A Study of Exception Handling and Its Dynamic Optimization in Java

Takeshi Ogasawara
takeshi@jp.ibm.com

Hideaki Komatsu
komatsu@jp.ibm.com

Toshio Nakatani
nakatani@jp.ibm.com

Tokyo Research Laboratory, IBM Japan
1623-14 Shimotsuruma, Yamato-shi, Kanagawa, Japan 242-8502

ABSTRACT

Optimizing exception handling is critical for programs that frequently throw exceptions. We observed that there are many such exception-intensive programs in various categories of Java programs. There are two commonly used exception handling techniques, stack unwinding and stack cutting. Stack unwinding optimizes the normal path, while stack cutting optimizes the exception handling path. However, there has been no single exception handling technique to optimize both paths.

We propose a new technique called *Exception-Directed Optimization (*E *)*, which optimizes exception-intensive programs without slowing down exception-minimal programs. E , a feedback-directed dynamic optimization, consists of three steps, exception path profiling, exception path inlining, and throw elimination. Exception path profiling attempts to detect *hot* exception paths. Exception path inlining compiles the catching method in a hot exception path, inlining the rest of methods in the path. Throw elimination replaces a `throw` with the explicit control flow to the corresponding `catch`. We implemented E in IBM's production Just-in-Time compiler, and obtained the experimental results, which show that, in SPECjvm98, it improved performance of exception-intensive programs by up to 18.3% without affecting performance of exception-minimal programs at all.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*incremental compilers, optimization, runtime environment*

General Terms

Performance, Experimentation, Languages

Keywords

Feedback-directed dynamic optimization, dynamic compilers, exception handling, inlining

OOPSLA 2001 Tampa Bay, FL, USA

1. INTRODUCTION

Language-supported exception handling mechanisms, such as in Ada [10], Modula-3 [13] and C++ [43], allows a programmer to write exception handlers for a region of the program. Java [29] is one of the modern languages that support exception handling. Using exceptions to change the control flows of the program is popular in Java [54, 56], and more and more Java programs tend to use exceptions.

There are two commonly used techniques to implement exception handling [53, 55]: *stack unwinding* and *stack cutting*. When an exception is thrown, stack unwinding is to unwind the stack frames to search for the corresponding exception handler. Here there is no penalty in *the normal path*, while there is a substantial overhead in *the exception handling path* to find the right exception handler by unwinding stack frames one by one [1]. On the other hand, stack cutting is to search for the list of the registered exception handlers. Here there is some overhead to register and deregister exception handlers in the normal path, while there is a smaller overhead in the exception handling path to find the right exception handler owing to the registered list.

Earlier research focused on optimizing the normal path by sacrificing the performance of the exception handling path, assuming that exceptions are rarely thrown in programs written in major programming languages [49, 34, 25]. In Java, however, there are many programs that frequently throw exceptions. For example, the SPECjvm98 benchmark suite has two exception-intensive programs, _228_jack and _213_javac, out of seven programs [19, 46]. Other five programs, such as _227_mtrt, _202_jess, _201_compress, _209_db, and _222_mpegaudio, throw much fewer exceptions. As shown in Section 4.4, the performance of _228_jack becomes 8.40% slower while that of _213_javac becomes 21.7% faster, when our Java Just-in-Time (JIT) compiler [57] switches the exception handling mechanism from stack cutting to stack unwinding.

There has been no single exception handling mechanism to optimize both the normal path and the exception handling path. In this paper, we propose a new optimization technique, called Exception-Directed Optimization (E), based on the feedback directed dynamic optimizations. E consists of three steps: 1) exception path profiling, 2) exception path inlining, and 3) throw elimination. When an exception is thrown and caught, exception path

[1]Here, the normal path is the code that is executed when no exceptions are thrown, while the exception handling path is the code that is executed only when an exception is thrown.

profiling records into a repository *the exception path*, which includes all the methods from the thrower to the catcher, and increments the counter associated with the path. Exception path inlining searches the repository for hot exception paths. For each hot exception path, it inlines into the catcher the rest of the methods. Finally, given a pair of throw and catch, throw elimination replaces the throw with the explicit control flow to the catch. Our new approach does not incur any penalty in the normal path, while it optimizes the frequently-executed exception handling paths by aggressive method inlining. It is applicable to any Java virtual machine (VM) regardless of whether it uses both an interpreter and a compiler [41, 59] or has only a compiler [7, 19]. With an interpreter, the first compilation can exploit the profiling information to compile the target method.

In this paper, we collected interesting runtime statistics; such as the exception counts and the lengths of the top three hot exception paths, for various categories of programs. We show that there are two types of programs, exception-intensive ones and exception-minimal ones, and that the majority of exceptions in most of the exception-intensive programs, except for _228_jack, are handled by a few hot exception handling paths of four or fewer invocations. The top three hot exception handling paths of _228_jack throw only 19% of the total exceptions.

We implemented three techniques, E , stack unwinding, and stack cutting, in our Java JIT compiler. Our preliminary experiment using SPECjvm98 shows that E improves the best performance of _228_jack by 18.3% (8.40%), javac by 13.8% (38.4%), and the geometric mean by 4.37% (6.91%), over stack unwinding (stack cutting). It also shows that E reduced the number of exceptions to occur at runtime for jack by 83.2% and javac by 100%. We also show that our results are better than those with Sun's HotSpot VMs [59] for the majority of the benchmark programs.

This paper makes the following contributions:

- A new approach, called Exception-Directed Optimization (E), for optimizing exception-intensive programs.
- A collection of runtime statistics, such as exception counts and the lengths of the top three hot exception handling paths, for various categories of programs.
- Evaluation of our new approach against two common techniques, stack cutting and stack unwinding, for the performance of SPECjvm98.

The rest of this paper is organized as follows. In Section 2, we explain the exception model of Java and discuss the overhead in the normal path and the exception handling path for stack unwinding and stack cutting. In Section 3, we introduce our new technique, E , for optimizing exception-intensive programs. In Section 4, we present a collection of statistics about exception behaviors for various categories of programs, and show performance results for SPECjvm98 in order to compare three approaches. We ran SPECjvm98 with three versions of IBM's production JIT compiler, which implement E , stack unwinding, and stack cutting, respectively. In Section 5, we discuss the related work. Finally, in Section 6, we give our conclusions.

2. RUNTIME OVERHEAD OF EXCEPTION HANDLING

Two common approaches to exception handling are *stack unwinding* and *stack cutting*. Stack unwinding is not as efficient as stack cutting in the exception handling path, while the former approach is more efficient in the normal path than the latter. The inefficiency of stack unwinding is due to the overhead for unwinding the stack itself.

In this section, we present the exception-handling model of Java, followed by the detailed descriptions of two approaches.

2.1 Exception handling in Java

In Java, an exception is an object from a subclass of the `Throwable` class. Programmers can define their exceptions and exception handlers by creating subclasses of the `Throwable` class. An exception is thrown with a `throw` statement. In a Java program, a `try` statement specifies a *try block*, which is a program region associated with one or more exception handlers. A `catch` statement specifies an exception handler and associates an exception class and its subclasses with the exception handler. These exception classes associated with an exception handler restrict exceptions that the handler can catch. An exception handler can catch only the exceptions that are instances of these exception classes. When an exception is thrown within a try block, if the exception cannot be caught by any exception handlers that are associated with the try block, an outer try block of the current try block is examined. The outer try block may exist within the same method or in a caller method. This search of try blocks continues until the handler that can catch the exception is found.

A complication arises in Java's exception handling because of *synchronized* methods. Before the handler found is executed, if the stack frames for synchronized methods are discarded, the exception handling mechanism must release the locks acquired by these methods [30]. Figure 1 illustrates an example of a Java program, where an exception, whose class is `Exc`, is thrown at method `work` and is caught by the handler of method `main`, which calls `work`.

2.2 Stack unwinding

Searching for the handler is performed during the execution of programs when an exception is thrown. There are three key steps in handling an exception: 1) *mapping* a program context, typically described by a program counter (PC), through which an exception is thrown to the corresponding try block, 2) *filtering* the exception, and 3) *searching* for the try blocks of caller methods. Figure 2 describes an abstract procedure of searching for the handler in the stack unwinding approach [53]. First, we explain each step in stack unwinding and then compare stack unwinding with stack cutting.

When an exception is thrown through a PC, we first examine which try block covers the program range including that PC. For the *table-driven* exception handling [55, 20], which is commonly used to implement exception handling, as in the Java VM [48], the compiler constructs the table that maps a range of PCs to a try block. With the table, the PC is searched for and is mapped to a try block. If a try block is found, we then examine whether the handler can catch the exception by comparing the class of the handler with that of the exception. We need this exception filtering because in Java a handler can only catch exceptions whose classes are subclass of the handler's class. A try block can have more than one handler, so

```
 1: class Exc extends Throwable {
 2:   static void work() throws Exc {
 3:     try {
 4:       try {
 5:         throw new Exc();
 6:       }
 7:       catch (NullPointerException e) {
 8:         /* some action */
 9:       }
10:     }
11:     catch (ArrayIndexOutOfBoundsException e) {
12:       /* some action */
13:     }
14:   }
15:   public static void main(String arg[]) {
16:     try {
17:       work();
18:     }
19:     catch (Throwable e) {
20:       /* some action */
21:     }
22:   }
23: }
```

Figure 1: An example of a Java program with try blocks

```
begin Searching for the handler
  Exception = the current exception
  Frame = the frame where Exception occurs
  Pc = the program counter where Exception occurs
  Handler = the default handler
  do
    while TryBlk = Search for a try covering Pc /*maping*/
      for each HandlerCandidate of TryBlk
        /* filtering class begin */
        if HandlerCandidate catches Exception then
          Handler = HandlerCandidate
          goto Done
        endif
        /* filtering class end */
      endfor
    endwhile
    /* searching for a try block (unwinding) begin */
    restore the context, such as PC and callee-saved registers
    release objects that are locked but not released in Frame
    Frame = the pointer to Frame's previous frame
    Pc = the last PC of Frame
    /* searching for a try block (unwinding) end */
  while Frame is not empty
Done:
  release locked objects to be released before Handler
  transfer the control to Handler
end
```

Figure 2: Searching for the handler in the stack unwinding approach

that the exception filtering is performed for each handler of the try block. For example, in Figure 1, the exception at line 5 is not caught by any handlers in the method `work()`, but it is caught by the handler in method `main` because the exception's class `Exc` is a subclass of class `Throwable` but not of class `NullPointerException` or `ArrayIndexOutOfBoundsException`. If the exception handler that can catch the exception is not found in the current method, we unwind the current stack frame and continue the above procedure in the caller method. Stack unwinding includes restoring the context of the caller's frame, such as the stack pointer and callee-saved non-volatile registers (NVRs), and releasing any unreleased locks on the objects that have been locked during the execution of the current frame.

Figure 3 shows an example of the call stack for the example of Figure 1. We use this example to show why unwinding the stack costs much more than a normal method return, particularly if the housekeeping data has been removed. Method `work` is called by method `main` in the last two frames. As shown in Figure 1, `work` has two try blocks and `main` has one try block. Let us assume that the runtime library unwinds the stack. When an exception is thrown at *PC1* in the top frame, the control is transferred to the code for handling the exception. If stack unwinding is implemented by compiler-generated code, the code is placed as the dummy epilog corresponding to the innermost try block of `work`. Otherwise, if it is implemented by the runtime library code, the code searches for the handler information for *PC1* in the *code database*, which maps each PC of the compiled code to its corresponding *code descriptor*. In both cases, the code performs the exception filtering, but it cannot find any handler that can catch the exception. As a result, it starts to unwind the current frame of `work`. It restores all callee-saved registers, but this example does not release any locked objects because no object is locked in `work`. As the final step, the code searches for the code descriptor of *PC2*, which is restored as the program counter of the call site. If the code descriptor is found, it provides the handler information or the dummy epilog. Otherwise, `main` has not been compiled, so that the code performs unwinding the stack and searching for the handler like as the interpreter does. In both cases, the code eventually finds the handler.

2.3 Stack cutting

Stack cutting [53] generates the housekeeping code in the normal path in order to improve performance of the exception handling path. The housekeeping code maintains a *try list* and a *lock list*. When the execution enters into a try block, the code pushes the try block into the try list. When the execution exits from the try block, the code pops the try block from the try list. Likewise, when calling a synchronized method, the code pushes the corresponding lock in the lock list. When returning from the synchronized method, the code pops the lock from the lock list. An indirect overhead in the normal path is saving non-volatile registers (NVRs) before calling a method in a try block, since the callee-save convention cannot be used for such NVRs in stack cutting [12]

In handling an exception, stack cutting does not have to do anything in the mapping step, while it simply scans the try list in the searching step. Thus, stack cutting allows faster exception handling than stack unwinding.

Stack cutting is typically implemented with `setjmp` [12, 22, 20], exception handler registration [38], or structured exception han-

	normal path execution			exception handling path execution					
	try	non-volatile registers (NVRs)	sync	try search	PC-to-try map	class filter	try search	NVR restore	locked objects release
Unwinding	no	callee-save	no	stack	yes	yes	stack	yes	frame
Cutting	yes (try list)	caller-save	yes (lock list)	try list	no	yes	try list	no	lock list

Table 1: Exception handling techniques and associated overhead

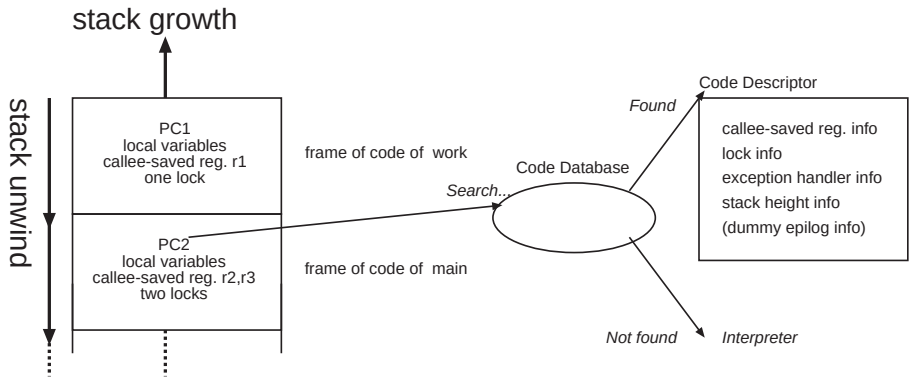


Figure 3: An example of optimized frames and unwinding them during exception handling.

dling (SEH) [50]. Table 1 summarizes differences between stack unwinding and stack cutting.

3. EXCEPTION-DIRECTED OPTIMIZATION

In this section, we explain our technique called *Exception-Directed Optimization (*E *)* to optimize exception-intensive programs. Although our approach is applicable to stack cutting as well as stack unwinding, we use stack unwinding as a basis in this paper because our interest is not only to optimize exception-intensive programs but also to avoid extra overhead in the normal path. Our approach consists of the runtime part and the compiler part. We first show the overview of our JIT compiler in Section 3.1, and then explain the runtime part in Section 3.2 and the compiler part in Sections 3.3 and 3.4.

3.1 JIT compiler overview

Our JIT compiler selectively compiles methods that are frequently invoked or have hot loops [57]. Methods are first executed by the interpreter. When the invocation count of a method reaches a threshold, the method is compiled by the first level compilation [58], and the compiled code is executed. Here the first level compilation quickly generates code by restricting time-consuming optimizations, such as method inlining. When the interpreter predicts that a loop in a method will be hot, the interpreter invokes the compiler and transfers to the compiled code.

There are multiple profiling systems in the JIT compiler, such as the sampling-based profiler [61] and the instrumenting profiler [58]. They accumulate information of which method should be compiled or recompiled during the program execution. If a profiler detects that a method that was already compiled should be further optimized, it requests recompilation of the method to the compile thread.

When the sampling-based profiler identifies a *hot* method, the JIT compiler aggressively inlines methods including virtual methods using class hierarchy analysis [41] along with other time-consuming optimizations [58]. Inlining methods is very important to improve the performance of Java programs. For example, the JIT compiler improves the best SPEC ratio of _228_jack by 11.3% and that of _213_javac by 6.47% even without E . The code expansion caused by inlining methods is controlled by heuristics about the depth of method invocations and the increase from the original method size.

3.2 Profiling exception paths

To optimize exception-intensive programs, we collect information on the behavior of each exception handling and save it in the *profile repository* as a *profile*. There are three values in each profile: *exception path*, the start time, which is the time when the profile was created, and the counter, which is the number of occurrences of the same exception path. An exception path is the stack trace from the method that throws an exception through a method that catches the exception and is denoted by $(\{m_0, pc_0\}, .., \{m_{n-2}, pc_{n-2}\}, \{m_{n-1}, \phi\})$. m_i denotes the i-th method from the root method on the exception path. m_i invokes m_{i+1}. pc_i denotes the program counter of the call site where m_i invokes m_{i+1}. Exceptions are thrown at some program points within m_{n-1} and are caught by some handlers within m_0.

A profile is allocated or updated as follows. During exception handling, the runtime library unwinds the stack and it also saves the program counter for each stack frame. In an environment using selective compilation, the program counter can point to bytecode or to compiled code. After the runtime library finds the handler, it searches the profile repository for the profile whose exception path is the same as the current exception path. This search should be as quick as possible so that the profiling does not add an extra penalty to the cost of exception handling. For instance, the profile repository can be implemented by using a hash table using as keys the

pairs consisting of the PC of a program address throwing an exception and the PC of the corresponding handler. If the profile is not found, a profile is created in the profile repository. The current exception path and the current time are saved as the exception path and the start time of the profile, respectively, and the counter of the profile is zeroed. If there is already a profile whose exception path is the same as the current exception path, the counter of the profile is incremented. The runtime library also checks if the profile has not been passed to the dynamic compilation system as a request for dynamic compilation. This check avoids not only redundant profiling but also redundant requests for dynamic compilation because there can be a time lag between a request and its processing. If the buffer of profiles overflows, these profiles which have smallest counts are discarded.

If the profile has never been used to request dynamic compilation, we examine the profile to determine whether to request dynamic compilation using the profile. We can calculate the frequency of the exception path from the profile with the counter and the start time of the profile and the current time as follows:

$$PathFrequency = \frac{ProfileCounter}{CurrentTime - ProfileStartTime}.$$

Here we need a $ProfileCounter$ large enough to keep the accuracy. If the frequency reaches a certain threshold, the profile is passed to the dynamic compilation system. At the same time, those profiles, whose catching methods are the same as that of the profile and whose frequencies are near the threshold, are searched for, and if any such a profile is found, it will also be sent for compilation. Those profiles, whose catching methods are the same as that of the profile but whose frequencies are still low, become useless after the compilation of the method. They will be reinitialized with the newly compiled method.

The threshold should be adjusted depending on the speed of the executed code. The frequency of an exception path executed by the interpreted code tends to be lower than that by the compiled code. Similarly, the frequency by the unoptimized code tends to be lower than that by optimized code.

3.3 Exception path inlining

When the compiler is invoked to compile a method m_1, it searches the profile repository for the exception paths whose catching methods are the same as the target method. If such exception paths are found, the compiler examines each of the exception paths to make a decision on inlining. Previous work on inlining [15, 17, 9, 8] reduced it to the knapsack problem, as explained in the related work section above. The framework in these projects is based on a dynamic call graph, whose edge represents the relationship between a call site and its callee and whose node represents a method. Each edge has a frequency that represents how often the method is invoked from a call site. In contrast, the frequency of an exception path represents how often the method at the end of the path returns to the other end of the path with an exception. The compiler should treat method invocations on the exception path as a whole. This is because the total benefit of inlining the whole exception path is not a simple summation of the benefits of inlining each invocation on the exception path. Unless all the invocations on the exception path are inlined, we cannot eliminate the cost of PC-to-try mapping and exception filtering by replacing the `athrow`s with `goto`s. We inline all the methods on each exception path, ignoring the code expansion. Although we currently inline the whole method body, we can apply a technique, called *partial inlining* [33], to reduce the code size expansion by selecting program regions to be inlined. We evaluate how inlining the whole exception path affects the total compilation time in Section 4.4. Overall, the increase in the compilation time caused by inlining all exception paths is small in our experiments.

An exception path can include virtual invocations. The exception path profiling collects information on which methods are invoked by the virtual invocations in the stack when an exception is thrown. To inline these virtual methods, the compiler applies *devirtualization* with tests [11, 31, 6, 24], with dynamic patching [41], or with the combination of these two approaches [19]. In devirtualization with tests [11, 31, 6], the compiler generates the code that guards the inlined code and then inlines each virtual method. The guard code checks if the class of each receiver object for the inlined virtual method is the same as that of the inlined virtual method itself. If the guard code test fails, a normal virtual invocation is performed. If different virtual methods are invoked at the same call site, the compiler may generate guard code for every class of the virtual methods to inline each of them. Instead of testing the class of a receiver object, the guard code may test the method of a receiver object specified at the call site [24]. *Type inference* [52, 5, 14, 27] computes the set of possible classes for every object reference in a program and narrows down the set of the classes reachable at each virtual call site. In conjunction with *class hierarchy analysis* [23], which tracks how methods are overridden in the whole program, the guard code can be removed if it can be asserted that only a single method is invoked at a virtual call site. However, since Java has dynamic class loading to allow a method to be dynamically overridden, it is hard to say that such an assertion is valid throughout the program execution, except for `final` methods. Earlier research presented techniques of canceling optimizations that are no longer valid because of changes in the class hierarchy [36, 40]. By applying these techniques, the compiler can inline the method without the guard code. If the method is overridden later, the compiler will cancel inlining by either discarding the whole method or switching to the original virtual method.

An exception path can include invocations of synchronized methods or methods with try blocks. For synchronized methods, there can be two approaches: the one without and the other with an extension of the Java VMs. One specifies a try block that covers the code of an inlined method and associates the try block with a new `finally` block that simply releases the lock for the receiver object of the inlined method and re-throws the same exception. This implementation is the same as that of Java's `synchronized` block [29], and it needs no extensions of the Java VMs. However, it requires a control transfer for entering and leaving a finally block in order to release locks. Also, it introduces extra work in the dataflow analysis due to the try block. The other approach maintains special data that keep track of locked receiver objects. For example, receiver objects are saved in the stack in the order of their locks. The compiler can generate a lock control table that describes how many objects have been locked at a given program point. If an exception is thrown through a PC, the system does a lookup in the lock control table to obtain the number $n_{sync}(pc_{xcpt})$ of locks at that PC. If the handler that catches the exception is found, the system also obtains the number $n_{sync}(pc_{hdlr})$ of locks at the handler. Since inlined methods are nested, the handler releases the last

```
/* Case 1 */
    new #7 <Class Exc>
    dup
    invokespecial #18 <Method Exc()>
    athrow

/* Case 2 */
    getstatic #5 <Field Exc s_var>
    athrow
```

Figure 4: Two bytecode sequences that throw exceptions.

```
/* Case 1 */
    L1 = new Exc()
       = invokespecial L1.Exc()
       = athrow L1

/* Case 2 */
    L2 = s_var
       = athrow L2
```

Figure 5: Two 3-address-code sequences that throw exceptions.

$(n_{sync}(pc_{xcpt}) - n_{sync}(pc_{hdlr}))$ locks in the stack frame. Otherwise, the system releases all the locks. In an actual implementation, the number $n_{sync}(pc_{hdlr})$ is available from the exception filtering. Searching for the number $n_{sync}(pc_{xcpt})$ is typically a binary search of the lock control table. However, by caching the search results, the overhead of the search can be reduced. As a result, we took the second approach since it releases locks faster than the first approach without using a control transfer.

For methods with try blocks, exception tables for the try blocks are merged while the program ranges are adjusted. This is possible because Java uses table-driven exception handling [55, 20] as explained in Section2. To allow succeeding optimizations to reorder the basic blocks, the compiler assigns an identifier, called a *try ID*, to each basic block. The try ID is associated with information on the handlers that cover the basic block. The compiler compiles merged exception tables and generates a try ID table, which maps a try ID to the corresponding handler information. Any basic block that is not associated with any try block will have the default try ID. Once the try ID table is generated and try IDs are assigned to each basic block, the exception table will no longer be accessed.

Additionally, other method inlining optimizations, such as the one based on the frequency of edges in the dynamic call graph [21, 9, 8, 7], may be performed along with the exception path inlining.

3.4 Throw elimination

After exception paths are inlined within the same code block, the compiler examines each `athrow` in the code block and, if possible, removes it and link the basic block that ends with the `athrow` to the corresponding handler. To perform such code transformation, called *throw-catch linking*, the compiler performs *exception class resolution* and *compile-time intra-method exception handling analysis*.

3.4.1 Exception class resolution

Figure 5 shows the 3-address-code corresponding to the bytecode in Figure 4. The compiler then resolves the class of every exception

```
1: class ExactClass {
2:   static void work2() {
3:     try {
4:       /* some code */
5:       throw new Exc();   /* may be guarded with if-statements */
6:       /* some code */
7:     }
8:     catch (Exc e) {
9:       /* some action */
10:    }
11:  }
12: }
```

Figure 6: Java source that throws exceptions of the exact class.

object that is thrown at each `athrow` in the code. This is called *exception class resolution (ECR)*. This information is used in the next step. To resolve these classes, the compiler performs *type inference* [52, 5, 14, 27]. In our exception class resolution, the compiler attempts to compute the static type of each object reference within a method using the similar approach to [27]. This is less expensive and thus more suitable for dynamic compilers than computing concrete types [52, 5], which would require analyzing the whole program to compute the set of possible classes of every object reference. The compiler converts the bytecode to a 3-address-code as in [27]. Figure 4 shows two typical bytecode sequences that throw exceptions. In Case 1, an exception object of class `Exc` is created and then immediately thrown. In Case 2, the static variable `s_var` holds an object of class `Exc` or its subclass, and the object held in `s_var` is thrown. Unlike receiver objects, these two patterns are dominant in throwing exceptions, so that the compiler can efficiently compute the static type of each exception object that is thrown in most cases.

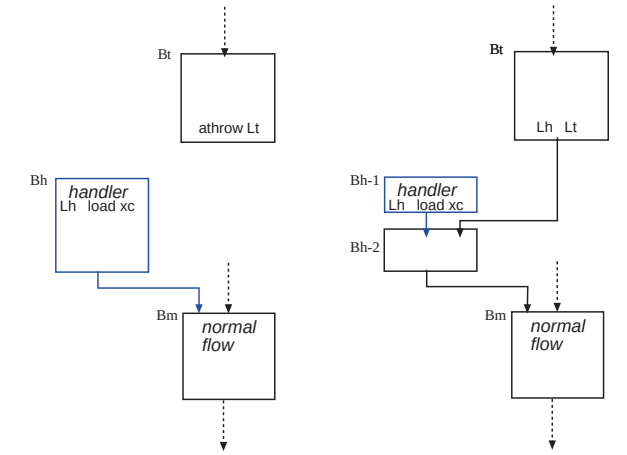
3.4.2 Compile-time intra-method exception handling analysis

With the types of exception objects provided by ECR, within the code where exception paths are inlined, the compiler attempts to find one or more handlers that can catch exceptions from each `athrow`. That is called *Compile-time Intra-method Exception Handling Analysis (CIEHA)*. CIEHA is similar to the normal exception handling, explained in Section 2.2, except for the fact that CIEHA knows only the set of possible classes (or types) of exception objects but not the class of the actual exception object. CIEHA saves the result of its analysis for use in the next step. In the simplest case, ECR computes the exact class of the exception objects and CIEHA finds the corresponding handler. For example, in Case 1 of Figure 5, ECR computes the exact class of `L1`, or `Exc`. Assuming that a try block covers the code "`= athrow L1`" and its handler can catch the object of class `Exc`, CIEHA finds the same handler at compile time as the one exception handling can at runtime. Figure 6 shows an example of Java source corresponding to this trivial situation. In this case, CIEHA saves the analysis that "`= athrow L1`" is always caught by this handler.

Unless ECR computes the exact class for the exception objects, CIEHA needs to do more work than what exception handling must do at runtime. A typical case is that ECR computes a set of classes belonging to a class hierarchy and CIEHA finds the handlers corresponding to those classes. For example, in Case 2 of Figure 5, assume that: 1) ECR determines two possible classes of `L2` such

```
1: class NotExactClass {
2:   Exc s_var;
3:   static void work3() {
4:     try {
5:       /* some code */
6:       throw s_var;   /* may be guarded with if-statements */
7:       /* some code */
8:     }
9:     catch (SubExc e) {
10:       /* some action */
11:    }
12:    catch (Exc e) {
13:       /* some action */
14:    }
15:  }
16: }
```

Figure 7: Java source that potentially throws exceptions of multiple classes.



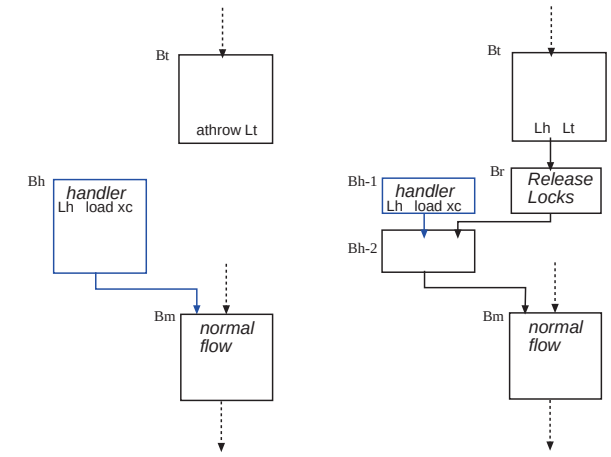
(a) Before throw-catch shortcutting *(b) After throw-catch shortcutting*

Figure 8: Throw-catch linking without compensation code.

as `Exc` and its subclass, `SubExc`, 2) a try block covers the code "`= athrow L2`", and 3) two handlers are associated with the try block in the exception table: the first one can catch `SubExc` and the second one can catch `Exc`, which are examined by exception filtering in this order, Figure 7 shows an example of Java source code that satisfies these assumptions. In this case, the saved CIEHA result is that "`= athrow L2`" is always caught by the first handler, if the class of a thrown exception is `SubExc`, or by the second handler, if the exception class is `Exc`.

3.4.3 Throw-catch linking

Using the results of CIEHA, the compiler attempts to transform each `athrow` to an unconditional branch or a conditional branch to the corresponding handler in the code. This is called *throw-catch linking (TCL)*. We illustrate how TCL transforms the code in this section. Figure 8 is a diagram that shows the blocks of code and the control flow where TCL links the block that ends with an `athrow`, denoted by Bt, with the handler block that catches exception Lt, denoted by Bh. Bm denotes the block at which the control flow merges from the handler and from the normal path. Code "`Lh = loadExc`" makes the current exception visible in the scope of the



(a) Before throw-catch shortcutting *(b) After throw-catch shortcutting*

Figure 9: Throw-catch linking with compensation code.

code. TCL extracts the code from Bh as a new handler, denoted by Bh-1. Also, TCL replaces the code "`= athrow Lt`" with the code "`Lh = Lt`", which copies the exception Lt to Lh. TCL then links Bt with the rest of Bh, denoted by Bh-2. If such a "shortcut" goes across the boundary of two synchronized methods that are inlined, TCL generates compensation code to release a lock. Figure 9 is the same as Figure 8 except that it has a block, denoted as Br, inserted between Bt and Bh-2 to release the locks of the synchronized methods. If the exceptions caught by the handler are thrown only by the removed `athrow`, block Bh-1 becomes unreachable and the compiler can eliminate it after TCL. CIEHA may find more than one handler for an `athrow`. In these cases, TCL appends the code to perform conditional branches to these handlers after Bt. CIEHA could also detect that some exceptions from the `athrow` are caught within the code. In this case, TCL appends the code to perform an `athrow` only if the classes of exceptions are not handled within the code. Thus, the compiler simplifies the complex control flows, which require runtime intra-method exception handling.

4. PERFORMANCE EVALUATION

In this section, we evaluate E using our system. We first experimented with ten programs, which include various categories of applications, to investigate how many exceptions are thrown during the execution of programs. Table 2 shows those results, where the first two columns describes information about programs, the third describes the parameters used for executing the programs, the fourth column shows total exception counts, the fifth to seventh columns show top three exception counts per exception path and the corresponding path lengths, and the last column describes what operations were performed or which benchmark was executed. According to the results, there are two typical types of programs, exception-intensive programs and exception-minimal programs. Exception-intensive programs are programs that throw more than ten thousand exceptions throughout their execution or throw thousands exceptions in some phases such as the startup phase or the file input phase. For example, _228_jack and _213_javac in SPECjvm98, Euler in Java Grande Forum Benchmark, and XML parser benchmark are typical exception-intensive programs. In contrast, exception-minimal programs throw tens to hundreds of exceptions throughout their execution. For example, the section three

Category	Program	Parameters	Exception counts				Scenario
			Total	per exception path			
				Top (path len.)	2nd(path len.)	3rd(path len.)	
Web browsers	ICE Browser 5.05	n.a.	1245	406 (1)	406 (3)	344 (3)	open www.acm.org
	XBrowser 3.0	n.a.	2892	1018 (1)	1018 (3)	356 (3)	open www.acm.org
word processor	Ichitaro Ark 1.0	n.a.	3632	1082 (1)	1082 (3)	691 (4)	startup
			4040	1169 (1)	1169 (3)	718 (4)	startup and open pubform.doc
multimedia player	JOS Media Player 2.0	n.a.	1075	506 (1)	506 (3)	26 (3)	startup
			5321	2512 (2)	1066 (1)	1066 (3)	startup and load an AVI file
			8205	2516 (2)	2506 (1)	2506 (3)	startup and load an MP3 file
XSLT processor benchmark	XSLTMark 1.2.1	run once	7116	2143 (3)	2143 (1)	1099 (3)	Saxon (Michael Kay)
			2718	349 (3)	305 (1)	305 (3)	XalanJ (Apatch Project)
			1865	439 (3)	439 (1)	439 (3)	XT (James Clark)
XML parser benchmark	XML benchmark	run once	17640	11961 (3)	3944 (3)	1129 (2)	XML4J (IBM)
			17640	11961 (3)	3944 (3)	1129 (2)	XP (James Clark)
chat server benchmark	VolanoMark 2.1.2	count=100	736	200 (4)	110 (4)	105 (2)	server
			217	169 (4)	16 (1)	16 (3)	client
business benchmark	SPECjbb2000 1.0	1 warehouse	506	99 (1)	96 (1)	96 (3)	n.a.
client benchmark	SPECjvm98 1.03	1st run; compliant mode	240	99 (1)	39 (3)	39 (1)	_200_check
			226	75 (3)	75 (1)	25 (2)	_227_mtrt
			1324	441 (3)	441 (1)	147 (2)	_202_jess
			100	33 (3)	33 (1)	11 (2)	_201_compress
			19	6 (3)	6 (1)	2 (4)	_209_db
			451	150 (3)	150 (1)	50 (2)	_222_mpegaudio
			242318	18935 (4)	14110 (3)	14110 (4)	_228_jack
			23849	19108 (3)	2656 (2)	608 (2)	_213_javac
large application benchmark	Java Grande Forum Benchmark Suite v2.0 Section 3	size A (small data set)	27	9 (3)	9 (1)	9 (3)	Search
			41222	41198 (2)	8 (3)	8 (1)	Euler
			24	8 (3)	8 (1)	8 (3)	MD
			54	18 (3)	18 (1)	18 (3)	MC
			48	16 (3)	16 (1)	16 (3)	Ray Tracer

Table 2: Exception counts and exception path lengths in various programs

benchmark programs in the Java Grande Forum Benchmark except for Euler and SPECjbb2000, which run for relatively long times, are obviously exception-minimal programs. Also, Table 2 shows that the exception counts for just the top three exception paths cover about 90% of the total exceptions in those exception-intensive programs and that the path lengths of those exception paths are shorter than five. Therefore, there is a good opportunity for optimizing those exception-intensive programs using E . However, in _228_jack, the top three exception paths throw only 19% of the total exceptions. Therefore, optimizing _228_jack is more challenging for E . In the rest of this section, we used SPECjvm98 as a typical example of a set of exception-intensive programs. Section 4.1 explains the methodology of our evaluation. Section 4.2 explains the system used in our experiments. Section 4.3 explains the details of the benchmark programs used in our experiments, focusing on the exceptions thrown. Section 4.4 presents the results and discusses them.

4.1 Experimental methodology

For practicality, we used the SPECjvm98 benchmark suite [60] in the *compliant mode* throughout our experiments. We satisfied all program execution rules[2]. There are two modes in the applets of SPECjvm98, the compliant mode and the test mode. The compliant mode is used when publishing the scores of SPECjvm98 while the test mode can be used for other purposes such as research [19, 7, 40].

[2]In terms of the reporting rules, we did not satisfy this specific rule: *The entire system, including the hardware and all software features, is required to be publicly available within three months of the date of publication of the results*.

We compare E , stack unwinding (denoted as Unwind), and stack cutting (denoted as Cut) with each other. We found that E improves Unwind in exception-intensive programs and still adds no penalty to exception-minimal programs. Note that sections with different exception characteristics coexist in the same program in our experiments. As explained in Section 2, Cut may be better than Unwind in exception-intensive programs. We found support for this expectation in our experiments. We also show that those results of E are practical by comparing them to two state-of-the-art Java VMs, the HotSpot Client VM and the HotSpot Server VM (both are build 1.3.1-b24, mixed mode) [59].

4.1.1 Exception handler registration

We used *exception handler registration (EHR)* [38], also known as *structured exception handling (SEH)* [50], as Cut. EHR is supported by operating systems such as IBM OS/2 and Microsoft Windows [45]. In EHR, the compiler generates prolog code that registers an *exception registration record (ERR)* as well as epilog code that deregisters it for any compiled code segment that has try blocks [38]. Each thread has its own ERR list, which is the last-in-first-out list of ERRs. When the code is executed in a thread, it maintains the ERR list of the thread by adding and removing its ERR from the ERR list at runtime. Also, the code maintains a field in its ERR that identifies the current try block at runtime. Non-volatile registers (NVRs) (i.e. EBX, ESI, EDI, and EBP in our compiler) that are not used in the code and are exposed to callees are conservatively saved in the code, because those exposed NVRs that could be saved

in the callee code are not restored during exception handling. For synchronized methods, the compiler generates code that not only obtains the lock on an object but also registers the object to the top-most ERR in the ERR list. As a result, the system can release the locks when the ERR catches an exception without traversing the stack.

When an exception is thrown, the system traverses the handler list, performing exception filtering, until a handler that can catch the exception is found. Thus, no unwinding the stack occurs during exception handling in EHR.

4.2 Environment

We implemented E on our Java Just-In-Time (JIT) compiler [57, 41, 40, 42], which works with the IBM Developer Kit for Windows, Java 2 Technology Edition [37], Version 1.3.1.

Throughout the measurements, we used the same parameters for a Java VM and SPECjvm98 that Sun used to submit the results of their HotSpot VMs to the SPEC. As for the Java parameters, the applets of SPECjvm98 were executed with a loopback network, such as `appletviewer.exe http://localhost/SpecApplet.html`. The initial and maximum amounts of Java heap space were 42 MB, specified with the parameters `-Xms42m -Xmx42m`. Also, to measure the HotSpot VMs, the tuning options `-XX:NewSize=13m -XX:MaxNewSize=13m` were specified. As for SPECjvm98 parameters, `spec.initial.automin=5` causes each benchmark program to be executed for a minimum of five times.

In the compliant mode, SPEC ratios are calculated by the applets after the benchmarks are finished. SPEC ratios for benchmark programs and their geometric mean for the second execution of the applet immediately after rebooting the machine were reported in the rest of this section.

The measurements were performed on an IBM Personal Computer 300PL Model 6892-44J, which has a Pentium-III 800MHz CPU, and 256 MB physical memory, running Microsoft Windows NT Server Version 4.0 and the Microsoft Internet Information System 4.0 with Service Pack 6.

4.3 Characteristics of the benchmark programs

This section gives an overview of the characteristics of the benchmark programs. SPECjvm98 is a suite of benchmark programs [60] and is currently accepted as one of the major Java benchmarks for evaluating Java VMs [46, 57, 42, 41, 40, 7, 19]. SPEC-compliant runs, whose SPEC ratios can be submitted to the SPEC if the reporting rules are satisfied, should be performed via the applet `SpecApplet`. Table 3 shows the number of exception paths and the exception counts for each benchmark program of SPECjvm98 during the two classes of runs, the first run and any run after the first run (i.e., the second through the fifth runs). During one execution of the SPECjvm98 applet, each benchmark program was executed for a minimum of five times, as explained in the previous section. We collected the data in Table 3 during each execution of those five runs. As Table 3 shows and earlier research [46, 19] reports, _213_javac and _228_jack are exception-intensive programs in the SPECjvm98 benchmark. The other programs (exception-minimal programs: _227_mtrt, _202_jess, _201_compress, _209_db, and _222_mpegaudio) throw few exceptions once initialized. These exception-minimal programs and exception-intensive programs run in the same execution environment.

	the 1st run		after the 1st run	
	Total except. counts	Num of except. paths	Total except. counts	Num of except. paths
_227_mtrt	226	6	10	1
_202_jess	1324	5	0	0
_201_compress	100	5	0	0
_209_db	19	5	0	0
_222_mpegaudio	451	5	0	0
_228_jack	242318	170	241877	165
_213_javac	23849	8	22373	3

Table 3: The original characteristics of SPECjvm98 benchmark programs

	total exception counts				
benchmark	1st run	2nd	3rd	4th	5th
_227_mtrt	226	10	10	10	10
_202_jess	1309	0	0	0	0
_201_compress	66	0	0	0	0
_209_db	12	0	0	0	0
_222_mpegaudio	300	0	0	0	0
_228_jack	113715	74097	59938	43505	40724
_213_javac	3885	44	0	0	0

Table 4: The total exception counts using EDO

4.4 Results and discussion

4.4.1 Total exception counts and numbers of exception paths

Table 4 shows the total number of exceptions for SPECjvm98 benchmark programs during each run. During the five runs, E reduced 83.2% of the original value (241877) for _228_jack and 100% for _213_javac.

Table 5 shows that E reduced the number of exception paths for SPECjvm98 benchmark programs. The number of hot execution paths, 954, is larger than the original 170 hot execution paths in Table 3 because selective compilation changes the address of the same call site from the bytecode address to the compiled code address. As a result, many exception paths can be collected for some logically equivalent exception path. In other words, the number of exception paths can increase because of compilation and recompilation, while E normally reduces the number of exception paths.

	total exception paths				
benchmark	1st run	2nd	3rd	4th	5th
_227_mtrt	6	1	1	1	1
_202_jess	6	0	0	0	0
_201_compress	4	0	0	0	0
_209_db	4	0	0	0	0
_222_mpegaudio	4	0	0	0	0
_228_jack	954	301	320	264	229
_213_javac	11	1	0	0	0

Table 5: The number of exception paths using EDO

benchmark	E	Unwind	Cut	HSC	HSS
_227_mtrt	143	144	146	59.4	128
_202_jess	77.3	77.0	76.0	69.4	86.6
_201_compress	78.0	77.8	78.0	63.4	73.1
_209_db	26.9	26.8	26.2	27.5	29.1
_222_mpegaudio	153	152	148	82.1	103
_228_jack	142	120	131	102	122
_213_javac	56.9	50.0	41.1	34.7	40.5
Geom. Mean	83.6	80.1	78.2	57.5	73.9

Table 6: SPEC ratios: Best

benchmark	E	Unwind	Cut	HSC	HSS
_227_mtrt	82.6	79.6	85.1	58.3	59.1
_202_jess	56.2	56.1	57.1	61.1	60.1
_201_compress	76.6	76.1	76.1	62.2	68.5
_209_db	26.4	26.3	25.8	23.6	24.4
_222_mpegaudio	123	124	124	78.1	79.1
_228_jack	80.1	85.6	90.1	88.4	37.0
_213_javac	34.3	32.0	29.6	29.5	14.5
Geom. Mean	61.1	60.8	61.1	52.3	42.6

Table 7: SPEC ratios: Worst

The increase from the second run to the third run for _228_jack in Table 5 is an example of such a case. As the JIT compiler compiles more bytecode, most of the addresses of call sites on exception paths become those of JIT-compiled code. However, the number of exception paths was eventually reduced by E to 24.0% of the value (954) for the first run.

4.4.2 SPEC ratios

We present SPEC ratios for three approaches: E , Unwind, and Cut. We also present SPEC ratios for the HotSpot Client VM (denoted as HSC) and the HotSpot Server VM (denoted as HSS). The SPECjvm98 applet calculates the best and the worst SPEC ratios for each benchmark and the geometric mean of all the SPEC ratios. Table 6 and Table 7 show the best and the worst SPEC ratios, respectively. The SPEC ratio of each benchmark in Table 6 is calculated based on the best time for five runs of the benchmark. Similarly, the SPEC ratio in Table 7 is calculated based on the worst times. With these tables, we evaluate how much E improves SPEC best ratios, while it introduces only small penalties.

We compare E with Unwind. E improved _228_jack by 18.3%,

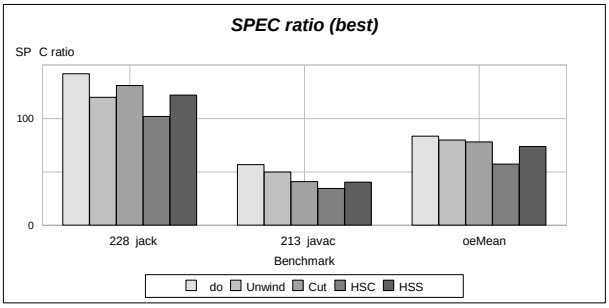


Figure 10: SPEC Best ratios for exception-intensive programs

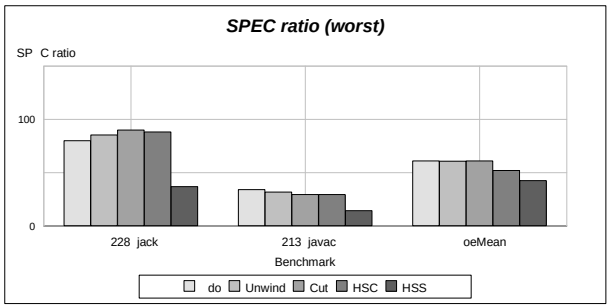


Figure 11: SPEC Worst ratios for exception-intensive programs

_213_javac by 13.8%, and the geometric mean of seven programs by 4.37% in terms of the best SPEC ratios (see Figure 10). In _213_javac, the compiler eliminated the code that frequently allocates and initializes exception objects. Once E replaces a `throw` with the explicit control flow to the corresponding `catch`, if an exception object that reaches the `throw` is no longer used by the `catch`, the compiler eliminates the allocation and initialization code for the object. The mechanism of this elimination is different from lazy exceptions [19], which postpones creation and initialization of an exception object until they are actually used. However, both can reduce the overhead for exception object creation. For the worst SPEC ratios, E degraded _228_jack by 6.43%, although it reduced the overhead for exception handling path by 50%. The degree of this degradation depends on how the system recompiles *hot* methods. In the first run of _228_jack, where the worst SPEC ratio was measured, time-consuming optimizations such as DAG-based optimizations and dataflow-based optimizations [58] consumes additional overhead (19%) for *hot* methods, however, the resulting code cannot compensate for the overhead within the first run. E does not affect the other programs. Thus, the results demonstrate that the performance of exception-intensive programs is dramatically improved while that of exception-minimal programs is not degraded.

Table 6 also shows the performance trade-off between Unwind and Cut. Cut shows an 8.40% improvement over Unwind in _228_jack, however, it degrades _213_javac by 21.7%This is because methods that have try blocks are heavily invoked and the overhead of maintaining the ERRs is not negligible in these programs. Thus, the results show that the runtime overhead specific to Cut and its fast exception handling are traded off and the drawback of the former counteracts the benefit of the latter.

5. RELATED WORK

Lang et al. [45] includes a survey of exception handling mechanisms, both those based on programming languages and those based on operating systems. Implementation techniques of programming-language-based exception handling mechanisms are described for CLU [49], Modula-2 [25], C++ [43, 12, 55, 20], a dialect of C [28], and intermediate languages [22, 53]. Ramsey and Jones [53] categorize these techniques into two, *stack unwinding* and *stack cutting*, based on whether or not the stack is walked. We use these terminologies in this paper.

Drew et al. [25] show that unwinding the stack is expensive if

the normal path is highly optimized and contains no housekeeping code for exception handling. They propose an efficient implementation of stack unwinding for Modula-2. Their compiler generates a *dummy epilog* for each procedure, which restores callee-saved non-volatile registers and discards the stack frame. The dummy epilog includes the code to search for the caller's dummy epilog. To eliminate the overhead of searching, they also mention the *hidden argument approach*, where a special extra argument is passed at each function call. However, this adds a penalty to the normal path.

Krall and Probst [44] propose an implementation of stack unwinding for JIT-based Java VM, called CACAO, with an optimization similar to the hidden argument approach [25]. JUDO [19], which is a *compile-all* Java VM, uses stack unwinding with no housekeeping code in the normal path. They perform a binary search of the lookup table to map the program counter to the unwind data structure. To reduce the search overhead, JUDO caches the lookup results, which gives a significant improvement in _228_jack. Notice that our base compiler used in Section 4 includes this optimization. They also present a technique to eliminate unused exception objects that have no side effects.

Lee et al. [47] propose *exception handler prediction* to reduce the overhead of Java's exception filtering in their Java VM, LaTTe. Their technique optimizes the cases where they could predict that a handler in the same method catches a `throw`. In such cases, they simply generate the jump instruction together with the code to check if the current exception's class is the same as the predicted one. By contrast with their dynamic approach, our approach uses exact results from the dataflow analysis. Thus, no checking code needs to be generated. In [46], they experimented with extending their technique across multiple methods by controlling the method inlining heuristics. While they mention that such an extension is possible, they conclude that it is impractical because it spends more than twice of the compile time. They do not describe how they changed the method inlining heuristics; however, their report implies that aggressive method inlining can cause an explosion of code size and compilation time.

Method inlining is one of the most widely applied optimizations in both static compilers [15, 17, 9] and dynamic compilers [35, 7, 51]. The central issue in the optimization is how aggressively inlining should be performed. Different compilers adopt different heuristics for addressing the issue. However, to the best of our knowledge, there is no heuristics that takes exceptions into account.

There are many studies that focus on optimizing the normal path of programs in the presence of exceptions [34, 16, 18, 26, 32]. Our focus is on reducing the overhead due to exception handling rather than such code optimization. Their results and our approach can complementarily improve exception-intensive programs.

6. CONCLUSIONS

This paper has presented a novel approach to adaptive optimization of exception-intensive programs. It consists of exception path profiling, exception path inlining, and throw elimination. Exception path profiling records an exception path into a repository and updates the profile associated with the path when an exception is thrown. Exception path inlining searches the repository for hot exception paths and inlines the methods on these exception paths into the catcher. Finally, given a pair of throw and catch, throw elimination replaces the throw with the explicit control flow to the catch. Our new approach does not incur any penalty in the normal path, while it optimizes the frequently-executed exception handling paths.

We have presented experimental data about exception counts and the number of exception paths, using various categories of programs. Using those data, we have shown that some of the programs throw many exceptions during their executions and performance suffers from the overhead for exception handling. We have presented experimental results with IBM's production Just-in-Time compiler and discussed the effectiveness of our approach. Experimental results show that, in exception-intensive programs, our approach greatly reduced exception counts by 83.2% for _228_jack and by 100% for _213_javac and greatly improved the best SPEC ratios for stack unwinding by 18.3% for _228_jack and by 13.8% for _213_javac, and that these results are practical compared with Sun's HotSpot VMs.

7. ACKNOWLEDGMENTS

We are grateful to the people in the Network Computing Platform group at Tokyo Research Laboratory for implementing our special JIT compiler.

8. REFERENCES

[1] *Proceedings of the 10th annual conference on Object-oriented programming systems, languages and applications* (New York, NY, USA, Oct. 1995), ACM Press.

[2] *ACM SIGPLAN '00 Conference on Programming language design and implementation* (New York, NY, USA, May 2000), ACM Press.

[3] *Proceedings of the ACM 2000 Conference on Java Grande* (New York, NY, USA, June 2000), ACM Press.

[4] *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2000)* (New York, NY, USA, 2000), ACM Press.

[5] A , O., H¨ , U. Type feedback vs. concrete type inference: a comparison of optimization techniques for object-oriented languages. In ACM [1], pp. 91–107.

[6] A , G., H¨ , U. Eliminating virtual function calls in C++ programs. In *Proceedings of the 10th European Conference on Object-Oriented Programming - ECOOP '96 (Lecture Notes in Computer Science, Vol. 1098)* (Berlin, July 1996), Springer-Verlag, pp. 142–166.

[7] A , M., F , S., G , D., H , M., S , P. F. Adaptive optimization in the Jalapeño JVM. In ACM [4], pp. 47–65.

[8] A , M., F , S., S , V., S , P. F. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization* (New York, NY, USA, Jan. 2000), ACM Press, pp. 52–64.

[9] A , A., G , R., S , R. Aggressive inlining. In *ACM SIGPLAN '97 Conference on Programming language design and implementation* (New York, NY, USA, June 1997), ACM Press, pp. 134–145.

[10] B , T. P., R , G. A. Implementing Ada exceptions. *IEEE Software 3*, 5 (Sept. 1986), 42–51.

[11] C , B., G , D. Reducing indirect function call overhead in C++. In *Proceedings of the ACM SIGPLAN '94 symposium on Principles of programming languages* (New York, NY, USA, Jan. 1994), ACM Press, pp. 397–408.

[12] C , D., F , P., L , D., M , M. A portable implementation of C++ exception handling. In *Proceedings of the C++ Conference* (Aug. 1992), USENIX Association, pp. 225–243.

[13] C , L., D , J., G , L., J , M., K , B., N , G. Modula-3 report (revised). Tech. Rep. SRC Research Report 52, Digital Equipment Corporation, Systems Research Center, 1989.

[14] C , P. R., S , H., H , M. Flow-sensitive type analysis for C++. Tech. Rep. Research Report RC20267, IBM, 1995.

[15] C , P. P., M , S. A., C , W. Y., H , W. W. Profile-guided automatic inline expansion for c programs. *Software: Practice and Notices 22*, 5 (May 1992), 349–369.

[16] C , R., R , B. G., L , W. Complexity of concrete type-inference in the presence of exceptions. In *ESOP'98, 7th European Symposium on Programming, Proceedings (Lecture Notes in Computer Science, Vol. 1381)* (Berlin, Apr. 1998), Springer-Verlag, pp. 57–74.

[17] C , W. Y., C , P. P., C , T. M., H , W. W. The effect of code expanding optimizations on instruction cache design. *IEEE Trans. Comput. 42*, 9 (Sept. 1993), 1045–1057.

[18] C , J., G , D., H , M., S , V. Efficient and precise modeling of exceptions for the analysis of Java programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Programming analysis for software tools and engineering* (New York, NY, USA, Sept. 1999), ACM Press, pp. 21–31.

[19] C , M., L , G., S , J. M. Practicing JUDO: Java under dynamic optimizations. In ACM [2], pp. 18–21.

[20] D , C. C++ exception handling. *IEEE Concurrency 8*, 4 (2000), 72–79.

[21] D , J., C , C. Towards better inlining decisions using inlining trials. In *Proceedings of the 1994 ACM conference on LISP and functional programming* (New York, NY, USA, June 1994), ACM Press, pp. 273–282.

[22] D , J., D F , G., G , D., L , V., C , C. Vortex: an optimizing compiler for object-oriented languages. In *Proceedings of the 7th annual conference on Object-oriented programming systems, languages and applications (OOPSLA-1996)* (New York, NY, USA, Oct. 1996), ACM Press, pp. 83–100.

[23] D , J., G , D., C , C. Optimization of object-oriented programs using static class hierarchy. In *Proceedings of the 9th European Conference on Object-Oriented Programming - ECOOP '95 (Lecture Notes in Computer Science, Vol. 952)* (Berlin, Aug. 1995), Springer-Verlag, pp. 77–101.

[24] D , D., A , O. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming - ECOOP '99 (Lecture Notes in Computer Science, Vol. 1628)* (Berlin, June 1999), Springer-Verlag, pp. 258–278.

[25] D , S., G , K. J., L , J. Implementing zero overhead exception handling. Tech. Rep. Technical Report 95-12, Faculty of Information Technology, Queensland University of Technology, 1995.

[26] F , R., K , T. B., R , E., S , B., T , D. Marmot: An optimizing compiler for java. *Software: Practice and Notices 30*, 3 (Mar. 2000), 199–232.

[27] G , E. M., H , L. J., M , G. Efficient inference of static types for Java bytecode. In *Static Analysis. 7th international symposium, SAS 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1824)* (Berlin, June 2000), Springer-Verlag, pp. 199–219.

[28] G , N. H. Exceptional C or C with exceptions. *Software: Practice and Notices 22*, 10 (Oct. 1992), 827–848.

[29] G , J., J , B., S , G. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Aug. 1996.

[30] G , J., J , B., S , G. *The Java Language Specification*. In *The Java Series* [29], Aug. 1996, ch. 11.

[31] G , D., D , J. Profile-guided receiver class prediction. In ACM [1], pp. 108–123.

[32] G , M., C , J., H , M. Optimizing Java programs in the presence of exceptions. In *Proceedings of the 14th European Conference on Object-Oriented Programming - ECOOP '00 (Lecture Notes in Computer Science, Vol. 1850)* (Berlin, June 2000), Springer-Verlag, pp. 422–446.

[33] H , R. E., H , W. W., R , B. R. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th annual international symposium on Microarchitecture* (New York, NY, USA, Nov. 1995), ACM Press, pp. 158–168.

[34] H , J. Program optimization and exception handling. In *Conference record of the 8th annual ACM symposium on Principles of programming languages* (New York, NY, USA, Jan. 1981), ACM Press, pp. 200–206.

[35] H¨ , U., C , C., U , D. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming - ECOOP '91 (Lecture Notes in Computer Science, Vol. 512)* (Berlin, July 1991), Springer-Verlag, pp. 21–38.

[36] H¨ , U., C , C., U , D. Debugging optimized code with dynamic deoptimization. In *ACM SIGPLAN '92 Conference on Programming language design and implementation* (New York, NY, USA, June 1992), ACM Press, pp. 32–43.

[37] The IBM Developer Kit, Java 2 Technology Edition. http://www.ibm.com/developerworks/java/jdk/.

[38] *IBM VisualAge C++ for OS/2 Programming Guide*, third ed. In IBM [39], May 1995, ch. 14. Signal and OS/2 Exception Handling.

[39] *IBM VisualAge C++ for OS/2 Programming Guide*, third ed., May 1995.

[40] I , K., K , M., Y , T., K , H., N , T. A study of devirtualization techniques for a Java Just-In-Time compiler. In ACM [4], pp. 294–310.

[41] I , K., K , M., Y , T., T , M., O , T., S , T., O , T., K , H., N , T. Design, implementation, and evaluation of optimizations in a Java Just-In-Time compiler. *Concurrency: Practice and Experience 12*, 6 (2000), 457–475.

[42] K , M., K , H., N , T. Effective null pointer check elimination utilizing hardware trap. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)* (New York, NY, USA, Nov. 2000), ACM Press, pp. 118–127.

[43] K , A., S , B. Exception handling for C++ (revised). In *Proceedings of the C++ Conference* (Apr. 1990), USENIX Association, pp. 149–176.

[44] K , A., P , M. Monitors and exceptions: How to implement Java efficiently. In *ACM 1998 Workshop on Java for High-Performance Network Computing* (New York, NY, USA, 1998), ACM Press, pp. 15–24. Also published as *Concurrency: Practice and Experience*, **10**(11–13), September 1998, CODEN CPEXEI, ISSN 1040-3108.

[45] L , J., S , D. B. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Trans. Program. Lang. Syst. 20*, 2 (March 1998), 274–301.

[46] L , S., Y , B.-S., K , S., P , S., M , S.-M., E ˇ , K. Efficient Java exception handling in Just-in-Time compilation. In ACM [3], pp. 1–8.

[47] L , S., Y , B.-S., K , S., P , S., M , S.-M., E ˇ , K., A , E. On-demand translation of Java exception handlers in the LaTTe JVM just-in-time compiler. In *Proceedings of the 1999 Workshop on Binary Translation* (Oct. 1999).

[48] L , T., Y , F. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Sept. 1996, ch. 4.7.4.

[49] L , B., S , A. Exception handling in CLU. *IEEE Trans. Softw. Eng. 5*, 6 (1979), 546–558.

[50] M . MSDN Online. http://msdn.microsoft.com/.

[51] P , M., V , C., C , C. The Java HotSpot server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* (Apr. 2001), USENIX Association, pp. 1–12.

[52] P , J., S , M. I. Object-oriented type inference. In *Conference proceedings on Object-oriented programming systems, languages and applications* (New York, NY, USA, Oct. 1991), ACM Press, pp. 146–161.

[53] R , N., P J , S. A single intermediate language that supports multiple implementations of exceptions. In ACM [2], pp. 285–298.

[54] R , B. G., S , D., K , U., G , M., S , N. A static study of Java exceptions using JESP. Tech. Rep. dcs-tr-406, Rutgers University, Department of Computer Science, 1999.

[55] S , J. L. Optimizing away C++ exception handling. *ACM SIGPLAN Notices 33*, 8 (Aug. 1998), 40–47.

[56] S , S., H , M. J. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng. 42*, 9 (2000), 849–871.

[57] S , T., O , T., T , M., Y , T., K , M., I , K., K , H., N , T. Overview of the IBM Java Just-In-Time compiler. *IBM Syst. J. 39*, 1 (2000), 175–193.

[58] S , T., Y , T., K , M., K , H., N , T. A dynamic optimization framework for a Java Just-In-Time compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2001)* (New York, NY, USA, 2001), ACM Press.

[59] S . Java HotSpot technology. http://java.sun.com/products/hotspot/, Oct. 2000.

[60] T S P E C (SPEC). JVM Client98 (SPECjvm98). http://www.spec.org/osg/jvm98/, 1998.

[61] W , J. A portable sampling-based profiler for Java virtual machines. In ACM [3], pp. 78–87.