RT 0529 (04/03/2003)
Computer Science

IBM Research Report

# Handler Cloning for Optimizing Exception Handling

Takeshi Ogasawara

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

# Handler Cloning for Optimizing Exception Handling

Takeshi Ogasawara
IBM Tokyo Research Laboratory

**Abstract**

*Handler cloning* is a technique for optimizing exception handling. For a specific exception handler catching and rethrowing multiple classes of exceptions, it creates a clone of the original handler and registers the clone as a handler that catches only a single class or subset of the classes for the original handler. The rethrown exceptions from the clone are determined at the compile time.

## 1 Background

Language-supported exception handling mechanisms, such as in Ada [1], Modula-3 [2], and C++ [4], allow a programmer to write exception handlers for a region of the program. Java [3] is one of the modern languages that support exception handling. Using exceptions to change the control flows of the program is popular in Java [8, 9].

Optimizing exception handling is critical for programs that frequently throw exceptions. There are many exception-intensive Java programs [6] in various categories. These programs suffer from the overhead of exception handling. There are two sources for this overhead. One is the overhead of throwing the exceptions. The majority of this overhead consists of creating exception objects and initializing them. The other is the overhead of catching exceptions. The majority of this overhead consists of traversing the stack frame-by-frame and searching for the appropriate exception handler.

The problem is that we should avoid any penalty on *the normal path* while at the same time optimizing *the exception-handling path*. 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. For the existing techniques [7], *stack* *unwinding* and *stack cutting*, either the normal path or the exception-handling path can be optimized.

We proposed a novel approach, *exception-directed optimization (EDO)* [6]. During the program execution, the runtime part of EDO profiles the exception-handling path and requests the optimization of the Java methods that frequently throw and catch exceptions. The compiling part of EDO then optimizes the methods and removes the overhead for exception handling, for the throwing and catching of exceptions. For a standard benchmark suite, SPECjvm98, EDO significantly improved two exception-intensive tests, jack and javac, by 18.3% and 13.8%, respectively, with no degradation for exception-rare tests.

To remove the overhead of exception handling, EDO first inlines the methods on the target *exception path*. The exception path is the sequence of the method invocations when an exception is thrown and caught, which includes all the methods from the thrower to the catcher. EDO then analyzes the classes of the exceptions at the throwing points and finds their corresponding exception handlers. Finally EDO links the throwing points to the corresponding handlers. For the optimized exception paths, there is no overhead for catching exceptions. If the exception objects are not used, the code creating and initializing the objects is dead code and is eliminated by the compiler optimization. In that case, there is no overhead for throwing exceptions.

The problem is that EDO cannot find the corresponding handlers if the classes of the exceptions cannot be determined. A typical example is when `finally` blocks rethrow the exceptions that they have caught. The Java Virtual Machine Specification [5] defines that a special exception handler is created for each `finally` block that catches any classes of exceptions and rethrows them. Since the special handler can catch any class of exceptions, the analysis of the program by the JIT compiler cannot resolve the class of these exceptions. Therefore, even though the runtime part of EDO detects such a special handler as frequently executed, the compiling part cannot map the rethrown exceptions to the corresponding handler. For such special handlers, the overhead of exception handling remains. In general the same problem exists for any exception handlers that can catch two or more exception classes and rethrow the exceptions.

## 2 Handler Cloning

We propose *handler cloning* to enable EDO to optimize in the problematic case explained in the previous section. The idea is to create a clone of the target `finally` block and to register the clone as a new handler catching the exception classes that are actually known to be caught and rethrown by the `finally` block.

Figure 1 shows an example of how the handler cloning transforms the program to op-

```
 1|try {    /* first try block */
 2|    try {    /* second try block */
 3|        :
 4|        if (cond) {
 5|            throw new E();
 6|        }
 7|    } catch (Any e) {  /* finally */
 8|        /* some action 1 */
 9|        throw e;
10|    }
11|} catch (E e) {
12|   /* some action 2 */
13|}
```

(a) The original program

```
 1|  try {    /* first try block */
 2|      try {    /* second try block */
 3|          :
 4|          if (cond) {
 5|              throw new E();
 6|          }
 +|      } catch (E e) { /* clone */
 +|          /* some action 1 */
 +|          throw e;
 7|      } catch (Any e) { /* finally */
 8|          /* some action 1 */
 9|          throw e;
10|      }
11|  } catch (E e) {
12|      /* some action 2 */
13|  }
```

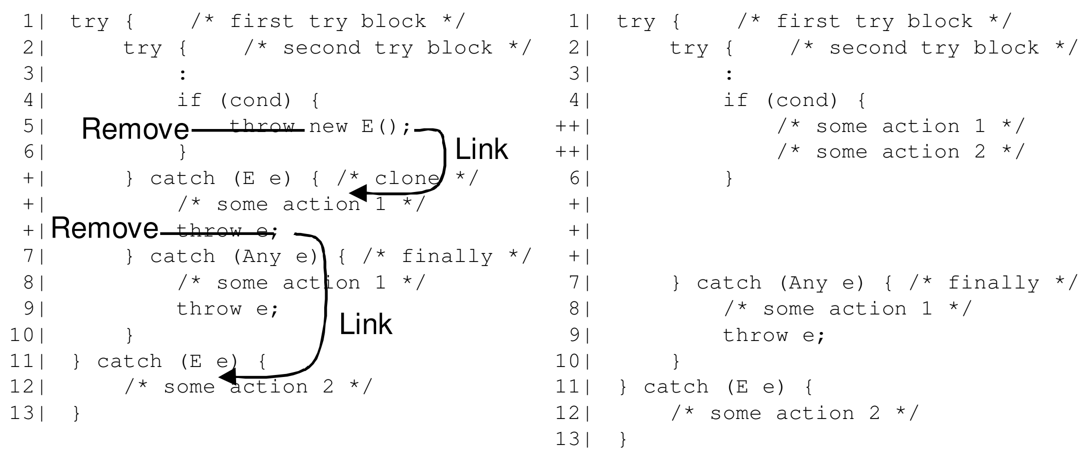
(b) Handler cloning

Figure 1: An example of handler cloning

timize the overhead of exception handling for a `finally` block. For the sake keeping the explanation simple, the exception handler corresponding to the `finally` block is explicitly drawn in this figure, though it is implicitly generated by the system [5] and does not appear at the source code level. We first explain why the overhead of exception handling still remains for the target `finally` block after performing EDO and then the handler cloning enables EDO to optimize this overhead.

Figure 1a shows a code fragment suffering from the overhead of exception handling. The exception handling is performed in this code as follows. The code creates and throws an exception of class `E` at Line 5, implicitly calling a method that saves the information of what methods have been invoked for the thread, `fillInStackTrace()`. If the exception is thrown, since Line 5 throwing the exception is surrounded by two `try` blocks, the inner or second `try` block is first examined. The handler for the `finally` block at Line 7 is associated with the second `try` block and is checked if it can catch the exception of class `E`. Since this handler can catch any exception class of exceptions, it is executed and the same exception is thrown again at Line 9 when the handler is finished. Next the outer or first `try` block is examined. The handler at Line 11 is associated with the first `try` block and is checked to see if it can catch the exception of class `E`. Since this handler is declared to catch exceptions of class `E`, it is executed.

For the original code, *exception-directed optimization (EDO)* [6] can analyze that the class of exceptions thrown at Line 5 is `E` and that these exceptions are caught by the finally block at Line 7. Then it removes the throwing code and links the throwing point (Line 5) to



```
 1|  try {     /* first try block */
 2|      try {     /* second try block */
 3|          :
 4|          if (cond) {
 5|              ~~throw~~ new E();
 6|          }
 +|      } catch (E e) { /* clone */
 +|          /* some action 1 */
 +|          ~~throw e;~~
 7|      } catch (Any e) { /* finally */
 8|          /* some action 1 */
 9|          throw e;
10|      }
11|  } catch (E e) {
12|      /* some action 2 */
13|  }
```

(a) Replace throw with goto

```
 1|  try {     /* first try block */
 2|      try {     /* second try block */
 3|          :
 4|          if (cond) {
++|              /* some action 1 */
++|              /* some action 2 */
 6|          }
 +|
 +|
 +|
 7|      } catch (Any e) { /* finally */
 8|          /* some action 1 */
 9|          throw e;
10|      }
11|  } catch (E e) {
12|      /* some action 2 */
13|  }
```

(b) Align the code and remove the unnecessary code

Figure 2: Optimization by EDO

the entry of the finally block (Line 7). However, EDO analyzes that the class of exceptions thrown at Line 9 can be any class. Therefore, it cannot match this throwing point to the handler catching only the exception class `E` at Line 11. As a result, since the exception object has to be created at Line 5 because the object is used for rethrowing at Line 9, the overhead of throwing exceptions remains. The overhead of catching exceptions also remains since the handler that can catch the exception thrown at Line 9 must be searched for.

Figure 1b shows the code after performing the handler cloning. The lines between 6 and 7 are added to the original Figure 1a. The handler cloning focuses on the `finally` block (Line 7) that throws the exception class `E` and another handler (Line 11) that catches the exception at runtime. The handler cloning generates a copy of the `finally` block and then registers the copy as an additional handler that catches the exception class `E` prior to the `finally` block. The additional handler also rethrows the exception.

Figure 2 shows the optimization of the example by using EDO. EDO can analyze that the class of exceptions thrown by the new handler created by the handler cloning (Lines denoted by +) is `E` and these exceptions are caught by the handler at Line 11. Therefore, it removes the throwing code and links the throwing points to the entries of the matching handlers, as shown in Figure 2a. Since EDO can determine that the exception created at Line 5 is not used, the code creating the exception can be eliminated. Figure 2b shows the optimized code after performing the handler cloning and EDO. Line 5 creating the exception is removed. The sequence of the code instructions executed when the program execution reaches Line 5 in Figure 2a appears between 4 and 6. By combining handler cloning with EDO, the overhead of exception handling for the example code is successfully removed.

## 3 Summary

In this paper, we explained the idea of handler cloning to address the problem where EDO cannot optimize the overhead of exception handling because the problem is throwing a larger set of exception classes than the handler can catch. Combining handler cloning with EDO, we showed that EDO can successfully remove this overhead.

## References

[1] Baker, T. P., and Riccardi, G. A. Implementing Ada exceptions. *IEEE Software 3*, 5 (Sept. 1986), 42–51.

[2] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report (revised). Tech. Rep. SRC Research Report 52, Digital Equipment Corporation, Systems Research Center, 1989.

[3] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Aug. 1996.

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

[5] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Sept. 1996, ch. 4.9.6.

[6] Ogasawara, T., Komatsu, H., and Nakatani, T. A study of exception handling and its dynamic optimization in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2001)* (New York, NY, USA, 2001), ACM Press, pp. 83–95.

[7] Ramsey, N., and Peyton Jones, S. A single intermediate language that supports multiple implementations of exceptions. In *ACM SIGPLAN '00 Conference on Programming language design and implementation* (New York, NY, USA, May 2000), ACM Press, pp. 285–298.

[8] Ryder, B. G., Smith, D., Kremer, U., Gordon, M., and Shah, N. A static study of Java exceptions using JESP. Tech. Rep. dcs-tr-406, Rutgers University, Department of Computer Science, 1999.

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