Vol. 47 No. SIG 6(PRO 29) IPSJ Transactions on Programming May 2006

Regular Paper

Efficient and Portable Implementation of Java-style Exception Handling in C

Seiji Umatani,† Hirokazu Shobayashi,†† Masahiro Yasugi† and Taiichi Yuasa†

An important issue in implementing high-level programming languages for use by translators into C is how to support high-level language features not available in C. Java's exception handling is one such feature, and translating it into portable C, which only uses C-style control structures, involves some challenges. Previous studies have proposed ways of translating the Java-style `try-catch` construct into C. In this paper, we propose a new scheme for implementing it in an efficient and portable manner. We use our parallel language OPA, which is an extended Java language, and its translator. In our scheme, we do not use troublesome `setjmp/longjmp` routines for non-local jumps. Instead, we check the occurrences of exceptions using functions' return values. Java also has the `try-finally` construct, mainly used for cleaning up, which cannot be translated directly into C-style control structures. To implement it, we developed a new scheme with integer values corresponding to continuation targets. Compared with other techniques, ours has advantages in both the runtime overhead and the generated code size. For these two features, using some benchmark programs, we measured the performance of our scheme and compared it with those of some other schemes.

1. Introduction

Java [6] is an object-oriented language designed by Sun Microsystems that supports portable code, i.e., bytecode that runs on a variety of platforms. Java's portability is achieved by compiling its source programs into a distribution format called a class file. A class file contains information about the Java class, including bytecode that is an architecturally independent representation of the instructions associated with the class's methods. A class file can be executed on any computer supporting the Java Virtual Machine (JVM). Java's code mobility, therefore, depends on both architecture-independent class files and the implicit assumption that the JVM is supported on every client machine.

Most JVM implementations execute bytecode through interpretation or Just-In-Time (JIT) compilation, which compiles the bytecode into machine code at run time. Thus, Java's portability comes at a price, namely, the cost of interpreting or JIT-compiling the bytecode every time the program is executed. This incurs a significant runtime overhead, which is unnecessary in applications running many times without change.

† Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University
†† NTT West Co., Ltd.

To overcome these inherent performance penalties, there are many Java implementations that pre-compile Java programs (or bytecode) into machine code [7),8),12),13)]. Some of these systems first translate Java programs (or bytecode) into C code, then compile the C code into machine code, using existing C compilers. These systems compile Java programs into machine code during program development, eliminating the need for interpretation or JIT compilation of bytecode.

Java-to-C translators have several advantages over interpretation or JIT-compilation. First, because they can employ optimizing C compilers as their backend, highly efficient executables can be generated at a low development cost. Moreover, because they create a C-equivalent to the Java program, the standard C debugging and profiling tools can be used for these executables. Second, because such an executable usually includes all the information about the class files it uses, there is no possibility of an application suddenly ceasing to execute because of a change in available class files. For these reasons, we believe that ahead-of-time translation into C is valuable for the development of efficient Java programs.

In this paper, we use our language OPA [15),16)] (Object-oriented language for PArallel processing), which is an extended Java language. In order to write parallel programs easily, we remove specifications about threads from Java,

```
void tryCatchFinally() {
  try {
    ⟨try-body code⟩
  } catch (MyExcep e) {
    ⟨handler code for MyExcep class⟩
  } finally {
    ⟨finally-body code⟩
  }
}
```

Fig. 1 A sample `try-catch-finally` method.

then add new features such as structured multi-threading constructs for OPA. OPA is intended for high-performance parallel processing, and therefore our OPA implementation translates OPA programs into C for efficiency.

Java's exception handling is one of the features that does not directly correspond to C-style control structures; thus, translating it involves some challenges. In Java, exception handling is described by a `try-catch-finally` construct (whose syntax is shown in **Fig. 1**). In *Java Language Specification* [6], it is defined as follows:

> When an exception is thrown, control is transferred from the code that caused the exception to the nearest dynamically-enclosing `catch` clause of a `try` statement that handles the exception.
>
> A statement or expression is *dynamically enclosed* by a `catch` clause if it appears within the `try` block of the `try` statement of which the `catch` clause is a part, or if the caller of the statement or expression is dynamically enclosed by the `catch` clause.
>
> ...
>
> In situations where it is desirable to ensure that one block of code is always executed after another, even if that other block of code completes abruptly, a `try` statement with a `finally` clause may be used.

Java's `try-catch-finally` construct:

```
try{
  ...
}catch(Excep e){
  ...
}finally{ ... }
```

can be separated into two constructs like:

```
try{ try{
       ...
     }catch(Excep e){ ... }
}finally{ ... }
```

Therefore, we discuss only the implementation of `try-catch` and `try-finally` in the remainder of this paper.

To implement the `try-catch` construct in C, we do not use non-local jumps such as `setjmp/longjmp`, since this would incur some undesirable overhead. Furthermore, we implement the `try-finally` construct neither by using an independent subroutine nor by inlining the `finally` body at all points where execution of the `finally` body is needed. The former has some overhead for each function call, while the latter may result in a huge amount of code, and therefore we propose other schemes for implementing them. Our schemes do not incur unnecessary runtime overhead and do not increase its code size superfluously. They also have portability; that is, the generated C code can be compiled and executed in environments where the standard C language is supported.

The rest of this paper is organized as follows. Section 2 describes our implementation of the `try-catch` construct and Section 3 describes our implementation of the `try-finally` construct. In Section 4, we describe the results of benchmark tests for some programs.

2. Implementation of `try-catch`

In this section, we explain how our OPA compiler translates `try-catch` statements and `throw` statements into C. In our scheme, we do not use troublesome `setjmp/longjmp` routines for non-local jumps, and the generated C code fully confirms to the ANSI C standard.

Exceptions are handled through the following steps:

(1) If an exception can be caught within the current method, control is transferred to the corresponding handler code.
(2) Otherwise, the method throws the uncaught exception toward its caller.
(3) The above steps are repeated until the exception is caught.

We explain how case (1) is implemented in Section 2.1 and case (2) in Section 2.2. Related work is discussed in Section 2.3.

2.1 Exceptions within a Method

Within the C function that is generated for a method, the control is transferred to exception handlers simply by using `goto` statements when an exception is raised. For example, the following code:

```
...;
ex = ⟨code for creating Excep1 object⟩;
goto CATCH_0;
...;
if(0){
  CATCH_0:
  if(instanceof(ex, c__Excep1)){
    ⟨code for handler 1⟩
  }else if(instanceof(ex, c__Excep2)){
    ⟨code for handler 2⟩
  }else{
    ⟨code for rethrowing the exception⟩
  }
}
```

Fig. 2 The C code generated for `try-catch` in our scheme.

```
try{
  ...; throw new Excep1(); ...
} catch(Excep1 e){
  ⟨code for handler 1⟩
} catch(Excep2 e){
  ⟨code for handler 2⟩
}
```

is translated into the code shown in **Fig. 2**. The local variable `ex` is used for saving the exception. `c__Excep1` is the class descriptor for the class `Excep1`, and `instanceof()` checks whether `ex` can be caught by the corresponding `catch` clause. If no handler can catch the exception, the exception is rethrown in the same manner.

If a `throw` statement is in the `try` body of a `try-finally` statement, not of a `try-catch` statement, its `goto` target becomes the label corresponding to the `finally` clause. Implementation details of `try-finally` are explained later, in Section 3.4.

2.2 Exceptions across Method Call

As mentioned previously, OPA provides simple and efficient multithreading features. To achieve efficiency, each thread in OPA normally executes on the C stack. When the current thread running on the stack is suspended, its method frames are allocated in the heap area and the current states of the stack frames are saved there. To realize this, each method on the stack must inform its parent method (caller) that it has been suspended and its state is saved into a certain frame in the heap; here, the address of the callee's frame must be known to the caller, so that method frames within one thread are organized as a linked list in the heap.

Therefore, a callee must return a pointer to its own heap frame (if it exists; otherwise it returns 0) along with the method's return value. In OPA, the pointer may be returned as a return value of a C function call, and the method's return value may be stored in another place:

```
child_fr = f__foo(pr, ...);
if(child_fr != 0){
  ⟨code for suspension⟩
} else {
  ret = pr->ret;
}
```

Here, `pr` points to the per-processor data area and the method's return value is stored in it (`pr->ret`). This means that each method call must be followed by this memory access. To reduce this overhead, the actual OPA system returns [16] the method's return value as a return value of the C function call, and the special value `SUSPEND` (which is selected from rarely used values, e.g., `-5`) indicates the suspension of the callee. In such a case, the system checks further to determine whether a pointer to the callee frame is stored in a fixed place (`pr->child_fr`):

```
ret = f__foo(pr, ...);
if(ret==SUSPEND && pr->child_fr!=0){
  ⟨code for suspension⟩
}
```

Our scheme involves quite a small overhead, because the return value of a C function is usually in a register and, in most cases, the caller executes only one additional branch instruction.

The above technique can also be applied for propagating an exception accross a method call. That is, if a callee completes abruptly because of an exception, it returns `SUSPEND` (and `pr->child_fr` is set to any non-zero value), and the uncaught exception is stored in `pr->ex`:

```
ret = f__foo(pr, ...);
if(ret==SUSPEND && pr->child_fr!=0){
  if(pr->ex){
    ⟨code for rethrowing the exception⟩
  }
}
```

2.3 Related Work

There are three existing schemes for implementing `try-catch` statements:

- use static `try` block tables and search these tables at runtime,
- dynamically create a linked list of `try` block data structures using `setjmp/longjmp`,

- use functions with two return values.

The first scheme is used in most JVM interpreters and C++ compilers [9]. In this scheme, each exception handler is recorded as a single entry of the table, specifying the range of offsets into the JVM code implementing the method for which the exception handler is active, describing the type of exception that the exception handler is able to handle, and specifying the location of the code that is to handle that exception. Here, offsets and locations are specified as program counters, and JVM can easily interpret these tables and transfer control to the corresponding handler code. However, since the program counter is not visible in C, this scheme is not suitable for Java-to-C translators. Another approach is to realize our compiler as a Java-to-C++ translator. In Section 4, we measure the performance of C++'s exception programs and compare it with those of our scheme in C.

The second scheme is used in C++ [1] and Ada [5]. A linked `try-catch` data structure is created when entering a `try` block, and the structure is discarded when leaving the `try` block. When an exception is raised during the time that some method is called in a `try` block, the callee-save registers must be restored to their original values. The data structure can be used to store such information. This scheme can be applied to Java-to-C translators, but its disadvantage is that creating and discarding the data structure incurs some runtime overhead, even if an exception is never raised.

The third scheme is used in Java2C [2] and the early implementation of CACAO [10]. In this scheme, a function has two return values: one for the method's return value and the other for an exception, as in our scheme. In CACAO, an register is exclusively dedicated to storing an exception. After each method call, the exception register is checked and, if it is nonzero, the exception handling code is executed. In Java2C, instead of reserving the proprietary register for exception handling, a raised exception is stored in the thread object of the current thread. In both cases, unlike our OPA implementation, the caller always checks the return value for an exception: CACAO always reserves one register (which is impossible in portable C) and Java2C requires some memory accesses even if an exception is never raised. In contrast, in our OPA implementation, the caller checks only the callee's return value and performs further checking (memory accesses) only when the callee might throw an exception.

3. Implementation of `try-finally`

In this section, we explain four schemes used for compiling Java's `try-finally` statements into C. Since all of these schemes except for ours emulate the JVM's bytecode implementation of `try-finally` statements, we first review the JVM's scheme.

The Java Virtual Machine Specification [11] includes the `jsr` (Jump SubRoutine) instruction and the `ret` (RETurn from subroutine) instruction for the implementation of `try-finally` statements. When '`jsr` ⟨*offset*⟩' is executed, control is transferred to that ⟨*offset*⟩ from the address of this `jsr` instruction. At the same time, the address of the instruction immediately following this `jsr` instruction is pushed onto the operand stack. Note that the target address specified by the ⟨*offset*⟩ must be within the method that contains this `jsr` instruction. When '`ret` ⟨*i*⟩' is executed, the contents of the local variable ⟨*i*⟩ are used for the return address; that is, they are written into the JVM's program counter register and execution continues there.

For example, a typical `try-finally` statement is compiled to the code shown in **Fig. 3**. The control flow of this code is shown in **Fig. 4**. There are four ways for control to pass outside of (or escape from) the `try` statement: by falling through the bottom of that block, by raising an exception, by returning, or by executing a `break` or `continue` statement. (In this example, only the former two cases are described, but the others can be compiled in a similar manner.)

```
 1  ⟨try-body code⟩
 2  jsr FIN_BODY     ; call finally block
 3  return           ; method return
 4  astore_1         ; rethrow handler
                     ; save thrown value
 5  jsr FIN_BODY     ; call finally block
 6  aload_1          ; restore thrown value
 7  athrow           ; rethrow the value
 8 FIN_BODY:
 9  astore_2         ; save return address
10  ⟨finally-body code⟩
11  ret 2            ; return from FIN_BODY
```

Fig. 3 Bytecode generated from a typical `try-finally` statement.

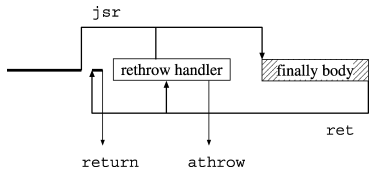


Fig. 4 Control flow of `try-catch-finally` code.

When the `try` body completes normally, a subroutine call for the `finally` body is executed at line 2. At line 2, the `jsr` instruction pushes the address of the following instruction (`return` at line 3) onto the operand stack before jumping. The `astore_2` instruction at line 9 saves the address on the operand stack into local variable 2. The code for the finally block (line 10) is executed. Assuming that execution of that code completes normally, the `ret` instruction retrieves the address from local variable 2 and resumes execution at that address. The method then returns normally at line 3.

If an exception is thrown during the execution of the `try` body, since there is no exception handler inside this `try-finally` statement, control is transferred to the *rethrow handler* at lines 4–7. In this case, the `finally` body is also called from line 5 before rethrowing the exception at line 7. Local variable 1 is used for saving the exception during the execution of the `finally` body.

3.1 Function Call

The first scheme is based on the straightforward observation that the `jsr` and `ret` instructions of JVM are almost the same as the machine instructions used by C compilers to translate C's function calls (e.g., `call` and `ret` instructions in the IA-32 architecture). (The only difference between them is that, in JVM, the target addresses of an `jsr` instruction must be within the method that contains this instruction.) Thus, in Java-to-C translators, it seems to be natural to realize a `try-finally` statement with the function definition which contains the corresponding `finally` body code (in C) and with calls to this function.

The drawbacks of this scheme arise from the difference between the `jsr` instruction and C function calls. First, a C function call contains a certain amount of runtime overhead: some register values must be saved into the C stack at the call and restored at its return, the stack pointer (and the frame pointer) may be adjusted, and so on. In addition to these overheads, if `try-catch` statements are implemented with the two return value schemes described in Section 2.3 or our scheme in Section 2.2, an exception check must be performed after each function call, since the function may throw an exception. Second, in order to access local variables of the caller from inside the callee, their addresses must be passed as parameters to it. This may prevent these variables from remaining in registers, thus causing performance degradation.

```
void sample(int p) {
  int a = 0;
  try {
    try {
      if (p == 0) return;
      f();  // may throw an exception
    } finally { // at level 1
      a += 2;
    }
  } finally { // at level 0
    a += 1;
  }
  return;
}
```

Fig. 5 Nested `try-finally` code.

3.2 Inline Expansion

To overcome the drawbacks of the function call scheme described in the previous section, simply inlining `finally` bodies at `jsr`'s call sites is effective, especially since:

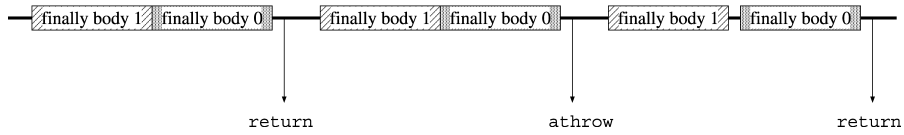
- it can eliminate the function call overhead,
- local variables can be directly accessed within `finally` bodies.

This scheme is used in the Java2C [2] translator.

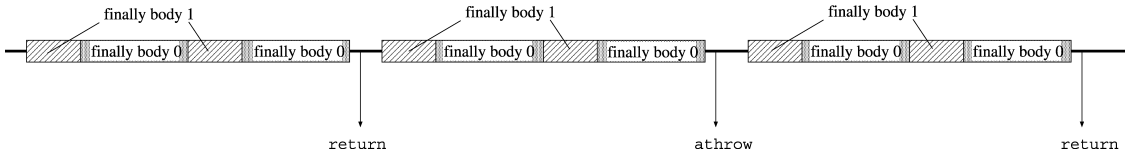
As a simple example, the Java code in **Fig. 5** is translated as in **Fig. 6** (a). Clearly, the inline expansion scheme has a considerable disadvantage in terms of code size. Moreover, if a certain `finally` body contains several escape points, all the outer finally bodies are expanded into these points. For instance, if `finally` body 1 contains a `throw` (or `return`, `break`, `continue`) statement in its middle, the generated code expands further, as shown in Fig. 6 (b). At a rough estimate, assuming that the nested level of `try-finally` statements is N and that every `finally` block contains k escape points, $O(k^N)$ pieces of the expanded `finally` bodies are included in the generated code.

3.3 GCC's Label Value

The third scheme employs one of the GCC's C extensions [3]: labels as values. We can obtain



(a) Inline expansion



(b) Recursive inline expansion

Fig. 6 Control flow of nested `try-finally` code.

the address of a label defined in the current function by using the unary operator '`&&`', and we can use label values with the computed `goto` statement, '`goto *`⟨*label val*⟩'.

Using this extension, GCJ [14] emulates JVM's `jsr` and `ret` instructions; the control flow of the code generated by GCJ is the same as JVM's. More precisely, '`jsr` ⟨*target address*⟩' is translated as follows:

```
  void *ra0 = &&CONT_0;
  goto ⟨target label⟩;
CONT_0: ...
```

where `CONT_0` is the return address of this code, i.e., the continuation of `jsr`. The variable `ra0` is prepared by the compiler for each finally block. The `ret` instruction is more straightforward:

```
  goto *ra0;
```

This scheme is simple and efficient. However, it has two disadvantages. First, the generated C code is not portable: it relies on the GCC's extension. Consequently, many other optimizing compilers seem to be unable to compile it.

Second, because the target addresses of computed `goto` statements cannot be decided at compile-time, the compiler must assume that each computed `goto` statement may jump to all labels in the current function. Thus, if multiple `finally` blocks exist in the same method, their sets of return addresses are indistinguishable from one another. This can have negative effects on the compiler's control flow analyses, such as liveness analysis, register allocation, and so on.

3.4 Our Scheme

The GCJ's problem is that continuation targets after the execution of `finally` blocks cannot be hard coded into label identifiers (constants). We solved this problem by representing continuation targets not as label values but as integer values. Furthermore, after executing a `finally` block, we know that the execution does not need to be resumed at its caller site (escape point); instead, control is directly transferred outside of the `try-finally` statement, where the exact target statement is determined according to the way of escape from the `try` block.

For each of the four ways of escape, our scheme generates the C code as follows:

(1) If the `try` statement completes normally, the `finally` block is executed and control is simply transferred to the next C code of the `try-finally` where the Java statement following the `try-finally` is translated. That is,

```
try {
  ⟨try-body⟩
} finally {
  ⟨finally-body⟩
}
```

is simply translated as

```
  ⟨C code for try-body⟩
FIN_BEGIN:
  ⟨C code for finally-body⟩
```

(2) If a `return` statement is performed inside the `try` block, the return value is saved, and then control is transferred to `FIN_BEGIN`. After the execution of the `finally` block, the function returns with the saved value.

```
try {
  ...; return x; ...
} finally {
  ⟨finally-body⟩
}
```

is translated as

```
...;
ret = x; goto FIN_BEGIN;
...;
```

```
FIN_BEGIN:
  ⟨C code for finally-body⟩
  return ret;
```

Note that there is a single `ret` variable in each function, since it cannot have multiple return values.

(3) If a `break` (or `continue`) statement is performed inside the `try` block and if there are any `try-finally` statements within the `break` (or `continue`) target whose `try` blocks contain the `break` (or `continue`) statement, then any `finally` blocks of those `try-finally` statements must be executed before control is transferred to the `break` (or `continue`) target. In our OPA implementation, `break` and `continue` statements are implemented with labels and `goto`s. We change these `goto`s' target to the innermost `finally`'s `FIN_BEGIN`. The original target label is saved (precisely speaking, a non-negative integer value is assigned for each label) and used after the execution of all `finally` blocks. For instance,

```
while(true) {
  try {
    ...; break; ...
  } finally {
    ⟨finally-body⟩
  }
}
```

is translated as

```
while(TRUE) {
  ...;
  gt = 3; goto FIN_BEGIN;
  ...;
 FIN_BEGIN:
  ⟨C code for finally-body⟩
  if(gt == 3)
    goto L3;
}
L3: ...
```

where the next statement of the `while` loop is labeled as `L3`.

(4) If an exception is thrown during the execution of the `try` block, the exception is saved, and then control is transferred to `FIN_BEGIN`. After the execution of the `finally` block, the exception is rethrown. For instance,

```
void f() throws Excep {
  try {
    ...; throw new Excep(); ...
  } finally {
    ⟨finally-body⟩
  }
}
```

is translated as

```
f_frame *f() {
  ...;
  ex0 = pr->ex; goto FIN_BEGIN;
  ...;
 FIN_BEGIN:
  ⟨C code for finally-body⟩
  pr->ex = ex0; return SUSPEND;
}
```

The integer values mentioned above are used to distinguish all these cases, and also for the `gt` variable in the case of `break` (or `continue`). An integer value i is set in the `fin_flag` variable before jumping to `FIN_BEGIN`, and its meaning is interpreted as follows (`NORMAL`, `RETURN`, `EXCEP` are defined as negative constants):

i) if $i =$ `NORMAL`, it means that the `try` block completes normally.
ii) if $i =$ `RETURN`, it means that a `return` statement is performed.
iii) if $i =$ `EXCEP`, it means that an exception is thrown.
iv) if $i \geq 0$, a `break` or `continue` is performed. Its target is labeled as '`L`i`:`'.

Putting all this together, the sample code in Fig. 5 is translated into the code shown in **Fig. 7**, and its control flow is shown in **Fig. 8**. It is conceivable that `try` statements usually complete normally. Thus, at the end of each `finally` block in the generated code, `fin_flag` is first compared with `NORMAL` before being checked with `switch` statements in order to avoid the overhead of executing `switch` statements.

4. Performance

In this section, we give measurement results for some programs that are influenced by implementation schemes of the `try-catch` and `try-finally` constructs. The configuration of the shared-memory parallel computer used for these measurements is shown in **Table 1**.

First, we measured the overhead caused by exception checks, which are necessary for implementing the `try-catch` construct in OPA. For comparison, we use two OPA implementations, one employing our proposed scheme, i.e., checking with `SUSPEND` ("fast"), the other always checking `pr->ex` ("slow"). For benchmark programs, we used four Cilk[4] multithreaded

```
void f__sample(penv *pr, int p){
  int a = 0; int fin_flag;
  void *ex0, *ex1;
   if (p == 0){
    fin_flag = RETURN;
    goto FIN_BEGIN1:
  }
  ret = f();
  if(ret==SUSPEND && pr->child_fr)
    if(pr->ex){
      fin_flag = EXCEP;
      ex1 = pr->ex;
      goto FIN_BEGIN1;
    }
  fin_flag = NORMAL;
 FIN_BEGIN1:
  a += 2;
  if(fin_flag!=NORMAL){
    switch(fin_flag){
      case RETURN:
        goto FIN_BEGIN0;
      case EXCEP:
        ex0 = ex1;
        goto FIN_BEGIN0;
    }
  }
 FIN_BEGIN0:
  a += 1;
  if(fin_flag!=NORMAL){
    switch(fin_flag){
      case RETURN:
        return;
      case EXCEP:
        pr->ex = ex0;
        return SUSPEND;
    }
  }
  return;
}
```

Fig. 7 Code generated by our scheme for `try-finally`.

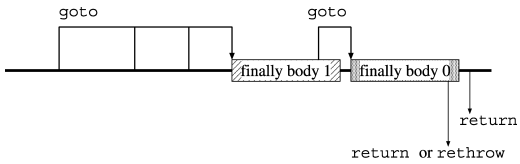


Fig. 8 Control flow of `try-finally` code generated by using our scheme.

programs (`fib`, `knapsack`, `cilksort`, `matmul`) by porting them into OPA. The results are shown in the upper part of **Table 2**. The results show that our scheme is able to reduce the overhead for exception checks considerably.

Even in other languages where no suspension check is carried out after each method call, our techniques (both "fast" and "slow") may be used to implement the `try-catch` construct. To estimate the overhead of exception checks in those languages, we prepared the sequential OPA implementation, which does not generate multithreaded code (e.g., suspension checks). Using this sequential implementation, we also measured the overhead caused by exception checks. The lower part of Table 2 shows the results: "nochk" without exception checks, "fastchk" with fast version exception checks, and "slowchk" with slow version exception checks. From the results, we can show that the overhead of slow version exception checks in the sequential implementation is greater than that of fast version ("exception checks").

Second, we compared the performance of our `try-catch` implementation in C (fast version) with C++'s `try-catch` implementation. The C++ compiler is g++ (gcc) 3.0.3. It uses the first scheme described in Section 2.3; that is, an exception table and a hardware PC are employed. We used a simple `try-catch` program that repeatedly throws and catches exceptions ("throw"). We also used a program which is almost the same except that it does not throw any exception ("no throw"). The results are shown in **Table 3** (1, 2, and 4 indicate the number of `try-catch` nestings). In the case of "no throw," our scheme is slightly slower than C++ because of the time spent on exception checks. However, in the case of "throw", the C++ program requires an amount of time about three orders of magnitude larger than that needed for "no

Table 1 Computer settings.

Machine	Sun Fire 3800
CPU	Ultra SPARC III 750 MHz, 8 MB L2 cache
Main memory	6 GB
Compiler	gcc 3.0.3 (with `-O3 -mcpu=ultrasparc` option)

Table 2 Execution times of `try-catch` benchmarks (sec).

	fib	knapsack	cilksort	matmul
fast	1.42	7.13	2.35	9.26
slow	1.58	7.26	2.52	9.44
nochk	1.41	5.92	2.31	9.15
fastchk	1.42	6.10	2.37	9.17
slowchk	1.48	6.30	2.43	9.30

Table 3 Comparison of `try-catch` with C++ (msec).

		1	2	4
throw	fast	4.48	5.16	6.24
	C++	1730	2750	4750
no throw	fast	0.88	1.18	1.77
	C++	0.72	0.99	1.52

Table 4 Execution times for three `try-finally` programs (μsec).

	A	B	C
func	5.57	3.00	427.4
inline	0.49	3.04	86.5
lvalue	1.23	3.01	55.1
intval	1.55	2.95	52.3

throw", while in our scheme the cost of throwing an exception is relatively low.

Third, we measured the execution time of three `try-finally` programs. The behavior of these programs can be summarized as follows:

prog. A: it does no useful work in the `finally` block.

prog. B: it calls a method in the `finally` block.

prog. C: it performs a `while` loop in the `finally` block.

We use four OPA implementations: function call ("func"), inline expansion ("inline"), gcc's label value ("lvalue"), and our proposed scheme ("intval"). The results are shown in **Table 4**. From the results, we can show that, in terms of execution time, the function call scheme clearly has some disadvantages, especially for programs A and C. It seems that the cost of accessing local variables via the pointers to them from inside the called function is relatively high. In contrast, the inline expansion scheme achieves good execution time results. However, it has some disadvantages in terms of code size. Our scheme achieves relatively good performance results in both contexts.

5. Conclusions

In this paper, we have proposed efficient and portable implementation techniques for Java-style exception handling in Java-to-C translators. A Java-style `try-catch` construct has been implemented efficiently; it incurs no overhead except for simple checks, which are already included in our multithreading implementation, if an exception is never raised. A Java-style `try-finally` construct can be implemented in several ways, and our proposed scheme has various benefits in terms of its portability, runtime overhead, and generated code size.

References

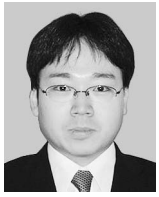
1) Cameron, D., Faust, P., Lenkov, D. and Mehta, M.: A Portable Implementation of C++ Exception Handling, *C++ Technical Conference*, USENIX, pp.225–243 (1992).
2) Chiba, Y.: Implementation of Exception Handling in a Java2C Translator, *IPSJ Transactions on Programming*, Vol.42, No.SIG11 (PRO 12), pp.14–24 (2001) (in Japanese).
3) Free Software Foundation, Inc.: *Using the GNU Compiler Collection*, for gcc 3.4.3 edition (2004).
4) Frigo, M., Leiserson, C.E. and Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language, *ACM SIGPLAN Notices* (*PLDI '98*), Vol.33, No.5, pp.212–223 (1998).
5) Giering, E.W., Mueller, F. and Baker, T.P.: Features of the Gnu Ada Runtime Library, *TRI-Ada '94*, ACM, pp.93–103 (1994).
6) Gosling, J., Joy, B., Steele, G. and Bracha, G.: *Java Language Specification*, Second Edition, Addison-Wesley Longman Publishing Co., Inc. (2000).
7) Hsieh, C.-H.A., Conte, M.T., Johnson, T.L., Gyllenhaal, J.C. and Hwu, W.W.: Optimizing NET Compilers for Improved Java Performance, *Computer*, Vol.30, No.6, pp.67–75 (1997).
8) Hsieh, C.-H.A., Gyllenhaal, J.C. and Hwu, W.W.: Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results, *MICRO 29: Proc. 29th Annual ACM/IEEE International Symposium on Microarchitecture*, IEEE Computer Society, pp.90–99 (1996).
9) Koenig, A. and Stroustrup, B.: Exception Handling for C++, *Journal of Object Oriented Programming*, Vol.3, No.2, pp.16–33 (1990).
10) Krall, A. and Grafl, R.: CACAO — A 64-Bit JavaVM Just-in-Time Compiler, *Concurrency: Practice and Experience*, Vol.9, No.11, pp.1017–1030 (1997).
11) Lindholm, T. and Yellin, F.: *Java Virtual Machine Specification*, Second Edition, Addison-Wesley Longman Publishing Co., Inc. (1999).
12) Muller, G., Moura, B., Bellard, F. and Consel, C.: Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code, *Proc. Third USENIX Conference on Object-Oriented Technologies and Systems* (*COOTS*), pp.1–20 (1997).
13) Proebsting, T.A., Townsend, G.M., Bridges, P.G., Hartman, J.H., Newsham, T. and Watterson, S.A.: Toba: Java for Applications — A Way Ahead of Time (WAT) Compiler, *Proc. Third USENIX Conference on Object-Oriented Technologies and Systems* (*COOTS*), pp.41–54 (1997).
14) Tromey, T.: GNU gcj, http://gcc.gnu.org/java/ (2002).
15) Umatani, S., Yasugi, M., Komiya, T. and Yuasa, T.: Lazy Normalization Techniques for

an Object-Oriented Parallel Language OPA, *IPSJ Transactions on Programming*, Vol.45, No.SIG5 (PRO 21), pp.12–25 (2004) (in Japanese).

16) Yasugi, M., Umatani, S., Kamada, T., Tabata, Y., Ito, T., Komiya, T. and Yuasa, T.: Code Generation Techniques for an Object-Oriented Parallel Language OPA, *IPSJ Transactions on Programming*, Vol.42, No.SIG11 (PRO 12), pp.1–13 (2001) (in Japanese).

(Received February 21, 2005)
(Accepted July 1, 2005)

Seiji Umatani was born in 1974, and received a B.E. degree in informatics and mathematical science, and M.E. and Ph.D. degrees in informatics from Kyoto University, Kyoto, Japan, in 1999, 2001, and 2004, respectively. Since 2004, he has been a research staff member in the Graduate School of Informatics at Kyoto University, and he was appointed to an assistant professor in 2005. His current research interest includes programming languages, compilers, and parallel/distributed systems. He is a member of the ACM and the Japan Society for Software Science and Technology.



Hirokazu Shobayashi was born in 1979. He received a B.E. in Information Science in 2002 and the Master of Informatics degree in 2004, both from Kyoto University, Kyoto, Japan. He is currently working at NTT West Co., Ltd. Shizuoka Branch. His research interests include programming languages and parallel processing.

Masahiro Yasugi was born in 1967. He received a B.E. in electronic engineering, an M.E. in electrical engineering, and a Ph.D. in information science from the University of Tokyo in 1989, 1991, and 1994, respectively. In 1993–1995, he was a fellow of the JSPS (at the University of Tokyo and the University of Manchester). In 1995–1998, he was a research associate at Kobe University. Since 1998, he has been working at Kyoto University as an assistant professor (lecturer) and an associate professor (since 2003). In 1998–2001, he was a researcher at PRESTO, JST. His research interests include programming languages and parallel processing. He is a member of the ACM and the Japan Society for Software Science and Technology.

Taiichi Yuasa received a Bachelor of Mathematics degree in 1977, the Master of Mathematical Sciences degree in 1979, and a Doctor of Science degree in 1987, all from Kyoto University, Kyoto, Japan. He joined the faculty of the Research Institute for Mathematical Sciences, Kyoto University, in 1982. He is currently a Professor at the Graduate School of Informatics, Kyoto University, Kyoto, Japan. His current areas of interest include symbolic computation and programming language systems. He is a member of the ACM, the IEEE, the Institute of Electronics, Information and Communication Engineers, and the Japan Society for Software Science and Technology.