SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2001; **31**:1109–1123 (DOI: 10.1002/spe.404)

Exploiting exceptions

Michael Zastre*,† and R. Nigel Horspool

Department of Computer Science, University of Victoria, P.O. Box 3055 STN CSC, Victoria, B.C., Canada V8W 3P6

SUMMARY

A novel compiler optimization for loops is presented. The optimization uses exceptions to eliminate redundant tests that are performed when code is interpretively executed, as is the case with Java bytecode executed on the Java Virtual Machine. An analysis technique based on abstract interpretation is used to discover when the optimization is applicable. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: exceptions; Java; bytecode; virtual machines; optimization; abstract interpretation

INTRODUCTION

Most programmers use exceptions to handle exceptional events. However, exceptions can also be used to simplify the control logic of a program and enhance readability [1]. As we will demonstrate, there are situations where exceptions can be used as a standard programming pattern to make programs execute faster. These situations are opportunities to increase the speed of Java programs by changing bytecode within methods with a space cost of, at most, a few extra instructions. Such modifications apply knowledge of which run-time actions and checks are performed by a virtual machine as bytecode instructions are executed. For instance, within Sun's Java Virtual Machine (JVM), all object dereferences are preceded with a run-time check for a null value; if null, a NullPointerException is thrown; if not null, the object dereference proceeds. As observed by Orchard [2], this exception can be exploited in loops whose control expressions involve an explicit null check since the expressions may be redundant—the exception thrown as a result of dereferencing a null pointer may be used to transfer control out of the loop (Figures 1(a), (b)).

Array-bounds checks present another opportunity. Before accessing any element of an array, the run-time system first checks whether the array index is within the array's bounds; if not, the JVM throws an ArrayIndexOutOfBounds exception; otherwise the array access proceeds. In a similar manner to the previous example, the exception may be exploited in any loop whose control expression involves

*Correspondence to: Michael Zastre, Department of Computer Science, University of Victoria, P.O. Box 3055 STN CSC, Victoria, B.C., Canada V8W 3P6.
†E-mail: zastre@csr.uvic.ca

Copyright © 2001 John Wiley & Sons, Ltd.

Received 31 August 2000
Revised 30 April 2001
Accepted 21 May 2001

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

(a) original

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

(b) transformed

Figure 1. Eliminating a redundant null check.

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

(a) original

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

(b) transformed

Figure 2. Eliminating redundant array-bounds check.

a comparison between a loop variable and an array's length. The check may be redundant where the same action is performed for every array access, and we may use the exception to transfer control out of the loop. This eliminates one check on every loop iteration (Figures 2(a), (b)); using Sun's HotSpot JVM on a 700 MHz Pentium 3, the transformed code is faster than the original code (13 ms vs. 20 ms) when the array is large (A.length $> 500\,000$). The speedup on any JVM clearly depends on the cost of throwing an exception and on the number of loop iterations.

The transformations exploit the termination semantics of Java exceptions, transferring flow of control through the use of try-catch statements. Exceptions are subclasses of the Java Exception class, and instances of exceptions are either thrown explicitly via the throw keyword, or implicitly through the failure of some run-time check. The programmer may use a try-catch statement to specify the exception handler (catch clause) for a specific block of code (try clause). When an exception is thrown within a try block, the JVM checks if a handler for this exception class exists within a catch block.

Copyright © 2001 John Wiley & Sons, Ltd.

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

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

Figure 3. Bytecode for Figure 1(b).

If so, control is transferred to the first instruction in the catch block; if not, the exception is propagated to the dynamically enclosing scope where the search is repeated.

In Figure 1(a), the loop terminates when p is null, after which the linked list size is returned. The transformed code in Figure 1(b) also terminates when p is null:

- the expression p.next dereferences p, so the JVM checks if p is null;
- when p is null, the dereference causes a NullPointerException to be thrown;
- control is transferred to the catch block for a NullPointerException—in this case, the block is empty;
- control continues to the next statement after the end of the catch block, in this case the return statement;

At the level of bytecode, an exception table associated with each method contains the information represented by source-level try-catch statements. The bytecode of Figure 3 corresponds to that generated for the example in Figure 1(b) (i.e. as would be output from a classfile disassembler such as Sun's javap -c). At the end of the code listing is a table having a single row, with numbers referring to statements within the method and a string referring to an Exception class. The first two numbers correspond to the try-block scope, the third number to the first bytecode of the catch block, and the class name is used at run-time to match exceptions with local handlers.

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

Copyright © 2001 John Wiley & Sons, Ltd.

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

(a) original

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

(b) transformed with check

Figure 4. Use of additional checks in transformed code.

Not all loops may be transformed to exploit exceptions quite so simply. For instance, if the order of increment and dereference statements are reversed in Figure 1(a), then the effects of a *spurious update* would be seen at the print statement in the transformed code, giving an off-by-one error for the list size. This occurs because the semantics of the original code would not be preserved in the transformed code. Where the original never increments the variable a when p is null, i.e. the loop-control expression evaluates to false, the transformed code instead increments the variable before the NullPointerException is thrown and control transferred out of the loop. In this latter case, the transformation is not possible without making other changes to the code.

Another instance is where the loop body may also contain a dereference of another variable. Any NullPointerException resulting from this dereference must not be consumed by the catch block, but propagated out of the block with a throw instruction. Figure 4(a) shows a modification of the first example: the dereference of q might cause a NullPointerException to be thrown. In this instance, the transformed code has an additional check within the catch block to re-throw any unexpected NullPointerException. Figure 4(b) contains the transformed code with the check in place.

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

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

Copyright © 2001 John Wiley & Sons, Ltd.

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

CODE ANALYSIS

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

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

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

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

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

For the false branch, we must know

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

We can capture the needed information by computing *must-ref states*, with one state value for each flowgraph edge (i.e. for each program point). Each state is a set of tuples of the form

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

Copyright © 2001 John Wiley & Sons, Ltd.

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

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

An isnull check of an object reference p is considered redundant if p appears as an object reference within a state tuple *at the program point immediately before the true branch of the check*, i.e. an isnull check is indeed implicitly performed by the JVM on all true program paths. Then the set of variable names is compared against the set of live variables at the program point *immediately before the false branch of the check*, i.e. a variable is live at a program point if it is used on some program path starting from that point. If the intersection of the two sets is empty, then the transformed program is guaranteed to have the same meaning as the original program. (Note that we refer to programs rather than just loops.) We use the set of bytecode positions to either construct a new exception table for the method or to modify an existing table—each new row will correspond to a one-bytecode-sized try block. A catch block is also constructed for each new try block, and simply contains a goto instruction to the first instruction of the false branch. A developed example is shown in Figures 6 and 7.

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

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

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

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

$$\sigma(n) = \{(r, l \cup l', v \cup v') \mid \exists(r, l, v) \in \sigma(n_1), \exists(r, l', v') \in \sigma(n_2)\}, \quad n_1, n_2 \text{ successors to } n$$

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

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

- Assignments: there are two groups of cases—one for the left-hand side of the assignment, and another for the right-hand side. Each group has three sub-cases—pointer dereference, scalar variable and object reference. One combination may be ignored as impossible, e.g. *lhs* is a pointer

Copyright © 2001 John Wiley & Sons, Ltd.

reference with *rhs* a scalar variable. All other combinations transform state by first applying the *rhs* rule to the incoming state, then the *lhs* rule to this result. We use $lhs(n)$ to refer to the left-hand side variable in the statement following program point n; $rhs(n)$ is defined similarly; $lnum(n)$ is the bytecode position of that statement.

(1) Any pointer dereference will either generate a new tuple, which must be added to the incoming state, or if such a tuple already exists, will replace the information already gathered for that tuple. The *lhs* and *rhs* cases have the same rule. A pointer dereference is of the form $r.e$, where r is an object reference, and e is a field accessed by dereferencing r.

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

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

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

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

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

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

$$simplify(\sigma(n)) = \Big\{(r, \lambda, \delta) \mid (r, l, v) \in \sigma(n),\ \lambda = \bigcup\{l' \mid (r, l', v'') \in \sigma'(n)\}, \\ \delta = \bigcup\{v' \mid (r, l'', v') \in \sigma'(n)\}\Big\}$$

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

The technique above is a form of abstract interpretation [3]. We start not knowing any of the must-ref states, but if we initialize all states to empty and repeatedly apply the rules, then we converge to the solution.

Before using must-ref states in a transformation algorithm, we must still account for one other complication added by aliasing. Aliasing introduces the possibility that an object dereference, and hence the run-time isnull check, is applied to some object other than the one in the loop-control expression. For instance, in Figure 5 the loop-control expression involves an explicit isnull check on p, but one program path from the check to the dereference of p has an assignment of the form p = q. A null pointer exception thrown at the dereference of p may be caused by a null value originating from either (1) the object pointed to by p at the start of the loop or (2) the object pointed to by q. If the latter, then our catch block *must re-throw the exception*.

Copyright © 2001 John Wiley & Sons, Ltd.

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

Figure 5. Aliasing of object references.

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

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

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

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

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

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

TRANSFORMATION ALGORITHM

Input.

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

Output.

- Modified code

Copyright © 2001 John Wiley & Sons, Ltd.

Algorithm. For each statement s which tests if object reference R is null, where n is the program point corresponding to the *true* edge, and m is the program point corresponding to the *false* edge, do the following:

(1) If there exists a tuple $(r, l, v) \in \sigma_{\mathrm{must}}(n)$ with $r = R$ and if $v \cap livevar(m) = \emptyset$, then:

(a) For every line number in l, create a null-pointer exception *catch* block entry in the exception table such that

(i) the `try`-block `start` and `end` labels are both set to l;

(ii) the destination is a new globally unique label attached to the following new code sequence[‡]

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

(b) For each tuple (r', l', v') in $\sigma_{\mathrm{may}}(n)$ such that $r' \neq r$ and $l' \cap l \neq \emptyset$, do

(i) set $check = l' \cap l$;

(ii) set `R` to the *ObjRef* corresponding to r;

(iii) for each $lnum \in check$, modify the catch block created in the previous step for this line number such that it reads

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

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

(2) Delete statement s.

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

- According to Sun's JVM specification [4], an athrow instruction's effect on the operand stack is to discard all items below the top item, i.e. the exception object becomes the sole stack contents. For the transformation to be 'verifier neutral' (i.e. not a cause of verification failure), no stack item may be live at program point m and at the program point immediately preceding s. Here we consider a stack item to be 'live' if it is ever popped from the stack. The condition is trivially true at any program point where the stack is empty.
- If the transform is applied to bytecode, the semantics of code using a finally clause are preserved without extra work. For example, a line l could be nested within a try block with a finally clause. Dereferencing a null pointer at l in such a case must take us out of the transformed loop *without* executing any of the finally code. The inserted handler is therefore safe as it transfers control

[‡]The Java Virtual Machine specification [4] requires that 'the only entry to an exception handler is through an exception. It is impossible to fall through or 'goto' the exception handler'. We can interpret this to mean either (a) even when within an exception handler, code cannot use 'goto' to jump to the beginning of another handler, nor fall through to an instruction which happens to be the start of another handler, or (b) once within a handler, code may jump into other handlers without restriction. The first interpretation has been used for this algorithm.

Copyright © 2001 John Wiley & Sons, Ltd.

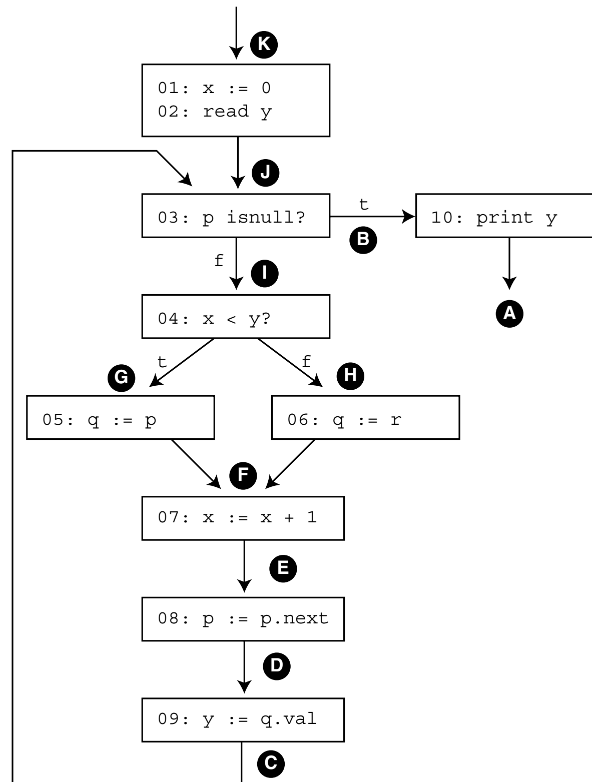

Figure 6. Flowgraph with redundant loop-control expression.

directly to program point m after catching the null-pointer exception. This assumes that handlers appear in proper order in the method's exception table.

EXAMPLE

In our example analysis, we examine a method whose flowgraph corresponds to that in Figure 6. The loop contains a conditional expression whose isnull test may be eliminated if

(a) all paths from program point corresponding to letter `I` (n of the algorithm) will dereference `p`, i.e. a *null pointer exception* thrown by the virtual machine can transfer control out of the loop; and
(b) all variable definitions occurring after the dereference of `p` *do not reach outside the loop*, i.e. those defined variables are *dead* at program point B (m in the algorithm).

Table I. States for Figure 6.

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

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

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

We use *must-deref* information to determine (a) and (b), and *may-deref* information to generate the check code required by (c). The values of *must-ref* and *may-ref* states for each program point are shown in Table I. Since the live-variable set for program point B is equal to $\{y\}$, the isnull check at statement 3 may be eliminated, and the code in Figure 7(b) is the output from the transformation algorithm.

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

COST OF EXCEPTIONS

Our goal is to save more time by eliminating redundant loop-control expressions than is spent in throwing an exception. Much depends upon the relative time cost of isnull instructions compared to the processing of throws; in some implementations of the Java VM, we have found that throwing and catching a NullPointerException is from 500 to 4000 times more expensive than executing a simple isnull check. The high cost can be attributed to several factors.

- Depending on the underlying object model implementation, a null-pointer dereference may cause a host system null-pointer dereference. In this case, the memory fault is caught by the host OS—already an expensive operation—and the OS delivers a signal to the process executing the JVM [5].

Copyright © 2001 John Wiley & Sons, Ltd.

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

(a) original

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

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

(b) transformed

Figure 7. Original and transformed code for working example.

- The method's exception table must be searched for the proper catch block given the thrown exception's class. Large exception tables are possible, and the search for a handler is not a simple equality test. An instanceof check must be performed since the subclass relationships among Exception classes may be non-trivial, i.e. if exception class B is a subclass of exception class A, then a thrown instance of B could be caught by a handler for A [6].
- The Exception class in Java provides a method for printing a stack trace. Therefore, when an exception is thrown, a stack trace must be built at the throw site and stored in the exception object pushed onto the stack. Constructing this trace requires a traversal through the chain of stack frames for all methods which have not yet exited, and includes significant other work including many lookups into the constant pool. If the exception object is not used, then the effort expended is lost [7].

The systems-programming community has traditionally considered exceptions to be rare events, and optimizing their execution in a JVM might be considered as adding needless extra complexity. However, one recent study shows that the exception style of programming is becoming more common [1]. Although this particular study is meant to inform implementors of optimizing compilers, it also indicates the evolving nature of typical Java programs and what can be expected to be executed on JVMs

Copyright © 2001 John Wiley & Sons, Ltd.

in the future. Even though our own transformation depends upon runtime (as opposed to user-defined) exceptions, there are several simple special cases for which optimizations should exist in a JVM.

- Null pointer checks should not depend upon the underlying host system's memory traps for detection.
- An inexpensive test can be constructed for the simple case of an exception thrown within a try block where the handler is declared within an accompanying catch block.
- The construction at exception-throw time of any data structures (e.g. the stack trace) can be done lazily, i.e. as late as possible. For instance, the stack trace could be built incrementally as each scope is exited.
- In many cases, a search of the exception table should not be required at run time. Even when it is required, the search can be implemented by table look-up or by perfect hashing.

Given these optimizations and where the Exception object is subsequently unused, we confidently expect the cost of a NullPointerException to be no more than ten times more expensive than an isnull check. For loops through null-terminated structures, this can result in a speed optimization for loops iterating 100 times or more. We further expect that a faster implementation would help promote exceptions as a standard programming pattern.

CODE ANALYSIS RESULTS

A classfile analyser was written using Purdue's BLOAT framework [8], and we selected a representative variety of Java packages for our tests. The analyser reads the methods of a classfile and then reports the number of ref-chasing and array-indexing loops; the total such number is reported for each package under 'all'. The number which are transformable according to our analysis are reported under 'transform'. Results are shown in Table II.

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

We observe that our simple analysis rejects loops having a method call on a program path from the loop-condition expression to a suitable runtime-checked instruction. This is too conservative: many method calls have no side effects that would invalidate must-ref states (e.g. an append() to a string, clone() on an Object). Beyond modified local variables in a loop, there are other cases involving some global variables (i.e. instance and class), and these cases could be enumerated and integrated into the loop analysis at the cost of a little complexity.

RELATED WORK

We do not know of any comparable transformation to that presented in this paper. There is recent work in analyzing the use of exception handling in Java code ([9], [10]) in the context of software engineering and program analysis (i.e. effect on slicing, building of program dependency graphs).

Copyright © 2001 John Wiley & Sons, Ltd.

Table II. Classfile loop analysis.

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

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

There is also recent work in eliminating the overhead of null-pointer and array-bounds checks for Java multi-dimensional arrays when used in computationally intensive tasks [11], but this focuses on the implementation of a specially-tuned Array class. The growing body of work on Java Just-In-Time (JIT) compilers [12] includes some remedies for the constraints placed upon code-motion optimizations by the precise-exception semantics of Java [13], but these do not address the cost of throwing and handling exceptions. Advice for Java programmers concerned about program performance in the presence of exception handling is also now making its way into programming practitioners' literature [14].

Several research efforts have specifically addressed exception-handling costs. Krall and Probst modified the exception handling mechanism for their CACAO system, and from this obtained free null-pointer checks [15]. Lee *et al.* have implemented a JIT which takes advantage of local exception handler cases, i.e. handlers for which no stack unwinding is necessary [16]. Both of these solutions depend upon native code characteristics to achieve performance improvements, and it is not yet clear how these approaches would contribute to a generally faster handler mechanism within an interpreter.

CONCLUSIONS

We have presented a novel loop optimization for Java bytecode which results in faster code running on a JVM, and which does not depend on the use of a JIT compiler. It achieves this speedup by eliminating loop-control expressions which are already implicitly performed by the JVM as part of its support for the Java language specification. For example, a loop like that shown in Figure 1 is reduced from 6 bytecode instructions to 5. The transformation is not limited to loops—the analysis may be applied to

Copyright © 2001 John Wiley & Sons, Ltd.

any control-flow graph where a conditional expression contains some check already performed along the execution path followed when the expression evaluates to true.

Possibilities for implementing the optimization include performing it as part of a class loader since only bytecode is modified. Profiling data could also be used to determine when the optimization is profitable. Similarly, two bodies of a loop could be generated (one transformed, the other not) and the appropriate loop version selected at runtime based on some dynamic value.

An important conclusion is that exception handling can and should be fast. The assumption that they are rare events is no longer tenable—the increased use of the exception-handling style of programming will lead to exceptions being considered a normal programming construct. We can expect programs to throw and catch exceptions much more frequently in the future. The analysis and transformation presented in this paper is one way in which the emerging coding idiom may translate into faster code.

ACKNOWLEDGEMENTS

The authors thank Dave Orchard, IBM Pacific Development Centre, for the original observation that loops may run faster if exited by catching an exception; John Aycock and Piotr Kaminski at the University of Victoria, and Jan Vitek at Purdue University, for helpful discussions; and the anonymous referees for their suggestions.

REFERENCES

1. Ryder BG, Smith D, Kreme U, Gordon M, Shah N. A static study of Java exceptions. *Proceedings of the 9th International Conference on Compiler Construction* (*Lecture Notes in Computer Science*, vol. 1781). Springer Verlag: Berlin, 2000; 67–81.
2. Orchard D. Better performance with exceptions in Java. *BYTE Magazine*; 1998; 53–54.
3. Abramsky S, Hankin C. An introduction to abstract interpretation. *Abstract Interpretation of Declarative Languages*, Abramsky S, Hankin C (ed.). Ellis Horwood, 1987; 9–31.
4. Lindholm T, Yellin F. *The Java (tm) Virtual Machine* (2nd edn). Addison-Wesley: Reading, MA, 1999.
5. Alpern B *et al*. The Jalapeño virtual machine. *IBM Systems Journal* 2000; **39**(1):211–238.
6. Vitek J, Horspool N, Krall A. Efficient type inclusion tests. *Proceedings of OOPSLA '97*, Atlanta, GA. ACM Press, 1997; 142–157.
7. Cierniak M, Lueh G-Y, Stichnoth J. Practicing JUDO: Java under dynamic optimization. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada. ACM Press, 2000.
8. Whitlock DM. *The BLOAT book*. Programming for Persistent Systems Research Group, Purdue University. http://www.cs.purdue.edu/homes/hosking/bloat/ [October 1999].
9. Sinha S, Harrold MJ. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering* 2000; **26**(9):849–871. ACM Press.
10. Robillard MP, Murphy GC. Analyzing exception flow in Java programs. *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering* 1999; 322–337.
11. Moreira JE, Midkiff SP, Gupta M, Artigas PV, Snir M, Lawrence RD. Java programming for high-performance numerical computing. *IBM Systems Journal* 2000; **39**(1):21–56.
12. Sarkar V. Optimizing compilation of Java programs. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vancouver, Canada, 2000. Tutorial notes.
13. Gupta M, Choi J-D, Hind M. Optimizing Java programs in the presence of exceptions. *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, Cannes, France (*Lecture Notes in Computer Science*, vol. 1850). Springer Verlag: Berlin, 2000.
14. Shirazi J. *Java Performance Tuning*. O'Reilly, 2000.
15. Krall A, Probst M. Monitors and exceptions: How to implement Java efficiently. *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*. ACM Press, 1998; 15–24.
16. Lee S, Yang B-S, Kim S, Park S, Moon S-M, Ebcioglu K, Altman E. On-demand translation of Java exception handlers in the LaTTe JVM just-in-time compiler. *Proceedings of the Workshop on Binary Translation*, October 1999. Technical Committee on Computer Architecture (TCCA) Newletter, IEE.

Copyright © 2001 John Wiley & Sons, Ltd.