Two Techniques for Improving the Performance of Exception Handling

Michael J. Zastre and R. Nigel Horspool
{zastre,nigelh}@cs.uvic.ca

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

Abstract. Two new optimizations using whole-program analysis are presented for languages using `try-catch` exceptions clauses (such as Java and C#) – *farhandlers* reduce the time needed to locate the handler for a thrown exception, and *throwsite reduction* eliminates unnecessary run-time overhead at throw instructions. Also presented are experimental results from a Java VM modified to implement these optimizations.

1 Introduction

Exceptions and their handling occupy an odd place amongst the set of programming-language constructs in languages such as Java. Exceptions are absolutely necessary in that Java's API simply requires their use in many cases, the most notable being for I/O. At the same time, however, exceptions are deemed to be rather expensive, and so in dispensing practical advice regarding performance Java programmers are encouraged to avoid them if at all possible in their own code [Shir00]. The advice seems reasonable especially given the run-time expense of exceptions – for instance, with many implementations of the Java VM the time taken to throw and catch a `NullPointerException` when dereferencing a null pointer is thousands of times more expensive than explicitly comparing an object reference with the value `null` [Zast05]. Nor can programming-language implementors be blamed for this run-time expense as they have been led to believe that exceptions are rare, and given the complexity of compilers, interpreters and VMs, they (rightly) choose to concentrate on what they consider to be more profitable optimizations. Indeed they often follow the advice given by designers of the programming language, a rare exemplar of which is shown here in text taken from the Modula-3 report [Card89, p. 17]:

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

The result for most languages appears to be something of a classic "catch-22": Before exception implementations become faster there needs to be more programmers using exceptions, yet programmers tend to avoid exceptions because

of their negative impact on program performance. Therefore there exists a danger that the innovative uses of exceptions in the structuring of programs may not get much purchase amongst practitioners if the run-time costs are deemed to negate the gains to program clarity and expressibility.

Our contribution to breaking out of the "catch-22" is to propose two optimizations that reduce the run-time cost of exceptions. These optimizations do not require programmers to write their code any differently than they would when using exceptions, nor are programmers expected to supply special annotations or pragmas. The optimizations are also applicable when throw and handler sites are located within different methods. We introduce the first of these, called *farhandlers* in the next section. After that we introduce the second technique, called *throwsite reduction*. Next appears some experimental results obtained from a modified Java VM, and this is followed by a brief description of related work.

2 Farhandlers

Consider the callgraph in Figure 1. If the site calling a method is enclosed within an exception handler, then the edge corresponding to that method call is labelled. For example, there is a call to method `b` within method `a`, and this call occurs within a handler for exceptions of class `E`. In this particular callgraph there appear four different handlers for exceptions `E`; handlers are numbered within parentheses. (Note: This numbering is not programmatic – that is, a programmer does not provide this numbering.) The question posed by the diagram is: If an exception of class `E` is thrown in method `m`, which of the four handlers will catch it?

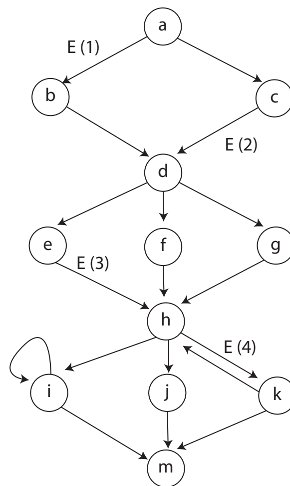


Fig. 1. Callgraph example

The answer depends, of course, on the path through the graph from node `a` to node `m`, and here we assume that there is no handler for `E` local to `m`. In practice most exception-handling mechanisms *unwind the stack*: a local handler for `E` in `m` is sought at run-time, and if one is not found, the search continues within the method that called `m`, with the search proceeding through the callstack until a local handler is found.

The goal of a *farhandler table* is to reduce both the number of handler-table lookups and the number of times stack-unwinding occurs by taking into account information about the callpath available on the runtime stack. For example, if an exception of class `E` is thrown on an invocation of `m`, then we can examine the return address stored for that invocation. If this return address indicates a call from either `i` or `j`, then unwinding to the first invocation of `h` is needed to help determine the location of a handler. However, if the return address indicates a call from `k`, then the handler location is known (that is, we unwind to the most recent invocation of `h` and transfer control to the first instruction in the handler `E(4)`). Therefore even if an exception-handler table exists in nodes `i` or `j` (that is, for other exception classes), we do not examine the tables; as we need not check for such tables, we can unwind the stack faster than if we had to examine the tables.

Extending the example somewhat, consider the callpath `a, b, d, f, h, j, m`. The callgraph indicates that if such a path has been followed at runtime, handler `E(1)` would be the appropriate handler when an instance of `E` is thrown in `m`. At runtime, however, we would normally be able to piece together the callpath only by examining the call stack one frame at a time. (The implication here is that we choose not to keep a copy of the callpath separate from the call stack itself as that would introduce possibly unnecessary work at each method invocation.) That being the case, what is the smallest number of distinct unwind/lookup steps needed to locate the handler? Here the answer is “3”:

1. At the invocation of `m`, the return address stored indicates that `m` was called from `j`; therefore we unwind to `h`.
2. Now that we are in `h`’s context, the return address stored here indicates that this invocation of `h` is the result of a call from `f`; therefore we can unwind to `d`.
3. In `d`’s context, we discover from the return address stored here that `d` was called from `b`, and that `b` itself was called from `a` and this from within a handler to `E` (handler 1 from the callgraph). Therefore we can unwind to `a` and then transfer control to the first instruction in `E(1)`.

To implement this new form of handler lookup we introduce a *farhandler table*; the name is meant to suggest an aid for finding non-local handlers. Such a table for the example callgraph appears in Table 1. Here are a few items to observe about this table:

– Each row in this table has an edge in the callgraph (*i.e.,* one-to-one mapping from edges to row).

– At exception-throw time, the current stackframe context (*i.e.,* the node) and the return address stored in the context are used to find a single row in the table.
– Each row in the table has either an entry in the *Dispatch Address* column or in the *Handler Info* column, but not both.

Entries in *Dispatch Address* indicate both an unwind point *and* an address to which control flow is transferred. For example, `PC(E1)` denotes the first location in handler `E(1)`. No further lookups or unwinding are necessary after using information in *Dispatch Address*. If the location of a handler is not yet known, then *Handler Info* is used to indicate the point to which the stack must be unwound. For example, the first step given the program path above would result in a lookup of the second-to-last row, and this contains $\mathcal{H}(\texttt{h})$ – therefore the runtime stack is unwound to the earliest instance of `h` and handler lookup continues in `h`'s context.

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

Table 1. Program-wide handler table for callgraph example

Computing these tables is relatively straightforward and consists of three separate steps:

1. For each node, a set of reachable handlers for a given exception class is computed; set items are denoted by a triple of the form $\langle n_s, \Sigma, n_d\rangle$, where n_s is a calling (or *source*) node, n_d the called (or *destination*) node, and Σ the exception handler enclosing the call of m to n. (*i.e.,* these are representations of labelled edges in our callgraph). The sets of reachable handlers from a node such as `m`, *i.e.,* those later in the call sequence, are larger than a node such as `d`, *i.e.,* those earlier in the call sequence.

2. We then identify those nodes where control-flow paths merge together. These nodes are called *mergehandlers*. In our callgraph example, such nodes are `d`, `h` and `m`; the root node of a callgraph is considered to be a trivial merge node (*i.e.,* `a` in our example). All other nodes can be reached from only one other node. Each node in the callgraph will have associated with it a mergehandler, which is either the node itself (if the node is a mergehandler) or the first mergehandler encountered when traversing up the callgraph (*i.e.,* towards the root node).
3. With the previous two items we can now construct the farhandler table row by row. Each callgraph edge corresponds to a table row. The first field in the row is the edge's destination node. The second field is the return address corresponding to the callsite (*i.e.,* next instruction following the callsite in the edge's source node). If there is exactly one item in set computed for the destination node in (1) above, then the corresponding handler is used to fill in the third field. Otherwise the third field remains blank and the fourth field is filled with the mergehandler for that node.

Farhandler tables can be extended to deal with more than one exception type by adding extra columns to the table and repeating the analysis for each additional exception type.

There has been no mention of `finally` clauses so far. Previous work has shown that these clauses are rare in practice [Ryde00]. We nevertheless have described one way of dealing with such clauses in [Zast05].

3 Throwsite Reduction

The actual run-time cost of throwing and catching an exception can be broken down into approximately four separate categories:

- the effort required to allocate an exception object on the heap;
- the construction of a stack trace;
- the cost of unwinding the stack; and
- other activities such as handler table lookup, actions of the VM, garbage collection during unwinding, etc.

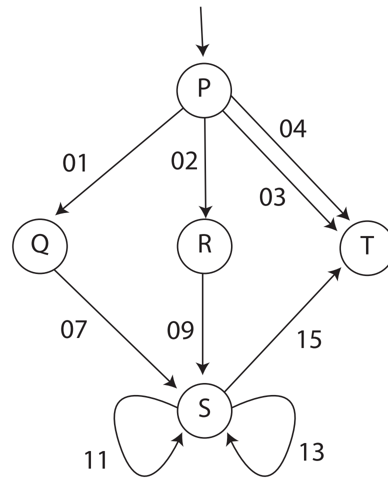
These activities can often take surprisingly long periods of time, and one or two of these consume large proportions of the overall effort needed for throwing and catching [Zast05]. For example, stack-trace construction on the Java Classic VM (version 1.4.2) makes up at least 50% of the effort, and the deeper the callstack the longer the time required (*i.e.,* the complete stacktrace comprises information starting at program's `main` entry point and leading down to the throwing method itself).

What is somewhat more shocking is that there is no requirement stating a handler must use the exception object, nor need the handler even reference the stacktrace itself. Many handlers in fact do not use the exception object to transfer information from the throwsite to the handler, and instead depend

upon exceptions as a form of control-flow transfer. Therefore we propose an optimization called *throwsite reduction*, implying that the work performed at the throwsite (such as exception-object creation and stacktrace construction) is eliminated if certain facts are true about handlers reachable from the throwsite.

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

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

   R() {
09   S();
   }

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

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

(a) code



(b) call graph

Fig. 2. Code and callgraph

Consider the code appearing in Figure 2 and the corresponding callgraph appearing to the right of the code. The labels on callgraph edges now correspond to line numbers in the code example (*i.e.,* the edge from `R` to `S` corresponds to line `09` where `R()` makes a call to `S()`). Nearly every call is enclosed by some `try` block. The start of handler blocks are notated with either the word `use` or `discard`; this indicates whether or not the exception object is used by the handler. The questions raised by this code are: Does an exception object need to be created at the throwsite on line 16? or on line 17?

If all handlers that can catch an exception thrown at line 16 do not use the object, then we can annotate the throwsite at line 16 as being eligible for *tsr* (throwsite reduction). When the bytecode corresponding to line 16 is evaluated at runtime, a virtual machine could check if a *tsr* annotation exists for this `athrow` bytecode, and if this does exist then a dummy placeholder object is pushed onto the virtual machine's operand stack.

Code inspection in this case reveals there exists *no* handler in the example which would use an exception objected created at line 16. The same cannot be said, however, for the instance of `F` thrown at line 17: As `T` is reachable from `S` – via a call to `S` in line 11 followed by a call to `T` in line 15 – and as the handler at line 12 uses the exception object, we cannot mark line 17 as *tsr*.

The computations for determining *tsr* annotations uses some of the intermediate values prepared for farhandler tables (*i.e.,* the set of reachable handlers for each node in step 1). We need add only one extra bit of information about each handler – that is, whether or not the exception object is used by the handler – and use the following for each throwsite:

1. If a local handler around the throwsite for the exception class exists, then check if the handler uses the exception object. If so, then the throwsite cannot be annotated *tsr* and we proceed to examine the next throwsite.
2. Otherwise we examine all of the handlers for the exception class reachable from the throwsite. If no handler uses the exception object, then the throwsite is annotated as *tsr* and we proceed to the next throwsite.
3. Otherwise the throwsite is left unannotated.

The usefulness of the *tsr* annotations, not to mention that of handler information in farhandlers, is directly affected by the analysis used to build the callgraph. In the following section we present experimental results where callgraphs were built using a *flow-insensitive* analysis, and we can do better than this but only if we are willing to pay the extra cost at compile time to perform a flow-sensitive analysis. The analyses now available for object-oriented programming languages (such as Class Hierarachy Analysis [Dean95]) can provide an even greater levels of precision (*i.e.,* fewer callgraph edges).

4 Experimental Results

We implemented both farhandlers and throwsite reduction using an analyzer based on the `Soot` bytecode-manipulation framework [Vale99] and modified an existing Java virtual machine called `SableVM` [Gagn01]. Timings provided in the next two subsections were produced on a Pentium 3 running at 750 MHz and 128 MB RAM running RedHat Linux 7.2. Our experiments were in two groups: the first was our *validation* group in which we used the SPECjvm98 benchmarks, and the other *exception-idiom usage* group in which a standard algorithm was converted into an exception-handling style of code.

4.1 Validation group

As with medical doctors who must administer treatment to patients, our modification of the VM should follow the rule from medicine of *primum non nocere*—"first of all, do no harm." Therefore our modified VM should not produce poorer performance for programs, regardless of whether or not they use exceptions. The SPECjvm8 benchmark suite allows us to check for this; the version of we used is maintenance release 1.04, but with two omissions:

– `_227_mtrt` (a ray-tracing program) raises a `ClassCastException` that causes program failure when run with either the unmodified VM or the modified VM. (The same error occurs when using the HotSpot VM from Sun.)
– `_213_javac` (Sun's Java compiler from JDK 1.0.2) causes our `Soot`-based analyzer to fail from an `OutOfMemoryException` (one of life's little ironies!) and therefore no farhandler tables or *tsr* annotations can be generated.

Of the SPECjvm98 benchmark programs tested here, only `_228_jack` makes significant use of exceptions, and even then its programmers appear to have taken special care to eliminate a lot of exception-handling overhead (*i.e.,* the thrown exceptions are previously created objects, with the object creation cost amortized over the many throws which use it).

Each benchmark was run on four different VM configurations:

– original: This is the unmodified SableVM;
– fh: modified VM using only the farhandler table;
– tsr: modified VM using only the *tsr* annotations;
– fh+tsr: modified VM using both the farhandler table and the *tsr* annotations.

The timings are shown in Table 2. Only `_200_check` and `_228_jack` throw any exceptions at all; the former throws 104 exceptions caught by local handlers, while the latter throws 241,876 exceptions caught by non-local handlers. What the results show is that the benchmarks run as fast—if not faster—under the modified VM as they do under the unmodified VM.

We make two general observations about this data:

1. Only `_228_jack` throws a significant number of exceptions—all of them to handlers outside of the throwing method—and the benchmark's speed is improved (about 1% on average, with 0.7% in the worst case and 1.2% in the best case). This gain is significant considering that much other computation is being performed by the benchmark program.
2. For all of the other benchmark programs, there is no observable difference (*i.e.,* "no harm").

4.2 Exception-Idiom Usage group

As a test of the effect of our two optimizations on a practical problem, we have chosen one for which a file of words must be examined, and a histogram

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

Table 2. SPECjvm98 benchmark timings (milliseconds)

of those words produced (as might be needed by a compression algorithm, for instance). As each word is input, a binary tree is searched. If the word is found, the corresponding tree node's `frequency` field is incremented. If the word is not found, then a new node must be created and linked into the existing tree.

Several versions of the program were written:

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

Text files from the *Calgary Compression Corpus* provided the workload for various programs [Bell90]. Timing results for for `SearchNonLocal` are in Table 4. Each of the individual tests in the corpus corresponds to a table row. The column labelled "w/o exceptions" is the time (in milliseconds) taken by the unmodified VM to process the test file with an algorithm that *does not* use exceptions. There follow two pairs of columns: the first pair is for a version of the VM not using the optimizations described here, while the VM of the second pair *does* support both farhandlers and throwsite reduction. Each of the "exception cost" columns represents the contribution made by exceptions to processing a file (*e.g.,* when computing the histogram for words in `bib`, exception-handling in the original VM results in a program running 353% longer than the version of the program without exceptions, while in the modified VM exception-handling the program runs only 0.3% longer).

The modified VM is clearly a win. The overhead of using exceptions (*i.e.,* the time difference between an exception-free program and exception-rich one) is low, ranging from .3% to 1.1% for `SearchNonLocal`. A pleasant surprise from `SearchLocal` (results not shown here) is that in some cases there is a *speedup* as in that for `book1` of about 0.6% [Zast05].

file	w/o exceptions	*Using original VM* w/ exceptions	exception cost	*Using modified VM, fh + tsr* w/ exceptions	exception cost
`bib`	2,445	11,081	353%	2,453	0.3%
`book1`	20,661	73,255	254%	20,731	0.3%
`book2`	15,240	48,962	218%	15,338	0.6%
`geo`	819	2,279	178%	821	0.2%
`news`	8,645	42,262	388%	8,725	0.9%
`obj1`	223	1,095	391%	226	1.3%
`obj2`	2,737	14,949	447%	2,762	0.9%
`paper1`	1,198	5,895	392%	1,210	1.0%
`paper2`	1,941	8,269	326%	1,958	0.9%
`paper3`	1,028	5,922	476%	1,049	2.0%
`paper4`	276	1,596	478%	279	1.1%
`paper5`	266	1,573	491%	269	1.1%
`paper6`	916	4,167	354%	921	0.5%
`progc`	823	4,160	405%	828	0.6%
`progl`	1,426	5,281	270%	1,434	0.6%
`progp`	805	3,644	353%	809	0.5%
`trans`	1,647	6,713	307%	1,655	0.5%

Table 3. SearchNonLocal timings (milliseconds, 10,000 iterations)

5 Related Work

Some optimizations for improving exception-handling performance are applied to cases where the throwsite and its handler are within the same method, as in the `LaTTe` system [Lee99]. If some method inlining is acceptable, then *Exception-Directed Optimization* may be suitable; paths through a callgraph are profiled, and those paths with a high execution frequency are inlined into one large method and local optimizations then applied [Ogas01].

A stack-unwinding optimization was proposed by Drew *et al.* [Drew95] in which as little state as possible is restored when moving from a frame to its calling procedure's frame. State is instead restored incrementally, *i.e.,* only when a handler is found is the complete state of a procedure's context restored.

6 A word about Just-In-Time compilation

The results described in the previous section were obtained using a VM supporting only interpreted bytecode. Much recent work on improving the run-time performance of languages such as Java and C# have focused on *Just-In-Time* compilers, *i.e.,* where individual methods are compiled such that they run at the speed of the underlying machine's native code (or more precisely, the interpreted VM may invoke both bytecode and native-code version of methods). One assumption therefore may be that the cost of exceptions can be eliminated by a JITter without any extra analysis or algorithms.

Unfortunately this assumption is woefully inaccurate. Without extra analysis, a JIT may lead a programmer to believe that exception-handling has an even *higher* cost relative to code written without exceptions. This is due to the way in which a language's runtime deals with exceptions, *i.e.,* non-local exceptions

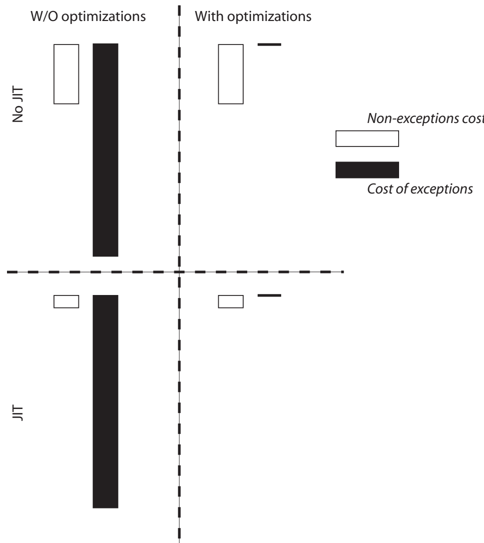


Fig. 3. Interaction of JIT-compilation with optimizations

invoke code within the interpreted VM for transferring control from the throwsite to the handler. One way of visualizing this is shown in Figure 3; each of the quadrants represents the contribution to the overall cost of the execution (in this case the processing of `bib` from the Calgary Compression Corpus in the experiments just described). If a JIT produces a modest five-fold increase in execution speed for JIT-compiled methods and exceptions are still dealt with as previously, then what the programmer experiences appears in the left side of the diagram – the relative contribution of exceptions relative to the "unexceptional" code – appears much larger even though the absolute contribution is unchanged.

We argue that our optimizations are even *more* important for a JIT compiler than for a purely interpreted VM. This can be seen in the right-hand side of Figure 3 where the exception cost is now very small, such that the five-fold increase in performance offered by the JIT is – for all intents and purposes – achieved in the presence of exception handling.

7 Conclusion

We have presented two new optimizations which are designed to improve the performance of exception handling. Our main goal, however, is to make the use exceptions more attractive to practioners such that they need not be concerned about the runtime cost (or at least not unduly concerned). We have shown that in cases where exceptions are used to express control flow, nearly all the overhead of exceptions can be eliminated. Extensions to these techniques to cover a larger set of cases, such as those where some handlers use the stack trace but not all do (*lazy*

stacktrace construction), or where some handlers use the exception object but not all do (*lazy exception-object creation*). Our current analyses require access to the whole program, but a more incremental approach towards farhandler-table construction (and *tsr* annotations) would combine well with JIT technology (*i.e.,* perform analysis as classes are loaded into the VM). However, much work is still needed to improve the performance of code in the presence of exceptions, specifically analyses to mitigate the negative impact of exceptions on traditional compiler optimizations (for example, Factored Control-Flow Graphs [Choi00]. There also remains the hard work of convincing programmers that exceptions can improve the readability and maintainability of programs, and perhaps this can be achieved via the identification of useful *exception idioms* or *exception patterns*.

References

[Card89] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kaslow, B. and Nelson, G.: Modula-3 Report (revised), Digital Systems Research Centre. 1989.

[Choi00] Choi, J., Grove, D., Hind, M., and Sarkar, V.: Efficient and Precise Modelling of Exceptions for the Analysis of Java Programs, in *Proceedings of the SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99)*, pp. 21–31.

[Dean95] Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis, in *Proceedings of the European Conference on Object-Oriented Programming '95*, pp. 77–101, Springer-Verlag. 1995.

[Drew95] Drew, S., Gough, K. and Ledermann, J.: Implementing Zero Overhead Exception Handling. Technical Report FIT 95-12, Queensland University of Technology. 1995.

[Gagn01] Gagnon, E. and Hendren, L.: SableVM: A Research Framework for the Efficient Execution of Java Bytecode, in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01')*, pp. 27–40, ACM Press. April 2001.

[Lee99] Lee, S., Yang, B., Kim, S., Park, S., Moon, S., Ebcioğlu, K., and Altman, E.: On-Demand Translation of Java Exception Handlers in the LaTTe JVM Just-In-Time Compiler. In *Proceedings of the Workshop on Binary Translation*. October 1999.

[Ogas01] Ogasawara, T., Komatsu, H., and Nakatani, T.: A study of exception handling and its dynamic optimzation in Java. In *Proceedings of the OOPSLA '01 Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 83–95, ACM Press. October 2001.

[Ryde00] Ryder, B., Smith, D., Kremer, U., Gordon, M., and Shah., N.: A Static Study of Exceptions Using JESP. In *Proceedings of Compiler Construction 2000*, pp. 67–81. April 2000.

[Shir00] Shirazi, J.: Java Performance Tuning, O'Reilly and Associates, Inc. 2000.

[Vale99] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagon, E., and Co, P.: Soot – A Java Optimization Framework, in *CASCON 1999*, pp. 125–135. September 1999.

[Bell90] Bell, T., Cleary, J., and Witten, I.: Text Compression, Prentice-Hall, Inc. 1990.

[Zast05] Zastre, M.: The Case for Exception Handling. PhD Thesis, University of Victoria (2004).