Efficient Exception Handling in Java Bytecode-to-C Ahead-of-Time Compiler for Embedded Systems

Dong-Heon Jung JongKuk Park Sung-Hwan Bae Jaemok Lee Soo-Mook Moon

School of Electrical Engineering and Computer Science
Seoul National University, Seoul 151-742, Korea
{clamp, uhehe99, seasoul, jaemok, smoon}@altair.snu.ac.kr

Abstract
One of the most promising approaches to Java acceleration in embedded systems is a bytecode-to-C ahead-of-time compiler (AOTC). It improves the performance of a Java virtual machine (JVM) by translating bytecode into C code, which is then compiled into machine code via an existing C compiler. One important design issue in AOTC is efficient exception handling. Since the excepting point and the exception handler may locate in different methods on a call stack, control transfer between them should be streamlined, while an exception would be an "exceptional" event, so it should not slow down normal execution paths. Previous AOTCs often employed *stack cutting* based on a setjmp()/longjmp(), which we found is involved with too much overheads. This paper proposes a simpler solution based on an exception check after each method call, merged with garbage collection check for reducing its overhead. Our evaluation results on SPECjvm98 on Sun's CVM indicate that our technique can improve the performance of stack cutting by more than 25 %.

Categories and Subject Descriptors
D3.3 [Programming Languages]: Language Constructs and Features; D.3.4 [Processors]: Compilers, Run-time environments; D.4.7 [Operating Systems]: Organization and Design-real-time and embedded systems;

General Terms
Design, Experimentation, Performance, Languages

Keywords
Bytecode-to-C, Java ahead-of-time compiler, exception handling, stack cutting, Java virtual machine, J2ME CDC

1. Introduction
Many embedded systems including mobile phones, digital TVs, and telematics have employed Java as a standard software platform, due to its support for platform independence, security, and faster development of reliable software contents [1]. Platform independence is achieved by installing the Java virtual machine (JVM) on each platform, which executes Java's compiled executable called *bytecode* via *interpretation* [2]. Since this software execution is much slower than hardware execution, compilation techniques that translates the bytecode into machine code has been employed, such as *just-in-time compiler* (JITC) [3] and *ahead-of-time compiler* (AOTC) [4-12]. JITC and AOTC perform the translation during runtime and before runtime, respectively. For embedded systems, AOTC is more useful since it does not need runtime translation and memory overheads of JITC, which is likely to waste the limited computing power and memory space of embedded systems.

There are two approaches to AOTC. One is translating bytecode directly into machine code [7-11], and the other is translating bytecode into C code first, which is then compiled into machine code by an existing compiler [4-6]. The *bytecode-to-C* (b-to-C) approach is more practical since it allows faster time-to-market by resorting to full optimizations of an existing compiler and by using its debugging and profiling tools. It also allows a portable AOTC.

One of the important issues in designing a b-to-C AOTC is exception handling.[20] The Java programming language provides try blocks and catch blocks such that if an exception occurs in a try block, it is supposed to be caught by an appropriate catch block depending on the type of the exception. The problem is that the excepting try block and the exception-handling catch block might be located in different methods on the call stack, so if there is no catch block that can handle the exception in the excepting method, the methods in the call stack should be searched backward to find one that has an exception-handling catch block [2].

Existing b-to-C AOTCs [4-5] employ a technique called *stack cutting* [14-15] based on C's library setjmp()/longjmp() pairs such that a setjmp() is executed in the method that has a catch block while a longjmp() is executed at the excepting method, which allows direct control transfer from an excepting try block to an exception-handling catch block. Unfortunately, we found from experiments that stack cutting is inefficient since it affects the performance for normal execution.

This paper proposes a different approach to exception handling where we check if exception occurred whenever a method returns in order to transfer control to an appropriate catch block. This is inspired by JNI's exception handling mechanism but with additional improvements. We evaluated the proposed technique on

This research was supported in part by Samsung Electronics.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EMSOFT'06, October 22–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-542-8/06/0010...$5.00.

a CVM in Sun's J2ME CDC [12] environment where we developed a b-to-C AOTC. We implemented both approaches and performed experiments.

The rest of this paper is organized as follows. Section 2 briefly overviews the JVM machine model and our b-to-C AOTC. Section 3 describes the stack cutting technique of previous b-to-C AOTCs. Section 4 introduces our proposed technique based on exception checks. Section 5 describes our experimental results. A summary follows in Section 6.

2. Overview of Our Bytecode-to-C AOTC and JVM

The structure of b-to-C AOTC which works with Sun's CVM is depicted in Figure 1. The AOTC translates bytecode into C code, which is then compiled with the CVM source code using a GNU C compiler [13]. Our AOTC selectively translates methods in class files using a profile feedback in order to reduce the code size. This is fine since our AOTC can work with the interpreter, so AOTC methods and interpreter methods can run concurrently. This is useful for an environment where we also need to download class files dynamically (e.g., in digital TVs the Java middleware is AOTCed while the *xlets* downloaded thru the cable line is executed by the interpreter).

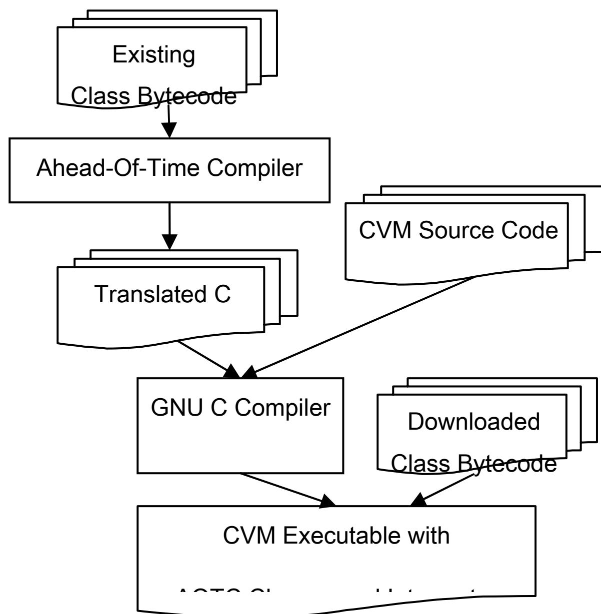

Figure 1. The Structure of our AOTC process

The Java VM is a typed stack machine [2]. All computations are performed on the operand stack and temporary results are saved in local variables, so there are many pushes and pops between the local variables and the operand stack.

Each thread of execution has its own Java stack where a new activation record is pushed when a method is invoked and is popped when it returns. An activation record includes state information, *local variables* and the *operand stack*. Method parameters are also local variables which are initialized to the actual parameters by the JVM.

Our b-to-C AOTC first analyze the bytecode and decides the C variables that need to be declared. Each local variable and each stack slot is translated into a C variable (which is called a local C variable and a stack C variable, respectively). Since the same stack slot can be pushed with differently-typed value during execution, a type name is attached into a stack C variable name such that a stack slot can be translated into multiple C variables. For example, s0_ref is a C variable corresponding to a reference-type stack slot 0, while s0_int is a C variable corresponding to an integer-type stack slot 0.

It then translates each bytecode one-by-one into corresponding C statements, with the status of the operand stack being kept track of. For example, iload_1 which pushes an integer-type local variable 1 onto the stack is translated into a C statement s0_int = l1_int if the current stack pointer points to the zero-th slot when this bytecode is translated. Figure 2 shows an example.

(a) **Java Method**

```
public int max(int a, int b) {
  return (a>=b)? a:b;
}
```

(b) **Bytecode**

```
0: iload_0
1: iload_1
2: if_icmplt 9
5: iload_0
6: goto 10
9: iload_1
10:ireturn
```

(c) **Translated C Code**

```
int Java_java_lang_Math_max_II(JNIEnv *env,
int l0_int, int l1_int)
{
  int s0_int;
  int s1_int;

  s0_int = l0_int;              // 0:
  s1_int = l1_int;              // 1:
  if(s0_int < s1_int) goto L9   // 2:
  s0_int = l0_int;              // 5:
  goto L10;                     // 6:
L9:   s0_int = l1_int;          // 9:
L10: return s0_int;             // 10:
}
```

Figure 2. An example of Java code and translated C code

Our AOTC method follows the JNI standard including its name. The first argument of every function is `CVMExecEnv *ee` which contains the interpreter stack, data structures for exception handling and garbage collection, and so on. Other Java arguments then follow. For synchronized method, additional C code for locking the object is included right after variable declaration.

Although we resort to C compiler's optimization flags for traditional optimizations, we also perform some Java-specific optimizations such as method inlining, null check elimination, array bound check elimination, class initialization check

elimination, etc. since the C compiler is not aware that the translated C code is from Java bytecode.

3. Stack Cutting based on setjmp()/longjmp()

The Java programming language provides an exception handling mechanism for elegant error handling [2]. When an error occurs during execution of code in a try block, the error is caught and handled by an exception handler located in one of subsequent catch blocks associated with it. If no catch block in the method can handle the exception, the method is terminated abnormally and the JVM searches backward through the call stack to find a catch block which can handle the error. This mechanism of searching the call stack is called *stack unwinding*[14] in CVM.

Since an exception would be an "exceptional" event, exception handling should be implemented in a way that normal execution is affected as little as possible. Stack unwinding in the interpreter mode of the CVM is certainly one such an implementation. In a b-to-c AOTC, however, stack unwinding is hard to implement since it is difficult in C to make a direct control transfer from the excepting point to the exception handler, if they are located in different methods on the call stack. One possible solution is using *setjmp()/longjmp()* C libraries [16,17].

The setjmp() function saves the current environment at a `jmpbuf`-type object for later use by the longjmp(). The longjmp() function with a `jmpbuf`-type object argument restores the environment saved by the last call of setjmp() which created the object. After the longjmp() completes, program execution continues as if the corresponding call to setjmp() had just returned [17]. This is perfect for implementing cross-method jumps in C.

In fact, all previous b-to-C AOTCs employed a technique called *stack cutting* using setjmp()/longjmp() functions. The idea is that a setjmp() is executed at every method that has a catch block, and the `jmpbuf`-type object created by the setjmp() is pushed on a global stack of `jmpbuf`-type objects. Later, if a method has an uncaught exception, a longjmp() is executed with the top object of the global stack, which will transfer the control to the method that created the top object. If the exception cannot be handled in that method, the global stack is popped and a longjmp() is executed again with a new top object. This process repeats until the exception handler is found. In normal execution when no exception occurs within the scope of a method where a setjmp() is executed, the top object of the global stack is popped before the function returns since the method cannot be a target of a longjmp() any more.

There are other works to do for restoring the environment when returned to the setjmp(). One is restoring the Java stack frame of the returned method. The other is releasing locks from all synchronized methods located between the longjmp() method and the setjmp() method on the call stack. In order to implement this, we need to maintain another global stack of locked objects such that all synchronized methods are required to push their locked object on the global stack at the beginning and pop it at the end (even if there is no try block in the method). When returned to the setjmp() of a synchronized method, we pop all locked objects from the stack until this method's locked object is exposed. Then, we release all of popped objects' locks.

After restoring the environment at the setjmp(), we need to jump to an appropriate catch block. There have been two approaches depending on where to place the setjmp(), especially if there are multiple try blocks in a method. One is placing a single setjmp() at the beginning of a method. The other is placing a setjmp() at the beginning of each try block. The setjmp()-per-method approach is advantageous if more than one try block is executed in the method since the setjmp() will be executed only once, whereas multiple setjmp()s will be executed with the setjmp()-per-try approach (also the two global stacks are pushed/popped at the beginning/end of each try block). However, when returning from longjmp(), setjmp()-per-method requires finding which try block caused the exception before searching for the catch block associated with it. In order to help this, we should save the bytecode PC before making a method call within a try block, which will be used later to consult the exception table for finding appropriate catch blocks to jump to. For setjmp()-per-try, this is not necessary since we know which catch blocks are associated with each try block. The setjmp()-per-try will be advantageous if no try blocks are executed in a method. Figure 3 depicts some part of simplified pseudo code for stack cutting.

One more overhead of stack cutting is that for a method that has a try block with a method call in it we need to keep all local C variables in memory, not in registers, by declaring them volatile. Since longjmp() restores register values as was when setjmp() was executed, if a local variable is allocated to a register and if it is changed after setjmp() is executed, longjmp() will incorrectly restore its previous value. Stack C variables are exempted from this requirement and can be allocated to registers since they keep only temporary values and thus are not used after setjmp().

The setjmp()-per-method approach was employed in [4,6], while the setjmp()-per-try approach was employed in [5], but there have been no evaluation of both techniques. In this paper, we perform such an evaluation for them along with our proposed technique. We implemented stack cutting as efficiently as possible to make a fair comparison. For example,

- We place setjmp() and its related code in Figure 3 only when there is a method call within a try block. If there is no method call, there is no need for placing a setjmp().
- Instead of saving bytecode PC in setjmp()-per-method, we save the label of the first catch block associated with a try block such that when returned from longjmp(), we jump to the first catch block directly instead of consulting with an exception table (this works for gcc only). The catch block will test the exception type and jump to the next one if not matched.
- When there is an exception in a method and if it can be handled there, the search process for the catch block is the same as in our proposed technique which is based on sequential search.

(a) At the method entry or at the try block entry

`Push a new jmpbuf object on the global jmpbuf stack`
`Push "this" object on the global locked object stack`
`if( setjmp(jmpbuf) ) { // setjmp() return 0 initially, 1 when returned from longjmp()`
`  If (synchronized method)`
`    pop locked objects from the global stack upto "this" object and release their locks`
`  Restore Java stack`
`  s0_ref = global exception object`
`  Find an appropriate exception handler`
`  Jump to the exception handler;`

(b) At the method exit or at the try block exit
`Pop a jmpbuf object from the global jmpbuf stack` `Pop an object from the global locked object stack`
(c) When there is an exception that cannot be handle in the method
`Make and save an exception object to global variable;` `Longjmp ( the top jmpbuf object at the global jmpbuf stack, 1 );`

Figure 3. Overhead of Stack Cutting on Normal Execution Paths

4. Proposed Exception Handling based on Exception Checks

Stack cutting described in the previous section includes many overheads for normal execution paths. The setjmp() library call itself is a very costly operation. We should maintain global stacks of `jmpbuf` objects and locks at the entry/exit of methods or try blocks, even when there is no exception raised. Local C variables for methods with try blocks should be kept in memory. Bytecode PC should be saved at memory before making a call in setjmp()-per-method.

In order to reduce these overheads for normal execution, we propose a simpler solution based on exception checks. When an exception occurs in a method but there is no catch block that can handle it, the method simply returns to the caller. In the caller, we check if an exception occurred in the callee and if so, try to find an exception handler in the caller. If there is no exception handler, the method also returns and this process repeats until an exception handler is found. This means that we need to add an exception check code after every method call, which would certainly be an overhead, especially because its dynamic count will be much higher than that of setjmp()s. However, it is a single comparison while stack cutting includes substantial work to do.

In fact, we can reduce this overhead by merging the exception check code with the garbage collection (GC) check code. CVM is employing precise GC with a moving GC algorithm [18], which can move objects during GC. If there is a GC in a callee method, the caller method needs to restores all live reference-type C variables from the Java stack frame because their reference values (addresses) might be incorrect if their referenced object moved during GC. So after every method call, there is GC check code. Since GC would also be an "exceptional" event like an exception, and since both should be checked after a method call, we can merge both checks.

Our idea is declaring a global variable called "`global_gc_and_exception_count`" which counts the number of GCs or exceptions occurred during execution. There is a local variable declared in each method called "`local_gc_and_exception_count`". At the beginning of a method, `local_gc_and_exception_count` initialized by `global_gc_and_exception_count.` After a method call, we check if both variables are still the same. If not, we perform appropriate actions depending on if GC or an exception occurred. The point is that both variables will be the same most of the time, which can obviate the overhead of a separate exception check. Figure 4 depicts the merged GC and exception check code.

JNI also employs the exception checks for the callee, yet we merged the exception checks with GC checks by using a global variable, which makes the check overhead to zero cost.

When an exception occurs in a method and if there is a catch block that can handle the exception in the same method, it should obviously be handled there instead of returning to the caller. In order to handle this case together with the return-and-exception-check case, our exception handling proceeds as follows. Each exception handler code is divided into two parts: the check code to see if a given exception can be handled there and the handler code itself. The AOTC analyze the bytecode to figure out exception types that can possibly occur in a method and exception types that can be handled in the method. If there is a match, we make a direct jump to the corresponding handler code. For other exceptions, we simply jump to the first exception check code that can possibly handle them. If they cannot be handled there, there is a jump to the next exception check code, and so on. If the last exception check code cannot handle them, jump to the method epilog where save the exception object in a global variable and return to the caller. For stack cutting we implemented this case in the same way.

```
Some_method_call();
If( local_gc_and_exception_count!=global_gc_and_exception_count ) {
  local_gc_and_exception_count = global_gc_and_exception_count;
  Restore all live local reference variables from Java stack frame;
  AOTCExceptionOccurred(ee, exc);
  If( exc != NULL ) {
    s0_ref = exc;
    Find and jump to the exception handler;
  }
}
```

Figure 4. The check code after method call.

5. Experimental Results

Previous sections described stack cutting techniques based on setjmp()/longjmp() and our proposed techniques based on exception checks. Although exception checks would occur much more frequently than execution of setjmp() during normal execution, the individual overhead of setjmp() and its related work to do would be higher than that of an exception check. It would be worthwhile to evaluate these techniques on the same environment.

5.1 Experimental Environment

We experimented with Sun's CVM for which we implemented a bytecode-to-C AOTC. On our AOTC we implemented four cases of exception handling: two stack cutting techniques, setjmp()-per-method and setjmp()-per-try, and two exception check techniques, merged check and separate check. We experimented with these four cases.

The experiments were performed on an Intel Pentium4 2.40 GHz CPU with 512M RAM and the OS is Debian Linux with kernel 2.6.8-2. The translated C code is compiled by GNU C compiler (GCC) version 3.3.5. The CVM is constrained to have 32M memory. The benchmarks we used are SPECjvm98 (except for mpegaudio for which CVM cannot read its class files).

5.2 Performance Comparison

We first measured how many exceptions occur in the benchmarks. Table 1 shows the number of exceptions raised and the average number of call depth differences between the excepting method and exception handling method[19]. It shows that exceptions indeed occur rarely except for javac and jack (all exceptions in others are UnsupportedEncodingException, which occur due to the character encoding in our Linux machine but does not affect the execution results). And the call depth difference is short, so the processing overhead for raised exceptions would be small.

Table. 1 Exception behavior of SPECjvm 98 in CVM

Benchmarks	Exception Count	Call Depth Difference
compress	7	2.0
Jess	10	2.0
Db	10	2.0
Javac	22408	1.9
Mtrt	7	2.0
Jack	241934	2.5

We first compare the running time of those four cases, which is depicted in Figure 5. The graph shows that exception checks are generally better than stack cutting, especially for javac, db, jess and jack. For stack cutting, setjmp()-per-try is a little better than setjmp()-per-method. For exception checks, merged exception check is slightly better than separate exception check. Figure 6 shows the performance percentage compared to merged exception checks.

In order to evaluate the overhead of stack cutting for normal execution paths, we performed another experiment where we added the overhead code of the setjmp()-per-method or the setjmp()-per-try in Figure 3 (a) and (b) to our merged exception check code so that setjmp() code is executed additionally for normal execution paths. Since exceptions will be handled by our exception check code, this will just simulate what stack cutting does for normal paths. We found that the running time difference between this simulation code and our original exception check code is the same as the difference between stack cutting and merged exception check in Figure 5. This means that the overhead code of stack cutting is the reason for its longer running time in Figure 5 and the actual exceptions raised do not affect the running time seriously.

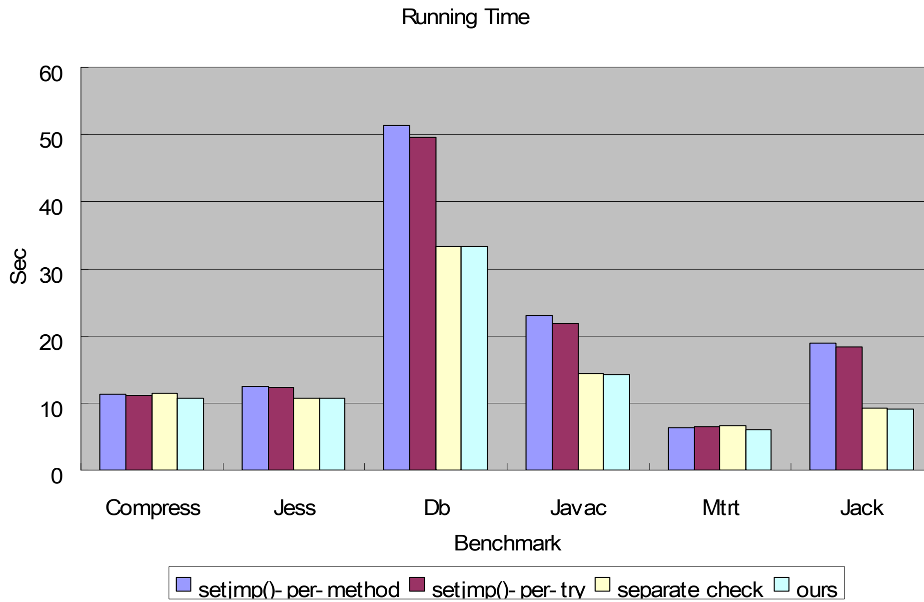

Figure 5. Running time for each case (seconds).

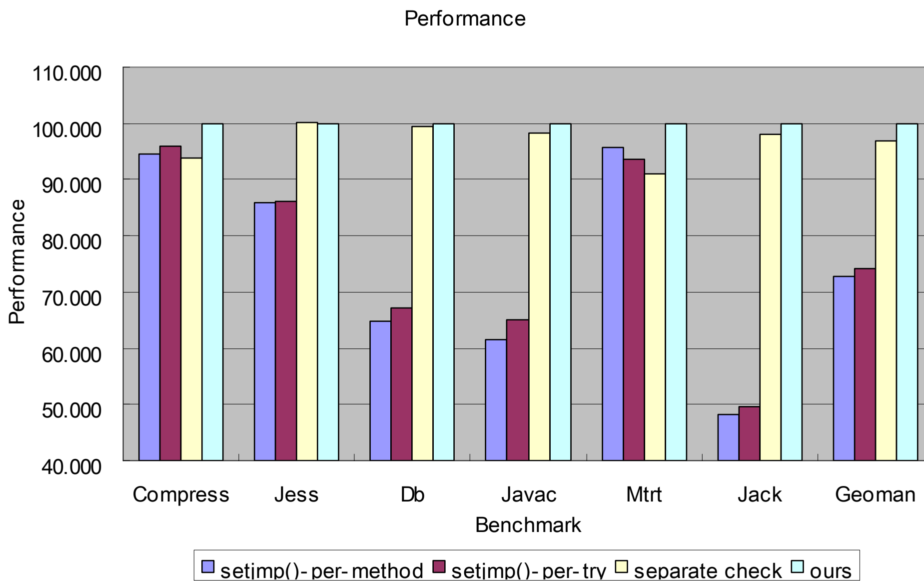

Figure 6. Performance compared to merged exception checks

We also measured the dynamic numbers of setjmps() in stack cutting cases, which are shown in Table 2. We also measured the dynamic numbers of exception checks in Table 3. Generally, the ratio of setjmp() frequency to exception check frequency is higher for those benchmarks where stack cutting is slower than exception checks in Figure 5. This also indicates that the overhead caused by setjmps() and its related code in Figure 3 (a) and (b) would be the reason for the performance loss. There does not appear to be a direct relationship between the setjmp() frequency and the performance between setjmp()-per-method and setjmp()-per-try, though, probably due to insufficient difference of frequencies to make a correlation.

Table 2. Dynamic Counts of setjmp() (thousands)

Benchmarks	setjmp()-per-method	setjmp()-per-try
compress	3.1	20
jess	35	67
db	158	93
javac	1,615	62
mtrt	32	270
jack	4,365	35

Table 3. Dynamic Counts of Exception Checks (millions)

Benchmarks	merged checks
compress	20
jess	67
db	93
javac	62
mtrt	270
jack	35

6. Summary

A bytecode-to-C AOTC is one of the most promising approaches to embedded Java acceleration but exception handling should be implemented efficiently in order not to affect performance, especially in normal execution since exception would be a rare event. Stack cutting based on setjmp()/longjmp() includes too much overhead for normal execution paths. We proposed a simpler technique based on an exception check after every method call. We found this is more efficient than stack cutting, especially when the exception check is merged with the GC check in the CVM, which leads to 25% better performance than stack cutting.

References

[1] Sun Microsystems, White Paper "CDC: An Application Framework for Personal Mobile Devices"

[2] J. Gosling, B. Joy, and G. Steele, The Java Language Specification Reading, Addison-Wesley, 1996.

[3] J. Aycock. "A Brief History of Just-in-Time", ACM Computing Surveys, 35(2), Jun 2003

[4] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: Java for Applications A Way Ahead of Time (WAT) Compiler," Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, Portland, Oregon, 1997

[5] G. Muller, B. Moura, F. Bellard and C. Consel, "Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code", In Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems pages 1-20, Berkeley, June 16-20 1997.

[6] A. Varma, "A Retargetable Optimizing Java-to-C Compiler for Embedded Systems," MS thesis, University of Maryland, College Park, 2003

[7] M. Weiss, F. de Ferriere, B. Delsart, C. Fabre, F. Hirsch, E. Andrew Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler, "TurboJ, a Java Bytecode-to-Native Compiler", 1998.

[8] M. Serrano, R. Bordawekar, S. Midkiff and M. Gupta, Quicksilver: A Quasi-Static Compiler for Java, In Proceedings of the ACM 2000 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'00). Pages 66-82, 2000.

[9] V. Mikheev, N. Lipsky, D. Gurchenkov, P. Pavlov, V. Sukharev, A. Markov, S. Kuksenko, S. Fedoseev, D. Leskov, A. Yeryomin, "Overview of Excelsior JET, a High Performance Alternative to Java Virtual Machines", in Proceeding of the third international workshop on Software and performance WOSP'02, ACM Press. 2002.

[10] R. Fitzgerald, T. B. Knolock, E. Ruf, B. Steensgaard, and D. Tarditi, "Marmot: An Optimizing Compiler for Java", Microsoft Technical Report 3. Microsoft Research, March 2000.

[11] Instantiations, Inc. JOVE: Super Optimizing Deployment Environment for Java, http://www.instantiations.com., July 1998

[13] Sun Microsystems, CDC CVM, http://java.sun.com/products/cdc

[13] GNU, Gnu Compiler Collection, http://gcc.gnu.org.

[14] T. Ogasawara, H. Komatsu and T. Nakatani, "A Study of Exception Handling and Its Dynamic Optimization in Java", In Proceedings of ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA'01), pages 83-95, 2001

[15] Ramsey, N., and Peyton Jones, S. "A single intermediate language that supports multiple implementations of exceptions". Proceedings of the ACM SIGPLAN 20000 Conference on Programming Language Design and Implementation (PLDI'00), pages 285-298, May 2000.

[16] de Dinechin, C. C++ exception handling. IEEE Concurrency, 8(4), pp. 72-79, 2000.

[17] Cameron, D., Faust, P., Lenkov, D., And Mehta, M. A portable implementation of C++ exception handling. In Proceedings of the C++ conference, USENIX Association, pp.225-243, Aug. 1992.

[18] Cohen, J., AND Nicolau, A. "Comparison of compacting algorithms for garbage collection". ACM Transactions on Programming Languages and Systems 5(4), pp. 532-553, October 1983.

[19] S. Lee, B. Yang, S. Moon, et al. "Efficient Java exception handling in just-in-time compilation" In Proceedings of the ACM 2000 Conference on Java Grande (New York, NY, USA, June 2000), ACM Press, pp. 1-8

[20] J. Lang, D. B. Stewart, "A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology" TOPLAS'98