Monitors and Exceptions: How to implement Java efficiently

Andreas Krall and Mark Probst
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8
A-1040 Wien
http://www.complang.tuwien.ac.at/andi/
http://www.unix.cslab.tuwien.ac.at/~schani/

Abstract

Efficient implementation of monitors and exceptions is crucial for the performance of Java. One implementation of threads showed a factor of 30 difference in run time on some benchmark programs. This article describes an efficient implementation of monitors for Java as used in the CACAO just-in-time compiler. With this implementation the thread overhead is less than 40% for typical application programs and can be completely eliminated for some applications. This article also gives the implementation details of the new exception handling scheme in CACAO. The new approach reduces the size of the generated native code by a half and allows null pointers to be checked by hardware. By using these techniques, the CACAO system has become the fastest JavaVM implementation for the Alpha processor.

1 Introduction

Java's [1] main success as a programming language results from its role as an Internet programming language through the machine independent distribution of programs with the Java virtual machine [2]. Additional reasons for its success are:

- easy to use object-oriented language
- security and safety (bound checks and exception handling)
- support for multithreading
- integrated garbage collection

Some of these features of Java can decrease the performance of Java applications drastically if implemented in the wrong way. A bad decision in the object layout can lead to an unnecessary indirection for a field access. If run time bound checks are not removed they can consume a large percentage of the run time. Inefficient implementation of synchronization can also lead to a huge performance degradation. The next section introduces basic implementation techniques for Java. The remaining parts of this article give the details of the thread and exception implementations.

1.1 JavaVM implementation basics

For portability reasons, the first JavaVM implementations were interpreter based and have therefore been very slow. Faster implementations are possible using just-in-time compilers which translate Java byte code on demand into native code. We developed such a JIT-based JavaVM system called CACAO, which is described in [3]. CACAO is freely available via the world wide web.

Conventional compilers are designed for producing highly optimized code without paying much attention to their compile time performance. The design goals of Java just-in-time compilers are different: they must produce fast code in the smallest possible compilation time. CACAO uses a very fast linear time algorithm for translating JavaVM byte code to high quality machine code for RISC processors. It has three passes: basic block determination, stack analysis and register preallocation, final register allocation and machine code generation. The most important optimization is elimination of unnecessary copy operations and is done implicitly during stack analysis. Stack analysis tracks the definition and use of local variables, establishing correspondences between stack locations and local variables. Register allocation afterwards uses local variables instead of stack locations.

The SUN JDK represents an object by a cell with two pointers: the first points to the instance data of the object, the second to the class descriptor [4]. CACAO's representation eliminates one unnecessary indirection [3]. The object itself contains the pointer to the class descriptor and the instance data. In addition to other information, the class descriptor contains the virtual function table and, at negative offsets, pointers to the interface virtual function tables.

2 Threads

2.1 Introduction

Threads are an integral part of the Java programming language. A Java Runtime Environment (JRE) has to implement threads to be compliant. A thread implementation includes the creation, execution, switching, killing and synchronization of threads. In Java the latter is provided by monitors. Monitors ensure that, for a given object, only one thread at a time can execute certain methods, known as synchronized methods, and blocks which are marked by the keyword `synchronized`.

Monitors are usually implemented using mutex (mutual exclusion). A mutex is a data structure which contains the necessary information to guarantee that only one unit of execution can perform a critical section at the same time [5].

As we show in section 2.4 a fast implementation of the synchronization mechanism is crucial for the efficiency of Java. One implementation produced more than 800% overhead in one of our tests.

2.2 Related work

Java threads can be implemented using threads provided by the operating system kernel, as user-level libraries, or as a combination of the two. There exist a number of articles describing different thread implementation aspects but none captures the problem of efficient monitor operations for objects.

Sun's first implementation of the JavaVM on Solaris was based on user-level threads. The current implementation uses a combination of kernel and user-level threads. Some of the advantages of this approach are outlined in [6].

The freely available JavaVM implementation `kaffe` by Tim Wilkinson uses user-level threads [7]. Until version 0.9, each object contained the complete mutex data structure. This enabled a fast monitor implementation but used a lot more memory than necessary.

Apart from thread implementations used in JavaVM's there are many other thread standards and implementations, the most notable being the IEEE POSIX extension [8].

In [9], Mueller describes a library implementation of POSIX threads on a standard UNIX system. This library is a user-level implementation which need no support from the operating system. A very popular library of user-level primitives for implementing threads is *QuickThreads* by David Keppel, described in [10].

Bershad et al. present a fast mechanism for mutual exclusion on uniprocessor systems [11]. They have a software solution for the implementation of an atomic test-and-set operation which is faster than the corresponding hardware instruction. However, their implementation relies on assistance from the operating system.

2.3 Implementation basics

A complete thread implementation for Java has to provide:

- thread creation and destruction,
- low level thread switching (usually implemented in assembly language),
- thread scheduling,
- synchronization primitives,
- a thread safe non-blocking input/output library.

Cacao's current implementation of threads is based mainly on the threading code of `kaffe` version 0.7, which has been released under a BSD-style license and can thus be used freely [7]. As mentioned above, `kaffe`'s threads are completely user-level, which means, for example, that they cannot take advantage of a multiprocessor system.

There are several reasons why we chose this approach:

- Thread support differs from operating system to operating system. Not only do some operating systems provide different APIs to other systems, but even if they do provide the same interface (most often in the form of POSIX threads), the costs of various operations are often very different across platforms.
- It was a complete implementation, tailored for the use in a JavaVM and thus made it possible for us to get thread support up and running quickly.

		mutex test			tree test		
		run time in secs		call cost	run time in secs		call cost
Machine	JavaVM	no sync	sync	in μs	no sync	sync	in μs
DEC 21064A 289MHz	OSF JDK port (1.0.2)	1.20	4.14	9.8	8.37	34.69	9.8
DEC 21064A 289MHz	DEC JDK interpreter	1.71	12.80	36.97	12.25	143.10	39.93
DEC 21064A 289MHz	DEC JDK JIT	1.06	11.05	33.30	7.80	130.50	37.45
Pentium-S 166MHz	Linux JDK 1.1.3	0.96	1.69	2.43	7.93	16.12	2.50
AMD K6 200MHz	Sun JPP JIT (WinNT)	0.57	0.94	1.23	2.8	9.8	2.13
AMD K6 200MHz	Navigator 4.04 (WinNT)	0.03	0.77	2.46	2.3	9.2	2.11
DEC 21064A 289MHz	Cacao	0.28	0.40	0.40	1.46	4.71	0.99

Table 1: Overhead for calling synchronized methods (one object)

- Parts of Cacao are not yet thread-safe (most notably the compiler and the garbage collector), thus making it difficult to use a kernel-supported solution.

Each thread in a Java program corresponds to an instance of the `java.lang.Thread` class. In order for the Java run time environment (JRE) to associate platform specific information with such an instance, it has a private member called `PrivateInfo` of type `int`, which can be used by the JRE. Kaffe version 0.7 used this member as a pointer to a context structure. Since pointers are 64-bit values on the Alpha, we use an array of context structures. `PrivateInfo` is then used as an index into this array.

A fixed-size stack is allocated for each thread. The stack size for ordinary threads can be specified as a command-line parameter. Special threads (such as the garbage collector) have their own stack sizes. In order to catch stack overflows without the burden of checking the stack top at each method entry, we simply guard the stack top with one or more memory pages. The memory protection bits of these pages are modified to cause a memory protection violation when accessed. The number of guard pages can be specified on the command-line.

Thread switching is implemented straightforwardly, namely by saving callee-save registers to the stack, switching to the new stack, restoring callee-save registers and performing a subroutine return.

Scheduling is very simple in Cacao: higher priority threads always get precedence over lower priority threads, and threads of the same priority are scheduled with a round-robin algorithm.

I/O in user-level threads is a problem since UNIX I/O calls are, by default, blocking. They would block not just the current thread but the whole process. This is undesirable. It is thus common practice for thread libraries to use non-blocking I/O and, in the case of an operation which would block, suspend the corresponding thread and perform a multiplexed I/O operation (`select(2)` in UNIX) on all such blocked files regularly, especially if there are no runnable threads available.

2.4 Motivation

To optimize an implementation it is necessary to find the parts which are critical to performance. Therefore, we analysed the cost of monitors with two small test programs. The *mutex test* program simply invoked a method 300000 times, once with the method being declared `synchronized` and once without. The other test, *tree test*, allocated a balanced binary tree with 65535 elements and recursively traversed it 50 times using a method, again once being synchronized and once being not.

Table 1 shows the differences in run-time for the two versions of the programs for several JavaVM's. The table includes the run times for both versions of the programs in seconds. The column 'call cost' gives the overhead of a call of a synchronized method compared to a call of a non-synchronized one. From these numbers it is obvious that calls to synchronized methods, or monitors in general, are much slower than ordinary method calls.

The question that arises is now whether this has any influence on common Java programs. To answer this question, we have used a modified version of `kaffe` to gather statistics about monitor usage. The results are summarized in table 2.

Javac is an invocation of Sun's `javac` on the Toba source files [12] and is thus single-threaded. Execution of this program takes only a few seconds using Cacao with threads disabled. Biss is a more or less typical working session with the Java development environment of the Biss-AWT [13]. It is slightly multithreaded. Jigsaw invokes the HTTP server Jigsaw [14]

Application	Objects allocated	Objects with mutex	Monitor operations	Parallel Mutexes
javac	111504	13695	840292	5
Biss	84939	13357	1058901	12
Jigsaw	215411	23804	855691	25

Table 2: Mutual exclusion statistics

of the World Wide Web Consortium and lets it serve identical parallel requests from seven hosts, amounting to about one megabyte split across 200 files for each request. This application is highly multithreaded.

The table contains the number of objects allocated during execution and the number of objects for which a monitor has been entered. The 'Monitor operations' column gives the total number of operations performed on monitors, while the numbers in the 'Parallel Mutexes' column show the greatest number of mutexes that were locked simultaneously.

These numbers demonstrate that the performance of monitor operations is of paramount importance for a fast JavaVM implementation. We did not include the number of blocking monitor operations because there were just two of them, namely in the Biss application. It was only after we modified `kaffe` to switch to another thread before each `monitorenter` operation that the Biss and Jigsaw applications performed a few thousand blocking `monitorenter`s.

2.5 Mutex implementation

Our mutex data structure includes a pointer to the thread that has currently locked the mutex (`holder`). If the mutex is not locked at all, this is a null pointer. Because one thread can lock the same mutex multiple times, we need a count of how many lock operations without corresponding unlocks have been performed on the mutex (`count`). When it goes down to zero, the mutex is not locked by any thread. Furthermore, we need to keep track of the threads which have tried to lock the mutex but were blocked and are now waiting for it to become unlocked (`waiters`).

The data structure is defined as follows:

```
struct mutex {
  int count;
  thread *holder;
  thread *waiters;
}
```

The locking of a mutex can now be specified as in figure 1.

```
void lockMutex (struct mutex *mux) {
  disablePreemption();

  if (mux->holder == NULL) {
    mux->holder = currentThread;
    mux->count = 1;
  } else if (mux->holder == currentThread) {
    mux->count++;
  } else {
    blockUntilMutexIsNotLocked(mux);
    mux->holder = currentThread;
    mux->count = 1;
  }

  enablePreemption();
}
```

Figure 1: Code for `lockMutex()`

The macro `disablePreemption()` simply sets a global flag indicating that a critical section is currently being executed and that preemption must not take place. `enablePreemption()` unsets the flag and checks whether a thread switch is necessary (see below). On a multiprocessor system this would need to be implemented using a test-and-set instruction, or some equivalent.

The inverse operation, namely the unlocking of the mutex, can be expressed as in figure 2.

The function `resumeThread()` sets a flag which results in a thread switch to the given thread after preemption has been re-enabled.

These algorithms are simple, straightforward and reasonably efficient.

2.6 Object-Mutex relation

Since the JavaVM specification states that each object has a mutex associated with it, the obvious solution seems to be to include the mutex structure in each object. The mutex structure could be reduced to a length of eight bytes if we used thread numbers instead of pointers. But, using such a solution, the javac application would allocate nearly one megabyte of mu-

```
void unlockMutex (struct mutex *mux) {
  disablePreemption();

  --mux->count;
  if (mux->count == 0) {
    mux->holder = NULL;
    if (mux->waiters != NULL) {
      thread *next = mux->waiters;
      mux->waiters = next->next;
      resumeThread(next);
    }
  }

  enablePreemption();
}
```

Figure 2: Code for `unlockMutex()`

tex data, just for a few seconds of execution. This is unacceptable.

On the other hand, the figures show that it is very seldom that more than 20 mutexes are locked at the same time. This suggests that using a hash table, indexed by the address of the object for which a monitor operation is to be performed could be very efficient. This is exactly what Cacao does.

We use a hash function which uses the $2n$ least significant bits of the address where 2^n is the size of the hash table. This function can be implemented in four RISC instructions. Since we ran into performance problems with overflow handling by internal chaining, we now use external chaining with a preallocated pool of overflow entries managed by a free list.

An entry in the hash table has the following structure:

```
struct entry {
  object *obj;
  struct mutex mux;
  struct entry *next;
}
```

In order to minimize the overhead of the locking/unlocking operations, we should strive for code sequences small enough to be inlined, yet powerful enough to handle the common case. We have chosen to handle the first entry in the collision chain slightly differently from the rest by not destroying the associated mutex when the count goes down to zero. Instead the decision is deferred until when a mutex with the same hash code gets locked and would thus use this location. The major benefit of this approach is that the code to lock the mutex need not (in the common case) check

```
void monitorenter (object *o)
{
  entry *e;
  disablePreemption();

  e = firstChainEntry(o);
  if (e->obj == o
      && e->mux.holder
         == currentThread)
    e->mux.count++;
  else
    lockMutexForObject(e,o);

  enablePreemption();
}
```

Figure 3: Code of `monitorenter()`

```
void monitorexit (object *o)
{
  entry *e;
  disablePreemption();

  e = firstChainEntry(o);
  if (e->obj == o)
    e->mux.count--;
  else
    unlockMutexForObject(e,o);

  enablePreemption();
}
```

Figure 4: Code of `monitorexit()`

for the location to be free, since each hash table location will, during the course of execution, only be free at the beginning of the program and will then never be freed again. The unlocking code is simplified by the fact that the code need not check whether the hash table location should be freed. Based on the experience that a recently locked mutex is likely to be locked again in the near future, this technique also improves overall performance.

See figures 3 and 4 for the code of the corresponding JavaVM instructions. These functions handle (as we show below) the most common case and depend on the two functions for the general case presented in figures 5 and 6 (we now assume that `enablePreemption()/disablePreemption()` pairs can be nested).

Even if they are not short enough to be inlined in every synchronized method, these functions are certainly small enough to make the effort of coding them as specially tailored, assembly language functions worth-

Program	Line 6	Line 10	count 0	Line 12	Ratio 12/6
javac	420147	405978	381019	14169	3.372 %
Biss	384350	370171	425038	14179	3.689 %
Jigsaw	426695	391680	294957	35015	8.206 %

Table 3: Execution statistics for `monitorenter()`

```
 1 void lockMutexForObject (entry *e,
 2                          object *o) {
 3   disablePreemption();
 4
 5   if (e->obj != NULL)
 6     if (e->mux.count == NULL)
 7       claimEntry(e,o);
 8     else
 9       while (e->obj != o) {
10         if (e->next == NULL) {
11           e = e->next = allocEntry(o);
12           break;
13         }
14         e = e->next;
15       }
16   else
17     e->obj = o;
18
19   lockMutex(&e->mux);
20
21   enablePreemption();
22 }
```

Figure 5: Code for `lockMutexForObject()`

```
 1 void unlockMutexForObject (entry *e,
 2                            object *o) {
 3   disablePreemption();
 4
 5   if (e->obj == o)
 6     unlockMutex(&e->mux);
 7   else {
 8     /* Assuming entry is there */
 9     while (e->next->obj != o)
10       e = e->next;
11     unlockMutex(&e->next->mux);
12     if (e->next->mux.count == 0)
13       e->next = freeEntry(e->next);
14   }
15
16   enablePreemption();
17 }
```

Figure 6: Code for `unlockMutexForObject()`

Program	Line 6	Line 8	Line 10	Ratio 10/6
javac	420146	419815	331	0.078 %
Biss	384368	383281	1087	0.282 %
Jigsaw	428997	416890	12107	2.822 %

Table 4: Execution statistics for `monitorexit()`

while. This bypasses the standard subroutine linkage conventions, gaining a little extra speed.

2.7 Results

To demonstrate that nearly all cases are indeed handled by these small routines, we have written a small application which simulates the locking and unlocking operations of the three applications we used above (tables 3 and 4). As can be seen, only a small percentage of cases need to be handled in the general routines `lockMutexForObject()` and `unlockMutexForObject()`.

The column 'count 0' gives the number of mutex locks where the `count` of the mutex before locking is zero. We present this number to evaluate an implementation where `monitorenter()` checks the `count` instead of the `holder` to determine if the mutex can be locked immediately. The numbers seem to be in favour of checking `holder` but at least indicate that the difference is neglectable. Furthermore, locking the mutex after only having checked `count` needs an additional assignment (setting `holder`).

We have also considered the possibility of using a cache of recently used mutexes to improve performance, similar to a translation-lookaside buffer in microprocessors which cache the mapping between virtual and physical memory pages. To evaluate whether this would be worthwhile, we have simulated caches with one, two, four and eight elements using the three applications as test candidates. We have used least-recently-used as the cache replacement strategy. Though this is not easily implemented in software, it provides a good estimate of the best hit rate that can be achieved with an efficient implementation. Table 5 summarizes the results.

Using an implementation supporting the monitor

Application	miss rates by size of cache 1 Element	2 Elements	4 Elements	8 Elements
javac	15.076 %	9.757 %	4.931 %	3.193 %
Biss	13.488 %	8.349 %	4.765 %	3.141 %
Jigsaw	43.694 %	37.700 %	22.680 %	5.182 %

Table 5: Results of cache simulation

	JavaLex	javac	espresso	Toba	java_cup
run time without threads	2.82	4.91	3.23	4.32	1.35
run time with threads	3.89	5.40	3.23	5.53	1.56
overhead (optimized impl.)	37 %	10 %	0 %	28 %	15 %
number of lock/unlock pairs	1818889	420146	2837	1558370	124956

Table 6: Overhead of monitor operations

routines, as discussed in section 2.6, and one implementation without thread support, we have run several applications on a 300MHz DEC 21064A (see table 6). For these single threaded applications, the overhead introduced by monitor operations ranges from 0% to 37%, depending on the number of monitor operations in the applications. Note, however, that this cannot be compared to the overhead figures given in table 1, since these applications do more than just entering and exiting a monitor.

Using the implementation described, the *mutex test* application for table 1 took 0.40 seconds with a synchronized and 0.28 seconds with an ordinary method to complete. In this program the time spent on locking/unlocking a mutex is 0.40 microseconds. The reason for the higher cost of mutex operations in the *tree test* is that this test violates the locality of monitor operations. Overall, these numbers compare very favorably with the other implementations.

For most single-threaded applications, the monitor overhead can be eliminated completely. If it is possible to determine statically that the dynamic class-loader and the `java.lang.Thread` class are not used, synchronization code need not be generated.

3 Exception handling

3.1 Introduction

Exceptions in Java occur either implicitly or explicitly. Typical implicit exceptions are references to the `null` pointer, array index out of bounds and division by zero. Exceptions also can be raised explicitly with the `throw` instruction. To handle exceptions occurring during execution, code which can raise an exception is included in a `try` block. An efficient implementation of exception handling has to take care of managing `try` blocks and to check for implicit exceptions efficiently .

3.2 Known implementation techniques

Three standard methods exist for implementing exception handling:

- dynamically create a linked list of `try` block data structures,
- use static `try` block tables and search these tables at run time (suggested for JavaVM interpreters),
- use functions with two return values.

The first method has been used in portable implementations of exception handling for C++ [15] or Ada [16] using `setjmp` and `longjmp`. A linked exception handling data structure is created when entering a `try` block and the structure is discarded when leaving the protected block. Java requires precise exceptions. It means that all expressions evaluated before the exception raising instruction must have finished and all expressions after the raising instruction must not have been started. Therefore, in practice, some instructions may not be moved freely. In the case of subroutine calls, the callee-saved registers must be restored to their original value. The data structure can be used to store such information. The disadvantage of this method is that creating and discarding of the data structure takes some time even if an exception is never raised.

The second method has been suggested for an efficient exception handling implementation of C++ [17] and is used in Java implementations. For every method, the JavaVM maintains an exception table. This exception table contains the program counter of the start and the end of the `try` block, the program counter of the exception handler and the type of the exception. A JavaVM interpreter can easily interpret this structure and dispatch to the corresponding handler code. If the byte code is translated to native code, the equivalent technique is more complicated.

To simplify restoration of the registers, the old CACAO implementation used a different scheme [3]. A method has two return values: the real return value and an exception value stored in a register. After each method call, the exception register is checked and, if it is non-zero, the exception handling code is executed. Since an exception is rarely raised, the branch is easy to predict and cheap. Entering and leaving a `try` block have no associated costs. At compile time, the dispatch information contained in the exception table is translated into method dispatching native code.

Run time checks for null pointers and array bounds are quite frequent, but can be eliminated in many cases. It is often possible to move a loop invariant null pointer check before the loop or to eliminate a bound check. Some more sophisticated compilers use these code motion techniques.

3.3 Motivation for a change

The old CACAO implementation was simple, but it only makes sense if the number of try blocks is high. We made an empirical study to count the numbers of static occurrences of method invocations and of `try` blocks in some applications (see table 7). The number of method invocations is two magnitudes bigger than the number of `try` blocks. Furthermore, an exception is rarely raised during program execution. This led us to a new implementation of exception handling.

3.4 The new exception handling scheme

The new exception handling scheme is similar to that in a JavaVM interpreter. If an exception occurs, information in the exception table is interpreted. However native code complicates the matter.

The pointers to Java byte code must be replaced by pointers to native code. It requires that, during native code generation, the order of basic blocks not be allowed to change. If basic blocks are eliminated because of dead code, the information about a block can not be discarded if there is a pointer to it in the exception table.

A CACAO stack frame only contains copies of saved or spilled registers. There is no saved frame pointer. The size of a stack frame is only contained in the instructions which allocate and deallocate the stack. Therefore, to support exception handling, additional information has to be stored elsewhere.

The code for a method needs access to constants (mostly address constants). Since a global constant table would be too large for short address ranges and, because methods are compiled on demand, every method has its own constant area which is allocated directly before the start of the method code (see fig. 7). A register is reserved which contains the method pointer. Constants are addressed relative to the method pointer.

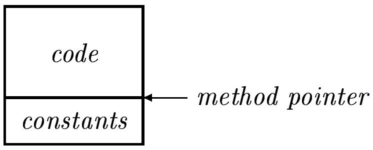

Figure 7: CACAO method layout

During a method call, the method pointer of the calling method is destroyed. However the return address is stored in a register which is preserved during execution of the called method and has to be used for returning from the method. After a method return, the method pointer of the calling method is recomputed using the return address. The following code for a method call demonstrates the method calling convention:

```
LDQ cp,(obj)      ; load class pointer
LDQ mp,met(cp)    ; load method pointer
JSR ra,(mp)       ; call method
LDA mp=ra+offset ; recompute method pointer
```

At the beginning of the constant area, there are fields which contain the necessary information for register restoration:

`framesize`	the size of the stack frame
`isleaf`	a flag which is true if the method is a leaf method
`intsave`	number of saved integer registers
`floatsave`	number of saved floating point registers
`extable`	the exception table – similar to the JavaVM table

The exception handler first checks if the current executing method has an associated handler and may dispatch to this handler. If there is no handler, it unwinds the stack and searches the parent methods for a handler. The information in the constant area is used to

	JavaLex	javac	espresso	Toba	java_cup
null pointer checks	6859	8197	11114	5825	7406
method calls	3226	7498	7515	4401	5310
try blocks	20	113	44	28	27

Table 7: Number of pointer checks, method invocations and try blocks

	JavaLex	javac	espresso	Toba	java_cup
CACAO old	61629	156907	122951	67602	87489
CACAO new	37523	86346	69212	41315	52386

Table 8: Number of generated native instructions

restore the registers and update the stack pointer. The return address and the offset in the immediately following LDA instruction is used to recompute the method pointer.

The change to the new scheme allowed us to implement the null pointer check for free. We protect the first 64 Kbyte of memory against read and write access. If an bus error is raised, we catch the signal and check if the faulting address is within the first 64K. If this is the case, we dispatch to our exception handler, otherwise we propagate the signal. This gives faster programs and reduces the work of the compiler in generating pointer checking code. As shown in table 7, the numbers of null pointer checks are quite high.

3.5 Results

We measured the improvement of the new exception handling scheme. We could not measure a noticeable improvement in the run time (compilation time not included). The improvement seemed to be around 3% but the inaccuracy of the measurements were in the same range. The cost of the exception register check was too small. But the total execution time was smaller because of faster compilation due to the reduction in size of the generated code (see table 8). The code size was nearly halved. One reason was that, for simplicity, the old compiler did not share exception dispatch code and this led to additional code growth.

4 Conclusions and further work

We have presented an efficient implementation of monitors and exceptions for Java. The thread overhead is less than 40% for typical application programs with our implementation and can be removed completely for some applications. A new exception handling implementation halved the size of the generated native code compared to our previous implementation. The CACAO system using these techniques is currently the fastest JavaVM implementation for the Alpha processor. CACAO can be obtained via the world wide web at `http://www.complang.tuwien.ac.at/java/cacao/`.

Acknowledgement

We express our thanks to David Gregg, Michael Gschwind and Nigel Horspool for their comments on earlier drafts of this paper. We would also like to thank the reviewers for their helpful suggestions.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[2] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.

[3] Andreas Krall and Reinhard Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[4] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'29)*, 1996.

[5] William Stallings. *Operating Systems.* Prentice Hall, 1995.

[6] Java threads whitepaper. `http://java.sun.com/`, 1997.

[7] Tim Wilkinson. KAFFE: A free virtual machine to run Java code. `http: //www.kaffe.org`, 1997.

[8] Standard for threads interface to POSIX. IEEE, P1003.1c, 1996.

[9] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, January 1993.

[10] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[11] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 223–233. ACM, October 1992.

[12] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications. Technical report, University of Arizona, Tucson, AZ, 1997.

[13] Peter Mehlitz. Biss AWT. `http://www.biss-net.com/biss-awt.html`, 1997.

[14] Jigsaw. `http://www.w3.org/Jigsaw/`, 1997.

[15] Don Cameron, Paul Faust, Dmitry Lenkov, and Michey Mehta. A portable implementation of C++ exception handling. In *C++ Technical Conference*, pages 225–243. USENIX, August 1992.

[16] E. W. Giering, Frank Mueller, and T. P. Baker. Features of the Gnu Ada runtime library. In *TRI-Ada '94*, pages 93–103. ACM, 1994.

[17] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. *Journal of Object Oriented Programming*, 3(2):16–33, July/August 1990.