A Retargetable JIT Compiler for Java

Michael Golm, Hans Kopp,
Jürgen Kleinöder

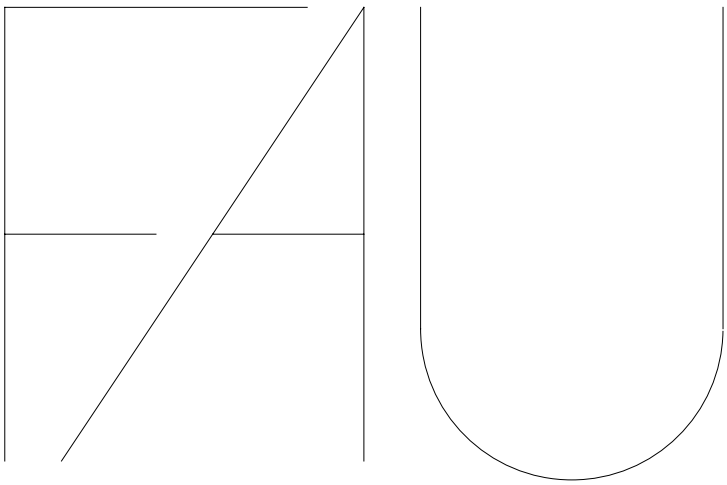
Jan 1999 TR-I4-99-10

Technical Report

Computer Science
Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany



A Retargetable JIT Compiler for Java

Michael Golm, Hans Kopp, Jürgen Kleinöder
University of Erlangen-Nürnberg
Dept. of Computer Science IV (Operating Systems)
Martensstr. 1, D-91058 Erlangen, Germany
{golm, kopp, kleinoeder}@informatik.uni-erlangen.de

Abstract

Interpreted Java has poor performance. Therefore most Java Virtual Machines (JVM) contain a just-in-time compiler (JIT) that compiles the bytecode to the processor's native instruction set. Most of these compilers are written in C and tightly integrated with the JVM. We describe a JIT compiler that is written in Java and retargetable to a different JVMs and CPUs.

1 Introduction

Modern JVMs contain a JIT compiler that improves the JVM performance for about 5 to 10 times. However, most of these JIT compilers have two fundamental deficiencies. They are written in C for the reason of execution speed and they are tightly integrated with the JVM.
Writing the compiler in C means that it is very difficult to extend it and integrate new optimization algorithms once they become available. It is also impossible to make application specific optimizations by downloading code into the compiler. Writing the compiler in Java means that the compiler can be customized to include application specific optimizers (as in [Mats98]).
Integrating the JIT with the JVM allows the compiler to make assumptions about the JVM that ease the task of code generation and allow several performance optimizations. However, it is difficult to enhance the JIT and JVM independently and it is impossible to use one JIT for different JVMs.
This paper makes the following contributions: It describes the problems that must be solved when a Java written compiler interacts with other parts of the JVM. The paper describes an object-oriented implementation of such a compiler that works with our own JVM metaXa [Gol97]. The paper is structured as follows.
The first part describes how our compiler can be ported to a different CPU. The second part discusses the problems of retargeting the compiler to a different JVM. Then we describe the architecture of our compiler together with our solutions to the retargetability problems. We continue by describing several optimizations that are implemented in our compiler or could be integrated in the compiler. Finally we give some preliminary performance figures.

2 Retargeting to different CPUs

There are several approaches to make a compiler retargetable to different CPUs [Fras96]. One is to use a register transfer language (RTL) [Tie89]. We decided not to use an RTL, because it imposes an overhead that seems not acceptable for a JIT compiler. Instead we separated the compiler into a frontend and backend in a similar way as in the kaffe JIT [Wil+97]. The following section describes the backend, that must be changed when compiling for a different CPU.

2.1 The backend

The backend encapsulates the details of the target CPU. It is implemented by the class IMCode (see Figure 1). The interface of IMCode contains many methods that start with the prefix "add". These methods can be considered an intermediate language. Invoking such a method generates code for a new instruction of this intermediate language. The IMCode interface also contains methods to generate code for relative jumps. When all instructions of a method are generated, jump offsets for these relative jumps can be computed.
The current implementation of IMCode generates code for the Intel x86 family of CPUs.

```
class IMCode {
    void addMoveIntS_S(int dstSlot, int srcSlot); // copy between slots
    void addMoveIntS_MB(int dstSlot, int baseSlot, int offset); // load memory cell into slot
    void addInstrIntS_S(int operator, int dstSlot, int srcSlot); // arithmetic operations
    void addInstrCnvsS(int slot, int dstType, int srcType); // type conversions
    void addJump(UnresolvedJump jumpObject); // unconditional jump
    void addJumpTarget(UnresolvedJump jumpObject); // jump target
    void addCondJumpIntS_S(int condition, int slot1, int slot2, UnresolvedJump jumpObject);
                    // conditional jump
}
```

Figure 1:The backend IMCode

Besides the code generator the backend also contains a register allocator that uses a virtual stack.

2.2 The virtual stack

The virtual stack is an array that has the same size as the stack frame of the method. Every array element describes where the value of the stack element is located (register, main memory) or how it can be computed. Register allocation is done using the virtual stack. If an machine instruction needs its operands in a register, the virtual stack is instructed to move contents the stack cell into a register. It remembers that the stack cell is now available in a register. To find a register it first tries to find an unused register. If there is no unused register, the virtual stack selects a used register, remembers to transfer the contents of a register to main memory, and use this register. The virtual stack is implemented by the class RegisterAllocStack (see Figure 2).

The method in Figure 3 uses the IMCode code generation interface and the RegisterAllocStack register allocator to generate code for an integer addition of two slots. The first slot is loaded into a register which is then added to the second slot. If the first slot was already associated with a register, RegisterAllocStack keeps track of this and does not generate code for loading the slot into a register in such a case.

3 Retargeting to different JVMs

This section describes the problems that must be coped with when retargeting the compiler to a different JVM. The discussed retargetability problems must be coped with even if the compiler is written in C. The Microsoft JIT compiler [MS98], which is written in C, defines an interface between the JIT compiler and the JVM to retarget the JIT to a different JVM. However, their compiler is allowed to make too many assumptions about the structure of the JVM. Some of these assumptions that do not hold for other JVMs are that the first element of the object's data is the vtable pointer or

```
class RegisterAllocStack {
    int allocRegisterFor(int slot); // allocate register for a specific slot, return register ID
    int allocFreeRegisterFor(int slot); // load register with slot value, but don't assiciate register with slot
    void freeSlot(int slot); // write register contents back into stack slot
    void assignRegisterTo(int slot, int register); // assign a register to a slot
    InstrOperand slotOpConst(int slot, InstrOperand op); // get operand for slot access
}
```

Figure 2:The virtual stack RegisterAllocStack

```
void makeCodeIntAdd {
    int srcReg1 = virtualStack.allocFreeRegisterFor(srcSlot1); // allocate register for first operand
    targetCode.addInstr(IntInstr.ADD, i1.register(srcReg1), virtualStack.slotOpConst(srcSlot2, i2));
            // generate code
    virtualStack.assignRegisterTo(dstSlot, srcReg1); // assign result register to target slot
}
```

Figure 3:Using IMCode and RegisterAllocStack

that the calling convention of compiled methods is __pascal. These assumptions are not valid for other JVMs, like Sun's JDK which encodes the vtable pointer in the objects handle [YeLi97]. One could also imagine a JIT compiler that uses the C calling convention instead of the __pascal calling convention. In fact, our JIT compiler uses the C calling convention.
We think that making it easy to port the compiler to a different JVM requires a clean interface between the native execution environment and the Java execution environment (see Figure 4).
Major obstacles in the way of making the compiler retargetable are exception handling, the garbage collector, stack layout, and calling conventions.

3.1 Exception handling

When an exception occurs, the appropriate exception handler must be found. If no matching handler can be found in the current stack frame, the stack must be unrolled until a handler is found or the stack contains no more stack frames (in which case the thread terminates). Stack unrolling could be difficult if the thread has executed code that uses different stack layouts. This could happen if native code or JIT compiled code was executed alternating with interpreted Java code. In ordinary JITs such a problem does not appear because all code is compiled. When compiled code and interpreted code is executed alternating, the layout of both native and interpreter stack must be known to unroll the stack. Our architecture aims at encapsulating details like the stack layout between the interpreter and the compiler. Therefore we do not unroll the stack at the exception point but use another technique, which is also used in [Gra97]. In [Gra97] every method indicates with a second return value whether an exception occurred. The method checks this return value and unrolls only its own stack frame when an exception is indicated. This way the method only has to know its own stack layout and information about its own exception handlers. The drawback of this solution is the additional overhead of checking this return value after each method call. In metaXa an exception is indicated in a variable in thread specific storage. When the compiled code tests this variable it has to look for a handler in the current method. The JVM provides a helper function for this purpose. The helper function gets the instruction pointer and the method id and looks in a table if a handler for the exception exists. It returns the address of this handler or a null pointer if no handler was found. The native code then jumps to this address.

3.2 Garbage Collection

A variety of garbage collection (GC) algorithms [JoLi96] is used in today's JVMs. GC algorithms can be classified in conservative GC and exact GC.

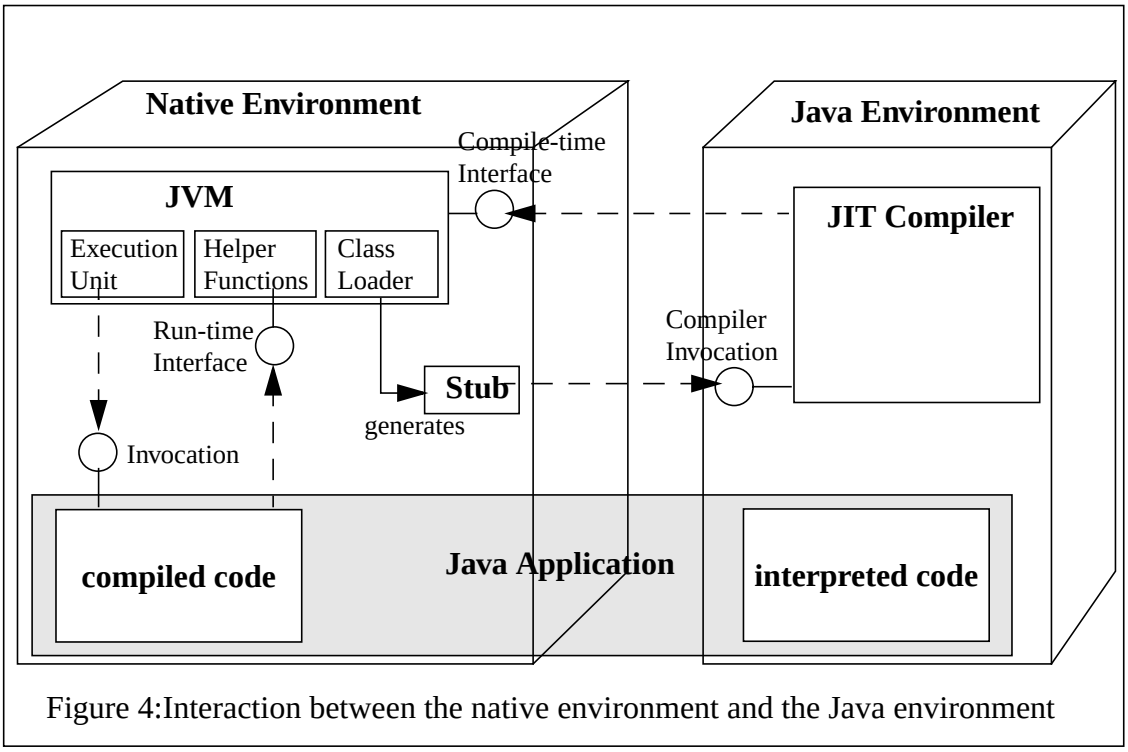


Figure 4:Interaction between the native environment and the Java environment

The algorithm that places the least burden to a JIT compiler is conservative garbage collection. In a conservative GC each value is considered a reference. There is no need for a JIT compiler to cooperate with a conservative GC.
An exact (or accurate) garbage collector needs to know all references that are hold by the JIT compiled code. At least two techniques can be used to provide this information. JIT compiled code can be treated like native code written in C. As in the JNI system [Sun97], each reference that is passed to the method as a return value of a method or as method parameter is registered. Using this technique, JIT compiled methods are treated like C-compiled native methods. Every method is passed a JNIEnv pointer to a thread specific data structure that is used to track references and so to support the garbage collector. This is not the fastest way but is has a simple implementation. In a more efficient technique the JIT compiler computes a map of reference locations for each code position where the thread can be preempted by the garbage collector [MS98], [Cram+97]. These code positions are called gc points [Diw+92]. To reduce the overhead of map creation and storage the garbage collector is only allowed to run at dedicated gc points, like method calls, and object allocations, or at the backward branch of a loop.

3.3 The compile-time interface

The compile-time interface (Figure 4) plays a central role in hiding JVM details from the compiler. The compile-time interface is the interface of the JVM that is used by the JIT compiler to get static information. To compile a method the compiler needs information about the class and the method that must be compiled. We extended the Reflection API allowing the compiler to obtain this information. The extended API provides the bytecodes of a method, the constantpool, the maximal depth of the operator stack, the number of local variables, and the exception tables. This extended Reflection API is only one part of the compile-time interface.
The other part of the compile-time interface gives information about the structure of the JVM. Using this interface the compiler can obtain

- method IDs
- the index of a method in the vtable
- class IDs
- the offset of a field in an object
- the address of static fields
- the address of String objects in the constant pool

This information can only be obtained after resolving the constant pool. However, this is allowed according to the JVM specification.

4 Compiling a method

In order to allow easy adoption of new compilation techniques we designed and implemented the JIT in a strictly object-oriented manner. The compiler is structured into a frontend and a backend. The frontend is mostly machine independent. The backend is used for native code generation and is therefore machine dependent.

4.1 Time of Compilation

The decision when the compiler is to be invoked is not hardwired into the VM. Most JIT compilers generate a method-table that contains compiler-invocation stubs when initially loading a class [Crel97]. This means that a method is compiled before its first use. Because compilation consumes time, one could use several heuristics to decide when to compile a method and when to interpret it [Gris98]. Our JIT compiler calls a class that decides whether a stub for compiler invocation or for interpreter invocation should be used. This class currently bases its decision only on the class name and the method name. However, it also could count method invocations and compile the method when the invocation count exceeds a certain threshold.

4.2 Method compilation

The compilation of a method proceeds as shown in Figure 5. The frontend first obtains the method's bytecode using an extended reflection API. An object is created for every bytecode of the method. Then the position of the bytecode's operands is computed and this way the stack oriented code is transformed into a two-address code. The next step is the actual code generation, where all bytecode objects are asked to translate themselves. They do the translation using the platform-dependent classes of the backend. JVM dependent bytecodes, such as invokevirtual, delegate the translation to the JVM dependent code. The CPU dependent part then does the register allocation and then the creation of machine code. If the machine code of the complete method is created, the jump offsets can be computed.

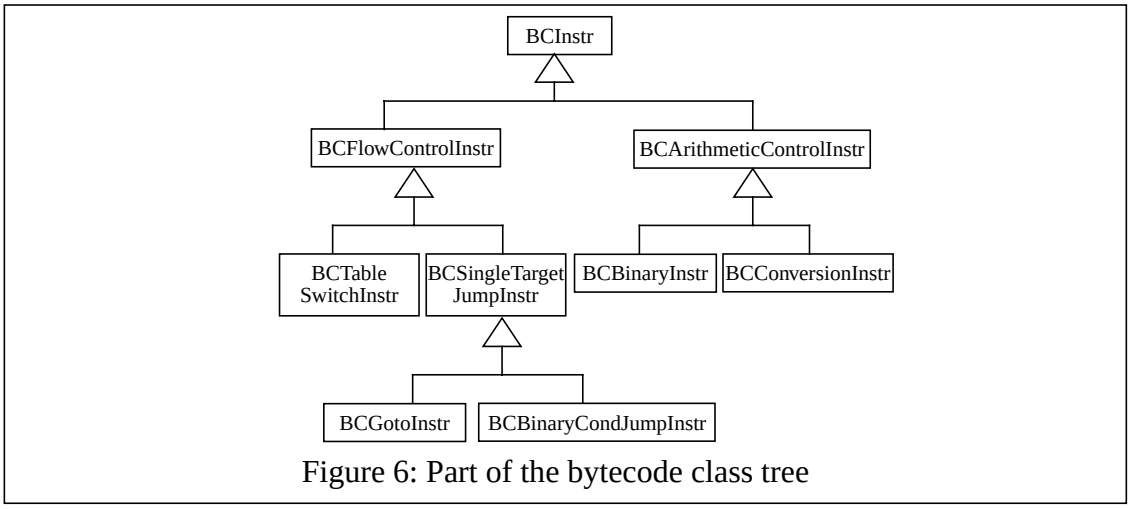


Figure 6: Part of the bytecode class tree

4.3 Bytecode objects

All bytecodes of a method are represented by objects. The classes of these bytecode objects are organized in a inheritance hierarchy as shown in Figure 6.
Jump instructions are a bit more complicated. At the target of the jump instruction a pseudo instruction JumpTarget is insert into the list of instruction objects. The Jump instruction object has a reference to this object. To create an instruction object, the prototype design pattern is used. An array of bytecode objects contains a prototype of the bytecode object at the index of the bytecode number. This object is cloned. During this process many new objects are cloned, which is a very expensive operation. We currently work on restructuring this process to reuse the bytecode objects in the compilation of the next method by employing the flyweight design pattern.

4.4 Using the bytecode objects

Using an object for each bytecode allows us to implement operations that can easily be applied to the whole method. A traverser object is responsible for traversing the bytecode. It determines the order, in which a certain operation is applied to the bytecodes. These operations could be

- setting the operands of the bytecode
- finding jump targets
- generation of native code
- copy propagation
- interpretation of the bytecodes
- eliminating unnecessary null-reference checks
- dead code elimination

4.4.1 Operand position

The JVM is a stack-oriented machine. This means that operands are passed at the stack. Most CPUs are register oriented, therefore the compiler should transform the stack-oriented code into an register oriented code. The first step is to find the stack positions of the operands. This position depends on the stack depth at this code position. And the stack depth depends on all bytecodes that are executed before. A property of the JVM is that the stack depth depends not on the code path on which a certain code position was reached [Gos95], [Age+98].

4.5 JVM dependent compilation

The JIT compiler must known certain details about the internal structure of the JVM when compiling code. The JVM provides an interface to obtain all the information that is necessary to compile the following operations that are dependent on the JVM structure (see Figure 2). This interface must be replaced when the compiler is ported to a new JVM. The compiler makes several assumptions about the

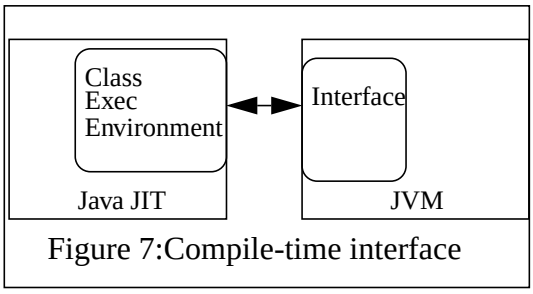


Figure 7:Compile-time interface

structure of the JVM. Most of these assumptions are encapsulated in the class ExecEnvironment and could be changed if the compiler is ported to a JVM where these assumptions do not hold.

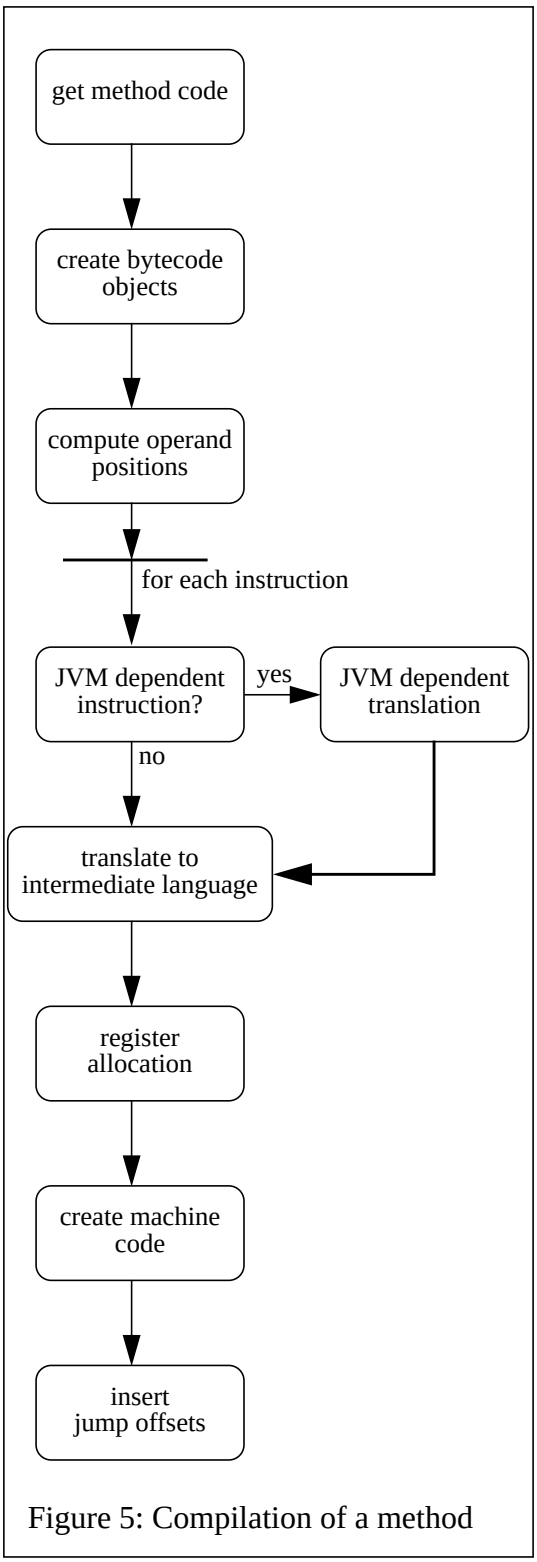


Figure 5: Compilation of a method

As shown in Figure 4 the compiler uses several helper functions that are accessed through a run-time interface. During JVM initialization several fields in the ExecEnvironment class are initialized with function pointers to these helper functions. The compiler generates a procedure call (a CALL instruction when compiling for the x86) using the function pointer as the target address. In the following we will explain which information is obtained from the ExecEnvironment class and for which purposes helper functions are used. The bytecodes that perform the discussed operations are listed in parenthesis.

Access to fields (getfield, putfield). To generate code for a field access the compiler must know the structure of the object store and the layout of an object. An object can be addressed directly or indirectly. Direct addressing means that an object reference contains a pointer to the object data. In this model, the first word of the object data is usually a pointer to the objects class. Indirect addressing means that an object reference contains a pointer to an object handle. This handle contains a pointer to the object data and a pointer to the class of the object. Using handles slows down field access but makes it easier to move objects during a garbage collection. The JVM provides the information which model of the object store is used and it also provides the information that is necessary to compile code for a field access. In case of a handle-based model the offset of the object pointer in the handle must be provided by the JVM.

Access to arrays (arraylength, aaload, aastore, ...). We assume that every JVM stores array elements sequentially and that the array length can be obtained using the array reference. The JVM gives an index of the length information in the array data and the offset of the array elements.

Invocation of virtual methods (invokevirtual). Virtual methods can be invoked by calling a helper function of the JVM or directly by looking into the method table. The JVM interface can be asked to give the offset of the method in the method table.

Invocation of static methods (invokestatic). The static method invocation can be compiled to a direct call to the method. The JVM interface gives the address of this method.

Interface invocation (invokeinterface). The compiler currently uses a helper function of the JVM to invoke an interface method. It would be faster if the compiler would generate code to directly invoke the interface method. To do this, the compiler must know the layout of the method tables for interfaces.

Type checking (instanceof, checkcast). The JVM provides helper functions to do runtime type checking.

Throwing exceptions (athrow). An exception must be thrown either because the athrow bytecode is executed or a null reference or an other run-time error occurred. The

native code then passes a reference to the exception object to a JVM helper function. This helper functions indicates, that an exception occurred in this thread.

Object creation (new). The JVM provides a helper function for object creation. This helper function gets a class ID and it returns an object reference.

Array creation (newarray, anewarray, multianewarray). Similar to object creation the JVM provides a helper function for array creation. This helper function gets a class ID for the array elements and the length of the array. It returns an array reference.

Monitors (monitorenter, monitorexit). There are helper functions that are used to acquire or release a monitor. These helper functions get the object reference that is associated with the monitor.

Certain run-time checks. (Certain tests that are done at runtime are also dependent on the JVM. These tests include the null-reference check, test for array bounds, integer division by zero, and checkcast. A JVM can use the MMU hardware to detect null-pointer accesses. If such a JVM is used, no null-reference checks are generated by the compiler.

There are some other parts of the compiler that are dependent on the JVM:

- calling convention of the compiled method
- stack layout of the compiled method
- calling convention of methods and helper functions
- exception handling

64-bit arithmetic (ldiv, lrem, lcmp, lshr, lshl, lushr). The compiler does not generate code for arithmetic operations with 64-bit integers. Istead it calls helper functions of the JVM interface to perform operations.

4.5.1 Stack Management

To be portable to different JVMs the compiler must be able to cope with different layouts of stack frames. Stack frames are typically used to store parameters, local variables, temporary variables, and the return address of the method. Our compiler encapsulates all code generation that depends on the stack layout in a separate class.

4.6 Installation of compiled code

If a method is loaded that has to be compiled, the JVM generates a stub to invoke the compiler and installs this stub as the method code by writing a pointer to this stub in the method table. After the compiler compiled the method, the pointer to the stub is replaced by a pointer to the compiled code.

5 Optimizations

5.1 Adaptive optimization

As shown in [Ro+96] different Java programs would profit from different optimization strategies. Simple and fast optimizations that are supported by most JIT compilers are register allocation, delayed code generation using a virtual stack, elimination of redundant array range checks, and elimination of redundant null pointer checks. Some applications would profit from other optimizations, so an open JIT architecture, where applications can guide the optimizer, is useful [Mats98].

5.2 Inlining

Our compiler supports inlining of static methods. Inlining methods results in speedup, because method call overhead is eliminated and inter-method optimizations can be applied. These optimizations could perform a better register allocation and elimination of unnecessary null-reference tests.
All methods that are private, static, or final can be inlined. The principle of inlining is very simple. The code of the method call (e.g. invokestatic) is replaced by the method code. The stack frame of the inlined method must be merged into the stack frame of the caller method. The following example illustrates this merging. If a method test() contains the method call a = addInt(a, 2), this call is compiled to

```
aload 0
iload 1
iconst_2
invokenonvirtual addInt(II)I
istore 1
```

The method int addInt(int x, int y) { return x+y; } is compiled to

```
iload 1
iload 2
iadd
ret
```

The merged stack frame has the following layout:

merged stack	stack of test()	stack of addInt()
0	local 0	
1	local 1	
2	operand 0	local 0

merged stack	stack of test()	stack of addInt()
3	operand 1	local 1
4	operand 2	local 2
5		operand 0
6		operand 1

The created code would look as follows (bytecodes have stack positions as operands):

```
aload 0 -> 2
iload 1 -> 3
iconst #2 -> 4
checkreference 2
iload 3 -> 5
iload 4-> 6          addInt()
iadd 5, 6 -> 5
istore 5 -> 2
istore 2 -> 1
```

Several optimizations can now be applied to the code. Provided that the method test only contains the addInt() method call, the code can be transformed using copy propagation and dead code elimination [Aho+85] to

```
checkreference 0
iadd 1, #2 -> 5
istore 5 -> 2
istore 5 -> 1
```

Virtual methods can be inlined, if no subclasses override the method. If the JVM loads a subclass that overrides the method, the inlining must be undone. This code modification must be done very carefully, because a different thread could be executing this code.

5.3 Storing compiled code

Storing the JIT compiled code means that the compilation must not be done just-in-time. If a class is loaded in a separate run of the VM, the stored native code can be used as long as the class has not changed. One could also construct an offline compiler that compiles a whole application and stores the compiled code. All data that depends on the current instantiation of the JVM must be set during a linking phase when the stored code is loaded into the JVM. This must be done for the following data:

- addresses of helper functions
- addresses of other compiled non-virtual methods
- addresses of static fields
- IDs of classes and methods
- offsets of methods in the method table
- offsets of fields in the object

To set this information when the code is loaded into the JVM we store a symbol table with every compiled method. This symbol table contains the offset of the constant within the compiled code, the type of the constant, and information to compute the constant.

5.4 Low-level optimizations

Our compiler uses several optimizations that are “standard” for JIT compilers, like register allocation, virtual stacks, copy propagation, dead code elimination, elimination of redundant null-reference checks, and elimination of redundant array range checks.

6 Performance

We did not carry out detailed performance measurements yet. Our compiler uses optimizations that are used be other JIT compiler, so we expect the generated code to be of a comparable quality.
The performance of the compiler itself is worse than the performance of other JIT compilers, because we did not compile the compiler with itself. It currently compiles about 3000 bytecodes per second on a 100 MHz Pentium processor. Compiling the compiler would give a 5 to 10 times speedup, so the compiler would be comparable to JIT compilers written in C, which compile about 30000 bytecodes per second.

7 Related Work

There exist a number of JIT compilers that are written in C (JBuilder [Crel97], Sun [Cram+97], Cacao [Gra97], Microsoft [MS98]). There are also some compilers that operate offline by compiling the bytecode to C code and then translating this C code ([Proe+97], [Mul+97]). We know only one other JIT compiler that is written in Java [Mats98]. This compiler was derived from an existing C-written JIT compiler and it is not designed to be JVM independent.

8 Status and Future Work

We extended our own JVM, which is described in [Gol97] for the interfaces described in the above sections. We did not used the compiler with another JVM, although we intend to do this. To use the compiler with another JVM, the target

JVM must provide the compile-time and run-time interfaces described in this paper. A detailed description of the compiler is contained in [Ko98].

9 References

Age+98 O. Agesen, D. Detlefs, J. Moss, Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines, Proc. Conf. on Programming Languages Design and Implementation, PLDI '98

Aho+85 V. A. Aho, R. Sethi, D. J. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison Wesley, 1985

Cram+97 T. Cramer et al., Compiling Java Just-in-time. *IEEE Micro*, May/June 1997

Crel97 R Crelier, Interview about the Borland JBuilder JIT Compiler.

Diw+92 A. Diwan, J. Moss, R. L. Hudson, Compiler support for garbage collection in a statically typed language, Proc. Conf. on Programming Languages Design and Implementation, PLDI '92, San Francisco 1992

Fras96 C. W. Fraser, D. A. Hanson, *A retargetable C compiler*, Redwood City, Calif. Benjamin/Cummings 1995

Tie89 M. D. Tiemann. The GNU Instruction Scheduler,, June 1989

Gol97 M. Golm, *Design and Implementation of a Meta Architecture for Java*. Diplomarbeit (masters thesis), Universität Erlangen, IMMD 4, Jan. 1997.

Gos95 J. Gosling, Java Intermediate Bytecodes, Workshop on Intermediate Representations, IR 95, ACM Sigplan Notices Vol. 30, No. 3, March 1995

Gra97 R. Grafl, *Cacao, Ein 64-Bit-JavaVM-Just-In-Time-Compiler*, Diplomarbeit (masters thesis), Technische Universität Wien, Institur für Computersprachen, 1997.

Gris98 D. Griswold, *The Java HotSpot Virtual Machine Architecture*, A White Paper About Sun's Second Generation JavaTM Virtual Machine,, March 1998

Hsi+97 A. Hsieh et al., Optimizing NET Compilers for Improved Java Performance, *IEEE Computer* Vol. 30, Iss. 6, June 1997

JoLi96 R. Jones, R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, John Wiley & Sons, New York, 1996.

Ko98 H. Kopp, Design und Implementierung eines maschinenunabhängigen Just-in-Time-Compilers für Java, Diplomarbeit (masters thesis), Universität Erlangen, IMMD 4, Oct. 1998

Mats98 Matsuoka, OpenJIT - A Reflective Java JIT Compiler, *OOPSLA 98 Workshop on Reflective Programming in C++ and Java*, UTCCP Report 98-4, Center for Computational Physics, University of Tsukuba, Japan, Oct. 1998.

MS98 Microsoft Web Page. http://www.microsoft.com/java/sdk/20/vm/Jit_Compiler.htm

Mul+97 G. Muller, B. Moura, F. Bellard, C. Consel, Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code, *The Third Usenix Conference on Object-Oriented Technologies and Systems*, Oregon 1997

Proe+97 T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, S. A. Watterson, Toba: Java for Applications - A Way Ahead of Time (WAT) Compiler. *The Third Usenix Conference on Object-Oriented Technologies and Systems*, Oregon 1997.

Ro+96 T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad H. M. Levy. The Structure and Performance of Interpreters, *ASPLOS VII*, October 1996

Sun97 Java Native Interface Specification, Sun Microsystems, May 1997

Wil+97 T. Wilkinson et al., *Kaffe Homepage*, www.kaffee.org

YeLi97 F. Yellin, T. Lindholm, *Java Runtime Internals*, Presented at JavaOne 1997, available at http://java.sun.com/javaone/sessions/slides/TT27/index.html