FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Invokedynamic for the CACAO JVM

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Fabian Gruber
Matrikelnummer 0726905

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr. Andreas Krall

Wien, 24.07.2014

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Invokedynamic for the CACAO JVM

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Fabian Gruber
Registration Number 0726905

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr. Andreas Krall

Vienna, 24.07.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Fabian Gruber
Ahornweg 11, 4910 Ried im Innkreis

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

I want to thank Prof. Andreas Krall for giving me the opportunity to work on the CACAO VM and for supporting my work. Furthermore, I want to the thank the entire CACAO team, especially Stefan Ring, for their ideas, constructive criticism and finding lots of bugs introduced by me.

Thanks to Katharina Weinhäupl for critical reading and moral support.

Finally, I want to thank my family for supporting me throughout my studies.

Abstract

The Java Specification Request 292 (JSR-292)[51] adds features to the Java Virtual Machine (JVM) specification[34] that facilitate high performance implementations of dynamically typed languages, most notable the new `invokedynamic` bytecode.

The aim of this thesis is to implement all new features of JSR-292 in the CACAO Java Virtual Machine[30],[31]. Implementing the `invokedynamic` instruction requires several changes to the JVM, which interact with various subsystems such as the class file loader, the bytecode verifier, the JIT code generator and even the garbage collector.

The OpenJDK class library contains a runtime bytecode generator[57] which implements parts of JSR-292. Unfortunately, the generator relies on a number of internal APIs of the HotSpot JVM and non standard extensions to the bytecode instruction set and a large part of this work was dedicated to porting these to CACAO. Most notable among these is that the generator assumes that its host VM can handle changes to a methods stack behavior during execution, which HotSpot handles by falling back to interpreted mode. With CACAOs JIT only approach to executing bytecode this is currently not feasible, as a workaround OpenJDKs generator is reverse engineered and the dynamic behaviour of the bytecode is anticipated statically where necessary.

Finally, the JSR-292 implementation of CACAO is compared to that of HotSpot and JamVM, two JVMs that also use the same bytecode generator framework, in order to gain insights on how low level implementation decisions impact performance.

Kurzfassung

Der Java Specification Request 292 (JSR-292)[51] erweitert die Java Virtual Machine (JVM) Spezifikation[34] um die effiziente Implementierung von dynamisch typisierten Programmiersprachen zu ermöglichen. Der `invokedynamic` Bytecode ist die wichtigste Neuerung in diesen Zusammenhang.

Das Ziel dieser Arbeit ist es alle neuen Features von JSR-292 in der CACAO JVM[30],[31] zu implementieren. Die Umsetzung der neuen Instruktionen und Schnittstellen hat Auswirkungen auf eine Vielzahl von Modulen der virtuellen Maschine, darunter den Classloader, den Bytecode-Verifizierer, den Just-In-Time (JIT) Codegenerator und den Garbage-Collector.

Die OpenJDK-Klassenbibliothek enthält einen Laufzeit-Bytecode Generator[57], welcher Teile von JSR-292 implementiert. Unglücklicherweise verwendet dieser Generator mehrere interne Schnittstellen der HotSpot JVM und Erweiterungen des Bytecode Befehlssatzes. Ein großer Teil dieser Arbeit war ihrer Reimplementierung in CACAO gewidmet. Das größte Problem stellte die Annahme des Generators dar, dass die virtuelle Maschine Bytecode unterstützt, der sein Stackverhalten zur Laufzeit verändert. Die HotSpot JVM verwendet in solchen Fällen einen Bytecode-Interpreter, für den Änderungen des Operandenstack kein großes Problem darstellt. Da CACAO sich zur Ausführung von Bytecode vollständig auf einen JIT-Übersetzer verlässt, ist dieser Ansatz derzeit nicht möglich, es war daher notwendig OpenJDKs Generator Framework zu analysieren um das dynamische Verhalten des generierten Bytecode vorhersagen zu können.

Abschliessend vergleichen wir die Implementierung von JSR-292 für CACAO mit der von HotSpot und JamVM, zwei weiterer JVMs die den selben Bytecode-Generator verwenden, um Erkenntnisse darüber zu gewinnen wie sich low-level Implementierungsentscheidungen auf die Effizienz auswirken.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 The Java Virtual Machine

The Java programming language is a high level, object oriented programming languages designed in the nineties at SUN Microsystems. Contrary to other languages such as C or C++ Java programs are usually not compiled to machine code intended for direct execution by a CPU. Instead they are translated to an intermediate program representation, so called bytecode[1], that is then executed by a virtual machine, the JVM.

The JVM is a stack machine, that means that the operands of an instruction are taken from and its result is stored on an operand stack. Contrary to other stack based systems, like Forth[38], each activation record in the JVM has a private operand stack and each method call can only return only one value. Furthermore, the JVM is not a pure stack machine, in addition to the operand stack every method has access to an array of local variables which is also used for parameter passing in a method call.

Since the JVM was specifically designed to execute Java programs many of the language features are directly mirrored in the JVMs instruction format. For example, bytecode is strongly typed and its type systems mirrors that of the Java language. It distinguishes between a fixed set of primitive types, consisting of several types of integers and floating point numbers, and object references. The bytecode type system is also aware of the fields and methods of object types, and ensures that they are accessed and called correctly.

Despite it's tight coupling to the Java language the JVM has become a compilation target for many different languages such as Ruby[40], JavaScript[39],[46], Scala[41], Clojure[25] and many more[3],[59]. This is in part due to bytecode being much more high level than regular machine code but also because there are high quality implementations of the JVM for most computer architectures and operating systems. Consequently any language that compiles down to bytecode is automatically portable to a wide variety of platforms.

.

[1] Bytecode derives it's name from the fact that most instructions fit in one byte.

1.2 Invokedynamic

1.2.1 Method calls on the JVM

The further a languages semantics differ from those of Java, the harder it gets to implement it efficiently on top of the JVM. Core features of modern dynamic languages, like methods as first class objects, the ability to change and extend classes at runtime or duck typing, are not directly supported by bytecode. When implementing these features with bytecode a lot of effort has to be expended on bridging this semantic mismatch[49]. This is most apparent in the code for method invocation and field access for dynamic languages.

There are four instructions in the JVM for method invocation which correspond directly to the four kinds of methods in the Java language. Namely virtual methods (`invokevirtual`), static methods (`invokestatic`), methods on interfaces (`invokeinterface`) and non-virtual instance methods (`invokespecial`). On the JVM all call sites are linked lazily the first time they are executed. Furthermore calls made via `invokevirtual` and `invokeinterface` are bound late, meaning the actual method to execute is looked up in the first argument of the call, the receiver. Even though these instructions are able to express a wide spectrum of object oriented behaviour the linking and dispatch semantics are fixed, other ways of method invocation that differ from Java's must be simulated, which can lead to a serious performance penalty. Furthermore, all these instructions have in common that methods are statically referenced by name and type, meaning there is no support for anonymous functions and that even for virtual calls the type and number of arguments of the executed method must match that of the call site.

1.2.2 Method calls in JRuby

Rose lists several problems Java's model of method calls present for dynamically typed programming languages[49]. We will use JRuby, an implementation of the Ruby programming language for the JVM[40], as an example to elucidate them.

Methods in Ruby are regular objects that can be passed around and manipulated. In Java bytecode on the other hand, methods are not values and can neither be stored on the operand stack nor in local variables, they are only available as constants that the invoke instructions reference. Thus JRuby wraps references to methods in so called method simulator objects, which are represented by the class `DynamicMethod`. This class has an abstract method `call` and for every Ruby method in a program there is a subclass of `DynamicMethod` whose implementation of `call` contains the code for that Ruby method. Since `call` simulates methods of different arities its arguments must be wrapped in an array at every call site and the actual implementation must then unwrap the argument array. Furthermore all arguments of a non object type, such as `int` or `double`, must be wrapped in an object so that they can be stored in the argument array. This constant allocation of course puts high pressure on the VMs garbage collector.

To alleviate the allocation overhead `DynamicMethod` provides overloads for `call` with zero to three arguments, any method with that number of arguments can then be invoked without creating an array. There are no overloads for primitive arguments though, since that would soon lead to a combinatory explosion in the number of overloads.

The problem with this representation of Ruby methods is that it prevents many optimizations, most importantly devirtualization. On many JVMs a monomorphic call, that is, a virtual call to a method with only one implementation, can avoid the overhead of late binding and directly jump to the target method. Knowing that there is only one, or a small finite number, of possible targets for a call also helps other optimizations, such as inlining. But even when a given Ruby method is monomorphic all calls to it are funneled through the `DynamicMethod.call` interface, which is not.

Efficiently finding the right `DynamicMethod` object to invoke also presents several problems. Note that in Ruby methods, and also fields, are looked up in a hash table by name. This allows for a more flexible, but also more expensive, form of dynamic dispatch than that supported natively by the JVM via `invokevirtual` and `invokeinterface`, where the lookup is usually implemented as an array access with a constant index. In practice JRuby speeds up calls by caching the result of such method lookups in an `org.jruby.runtime.CallSite` object associated with every method call site. These `CallSite` objects are effectively used to simulate inline caching[26] at the bytecode level. Unfortunately the JVM specification does not allow altering a methods code once it has been loaded, even though some VMs support this. As a consequence the inline cache can not lazily patch in new code in case of a cache miss, instead every call site requires code to check and if necessary invalidate the cache.

A further complication of this scheme is that each method object requires at least one class to hold its bytecode, once a method is specialized or inlined creation of further classes can become necessary. Once a method class is no longer used a JVM implementation is allowed to unload it and free the resources associated with it. Unfortunately every Java class loader holds a strong reference to each class it has loaded. Thus to allow method classes to be garbage collected individually each such class has be loaded by its own class loader. The problem with this is that we now have to create at least three objects, the method itself, a class and a class loader, for each method in the Ruby code.

1.2.3 Method calls with invokedynamic

With the DaVinci or Multi Language Virtual Machine[10] Sun Microsystems started an effort to make the JVM more viable as execution platform for other programming languages, with a focus on dynamic languages. At the core of this effort lies the new `invokedynamic` bytecode and the so called `MethodHandle` and `CallSite` classes. `invokedynamic` is a new instruction for invoking methods that allows a program to take control of the linking mechanism. A so called bootstrap method, which is just a regular Java method, is run to determine the actual method to link to a given call site. `MethodHandle` and `CallSite` correspond roughly to the `DynamicMethod` and `CallSite` types of JRuby. An advantage of the new APIs is that the JVM is aware of them, it can take adavantage of the information reified in those objects and they do not pose an optimization barrier as large as JRuby's equivalents.

A comparison of the bytecode for method calls in JRuby with and without JSR-292 can be seen in figure 1.1. In the first bytecode listing, which does not use JSR 292, JRubies implementation of inline caching can be seen. In the second snippet, which utilizes the new `invokedynamic` instruction, the `CallSite` mechanism is not exposed to the bytecode, giving the JVM much more leeway in how it implements this call.

```
public abstract class DynamicMethod {
 IRubyObject call(IRubyObject self, String name, IRubyObject arg1);
}
public class CallSite {
 private String name;

 DynamicMethod getCache() {...}

 IRubyObject call(IRubyObject self, IRubyObject arg1) {
  return getCache().call(self, name, arg1);
 }
}


# Ruby code
a = 5
b = 3
c = a + b


// Bytecode without JSR-292
invokevirtual Module.getCallSite0; // load CallSite object
aload 1                            // load variable a
aload 2                            // load variable b
invokevirtual CallSite.call        // invoke call site object,
                                   // forwards to DynamicMethod.call


// Bytecode with JSR-292
aload 1                 // load variable a
aload 2                 // load variable b
invokedynamic "call:+"  // directly invoke method
```

Figure 1.1: Method call with and without JSR-292 in JRuby (abridged)

Work on implementing these new APIs in the JVM was started with JSR 292[51]. The similarity between the two APIs is not coincidental since key members of the JRuby team where also involved in the JSR and both APIs evolved alongside each other. After several iterations and design changes JSR 292 was added to the JVM Specification Version 7[34] in July 2012.

In March 2014 Oracle released JDK 8 which contains among other changes Nashorn[46], a new implementation of the JavaScript scripting language running on top of the JVM to replace the older Rhino JavaScript VM[39]. Nashorn makes extensive use of method handles and the new invokedynamic instruction. Marcus Lagergren of the Nashorn team states that on average every tenth bytecode emitted by Nashorn is an `invokedynamic` instruction[32].

1.2.4 Other uses of JSR 292

Besides building the basis of implementations for dynamic languages on the JVM, JSR-292 has been used for a number of different projects.

The compilation of lambda expressions in Java 8 for example uses invokedynamic and the

bootstrapping mechanismn to create custom factory functions for the closure object backing a given lambda expression[23]. This compilation scheme was chosen since it allowed to hide the actual implementation of closure generation behind the bootstrap method. Consequently the mechanism used by the VM to create closure objects can be changed without breaking binary compatibility with already existing class files[23].

JooFlux is a JVM agent that transforms bytecode as it is loaded in order to allow methods to be replaced at runtime. The agent rewrites the traditional method call instructions to invokedynamic instructions and also provides a JMX interface that allows the user to alter the code for individual methods of a running application[45].

1.3 CACAO

The CACAO virtual machine[30],[31] is a JVM from the Institut für Computersprachen der Technischen Universiät Wien, first developed in 1996 for the 64-bit Alpha architecture. It has since then been ported to many operating systems and CPU architectures, including Linux and MacOS X on ARM, Intels x86 architecture[56] (32 and 64-bit) and PowerPC[33] (32 and 64-bit).

CACAO uses a JIT compiler for executing Java programs, that means that bytecode is translated to machine code at runtime just as it is about to be executed. Even though a bytecode interpreter has been developed for CACAO[24],[16] the standard configuration of the VM solely relies on the JIT compiler. To keep startup times low the JIT focuses on being fast instead of spending a large amount of time on optimizing the produced code.

Traditionally CACAO has used GNU Classpath[22] as its class library but it can also be run with the OpenJDK[42] class library instead.

Currently CACAO supports version 6 of the JVM specification, work on supporting version 7 has started but the new features of JSR 292 have not yet been incorporated. The aim of this thesis is to implement all these new features in CACAO. This requires several changes to the JVM, which interact with various subsystems such as the class file loader and bytecode parser, the bytecode verifier, the JIT code generator, the code patcher and even the garbage collector.

1.4 Structure

Chapter 2 explains relevant details of the JSR-292 specification for invokedynamic. Chapter 3 presents other JVMs and their implementation of invokedynamic. A focus is put on the implementation in the HotSpot JVM since it serves as a basis for CACAOs version. Chapter 4 contains a short overview of the architecture of the CACAO JVM and then presents details on how HotSpots code was adapted for CACAO. Chapter 5 shows the results of standards conformance and performance testing. Chapter 6 discusses the methods and results of the thesis and points out possible future work.

CHAPTER 2

JSR-292

JSR 292 adds three new concepts to the JVM: method handles, which allow code to be passed around and treated as a first class entity, method types which describe the runtime type of a method handle and the invokedynamic instruction which allow for custom linking and dispatch semantics for method calls which differ from those of the Java language.

Generally speaking the new features introduced by JSR 292 are designed to be as non invasive as possible and change only very few already existing aspects of the JVM. In the spirit of backwards compatibility the semantics of no already existing Java program, compiled to bytecode or in source form is altered.

2.1 Method handles

The most important and complex new concept introduced by JSR 292 is that of a method handle. Contrary to its name a method handle is not just a pointer to a Java method but an executable reference to a low level VM operation such as a method invocation, object construction or a field access[34].

No new kinds of types for function objects are introduced for method handles and there are no new bytecodes to operate on them. From the perspective of Java bytecode a method handle is just a regular object of class `MethodHandle`[11]. At first glance the API presented by `MethodHandle` is very similar to that of `Method`[13], an older facility for treating methods as objects. Both classes have no accessible fields or constructors, and, as can be seen in figure 2.1, similar methods for invoking the underlying operation.

Even though all these invoker methods can take any argument on the Java source level, the way they are compiled is very different. `Method.invoke` is a regular Java method and every time it is invoked the VM has to allocate an array to store the variable number of arguments, which also includes a heap allocation for boxing any parameter of a primitive type. `MethodHandle.invoke` and `MethodHandle.invokeExact` are compiled as if there were an overload for the given types of parameters, meaning there is no boxing overhead whatsoever.

```
public abstract class MethodHandle {
  @PolymorphicSignature
  public final native Object invokeExact(Object... args) throws Throwable;

  @PolymorphicSignature
  public final native Object invoke(Object... args) throws Throwable;
}

public final class Method {
  public Object invoke(Object obj, Object... args) throws ...;
}
```

Figure 2.1: API of `MethodHandle` and `Method` (abridged)[42]

Invocation of a method handle via `invokeExact` are from now on referred to as exact invocation and calling it via `invoke` as generic invocation[1].

The bytecode generated by a simple use of the two APIs is illustrated in figures 2.2 and 2.3 [2]. Observe the different type signatures for the calls to `Method.invoke` and `MethodHandle.invoke`, the signature of `Method.invoke` shows that all parameters have been boxed in an array while the invocation of the method handle is compiled like a regular method call. `Method.invoke` also differs in taking the receiver argument separately from all other arguments, but `MethodHandle.invoke` treats it the same as all other arguments. This stems from the fact that a `Method` object is always a handle to a method while a method handle can point to a computation that does not even have a notion of a receiver. Also note that the reflection API throws away all static type information and can only rely on the dynamic types of the passed arguments, while `MethodHandle.invoke` has both static and dynamic types available.

For any other method than `MethodHandle.invoke` and `MethodHandle.invokeExact` this compilation scheme would simply lead to a runtime error the first time the code is executed since the JVM would not be able to find a corresponding overload for the given argument types. However, JSR 292 extends the behaviour of the `invokevirtual` instruction exactly for these two methods. They can be invoked with any type signature and linking and type checking of such a call will always succeed, provided the receiver is a `MethodHandle` object. This special behaviour unique to these two invoker methods of `MethodHandle` is referred to as *signature polymorphism*.

The JSR 292 specification states that neither invoke nor invokeExact can be called via Java's reflection API. While it is possible to obtain a Method object for them with the generic signature that takes an object array and returns an object, and not any other, trying to invoke that Method object must throw a runtime exception.

[1] In an earlier version of the JSR 292 specificiation `invoke` was actually called `invokeGeneric` and the internal source code of HotSpots implementation still refers to it as such.

[2] All bytecode listings are produced with the `javap` tool distributed with OpenJDK and, where necessary, were shortened to fit the page

```
Method m    = ...
Foo    self = ...
int    i    = m.invoke(self, 2);


aload_0; // load Method object
aload_1; // load receiver
// box arguments in an array
iconst_1;
anewarray "java/lang/Object";
dup;
// box int argument in an object
iconst_0;
iconst_2;
invokestatic "Integer.valueOf:(I)LInteger;";
aastore;
// invoke underlying method
invokevirtual "Method.invoke:(LObject;[LObject;)LObject;";
// cast returned object back to an int
checkcast     "Integer";
invokevirtual "Integer.intValue:()I";
```

Figure 2.2: Bytecode generated for `Method.invoke`

```
Method m    = ...
Foo    self = ...
int    i    = m.invoke(self, 2);


aload_0;   // load MethodHandle object
aload_1;   // load receiver
iconst_2;  // load int parameter
// invoke underlying method
invokevirtual "MethodHandle.invoke:(LFoo;I)I";
```

Figure 2.3: Bytecode generated for `MethodHandle.invoke`

2.1.1 Method types

Just as any piece of bytecode the underlying operation represented by a method handle expects a certain number of arguments on the stack and requires those to conform to some type. However, since, contrary to any other bytecode, an invocation of a method handle is not statically checked to have a valid type this check must be done at runtime.

Just as a method handle makes the concept of code a first class object the class `MethodType`[12] reifies the notion of a bytecode type signature and allows it to be inspected and manipulated at runtime. The individual parameter types and the return type of a method type are represented by Class objects, allowing for introspection via the regular reflection framework of the JVM.

Each method handle has a fixed method type and each time it is invoked via the `invokeExact` method that method type is compared against the signature of the call site, if the two types do

not match exactly an exception is thrown and no computation is performed.

The `invoke` method is more lenient, and it tries to apply the same type conversions to arguments as the `javac` compiler would, but at runtime. Calling, for example, `invoke` on a handle that expects a `long` argument with an `int` argument simply means that the `int` is widened to a `long`, while `invokeExact` would throw a runtime error. `invoke` also boxes and unboxes primitive arguments and casts reference parameters as needed by the handles type and only if one of these conversions fails, the call is aborted.

2.1.2 Creating method handles and types

The classes `MethodHandle` and `MethodType` have no accessible constructors and cannot be directly instantiated by user code. Instead there is a specialized API, once more similar to the older reflection API, for obtaining instances of these classes. This API allows the creation of handles that directly reference a method, constructor or field. Such method handles, and their types, can also be directly stored in the constant pool of a class, just as string or primitive constants.

For each such direct method handle there is a sequence of bytecodes that perform the exact same operation as that handle would when called, that sequence is called the handles bytecode behaviour. The kinds of handles that can be created this way are listed in table 2.1, along with their corresponding bytecode behaviour and type signature.

Note that it is not possible to obtain a handle that directly invokes a constructor, a special `<init>` method, since that would allow bypassing safety constraints of Java bytecode. Before any other operation can be performed on a freshly allocated object it must first be initialized by calling a constructor on it, guaranteeing that objects are always initalized properly. The reverse also holds, i.e., the receiver of a constructor invocation can only be an uninitialized object. Thus a handle to a constructor, as identified by kind `REF_newInvokeSpecial` (see table 2.1), calls both `new` and `<init>` before returning the created object.

Besides checking if a bytecode operation, for example a field access, has the correct type the JVM also validates that the class containing the bytecode is actually allowed to access that field. Usually these checks are performed statically and do not incur any performance penalty at runtime, when using the reflection API such checks are performed on every access meaning the safety constraints are upheld. Even if a class inadvertently leaks a Method object for a private method trying to use that reflective object will result in an exception. Such runtime access checks are relatively expensive since they have to scan the call stack to see which class the code was called from. Method handles avoid this overhead by only performing such checks when they are created. As a consequence they can pose a security risk when passed to untrusted code.

2.1.3 MethodHandle combinators

Direct method handles alone suffice to implement dynamic direct calls to Java methods, but more advanced use cases such as handling methods with a different calling convention or inline caches still would require the user to generate adapter bytecode.

To enable this JSR 292 introduces method handle combinators, which are method handles that alter the behaviour of other method handles, for example by dropping, converting or inserting arguments of call.

Kind	Bytecode behaviour	Method descriptor
REF_getField	getfield C.f:T	(C)T
REF_getStatic	getstatic C.f:T	()T
REF_putField	putfield C.f:T	(C,T)V
REF_putStatic	putstatic C.f:T	(T)V
REF_invokeVirtual	invokevirtual C.m:(A*)T	(C,A*)T
REF_invokeStatic	invokestatic C.m:(A*)T	(A*)T
REF_invokeSpecial	invokespecial C.m:(A*)T	(C,A*)T
REF_newInvokeSpecial	new C; dup; invokespecial C.<init>:(A*)void	(A*)C
REF_invokeInterface	invokeinterface C.m:(A*)T	(C,A*)T

The kinds of MethodHandles that may appear in the constant pool of a class. C is a Java class, f a field, m a method, A* are the types of the arguments of m and T is either the type of f or the return type of m[34].

Table 2.1: MethodHandle kinds

Contrary to direct method handles combinators cannot be represented in the class file format and must be constructed at runtime by calling factory methods of class `MethodHandles`. Besides combinators that API can also create method handles that read or write from an array or return a constant value.

Combinators themselves can of course also be used as arguments for other combinators, making it possible to build more complex method handles out of the simpler ones predefined by JSR 292. Combinators and direct method handles together present a tree shaped representation of code, opposed to the sequential one of bytecode.

For an example use case of method handle combinators imagine a dynamically typed progamming language in which functions, represented as method handles, can be passed around as first class values. Furthermore, every function expects, to avoid the use of thread local storage, some thread state object as first argument. Handles to regular Java methods could then not be used interchangibly with functions of the language since the Java method does not expect the additional thread state argument.

In order to allow seamless interoperation each handle to a Java method is wrapped in the `dropArguments` combinator, which discards the first argument of a call before forwarding the rest to another handle. When the handle to this Java method is called from the other language the unwanted thread state argument is now automatically dropped and the target method gets exactly the parameters it expected.

Another use case of combinators would be to provide an easy to use `println` function for printing to `stdout` in this imanginary language. The standard way of achieving this in Java is to use the famous global variable `System.out`, which contains a `PrintStream` object whose methods write directly to `stdout`. In order to not force users of the language to always pass this global variable as an argument to every call to the `println` function we use the `bindTo`

```
// obtain a handle to the println method in PrintStream
MethodHandle raw_println = lookup().findVirtual(
  PrintStream.class,
  "println",
  methodType(PrintStream.class, String.class));

// always use System.out as first argument to println
MethodHandle System_out_println = raw_println.bindTo(System.out);

// discard one leading argument (thread state)
MethodHandle println = dropArguments(
  System_out_println,
  1,
  ThreadState.class);

// print to stdout
println.invoke(thread_state, "Hello, World!");
```

Figure 2.4: An example usage of method handle combinators

combinator. `bindTo` can be seen as the opposite of the `dropArguments` combinator, i.e. it inserts an additional paramater into the call to a handle.

In this example we would first obtain a handle referencing the `println` method of `PrintStream`, and then we bind that handle to always use `System.out` as the target stream for printing. The result is a handle that takes only a `String` as argument and when invoked prints that string to `System.out`. The API calls to create the `println` function can be seen in figure 2.4.

As a side effect virtual calls through a bound handle always use the same receiver class, allowing the VM to eliminate the overhead of the virtual dispatch.

An example for a combinator that depends on more than one other handle is `filterArguments`. It transforms individual arguments of its call with with other method handles. An example for this would be a function that concatenates two strings, but before concatenation the first argument is converted to uppercase and the second one to lower case. The API calls and combinator tree for such a function is shown in figure 2.5.

2.2 The invokedynamic instruction

The semantics of the `invokedynamic` instruction is largely defined via that of method handles. The first time an `invokedynamic` is executed a linking process similar to that for other invoke instructions is initiated. The difference for the new instruction is that this linking does not follow fixed semantics but is achieved by calling a user defined bootstrap method.

Each invokedynamic instruction in a class is associated with a call site descriptor that contains a reference to the bootstrap method, a name and type descriptor for the method to resolve as well as a list of additional arguments to be passed to the boostrap method.

Fully resolving the invokedynamic instruction then requires three steps:

```
// obtain a handle to the concat method in String
MethodHandle concat = lookup().findVirtual(
  String.class,
  "concat",
  methodType(String.class, String.class));

MethodHandle upper = lookup().findVirtual(
  String.class,
  "toUpperCase",
  methodType(String.class));

MethodHandle lower = lookup().findVirtual(
  String.class,
  "toLowerCase",
  methodType(String.class));

// filter arguments of concat
MethodHandle mh = filterArguments(concat, 0, lower, upper);

String s = mh.invoke("Hello, ", "world!");

assert s == "hello, WORLD!";
```

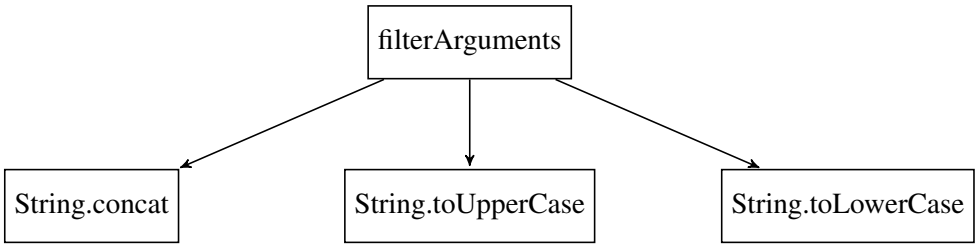


Figure 2.5: `filterArguments` combinator example and tree

- First a handle to the bootstrap method and the method type for the call site must be loaded from the constant pool of the class containing the instruction.
- Next the list of additional arguments for the bootstrap method is loaded from the constant pool. These arguments can be used to convey additional information about the call site to the bootstrap method.
- Finally the bootstrap method is invoked resulting in a `CallSite` object.

The `CallSite` object itself is simply a reference to a method handle to be invoked every time the invokedynamic instruction is executed. Since method handles are opaque and stateless directly returning a handle from the bootstrap method would mean the call sites target is forever fixed. To directly support the evolution of code at runtime a call sites target can be altered by application code.

The `CallSite` mechanism allows for a straightforward encoding of inline caches in bytecode[49]. Here the target handle of the call site performs a fast check to see if the actual method to invoke has changed, and if it has not it directly invokes the methods. If the actual method to call has changed, for example if the invokedynamic instruction was called with a new receiver type, the current target handle creates a new handle that also checks for the new type and sets it as the new target of the `CallSite`.

Even though the bytecode type system can represent method types for call sites, there are no method types for values on the stack or in local variables. Since JSR-292 intends to add support for dynamic languages it does not add such static method types for values. So all different types of methods are represented by the single `MethodHandle` type and all invocations of a handle must be dynamically checked.

Note that while it is possible to call method handles from Java programs, there is no feature in the Java language version 7 for which the Java compiler emits an `invokedynamic` instruction.

CHAPTER 3

Related Work

3.1 HotSpot

HotSpot is one of the most advanced Java virtual machines available today, originally developed by SUN Microsystems and now Oracle. This is also the VM where the original implementation work for JSR 292 was done on.

Besides the closed source version Oracle there is also an open source version of this JVM, developed by the OpenJDK project[42]. We subsequently only refer to the open source variant of the VM.

HotSpot uses a hybrid approach to execution of bytecode, where code is initially executed with an interpreter and only once it is considered worthwile it is optimized and compiled to machine code. There are two JIT compilers available for this VM, the client and the server compiler. The server compiler applies a larger set of optimizations to the target bytecode, trading off larger compilation time for quality of the generated code.

HotSpots JSR 292 implementation has gone through two iterations. The original one relied on a large body of hand optimized machine code routines to implement type checking, dispatch and combinators for method handles[57]. The basic idea behind that design is very similar to that of HotSpots template based interpreter[53], where specialized machine code routines are assembled at runtime for a given bytecode, or in this case method handle.

Since method handle combinators can insert or remove arguments from a call, new stack frames, so called *ricochet frames*, had to be introduced. As a consequence HotSpots stack walking code, which is required for garbage collection as well as for security checks also had to be adjusted. The problem with this approach is that for each target platform a new version of all machine code stubs has to be maintained. Another caveat is that execution of a method handle tree requires an indirect branch to the code for each visited node.

The server compiler which is able to perform aggressive optimizations on bytecode used a different approach. Instead of reimplementing optimizations for the new IR of method handle trees the server compiler first transforms such trees to regular sequences of bytecode which it can then translate to efficient machine code.

To increase portability and ease maintenance and evolution of the code base the second iteration of the JSR 292 implementation replaced the machine code stubs with an interpreter for combinators written entirely in Java[50]. Furthermore the compilation of combinator trees to bytecode is now also handled by Java code, meaning both the client and server compiler no longer require any specialized logic for optimizing method handles.

The following will focus on the newer implementation of JSR 292 since the Java framework for method handles is reused by CACAO.

3.1.1 Method handles in HotSpot

Method handles can represent a plethora of different operations, such as field and array accesses, calls to static or virtual methods, argument transformations for other handles and conditionals. Furthermore, they are dynamically typed and calls to handles require not only type checks but context dependent type conversions for arguments and return values. The rich semantics mean HotSpot has to overcome a number of hurdles to achieve high performance with its implementation.

Invoking a method handle generally involves the following steps:

- Checking the runtime type of the handle against the static one of the call site, and, if necessary, applying conversions to the arguments and return value.
- If the direct handle is wrapped in any method handle combinators, its argument transformations must be applied.
- Finally, linking to the correct code that implements the underlying VM operation described by the handle, i.e. calling a method or accessing a field.

The following sections explain in detail how each of these steps is implemented in HotSpot.

3.1.2 Member names

As mentioned before a method handle is not just a function pointer as in C but can represent a number of operations. Internally HotSpot introduces another concept, called a `MemberName`, used for directly referencing methods and fields of a class. Just as the MethodType class reifies the bytecode concept of a type descriptor to make it available at runtime, the `MemberName` class reifies references to a named member of a class.

Like references in the constant pool `MemberName` objects are resolved lazily, that is, only when the member is about to be used. During resolution the VM stores a pointer to its internal C++ representation, also used by HotSpots bytecode interpreter and JIT, of the resolved member is saved in the `MemberName` object. If a `MemberName` has been resolved to a method that C++ object contains both the bytecode and, if it has been compiled, the machine code for that method. `MemberName` that are resolved to fields also contain the offset of that field in its class. Both bytecode interpreter and JITted code can thus call a method or load a field stored in a `MemberName` with only a handful of instructions.

For actually invoking methods stored in a `MemberName` HotSpot introduces additional signature polymorphic methods to class `MethodHandle` called linker methods. As shown in figure 3.1

```
class MethodHandle {
  // unchecked call to an arbitrary MethoHandle
  @PolymorphicSignature
  final native Object invokeBasic(Object... args) throws Throwable;

  // unchecked call to a MemberName which references a virtual method
  @PolymorphicSignature
  static native Object linkToVirtual(Object... args) throws Throwable;

  // unchecked call to a MemberName which references a static method
  @PolymorphicSignature
  static native Object linkToStatic(Object... args) throws Throwable;

  // unchecked call to a MemberName which references a special method,
  // i.e. a constructor or private method
  @PolymorphicSignature
  static native Object linkToSpecial(Object... args) throws Throwable;

  // unchecked call to a MemberName which references an interface method
  @PolymorphicSignature
  static native Object linkToInterface(Object... args) throws Throwable;
}
```

Figure 3.1: Additional signature polymorphic methods for HotSpot

there are four such linker methods, one for each instruction used to invoke methods on the JVM, The fifth additional method, `invokeBasic`, is used to call method handles of an unknown kind and is explained later.

Contrary to `invoke` and `invokeExact` the linker methods are static, that is they do not require a `MethodHandle` as a receiver argument. In fact a call to a linker requires exactly the same arguments as one to the method stored in the `MemberName` would, since the argument are simply forwarded to that method. The `MemberName` object itself is passed to the linker as an additional last argument, but this constraint cannot be expressed via Java's type system.

Member names can be created for any kind of member that can also be referenced via a direct method handle stored in a classes constant pool. The only difference is that a `MemberName` pointing to a constructor really points to the initialization method and does allocate any new objects itself. As a consequence `MemberName` objects can be used to circumvent the safety constraints of the JVM since a `MemberName` that points to a constructor method allows calling that constructor without guaranteeing that the receiver argument is an uninitialized object. Neither the interpreter nor the machine code generated from it in fact perform any security checks at runtime when using a `MemberName`. It is for example possible to call a virtual method as if it were a static one via `linkToStatic`, but since those two kinds of calls require different machine code this would in all likelihood crash the VM. To uphold the JVMs promise of safety `MemberName` objects and the linker methods of `MethodHandle` are inaccessible from user code.

Note that there are no linker methods for reading or writing a field reference by a `MemberName`. For this HotSpot uses already existing methods in the class `Unsafe` which allow unchecked reads

and writes from an object at any offset.

Both the linker methods as well as the methods in `Unsafe` cannot be expressed in Java bytecode and are instead implemented directly in machine code. For the methods `Unsafe` this code can be created ahead of time but since the linker methods can be called with arbitrary arguments new code is created for each type signature. Since HotSpot knows the exact semantics of all these private methods its JIT compiler is able to inline them, even though that is, in general, impossible for native methods.

3.1.3 Lambda forms

With `MemberName` objects HotSpot can directly represent and call function pointers. To implement method handle combinators and handles to constructors HotSpot could just generate bytecode at runtime and store these synthetic methods in a `MemberName`. However creating a method at runtime also entails defining a new class as a container for the methods. To analyse and profile the bytecode of every class the HotSpot VM additionally allocates a number of internal data structures. The current process of class generation also involves emitting a whole class file into memory and then parsing and typechecking it. As a consequence generating custom bytecode for each combinator is quite expensive and should be avoided.

For that reason HotSpot uses a new intermediate representation called lambda forms to represent combinators. The first few times a method handle is executed its lambda form is run by an interpreter written in Java, only after certain threshold is reached the lambda form will be compiled to bytecode. All subsequent executions will go directly through that code allowing HotSpots compiler to analyse and optimize it like any other piece of code.

Internally lambda forms use neither the direct tree representation implied by combinators nor the graph based SSA[2],[43],[29] IR used by its JIT compiler. Instead they use a kind of administrative normal form[17] (A-normal form), a program representation otherwise used by compilers for functional programming languages. Expressions in A-normal form consist solely of function calls and variable bindings. Furthermore the normal form stipulates that the result of every expression is bound to a variable and that expressions do not nest, meaning that the arguments of every call are all variable references. Lambda forms thus make the evaluation order of a combinator tree explicit by flattening the method handle tree to a sequence of calls.

Furthermore, the lambda form compiler knows the exact semantics of all method handle combinators, they can always be inlined, meaning a whole combinator tree can be compiled into a single method that forwards to a direct method handle, which greatly reduces the method call overhead.

Since lambda forms are mainly intended to express sequences of argument transformations they do not have any facilities for conditionals, looping or recursion. Guard conditions are still expressible with the help of a simple helper method written in Java that takes a boolean flag and two method handles and returns one of the given handles depending on the value of the flag.

The bytecode compiler for lambda forms actually takes care to emit bytecode patterns that are known to produce good machine code when processed by HotSpots JIT compiler. For instance, the helper method for conditionals is compiled to a pattern that leads the JIT to speculatively inline the first of the two handles, meaning it can be used to implement efficient inline caches.

Besides method handle combinators lambda forms are also used for handles that call constructors, read or write to object fields or access arrays. All these kinds of handles are implemented with lambda forms that forward to utility methods written in Java. Only constructors require a sequence of calls, to both allocate and initialize the new object.

Together lambda forms and `MemberName` allow user code to invoke a method handle without knowing its kind. Each `MethodHandle` object stores a lambda form that in turn contains a `MemberName` that always points to a static method. The method `invokeBasic` of `MethodHandle` loads the `MemberName` object and calls its static method. When the lambda form has not yet been compiled that method will simply enter the lambda form interpreter. Once it has been compiled the `MemberName` is replaced with a new object that directly points to the newly generated code. Since the bytecode generated for a direct method handle knows what kind of method it references it can directly invoke the correct linker method and does not require any checks at runtime.

3.1.3.1 Invokers

In principle the JSR 292 specification states that the type safety of every call to a method handle requires a runtime type check to uphold the safety constraints of the JVM. Since complex dispatching code can require a large tree of method handle combinators calling a single method via handles could involve a large number of such checks.

The number of checks actually performed at runtime can be reduced considerably using the observation that they are only necessary when entering a method handle combinator tree from untrusted code. Since combinators are stateless and do not change their behaviour once created they can check the type of their target handles during initialization. Any call to an inner handle of the expression tree is then type safe by construction and can be performed via a linker method or `invokeBasic`, which do not incur the overhead of a type check.

To exploit this HotSpot separates the invocation of handles into two distinct phases. The first phase can be initiated by any user code by calling either MethodHandle.invoke or MethodHandle.invokeExact, it checks the static type of the call site against the runtime type of the invoked method handle and also applies the necessary conversions for a generic invocation. The second phase, which is responsible for performing the actual operation for the handle, is implemented via lambda forms as described above.

Since only trusted code can be allowed to call a method handle unchecked, the bytecode for type checks and conversions is not emitted directly at each call site but implemented as a separate static method, called an invoker. Every method handle call site is then rewritten to an invocation of that invoker.

To prevent an explosion of code size invokers are not actually generated per call site but per method type, once for exact and once for generic invocation, all call sites of the same type can then share the same invoker.

The number of necessary invoker methods are reduced even further by exploiting two simple observations. First, that method handle combinators treat all reference parameters the same, no matter what their actual class is, that is bytecode behaviour of no combinator actually ever cares about the class of an object. Furthermore, on the operand stack and in local variables all primitive types smaller than `int`, i.e. `boolean`, javachar and `short`, are represented by an `int`. This means

any method type used by method handles after the initial method type check has been performed can be simplified to a so called basic type, that does not make these unecessary distinctions. Basic method types simply erase all object types to `java.lang.Object` and all small primitive types to `int`, now a much larger range of call sites can be serviced by a single invoker method.

A consequence of one invoker handling call sites of different types is that the expected method type of a call site has to be provided as an argument to the invoker since it does no longer know about the precise type signature. HotSpot solves this by appending the expected method type to the argument list of the invoker, simply by pushing it on top of the Java operand stack. This additional argument is referred to as the appendix.

One problem with this scheme is that the invoker method now has one more parameter than the actual invoked handle, since the JVM limits argument lists to a length of 255 this transformation might cause an invalidly long argument list. In such rare cases HotSpot does not perform any type erasure and creates a custom invoker method tailored to the exact method type, which does not require an appendix argument. Note that in bytecode adding the appendix to a call only amounts to pushing another value onto the stack. When generating machine code, though, the JIT must allocate an additional register or stack space to the call, meaning it has to detect ahead of time if a method handle invocation uses an appendix.

Since the invoker method is statically known and usually only a few instructions long it can easily be inlined by HotSpot meaning there is essentially no overhead to it in hot code. If the invoked handle is also inlined speculatively, the type check can be removed completely.

To guarantee that the unsafe methods in `MethodHandle` are only used by trusted code HotSpot reuses the JVMs visibility mechanism. Since all additional signature polymorphic methods in `MethodHandle` are package private, they can only be accessed by the trusted code in the `java.lang.invoke` package. Invokers and all other classes generated for method handles by the VM at runtime are also placed in that package in order to give them access to these methods. On the other hand the JVMs own class loading policy prevents any external class loader from defining a class that is in the `java.lang` package or any package nested therein, making them unreachable from user code. Just as with `invoke` and `invokeExact` it is also impossible to call these methods via reflection, trying to do so will result in a runtime exception being thrown.

3.1.4 `invokedynamic` in HotSpot

Since it can reuse the whole machinery created for method handle combinators the implementation of `invokedynamic` in HotSpot is comparatively simple.

Similar to an `invokeExact` call an `invokedynamic` instruction is rewritten to a call to a custom invoker method that on its first execution resolves a method handle by invoking the call sites bootstrap method and from then on forwards to that handle. To allow call sites with the same signature to share invoker methods the `CallSite` object is provided to the invoker by the VM in the same way as the `MethodType` for an `invokeExact` call.

As an optimization HotSpots JIT assumes that the target handle of an `invokedynamic` call site does not change often and speculatively inlines it. Each time the target does change, the specialized machine code is invalidated so that specialized code for the new target can be generated. If a call site changes too often, the VM stops trying to inline the handle and generates code that calls the target handle normally.

3.2 JamVM

JamVM[35] is a small interpreter only JVM developed by Robert Lougher. It can be used with GNU classpath and OpenJDK, though JSR 292 support is only enabled when using OpenJDK, since it reuses the lambda form and bytecode generator framework.

Since it relies solely on an interpreter for executing bytecode it cannot compete with a VM such as HotSpot in terms of performance, on the other hand its source code is an order of magnitude smaller, than that of HotSpot.

To speed up execution of bytecodes that require some resolution the first time they are executed, such as method calls and field accesses, JamVM rewrites individual instructions as they are executed. For this purpose it uses the space of unassigned opcodes ranging from 203 to 253. An `invokevirtual` instruction is converted to an `invokevirtual_quick` instruction once the method that should be called has been fully resolved. The `invokevirtual_quick` instruction then no longer has to perform any checks and can directly dispatch to the target method.

JamVM also rewrites calls to the special signature polymorphic methods of `MethodHandle` to internal bytecodes. This is done for `invoke`, `invokeExact` and `invokeBasic` as well as all four linker methods, though only `invoke` and `invokeExact` are split up into an unresolved and resolved bytecode since other methods never require resolution.

The problem of adding the appendix arguments to a call site are non existent in JamVM since it uses an array as Java operand stack and can simply push another value on top of it. JamVM also directly stores a pointer to its internal representation for methods used during dispatch, the `methodblock` struct, in every resolved MemberNames vmtarget field. Direct invocation of a method handle thus requires only a few additional load instructions compared to a regular method invocation.

Since the source code of JamVMs JSR 292 implementation is much more compact and readable the CACAO implementation is influenced by it rather than directly mimicking HotSpots.

3.3 Dynamic Language Runtime

The dynamic language runtime (DLR)[58] is a framework for implementing dynamic languages on top of Microsofts Common Language Runtime (CLR). The CLR is a stack based virtual machine and its instruction set, the common intermediate language (CIL), shares many characteristics with the JVMs bytecode, though unlike the JVM it was designed from the ground up to be used as a host for different languages[36].

Since the CLR has had support for a form of method handles, called delegates, before work on the DLR began no changes to the bytecode instruction format were necessary for this framework.

The internal representation for code used by the DLR also extends a pre existing facility of the CLR, the so called expression trees initially introduced for the language integrated query (LINQ) feature[60]. Expression trees, like method handle combinators, are a tree based code representation, though expression trees are much more fine grained. While method handles and combinators always represent a directly callable function object, an expression tree can encode a single bytecode instruction, such as integer multiplication or an access to a local variable.

```
if (SiteContainer.Site1 == null) {
  SiteContainer.Site1 = CallSite<Func<CallSite, object, object, object>>.Create(
    CSharpBinaryOperationBinderFactory.Create(
      ExpressionType.Add, false, false,
      new CSharpArgumentInfo[] {
        new CSharpArgumentInfo(0, null),
        new CSharpArgumentInfo(0, null)
      }
    )
  );
}
object c = SiteContainer.Site1.Target(SiteContainer.Site1, a, b);
```

Figure 3.2: A dynamic call to '+' in C#[52]

The DLR also provides a whole framework for inline caching and other important dynamic optimization techniques, while JSR 292 only makes it possible to encode them efficiently.

Method handles are aimed at implementing dynamic dispatch code that calls methods written in regular bytecode, while expression trees can be used for all parts of the code generator for a language hosted on the CLR. IronPython, an implementation of the Python programming language implemented on the CLR, for example uses expression trees as its IR[61].

Contrary to the Java compiler the C# compiler can emit code for performing dynamic dispatch. Since the DLR does not rely on specialized instructions by the VM it requires some support code at every dynamic call site. Figure 3.2 shows the code emitted by the C# compiler for a dynamic call to the binary '+' operator in a class called `SiteContainer`. Note that there is no separate bootstrap method required, but a new static field for the `CallSite` object must be added to the class containing the call.

Mono, an open source implementation of the CLR, also supports the DLR.

3.4 IKVM

IKVM[27] is an implementation of the JVM on top of the CLR. Similar to JamVM it reuses HotSpots class library, but contrary to Jam IKVM replaces the entire lambda form and `MemberName` framework with its own implementation.

Since the CLR, as mentioned before, already has support for function objects in the form of delegates IKVM simply reuses those facilities and directly generates delegates and CIL bytecode for method handles and their combinators. Similar to the implementation in HotSpot an `invokedynamic` instruction is compiled to a load of a static delegate object, that when invoked boostraps the call site, or if that has already been done calls the target method handle.

IKVM does not reuse the DLRs expression tree facility though and prefers to use its own bytecode compiler.

```
int i = $indy_stub0plus$(5, 6);

private static int $indy_stub0plus$(int a, int b) {
  CallSite     cs     = SiteContainer.$indy_stub0plus$cs;
  MethodHandle target = cs.target;

  Integer aBox = Integer.valueOf(a);
  Integer bBox = Integer.valueOf(b);

  Integer i = (Integer) target.invoke(aBox, bBox);

  return i.intValue();
}
```

Figure 3.3: Code generated for an `invokedynamic` by JSR 292 backport

3.5 JSR 292 backport

JSR 292 backport[19],[20] is a project by Remi Forax, who was also a member of the JSR 292 expert group. It implements a bytecode transformer that rewrites invokedynamic instructions and uses of method handles so that they can be executed by a JVM that only supports version 5 or 6 of the JVM Specification.

JSR 292 backport can be used either as a runtime agent that transforms classes as they are being loaded or as an offline tool that rewrites class files. The implementation of method handles used by this implementation of JSR 292 uses essentially the same techniques as JRuby. Direct method handles are implemented using Java's reflection API and all kinds of method handle combinators are hand coded Java classes. For efficiency the invoker methods of `MethodHandle` are overloaded up to a fixed arity of eight, allowing for example a method handle that reads a field to completely avoid array boxing of arguments.

Like the C# compiler the backport transformer inserts static fields to hold the `CallSite` objects into the class containing the call site, additionally it must also inserts static fields for `MethodHandle` and `MethodType` objects in the classes constant pool. Like HotSpot the backport transformer also creates invoker methods for each `invokedynamic` call and invocations of method handles which are responsible for lazy creation of the `CallSite`, argument boxing and return value unboxing. Unlike in HotSpot though these invokers are created eagerly in the class containing the call and not lazily in a separate invoker class.

Figure 3.3 shows an unoptimized piece of code generated by JSR-292 backport for an `invokedynamic` instruction which calls a method called 'plus'. Note that the call to the target handle requires the primitive `int` arguments to be boxed in `Integer` objects and that the return value must be unboxed.

The online transformer also supports an optimizer that rewrites dynamic calls that are executed often using a strategy very similar to that of OpenJDKs lambda forms. Here the tree of method handle combinators is also regarded as a tree which Forax's optimizer walks during code generation. The generator knows about all types of method handle combinators and is able to directly emit an optimized version, effectively inlining all combinators. This way the overhead of

```
MethodHandle target = ...; // handle to method "foo";
MethodHandle cache  = SiteContainer.$method_handle_stub0$mh;

if (target == cache) {
  // direct invocation
  return foo(5, 6);
} else {
  // create new version of this method
  Optimizer.deoptimize();
}
```

Figure 3.4: Optimized code generated for calling a direct method handle with JSR 292 backport

invoking a method handle is reduced to two `invokevirtual` instructions, one to call the inlined combinator tree and one reflective call to the direct handle. When the direct handle is a constant even the reflective overhead can be avoided and a direct call is emitted.

The optimizer furthermore speculatively inlines method handles at call sites that are known to not change very often, inserting a deoptimization guard if the handle does actually change. An example of an optimized call is shown in figure 3.4.

The optimizer requires the underlying JVM to support reloading of classes, which CACAO unfortunately currently does not.

The current version of the JSR 292 backport does not check the method type of a method handle against the method type of a call site and it also does not correctly implement the type conversions required by `MethodHandle.invoke`. This is done in order to speed up execution, since Java's reflection API will check the type of the invoked method, an exception will still be thrown in case of an invalid call.

3.6 Truffle

Truffle[62] is a framework for creating virtual machines with a highly optimizing JIT compiler from simple abstract syntax tree (AST) intepreters written in Java.

Execution with Tuffle starts in the AST interpreter, and once a method has been executed a number of times the framework performs partial evaluation on the interpreter code using the AST of the target method as input. The result of this partial evaluation, a form of Futamura[21] projection, is the compiled code for the given method.

This is quite similar to the way the lambda form compiler operates, but AST nodes in Truffle are not limited to a fixed set. Instead arbitrary new node types can be created, provided they are expressible using Java and a number of intrinsic methods provided by the framework.

Dynamic profiling driven optimizations in Truffle are enabled by AST rewriting. Nodes in Truffles AST can replace themselves or their children with more efficient computations, for example a multiplication node can replace itself with a binary bit shift if one of its arguments is always a power of two. Such rewrites automatically trigger a recompilation of the nodes method, meaning the JIT compiler can incorporate new information as it becomes available.

Truffle also has an API checking for assumptions which are relatively stable. An Assumption object represents a boolean flag that is expected to change very infrequently and allows user code to check if the assumption still holds and to invalidate it. The JIT then creates code that does not actually checks the flag but records that a given method depends on an assumption, it shifts this burden to the invalidation logic. When an assumption does eventually become invalid the JIT also invalidates all code that depends on this assumption and recompiles it when it is reached again.

The actual machine code generation for Truffle is performed by Graal[9],[14] a highly optimizing compiler written in Java which was originally developed for the Maxine JVM and later ported to the HotSpot JVM.

The goal of Truffle is not necessarily to generate VMs that outperform those hand tuned for a given language, but to make it easy to write implementations that are faster and easier to maintain than interpreters written in C or other low level languages with relatively little effort.

CHAPTER 4

JSR-292 on the CACAO JVM

The second

Since CACAO is already able to be run with HotSpots class library we chose to reuse its bytecode generation framework for method handles. For low level and CPU architecture dependent tasks the framework depends on a number of private native methods which must be implemented by the underlying VM which had to be reverse engineered or ported to CACAO. Unfortunately these APIs are tailored closely to HotSpots architecture and require a translation layer for converting its data structures to those expected by CACAO and vice versa.

CACAOs implementation of JIT support for signature polymorphic methods underwent two iterations. The first design was based closely on that of JamVM, here each polymorphic invocation was translated to a static call to a lazily generated stub function. These stubs would, if necessary, load the appendix argument, perform any required dispatch and then tail call the target method of the invoked handle.

The main benefit of this scheme was that the stack analyser and verifier were unaware of any appendix arguments and did not have to be changed at all. However, each polymorphic invocation now had the additional overhead of calling a stub function and code memory had to be allocated for each of these stubs.

The second, and current, design inlines the stubs for most signature polymorphic calls, avoiding the cost of an indirect branch and reducing code size. On the downside this approach required deeper changes to the earlier passes of the JIT for handling appendix arguments.

4.1 Architecture of the CACAO JVM

The CACAO JVM uses a fast just in time compiler to translate Java bytecode to machine code for execution. Unlike other JVMs such as HotSpot it does not use a hybrid approach where methods are first run with an interpreter to avoid the overhead of compiling code that is only used once. This is feasible since CACAOs JIT is very fast, but as a trade off it does not perform many optimizations.

A good overview of CACAOs architecture is given by Christian Thalinger in his diploma thesis[56], the following is a short outline of the several modules of CACAO a Java class passes through in it's life cycle. Modules relevant to the thesis are explained in more detail later.

Class loader The class loader is invoked either explicitly, via a call to the method `loadClass` method of a class loader object, or implicitly because a reference in another class is used. At this point CACAO loads the constant pool, which contains numeric constants as well as references to other classes, fields and methods, and the classes attributes, which store Java annotations, debug information and other meta data. Class loading checks several static constraints imposed on Java class files, such as validity of method type descriptors, but does not check the actual bytecode for any method.

Linker The linker determines the offsets of fields and the indexes of methods in a classes virtual function table (vtable) used for dynamic dispatch. Since no methods have been compiled when a class is being linked the vtable is filled with stub functions that invoke the JIT compiler when executed. Just like class loading linking is performed lazily.

Bytecode parser This is the first phase of the JIT compiler, it is triggered either by an uncompiled method being invoked the first time or if a method has been marked for recompilation by the optimizer. Its main responsibility is parsing the Java bytecode of a method into intermediate commands (ICMD), CACAOs intermediate code representation. The bytecode parser also determines basic blocks, checks that all local jumps stay within the current method, and pre-allocates registers used for function calls. This pass furthermore ensures that all constants from the constant pool are used correctly, for example that a reference to a method is not interpreted as a numeric constant.

Stack analysis Every method on the JVM must predeclare the maximum depth the operand may reach during its execution, and the maximum number of local variables it may use. The stack analyser checks that neither operand stack underflows or overflows may occur at runtime, validates the local variable indices appearing in the program and performs a type analysis of untyped byte codes. Since all new IR instructions introduced for JSR-292 behave similiarly to pre-existing ones from the perspective of the stack analyser only limited changes to this pass were required.

Verifier The JVM specification requires that the operand stacks and local variables are used in a type consistent way, meaning it is, for example, forbidden to use integer instructions on floating point numbers. The verifier makes sure that these and other constraints are upheld by every method, but due to the JVMs lazy class loading it can not always do all of these checks ahead of time. When a field or method of an unloaded class is used the verifier stores the relevant constraints, the code generator will then emit code that validates them when the code is actually executed. The design of CACAOs verifier follows that proposd by[8]. CACAO has not always had a verifier, which is still apparent in the fact that it can be disabled if desired. In that case the bare minimum of checks necessary for the VM to function are performed in the bytecode parser and stack analysis.

Optimizer A number of optimizations, such as array bounds check elimination, if-conversion[1], escape analysis for stack allocation of objects[37] and inlining[55], have been implemented for the CACAO virtual machine.

Register allocator There are currently two register allocators available for CACAO, the default is a fast and simple allocator described in[30], the other one is a linear scan register allocator[44].

Code generator The code generator performs a straightforward translation of each individual intermediate instruction to a small piece of machine code.

Patcher As mentioned before accesses to fields and methods of unloaded classes have to be treated specially by the code generator, for these cases it emits a trap instruction. When the CPU encounters that trap instruction it suspends execution and calls a signal handler function installed by CACAO. This function then performs any deferred class loading, resolution of methods or verification. Once this is done the trap instruction is overwritten with a no-op and execution is resumed.

The same technique is used to implement lazy compilation. Every invocation of an uncompiled static method is prefixed with a trap, and as mentioned before linking fills a classes vtable with stubs that consist only of a trap instruction. These compiler traps simply invoke the JIT and cause it to compile the requested method. Once the method has been compiled, the static call site or vtable slot is updated to now point to the newly created code.

4.1.0.1 Other class libraries

Besides the OpenJDK class library CACAO also supports the GNU Classpath class library, which currently only contains classes for version 5 of the JVM specification. Since the two class libraries require a different interface from the underlying JVM CACAO uses preprocessor directives to select the right code to compile.

Since the large number preprocessor directives makes the code harder to read and maintain we chose to use a different approach for the new API. All new types and function prototypes required for JSR-292 are located in the C++ namespace `MethodHandles` which is contained in a single header, which is used for any configuration of CACAO. The build system then selects between two different source files containing the implementations for the two different class libraries.

For GNU Classpath all new APIs are implemented as short inline functions that always throw an exception or return a dummy result. With this approach we can still achieve zero runtime overhead for unused APIs. Consider for example CACAOs class file loader, due to some requirements of JSR-292 it now contains several conditionals that never evaluate to true when using GNU Classpath. Since the check for that code is an inlinable function that always returns false any modern C++ compiler can eliminate these if-statements completely.

For version 6 of the OpenJKD class library, which also does not contain the classes required for JSR-292, we use exactly the same approach as for GNU Classpath. In fact most of the functions used by the JIT from namespace `MethodHandles` are used by both configurations.

4.2 The class file loader

The actual JSR-292 specification requires only a few changes to the class loader, but under the covers HotSpots bytecode generator uses a number of features that needed to be added to CACAO.

4.2.1 The constant pool

The constant pool of a class contains numeric and string constants as well as the names and types of all classes, fields and methods used by the bytecode. Furthermore, to save space, names and type descriptor strings are also stored as entries in the pool, allowing methods with the same name or type to share the same string constant.

Every entry in a classes constant pool is identified by a 16 bit number which is used by bytecode to refer to it and also to implement references within the constant pool itself.

JSR-292 adds two new kinds of constant pool entries, one for method handles and one for method type objects. The code for parsing these new entry types is straightforward and does not differ much from that for any other type of entry.

Each entry of the constant pool starts with a tag byte that identifies its type followed by a body whoose size and contents depends on its type. A few example constant pool types can be seen in figure 4.1.

Since bytecode is statically typed the Java class file format has always had a way to encode the types of a methods arguments and its return type via so called method type descriptors. A method type descriptor in the constant pool is an UTF-8 string containing the type in human readable format. Method type descriptors are not first class values, though, and can only be referenced from a method invocation bytecode. As a consequence they can not be pushed on the operand stack or stored in a local variable. In the constant pool `MethodType` objects are simply a wrapper for a method type descriptor string and they can be loaded as a regular object and pushed on the operand stack via the `ldc` instruction.

A `MethodHandle` constants body consists of a second tag byte specifiying its kind (see figure 2.1) and a reference to another constant pool entry that must be either a reference to a field or method.

The C++ objects used to represent the new constant pool entry types are called `constant_MethodHandle` and `constant_MethodHandle` respectively.

The class loader only checks that constant pool entries are well formed, i.e. that a method type descriptor is syntactically correct or that the reference in a `MethodType` really points to a type descriptor. It does not validate if the referenced classes, fields or methods really exist, this is only done lazily when the references are actually used at runtime.

At runtime CACAO stores the constant pool in two arrays of the same size, one for the type tags and one for pointers to the actual data structures which represent the constant pool entries. These arrays are fully populated by the loader and not written to afterwards, allowing multiple threads to access them without the need for synchronization.

Once a constant pool entry has been resolved the result is cached in the reference object to speed up future resolution attempts. Since resolution does not allocate any objects but only

```
// u1 ...  8 bit unsigned int
// u2 ... 16 bit unsigned int
// u4 ... 32 bit unsigned int

// general layout for all constant pool entries
cp_info {
  u1 tag;    // identifies type of entry
  u1 info[]; // body of entry
}

// UTF-8 string in the constant pool
//  used for method name & method type descriptor
CONSTANT_Utf8_info {
  u1 tag;               // set to CONSTANT_Utf8
  u2 length;
  u1 bytes[length];
}

// encode method name and types for invoke* instructions
CONSTANT_NameAndType_info {
  u1 tag;               // set to CONSTANT_NameAndType
  u2 name_index;        // reference to name of method
  u2 descriptor_index;  // reference to method type descriptor
}

// MethodType object in the constant pool
CONSTANT_MethodType_info {
  u1 tag;               // set to CONSTANT_MethodType
  u2 descriptor_index;  // reference to a method type descriptor
}

// MethodHandle object in the constant pool
CONSTANT_MethodHandle_info {
  u1 tag;               // set to CONSTANT_MethodHandle
  u1 reference_kind;    // type of handle (see table 2.1)
  u2 reference_index;   // reference to a method or field descriptor
}
```

Figure 4.1: Layout of entries in the constant pool

performs a lookup this cache also requires no synchronization. If two threads try to resolve a reference concurrently they will always both fail or store the same object in the cache.

4.2.1.1 Garbage collection issues

The VM must be aware of all references to objects in the heap since otherwise it could erroneously conclude that an object is unreachable and reclaim its memory. In CACAO traditionally only Java objects are allocated on the garbage collected heap, all internal data structures are placed on the C heap and do not point to the Java heap.

The only type of constant pool entry that could contain a Java object previous to JSR-292 is

CONSTANT_String, which resolves to a `java.lang.String` object. Since CACAO instantiates itself these constant `String`s it can allocate their storage on the C heap, meaning the garbage collector does not have to be aware of them or any pointers to them.

With method handles and method types there are two new types of entries for which this solution does not work. The `MethodHandle` and `MethodType` objects for these entries are created by HotSpots JSR-292 framework and CACAO is not able to control their allocation. To make the garbage collector aware of the objects `constant_MethodHandle` and `constant_MethodType` objects must also be allocated on the Java heap, allowing the garbage collector to scan them for pointers to other objects.

CACAO can use two different garbage collectors, the conservative Boehm-Demers-Weiser[5],[4] (often simply referred to as BoehmGC) and a precise garbage collector written by Michael Starzinger[54]. Currently only the BoehmGC can handle non-Java objects that may point Java objects, the precise collector will require further work to support this solution.

4.2.1.2 Optimizations

The class file format does not place any restrictions on the order in which entries of the constant pool may appear. This makes a simple one pass parsing of the constant pool impossible since an entry might refer to another one that has not yet been encountered. To solve this problem CACAO stores all entries of the constant pool that may refer to a yet unparsed entry in a list and then resolves forward references. By design the constant pool cannot contain reference cycles so the resolution can be done with one pass per kind of entry.

Previously the data structures representing class references and field type descriptors in the constant pool were cached per class and allocation of the cache was delayed until the last possible moment in the class loading process. This lead to some contortions in the code and required two additional passes through the constant pool array to fully initialize all data structures.

To simplify the code base class references are now cached globally and created as soon as possible, eliminating the need for the additional back patching passes. Since field type descriptors fit in two words they are no longer allocated on the heap and instead are directly embedded in the data structures that use them.

4.2.1.3 Method type descriptors

Method type descriptors are strings stored in the constant pool and are used in several places in the Java class file format to specify a methods parameters and return types. They do not, however, encode the type of the methods receiver argument, or whether it even expects a receiver.

At runtime CACAO represents the type descriptors with struct `methoddesc`, which are parsed from the descriptor strings during class loading. For methods of the class itself the class file encodes whether they are static or require a receiver, but for methods in the constant pool it does not. The loader thus only half initializes the `methoddesc` for references and leaves enough space for a receiver argument. Only during bytecode parsing, when a method reference is used from a method invocation instruction, is that slot filled in. When the refering bytecode is an `invokestatic`, the slot remains ununsed, otherwise all parameters are copied one slot to the right and the first one is filled in with the type of the receiver. The bytecode parser then also

allocates a second data structure used to hold the registers assigned to each parameter by the platforms calling convention.

The appendix arguments used by HotSpots bytecode generator are also not encoded in method type descriptors. With the current system we would have to allocate space for one more parameter in each `methoddesc` in the loader, even though it would only be needed in a small number of cases, and then initialize that slot in the parser. To avoid this unnecessary overhead the process of parsing method descriptors for method references is moved entirely to the bytecode parsing pass. When creating the `methoddesc` at this later stage all necessary information for fully initializing them is available and static method descriptors do not incur the penalty of an unused parameter slot.

Another advantage of this approach is that the two arrays for describing a methods Java types and its register allocation information can be merged into one, saving an additional allocation. It furthermore allows us to cache `methoddesc` objects globally, which means there are no duplicate instances for methods with the same type in different classes.

Determining if a method descriptor should have an appendix or not is unfortunately not as straigtforward as it is with the receiver argument. HotSpots bytecode generator makes this decision if an appendix is necessary when a method handle call site is linked at runtime. Since HotSpot uses an interpreter for executing bytecode the first time this does not pose a problem, if there is an appendix that additional argument is simply pushed onto the interpreter stack. CACAO on the other hand uses a compile only approach and its needs to allocate machine registers and stack space during compilation before the call site is ever reached.

It was thus necessary to reverse engineer the decision criterion used by the bytecode generator. As described in section 3.1.3.1 this depends on the number of arguments for a call, an appendix is only used if adding it does not exceed the limit of 255 arguments. In addition to this restriction HotSpot also reserves two additional parameter slots for internal use in generic method handle invocations. CACAO now uses the same rules for parsing method type descriptors for method handle call sites and to allocate registers for them.

The bytecode generator uses three constants to calculate the threshold for using an appendix: `MAX_MH_INVOKER_ARITY` (254, the maximum arity of a method minus the receiver), `MTYPE_ARG_APPENDED` (one slot for the appendix) and `GENERIC_INVOKER_SLOP` (two additional slots for `invoke`). The values of `MAX_MH_INVOKER_ARITY` and `GENERIC_INVOKER_SLOP` are static constants in the helper class `java.lang.invoke.Invokers` and can be checked by CACAO at startup, `MTYPE_ARG_APPENDED` is only a local variable which is always assigned the value one.

When first linking a method handle call site CACAO checks if it predicted correctly if HotSpot would use an appendix or not. If a version of HotSpots class library that uses different criteria than those above is used the machine code for the call site would be invalid, we thus throw an `InternalError` exception which aborts execution of the method and prevents the invalid code from being reached.

For the `invokedynamic` instruction all these problems do not exist and we can always assume that the call requires an appendix. Remember that executing a resolved and bootstrapped `invokedynamic` instruction proceeds as follows: First the `MethodHandle` target of the `CallSite` object is retrieved, then we call the `MethodHandle` via `invokeExact`. Since `invokeExact` is a virtual method it can have at most 254 arguments plus the receiver, the linker method for the

`invokedynamic` thus gets at most 254 arguments plus the `CallSite` objects. Thus a call to the linker method never exceeds the JVM parameter limit. Whithin the linker we only retrieve the `CallSite`s target, push up to 254 arguments, and then invoke it, obviously the argument limit is also preserved here.

4.2.2 Attributes

Attributes in the Java class file format are used to store additional information of a class or its fields and methods. A methods bytecode and debug information or the value of a static final field are also encoded as attributes.

JSR-292 adds the `BootstrapMethods` attribute, shown in figure 4.2, which contains the bootstrap methods for a classes `invokedynamic` instructions. In the naming convention of the JVM specification a `u2` or `u4` is an unsigned two or four byte integer.

```
BootstrapMethods_attribute {
  u2 attribute_name_index;
  u4 attribute_length;
  u2 num_bootstrap_methods;
  {
    u2 bootstrap_method_ref;
    u2 num_bootstrap_arguments;
    u2 bootstrap_arguments[num_bootstrap_arguments];
  } bootstrap_methods[num_bootstrap_methods];
}
```

Figure 4.2: The `BootstrapMethods` attribute

Each bootstrap method in the attribute contains a reference to a method handle that will be used as the bootstrap method, and a variable number of references into the constant pool which encode the additional arguments for the bootstrapping process.

In the loader we only have to check if the constant pool references in the attribute are valid, resolution of the method handle and arguments occurs much later when `invokedynamic` instructions are executed.

4.2.3 Anonymous classes

Anonymous classes, not to be confused with the Java language feature of the same name, are the first of the additional features required for HotSpots bytecode generator framework.

Usually a class in the JVM is identified uniquely by its name and the class loader that loaded it. Classes are also cached by the VM and attempting to load an already loaded class always results in the same `Class` object. Furthermore, every class loader keeps a strong reference to every class it has loaded, preventing these classes from being unloaded as long as it is alive.

Anonymous classes were initially introduced to reduce the memory overhead caused by the many classes generated at runtime by JRuby. Since the JSR-292 framework also generates a class for each compiled lambda form and direct method handle it also benefits greatly from this mechanism.

Anonymous classes simply allow a class to be loaded without consulting the class cache and without adding a reference to it to a class loader. The benefit is that many anonymous classes can share one class loader and that the class loader does not have to contain a list of all classes it loaded.

Loading of anonymous class is exposed to Java via the native method `defineAnonymousClass` of the `Unsafe` class, which contains a number of private APIs that break the safety promises of the JVM, such as unsafe memory access. `defineAnonymousClass` completely bypasses the Java classpath and class loader infrastructure, it expects the contents of the class file to be passed as an in memory byte array and copies the class loader from a host class into the new class. Unlike with regular class loading now methods of the class loader are called, since they would register a reference to it in the loader.

To reduce the memory footprint of the API even more `defineAnonymousClass` also allows the constant pool of the loaded class to be patched. As a consequence loading two classes that differ only in the contents of their constant pool requires only one class file to be created.

HotSpot also uses this patching mechanism to store objects in the constant pool that could usually not occur there. A prime example of this are instances of `Unsafe`, which are expensive to construct since that entails a call stack inspecting security check. Previously CACAO never stored Java objects directly in the constant pool, even `String` constants are stored as UTF-8 strings and created as needed. Even though class objects themselves are visible to the garbage collector, the constant pool array is not, thus, just as with `constant_MethodHandle` we need to make the garbage collector aware of these pointers into the Java heap. One option would be to allocate all constant pool arrays from the Java heap, but since only a small portion of all constant pool entries is likely to be patched this would put unnecessary pressure on the garbage collector. Instead a new uncollectable pointer sized object is allocated on the heap and a reference to the patch object is stored in it. This ensures that patch objects are always reachable by the GC and also makes them effectively uncollectable.

In the classes created by HotSpots JSR-292 framework constant pool entries that are intended to be patched always contain dummy string entries. To distinguish them patched entries are marked with the new tag `CONSTANT_Object`, allowing the various reflection and debugging procedures in CACAO to recognize them.

4.2.4 Field injection

The class `MemberName` is special since it has a field `vmindex` that contains a pointer to machine code. There are two plausible options for representing that field in the class file for `MemberName`, either with an object type or with an integer type. Unfortunately both these approaches have severe drawbacks that make them undesirable.

Using an object type for the code pointer would open the door to memory corruption exploits. One possible attack abuses the memory layout of Java objects in CACAO, which can be seen in figure 4.3. Every object starts with a pointer to its vtable, which in turn contains a pointer the corresponding `Class` object. An attacker first obtains the code pointer ‘object’ from an instance of `MemberName` via Javas reflection API and then tries to cast the object to another type. The code generated by CACAO for the `checkcast` bytecode loads the `Class` object for the required runtime type check. Since the vtable pointer loaded from the code ‘object’ is actually a piece of

machine code the address is a random location in memory, rather than that of a `Class` object. Trying to use that `Class` will most likely crash the VM with an invalid memory access, but using a `MemberName` pointing to a specially crafted code sequence could probably used for other attacks.

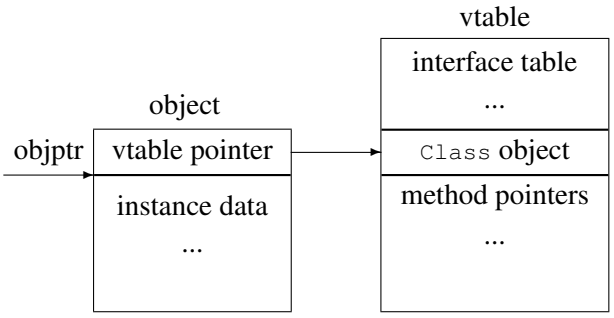


Figure 4.3: CACAO object and vtable layout

The downside of using an integer to represent `vmindex` is that the JVM does not offer a platform independent pointer sized integer. The only available types are the 4 byte `int` and the 8 byte `long`. To support 64 bit platforms `long` would have to be used, but this would waste 4 bytes of space per `MemberName` instance on a 32 bit system.

To work around these restrictions HotSpot chooses to not encode a field for the code pointer in the class file for `MemberName` at all. Instead the class loader has to be altered to inject a synthetic field with an integer type wide enough to fit a pointer for the current platform into that class.

For each class the loader now consults the function `MethodHandles::injected_fields` that returns an array containing the names and types of any fields that should be injected. With HotSpots class library `MemberName` is the only class for which that array is not null. The implementation of the function thus consists of an inlinable check that against the name of the class currently being loaded and a slow path if it matches that of `MemberName`. For GNU Classpath `MethodHandles::injected_fields` always returns null and the check can be removed completely.

Since CACAO uses a different implementation for invoking interface methods than HotSpot which requires two 32 bit indices the `vmindex` field is always 8 bytes wide, even on 32 bit architectures.

In HotSpots implementation of JSR-292 `MemberName` also contains another injected field called `vmtarget` that holds a `Method` or `Class` object describing the pointed to field or method. This field could actually be included directly in the class file of `MemberName`, but isn't, either for historical reasons or to allow the VM to alter its internal implementation without changing the class library. In CACAOs implementation we do not require that descriptor object and don't inject it into `MemberName`.

4.3 The JIT compiler

The main changes to the JIT compiler for JSR-292 handle the new `invokedynamic` instruction and the special signature polymorphic methods of `MethodHandle`. Besides the previously discussed characteristics signature polymorphic methods are also distinguished from normal methods in that the VM does not create a stack frame for these calls. Instead they behave like new bytecodes and CACAO also treats them as such, introducing new IR instructions to represent them. Accordingly no separate method objects are created to implement them and their machine code is directly inlined at every use, just as there are no separate method implementing, say, the `invokevirtual` instruction.

The following chapters each describe changes made to one pass of the JIT compiler required for the implementation of JSR-292.

4.3.1 Bytecode parser

As mentioned before the bytecode parser converts Java bytecode into CACAOs intermediate representation. The bytecode parser is the only pass in CACAO that is aware of bytecode, all subsequent passes work solely on ICMDs.

CACAOs IR uses a RISC like encoding where each instruction uses the same amount of space. All different intermediate instructions are represented by the C++ struct `instruction`, which consists of fields for the instruction opcode, flags, up to three operands and an indicator where the instruction stores its result.

All IR instructions for a method are allocated in one contigous array, which is very cache friendly but makes it difficult to insert instructions into a method during compilation.

Parsing of bytecode to ICMDs is not a strict one to one mapping, some bytecodes are translated into multiple IR instructions. The various dup bytecodes for example are translated into a series of move and copy instructions, which are easier to translate to register based machine code.

Since invoking method handles via `invoke` or `invokeExact` or calling linker methods via the `linkTo` methods has very different semantics than a regular call we introduce new ICMDs for all these cases, including an IR instruction for pushing the appendix argument. We of course also create a new ICMD corresponding to the `invokedynamic` instruction. Furthermore, there are distinct ICMDs for `invoke`, `invokeExact`. One with an appendix argument and one without. The reason for this seperation is explained in detail in section 4.3.3 which discusses changes to the patcher and resolver. Table 4.1 lists all new ICMDs and their purpose.

For simplicty the new ICMDs are grouped into three categories, one for loading the new constant types, one for the public JSR-292 interface (`invokedynamic`, `invoke` and `invokeExact`) and one for HotSpots private polymorphic methods.

4.3.1.1 Detecting polymorphic calls

According to the JVM specification the check whether an invoke bytecode is signature polymorphic should be done when the instruction is linked, that is when it is executed for the first

time. In a dilemma quite parallel to that of the appendix argument described in section 4.2.1.3 CACAO cannot delay detecting polymorphic calls that long.

JamVM for example uses regular C arrays to represent a methods bytecode, rewriting an `invokevirtual` instruction that invokes a handle to a specialized one during linking relatively easy. The appendix argument also presents no problem since JamVM directly implements the JVMs stack machine model so it can simply push another value onto the stack without a problem.

The HotSpot JVM does compile bytecode to machine code for faster execution but is able to fall back to an interpreter while executing a method. When a call in compiled code is detected to be signature polymorphic the VM can switch to the interpreter, link in an invoker for the call, and continue execution. If the method is executed often enough it will be recompiled to machine code and since the JIT is now aware of the polymorphic call site it can directly emit specialized code for it.

While CACAO has support for on stack replacement of currently executing methods this feature is optional, and it is desirable that the VM can be used with it disabled. If we compiled `invoke` and `invokeExact` like normal virtual invocations the instructions at the call site would load a method from the receivers vtable, but in fact these calls should have the form of an `invokestatic` to the invoker method. Furthermore the register allocator must be made aware of the appendix argument so it can assign it a register or stack slot. For all these reasons it is necessary to detect polymorphic calls before they are first executed

We choose to do this ahead of time resolution in the bytecode parser, so that no later phase of the JIT has to care about the new behaviour of the `invokevirtual` instruction.

When parsing an `invokevirtual` or `invokestatic` instruction the method `MethodHandles::parseSignaturePolymorphic` is called to handle any signature polymorphic invocations and if it fails we fall back to the normal parsing behaviour for those bytecodes. For GNU classpath we use the same inlining trick as with `MethodHandles::injected_fields` in the loader, meaning this check is optimized away completely.

When configured for HotSpots class library we can also speed up this check using the observations that all signature polymorphic method are members of `MethodHandle`. A fast pointer comparison of the class of the method being parsed against class `MethodHandle` thus rules out most normal methods. For members of `MethodHandle` a small table with descriptors for all polymorphic methods is scanned. If the name and flags of the method being called matches one in the table the invocation is polymorphic and we rewrite it to one of the new ICMDs.

4.3.1.2 The appendix argument

For some more complex bytecodes CACAO also utilizes a scheme similar to HotSpots appendix argument. A `new` bytecode for example is translated into an ICMD that pushes the class of the object to allocate and one that calls a memory allocation function implemented in C++ with that class as its only argument. That single argument contains all information necessary for the garbage collector, such as the size of an instance or if the object contains pointers to other objects. The class object thus serves essentially the same purpose as an appendix, since it allows CACAO to use a generic allocation routine for all objects, just as a method type appendix allows for generic invoker routines.

ICMD_RESOLVE_METHODHANDLE	Resolve and push a MethodHandle object constant
ICMD_RESOLVE_METHODTYPE	Resolve and push a MethodType object constant
ICMD_INVOKEHANDLE	Call the invoke or invokeExact invoker for a method handle
ICMD_INVOKEDYNAMIC	Bootstrap and/or call target handle of CallSite
ICMD_INVOKEHANDLE_WITH_APPENDIX	Like ICMD_INVOKEHANDLE with an appendix argument
ICMD_APPENDIX	Push the appendix argument for ICMD_INVOKEHANDLE_WITH_APPENDIX or ICMD_INVOKEDYNAMIC
ICMD_INVOKEBASIC	Call a method handles target (implements invokeBasic)
ICMD_LINK_TO_STATIC	Call target of a MemberName as a static method
ICMD_LINK_TO_SPECIAL	Call target of a MemberName as a private or final method
ICMD_LINK_TO_VIRTUAL	Call target of a MemberName as a virtual method
ICMD_LINK_TO_INTERFACE	Call target of a MemberName as an interface method

Table 4.1: New ICMDs for JSR-292

Since pushing an additional object onto the operand stack might cause the verifier to detect a stack overflow the inserted instruction is marked so its result does not count against the methods stack depth limit.

The main difference between CACAOs pre-existing appendix system and that introduced for JSR-292 is that CACAOs builtin functions all have a fixed stack behaviour, that is they always use an appendix of the same type. The stack analyser, verifier and register allocator thus have hard coded paths for dealing with these instructions. Each invocation of a method handle on the other hand can have a different behaviour and might or might not require an appendix argument, depending on the calls type descriptor. The stack behaviour of method calls, normal ones as well as those to method handles, is taken from a methoddesc stored in the invoking instruction

When MethodHandles::parseSignaturePolymorphic detects that a call requires an appendix argument an ICMD_APPENDIX instruction is inserted. Splitting up a handle invocation into an instruction that pushes the appendix and a call that immediately consumes it helps reduce the impact of these new instructions to the subsequent JIT passes. As a consequence the stack analyser, verifier and register allocator only require very minimal changes to accomodate the new IR instructions.

4.3.1.3 Object constants

The ldc family of bytecodes is responsible for pushing constants from the constant pool onto the operand stack. In CACAO these bytecodes are translated to different ICMDs depending on the type of constant pushed. There are distinct instructions for each primitive Java type and

ICMD_ACONST (short for address constant) for pushing the address of an object.

Previous to JSR-292 ICMD_ACONST only had to handle two different kinds of constants, String and Class objects, the flag INS_FLAG_CLASS in the current instruction was used to distinguish between the two cases. Another flag, INS_FLAG_UNRESOLVED, is used to signal that the class object to be loaded has not yet been resolved, meaning the code generator has to insert a trap instruction here.

The new MethodHandle and MethodType constants add four new cases ICMD_ACONST has to handle, one for a resolved and unresolved constant each. If we continued to use a flag based approach we would thus have to double the number of flags, leading to long if-cascades in the code handling this IR instruction. Furthermore, patched constant pool entries of anonymous classes can contain any object type adding one more case to ICMD_ACONST.

Instead of addding a great number of new flags for this instruction, the behaviour of ICMD_ACONST is split up into several distinct ICMDs. We introduce three new opcodes, one for each possible unresolved constant type (ICMD_RESOLVE_CLASS, ICMD_RESOLVE_METHODHANDLE and ICMD_RESOLVE_METHODTYPE), and change the behaviour of ICMD_ACONST so it now always pushes a fully resolved object constant and is the only variant that does not require a trap instruction.

4.3.2 Stack analysis, verifier and register allocation

All information required for a call instruction by stack analysis and register allocation in CACAO is provided by the associated methoddesc. Since we have already taken care of the special behaviour of method handle invocation and invokedynamic in the parser these passes required only very small adaptations.

4.3.3 Code generator and patcher

The code generator is the very last pass of the JIT and is responsible for translating each ICMD to its corresponding machine code. Here each IR instruction is considered on its own and no reordering or coalescing of instruction takes place, all such optimizations have been taken care of by previous passes.

Naturally this process is highly dependent on the CPU and operating system CACAO is running on, as a consequence this pass is implemented differently for each supported platform. There are a few helper functions for emitting machine code which are available on all systems and a limited number of IR instructions can be handled using only these. The code for emitting the copy instructions for arguments to a method call is also platform independent, the architecture specific part is only responsible for emitting the code for loading the target method and jumpfing to it.

Due to the JVMs lazy linking semantics CACAO cannot compile all methods ahead of time. It is thus possible to call methods which have not yet been compiled or even resolved. When emitting code for such calls the JIT leaves out the address of the function to call and inserts a trap instruction. When the call is first encountered the patcher is invoked, it is then responsible for resolving the target method and fixing the callers machine code to perform a valid method invocation. For methods that have not been compiled yet the call will then jump to a stub method

that contains yet another trap instruction, this trap then invokes the JIT compiler to compile the real target method. After the JIT has finished the patcher updates the call site once more to finally point to the compiled code. Each subsequent execution can then directly perform the call without triggering any traps.

Translation of the new ICMDs for JSR-292 use features that vary greatly between platforms and need to be implemented separately for each CPU. Currently the only platform for which this has been done is x86_64, porting to other 64 bit systems should not be very difficult though. Porting to a 32 bit architecture would require some changes to the current design regarding handles to interface methods. These will be explained in more detail later.

The main effort in implementing the JSR-292 ICMDs does not lie in translating them to machine code though. The IR instructions for pushing method handle and type constants and calls to `invokedynamic`, `invoke`, and `invokeExact`, are translated to essentially the same code as a `ldc` or `invokestatic` instruction would be. Furthermore, the machine instructions for the `linkTo` methods are variations of those for the regular invoke instructions. Only `invokeBasic` call sites are more novel and require different machine instructions.

The need to interact with HotSpots APIs for lazy resolution for these instructions required some changes to CACAOs patching infrastructure though. The C API for patching was ported to object oriented design implemented in C++ modeled after a proposal by Josef Eisl, another contributor to the CACAO VM.

The following sections describe each of the three categories of new ICMDs in more detail, highlighting similarities and differences to how pre-existing instructions are handled in CACAOs.

4.3.3.1 Loading object constants

Each compiled method in CACAO has its own data segment which contains constants and addresses of other methods called by it. An `ICMD_ACONST`, `ICMD_RESOLVE_METHODHANDLE` or `ICMD_RESOLVE_METHODTYPE` instruction can thus be translated to a single load instruction that loads an object pointer from the data segment.

Since that object has not yet been created when an `ICMD_RESOLVE_METHODHANDLE` or `ICMD_RESOLVE_METHODTYPE` is translated we additionally insert a patcher trap. In this case the slot for the object constant in the data segment is allocated but initialized to zero.

So far this does not differ from the code generated for an `ICMD_RESOLVE_CLASS`. In the patcher though, instead of looking up an object from the constant pool we have to call a Java method in HotSpots JSR-292 framework to create a `MethodHandle` or `MethodType` object.

As a consequence the bytecode generator creates a direct method handle with a `MemberName` referencing the target member. The task of resolving this member and initializing the hidden `vmtarget` and `vmindex` fields is then delegated back to the native methods `MethodHandleNatives.resolve` and `MethodHandleNatives.init` implemented by CACAO. For handles to fields the framework queries the VM for the fields offset via `MethodHandleNatives.objectFieldOffset`, the actual read or write of the field is implemented, as mentioned in section 3.1.2, via native methods in `Unsafe`. The Java API for creating method handles also uses these hooks for populating `MemberName` objects.

Once the `MethodHandle` or `MethodType` object has been created it is then patched back into the data segment, so the load instruction can fetch it when it is executed. Since the garbage collector does not scan data segments we have to make the appendices uncollectable just as was necessary with constant pool patches of anonymous classes.

4.3.3.2 `invokedynamic`, `invoke` and `invokeExact`

An `ICMD_INVOKEHANDLE` instruction without an appendix looks exactly like a static invocation of the invoker method, the only difference being where the patcher gets the target method from. Same as for resolving method handle constants this requires a call to the JSR-292 framework which will generate the required bytecode.

For invocations with an appendix the situation is more complicated. Recall that these are always expressed as an `ICMD_APPENDIX` instructions followed by `ICMD_INVOKEHANDLE_WITH_APPENDIX` or `ICMD_INVOKEDYNAMIC`. Here both the invoker method and the appendix argument for these is created by a single API call to the JSR-292 bytecode generator. This means a single patcher trap is responsible for updating both locations in the data segment. As a consequence we can't process the `ICMD_APPENDIX` and its invoke instruction separately.

Since the code generator has no concept of two subsequent IR instructions interacting the `ICMD_APPENDIX` and all necessary code is emitted when processing the `ICMD_INVOKEHANDLE_WITH_APPENDIX` or `ICMD_INVOKEDYNAMIC`.

4.3.3.3 `linkTo` methods

The ICMDs for these methods always read their target method from a `MemberName` object and thus require no patcher traps for resolution. Since they also only use basic method types where all reference types have been erased to `Object` delayed type checking or lazy loading of classes is also never necessary.

Remember that when the code generators architecture specific code reaches one of these IR instructions registers have already been allocated and the parameters for the function call have already been loaded into registers or pushed onto the stack. Since the call has already been set up only the three scratch registers set aside by CACAO are available for loading the target methods code and jumping to it.

As an additional restriction the patcher expects all calls to be made by loading the target address into a register and then jumping to the contents of that register. It is also of importance which of those three registers is used since the patcher for uncompiled methods examines the call sites machine code in order to decide where it should store the address of the target methods machine code. Because two of the three scratch registers are already reserved for regular method invocations only the register `%rax` can be used for direct method handle invocations.

Since CACAO implements the four method invocation bytecodes differently than HotSpot the fields `vmindex` and `vmtarget` are also used differently. When invoking handles to static methods for example HotSpot stores nothing in `vmindex` and instead uses the `Method` object in `vmtarget` to get the target methods address. In CACAO we instead directly store the address of the static methods machine code in `vmtarget` which means we can fetch with a single load

```
;; invokestatic
mov    -0x2c(%rip), %r10  ; load code from data segment (at offset 0x2c)
callq  *%r10              ; call method

;; linkToStatic
mov    0x40(%rdi), %rax   ; load vmindex field of MemberName (at offset 0x40)
callq  *%rax              ; call method
```

Figure 4.4: Difference between machine code for `invokestatic` and `linkToStatic`

instruction. The downside of this is that we have to patch the `vmtarget` field of a `MemberName` if it points to a compiler stub or if the target method has been recompiled.

Since virtual methods can be invoked both via `invokeinterface` and `invokespecial` we have to store both a pointer to the methods code and the methods virtual table index in its `MemberName`. Consequently `vmtarget` always holds the code of the method, or its compiler stub, and `vmindex` holds the virtual table index or interface table indices.

Because this representation allows a `MemberName` to be invocable via either `linkToVirtual` and `linkToInterface`, but not both, we have to take care when creating a handle to one of the public methods of `Object`, since these can legally be invoked both ways. As a solution we alter some internal flags used by HotSpot forcing it to always invoke these methods via `linkToVirtual`.

JamVM on the other hand only uses the `vmtarget` field and does not inject `vmindex`. Here `vmtarget` contains the JamVM equivalent of CACAOs `methodinfo`, the actual invocation then uses the same interpreter functions as reflection would.

Besides the register used for the actual jump the machine code generated for calls to the `linkTo` methods is only a variation of that emitted for the four traditional method invocation bytecodes of the JVM. The only difference being that the address or virtual table index of the target method is not a constant loaded from the data segment or encoded as an immediate value but can change each time the instruction is executed and must thus be loaded from a `MemberName` object.

Example code generated for an `invokestatic` instruction and a `linkToStatic` call are shown in figure 4.4. Note that this depicts the case where the `MemberName` argument resides in register `%rdi`, if it is stored on the stack we also have to emit code for loading it first.

The code for `linkToSpecial` looks exactly the same as for `linkToStatic`, adding only a null pointer check for the first argument of the call.

`linkToVirtual` is the linker method whoose machine code looks most alike to that generated by HotSpot. Here the target address is loaded from the receivers virtual table, the only difference to `invokevirtual` is that the table index is read from `vmindex` instead of being embedded directly in the machine code.

Invocation of interface methods in CACAO[6] uses two 32 bit indices, one for locating the interface method table for a given interface in the vtable, and another one for indexing into that interface method table. On a 64 bit architecture like x86_64 this makes no difference since

vmindex is 8 bytes wide anyways. For 32 bit architectures this forces a waste of 4 bytes per MemberName object.

Even though the machine code for the linkTo methods is very similar to that of the invoke bytecodes the patching required if a previously uncompiled method differs greatly. Since compilation does not invalidate virtual table or interface table indices patching is not strictly necessary for linkTo or invokeBasic calls. Without it every invocation would hit a compiler trap. Though the compiler is intelligent enough to not needlessly recompile the target method invoking such a stub involves raising a processor signal and is orders of magnitude more expensive than a direct call.

Instead of patching the data segment or machine code of the call site we have to patch the MemberName argument of the call and the virtual or interface table of the receiver argument. For regular invocations patching is relatively simple since the address of the data segment slot or instruction to update is directly encoded in the instructions of the call site. Polymorphic invocations on the other hand don't encode the register or stack slot containing which holds the MemberName. Furthermore, for the actual call of the target method the MemberName argument must be discarded since the called code does not expect it. Because the x86_64 calling conventions allows invoking a function with more arguments than it actually uses the additional parameter is not actually removed but just ignored. However, since the target method does not use the MemberName its methoddesc does record where that parameter is stored, and thus this information is also not available in the patcher. To recover this information a new function in the register allocator was introduced which computes the location of the additional argument based on the known assignment of all other parameters. Once the location of the MemberName pointer is known retrieving it is straightforward, since CACAOs signal handler records the contents of all registers, including the stack pointer, at the point where the compiler stub was called.

What remains is to update the code pointer in the vmtarget field to the entry point of the freshly compiled method and patching the virtual or interface table of the owner of the method.

4.3.3.4 invokeBasic

invokeBasic is the only linker method where the MemberName object is not passed as an additional parameter, here it must be loaded from the MethodHandle receiver argument. Figure 4.5 shows the memory layout of this MethodHandle and its nested objects which must be traversed for every call. The diagram also shows all different possible contents of the vmindex field, but in the case of invokeBasic it is always zeroed out since static method have virtual or interface table index.

Same as with the linkTo methods an invokeBasic call can trigger compilation which requires patching of the handles MemberName. Since invokeBasic also has to use the same register, %rax, for its callq the patcher also has to inspect more instructions to distinguish this case. Unfortunately the last two instructions of an invokeBasic call look exactly the same as those of an linkToStatic where the MemberName argument was loaded from the stack. Consequently we need to examine three instruction to correctly distinguish an invokeBasic.

Locating the MemberName of the call, though, is easier than with the other linker methods since the receiver argument is always located in register %rdi. For invokeBasic that MemberName always points to a static method and we thus only have to update its vmtarget.

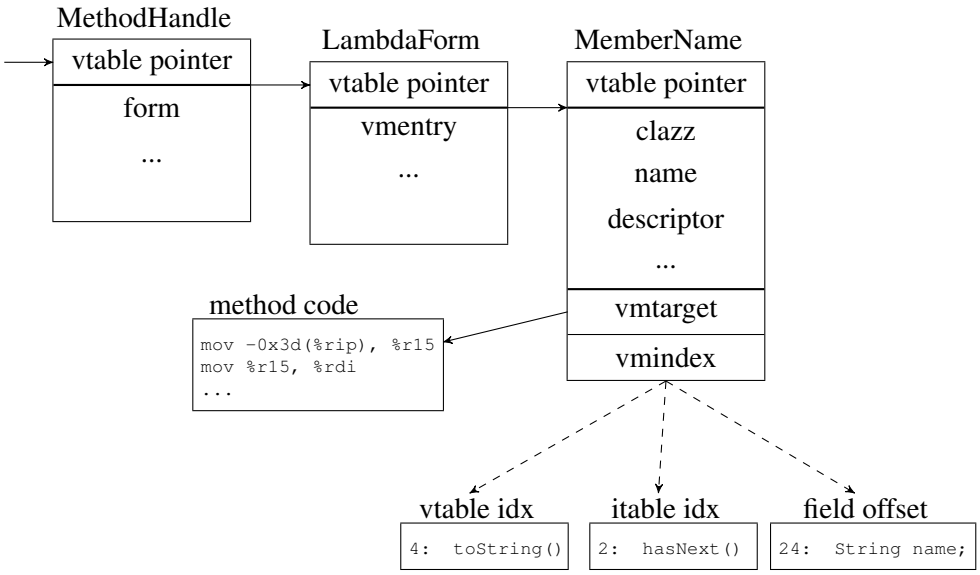


Figure 4.5: Memory layout of a `MethodHandle` in CACAO

4.3.3.5 The lambda form interpreter

As mentioned in section 3.1.3 HotSpots JSR-292 framework uses an interpreter for executing lambda forms before they are compiled to bytecode. To perform this task the interpreter uses a small number of method handles with pre compiled lambda forms, among those handles to the private signature polymorphic methods of class `MethodHandle`.

Since it is possible to create handles to these methods we can no longer guarantee that all invocations of these methods can be inlined at their call site. Fortunately these handles are not required to also behave like a signature polymorphic method, instead they are fixed to a given method type. For such handles we create a small stub method containing nearly the same machine code as that emitted when inlining the call. The only difference is that instead of invoking the target method via `callq` and then returning with `ret` sequence these stubs perform a tail call via the `jmpq` instruction, which means we never have to allocate a stack frame for them.

HotSpots stack inspection logic assumes that signature polymorphic methods never appear in the call stack, since we use `jumpq` this happens automatically and CACAOs stack walking routines don't have to be altered. Furthermore, since the `jmpq` and `callq` differ only in 4 bits it is trivial to extend the patcher to also detect these calls.

The patching logic for `linkToVirtual` call sites becomes more complicated since the target of the invocation can be either a regular Java method or a stub function for `invokeBasic` and both cases require different patching behaviour. For a Java method we must update the methods owners virtual table, for `invokeBasic` the method handle receiver must be patched. To distinguish the two cases we ensure that there is only one stub function for `invokeBasic`, the patcher

can then check if the `vmtarget` of the `MemberName` of the current call contains that pointer.

Note that it is also possible, even for unprivileged user code, to obtain handles to `invoke` and `invokeExact`. This requires some additional resolution logic in CACAO but no stub methods have to be generated since HotSpot creates customized bytecode with an inlinable call.

4.3.3.6 Optimizing `invokeExact`

Remember that for every `invokeExact` call site the JSR-292 generates a method that checks the incoming `MethodHandle` s method type against that of the call site, and then invokes the handle via `invokeBasic`. Since method type objects are interned the type check only consists of a pointer equality check.

To speed up these invocations we force inlining of these invoker methods at every call site even when CACAOs inliner is disabled. As a consequence the type check boils down to three instructions and `invokeExact` calls are sped up by a factor of more than two. Lazy verification of the call site and creation of the `MethodType` appendix object still require a patcher trap though.

The machine code for creating and raising an exception object in case of a failed type check is several times larger than that for the actual call. Since we expected that in a correct program the type check for `invokeExact` will succeed most of the time we implement the uncommon case with a single trap instruction. That trap gives control back to CACAOs signal handler which is then responsible for raising and handling the exception.

CHAPTER 5

Evaluation

The JSR-292 implementation for CACAO is evaluated on three different criteria. First we confirm that it follows the rules of the JVM specification. Next we run a number of microbenchmarks to establish the raw exection time for method handle invocation and `invokedynamic`. Lastly a number of JRuby benchmarks are run to determine the performance in a more realistic setting.

Kaewkasi[28] proposed a different way measuring the performance of JSR-292. They rewrote all method invocations in SciMark, a benchmark suite for scientific computing in Java, to use `invokedynamic` and method handles. Unfortunately their work uses an older version of `invokedynamic`, which has different semantics from those specified in JSR-292 and is not compatible with CACAOs implementation.

5.1 Methodology

All benchmarks were run on an Intel® Core™ i7-M620 2.67GHz processor with Ubuntu linux with kernel version 3.11.

CACAO was configured for a default release build with statistic gathering enabled. When measuring class loading and compilation time the real-time timing subsystem was also enabled.

For HotSpot a standard release configuration of version 1.7.0_55 build 24.51-b03 was used. JamVM was also compiled for with default options, plus those to enable JSR-292 support.

5.2 Standards conformance

The OpenJDK project contains an extensive test suite for the features of JSR-292. With these and another smaller set of tests tailored to CACAO the behaviour of its JSR-292 implementation was validated to conform to the specification.

5.2.1 The Indify tool

As mentioned before the `javac` compiler cannot be used to emit class files using the `invokedynamic` instruction or containing `MethodHandle` or `MethodType` constants.

To be able to test all features of JSR-292 the OpenJDK test suite uses the Indify tool[48] written by John Rose. This tool in a way performs the opposite of what Foraxs JSR-292 backport does, i.e, it rewrites class files converting calls of the Java method handle API to directly use the new bytecode and constant pool types.

Traditionally CACAO has used the jasmin bytecode assembler for creating class files for testing, but unfortunately that tool does not have mature support for the newer class file format features. As a consequence we also use Indify for processing our own tests.

5.2.2 Results of the OpenJDK test suite

The OpenJDK test suite consists of about 120 JUnit tests, but since some of these tests actually generate new and randomized tests at runtime the suite exhaustively covers the required behaviour of the JVM specification.

Of these 120 tests four fail when run with CACAO, all of these failures are caused by problems not related to JSR-292. Two tests trigger a previously unknown problem in the JIT code for the handling of debug line number information and one hits a problem on the bytecode verifier. The fourth test checks if call stack overflow detection works with method handles, it fails because CACAO currently does not check for this.

Besides these tests the suite also contains a number of regression tests for bugs in HotSpots implementation of JSR-292, CACAO passes all of these tests.

5.3 Measuring invocations

To measure the time individual method handle invocations take we created a set of microbenchmarks. The benchmarks invoke the four kinds of methods on the JVM (static, final, virtual and interface) directly, via reflection, method handles and `invokedynamic`.

Each benchmark run lasts for five seconds and counts the number of invocations that were performed in that time. From this we can estimate the time required for a single call. All benchmarks are run ten times and the final results are the mean of the measured timings.

Each benchmark also contains a warmup phase which performs the method invocation a few million times to ensure all code has been compiled when the actual measurements are done.

Since we wanted to measure the performance of method invocation the target methods only return a constant value. This of course makes these methods a perfect target for inlining and other optimizations, which would make the benchmarks pointless since it is the actual invocation we want to measure.

CACAO for example detects monomorphic virtual methods and performs no virtual table lookup when they are called. HotSpot even speculatively inlines virtual methods with multiple overrides at monomorphic call sites. To prevent the different JVMs from optimizing away the virtual and interface calls the target methods were overriden in several methods and a list of receveiver arguments was cycled through. Since HotSpot not only speculatively inlines at

monomorphic but also at bimorphic call sites at least three different implementations are necessary for each method. For the static and final method this approach does not work since they are always monomorphic, here we employ special command line flags such as HotSpots `-XX:CompileCommand:noinline`.

5.3.0.1 Direct and reflective invocation

To give a frame of reference for the following results timings for direct and reflective method invocation are included in figure 5.1. The `invokestatic` benchmark, for example, performs around two billion direct calls and six million reflective calls in five seconds.

Static and special invocations in HotSpot are marginally faster than in CACAO since they are compiled to direct jumps while CACAO always jumps through a register. Virtual calls are implemented essentially the same in HotSpot and CACAO, interface invocations in HotSpot on the other hand use a linear search. Both virtual and interface calls in HotSpot are sped up with an inline cache, but because the targets of those caches are regularly invalidated in the benchmarks CACAO is slightly faster here. Method invocation in JamVM is consistently slower by an order of magnitude.

For reflective invocation HotSpot greatly outperforms both CACAO and JamVM since its implementation does not require a transition to native code for every call. In the benchmarks for reflective invocation the interface method is the only target that requires no access check, consequently these calls are much faster. When factoring out the boxing overhead reflective interface calls on HotSpot are as fast as direct calls. Note that in the current results we prevent inlining of `Method.invoke`, otherwise HotSpot even optimizes away the array boxing of arguments and the reflective invocations would be inlined.

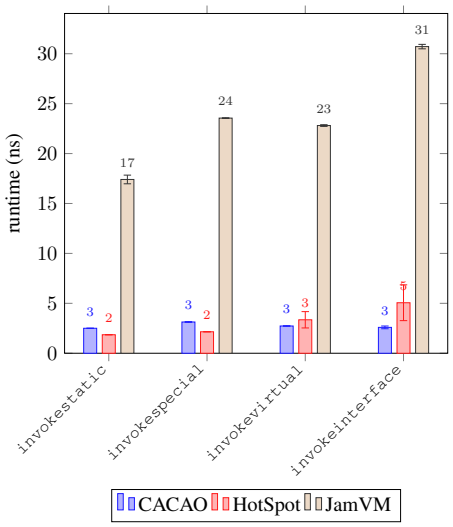
As witnessed by figure 5.1c the stack based access check slows down reflective invocation in CACAO by a factor of two. In HotSpot unchecked execution is even faster since it does not require a transition into VM code. In the `invokeinterface` case the access check can be omitted since interface methods are always public.

In the first release of OpenJDK that contained support for JSR-292 invocation of methods via method handles was reportedly slower than invocation via reflection[18]. This was due to the fact that in that first iteration method handle invocation was implemented with chains of assembly stubs that were opaque to the JIT and could not be inlined or optimized in any other way. Reflection, on the other hand, has been sped up by generating customized bytecode since version 1.4 of HotSpot, similarly to the way JSR-292 is implemented now. In the current release `invoke` is slightly faster than reflection and `invokeExact` is nearly as fast as a direct call.

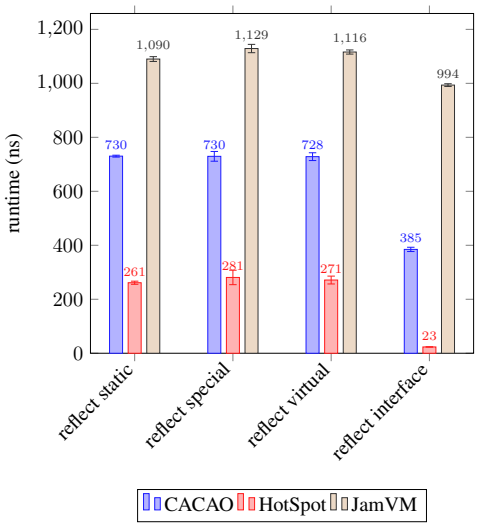
CACAO currently does not optimize reflective accesses, instead they are implemented via a native method that does some validation and forwards to a platform dependent assembly stub for the actual invocation. As a consequence reflective method invocation is much slower on it compared to HotSpot. Method handle invocation via `invoke` on CACAO is still slower than reflection but `invokeExact` outperforms it by two orders of magnitude.

5.3.0.2 Method handle invocation

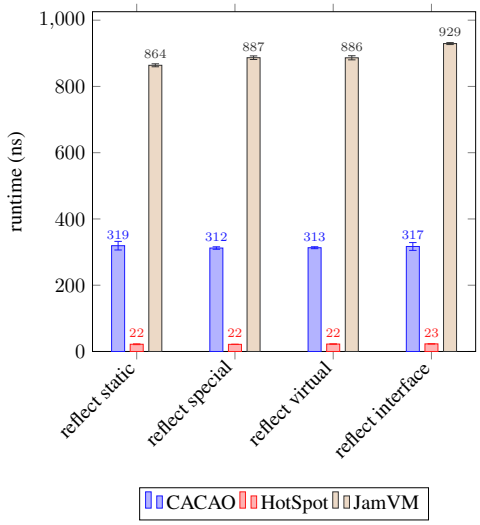
The results for method handle invocation via `invoke` and `invokeExact` can be seen in figure 5.2.



(a) Direct invocation



(b) Reflective invocation



(c) Reflective invocation without access checks

Figure 5.1: Direct and reflective invocation

Since both CACAO and HotSpot inline `invokeExact` calls at their call site this way of invoking a method handle has very little overhead. For direct method handles the type check and indirection leads to a factor of two slowdown compared calls via the `invoke` bytecodes. The type check is actually very cheap compared to the cost of the argument shuffling introduced by method handles. When it is disabled the micro benchmarks for direct invocation are sped up by 7%, but in the JRuby benchmarks there is no measurable improvement.

`invoke` on the other hand requires much more elaborate type checking and is significantly slower, even slower than reflection. The cause of the slowdown is that every `invoke` call has to look up, or if necessary create, a combinator handle that performs any required argument conversions. This look up involves `MemberName` resolution and thus requires a slow JNI call to a native method. Execution of the combinator also always uses the lambda form interpreter. The most used internal methods of the interpreter have a special annotation which instructs HotSpot to always inline them, as a consequence the interpreter and thus `invoke` calls are much faster with that VM.

In addition to invocation of regular Java methods the benchmark for method handles also measures timings for invoking method handle combinators. The first combinator benchmark, `dropArgument`, calls a handle to the static method that adds two `long`s wrapped in the `dropArguments` combinator. The second combinator, `dropAndFilter`, additionally filters the arguments of the addition method with two handles that add or substract one from the argument `long`. The last combinator, `addFive`, binds the first argument of the addition method to `5`, producing a handle that adds five to any passed argument. As can be seen in figure 5.2 the cost of invoking combinators on CACAO is simply the sum of the costs of the invdividual handle calls inside the combinator.

For comparison we also manually implement methods equivalent to these combinators. Since the bytecode for combinators always involves one more level of indirection compared to direct handles the direct versions are marginally faster for both CACAO and HotSpot.

5.3.1 JSR-292 backport

We also evaluate Foraxs backport implementation of JSR-292 when running with CACAO. Since this implementation mostly uses reflection for method handles the direct JSR-292 implementation outperforms it by two orders of magnitude.

The results for these benchmarks are listed in figure 5.3. We do not compare performance for method handle calls via `invoke` since the backport implementation is actually not correct since it does not perform any argument conversions. The `invokespecial` benchmark is also omitted since it can't be run with the Foraxs backport. The reason for this is that reflection cannot faithfully reproduce the semantics of a handle to `REF_invokeSpecial` and only supports the case where such a handle would behave exactly like a `REF_invokeVirtual`.

Also note that JSR-292 backport disables access checks for reflection when possible. In some environments disabling these security checks is prohibited which means the backport would be slowed down by a factor of two.

Furthermore note that interface calls are faster with the backport implementation since for public methods an invoker class akin to what HotSpots framework produces is generated to avoid the overhead of reflection.

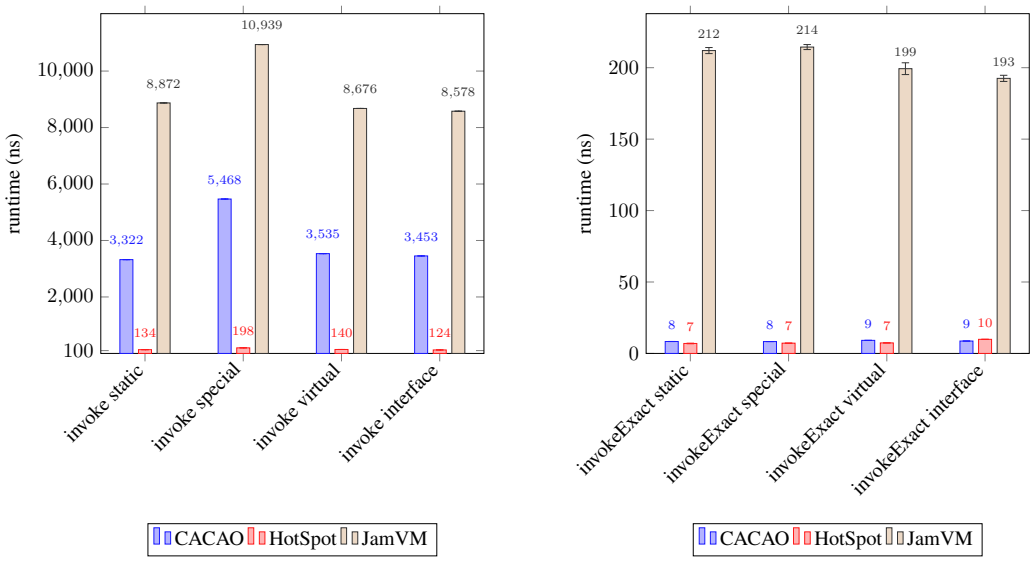


Figure 5.2: Invocation via method handles

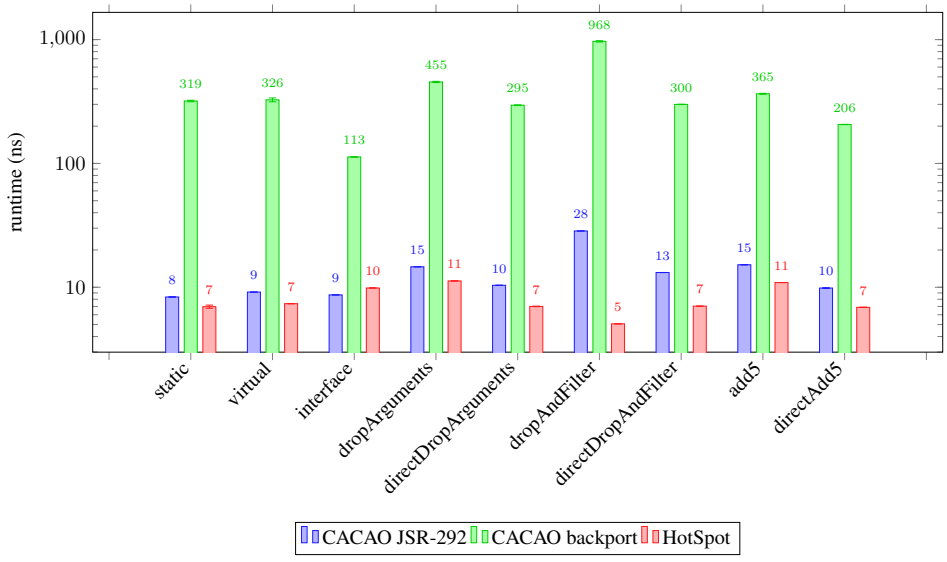


Figure 5.3: Comparison with JSR-292 backport

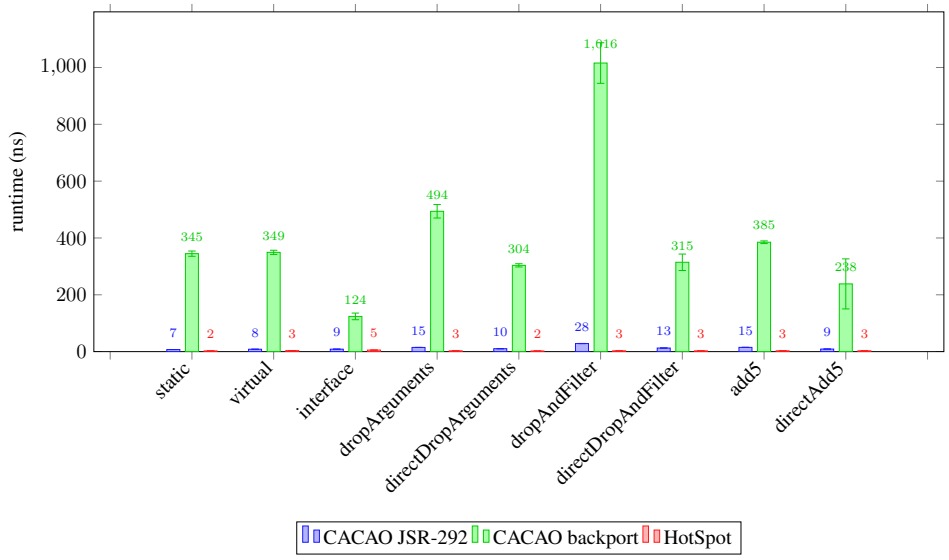


Figure 5.4: Invocation via `invokedynamic`

5.3.1.1 `invokedynamic`

The benchmarks for the `invokedynamic` instruction call the same methods as those for method handle invocation, but here the target handles are not local variables on the stack but linked in via the `invokedynamic` bootstrapping mechanism. Figure 5.4 shows the measurements obtained from these benchmarks.

The results of these benchmarks are not surprising and the time required for an `invokedynamic` call in CACAO is exactly that for the static call to the linker method and the `invokeBasic` for the target handle.

HotSpot on the other hand assumes that the target handle of the `invokedynamic` call site does not change often and compiles an optimized version that does a direct call to the handles target method. There is no check whether the call sites target has changed in the compiled code, instead the `setTarget` method of the `CallSite` object invalidates all methods that use it. As a consequence HotSpots implementation of `invokedynamic` is as fast as a direct call.

We also ran a suite of benchmarks where on every thousandth `invokedynamic` call the target handle of the `CallSite` is changed to a different `MethodHandle` with equivalent behaviour. This only marginally changes the timing results for CACAO, but causes a twentyfold slow down for HotSpot, since each invalidation triggers a recompilation of the benchmark method.

Call site invalidation does not slow down the JSR-202 backport implementation, since it does not trigger any recompilation, but it is again two orders of magnitudes slower overall.

5.3.2 JRuby

We tested the performance of CACAOs JSR-292 implementation with a number of benchmarks from JRubies own benchmark suite[40] and from the Ruby Benchmark Suite[7].

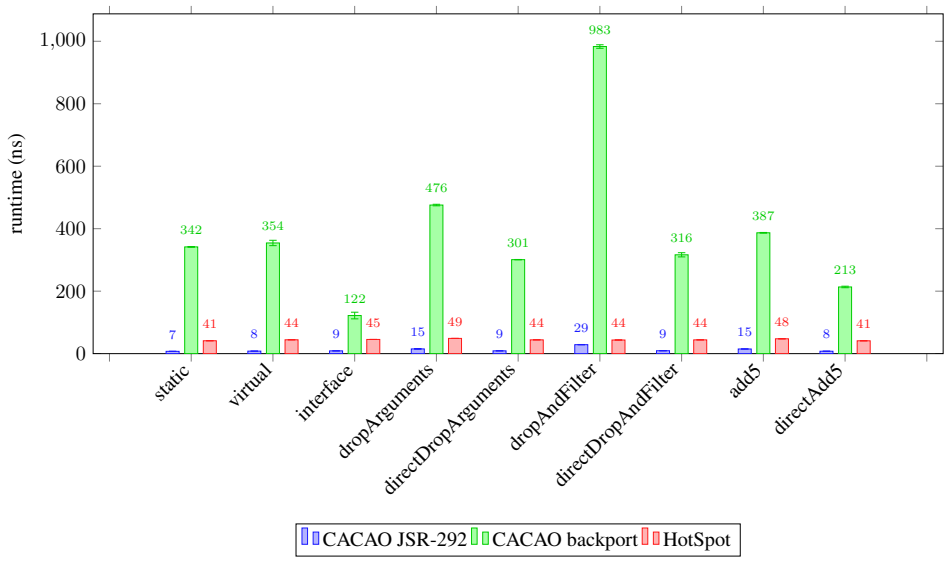


Figure 5.5: `invokedynamic` with call site invalidation

As explained in section 1.2.2 JRuby can be used both with and without JSR-292, allowing for a direct performance comparison between the two approaches.

For these benchmarks the official release version 1.7.12 of JRuby was used, all of them were run on CACAO and HotSpot. Whether JRuby emitted bytecode using JSR-292 was controlled via the `invokedynamic.compile` command line flag.

5.3.2.1 New builtin functions

JRuby without `invokedynamic` support enabled compiles calls to well known methods, such as those for arithmetic, in roughly the following way:

```
if (a instanceof RubyFloat && b instanceof RubyFloat)
  result = JRuby.addFloat(a, b);         // fast static call
else
  result = a.getMethod('+').call(a, b); // slow dynamic call
```

Since CACAO has a reliably fast implementation of subtype checking[47] this does really speed up calls to these methods.

With JSR-292 enabled the `instanceof` bytecodes are replaced with calls to the method `Class.isInstance`. As a consequence the number of calls to this method in our JRuby benchmark suite has increased from a few millions to multiple billions. Since `isInstance` is a native method it requires an expensive transition to native code. To speed up JRubies method call idiom we promote `Class.isInstance` to a builtin function, meaning the JIT can emit specialized code for it.

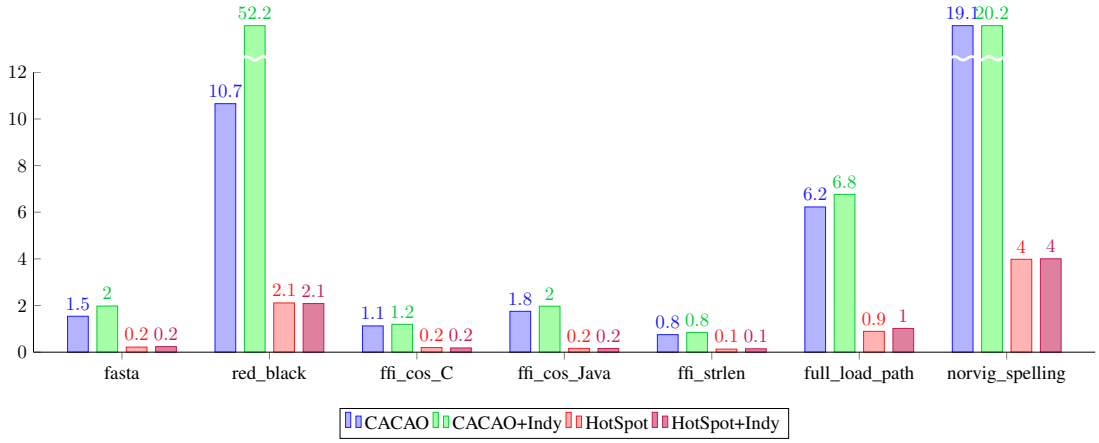


Figure 5.6: JRuby with and without JSR-292

We observed that roughly 90% percent of all calls to `Class.isInstance` in our JRuby benchmarks do not require a full subtype check, instead a fast pointer comparison of the expected class and the arguments actual type suffices. Exploiting this, calls to `Class.isInstance` are now translated to the following code:

```
if (class != obj.class)
  result = isInstance(class, obj); // slow function call
else
  result = true;                   // fast path
```

Here `isInstance` is a C function which performs a full flegded subtype check taking interface and array types into account. For most calls no function call, native or Java, is required at all.

With this optimization the execution time of the `bench_red_black` benchmark drops from 79 seconds to 58.

The number of calls to `Object.getClass` also increase greatly with JSR-292 enabled, it is thus also promoted to a builtin function and inlined wherever possible.

5.3.2.2 Benchmark results

Since every method handle invocation and `invokedynamic` calls are more expensive than invocations via the other `invoke` bytecodes method calls in JRuby actually become slower on CACAO with JSR-292.

For arithmetic and other fast operations where the method call is much more expensive than the operation actually performed this slow down is quite large, as can be seen in the `red_black` benchmark. FFI calls to Java or native code on the other hand, which are already quite expensive, are not substantially slower, as witnessed by the `ffi_cos_Java`, `ffi_cos_C` and `ffi_strlen` benchmarks.

5.3.2.3 Class loading and compilation time

All previously mentioned benchmarks only measured execution time for a fully loaded and jitted program. To determine the effects of JSR-292 on class loading and compilation time we additionally run the JRuby benchmark suite with CACAOs real-time timing support enabled. The results of these measurements are displayed in figure 5.7.

On average 902 additional classes are loaded when JSR-292 support is enabled in JRuby, 732 of those are anonymous invoker classes, 28 array classes, and 57 are from the JSR-292 support code. Since these classes are all very small and mostly contain a single short method the overall time spent in the class loader increases only by 0.5%.

Even though the total number of compiled methods increases by 12% the overall time spent in the JIT decreases by 58%. The amount of memory used for machine code increases by 9.4% and that for data segments by 12.9%.

Note that the time spent in the heap allocator is nearly doubled when JSR-292 is enabled while the maximum memory usage only increases by 10%.

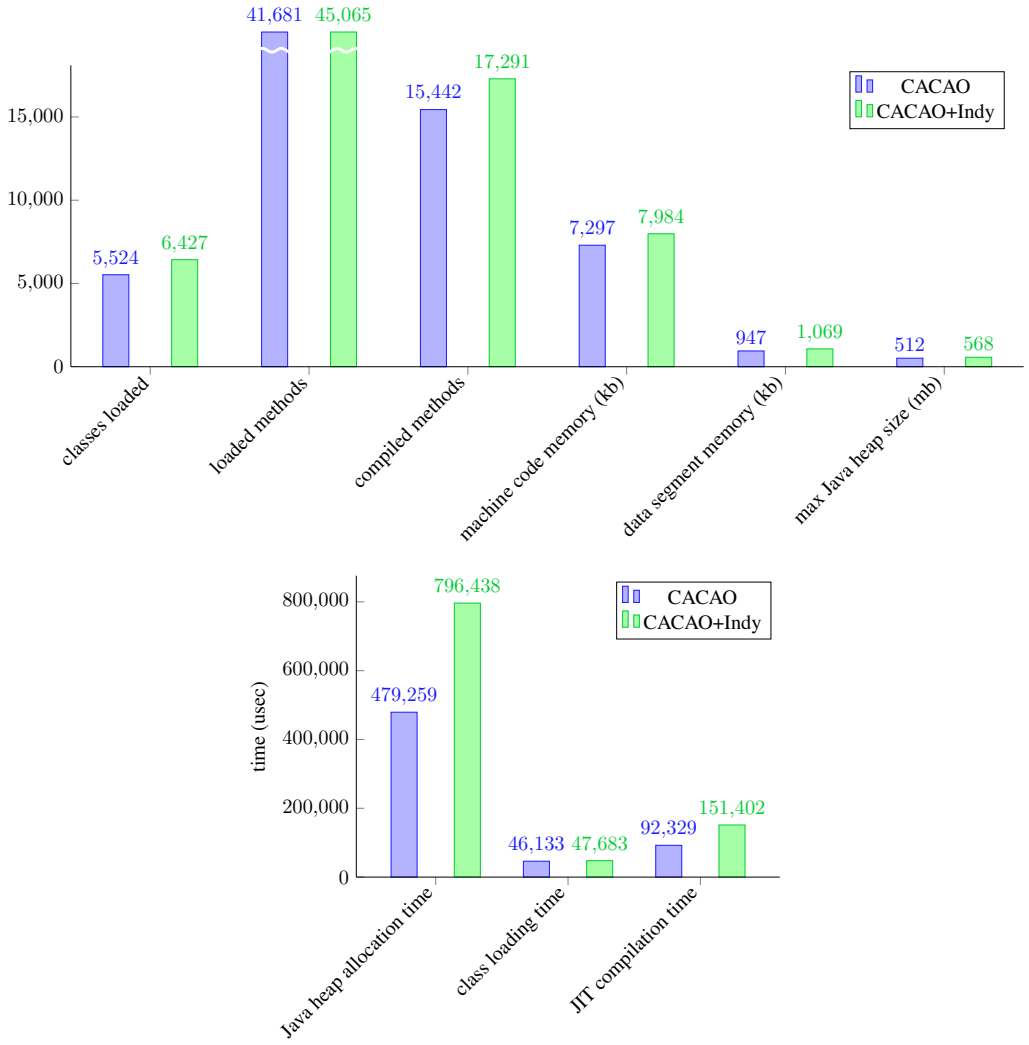


Figure 5.7: Class loading and JIT compilation statistics

CHAPTER 6

Conclusion

Around the time work on this thesis was started the second iteration of JSR-292 for the HotSpot VM containing the bytecode generator framework was released. The decision was made to use this framework since it was believed this would substantially reduce the amount of work needed to implement JSR-292 on CACAO. In retrospect this decision turned out to be a double-edged sword since the differences between HotSpot and CACAO meant that a large number of APIs had to be ported or reverse engineered. The architecture of the bytecode generator framework furthermore requires that nearly all features of JSR-292 are implemented to even initialize it at runtime. As a consequence, for a large part of the development time not even toy examples could be executed. On the other hand, once it was possible to start the bytecode generator it took care of many corner cases in the JSR-292 specification and surprisingly little work was necessary to make sure that CACAO conformed to the standard.

6.0.3 Initialization of HotSpots JSR-292 framework

From the perspective of the VM HotSpots bytecode generation framework is encapsulated in the class `MethodHandleNatives`, which contains all relevant native methods used by the bytecode generator.

On the Java side the framework uses several intern tables and other global data structures which are spread out over the `java.lang.invoke` package and constructed via their corresponding classes static initializer methods. Due to Java's lazy class initialization semantics this ensures that all globals used by the framework are created when the API is used. Another consequence of this is that any program that does not use any feature of JSR-292 does not incur the overhead in start-up time of initializing the framework.

Unfortunately the current version of the code is relatively brittle since the individual initializers are highly interdependent. The code only works if the initializers are being run in the correct order. In order to successfully initialize the framework with CACAO a small patch to the OpenJDK class library is required.

The patch moves the initialization of a static `String` array in `MethodHandleNatives` to the top of the file. When a class is first linked all static fields are set to null and then all static initializers are run in the order in which they appear in the Java source file. Moving the array to the top ensures that it is created before any code can access it. Since it furthermore does not rely on any other global variable this does not create any other start up problems.

Without this patch the array is used from a static method before it is properly created which leads to a null pointer exception. Several attempts were made to reverse engineer the proper time and place to initialize `MethodHandleNatives` but in the end it was easier to patch OpenJDK.

This problem does not appear to occur if the JSR-292 framework is initialized eagerly during start-up of the VM. Unfortunately this exposes another problem since a static initializer in the private implementation class `sun.invoke.util.VerifyType` tries to load the class `java.lang.Null`, which was at one point supposed to be standardized with version 7 of OpenJDK, but was never added to the class library. The initializer in `VerifyType` does catch and ignore the `ClassNotFoundException` raised in the course of attempting to load a non-existent class, but since CACAO aborts whenever any exception is raised during start-up that exception handler is never even reached. Furthermore, eager initialization of the JSR-292 framework forces an unnecessary overhead onto all programs that do not wish to use its features.

6.1 Future work

6.1.1 Second stage compiler

In his masters thesis Josef Eisl has implemented an optimizing second stage compiler for the CACAO JVM. This new compiler recompiles hot methods that are executed frequently with more sophisticated optimizations than those provided by the baseline compiler[15]. The second stage compiler features a new SSA based intermediate representation which is much more targeted for ease of use and maintainability than that of the baseline compiler.

Currently the second stage compiler is not as mature and well tested as the baseline compiler and does not yet implement the complete bytecode instruction set. Once work on the compiler has progressed further the JSR-292 support can be ported to it and optimizations that are hard or impossible to implement on top of the old IR can be tackled. Even so it still remains desirable that the baseline compiler be able to handle bytecode using JSR-292.

6.1.2 Inlining

Adaptive inlining for the CACAO JVM has been implemented by Thalinger and Steiner[55], currently the target method of a method handle invocation is never inlined. The two most important targets for inlining are calls to method handle constants and inlining of the target of an `invokedynamic` instruction. Constant method handles are a lucrative target since they are used extensively in the bytecode generated for lambda forms. That this is also the case for `invokedynamic` calls is evidenced by the evaluation of this thesis where HotSpot outperforms CACAO by a large margin.

The problem with inlining calls to constant handles is that the instruction that loads the handle can be arbitrarily far away from the actual call. Unfortunately CACAOs IR does neither

maintain the source instruction for values nor whether a given variable or stack slot contains a constant. Detecting if a call site is eligible for inlining would require a separate analysis pass. Since the second stage compilers SSA based IR directly contains this information this optimization would be relatively easy to implement. We thus postpone implementing inlining for JSR-292 until the second stage compiler is ready to support it.

6.1.3 Reclaiming code memory

`MemberName` and `Method` objects have the unique property that they can point to code memory. Currently CACAO never frees code memory, but when code memory reclamation is added, these code pointers have to be respected. The Boehm GC does provide custom marking routines, which can be used for scanning objects for pointers to code. Unfortunately these have to be specified when memory is allocated and as has been mentioned before, `MemberName` objects are allocated by Java code not by C++ code under our control. One solution would be to add special logic to `ICMD_NEW` (the ICMD for allocating objects), to detect the special case of allocating a `MemberName` and use a special allocation function.

APPENDIX A

Source Code Reference

The official source code repository for CACAO can be found at:

`http://www.cacaojvm.org`

Development of CACAO is done publicily in a Mercurial repository at:

`https://bitbucket.org/cacaovm/cacao-staging`

The source code for the JSR 292 implementation based on OpenJDKs bytecode generator framework can be found at:

`https://bitbucket.org/faderAvader/cacao-staging`

The source code for all benchmarks used in the evaluation can be fount at:

`https://bitbucket.org/faderAvader/cacao-JSR292-bench`

Bibliography

[1] J. R. Allen et al. “Conversion of Control Dependence to Data Dependence”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’83. Austin, Texas: ACM, 1983, pp. 177–189. DOI: 10.1145/567067.567085.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. “Detecting Equality of Variables in Programs”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’88. San Diego, California, USA: ACM, 1988, pp. 1–11. DOI: 10.1145/73560.73561.

[3] Bard Bloom et al. “Thorn: Robust, Concurrent, Extensible Scripting on the JVM”. In: *SIGPLAN Not.* 44.10 (Oct. 2009), pp. 117–136. DOI: 10.1145/1639949.1640098.

[4] Hans Boehm, Alan Demers, and Mark Weiser. *A garbage collector for C and C++*. URL: http://www.hpl.hp.com/personal/Hans_Boehm/gc/ (visited on 12/11/2013).

[5] Hans-Juergen Boehm and Mark Weiser. “Garbage Collection in an Uncooperative Environment”. In: *Softw. Pract. Exper.* 18.9 (Sept. 1988), pp. 807–820. ISSN: 0038-0644. DOI: 10.1002/spe.4380180902.

[6] Manfred Brockhaus, Anton Ertl, and Andreas Krall. *Weiterführender Übersetzerbau, Institut für Computersprachen. Arbeitsbereich für Programmiersprachen und Übersetzerbau.* Vienna University of Technology, 2009.

[7] Antonio Cangiano. *The Ruby Benchmark Suite*. URL: https://github.com/acangiano/ruby-benchmark-suite (visited on 07/22/2014).

[8] Alessandro Coglio. “Improving the official specification of Java bytecode verification”. In: *Concurrency and Computation: Practice and Experience* 15.2 (2003), pp. 155–179. ISSN: 1532-0634. DOI: 10.1002/cpe.714.

[9] OpenJDK Community. *Graal Project*. URL: http://openjdk.java.net/projects/graal/ (visited on 03/27/2014).

[10] *Da Vinci Machine project*. URL: http://openjdk.java.net/projects/mlvm/ (visited on 11/03/2013).

[11] *Documentation for java.lang.invoke.MethodHandle*. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandle.html (visited on 03/25/2014).

[12] *Documentation for java.lang.invoke.MethodType*. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodType.html (visited on 03/25/2014).

[13] *Documentation for java.lang.reflect.Method*. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Method.html (visited on 03/25/2014).

[14] Gilles Duboscq et al. "Graal IR: An extensible declarative intermediate representation". In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 2013.

[15] Josef Eisl. "Optimization Framework for the CACAO VM". MA thesis. Vienna University of Technology, 2013.

[16] Martin Anton Ertl, Christian Thalinger, and Andreas Krall. "Superinstructions and Replication in the Cacao JVM interpreter". In: *Journal of .NET Technologies* Vol. 4.ISBN 80-86943-13-5 (2006), pp. 25–32.

[17] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *SIGPLAN Not.* 28.6 (June 1993), pp. 237–247. DOI: 10.1145/173262.155113.

[18] Remi Forax. *Invokedynamic slower than reflection?* URL: http://mail.openjdk.java.net/pipermail/mlvm-dev/2010-June/001768.html.

[19] Rémi Forax. *JSR 292 Backport*. URL: http://code.google.com/p/jvm-language-runtime/ (visited on 03/27/2014).

[20] Rémi Forax. "JSR 292 Backport (first year report)". In: *Java Language Summit*. 2009. URL: http://wiki.jvmlangsummit.com/JSR_292_Backport_Deep_Dive.

[21] Yoshihiko Futamura. "Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler". In: *Higher Order Symbol. Comput.* 12.4 (Dec. 1999), pp. 381–391. DOI: 10.1023/A:1010095604496.

[22] *GNU Classpath*. URL: https://www.gnu.org/software/classpath/ (visited on 11/03/2013).

[23] Brian Goetz. *Lambda Expressions in Java*. http://wiki.jvmlangsummit.com/images/7/7b/Goetz-jvmls-lambda.pdf. 30 July 2012.

[24] David Gregg, M.Anton Ertl, and Andreas Krall. "Implementing an Efficient Java Interpreter". In: *High-Performance Computing and Networking*. Ed. by Bob Hertzberger, Alfons Hoekstra, and Roy Williams. Vol. 2110. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 613–620. DOI: 10.1007/3-540-48228-8_70.

[25] Rich Hicky et al. *Clojure*. URL: http://clojure.org/ (visited on 03/27/2014).

[26] Urs Hölzle, Craig Chambers, and David Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches". In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP '91. London, UK, UK: Springer-Verlag, 1991, pp. 21–38. ISBN: 3-540-54262-0.

[27] *IKVM.NET*. URL: http://www.ikvm.net/ (visited on 05/13/2014).

[28] Chanwit Kaewkasi. "Towards Performance Measurements for the Java Virtual Machine's Invokedynamic". In: *Virtual Machines and Intermediate Languages*. VMIL '10. Reno, Nevada: ACM, 2010, 3:1–3:6. DOI: 10.1145/1941054.1941057.

[29] Thomas Kotzmann et al. "Design of the Java HotSpot Client Compiler for Java 6". In: *ACM Trans. Archit. Code Optim.* 5.1 (May 2008), 7:1–7:32. ISSN: 1544-3566. DOI: 10.1145/1369396.1370017.

[30] Andreas Krall. "Efficient JavaVM just-in-time compilation". In: *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*. IEEE. 1998, pp. 205–212.

[31] Andreas Krall and Reinhard Grafl. "CACAO — A 64-bit JavaVM just-in-time compiler". In: *Concurrency: Practice and Experience* 9.11 (Nov. 1997). Special Issue: Java for computational science and engineering — simulation and modeling II., pp. 1017–1030. ISSN: 1040-3108.

[32] Marcus Lagergren. *Nashorn War Stories*. http://www.slideshare.net/lagergren/lagergren-jvmls2013final.

[33] Roland Lezuo. "Porting the CACAO Virtual Machine to POWERPC64 and Coldfire". MA thesis. Vienna University of Technology, 2007.

[34] Tim Lindholm et al. *The Java Virtual Machine Specification, Java SE 7 Edition*. 1st. Addison-Wesley Professional, 2013. ISBN: 0133260445, 9780133260441.

[35] Robert Lougher. *JamVM*. 2013. URL: http://jamvm.sourceforge.net (visited on 11/03/2013).

[36] Erik Meijer and John Gough. *Technical overview of the Common Language Runtime, 2000*. 2000. URL: https://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf (visited on 03/27/2014).

[37] Peter Molnar, Andreas Krall, and Florian Brandner. "Stack Allocation of Objects in the CACAO Virtual Machine". In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. PPPJ '09. Calgary, Alberta, Canada: ACM, 2009, pp. 153–161. DOI: 10.1145/1596655.1596680.

[38] Charles Moore. *ColorForth*. URL: http://www.colorforth.com/ (visited on 07/22/2014).

[39] Mozilla. *Rhino JavaScript VM*. URL: https://developer.mozilla.org/en-US/docs/Rhino (visited on 03/27/2014).

[40] Charles Oliver Nutter et al. *JRuby*. URL: http://jruby.org/ (visited on 12/11/2013).

[41] Martin Odersky et al. "An overview of the Scala programming language". In: ().

[42] *OpenJDK*. URL: http://openjdk.java.net/ (visited on 11/03/2013).

[43] Michael Paleczny, Christopher Vick, and Cliff Click. “The Java HotSpot™ server compiler”. In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM’01. Monterey, California: USENIX Association, 2001, pp. 1–1.

[44] M. Poletto and V. Sarkar. “Linear Scan Register Allocation”. In:

[45] Julien Ponge and Frédéric Le Mouël. “JooFlux: Hijacking Java 7 InvokeDynamic To Support Live Code Modifications”. In: *arXiv* abs/1210.1039 (2012).

[46] *Project Nashorn*. URL: http://openjdk.java.net/projects/nashorn (visited on 03/28/2014).

[47] Stefan Ring. “Implementation of native threads and locks in the CACAO Java Virtual Machine”. MA thesis. Vienna University of Technology, 2008.

[48] John Rose. *Indify repository*. URL: https://kenai.com/projects/ninja/sources/indify-repo/show (visited on 07/22/2014).

[49] John R. Rose. “Bytecodes meet combinators: invokedynamic on the JVM”. In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. VMIL ’09. Orlando, Florida: ACM, 2009, 2:1–2:11. DOI: 10.1145/1711506.1711508.

[50] John R. Rose. *JEP 160: Lambda-Form Representation for Method Handles*. Dec. 2012. URL: http://openjdk.java.net/jeps/160 (visited on 01/05/2014).

[51] John R. Rose. *JSR 292: Supporting Dynamically Typed Languages on the Java™Platform*. 2008. URL: http://jcp.org/en/jsr/detail?id=292 (visited on 11/03/2013).

[52] *Sites, Binders, and Dynamic Object Interop Spec*. July 2009. URL: http://nuxleus.googlecode.com/svn/trunk/nuxleus/Source/Vendor/Microsoft/Codeplex-DLR-0.9/docs/sites-binders-dynobj-interop.doc (visited on 03/27/2014).

[53] Todd Smith et al. “Experiences with Retargeting the Java Hotspot(Tm) Virtual Machine”. In: *Proceedings of the 16th International Parallel and Distributed Processing Symposium*. IPDPS ’02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 146–. ISBN: 0-7695-1573-8.

[54] Michael Starzinger. “Exact Garbage Collection for the Cacao Virtual Machine”. MA thesis. Vienna University of Technology, 2011.

[55] Edwin Steiner, Andreas Krall, and Christian Thalinger. “Adaptive Inlining and On-stack Replacement in the CACAO Virtual Machine”. In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. PPPJ ’07. Lisboa, Portugal: ACM, 2007, pp. 221–226. DOI: 10.1145/1294325.1294356.

[56] Christian Thalinger. “Optimizing and Porting the CACAO JVM”. MA thesis. Vienna University of Technology, 2004.

[57] Christian Thalinger and John Rose. “Optimizing invokedynamic”. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. PPPJ ’10. Vienna, Austria: ACM, 2010, pp. 1–9. DOI: 10.1145/1852761.1852763.

[58] *The Dynamic Language Runtime*. URL: https://dlr.codeplex.com/ (visited on 03/27/2014).

[59] Robert Tolksdorf. *Programming languages for the Java Virtual Machine JVM and Javascript*. URL: http://www.is-research.de/info/vmlanguages/ (visited on 12/11/2013).

[60] Mads Torgersen. "Querying in C#: How Language Integrated Query (LINQ) Works". In: *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: ACM, 2007, pp. 852–853. DOI: 10.1145/1297846.1297922.

[61] Greg Wilson. "The Dynamic Language Runtime and the Iron Languages". In: *The Architecture of Open Source Applications, Volume II*. Kristian Hermansen, 2012. URL: http://aosabook.org/en/ironlang.html.

[62] Thomas Würthinger et al. "One VM to Rule Them All". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 187–204. DOI: 10.1145/2509578.2509581.