Appendix A - An Optimization

The following set of pseudo-instructions suffixed by *_quick* are variants of Java virtual machine instructions. They are used by the LiveOak project to improve the execution of compiled code on our bytecode interpreter. They are not part of the virtual machine specification or instruction set, and are invisible outside of an Java virtual machine implementation. However, inside a virtual machine implementation they have proven to be an effective optimization.

A compiler from Java to the Java virtual machine instruction set emits only non-*_quick* instructions. If the *_quick* pseudo-instructions are used, each instance of a non-*_quick* instruction with a *_quick* variant is overwritten on execution by its *_quick* variant. Subsequent execution of that instruction instance will be of the *_quick* variant.

In all cases, if an instruction has an alternative version with the suffix *_quick*, the instruction references the constant pool. If the *_quick* optimization is used, each non_*quick* instruction with a *_quick* variant performs the following:

- Resolves the specified item in the constant pool
- Signals an error if the item in the constant pool could not be resolved for some reason
- Turns itself into the *_quick* version of the instruction. The instructions **putstatic**, **getstatic**, **putfield**, and **getfield** each have two *_quick* versions.
- Performs its intended operation

This is identical to the action of the instruction without the *_quick* optimization, except for the additional step in which the instruction overwrites itself with its *_quick* variant.

The *_quick* variant of an instruction assumes that the item in the constant pool has already been resolved, and that this resolution did not generate any errors. It simply performs the intended operation on the resolved item.

Pushing Constants onto the Stack (_quick variants)

ldc1_quick

Push item from constant pool onto stack

Syntax:

ldc1_quick = 199
indexbyte1

Stack: ... => ..., *item*

indexbyte1 is used as an unsigned 8-bit index into the constant pool of the current class. The *item* at that index is pushed onto the stack.

ldc2_quick

Push item from constant pool onto stack

Syntax:

ldc2_quick = 200
indexbyte1
indexbyte2

Stack: ... => ..., *item*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The *constant* at that index is resolved and the *item* at that index is pushed onto the stack.

ldc2w_quick

Push long integer or double float from constant pool onto stack

Syntax:

ldc2w_quick = 201
indexbyte1
indexbyte2

Stack: ... =>=> ..., *constant*-word1, *constant*-word2

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The *constant* at that index is pushed onto the stack.

Managing Arrays (_quick variants)

anewarray_quick

Allocate new array of objects

Syntax:

anewarray_quick = 216
indexbyte1
indexbyte2

Stack: ..., *size* => *result*

size should be an integer. It represents the number of elements in the new array.

indexbyte1 and *indexbyte2* are are used to construct an index into the constant pool of the current class. The entry should be a class.

A new array of the indicated class type and capable of holding *size* elements is allocated. Allocation of an array large enough to contain *nelem* items of the given class type is attempted. All elements of the array are initialized to zero.

If *size* is less than zero, a `NegativeArraySizeException` is thrown. If there is not enough memory to allocate the array, an `OutOfMemoryException` is thrown.

Manipulating Object Fields (_quick variants)

putfield_quick

Set field in object

Syntax:

putfield_quick = 203
offset
unused

Stack: ..., *handle*, *value* => ...

handle should be a handle to an object. *value* should be a value of a type appropriate for the specified field. *offset* is the offset for the field in that object. *value* is written at *offset* into the object referenced by *handle*. Both *handle* and *value* are popped from the stack.

If *handle* is null, a `NullPointerException` exception is generated.

putfield2_quick

Set long integer or double float field in object

Syntax:

putfield2_quick = 205
offset
unused

Stack: ..., *handle*, *value*-word1, *value*-word2=> ...

handle should be a handle to an object. *value* should be a value of a type appropriate for the specified field. *offset* is the offset for the field in that object. *value* is written at *offset* into the object referenced by *handle*. Both *handle* and *value* are popped from the stack.

If *handle* is null, a `NullPointerException` exception is generated.

getfield_quick

Fetch field from object

Syntax:

getfield_quick = 202
offset
unused

Stack: ..., *handle* => ..., *value*

handle should be a handle to an object. The value at *offset* into the object referenced by *handle* replaces *handle* on the top of the stack.

If *handle* is null, a `NullPointerException` exception is generated.

getfield2_quick

Fetch field from object

Syntax:

getfield2_quick = 204
offset
unused

Stack: ..., *handle* => ..., *value*-word1, *value*-word2

handle should be a handle to an object. The value at *offset* into the object referenced by *handle* replaces *handle* on the top of the stack.

If *handle* is null, a `NullPointerException` exception is generated.

putstatic_quick

Set static field in class

Syntax:

putstatic_quick = 207
indexbyte1
indexbyte2

Stack: ..., *value* => ...

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. *value* should be the type appropriate to that field. That field will be set to have the value *value.*

putstatic2_quick

Set static field in class

Syntax:

putstatic2_quick = 209
indexbyte1
indexbyte2

Stack: ..., *value*-word1, *value*-word2 => ...

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. That field should either be a long integer or a double precision floating point number. *value* should be the type appropriate to that field. That field will be set to have the value *value.*

getstatic_quick

Get static field from class

Syntax:

getstatic_quick = 206
indexbyte1
indexbyte2

Stack: ..., => ..., *value*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The value of that field will replace *handle* on the stack.

getstatic2_quick

Get static field from class

Syntax:

getstatic2_quick = 208
indexbyte1
indexbyte2

Stack: ..., => ..., *value*-word1, *value*-word2

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The constant pool item will be a field reference to a static field of a class. The field should be a long integer or a double precision floating point number. The value of that field will replace *handle* on the stack

Method Invocation (_quick variants)

invokevirtual_quick

Invoke class method

Syntax:

invokevirtual_quick = 210
offset
nargs

Stack: *...*, *handle*, [*arg1*, [*arg2* ...]] => *...*

The operand stack is assumed to contain a *handle* to an object and *nargs* arguments. The method block at *offset* in the object's method table is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked *synchronized* the monitor associated with *handle* is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to *handle* on the stack, making *handle* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowException` is thrown.

invokevirtualobject_quick

Invoke class method

Syntax:

invokevirtualobject_quick = 214
offset
nargs

Stack: *...*, *handle*, [*arg1*, [*arg2* ...]] => *...*

The operand stack is assumed to contain a *handle* to an object or to an array and *nargs* arguments. The method block at *offset* in the object's method table is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked *synchronized* the monitor associated with *handle* is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to *handle* on the stack, making *handle* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowException` is thrown.

invokenonvirtual_quick

Invoke superclass method

Syntax:

invokenonvirtual_quick = 211
indexbyte1
indexbyte2

Stack: ..., *handle*, [*arg1*, [*arg2* ...]] => ...

The operand stack is assumed to contain a *handle* to an object and some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked *synchronized* the monitor associated with *handle* is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to *handle* on the stack, making *handle* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowException` is thrown.

invokestatic_quick

Invoke a static method

Syntax:

invokenstatic_quick = 212
indexbyte1
indexbyte2

Stack: ..., [*arg1*, [*arg2* ...]] => ...

The operand stack is assumed to contain some number of arguments. *indexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains a method slot index and a pointer to a class. The method block at the method slot index in the indicated class is retrieved. The method block indicates the type of method (native, synchronized, etc.) and the number of arguments (*nargs*) expected on the operand stack.

If the method is marked *synchronized* the monitor associated with the method's class is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to the first argument on the stack, making the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowException` is thrown.

invokeinterface_quick

Invoke interface method

Syntax:

invokeinterface_quick = 213
idbyte1
idbyte2
nargs
guess

Stack: ..., *handle*, [*arg1*, [*arg2* ...]] => ...

The operand stack is assumed to contain a *handle* to an object and *nargs*-1 arguments. *idbyte1* and *idbyte2* are used to construct an index into the constant pool of the current class. The item at that index in the constant pool contains the complete method signature. A pointer to the object's method table is retrieved from the object *handle*.

The method signature is searched for in the object's method table. As a short-cut, the method signature at slot *guess* is searched first. If that fails, a complete search of the method table is performed. The method signature is guaranteed to exactly match one of the method signatures in the table.

The result of the lookup is a method block. The method block indicates the type of method (native, synchronized, etc.) but unlike *invokemethod* and *invokesuper*, the number of available arguments (*nargs*) is taken from the bytecode.

If the method is marked *synchronized* the monitor associated with *handle* is entered. The exact behavior of monitors and their interactions with threads is a runtime issue.

The base of the local variables array for the new Java stack frame is set to point to *handle* on the stack, making *handle* and the supplied arguments (*arg1*, *arg2*, ...) the first *nargs* local variables of the new frame. The total number of local variables used by the method is determined, and the execution environment of the new frame is pushed after leaving sufficient room for the locals. The base of the operand stack for this method invocation is set to the first word after the execution environment. Finally, execution continues with the first instruction of the matched method.

If the object handle on the operand stack is null, a `NullPointerException` is thrown. If during the method invocation a stack overflow is detected, a `StackOverflowException` is thrown.

guess is the last guess. Each time through, guess is set to the method offset that was used.

Miscellaneous Object Operations (_quick variants)

new_quick

Create new object

Syntax:

new_quick = 215
indexbyte1
indexbyte2

Stack: ... => ..., *handle*

indexbyte1 and *indexbyte2* are used to construct an index into the constant pool of the current class. The item at that index should be a class. A new instance of that class is then created and a *handle* for it pushed on the stack.

checkcast_quick

Make sure object is of given type

Syntax:

checkcast_quick = 217
indexbyte1
indexbyte2

Stack: ..., *handle* => ..., *handle*

handle should be a handle to an object. i*ndexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The object at that index of the constant pool should have already been resolved.

checkcast then determines whether *handle* can be cast to an object of class *class*. A null *handle* can be cast to any *class*, and otherwise the superclasses of *handle* are searched for *class*. If class is determined to be a superclass of *handle*, or if *handle* is null, *object* can be cast to *class* and execution proceeds at the next instruction, and the handle for *handle* remains on the stack.

If *handle* cannot be cast to *class*, a `ClassCastException` is thrown.

instanceof_quick

Determine if object is of given type

Syntax:

instanceof_quick = 218
indexbyte1
indexbyte2

Stack: ..., *handle* => ..., *result*

handle should be a handle to an object. i*ndexbyte1* and *indexbyte2* are used to construct an index into the constant pool of the current class. The item of class *class* at that index of the constant pool is assumed to have already been resolved.

instanceof determines whether *handle* can be cast to an object of the class *class*. A null *handle* can be cast to any *class*, and otherwise the superclasses of *handle* are searched for *class*. If *class* is determined to be a superclass of *handle*, or if *handle* is null, *handle* is overwritten by 1. Otherwise, *handle* is overwritten by 0.

Constant Pool Resolution

When the class is read in, an array `constant_pool[]` of size `nconstants` is created and assigned to a field in the class. `constant_pool[0]` is set to point to a malloc-ed array which indicates which fields in the constant_pool have already been resolved. `constant_pool[1]` through `constant_pool[nconstants - 1]` are set to point at the "type" field that corresponds to this constant item.

When an instruction is executed that references the constant pool, an index is generated, and `constant_pool[0]` is checked to see if the index has already been resolved. If so, the value of `constant_pool[index]` is returned. If not, the value of `constant_pool[index]` is resolved to be the actual pointer or data, and overwrites whatever value was already in `constant_pool[index]`.