Static Analysis of Dynamic Languages

Jennifer Strater

2017-05-16

Table of Contents

Introduction

The study of static analysis in the context of dynamic languages is an interesting and complex problem because it presents different challenges than the ones found in traditional static languages. For this study, I chose the dynamic programming language Groovy because of my background with the language and knowledge of some of the inner workings. Additionally, the bytecode manipulation and analysis framework ASM[1] was used because of its use within the current Groovy compiler.

Since dynamic languages, particularly Groovy, have custom logic to support their dynamic features, the implementation may vary from other compilers such as ones for Java. Investigating the differences in performance and implementation are the topic of this project.

Additionally, since the language is primarily developed by volunteer open source contributors and not a full time dedicated support staff with resources to fix it, some implementations of the language features are not as optimal as they could be. Particularly when we look at benchmarking comparisons of Groovy to standard core Java and the quality of the output of the bytecode prouced by the Groovy compiler, one can see the differences. It's also been very useful and quite a unique experience to work with a working compiler hands-on used in many production systems around the world and to have direct access to the users and community.

The Three Compiler Options

For background, within the groovy project, there are three different ways to compile groovy code: the legacy compiler also labeled `groovyc`, the indy option, and the static compilation flag.

The main differences between the three have to do with method invocation. Let's look at the differences in the context of the bytecode generated for a simple helloworld.groovy program. The smallest `Hello, World!` example would be the simple script:

```
println 'Hello, World!'
```

Legacy Compiler

In the legacy compiler, methods are run using a callsite caching mechanism[2] as seen in Compilation with Groovyc which has much worse performance than other invocation strategies.

Compilation with Groovyc

```
  public static void main(java.lang.String...);
    Code:
       0: invokestatic  #17                 // Method
$getCallSiteArray:()[Lorg/codehaus/groovy/runtime/callsite/CallSite;
       3: astore_1
       4: aload_1
       5: ldc           #27                 // int 0
       7: aaload
       8: ldc           #29                 // class
org/codehaus/groovy/runtime/InvokerHelper
      10: ldc           #2                  // class HelloWorld
      12: aload_0
      13: invokeinterface #35,  4           // InterfaceMethod
org/codehaus/groovy/runtime/callsite/CallSite.call:(Ljava/lang/Object;Ljava/lang/Objec
t;Ljava/lang/Object;)Ljava/lang/Object;
      18: pop
      19: return
```

Invoke Dynamic (Indy)

Invoke Dynamic, aka *Indy*, is a flag set at compilation to specify that the compiler should replace old style callsite caching mechanism with the Java7 InvokeDynamic instruction. It also requires a version of groovy that supports the *indy* option such as the `groovy-all` or the `groovy-indy` jars[3].

```
groovyc --indy helloworld.groovy
```

The source code below is the same helloworld source code as the example above, but with the indy flag set. Notice the difference in the two main methods. The overhead of the legacy compiler, in particular for maintaining the callsite information, is much higher than for using Indy.

```
  public static void main(java.lang.String...);
    Code:
       0: ldc           #24                 // class
org/codehaus/groovy/runtime/InvokerHelper
       2: ldc           #2                  // class HelloWorld
       4: aload_0
       5: invokedynamic #38,  0             // InvokeDynamic
#0:invoke:(Ljava/lang/Class;Ljava/lang/Class;[Ljava/lang/String;)Ljava/lang/Object;
      10: pop
      11: return
```

Static Compilation

Another option for compiling groovy code is static compilation which is set through the config script as shown in Static Compilation Configuration Script. Static Compilation adds some additional

functionality such as type checking and limiting the scope of Groovy's metaprogramming capabilities. In small examples, there are few differences between Indy and Static Compilation. However, with larger more complex programs, Static Compilation is often the way to go. However, because it is also the strictest compilation option, some programs may not compile if they use features of the Groovy language that violate static compilation.

Static Compilation Configuration Script

```
withConfig(configuration) {
    ast(groovy.transform.CompileStatic)
}
```

```
  public static void main(java.lang.String...);
    Code:
       0: ldc           #2                  // class HelloWorld
       2: aload_0
       3: invokestatic  #28                 // Method
org/codehaus/groovy/runtime/InvokerHelper.runScript:(Ljava/lang/Class;[Ljava/lang/Stri
ng;)Ljava/lang/Object;
       6: pop
       7: return
```

Scripts vs Applications

The bytecode generated as seen in Complete Bytecode Example for HelloWorld includes a lot more than a simple print statement. In order for the code to run on the JVM, Groovy adds a wrapper around the code of type `groovy.lang.Script`. The script contains a `run` method with all of the code for the script. The main method runs the simple script. There's also a MetaClass, that contains information about the Script. This isn't used in the simple Hello World example, but is used for complex dynamic features of Groovy.

Complete Bytecode Example for HelloWorld

```
Compiled from "HelloWorld.groovy"
public class HelloWorld extends groovy.lang.Script {
  public static transient boolean __$stMC;

  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #13                 // Method groovy/lang/Script."<init>":()V
       4: return

  public HelloWorld(groovy.lang.Binding);
    Code:
       0: aload_0
       1: aload_1
       2: invokespecial #18                 // Method
```

```
groovy/lang/Script."<init>":(Lgroovy/lang/Binding;)V
       5: return

  public static void main(java.lang.String...);
    Code:
       0: ldc           #2                  // class HelloWorld
       2: aload_0
       3: invokestatic  #28                 // Method
org/codehaus/groovy/runtime/InvokerHelper.runScript:(Ljava/lang/Class;[Ljava/lang/Stri
ng;)Ljava/lang/Object;
       6: pop
       7: return

  public java.lang.Object run();
    Code:
       0: aload_0
       1: checkcast     #2                  // class HelloWorld
       4: ldc           #34                 // String Hello, World!
       6: invokevirtual #38                 // Method println:(Ljava/lang/Object;)V
       9: aconst_null
      10: areturn
      11: aconst_null
      12: areturn

  protected groovy.lang.MetaClass $getStaticMetaClass();
    Code:
       0: aload_0
       1: invokevirtual #46                 // Method
java/lang/Object.getClass:()Ljava/lang/Class;
       4: ldc           #2                  // class HelloWorld
       6: if_acmpeq     14
       9: aload_0
      10: invokestatic  #52                 // Method
org/codehaus/groovy/runtime/ScriptBytecodeAdapter.initMetaClass:(Ljava/lang/Object;)Lg
roovy/lang/MetaClass;
      13: areturn
      14: getstatic     #54                 // Field
$staticClassInfo:Lorg/codehaus/groovy/reflection/ClassInfo;
      17: astore_1
      18: aload_1
      19: ifnonnull     34
      22: aload_0
      23: invokevirtual #46                 // Method
java/lang/Object.getClass:()Ljava/lang/Class;
      26: invokestatic  #60                 // Method
org/codehaus/groovy/reflection/ClassInfo.getClassInfo:(Ljava/lang/Class;)Lorg/codehaus
/groovy/reflection/ClassInfo;
      29: dup
      30: astore_1
      31: putstatic     #54                 // Field
$staticClassInfo:Lorg/codehaus/groovy/reflection/ClassInfo;
```

```
      34: aload_1
      35: invokevirtual #63                 // Method
org/codehaus/groovy/reflection/ClassInfo.getMetaClass:()Lgroovy/lang/MetaClass;
      38: areturn
}
```

In that last example, we see the first mention of running a script. It is worth noting that in Groovy, you can run full applications with a main method just like in Java or there is a built-in mechanism for running small scripts. The scripting functionality works by wrapping the code to be executed into a script class with a main method called run. This additional overhead can have affects on the performance of the code. To look at more comparisons of the bytecode optimizations for scripts and applications/jars, see the bytecode report or Appendix A from the pdf.

Identification of Potential Optimizations

To begin any investigation into potential optimizations, it is important to first look at the cause and effect of various parts of the system.

The Source of the Problem

When looking at the bytecode generated from each of the three compiler options, there are some common themes. This is particularly true when looking at method execution.

The main source of the extra overhead in Groovy bytecode is from the build up and tear down of the stack. This happens to support the return by default feature of the Groovy programming language. The compiler keeps track of the result of each statement in case it needs to be returned. However, if the method is void or has no possible route from that statement to the end, then the operations are not needed.

Solutions

Post Pass Optimization

In order to solve this problem in the short-term, we propose a post-pass optimization to identify sequences of operations that are just doing busy work. The two first identified are load and pop as shown in lines 29 and 30 of the

In collectloop with static compilation in the run() method

```
      29: aload_1
      30: pop
```

and

In nsieve with indy compilation in the countSieve() method

```
64: dup
65: astore_3
66: pop
```

When a load is immediately followed by a pop like in the first example, both operations can be deleted. When the sequence is dupe, store, then pop like in the second example, the dupe and pop can be removed and the store should be left. There are of course, many other variations of these duplicated efforts throughout the examples. For simplicity, we just chose these two. Adding more small tweaks would have diminishing returns. Ultimately, the root cause of the problem should be fixed in the compiler itself instead of in a post-pass optimization.

Project

The source code in this project is published to Github. It contains a simple Groovy application with tests and documentation as well as some example scripts, and jars.

Setup

The project is setup using the build tool Gradle. For scripts, the basic application takes each script from the resource folder and applies each of the three compilation options using the command line execution feature of Java. After the bytecode is generated, it is fed back into the program using an ASM reader. Using the ASM framework, the bytecode is optimized to remove the extraneous instructions. A separate pass is done to ensure the lines removed are not the target of a jump/goto operation. The number of lines removed are counted for the reports.

Test Cases

To test the efficacy of the bytecode optimizations, a variet of different test cases were used. There were a number of scripts included in the resources directory that covered a wide variety of typical Groovy code operations including the simple hello world example, looping, complex branching, and pattern matching. To ensure no breaking changes, the bytecode was run before and after the changes to ensure the correct output.

In addition to the small scripts, a number of large jar files including the groovy-all distribution and this application were added to the test suite. For simplicity, the before and after source code for applications and jar files were not included in the report — only the number of lines removed.

Performance Differences

Since the optimization removes a lot of busy work operations, it was expected that the optimizations would result in a performance difference. However, probably due to the high overhead of the groovy library in comparison to the scripts and applications run, we were unable to detect a statistically significant difference in the performance for any of the test cases.

Jar Size Differences

After asking the Groovy compiler creators about the potential optimization on the community mailing list[3], it was revealed that the performance of the JVM is highly tuned and therefore the optimization I was proposing would not be expected to have a major performance impact. However, one way it would have an impact is in the size of the output. For most cases, the size of the jar doesn't matter; however, in very large applications, the size of the bytecode may exceed the limit of the JVM. In that case, we could show that this optimization would have an advantage. However, I do not have any examples that large to test.

Challenges

As mentioned in the section on jar size differences, one of the major challenges of this project was coming up with large examples that would be testable. Many large groovy applications have many dependencies that are actually Java and not Groovy or are packaged in a way that I could not use the traditional Java command line jar option to put it back together again after running the optimization.

Next Steps

In the immediate future, I would like to submit my final post-pass optimzition as a pull request to the current Groovy compiler. This would require integrating the current reader and writer into the existing ASM framework.

Fix the Compiler

The long term goal of this project would be to fix the underlying problems with the compiler when it's generating these unnecessary instructions. The potential spots for optimization would be in the handling of the return statement. First, if we know that the method is void, then the compiler does not need to keep track of anything on the stack. Next, if the method is not void then I would start by doing a backward analysis of the code to find the potential exits. After identifying the exits, only those statements need to be tracked on the stack and the rest of the statements can be eliminated.

Bibliography

- [1] ASM. 'ASM - Home Page'. 23 Dec 2016. http://asm.ow2.org
- [2] Cedric Champeau. 'Using Groovy to play with invokedynamic'. 31 Jan 2013. http://melix.github.io/blog/2013/01/31/using_groovy_to_play_with.html
- [3] Strater, Jennifer, Cédric Champeau, and Jochen Theodorou. "Potential Bytecode Optimization." Groovy Users Mailing List. Apache Groovy, 06 Apr. 2017. Web. 16 May 2017. http://mail-archives.apache.org/mod_mbox/groovy-users/201704.mbox/browser