Static Analysis of Dynamic Languages

Jennifer Strater

2017-05-18

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, the dynamic programming language Groovy, an Apache project, was chosen because of previous knowledge and experience with the language. Additionally, the bytecode manipulation and analysis framework ASM[1] was used because of its use within the current Apache 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 between Java and Apache Groovy is the topic of this project.

The language is primarily developed by volunteer open source contributors and not a full-time dedicated support staff with resources to fix it. Therefore, some implementations of the language features are not as optimal as they could be. Particularly when we look at benchmarking comparisons of Apache Groovy to standard core Java and the quality of the output of the bytecode produced by the Apache Groovy compiler, one can see the differences. It has also been very useful and quite a unique experience to work with a 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 Apache 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. The next few examples look at the differences of the bytecode generated for a simple helloworld.groovy program. The smallest Hello World example in Apache Groovy would be the simple script:

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

Legacy Compiler

In the legacy compiler, implemented in a pre 2.0 version of the language, methods are run using a callsite caching mechanism[2] as seen in Bytecode from Compilation with Groovyc which has much worse performance than other invocation strategies. However, when using Apache Groovy with versions of Java less than Java7, this is the only option.

Bytecode from 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 during compilation to specify that the compiler should replace the outdated callsite caching mechanism with the Java7 InvokeDynamic instruction. It also requires a version of Apache Groovy that supports the *indy* option such as the `groovy-all` or the `groovy-indy` jars[3]. From a command line installation of groovy-all, that can be done as seen in Compiling With Invoke Dynamic.

Compiling With Invoke Dynamic

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

The source code in Bytecode from Compilation with Invoke Dynamic below is the same helloworld source code as Bytecode from Compilation with Groovyc, 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*.

Bytecode from Compilation with Invoke Dynamic

```
  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 Apache 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.

Static Compilation Configuration Script

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

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 Apache Groovy language that violate static compilation such as the def type notation.

```
  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. For the code to run on the JVM, Apache 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 is also a MetaClass which contains information about the Script. The MetaClass class is not used in the simple Hello World example, but it is used for complex dynamic features of Groovy. Groovy applications can also be written with a main class in which case the scripting wrapper is not used.

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
}
```

The additional overhead from the metaclassing 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.

Process

Identification of Potential Optimizations

To begin any investigation into possible optimizations, it is important to first look at the cause and effect of various parts of the system, in this case, the Apache Groovy compiler itself.

The Source of the Problem

Starting with the bytecode generated from each of the three compiler options, we found some common themes. The main source of the extra overhead in Apache Groovy bytecode is from the build up and tear down of the stack. The extra actions occur from the implementation of the return by default feature of the Apache Groovy programming language. In order to make the return mechanism work, the compiler keeps track of the result of each statement as it is executed in case that is the value that needs to be returned. However, if the method is void, and thus should not return anything, or has no possible route from the current statement to the end, then the operations to store the value on the stack are not needed, and the beginning of the next statement compilation adds a pop to remove this superfluous value. One example of this is shown below in Groovy Compiler Implementation for Block Statements which comes from the StatementWriter class which is run during the class generation phase of the Apache Groovy compiler.

Groovy Compiler Implementation for Block Statements

```
public void writeBlockStatement(BlockStatement block) {
        CompileStack compileStack = controller.getCompileStack();

        //GROOVY-4505 use no line number information for the block
        writeStatementLabel(block);

        int mark = controller.getOperandStack().getStackLength();
        compileStack.pushVariableScope(block.getVariableScope());
        for (Statement statement : block.getStatements()) {
            statement.visit(controller.getAcg());
        }
        compileStack.pop();

        controller.getOperandStack().popDownTo(mark);
    }
```

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 Load then Pop Example.

In collectloop with static compilation in the run() method

Load then Pop Example

A superfluous load then pop in the generated bytecode from the collectloop.groovy script with static compilation.

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

and

Dup Store Pop Example

An example from the nsieve.groovy script after indy compilation. This extract is from the countSieve() method.

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

These types of optimizations can also be called peephold optimizations because they look at small parts of the bytecode without the rest of the context and after it has been generated. By running this type of analysis, we can see the cases defined above. Then, when we find a load instruction is immediately followed by pop, such as in the first example, we can delete both the load and pop operations from that method's instruction set. Alternatively, when the sequence is dupe, store, then pop, such as 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 types of optimizations that can be identified and removed. The output as shown in the bytecode reports from the test cases of the project can be used to find more. For simplicity; however, we chose to implment just 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 Apache Groovy application with tests and documentation as well as some example scripts and jars.

Setup

The project is setup using the build tool Gradle with several plugins for making fat jar files, running a Groovy application, a static analysis tool for Apache Groovy, and generating documentation. The project can be run using the Gradle wrapper included with the source code. For more information about running the project, see the README.md.

Implementation

For scripts, the application takes the script specified as an argument to the program and applies each of the three compilation options using the command line execution API native to Java. After the bytecode is generated using the currently available version of the Apache Groovy compiler via command line, the generated bytecode is imported 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 operation. Then, the different versions of the bytecode are benchmarked, and a report generated to compare the bytecode both visually and in runtime. When no parameters are specified, the optimizer is run against the source code for the application itself and the benchmark is run with the helloworld script as a parameter for the jar file.

Test Cases

A variety of different test cases were used to test the efficacy of the bytecode optimizations. There were a number of scripts included in the resources directory that covered a wide variety of typical

Apache Groovy code operations including the simple hello world example, looping, complex branching, and pattern matching. The example programs come from both the the benchmark package in the Apache Groovy source, and interesting examples from the course. To ensure the optimizations did not break the functionality of each program, 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 reports — only the number of lines removed.

Reports

The project includes a number of reports and statistics that were used for comparing the performance before and after as well as the size of the output bytecode and a visual output of what was generated. To visualize the bytecode, the bytecode was run through the command line program javap.

Results

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 Apache 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 Apache Groovy compiler creators about the potential optimization on the community mailing list[3], it was revealed that most JVMs are highly tuned and therefore the optimization proposed 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 does not 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, there were not any examples that large to test. In the scripts tested, the optimization could reduce the size of bytecode by around 10%. For larger jar files, for example, the groovy-all and codenarc jars, the optimization removed about 10.000 lines of bytecode instructions.

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 Apache Groovy applications have many dependencies that are actually Java and not Groovy or are packaged in a way that could not be reproduced using the traditional Java command line jar option to put it back together again after running the optimization.

Conclusion

In conclusion, this coursework was highly informative and revealed a lot of helpful information about the inner workings of the compiler for the Apache Groovy programming language. Now that we have identified some potential places for optimization, we can continue to work towards correcting these problems.

Next Steps

In the immediate future, the final post-pass optimization will be submitted as a pull request to the current Apache Groovy project on Github. This would require integrating with the current ASM reader and writer for the Apache Groovy project and adding a cleaner version of the BytecodeOptimizer class from this project. It may cause some significant slowdowns of compilation time as the compiler will be required to run two additional passes over every method. Perhaps it would be better to have this as an optional feature set via a configuration option. The current Apache Groovy project team can help decide the best way forward for this contribution.

Fix the Compiler

The long-term goal would be to fix the underlying problems with the compiler when it is generating the 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 a backward analysis can detect the program's 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