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.

Since dynamic languages, particularly Groovy, have custom logic to support their dynamic features and the language is primarily developed by volunteer open source contributors, 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, one can see the differences. Investigating these differences in performance and implementation are the problem space for this project.

The Three Compiler Options

Within the groovy project, there are three different ways to compile groovy code: the legacy compiler, the indy option, and the static compilation flag. The main differences between the three have to do with method invocation. For the differences, let's look at the bytecode generated for a simple helloworld.groovy program.

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

Legacy Compiler

In the legacy compiler methods are run using a callsite caching mechanism[1] as seen in Figure 1.

```
  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 time as seen in the snippet below to specify that the code should be compiled using the Java7 feature InvokeDynamic instead of using the callsite caching mechanism of the legacy compiler.

```
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. 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.

```
  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

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.

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. Particularly when looking at method execution.

Return Last Result By Default

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

Setup

Test Cases

Performance Differences

Jar Size Differences

Challenges

Next Steps

Fix the Compiler

Bibliography

- [1] 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