Performance of HotSpot with various Flags

Boyuan Feng

Section 1. Introduction

HotSpot is an implementation of Java virtual machine and is widely used in many tasks. It utilizes a combination of methods like interpretation and compilation to improve program performance. To allow finetune the performance, HotSpots provides a large number of flags on different configurations, like whether using garbage collection or not and when compilation should be used. In theory, we know that interpretation will be much slower than compilation (low efficiency) and the compilation may introduce too much overhead (high latency) if the method will not be called for many times. Thus there is a trade-off between efficiency and latency, which may varies on different workload. We want to explore this trade-off and find scenarios that favor different configurations.

In this project, we are going to compare the performance of HotSpot with various flags measured in running time. The comparison will focus on the effect of interpretation, compilation, and garbage collection. Six flags are selected to tune the performance of program, i.e., disabling background compilation, disabling compilation at all, forcing the compilation of methods on first invocation, etc. To evaluate the effect of flags under different workload, eleven programs from website [1] are selected. These programs covers various workload including intensive IO and heavy workload on memory. Through a series of experiments, we show the following results.

1. Performance of programs varies a lot even with same code and on the same machine.
2. Interpretation runs much slower than compilation when there are lots of calls to a method and disabling compilation in this scenario would introduce a disaster in performance.
3. Force compilation of methods on first invocation would introduce a high overhead, which shows the importance of interpretation on “cold” method, which does not run many times.
4. In most of the time, disabling garbage collection saves little time.

Section 2. Selected Flags

HotSpot maintains hundreds of flags [2] covering various configuration related to the program performance, i.e., garbage collection, interpretation, and compilation. We went through all the flags and choosed six flags to evaluate the performance of HotSpot on various kinds of workload.

1. Default. Default means run the original HotSpot without any flag. *By default, the JVM performs 10,000 interpreted method invocations to gather information for efficient compilation* [3]. Thus HotSpot will run the method using interpretation in the first 10,000

calls. In the same time, a thread will run in background and compile the code to native code. After that, HotSpot will run the program in native code to achieve speedup in hot method and avoid unnecessary compilation in *cold* method, which means the method that does not run many times.

2. -Xbatch. This flag will disable background compilation so that compilation of all methods proceeds as a foreground task until completed. This would introduce a little overhead since concurrent compilation is disabled and the thread running the program must wait for the compilation to be completed.
3. -XX:+AggressiveOpts. This flag enables the use of aggressive performance optimization features.
4. -Xint: This flag indicates running the program in interpreted-only model and compilation to native code is disabled. Using this flag we can compare the performance with compilation and without compilation to show the importance of compilation in speeding up hot methods.
5. -Xcomp: This flag will force the compilation of methods on first invocation. In other word, not only hot methods but also the cold methods will be compiled, while the compilation of cold methods is unnecessary and will introduce a heavy overhead, especially if there are lots of cold methods.
6. -Xnoclassgc: This flag disables garbage collection of classes to save some garbage collection time. For programs with little memory allocation and deallocation, garbage collection will not happen thus this flag will not show effect. However, for programs with heavy workload on memory, this flag may make dramatic difference.

Section 3. Selected programs

For comparing flags, we argue that different flags favor different side in the trade-off and there is no flag performs consistently better than others under all kinds of workload. Thus selecting programs covering various workload is critical for evaluating the performance. To better evaluate the effect of flags, we selected eleven programs covering various kinds of workload, including calling a hot methods tens of thousands of times, allocating and deallocating memory frequently, as well as introducing heavy workload IO.

1. The first group of programs contains sqrt() and pow(x, 0.5). This group is interesting because they are both built-in functions and they perform similar job in different ways. We expect sqrt() to be faster than pow(x,0.5) since pow(x,0.5) could do more tasks.
2. The second group of programs contains *Binary-tree,* which will fully create perfect binary trees. This program will introduce a heavy workload on allocation and deallocation, thus is a good way to test garbage collection.
3. The third group contains programs performing various tasks on DNA, i.e., generating a long sequence of DNA and store the data into a file and generating the complement of a given DNA sequence.
4. We also selected a group of programs running different math algorithms, i.e., calculating eigenvalue, calculating pi to a given precision, and so on.

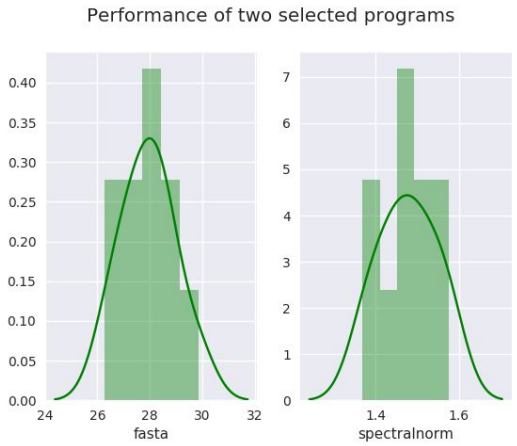

Figure 1: Variance of performance

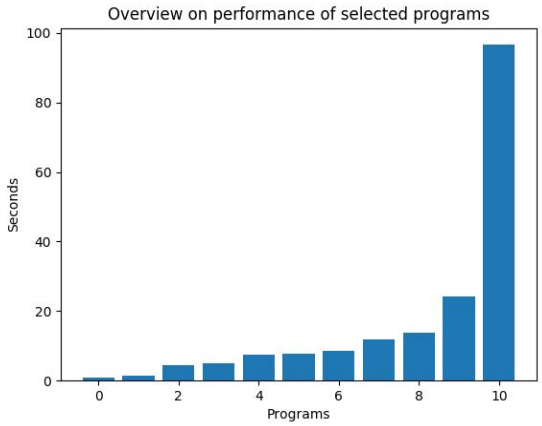

Figure 2: Performance of selected programs

Section 4. Experiments

In this section, we are going to show the interaction between flags and various workloads. First, we show the existence of variance in program performance with same flag and same machine. Second, we exhibit the performance of selected programs to show that these programs have a wide range of performance and thus compose a valid benchmark for evaluating flags. We proceed to show that disabling compilation and allowing only interpretation would introduce a disaster in most of programs. Finally, we show that compilation all methods including those cold methods is also a bad idea. A mixing of interpretation and compilation would give better performance under various kinds of workload.

4.1 Experimental settings

The performance is measured in running time, which could be generated using the built-in function “System.currentTimeMillis()”. To eliminate the effect of randomness, every combination of program and flag is measured for 5 times and the average running time is reported. The performance is measured on a quad-core 2.2 GHz Intel i7-6560U CPU with 8GB of RAM and a 220GB SSD with Ubuntu 16.04.4 LTS.

4.2 Experiments on the variance of performance

Using average running time instead of the result from a single run is important due to the huge variance in the program performance even with the same flag and same machine. In this section, we will show the existence of variance in program performance, see fig. 1. Two programs, fasta and spectralnorm, under default mode are run for 10 times to simulate the distribution of performance. X axis stands for the performance of program measured in seconds. We can see that the performance of same program with the same flag and on the same machine may still vary a lot. There is a 4 seconds variance for fasta between the maximum and

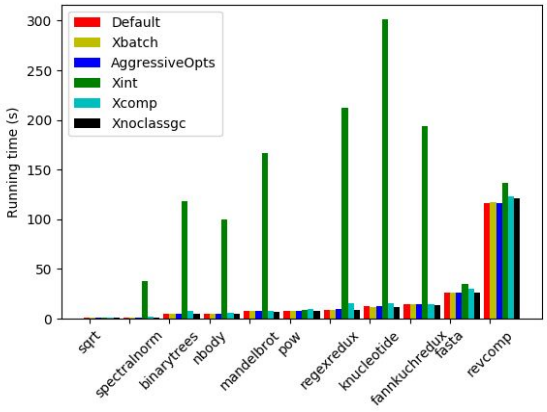

Figure 3: Performance by programs and flags

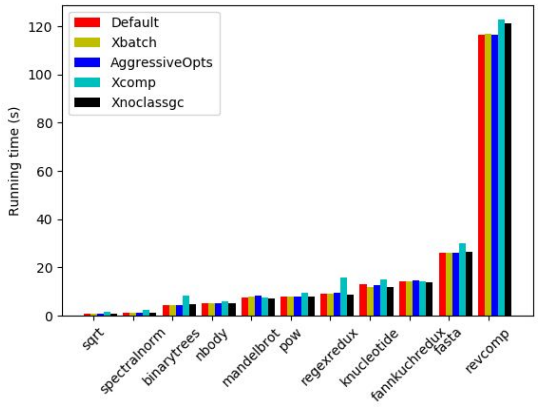

Figure 4: Performance without xint

variance for spectrainorm decreases to 0.2 seconds while the average running time also decrease to 1.5 seconds. This decrease in variance is due to the decrease in average running time instead of program since the ratio between variance and average running time is 15% for both programs. Thus we conclude that a variance of program running time appear naturally and a small difference between average running time for different flags does not necessarily indicate an intrinsic performance difference between flags.

4.3 Running time of selected programs

We select programs with a wide range of running time, see fig 2. The program with smallest running time is sqrt, which takes less than 1 seconds with default mode. The program with highest running time is revcomp, which lasts for around 100 seconds. We argue that a wide range of running time is important to the validity of the comparison.

4.4 Performance by programs and flags

In this section, we display the performance of various flags grouped by programs, see fig. 3 and fig. 4. To eliminate the effect of randomness, every combination of program and flag is measured for 5 times and the average running time is reported. The x axis represents the name of programs and y axis is the running time in seconds. Each group stands for the results for a single program and the six bars inside each group represent the six flags. Across different groups, same flag has same color.

From fig. 3, we can see that the most significant result is the green bar, which stands for Xint, only interpretation without a single compilation. For most of the program, this flag would destory the performance totally. For example, the running time of knucleotide without this flag is around 12 seconds while the number with this flag increase to around 300 seconds, which is 30 times slower than the former one. Most of the other program also exhibits a similar behavior. This observation coincide with theoretical analysis. Flag -xint disables compilation at all and will not

allow HotSpot to compile the code into native code. So the code could only be run in interpretation way, which is much slower than running native code. Although it is fine for cold methods, it would harm the performance a lot for these hot ones, since the large overhead of interpretation would appear again and again.

To compare other flags better, we deleted performance for xint and only display results for other flags, see fig. 4. We find that the blue line becomes higher than others frequently, which represents “force the compilation of methods on first invocation”. This flag would introduce a little bit overhead consistently. For example, in regexredux, the average running time without this flag is 8.9 seconds while the number with this flag is 15.955, which is almost twice as the former one. From the point view of theory, forcing the compilation of methods on first invocation would force the compilation of both hot methods and cold methods. For hot methods, this does not hurt while it does hurts a lot for cold methods, since the cold methods could be called in an interpretation way. The overhead of compilation for cold methods cannot be made up by the decrease in running time since these methods are called too little times.

Besides these two flags, other flags performs similar on our selected programs, especially when we consider the variance on performance shown in section 4.2. We can conclude that neither only using interpretation nor compiling all the methods in the first invocation is a good way and may both hurt the program performance.

Section 5. Conclusion

In this project, we evaluate the performance of six HotSpot flags on eleven programs. The six flags cover various configurations, like interpretation-only, compilation-only, mixed method, and no garbage collection. Through experiments, we found that the mixed method performs consistently better than others and the garbage collection introduces little overhead.

Reference

[1] The Computer Language Benchmark Games
https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/java.html

[2] Summary of HotSpot flags.
http://www.oracle.com/technetwork/articles/java/vmoptions-jsp-140102.html

[3] Java Platform, Standard Edition Tools Reference
https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html