Optimization of Java Virtual Machine Flags using Feature Model and Genetic Algorithm

Work-In-Progress Paper

Felipe Canales
felipe.canales.c@ing.uchile.cl
ISCLab, Department of Computer Science, University of Chile

Geoffrey Hecht
ghecht@dcc.uchile.cl
ISCLab, Department of Computer Science, University of Chile

Alexandre Bergel
abergel@dcc.uchile.cl
ISCLab, Department of Computer Science, University of Chile

ABSTRACT

Optimizing the Java Virtual Machine (JVM) options in order to get the best performance out of a program for production is a challenging and time-consuming task. HotSpot, the Oracle's open-source Java VM implementation offers more than 500 options, called flags, that can be used to tune the JVM's compiler, garbage collector (GC), heap size and much more. In addition to being numerous, these flags are sometimes poorly documented and create a need of benchmarking to ensure that the flags and their associated values deliver the best performance and stability for a particular program to execute.

Auto-tuning approaches have already been proposed in order to mitigate this burden. However, in spite of increasingly sophisticated search techniques allowing for powerful optimizations, these approaches take little account of the underlying complexities of JVM flags. Indeed, dependencies and incompatibilities between flags are non-trivial to express, which if not taken into account may lead to invalid or spurious flag configurations that should not be considered by the auto-tuner.

In this paper, we propose a novel model, inspired by the feature model used in Software Product Line, which takes the complexity of JVM's flags into account. We then demonstrate the usefulness of this model, using it as an input of a Genetic Algorithm (GA) to optimize the execution times of DaCapo Benchmarks.

KEYWORDS

Java Virtual Machine, Optimization, Auto-tuning, Feature Model, Genetic Algorithm

ACM Reference Format:
Felipe Canales, Geoffrey Hecht, and Alexandre Bergel. 2021. Optimization of Java Virtual Machine Flags using Feature Model and Genetic Algorithm: Work-In-Progress Paper. In *Companion of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21 Companion), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3447545.3451177

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICPE '21 Companion, April 19–23, 2021, Virtual Event, France
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8331-8/21/04.
https://doi.org/10.1145/3447545.3451177

1 INTRODUCTION

Oracle's HotSpot JVM latest release, distributed with Java SE 15, provide a list of 528 flags considered as final (i.e., excluding experimental, diagnostic and others specifics set of flags) via the command `java -XX:+PrintFlagsFinal -version`. Hundreds of these flags can be used to optimize the performance of the JVM by tuning the behaviors of various internal processes including memory management, garbage collection process, and compiler behaviors. For example, the `-Xmx2G` flags set the maximum size of the heap to 2 GB of RAM, whereas the `-XX:+UseParallelGC` flags specify that the JVM should use the parallel garbage collector.

Although in general, the default parameters of the JVM are enough to run an application, the task becomes more complicated when practictioners want to optimize the performance of their program. Indeed, the large quantity of flags can take a plethora of values (boolean, integrer or string) which must be adapted to the application and the system that runs it. A great deal of expertise is therefore required, especially since not all the options are properly documented in the official documentation, sometimes requiring to dive into the source code of the JVM [9] or other experts website to understand them properly.

An example of such configuration done manually can be found in Listing 1. This popular configuration for a Minecraft game server aims at optimizing the use of the memory and the garbage collector. According to its author, it is "After many weeks of studying the JVM, Flags, and testing various combinations [...] the result of a ton of effort and results of seeing it in production on various server sizes, plugin lists and server types" [6].

Previous approaches have already been proposed to automatize this process of optimizations as showed in Section 2. However, their mechanisms to handle the dependencies and incompatibilities of the various flags are limited. As a consequence, the search space of flags is not optimal which makes harder and more time consuming the discovery of the best configurations of flags.

Consider Listing 1 as the result of a manual flag tuning of a Minecraft server. The presented configuration of the JVM is complex, for example, the `-XX:G1HeapRegionSize=8M` flags is only valid because the `-XX:+UseG1GC` is also used in the same configuration. Also the `-XX:+UseParallelGC` flag (as well as all the flags who need it as a prerequisite) could not be used in this configuration since it is conflicting with `-XX:+UseG1GC`. Such restrictions are not apparent in the configuration and therefore solely rely on a deep and necessary knowledge of the performance engineer.

Previous studies [9, 12] also present results for OpenJDK7 or OpenJDK8 which are now outdated. We expect our work to allow

for replicating previous results from the performance engineering community but on a newer JVM.

Our approach. To solve this problem, we propose to organize the flags of the JVM as it is done for features in feature model. We then use this convenient model as an input for a genetic algorithm capable of interpreting its specificities. We have experimented our approach on several benchmarks of the DaCapo benchmark suite [2]. Our genetic algorithm determined a list of options that improve some benchmark executions by up to 23% when compared to the default parameters of the JVM.

```
java −Xms10G −Xmx10G −XX:+UseG1GC −XX:+AlwaysPreTouch
−XX:MaxGCPauseMillis=200 −XX:+UnlockExperimentalVMOptions
−XX:+DisableExplicitGC  −XX:+ParallelRefProcEnabled
−XX:G1NewSizePercent=30 −XX:G1MaxNewSizePercent=40
−XX:G1HeapRegionSize=8M −XX:G1ReservePercent=20
−XX:G1HeapWastePercent=5 −XX:G1MixedGCCountTarget=4
−XX:InitiatingHeapOccupancyPercent=15
−XX:G1MixedGCLiveThresholdPercent=90
−XX:G1RSetUpdatingPauseTimePercent=5 −XX:SurvivorRatio=32
−XX:+PerfDisableSharedMem −XX:MaxTenuringThreshold=1
−Dusing.aikars.flags=https://mcflags.emc.gs
−Daikars.new.flags=true −jar paperclip.jar nogui
```

Listing 1: Example of Hotspot Flags recommended for a Minecraft Server

Outline. This paper is structured as follows. Section 2 gives a brief overview of the related work. Section 3 describes our approach to employ a feature model and a genetic algorithm to optimize the JVM configuration for a particular software execution. Section 4 details the experiments we conducted. Section 5 concludes and outlines our future work.

2 RELATED WORK

A number of automatic optimization or auto-tuning approaches have been proposed for various domains such as compilers [10], runtime systems [16] or deep learning [5]. There is even framework such as ParamILS [8] or OpenTuner [1] allowing one to build domain-specific and customizable program autotuners.

Various works have focused on the optimizations of the JVM flags, focusing on a subset of specialized flags carefully selected [3, 4, 11, 14, 15], thus avoid the need of modeling the dependencies and incompatibilities between flags. Two studies [9, 12] are closely related to our work since they tackle a wider range of flags.

Automatic tuning of GC parameters. Lengauer and Mössenböck [12] proposed an approach, built-on top of ParamILS [8] and its hill climbing, to automatically optimize all GC flags available for OpenJDK8. They were able to consistently improve the performance of benchmarks of DaCapo and SPECjbb 2005. They provide an in-depth analysis of their results, useful to guide practitioners willing to optimize their programs. To model dependencies between flags, they use the mechanism of conditional parameters provided by ParamILS, which allows a flag B to be set only if a flag A has certain values (typically when set to true). However, the authors mention that this mechanisms may still lead to improper configurations (called false positive), since with the hill climbing approach, the value of the flag B might be set and kept between iterations while the flag A has been switched to an incompatible value. ParamILS also allows for the definition of forbidden combinations of parameters but the authors did not mention using it. It seems impractical to use in the case of Java flags since it requires to list all the combinations of forbidden values (e.g., `{flagA=valueA1,flagB=valueB1,...}` `), {flagA=valueA2,flagB=valueB2,...}`.

HotSpot Auto-tuner. Jayasena et al. [9] proposed a similar approach named *HotSpot Auto-tuner* but this time considering all the flags available in OpenJDK7 and relying on OpenTuner [1]. They were able to improve the performance of SPECjvm2008 and DaCapo benchmarks, they found that mixing all types of flags (i.e., not only using a subset of GC or compiler-related flags) provided the best results. Interestingly, with the Dacapo Benchmark the performance worsen with the subset of GC flags, contrary to what was achieved by Lengauer and Mössenböck [12]. By default, OpenTuner proposes a more sophisticated approach than ParamILS to explore the search space, relying on an AUC Bandit meta technique which combines greedy mutation, differential evolution, and two hill climber algorithms. However it lacks mechanisms to handle dependencies and incompatibilities between flags, therefore reducing the effectiveness of the tuning process. To overcome this problem, the authors propose to use a hierarchy of flags which groups together alternative flags (e.g., `-XX:+UseParallelGC` and `-XX:+UseG1GC`). Although this approach helps to explore the search space more rapidly, it is still prone to generate incorrect configurations of flags. Indeed, it cannot express prerequesite between flags, e.g., `-XX:UseAdaptiveSizePolicy` is only valid with `-XX:+UseParallelGC` or `-XX:+UseG1GC` but not the alternatives GC.

3 MODELING FLAGS USING FEATURES AND GENETIC ALGORITHM

In order to solve these problems of dependencies and incompatibilities between flags, we propose a model of the flags inspired by a feature model.

Feature model. Figure 1 describes a feature model of the JVM flags related to the garbage collector (GC). Our feature model also relies on a hierarchy (as with *HotSpot Auto-tuner*), but it offers a better granularity and flexibility by supporting:

- *Alternative* between flags, e.g., the choice of a unique garbage collector;
- *Combination* of flags, e.g., Uncommit and CollectionInterval could be both used in some configurations;
- *Mandatory* or *optional* flags, e.g., a garbage collector must be used, in most cases a default value is set by the JVM, whereas Gensize flags are not always required;
- Constraints between flags using *logical operators* such as implies ($\implies$), and ($\wedge$), or ($\vee$) and negation ($\neg$). For example, the ZGC is incompatible with AdaptiveSizePolicy and SurvivorRatio since ZGC uses only one generation for the whole heap.

Flag optimization. We have chosen to experiment the potential of this model with a Genetic Algorithm (GA). GA is inspired by natural selection, in its simple form it consists of: genes representing properties or values, individual corresponding to a chain of genes and generations which regroups a number of individuals. The first generation of individuals are created from the random selection

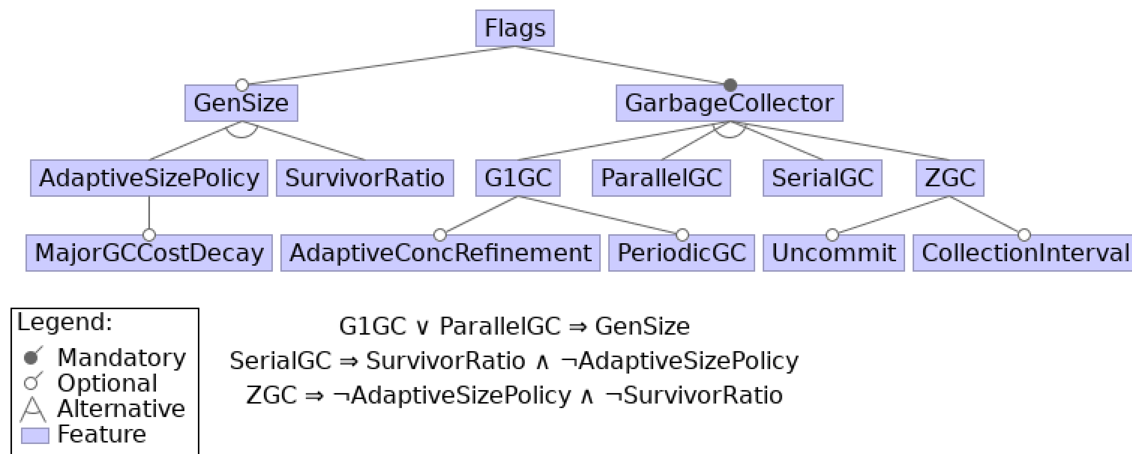

Figure 1: Partial Feature Model of flags related to garbage collector

of values for each gene. Then, the performance of each individual is evaluated by calculating an objective function. Subsequently, the worst performers of the generation are then discarded and the new generation is obtained by creating new individuals from the crossover of the gene of the best performing individuals. After this, a mutation step is usually performed on some of the new individuals to add variability to the population.

We decided to use GA to model flags as follows: a gene represents a flag and the gene value represents the value of the flag (e.g., boolean, integer or decimal). An individual in our GA is an ordered collection of flags. The objective function takes as argument a collection of flags and returns a numerical fitness value, calculated by measuring the running time of a program launched with the provided set of flags. In addition, to the necessity of indicating the type and range of values that the flags can take, we extended the sequence of genes with a set of *control genes* as a means to express the constraints of our particular model.

The control genes allow us to form groups of alternate genes, when the control gene is active then a variant of the gene is chosen (e.g., a variant of GC). They are therefore a way to implement alternative of the feature model (see Figure 1), whereas combination of flags is simply the chain of genes. Futhermore, to implement the other constraints of our model, when a gene is selected it may disable another gene (regular or control) to express incompatibilities between flags. Similarly, a gene may enable other genes to express the implies and mandatory relationships of our model. Since the JVM often provides default values for most of the mandatory relation, this relation between gene is sometimes implicit.

For example, considering the model of Figure 1, a control gene represents the alternative between GarbageCollectors. If SerialGC is selected for activation during the creation of an individual then AdaptiveSizePolicy is automatically disabled, whereras SurvivorRatio is enabled.

4 EXPERIMENTS

We decided to experiment our approach on the DaCapo benchmarks [2] version 9.12 which is commonly used for Java benchmarking, especially in efforts related to ours [9, 12]. From the various benchmarks offered by DaCapo, only *avrora, fop, jython, luindex, lusearch, lusearch-fix, pmd and xalan* are used. Those not mentioned present stability problems with some flags (including some default flags).

Our goal of our experiment is to assess whether our approach is able to minimize the execution time of the benchmarks. Since it is a work-in-progress, so far, we have modeled 70 of the 528 flags, focusing on GC flags. Our future work contemplates incorporating the remaining flags. The genetic algorithm is run with a population of 32 individuals along ten generations. The 16 least performing individuals are discarded to generate the next generation using crossover of individuals. Moreover when a gene is activated we kept the default value proposed by the JVM in 20% of the cases, otherwise a random value is generated according to the min, max and step values provided in a JSON file.

The experiment was run with the following configuration: Hotspot of Java SE 15.0.2 on Windows 10 Professional (64 bits) with an AMD Ryzen 2600X 3.6GHZ (6 cores) and 16 GB of RAM memory.

During the execution of GA, for each individual the benchmark was run twice after the Dacapo warmup phase, and the average of these two runs was used to rank individuals for the next generation. At the end of the 10 generations, the best performing individual is considered as our optimized set of flags.

We decided to compare its performance with the benchmark executed with the default options of the JVM. In order to obtain rigorous comparison between the two configurations of flags, each of the configurations (optimized and default) was executed 30 times after warmup as recommended in previous work [7].

Table 1 gives the results of the experiments. In all cases our approach was able to reduce the execution time of the benchmark. For all benchmarks, an independent samples t-test gave us a two-tailed P value of less than 0.0001, confirming that the differences are statistically significant. We can therefore conclude that in our

Benchmark	Average time (ms)		Percentage of	Cohen's d
	Default	Optimized	Improvement	effect size
avrora	3552.87	3506.23	1.31%	3.75 (h)
fop	198.07	151.67	23.43%	4.50 (h)
jython	2778.07	2690.70	3.14%	3.02 (h)
luindex	1460.17	1434.63	1.75%	1.08 (l)
lusearch	389.57	351.50	9.77%	4.19 (h)
lusearch-fix	389.17	348.30	10.50%	4.38 (h)
pmd	1317.30	1244.37	5.54%	4.11 (h)
xalan	461.90	420.30	9.01%	1.67 (vl)

Table 1: Comparison of the execution time of the benchmarks with and without optimization on 30 iterations. (l) means that the effect size is large, (vl) means very large and (h) means huge

setting, these results are not by chance, our approach is able to reduce the benchmark execution time for all the benchmarks.

As an effort to estimate the strength of the statistical difference we found, we use (i) the Cohen's d effect size with a confidence 95% confidence level and (ii) the descriptor of magnitude recommended by Sawilowsky [13]. The benchmark *luindex* has the smallest effect size but it could still be described as a large effect size, whereas *xalan* is very large and the others benchmarks all have a huge effect size. Therefore, even if the magnitude of improvements vary between benchmarks, it is never marginal, on the contrary, the reduction of the execution time is always noticeable. This is particularity true for the *fop* benchmark with 23.43% percentage of improvements on the average execution time. The Listing 2 shows the set of flags selected by our approach, we can observe that the ParallelGC was selected as well as an AdaptivePolicy and multiple flags related to this policy. This is consistent with the model presented in Figure 1.

```
-XX:+UseParallelGC -XX:+UseAdaptiveSizePolicy
-XX:-UseAdaptiveSizeDecayMajorGCCost
-XX:+UseAdaptiveSizePolicyWithSystemGC
-XX:-UseAdaptiveGenerationSizePolicyAtMajorCollection
-XX:AdaptiveSizePolicyCollectionCostMargin=25
-XX:AdaptiveSizePolicyInitializingSteps=25
-XX:AdaptiveSizePolicyWeight=0
-XX:AdaptiveSizeThroughPutPolicy=0
-XX:GCTimeRatio=99 -XX:MaxGCPauseMillis=150
-XX:TenuredGenerationSizeIncrement=15
-XX:TenuredGenerationSizeSupplement=80
-XX:YoungGenerationSizeIncrement=20
-XX:AdaptiveSizeDecrementScaleFactor=3
-XX:-UseDynamicNumberOfGCThreads
-XX:-UseMaximumCompactionOnSystemGC
-XX:HeapMaximumCompactionInterval=20
-XX:ParallelGCBufferWastePct=15
```

Listing 2: Optimized set of flags selected by our approach for the *fop* benchmark

5 CONCLUSION AND FUTURE WORKS

In this paper, we proposed an approach to automatically optimize the set of flags for a specific application running on a JVM. The novelty of this approach is in the use of a feature model to describe the dependencies and incompatibilities between the numerous flags of the JVM. In addition to avoiding spurious configurations of flags, our model can reduce the search space for an optimal solution. This model is then used as in input of genetic algorithm, which was consistently able to strongly improve the performance of multiple DaCapo benchmarks.

Encouraged by these positive results, our future work includes extending the set of flags considered by our approach and integrating it to existing frameworks of auto-tuning. This will allow us to compare the efficiency of auto-tuner with or without our model. Futhermore, we plan to test our approach on more benchmarks and applications. In particular, we want to see if our approach is capable of matching or even surpassing the performance of a set of flags that has been manually tuned by an expert.

ACKNOWLEDGMENTS

This work is supported by Proyecto ANID/FONDECYT Postdoctorado N°3180561, ANID/FONDECYT Regular project 1200067, and Lam Research.

REFERENCES

[1] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.

[2] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 169–190.

[3] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham. 2001. Controlling garbage collection and heap growth to reduce the execution time of Java applications. *ACM Sigplan Notices* 36, 11 (2001), 353–366.

[4] Guangyu Chen, R Shetty, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, and Mario Wolczko. 2002. Tuning garbage collection in an embedded Java environment. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*. IEEE, 92–103.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[6] Daniel Ennis. 2020. Tuning the JVM – G1GC Garbage Collector Flags for Minecraft. https://aikar.co/2018/07/02/tuning-the-jvm-g1gc-garbage-collector-flags-for-minecraft/. [Online; accessed 22 January-2021].

[7] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices* 42, 10 (2007), 57–76.

[8] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36 (2009), 267–306.

[9] Sanath Jayasena, Milinda Fernando, Tharindu Rusira, Chalitha Perera, and Chamara Philips. 2015. Auto-tuning the java virtual machine. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 1261–1270.

[10] Herbert Jordan, Peter Thoman, Juan J Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. 2012. A multi-objective auto-tuning framework for parallel codes. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[11] Iffat H Kazi, Howard H Chen, Berdenia Stanley, and David J Lilja. 2000. Techniques for obtaining high performance in Java programs. *ACM CSUR* 32 (2000), 213–240.

[12] Philipp Lengauer and Hanspeter Mössenböck. 2014. The taming of the shrew: increasing performance by automatic parameter tuning for java garbage collectors. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. 111–122.

[13] Shlomo S Sawilowsky. 2009. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods* 8, 2 (2009), 26.

[14] Jeremy Singer, Gavin Brown, Ian Watson, and John Cavazos. 2007. Intelligent selection of application-specific garbage collectors. In *Proceedings of the 6th international symposium on Memory management*. 91–102.

[15] Jeremy Singer, George Kovoor, Gavin Brown, and Mikel Luján. 2011. Garbage collection auto-tuning for java mapreduce on multi-cores. *ACM SIGPLAN Notices* 46, 11 (2011), 109–118.

[16] Cristian Tapus, I-Hsin Chung, and Jeffrey K Hollingsworth. 2002. Active harmony: Towards automated performance tuning. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. IEEE, 44–44.