Towards Benchmark Classification for GC Tuning

Olivier Flückiger

April 12, 2018

Abstract

There are many GC algorithms described in the literature and used in practice. However, an often overlooked detail are the heuristics, that all of them need to rely on. When to grow the heap? How aggressively to shrink it again? Trigger a GC cycle early? There are many knobs to turn, but a distinct lack of principled advice on how to do so. In this article we propose a methodology for clustering programs by similar behavior, from the point of view of the GC. We find that this method succeeds at identifying similarly behaving programs, but as such is not strong enough to predict the performance of a certain choice of parameters.

1 Introduction

A typical GC implementation has several tuneable parameters, such as heap grow rate, various thresholds for triggering collection cycles, and so on. Picking an optimal set of parameters is a multi-dimensional optimization problem, and the resulting default parameters are always a compromise. To better understand GC tuning, we propose to develop a methodology for classifying benchmarks (also called mutators) according to their behavior under varying GC parameters.

There are several applications of such a classification. For a fair and unbiased comparison, we want to ensure, that we covered all facets of a GC algorithm. For efficient testing, we might want to run the programs with the most nuanced behaviors first. Also, we can provide more insightful evaluations, if we report performance numbers for each class. Finally, if we know which classes of programs exist, we can detect them and adjust tuning parameters at runtime, overcoming the one size fits all nature of ahead-of-time GC tuning.

To cluster mutators we proceed in two steps. First, we idealize their response to changing GC heuristics, by fitting an SVM model. Second, we automatically cluster the generated models.

All the presented results in this article are based on the latest version of the reference R bytecode interpreter [R Core Team, 2017] and a standard R benchmark suite called *rbench*[1]. The R GC uses a traditional treadmill [Baker, 1992] design. We can't say how the results would generalize to other algorithms. Source code and raw data can be found at[2].

[1] https://github.com/rbenchmark/benchmarks

[2] https://github.com/o-/ml_hw/tree/master/project

2 Technical Approach

This section describes our data gathering, the preprocessing and the clustering approach. As a first step, we gather benchmark execution times over a grid of GC parameter values. In this experiment we looked at the following set of parameters:

R_NGrowFrac The occupancy percentage trigger value for heap growth.

R_NGrowIncrFrac The heap growth factor.

R_MinFreeFrac The trigger value for an old generation collection cycle.

R_MaxKeepFrac How many empty pages to retain.

The benchmarks where executed on a Lenovo X220 machine with an Intel Core i7-3520M CPU with a clock cycle of 2.90GHz and 16Gb of RAM running Fedora Linux version 27. For each benchmark we identified the fastest execution time. We define a run to be *fast* if it is within 3% of the fastest one and *slow* otherwise. Thus our input data looks as follows:

$$\text{Program}, \quad P_1, \quad P_2, \quad \ldots, \quad P_n, \quad \text{Time}, \quad \text{Slow/Fast}$$

Then we train an SVM classifier, to predict *fast* vs. *slow* for a given set of GC parameters. We use the SVM classifier by [Chang and Lin, 2011] from the *e1071* R package with a radial basis kernel. Hyperparameter selection is done by autotuning using gridsearch with 10-fold cross validation. As an example see the preprocessing for the binary trees benchmark in Figure 1. Several benchmarks are not sensitive to changing GC parameters, since, for example, they do not allocate a significant number of objects. We do not generate SVM models for benchmarks where more than 95% of all runs are *fast*.

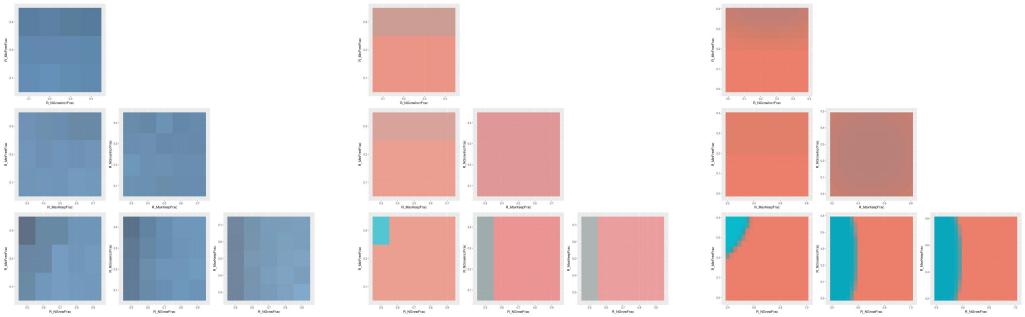

Figure 1: The four dimensional feature space in the main projections. From left to right: execution time (darker means longer), classification fast or slow, SVM prediction.

In a next step we use the k-means algorithm to cluster benchmarks with similar models. We use the algorithm by [Hartigan and Wong, 1979], implemented in the *NbClust* R package.

3 Experimental Results

The collection of benchmark results for the grid search through all GC parameters is a rather slow process. In the time available we managed to collect and incorporate

roughly 36′000 executions. We evaluate 92 benchmarks, therefore there are roughly 400 measurements per benchmark.

As for the SVM models to describe each individual benchmark, due to the noisy nature of time measurements, the observed classification errors range between 5 and 20%.

In the resulting classification, there are two very large clusters. First, the cluster of allocators which have no correlation with the GC parameters. Those are the ones we excluded in the preprocessing phase. Second, the cluster of very noisy and weakly correlated allocators. Unfortunately those make up over half of all investigated benchmarks. See the appendix for a list of the identified clusters.

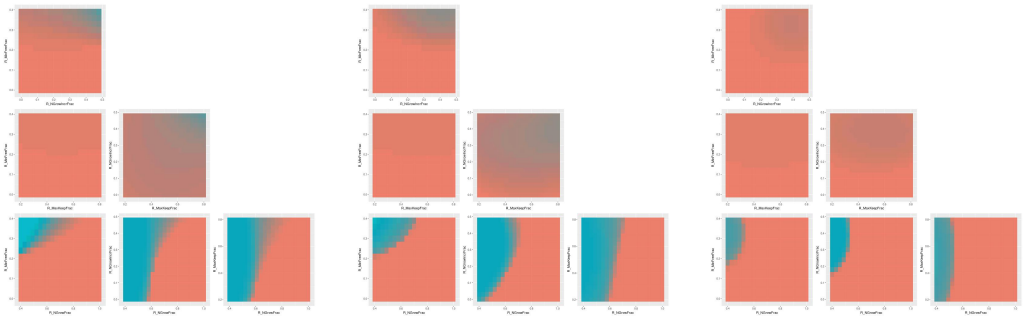

Figure 2: Cluster 7: binary-trees_2, binary-trees_list, ica_lapply.

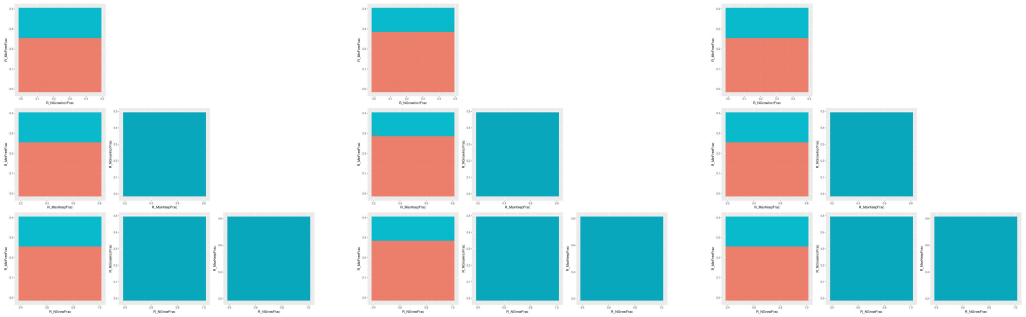

Figure 3: A sample of cluster 9: black_scholes, mandelbrot-ascii and raysphere.

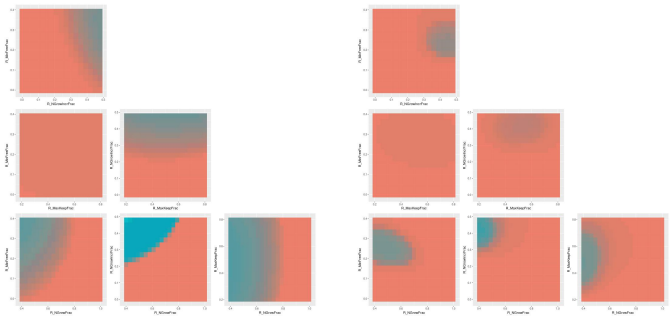

Figure 4: Cluster 14: LR-1var_ols_lapply and Pi_lapply.

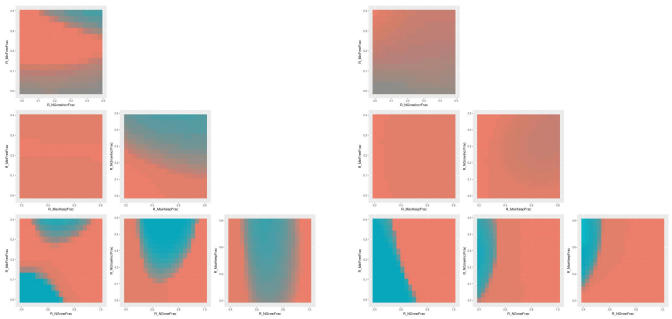

Figure 5: Miss classified cluster 5: k-means_lapply and k-means-1D_lapply.

We need a very high value of $k = 15$ and most interesting clusters turned out to be very small. A number of them is however quite interesting, such as cluster 7 in Figure 2, cluster 9 in Figure 3, cluster 14 in Figure 4 and the miss classified[3] cluster 5 in Figure 5.

4 Evaluation

To evaluate the performance of the classification, we performed a synthetic GC parameter tuning. In particular we tested the change from *R_NGrowFrac R_NGrowIncrFrac* from 0.7 and 0.1 to 0.2 and 0.2. *R_MinFreeFrac* and *R_MaxKeepFrac* where kept at their default value of 0.1 and 0.5.

Our hypothesis was, that clustered benchmarks would show a similar speedup. However, the prediction accuracy was underwhelming and we could not find much support for this hypothesis.

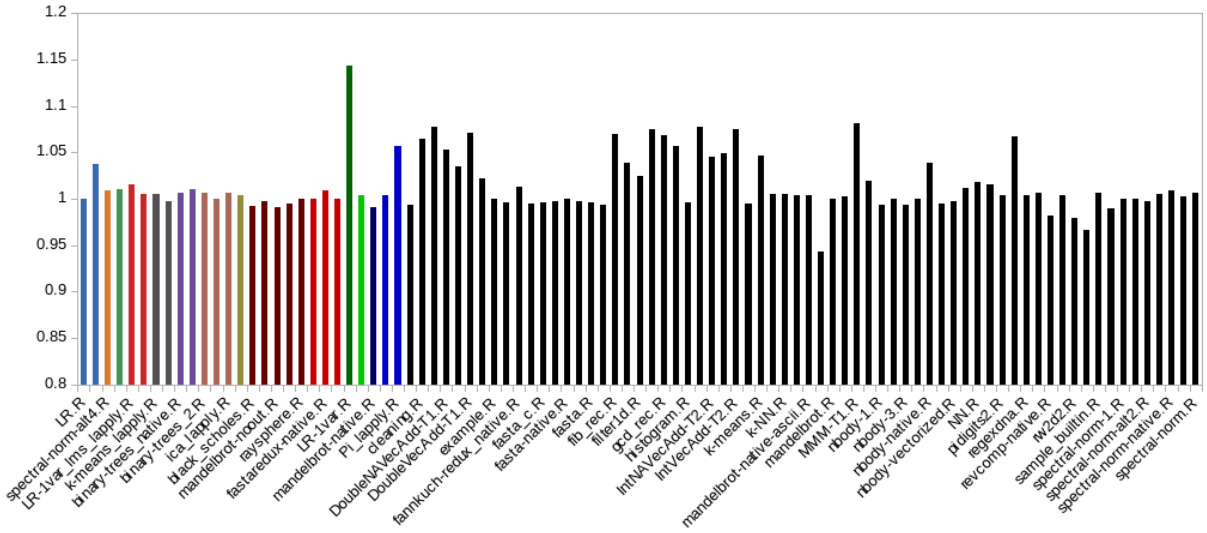

Figure 6: Speedup of an artificial GC parameter change. The black class are the ones classified as noisy or non correlated.

[3]As can be seen, those two benchmarks have been more accurately modeled since the final presentation.

The relative speedup of this GC tuning is shown in Figure 6. As an explanation for our result we would like to point out the blue class, left of the big black cluster. There are two benchmarks with roughly 10% difference in runtime, but in the same class. As a matter of fact, they represent cluster 5 from Figure 5. It seems, even with the high number of k, we still end up clustering very different benchmarks into similar categories.

In conclusion we believe that most benchmarks exhibit unique or unpredictable behavior. Even for benchmarks with similar responses, small differences in the *slow*, *fast* boundary can have big effects in practice. Therefore, we think it is mostly not the case, that one benchmark can subsume another one. Based on those findings we do not recommend retiring or skipping benchmarks, and conclude that more work is necessary to accurately predict the effects of GC heuristics changes.

References

[Baker, 1992] Baker, H. G. (1992). The treadmill: real-time garbage collection without motion sickness. *ACM Sigplan Notices*, 27(3):66–70.

[Chang and Lin, 2011] Chang, C.-C. and Lin, C.-J. (2011). Libsvm: a library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):27.

[Hartigan and Wong, 1979] Hartigan, J. A. and Wong, M. A. (1979). Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108.

[R Core Team, 2017] R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

5 Appendix

Identified Clusters:

Not correlated to GC parameters

- binary-trees.R,
- cleaning.R,
- crt.R,
- DoubleNAVecAdd-T1.R,
- DoubleNAVecAdd-T2.R,
- DoubleVecAdd-T2.R,
- example.R,
- fannkuch-redux_native.R,
- fasta-2.R,
- fasta-native2.R,
- filter1d.R,
- ForLoopAdd.R,

- IntNAVecAdd-T1.R,
- IntNAVecAdd-T2.R,
- IntVecAdd-T1.R,
- IntVecAdd-T2.R,
- k-means.R,
- k-NN_lapply.R,
- k-NN.R,
- mandelbrot-noout-native.R,
- mandelbrot.R,
- MMM-T3.R,
- nbody-1.R,
- nbody-2.R,
- nbody-3.R,
- nbody-native.R,
- nbody-native2.R,
- prime.R,
- regexdna.R,
- revcomp-1.R,
- revcomp-native.R,
- rw2d2.R,
- rw2d3.R,
- sample_builtin.R,
- sample.R,
- spectral-norm-1.R,
- spectral-norm-alt.R,
- spectral-norm-math.R,
- spectral-norm-native.R,
- spectral-norm.R,

Weak or very noisy correlation

- DoubleVecAdd-T1.R,
- fannkuch-redux_2.R,
- fannkuch-redux.R,
- fasta_c.R,
- fasta-native.R,
- fasta.R,
- fastaredux.R,
- fib_rec.R,
- fib.R,
- gcd_rec.R,
- gcd.R,

- histogram.R,
- k-means-1D.R,
- LR_lms_lapply.R,
- mandelbrot-native-ascii.R,
- meteor-contest.R,
- MMM-T1.R,
- nbody-vectorized.R,
- NN_lapply.R,
- NN.R,
- pidigits.R,
- pidigits2.R,
- rw2d1.R,
- spectral-norm-alt2.R,
- spectral-norm-vectorized.R,

Cluster 1 with

- LR.R,
- MMM-T2.R

Cluster 2 with

- spectral-norm-alt4.R

Cluster 3 with

- LR-1var_lms_vec.R

Cluster 4 with

- LR-1var_lms_lapply.R,
- spectral-norm-alt3.R

Cluster 5 with

- k-means_lapply.R,
- k-means-1D_lapply.R

Cluster 6 with

- binary-trees_native.R,
- smv.R

Cluster 7 with

- binary-trees_2.R,
- binary-trees_list.R,
- ica_lapply.R

Cluster 8 with

- smv_builtin.R

Cluster 9 with

- black_scholes.R,

- mandelbrot-ascii.R,
- mandelbrot-noout.R,
- mandelbrot1.R,
- raysphere.R

Cluster 10 with

- fasta-3.R,
- fastaredux-native.R,
- nbody.R

Cluster 11 with

- LR-1var.R

Cluster 12 with

- LR_ols_lapply.R

Cluster 13 with

- mandelbrot-native.R

Cluster 14 with

- LR-1var_ols_lapply.R,
- Pi_lapply.R