Time Matters: Minimizing Garbage Collection Overhead with Minimal Effort

Günther Blaschek
Institute for System Software
Johannes Kepler University
Linz, Austria
gue@jku.at

Philipp Lengauer
Institute for System Software
Johannes Kepler University
Linz, Austria
philipp.lengauer@jku.at

ABSTRACT
Parameterization of garbage collectors can help improving the overall run time of programs, but finding the best parameter combination is a tedious task. We used a simple brute-force optimization algorithm for the Java Parallel GC to study the behavior of benchmarks with thousands of configurations. As a result of this study, we propose a practically usable strategy for finding a "good" parameter combination with a small number of experiments. Using this strategy, only 10 configurations need to be tested in order to reduce the garbage collection time down to 21% (56% on average). For the studied benchmarks, this results in an overall performance gain of up to 13%.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Processors – *Memory management (garbage collection).*

General Terms
Measurement, Performance, Experimentation.

Keywords
Garbage Collection, Parameterization, Optimization.

1. INTRODUCTION
Garbage collection is a convenient feature of programming languages that helps to avoid many memory-related programming errors. However, garbage collection adds to the run time of programs and thus decreases the performance. In Java, garbage collectors can be tuned with a number of parameters, but finding the best combination for a given program is anything but easy.

The Parallel GC, as the default garbage collector, is used in most Java desktop applications. We therefore concentrated on this garbage collector. Since different garbage collectors are controlled by different parameters, the results from our investigation are of direct benefit only for programs in which the Parallel GC is used.

In an earlier project [1], the ParamILS framework [2] was used to find the optimum parameter combination. The results from this project raised a few questions, which we attempted to answer in a follow-up project:

- Can we achieve better results by spending more time on a brute force exploration of the search space?
- Which parameters have the most relevant influence on the garbage collection overhead?
- Can we find a strategy that allows us to find a "good" parameter configuration with only a few experiments?

In the remainder of this paper, we first outline the problem and then describe the experiments that finally led to a small number of configurations that need to be tested. We finally recommend a simple series of experiments for finding a promising garbage collector configuration.

2. PROBLEMS
The Parallel GC can be controlled with 37 parameters. Many of these are on/off switches, but some are numeric with wide ranges of possible values. Due to logical dependencies, not all parameter combinations make sense.

To simplify optimization of the parameters, we reduced value ranges and described dependencies between parameters. As a result, we ended up with 31 parameters. Although the search space became smaller, there are still $7.2 \cdot 10^{35}$ possible combinations.

Another problem is the execution time required for a single measurement. A program must be executed long enough to activate the garbage collector a number of times, so that the GC overhead can be measured. A number of iterations is required for warm-up, and the whole measurement must be repeated several times in order to statistically eliminate unwanted influences. For the benchmarks that we used, it took 18 minutes on average to measure the GC overhead for a single parameter configuration. If we need to find a good parameter configuration *quickly*, only a limited number of experiments can be made.

Whereas other approaches suggest a feasible parameter configuration by means of profiling [3], we decided to actually measure the actual GC overhead for concrete configurations.

3. BRUTE-FORCE EXPLORATION
The goal of the first phase was a) to find out whether we can find better results than ParamILS and b) to gain insight about the influences of individual parameters. The primary goal was exploring the search space rather than quickly finding the best possible combination. We therefore did not use optimization tools (such as HeuristicLab [4] or OptLets [5]), but rather developed a simple brute-force algorithm that systematically varies GC parameters of already known configurations. Our algorithm collects all tested parameter configurations; it then heuristically selects a "promising" configuration as the starting point for variation of parameters. The results are again added to the collection.

To enable comparison with previous results, we ran our experiments with the same set of DaCapo benchmarks that were used in [1]: h2, jython, sunflow, tomcat, tradesoap, and xalan. In the following, we use the term *experiment* for a statistically significant measurement of a single parameter configuration. The result of an experiment is the lowest GC time (peak performance) of three *runs*, where each run consists of 10 *iterations*. Within a run, the first 5 iterations are discarded as part of the warm-up phase; the lowest GC overhead of the last 5 is used as the result of a run. The following table shows the results of the brute-force algorithm in comparison with ParamILS.

Table 1. Brute Force results vs ParamILS

benchmark	# exp.	time/exp. [s]	GC [%]	ParamILS [% def]	Brute [% def]
h2	2466	1917	2.0	45	19
jython	3261	837	1.7	50	44
sunflow	2019	426	9.3	87	94
tomcat	2726	435	1.9	67	46
tradesoap	2180	2130	12.0	86	77
xalan	2227	672	18.2	23	24
	∑14879	Ø 1070	Ø 7.5	Ø 60	Ø 51

The total run time of the brute force experiments was about half a year. During that time, 14879 parameter configurations were tested, where a single experiment took about 18 minutes on average. The GC column in Table 1 shows the garbage collection overhead as a percentage of the total run time; the last two columns compare the results achieved by our brute-force algorithm with those of ParamILS. These numbers show the GC time as a percentage of the default configuration. With one exception, the new results were equal to or better than those produced by ParamILS.

4. FINDING RELEVANT PARAMETERS

In the next phase, we tried to find those parameters that have the most significant influence on GC time. To do so, we extracted subsets from the collected experiments, where the configurations within a subset differ only in a single parameter. In these subsets, we analyzed the effect of a deviation from the parameter's default value. For example, the data for h2 contained 177 cases where deviation from the default value of *SurvivorPadding* reduced the GC time, and 107 cases where the change increased the GC time.

Statistical aggregation of the parameter effects over all six benchmarks showed that only 10 out of 31 parameters either yield significant improvement or have a good chance for improvement. This allowed us to exclude many parameters from further investigation.

UseAdaptiveGCBoundary turned out to be the most significant parameter. By default, this option is disabled, but turning it on reduces the GC overhead in 84% of all cases, and in those cases the average reduction is almost 50% of the GC overhead. We therefore expect that turning this parameter on would increase performance for most programs.

This analysis showed which parameters had a chance to reduce the GC overhead, but we still needed to find reasonable alternative values. In the case of on/off switches, there is only one alternative, but some parameters have numerical value ranges. As an example, the following diagrams show the effects of the *SurvivorPadding* parameter in all six benchmarks. Each line represents a subset in which only the *SurvivorPadding* parameter is modified. The vertical axis shows the costs of changes in terms of GC time, where the best possible value in each subset sits on the baseline.

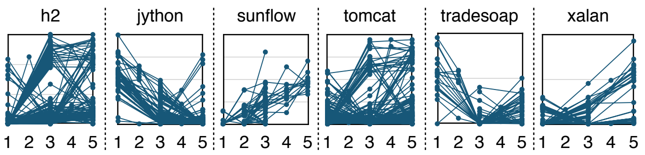

Figure 1. Effect of SurvivorPadding parameter on GC time

The diagrams show that there is no single best value of the *SurvivorPadding* parameter, but the values 1, 3, and 5 seem to be good choices for experiments.

We manually studied these diagrams for all 31 parameters. In combination with the statistical evaluation, we identified the following 10 promising parameters:

Table 2. Most relevant parameters

№	parameter	def	alt	chance [%]	Ø impr. [% def]
1	AdaptiveSizeDecrement-ScaleFactor	4	5, 6	44	9.5
2	MaxTenuringThreshold	15	1	53	21.1
3	MaxHeapFreeRatio	70	50	38	6.3
4	UseAdaptiveGCBoundary	0	1	84	49.1
5	SurvivorPadding	3	1	45	16.5
6	AdaptiveSizePolicyWeight	10	50	43	10.2
[7]	MinHeapFreeRatio	40	20, 25	29	7.9
8	YoungPLABSize	4096	1024	25	16.2
9	UseAdaptiveSizePolicy-WithSystemGC	0	1	43	10.2
[10]	NewRatio	2	1	11	27.1

The *def* column shows the parameter's default value; the *alt* column lists suggested alternative values. The *chance* column shows the percentage of cases where variation of the parameter resulted in an improvement during the brute-force runs, and *Øimpr.* shows the average improvement in percent of the default GC time.

5. FINDING A LINEAR SEQUENCE

With the reduced parameter set, there are still 2304 possible configurations. If an experiment with a single configuration takes 18 minutes (the average in our setup), we would need about 29 days to run all experiments. This is far too long for practical application, in particular when we want to repeat the experiments in case of a platform change or after maintenance.

To reduce the experimental effort, we used a variant of a hill-climbing algorithm to find a good parameter combination. Starting with the default configuration, we try all alternative parameters listed in Table 2. Whenever one such variation produces a better result than the default value, the algorithm recursively tries all variations that can be reached from there. We call this algorithm *MultiHill*, because it uses a hill climbing approach with multiple starting locations. This approach still turned out to be too expensive; it took 453 experiments to find the best possible parameter combination for h2.

When multiple parameters are involved in optimization, the order in which the parameters are modified makes a difference. For example, starting with parameter A may yield a big improvement but lead to a dead end. We therefore tried to find a linear sequence of parameters that a) leads to the best possible result, b) works equally well for all investigated benchmarks. The goal was to tackle each parameter exactly once, in order to reduce the number of required experiments.

We tried all possible permutations, but could not find a single parameter order that produces the best results for *all* benchmarks. We therefore selected the parameter order with the lowest costs over all benchmarks. This order does not always produce the same result as the *MultiHill* optimization, but the deviation is less than 1% for most benchmarks (with a maximum of 5% for jython).

The parameters in Table 2 are listed in the best order that we found in our experiments. Based on this order, we suggest the following algorithm for finding a good parameter combination:

Measure the GC time with the default parameter settings.

Go through the parameters in Table 2 from top to bottom and try the alternate values. Set that parameter to the value that yields the best result.

This way, testing all 10 parameters takes 13 experiments, including the initial experiment with the default values.

6. OMITTING PARAMETERS

In the next step, we studied the progress made with this parameter sequence for all benchmarks (see Figure 2). We hoped to find parameters that have only a small influence on the final result, so they could be omitted.

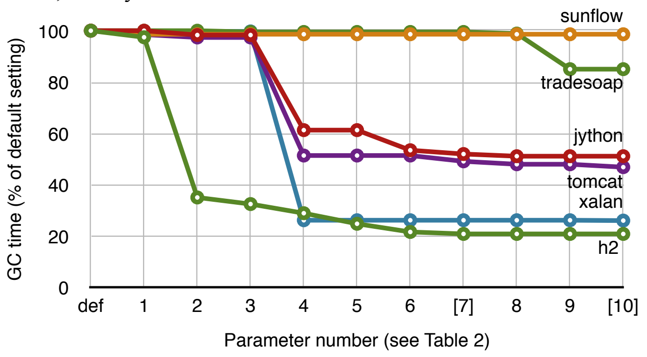

Figure 2. Progress during parameter modifications

The diagrams show that some of the parameters, notably 2 and 4, but also 6 and 9, have a significant influence. This leaves six parameters that could potentially be omitted. However, Figure 2 shows only the direct influences of the parameters, but a seemingly weak parameter change may be necessary to make subsequent improvements possible. To find out which parameters could be omitted, we tried all combinations of omissions of the parameters 1, 3, 5, 7, 8, and 10. It turned out that even the omissions with the smallest influence increase the GC overhead by a few percent. But if we are ready to accept an increase of up to 5%, we can omit the parameters 7 and 10 (MinHeapFreeRatio and NewRatio, in brackets in Table 2 and Figure 2). Omitting these two parameters reduces the number of required experiments from 13 to 10, with only a minor increase in GC time.

7. COMPARISON OF METHODS

Figure 3 shows the normalized GC times for the various approaches described in this paper. The first bar represents the default configuration (100%). The second bar shows the results achieved with ParamILS. The third bar represents the best result achieved with our brute-force algorithm. The fourth bar shows the results of the hill climbing algorithm with the reduced parameter set. The last two bars show the results with our recommended parameter sequence (for all 10 parameters and the 8 most relevant parameters).

The final results with the top 8 parameters are about as good as the results with ParamILS [2], in two cases even significantly better. Only for sunflow, our approach falls behind ParamILS, but this is a benchmark for which the default configuration is already “quite good”, so that tweaking the parameters does not yield significant improvements with any method.

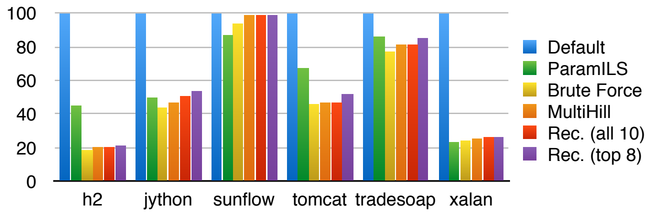

Figure 3. GC times for multiple approaches

Table 3 shows the results of running 10 experiments with the reduced “top 8” parameter set. *GC time* is the final garbage collection time in percent of the default configuration (this corresponds to the rightmost bar for each benchmark in Figure 3). The last column shows the overall reduction of run time. xalan is an extreme case because the garbage collector consumes 18.2% of the total run time. In this case, our approach is able to reduce the run time by 13.4% with 10 experiments that take less than 2 hours. Even for the other benchmarks (where the GC’s share on overall run time is much smaller), the run time improvements are typically in the range of 1 to 2 percent.

Table 3. Effort and improvements

benchmark	time for 10 exp. [h]	GC time [%]	saved run time [%]
h2	5.3	21.5	1.6
jython	2.3	53.4	0.8
sunflow	1.2	98.6	0.1
tomcat	1.2	51.3	0.9
tradesoap	5.9	85.0	1.8
xalan	1.9	26.1	13.4
Ø	3.0	56.0	3.1

8. CONCLUSIONS

Our approach shows that “good” parameter combinations for controlling a garbage collector can be found with a small number of experiments in a few hours. The benefit is reduced run time and reduced delays caused by interruptions during program execution. The average improvement over the tested benchmarks is about 3% of overall time.

The downside of our approach is that it cannot be easily extended to other parameter sets, such as different GC implementations. Applying our approach to a different problem class with different parameters requires analysis of large data sets based on many time-consuming experiments.

9. REFERENCES

[1] Lengauer P., Mössenböck H. The Taming of the Shrew: Increasing Performance by Automatic Parameter Tuning for Java Garbage Collectors. *Proc. of the 5th ACM/SPEC intl. conf. on performance engineering (ICPE '14). 111-122*

[2] Hutter, F., Hoos, H., Leyton-Brown, K., and Stutzle, T. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research, 36:267–306, 2009.*

[3] Singer J., Brown G., Watson I., Cavazo J. Intelligent Selection of Application-specific Garbage Collectors. *Proc. of the Intl. Symp. on Memory Management, 91–102, 2007.*

[4] Affenzeller M. Architecture and Design of the HeuristicLab Optimization Environment. *Advanced Methods and Applications in Computational Intelligence, 197–261, 2014.*

[5] Breitschopf C., Blaschek G., Scheidl T. OptLets: A Generic Framework for Solving Arbitrary Optimization Problems. *Proc. of the 6th WSEAS Int. Conf. on Evolutionary Computing, 49-54, 2005*