Garbage Collector Tuning in Pathological Allocation Pattern Applications

Nahuel Palumbo, Sebastian Jordan Montaño, Guillermo Polito, Pablo Tesone, Stéphane Ducasse

Garbage Collector Tuning in Pathological Allocation Pattern Applications

Nahuel Palumbo[1], Sebastian Jordan Montaño[1], Guillermo Polito[1], Pablo Tesone[1] and Stéphane Ducasse[1]

[1]*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL F-59000 Lille, France*

Abstract

Automatic memory management is often supported by Garbage Collectors (GC). GC usually impacts running application performance. For tuning properly, they expose some parameters to support the adaptation of their algorithms to specific applications' scenarios. In some cases, the developers should modify the GC parameter values to achieve high performance.

However, many application developers cannot be expected to perform expert analysis to determine which parameter values are the best for their application. There are techniques to find "good enough" parameter values. But, even if the overhead was reduced, it is still unknown the cause of the problem and how the GC tuning managed it.

In this paper, we present a methodology to identify the causes of GC overhead in Pharo applications for tuning GC parameters. We describe the GC inside the PharoVM and its parameters, looking at how their variations change the allocation behaviour. We were able to analyse, identify and understand the GC performance issues present in one real application and suggest specific GC tuning actions. Using the suggested parameter values, we improved its performance by up to 12x and reduced GC overhead by up to 3.8x.

During the experiments, we also found: 1) a bug in the production PharoVM concerning the tenuring policy, 2) a misconception about one GC parameter even for the VM developers, and 3) some possible improvements for the current GC implementation.

Keywords

garbage collector, memory profiler, memory management, virtual machine, Pharo, Smalltalk

1. Introduction

Automatic memory management is often supported by Garbage Collectors (GC) [1, 2]. GC deals with different programs that manipulate different data. This data must be allocated in the memory (heap) and accessible when the application uses it. We refer to the *allocation pattern* as the ways that applications allocate and use their data.

There are different GC strategies to decide where to allocate, keep track and, eventually, free the application data [3, 4, 5]. Garbage Collectors respond to the *allocation pattern* of running applications for better performance. So, they implement parametric algorithms to adapt their complex behaviour to each specific application [4, 6, 7].

IWST 2023: International Workshop on Smalltalk Technologies, August 29-31, 2023, Lyon, France

nahuel.palumbo@inria.fr (N. Palumbo); sebastian.jordan@inria.fr (S. Jordan Montaño); guillermo.polito@inria.fr (G. Polito); pablo.tesone@inria.fr (P. Tesone); stephane.ducasse@inria.fr (S. Ducasse)

There are reports about how garbage collection impacts application performance if not tuned properly [8, 9]. In the PharoVM [10], GC parameters start with default values and application developers are able to override them from the image. If the GC parameter values respond to the application allocation pattern wrongly, the GC will increase overhead and the application performance will be degraded (see Section 2). In that case, we say that the application presents a *pathological allocation pattern* (see Section 6). Thus, applications exhibiting allocation patterns unexpected by the GC should modify these parameter values to optimise their performance.

Worst, most application developers do not know how the Pharo GC works or have misconceptions (see Section 4). Even for experts, it is hard to identify the issues between GC and the running application. There are techniques to achieve "good enough" performance based on trial and error of GC parameter values [8]. But, even though the overhead was reduced, the root cause of the issue and how the GC tuning solved it remains unknown in the end [11, 6].

To solve this problem, we propose a methodology to identify the causes of pathological allocation patterns in running applications. Using our modified version of the PharoVM, we log data about each GC execution into a file. By analysing the logs, we are able to understand the application's allocation issues and take better decisions about how to improve performance (see Section 3).

We reduced the time spent in GC by a factor of 3x for all benchmarks of the real application studied in this paper. Our results in Section 6.3 show performance improvements in total execution time up to 12.5x.

The main contributions of this work are:

- A methodology to identify application pathological allocation patterns and tune GC parameters for avoiding them.
- A full description of the impact of Pharo GC parameters on the application allocation behaviour.
- We also found a bug in the production PharoVM concerning the tenuring policy, which was fixed and integrated into the next version (see Section 5).

2. Motivation

Data size	Total time (sec)	GC time (sec)	GC overhead
529 MB	43	7	16%
1.6 GB	150	38	25%
3.1 GB	5599	5158	**92%**

Table 1
Benchmarks for Data Frame application on load data files with different sizes. Garbage collection time increases exponentially related to the loaded data.

We found a performance issue in the Data Frame[1] open source project. A *Data Frame* is a data structure to manipulate and analyse large amounts of data. This project is used in real applications, for example in AI-related algorithms.

[1] https://github.com/PolyMathOrg/DataFrame

We observed that the execution time grows exponentially related to the amount of data loaded into the Data Frame. We make some benchmarks just by loading data from a file into the DataFrame[2]. Table 1 shows the relation between the total execution and GC running time on loading data files with different sizes into a Data Frame[3]. For the case of a 3.1 GB data file, the application spends almost 90% of the execution time just garbage collecting. This is a sign of a *pathological allocation pattern*.

Knowing the GC overhead is useful to identify possible performance issues related to allocations. However, we need more information about how the Garbage Collector interacts with the application to understand the causes of those problems.

3. Solution

For garbage collecting, the VM analyses the data accessible from some roots and "discards" not accessible data. the GC performs two strategies to analyse if some data is accessible: a *Scavenge* using a *Remembered Set* as root, or a *FullGC* using global fixed roots in the VM. There is a detailed overview of the Pharo Garbage Collector in Section 4.

We modified the VM's code to collect information each time the Garbage Collection is triggered.

From Scavenges, we get:

- Amount of memory used to allocate objects in the Eden (before).
- Total survivor's memory size (before and after).
- Number of objects in the Remembered Set (before and after).
- Amount of tenured data (before and after).
- Tenured threshold (if objects were tenured in the pass).
- Executed time.

And from FullGC:

- Time spent marking.
- Time spent sweeping.
- Time spent compacting.
- Total executed time.

Using our version of the PharoVM, we log all listed data into a file. Then, analysing this file, we identify the causes of application pathological allocation patterns and adapt the set of GC

[2]Evaluating: `DataFrame readFromCsv: pathToFile asFileReference.`

[3]Data files used are:

- https://www.kaggle.com/datasets/rahulbanerjee123/aws-product-length?resource=download (529 MB and 1.6 GB)
- https://www.kaggle.com/datasets/antoninadolgorukova/proteinrna-vs-rna-spearman-correlation-data?select=Prot-RNA_corr_63gr.csv (3.1 GB)

parameters to achieve better performance. Our results show performance improvements in total execution time up to 12.5x for the Data Frame application presented in 2. We reduced the time spent in GC by a factor of 3x for all benchmarks (see Section 6).

In the next section, we introduce the Pharo Garbage Collector and its parameters. Profile data also help to understand how each parameter affects the application allocations.

4. Background: Pharo Garbage Collector

The PharoVM[4] has automatic memory management based on a Generational Garbage Collector [12]. Memory heap is divided into two areas: *New Space* and *Old Space*. Each space has a different structure and strategies for allocation and garbage collection. A big picture of this organization is shown in Figure 1.

This design is based on the *weak generational hypothesis*: most of the objects die young [12, 13]. So, this kind of GC allocates newly instantiated objects and old ones in different spaces. The New Space will be garbage collected often by a *Scavenger* and the Old Space occasionally by a *FullGC* since it is bigger and more expensive to scan.

4.1. Pharo Generational Collector Structure

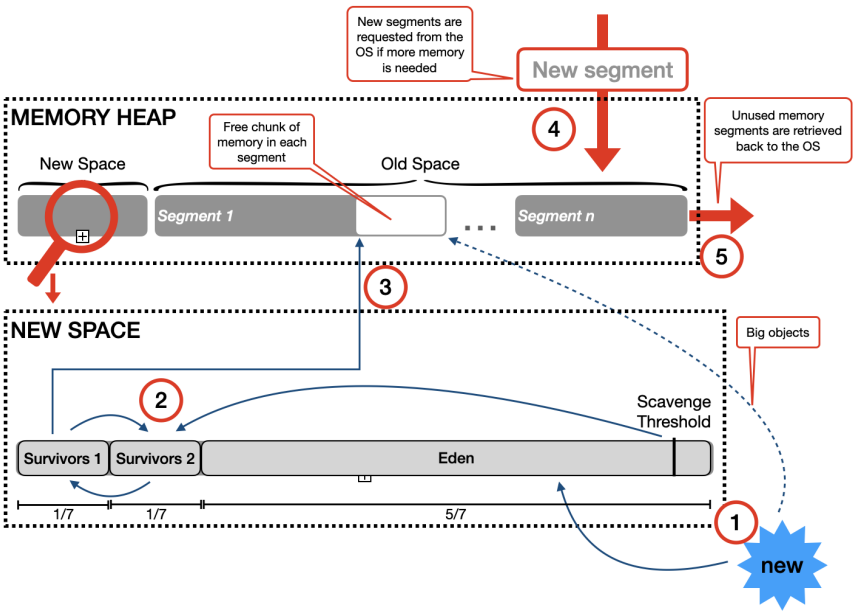

Figure 1: Overview of the memory heap structure by the Generational Garbage Collector. There are two sections: New Space and Old Space. We zoomed the New Space to show inner divisions. Blue arrows show the data allocation flow. Red letters and arrows are for references in the paper.

The New Space is divided into 3 sections: the Eden and two semi-spaces. The amount of

[4]https://github.com/pharo-project/pharo-vm

memory for each maintains always the same proportion of $\frac{5}{7}$ for the Eden and $\frac{2}{7}$ for the semi-spaces. The Eden is the place where new objects are allocated and the two semi-spaces keep the survivors. Big enough objects are created directly in the Old Space. This is referenced by (1) in Figure 1.

When the amount of memory used in the Eden reaches a threshold, a Scavenge pass is automatically scheduled. The Scavenger performs a copy-collection, copying all survivors to one dedicated semi-space. It is referenced by (2) in Figure 1. All non-copied data will be overridden by new allocations. To know which objects survive, *root references* should be followed. The roots for the New Space are the references coming from objects in the Old Space. For avoiding to scan all objects in the Old Space looking for roots, there is a *Remembered Set* with the objects in the Old Space that have any reference to an object in the New Space [12]. This Remembered Set is updated at each store (write barrier), based on the object-allocated spaces.

After many passes, the amount of survivor objects could increase and not fit in the survivor's semi-space. Then, some objects are *tenured* to the Old Space, based on some policy [14]. This policy could be based on the object's age, memory location or a specific goal as reducing the number of objects in the Remembered Set. In the current implementation, the Scavenger triggers the tenuring objects when survivors in the semi-space fill the 90%, this value is hardcoded in the GC code. Tenured objects are referenced by (3) in Figure 1.

The Old Space is the memory region where most objects live, being at least 3 times bigger than the New Space. It is organised in segments and incorporates more segments dynamically to have enough free space for the demanded amount of the objects/memory by the application. On the contrary, if the system detects that much of managed memory is free and not being used, it returns the free segments to the OS. This process is referenced by (4) and (5) in Figure 1.

As the Old Space could be very big and expensive to traverse, this space is garbage collected occasionally. The garbage collection in this space is performed by a *FullGC*. On FullGC, objects in both, New and Old Spaces, are tracked and non-accessible memory is freed.

The system automatically decides when to run the Scavenge or FullGC passes. A Scavenge pass is triggered when the Eden is almost full, to free memory for new allocations. The reasons for triggering FullGC are mainly because the system does not have enough memory in the Old Space to allocate one object. In this case, GC tries to free the required amount of memory before requesting them to the Operating System. Applications can also execute a FullGC programmatically.

4.2. Analysis of GC Parameters

There are five parameters in the PharoVM to control the GC behaviour: *Desired Eden size*, *Survivors to keep*, *Full GC ratio*, *Grow headroom* and *Shrink threshold*. Next, we will explain each and see their impact on allocations.

Desired Eden size It is the desired memory size for the Eden. This parameter also controls the size of the entire New Space to maintain the proportion of each section. By default, the Eden size is 16MB.

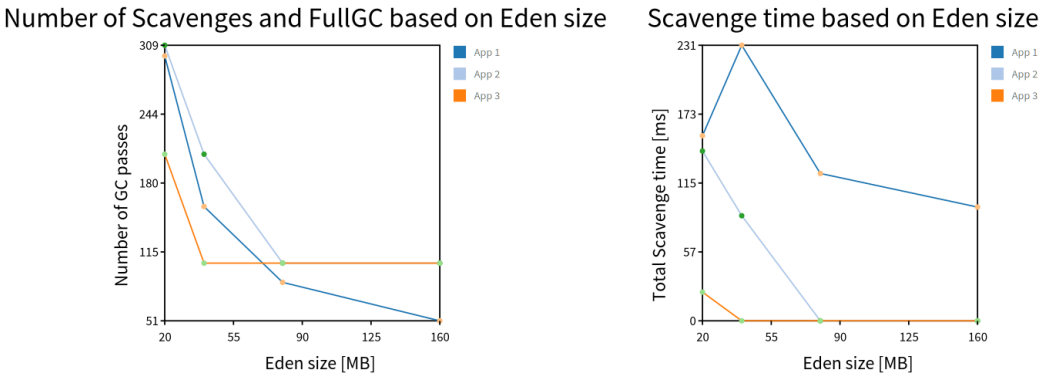

Figure 2: Garbage Collector behaviour based on Eden size for different applications. On the left, the number of garbage collection passes decreases asymptotically while the Eden size is bigger. On the right, the amount of time Scavenging is not fully dependent on the Eden size.

A bigger Eden implies that the GC will support more objects between passes. Thus, GC will perform fewer automatic garbage collections. Figure 2 shows the number of garbage collection passes based on Eden size for different applications. The total number of garbage collections (FullGCs and Scavenges) decreases asymptotically while the Eden size grows. The minimum value depends on how many allocations and FullGC are triggered by the application [5].

Counterintuitively, Figure 2 also shows no clear correlation with the Scavenge time, it also depends on the number of survivors and objects in the Remembered Set (see Section 6.2).

Survivors to keep This value is an integer with the desired number of survivor objects to keep in the New Space when the Scavenger decides to tenure objects.

The VM estimates the proportion of objects to keep in the semi-space based on an *average object size* of 64 bytes. Using the estimated memory size, the GC computes a *tenuring threshold* inside the survivor's semi-space to decide which objects will be tenured to the Old Space (i.e. the objects before the threshold). By default, only 10% of survivors in the semi-space will be tenured.

Figure 3 shows how the total amount of data tenured changes with different values of survivor objects to keep for different allocation patterns. The decision about which objects will be tenured has an impact on applications with different object lifetimes. For example, the application at the bottom of Figure 3 presents a critical value where long lifetime objects keep in the New Space and more data is tenured.

For applications that use enough amount of data for a long time, this threshold does not have a big impact. Most of the objects will be tenured anyway because of the application usage, independently of the tenuring policy. This is the case of the application on top of Figure 3.

As this parameter is a raw quantity, we need to consider the survivors' semi-space size. If the number of objects to keep is big enough to consider all survivors, then GC will not tenure objects from the survivors' semi-space. Instead, it will tenure all new incoming survivors from the Eden directly to the Old Space because there is no more space in the semi-space. The last point for each application in Figure 3 represents that scenario.

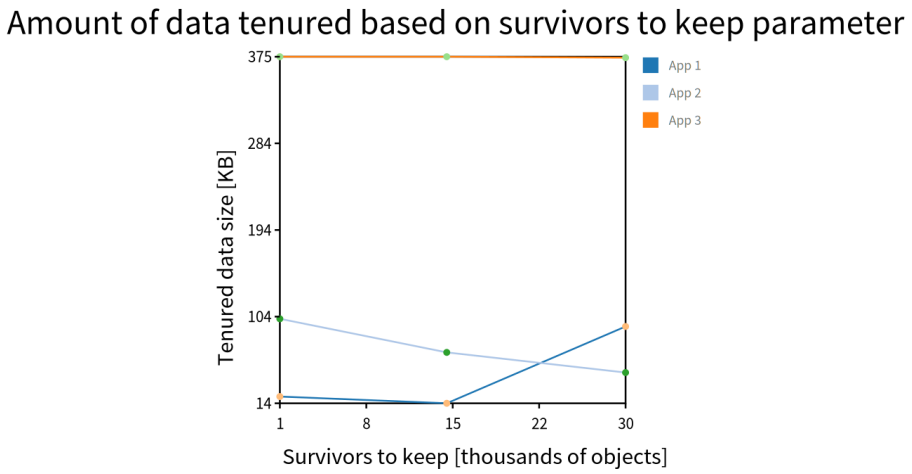

Figure 3: Total amount of tenured data based on the number of survivor objects to keep in the New Space. Points at the end represent the case where all survivors in the semi-space are kept and only new survivors from the Eden are tenured.

Tenuring fewer objects reduce the chances of *nepotism*. It reduces the stress of FullGC and then improves the Garbage Collector performance [**?**].

FullGC ratio This ratio is a threshold for triggering a FullGC when the old space grows more than expected. By default, this value is set at 33%. Thus, when the old space grows 1/3 of its space (from the last FullGC) it will perform a FullGC. Increasing this value will reduce the number of FullGCs automatically performed by the VM. This parameter does not modify the behaviour of the Scavenger.

Figure 4 shows how the number of FullGC decreases by a bigger FullGC ratio value. This effect was only observable in applications with intensive use of memory, i.e. many allocated objects that finish in the old space.

Grow headroom The grow headroom is the minimum amount of memory that the GC will order from the OS. It is used when the system detects that there is not sufficient space in the Old Space, so it will be expanded. The value is 16MB by default.

This parameter is important to avoid requesting many small chunks of memory from the OS. If the Old Space is filled and new small objects need to be allocated there, the space should grow with sufficient memory for many of them for better performance. Thus, the value for this parameter should be relative to the average size of the new allocations for avoiding unnecessaries FullGC. On the other hand, if the requested memory is much bigger than the total of newly allocated objects size, it wastes resources with unused memory.

To show this effect, we created an artificial tiny benchmark creating 100 objects of 24MB each. We run it with different headroom values, results are in Figure 5. The first configuration was executed with a headroom of 16MB, less than 24MB, so each object allocation triggered a FullGC. The number of FullGC decreases proportionally to the number of new objects that enter the headroom. In this extreme scenario, the last configuration was 5x faster than the first one.

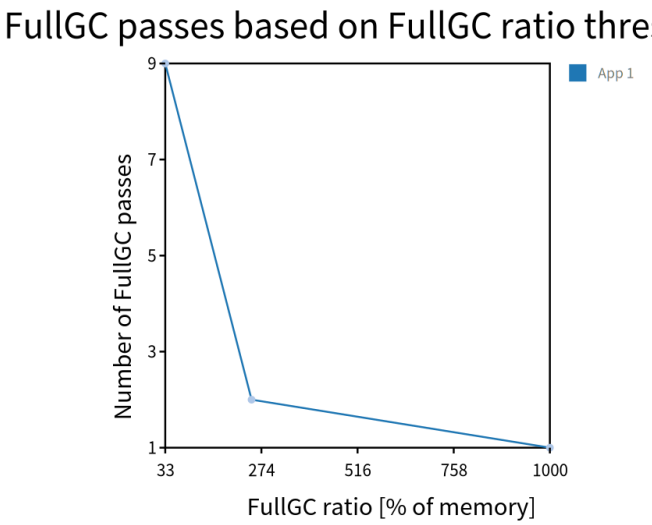

Figure 4: Number of FullGC based on Full GC ratio. Higher ratios imply fewer FullGCs automatically triggered.

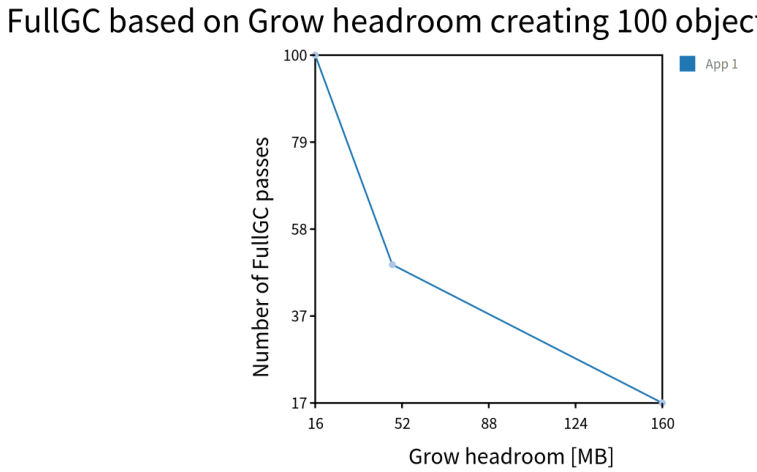

Figure 5: Number of Full GC for different Grow headroom values. The number of Full GC is proportional to the number of new objects that fit the headroom.

Shrink threshold This threshold determines the maximum amount of free memory (without objects) in the Old Space that the GC will manage before trying to get it back to the OS. In that case, it will return to the OS all chunks of free memory until the amount of free memory is close to the Grow headroom, introduced before. By default, the value of this parameter is 32MB.

In our experiments, we did not see any observable effect based on this parameter, maybe because we did not find the best application allocation pattern for it. We may need more integral profiling techniques for the analysis of this parameter.

5. Bug in the Scanvenge's tenuring policy

We observed a bug in the tenuring objects policy during the profiling sessions. The *tenuring threshold* value was computed outside the current survivors' semi-space for all cases where the GC should tenure objects from the New to the Old Space. It makes no sense since this threshold should divide the survivor objects that will and will not be tenured, so it will be at some part inside the current survivors' semi-space.

Checking the code, we found that the threshold was mistakenly computed in the wrong semi-space. So, any time that the scavenger wants to tenure objects, it was tenuring either none or all survivors, depending on the position of the semi-spaces in the memory.

We fixed it to compute the value in the correct semi-space. Figure 6 shows the amount of data that survives in the semi-space on each Scavenge before and after the fix respectively. Marked dots represent the value of the tenuring thresholds. The horizontal dotted line is the limit of the semi-space.

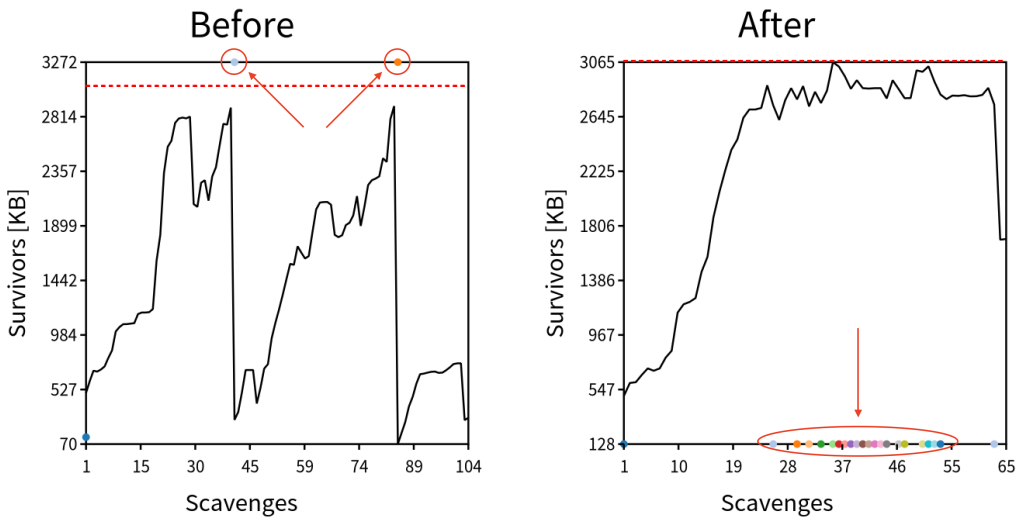

Figure 6: Bug on computing the tenuring threshold in Scavenger. Black lines are the survivor data amount in the semi-space, dots represent the tenuring threshold when the Scavenger decides to tenure (pointed by arrows and circles) and the red horizontal dotted line is the limit of the semi-space. At the left, the threshold is set outside the current semi-space, so all objects are tenured. At the right, the threshold is computing in the correct place, only 128KB of objects are tenured in each Scavenge.

Before the fix, the threshold is outside the semi-space and all survivors are tenured. After the fix, more scavenges perform tenuring but much less amount of data is tenured on each. Summarising them, the total amount of tenured data in this example decreases from 4 to 2.5 MB.

For the rest of the analysis in this paper, we will use the fixed version of the VM. This fix is already integrated into the PharoVM and will be present in the next release.

6. Case study: pathological allocation patterns

In this section, we will show the analysis of the pathological allocation patterns discovered for the Data Frame application presented in Section 2 based on the profile data collected by our instrumented GC. Our methodology is based on identifying what are the causes of performance issues and suggesting some changes in the GC parameters values for solving them. Then we rerun the benchmarks with the suggested changes to see if the observed problems were solved.

6.1. Reducing the number of FullGC passes

Looking at the profile data, the first thing that called our attention was the number of FullGC triggered together. Performing many FullGC together increases overhead because scanning all the Old Space is an expensive task. It is better to wait for some Scavenges before, so the application has more time to use the objects and then the GC has more chances to free their memory. Figure 7 shows the relation between Scavenges and FullGC.

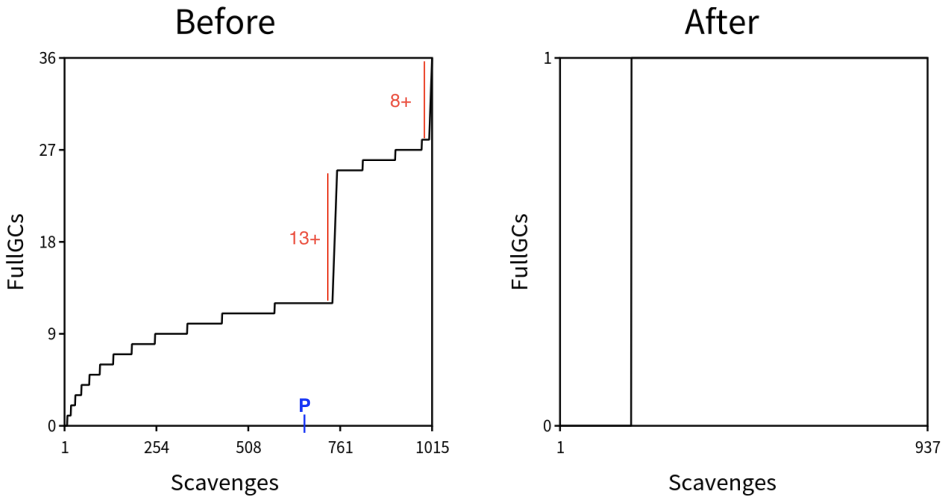

Figure 7: Relation between Scavenges and FullGC in Data Frame benchmarks. In red, the number of FullGC triggered together. In blue, point P is used as a reference between the two different patterns. After the fix, only one FullGC is performed

Analysing the first chart in Figure 7 we observed two patterns of FullGC: during data loading (before point P), FullGC are triggered periodically in time; and during Data Frame building (after point P), FullGC are triggered together.

The first group is due to the loaded data being tenured to the Old Space, growing it and triggering the FullGC ratio, as we introduced in Section 4.2. In this case, the application is loading a lot of non-temporary data, so the FullGC will not free memory. We should adapt the FullGC ratio relative to the amount of loaded data to avoid most of them.

The number of FullGC triggered together is due to the size of the Data Frame structures. They are big enough for being allocated directly in the Old Space. If the Old Space does not have enough free memory, FullGC is triggered as explained in Section 4.2. To avoid this anomaly, we

need to set a Grow Headroom according to the expected Data Frame size.

So, we customised the FullGC ratio to a very big value, simulating an infinite ratio, and the Grow headroom to the same as the loaded file, depending on the benchmark. We ran the benchmarks again to compare performance and continue the analysis. These GC parameter values reduced the number of FullGC from 36 to only 1 time and increased the application performance between 1% and 5%.

6.2. Reducing Scavenge time

Analysing the newly profiled data, we saw a big increase in the time of Scavenges when the Data Frame is being built. Scavenges are 52x slower now! This increment in Scavenge time starts when the Data Frame is created and persists until the end of the benchmark, as we can observe in Figure 8, representing 25% of the running application time.

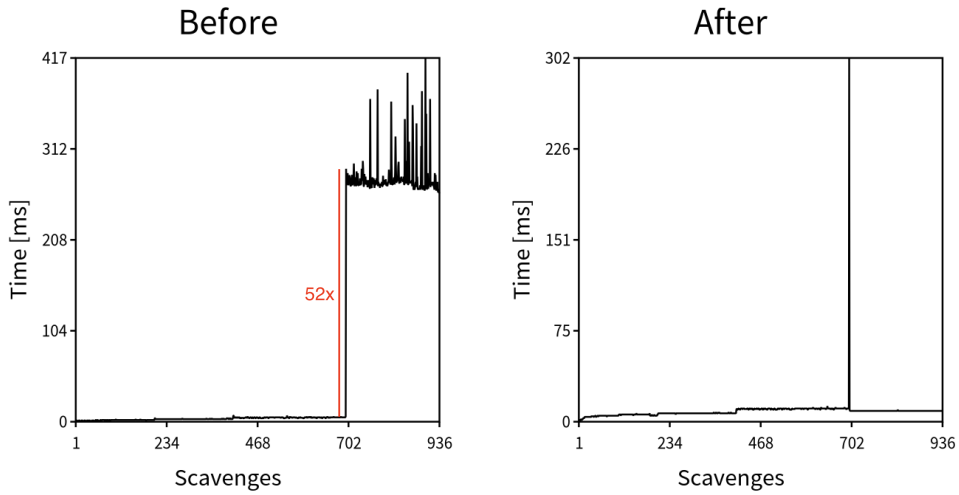

Figure 8: Time of each Scavenge in Data Frame benchmarks. After the red mark, all Scavenge passes at the end have increased their time by more than 52x. After the fix, only one Scavenge has this overhead.

This pathological scenario is due to a big object inside the *Remembered Set*, probably the Data Frame structure. To know which objects survive in each scavenge, all references from every object in the Remembered Set should be scanned. If an object with so many references arrives in the Remember Set, it could take a long time to scan it. Looking at the number of objects in the Remembered Set in each scavenge, we know that only one (or a very small group of objects) are the cause of this behaviour.

We should avoid spending so much time scanning objects on each Scavenge because it is very frequent. But, as there is no way to control the objects in the Remembered Set, it is not an easy task. We tried with different values for the tenuring threshold introduced in Section 4.2 to control which survivor objects in the New Space are tenured to the Old Space. None of the possible values could avoid this situation, since the pointed new objects are continually created and the big object always stays in the Remembered Set.

The solution that we found for this issue was to configure the tenuring threshold value to keep all survivors in the semi-space. Then, after the survivor's semi-space fills up, all new survivor objects will be automatically tenured from the Eden. Using this strategy, all survivor objects created at the end of the benchmark will live together in the Old Space. Thus, we avoid having big objects in the Remembered Set for almost all Scavenges.

6.3. Final configuration and Results

Given the previous observations, we tuned GC parameters to:

1) Have an infinite FullGC ratio To reduce the number of FullGC when the Old Space grows.

2) Have a grow headroom equal to the loaded file To avoid many FullGC together.

3) Keep all survivors in the semi-space To tenure new objects to the Old Space quickly.

Data size	Total secs before	GC overhead before	Total secs after	GC overhead after
529 MB	43	16%	37 (1.1x)	5% (3.2x)
1.6 GB	150	25%	122 (1.2x)	7% (3.6x)
3.1 GB	5599	**92%**	440 (12.5x)	**24%** (3.8x)

Table 2
Results of tuning GC parameters for Data Frame application benchmarks. We fixed the pathological allocation patterns to avoid GC overhead. Parenthesis values show performance improvement.

We improved the time spent in GC for the Data Frame application benchmarks presented in Section 2 by a factor bigger than 3x, as we show in Table 2.

6.4. Take aways

Learnings about this experience are:

- It is possible to tune the GC parameters for a specific application based on the methodology presented in this paper.
- Using profile techniques on running applications we were able to identify pathological allocation patterns.
- We understood how variations in the GC parameters are related to the application pattern allocations. It is important to have a good overview of the GC algorithm to understand what the effect of changing each parameter is.
- Mixing the items above, we have a set of tools to analyse the interactions between applications and VM garbage collection. We are able to make better decisions about tuning applications for performance improvements.

7. Related work

The work presented in this paper is a continuation of previous work [15]. Even if we did not use Vicoca, the tool presented in that paper, we reuse part of the infrastructure developed on top of the Pharo VM. While previous work aims to identify Code Cache issues, in this work we use profiling via GC events to identify allocation issues. Both approaches are similar: profiling VM events to analyse specific applications' behaviour and, eventually, tune VM parameters to achieve better performance.

Kaleba *et al.,* [16] have extended an existing VMProfiler for the same VM as us. They were interested in profiling the generated machine code to improve the JIT compiler, different to our goal of GC tuning for applications.

There are many works exploring techniques to find an optimal set of GC parameter values for a specific application [8, 9, 17]. They present the problem of having too many parameters for tuning, so they use machine learning and hill-climber algorithms to find good values. Our approach is based on analysis and understanding the application allocation patterns for consciously tuning GC parameters. They also agree about the importance of tuning GC parameters to improve applications' performance.

The master thesis of Neu is the most similar to our work [7]. He describes the parameters in the OpenJ9 Java VM showing the impact of each on applications performance, similar to us. His work also uses profile techniques to log VM information and a tool to analyse it later, such as ours. Both approaches intend to help users to get information about their running applications' behaviour and be able to manually tune GC parameters.

VM profiling techniques are commonly used in *auto-tuning* research [6, 11]. We agree about the difficulty of tuning GC parameters for user developers. To solve this problem, they propose a VM implementation which tunes itself online to improve the predictions about application behaviour. Instead, we prefer to build tools that users can use to learn about its applications and take the decision about the best way to improve its performance.

Finally, Ungar and Jackson already suggest a low effectiveness of a "simple two-generation scavenger" in programs that hang onto large amounts of temporary data for extended periods of time [**?**]. This is exactly the main cause of our pathological case study. Next years after that, some researchers proposed solutions to this problem thru *pretenuring* [18, 19, 20, 21]. They use profiling techniques and sampling to predict allocations of objects with long lifetimes.

8. Future work

8.1. Available for the community

Our next goal is that Pharo developers can profile the Garbage Collector on their applications and adapt its parameters if needed. For that, we need to integrate the work done in the PharoVM to be able to profile Garbage Collection passes and an API for users[5]. We also want to publish an application with tools for analysing and visualising the profiled data [22]. Using that tool and the methodology presented in this paper, developers will be able to adapt the Garbage Collector behaviour to their applications' specific allocation patterns.

[5] https://github.com/pharo-project/pharo-vm/pull/606

8.2. Change GC parameters

We also have two suggestions to improve the API for the current GC parameters:

1. Change the *Survivors to keep* parameter for a *Tenuring proportion*. This parameter confuses even for VM developers, we saw erroneous comments in the GC code. The new suggested parameter will directly represent the percentage of survivor data to tenure when New Space is full.

2. Add a new parameter to manage the *Tenuring criterion*. Now the Scavenge have different criteria for tenuring objects. In this paper, we just presented the tenuring by age, because is the normal criterion. Using this parameter, we could set a criterion for reducing the Remembered Set, for example.

8.3. Improve GC in PharoVM

We are also excited to improve the Pharo Garbage Collector implementation to automatically avoid some of the pathological scenarios seen in Section 6.

First, we want to avoid big objects in the Remembered Set. Since the design of a Generation Garbage Collector is based on fast Scavenges, avoiding scanning big objects on each pass could improve its performance.

Another idea is to have a *pretenuring* strategy for long-life objects. We have an idea to identify these objects based on the allocation site (the method where the object is instantiated).

8.4. Continue profiling

Continuing this work, we want to increase the data collected by the profiling sessions to be able to make better analyses. During the experiments, some data about FullGC was missing for a better understanding of the passes. Cause of the pass, survived data size, free chucks, and segments information are some examples. With more information, we will be able to understand the application allocation patterns better and, for example, find better examples for each GC parameter change.

9. Conclusions

In this paper, we presented a methodology to identify the causes of performance issues with the Pharo Garbage Collector (GC) for running applications. It is based on profile data collected each time that the GC is executed. Using the profile data, we described the GC used by the PharoVM and its parameters. We showed how each parameter affects the underlined behaviour of the GC.

We introduced a real application scenario presenting performance issues with the GC. Then, we were able to identify the causes of these *pathological allocation patterns*. Using our methodology, we were able to tune the GC parameter values for the specific application allocation pattern. Our results showed performance improvements up to 12.5x in execution time and reduced the GC overhead by up to 3.8x, for the analysed benchmark.

This work highlights the importance of tuning GC parameters for specific application behaviour. Our descriptions of the GC and its parameters, the profiling tools and the presented methodology are useful to help developers with this task.

References

[1] L. P. Deutsch, D. G. Bobrow, An efficient, incremental garbage collector, CACM 19 (1976) 522–526.

[2] R. Jones, A. Hosking, E. Moss, The garbage collection handbook: the art of automatic memory management, CRC Press, 2016.

[3] R. Garner, The design and construction of high performance garbage collectors, Ph.D. thesis, The Australian National University, 2011.

[4] S. M. Blackburn, P. Cheng, K. S. McKinley, Oil and water? high performance garbage collection in java with mmtk, in: Proceedings. 26th International Conference on Software Engineering, 2004, pp. 137–146. doi:10.1109/ICSE.2004.1317436.

[5] M. Hertz, E. D. Berger, Quantifying the performance of garbage collection vs. explicit memory management, in: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 2005, pp. 313–326.

[6] S. Jayasena, M. Fernando, T. Rusira, C. Perera, C. Philips, Auto-tuning the java virtual machine, in: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IEEE, 2015, pp. 1261–1270.

[7] N. Neu, Automatic application performance improvements through VM parameter modification after runtime behavior analysis, Ph.D. thesis, University of New Brunswick, 2014.

[8] P. Lengauer, H. Mössenböck, 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, 2014, pp. 111–122.

[9] G. Vijayakumar, R. Bharathi, Predicting jvm parameters for performance tuning using different regression algorithms, in: 2022 Fourth International Conference on Emerging Research in Electronics, Computer Science and Technology (ICERECT), IEEE, 2022, pp. 1–8.

[10] E. Miranda, The cog smalltalk virtual machine, in: Proceedings of VMIL 2011, 2011.

[11] J. Singer, G. Kovoor, G. Brown, M. Luján, Garbage collection auto-tuning for java mapreduce on multi-cores, ACM SIGPLAN Notices 46 (2011) 109–118.

[12] D. Ungar, Generation scavenging: A non-disruptive high performance storage reclamation algorithm, ACM SIGPLAN Notices 19 (1984) 157–167. doi:10.1145/390011.808261.

[13] H. Lieberman, C. Hewitt, A Real Time Garbage Collector Based on the Lifetimes of Objects, AI memo no 569, MIT, 1981.

[14] D. Ungar, F. Jackson, Tenuring policies for generation-based storage reclamation, in: Proceedings OOPSLA '88, volume 23, 1988, pp. 1–17.

[15] P. Tesone, G. Polito, S. Ducasse, Profiling Code Cache Behaviour via Events, in: MPLR '21, Münster, Germany, 2021. URL: https://hal.inria.fr/hal-03332040. doi:10.1145/3475738.3480720.

[16] S. Kaleba, C. Béra, A. Bergel, S. Ducasse, A detailed vm profiler for the cog vm, in: International Workshop on Smalltalk Technology IWST'17, Maribor, Slovenia, 2017. URL: https://hal.inria.fr/hal-01585754.

[17] J. Singer, G. Brown, I. Watson, J. Cavazos, Intelligent selection of application-specific garbage collectors, in: Proceedings of the 6th international symposium on Memory

management, 2007, pp. 91–102.

[18] P. Cheng, R. Harper, P. Lee, Generational stack collection and profile-driven pretenuring, in: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 1998, pp. 162–173.

[19] T. L. Harris, Dynamic adaptive pre-tenuring, in: Proceedings of the 2nd International Symposium on Memory Management, ISMM '00, Association for Computing Machinery, New York, NY, USA, 2000, pp. 127–136. URL: https://doi.org/10.1145/362422.362476. doi:10.1145/362422.362476.

[20] M. Jump, S. M. Blackburn, K. S. McKinley, Dynamic object sampling for pretenuring, in: Proceedings of the 4th international symposium on Memory management, 2004, pp. 152–162.

[21] D. Buytaert, K. Venstermans, L. Eeckhout, K. De Bosschere, Garbage collection hints, in: High Performance Embedded Architectures and Compilers: First International Conference, HiPEAC 2005, Barcelona, Spain, November 17-18, 2005. Proceedings 1, Springer, 2005, pp. 233–248.

[22] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, J. Laval, Agile visualization with Roassal, in: Deep Into Pharo, Square Bracket Associates, 2013, pp. 209–239.