Improving Performance Through Object Lifetime Profiling: the DataFrame Case

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

[1]*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Park Plaza, Parc scientifique de la Haute-Borne, 40 Av. Halley Bât A, 59650 Villeneuve-d'Ascq, France*

Abstract

Being capable of profiling the object lifetimes of an application gives information that can be used to optimize the GC performance and improve overall execution time. One can pre-tenure objects based on profiler information, tune the GC parameters, or take decisions about pre-allocating bigger memory segments. However, accessing object lifetimes is difficult because it requires monitoring any object GC reclamation. We developed an open-source lifetime profiler. Our current implementation does not require Virtual Machine modification. It is based on ephemerons and method proxies. We profiled *DataFrame* and we observed a significant number of objects that lived a long time. We used this information to tune the garbage collector parameters and we got up to 6.8 times of performance improvements.

Keywords

memory profiler, garbage collector, object lifetimes, optimization

1. Introduction

Modern programming languages offer automatic memory management through garbage collectors (GC) [1]. This takes the responsibility of allocating objects and freeing the allocated memory from of the developer. Pharo has a two-generation GC with a scavenging algorithm for the new generation and a stop-the-world mark-and-compact algorithm for the old generation [2, 3]. The Pharo GC periodically traverses the memory to detect the objects that are not reachable (an object is not reachable when it is no longer accessible nor usable). After the memory traversal, the GC frees the unreachable objects' memory.

There are some applications in which a significant part of the execution time is spent in garbage collections. The default GC parameters are rarely ideal for any given application [4]. Consequently, there is considerable potential for optimizing such applications to mitigate garbage collection overhead. Profiling the object lifetimes gives information that can be used to optimize GC performance and improve the overall execution time. One can pre-tenure objects based on profiler information [5], tune the GC parameters [6, 7, 4], or take decisions of pre-allocating bigger memory segments. We define an object's lifetime as the difference

IWST 2023: International Workshop on Smalltalk Technologies. Lyon, France; August 29th-31st, 2023
sebastian.jordan@inria.fr (S. Jordan Montaño); nahuel.palumbo@inria.fr (N. Palumbo); guillermo.polito@inria.fr (G. Polito); stephane.ducasse@inria.fr (S. Ducasse); pablo.tesone@inria.fr (P. Tesone)
https://github.com/jordanmontt/ (S. Jordan Montaño)

© 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
CEUR Workshop Proceedings (CEUR-WS.org)

CEUR-WS.org/Vol-3627/paper09.pdf

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

between its allocation time and its finalization time $object's\ lifetime = finalization\ time - allocation\ time$. An object's finalization time is when the GC decides to collect it.

Understanding object lifetimes requires precise profiling information on the allocations and deallocations (when an object is reclaimed by the GC). Object allocations can be precisely identified by instrumenting all allocation sites (*e.g.,* send the message basicNew). However, in generational GCs is not possible to precisely know when an object becomes unreachable. It will be detected when the GC traverses the memory. A long time may have passed when an object becomes unreachable and when it is reclaimed. Understanding object deallocation requires instrumenting automatic reclamation algorithms such as GCs. Furthermore, finalization mechanisms such as ephemerons [8] introduce a high memory overhead that reduces performance [9].

We developed an Object Lifetimes Profiler as a plugin of Illimani[1]. Illimani is a memory profiler for Pharo[10] and it runs on an unmodified Pharo virtual machine. It is available under an open-source MIT license. Our profiler tracks the object lifetimes of a given application via instrumentation by controlling the execution whenever an object is allocated. MethodProxies[2] is a library that decorates and controls the method execution by executing user-defined behavior before or after a method's execution. We used this library to do the instrumentation.

We evaluated our solution by observing how object lifetimes relate to performance improvements when tuning the GC. We choose as a case study the loading of a 500 MB dataset into a *DataFrame* [11]. We have selected *DataFrame*[3] library for our study because it is often used in memory-intensive applications such as machine learning, data mining, and data analysis [12]. The profiler gave us object lifetimes.

We observed that our case study has 25% of long-lived objects that represent 40% of the allocated memory (Figures 4, 5). Applications that have many objects that live a fairly long time suffer from performance issues [13]. Increasing the GC heap size has a significant impact on GC performance [7, 4]. With this information, we decided to tune the GC parameters to see if we can get performance improvements [14, 13, 6]. We chose 5 non-default GC parameter configurations that increase the heap size and we then benchmarked the loading of the *DataFrame* with these configurations. We obtained improvements of up to 6.8 times compared to the default GC parameters when the number of full garbage collections is reduced.

Paper's contributions. This paper makes the following contributions:

- It presents the challenges of lifetime profiling opening the door for future profiler improvements.
- It introduces a lifetime profiler capable of tracking the object lifetimes for a given application with the unmodified Pharo VM.

Paper's outline. Section 2 explains the automatic memory management in Pharo and the challenges of computing the object lifetimes; Section 3 explains our solution; Section 4 explains the validation of our solution profiling a memory-intensive application; Section 5 presents the discussion; Section 6 presents related work; and Section 7 finishes with the conclusion and the future work.

[1] https://github.com/jordanmontt/illimani-memory-profiler
[2] https://github.com/pharo-contributions/MethodProxies
[3] https://github.com/PolyMathOrg/DataFrame

2. Automatic memory management in Pharo

Pharo has a two-generation garbage collector with a young and an old generation. The newly allocated objects are allocated in the new generation, also called new space. The old space is where the objects, after surviving a certain number of GC cycles are promoted. This promoting process is called *tenuring*.

Generational GCs are designed to leverage an empirical property of objects: young objects are more prone to die while older ones tend to persist in memory [13]. These GCs are configured to exploit this empirical property.

The new space uses a scavenging algorithm while the old space uses a stop-the-world mark-and-compact algorithm [2, 3]. An incremental garbage collection executes the scavenging algorithm while a full garbage collection executes both a scavenging and the stop-the-world mark-and-compact algorithm. Running a full garbage collection is slower than an incremental one by orders of magnitude [14]. For this reason, the scavengers are executed orders of magnitude more often.

Each time that a full garbage collection is executed, the GC traverses the objects that are in the old space to detect the ones that are not reachable to then reclaim them. An object is not reachable when it stops being accessible and usable by the application. Afterwards, the GC frees the memory of all the non-reachable objects. Ephemerons [8] are a new finalization mechanism that is implemented in the Pharo Virtual Machine version 10[4]. They allow one to perform a user-defined action when an object is about to be reclaimed by the GC.

In Pharo, almost all computations are done by sending a message [15]. This is the case for allocating an object too. In Pharo 11, 4 methods are responsible for allocating objects: Behavior»basicNew , Behavior»basicNew: , Number»@ , and Array class»new: . These methods are special methods that are executed natively.

Understanding object lifetimes requires precise profiling information about the allocations and the garbage collections. Object allocations can be precisely identified by instrumenting the allocation sites (*e.g.,* the allocator methods). However, as Pharo has a two-generation GC is not possible to precisely know when an object becomes unreachable (stops being accessible). A long time may pass until the object became unreachable and when the GC decided to finalize it. Detecting when an object is being reclaimed by the automatic memory manager requires instrumenting the GC. The available finalization mechanisms in Pharo, such as ephemerons [8], introduce a high memory overhead that reduces performance [9].

3. Profiling object lifetimes

We instrumented the 4 allocator methods (Behavior»basicNew , Behavior»basicNew: , Number»@ , and Array class»new:) to intercept whenever they are invoked using MethodProxies. MethodProxies allow one to decorate and control the execution of a method. With the instrumentation, we are able to capture the exact allocation time. Within the instrumentation, we also installed an ephemeron for each of the allocated objects that will set its finalization time when the object will be reclaimed by the GC. We also store the object's size in memory.

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

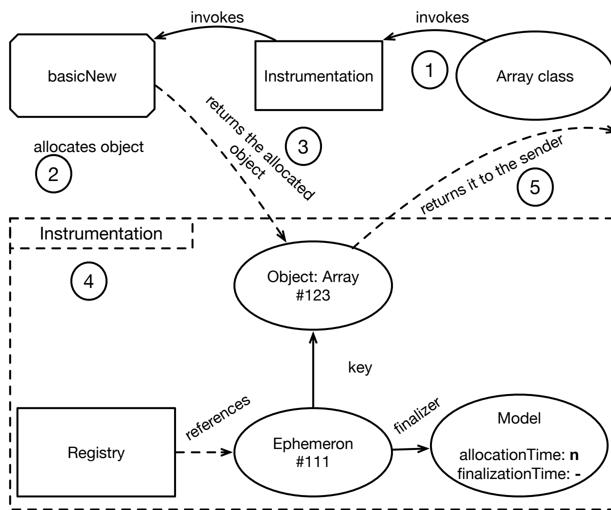

Figure 1: The allocation of an object at a time n

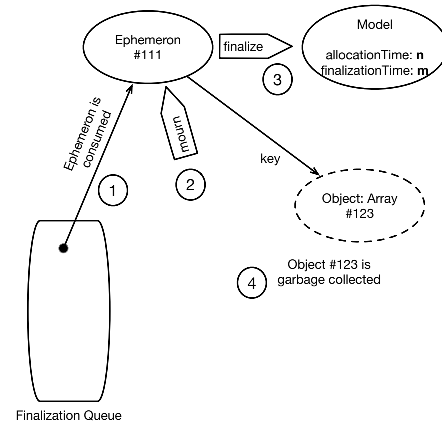

Figure 2: The finalization of an object at a time m

Figure 1 describes an object's allocation execution. First, a sender requests an object allocation. For example Array class is requesting an object allocation invoking the method Behavior»basicNew . After the allocation is produced, the instrumentation captures the execution before the object is returned to the sender. Inside the instrumentation, an object's model is created that will hold the object's allocation and finalization time and its size in memory. The object's allocation time is set and an ephemeron is instantiated.

When the object is being garbage collected the ephemeron will set its finalization time in the object's model. Figure 2 shows an object's finalization. First, the GC detects that an ephemeron can be finalized because it is unreachable. The GC will put the ephemeron into a finalization queue. Then, the GC will consume the ephemerons from the finalization queue one by one. The message mourn will be sent to the ephemeron as part of the finalization process. The ephemeron will send the message finalize to its finalizer, which is the object's model. This finalizer will set the object's finalization time. Finally, the GC frees the memory of the object.

An object's finalization time is not exact, it rather depends on when the GC detects that the object is unreachable. As discussed in Section 2, an object is detected as unreachable when the GC traverses the memory in both, the new and the old space.

4. Improving *DataFrame* performance through lifetime profiling

This section describes the effectiveness of our solution. We answer the following research question:

- *Does our lifetime profiler provide information that can lead to improvements in GC performance?*

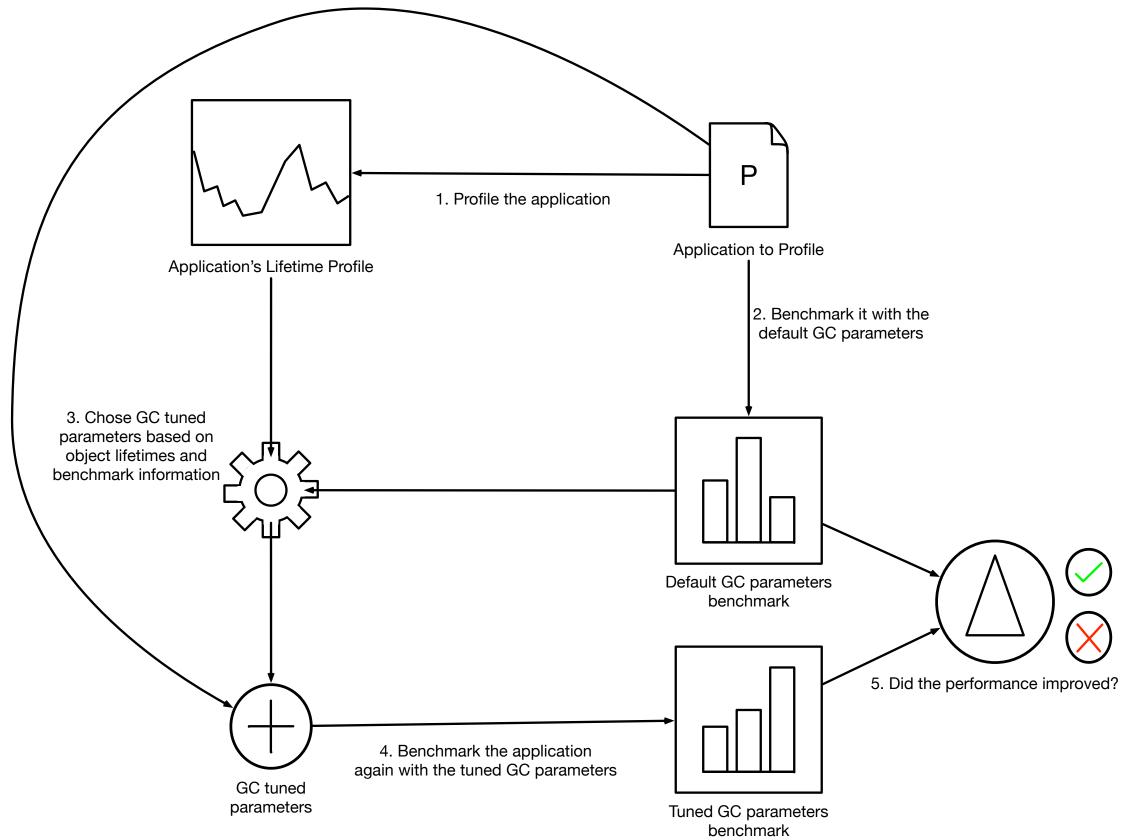

Figure 3: Validation methodology

We respond to this question positively. We evaluated our profiler by profiling the execution of loading a 500 MB CSV file into a *DataFrame*. We choose *DataFrame* as our target application because it is a memory-intensive application that supports loading big files to do data engineering and machine learning [12].

4.1. Methodology

We applied the following methodology for validating our profiler: We profile the execution of a given application and we then benchmark it to know how much time was spent on garbage collections. Then, we choose non-default GC parameters taking into consideration the object lifetimes and the GC spent time. If the profiler reports a significant quantity of long-lived objects, we choose non-default GC parameters that increase the memory heap and reduce the number of full garbage collections. With these custom GC parameters, we benchmark again our target application. We finally validate our profiler comparing the performance with the tuned GC parameters against the default ones. Figure 3 explains this process.

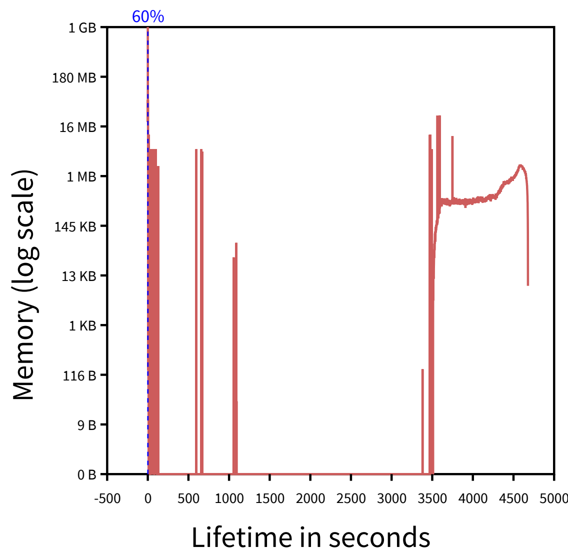

Figure 4: Object lifetimes profile by memory for a 500MB dataset

4.2. Lifetime profiling results

The profiler gives a density chart of the object lifetimes. We grouped the object lifetimes by intervals of one second. All objects whose lifetime duration has the same second will be in the same bucket. In Figure 4, we calculated the density as a function of the actual memory size occupied by the objects. We can observe that around 40% of memory stayed referenced for a long time.

In Figure 5 we calculated the density but in function by the number of objects instead of the occupied memory. Crossing this information with Figure 4 we get that 25% of the objects that represent 40% of the GC memory stay referenced for a long time.

4.3. Benchmarking *DataFrame*

We benchmarked the loading of a *DataFrame* but this time without the instrumentation. We used the default GC parameters when running these benchmarks. To improve the reproducibility of benchmarking, we used the best developer techniques for the benchmarks [16]: we cut the internet connexion and stopped all non-vital applications. We run each of the benchmarks n-times and then we reported the average execution time with the standard deviation. We used *benchy*[5] as the infrastructure for running the benchmarks. *Benchy* is an open-source application that serves as configurable infrastructure to run customizable benchmarks in Pharo. We benchmarked the loading of 3 CSV files of different sizes: 500 MB, 1.6 GB, and 3.1 GB.

We present the results of the benchmarks for the 3 different CSV files in Table 1.

[5]https://github.com/tesonep/benchy

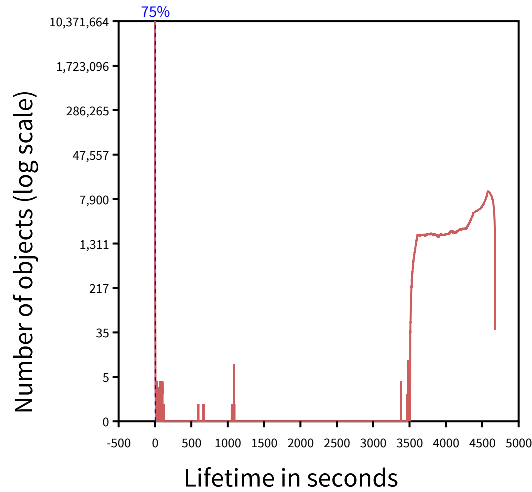

Figure 5: Object lifetimes profile for a 500MB dataset by number of objects

Table 1
Benchmark when loading a *DataFrame* with the default GC parameters

Dataset	# of scavengers	# of full GCs	GC time	Total time	GC time in %
500 MB	266	18	11 sec	1 min 11 sec	15%
1.6 GB	304	36	1 min	4 min 8 sec	22%
3.1 GB	1143	309	1 h 3 min 13 sec	1 h 11 min 5 sec	89%

The benchmarks that are presented in Table 1 show that there is a significant part of the execution time spent doing garbage collections, going from 15% to 89%, with the default GC parameters.

4.4. Tuning garbage collector parameters

Generational GCs suffer from poor performance when dealing with memory-intensive applications that generate numerous intermediate objects, which live a fairly long time under the default GC parameters [17, 13]. Our profiler showed that *DataFrame* is an application that produces long-lived objects. *DataFrame* has 25% of the objects that represent 40% of the allocated memory that live for a fairly long time (Figures 4, 5). Increasing the GC heap size has a significant impact on GC performance [7, 4]. One can reduce GC time by tuning the GC parameters [6]. The benchmarks exhibited optimization opportunities by reducing the garbage collection time.

Generational GCs are not optimized for applications with a substantial number of long-lived objects. Instead, they are specifically configured for applications where objects tend to die young [13]. We discussed with Pharo experts about Pharo's GC implementation details. With this internal knowledge of Pharo's GC implementation and the DataFrame internals, we chose

the 5 custom GC parameters. We chose by hand these 5 custom GC parameters that we knew will increase the heap size and reduce the number of garbage collections.

The Pharo GC has 4 available customizable parameters. In Pharo, the heap size can be seen as the sum of the new and the old space. One can control the heap size by tuning these customizable GC parameters.

- **Eden size** represents 5/7 of the new space and it is where the objects are allocated; the other 2/7 of space is used for copying objects.
- **Growth headroom** is the minimum amount of space the GC will request to the operating system to allocate
- **Shrink threshold** is the amount of free space that the GC will manage before giving it back to the OS.
- **GC ratio** is a percentage that triggers a full garbage collection when the heap will grow that percentage.

Then as explained in Figure 3, we benchmarked the loading of the *DataFrame* with the 3 different CSV files with these 5 GC parameter configurations to see if we reduce the garbage collection time. We obtained improvements of up to 6.8 times compared to the default GC parameters.

Table 2 describes the 5 non-default GC parameters that we chose.

Table 2
GC tuning parameter configurations

Configuration	Eden size	Growth headroom	Shrink threshold	GC ratio
Default	15MB	16MB	32MB	33%
Configuration 1	64MB	64MB	128MB	250%
Configuration 2	150MB	128MB	128MB	250%
Configuration 3	300MB	128MB	128MB	500%
Configuration 4	300MB	256MB	256MB	1000%
Configuration 5	300MB	512MB	512MB	1000%

In the following tables, we present the results that we obtained re-running the 3 benchmarks with the 5 configurations. For the 500 MB CSV, we got from the benchmark in Table 1 that 15% of the time is spent on garbage collecting. In Table 3 we observe that we increased the performance up to 1.2 times.

In Table 4 we observe that we increased the performance up to 1.2 times for loading the 1.6 GB CSV file into the *DataFrame*. For this CSV file, we got 22% execution passed doing garbage collections. The performance improvements are the same as the precedent benchmark.

In Table 5 we observe that we increased the performance up to 6.8 times with configuration 5. This enhancement can be attributed to the fact that, for this 3.1 GB, 89% of the execution time is spent doing garbage collections. Configuration 5 has a larger memory footprint compared to the default parameters. Comparing Configuration 5 with the default GC configuration, when the GC requires more memory it allocates segments of 512 MB instead of 32 MB and the new space starts with 512 MB instead of 15 MB.

Table 3
Changing the parameters for the 500 MB *DataFrame*

GC Configuration	GC spent time	Total execution time	Improved performance
Default	11.18 sec	70.78 sec	1×
Configuration 1	4.08 sec	63.72 sec	1.1×
Configuration 2	2.91 sec	64.10 sec	1.1×
Configuration 3	2.07 sec	61.01 sec	1.1×
Configuration 4	2.21 sec	60.78 sec	1.2×
Configuration 5	2.17 sec	60.58 sec	1.2×

Table 4
Changing the parameters for the 1.6 GB *DataFrame*

GC Configuration	GC spent time	Total execution time	Improved performance
Default	60.31 sec	4 min 8 sec	1×
Configuration 1	42.48 sec	3 min 54 sec	1.1×
Configuration 2	17.97 sec	3 min 30 sec	1.2×
Configuration 3	14.99 sec	3 min 23 sec	1.2×
Configuration 4	11.69 sec	3 min 19 sec	1.2×
Configuration 5	12.93 sec	3 min 20 sec	1.2×

Table 5
Changing the parameters for the 3.1 GB *DataFrame*

GC Configuration	GC spent time	Total execution time	Improved performance
Default	58 min 18 sec	1 h 6 min 18 sec	1×
Configuration 1	9 min 41 sec	17 min 46 sec	3.7×
Configuration 2	4 min 57 sec	12 min 54 sec	5.1×
Configuration 3	5 min 8 sec	13 min 2 sec	5.1×
Configuration 4	2 min 42 sec	10 min 37 sec	6.2×
Configuration 5	1 min 47 sec	9 min 42 sec	6.8×

5. Discussion

We describe some challenges of object lifetimes profile that we faced during the writing of this paper:

GC custom parameters. We chose the GC parameters arbitrarily based on expert knowledge of Pharo's GC implementation and the object lifetimes of our target application. The parameters were chosen with the objective of increasing the heap size and reducing the number of garbage collections.

Cost of the instrumentation. Our implementation relies on instrumentation. We measured the impact of it and we got an overhead of at least 50 times According to our measurements, the ephemerons have the biggest impact on the overhead. For this reason, we were not able to profile loading a bigger CSV file. Nevertheless, the lifetime profile that we got was useful for taking GC optimization decisions for the 3 different CSV files that we benchmarked.

GC Stress. For tracking the object lifetimes we allocate an object's model that stores the allocation and finalization time of the object as long as the size in memory. We also allocate an

ephemeron to finalize the object when it will be collected. This introduces stress to the GC as we are multiplying the allocations by three.

Ephemeron's Queue. When the GC detects that an ephemeron is a candidate for collection, it puts it into a finalization queue. Once in the queue, it creates a strong reference to its key object, the one that is being garbage collected. If one of the objects in the queue is the root of a sub-graph of unreachable objects, this strong reference will avoid the whole object sub-graph from being collected and it will only collect the root. This can delay the collection of some objects, increasing their lifetime.

Nepotism. Nepotism [13] happens when tenured garbage makes its referenced objects, which are also garbage, to be tenured too. This happens when an object that is about to die gets tenured. As the object is now in the old space, until a full GC is triggered, the GC will not detect that the object became unreachable, it will treat it as reachable until the next run of the full garbage collection. If this tenured garbage references other objects, those objects become candidates for being tenured and can eventually get tenured too. This affects negatively the GC performance since it passes time copying and updating references of garbage. This problem is not specific to our profiler nor to the instrumentation but to the generational GCs themselves.

6. Related work

Automatic GC parameter tuning. Lengauer et al. [6] proposed a technique to automatically tune the GC parameters for the Java programming language. They used an optimization algorithm that adjust the parameters with the objective function being the aggregated garbage collection time. The authors ran their experiments on the Java 8 Hotspot™ VM. The difference with our work is that they optimized the GC performance automatically in a black-box manner while we used the object lifetimes information given by our profiler to take decisions about the appropriate GC parameters.

Adaptative tenuring policies. Ungar et al. [13] instrumented a Smalltalk VM to track the object lifetimes. They use this information to change the tenuring policies to reduce the amount of tenuring garbage. Their objective was to optimize the GC performance. Their work differs from ours in the sense that they used to object lifetimes information to change the tenuring policies while we used it to tune the GC parameters.

GC infrastructure evaluation. Kaleba et al. [17] benchmarked the GC of the Cog VM for one memory-intensive application. The difference with our work is that we used the object lifetimes profile to choose the custom GC parameters for our application. We then benchmarked our application with these different GC configurations and compared their performance. Kaleba et at. ran their benchmarks with different GC algorithms and modified only one GC parameter: the Eden size. They chose the Eden size value with their application's knowledge without any profile information.

Profiled-based pretenuring Blackburn [5] et al. used profile information to make decisions for pre-tenuring objects. In their approach, they took objects allocation information such as the object lifetimes to pre-tenure objects for increasing performance. Our approach is different in the sense that we use the object lifetimes to tune the GC parameters.

Hybrid finalization mechanism. Valloud [9] implemented a new finalization mechanism

for the HPS Smalltalk VM. Valloud explains that there were performance issues with the two available finalization mechanisms: Weak Arrays and Ephemerons. Weak arrays are described as inefficient and ephemerons introduce a large memory overhead. The author developed a new finalization mechanism that combines both approaches for improving performance. We observed the same large memory overhead caused by the ephemerons in our use case.

7. Conclusion and future work

We developed a lifetime profiler that tracks the lifetime of objects that were allocated during the execution of an application. It tracks the object lifetimes of a given application via instrumentation by controlling the execution whenever an object is allocated. We profiled the object lifetimes of a *DataFrame* when loading a big CSV file. We evaluated our solution by observing how object lifetimes relate to performance improvements when tuning the GC. We obtained a profile that exhibited that there were 25% of objects that represent 40% of the memory that lived for a fairly long time.

We discussed with Pharo experts about the implementation details of Pharo's GC. With the knowledge of how Pharo's GC is implemented and the fact that *DataFrame* produces a fairly large number of long-lived objects, we chose 5 GC parameter configurations that increase the heap size. We then benchmarked the loading of 3 big CSV files into a *DataFrame* with these 5 custom parameters and we observed a performance increase up to 6.8 times compared with the default GC parameters.

As future work, we plan to profile the object lifetimes at the virtual machine level to reduce the overhead that the instrumentation introduces. To mitigate the GC stress we want to explore the idea of sampling the object allocations and compare the sampling precision with taking all the allocations. We also aim to explore automated techniques for detecting an optimal set of GC parameters to enhance an application's performance. Finally, we also want to explore taking pre-tenuring decisions using the object lifetimes profile information.

8. Acknowledgments

We express our gratitude to the anonymous reviewers for providing valuable comments on the draft of this paper.

References

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

[2] G. Polito, P. Tesone, J. Privat, N. Palumbo, S. Ducasse, Heap fuzzing: Automatic garbage collection testing with expert-guided random events, in: International Conference on Software Testing, 2023.

[3] E. Miranda, C. Béra, E. G. Boix, D. Ingalls, Two decades of Smalltalk VM development: live VM development through simulation tools, in: Proceedings of International Workshop on

Virtual Machines and Intermediate Languages (VMIL'18), ACM, 2018, pp. 57–66. doi:10.1145/3281287.3281295.

[4] T. Brecht, E. Arjomandi, C. Li, H. Pham, Controlling garbage collection and heap growth to reduce the execution time of java applications, ACM Sigplan Notices 36 (2001) 353–366.

[5] S. M. Blackburn, M. Hertz, K. S. Mckinley, J. E. B. Moss, T. Yang, Profile-based pretenuring, ACM Transactions on Programming Languages and Systems (TOPLAS) 29 (2007) 2–es.

[6] 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.

[7] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, J. E. B. Moss, Automatic heap sizing: Taking real memory into account, in: Proceedings of the 4th international symposium on Memory management, 2004, pp. 61–72.

[8] B. Hayes, Ephemerons: A new finalization mechanism, in: International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'97), 1997. doi:10.1145/263700.263733.

[9] A. Valloud, Linked weak reference arrays: A hybrid approach to efficient bulk finalization, in: Proceedings of the International Workshop on Smalltalk Technologies, 2015, pp. 1–6.

[10] S. Ducasse, G. Rakic, S. Kaplar, Q. D. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, M. Denker, Pharo 9 by Example, Book on Demand – Keepers of the lighthouse, 2022. URL: http://books.pharo.org.

[11] O. Zaytsev, N. Papoulias, S. Stinckwich, Towards exploratory data analysis for pharo, in: Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies, 2017, pp. 1–6.

[12] O. Zaitsev, S. Jordan Montaño, S. Ducasse, How fast is ai in pharo? benchmarking linear regression, in: IWST 2022-International Workshop on Smalltalk Technologies, 2022.

[13] D. Ungar, F. Jackson, An adaptive tenuring policy for generation scavengers, ACM Transactions on Programming Languages and Systems (TOPLAS) 14 (1992) 1–27.

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

[15] A. Bergel, Counting messages as a proxy for average execution time in pharo, in: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11), LNCS, Springer-Verlag, 2011, pp. 533–557. URL: http://bergel.eu/download/papers/Berg11c-compteur.pdf.

[16] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous java performance evaluation, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '07, Association for Computing Machinery, New York, NY, USA, 2007, pp. 57–76. URL: https://doi.org/10.1145/1297027.1297033. doi:10.1145/1297027.1297033.

[17] S. Kaleba, C. Béra, E. Miranda, Garbage collection evaluation infrastructure for the cog vm, in: Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems Workshop, ICOOOLPS'18, 2018.