Evaluating Finalization-Based Object Lifetime Profiling

Sebastian Jordan Montaño
Univ. Lille - Inria - CNRS - Centrale Lille - UMR 9189 CRIStAL
Villeneuve D'Ascq, France
sebastian.jordan@inria.fr

Guillermo Polito
Univ. Lille - Inria - CNRS - Centrale Lille - UMR 9189 CRIStAL
Villeneuve D'Ascq, France
guillermo.polito@inria.fr

Stephane Ducasse
Univ. Lille - Inria - CNRS - Centrale Lille - UMR 9189 CRIStAL
Villeneuve D'Ascq, France
stephane.ducasse@inria.fr

Pablo Tesone
Univ. Lille - Inria - CNRS - Centrale Lille - UMR 9189 CRIStAL
Villeneuve D'Ascq, France
pablo.tesone@inria.fr

Abstract

Using object lifetime information enables performance improvement through memory optimizations such as pretenuring and tuning garbage collector parameters. However, profiling object lifetimes is nontrivial and often requires a specialized virtual machine to instrument object allocations and dereferences. Alternative lifetime profiling could be done with less implementation effort using available finalization mechanisms such as weak references.

In this paper, we study the impact of finalization on object lifetime profiling. We built an actionable lifetime profiler using the ephemeron finalization mechanism named FiLiP. FiLiP instruments object allocations to exactly record an object's allocation time and it attaches an ephemeron to each allocated object to capture its finalization time. We show that FiLiP can be used in practice and achieves a significant overhead reduction by pretenuring the ephemeron objects. We further experiment with the impact of sampling allocations, showing that sampling reduces profiling overhead while maintaining actionable lifetime measurements.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; **Software performance**; **Garbage collection**.

Keywords: profiling, garbage collection, finalization

ISMM '24, June 25, 2024, Copenhagen, Denmark
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0615-8/24/06
https://doi.org/10.1145/3652024.3665514

ACM Reference Format:
Sebastian Jordan Montaño, Guillermo Polito, Stephane Ducasse, and Pablo Tesone. 2024. Evaluating Finalization-Based Object Lifetime Profiling. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management (ISMM '24), June 25, 2024, Copenhagen, Denmark.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3652024.3665514

1 Introduction

Object lifetime information is crucial to optimize performance through memory optimizations such as pre-tenuring or tuning garbage collector parameters [7, 17]. Implementing algorithms that compute object lifetimes in a precise and scalable manner is nontrivial and requires often a modified virtual machine for execution [4, 12, 13, 28? ?]. For example, Hertz et al. [12, 13] introduced a perfect tracing algorithm that computes object lifetimes called Merlin. However, Merlin has an overhead between 70 and 300 ×.

One practical alternative used in the past is to use finalization mechanisms such as weak references to estimate object lifetimes [1, 22]. However, the topic in question was never explored in depth. In this paper, we explore the profiling of object lifetimes using the ephemeron finalization mechanism [11]. In a nutshell, we instrument object allocations to trace object *birthtime* and we attach an ephemeron to each object to be notified when the object becomes collectible. The main challenge is that naively using such a mechanism attaches an ephemeron to each allocated object stressing the memory manager, adding extra runtime overhead and negatively impacting the precision of the lifetime measures.

To address these challenges, we study the impact of sampling in lifetime profiling. For this purpose we have developed FiLiP, a lifetime profiler working on top of the Pharo Virtual Machine and available under the MIT open-source license. Our solution utilizes the underlying ephemeron support with Pharo's generational scavenger memory manager and implements sampling to scope instrumentation.

We first validate that our profiler is actionable [21], verifying that there is a causal connection between the benchmarked applications and our measurements*i.e.,* If we action on application source code to increase object lifetimes, we should see a change in the measured object lifetimes. Second, we show that pretenuring ephemerons and sampling allocations significantly reduce profiling overhead. Finally, we show that our profiler achieves similar precision across different sampling rates when measuring object lifetimes. These results indicate that finalization and sampling are promising solutions to build object lifetime profilers.

The contributions of this paper are:

- Empirical evidence that a finalization-based profilers can be weakly actionable and have low overhead.
- Empirical evidence of how pre-tenuring ephemerons significantly reduces overhead in the stressful setting that is allocation profiling, making profiling practical.
- Empirical evidence that sampling allocations reduce further profiling overhead while still providing relevant object lifetime information.

Paper's outline. The paper is structured as follows: Section 2 presents the problems of profiling object lifetimes using a finalization mechanism. Section 3 explains the core design of our finalization profiler used for experimentation. Section 4 presents our experimental setup, research questions, and experimental methodology. Section 5 presents our validation and answers the research questions. Section 7 presents the related work and Section 8 concludes the paper.

2 Challenges of finalization-based object lifetimes

The ideal lifetime of an object spans from the moment it is allocated to the moment it becomes unreachable. Measuring such an ideal lifetime requires instrumenting the memory manager to track allocations and dereferences with algorithms such as Merlin [12, 13]. In this paper, we explore the tradeoffs of measuring object lifetimes with a finalization mechanism. Using the finalization mechanism available in existing implementations allows for retrofitting lifetime profiling to an existing language runtime without major changes in the memory manager.

2.1 Ephemeron finalization

Finalization is a runtime mechanism that allows developers to hook into object deallocation. The objective of finalization is to notify about object deallocation, generally by executing a user-defined callback when this happens. Such a mechanism is useful to *e.g.,* dynamically free runtime resources such as file descriptors when objects become unreachable.

One popular finalization mechanism is ephemeron finalization [11], present in languages such as Lua [14], OCaml [27], Racket [9], Squeak [2] and Pharo [24]. Ephemerons are key-value pairs that strongly hold their values as long as their keys are referenced. When the key becomes unreachable in the object graph by objects other than the ephemeron, the memory manager *mourns* the ephemeron and triggers the finalization callback.

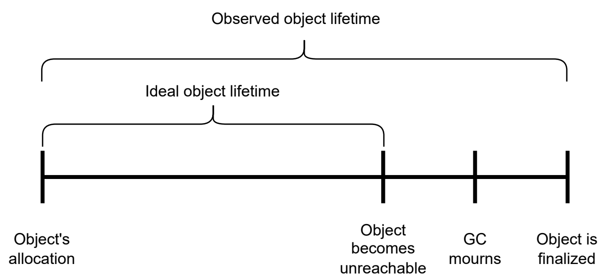

Figure 1. An object's lifetime.

Mourning typically happens during garbage collection (GC), thus the mourned ephemerons found during a GC cycle are queued and not treated immediately. Finalization callbacks are then executed when control returns to the execution engine: ephemerons are dequeued one by one and their attached objects are finalized.

2.2 Object lifetime and finalization

Let us consider a profiler that computes object lifetimes using the finalization approach described before. Such a profiler will store a timestamp when an object is allocated and register an ephemeron for it. When the object is finalized, the profiler stores a second timestamp. Such a profiler suffers several issues on how lifetimes are measured, as illustrated in Figure 1.

Problem 1: Delayed observation time. There is a potentially large distance imposed by design between the moment when an object becomes unreachable and its finalization. Ideally, an object's lifetime should indicate the time when it was dereferenced and became unreachable. However, since mourning happens only during GC cycles, a profiler will never observe lifetimes under the period between GCs. Moreover, the mourning process queues objects for later finalization by the runtime engine. Thus, the time interval between an object becoming unreachable and its finalization time is influenced by the latency of the program executing the finalization. This difference between the dereferencing time and the finalization time can lead to imprecise measurements.

Problem 2: Ephemeron contamination. Ephemerons, generally treated by implementations as normal objects, add additional stress to the memory manager. This stress may negatively impact the period and overhead of GCs, and hurt the lifetime measurement. This increased overhead may even render the solution unaffordable in terms of performance, thus posing a significant challenge.

3 Finalization-based lifetime profiling

In this paper, we explore the profiling of object lifetimes using the ephemeron finalization mechanism [11]. We have developed FɪLɪP, a lifetime profiler working on top of the Pharo Virtual Machine, available under the MIT open-source license and based on the Illimani memory profiling framework [1]. In a nutshell, FɪLɪP instruments object allocations to trace object *birthtimes* and attaches ephemerons to each object to be notified when the object becomes collectible. To investigate the impact of the delayed observation time and ephemeron contamination, FɪLɪP improves Illimani with two key extensions: customized pre-tenured ephemerons and configurable object sampling.

In the rest of this section, we explain the inners of our profiler. Later, in Section 4 we explain our experimental setup and methodology, and Section 5 presents our experimental results on five different application benchmarks.

3.1 Instrumenting allocations

FɪLɪP dynamically instruments object allocations by taking advantage of that in Pharo all allocations occur through message sending [3]. It is thus enough to instrument the methods responsible for allocation. We implement such an instrumentation with a method proxy library re-implementation of method wrappers [5]. Method wrappers are user-defined hooks that are invoked before and after an instrumented method is invoked. The method wrapper library makes sure that allocations produced by the profiler itself are not instrumented as well, as may be the case in a reflective environment such as Pharo.

When an allocator method is invoked, the after hook will contain the return value of the original method, which is the newly allocated object. We then register its allocation time and attach an ephemeron to it.

3.2 Customizing finalization

The standard ephemeron framework in Pharo produces two objects per ephemeron: the ephemeron key-value object provided by the standard library, and the user-defined finalizer. To minimize the pressure in the memory manager we customize ephemerons in our profiler to use a single object fulfilling both roles. The ephemeron is thus also a finalizer to avoid additional allocations.

Furthermore, as we know in advance that the ephemerons will live at least longer than the object they observe, we optimize the allocation by pre-tenuring ephemerons. This approach reduces the pressure on the scavenger algorithm, which needs to run quickly and frequently. They will still impose some overhead because ephemerons will be tracked in the memory manager's remembered set because their key will be allocated as young objects.

[1]https://github.com/jordanmontt/illimani-memory-profiler

3.3 Sampling support

FɪLɪP introduces built-in support for configurable sampling allocation events. All our experiments below use different sampling rates. Each sampling rate is represented as a reduced fraction. For example, a 20% sampling rate is represented as 1/5, meaning the profiler will capture 1 out of every 5 produced allocations. We leave for future work the study of different sampling approaches, such as using time windows.

4 Experimental setup

This section presents the general context of our experiments, including our research questions, the methodologies followed, and the list of benchmarks we profiled. The detailed results of our experiments are shown and discussed in Section 5.

4.1 Research questions

This paper studies the following research questions on the FɪLɪP profiler and its design.

- **RQ.1 - Actionability.** Is our profiler actionable? If we act on the application source code to extend the object lifetimes, does the profiler report reflect these changes? This first question aims to validate our profiler implementation.
- **RQ.2 - Object lifetime precision across sampling rates.** How do different sampling rates impact the computed object lifetimes? Do the object lifetimes vary across these different sampling rates? This second question aims to evaluate the precision of the profiler, and how sampling impacts the measurements.
- **RQ.3 - Execution time overhead.** Does sampling object allocations reduce the overhead introduced by the finalization profiler, enabling its use without significant performance impact? This third question aims to validate how practical is such a profiler.
- **RQ.4 - Memory overhead.** How much memory overhead does our profiler introduce? This fourth question also aims to validate how practical our profiler is.

4.2 Experimentation platform

Our implementation resides in the Pharo programming language [8], which has a mature virtual machine implementation in stable production usage for over a decade [16, 19, 20, 23, 24]. Pharo implements generational scavenger [**?**] with a remembered set implemented as a sequential store buffer, and a mark-compact GC for the older generation [**?**]. Pharo implements Ephemerons [24], a finalization mechanism allowing the execution of user-defined actions when an object is about to be garbage collected [11]. We run the experiments on a MacBook M2 Pro with 16 GB of memory running OSX 14.3.1.

4.3 Benchmarks

We run all our experiments in the following set of benchmarks. These benchmarks are programs extracted from existing production libraries/applications.

- **DataFrame**: DataFrame [25] is a tabular data structure commonly used for data analysis. We load a synthetic dataset that follows a linear distribution with some noise. The dataset weighs 230 MB and it has 2000000 rows and 6 columns. This scenario is interesting because of the amount of objects manipulated and the execution time of the computation.
- **Honey Ginger**: A smoothed-particle hydrodynamics simulator with rich visualization and interactivity. We render one simulation for 1000 rendering cycles.
- **Re:Mobidyc**: A multi-agent simulator for individual-based modeling in population dynamics and ecotoxicology. We run a simulation where wolves chase and eat goats in a grass field. The simulation shows the evolution of the wolves and goats population. The simulation takes about 2-3 minutes to finish.
- **Bloc**: Bloc is a low-level UI infrastructure and UI framework for Pharo. We executed the Boids benchmark, which renders moving figures that simulate the flocking behavior of birds. We let the benchmark execute for 10 seconds.
- **Moose**: Moose is an open-source extensive platform for software and data analysis. It offers multiple services ranging from importing and parsing data to model, measuring, querying, mining, and building interactive and visual analysis tools. We loaded a large software database from a private company into the Moose meta-model. The software model has 13521 classes and 48087 methods.

4.4 General methodology

We applied the following methodology to validate FiLiP and to respond to the research questions. We profiled each benchmark using four different sampling rates: 0.1%, 1%, 50%, and 100%. These sampling rates were chosen to provide a wide range of comparisons.

When presenting results based on time (e.g., lifetimes and overhead) we use execution time relative to the uninstrumented benchmark which serves as baseline. For example, instead of expressing that an object lived for 62 out of 77.5 seconds, we will state that it lived for 80% of the total execution time. This is because we expect the profiler to introduce execution overhead, which will vary according to the sampling rate. With a higher sampling rate, we will exert more stress on memory, thereby increasing garbage collection time, and thus affecting the absolute measurements.

Definitions.

- **Most allocated classes.** We consider a class as 'most allocated' if its instances represent at least 1% of the total allocations. If an application has too few or too many such classes, the threshold can be adjusted, although for our analysis the 1% threshold showed good results.
- **Short-lived and long-lived object.** We consider all instances of a class as *short-lived* if their average lifetime is less or equal to 5% of the benchmark execution time. We consider all other instances as *long-lived*. Notice that although this threshold will vary depending on the application, we use it only to classify instances in our actionable experiments and to present results.

4.5 Actionability experiments

As discussed in Section 2, our approach does not provide the exact object lifetimes, but an approximation. To validate FiLiP, we adopt the definition of actionable profilers proposed by T. Mytkowicz [21]. T. Mytkowicz [21] et al. define actionable profilers as follows:

"To evaluate if a profiler is actionable, we use causality analysis. Causality analysis works by intervention: we change our system (the intervention) and then check if the intervention yields the predicted performance. If the prediction holds, then causality analysis gives us confidence that the profiler is actionable; if the prediction does not hold, causality analysis indicates that the profiler is not actionable."

In our case, since we are evaluating memory rather than performance, we intervene in the application's source code to alter object lifetimes. Given the difficulty of reducing object lifetimes, we opted for increasing the lifetime of short-lived objects. We will name *baseline profiler* the finalization profiler, for which no action was taken. Conversely, the profiler in which we took action to increase lifetimes is named the *actionable profiler*.

In our actionability experiments, we instrument each benchmark to keep references to instances of short-lived classes at allocation time, to prevent them from being garbage collected until the end of the application. We compare the average lifetimes of the actionable profiler with those of the baseline profiler. We expect that lifetimes measured by the actionable profiler should increase. Additionally, for the *DataFrame* benchmark we present an in-depth analysis of the most allocated classes across the different sampling rates.

4.6 Object lifetime precision across sampling rates

To study the precision of our profiler, we compare the overall lifetime frequencies for each sampling rate relative to the total benchmark execution time. Allocation histograms bin allocations by second. We compare the results across all sampling rates to assess the degree of variation between them. We expect consistent results across all sampling rates.

Additionally, for the *DataFrame* benchmark we present a in-depth analysis of the most allocated classes across the different sampling rates.

4.7 Execution time and memory overhead

To study the overhead of our profiler, we executed each benchmark across 5 configurations: without the profiler and with the profiler active at sampling rates of 100%, 50%, 1% and 0.1%. We used ReBench [18] to drive our benchmark execution, using 30 VM iterations with 1 benchmark iteration each [10]. We strongly believe that one benchmark iteration per VM invocation does not alter our results because our target language implementation, the Pharo VM, has a non-optimizing baseline JIT compiler that JIT compiles on the second method invocation and does not apply optimizations that impact allocation behavior such as scalar replacement. For each configuration, we report the mean value of the measurements. To avoid interference from other applications during benchmark execution at maximum during benchmark execution, we stopped all OS applications that could be stopped and we cut the internet connection.

We measured baseline memory consumption with our profiler. We added up the size of all allocated objects at 100% of sampling. We then estimated the incurred overhead by computing the size of a FiLiP ephemeron instance multiplied by the total number of allocations at each sampling rate.

5 Results

This section presents the results of our research questions using all our benchmarks. When applicable, we perform an in-depth analysis on the DataFrame [25] benchmark.

5.1 RQ.1 - Actionability

This section presents our actionability experiments. First, we show how actionable are the overall averages of lifetimes in the different benchmarks. Then we dive into the DataFrame case study and show a histogram of lifetime variations of both overall lifetimes and per most allocated class.

5.1.1 Acting on lifetimes. This subsection shows the effect of actioning on object allocation for all of our benchmarks. As it was said in Section 4.5, we act on the *most allocated short-lived classes* of each benchmark. Following we list the classes and the percentage of objects allocated.

- **DataFrame.** NumberParser (27.9%), Fraction (27.9%), and ReadStream (27.9%).
- **HoneyGinger.** Rectangle (1.02%), Array (0.74%), and Form (0.22%).
- **Re:Modidyc.** Array (28.7%), Dictionary (14.9%), and RMDFishmanMooreRandomGenerator (7.4%).
- **Bloc.** BlChildrenSortedByElevation (30%), BlVector2D (1.6%), and WriteStream (2%).
- **Moose.** WideString (15%), ValueLink (1.7%), and Set (1.4%).

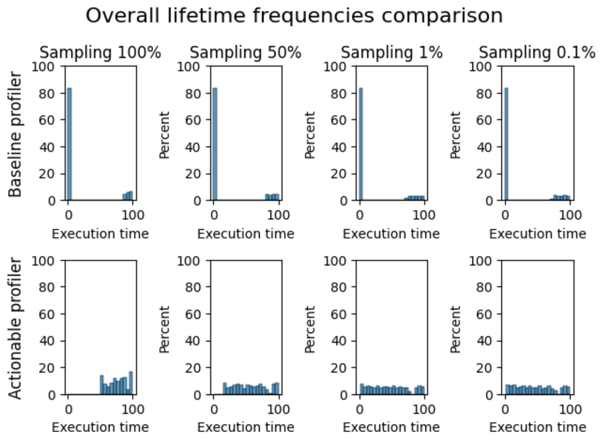

Figure 2. Comparison of the overall lifetime relative frequencies: baseline vs. actionable profiler in DataFrame.

Table 1 shows the difference in average lifetime between the baseline and the actionable profilers when applying the different sampling rates (columns) per benchmark (rows).

5.1.2 Actionable lifetime averages in DataFrame. In this section, we dive into the DataFrame benchmark to study how actionability in object lifetimes happen at the class level. Table 2 presents the difference in the average lifetimes between the actionable and the baseline profiler for the most allocated classes in the benchmark. Two groups exist: the short-lived and the long-lived ones. The short-lived group shows differences ranging from 72% to 38%. The long-lived shows differences from 4% to 6%. The classes on which we took action, NumberParser, Fraction, and ReadStream, exhibit significant differences in their average lifetimes. This difference in variation is expected as we did not act on the long-lived objects.

5.1.3 Actionable lifetime frequencies in DataFrame. Figure 2 presents the overall lifetime frequencies for all allocated objects across the four different sampling rates. Each row plots the overall lifetime at a different sampling rate. The first column shows that we go from a situation (in the top) where most of the objects are short-lived to a situation (in the bottom) where objects lived from 50% to 100% of the execution time. In addition, the plots of the second row (actionable profiler) show that the lifetime is spread from 50% to 100% for the first plot and from 0% to 100% for the three last plots.

Figure 3 splits the same analysis for the short-lived classes (Fraction, NumberParser, and ReadStream). Similarly to the overall lifetime, the results show an increased lifetime for the classes that we acted upon.

Table 1. Differences in overall lifetimes in the baseline vs. actionable profiler for all benchmarks.

	100% Sampling	50% Sampling	1% Sampling	0.1% Sampling
DataFrame				
Difference in average lifetimes	+60%	+42%	+32%	+33%
HoneyGinger				
Difference in average lifetimes	+76%	+56%	+47%	+47%
Re:Mobidyc				
Difference in average lifetimes	+34%	+32%	+24%	+23%
Bloc				
Difference in average lifetimes	+25%	+13%	+13%	+14%
Moose				
Difference in average lifetimes	+10%	+9%	+10%	+10%

Table 2. Differences in average lifetimes in the baseline vs. actionable profiler in DataFrame.

	Fraction	NumberParser	ReadStream	ByteString	Array
100% Sampling					
Difference in average lifetimes	+72%	+72%	+72%	+4%	+4%
50% Sampling					
Difference in average lifetimes	+49%	+49%	+49%	+4%	+4%
1% Sampling					
Difference in average lifetimes	+38%	+38%	+38%	+6%	+6%
0.1% Sampling					
Difference in average lifetimes	+38%	+38%	+38%	+6%	+6%
		short-lived		**long-lived**	

RQ.1 Conclusion. Acting on objects resulted in a noticeable increase in the observed average lifetimes. We observed significant differences across all sampling rates, ranging from 10% to 76%. Object graphs revealed an anchoring effect: all objects referenced by an affected object will also increase their lifetimes. These findings suggest that our finalization profiler provides **weak actionability**: while overall average lifetimes are increased, graph anchoring prevents predicting the expected increase.

5.2 RQ.2 - Object lifetime precision across sampling rates

This section presents the results of our profiler's precision across sampling rates. First, we show how the average measured lifetimes vary across sampling rates for each application. Then we dive into the DataFrame case study and show a matrix of lifetime histograms for both overall lifetimes and per most allocated class.

5.2.1 Precision across average lifetimes. This subsection shows how lifetime measurements are affected across different sampling rates. As stated in Section 4.4, we report lifetime measurements relative to the benchmark execution time to cope with the instrumentation-induced overheads. Table 3 shows the average lifetime applying the different sampling rates (columns) per benchmark (rows).

Our results show that sampling shows little variation in the observed lifetimes for each application, and that variation is particularly low when average lifetimes tend to be longer.

5.2.2 Precision per most-allocated class in DataFrame. In this section, we dive into the DataFrame benchmark to study how object lifetimes vary at the class level. Table 4 presents per most-allocated class and across the different

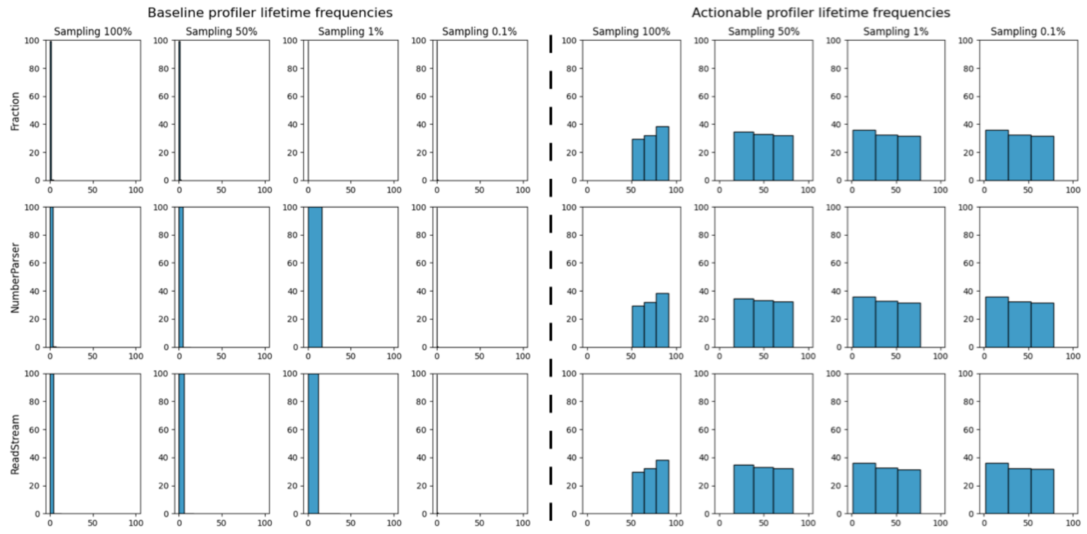

Figure 3. Baseline vs. actionable profiler relative lifetime frequencies by class.

sampling rates the following information: number of allocated objects relative to the number of objects, amount of memory allocated relative to the total memory allocated, and relative average observed lifetimes. The left part of the table shows long-lived classes, and the right short-lived classes.

The table shows that the percentage of captured allocations does not vary across sampling rates: 27.9% for Fraction, 27.9% for NumberParser, 27.9% for ReadStream, 13.9% for ByteString, and 2.3% for Array. Memory usage varies across sampling rates *e.g.,* the class NumberParser has 27.9% of allocations but the occupied memory varies from 52.3% to 34% because sampling affects the number of total objects captured. However, the percentage of memory used per class remains always in the same order of magnitude.

Regarding the average lifetimes, we observe slight reductions as we reduce the sampling rate while staying always in the same order of magnitude: for the short-lived objects, the average relative lifetime varies from 0.06% to 0.04% for Fraction (0.01% difference), and for the long-lived objects from 93.3% to 86.3% for ByteString (7% difference). This shows that although most instances of ByteString are long-lived, there are several short-lived instances and the results affected if less long-lived instances are sampled.

5.2.3 Lifetime frequencies in DataFrame.

Figure 4 presents the lifetime frequencies for all objects across four different sampling rates. We observe a bimodal distribution across all sampling rates. Approximately 85% of the total allocated objects exhibit lifetimes close to 0% of the execution time. The other 15%, in the three bins on the right, are below 10% and have a lifetime closer to the one of the benchmark.

It is worth noting the following about the use of a relative axis in the plots. Since the instrumentation of the profiler

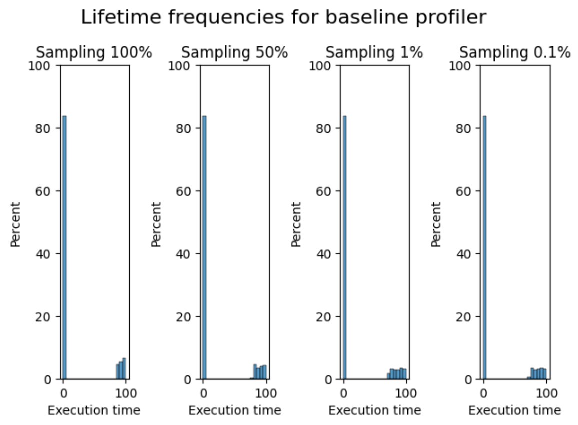

Figure 4. Lifetime relative frequency histograms for DataFrame.

may change the execution speed of the application, the execution time varies according to the number of captured. Not using the normalized histogram would make the comparison unfair. Time normalization is essential to the analysis because the overhead varies on the sampling rate.

5.2.4 Lifetime frequencies per most allocated class in DataFrame.

Figure 5 presents the lifetime frequencies per most allocated class in the DataFrame benchmark. Each row shows the lifetimes of a class across sampling rates. The three upper rows show lifetime frequencies for short-lived classes. The two lower rows show lifetime frequencies for long-lived classes. Reading this figure per column shows different classes per sampling rate, reading it per row shows class lifetime variation across sampling rates. Figure 5 shows that the lifetime distributions per class remain similar across sampling rates.

Table 3. Overall lifetimes for all benchmarks.

	100% Sampling	50% Sampling	1% Sampling	0.1% Sampling	**Avg±stdev**
DataFrame					
Overall average lifetimes	15.24%	14.68%	14.09%	14.21%	14.55%±0.45
HoneyGinger					
Overall average lifetimes	0.09%	0.08%	0.05%	0.06%	0.07%±0.014
Re:Mobidyc					
Overall average lifetimes	0.5%	0.36%	0.46%	0.18%	0.37%±0.12
Bloc					
Overall average lifetimes	6.27%	6.27%	6.51%	6.63%	6.42%±0.15
Moose					
Overall average lifetimes	13.19%	13.33%	13.67%	12.82%	13.25%±0.3

Table 4. Comparison of DataFrame with different sampling rates.

	Fraction	NumberParser	ReadStream	ByteString	Array
100% Sampling rate					
Allocations (Total: 86,002,607)	27.9%	27.9%	27.9%	13.9%	2.3%
Memory	11.4%	45.6%	15.2%	7.6%	20.2%
Avg. Lifetime	0.06%	0.06%	0.06%	93.3%	93.3%
50% Sampling rate					
Allocations (Total: 43,001,311)	27.9%	27.9%	27.9%	13.9%	2.3%
Memory	11.2%	44.6%	14.9%	7.4%	21.9%
Avg. Lifetime	0.05%	0.05%	0.05%	89.9%	89.9%
1% Sampling rate					
Allocations (Total: 860,026)	27.9%	27.9%	27.9%	13.9%	2.3%
Memory	13%	52.3%	17.4%	8.7%	8.5%
Avg. Lifetime	0.04%	0.04%	0.04%	86.3%	86.3%
0.1% Sampling rate					
Allocations (Total: 86,003)	27.9%	27.9%	27.9%	13.9 %	2.3%
Memory	8.5%	34%	11.3%	5.7%	40.4%
Avg. Lifetime	0.04%	0.04%	0.04%	87.1%	87%
		short-lived		**long-lived**	

RQ.2 Conclusion. We observe consistent lifetime frequencies across the various sampling rates, including the lowest one. This consistency holds even for the most-allocated classes, where we find no significant changes. This evidence suggests that our profiler reports **precise measurements relative to the benchmark execution time**.
We observed that results vary depending on the sampling rate and the profiling application. If the application makes a considerable number of allocations, we recommend using a small sampling rate and gradually incrementing it if needed. We obtained weakly actionable results using a 0.1% sampling rate.

5.3 RQ.3 - Execution time overhead

This section studies the overhead introduced by the profiler in the benchmark execution in two scenarios: with and without ephemeron pretenuring. Our profiler introduces overhead because we instrument all allocator methods and allocate an ephemeron object for each produced allocation.

Table 5 displays the introduced execution time overhead per application across different sampling rates. For each application, the table presents three rows: the measurements without pretenuring, the measurements with pretenuring and the estimated improvement of pretenuring over not doing it, computed as:

Table 5. Profiler overhead comparison with different sampling rates.

Sampling rate	**100%**	**50%**	**1%**	**0.1%**
DataFrame - pretenuring	5.8 × ±0.017	3.8 × ±0.019	2.25 × ±0.019	2.27 × ±0.012
DataFrame + pretenuring	2.29 × ±0.03	1.64 × ±0.01	1.08 × ±0.001	1.12 × ±0.001
ΔPretenuring	2.53×	2.32×	2.08×	2.03×
HoneyGinger - pretenuring	9.2 × ±0.01	8.91 × ±0.03	8.4 × ±0.6	7.8 × ±0.04
HoneyGinger + pretenuring	3.2 × ±0.10	2.5 × ±0.10	2.2 × ±0.06	1.8 × ±0.05
ΔPretenuring	2.88×	3.56×	3.82×	4.33×
Bloc - pretenuring	1.04 × ±0.002	1.04 × ±0.001	1.04 × ±0.0008	1.04 × ±0.0006
Bloc + pretenuring	1.03 × ±0.001	1.03 × ±0.001	1.03 × ±0.0007	1.03 × ±0.0007
ΔPretenuring	1.01×	1.01×	1.01×	1.01×
Moose - pretenuring	3.2 × ±0.02	2.6 × ±0.008	2.02 × ±0.007	2.00 × ±0.005
Moose + pretenuring	2.02 × ±0.03	1.6 × ±0.007	1.1 × ±0.003	1.09 × ±0.002
ΔPretenuring	1.58×	1.63×	1.84×	1.84×
Re:Mobidyc - pretenuring	32.3 × ±2.2	13.8 × ±0.11	9.14 × ±0.13	9.01 × ±0.08
Re:Mobidyc + pretenuring	2.06 × ±0.005	1.5 × ±0.004	1.3 × ±0.01	1.25 × ±0.05
ΔPretenuring	15.68×	9.20×	7.03×	7.21×
Avg. + pretenuring	2.12×	1.65×	1.34×	1.26×

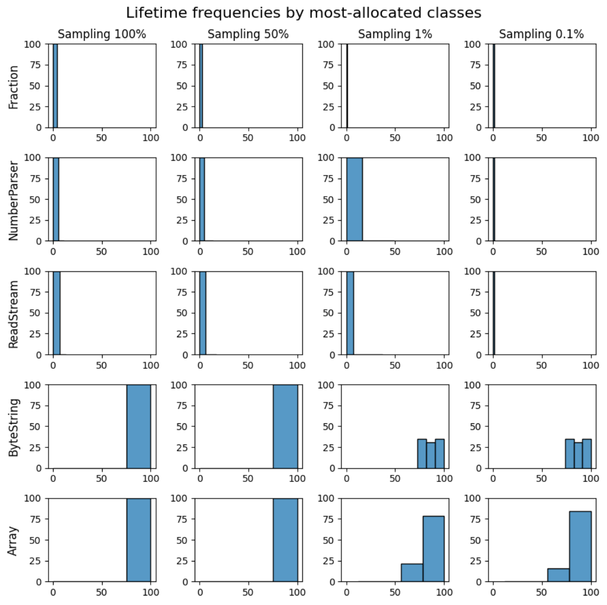

Figure 5. Relative lifetime frequencies by most-allocated classes. The x-axis represents the relative execution time of the application and the y-axis is the percentage of objects.

$$\Delta Pretenuring = \frac{Mean \quad without \quad pretenuring}{Mean \quad with \quad pretenuring}$$

The table shows that both the sampling rate and the allocation ephemeron incur significant overhead on benchmark execution, except for the Bloc benchmark arguably because it is less allocation intensive. Our results show that pretenuring ephemerons has a 3.68x improvement against not pretenuring on average. Moreover, ephemeron pretenuring shows an improvement of 15.68× the Mobidyc benchmark when performing 100% of sampling.

Overall, the profiler using the pretenured ephemeron configuration shows an average overhead of 1.59x across all sampling rates, diminishing with the sampling rate.

As we have shown that not pre-tenuring of objects significantly increases the overhead introduced by the profiler, we have chosen not to further investigate the impact of pretenuring on the computed overall lifetimes.

RQ.3 Conclusion. The impact of the overhead introduced by the profiler varies depending on the sampling rate and the ephemeron pretenuring configuration. Pretenuring ephemerons reduces the overhead by 3.68x on average. The pretenured configuration presents an average overhead of 1.59x across all sampling rates. This makes finalization profiling a **practical alternative** to estimating object lifetimes.

5.4 RQ.4 - Memory overhead

This section studies the memory overhead introduced by the profiler in the benchmark execution. Our profiler introduces memory overhead because one ephemeron is allocated by each application allocation.

Table 6 presents the baseline memory consumption of each benchmark when the profiler is not present and the memory overhead relative to the baseline of each sampling configuration. In this table, each column is each of our benchmarks. The rows show respectively the baseline absolute measurements and the overhead relative to the baseline at the different samplings. We computed the estimated memory overhead as follows:

$$Overhead_{sampling} = Baseline + size_e * allocations_{sampling}$$

where *Baseline* represents the absolute memory consumption, $size_e$ represents the size of a FiLiP ephemeron instance, and $allocations_{sampling}$ represents the number of allocations for a given sampling rate.

The table shows that sampling significantly impacts memory overhead, with a linear tendency for all applications. In the worst scenario using 100% of sampling, overheads range between 2.36 × and 3.34 ×, suggesting that FiLiP ephemeron objects are between 36% and 134% bigger than the average objects in each benchmark. This shows that there is a potential optimization on the size of ephemerons that could both reduce memory overhead and improve locality.

RQ.4 Conclusion. Our results show that low sampling rates have negligible memory overhead. In the worst case, we have an overhead of up to 3.34 ×, and 2.75 × in average.

6 Discussion and threats to validity

6.1 Benchmark selection and homogeneous workloads

In our experiments, we selected 5 different benchmarks for our evaluation. We did our best to choose benchmarks with different memory allocation profiles. However, benchmarks like DataFrame are designed to work with a homogeneous workload: a large CSV file with the same structure and similar data in each row. This paper does not analyze the impact of workload homogeneity on our results. It is left to future work to guarantee that the results are generalizable to all kinds of applications.

6.2 Strong vs. weak actionability

Our results in RQ. 1 show that acting on object lifetimes allows us to observe an increase in the overall average lifetimes as expected. However, as discussed previously, our profiler exhibits weak actionability: object graphs have an anchoring effect that makes it difficult to precisely predict the outcome of our actions on lifetimes. Our experiments show such weak actionability property is possible but more work is required to guarantee strong actionability.

7 Related work

Weak references to profile object lifetimes. Agesen et al. [1] investigated profiling object lifetimes in the Java programming language using weak references and various sampling techniques. One sampling technique involves attaching a weak reference to every X-th allocated instance, for example once every 1000th allocation. Other discussed sampling techniques include sampling objects causing local allocation buffer (LAB) refills, custom allocator routines, and thread-specific sampling. Weak references are attached to object allocations, and upon garbage collection, the finalization time of objects is recorded. To store information about the object, such as its type, memory size, and allocation time, the authors extended the `PhantomReference` class, ensuring that the object won't be revived at finalization. To the best of our knowledge, the work by Agesen et al. is the most relevant to ours. We attach an ephemeron to an object allocation instead of relying on weak references. In Pharo, ephemerons are not a class but a memory layout, allowing us to customize ephemerons to *e.g.,* make the ephemeron its finalizer. We also ensure that the object won't be revived during finalization. Although relevant, Agesen et al.'s work is descriptive but misses a report on their experiments and validation. Our paper provides an analysis of the impact of sampling and analyses the effect of ephemeron contamination and a validation methodology.

Pearce et al. [22] use weak references to profile object lifetimes for the AspectJ programming language. Similarly to Agesen et al., they attach a weak reference instance when an object allocation is produced to record the finalization time of the object.

Object lifetime profiling. Hertz et al. [12, 13] introduced the Merlin algorithm, a perfect tracing algorithm that computes object lifetimes by reconstructing *exactly* when objects were last reachable. Implemented on a modified virtual machine, Merlin determines object lifetimes offline, after the application's execution, albeit with an overhead ranging between 70 and 300 times. In contrast, our approach is implemted on top an unmodified Pharo VM and exhibits overheads up to 2.29 times.

Bruno et al. [?] developed an object lifetime recorder (OLR) that calculates object lifetimes in terms of garbage collection generations. It incorporates an allocation recorder that tracks allocation sites and provides notifications when a collection is completed. Subsequently, it generates incremental heap dumps upon completion of a collection. After the execution, it analyzes the object graph, incorporating allocation sites from the heap dumps to determine which objects should belong to the same generation. Bruno et al. [?] introduced a lifetime profiler called ROLP (Runtime Object Lifetime Profiler) based on OLR. ROLP predicts the object lifetimes based on its allocation context, which comprises

Table 6. Baseline memory consumption and memory overhead.

	DataFrame	HoneyGinger	Bloc	Moose	Re:Mobidyc
Total allocated memory	5053.51 MB	3312.44 MB	9.86 MB	3081.95 MB	4906.64 MB
Memory overhead (100% sampling)	2.36 ×	3.34 ×	2.9 ×	2.61 ×	2.53 ×
Memory overhead (50% sampling)	1.68 ×	2.25 ×	2.0 ×	1.81 ×	1.75 ×
Memory overhead (1% sampling)	1.01 ×	1.03 ×	1.02 ×	1.02 ×	1.01 ×
Memory overhead (0.1% sampling)	1.001 ×	1.003 ×	1.002 ×	1.002 ×	1.001 ×

the allocation site plus the thread stack state. Upon allocation, objects are annotated in their header with the allocation context. Different from us, it relies on virtual machine modifications to obtain good scalability at runtime.

Evaluating profilers. Mytkowicz et al. [21] studied the issue of disagreement among commonly used Java profilers in identifying hot methods. The paper employs causality analysis to evaluate profiler correctness, revealing biases and observer effects contributing to incorrect profiles. It introduces the concept of actionable profilers and proposes causality analysis as a method to evaluate profiler accuracy without relying on a correct profiler.

Burchell et al. [6] conducted a study on the precision of Java profilers. They evaluated the precision, reliability, and overhead of six actively maintained Java profilers using deterministic benchmarks. The study evaluates six actively maintained profilers, finding them relatively reliable but highlighting application-specific variations and disagreements among profilers in identifying hot methods. This study builds upon the work of Mytkowicz et al. [21], conducted 13 years prior.

Our validation was inspired by these two works. We applied Mytkowicz's methodology to evaluate whether our profiler is actionable. We based our precision comparison by doing differential profiling across different sampling rates.

Hybrid finalization mechanism. Valloud [29] implemented a new finalization mechanism for the HPS Smalltalk VM due to performance issues with the existing mechanisms: Weak Arrays and Ephemerons. Weak arrays were deemed inefficient, while ephemerons introduced a significant memory overhead. To address these issues, the author developed a new finalization mechanism that combines elements from both approaches to improve performance. While our paper did not specifically examine the memory overhead caused by ephemerons, our experiments show that ephemerons add a noticeable overhead when used in stressful scenarios such as profiling. Moreover, even when allocating one ephemeron per sampled allocation, the computed object lifetimes remain actionable and the introduced overhead is manageable. It's important to note that the implementation of ephemerons in Pharo differs from that in HPS Smalltalk.

Allocation site optimizations. Clifford et al. [7] introduced an instrumentation technique called allocation mementos. Mementos are small objects placed in the new space alongside allocated objects. They are designed not to survive any garbage collection cycle. These mementos contain information such as the allocation site and whether the object was tenured or not. This information is used to apply dynamic memory optimizations, including pre-tenuring, pre-transitioning, and presizing. The authors proposed a new instrumentation technique for applying memory optimizations. In contrast, our approach focuses on providing an evaluation of a finalization-based object lifetime profiling that targets application developers and manual optimizations.

In our previous work [15], we introduced Illimani, a memory profiling framework designed to operate on the stock Pharo VM. This work explores a case study to tune garbage collector parameters from object lifetime information. In this paper, we use Illimani as the underlying infrastructure for developing FiLiP.

Most allocated objects. Shuf et al. [26] introduced a novel garbage collection algorithm based on object types. They introduced the concept of prolific and non-prolific types, categorizing them based on their allocation frequency. The prolific types are the types of objects that are most allocated, and the non-prolific ones are those rarely allocated. They delineated this categorization using a threshold of 1%. If the total allocated instances of a type are above 1%, that type is considered prolific. However, in our study, we encountered objects with 2.3% allocations that exhibit long lifetimes, thus contradicting the authors' proposed 1% threshold for classifying an object as prolific.

8 Conclusion

In this paper, we evaluated the actionability, precision, and overhead of finalization-based object lifetimes profilers. By attaching an ephemeron to allocated objects and recording their finalization times, we approximate computed object lifetimes without requiring modifications to the virtual machine. Our approach offers actionable insights into object lifetimes that can be used for memory performance optimization, such as pre-tenuring, and garbage collector tuning.

Our profiler is weakly actionable: it presents significant increases in the average lifetimes of targeted classes when taking action to extend their lifetimes even when considering low sampling rates. Furthermore, our results show that pretenuring ephemerons and sampling object allocations effectively reduce the overhead introduced by the profiler, particularly in allocation-intensive applications. The most impactful optimization is the pretenuring of the ephemerons with an overhead reduction of 3.68x on average.

We observed that results vary depending on the sampling rate and the profiling application. When profiling allocation-intensive applications, we recommend developers use an iterative methodology starting with a small sampling rate and gradually incrementing if needed.

References

[1] O. Agesen and A. Garthwaite. Efficient object sampling via weak references. *ACM SIGPLAN Notices*, 36(1):121–126, 2000.

[2] C. Béra, E. Miranda, and E. G. Boix. Lazy pointer update for low heap compaction pause times. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*, pages 15–27, 2019.

[3] 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, pages 533–557. Springer-Verlag, July 2011.

[4] S. M. Blackburn, M. Hertz, K. S. Mckinley, J. E. B. Moss, and T. Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1):2–es, 2007.

[5] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[6] H. Burchell, O. Larose, S. Kaleba, and S. Marr. Don't trust your profiler: An empirical study on the precision and accuracy of java profilers. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, pages 100–113, 2023.

[7] D. Clifford, H. Payer, M. Stanton, and B. L. Titzer. Memento mori: Dynamic allocation-site-based optimizations. *ACM SIGPLAN Notices*, 50(11):105–117, 2015.

[8] S. Ducasse, G. Rakic, S. Kaplar, Q. D. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, and M. Denker. *Pharo 9 by Example.* Book on Demand – Keepers of the lighthouse, 2022.

[9] M. Flatt, C. Derici, R. K. Dybvig, A. W. Keep, G. E. Massaccesi, S. Spall, S. Tobin-Hochstadt, and J. Zeppieri. Rebuilding racket on chez scheme (experience report). *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–15, 2019.

[10] A. Georges, D. Buytaert, and 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, pages 57–76, New York, NY, USA, 2007. Association for Computing Machinery.

[11] B. Hayes. Ephemerons: A new finalization mechanism. In *International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'97)*, 1997.

[12] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: How to cheat and not get caught. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 140–151, 2002.

[13] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):476–516, 2006.

[14] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes. A look at the design of lua. *Communications of the ACM*, 61(11):114–123, 2018.

[15] S. Jordan Montaño, N. Palumbo, G. Polito, S. Ducasse, and P. Tesone. Improving Performance Through Object Lifetime Profiling: the DataFrame Case. In *IWST 2023 - International Workshop on Smalltalk Technologies*, Lyon, France, Aug. 2023.

[16] S. Kaleba, C. Béra, and 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.

[17] P. Lengauer and 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*, pages 111–122, 2014.

[18] S. Marr. Rebench: Execute and document benchmarks reproducibly, aug 2018. Version 1.0.

[19] E. Miranda and C. Béra. A partial read barrier for efficient support of live object-oriented programming. In *International Symposium on Memory Management (ISMM '15)*, pages 93–104, Portland, United States, June 2015.

[20] E. Miranda, C. Béra, E. G. Boix, and 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)*, pages 57–66. ACM, 2018.

[21] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. *ACM Sigplan Notices*, 45(6):187–197, 2010.

[22] D. J. Pearce, M. Webster, R. Berry, and P. H. Kelly. Profiling with aspectj. *Software: Practice and Experience*, 37(7):747–777, 2007.

[23] G. Polito, P. Tesone, E. Miranda, and D. Simmons. Gildavm: a non-blocking i/o architecture for the cog vm. In *International Workshop on Smalltalk Technologies*, Cologne, Germany, Aug. 2019.

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

[25] L. Safina, O. Zaitsev, C. Ferlicot-Delbecque, and P. I. Sow. Pharo dataframe: Past, present, and future. In *International Workshop on Smalltalk Technologies IWST'23*, Lyon, France, Aug. 2023.

[26] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–306, 2002.

[27] K. Sivaramakrishnan, S. Dolan, L. White, S. Jaffer, T. Kelly, A. Sahoo, S. Parimala, A. Dhiman, and A. Madhavapeddy. Retrofitting parallelism onto ocaml. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020.

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

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

Received 2024-03-22; accepted 2024-05-10