ILLIMANI Memory Profiler -A Technical Report

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

ILLIMANI Memory Profiler - A Technical Report

Sebastian Jordan Montaño
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
sebastian.jordan@inria.fr

Guillermo Polito
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
guillermo.polito@inria.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
stephane.ducasse@inria.fr

Pablo Tesone
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
pablo.tesone@inria.fr

***Abstract*—Modern programming languages provide automatic memory management with an efficient garbage collector making the memory management of an application transparent to the developer. There is a need for practical tools to support developers in their understanding of the memory consumption of their applications. In this paper, we present a prototype version of ILLIMANI: a precise object allocation profiler. It has a rich object model that provides information about the objects' allocation context, the evolution of memory usage, and garbage collector stress.**

We were able to find an object allocation site in the class UITHEME that was making 99,9% redundant allocations. We developed a Color Palette cache at the domain level that removed all the redundant allocations. We were also able to identify 2 other object allocation sites in the methods MARGIN»#INSETRECTANGLE and NUMBER»#ASMARGIN.

***Index Terms*—Memory allocation, profiling, Pharo, allocation site**

I. INTRODUCTION

Modern programming languages, such as Java, Python, or Pharo, provide automatic memory management with an efficient garbage collector. This makes the memory management of an application transparent to the developer. Debugging memory issues is known for being a tedious activity [1]. There is a need for practical tools to support developers in their understanding of the memory consumption of their applications.

In this paper we present prototype version ILLIMANI[1] [2] : a precise object allocation profiler for the Pharo Smalltalk programming language [2]. It profiles the object allocations that are produced during the execution of an application. It provides information about the allocation context for each of the allocated objects, the evolution of memory usage, and garbage collector stress. It instruments the code to control the execution of the methods that are responsible for allocating objects using method wrappers. It also calculates the application's memory consumption.

[1]The version of the tool and the technical report are dated March 2023

We used ILLIMANI to profile the object allocations in the Morphic UI, a Pharo framework that is used to draw the Pharo IDE. We were able to detect object allocation sites. We found a color object allocation site in the class UITHEME. We analyzed the allocated objects and we discover that 99,9% of the allocated colors were redundant. We developed a Color Palette at the domain level introducing an important missing architectural element that serves as a natural cache. With the Color Palette, we reduced the memory stress of the application by removing all the redundant allocations. We were also able to identify 2 other object allocation sites in the methods MARGIN»#INSETRECTANGLE and NUMBER»#ASMARGIN.

Outline. Section II gives an insight of ILLIMANI and its features; Section III explains the functioning mechanisms of the profiler; Section IV presents a use case example where we profiled the object allocation during the execution of opening 30 Pharo tools and rendering each of them for 100 rendering cycles; Section V talks about the related work; and Section VI and Section VII finalize explaining the limitations, the conclusion, and the future work.

II. ILLIMANI MEMORY PROFILER

ILLIMANI[3] is a memory profiler developed for the Pharo Smalltalk [2] programming language that is available under an open-source MIT license. Pharo is a dynamically typed, purely object-oriented, and reflective modern programming language. ILLIMANI profiles and extracts relevant information of the profiled application, such as the objects' allocation context, memory usage, and garbage collector stress. It presents this information with memory usage tables, accumulative allocation evolution charts, and a heat map visualization. It is also possible to query the profiler to make a custom analysis. ILLIMANI is capable of filtering the profiling for a given specific domain. In this paper, we present a prototype version of ILLIMANI. The user can specify which types of objects she wants to capture or to capture the allocations that a given set

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

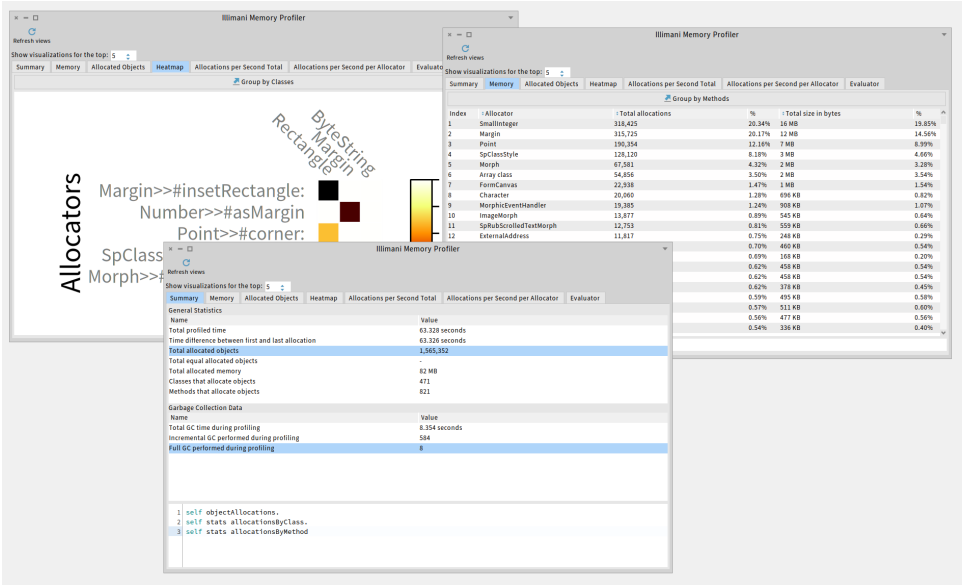

Fig. 1: Overview of ILLIMANI user interface

of classes produce. Figure 1 shows an overview of the user interface of ILLIMANI.

ILLIMANI offers the following features:

- Summary and statistics report
- Memory usage tables
- Navigation of query results
- Heat map visualization with the relationship: allocator - allocated
- Accumulative allocation evolution

A. *Summary and Statistics*

ILLIMANI provides a summary of the studied execution. It provides information on the total allocated objects, the allocator classes and methods, the memory usage, and the garbage collector stress.

ILLIMANI shows information on how many garbage collections were made, both incremental and full, and the time spent doing garbage collections. Pharo has a two-generation garbage collector [3]. It has a young and an old space. The newly allocated objects are allocated in the young space and after they survive a threshold of garbage collections they are moved to the old space. The garbage collections done in the old space are orders of magnitude slower than the ones done in the young space [4], [5].

The profiler groups the allocations by allocator classes or methods. It shows this information in memory tables that can be sorted by the number of allocations or by the total memory size in bytes.

Different executions have different allocation paths. ILLIMANI provides a chart with the allocation paths for the top allocators. The number of top allocators is a customizable parameter that can be changed by the user. On the one hand, Figure 2 shows that the class GRAFPORT, the second most allocator, allocates all of its objects in the first moment

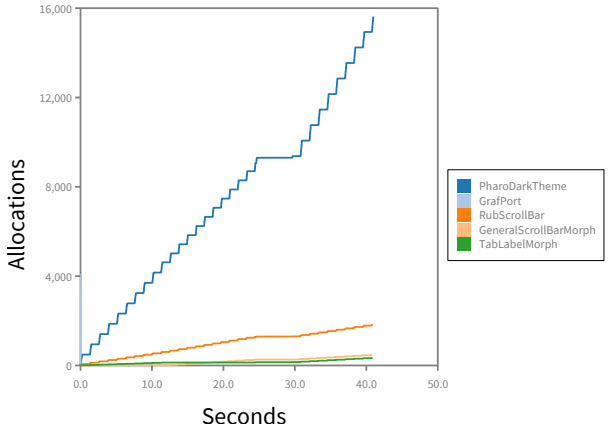

Fig. 2: The 5 top most allocator classes

and then stops allocating. On the other hand, the other four classes allocate the objects continuously during the execution. These different execution paths are identifiable thanks to the allocation path chart.

B. *Allocation queries*

ILLIMANI gives access to the raw information of the allocated objects such as the allocator class and method, the object's total size in memory, the object's allocation time, and its allocation context stack. Pharo supports full-stack

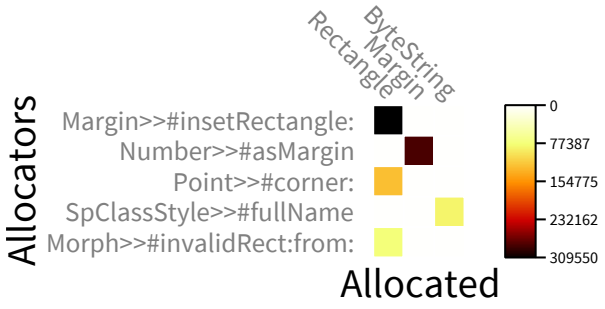

Fig. 3: Allocator methods heat map

reification thanks to its reflective properties. ILLIMANI uses this language property to copy the full execution stack of each one of the allocations. The user can query ILLIMANI to extract this information and to make a custom analysis. We developed custom data structures with constant time insertion and accessing to support the queries.

```
"Allocations bigger than 4 KB"
profiler objectAllocations select: [ :e |
    e totalSizeInBytes > 4096 ].
"Most allocator methods grouped allocations"
profiler allocationsByMethod
    first groupedAllocations.
```

Listing 1: Querying ILLIMANI

C. Heat map

ILLIMANI presents the information with a heat map. It shows the relationship between the most allocator classes, or methods, and the most allocated objects. Key questions developers ask about memory are related to who is responsible for most creating instances and of each class, or method [6]. Heat map visualizations are particularly adapted to display such relationships. Their matrix architecture supports the identification of key players: most created vs. most creating classes per entity [7]–[9].

The most allocators are ordered from top to bottom, the top is the one that allocates the most and the bottom one is the one that allocates the less. The allocated classes are ordered from right, the most allocated, to the left, the less allocated. ILLIMANI groups the allocations by classes and by methods. This heat map supports a drill-down version where methods creating most objects are displayed instead of the classes: Figure 3 shows that method MARGIN»#INSETRECTANGLE is the one creating all rectangle objects.

III. PRECISE MEMORY PROFILING

In Pharo, almost all computations are done by sending messages (invoking methods) [10]. Allocating an object is done also by sending a message. The methods BEHAVIOR»#BASICNEW, BEHAVIOR»#BASICNEW:, ARRAY CLASS»NEW:, and NUMBER»@ are the four methods that allocate objects in Pharo.

ILLIMANI automatically instruments the execution of the profiled application to capture its object allocations. It instruments the four allocator methods to control their execution before and after being invoked. We use the library *MethodProxies*[4] for instrumenting the methods. *MethodProxies* add defined actions to be executed and or after the method that is being instrumented.

Each time that one of those methods is invoked, ILLIMANI intercepts the call and registers important information about the allocation context, the object's type, and its size in memory. Right after the allocation is made, we store the allocated object's type, its allocator class and method, and different information about the allocation's context. The *MethodProxies* architecture ensures that the code will be de-instrumented after the profiling is finished. None of the allocations that are made in the process of extracting the information are intercepted.

IV. USE CASE: IDENTIFYING ALLOCATION SITES

A Pharo expert had a hint about a memory leak of objects of the type COLOR. ILLIMANI provides the possibility of filtering the profiling for a given specific domain. We configure the profiler to capture all the COLOR allocations that an application creates. We run the profiler on Pharo 11, commit `1a5afe1`.

Our target application was *MorphicUI*, a graphics framework for Pharo. It has 669 classes with 11236 methods in Pharo 11.

We opened 30 Pharo core tools and we let each of the instances of the tools render for 100 Morphic rendering cycles. Through this we are able to control how many times each of the tools is rendered, making the profiling reproducible. The tools are: Iceberg, Playground, and the Pharo Inspector. We opened 10 of each making 30 in total. The code to reproduce the experiment is available as a script[5].

Figure 2 presents an allocation paths plot for the top 5 allocator classes. One can observe that the class PHARODARKTHEME is the allocation site with the most allocations. PHARODARKTHEME is a subclass of UITHEME. UITHEME is a central class in Pharo that is responsible for setting drawing configurations and also to provide the theme colors that are used in the Pharo IDE.

During the application's execution, we observed 23,686 total COLOR object allocations. Table I shows that the PHARODARKTHEME class is responsible for 66% of all the COLOR allocations. Using the customizable queries of ILLIMANI we analyzed the allocated objects by the UI Theme and we detected that only 15 out of 23,686 colors were different, meaning that 99,9% of the allocations were redundant.

Summary: Using ILLIMANI we identified an allocation site in the class PHARODARKTHEME that was allocating 66% of all the colors with 99,9% redundant allocations.

[4]https://github.com/pharo-contributions/MethodProxies
[5]https://gist.github.com/jordanmontt/05c51c5527bf1c8e375117a3b9020c1e

TABLE I: Top 5 color allocations when opening 30 Pharo tools

Allocator class	Allocated colors	%
PharoDarkTheme	15,629	66%
GrafPort	4,096	17%
RubScrollBar	1,842	8%
GeneralScrollBarMorph	480	2%
TabLabelMorph	346	1%
Rest of the classes	1293	2%

A. Color Palette

Looking at the implementation of UITHEME we identified the cause of the redundant allocations: each time that a user asks for a color, the UITHEME class creates a new instance of it. We developed a Color Palette as a solution. The Color Palette lazily allocates the colors when they are requested by a user and caches them once created. When the UITHEME changes, for example changing from a dark theme to a light theme, the caches are invalidated and they are recalculated on demand. The Color Palette was integrated into Pharo 11 in the commit `d540bcf`.

With the Color Palette fix, we profiled again the same execution to compare the baseline implementation against the Color Palette. In Table II and in Figure 4 we see that with the Color Palette implementation, the PHARODARKTHEME does no longer allocate COLOR objects.

TABLE II: Object Allocations in Baseline vs Color Palette implementation

Allocator class	Baseline	Color Palette	Diff
PharoDarkTheme	15,629	0	∞
RubScrollBar	4,096	4,096	1×
Total Allocations	23,686	7,974	3×

Summary: With the Color Palette implementation the PHARODARKTHEME class does not longer allocates redundant colors.

B. Other allocation sites

We profiled the same execution setup, opening 30 Pharo tools, this time not filtering the allocated objects but capturing all of the allocations. We observe in Table III that the classes RECTANGLE and MARGIN are the ones that are allocated the most, summing 64% of all the allocations between them. The methods MARGIN»#INSETRECTANGLE and NUMBER»#ASMARGIN are the two methods producing all of those allocations. We did not investigate further the causes of these object allocation sites. The heat map presented in Figure 3 shows the relationship between the most allocators and the most allocated for this profile. One observes that the method MARGIN»#INSETRECTANGLE allocates mostly RECTANGLE objects while NUMBER»#ASMARGIN allocates MARGIN objects.

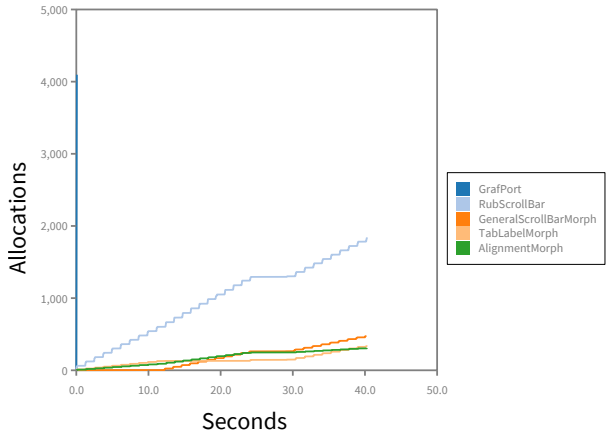

Fig. 4: The 5 topmost color allocator classes with the Color Palette implementation

TABLE III: Allocated Objects

Allocated object class	Allocations	%
Rectangle	699,625	45%
Margin	300,663	19%
ByteString	111,662	7%
OrderedCollection	78,474	5%
WriteStream	60,448	4%
Rest of the classes	314,480	20%

Summary: ILLIMANI reports other two object allocation sites that allocate objects of type RECTANGLE and MARGIN that represent 64% of the allocations.

V. RELATED WORK

Visualizing object allocation sites. Infante et al. [12] have developed a Pharo memory profiler that reports object production sites and memory usage called Memory Blueprint. Memory Blueprint shows a call graph with the methods that produce objects and a visualization of the memory usage of the allocated objects. Their work focuses on identifying object production sites while ILLIMANI also has a rich model for representing the object allocations that allows the user to query them to calculate other statistics. It can be also queried without the user interface as it is independent of it and it allows inspecting the reified stacks of the allocation contexts. Fernandez et al. [13] made a visualization related to Memory Blueprint, but that keeps the context of the method and the source code of it. ILLIMANI also provides for each of the allocated objects the allocator method and class as well as the allocation context. Fernandez et al. [14] have developed

a call graph visualization, similar to their previous work but analyzing the memory consumption of Python applications.

Allocation matrix. De Pauw et al. [9] developed an Allocation matrix, which is a visualization that shows the relationship between the most allocators and the most allocated for the C++ programming language. We used this idea to implement our heat map visualization.

Resources consumption reduction. Bergel et al. [15] have worked on analyzing the memory footprint of the expandable collections of Pharo. They analyzed the allocation of internal arrays and their use to propose an optimized version of the expandable collections. Our work is different but complementary in the sense that we profiled the memory allocation of Pharo tools to identify allocation sites to reduce its memory footprint.

Domain specific profilers. Ressia et al. [16] developed a domain-specific profiler framework in Pharo that the user can specify the profiled domain to allow the profiler to give more precise information about the specific domain. They developed this technique to profile a know performance issue that they had and the profiling tools that were available at that moment were not sufficient to localize the problem. The profiler that we developed allowed us to identify a problem that we were not aware of. Thanks to its architecture, ILLIMANI also allows filtering the profiling for a given specific domain. The user can specify which set of classes wants to profile and which type of object allocations she wants to capture.

Tracking of Allocation Sites. Odaira et al. [17] have working on tracking object allocation sites in Java. They developed two new approaches to trace the object allocation sites with a minimum slowdown of the execution. They store the allocation information on the object hash code field or in the object header. They use the allocation information to optimize the garbage collector copying directly into a tenured space certain objects. Our work does not focus on the tracing technique, we use method wrappers. Our solution does not require running the profiler on a modified virtual machine and it has a rich model that offers the possibility of extracting information from the allocation context.

VI. LIMITATIONS

In Pharo, there are special methods, called primitives that are executed natively. At the moment of writing this paper, we were not able to wrap these special methods. In Pharo 11 we identified in total 3 primitive methods that allocate objects of classes ARRAY, POINT and INTERVAL.

Our profiler has an overhead of around 10×. However, this not affects the precession of the measurements as the application will make the same number of allocations.

ILLIMANI does 3 object allocations that are registered during the profile. The number does not vary no matter the code that is being profiled. This is not a significant perturbation for our profiles.

VII. CONCLUSION AND FUTURE WORK

ILLIMANI is a memory profiler that can precisely capture object allocations. It provides a rich object model that allows a user to query and group the object allocations. ILLIMANI also shows the information with tables and visualizations. We profiled the execution of opening 30 Pharo tools: Iceberg, Playground, and the Inspector. We opened 10 times each making 30 in total and we let render each of the applications for 100 Morphic cycles. We were able to detect object allocation sites. ILLIMANI reported that the class UITHEME was making 99,9% redundant object allocations. We were able to fix the excessive allocations by introducing the architectural concept of a Color Palette removing all the redundant allocations that the UITHEME was making.

We were able to identify other two allocation sites related to the *MorphicUI* framework. For the same execution of opening the 30 Pharo tools, 1,565,352 objects were allocated in total and the classes RECTANGLE and MARGIN are the ones that are allocated the most, summing 64% of all the allocations. The methods MARGIN»#INSETRECTANGLE and NUMBER»#ASMARGIN are the two methods producing all of those allocations.

There is previous work done on energy profiling in Pharo [18]. We want to continue this work to include energy measurements along with the object allocations. We will also develop a solution at the bytecode level to be able to wrap the primitive methods too.

REFERENCES

[1] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy, "Patterns of memory inefficiency," in *ECOOP 2011–Object-Oriented Programming: 25th European Conference, Lancaster, Uk, July 25-29, 2011 Proceedings 25*. Springer, 2011, pp. 383–407.

[2] 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. [Online]. Available: http://books.pharo.org

[3] D. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *ACM SIGPLAN Notices*, vol. 19, no. 5, pp. 157–167, 1984.

[4] 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.

[5] 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)*. ACM, 2018, pp. 57–66.

[6] J. Sillito, K. D. Volder, B. Fisher, and G. Murphy, "Managing software change tasks: An exploratory study," in *Proceedings of the International Symposium on Empirical Software Engineering*. IEEE Computer Society, 2005, pp. 23–32.

[7] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Revised Lectures on Software Visualization, International Seminar*. London, UK: Springer-Verlag, 2002, pp. 151–162.

[8] W. De Pauw and G. Sevitsky, "Visualizing reference patterns for solving memory leaks in Java," *Concurrency: Practice and Experience*, vol. 12, no. 14, pp. 1431–1454, 2000.

[9] W. De Pauw, D. Kimelman, and J. Vlissides, "Modeling object-oriented program execution," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94)*, ser. LNCS, M. Tokoro and R. Pareschi, Eds., vol. 821. Bologna, Italy: Springer-Verlag, Jul. 1994, pp. 163–182.

[10] 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)*, ser. LNCS. Springer-Verlag, Jul. 2011, pp. 533–557. [Online]. Available: http://bergel.eu/download/papers/Berg11c-compteur.pdf

[11] J. Brant, B. Foote, R. Johnson, and D. Roberts, "Wrappers to the rescue," in *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, ser. LNCS, vol. 1445. Springer-Verlag, 1998, pp. 396–417.

[12] A. Infante and A. Bergel, "Efficiently identifying object production sites," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 575–579. [Online]. Available: https://bergel.eu/MyPapers/Infa15a-MemoryProfiling.pdf

[13] A. Fernandez Blanco, J. P. Sandoval Alcocer, and A. Bergel, "Effective visualization of object allocation sites," in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, 2018, pp. 43–53.

[14] A. Fernandez Blanco, A. Bergel, J. P. Sandoval Alcocer, and A. Queirolo Córdova, "Visualizing memory consumption with vismep," in *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022, pp. 108–118.

[15] A. Bergel, A. Infante, S. Maass, and J. P. S. Alcocer, "Reducing resource consumption of expandable collections: The pharo case," *Science of Computer Programming*, vol. 161, pp. 34–56, 2018, advances in Dynamic Languages. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642317302940

[16] J. Ressia, A. Bergel, O. Nierstrasz, and L. Renggli, "Modeling domain-specific profilers," *Journal of Object Technology*, vol. 11, no. 1, pp. 1–21, Apr. 2012.

[17] R. Odaira, K. Ogata, K. Kawachiya, T. Onodera, and T. Nakatani, "Efficient runtime tracking of allocation sites in java," *ACM Sigplan Notices*, vol. 45, no. 7, pp. 109–120, 2010.

[18] A. Bergel, "Power and energy code profiling in pharo," in *Proceedings of the International Workshop on Smalltalk Technologies (IWST'16)*, 2016.