Automatic Collection Selection using Machine Learning

Couderc, Noric

2022

Automatic Collection Selection using Machine Learning

Noric Couderc

Licentiate thesis, 2022

Department of Computer Science
Lund University

ISBN 978-91-8039-193-1 (electronic version)
ISBN 978-91-8039-194-8 (print version)
ISSN 1404-1200
Technical Report 110, 2022
Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: `noric.couderc@cs.lth.se`

Typeset using LaTeX
Printed in Sweden by Tryckeriet i E-huset, Lund, 2022

Abstract

Most recent programming languages include a collection framework as part of their standard library (or runtime). Examples are Java, C#, Python and Ruby. The Java Collection Framework provides a number of collection classes, some of which implement the same abstract data type, which makes them interchangeable. Developers can therefore choose between several functionally equivalent options. Since collections have different performance characteristics, and may be allocated in thousands of programs locations, the choice of collection has an important impact on performance. Unfortunately, programmers often make sub-optimal choices when picking their collections. In this licentiate thesis, we consider the problem of building automated tooling, which would help the developer choose among several collection implementations. We consider an existing tool called Brainy, which targets C++, and adapt it to the Java context. In doing so, we investigate how to synthesize benchmarks and analyze their behavior to create training data for automated classification. We propose one new generative model for collection benchmarks and present the challenges that porting JBrainy to Java entails. Finally, we compare JBrainy's suggestions versus greedy search, on five well known benchmarks. Our investigations show that JBrainy's suggestions were almost as effective than those of greedy search in minimizing the running time of programs. However, we also find that Brainy's benchmark synthesis methods do not apply well to the Java context, since they introduce some significant biases.

This thesis is a compilation consisting of an introduction, one paper, and a technical report.

Contributions of the author

List of included peer-reviewed publications by the thesis author.

Paper I Noric Couderc, Emma Söderberg, and Christoph Reichenbach. "JBrainy: Micro-benchmarking Java Collections with Interference". In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 42–45

The thesis author did all of the technical work and is the main author of sections 3-4.

Other publications

Paper II Noric Couderc, Christoph Reichenbach, and Emma Söderberg. *JBrainy, Effective Selection of Data-Structures in Java*. Tech. rep. Lund Tekniska Högskola, 2022

The thesis author is the main author of the paper, and did all the technical work regarding evaluation.

Contents

INTRODUCTION

1 The collection selection problem

Seen from very far, all software is the same: it reads data, processes it, and stores the processed data. This similarity can be misleading: even two programs that compute exactly the same result can be different. For example, there exists several ways to sort a list of numbers. How can two programs be different if they compute the same thing? The difference lies in *how* they compute the result: which operations are done, and in which order. Both these things have an impact on how much time and resources the process requires. Since computers have limited resources, and humans have limited time, time and resources are important aspects of software quality [Noa].

One way to make processing faster and storage more efficient is to look at how a piece of software stores data in memory. When software developers have to group data items together (for example, making a list of all the cities the system knows), they typically use what is called a *collection*. Many standard libraries for programming languages provide different collections implementations, each with different performance characteristics.

Unfortunately, developers often make sub-optimal choices when choosing collections, leading to both memory bloat (using too much memory), and runtime bloat (taking longer than optimal to finish a task). The first problem developers face is their limited knowledge on how data-structures work. In their review of software bloat, Mitchell et al. [MSS10] report that HashMaps and HashSets, two commonly used Java collections, take more space than is truly needed: they cache hashcodes, even if these are not expensive to compute. Another problem Mitchell et al. highlight is that framework implementers do not know how their software will be used: they argue it is difficult to make performance design decisions without knowing the context [MSS10]. To make things worse, even very small changes have a big impact: In one study [LR09], Google engineers noticed that changing a single line of code improved the running time of their program by 17%.

Choosing the *wrong* collection doesn't lead to obvious failure. Two collections from the same family (list, sets, maps) have roughly the same functional properties, swap one for the other, and it works anyway. This flexibility is appreciated by developers, but has an impact on performance [MSS10].

Why would choosing the right collection be the task of the developer? Compilers already automate many optimizations [WO18], why not include collection selection too? In this thesis, we will assume that we can indeed help developers choose the right collections, and explore techniques to do that.

2 A map of the territory

If we aim to have a tool look at the collections that a program uses and suggest a better configuration, there are several design choices we need to consider. In this section, we will review what design choices were made in existing systems. I chose to compare Chameleon [SVY09], Brainy [Jun+11], CoCo [Xu13], CollectionSwitch [CA18], and finally, Artemis [Bas+18]. Automated collection selection dates back to 1983 with the work of Freudenberger et al. on the SETL language [FSS83]: I chose to focus on systems which target Java (Brainy being the exception, for reasons which will become clear soon) and aim to improve running time.

There are systems which target other languages and use other definitions of efficiency, like energy usage. Perflint [LR09] improves the running time of C++ programs. On energy usage, the SEEDS system [MPC14] aims at improving energy usage, while Hasan et al. [Has+16] investigated why some collections require more energy than others.

Coming back to our comparison, I will focus on how different systems answer the following design questions.

- What language do we target?
- What are we trying to improve?
- What collections and optimizations do we consider?
- How do we perform replacements?

2.1 What language do we target?

Chameleon, CoCo, and CollectionSwitch target Java. Brainy targets C++, while Artemis supports both C++ and Java. The choice of target language is mostly an engineering choice, but as we will see, porting Brainy from C++ to Java introduces challenges.

2.2 What are we trying to improve?

All of these works claim to improve *performance*. What definition of performance do they use? Here, performance is a trade-off between two resources, which should not be wasted: running time, and memory used. Artemis and Chameleon try to optimize both simultaneously, while CollectionSwitch allows the user to choose between optimizing one, or the other. CoCo, on the other hand, trades memory usage with better running times (by sharing the data across several collections). Lastly, Brainy only optimizes running time.

2.3 What collections and optimizations do we consider?

A major design decision revolves around what changes the tool is allowed to do. What collections can it choose between? What are the changes the tool is allowed to make on the program to optimize?

Collection Tuning

Several collections (e.g. ArrayLists and HashMaps) have parameters that do not change their functional behavior, but change their performance. ArrayList takes a `capacity` parameter, which is the initial size of the underlying array. For HashMaps, the `load factor` specifies when the map's array is full enough to justify a full copy of it. These parameters do not matter that much when the collection becomes big, but in programs with many, small collections, constants matter [SVY09]. Both Artemis and Chameleon use collection tuning to improve performance. Brainy, CoCo, and CollectionSwitch do not.

Lazy Collections

Sometimes, a collection might remain empty. Even when that is the case, Hashsets and ArrayLists still allocate an array, which is then wasted space [MSS10]. In the case of the bloat DaCapo benchmark, Shacham et al [SVY09] were able to gain 20% memory usage but switching a LinkedList to a LazyLinkedList, which would only create a first node if one element was added to the list.

Array-based Maps

When Maps are small, it is actually faster and more memory-effective to look for the key-value pair linearly than to use a regular hash map. This technique is common among the tools we considered, since Chameleon, CoCo, and CollectionSwitch all use this type of Map. CollectionSwitch's authors implemented "Adaptive" maps and sets, which automatically switch between the "linear-search" mode, and the "hash-based mode". These collections are popular among CollectionSwitch's suggestions. Chameleon presented the same data-type (called SizeAdaptingSets and SizeAdaptingMaps). In the case of Chameleon, using ArrayMaps was important in getting 13.79% of reduction in memory space used by the program FindBugs.

Hash-based Lists

Converse to using lists of pairs to implement maps is the method of coupling an ArrayList and a HashSet in the same object, with the hope that it will speed up calls to the `contains` method. Both CollectionSwitch and CoCo use this method. CoCo's authors call their implementation an HashArrayList, while CollectionSwitch's AdaptiveList uses this technique and is also reported as an option often favored by the CollectionSwitch tool.

A word about consistency

We have listed some of the common techniques these tools use. However, I noted that the set of collections differs often between systems. CollectionSwitch uses

OpenHashSets (sets which use a map with open addressing) which weren't used in Chameleon, and CoCo. Likewise, Artemis uses synchronized collections which were not used in Chameleon, CoCo, and CollectionSwitch. Moreover, lazy collections, array-based maps, and hash-based lists are not popular among Java programmers [CASL17]. An interesting line of work would be to evaluate these methods on the same set of collections, to see how much the decision-making matters, compared to the features of each collection.

2.4 How do we perform replacements?

Developers usually work on an existing code base. The existing program likely already uses collections. The developer therefore faces the two following questions:

1. Which collections should we replace first?
2. When should we make replacements?
3. What replacements should we use?

Which collections should we replace first?

The question of choosing the most important collections in the program of interest can either be considered a problem that the tool should solve, or developer's responsibility.

Artemis and Chameleon take the most aggressive approach: every allocation site can be optimized. CollectionSwitch optimized allocation sites from which at least 1000 instances originated. CoCo's authors do not state how many allocation sites were switched to CoCo collections. Lastly, Brainy's authors manually chose one allocation site for each benchmark they considered.

When should we make replacements?

I could find two families of replacements: static replacements (before running the program) and dynamic replacements (while the program is running). Both methods have their advantages and drawbacks, and which method is best is still an open question.

One aspect that makes static replacements attractive is the lack of run-time overhead. Just like developers, tools that perform static replacements can think for as long as they need [1]. They can also use features which would be very expensive to get at run-time. As examples, Brainy gathers hardware performance counters and evaluates a neural network to decide which collection to use. Similarly, Artemis uses a multi-objective genetic search algorithm (NSGA-II), which evaluates modified variants of the program, taking 3.05 hours on average [Bas+18].

[1] managers might disagree with this statement

Chameleon, on the other hand, uses handwritten rules, which could be cheap, however it also uses expensive information about the state of the heap. Dynamic replacement tools cannot afford such expensive decision making.

Now, static replacements can suffer from their lack of adaptability. Static replacement tools can only perform allocation-site level replacements, which might be too coarse. For example, if the constructor is called in a loop, all collections will have the same type. Moreover, static replacements cannot take into account the program inputs, nor can they adapt naturally to code changes: If the developer changes their code, they have to re-run the tool.

How do we decide what collection to use?

Most of the tools use a hand-crafted performance model to select which collection to use. Chameleon uses hand-written heuristics to make a choice. For example, the rule:

```
ArrayList: #contains > X and maxSize > Y -> LinkedHashSet
```

Specifies that an ArrayList should be switched to a LinkedHashSet if the number of calls to contains and the maximum size of the instance exceed some fixed thresholds X and Y. Finding the thresholds X and Y is the user's task. CoCo uses a similar strategy, using hand-written rules, except these are evaluated at run-time. Perflint uses a similar model, mixing rules about asymptotic complexity, and considerations of CPU architecture.

Brainy and CollectionSwitch use a different approach: They try to learn this model with micro-benchmarking. Brainy being an offline tool, it can be more ambitious in this regard, and uses neural networks to predict the best collection to use, given hardware performance counters (like branch mispredictions and L1 cache miss rates). CollectionSwitch must make decisions at run-time, and must therefore use a simpler model. It evaluates the cost of an operation as a polynomial of the size of the collection at the moment. The parameters of the polynomial are learned offline, on hand-written micro-benchmarks.

Lastly, Artemis takes the minimalist-maximalist approach: It does not use a model at all. Instead, it evaluates variants of the program of interest in a multi-objective optimization algorithm: NSGA-II. The fitness function evaluates the test suite of the program, and measures how long it runs.

One may argue that using the test-suite of the program for optimization is not representative of the performance characteristics of the program. This issue extends to tools which use machine learning to deduce their performance model: What training data did they use? To some extent, the rules that Chameleon and CoCo use are also up to debate, where do they come from?

Brainy provides an interesting solution to the problem: They needed a lot of training data quickly, so they generated programs to serve as micro-benchmarks. How representative are these benchmarks? This is one of the main questions we will look at in this thesis.

2.5 Cutting corners: which one is the best?

A nice property of performance engineering is its clear definition of success: we want faster, less memory-intensive programs. How do these tools compare, in terms of numbers?

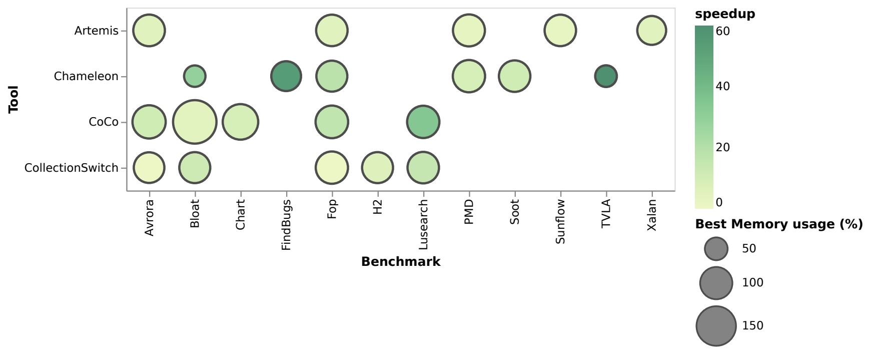


Figure 1: Comparison of speedups and memory usage for each tool and benchmark

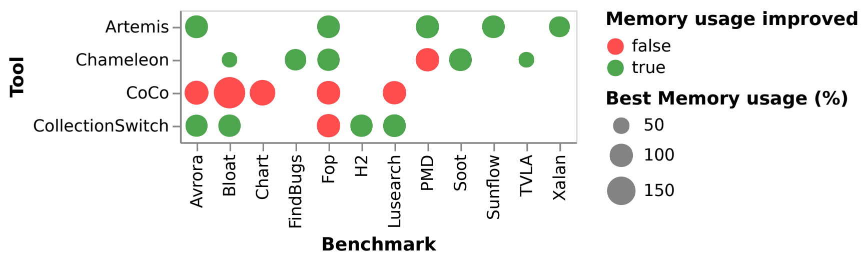


Figure 2: Comparison of memory usage for each tool and benchmark

Figure 1 shows the speedup and memory usage each tool gives, for each benchmark. Color represents the speedup (greener is better), and the size of the circle represents the memory usage. Greener, smaller circles are better. Figure 2 shows how much memory each benchmark uses, after optimization with each tool. When memory usage was lower than the original, the circle is green. Otherwise, it is red.

We can see that different tools were often tested on different benchmarks, as there are several “holes” in the plot. Only fop was tested with every tool. Chameleon was the most effective at reducing running time, while Artemis yielded the most modest improvements. Lusearch was the benchmark that was improved the most with CoCo and CollectionSwitch, but unfortunately, Chameleon was not tested on lusearch.

In terms of memory consumption, CoCo *increased* it, but that was the goal: trading memory consumption for speed. Once again, the most effective tool was Chameleon. Comparing the memory usage of programs is a bit difficult on the plot, but for good reason: apart from Chameleon, the improvements of other tools remain below 10%.

For completeness, we provide the data in tabular form.

Benchmark	Tool	Best running time (%)	Best memory usage (%)
Sunflow	Artemis	98	95
PMD	Artemis	97	98
Fop	Artemis	95	95
Avrora	Artemis	95	96
Xalan	Artemis	95	80
PMD	Chameleon	91	100
Soot	Chameleon	89	95
Fop	Chameleon	82	92
Bloat	Chameleon	71	44
FindBugs	Chameleon	46	85
TVLA	Chameleon	40	45
Bloat	CoCo	96	181
Chart	CoCo	91	121
Avrora	CoCo	89	106
Fop	CoCo	84	102
Lusearch	CoCo	66	100
Avrora	CollectionSwitch	100	90
Fop	CollectionSwitch	100	100
H2	CollectionSwitch	94	92
Bloat	CollectionSwitch	88	92
Lusearch	CollectionSwitch	85	95

If we compare with Brainy, we see that it found substantial improvements, getting up to an impressive 77% speedup. Brainy's authors highlight the importance of the CPU architecture, as their tool suggests different options, depending on what architecture the software is running on. This aspect is neglected by the works on Java. Artemis, in particular, optimizes programs by running them in the cloud. Could the importance of the CPU architecture explain the somewhat modest improvements provided by Artemis?

Benchmark	Tool	Best running time (%)	Architecture
Xalanbmk	Brainy	79	Core2
Chord Similator	Brainy	23	Atom
RelpimoC	Brainy	70	Atom
Raytrace	Brainy	84	Core2

3 Research questions

We have seen several tools which try to make both static and dynamic replacements, and we have seen that both techniques can yield speedups. A thorough comparison of both approaches is still lacking.

In this context, given that Brainy looked promising, we decided to port Brainy to Java. It would give an opportunity to compare the Brainy approach to CoCo and CollectionSwitch (Chameleon is harder to compare with, as it requires a modified JVM).

Another interesting set of questions around Brainy is connected to how Brainy generates its data. The micro-benchmark generation scheme that Brainy uses encodes theories about how collections are used. Do these assumptions make sense? How can we generate realistic micro-benchmarks, which show the strengths and weaknesses of the different collections we have available?

In the rest of this thesis, we will focus on the following questions.

- What is an effective model of how data-structure are used?
- Can we predict data-structure performance?

4 Contributions

4.1 JBrainy: Micro-benchmarking Java Collections with Interference (Work in Progress Paper)

In this paper, we consider the problem of evaluating the performance of collections directly, using micro-benchmarking.

In their study, Costa et al. [CASL17] evaluated collections, one method at a time. Brainy generated micro-benchmarks with random sequences of method calls. The latter method has an advantage over the former, in the sense that it may capture “interference” between different operations. For example, is it worth sacrificing a little time at insertion, if iteration is faster?

Now, a generative approach like that of Brainy’s requires a model of how collections are *used*. How realistic is Brainy’s model? We explain that Brainy’s model could be improved, and present an alternative, which we call Pólya profiles. We evaluate nine collections from the Java Collections Framework on synthetic benchmarks.

We report that we found ArrayList to yield the best running time in 90% of our benchmarks, in accordance with previous results. In contrast with previous works, we found LinkedHashSet to be faster than alternatives in 78% of cases, while TreeMap and LinkedHashMap yielded better performance than HashMap in 84% of cases.

We conjecture that LinkedHashSet and LinkedHashMap work so well in our case because our benchmarks exercise one of their strengths: They sacrifice a little time when doing insertions, to gain a lot when iterating over the whole collection. Single-operation benchmarks do not exercise this feature, but our multi-operation workloads trigger this behavior quite often.

Contributions of the paper

- A port of the Brainy benchmark generation process to Java.
- A new model for collection usage.
- A study of how Java collections react to workloads built with the model
- A comparison of the result with existing previous work on micro-benchmarking collections

4.2 Paper II

In this paper, we test our synthetic benchmarks on real-world programs. Chameleon, CoCo, and CollectionSwitch used benchmarks from the DaCapo benchmark suite. We ported the full Brainy approach to Java, and compared JBrainy's improvements with greedy search, on five benchmarks of the DaCapo suite [Bla+06]. JBrainy and greedy search could use any of the 9 most popular Java collections.

This comparison revealed that JBrainy's classifier suffers from biases in the training data. We describe how the benchmark synthesis model we borrowed from Brainy is biased in favor of ArrayList, LinkedHashSet, and LinkedHashMap. We highlight that despite these problems, it could nonetheless do as well as greedy search for a fraction of the time spent. JBrainy and greedy search were not as effective as the state of the art, and we suspect this difference is due to biases in the training data, but also to the set of collections we used: Chameleon, CollectionSwitch and CoCo use a wider set of collections than we did, including lazy collections, array-based maps, and hash-based lists.

Contributions of the paper

- A port of the full Brainy approach to Java.
- A study of the challenges of porting the Brainy approach to Java
- An evaluation of the JBrainy tool on five well-known benchmarks, which we compare with greedy search.

5 Conclusions and future work

In this thesis, we started with the Brainy study, by Jung et al. We have ported their benchmark generation method to Java. We compared the behavior of collections from the Java Collections Framework with results by Costa et al. [CASL17]. We ported the Brainy approach to Java, and compared its performance with greedy search on five Java benchmarks. We notice that JBrainy works roughly as well as greedy search, for a small fraction of the cost. However, JBrainy's benchmark synthesis models is biased, and the tool does not work as well as the state of the art. Since greedy search did not outperform JBrainy significantly, we conjecture that the poor performance of both methods is due to the restricted set of collections we have considered. Expanding the set of collections available to JBrainy is a natural next step in our work.

Next, the Brainy study highlights the importance of the CPU architecture in choosing the right collection for a program. Is it the same for Java programs? We still do not have an answer to this question.

Lastly, techniques like Brainy, which use machine learning to predict the cost of an optimization, are becoming mainstream [WO18]. Now, machine learning requires data, and labeled data about programs is not easy to come by. As Wang and O'Boyle explain [WO18]:

> The most immediate problem continues to be gathering enough sufficient high quality training data. Although there are numerous benchmark sites publicly available, the number of programs available is relatively sparse compared to the number that a typical compiler will encounter in its lifetime. This is particularly true in specialist domains where there may not be any public benchmarks. Automatic benchmark generation work will help here, but existing approaches do not guarantee that the generated benchmarks effectively represent the design space. Therefore, the larger issue of the structure of the program space remains.

Brainy used automatic benchmark generation. Unfortunately, JBrainy sometimes generates programs nobody would ever write, and our investigations show that the resulting training data is severely imbalanced. JBrainy does not know how real traces look like, and it doesn't know what constraints sequences of method calls should satisfy (e.g. why would you call `clear` on an empty collection?). In the future, we plan to work on more realistic generative models. We could either learn them from real-world data or by instrumenting collections to inform the generation of new benchmarks. Alternatively, we could derive a model from a user specification of the abstract data-type, with hints about run-time properties of implementations.

References

[Bas+18] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. "Darwinian data structure selection". en. In: *Proceedings of the 2018*

26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018. Lake Buena Vista, FL, USA: ACM Press, 2018, pp. 118–128.

[Bla+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss, Aashish Phansalkar, Darko Stefanovic, and Thomas VanDrunen. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". en. In: (2006), p. 22.

[CA18] Diego Costa and Artur Andrzejak. "CollectionSwitch: a framework for efficient and dynamic collection selection". en. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. Vienna, Austria: ACM Press, 2018, pp. 16–26.

[CASL17] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. "Empirical Study of Usage and Performance of Java Collections". en. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*. L'Aquila, Italy: ACM Press, 2017, pp. 389–400.

[FSS83] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. "Experience with the SETL Optimizer". en. In: *ACM Transactions on Programming Languages and Systems* 5.1 (Jan. 1983), pp. 26–45.

[Has+16] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. "Energy profiles of Java collections classes". en. In: *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 225–236.

[Noa] *ISO 25010*.

[Jun+11] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. "Brainy: effective selection of data structures". In: *ACM SIGPLAN Notices* 46.6 (June 2011), pp. 86–97.

[LR09] Lixia Liu and Silvius Rus. "Perflint: A Context Sensitive Performance Advisor for C++ Programs". In: *2009 International Symposium on Code Generation and Optimization*. Mar. 2009, pp. 265–274.

[MPC14] Irene Manotas, Lori Pollock, and James Clause. “SEEDS: a software engineer’s energy-optimization decision support framework”. en. In: *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. Hyderabad, India: ACM Press, 2014, pp. 503–514.

[MSS10] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. “Four Trends Leading to Java Runtime Bloat”. In: *IEEE Software* 27.1 (Jan. 2010). Conference Name: IEEE Software, pp. 56–63.

[SVY09] Ohad Shacham, Martin Vechev, and Eran Yahav. “Chameleon: Adaptive Selection of Collections”. en. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), p. 11.

[WO18] Zheng Wang and Michael OBoyle. “Machine Learning in Compiler Optimization”. In: *Proceedings of the IEEE* 106.11 (Nov. 2018). Conference Name: Proceedings of the IEEE, pp. 1879–1901.

[Xu13] Guoqing Xu. “CoCo: Sound and Adaptive Replacement of Java Collections”. In: *ECOOP 2013 Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–26.

INCLUDED PAPERS

JBRAINY: MICRO-BENCHMARKING JAVA COLLECTIONS WITH INTERFERENCE (WORK IN PROGRESS PAPER)

1 Introduction

Java developers use collections extensively and are often faced with the task of picking a collection class. The Java collection framework provides documentation describing each collection's functional properties in an interface, and supplies several classes implementing this interface. However, it can be difficult to pick the most appropriate implementation, and in practice software developers often make sub-optimal choices when picking collections [SVY09].

When developers are unsure which collection class to use, they can run benchmarks on their application and compare different solutions. This approach gives precise insight, evaluating collection classes in the context in which they are used. However, in practice developers may lack the time to benchmark each use of collections in their code. Instead they turn to existing guidelines and look for general strategies for datastructure selection.

Collections have different *usage profiles*, which we can think of as statistical distributions of sequences of operations. Different collection classes perform better for different usage profiles, e.g., a linked list may more efficiently support insert-at-the-beginning operations than an array-based vector, whereas profiles dominated by index-based lookup may be faster on the vector.

Therefore, to recommend a collection class to a programmer, we must (a) understand what the programmer's usage profile is, and (b) have a mechanism for predicting the performance of a given collection class for that usage profile. Our research question in this paper focuses on the second point: *how can we obtain a performance model that allows us to predict collection class performance with a level of precision that is adequate for giving effective recommendations?*

Related work has explored models for two kinds of profiles, which we here call *single-operation profiles* and *multi-operation profiles*. Single-operation profiles are the basis for the CollectionsBench study by Costa et al. [CASL17], in which the authors study Java collections from the standard and third-party libraries by examining one operation at a time. Multi-operation profiles are the basis for the Brainy approach [Jun+11], in which the authors synthesise benchmarks for C++ to exercise random sequences of operations.

Both kinds of profiles can produce guidelines for developers for picking data structures, but neither is perfect: single-operation profiles capture typical usage scenarios, but cannot capture *interference* between different operations (one operation affecting the performance of another). Multi-operation profiles can capture interference, but present a much larger and more challenging search space for benchmarking. To facilitate the comparison between these two approaches this paper makes the following contributions:

- a porting of the Brainy approach to Java via the JBrainy tool.
- *Pólya Profiles*, a refinement of multi-operation profiles.
- an evaluation of the JBrainy approach on Java collections.
- an initial comparison of JBrainy and CollectionsBench.

The rest of this paper is organised as follows: Section 2 describes the methods used in the experiments presented in Section 3. We discuss results and implications of the experiments in Section 6, review related work in Section 5, and conclude in Section 9.

2 Methods

In this section we describe the three approaches that we consider in this paper in terms of the usage profile they embody.

Single-Operation Profiles Costa et al.'s CollectionsBench system [CASL17] builds models for five hand-written usage profiles that test, respectively, element insertion, multi-element insertion, is-element-of checks, index-based lookup (lists only), and iteration. Except for iteration, all of these profiles capture the exclusive use of a single operation.

While these single-operation profiles represent some of the real-life usage of collections, they do not directly capture e.g. uses in which the code alternates between adding and deleting. If there is nontrivial statistical *interference* between the performance of addition and deletion operations for a given collection class, models built from single-operation profiles may be inaccurate.

Multi-Operation Profiles To account for the possibility of interference between different operations, Jung et al.'s Brainy system [Jun+11] explores a multi-operation usage profile that assumes that operations occur with a certain probability distribution but independently of any previously selected operations. Brainy uses this profile to generate a family of microbenchmarks, each a sequence of randomly selected operations, and executes the benchmarks to build a performance model.

Thus, Brainy's multi-operation profiles allow for construction of a model that can directly observe interference between operations, i.e., whether one operation coinciding with another may speed up or slow down that operation. On the other hand, Brainy is unlikely to generate microbenchmarks that correspond to CollectionsBench-style single-operation profiles, even though such profiles arguably correspond to practically relevant usage patterns.

Pólya Profiles To address the limitation with multi-operation profiles, we propose a third model, which we call *Pólya Profiles*. Pólya profiles are multi-operation profiles in which the probability distribution is biased through a Pólya urn [Mah03]: for the first operation, we are equally likely to select any of a collection's operations, but each time we choose an operation, we increase its likelihood of being picked again. Consequently, when we use Pólya profiles to generate microbenchmarks, we lean towards generating benchmarks that use a small number of operations frequently. However, when we consider all benchmarks, our approach favours no particular method, as all methods have an equal probability of being favoured in one benchmark. An example of such a generated profile is shown in Figure 1, in which the method addAll is called many more times than other methods.

3 Experiments

To explore the impact of Pólya profiles in generating more accurate performance models, we here compare the recommendations from CollectionsBench's single-operation profiles against recommendations from our own JBrainy system, which uses Pólya profiles.

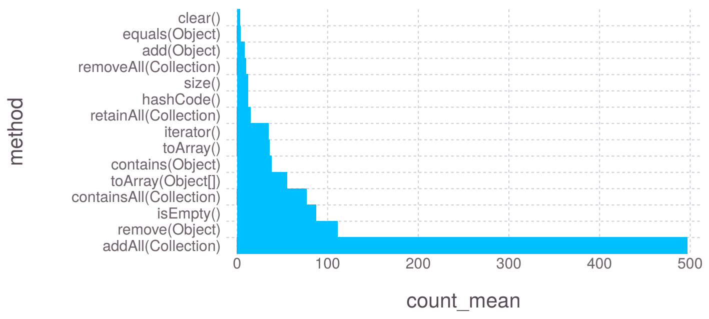


Figure 1: The distribution of method calls for one synthetic benchmark

3.1 Experimental setup

Our experiments focused on collections in the Java standard library, where we considered a selection of lists (ArrayList, LinkedList and Vector), sets (HashSet, LinkedHashSet and TreeSet), and maps(HashMap, LinkedHashMap, and TreeMap). Each collection was tested with integer elements, using the Java Microbenchmarking Harness [CBLA19] for compatibility with CollectionsBench and to simplify our evaluation methodology [Bla+08].

We ran our microbenchmarks on an Intel(R) Core(TM) i7-3820 CPU 3.60GHz with 16 GB of RAM, running Ubuntu 18.04 (Linux 4.18.0-15-generic), on OpenJDK 10.0.2. Each benchmark ran as many times as possible during 250ms, with three warm-up runs and five sampling runs.

We configured the microbenchmarks to execute 10, 100, and 1000 operations each, and initialised the collections to initially contain 0, 1000, or 10000 entries. Together, these two parameters yielded 3×3 different configurations. For brevity, we only report results aggregated over all configurations, the impact of benchmark size and collection size are briefly discussed in section 6.

CollectionsBench We re-ran CollectionsBench with the configuration that we reported above. The only changes that we made were to reconfigure CollectionsBench to use integers instead of strings as collection elements, and to analyse only collections from the Java standard library.

JBrainy For JBrainy, we first re-implemented Jung al.'s benchmarking strategy from their Brainy system in Java. We then augmented it to utilise Pólya profiles. For each interface of interest, we synthesised 4500 ($500 \times 3 \times 3$) microbenchmarks for each collection class that each exercised the methods declared in the interface.

Comparison of CollectionsBench and JBrainy To compare the two approaches, we first identified the *dominant operation* for each JBrainy microbench-

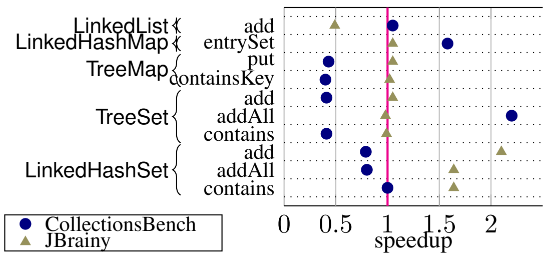


Figure 2: Comparison between speedup predictions by CollectionsBench and JBrainy for various operations

mark, i.e., the operation with the largest number of invocations in the benchmark. Second, we computed the speedup of each benchmark, compared with a *baseline collection*, for which we chose the most popular collections reported by Costa et al.: ArrayList for lists, HashSet for sets, HashMap for maps. For each single-operation profile in CollectionsBench, we then aggregated results from all JBrainy microbenchmarks with a matching dominant operation and compared median speedups for each tool.

3.2 Results

Figure 2 shows the ten largest differences between JBrainy's and CollectionsBench's results (out of 26 results in total). For example, CollectionsBench reports that LinkedList.add has roughly the same performance as ArrayList.add, while JBrainy reports it as being slower by approximately a factor of two. Conversely CollectionsBench reports a speedup of 0.41 for TreeSet.add compared to HashSet, while JBrainy reports these operations as having roughly comparable performance, and we observe a similar difference for TreeMap.put when compared to HashMap.

For completeness, we also report the recommendations that JBrainy gives for operations that CollectionsBench does not report on. Figure 3 shows the median speedups for each collection class and the dominant operation in each synthetic benchmark. We report medians instead of averages as the distribution of speedups is skewed (skewness ≈ 14.78).

In the case of lists, LinkedLists are approximately twice as slow as ArrayLists, while Vectors are approximately 1.1 times slower than ArrayLists. In the case of maps, LinkedHashMap is faster for most of the methods in the interface, and particularly for methods put (speedup ≈ 1.28), hashCode ($s \approx 1.20$), and remove ($s \approx 1.10$). TreeMap is only faster for benchmarks where the most common method is `clear`, with a median speedup of 1.07. Similarly in the case of sets, LinkedHashSet is faster for all of the methods that we considered, and particularly for methods toArray ($s \approx 2.96$), toArray ($s \approx 2.85$), and add ($s \approx 2.10$). TreeSet is faster on method clear with a median speedup of 1.18.

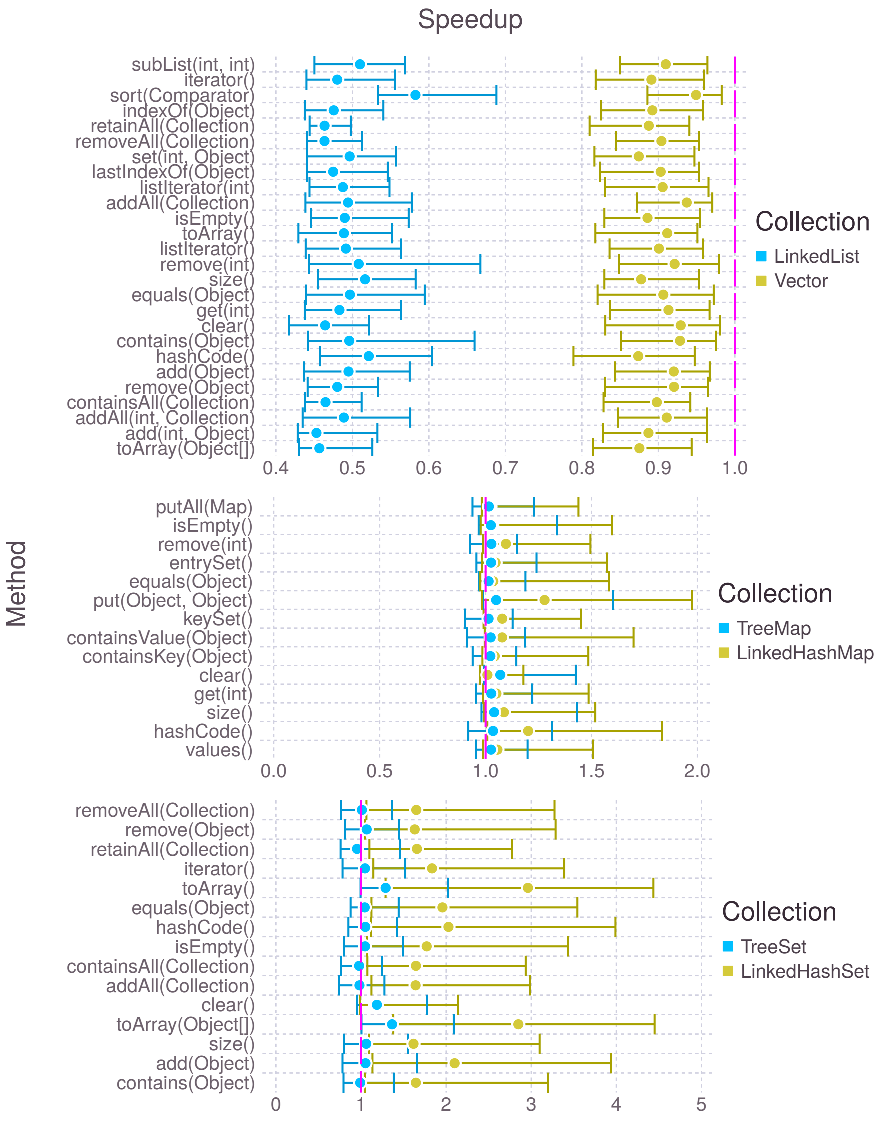


Figure 3: Median speedup of various collections compared to baseline (in magenta), with 25% and 75% quantiles

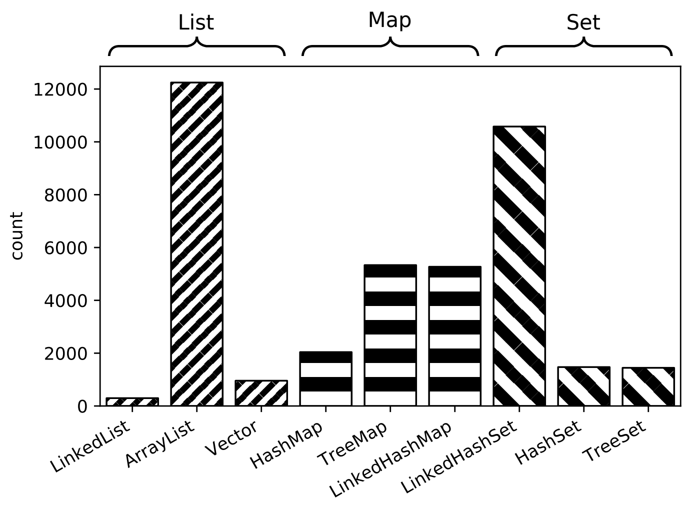


Figure 4: Count of fastest benchmarks depending on the collection class used.

Figure 4 summarises how often JBrainy found a particular collection class to be optimal for any of its benchmarks. For lists, ArrayList is fastest in 91% of our benchmarks, while Vector and LinkedList are the best fit in respectively 7% and 2% of all runs. This agrees with Costa et al.'s findings that ArrayList may be a good default choice. For maps, the situation is more nuanced. LinkedHashMap and TreeMap are the best fit for respectively 42% of benchmarks, while HashMap is the best fit for 16% of benchmarks. For sets, LinkedHashSet is the best data structure for 78% of our generated benchmarks, while HashSet and TreeSet are the best fit for 11% of benchmarks each.

4 Discussion

JBrainy does not explore iteration over lists directly. However, the implementation of the operations toArray() and hashCode() is dominated by iterating over the underlying collection, so we use these as a proxy for iteration performance, since adaptive inlining is likely to be equally effective for both sets of microbenchmarks.

We can conjecture why LinkedHashSet performs well on toArray() and similar operations: These operations iterating over all the elements of the set. In a HashSet, this iteration requires iterating over all buckets in the hash table, whereas for a LinkedHashSet, the iteration only goes through the set's internal linked list of the set elements. The same considerations apply to hashCode(), which requires iterating over all elements for both LinkedHashSet and LinkedHashMap.

We further note that LinkedHashMap's put and add operations perform surprisingly well. We conjecture that the additional overhead of these operations is amortised by later calls. In the case of TreeSet and TreeMap, the performance of the `clear` method comes about because clearing a tree only requires `NULL`ing the root node, while clearing (linked) hash maps requires iterating over all hash buckets.

For sets, Costa et al. focus on third-party alternatives to HashSet [CASL17], while our results show that LinkedHashSet is faster than HashSet in a majority of cases. For Maps, Costa et al. describe HashMap as providing solid performance, while our results show that LinkedHashMap often performs better. For Lists, our results confirm the findings of the CollectionsBench study: ArrayLists are significantly faster than LinkedLists in the majority of cases.

A key insight from our work is that LinkedHashSet and `LinkedHashMap`, which account for a small percentage of Java collection classes used in real-world programs [CASL17], can outperform more popular alternatives when the benchmark involves calling many different methods on the object. If binning by collection and benchmark size does have an effect on the median speedup, the fastest collection remains the same in 84% of cases.

Our results strongly suggest that there is interference between different operations in the interfaces that we examined. This in turn means that performance models based on Pólya profiles (or other multi-operation profiles) may provide more accurate suggestions for collection class selection than those of single-operation profiles.

Threats to Validity. While our initial results are very encouraging, we observe a number of threats to validity that we will explore in future work. Regarding internal validity, we have not yet systematically analysed the difference in recommendations from JBrainy and CollectionsBench, nor have we validated our models and recommendations by exploring their impact on the performance of existing software. Moreover, we have not yet explored fully the impact of collection size on results.

Regarding external validity, we have only benchmarked one hardware setup and one virtual machine, and not considered third-party collection classes.

5 Related work

Automatic datastructure replacement for Java has been explored e.g. by Shacham et al. [SVY09] who explored a modified Java VM that could automatically propose or perform container class migrations, though the authors only explored automatic migration for reducing memory footprint. Xu's CoCo system [Xu13] similarly enabled automatic dynamic collection class migration, but successfully targeted performance optimisation with the ability to migrate more than once at runtime. Both tools used hand-written rules for controlling migration. Recently, Costa et al. presented a dynamic migration technique [CA18] that improves over CoCo by utilising performance models generated from single-operation profiles [CASL17], for dynamic collection class selection instead of hand-coded rules. Hasan et al. [Has+16] similarly obtain energy usage models for container classes of varying sizes.

Similar ideas have also been explored for C++ [Jun+11], though research in automatic datastructure selection dates back further [FSS83].

6 Conclusions and Future Work

Developers are often faced with the need to pick a collection datastructure from options that appear functionally equal. One way to assist them is to providing decision support in the form of performance insights from micro-benchmarking.

We have explored one such micro-benchmarking approach in our tool JBrainy, which builds on the benchmark synthesis approach introduced in Brainy [Jun+11]. Using JBrainy and its novel Pólya profiles, we have run an initial performance evaluation experiment following the setup of the CollectionsBench study [CASL17]. While CollectionsBench focused on improvements from using third-party Java collections, we have focused our experiment on collections in the Java standard library. For lists, our results agree with those of CollectionsBench, finding ArrayList to be the best candidate for the vast majority of benchmarks. However, for maps and sets, our results show that less well-used collections such as LinkedHashMap or LinkedHashSet can improve the performance of benchmarks.

As an immediate next step we plan to include the third-party collections used in the CollectionsBench study in our work to get a better comparison between the two approaches, and to increase collection sizes further.

In addition, we plan to explore various threats to validity. Particularly, validating the recommendations from JBrainy on real-world software would allow to evaluate how realistic Pólya profiles and our configurations are and how much insight can be gained with more realism.

7 Acknowledgements

This work was partially supported by Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by Knut and Alice Wallenberg Foundation.

References

[Bla+08] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. “Wake up and smell the coffee: evaluation methodology for the 21st century”. en. In: *Communications of the ACM* 51.8 (Aug. 2008), pp. 83–89.

[CA18] Diego Costa and Artur Andrzejak. “CollectionSwitch: a framework for efficient and dynamic collection selection”. en. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. Vienna, Austria: ACM Press, 2018, pp. 16–26.

[CASL17] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. “Empirical Study of Usage and Performance of Java Collections”. en. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE ’17*. L’Aquila, Italy: ACM Press, 2017, pp. 389–400.

[CBLA19] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. “What’s Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks”. In: *IEEE Transactions on Software Engineering* 47.7 (2019). Conference Name: IEEE Transactions on Software Engineering, pp. 1452–1467.

[FSS83] Stefan M. Freudenberger, Jacob T. Schwartz, and Micha Sharir. “Experience with the SETL Optimizer”. en. In: *ACM Transactions on Programming Languages and Systems* 5.1 (Jan. 1983), pp. 26–45.

[Has+16] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. “Energy profiles of Java collections classes”. en. In: *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 225–236.

[Jun+11] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. “Brainy: effective selection of data structures”. In: *ACM SIGPLAN Notices* 46.6 (June 2011), pp. 86–97.

[Mah03] Hosam M Mahmoud. “Pólya Urn Models and Connections to Random Trees: A Review”. en. In: *Journal of the Iranian Statistical Society* (2003), p. 64.

[SVY09] Ohad Shacham, Martin Vechev, and Eran Yahav. “Chameleon: Adaptive Selection of Collections”. en. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), p. 11.

[Xu13] Guoqing Xu. “CoCo: Sound and Adaptive Replacement of Java Collections”. In: *ECOOP 2013 Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–26.

JBrainy: Effective Selection of Data Structures in Java

1 Introduction

There are many ways to make Java programs run faster. One way is to look at the collections they use. Can we make the program faster by switching an ArrayList to a LinkedList , or a HashMap to a TreeMap? Usually, knowing the answer to those questions is considered a matter of professional skill.

In this paper, we consider the problem of helping the developer choosing what data-structures to use, by providing them with suggestions. We could find solutions that targeted C++ and Java. For Java, we found three solutions. Chameleon [SVY09] requires to modify the JVM, so we decided that it was outside of the scope of this work. CoCo [Xu13] and CollectionSwitch [CA18] both use dynamic optimization: the developer doesn't need to choose, they provide collections which switch their underlying implementation at runtime. Another one, Brainy [Jun+11], targets C++ programs, and suggests static changes. It uses a machine-learning based classifier to give advice to the developer, based on runtime information.

Here is a picture of the design space:

	Java	C++
Dynamic	CoCo, CollectionSwitch	???
Static	???	Brainy

In this table, a C++ implementation of CoCo or CollectionSwitch would be an interesting engineering problem, but it's probably not a research problem.

On the other hand, adapting Brainy to Java poses new challenges. Brainy uses dynamic information (e.g. hardware performance counters) to take decisions, but

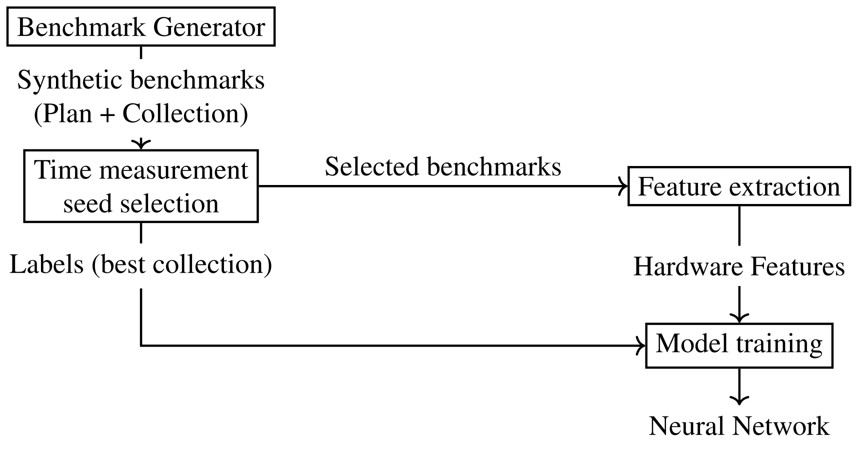


Figure 1: Overview of Brainy

benchmarking Java programs does not work like benchmarking C++ programs: the Java Virtual Machine uses many dynamic optimizations which impact the running time of programs in tricky ways. In this paper, we present our implementation of the Brainy tool for Java programs, and answer the following research questions:

- **RQ1**: How well can the Brainy approach by applied to Java?
- **RQ2**: How well does the Brainy approach work on Java programs?

2 Background: Brainy

Brainy tries to answer the question: *"what data-structure should I use, when the original data-structure behaves a certain way?"*. Provide Brainy with a program and a source location, and it will suggest which collection to use there.

2.1 Overview

Brainy uses a learned classifier to give a data-structure suggestion, based on information about the hardware performance of a data-structure. To do that, Brainy needs to know which data-structures are best, in which cases. Getting this information from real programs would take a lot of effort, so instead, Brainy builds random programs to test data-structures against. This information is then used to train a classifier.

Figure 1 presents an overview of Brainy's architecture. Brainy needs to generate benchmarks which would make a balanced training set. To do so, it generates many benchmarks and evaluates how long they take with different data-structures.

Some collections will win more often than others, so Brainy stops recording these benchmarks when it has enough. Once this is done, Brainy has both a set of selected benchmarks, and the best collection for these benchmarks. It can move onto the more expensive task of collecting hardware features for the benchmarks. Labels and features are then used to train a neural-network based classifier.

2.2 Building Brainy's training set

Brainy generates random micro-benchmarks, which it runs, to get information about how data-structures behave. These micro benchmarks are composed of a **plan** – the sequence of methods to use – and a data-structure we want to experiment with.

A plan can be uniquely identified with:

1. The abstract data-type (e.g. List, Map, Set) of the data-structures we'd like to compare.

2. Its seed. The seed will be used to pick methods at random, and generate arguments for these method calls.

Brainy's model for generating plans is fairly simple: First, Brainy assigns a weight to each method the abstract data-type supports, and then, it sample method names according to these weights.

With a plan and a data-structure, we get a **benchmark**, which is runnable. We measure how long the benchmark takes to run, which tells us how well the data-structure performs for this specific plan. By running the same plan with different data-structures, we can compare them. A data-structure **wins** for a plan if it runs at least 5% faster that all the alternatives.

Phase I: Time measurements and seed selection

When comparing data-structures, we might observe that some are more versatile than others: For example, in Java, ArrayList is the best data-structure for many plans, while LinkedList rarely wins. Since getting the features for a benchmark is expensive, we want a balanced dataset: Each data-structure should win for a reasonable portion of the plans. To enforce that constraint, Brainy uses rejection sampling: Each data-structure is associated with a fixed-size bucket of plans. When data-structure DS wins for plan p, p goes in the bucket for DS. If the bucket is already full, the plan (and its seed) are discarded.

Once this phase is finished, the result is a list of pairs with the seed used to generate the plan, and the best data-structure for that plan. Since each bucket is full, the data-set should be balanced.

Phase II: Getting the features

In this phase, we iterate over the results of Phase I. We already know which collection is the best for each plan, the goal of this phase is to teach Brainy how to recognize whether a data-structure is efficiently used or not. To do this, we try each plan with other data-structures and collect hardware features.

In the original study, the authors selected the following features:

- Branch mispredictions
- L1 cache misses
- L1 cache miss rate (L1 cache misses / accesses)
- Size of collection elements / cache block size
- Branch mispredicition rates (Branch mispredictions / branch instructions)
- Cost features

Cost Features Intuitively, choosing the right data-structure requires knowing what operations we want to do, and how much these cost: If we insert many elements in a data-structure, we will pick one for which insertion is cheap. Brainy therefore gathers the cost associated with insertions, deletions, and searches performed during the benchmark. The authors use different definitions of cost:

- Cost of insertion: Number of data elements moved forwards or backwards by the insertion.
- Cost of deletion: Number of data elements moved forwards or backwards by the deletion.
- Cost of search: Number of data elements accessed before finding the element of interest.

Training data The result of this phase is a list of samples, one for each benchmark. Each sample contains the following information:

- The data-structure used to collect features
- The features obtained with instrumentation
- The best data-structure for this benchmark's plan.

2.3 Classification

Brainy splits the training data, and trains one model for each data-structure to replace. For example, if the user wants to replace a vector, Brainy will feed the features into a classifier specialized in doing replacements of vectors. Brainy considers 9 collections: vector, list, deque, set, AVL_set, hash_set, map, AVL_map, and hash_map.

For classification, Brainy uses artificial neural networks. Unfortunately, the paper does not mention about the structure and size of the neural network.

2.4 Evaluation of Brainy

To evaluate Brainy, the authors report on both the training accuracy, and the effect of applying Brainy's suggestions to several C++ programs. They compare results on two architectures, described below.

Desktop	
CPU	Intel Core2 Quad Q6600 2.4 GHz
Caches	4×32 KB L1 data, 2×4 MB L2 Unified
Memory	2GB SDRAM, 200GB HDD
Operating System	64-bit Ubuntu Desktop 8.04
Compiler	GCC 4.5 with libstdc++ 4.5.0

Laptop	
CPU	Intel Atom N2700 1.6GHz with HyperThreading
Caches	32 KB L1 data, 512 KB L2 Unified
Memory	512MB SDRAM, 8 GB SSD
Operating System	32-bit Ubuntu Netbook Remix 9.10
Compiler	GCC 4.5 with libstdc++ 4.5.0

Training accuracy

To test the training accuracy of Brainy, the authors generate a test set of 1000 benchmarks, for each data-structure model. They report an accuracy between 80% and 90% for the Core2 architecture, and an accuracy between 70% and 80% for the second architecture.

Effect on real-world programs

The authors compare Brainy's suggestions and their effect on 4 programs: Xalancmbk, Chord Simulator, RelipmoC, and Raytrace. They compare inputs of 3 different sizes for Chord Simulator.

Brainy works well on these benchmarks. On average, Brainy improves performance by 27% on the Core2 architecture, and by 33% on the Atom architecture.

All benchmarks report at least 10% of improvement, and up to 77% of improvement in one case.

3 JBrainy

JBrainy is the tool we developed, which tries to adapt the same process to Java Programs. We tried to stay as close to the original work as possible. Figure 2 shows that the structure is similar to that of Brainy. The only structural difference is that Brainy measures the run-times of programs while doing seed selection, while we split this in two steps. This difference is explained in section 3.3, and it's impact is detailed in section 7. Here we list some key differences between our tool and Brainy.

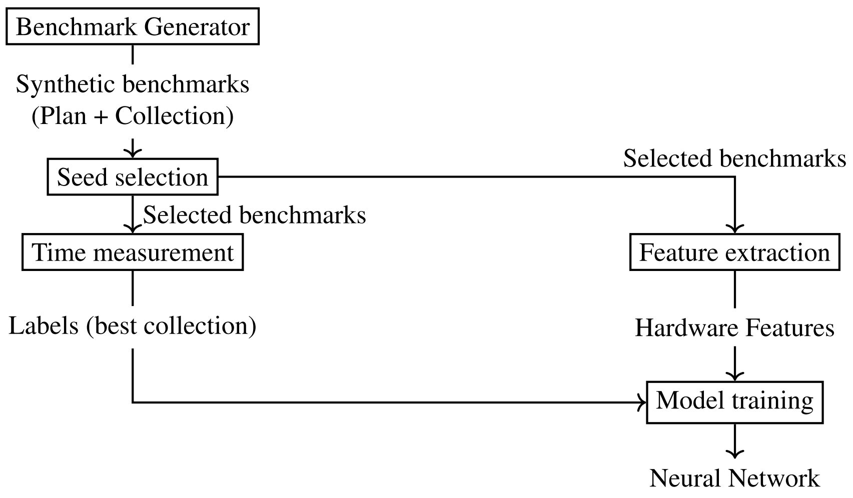


Figure 2: Overview of JBrainy

3.1 Candidates for replacement

In this work, we focus on collections in the Java Collections Framework that implement the interfaces List, Map, and Set in the java.util package, each of which represent an abstract datatype (ADT).

Figure 3 lists these interfaces. For each, we consider the three *implementations* of this data structure offered by the java.util package, as follows:

1. we consider every point in the program that invokes the constructor of such a data structure a **replacement slot**, meaning that we will consider performing **replacement** at that program point, and

List	Set	Map
LinkedList	HashSet	HashMap
ArrayList	LinkedHashSet	LinkedHashMap
Vector	TreeSet	TreeMap

Figure 3: The three Java collection interfaces (header row) that we consider, and the three data structures each that serve as the interfaces' implementations.

2. for any replacement slot for an implementation of some interface, we will consider a replacement by any other implementation of that same interface; in other words, we only allow sets to be replaced by other sets, etc.

   This restriction accommodates both fundamental differences in the operations offered by each interface and differences in the interfaces' semantics, at least where these are specified.

In practice, real-world programs have hundreds of replacement slots. Since we are not aware of any simple static strategies that allow pre-selecting the most significant replacement slots, JBrainy must be able to potentially perform replacement at any replacement slot. By contrast, Brainy (to the best of our understanding) exploited expert knowledge to select only one replacement slot per benchmark program.

3.2 Class replacement

Brainy performs replacements in two parts of its overall process: when generating synthetic applications for benchmarking, and when updating application code. To vary which data structure a given piece of code uses, Brainy uses C++ templates that represent ADTs at least for benchmarking. We expect that the original Brainy implementation uses the same or a similar technique for application code. Since Java lacks a mechanism that is equivalent to (compile-time evaluated and inlined) templates, JBrainy instead relies on dynamic inlining strategies whose performance characteristics are not obvious.

We therefore designed JBrainy to vary data structures by replacing constructor calls of data structures of interest by constructor calls to possible alternatives. Since different Java programs use different build systems and may rely on generated code[1], we perform replacement at the level of Java bytecode rather than at source code level. Advanced program transformation frameworks may implicitly perform changes or even optimizations [VR+10] as part of going through an intermediate representation; since this could cause interference with our measurements, we instead constructed a minimally invasive transformation tool based on the ASM library [Kul07].

[1]For example, the `fop` benchmark generates a number of Java files from XML specifications

Since Brainy only used a single replacement slot, it could rely on manual adaptations instead of requiring automation as we do here.

JBrainy's automatic replacement strategy induces a number of complications that we describe below.

Interface mismatch A replacement across two implementations of the same interface may not typecheck. For instance, consider the following code:

```
List<Integer> l1 = new LinkedList<>();
LinkedList<Integer> l2 = new LinkedList<>();
```

Here, the Java type system allows replacing the constructor call in the initialization of `l1` by `new ArrayList<>()`, but not the initialization of `l2`, since ArrayList is not a sub-type of LinkedList. Moreover, we cannot generally abstract the type of l2 to `List<Integer>`, since LinkedList is more that a List: it also implements the Deque interfaces, which offers operations not found in List. Meanwhile, ArrayList does not implement Deque, which means that it is not generally a viable replacement for LinkedList. In practice, a replacement for a concrete collection data structure must thus be a sub-type of the data structure.

Semantic changes Java interfaces such as List or Map have no mechanism for enforcing the behavioral properties of ADTs, and their (natural language) specifications are not usually comprehensive. As a consequence, two collections may implement the same Java interface, but but not implement the same ADT.

For example, HashMap, LinkedHashMap and TreeMap are all Maps, but their iteration orders can differ (e.g., iteration order for HashMap is undefined, but guaranteed to be in a specific order for TreeMap). If the iteration order is important for the program to be correct, some replacements will alter program behavior.

Constraints on Elements Similarly, some collections might implement the same interface but put different constraints on the type of elements they accept. For example, a TreeMap requires its elements to implement the Comparable interface, which is not a requirement for a HashMap.

Mitigating class replacement challenges

As we have seen, the Java collection interfaces themselves are not a sufficiently strong mechanism to ensure that their implementing data structures are mutually replaceable. Given a replacement slot that constructs an object of type A, the de-facto requirement for a replacement from A to B is that (1) B is *statically* a sub-type of any type of any variable that the object constructed in that slot may flow into, and (2) B *dynamically* expects no more constraints on elements as A

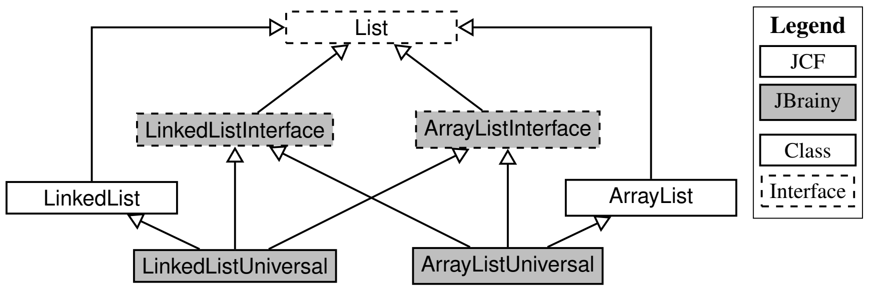


Figure 4: JBrainy sub-typing hierarchy for two container classes, ArrayList and LinkedList. For each such container *T*, *T*Interface serves as general interface, and *T*Universal serves as implementation (extending *T*) that implements *all* Interfaces.

and provides the same semantics as A for all possible program executions. We call these requirements the *shape* of the replacement slot.

Interface mistmatch We define the shape of replacement slots with Java interfaces, which define all the methods our replacement collection should support. For example, we created an interface for LinkedList -shaped collections, called LinkedList Interface. This interface defines all methods that LinkedLists support. Any collection we might use to replace a LinkedList must implement LinkedList Interface.

Now, the regular ArrayList does not implement LinkedList Interface, so how do we replace a LinkedList? We wanted universal replacements: Any collection should fit in any slot, as long as it's from the same family (List, Map, or Set). It does so by implementing the interface for every slot shape. We created wrapper classes which provide the methods we want, but use a different implementation. For example, ArrayListUniversal is a collection which implements both LinkedListInterface, ArrayListInterface, and VectorInterface, but it uses an ArrayList as the underlying implementation. It can fit in any list slot, but it performs like an ArrayList.

Figure 4 illustrates the resultant inheritance hierarchy.

To ensure that this replacement strategy has no significant impact on overall execution time, we evaluated the original benchmarks against benchmarks in which we had transformed almost[2] all types and all replacement slots as described above with the same methodology as described in changes in overall execution time.

[2]Due to constraints imposed by the Java compiler, we had to exempt the Java standard library from these transformations. As a consequence, we also had to replace a small number of static method calls to helper operations in the standard library by calls to equivalent custom static methods that we provided.

Semantic Changes and Constraints on Elements It is not generally possible to determine whether client code depends on semantic properties such as iteration order or universally satisfies constraints on elements. For the purposes of our evaluation, we approximated these properties by verifying that our benchmarks produced the expected output and otherwise manually identified and blocklisted replacements that were responsible for output changes.

3.3 Microbenchmarks

Our approach towards generating microbenchmarks in JBrainy follows the generative strategy used by Brainy, except that instead of generating microbenchmarks as source code, we directly generate them as bytecode. Since we lack information on how exactly Brainy assigns probabilities to method calls for each plan [Jun+11], we assign a weight of zero to each method, and increment the weight of a uniformly selected method, 100 times, then normalize.

Benchmarking Brainy targets C++ programs, while JBrainy targets Java Programs. The main consequence of this change is that the Java Virtual Machine performs JIT compilation and dynamic optimizations: The JVM "warms up", and calling the same program several times in the same JVM instance can make the program run faster. While the runtime of C++ programs can also vary due to operating-system level caching, the runtime of Java programs is generally subject to a more pronounced warmup effect [Bla+08].

We use three different methods for selecting seeds, gathering labels and features when generating JBrainy's training set. When selecting seeds, we run the benchmark 30 times, and record the running time. We selected this naive method because it is fast, as many seeds end up being rejected. After selecting interesting seeds, we measure the running time of the associated benchmarks more precisely, with the Java Microbenchmarking Harness (JMH). To warm up, the benchmark runs for $2 \times 2 \times 250$ms: We run the benchmark as many times as possible for 250ms, twice, for 2 JVM invocations. For measurements, the benchmark runs for $2 \times 5 \times 250$ms. We run the benchmark as many times as possible for 250ms, 5 times. We do this for 2 JVM invocations. To get the features, we run each benchmark 10 times within 1 JVM invocation. This seems complicated.

The complexity of this benchmarking scheme is justified by the tension between the JIT-compilation features of the JVM and the need for precise instrumentation. On the one hand, the JVM has elaborate dynamic optimization features, which motivates the use of JMH make sure the recorded times are realistic [3]. On the other hand, gathering the features [4], requires more control over instrumentation than JMH provides. These two aspects need to be balanced with the total time spent building the training set. We want to run as few runs as possible

[3]For example by preventing abusive dead code elimination

[4]particularly the time spent doing certain classes of operations, like insertions

while keeping the values stable enough for the classifier to work. Balancing those aspects was not easy and caused some problems, reviewed in section 7.

Benchmark generation When using Brainy, the user might configure the *size* of the elements to store in data-structures when running benchmarks. In the case of Java, we chose to use only integers, because collections store references to objects.

Feature extraction Brainy's original authors do not mention the tools they used for instrumentation. We used Java bindings to the PAPI library. Our experiments showed that starting and stopping the counters introduced an overhead, which we could avoid by reading the current value and subtracting the previously read value. With tracing, avrora runs 1.5 times slower with tracing, bloat runs 1.76 times slower, and fop runs 7.8 times slower.

For the cost of operations, Brainy's authors used metrics on the number of elements moved or accessed, while we used the number of cycles spent for each class of operations.

Feature selection Brainy's authors selected features using genetic algorithms. We use the same list of features the Brainy's authors have selected.

Classification Brainy uses Artificial Neural Networks for classification. We use the same method, but since there was no information in the paper about the size and structure of the networks, we used the defaults of the scikit-learn library.

4 Experimental setup

To evaluate JBrainy, we compare the effect of suggestions from JBrainy on a set of real-world programs. We compare both the jbrainy-optimized version with the original, but also with a ground truth, which is obtained by greedily trying different collections and comparing the results, which we discuss in section 5.

As the authors of the original Brainy emphasize, the hardware architecture of the machine influences the results, we therefore compare results on two different machines.

4.1 Benchmark suite

The benchmark suite we used was the Dacapo Benchmark suite [Bla+06]. We compare 6 programs: avrora, bloat, chart, fop, and lusearch. These benchmarks were selected because they have been used in CoCo and CollectionSwitch.

4.2 Machines used

We used one machine for running our benchmarks.

"Pascal"	
java version	1.8.0_292
CPU	11th Gen Intel(R) Core(TM) i7-11700K @ 3.60GHz
Caches	8×384 KB L1 data, 4 MB L2 unified, 16 MB L3 unified
kernel release	5.13.7-051307-generic
Memory	128 GB SDRAM, 2 GB Swap
Operating System	64 bit Ubuntu 21.04

We fixed the CPU frequency for the experiments to 3.6 GHz. We also enabled user-space RDPMC access for PAPI to enable fast readout of the performance counter registers.

4.3 Benchmarking programs

To get the runtime of one program, we run 20 JVM invocations with a heap size of 12GB. For each invocation, we run the program until it converges. In section 5, we compare the results of the first iteration with the results after convergence.

4.4 Ground truth

To get an idea of how good Brainy's predictions are, we need to compare Brainy-optimized programs with a baseline. Our baseline is a variant of greedily searching for the optimal collection. We generate possible alternative programs, and benchmark them, we then pick the best we found.

First, we select interesting replacement slots in the program. Not all replacement slots are equal: Some are the origin of a single large list used throughout the program, some are the origin of a large number of small maps, or one single list of maximum five elements. We measure the importance of a replacement slot by counting the number of method invocations on collections which share this origin. One could use other metrics, for example the total number of CPU cycles. Our choice is motivated by the overhead of accessing performance counters in Java: For example, tracing cycles for all collections on avrora took 13s, while tracing method invocations took less than 3.5s. The resulting ranking of replacement slots is not identical for both metrics, but the 10 busiest replacement slots were the same for avrora.

We select the 10 busiest slots and consider what possible replacements. Pure exhaustive search would generate every possible combination, but the number of candidates to consider is prohibitive: in the case of fop, we would need to benchmark 59049 candidates. Instead, we optimize each replacement point independently and merge the results to produce an "expected best" candidate. That way, we only need to benchmark 21 candidates. We then compare the running times of these candidates: Each benchmark is ran until convergence, 20 times, and the collection which yields the lowest median running time is selected.

5 Results

RQ1: How well can the Brainy approach be applied to Java?

The main challenge of porting Brainy to Java concerns benchmarking. When generating micro-benchmarks, we generated the bytecode directly. One could also generate Java code and compile it. We do not know the implications of that design choice yet. When benchmarking, as we highlighted in section 2.4, the complexity of the Java runtime system makes measuring the running time of many micro-benchmarks more error-prone and more expensive than it probably was for the original Brainy. While Jung et al. report 1000 micro-benchmarks for each collection as a low number, in our case, generating 100 benchmarks for each collection can already take 6 hours.

Next to benchmarking, instrumentation is a major challenge. Brainy requires fine-grained instrumentation for some features, like the cost of insertions. Supporting the same instrumentation required interfacing Java with a native library (PAPI), but this interfacing introduces some overhead, and this overhead influences the measurements!

RQ2: How well does the Brainy approach work on Java programs?

A good tool for collection suggestions must provide two things: it should be quicker than greedy search, and its suggestions should reduce effectively the running time of programs. We compare JBrainy with greedy search on several DaCapo benchmarks. Greedy search takes longer if we consider more replacement slots, so we only focus on the ten busiest. For speed, we compare how long optimizing a program with JBrainy takes, versus using greedy search. For effectiveness, we compare the running times of the greedily-optimized variants of the benchmark, with one optimized by JBrainy. Running times contain some noise, so we compare the median running time across 20 JVM invocations.

How fast is JBrainy?

Training the model takes approximately 6 hours. Seed sampling aims to find 100 seeds for each collection type. Some seeds were usable for benchmarks for all families (List, Maps, and Sets), so the number of unique seeds is lower: 577. We used 9 collections, for a total of $577 \times 9 = 5193$ benchmarks. Using JMH, getting the running time of these benchmarks took 5 hours 35 minutes. We obtained the feature vectors in 35 minutes, running each benchmark 10 times. Training the classifier takes less than a minute.

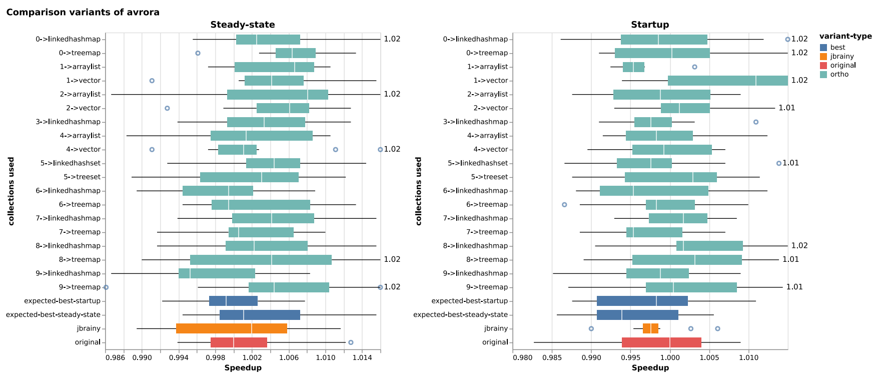
Once the model is trained, building a new JBrainy-optimized program takes around a minute. The classifier is quick to evaluate, so most of the time is spent collecting features from the original program. We provide a table for comparison.

Benchmark	Build time greedy (s)	Build time JBrainy (s)
avrora	3931	12
bloat	2093	70
chart	1334	80
fop	2085	56
lusearch	1320	12

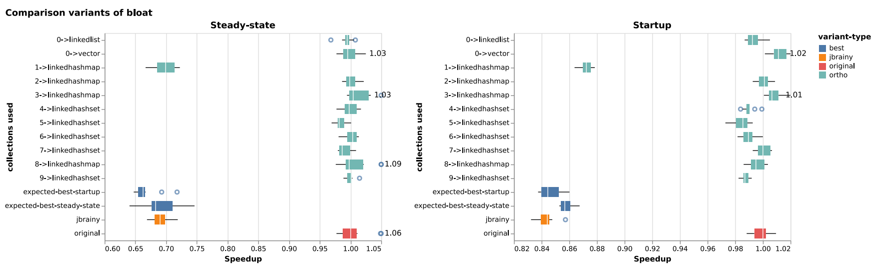
How effective is JBrainy?

Effect on real-world programs

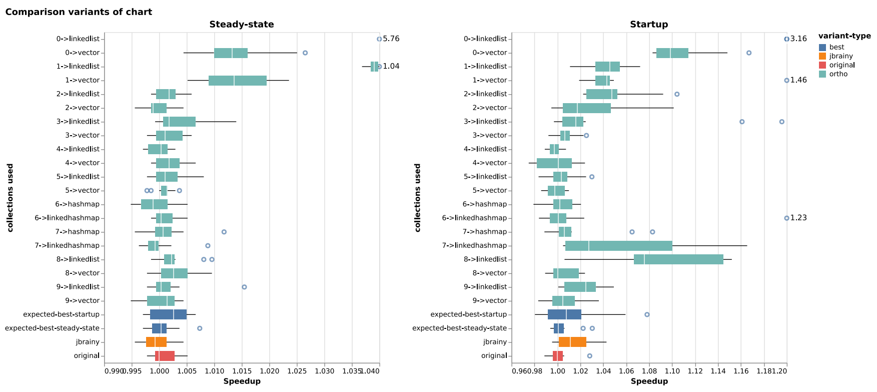
We plotted the best variants found by greedy search (in dark blue), which we compare with the original program (in red), and the program optimized with JBrainy (in orange). On the x-axis is the speedup, relative to the median running time of the original program, while the y-axis identifies which variant was run. To help identify why a variant is better than another, the teal bars denote programs where only one allocation site was changed to another collection. For example $5 \rightarrow$ linkedhashset means that this program was the same as the original, except a LinkedHashSet was used for the fifth busiest allocation site.



In the case of avrora, JBrainy does slightly worse than the original program, while greedy search does a little bit better. Since the difference is less than 1%, we assume it is negligible.



In the case of bloat, both JBrainy and greedy search find an important optimization: switching the busiest HashMap to a LinkedHashMap. This optimization yields 30% of improvement in steady-state performance.



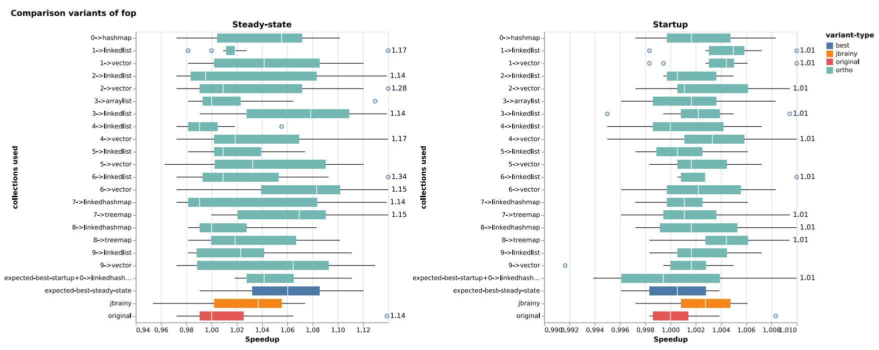
In the case of chart, greedy search found a change which multiplied running time by five: Switching the busiest list to a LinkedList . This is reassuring, as it shows the choice of collection did matter for that program. However, the original program was already quite close to the optimal program found by both greedy search and JBrainy.

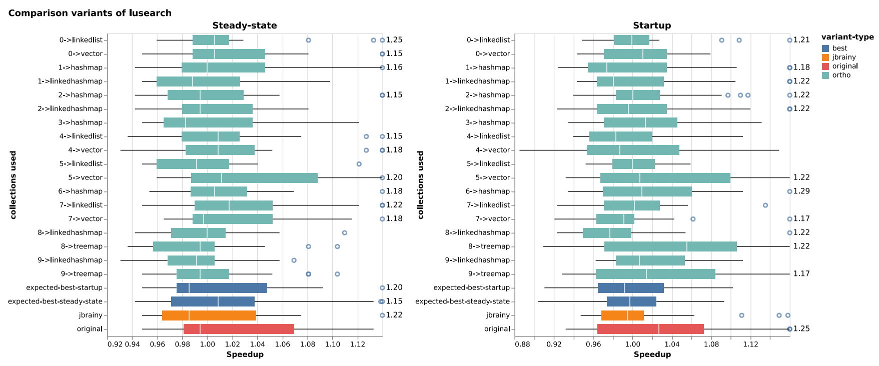
In the case of fop, greedy search *worsens* steady-state performance by 5%. We can see that the error bars cover most of the range of improvement, suggesting that it's not possible to improve the program much with the selected collections.

In the case of lusearch, Neither Greedy search nor JBrainy improved substantially the running time of the program.

Major issues with classification

Comparing the suggestions provided by greedy search and JBrainy, we notice that JBrainy agrees with greedy search in 36% of cases. Figure 5, comparing suggestions by JBrainy, reveals that JBrainy *always suggested the same things*. All lists





are replaced by ArrayList, all maps by LinkedHashMap, all sets by LinkedHash-Set. Our classifier is not useful. We come back to possible reasons in section 7.

6 Discussion

JBrainy's results are a bit disappointing: For four programs out of five, it did not find substantial optimizations, while the state of the art found optimizations in the tens of percents. It found *one* substantial optimization, for bloat. Interestingly, greedy search did not find large optimizations either. We notice two things:

First, the original programs performed quite well. Apart from bloat, there is no benchmark where greedy search found a substantially better program than the original. Bloat was the only benchmark where the original program was significantly worse than the JBrainy's optimized version.

Second, for avrora, fop, and lusearch, it does not seem like the choice of collection matters that much. For avrora, all variants stay within 1% of the original program.

All in all, we get a surprising result: it seems like collection choice does not matter a lot in 4 out of our 5 benchmarks. Now, related work did manage to find

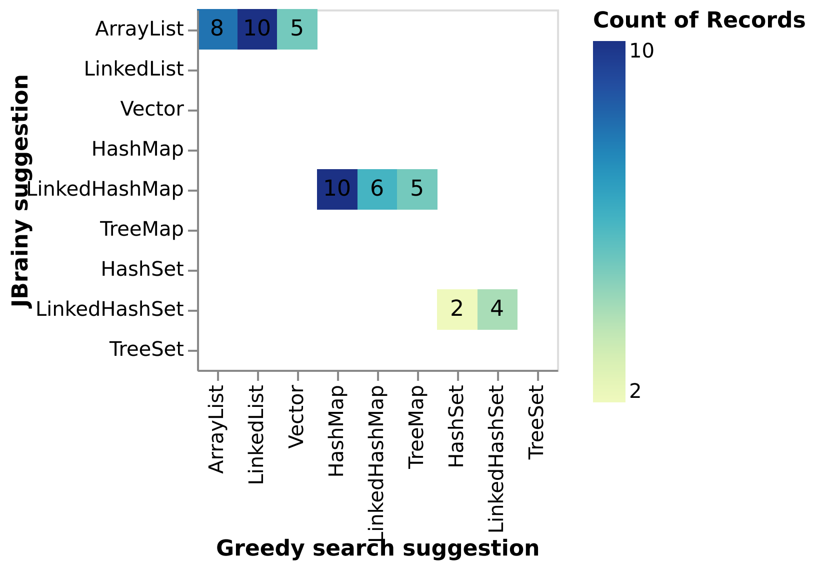


Figure 5: Comparison of Brainy's suggestions with greedy search suggestions

inefficiencies. Chameleon improved fop's performance by 20%. How did they do it?

We can answer this question by looking at the collections they have considered. In this work, we considered nine popular collections: ArrayList, LinkedList, Vector, HashSet, TreeSet, LinkedHashSet, HashMap, TreeMap, and LinkedHashMap. Chameleon [SVY09] and other tools like CoCo [Xu13] and CollectionSwitch [CA18], use different collections and techniques. These techniques mostly improve memory usage, but we suspect that they also improve running time by decreasing the time spent doing garbage collection. Now, is is unclear as to which technique improves running time the most, and when.

6.1 Lazy collections

Chameleon introduces a lazy ArrayList which does not allocate the array when it is created, but on the first update. They also introduce Lazy Sets and Lazy Maps, which do the same. Chameleon suggests to replace any list which remains empty by this LazyArrayList. They report that bloat allocates many LinkedList s which remain empty. Even when empty, a `LinkedList$Entry` object is allocated at the head of the linked list. Shacham et al. could save 20% of the heap by using LazyArrayList instead. Interestingly, Chameleon did not use LinkedHashMap, so it could not use the same optimization that JBrainy suggested. It is probably possible to optimize bloat further by combining both optimizations.

Collection	Number of wins
ArrayList	1731
LinkedHashSet	1722
LinkedHashMap	1713
HashMap	18
HashSet	6
TreeSet	3

Table 1: Number of wins for each collection in the training data

6.2 Array-based maps

Costa [CA18], and Shacham [SVY09] use array-based maps, which are simply arrays of entries. `get` does not run `hashcode`, it linearly searches through the map. Array-based maps are more memory efficient, and can be more efficient if the map remains small. Österlund [OL13] and Costa use collections which start by allocating an array-based map, and switch to a regular HashMap once the map reaches a certain size. Costa et al. replaced many maps in lusearch by their AdaptiveMap, since these contained less than 20 elements, resulting in a 15% improvement of running time.

6.3 Hash-based lists

Österlund and Costa present variants of ArrayList which also maintain a hash table. That way, the hash table speeds up calls to `contains`. Costa report that their AdaptiveList (which switches between hash-based lists and array-based lists) could speed up h2 by 6%, but that it did not improve fop's running time.

6.4 Collection tuning

The constructors of both ArrayList and HashMap have a parameter which sets how big the underlying array should be. Setting this size prevents resizing the collection as it grows, avoiding copies of the data with every resize. Shacham et al. [SVY09] report using this method on fop, and report 18% speedup. However, they used both collection tuning and array-based maps in this case, and it is unclear which technique was most effective.

7 Threats to validity

Figure 5 shows that JBrainy's classifier always make the same choices. This is explained by a very strong imbalance in the training data. The table below shows the number of wins per collection type in the training data.

Despite our efforts to select a balanced dataset, it didn't happen. There are four possible reasons for this.

Seed selection To ensure the training data it uses is balanced, Brainy uses seed selection, which we discussed in section 2.2. We can see in table 1 that this method was not effective for JBrainy.

The reason for that discrepancy lies in the disagreement of the naive benchmarking we perform for seed selection, compared to the use of JMH for measuring the time benchmarks take precisely. The first method runs the benchmark 30 times, and prints the value of the collection to `/dev/null`. In practice, the two methods often disagree.

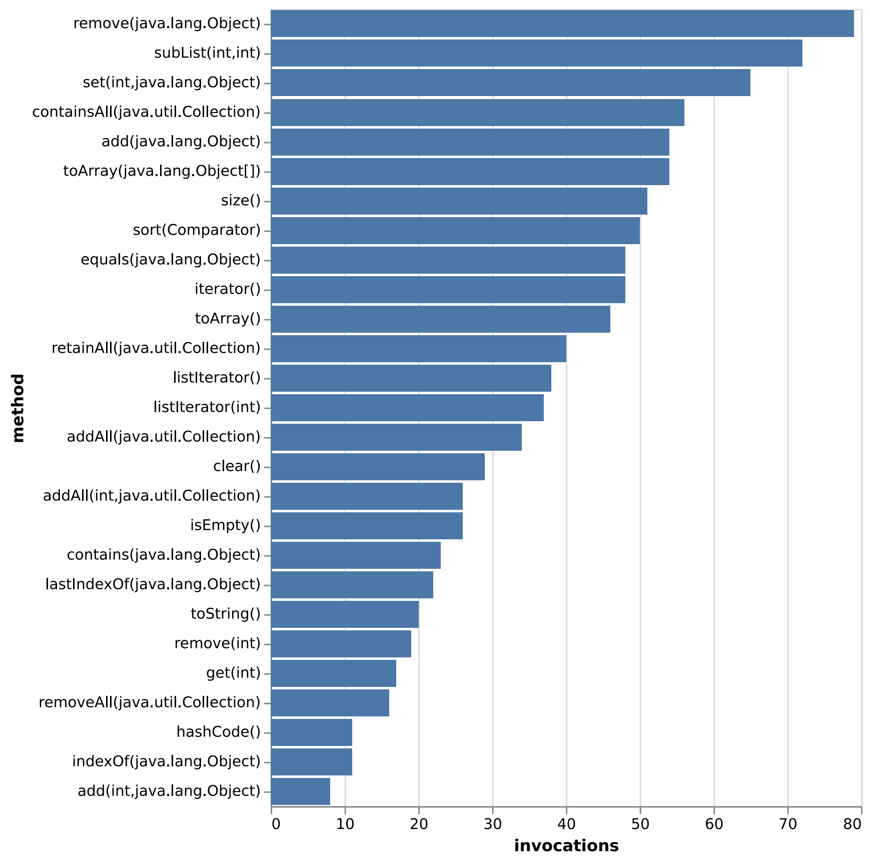


Figure 6: Count of invocations for a JBrainy benchmark

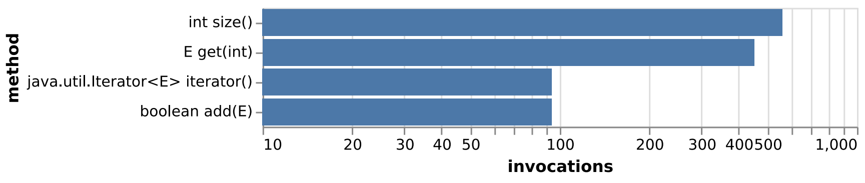


Figure 7: Count of invocations for an allocation site with roughly 1200 invocations

Bias in generated plans One possible reason for the large imbalances in training data could be that JBrainy's model of collection usage is not realistic. For a comparison between synthetic benchmarks and actual traces, figure 6 and 7 show the number of invocations for each method, for traces of similar size. Figure 6 shows a synthetic benchmark, while figure 7 shows data from a trace of fop. The synthetic benchmark calls 27 methods, between 10 and 80 times, while the actual trace shows that only 4 methods are called. Two are called approximately 100 times, and two are called more than 400 times. It seems that real method calls are more focused on a few methods, called many more times. For the busiest allocation site in fop, only 5 methods are called, but the number of invocations lie between 58 and more than 250000.

The differences, in hindsight, seem obvious. However, a similar model did work in the original Brainy study. It could be that this model doesn't work very well for Java, or that LinkedHashSet and LinkedHashMap are particularly adapted to workloads the model produces. To investigate this question, we compared seed selection with the Brainy model of collection usage, with traces generated by sampling markov chains. The markov chains were inferred using traces from avrora, fop, and lusearch. Figure 8 shows the progress for each data-structure. We see that the Brainy model does favor ArrayList, LinkedHashMap and LinkedHashSet, while the Markov model struggles to find benchmarks in which one collection runs at least 5% faster than the others.

Element types We chose to store integers in the collections we benchmark, but real programs manipulate more complicated objects. For example, our tool does not model the cost of methods like `hashcode`, `equals`, or `compareTo`. We suspect that using only integers could be detrimental to some collections.

Generation of method arguments Our model generates uniformly distributed random numbers to serve as arguments to methods, which might not show some collections' strengths. For example, a LinkedList typically performs better than an ArrayList when elements are inserted at the beginning: the ArrayList reallocates the whole array, which is expensive. Our model selects a uniformly distributed

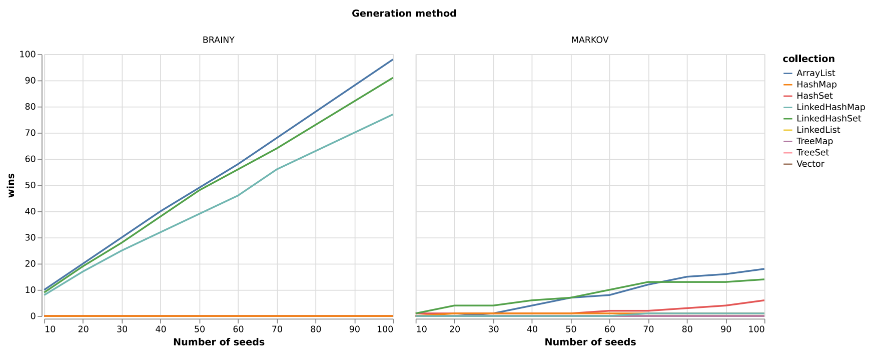


Figure 8: Wins relative to number of seeds for each collection, for two generation methods

index between 0 and the size of the list, so insertions at the beginning can happen, but the model doesn't try to prepend elements repeatedly.

8 Related work

Static collection replacement In 2009, Shacham et al. presented Chameleon [SVY09], a tool which used traces, heap information, and hand-written rules to select the collections to use. Artemis [Bas+18], presented in 2018, used a genetic algorithm to optimize the program directly, it does not try to build a cost model at all.

Our work is a direct port of Brainy, the tool presented by Jung et al in 2011 [Jun+11]. In contrast with Chameleon, we do not attempt to reduce memory consumption and do not use heap information. Instead of hand-written heuristics, we use a machine learning model. We also do not use the same collections, and restrict ourselves to the collections available in the Java Collections Framework.

Artemis is different from our work, since it does not try to build a cost model: The fitness function of its optimization algorithm runs the variant program directly. Basios et al. report that Artemis takes 3.05 hours on average to optimize a program. JBrainy goes faster, but Artemis was more effective: it improved fop and avrora's running time by approximately 5%.

Adaptive collections Xu [Xu13] and Österlund et al. [OL13] presented collections that can instead switch implementation, depending on their usage. Österlund et al. present lists which can switch between an array and a hashmap, and use a state machine to decide when enough calls to `contains` have been performed that it is worth switching. CoCo [Xu13], on the other hand, minimizes copies by spreading the data across several collections, elements are moved between collections on demand. Costa et al. presented CollectionSwitch [CA18] in 2018, which builds on these two works: They present smart constructor which select which type to instantiate based on monitoring of existing collections, and use adaptive lists similar to those presented by Österlund et al. in their tool. The key difference between our work and these adaptive collections is that they optimize at runtime. JBrainy does use runtime information, but the replacements are static.

9 Conclusion

In this paper, we presented our port of the Brainy approach to Java programs. We reported on the challenges that porting Brainy to Java entails. We conclude that porting Brainy to Java requires knowing about bytecode generation (to generate benchmarks), micro-benchmarking and interfacing Java with tracing libraries.

We evaluated JBrainy on five DaCapo benchmarks: avrora, bloat, chart, fop, and lusearch. Our results show that JBrainy can speed up programs for a fraction of the cost of greedy search, but we notice that JBrainy was less effective than the state of the art. Here, we list some directions for future work.

Collections available Since greedy search was not as effective as the state of the art, we suspect that the set of collections available to JBrainy could be expanded. We expect that collections like Lazy ArrayList, or ArrayMap were important in the improvements obtained by Chameleon, CoCo and CollectionSwitch, and could improve JBrainy's performance. Adding these collections to the options JBrainy and greedy search can use would allow for better comparisons with the state of the art.

Imbalanced training data We suspect that JBrainy's benchmark generation model is biased, considering the large imbalances of its training data. In future work, we plan to improve this situation by allowing the user to specify hints to guide the synthesis of benchmarks, so that the resulting training data is more balanced. We do not know yet which factor is most important between the distribution of methods, the properties of the element type, or the distributions of arguments.

Importance of CPU architecture Jung et al. [Jun+11] reported that Brainy's suggestions were different for different CPU architectures, is it the same for Java programs? We plan to compare JBrainy's suggestions on different CPU architectures, to see to which extent the importance of CPU architecture applies to Java.

10 Acknowledgements

This work was partially supported by Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP), funded by Knut and Alice Wallenberg Foundation.

References

[Bas+18] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. “Darwinian data structure selection”. en. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. Lake Buena Vista, FL, USA: ACM Press, 2018, pp. 118–128.

[Bla+06] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khan, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J Eliot B Moss, Aashish Phansalkar, Darko Stefanovic, and Thomas VanDrunen. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. en. In: (2006), p. 22.

[Bla+08] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. “Wake up and smell the coffee: evaluation methodology for the 21st century”. en. In: *Communications of the ACM* 51.8 (Aug. 2008), pp. 83–89.

[CA18] Diego Costa and Artur Andrzejak. “CollectionSwitch: a framework for efficient and dynamic collection selection”. en. In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization - CGO 2018*. Vienna, Austria: ACM Press, 2018, pp. 16–26.

[Jun+11] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. “Brainy: effective selection of data structures”. In: *ACM SIGPLAN Notices* 46.6 (June 2011), pp. 86–97.

[Kul07] Eugene Kuleshov. “Using the ASM framework to implement common Java bytecode transformation patterns”. en. In: (2007), p. 7.

[OL13] Erik Osterlund and Welf Lowe. “Dynamically transforming data structures”. en. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE).* Silicon Valley, CA, USA: IEEE, Nov. 2013, pp. 410–420.

[SVY09] Ohad Shacham, Martin Vechev, and Eran Yahav. “Chameleon: Adaptive Selection of Collections”. en. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2009), p. 11.

[VR+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. “Soot: a Java bytecode optimization framework”. In: *CASCON First Decade High Impact Papers*. CASCON ’10. USA: IBM Corp., Nov. 2010, pp. 214–224.

[Xu13] Guoqing Xu. “CoCo: Sound and Adaptive Replacement of Java Collections”. In: *ECOOP 2013 Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–26.