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

Noric Couderc
Department of Computer Science
Lund University
Lund, Sweden
noric.couderc@cs.lth.se

Emma Söderberg
Department of Computer Science
Lund University
Lund, Sweden
emma.soderberg@cs.lth.se

Christoph Reichenbach
Department of Computer Science
Lund University
Lund, Sweden
christoph.reichenbach@cs.lth.se

ABSTRACT

Software developers use collection data structures extensively and are often faced with the task of picking which collection to use. Choosing an inappropriate collection can have major negative impact on runtime performance. However, choosing the right collection can be difficult since developers are faced with many possibilities, which often appear functionally equivalent. One approach to assist developers in this decision-making process is to micro-benchmark datastructures in order to provide performance insights.

In this paper, we present results from experiments on Java collections (maps, lists, and sets) using our tool JBrainy, which synthesises micro-benchmarks with sequences of random method calls. We compare our results to the results of a previous experiment on Java collections that uses a micro-benchmarking approach focused on single methods. Our results support previous results for lists, in that we found `ArrayList` to yield the best running time in 90% of our benchmarks. For sets, we found `LinkedHashSet` to yield the best performance in 78% of the benchmarks. In contrast to previous results, we found `TreeMap` and `LinkedHashMap` to yield better runtime performance than `HashMap` in 84% of cases.

CCS CONCEPTS

• **Software and its engineering → Software performance**; *Software libraries and repositories*; *Reusability*.

KEYWORDS

collections, performance, Java

ACM Reference Format:
Noric Couderc, Emma Söderberg, and Christoph Reichenbach. 2020. JBrainy: Micro-benchmarking Java Collections with Interference (Work in Progress Paper). In *Proceedings of ICPE 2020: ACM / SPEC International Conference on Performance Engineering (ICPE 2020).* ACM, New York, NY, USA, 5 pages. https://doi.org/DOI~to~be~determined

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 [9].

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.

Considering the Java collections API, it is relatively rich, offering 28, 16 or 25 operations on lists, sets, and maps, respectively. Correspondingly, each collection can be utilised in many different ways: for instance, developers might initialise one of the collections and then only perform lookups, or they might repeatedly update the datastructure and only rarely perform lookups.

Consequently, 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. [3], in which the authors study Java collections from the standard library and from third-party libraries by examining one operation at a time. Multi-operation profiles are the basis for the Brainy approach [7], 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

ICPE 2020, Edmonton, Canada,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/DOI~to~be~determined

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 4, review related work in Section 5, and conclude in Section 6.

2 METHODS

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

2.1 Single-Operation Profiles

Costa et al.'s CollectionsBench system [3] 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.

2.2 Multi-Operation Profiles

To account for the possibility of interference between different operations, Jung et al.'s Brainy system [7] explores a single 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 allows 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. However, the price that Brainy pays for this approach is that it is unlikely to generate microbenchmarks that correspond to CollectionsBench-style single-operation profiles, even though such profiles arguably correspond to practically relevant usage patterns. For example, assuming uniform distribution, the probability of generating ten list additions in a row is only $\frac{1}{3\times10^{14}}$.

2.3 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 [8]: 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 generated benchmarks, our approach favours no particular methods, because all methods have an equal probability of being favoured in one benchmark.

3 EXPERIMENTS

To explore the utility 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.

3.1 Experimental Setup

Our experiments focussed 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 [4] for compatibility with CollectionsBench and to simplify our evaluation methodology [1].

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. We found no significant difference between these configurations, so in the following we only report on experiments and results aggregated over all configurations.

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 our JBrainy system, 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 microbenchmark, 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*, mapping to the most popular collections, as 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 the median speedups for each tool.

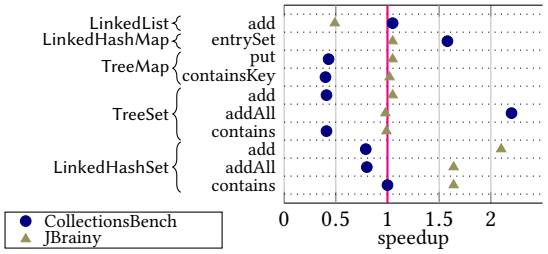

Figure 1: Comparison between speedup predictions by CollectionsBench and JBrainy for various operations

3.2 Results

Figure 1 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 2 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 $\approx$ 14.78).

In the case of lists, `LinkedLists` are on average 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 ($s \approx 1.28$), 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 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

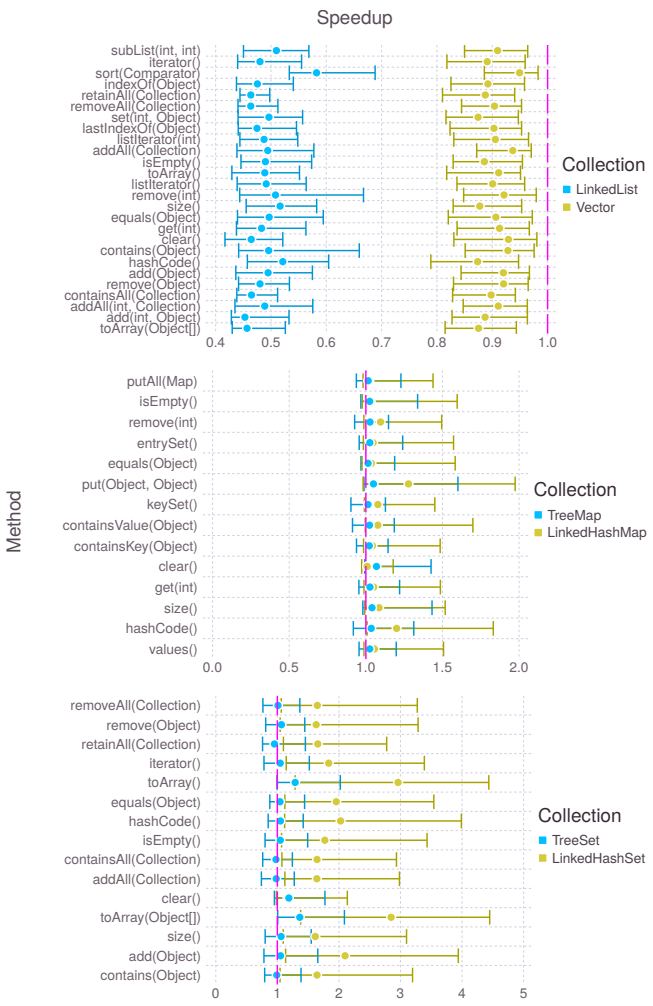

Figure 2: Median speedup of various collections compared to baseline (in magenta), with 25% and 75% quantiles

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 (i.e., it depends on the *capacity* of the table), whereas for a `LinkedHashSet`, the iteration only goes through the set's internal linked list of the set elements (i.e., only depends on the actual list size). 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

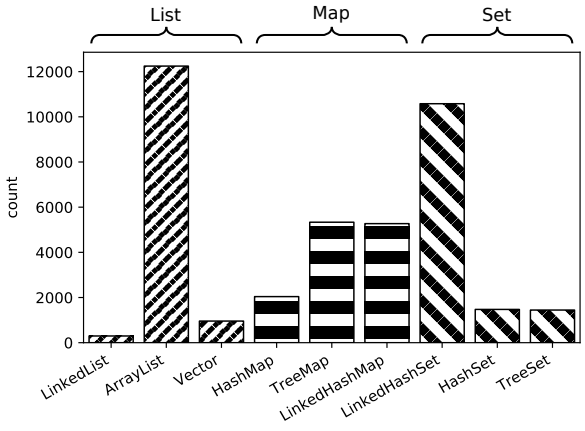

Figure 3: Count of fastest benchmarks depending on the collection class used.

node, while clearing (linked) hash maps requires iterating over all hash buckets.

For sets, Costa et al. focus on third-party alternatives to `HashSet` [3], 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 in the CollectionsBench study, 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 vast 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 [3], can outperform more popular alternatives when the benchmark run involves calling many different methods on the object.

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 recommendations by exploring their impact on the performance of existing software.

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. [9] 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 [10] 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 [2] that improves over CoCo by utilising performance models generated from single-operation profiles [3], for dynamic collection class selection instead of hand-coded rules.

Similar ideas have also been explored for C++ [7], though research in automatic datastructure selection dates back further [5].

6 CONCLUSIONS

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 [7]. Using JBrainy and its novel Pólya profiles, we have run an initial performance evaluation experiment following the setup of the CollectionsBench study [3]. 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 many benchmarks.

We find these initial results encouraging and see several directions for future work. As an immediate next step we plan to include the third-party collections used in the CollectionsBench study in our work, to search for further insights. In addition, we plan to explore various threats to validity, especially by validating the recommendations from JBrainy on real-world software.

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

[1] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, and et al. 2008. Wake up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Commun. ACM* 51, 8 (Aug. 2008), 8389. https://doi.org/10.1145/1378704.1378723

[2] D. Costa and A. Andrzejak. 2018. Collectionswitch: A framework for efficient and dynamic collection selection. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 16–26.

[3] D. Costa, A. Andrzejak, J. Seboek, and D. Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering - ICPE '17*. ACM Press, 389–400.

[4] D. E. D. Costa, C. Bezemer, P. Leitner, and A. Andrzejak. 2019. What's Wrong With My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering* (2019).

[5] S. M. Freudenberger, J. T. Schwartz, and M. Sharir. 1983. Experience with the SETL optimizer. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 1 (1983), 26–45.

[7] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. 2011. Brainy: Effective Selection of Data Structures. *SIGPLAN Not.* 46, 6 (2011), 86–97.

[8] Hosam M Mahmoud. 2003. Pólya urn models and connections to random trees: a review. *Journal of the Iranian Statistical Society (JIRSS)* (2003).

[9] O. Shacham, M. Vechev, and E. Yahav. 2009. Chameleon: Adaptive Selection of Collections. *SIGPLAN Not.* 44, 6 (June 2009), 408–418.

[10] G. Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP13)*. Springer-Verlag, Berlin, Heidelberg, 126.