LUND UNIVERSITY

On the Challenges of Software Performance Optimization with Statistical Methods

Couderc, Noric

2023

Document Version:
Publisher's PDF, also known as Version of record

Link to publication

Citation for published version (APA):
Couderc, N. (2023). *On the Challenges of Software Performance Optimization with Statistical Methods*. [Doctoral Thesis (compilation), Department of Computer Science]. Department of Computer Science, Lund University.

Total number of authors:
1

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Download date: 27. Sep. 2024

On the Challenges of Software Performance Optimization with Statistical Methods

Noric Couderc

Doctoral thesis, 2023

Department of Computer Science
Lund University

ISBN 978-91-8039-691-2 (electronic)
ISBN 978-91-8039-692-9 (print version)
ISSN 1404-1219
Dissertation 72, 2023
LU-CS-DISS 2023-03

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, 2023

© *2023 Noric Couderc*

Abstract

Most recent programming languages, such as Java, Python and Ruby, include a collection framework as part of their standard library (or runtime). The Java Collection Framework provides a number of collection classes, some of which implement the same abstract data type, making them interchangeable. Developers can therefore choose between several functionally equivalent options. Since collections have different performance characteristics, and can 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 selecting their collections.

In this thesis, we consider the problem of building automated tools that would help the programmer choose between different collection implementations. We divide this problem into two sub-problems. First, we need to measure the performance of a collection, and use relevant statistical methods to make meaningful comparisons. Second, we need to predict the performance of a collection with as little benchmarking as possible.

To measure and analyze the performance of Java collections, we identify problems with the established methods, and suggest the need for more appropriate statistical methods, borrowed from Bayesian statistics. We use these statistical methods in a reproduction of two state-of-the-art dynamic collection selection approaches: CoCo and CollectionSwitch. Our Bayesian approach allows us to make sound comparisons between the previously reported results and our own experimental evidence. We find that we cannot reproduce the original results, and report on possible causes for the discrepancies between our results and theirs.

To predict the performance of a collection, we consider an existing tool called Brainy. Brainy suggests collections to developers for C++ programs, using machine learning. One particularity of Brainy is that it generates its own training data, by synthesizing programs and benchmarking them. As a result Brainy can automatically learn about new collections and new CPU architectures, while other approaches required an expert to teach the system about collection performance. We adapt Brainy to the Java context, and investigate whether Brainy's adaptability also holds for Java. We find that Brainy's benchmark synthesis methods do not apply well to the Java context, as they introduce some significant biases. We propose a new generative model for collection benchmarks and present the challenges that porting Brainy to Java entails.

Acknowledgements

I thank Christoph Reichenbach and Emma Söderberg for being excellent supervisors. They are talented, driven, and supportive people, and provided a lot of insight and help.

I thank the members of the SDE group at Lund University, for the stimulating conversations, about anything and everything. I thank Görel Hedin for her support and wisdom. I thank Alexandru Dura for his intelligence and honesty. I thank Momina Rizwan for reminding me of all the meetings I had forgotten, and Faseeh Ahmad for his advice about ice cream. I thank Idriss Riouak for challenging me when we went bouldering, and Anton Risberg Alaküla for asking me absurd questions.

Je remercie ma mère, Dominique Couderc, pour son soutien indéfectible, et pour m'avoir inspiré par son courage et sa persévérance. Je remercie mon père, Jean-Jacques Couderc, pour avoir, depuis toujours, stimulé ma curiosité et mon intérêt pour les sciences. Je remercie ma sœur, Camille Couderc pour sa simplicité, et son ouverture d'esprit. Ces personnes m'ont appris que les sciences, les arts et les sports sont trois composantes d'une vie riche, et qu'à ce titre, elles méritent tout notre respect et toute notre attention.

Popular Summary

If you have ever used a computer, you probably noticed something: In a way, computers are really fast, but sometimes, they are really slow. Why is it that video games get more impressive every year, while a simple web-page can stall for seconds, displaying a frustrating spinning wheel?

This question is at the heart of *performance engineering*. Everything a computer does takes at least two resources: time, and memory. Since the very invention of the computer, engineers have worked to do things more efficiently, that is, using less time and less memory. When they are successful, we get exciting games, when they fail, we get… well, frustrated.

One way to make programs faster is to look at *collections*. When programmers want to group some data together, they use a collection. One example of collection is the list of contacts in a phone.

Each collection supports a few operations we can use in programming, to modify it. We can add to a collection, delete from it, and search for certain values in the collection. In the last fifty years, people have come up with different types of collections, which have different performance properties. For some, inserting is very cheap, but deleting is expensive. For some others, deleting is cheap, but it's search that is expensive.

So, if we want programs to be fast, we need to consider *how* a collection is used, and choose carefully. Software developers can think very hard about what collection to use, but because code is complicated, choosing wisely is difficult. The program still works, but it is slower than intended.

In this thesis, we try to speed up programs by suggesting to the developer which collections they should use, based on how they use them. We try to build a tool akin to a "code spellchecker", which will suggest fixes to the developer's code, so that the program runs faster. We focus on the Java Programming language, because it is very popular. To build such a tool, we need to answer a couple of questions.

The first one seems trivial, but it is not: **How do we even measure how long a program takes?** We can take a stopwatch and measure the time it takes to run the program. The problem is that computers are very complicated, so if we measure twice, we'll get different numbers. Indeed, there are so many things influencing our results! Worse, if we measure on another machine, we might get *completely* different numbers! A significant part of this thesis is about what influences the execution time of a program, and how to make meaningful comparisons between running times.

The second question is: **Can we predict the performance of a collection?** Indeed, if we want a tool that suggests option A over option B, it needs to have a rough idea of how long A and B would take to run. The tricky part is that we want to know that *without* running the programs, because running a program can

take seconds, minutes, hours, or days. Therefore, we would prefer the computer to *guess*.

To be able to guess, the computer needs to *learn*. This our third question: **How can we learn effectively about collections' strengths and weaknesses?** In this thesis, we ask the computer to *generate* artificial (short) programs, run them, and learn from them.

Here's a metaphor. To learn about the strength and weaknesses of collections, we'll build a "race-track" for collections. This race-track is a list of operations (insert this, delete that, search for something) in sequence. Then, we organize a race. We try different collections on the same race-track, to see how well they do. Usually, people might write these by hand. Small programs like these are called a *micro-benchmark*. Instead, we have the computer generate them. It makes a *lot* of them.

So, how well does this work? In our case, we run into a problem: some collections beat the competition in many, many cases. That sounds great: Then, why not use those all the time? It turns out that when we plug those collections into some real programs (that people have written), we do not observe significant differences. So in practice, the tool we've built is not effective at optimizing those programs.

We finish this thesis by looking at other approaches that people have tried in the past. We investigate how some real programs use collections, and try to reproduce the results that others have found earlier. We see that even if their methods worked when they published their studies, they do not seem to work very well anymore.

Contributions of the Author

This thesis is a compilation consisting of an introduction, and four papers.

Peer-reviewed Publications by the Thesis Author

Paper I

Noric Couderc, Emma Söderberg, and Christoph Reichenbach. "JBrainy: Microbenchmarking Java Collections with Interference (Work in Progress Paper)". In: *ICPE '20: 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.

Paper II

Noric Couderc, Christoph Reichenbach, and Emma Söderberg. "Performance Analysis with Bayesian Inference". In: *ICSE-NIER '23: Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results*. 2023

The thesis author suggested the article, did all of the technical work, and is the main author of the paper.

Paper III

Noric Couderc, Christoph Reichenbach, and Emma Söderberg. "Classification-based Static Collection Selection for Java: Effectiveness and Adaptability". In: *EASE '23: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2023*. June 2023

The thesis author implemented most of the system described, and wrote the main argument, which has however been significantly edited by secondary authors

Drafts

Paper IV

Automatic Collection Selection for Java: Comparing Static Approaches with Adaptive Collections.

To be submitted.

The thesis author implemented most of the system described, and wrote the core argument, with revisions from the co-authors.

Reponsibilities

The table below indicates the responsibilities the author had in each paper.

Paper	Writing	Concepts	Implementation	Evaluation
I	1/4	3/4	4/4	4/4
II	3/4	4/4	4/4	4/4
III	2/4	3/4	3/4	4/4
IV	3/4	4/4	3/4	4/4

Contents

Introduction **1**

I The Collection Selection Problem **3**
1 Research Questions . . . 4
2 Outline . . . 4

II Background **5**
1 Collection Type . . . 5
2 Collection Usage, Program and Input Data . . . 9
3 Hardware . . . 10
4 Java Virtual Machine . . . 11
5 Summary . . . 15

III Related Work **17**
1 Target Programming Language . . . 18
2 Performance Metrics . . . 18
3 Replacement Strategy . . . 18
4 Collections and Optimizations . . . 21
5 Evaluation . . . 22
6 Comparison of Collection Selection Tools . . . 23
7 Summary . . . 24

IV Machine Learning and Collection Performance Prediction **27**
1 Architecture . . . 28
2 Modeling Collection Usage . . . 29
3 Predicting Collection Performance . . . 31
4 Summary . . . 36

V Evaluating Java Programs **37**
1 Design of Experiments . . . 37
2 Design of Experiments and Causal Inference . . . 39
3 Summary . . . 41

VI Analyzing Experimental Results **43**

1 Hypothesis Testing 43
2 Confidence Intervals 45
3 Bayesian Statistics 46
4 Linear Regression 47
5 Hierarchical Models 54
6 Summary 61

VII Contributions 63
1 JBrainy: Micro-benchmarking Java Collections with Interference (Work in Progress Paper) 63
2 Performance Analysis with Bayesian Inference 64
3 Classification-based Collection Selection for Java: Effectiveness and Adaptability 65
4 Automatic Collection Selection for Java: Comparing Static Approaches with Adaptive Collections 66

VIII Threats to Validity 69
1 Internal Validity 69
2 External Validity 71
3 Summary 72

IX Conclusions and Future Work 73
1 Collection Selection 73
2 Benchmarking and Experimental Design 74
3 Conclusions 75
4 Final Words 75

Included Papers 85

Included Papers 85

I JBrainy: Micro-benchmarking Java Collections with Interference (Work in Progress Paper) 87
1 Introduction 87
2 Methods 88
3 Experiments 89
4 Discussion 93
5 Related work 94
6 Conclusions and Future Work 95
7 Acknowledgements 95
References 95

II Performance Analysis with Bayesian Inference 99
1 Introduction 99
2 Example: Collection Selection in Java 100
3 Model with interactions 105
4 Discussion 107
References 108

III Classification-based Collection Selection for Java: Effectiveness and Adaptability 111
1 Introduction 111
2 Related work 113
3 Brainy 114
4 Porting Brainy to Java: Brainy4J 116
5 Effectiveness of Brainy4J 118
6 Adaptability of Brainy4J 124
7 Obstacles to Effectiveness 127
8 Threats to Validity 129
9 Discussion 130
10 Conclusions 131
11 Acknowledgements 132
References 132

IV Automatic Collection Selection for Java: Comparing Static Approaches with Adaptive Collections 137
1 Introduction 137
2 Related Work 139
3 Background: What Influences Execution Time? 141
4 Methods 148
5 Statistical Analysis 151
6 Results 155
7 Discussion 163
8 Threats to Validity 164
9 Conclusion and Future Work 165
References 166

Introduction

The Collection Selection Problem

From a distance, all software looks the same: it reads data, processes it and stores the results. This similarity can be misleading: even two programs that compute exactly the same result can be different. How can two programs that compute the same thing be different? The difference lies in a key property of computers: every action do has a cost. How two algorithms compute a result is important: what operations are performed, and in what order, both affect how much time, and how much memory the process takes. Since computers have limited resources and people have limited time, both are important aspects of software quality [Noa].

One way to make processing faster and storage more efficient is to look at how a program 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) [MSS10; Xu+10]. Mitchell et al. identify two causes for such bloat [MSS10]. First, developers sometimes lack knowledge on how data-structures work. Second, even when they don't, they might have limited knowledge of how their software is going to be used: making performance design decisions is difficult without knowing the context.

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 execution 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, I will explore techniques to help developers choose the right collections. Doing this requires to solve two problems: first, we need to design optimizations, and second, we need to evaluate their effect on programs. The design of optimizations, in my case, required to understand how different Java collections work, and their properties regarding the cost of different operations on these collections. The evaluation of optimizations requires to first apply the optimization to a program, and then compare the optimized program to the original program. Unfortunately, there are many factors that can influence measurements, so the evaluation must be planned carefully.

1 Research Questions

In the rest of this thesis, we will focus on two main research questions, and their sub-questions.

- How to make effective suggestions for collections to Java developers?
    - How do other tools do it?
    - How to predict a collection's performance?
        - How do we model collection usage?
        - How can we effectively model a data-structure's strengths and weaknesses?
- How to evaluate our optimizations on Java software?
    - How to design experiments for performance analysis?
    - How to analyse benchmark results?

2 Outline

This thesis countains nine chapters, including this one. Chapter II presents the different factors that influence execution time. Chapter III reviews the related work on collection selection. Chapter IV discusses using machine learning to predict collection performance. Chapter V discusses the design of experiments to measure the performance of Java software. Chapter VI discusses the statistical analysis of benchmark results. Chapter VII lists the contributions of this thesis. Chapter VIII discusses the threats to validity. Chapter IX describes the future work.

Background

To reduce the execution time of their programs, developers need to understand what factors influence it. This is what this chapter is about. I will review how collections influence execution time, but also how other factors influence it too.

I divide the problem in two families of factors: those who are *internal* to the program under study, and those who are *external*. Internal factors can be manipulated by editing the source code of the program: for example, *which* collection you use, and what you do with it.

External factors concern the *environment* in which the software is compiled and run. For example, hardware matters, so the CPU architecture and memory matters. Likewise, the compiler and its options influence performance, but so does its runtime. In the Java world, the runtime is the Java Virtual Machine (JVM): it is the infrastructure that supports the program running, including things like garbage collectors and JIT-compilers.

To keep track of these factors and their influences, I will use *graphical causal models* [PGJ16], informally called "causal graphs". In a causal graph each vertex represents a factor. When a factor causes another, they are connected by an arrow. Some factors can be marked as "unobserved", in which case, I will display them in a box. Unobserved factors are factors that we cannot observe directly.

Figure 1 shows one such graphs, which summarizes what I just said. It shows the effect of the Hardware, the JVM, the collection type and collection usage on execution time. The variable "Collection usage" is marked as unobserved, because this is something that we cannot easily measure without disturbing the execution time of the program.

1 Collection Type

This thesis is concerned with Java data-structures, which are roughly divided in three families: Lists, Sets and Maps. The Java Collections Framework defines interfaces for these three data types, which specifies methods which are mandatory

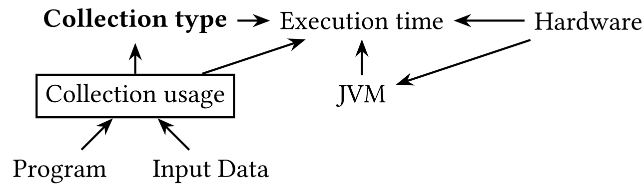


Figure 1: A causal graph of various causes of execution time for Java programs. Each arrow $X \rightarrow Y$ means X causes Y. Boxed nodes denote *unobserved* variables, which we cannot measure during our experiments. **Collection type** is the variable we will manipulate to reduce execution time.

to implement, and their types. I tried to find a source for these definitions, but did not find any.

The Java interface describes the type of methods of collections, but it does not specify their semantics, nor their performance characteristics. For each interface, there exist a number of collection implementations, which have different properties, regarding performance. We review the major collection implementations available in the Java Collections Framework.

1.1 Array-Based Lists

Array-based lists contain arrays which are re-allocated when the size of the list changes. *When* to resize is decided by the *capacity* of the list, which sets the initial size of the array. They offer access to any element of the array in constant time, while search is in linear time. Insertions are performed in constant time, but can take linear time in the worst case, where the full list needs to be copied to a new array. Likewise, deletions run in linear time in the worst case, where the full array must be copied to be kept contiguous in memory.

1.2 Linked Lists

Linked lists are implemented by having each element hold a pointer to the previous and the next element in the list. Access and search run in linear time in the worst case. Insertion at the front runs in constant time, since one just needs to create a new element which points to the first of the list. Deletion runs in constant time, but often requires search, and therefore runs in linear time in the worst case [Cor07].

Linked lists are slower than array-based lists, but they have the advantage that it is possible to take an existing data-structure, and add a linked list to it. An example of that is Java's LinkedHashMap. Which is a HashMap, with a linked list attached, which allows to speed up iteration over the map.

1.3 Hash Tables

Hash tables are used both to implement maps and sets. They consist of an array of slots and a hash function, which maps the key of interest into a slot. A convention in Java is that if two objects have an equal value, the hash function for these objects (called hashCode) should return the same result. However, the converse is not necessarily true. For example: the strings “BBBB” and “BBAa” have the same hashCode [1].

The *load factor* is a number that selects many filled slots there should be in the hash table before we resize it. When that happens, the hash table is *rehashed*, to spread the elements in the slots.

The properties of the hash function influence cost of operations on the hash table in two ways. First, the cost of hashing matters, since every insertion, deletion and search will require to hash. If the hash function runs in constant time, insertion, deletion and search then take constant time. Second, each hash function involves the probability of a *collision*. A collision is when the hash function maps the key to a slot that is already occupied.

When collisions happen, and there are several strategies for handling them. Cormen et al. [Cor07] present two: *chaining* and *open addressing*.

With chaining, instead of storing on single element in each slot, the hash table stores a linked list of elements. If there is a collision during insertion, you prepend the new element to the list. During search, if the slot contains a list, you iterate through the elements of the list. In the worst case, where every insertion caused a collision, a search amounts to a search in a linked list, which takes linear time.

With open addressing, instead of using a linked list, we transform the hash function so it can generate a *sequence* of slots for each key. During insertions, we try to insert in the first slot of the sequence, if there is an element there, we try the second slot, etc. During search, we try the first slot in the sequence, and if it does not match the key, we try the second slot in the sequence, etc. Deletion requires to store a special element in the deleted slot, to indicate that there was an element there (and therefore, possibly others later in the chain). Otherwise, during a lookup, we might conclude the element is not in the map, while there is, but it’s after the deleted element in the sequence.

Using open addressing, insertions and searches require to run the hash function several times. If this function is expensive, collisions can become very expensive.

Because hash functions are ran often, and because collisions increase the cost of operations, we would like to have hash functions which are fast but cause few collisions. It is difficult to satisfy both properties: reducing the probability of collisions requires more analysis, which takes time.

For example, take writing a hash function for strings. Analyzing the full string takes linear time. Therefore, to reduce the cost of hashing, we would be tempted to write a hash function which only looks at part of the string, for example, the

[1] I thank Louise Adolfsson for this information.

first few characters. But if we do that, we increase the probability of collisions, for example, Norway would collide with North Korea.

Sometimes, using comparison instead of hashing is a better option, which is the strategy used by the next data-structure we consider.

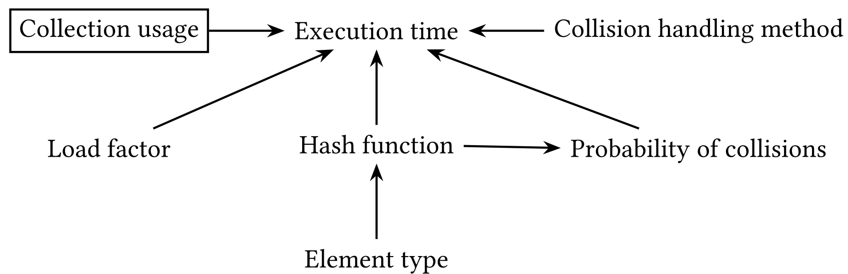


Figure 2: A causal graph for hash tables (like HashMap and LinkedHashMap)

1.4 Red/Black Trees

TreeMap and TreeSet are implemented using red/black trees, which are specialization of binary trees. Such data-structures rely on the *ordering* of the elements stored in the collection to reduce the cost of lookups.

Binary trees are composed of nodes. Each node contains the *key*, which is the element stored in the node. Then, three references to other nodes. The *parent* is the node above in the tree. The *left child* is a sub-tree which will contain nodes whose key is lower than the current key. The *right child* is a sub-tree which will contain nodes whose key is higher than the current key.

More formally, the structure of the tree is described in the *binary tree property*:

" Let x be a node in the binary search tree. If y in a node in the left sub-tree of x, then $\text{key}[y] \leq \text{key}[x]$. If y is a node in the right sub-tree of x, then$\text{key}[x] \leq \text{key}[y]$." ([Cor07])

Because of this property, we know that the cost of operations on the tree is proportional to the height of the tree h: insertions, deletions and lookups all run in $O(h)$ [Cor07].

Now, binary trees do not put constraints on the height of the tree, so it is possible to end up with an unbalanced tree. For example, inserting elements in a sorted order will create a tree that is equivalent to a linked list, so operations on the tree will have the same cost as for a linked list.

Red/black trees are binary trees which use a *balancing strategy*. They have different insertion and deletion procedures, to keep the tree balanced, so that it's height is the logarithm of the number of elements in the tree. Therefore, insertion, deletion and lookups take $O(\log(n))$ comparisons in the worst case [Cor07]. The cost of each comparison must however be considered too.

Figure 3: A causal graph for binary trees (like TreeMap)

Data-structure	Access	Search	Insertion	Deletions
Array-based list	$O(1)$	$O(n)/\Theta(n)^*$	$O(n)$	$O(n)$
Singly-linked list	$O(n)$	$O(n)/\Theta(n)^*$	$O(1)$	$O(1)$
Hash table		$O(1)$	$O(1)$	$O(1)$
Red/Black tree		$O(\log(n))$	$O(\log(n))$	$O(\log(n))$
Array-based map		$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Table 1: Complexity of operations for common data-structures [Cor07]. For array-based lists, I provide amortized costs. For linked lists, the cost of insertion assumes that we already have the position we want to insert / delete, which takes $\Theta(n)$ time to obtain. Asterisks denote complexity in the worst case.

1.5 Summary

I described how the operations performed on different collections, what they contain, and their size had an influence of execution time. In practice, most of the methods performed on collections will have constant, logarithmic, or linear complexity. Table 1 summarizes the cost of different operations on different data-structures.

2 Collection Usage, Program and Input Data

Of all the factors in this chapter, collection usage is probably the one that has the most impact on execution time. First, because each operation on the collection has a cost. Second, because it has an influence on the *size* of the collection, and the size of the collection influences the cost of subsequent operations, as shown in Table 1.

Unfortunately, tracing collection usage has an impact on execution time. That is why I marked it as unobserved: tracing collection usage will slow down the program. So we will either need to run it again to estimate the actual execution time, or hope that we do not disturb it too much.

Collection usage is influenced by the source code of the program, but also the inputs to the program. This point is important, because one can either create an offline tool that looks at the source code and suggests static changes, but then lack access to the inputs, or adaptive collections that can switch implementation at runtime, and have access to the input data, but add some overhead. I will come back to this issue in Chapter III.

3 Hardware

The "hardware" node in Figure 1 hides many details, but if we run the same program with the same input on two different machines, we should expect some variation in the resulting execution time. In this section, we list in more detail what hardware factors can influence execution time.

Instructions and Cycles An *instruction* is an operation performed by the CPU. CPUs have a clock which tick regularly, every tick is called a *cycle*. Modern computer performance is often described in *instructions per cycle* which denotes the number of instructions executed in one clock cycle. Some instructions take more than one cycle to execute, for example, reading from RAM takes hundreds of cycles. Two ratios have a key influence on execution time: instructions per cycle, and time per cycle. Several factors can affect both instructions per cycle, and time per cycle, as we will see below.

Clock Frequency The clock frequency sets the time spent per cycle, increasing the frequency decreases the time spent per cycle, at the cost of increasing the power consumption and temperature of the CPU. Modern CPUs can change the frequency automatically, depending on the temperature of the CPU.

Cache Behavior Modern CPUs all have a small amount of very fast memory, called a *cache*. Loading data from the cache takes very few cycles, compared to loading from the RAM, so using this cache appropriately speeds computations up.

One example of the effect of data-structures on the cache is the difference between array-based lists and linked lists. When iterating through an array-based list, the list operates on an array that is contiguous in memory. As a result the address of the next few elements is easy to predict, and the next elements will be loaded in the cache in advance. When iterating through a linked list, this is impossible: Each element comes with a pointer which tells *where* in RAM the next element is, and one need to load it from RAM. The cache is used less efficiently, and as a result, iterating through a linked list takes longer.

Branch (Mis)predictions CPUs have a *pipeline* of operations, which will be performed in sequence. If-statements are translated in conditional jumps, which

prevent the CPU from loading the next instructions to execute in the pipeline (since the result of the condition changes which code to execute next). To prevent this, modern CPU have a device called a *branch predictor* which will try to predict the result of the conditional *before* it's executed. If the branch predictor is correct, the next instructions are already loaded in the pipeline and run quickly. Otherwise, the CPU must discard the instructions, wasting cycles. As a result, increasing the number of branch mispredictions makes the program run slower.

Parallel Architectures One way to speed up computations is by doing things in parallel, rather than sequentially. As a result, hardware designers conceived multi-core architectures which allow several *threads* to run in parallel, sharing a central memory.

If each processor has its own memory (or cache), then the data that is in that memory can become outdated because of the actions of other processes running on other cores [HPAD07], a problem called *cache coherence*. The designer must therefore implement a protocol to either *update* or *invalidate* the values stored in the cache, when other processors make writes to memory, with a time penalty.

The presence of threads and cache coherence influence execution time in unpredictable ways. As a result, running once is not enough, maybe we just got lucky. To prevent this, we will run many *replications*: several runs of the same program, to obtain a distribution of results, which takes into account the uncertainty introduced by threads and disturbances by other processes.

Unfortunately, the speedup one could expect from parallelism is significantly influenced by the proportion of *sequential* operations: "The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used" [HPAD07].

Summary Figure 4 describes the effect of hardware behavior on execution time with a causal graph. CPU architecture and the code have an effect on the execution time. They act through branch mispredictions and cache misses. Some of these quantities are *unobserved*, they can't be measured without disturbing the execution time of the program. CPU temperature and other processes also influence execution time, and because experimenters cannot easily control them, they will observe noise in our measurements.

4 Java Virtual Machine

Compilers developers want to reduce execution times too, so they will often write compilers which try to exploit knowledge about hardware to reduce execution time. In the context of Java, the compiler is relatively simple, it is the Java Virtual Machine (JVM) which serves as an intermediate layer between code and hardware, it will perform dynamic optimizations. I will focus on Just-in-Time (JIT)

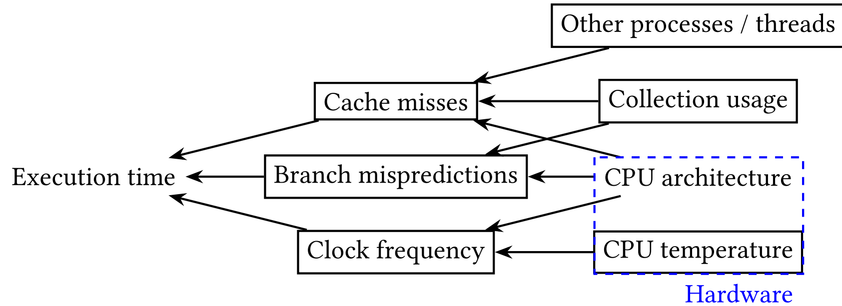


Figure 4: A causal graph describing what hardware properties influence execution time. The boxed nodes refer to unobserved variables, which we cannot measure without disturbing our measurements. The blue box refers to the "hardware node" in Figure 1

.

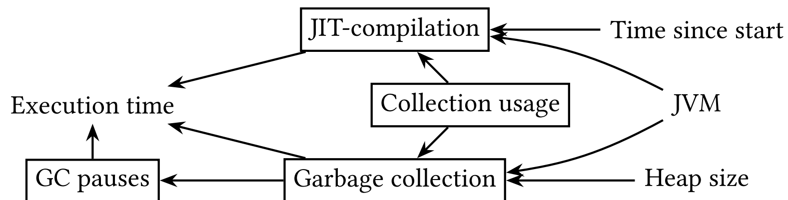


Figure 5: A causal graph describing how the JVM influence execution time. The boxed nodes refer to unobserved quantities

compilation, and garbage collection (GC). Figure 5 is the causal graph describing the influence of the JVM on execution time.

4.1 JIT-Compilation

One of the keys optimizations that the JVM performs is Just-in-Time compilation (JIT). The JVM will choose code that is "hot" (executed often) and will compile it to machine code, for quicker execution. As a result, running the same program several times *in the same process*, we expect to see its execution time decrease. Figure 6 shows an example of this behavior.

Performance engineers are often interested in so-called "steady-state performance". Which is the running-time to the right in Figure 6, that is, after JIT-compilation has finished optimizing. At that point, performance engineers expect the execution time to reach a plateau. To account for JIT compilation, experimenters run the program several times in the same process, [Bla+08; GBE07].

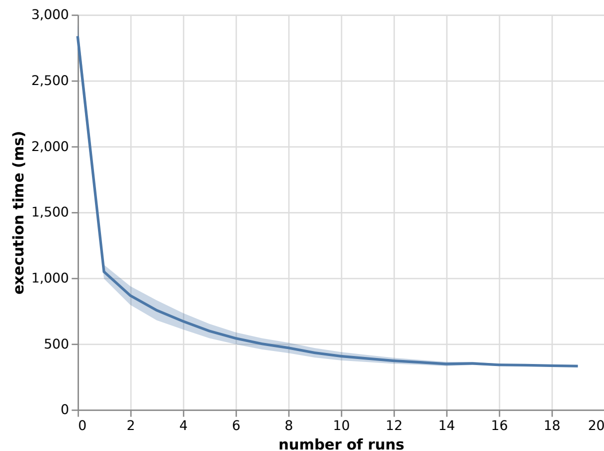


Figure 6: Execution time for the fop benchmark, relative to number of times it was run. Error bands show the standard deviation across 20 replications of the same experiment. The line shows the mean running time. We can see that the execution time of the benchmark drops significantly with warmup.

Determining *when* steady-state performance is reached has shown to be very tricky [Bla+08; Bar+17; Tra+22]. Figure 7 shows my hypothesis as to what the problem is. The problem is that JIT-compilation is *unobserved*: It is not something one can measure without disturbing the execution time of the program, so experimenters do not measure it. Researchers can control the number of in-process runs, and observe the execution times, but they cannot observe if JIT-compilation has completed or not. Therefore, they use execution time as a proxy for JIT-compilation: they try to *infer* the state of JIT-compilation based on execution time and the number of runs.

So far, scientists have focused on using cleverer statistical techniques to infer the state of JIT-compilation from the number of runs and execution time [GBE07; KJ13]. However, as we have seen, many *other* things influence execution time, so in practice, many benchmarks *never* reach a plateau [Bar+17; Tra+22]. Barett et al. for example, show how the execution times varies as garbage collection starts, or as the benchmark is migrated between cores [Bar+17]. I do not really have a solution yet, but I am skeptical we can talk about steady-state performance, as long as JIT-compilation remains unobserved.

4.2 Garbage Collection

Java is a managed language, which means that memory allocation and freeing are automatic. When starting, the JVM will claim a certain amount of memory, this is

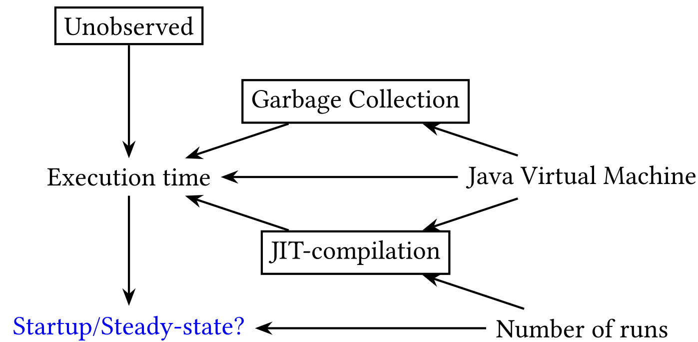


Figure 7: A causal diagram of how the JVM influences execution time. Unobserved variables are boxed, the blue node refers to a quantity that is inferred by the researcher. Researchers have tried to use execution time as a proxy for JIT-compilation, but the other influences on execution time disturb the measurements.

called the *heap*. This memory is claimed in the sense that other processes are not allowed to use it.

When the Java Programmer “creates” an object, a certain region of memory is assigned to that object. The JVM looks for a location in memory considered “free” (meaning that no important information is stored there), where information about that specific object will be stored. When an object is not used anymore, the space it occupied is considered free. The JVM automatically detects and marks memory to re-use, which is called *garbage collection* (As Nyström points out, the garbage collector doesn't *throw* memory away. So *recycling* would be a better metaphor [Nys21]).

Heap Graph How does the JVM keep track of what memory is not needed? In short, it tracks how objects reference other objects. We can see this structure as a graph. Objects are vertices and references are edges. We are interested in objects which become unreachable from certain special vertices in the graph, called the *roots*. A reachable object is called *live*.

Marking and Collection Garbage collection is divided in two phases: marking and collection. The marking phase is the process of marking reachable objects. Nodes that are not reachable are considered “garbage”, and will therefore be removed. Some garbage collectors, like the Serial GC, stop the entire program to mark the objects for removal, they are hence called *stop-the-world garbage collectors* [Nys21]. The time while the program is paused is called a GC pause. In the collection phase, the garbage collectors frees the unused (non-live) objects, meaning that it considers this memory as available for further allocations.

Latency and Throughput Every managed language pays a price for garbage collection [Nys21], and Java is no exception [Len+17]. Throughput is the total fraction of the time spent running user code instead of doing garbage collection. Latency (also called maximum pause time) is the *longest* continuous chunk of time where the user's program is completely paused. Different garbage collectors offer different trade-offs between those two different metrics, which is why there is a direct arrow between garbage collection and execution time in Figure 5. For batch jobs, throughput is more important, since it's the total amount of wasted time. For interactive applications, latency can be more important, otherwise the user interface might freeze while garbage is collected.

Summary The JVM influences execution time, mostly through two aspects. JIT-compilation speeds up programs by compiling java bytecode to machine code, while garbage collection occasionally slows down the program, reclaiming unused memory.

5 Summary

This chapter was a tour of how collections and other environmental factors, such as hardware architecture and the JVM, influence execution time.

In the Chapter III, I will review the related work about collection selection, in which we will see some optimizations that use some of the factors I mentioned here (e.g. load factor). In Chapter IV, we will also see how my work uses some consequences of collection usage to decide which collection to use (e.g. cache misses).

Related Work

To design a tool that optimizes running by changing which collections are used, there are several choices to consider. In this section, I will review what design choices were made in existing systems. I chose to compare Chameleon [SVY09], Brainy [Jun+11a], CoCo [Xu13], CollectionSwitch [CA18], Artemis [Bas+18], and finally, Cres [Wan+22]. 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 execution time.

There are systems which target other languages and use other definitions of efficiency, like energy usage. Perflint [LR09] improves the execution 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 axes.

- **Target Programming Language**: What language do the tool target?
- **Performance Metrics**: What is the tool trying to improve?
- **Collections and Optimizations**: What collections and optimizations can the tool use?
- **Replacement Strategy**: How does the tool perform replacements?
- **Evaluation**: How is the tool evaluated?
- **Comparison**: How is the tool compared to alternatives?

1 Target Programming Language

Chameleon, CoCo, CollectionSwitch and Cres target Java. Brainy targets C++, while Artemis supports both C++ and Java.

2 Performance Metrics

All of these works claim to improve *performance*. Here, performance is a trade-off between two resources: execution 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 execution times (by sharing the data across several collections). Lastly, Brainy and Cres only optimizes execution time.

3 Replacement Strategy

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

- **Granularity**: When should collections be replaced?
- **Prioritization**: Which collections should be replaced first?
- **Decision Making**: How to select the collection to use?

3.1 Granularity

First, a collection is born in a constructor call. We call the program location of that call an *allocation site*. Then, the collection receives method calls, we call the sequence of such method calls a *trace*. After it has been used, the collection is destroyed, either manually (as in C++), or automatically (in Java).

I could find three approaches to replace collections during the lifetime of collections: at the allocation site level, at the object level, or at the method level.

Replacements at the allocation site level replace the constructor calls, just like a software developer updated source code. Every collection that is created at the allocation site will still be of the same type. Brainy, Artemis and Cres, Chameleon did allocation site level replacements.

Replacements at the object level replace the constructor call by a factory method, which was the strategy that CollectionSwitch used. Each created collection carries some instrumentation to inform the factory about how they are being used. The factory can use that information to determine if a different type of collection should be created at the allocation site. As a result, two collections created at the same allocation site may be of different types.

Lastly, replacements can be done at the method level. In this case, created collections have the ability to switch their underlying implementation at any time. Each collection carries instrumentation for monitoring and decision making, and each method call may trigger a *switch*. This approach is also sometimes called *adaptive collections*. For CollectionSwitch, a switch triggers a copy to a new collection. For CoCo however, the data stored in the collection will be lazily copied over the new collection.

Static or Dynamic Replacements

I put the different replacement granularities in two families: static replacements (before running the program) and dynamic replacements (while the program is running). Static replacements replace collections at the allocation site level. Dynamic replacements may replace collections either at the object level, or at the method level. Both approaches 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]. 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.

Brainy, Artemis and Cres make static replacements. Chameleon makes static replacements but it can make dynamic replacements. Shacham et al report on the static replacements, and report that dynamic replacements were also effective, except in the case of the PMD benchmark, which was 6 times slower. CoCo and CollectionSwitch make dynamic replacements.

3.2 Prioritization

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

[1] management might disagree with that statement

Artemis and Chameleon take the most ambitious 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. Cres uses static analysis to identify locations where code could be optimized, in the code.

3.3 Decision Making

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. Cres also uses a hand-written model: they compare the complexity class of methods, and if two have the same complexity class, they specify an ordering: For example, we know that `put` on a LinkedHashMap should take longer than for a HashMap, because the former must maintain a linked list between elements. 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 , and uses neural networks to predict the best collection to use, given hardware performance counters (like branch mispredictions and L1 cache miss rates). CollectionSwitch 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.

CollectionSwitch must make decisions at run-time, and must therefore use a simpler model. Brainy being an offline tool, it can gather more expensive features.

Hand-written models of cost have shown to be effective, but what if the CPU architecture has such a strong impact of performance that you may want to use different collections? The CPU architecture was important in the case of Brainy: their tool considers recommends different collections for difference the CPU architectures. If the environment impact the effect of collection changes, using a learned model of performance has advantages. Likewise, if you add a new collection, the learned model of performance can inspect this new collection automatically, while a human expert has to update the hand-written model.

Not all tools have cost models. Artemis takes the minimalist-maximalist approach: 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.

4 Collections and Optimizations

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?

4.1 Collection Tuning

Several collections (e.g. ArrayList and HashMap) accept parameters that do not change their functional behavior, but change their performance (see Chapter II). ArrayList takes a capacity parameter, which is the initial size of the underlying array. For HashMap, the load factor specifies when the map's underlying array is full enough to justify a full copy of it. Some collection selection tools set these parameters, such as Artemis and Chameleon. Brainy, CoCo, and CollectionSwitch do not.

4.2 Lazy Collections

Sometimes, a collection might remain empty. Even when that is the case, Hashset and ArrayList 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.

For recent versions of the JDK (after version 8), this optimization is already implemented in standard collections. LinkedList uses lazy initialization since OpenJDK 7. ArrayList, HashMap, LinkedHashMap, HashSet, and LinkedHashSet support this optimization since OpenJDK 8.

4.3 Array-based Maps

When maps are small, it is 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 I 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”. Chameleon presented the same data-type (called SizeAdaptingSets and SizeAdaptingMaps). In the case of Chameleon, using ArrayMap was important in getting 13.79% of reduction in memory space used by the program FindBugs.

4.4 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 also uses this technique.

4.5 Multi-Interface Replacements

Some tools must replace a list by a list, some can replace it by a member of another abstract data type, for example a set. Chameleon can replace an ArrayList with a LinkedHashSet. Cres can replace an collection by another which implements another interface, for example, replace ArrayList by a HashSet. Cres automatically generates the surrounding code to make the replacement.

CollectionSwitch replaces classes by a class with the same interface, but they also provide a HashArrayList, which is essentially a wrapper for a HashSet, so the effect the same.

5 Evaluation

Different authors take different approaches for evaluating their collection selection tools. CollectionSwitch, CoCo, and Brainy use existing benchmarks, like the DaCapo benchmark suite [Bla+06b], but they select different benchmarks in the suite. Artemis and Cres run the test-suite of a number of selected repositories on GitHub.

I am not so fond of the latter approach. I can understand the appeal of picking GitHub repositories, which could be considered a more representative sample of the code developers actually write. In fact, we worked on DaCapo benchmarks, and I often wondered if there was really *any* performance left to squeeze out of them, after years of scrutiny by researchers. However, I doubt that the test suite of a program gives reliable information about its performance. I would suspect developers to generate small collections during tests, so they run quickly.

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.

6 Comparison of Collection Selection Tools

I used to think that comparing the effect of collection selection tools would be easy. After all, performance engineering has a clear definition of success: we want faster, less memory-intensive programs. Different tools were tested on different benchmarks, and they could choose different collections, but they *improved* performance on average. So I used to think we can just compare the numbers. I was wrong. We cannot compare these numbers directly, for several reasons.

First, different tools were tested on different benchmarks. CoCo, Chameleon and CollectionSwitch were tested on DaCapo benchmarks, but the set of benchmarks varies. Artemis was tested on five DaCapo benchmarks, but also on a corpus of popular GitHub projects. Only the fop benchmark was tested with every tool. Lusearch was the benchmark that was improved the most with CoCo and CollectionSwitch, but unfortunately, Chameleon was not tested on Lusearch.

Second, different studies measure and report different things. Chameleon does not state if they evaluated startup performance or steady-state performance, and does not report exact numbers, we have to read them from a plot. CoCo reports the median running time and its standard deviation. CollectionSwitch reports the mean speedup, for each benchmark. The authors remove results for which the improvement is not significant. Artemis measures the execution time of the test suite of a corpus of selected GitHub projects, as well as for a set of five DaCapo benchmarks. The authors report the median with 95% confidence interval.

Third, one may decide to ignore that studies different benchmarks and report slightly different statistics, and decide to compare the mean speedups between the studies. However, the context has also changed a lot between studies. Chameleon was released in 2009, CoCo was released in 2013, CollectionSwitch and Artemis were released in 2018. Between 2009 and 2018, CPU architectures, the JVM, and the the Java Collections Framework have evolved. For example, lazy collections were added in OpenJDK 7. If we tried to reproduce Chameleon today, it seems unlikely we would observe the same speedups. Because lazy collections are now standard, our *baseline* has changed.

What can one learn from these numbers, then? Most of the observed speedups are between 5% and 15%, even though they could be as high as 60%. Chameleon was the most effective at reducing execution time, while Artemis yielded the most modest improvements. Chameleon decreases execution times by between 8% and 60%. CoCo decreases execution times by 14.6% on average, but at the cost of +18.8% of memory space. CollectionSwitch decreases execution times by between 0 to 15%. Artemis improves the median of execution times by an average of approximately 4.6%, and memory consumption by an average of 4.6%. Their improvements range between 41.6% and 0.9%. For memory consumption, CoCo increased it, but other tools reduced memory consumption by roughly 10%. For completeness, Table 1 shows improvements reported by different studies. Remember that these numbers

measure different things (sometimes medians, sometimes means).

Brainy found substantial improvements, 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.

7 Summary

In this chapter, I have described the different design decisions that building a collection selection tool requires, reproduced below.

- **Target Programming Language**: What language do the tool target?
- **Performance Metrics**: What is the tool trying to improve?
- **Replacement Strategy**: How does the tool perform replacements?
- **Collections and Optimizations**: What collections and optimizations can the tool use?
- **Evaluation**: How is the tool evaluated?
- **Comparison**: How is the tool compared to alternatives?

I compared the approaches of Chameleon [SVY09], CoCo [Xu13], Collection-Switch [CA18], Artemis [Bas+18], and Cres [Wan+22].

I also included Brainy [Jun+11a], which targets C++, and not Java. Brainy can automatically adapt its replacement strategy to new collections, and considers the machine's CPU architecture when making decisions. That aspect was neglected by the works on Java, so I decided to port the Brainy approach to Java, which is the topic of the next chapter.

Benchmark	Tool	Execution time difference (%)	Memory usage difference (%)
Sunflow	Artemis	-2	-5
PMD	Artemis	-3	-2
Fop	Artemis	-5	-5
Avrora	Artemis	-5	-4
Xalan	Artemis	-5	-20
PMD	Chameleon	-9	0
Soot	Chameleon	-11	-5
Fop	Chameleon	-18	-8
Bloat	Chameleon	-29	-56
FindBugs	Chameleon	-54	-15
TVLA	Chameleon	-60	-55
Bloat	CoCo	-4	+81
Chart	CoCo	-9	+21
Avrora	CoCo	-11	+6
Fop	CoCo	-16	+2
Lusearch	CoCo	-34	0
Avrora	CollectionSwitch	0	-10
Fop	CollectionSwitch	0	-7
H2	CollectionSwitch	-6	-8
Bloat	CollectionSwitch	-12	-8
Lusearch	CollectionSwitch	-15	-5
bootique	Cres	-4.5	
mapper	Cres	-7.3	
incubator-eventmesh	Cres	-4.1	
google-http-java-client	Cres	-27.1	
light-4j	Cres	-5.2	
roller	Cres	-9.5	
IginX	Cres	-3.5	
sofa-rpc	Cres	-3.7	
Glowstone	Cres	-13.1	
dolphinscheduler	Cres	-5.3	
dubbo	Cres	-7.5	
iotdb	Cres	-6.3	

Table 1: Comparison of improvements provided by different tools, for different benchmarks. CollectionSwitch could optimize either execution time or memory, so the numbers refer to different runs.

Machine Learning and Collection Performance Prediction

In Chapter III, I said that there were two approaches to collection selection. CollectionSwitch and CoCo propose adaptive collections, which can dynamically switch between collections. Brainy, on the other hand, suggest static changes. Both approaches were effective on real-world programs.

Apart from their replacement strategies, tools also differ in their decision-making approach. CoCo, Cres and Chameleon use a hand-written model. While CollectionSwitch and Brainy learn parts of this model. CollectionSwitch used some amount of micro-benchmarking to learn a size threshold for which a list should be switched to a hash table. Brainy learned its full model, on and for the host machine.

Learning this model automatically makes the approach quite adaptable. First, if a new collection is added, updating the model to use this new collection is automatic. Second, if CPU architecture matters, then being able to learn the cost model on the target machine is valuable. This was the case in the original Brainy study.

Given that Brainy looked promising, we decided to port Brainy to Java. In this chapter, I will discuss two remaining research questions from the introduction:

- How can collection usage be modeled?
- Can we predict a data-structure's performance?

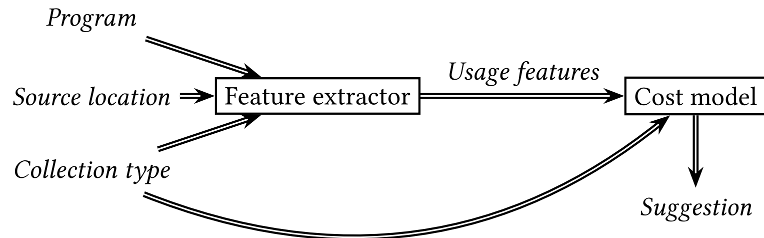


Figure 1: A diagram describing how to use Brainy4J. Nodes in italics represent data, boxed nodes represent programs, arrows represent data flow.

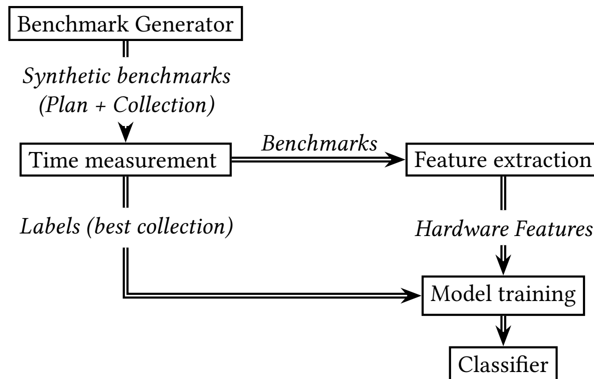


Figure 2: A diagram of the training process of Brainy4J. Nodes in italics represent data. Boxed nodes represent programs. Arrows represent data flow.

1 Architecture

Figure 1 describes how a user would use Brainy4J. The target program, source location and collection type are passed to a feature extractor, which instruments the target program, and runs it. The result is a number of usage features, which show how the collection is used. These features are passed to a cost model, which outputs a suggestion.

Brainy treats the problem of collection selection as a classification problem. What collections should one use, given the collection in place behaves in a certain way? Given the collection type, and usage features, Brainy tries to learn which collection would be a better alternative. Doing this requires training data.

Figure 2 shows how Brainy obtains training data for its cost model. Instead of relying on corpora of programs, Brainy synthesizes benchmarks. The synthetic benchmarks are tested with several different collections, to obtain the labels (the collection that yields the lowest execution times). Later, these benchmarks are

instrumented and run again, to obtain features on collection usage.

2 Modeling Collection Usage

Brainy's generative approach requires a model of how collections are *used.* In our work, we separated the generation of benchmarks in two steps. First, we generate a *plan*, the sequence of methods to call, with their arguments. Second, the plan is translated into Java bytecode, loaded as a class, and run.

A plan is similar to what I have earlier called a *trace*, which is a sequence of method calls on a collection. There are two differences. First, a trace is logged from the run of a program, while the plan is synthesized. Second, our traces did not log method arguments, but for our plans, we had to generate them. The generation of a plan requires to answer four questions:

1. How long is the plan?
2. Which methods are called on the collection?
3. What arguments are passed to the method calls?
4. What type of elements is stored in the collection?

I will explain our choices for each question, in turn. In our work on reproducing Brainy, we followed the same choices as in Brainy, but I think there are many opportunities for future work, here.

2.1 Length of the Plan

Longer plans take more time, and therefore take longer to benchmark. But they also influence the maximum size of the collection during the benchmark, since longer plans probably include more insertions.

In Paper I, we tried different plans of sizes 10, 10^2 and 10^3, while also varying the size of the collection at the beginning of the benchmark, with 0, 10^3 and 10^4 entries. In Paper III, we used plans of length 10^3, since this was what Brainy used. However, when we investigated the length of traces for collections, we observed that the distribution has a long tail. Many traces contain very few calls (sometimes none), and very few traces with up to 10^6 method calls. Table 1 shows the distribution of trace length of avrora, bloat, fop, and lusearch.

2.2 Methods to Call

Brainy models the distribution of method counts, meaning that the probability for a method to be called is fixed, it doesn't depend on previous of following method calls. In Paper I, we proposed an alternative model, using a Pólya Urn

Size of trace	Count
[0, 1]	10^6
(1, 10]	10^6
$(10^1, 10^2]$	2.16×10^4
$(10^2, 10^3]$	2.34×10^3
$(10^3, 10^4]$	537
$(10^4, 10^5]$	672
$(10^5, 10^6]$	27
$(10^6, 1.5 \times 10^6]$	1

Table 1: The distribution of size of traces for avrora, bloat, chart, fop and lusearch.

process: When a method is selected to be included in the plan, the same method is more likely to be called again later. The assumption is that a method call is often followed by a call to the same method, for example, if the collection is updated in a loop. This model introduces bias when generating a single benchmark, but this bias disappears when many benchmarks are generated. We argued that such a model was more realistic that a uniform distribution, because software developers usually use a subset of the methods available.

One argument against this type of model is that it does not consider domain-specific knowledge. For example, it allows for a plan where the clear is called repeatedly before the first insertion, which is not common in traces.

We tested another approach, which is to use traces to inform the generation. We tried to model collection usage using a Markov chain, in which the probability of the next method call depends on the current method call. We inferred the Markov chain, based on traces from avrora, fop, and lusearch. We observed that such Markov chain did encode some of the domain specific knowledge (e.g. the first method that is called on a set is often add). The Markov chain often contained loops: when a method was called, the following method call was often the same.

2.3 Method Arguments and Collection Elements

We chose to restrict ourselves to storing integers in our collections. For insertions, we generated a random index between 0 and the size of the collection. For insertions in maps, we picked random keys between 0 and 100, so the probability of a collision increased with the number of insertions in the map.

In Chapter II, I discussed how the cost of an insertion in a map is related to the cost of hashing the new key, or comparing it with those already in the map. We have not tested this approach. Using other data-types could help making our benchmarks more realistic (e.g. by varying the cost of hashCode). Unfortunately, we did not instrument our code to track method call arguments, or what type of elements were stored in collections. I suspect that storing different data-types in

collections would be an effective approach, but it opens a large design space, so there is even more of a need for efficient sampling.

2.4 The Realism/Diversity Trade-Off

An effective generative model for benchmarks requires two properties, which I call *realism* and *diversity*. Realism means that micro-benchmarks should match how collections are used in practice, and test common use-cases of collections. Diversity, on the other hand, means that micro-benchmarks should exercise all of the collection's API, allowing for the discovery of edge-cases where a collection shines (e.g. prepending is fast on LinkedList). I think these two properties have to be balanced: more realism restricts the set of possible plans to be close to the traces seen in programs, while more diversity means that we should expand the set of possible plans to include more unexpected plans.

2.5 Benchmark Selection

To increase the diversity of benchmarks, the original Brainy uses a selection process. First, a benchmark is generated, and Brainy tests it with each appropriate collection. It compare the execution times, and select the fastest collection. Then Brainy puts the benchmark into a "bin" associated with the winning collection. If the bin is full, the benchmark is discarded. Brainy repeats this process until there are enough benchmarks in each bin.

We implemented the same approach, but it becomes prohibitively slow. In our case, we tried to find 100 benchmarks for each of our 9 collections, and let it run for a month, and still was not finished. First, measuring the execution time of a benchmark with the Java Microbenchmarking Harness (JMH) take a few seconds. Second, we observed that some collections win far more often than others. In total, benchmark selection had tried 3 937 190 plans, without success. I suspect benchmark selection is slow because benchmark synthesis does not use data about existing benchmarks to produce new ones. It might be possible to make benchmark selection faster by reusing existing benchmarks to create new ones, but we haven't tested this approach.

3 Predicting Collection Performance

3.1 Feature Extraction

Brainy4J takes decisions based on how a collection is used. However, when we discussed influences on execution times in Chapter II, we said that collection usage was *unobserved* (see Figure 1). We cannot measure collection usage without affecting execution time.

Benchmark	Normal Startup	Normal Steady	Tracing Startup	Tracing Steady
avrora	7.6	6.5	9.9	9.5
bloat	3.7	2.7	5.9	4.4
chart	4.9	3.2	16.9	14.6
fop	2.8	0.4	3.9	3.2
lusearch	3.3	0.8	4.4	3.5

Table 2: Execution time (in seconds) without tracing, and when we trace CPU cycles, for 5 DaCapo benchmarks

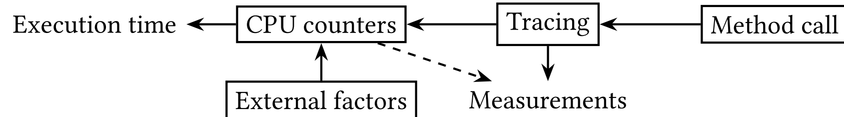


Figure 3: A causal graph describing the "observer effect" of tracing: A method call triggers tracing code which records CPU counters in the tracing results we will observe, but it influences the values we are measuring. Moreover, CPU counters are also affected by factors external to the program (CPU load, etc.) When we trace only method calls, the effect still affects the program, but the dashed arrow is not present. Our tracing code does not affect our measurements anymore.

All collection selection methods are affected by this, except Cres, which uses static program analysis. CoCo and CollectionSwitch count method calls, which incurs some overhead. Inspired by Brainy, we tried tracing number of method calls, and CPU counters such as L1-cache misses, branch mispredictions, and CPU cycles. Tracing such features incurs more cost than counting method calls, so we needed one run to measure execution time, and one run to measure collection usage. Table 2, shows that tracing had an significant overhead. Execution time increases, which is expected, but it also seems that JIT-compilation did not have as much of an effect. For example, fop's execution time drops a lot with warm-up, but when we trace, it does not.

The "Observer effect" of Instrumentation

Whether the tracing overhead is tolerable or not depends on what features of collection usage we trace. Figure 3 shows what I call the "observer effect" of instrumentation. To observe previously unobservable variables, we add instrumentation. Our instrumentation is tracing code, which is triggered by method calls. When tracing triggers, it records information that will be visible to us, as measurements. In Brainy4J, we want to be able to observe CPU counters, but the

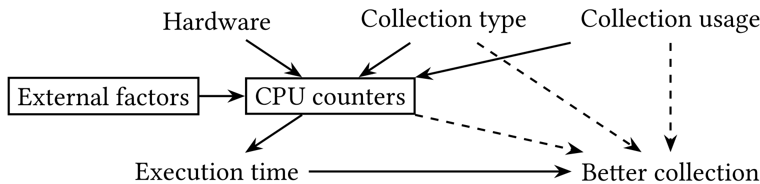


Figure 4: Causal graph describing the classification approach taken by Brainy. Black arrows denote the causal relationships in the training set. Brainy learns the type of collection to suggest, based on the collection type, CPU counters, and the collection usage (dashed arrows).

problem is that the tracing code *itself* affects the measurements.

On the one hand, if we want to predict execution time, we should trace a variable that is strongly correlated with it. For example, more cache misses and more branch mispredictions unambiguously translate to more execution time. Therefore, they give valuable information as to how well the collection is used.

On the other hand, if we trace something that we know is affected by the tracing code, how can we trust our measurements? In our case, we suspect that tracing affects CPU counters, because it hinders JIT-compilation. Consequently, if we cannot measure CPU counters in steady-state, we are probably missing on some effective optimizations. Instead, we may want to trace features that are unaffected by the tracing code, such as method calls. Then, we can trust our measurements, since they are unaffected by the instrumentation. When we compared several traces of our benchmarks, the number of method calls were stable. However, method calls are less obviously correlated with execution time, so their influence on execution time was harder to predict.

Collection Size

In Chapter II, I mentioned the complexity of operations on collections, which is usually related to the size of the collection. CoCo and CollectionSwitch both used the size of the collection to predict the cost of operations. In the setting of dynamic collection selection, the collection size at the time of the switch is a feature that is both easy to measure and correlated with execution time.

In the static replacement setting, like in the case of Brainy, the size of the collection is not available. For Brainy4J, we considered tracing it, but we know that collection size also varies with time. Should one use the max size, the mean size? As a proxy, we counted the number of cycles spent in insertions.

3.2 Classification

In Brainy, the problem of collection selection is described as a classification problem. Figure 4 shows a causal graph for the classification problem. In Paper III, we used the collection type and CPU counters, such as L1 cache misses, branch mispredictions, and clock cycles, to predict which collection would be most appropriate as a replacement.

Normalization

Since traces are of varying size, we must normalize them before passing them to a classifier. For software features, we divided the number of calls for each method by the total number of calls. For hardware features, we divided each feature by the total number of cycles.

Training

The original Brainy used neural networks for classification. In our case, we used random forests [HTF09, p. 587]. Random forests performed as well as neural networks in our case, while requiring fewer hyper-parameters. Moreover, they could also estimate the importance of features, so we could report it, just as they did in the original Brainy.

For each source collection (collection that could be replaced), we trained one classifier. We checked the accuracy of the classifier with 10-fold cross validation, which showed a reliable accuracy above 85% for all of them. However, these scores are to be taken with a grain of salt, because the labels are not balanced: some collections win more other than others.

Data Imbalance

When we generated benchmarks, we noticed that some collections win far more often than their alternatives. In a classification context, the consequence of this is that the classes to predict are not balanced. For example, ArrayList wins for *all* of the synthesized benchmarks for lists, in this case there is no need for a classifier. If the user uses a list, the classifier would suggest an ArrayList.

The imbalance in the data could have several causes: it could be that some collections are just better, when it comes to decreasing execution time, but it could also indicate a bias in the synthesized benchmarks. For example, in Paper I, we observed that for most of our benchmarks, ArrayList is faster than Vector, because Vector synchronizes each method call. Vector is more safe to use in a multi-threaded context, but synchronization has too high of a cost, and as a result, Vector is seldom used. In that sense, it seems fair to say that ArrayList dominates Vector. On other hand, most method calls are more costly on a LinkedList than on an ArrayList, but there is a case where LinkedList is faster:

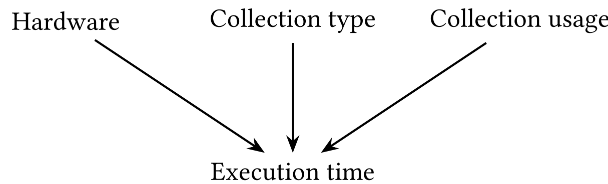


Figure 5: Causal graph describing the regression approach to collection selection. Black arrows denote the causal relationships in the training set. A regression approach would learn the relationship described by the arrows.

when we prepend to the list. In that case, it could be argued that a collection selection tool should be able to find such a use case.

There exist approaches to compensate for the imbalance in the training data. The original Brainy used benchmark selection: it kept sampling until the classes were balanced. Otherwise, there are options where one post-processes the imbalanced data, which I have not tried. One can under-sample the training data, rejecting some samples. Another approach is interpolate between existing samples to make new ones [Fer+18].

3.3 Regression

Instead of a classification approach, we can try to predict the execution time, given a collection type and statistics about collection usage. Collection Selection then becomes a matter of comparing the predicted execution times between different collections, and selecting the collection yielding the lower cost. Figure 5 shows the predicted causal graph of the relationship.

When using regression, collection selection requires an *intervention*: the regression model predicts *what would have happened* if we had used a different collection. In this case, one cannot use CPU counters as input features, because these are a *consequence* of running the program, with a specific collection. Regression only works with features which are independent of the collection type that was used in the program.

A regression-based approach has several advantages. During training, regression doesn't suffer from the problem of class imbalance. Every run yields valuable information, so there is no need for benchmark selection. Moreover, the regression model can be used both for prediction about collections, but also predict what would happen if we changed the benchmark itself, so we can use the model to generate new benchmarks. Afterwards, when we use it, a regression based approach can also tell *how much speedup* is expected after a collection change, which a classifier cannot do.

One problem that I observed with regression is that the order of method call matters. Using only the histogram of method calls for the trace might not work.

Another problem of a regression based approach is that of normalization: we need to normalize the inputs to the model, but also the output. Because different traces have different sizes, how to make a model that works with any size? I suspect that creating benchmarks of different sizes (e.g. $10^3, 10^4, \ldots$) and learning either a model for all, or a model for each, could work.

We have run some experiments to test if regression was possible, but without much conclusive results, I suspect this area could be developed further.

4 Summary

In this chapter, we considered the problem of modeling collection usage and generating synthetic benchmarks, to learn a predictor of collection performance.

I have listed four important decisions to make when modeling collection usage:

1. How many method calls should we make?
2. Which methods are called on the collection?
3. What arguments are passed to the method calls?
4. What type of elements is stored in the collection?

I have described different statistical models of collection usage that I experimented with. I mentioned the problem of realism and diversity of such a set of benchmarks. Should our synthetic benchmarks be close to how people use collections, or should they be different?

Once we have synthetic benchmarks, we need to extract information about collection usage in the target program. I have described the issue that instrumentation influences the features that we aim to measure.

Lastly, I compared two possible approaches to the prediction of collection performance. The first approach is to try to predict if a better option exists, using classification. The second approach is to predict the cost (e.g. time spent) of running a program with a collection, and suggest the option yielding the lowest cost.

In conclusion, collection selection can be seen as both a regression or a classification problem. However, the key problem is that of finding (or generating) training data that resembles the data that will be available when we use the model.

Now, at this stage of the design, are the issues that I highlighted in this section that critical? To know that, I needed to test my collection selection tool on actual programs. In the next two chapters, I will talk about the challenge of *evaluating* the effect of optimizations on Java software.

Evaluating Java Programs

Performance engineering is a bit like medecine. Engineers are interested in manipulating variables to reach a desired outcome (in our case, lower the execution time). To do this, they must run experiments.

I used to hate this part. In appearance, benchmarking is simple: try every option, and compare the numbers. This is not true, because as I tried to detect smaller and smaller effects, *everything* seemed to matter. Are there programs running in the background, changing our measurements? Is the CPU hot, or cold? Does the user-name influence the execution time, or not?

Avoiding such disturbances in my experiments required careful planning, which I had no experience with. At the time, I did not find good sources about experimental design in software engineering, so I looked elsewhere, in the statistics literature. This chapter, and the next, are about what I learned.

1 Design of Experiments

In experiments, scientists are interested in the *causal* effect of various *factors* (e.g. CPU architecture) on a *response variable* (e.g. execution time) [Mon01]. For each factor, one usually selects several discrete *levels*, which will be used in experiments. Sometimes, factors are called *independent variables*, and the response variable is called the *dependent variable*. Typically, computer scientists run *factorial experiments*, where they try every combination of levels.

1.1 Types of Factors

Montgomery calls factors which the experimenter is not interested in *nuisance factors* [Mon01]. Nuisance factors can be *controllable* or *uncontrollable*, and they can be *observable* or *unobservable*. Nuisance factors which are both uncontrollable and unobserved are often called noise. To derive causal relationships between independent variables and the response, the experimenter must remove influences

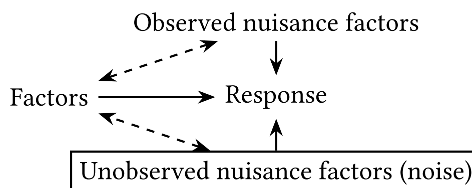


Figure 1: The different types of variables that can influence the response in an experiment. The experimenter is interested in the "Factors", which she sets, but observed and unobserved nuisance factors (noise) can also affect the response. The doubled dashed arrows represent influences that experiment design tries to remove.

between such variables and nuisance factors. To do so, experimenters shall use the three basic principles of experimental design: *replication*, *randomization* and *blocking* [Mon01]. Each technique is used to remove spurious influences between nuisance factors and factors of interest. Figure 1 displays the effect of different families of factors

1.2 Replication

A replication is when the experiment is ran several times [Mon01]. Because noise is not controlled nor observed, the results of the experiments will be slightly different each time. The purpose of replications is to allow the experimenter to separate the effect of the factors from the effect of the noise. This estimate of experimental error is required to make statistically significant comparisons. More replications also allow for more precise estimates of the mean effect of a treatment.

1.3 Randomization

Replications are useful for separating the factors from the noise, but factor levels and noise can still be correlated.

It seems the established practice among performance engineers is to try to "clean" the context, and remove noise altogether. Just like a biologist sterilizes their tools before performing experiments, we tried to remove as many sources of noise as possible. In our experiments, we tried to do this as well. We used some machines specifically for benchmarking, and disabled background daemons and some features like frequency scaling, to remove as much noise as possible. Some authors even restart machines between runs [Bar+17]. To me, this approach has the weakness of increasing the distance between experiments "in the lab" and observed behaviour "in the wild". For example, some authors turn-off CPU features, such as hyper-threading, during experiments [Tra+22], but end-users

will probably use hyper-threading. I think our experiments should be as close as possible to the context that users will experience.

Instead, randomization helps by ensuring that unobserved factors are *independent* of the relevant experimental factors [Mon01]. Then, unmeasured influences can be treated as noise, which one can "average away" with enough replications. In a randomized design, the levels of different factors are assigned at random to experimental units (individual measurements), and performed in a random order. For example, if I decide not to turn off hyper-threading, I should at least make sure it does not *systematically* affect my measurements. One way to achieve this would be to run experiments for different treatments in random order, and run several replications of the experiment.

In textbooks about experimental design, randomization is absolutely fundamental [Mon01; BHH05]. However, it did not seem to be used by experimenters working on collection selection. I suspect that for a computer scientist, randomization is not as obviously necessary, because we believe we have complete control over the machine. For example, the Java Microbenchmarking Harness can vary different parameters, and will follow a factorial design (try every combination of parameter values). JMH, however, does not randomize between configurations, it will always try them in the same order. Therefore, the experiments are not fully randomized, but JMH runs the statistical analysis as if they were.

I suspect this lack of randomization has an effect, but I haven't empirically tested this theory. In my research, I ran into the problem by running a benchmark on a machine I was working on. Because the order in which we considered treatments was always the same, treatments late in the list where favored, because by the time they ran, I would have left the machine alone. I think that since remove every source of noise is becoming increasingly difficult, we will rely more and more on randomization.

1.4 Blocking

Blocking is a technique to make estimates more precise by making use of controllable nuisance factors [Mon01]. We can select different levels for the nuisance factors, each level will form a block. Then we run a full randomized experiment within the block. Including the block in the statistical analysis reduce the variance and improve the precision of estimates. In our work, we used blocking by running the same experiment on different machines. Each machine is a block.

2 Design of Experiments and Causal Inference

The design of experiments and causal inference both try to infer a *causal* relationship between variables. The difference between the two is that experimental design is about how to *collect* the data, while causal inference is about how to *analyse* the data.

In their textbook about experimental design, Box, Hunter and Hunter warn against "happenstance data", which is not gathered through experiments. They say (emphasis theirs),

> To safely infer causality, the experimenter cannot rely on natural happenings to choose the design. She must intervene, choose the factors, design the experiment for herself, and, in particular introduce randomization to break the links with possible lurking factors **To find out what happens when you change something, it is necessary to change it.** [BHH05]

Causal inference, on the other hand, says quite the opposite: We could, in fact, infer causal relationships, if we are careful in our statistical analysis [PGJ16]. What you need is a causal graph, and to apply a set of principles.

A full introduction to causal inference is outside the scope of this thesis, because I am not dealing with happenstance data, I'm designing and running experiments. For interested readers, I suggest Pearl et al's textbook [PGJ16]. However, to show why I think causal graphs are interesting, I will express the concepts of design of experiments with causal graphs.

2.1 Causal Graphs in Design of Experiments

We can map types of factors to properties of factors in the causal graph. In causal inference, the scientist is interested in one specific edge of the graph, the causal effect on one variable on another. The dependent variable is the target of that edge. Independent variables are called *exogenous variables* [PGJ16]. They are variables which are not descendants of any other variable in the graph: they have no incoming edges. The experimenter may set arbitrary values to them. Variables which are descendents of other variables (they have incoming edges) are called *endogenous* variables. They can be confounders, but not necessarily. It depends of how they are connected to the dependent variable.

Nuisance factors are variables from which the dependent variable is a descendent, they can be either observed or unobserved. When a nuisace factor is unobserved, it is called "noise" or "error". An important question is whether nuisance factors have incoming edges. If there are edges between variables directly affecting the dependent variable, then one must be careful in the statistical analysis of the results [PGJ16].

If two variables are connected through a sequence of arrows (e.g. $X \to Y \to Z$), there is a *causal path* between them [McE20]. If they are connected through a path which contains one or more arrows connected "backwards" (e.g. $X \leftarrow Y \to Z$), then they are connected through a *non-causal* path. Non-causal paths are the confounders that can prevent us from inferring causal effects. Now, if we observe a correlation between two variables which are *not* connected through a path, then we know the graph is missing either some vertices or some arrows.

To measure correlations (which might indicate edges that we missed), we need several replications. Randomization on the other hand, is used to "cut" edges which disturb our inferences [PGJ16].

You can certainly do experimental design without drawing a causal graph, and classical textbooks [BHH05; Mon01] do not use such graphs. They were written before causal inference became widespread. But now that we have causal graphs, I think we should use them. Causal graphs clarify many informal discussions around which variables are independent, which variable is the dependent variable, and what the confounders are. They are simple and explicit, so researchers can compare them, and discuss them.

3 Summary

Domain knowledge is necessary to plan meaningful experiments. In Chapter II, I discussed it, expressing it using causal graph. In this chapter, I explained *why* I did it with causal graphs.

In my case the domain was discussed in papers, but in an informal way. I was lacking a language for compressing that information and make it usable. I think that causal graphs fill that gap.

In my work, causal graphs are now a necessary step in designing an experiment. They serve as a compact but rich framework to express the relationships at play in my domain. In the next chapter, we will see what we need to consider *after* we run an experiment, when we analyze the data.

Analyzing Experimental Results

The result of experiments is data. In this chapter, I will consider the problem of analyzing benchmark results. Classical statistics textbooks [BHH05; Mon01] propose tools like hypothesis testing, confidence intervals, and the Analysis of Variance. These tools are well established in empirical disciplines, but are getting increasingly criticized in these fields [WSL19; Hoe+14; Nuz14].

Eventually, I also became more critical of these techniques. I now believe that there are promising alternatives, and in Paper II, we describe a statistical model that uses some of them. Paper II is rather short, so in this chapter, I describe the content of the paper in more detail.

Next, I will describe how classical methods work in more detail, present alternatives, and why I prefer them.

1 Hypothesis Testing

Imagine the example of measuring execution times for two programs, is program A is faster than program B? An experimenter runs each program a number of times, measuring how long they take. If they compared the means execution times (μ_A and μ_B) directly, these will surely be different. When can one say that the difference of means is "large enough" to conclude that, A is slower (or faster) than B? To get a clear (yes/no) answer to that question, researchers use *statistical hypothesis testing*, or *confidence intervals*.

In the case of statistical hypothesis testing, researchers pick a *null hypothesis*, usually, that the means of the two groups are equal: $H_0 : \mu_A = \mu_B$. Then, they pick a *test statistic*, which is the probability of observing a difference between means *given* the null hypothesis is true: $P(\mu_A - \mu_B | H_0)$. This value is called the *p-value*. If the p-value is low enough (usually under 0.05) researchers conclude

that the observed difference is implausible under the null hypothesis, which is therefore false. They then *reject* the null hypothesis of *no* difference between the two groups, and conclude there is a *statistically significant difference*.

To me, rejection of the null hypothesis was very confusing. Why would I conclude that I *reject* an hypothesis of *no* effect? The answer has to do with what we're trying to achieve with statistics: induction. Loosely speaking, induction is the process of making predictions about the future, based on data about the past. Here is a simple example: If I see a large number of white swans, when can I state that all swans are white? In other words, when is a theory *confirmed*?

Karl Popper provided a simple answer to that question: A theory is *never* confirmed [GS03]. In other words, according to Popper, one can *never* prove a theory is *true*, no matter how many cases one finds that agree with it. Experiments can only show a theory is *false*, by presenting a counter-example. No matter how many white swans I see, Popper says I can never assume that all swans are white [1]. In accordance with Popper, it makes sense to build a (usually restrictive) null hypothesis, and check if it can be rejected. Because it is the only thing I can do.

Now, It seems unlikely that Ronald Fischer, who popularized the concepts of null hypothesis and p-values, had Popper in mind. He published his book *Statistical Methods for Research Workers* in 1925, while Popper's book, *The Logic of Scientific Discovery*, was published in German in 1935, and was translated to English in 1959. But even if Fischer did not base his design on Popper's ideas, they are popular among scientists [GS03], which explains why they still use null hypotheses.

It seems that Fischer originally considered p-values to be an informal test. It was only later included in a larger hypothesis testing framework, by authors who were not statisticians themselves [Nuz14]. In practice, p-values have been criticized for decades, because scientists frequently misinterpret them [Nuz14]. Some assume that p-values estimate the probability of the null hypothesis being true (or false), while that is not what they measure. They measure the probability of observing an event *if* the null hypothesis was true.

Even if scientists interpreted p-values correctly, their use is still criticized, because scientists use them as a binary outcome: Either a result is significant, or it is not. In practice, scientists use them to determine if a result is worth publishing or not [WSL19]. A yes/no answer gives a sense of certainty, but it is misleading. Like all tests, statistical significance will show true positives, but also false positives, and false negatives. A true positive is an interesting scientific result. A false positive is a result that fails to reproduce, and a false negative ends up in a drawer, never to be published [WSL19].

---

[1] Yes, some swans are black, and yes, they live in Australia.

2 Confidence Intervals

A proposed alternative is *confidence intervals.* A confidence interval proposes a range of plausible values of an unknown parameter, for example, a difference between the means of two groups. The term *confidence* refers to how reliable the procedure is: if I build 95% confidence intervals based on my data, 95% of these intervals will contain the true mean [Hoe+14]. The confidence interval of a mean grows with the confidence level, the variance of the samples, and shrinks with the number of samples. Usually, researchers compute the 95% confidence interval, and decide that if the interval does not overlap zero, the difference is significant. There is formally no null hypothesis to select, although in practice, the effect is the same: pass a hard threshold, and claim statistical significance.

Unfortunately, many researchers fail to correctly interpret confidence intervals [Hoe+14; Mor+16]. Perhaps surprisingly, if I observe a confidence interval of $[0.1, 0.4]$, I *cannot* assume this interval has a 95% probability of containing the true mean. Confidence intervals do not answer the question that scientists ask: Scientists are interested in the probability that the parameter of interest lies in the observed interval, but confidence describes how reliable the procedure is on average, it says nothing about the one single interval, computed from experimental data.

Aren't we getting lost in perfectionistic discussions, only relevant to statisticians and philosophers? In a way, yes. The *meaning* of our statistical inferences is counter-intuitive, and one may wonder if we are talking about practical problems, or philosophical details.

But we now have two problems on our hands. First, Scientists do not fully understand the tools they use. And second, they use these tools to determine what is published, and therefore what counts as scientific knowledge. But alternative *do exist,* so I think we should seriously consider the alternatives.

2.1 What Should One Do Instead?

The proper approach to resolve the situation is still a matter of hot debate, since it impacts large parts of the scientific community [WSL19].

One approach is to stop using hard thresholds. For example, to report continuous p-values (e.g. 0.023). Other authors suggest focusing on the effect size and its domain-specific implications: Does it really matter if A is 0.4% faster than B with a p-value of 0.043, if the variance of A is of 20%?

One ambitious solution I find particularly promising, is that of pre-registered (results-blind) studies [WSL19; Coc+20]. Instead of changing the statistics, it changes the publication process. A registered report is the same as a normal research paper, but it would be accepted or rejected *before* the study is conducted. It would contain everything a research paper contains (introduction, related work, methods) *except* the results and their discussion. Submissions are therefore

evaluated on the grounds of the research questions and the methods used, which are the most important.

For now, however, can we find a better option to report our uncertainty about estimates? Bayesian statistics provides promising options, which allow to answer the type of questions that scientists are interested in, such as:

- What is the probability that A is better than B?
- What is the probability the mean of A is in a given interval?

Since these options exist, I see no reason to use them. The Bayesian approach is what we will consider next.

3 Bayesian Statistics

To talk about Bayesian statistics, I need to discuss what a probability *means*. Hypothesis testing and confidence intervals come from *frequentist* statistics. In frequentist statistics, a probability meant a proportion of events as we made more tries: If I throw a coin many times, I will eventually observe 50% of heads, says the frequentist. For the Bayesian, a probability describes a *subjective belief* [GS03]. Before we throw the coin, we don't know which side will show up, so a Bayesian would say that we should assign a probability of 50% to each side. The rules of probability work just as well in both definitions. The key difference is that for the Bayesian, the researcher is allowed to give probability distributions to describe their *belief* about its plausible values.

3.1 Random Variables and Events

To make things a bit clearer, I need some definitions. A *random variable* is any quantity, or property we are interested in, it should be uncertain (that's why it's called random). An *event* is a mapping of a random variable to a value or set of values. Events are true or false, like $X = 2$, or $Y < 45$. The probability of an event expresses a belief that the event is true.

A *conditional probability* expresses "influences" between events. $P(A|B)$ represent the probability that A happens, given that B has happened. If $P(A|B) = P(A)$, then we say that A and B are *independent*: the fact that B happened doesn't influence A.

3.2 Bayes' Rule

The foundation of Bayesian statistics is the use of *Bayes' rule*, a simple relationship between conditional probabilities:

$$P(A)P(B|A) = P(B)P(A|B) \quad (1)$$

Both sides of the equation are just different ways to write $P(A \cap B)$. What is interesting in Bayes' rule is that it relates $P(B|A)$ and $P(A|B)$.

If we rename A and B to H (for hypothesis) and E (for evidence), and change the equation a bit, we get:

$$P(H|E) = \frac{P(H)P(E|H)}{P(E)} \tag{2}$$

$P(E|H)$ is called the *likelihood*, the probability of observing the evidence, given the hypothesis is true. In many cases, we know or can easily determine it [PGJ16]. For example, p-values measure $P(E|H_0)$, the probability of the observed data *given* the null hypothesis is true. $P(H)$ is the probability of the *hypothesis*, which is called the *prior probability*, it doesn't mention E for good reason, it's the probability of the hypothesis being true before we observed any data. $P(E)$ is the probability of the evidence (independently from H being true), in many cases, we won't estimate this quantity, because we're only interested comparing different hypotheses $H_1, H_2, \ldots$ with the same data, in which case $P(E)$ would stay constant.

Bayes' rule has profound implications: it describes how one can change one's mind. $P(H|E)$ is the *posterior probability*: The probability that the hypothesis is true, provided the evidence E has been observed. If $P(H|E) > P(H)$, then the evidence *confirms* the hypothesis, otherwise, it *refutes* it. While Popper described confirmation of theories as impossible, Bayesians answer that confirmation is in fact quite common. This conclusion is important: philosophers of science had struggled with the problem of confirmation for most of the twentieth century [GS03].

Now, there is some controversy attached to Bayes' rule [PGJ16]. In most cases, $P(H)$ cannot be estimated, so the statistician *chooses* a distribution, which means that the posterior probability is tainted a subjective choice of prior: For example, if some hypotheses are *impossible* according to the prior, then no amount of data can change that. To prevent this, Furia et al. [FFT19] suggest to test the same model with different priors, in what they call a *prior sensitivity analysis*.

I find the use of priors more sensible that picking null hypotheses. When a researcher chooses a prior, they can express their domain knowledge, for example by involving the related work. That is not what researchers do when picking null hypotheses. Moreover, if a scientist report posterior distributions in their work, Bayes' rule allows other researchers to build on this information in a principled way. Bayes' rule expresses elegantly the idea that current research builds on existing knowledge: Today's posteriors are tomorrow's priors.

4 Linear Regression

To show how I use the Bayesian Framework, I will present an example of a linear regression with simulated data. The argument will be close to that of Paper II, but in extended form, since the original paper is only four pages long.

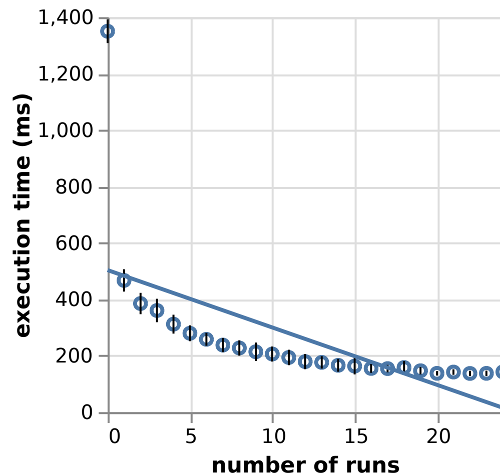


Figure 1: Execution times for fop (20 replications). The point denotes the mean, the error bar the standard deviation. The line denotes a linear regression. It would make poor predictions: after 25 iterations, it would predict the execution time is negative. But it does indicate the downward trend.

Linear regression models the relationship between factors and the response by a straight line, hence the name. Figure 1 shows a linear regression fit to execution times for fop. It is a simple model, but it has the advantage of being quite flexible, and somewhat interpretable.

I pick simulated and not real-world data, because when working with simulated data, I know what I'm looking for. I will compare the simulation parameters with the inferences from different statistical methods. When using real-world data, the full data-generation model is unknown, so I would never be sure that the inference is correct.

Here is the simulation model. We assume we have two machines M_1, M_2, and three treatments t_1, t_2, t_3. Both machine and treatment influence the execution time, as displayed in Figure 2. An example of a treatment could be to replace a ArrayList by a LinkedList and some program location. The parameters of the model are an array β of effect for each machine, and a matrix of effects α for each combination of machine and treatment. The simulation model is described in the equations below. $X \sim D$ means X is sampled for the distribution D.

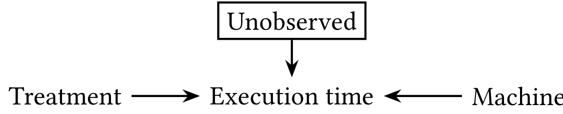


Figure 2: A causal diagram describing the influences present in the simulation.

$$
\begin{aligned}
\alpha[M_1, t_1] &= 0.0,\ \ \alpha[M_2, t_1] = 0 \\
\alpha[M_1, t_2] &= -0.2,\ \ \alpha[M_2, t_2] = 0.25 \\
\alpha[M_1, t_3] &= 0.2,\ \ \alpha[M_2, t_3] = 0.5 \\
\beta[M_1] &= -1 \\
\beta[M_2] &= 1 \\
\sigma &= 0.1 \\
\mu_i &= \beta[M_i] + \alpha[M_i, t_i] \\
\text{execution time}_i &\sim \text{Normal}(\mu_i, \sigma)
\end{aligned}
$$

If I implement this model and simulate 20 runs for each combination of machine and treatment, I obtain a table similar to Table 1. I assume execution times are relative to the mean so there are negative values. Figure 3 shows the plotted data.

Machine	Treatment	Execution time
1	1	-1.14584
2	1	0.846367
1	2	-0.990952
2	2	0.9508
...	...	...

Table 1: Example of data from a synthetic model

I will first analyze the data using ordinary least-squares regression, and then compare it with a bayesian model. This case contains categorical variables, which can take a finite number of values, called *levels*. These levels are unordered (M_2 is not more machine than M_1). Linear regression doesn't support categorical variables, which must be encoded.

4.1 Categorical Variables

There are several ways to use discrete values with linear regression. One is *one-hot encoding*: For each level, create a new input variable, which takes the value 1 if

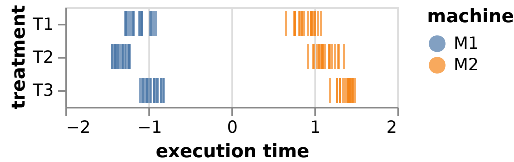


Figure 3: The results of the simulation of 20 runs. Treatment 1 is the baseline, so its effect is considered to be zero. Treatment 2 speeds up the program on M_1, but slows it down on M_2. Treatment 3 slows the program to varying degrees.

the value of the factor is of that level or 0 if it is not. For this example, I will create one input variable per machine, called M_1 and M_2. For each sample, I will set the variable associated with a used level to 1. Table 2 shows the updated data. For treatments, the procedure is essentially the same.

M_1	M_2	Treatment	Execution time
1	0	1	-1.14584
0	1	1	0.846367
1	0	2	-0.990952
0	1	2	0.9508
…	…	…	…

Table 2: Example of data from a synthetic model, with dummy encoding

Quite often, the experimenter is interested in the *difference* between a "base" level and other levels. In that case, there will be no column for that level, but instead, every other column for the same feature will be set to zero. In this example, that would correspond to removing column M_1 and keeping M_2. The coefficient in the linear model of M_2 will indicate the difference relative to M_1. This is called *dummy coding*. In a way, the base level is implicit, since there is no input variable to represent it.

We create a first baseline linear regression model, with dummy-coded categorical variables. We set t_1 as a baseline, so we have two variables: t_2 and t_3. β represents the intercept, which is the mean of execution times for t_1.

$$\text{execution time}_i = \beta + \alpha_{t_2} t_2 + \alpha_{t_3} t_3 + \epsilon$$

Ordinary least squares linear regression will pick the $\beta, \alpha_{t_1}, \alpha_{t_3}$ which minimize the mean square error. Figure 4 shows the estimated effects and their error compared to the parameters set when designing the simulation. Perhaps surprisingly, it was not necessary to add the machine to the regression to estimate to the real value of the effects. This is because the treatment is independent from the machine used, the two are independent.

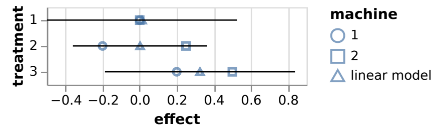


Figure 4: The estimated effect (triangle) vs the real effects for different machines (circles and square), for different treatments. The error bars display the error in the estimate. The linear model estimates the *mean* effects of treatments. As a result, it estimates treatment 1 has no effect on average, even if it would have a positive effect on machine 1.

One could instead use a model that included the machine, as below,

$$\text{execution time}_i = \beta + \beta_{M_2} M_2 + \alpha_{t_2} t_2 + \alpha_{t_3} t_3 + \epsilon$$

This models makes the *same* estimates for the mean effects. However, it makes more precise estimates: the confidence intervals are narrower. This is the effect of *blocking*, discussed in Chapter V.

With Bayesian inference, instead of dummy coding, one can use arrays of coefficients. The model is close to the equations of the simulation. The only difference is that parameters that took values in the simulation are now unknown, and one must therefore assign priors to them. For each sample i, the treatment t_i is assigned a coefficient $\alpha[t_i]$, from an array α. x is an observed variable (data provided to the model), and α denotes a parameter, which requires a prior distribution.

$$\begin{aligned}
\text{execution time}_i &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \alpha[t_i] \\
\alpha[t] &= \text{Normal}(0.0, 0.5), \forall t \in \{t_1, t_2, t_3\} \\
\sigma &= \text{Exponential}(0.2)
\end{aligned}$$

To estimate posterior distributions, I can use an inference engine such as Turing [GXG18]. Turing returns a vector of samples for each parameter, which represents its posterior distribution. Figure 5 shows the posterior distributions of the effect of treatments.

I could estimate the effect of the treatment, but I do not know if that effect varies for different machines. In a sense, this *mean* effect is a more generalizable result. It averages across different contexts, so if the effect is strong, it will be observed in more situations. To test how the effect varies with the context, one can add *interactions*.

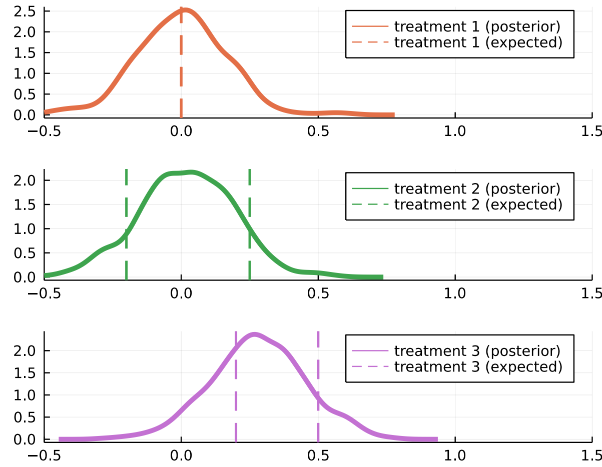


Figure 5: The posterior distribution of mean treatment effects. The posterior distributions match well the estimates in Figure 4, except in the shape of distributions instead of intervals. To report credible intervals (the Bayesian equivalent to confidence intervals) we can take quantiles of the posterior distributions.

4.2 Interactions

An *interaction* is when the treatment effect varies with the value of another variable [GHV21]. In this example, is the treatment just as effective for any machine, or is it only effective for some?

In an ordinary least-squares model, an interaction is the *product* of the dummy coded variables. For example, for a sample which used the machine M_2 and the treatment t_3, we get:

$$\text{execution time}_i = \beta + \beta_{M_2} M_2 + \alpha_{t_3} t_3 + \alpha_{t_3,M_2} t_3 M_2$$

All other factors like (such as $\alpha_{t_2} t_2$) are set to zero, because of dummy coding. Figure 6 shows the result of a regression with interactions.

In Bayesian inference, interactions for categorical variables are represented by a matrix of coefficients. For a factor with k levels, and another with l levels, I create a $k \times l$ matrix of coefficients. For this example, here is how a model with interaction will look like.

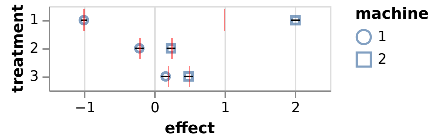


Figure 6: The effect of machine and the estimated effects of treatments, when using interactions. The ground truth is displayed as red ticks. Now, the treatment effect varies by machine, so two effects are estimated for each treatment. These estimates are much more precise, and the confidence intervals for the parameters overlap the values I set for the simulation. In the simulation, I set the effect of M_2 to 1.0, but in this model, the coefficients is 2, because of dummy coding: The coefficient measures the effect of M_2 relative to M_1 (which was -1.0) The treatment effects are relative to t_1.

$$
\begin{aligned}
\text{execution time}_i &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \beta[M_i] + \alpha[M_i, t_i] \\
\beta[M] &= \text{Normal}(0.0, 0.5), \ \forall M \in \{M_1, M_2\} \\
\alpha[M, t] &= \text{Normal}(0.0, 0.5), \ \forall M \in \{M_1, M_2\}, t \in \{t_1, t_2, t_3\} \\
\sigma &= \text{Exponential}(0.2)
\end{aligned}
$$

Figure 7 shows the posterior distribution of effects for the two machines and the different treatments. Because I used two separate coefficients for the effect of treatment (one for each machine), I obtain different posteriors. The model seems to infer that M_2 is slower than it actually is. Since the model does not know that t_1 is our baseline, it estimates the effect of M_2 to be closer to the mean of the execution times for M_2.

One can estimate the effect of treatments *relative* to t_1. Since the posterior of each parameter is a vector of numbers, I can estimate the difference between any two variables by subtracting the vectors. Figure 8 shows the result: the estimates match the ground truth.

4.3 Data Transformations

So far, our linear model estimated the effects in terms of differences in absolute execution times. Performance engineers instead prefer to work with ratios, so that we can express speedups in percentages (e.g. “A is 32% faster than B”). To do that, we just need to run a linear regression on the *logarithm* of the execution time. Then, coefficients can be interpreted as ratios.

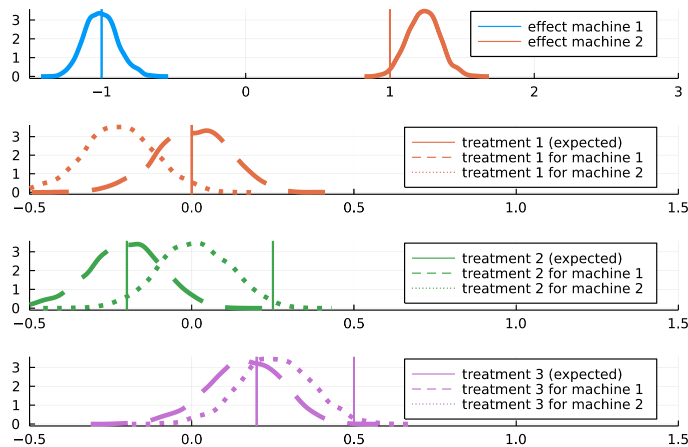


Figure 7: The posterior distribution of the effect of each machine (1), and the effect of treatment, for each machine (2, 3, 4). The interaction model seems to over-estimate the effect of M_2. As a result, the effects of treatments for M_2 are shifted, even if the real effect is covered by the posterior distribution. This happens because this model infers effects relative to the *mean* of the effects for M_2. Since we did not define the model so that t_1 is our baseline, it doesn't estimate its effect to be zero.

5 Hierarchical Models

So far, I used fixed priors for our effects, like $\text{Normal}(0.0, 0.5)$. What is my justification for such a prior? As we have seen, some might argue that priors allows the introduction of bias in the model.

If I do not want to commit to a specific prior, I can add a parameter to its definition. For example, one can replace $\text{Normal}(0.0, 0.5)$ by $\text{Normal}(0.0, \sigma)$, where σ is a new parameter. These are called a *hyperparameters*. Because σ_t is a parameter, it needs a prior, which is then called a *hyperprior*. A model with hyperpriors is called a *hierarchical* model.

Hyperparameters have a number of uses. First, they can be used to express uncertainty about our priors. If $\text{Normal}(0.0, 1.0)$ seems too "stiff", I may replace it with $\text{Normal}(\mu, 1.0)$, and set a prior distribution for μ. Plausible values of μ are learned from the data, so μ will get a posterior distribution too. Second, hyperparameters can be used to flexibly "connect" variables together and pool information. I will show an example soon. Third, the posterior distribution hyperparameters can be used to gain information about the structure of the model. I will show an example with the Bayesian ANOVA, from Gelman [Gel05].

We start with pooling information. In my model, coefficients are *clustered*:

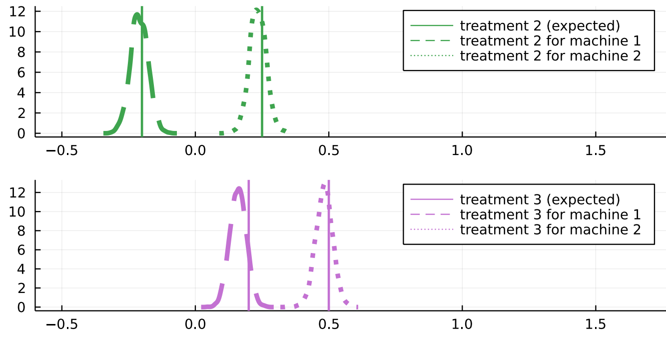


Figure 8: The posterior distribution of the effects of t_2 and t_3 *relative* to t_1. The posterior distributions are narrow and close to the values that we expected. They still show treatments vary by machine.

some concern effects of machines, others concern effects of treatments. I can exploit this clustering of coefficients to prevent overfitting, by pooling information about the parameters [McE20]. I do that by having the prior for all treatments coefficients depend on the *same* hyperparameter.

In the model below, all treatment effects are linked by the common parameter $\sigma_{\text{treatments}}$. A treatment t_j and another t_k both depend on $\sigma_{\text{treatments}}$. As a result, information about t_j influences t_k through $\sigma_{\text{treatments}}$. I assume that such treatment effects are sampled from the same *family*, the family of treatment effects. What the model learns about each treatment informs the posterior distribution of the family, which will inform the prior for the other treatment effects.

$$
\begin{aligned}
\text{execution time}_i &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \beta[M_i] + \gamma[M_i, t_i] \\
\beta[M] &= \text{Normal}(0.0, \sigma_{\text{machines}}), \forall M \in \{M_1, M_2\} \\
\gamma[M, t] &= \text{Normal}(\alpha[t], \sigma_{\text{interaction}}), \forall M \in \{M_1, M_2\}, t \in \{t_1, t_2, t_3\} \\
\alpha[t] &= \text{Normal}(0.0, \sigma_{\text{treatments}}), \forall t \in \{t_1, t_2, t_3\} \\
\sigma_{\text{treatments}} &= \text{Exponential}(0.1) \\
\sigma_{\text{machines}} &= \text{Exponential}(1.0) \\
\sigma_{\text{interaction}} &= \text{Exponential}(0.1) \\
\sigma &= \text{Exponential}(0.2)
\end{aligned}
$$

This pooling of information is particularly valuable if you do not have the same amount of data for different treatments [McE20]. But I can also use it to study the *variantion* between coefficients in different groups.

5.1 Analysis of Variance

Bayesian ANOVA

The Analysis of Variance (ANOVA) estimates the importance of each factor by looking at how much each factor contributes to the total variance. Bayesian ANOVA [Gel05] estimates the importance of each factor by looking at the variation in groups of coefficients in a linear regression. Groups of coefficients with higher variance are more important. [2]. To measure this variance, all we have to do is use a hierarchical model, like the model presented above. Then, we look at the posterior distributions of $\sigma_{\text{treatments}}$, σ_{machines}, etc. Figure 9 shows the posterior distributions of $\sigma_{\text{treatments}}$, σ_{machines} and $\sigma_{\text{interaction}}$. There is more variation between machines, so this factors contributes more to the variance of the data. Between the mean effects of treatment and the interactions, interactions vary a little bit more, but the difference is not very clear-cut.

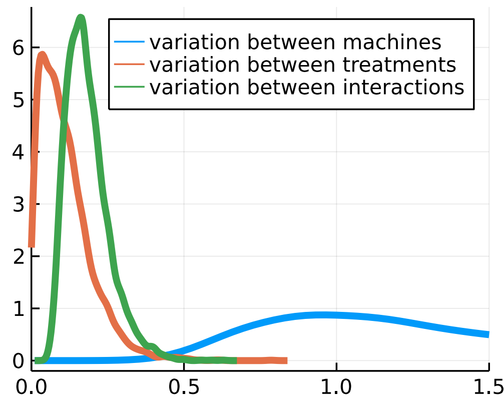


Figure 9: Posterior distributions of the standard deviation of treatment effects (orange), machine effects (blue) and interaction effects (green). This plot shows that there is more variation between machine than there is between interactions. It also seems like interactions vary more than the mean effect of treatments.

Classical ANOVA

In the case of classical ANOVA, the idea is essentially the same, it tries to estimate how much each factor contributes to the variance in the data.

[2] In the case where factors have different scales, we should standardize them, but in the case of categorical variables, we do not need to.

" The idea is to determine whether the discrepancies *between* the treatment averages are greater than could be reasonably expected from the variation that occurs *within* the treatments. " [BHH05]

Classical ANOVA achieves this through different means, so I need to introduce new concepts. Instead of using the variance, ANOVA uses the sum of squared deviations. The sum of squared deviations from the grand mean is decomposed as a sum of terms. Then, the experimenter needs to compare the computed sums of squares to a null hypothesis. In the case of ANOVA, the null hypothesis is that there are no differences between treatments. I will start with a description of the approach, taken from Montgomery [Mon01], and then show some examples.

I assume an experiment where I have have a treatments. For each treatment, I do the experiment n times. The total number of observations is $N = an$. The response for treatment i and replication j is noted y_{ij}.

To estimate the variance *between* treatments, I will look at the mean responses for each treatment. $y_{i\bullet}$ represents the sum of responses for treatment i. $\bar{y}_{i\bullet}$ is the mean of the responses for treatment i. Similarly, $y_{\bullet\bullet}$ is the grand sum of all responses, and $\bar{y}_{\bullet\bullet}$, is the grand mean.

More formally,

$$y_{i\bullet} = \sum_{j=1}^{n} y_{ij} \qquad \bar{y}_{i\bullet} = y_{i\bullet}/n \qquad i = 1, 2, \ldots, a$$

$$y_{\bullet\bullet} = \sum_{i=1}^{a}\sum_{j=1}^{n} y_{ij} \qquad \bar{y}_{\bullet\bullet} = y_{\bullet\bullet}/N$$

One uses the definitions above to compute the total sum of squares (TSS), the explained sum of squares (ESS) and the residual sum of squares (RSS). The total sum of squares is the sum of squared deviations from the grand mean. The explained sum of squares is the sum of squared deviations between the mean of each treatments and the grand mean. The residual sum of squares is the sum of squared deviations between the mean of each group and each datapoint.

$$\text{TSS} = \text{ESS} + \text{RSS}$$

$$\sum_{i=1}^{a}\sum_{j=1}^{n}(y_{ij} - \bar{y}_{\bullet\bullet})^2 = n\sum_{i=1}^{a}(\bar{y}_{i\bullet} - \bar{y}_{\bullet\bullet})^2 + \sum_{i=1}^{a}\sum_{j=1}^{n}(y_{ij} - \bar{y}_{i\bullet})^2$$

Null Hypothesis In the case of one single factor, the null hypothesis is that there are *no differences* between treatments, and all the variance is due to the effect of the residuals. If the null hypothesis is true, one can define the distribution of the expected TSS, ESS, and RSS. This distribution is called the *chi-squared* distribution, which parameters are called *degrees of freedom*. A chi-square distribution with

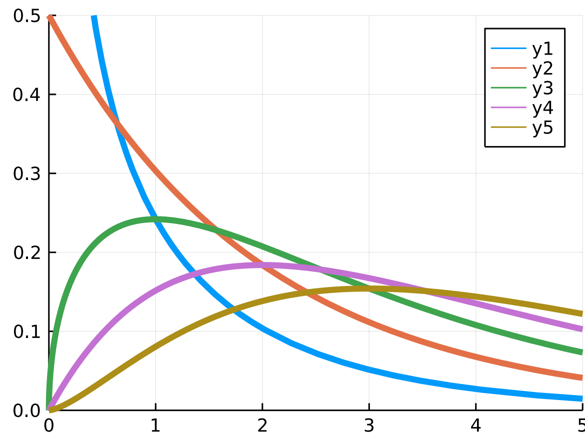


Figure 10: The distribution of the sum of squares of k standard normal random variables, called the chi-square distribution. For different values of k between 1 and 5.

k degrees of freedom represents the distribution of a sum of k squared random variables, if these random variables are independent, and follow a standard normal distribution. Figure 10 shows the distribution for different values of k.

Degrees of Freedom If the null hypothesis is true, all the variance in the data is due to the residuals. If that is true, and if the residuals are normally distributed with mean 0 and variance σ^2, then we can show that TSS/σ^2 follows a chi-squared distribution with $N-1$ degrees of freedom [Mon01].

Just like the sum of squares, the total degrees of freedom can be decomposed as sums of degrees of freedom: ESS/σ^2 follows a chi-squared distribution with $a-1$ degrees of freedom, and RSS/σ^2 follows a chi-squared distribution with $N-a$ degrees of freedom.

Test Statistic With the chi-square distributions, one could check if the ESS or RSS fit the null hypothesis, but that would be two distributions, giving two different p-values. Instead, ANOVA computes the F-statistic:

$$F_0 = \frac{\mathrm{ESS}/(a-1)}{\mathrm{RSS}/(N-a)}$$

Under the null hypothesis, the F-statistic will follow a distribution called the *F-distribution* which takes two parameters: The degrees of freedom $(a-1)$ and $(N-a)$. This distribution is used to compute the p-value of F_0.

Adding More Factors If there is more than one factor, one needs to add sums of squares to the decomposition. For example, in the case where there are two factors A and B with levels a and b, the sum of squares is decomposed as follows:

$$\text{TSS} = \text{ESS}_A + \text{ESS}_B + \text{ESS}_{AB} + \text{RSS}$$

Where,

$$\begin{aligned}
\text{TSS} &= \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(y_{ijk} - \bar{y}_{\bullet\bullet\bullet})^2 \\
\text{ESS}_A &= bn\sum_{i=1}^{a}(\bar{y}_{i\bullet\bullet} - \bar{y}_{\bullet\bullet\bullet})^2 \\
\text{ESS}_B &= an\sum_{j=1}^{b}(\bar{y}_{\bullet j\bullet} - \bar{y}_{\bullet\bullet\bullet})^2 \\
\text{ESS}_{AB} &= n\sum_{i=1}^{a}\sum_{j=1}^{b}(\bar{y}_{ij\bullet} - \bar{y}_{i\bullet\bullet} - \bar{y}_{\bullet j\bullet} + \bar{y}_{\bullet\bullet\bullet})^2 \\
\text{RSS} &= \sum_{i=1}^{a}\sum_{j=1}^{b}\sum_{k=1}^{n}(y_{ijk} - \bar{y}_{ij\bullet})^2
\end{aligned}$$

The degrees of freedom are decomposed as follows:

Effect	Degrees of Freedom
A	$a-1$
B	$b-1$
AB interaction	$(a-1)(b-1)$
Error	$ab(n-1)$
Total	$abn-1$

Examples

Table 3 shows the result of the ANOVA, for our simulation, as I computed it in R. In this case, it shows that the model for treatments predicts points close to the mean. Particularly, the F-statistic is close to 1, and the p-value is quite high, which means the treatment does not explain most of the variance in the data.

I will now compare the previous ANOVA with the same analysis on a model which predicts the execution time based on machine only (Table 4). Here, the

Feature	Df	Sum Sq	Mean Sq	F value	Pr(>F)
treatment	2	3.6	1.798	1.368	0.259
Residuals	117	153.7	1.314		

Table 3: ANOVA for a model which predicts the time based on treatment only. “Df” refers to degrees of freedom. “Sum Sq” is the computed sum of squares. “Mean Sq” is the sum of squares, divided by the degrees of freedom. “F-value” is the F-statistic, which evaluates the ratio between the mean explained, and residual sums of squares. “Pr(>F)” is the p-value of the F-statistic.

F-statistic is much higher, which means the machine used explains most of the variance in the data (the residuals are low).

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
machine	1	151.63	151.63	3158.55	<0.001
Residuals	118	5.66	0.05		

Table 4: ANOVA for a model where we predict the time based on machine only. The “Df” value is now 118 for the residuals, because there are three treatments, but two machines only. The p-value is low, meaning that the variance within machines is much lower than between machines.

I can combine the two, and compute an ANOVA with *both* the machine used, and the treatment used. The result is displayed in Table 5. Here, it shows that the machine used explains most of the variance in the data, but also that *once* we considered it, the treatment adds significant information.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
machine	1	151.63	151.63	8499.08	< 0.001
treatment	2	3.60	1.80	100.76	< 0.001
Residuals	116	2.07	0.02		

Table 5: ANOVA for model where we predict the time based on treatment and machine. The degrees of freedom of the residuals are equal to the number of data points (120), minus the degrees of freedom for other variables, minus 1 (120 - 2 - 1 - 1). Adding the machine to the model reduced the sum of squares of residuals, showing that the effect of treatment is significant (as the p-value on the right, shows).

Lastly, I add interactions. Because of the way I wrote the simulation, I know that they play a small, but significant role. The result is displayed in Table 6.

ANOVA allows to estimate the importance of each factor, but it doesn’t tell us which machine or treatment is the fastest. To do that, the experimenter should

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
machine	1	151.63	151.63	15446.90	< 0.001
treatment	2	3.60	1.80	183.13	< 0.001
machine:treatment	2	0.95	0.48	48.41	< 0.001
Residuals	114	1.12	0.01		

Table 6: ANOVA for model with machine, treament, and the interaction between the two. As expected, the interaction is significant.

look at the coefficients of a linear regression.

Now, to be able to use ANOVA, we've seen that we needed to learn several new concepts: degrees of freedom, sum of squares, and the F-statistics. Georges proposed the use of the ANOVA in Java performance evaluation but agreed that "[...] their output is often non-intuitive and in many cases hard to understand without deep background knowledge in statistics" [GBE07].

Gelman argues that computing the appropriate degrees of freedom for a given experimental design can become difficult for example in the case of unbalanced data [Gel05]. In this example simulation, if the experimenter cannot try every treatment on each machine, what degrees of freedom should she use? In that case, the order in which different factors are considered in the ANOVA influences the result. That is why Gelman argues for the use of hierarchical models instead.

I find the Bayesian approach easier. I used a linear regression and when I wanted to estimate other quantities (the variance of groups of coefficients), I added hyper-parameters.

6 Summary

In this section, I've considered the challenge of the analysis of experimental results. I argued that there are a number of problems with the interpretation of the methods that were (and are) used.

I tried to list some alternatives, which I find promising. Some related to changing the way we report our estimates, but for most of the chapter, I talked about the Bayesian approach to inference. Bayesian statistics have the advantage of being explicit, and inference engines like Turing allow for great flexibility in modeling.

Now, the cost of this approach is that Bayesian inference is slow. For example, least-squares linear regression takes seconds, while Bayesian linear regression with a Monte-Carlo Markov Chain sampler can easily take 10-30 minutes. I suspect that the problem of speed is not connected to what theory we use (Bayesian or frequentist), but is a consequence of the assumptions we make. If *general* Bayesian inference is slow, it is probably possible to make it much faster by restricting our models to specific likelihoods and specific classes of priors.

I have now discussed both the design of tools for collection selection, and the challenges of their evaluation. In the next chapter, I will discuss each paper in this dissertation, and their contributions.

Contributions

In this chapter, I will review each paper in this dissertation, and its contributions.

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

In Paper I, we consider the problem of evaluating the performance of collections using micro-benchmarking, discussed in Chapter IV.

In their study, Costa et al. [Cos+17] 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?

Brainy [Jun+11a] uses a random distribution to generate benchmarks. 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.

For lists, we found ArrayList to yield the best execution time in 90% of our benchmarks, in accordance with previous results. In contrast with previous works, we found LinkedHashSet to be the fastest set 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 during insertions, to gain a lot during iterations over the whole collection.

1.1 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 previous work on micro-benchmarking collections

2 Performance Analysis with Bayesian Inference

In Paper II, we consider the problem of analyzing benchmark data, discussed in Chapter VI. The state of the art for analyzing benchmark results is the Analysis of Variance (ANOVA). ANOVA has several problems.

First, the different columns of an ANOVA table can be difficult to understand: Software engineers are not necessarily familiar with concepts like degrees of freedom, sum of squares, the F-statistic, and p-values. Second, if ANOVA shows which factors are more *important*, it doesn't show the scientist which levels of those factors are higher/lower. ANOVA can tell us that *some* of our optimizations do have a significant impact on performance, but it doesn't tell us *which* optimizations matters most, and it doesn't say if this influence is positive or negative.

In this paper, we present an example of a Bayesian statistical model, to answer common performance engineering questions, such as:

1. What effect does each optimization have on execution time?
2. Is this effect influenced by other things (like just-in-time compilation, or the CPU architecture?)
3. Which factors are most important?

To study the first question, we use a linear regression on the logarithm of the execution time. This allows us to compare *ratios* instead of absolute time differences (which are not as important to performance engineers). To investigate which factors influence the *effect* of an optimization, we use *interactions*. To compare the importance of different factors, we use a Bayesian hierarchical model.

We conclude by arguing that Bayesian statistics are more flexible and more explicit than classical approaches, such as ANOVA. The assumptions of Bayesian models (e.g. if a variable is expected to follow a normal distribution) are explicitly part of the model, and we can easily modify such models to match our statistical assumptions.

2.1 Contributions of the Paper

A Bayesian statistical model for performance analysis, with the following properties:

- We use a log-transformation to easily reason about speedup ratios, instead of absolute time differences.

- We use interactions to determine if the effectiveness of an optimization depends of the context.
- We can measure the importance of different factors using an hierarchical model.

3 Classification-based Collection Selection for Java: Effectiveness and Adaptability

In Paper III, we test our synthetic benchmarks on real-world programs. CoCo, Chameleon and CollectionSwitch used benchmarks from the DaCapo benchmark suite. We ported the Brainy approach to Java, in a tool called Brainy4J. We compared Brainy4J's improvements with a greedy approach, which tries to optimize each allocation site separately. We evaluated both tools on five benchmarks of the DaCapo suite. Brainy4J and greedy search could use a set of 9 Java collections, which were used in either CoCo or CollectionSwitch.

We found that Brainy4J was not effective at optimizing the programs under study, while greedy search found one major optimization for one of our benchmarks. Brainy4J and greedy search did not seem to be as effective as the state of the art, including tools that make static replacements, like Artemis. We suspect that adding more collections to the set of options that Brainy4J can use, could help.

We mention several key challenges we encountered when porting the Brainy approach to Java. For example, the benchmark synthesis model we borrowed from Brainy is biased in favor of ArrayList, HashSet, and HashMap. Our port did not find any cases where LinkedList was the most effective collection. Even if the training data was not biased, tracing collection usage significantly hinders just-in-time compilation, which might introduce bias when Brainy4J decides which collection to use.

We conclude that the Brainy approach is not effective for Java. Benchmarking in Java takes much longer than C++, and getting low-level features to characterize collection usage is difficult. Moreover, we found that the effect of changes was not very sensitive to the choice of CPU architecture: A change on one machine would have roughly the same effect on another.

3.1 Contributions of the Paper

- A port of the full Brainy approach to Java.
- An evaluation on five well-known benchmarks and three difference CPU architectures, where we compare Brainy4J with a ground truth obtained with greedy search.

- A study of the challenges of porting the Brainy approach to Java.

4 Automatic Collection Selection for Java: Comparing Static Approaches with Adaptive Collections

In Paper III, we noticed that greedy search failed to find significant optimizations, except for one benchmark: bloat. However, the state of the art reported speedups of 5% or more for most benchmarks. In Paper IV we review possible causes for this difference.

4.1 Benchmarks

Perhaps these works used a different version of benchmarks, for which there were more opportunities for improvements. We investigate which versions of the DaCapo benchmarks were likely used in CoCo and CollectionSwitch. In total both studies used avrora, bloat, chart, fop, lusearch and h2. Both studies could have used two different versions of the fop benchmark, one of which takes six times as long to run as the other.

To investigate the limits of possible improvements on DaCapo benchmarks, we use our tracing framework to estimate the amount of collection usage in each of the six DaCapo benchmarks used in CoCo and CollectionSwitch. We find that avrora and fop do not use collections for more than 10% of their execution time. However, h2, lusearch and bloat use collections for more than 20% of their execution time. We find no benchmark using spending more that 35% of execution time on collections.

4.2 Evolutions in the Java Collections Framework

Since the publication of CoCo and CollectionSwitch, the Java Collections Framework has evolved. Some optimizations that were effective in CoCo are now part of the standard library. We investigate which JDK was used in each study, and find that CoCo probably used JDK 6, while CollectionSwitch probably used JDK 8. In our experiments, JDK 8 speeds up most benchmarks by approximately 10-15%, which might be due to changes in the collections framework, or to other optimizations.

4.3 Collections Options

Perhaps the limited effectiveness of greedy search in Paper III is due to not having enough collections to choose between. To check if adding more collections can help, we add LinkedHashSet and LinkedHashMap to the set of collections that

greedy search can use. We find that greedy search is now able to speed up bloat by approximately 50%, by switching a HashMap to a LinkedHashMap.

4.4 Replacement Strategy

In Chapter III, I have described three options for replacing collections. Greedy search makes replacements at the allocation site level, CollectionSwitch at the object and method levels, while CoCo makes replacements at the method level. It is possible that making static replacements at the allocation site level is not as effective as making dynamic replacements at the object or method level. To study the impact of adaptive collections, we reproduce CoCo and CollectionSwitch, and compare with greedy search.

To make a fair comparison of the results, we use a Bayesian statistical model, and use the numbers reported in the original studies as priors. We use the Bayesian approach described in Chapter VI, to express the idea that we are in a replication context. Our statistical model therefore expects the speedups reported in the original studies, and will use that information during inference. For example, if we used too little data, our estimates of effects would be nudged towards the previously reported effects.

We could not find evidence that dynamic collection selection significantly outperforms a static approach. In our experiments, CoCo slows bloat down by 5 to 10% and fop by 2 to 5%, and has no other effect on the other benchmarks. CollectionSwitch speeds up bloat by 1 to 5%, and slows down lusearch by 0.5 to 5%. In our case, our evidence overwhelms the priors, showing that our results reliably contradict the original findings. To explain this difference, we highlight a number of challenges we encountered during the replication.

4.5 Selected Sites

Applying a collection selection tool requires to select which allocation sites will be modified. One of the weaknesses of our reproduction is that we do not know which allocation sites were selected in the prior work. We report on possible differences between our approach and the original studies.

To understand the impact of collection site selection, we break down our analysis per allocation site, in two benchmarks: bloat and lusearch. We focus on bloat because it is the benchmark for which we found the most effective optimizations, and lusearch because it is the benchmark for which CoCo and CollectionSwitch were most beneficial. For lusearch, we could not find a site for which replacing the existing collection by a CoCo collection or a CollectionSwitch collection was beneficial.

For each benchmark, we select 20 allocation sites and replace the collection at each of the sites. We test with CoCo collections, CollectionSwitch collections, alternatives present in the JDK.

For bloat, for one allocation site, replacing with a CoCo collection was beneficial, but was detrimental for several other sites. We could not find a site for which CollectionSwitch sped up bloat.

4.6 Contributions of the Paper

- An investigation of collection usage of six DaCapo benchmarks, used in state of the art studies.
- A replication of two adaptive collection tools: CoCo and CollectionSwitch.
- A demonstration of the use of Bayesian inference to analyze experimental data in a replication context.
- An evaluation of adaptive collections compared with a static approach: greedy search.
- A study of the effect of replacements on specific allocation sites.

Threats to Validity

In this Chapter, I will review the threats to the validity of our work. Our work revolves around three replications, none of which succeeded. I know of several differences between our replications and the original works. First I will start by reviewing how these differences may have impacted our results. Second, I will talk about what might make our work less applicable to other contexts.

1 Internal Validity

In Paper III, we see that Brainy4J does not perform as well as the state of the art.

The performance of a collection selection tooling relies on four factors:

1. The collection usage, dictated by the target program and the workload.
2. The selection of promising program location for optimization.
3. The collections that the tool can use in its replacements.
4. Its decision making procedure, which depends of its cost model and data about collection usage.
5. How we evaluate performance of programs, to detect the impact of optimizations.

1.1 Data Imbalance

In Paper III, we synthesized micro-benchmarks to build a training set for our collection suggestion tool. Our training data was imbalanced: some collections were far more successful in our benchmarks than others.

I can see two possible causes for this: it could be due to a legitimate superiority of some collections against alternatives, or issues with our approach to benchmark synthesis.

Benchmark Synthesis

For benchmark synthesis, there are the following aspects to take into account: the size of the generated benchmarks, the methods selected, the arguments provided to such methods, and the type of elements stored in the collection.

Size of Traces To reproduce Brainy, we used traces of length 1000, but there is not much justification for this number. In actual programs, the number of method calls varies a lot between collections. We think that for some collections, such as ArrayMap, short traces would be better, because ArrayMap is efficient when the map stays small, and longer traces increase the probability of insertions in the map.

Selection of Methods The original generation scheme used in Brainy does not use any domain knowledge. As a results, Brainy4J can generate unrealistic benchmarks, for example, calling clear repeatedly, on an already empty collection. We tried to extract domain knowledge by modeling traces of method calls with Markov chains, with little success.

Generation of Method Arguments I know there is at least one case where Brainy4J's benchmark generator could be improved by encoding domain knowledge in the generation of method arguments: prepending to a LinkedList is less costly than for an ArrayList. In our case, it is unlikely that our benchmark generator would test this hypothesis, because it was unlikely to generate long sequences of insertions to the beginning of the list. As a result, our benchmark generator did not find a single benchmark for which a LinkedList was faster than an ArrayList.

Type of Elements Stored in the Collection Brainy4J only stores integers in collections. However, there is evidence that the type of element in the collection matters. For example, in Chapter II, I mentioned that hash tables call a hash function, and tree-based maps used the comparison. Therefore, testing collections with elements that exploit various trade-offs between the cost of hashCode and compare could help.

1.2 Measuring Collection Usage

Impact of Instrumentation

In Paper III we rely on CPU counters for measuring collection usage. In Chapter IV, I have mentioned the "observer effect" of instrumentation: we wanted to trace low-level features because they promised to contain more information about a collection's behavior. However, such low-level instrumentation is affected by the

tracing code itself, and in practice, instrumentation has a significant impact on JIT-compilation.

Tracing Iterations over Collections

In Paper III and IV, we traced method calls to collections, for example, to rank allocation sites by importance. That approach suffers from one weakness: It doesn't trace how iterators on the collections are used. It is possible that we missed a significant part of the time a collection is used, if it only iterated on, since we did not measure that.

1.3 Selecting Allocation Sites

In Paper III, we supposed that the low performance of greedy search could be due how we measured the potential of an allocation site. However, we traced collection usage for each of our sites, and saw that the first 10-20 busiest sites account for most of the collection usage. We also ran greedy search on *every* allocation site of our benchmarks, but we found no important optimizations.

In Paper IV, we tried to reproduce CoCo and CollectionSwitch, but we did not know which sites they selected. For CollectionSwitch, we know that our selection of sites is different than the selection that they used. It is possible that we couldn't reproduce their results because we missed some key allocation sites that they had selected.

1.4 Evaluation

In Paper III We used two approaches for benchmarking. For micro-benchmarking, we used the Java Microbenchmarking Harness (JMH). JMH makes many things easier, but there are still some traps one can fall into, we followed best-practices [Cos+19].

For DaCapo benchmarks, we wrote our own scripts that launch the JAR file. As I said in Chapter II, determining when steady-state performance is reached is difficult. We inspected manually the running times of benchmarks to check if they reached a plateau. It is probably possible to obtain more precise estimates of the effect of changes with more runs per benchmark.

2 External Validity

In this section, I will talk about what threats the generalizability of our work.

2.1 Target Programs

In our articles, we focused on benchmarks from the DaCapo benchmark suite. We analyzed the collection usage of our different benchmarks, and found significant differences between benchmarks. The avrora and chart benchmarks, for example, spends most of the time writing to a file. So there is little potential to optimize them by changing which collections they use. Existing work [Bas+18] has considered taking many projects from GitHub, and running them. This could be a good approach to cover the ways collections are used.

2.2 Collection Types

In Paper III, and IV, we used collections which could store any type of object, which were less likely to break the target program. However, there exist specialized collections for integers and floats [1], which probably perform better.

We also did not tune collections. In some cases, the most effective optimization might be to change the initialization parameters of the collection.

3 Summary

In this chapter, I reviewed the threats to validity of this work.

In Paper III, we've observed that our tool, Brainy4J, sometimes misses important optimizations. I can see two possible causes for this. First, I suspect that the training data it uses for its cost model is not representative of collection usage, in a number of ways. Second, Instrumentation impacted the JVM significantly, for example by hindering JIT-compilation, and we do not know how much of an effect that had on Brainy4J's decision making.

In Paper IV, we could not reproduce results from the original works, possibly because we did not select the same allocation sites.

We restricted ourselves to a set of well-known benchmarks, and used collections that could be swapped easily in such benchmarks. These aspects constraint the generalizability of our work. Other works have found effective optimizations on other programs. Moreover, there exist collections specialized to specific element types, which could be provide speedups, but we haven't tested them.

[1] FastUtils has specialized collections for storing ints and floats https://fastutil.di.unimi.it/

Conclusions and Future Work

1 Collection Selection

1.1 Regression Approaches

I mentioned the possibility of using a regression for collection selection, in which we would predict the execution time, using features about collection usage.

Regression provides a number of advantages over classification, so I think it should be explored further. First, it can tell *how much* an alternative is. Second, using Bayesian methods, we can also estimate the uncertainty of predictions. Third, it can also provide help with the *synthesis* of benchmarks, for example by generating benchmarks for which the uncertainty is high, or facilitating the search for interesting sections of the space. For example, searching for a benchmark which maximises the predicted speedup of LinkedList over ArrayList.

What I am describing here is basically *Bayesian Optimization*. The regression approach to collection selection is clearly an instance of black-box optimization. I have an expensive function (benchmarking a program), and I could use a regression model as a surrogate model. There is a lot of work in this area, but I haven't had the time to explore it.

1.2 Benchmark Selection

In Chapter IV, I said that benchmark selection is not efficient at finding relevant benchmarks. I suspect that a way to make it more efficient would be to update the synthesis model after each benchmark. One option could be to try to avoid making benchmarks similar to the ones that we already tried. Another option would be the opposite: to use existing benchmarks to generate variations of new benchmarks.

1.3 Benchmark Synthesis

I have explored several approaches for generating sequences of methods calls for collections. However, I have not spend much time on the synthesis of the data to *store* in the collection. I suspect that the type of element has a strong impact of collection performance, even though that seems like a big design space to explore.

1.4 Heterogeneity of Collections

There is some overlap in the set of collections that different systems use, but there are also discrepancies, which makes the tools harder to compare with one another. 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 [Cos+17]. 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.

1.5 Performance Metrics

In this work, I focused on reducting execution time. Originally, it was because execution time seemed easy to measure: start a stopwatch when the program starts, stop it when the program ends. As is now clear, the problem of measuring execution time is much more complicated, because many things can influence it, and the cause of fluctuations is not easy to pinpoint.

Many other studies have focused on memory usage instead. Memory usage is seemingly harder to measure exactly, but I suspect that fewer factors influence it.

It is possible that Java programs suffer from more memory bloat than they suffer from execution bloat. Likewise, energy usage is a performance metric that has not been considered as much as memory usage and execution time.

2 Benchmarking and Experimental Design

I see some opportunities for future work, at different levels. I will start by future work closest to my work, but also mention more general issues.

2.1 Sequential Experimental Design

One problem with my experiments is that new information does not trigger an update of the experimental plan. In other words, I sometimes waited several hours to obtain data that confirms something I already knew.

There is work on sequential experiment design [BHH05], which reduces the cost of experiments by revising the plan as we obtain new results. Similarly, the area of Bayesian experimental design uses Bayesian methods to update the plan.

2.2 Statistical Methods in Software Engineering

Causal inference and Bayesian statistics are applied successfully to answer questions about a variety of topics, from medecine to economics. I see no reason for software engineering not to use these techniques, whether it is to analyze the accuracy of a classifier, or the effect of optimizations on execution time. There is existing effort in this direction, and I consider it a very promising area of research.

3 Conclusions

In Paper I, we started with the Brainy study, by Jung et al. We ported their benchmark synthesis method to Java, and compared the behavior of collections from the Java Collections Framework with results by Costa et al. [Cos+17].

In Paper II, since I was skeptical of established approaches to analyze benchmark data (ANOVA and confidence intervals), we introduced a new Bayesian statistical model to analyse benchmark data, inspired by Gelman [Gel05]

In Paper III, we ported the full Brainy approach to Java, and compared its performance with greedy search on five Java benchmarks. We noticed that both methods give somewhat disappointing results, except for one benchmark, where greedy search finds a remarkable optimization. We highlight the challenges of adapting Brainy. Brainy4J's benchmark synthesis models is biased, and the tool does not work as well as the state of the art. The original Brainy study highlights the importance of the CPU architecture in choosing the right collection for a program. We have seen that this doesn't seem to be the case, for Java programs.

Since tools like CoCo and CollectionSwitch were effective. In Paper IV, we tried to reproduce their results, and compare to greedy search, on six benchmarks. We could not reproduce their results.

4 Final Words

Recently, at a conference, I was asked if I thought that my work would still be relevant, ten years from now. For our work on Brainy4J, I honestly doubt it: Our approach is not fundamentally new, and our results are mostly negative. Moreover, we have seen that reproducing results from ten years ago was difficult: software changes quickly. I do think however, that our proposition to use Bayesian inference for performance analysis (and our models) have a better chance to stand the test of time. In ten years from now, the model will still be valid, and potentially still be

useful. Therefore, If I think that my contribution to automatic collection selection was modest, I think I helped improve the methodology around benchmarking.

Bibliography

[Bar+17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. "Virtual machine warmup blows hot and cold". en. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–27.

[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.

[Bau+85] F.L. Bauer, M. Broy, B. Möller, P. Pepper, M. Wirsing, et al. *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L.* Lecture Notes on Computer Science 183. Berlin: Springer Verlag, Berlin, Heidelberg, New York, 1985.

[Bla+06a] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications.* ACM Press, Oct. 2006, pp. 169–190.

[Bla+16] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, and Andreas Zeller. "The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic

Guide to Assessing Empirical Evaluations”. In: *ACM Transactions on Programming Languages and Systems* 38.4 (Oct. 13, 2016), pp. 1–20.

[Bla+06b] 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.

[BHH05] George E. P. Box, J. Stuart Hunter, and William Gordon Hunter. *Statistics for experimenters: design, innovation, and discovery*. 2nd ed. Wiley series in probability and statistics. Hoboken, N.J: Wiley-Interscience, 2005.

[Coc+20] Andy Cockburn, Pierre Dragicevic, Lonni Besançon, and Carl Gutwin. “Threats of a replication crisis in empirical computer science”. en. In: *Communications of the ACM* 63.8 (July 2020), pp. 70–79.

[Cor07] Thomas H. Cormen, ed. *Introduction to algorithms*. eng. 2nd. ed., 10th pr. Cambridge, Mass.: MIT Press [u.a.], 2007.

[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.

[Cos+17] 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.

[Cos+19] 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.

[CRS23] Noric Couderc, Christoph Reichenbach, and Emma Söderberg. "Performance Analysis with Bayesian Inference". In: *ICSE-NIER '23: Proceedings of the 45th International Conference on Software Engineering: New Ideas and Emerging Results*. 2023.

[CRSne] Noric Couderc, Christoph Reichenbach, and Emma Söderberg. "Classification-based Static Collection Selection for Java: Effectiveness and Adaptability". In: *EASE '23: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering 2023*. June 2023.

[CSR20] Noric Couderc, Emma Söderberg, and Christoph Reichenbach. "JBrainy: Micro-benchmarking Java Collections with Interference (Work in Progress Paper)". In: *ICPE '20: Companion of the ACM/SPEC International Conference on Performance Engineering*. 2020, pp. 42–45.

[Dav10] Matthew J Davis. "Contrast coding in multiple regression analysis: Strengths, weaknesses, and utility of popular coding structures". In: *Journal of data science* 8.1 (2010), pp. 61–73.

[Fer+18] Alberto Fernandez, Salvador Garcia, Francisco Herrera, and Nitesh V. Chawla. "SMOTE for Learning from Imbalanced Data: Progress and Challenges, Marking the 15-year Anniversary". en. In: *Journal of Artificial Intelligence Research* 61 (Apr. 2018), pp. 863–905.

[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.

[FFT19] Carlo A. Furia, Robert Feldt, and Richard Torkar. "Bayesian Data Analysis in Empirical Software Engineering Research". en. In: *IEEE Transactions on Software Engineering* (2019). arXiv:1811.05422 [cs, stat], pp. 1–1.

[FTF22] Carlo A. Furia, Richard Torkar, and Robert Feldt. "Applying Bayesian Analysis Guidelines to Empirical Software Engineering Data: The Case of Programming Languages and Code Quality". In: *ACM Transactions on Software Engineering and Methodology* 31.3 (July 2022), pp. 1–38.

[GXG18] Hong Ge, Kai Xu, and Zoubin Ghahramani. "Turing: a language for flexible probabilistic inference". In: *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. 2018, pp. 1682–1690.

[Gel05] Andrew Gelman. "Analysis of variance—why it is more important than ever". In: *The Annals of Statistics* 33.1 (Feb. 2005).

[GHV21] Andrew Gelman, Jennifer Hill, and Aki Vehtari. *Regression and other stories*. eng. Analytical methods for social research. Cambridge New York, NY Port Melbourne, VIC New Delhi Singapore: Cambridge University Press, 2021.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 57–76.

[GS03] Peter Godfrey-Smith. *Theory and reality: an introduction to the philosophy of science*. Science and its conceptual foundations. Chicago: University of Chicago Press, 2003.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. "Semantic essence of AsmL". In: *Theor. Comput. Sci.* 343.3 (2005), pp. 370–412.

[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.

[HTF09] Trevor Hastie, Robert Tibshirani, and J. H. Friedman. *The elements of statistical learning: data mining, inference, and prediction*. 2nd ed. Springer series in statistics. New York, NY: Springer, 2009.

[HPAD07] John L. Hennessy, David A. Patterson, and Andrea C. Arpaci-Dusseau. *Computer architecture: a quantitative approach*. 4th ed. OCLC: ocm70830951. Amsterdam ; Boston: Morgan Kaufmann, 2007.

[Hoe+14] Rink Hoekstra, Richard D. Morey, Jeffrey N. Rouder, and Eric-Jan Wagenmakers. "Robust misinterpretation of confidence intervals". en. In: *Psychonomic Bulletin & Review* 21.5 (Oct. 2014), pp. 1157–1164.

[Noa] *ISO 25010*.

[JS02] Nathalie Japkowicz and Shaju Stephen. "The class imbalance problem: A systematic study". In: *Intelligent data analysis* 6.5 (2002), pp. 429–449.

[Jun+11a] 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.

[Jun+11b] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. "Brainy: Effective Selection of Data Structures". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 86–97.

[Kah99] Wolfram Kahl. "The Term Graph Programming System HOPS". In: (1999), pp. 136–149.

[KJ13] Tomas Kalibera and Richard Jones. "Rigorous benchmarking in reasonable time". In: *Proceedings of the 2013 international symposium on memory management*. ISMM '13. New York, NY, USA: Association for Computing Machinery, June 2013, pp. 63–74.

[Len+17] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008". en. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. L'Aquila Italy: ACM, Apr. 2017, pp. 3–14.

[LZ74] Barbara Liskov and Stephen Zilles. "Programming with Abstract Data Types". In: *SIGPLAN Not.* 9.4 (Mar. 1974), pp. 50–59.

[LW94] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), pp. 1811–1841.

[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.

[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.

[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.

[McE20] Richard McElreath. *Statistical rethinking: a Bayesian course with examples in R and Stan*. 2nd ed. CRC texts in statistical science. Taylor and Francis, CRC Press, 2020.

[McI68] M. D. McIlroy. "Mass-produced software components". In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968).

[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.

[Mon01] Douglas C. Montgomery. *Design and analysis of experiments*. 5th ed. New York: John Wiley, 2001.

[Mor+16] Richard D. Morey, Rink Hoekstra, Jeffrey N. Rouder, Michael D. Lee, and Eric-Jan Wagenmakers. "The fallacy of placing confidence in confidence intervals". en. In: *Psychonomic Bulletin & Review* 23.1 (Feb. 2016), pp. 103–123.

[Nuz14] Rigina Nuzzo. “Statistical errors: P values, the’gold standard’of statistical validity, are not as reliable as many scientists assume”. In: *Nature* 506.7487 (2014), pp. 150–153.

[Nys21] Robert Nystrom. *Crafting interpreters.* eng. Daryaganj Delhi: Genever Benning, 2021.

[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.

[Our+21] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. “Evaluating the Impact of Java Virtual Machines on Energy Consumption”. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* ESEM ’21. Bari, Italy: Association for Computing Machinery, 2021.

[Pap+21] Alessandro Vittorio Papadopoulos, Laurens Versluis, Andre Bauer, Nikolas Herbst, Joakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, Jose Nelson Amaral, Petr Tuma, and Alexandru Iosup. “Methodological Principles for Reproducible Performance Evaluation in Cloud Computing”. In: *IEEE Transactions on Software Engineering* 47.8 (Aug. 2021), pp. 1528–1543.

[Par72] David L Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, 1972, pp. 479–498.

[PGJ16] Judea Pearl, Madelyn Glymour, and Nicholas P. Jewell. *Causal inference in statistics: a primer.* Chichester, West Sussex: Wiley, 2016.

[Püs+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. “SPIRAL: Code Generation for DSP Transforms”. In: *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”* 93.2 (2005), pp. 232–275.

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2020.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. “An Automatic Technique for Selection of Data Representations in SETL Programs”. In: *ACM Trans. Program. Lang. Syst.* 3.2 (1981), pp. 126–143.

[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.

[Smi90] Douglas R. Smith. "KIDS: A Semi-Automatic Program Development System". In: *Client Resources on the Internet, IEEE Multimedia Systems '99*. 1990, pp. 302–307.

[SP14] Daniele G. Spampinato and Markus Püschel. "A Basic Linear Algebra Compiler". In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '14. Orlando, FL, USA: ACM, 2014, 23:23–23:32.

[Szy03] Clemens Szyperski. "Component Technology: What, Where, and How?" In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 684–693.

[Ter+10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting performance data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[Tra+22] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. "Towards effective assessment of steady state performance in Java software: are we there yet?" en. In: *Empirical Software Engineering* 28.1 (Nov. 2022), p. 13.

[Vig06] SEBASTIANO Vigna. "fastutil 5.0". In: (2006).

[VK11] Jan Vitek and Tomas Kalibera. "Repeatability, reproducibility, and rigor in systems research". In: *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT '11*. the ninth ACM international conference. ACM Press, 2011, p. 33.

[Wan+22] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. "Complexity-guided container replacement synthesis". en. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1 (Apr. 2022), pp. 1–31.

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

[WSL19] Ronald L. Wasserstein, Allen L. Schirm, and Nicole A. Lazar. "Moving to a World Beyond "$p < 0.05$"". In: *The American Statistician* 73.sup1 (Mar. 2019), pp. 1–19.

[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.

[Xu+10] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. “Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications”. en. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER ’10*. Santa Fe, New Mexico, USA: ACM Press, 2010, p. 421.

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. [Cos+17], 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+11a], 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 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.

Single-Operation Profiles Costa et al.'s CollectionsBench system [Cos+17] 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+11a] 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.

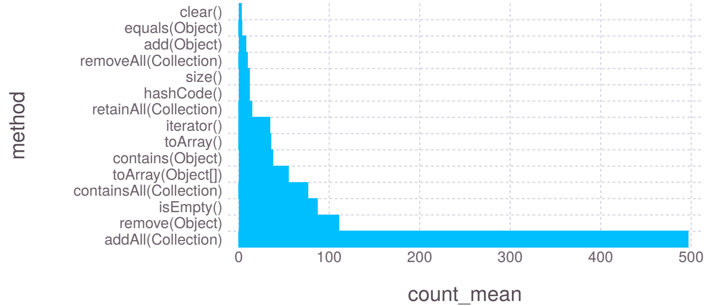


Figure 1: The distribution of method calls for one synthetic benchmark

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 [Cos+19] 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 4.

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-

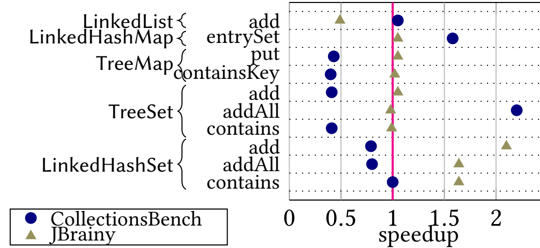


Figure 2: Comparison between speedup predictions by CollectionsBench and JBrainy for various operations

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 Collections-Bench'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.

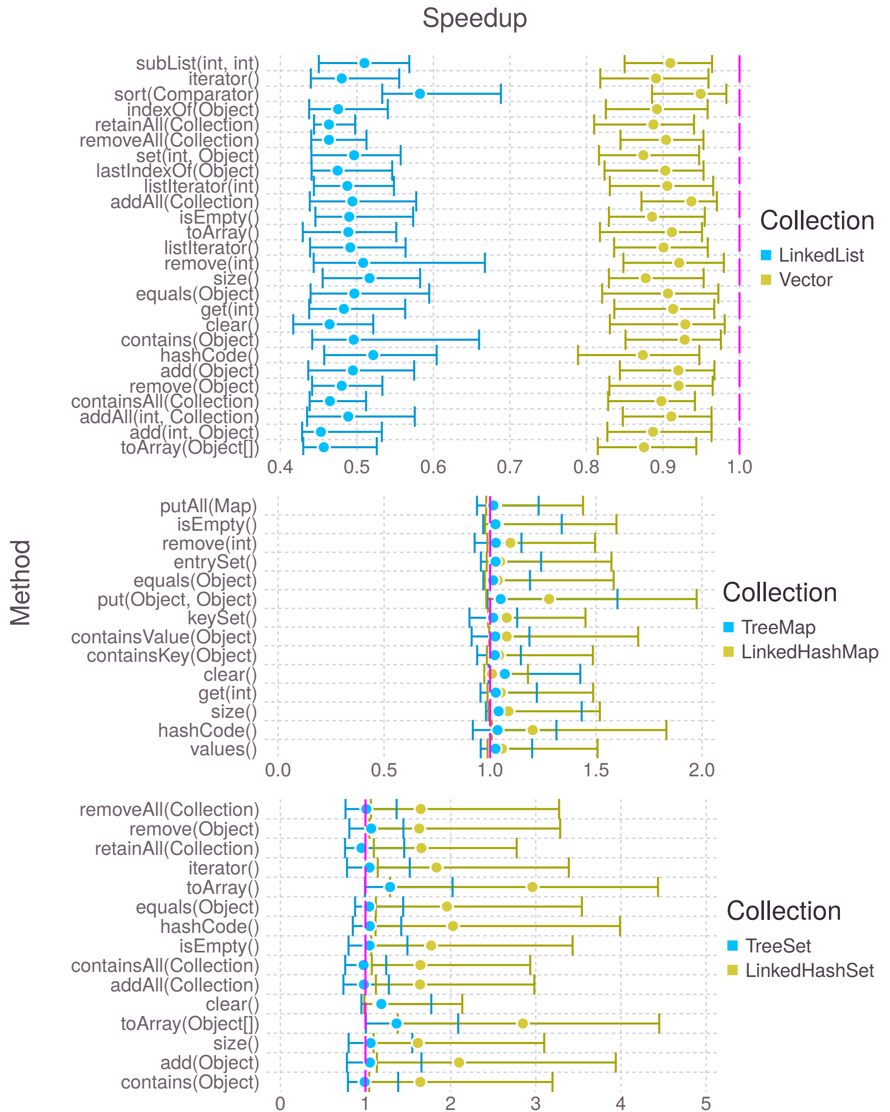


Figure 3: Median speedup of various collections compared to baseline (in magenta), with 25% and 75% quantiles

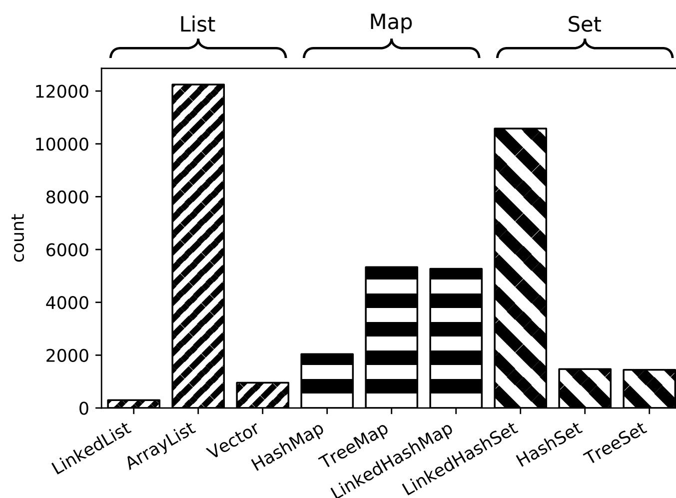


Figure 4: Count of fastest benchmarks depending on the collection class used.

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 NULLing 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 [Cos+17], 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 [Cos+17], 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 [Cos+17], 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+11a], 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+11a]. Using JBrainy and its novel Pólya profiles, we have run an initial performance evaluation experiment following the setup of the CollectionsBench study [Cos+17]. 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.

[Cos+17] 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.

[Cos+19] 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+11a] 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.

Performance Analysis with Bayesian Inference

1 Introduction

Consider the example of a researcher trying to test the efficiency of an optimization on a given program. They might have the following research questions:

Q1 What effect does the optimization have on run time?

Q2 Is this effect influenced by just-in-time compilation?

Q3 Is this effect constant if we use another machine?

Researchers know that many factors can influence measurements and use statistical tools to understand the uncertainty and noise in their results [GBE07]. However, for a non-statistician, comparing these tools is not easy. What is the difference between a Student's t-test and a Welch's test? What assumptions underlie ANOVA? McElreath [McE20] argues that while statisticians know how these tools work and how they may break, scientists often do not, and instead rely on ritualistic recipes. There is even a flowchart of 23 nodes to guide the working scientist to choose the right test statistic [McE20].

For performance analysis, the situation is the same. In 2007, Georges suggested ANOVA and Tukey's honestly significant difference test [GBE07] for benchmark comparison, but warned that their results are difficult to interpret for non-statisticians. In 2011, Vitek and Kalibera [VK11] noted that "[s]tatistical tests or analysis of variance (ANOVA) seem to be used only in papers about how statistics should be applied in the field". In 2021, Papadopoulos [Pap+21] found that cloud performance studies rarely used ANOVA and confidence intervals. In October 2022, of the 399 papers citing Georges' work indexed in Google Scholar, only 71 mention ANOVA.

More generally, Blackburn et al. [Bla+16] cast doubt on the relevance of classical statistical procedures for performance analysis, since their hidden assumption that samples are normally distributed does not always hold.

Recent work by Furia et al. introduced the software engineering community to *Bayesian inference* [FFT19; FTF22] as an alternative to classical *frequentist inference*. They argue that Bayesian inference is more flexible, more explicit, and easier to interpret, and demonstrate regression models for relating programming languages to program execution times and numbers of bugs.

In this paper, we show how to adapt Furia et al.'s work to fit the needs of performance analysis. First, we highlight the connection between Bayesian hierarchical models and ANOVA in an example (Section 2), building on the work by Gelman [Gel05]. The example also demonstrates how performance analysts can use log-space conversion to easily reason about speedup ratios instead of absolute time deltas. We then show how we can adapt a Bayesian model to analyze interactions between different factors that contribute to overall performance (Section 3), without the need for additional statistical tools. We summarize the paper with our view on the strengths and weaknesses of Bayesian statistics as a novel tool for improved performance analysis (Section 4).

2 Example: Collection Selection in Java

As example, consider a scientist who is optimizing a Java benchmark by replacing instantiations of collection classes (e.g., LinkedList) by instantiations of potentially faster collection classes [CA18] (e.g., ArrayList). Our scientist picks the bloat benchmark [Bla+06a] and two promising replacements, which we call *treatments* (following Gelman [GHV21, p. 339]).

Our scientist wants widely applicable results. To investigate if the treatments are machine-dependent [Jun+11b], they run all measurements on three different microarchitectures. To investigate the role of the just-in-time compiler, they measure performance both at program startup ("cold" JVM) and after just-in-time compiler warm-up ("hot" JVM). To account for noise, they use 20 replications (independent JVM runs) for each configuration. With three benchmark variants (one baseline plus two treatments) they obtain 360 data points of the form $\langle M[j], W[j], T[j], \text{running_time}_j \rangle$, representing the machine, JVM warmup, treatment, and run time (respectively) for measurement j.

2.1 Frequentist Inference with ANOVA

To understand the importance of our three "features" *machine*, *warmup*, and *treatment*, we now follow Georges in applying ANOVA [GBE07]: we fit a linear function to input/output values by minimizing error, using (essentially) linear regression. For example, we model the run time of each sample j as the sum of three families of coefficients that describe the effect of the machine ($\alpha_{M[j]}$),

warmup ($\beta_{W[j]}$) and treatment ($\gamma_{T[j]}$), plus some residual noise ϵ_j. Since inputs are discrete, we use a "contrast coding" to embed them (omitted for brevity).[1]

$$\text{running_time}_j = \alpha_{M[j]} + \beta_{W[j]} + \gamma_{T[j]} + \epsilon_j$$

This is the model we obtain if we naïvely plug our data into ANOVA: each coefficient contributes some number of seconds to the run time. In performance analysis, we more commonly work with speedups, so our scientist may prefer to model running time as a *product* $\alpha_{M[j]} \cdot \beta_{W[j]} \cdot \gamma_{T[j]}$. To adapt ANOVA to speedups, we can use log-space coefficients:

$$\log(\text{running_time}_j) = \alpha_{M[j]} + \beta_{W[j]} + \gamma_{T[j]} + \epsilon_j$$

Plugging our data into classical ANOVA, our scientist obtains Table 1.

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
machine	2	30.96	15.48	6054.87	< 0.001
warmup	1	31.44	31.44	12298.82	< 0.001
treatment	2	1.50	0.75	293.86	< 0.001
Residuals	354	0.90	0.00		

Table 1: ANOVA table for training data, generated with R 4.0.3 [R C20]

Intuitively, the table indicates how fit improves as we add more features to a linear regression model. However, to understand this table, the scientist must understand the concepts of *degrees of freedom* (notated Df), *sum of squares* (notated Sum Sq), and the *F-value*. A high F-value shows that the feature explains much of the variance in the data. In our case, warmup matters most, followed by machine. Treatment matters less, but remains significant, with an F-value far above 1. The table says nothing about the efficacy of each treatment, but our scientist can extract these from the regression coefficients γ_t of the linear regression model (Figure 1, left).

We can see that switching to t_2 sped up the program (since the coefficient is less than zero). To compute speedup ratios, we exponentiate differences between coefficients, e.g., the speedup of the *hot* JVM over the *cold* JVM is $\exp(\beta_{\text{hot}} - \beta_{\text{cold}})$. However, we are unaware of a method to compute confidence intervals for the exponential of a random variable.

2.2 Bayesian Inference

We can do the same study with a Bayesian model. Instead of using contrast coding, we can use an array of coefficients which we can index (see Figure 2).

[1] Davis [Dav10] describes six contrast codings; we select "dummy coding", which encodes enumerations with n options as $n - 1$ boolean flags.

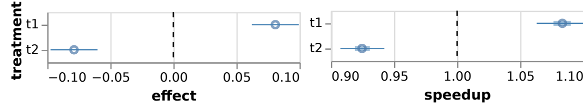


Figure 1: Effect of treatment, frequentist model on the left, Bayesian on the right. The left plot shows the estimate of the effect, with a 95% confidence interval. On the right plot, point estimate is the median of the posterior distribution. Thick error bars are the 50% uncertainty intervals, thin error bars are the 95% uncertainty intervals.

Machine coefficients α_m:	α_{M_1}	α_{M_2}	α_{M_3}
Warmup coefficients β_w:	β_{cold}	β_{hot}	
Treatment coefficients γ_t:	γ_{t_1}	γ_{t_2}	γ_{baseline}

Figure 2: The main coefficients for our Bayesian model

We again model the log running time as depending on machine, warmup state, and treatment, similarly to the frequentist model that implicitly underlies ANOVA, except we explicitely state that residuals are normally distributed:

$$\begin{aligned}\log(\text{running_time}_j) &\sim \text{Normal}(\mu_j, \sigma)\\ \mu_j &= \alpha_{M[j]} + \beta_{W[j]} + \gamma_{T[j]}\\ \sigma &\sim \text{Exponential}(0.1)\end{aligned}$$

The first line states that log(running_time) for data point j is a sample drawn (noted ∼) from a normal distribution with mean μ_j and standard deviation σ. The second line elaborates on μ_j, with no uncertainty: the expected running time μ_j is the sum of the three coefficients that describe experiment j. Since our measured $\log(\text{running_time}_j)$ comes from a distribution rather than an equation, we need no residual term ϵ_j as before. The third line describes a *prior probability* (or *prior*).

Priors in Bayesian inference Priors are key to Bayesian statistics and its main difference to frequentist statistics: for each coefficient, we *must* describe what distribution it comes from. In our model, we have made σ a coefficient with a prior of the exponential distribution with a mean of 0.1 (Figure 4). We thus state that we do not know what σ is, but our initial guess is that it must be ≥ 0, will be ≈ 0.1 on average, and unlikely to be > 0.4. Similarly, we set priors for our α, β and γ coefficients from Figure 2, following best practices from the literature [McE20]. These priors are not *constraints* on the distributions taken by the variables, so the posterior distributions can be quite different, if the data warrants it.

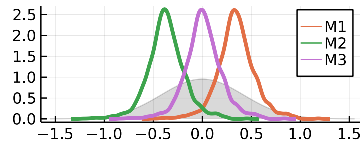


Figure 3: Posterior distributions of the effect of machines. The grey normal distribution in the background is the prior of machine effects ($\sigma = \sigma_{\text{machines}}$).

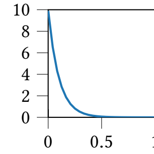


Figure 4: Exponential Distribution with mean at 0.1.

Machines

We set a prior α_m on the mean effect of each machine m, compared to the population mean. For example, we could use the prior Normal$(0.0, 0.4)$, which would mean that approximately 68% (i.e., the 1σ interval around the mean) of all machines yield a overhead between $\exp(-0.4) \approx 0.67$ and $\exp(0.4) \approx 1.49$ compared to the observed mean. While our scientist might be comfortable with selecting the normal distribution as a default [McE20], they may find the choice of 0.4 to be arbitrary. Instead, we can use a model that learns how fast (or slow) machines can be from the data: We introduce a fresh coefficient σ_{machines} that represents the standard deviation *between* coefficients α_m. A large σ_{machines} would show that machines are very different from each other. We thus set:

$$\begin{aligned} \sigma_{\text{machines}} &\sim \text{Exponential}(0.4) \\ \alpha_m &\sim \text{Normal}(0.0, \sigma_{\text{machines}}) \end{aligned}$$

Warmup

For simplicity, we do the same for the distribution of our two warmup coefficients, drawing $\beta_{\text{hot}}, \beta_{\text{cold}}$ from Normal$(0.0, \sigma_{\text{warmups}})$ for a $\sigma_{\text{warmups}} \sim \text{Exponential}(0.4)$. We could alternatively use more accurate priors from prior work.

Treatments

For treatment coefficients γ_t, we choose to be more conservative and set $\gamma_t \sim \text{Normal}(0.0, \sigma_{\text{treatments}})$ with $\sigma_{\text{treatments}} \sim \text{Exponential}(0.1)$. This roughly assigns a 50% probability of the treatment having an effect of less than 10%.

Posteriors Our scientist can now specify the prior beliefs over all coefficients, e.g. in the Turing language [GXG18], and run a Bayesian inference tool. Turing specifications are short: we needed 21 lines of code to encode our model.

Given prior distributions and the data, Turing will compute a list of samples for each coefficient, which describes its likely values, given both prior information and

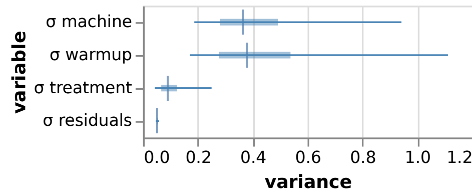


Figure 5: Posterior distributions of the σ values in the model, with medians marked. Horizontal lines show 50% (thick) and 95% (thin) uncertainty intervals.

evidence. We can either inspect posterior distributions directly (plots, statistics, or compute contrasts (differences) between them), or make new predictions, which will return distributions too.

For example, Figure 3 shows the posterior distributions of the effects of machine (α) and the "family" distribution they are sampled from: $\text{Normal}(0.0, \sigma_{\text{machines}})$. Information flows both ways: σ_{machines} influences samples for α, but so does evidence, so σ_{machines} will grow or shrink as the evidence demands. After inference, σ_{machines} informs our estimate of how important machines are, similarly to the ANOVA table.

Figure 5 shows intervals describing the posterior distribution of the various σ parameters in our model. A *p% uncertainty interval* (the analogue to a confidence interval [FFT19]) has a p% percent probability of containing the real value of the effect, and grows wider as p increases. We obtain them by taking percentiles of the posterior. We see the effects of warmup vary most, followed by the effects of the machines. Variation between treatment effects is comparatively low. This analysis is consistent with the ANOVA table.

One might argue that σ_{warmups} does not make much sense. Why estimate the standard deviation of two numbers? In this case, σ_{warmups} also serves as regularization, preventing under/over-fitting. For more details about how it works, we refer to McElreath [McE20, p. 413].

Figure 1 shows the uncertainty intervals of the speedup for each treatment. The two plots look similar, except that we can compute the distribution of $\exp(\gamma_t)$, by exponentiating samples of the posterior. We conclude that t_2 sped up the program by at least 5%.

Discussion To validate our models, we split our data points into training and validation sets, assigning 10 JVM runs of each configuration (50%) to each. Our models predict running times well, with an R2 of 0.98, in both the Frequentist and Bayesian case.[2] Our Bayesian model shows minor disagreement with the ANOVA table, as the former concluded that warmup mattered most, while in the latter warmup and machine matter equally.

[2]In the Bayesian case, we use Gelman's "Bayesian R2" [GHV21, p.170]

Figure 1 shows the effect of the treatment *on average*, answering our scientist's **Q1**, but we still do not know if that effect depends on warmup or machine (**Q2** and **Q3**). To understand *when* the effect is strongest, we add *interactions*.

3 Model with interactions

We now build a model which assumes that the machine used and JVM warmup influence the effect of the treatment. We transform what was array γ into two matrices of coefficients: γ^M and γ^W. $\gamma^M_{m,t}$ is the effect of treatment t on machine m, while $\gamma^W_{\text{cold},t}$ is the effect of treatment t when the JVM is starting up, and $\gamma^W_{\text{hot},t}$ is the effect of t in steady state.

$$\log(\text{running_time}_j) = \alpha_{M[j]} + \beta_{W[j]} + \gamma^M_{M[j],T[j]} + \gamma^W_{W[j],T[j]}$$

When computing the ANOVA table for the second model, we find that the F-value is influenced by the *order* in which the interactions appear in the model (Table 2). When the interaction of treatment and machine comes first, its F-value is 18× higher.[3] The degrees of freedom also changed, but we are unsure why. In the statistics literature, Gelman [Gel05] similarly reports on "many [...] difficulties in understanding and computing ANOVAs" and develops a Bayesian ANOVA framework to resolve them.

	Df	Sum Sq	Mean Sq	F value
machine	2	30.96	15.48	19301.98
warmup	2	30.96	15.48	19301.98
warmup:treatment	**4**	2.09	0.52	**650.38**
machine:treatment	**4**	0.04	0.01	**13.14**
Residuals	348	0.28	0.00	
machine	2	30.96	15.48	19301.98
warmup	1	31.44	31.44	39206.73
machine:treatment	**6**	1.54	0.26	**321.02**
warmup:treatment	**2**	0.58	0.29	**363.97**
Residuals	348	0.28	0.00	

Table 2: ANOVA table for two models, permuting interaction terms

Bayesian inference We modify our Bayesian model from Section 2.2 to support interactions:

$$\begin{aligned}
\log(\text{running_time}_j) &\sim \text{Normal}(\mu_j, \sigma) \\
\mu_j &= \alpha_{M[j]} + \beta_{W[j]} + \gamma^M_{M[j],T[j]} + \gamma^W_{W[j],T[j]} \\
\sigma &\sim \text{Exponential}(0.1)
\end{aligned}$$

[3] This effect disappears if the model retains the mean effect of treatment γ.

	α_{M_1}	α_{M_2}	α_{M_3}	β_{hot}	β_{cold}
$\gamma_{t_{\text{baseline}}}$	$\gamma^M_{t_{\text{baseline}},M_1}$	$\gamma^M_{t_{\text{baseline}},M_2}$	$\gamma^M_{t_{\text{baseline}},M_3}$	$\gamma^W_{t_{\text{baseline}},\text{hot}}$	$\gamma^W_{t_{\text{baseline}},\text{cold}}$
γ_{t_1}	$\gamma^M_{t_1,M_1}$	$\gamma^M_{t_1,M_2}$	$\gamma^M_{t_1,M_3}$	$\gamma^W_{t_1,\text{hot}}$	$\gamma^W_{t_1,\text{cold}}$
γ_{t_2}	$\gamma^M_{t_2,M_1}$	$\gamma^M_{t_2,M_2}$	$\gamma^M_{t_2,M_3}$	$\gamma^W_{t_2,\text{hot}}$	$\gamma^W_{t_2,\text{cold}}$

Figure 6: Main coefficients (excluding the σ coefficients) for our model with interactions, replacing the γ_t coefficients by γ^M and γ^W coefficients.

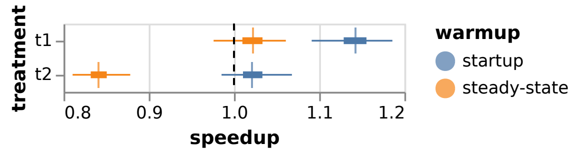


Figure 7: Relative effect of treatment for different warmups. Thick lines denote 50% uncertainty intervals, thin lines denote 95% uncertainty intervals.

Figure 6 visualizes the coefficients of our new model. This change grows our specification to 25 lines of Turing code. For the priors of interactions between machine and treatments (γ^M), we again use a normal distribution with a standard deviation coefficient $\sigma_{\text{tm}} \sim \text{Exponential}(0.1)$, with $\gamma^M_{m,t} \sim \text{Normal}(0.0, \sigma_{\text{tm}})$, and analogously for the interactions between warmup state and treatments (γ^W), we introduce a separate coefficient. We leave α_m and β_w unchanged.

Figure 7 shows how treatments interact with warmup. We can see that treatment t_2 has almost no effect on startup, but the effect becomes much stronger (almost -20%) when the JVM is in steady-state. Similarly, we can see that the effect of t_1 is detrimental on startup (+15%), but that effect disappears after warming up. Figure 8 shows how treatment interacts with machine. For most treatments, we see little difference, except for t_2, which runs slightly faster ($\approx -5\%$) on M_3.

Figure 9 shows a comparison of the σ values in the model: Treatment/machine interaction matters less than treatment/warmup. This matches what we find when comparing individual interactions: Fig. 7 spreads across the range 0.8–1.2, while Fig. 8 shows a flat posterior outside of 0.9–1.1.

Discussion The Bayesian and Frequentist models both match the data closely, with an R2 of 0.99 (vs 0.98 previously). Figure 7 and Figure 8 also answer questions **Q2** and **Q3**. Adding interactions is possible for both the frequentist and the Bayesian approach, but as we have seen, seemingly unimportant factors (the order of variables in the definition of the linear model) could lead us to conclude the effect of machine is much more important than it really is.

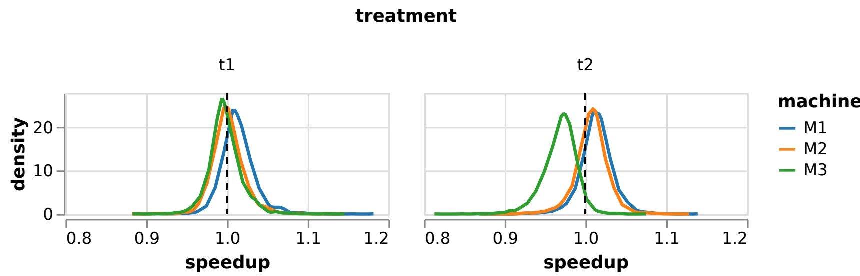


Figure 8: Posterior distribution of the effect of treatment for each machine. We see t_2 runs slightly faster on M_3, but not on M_1 or M_2.

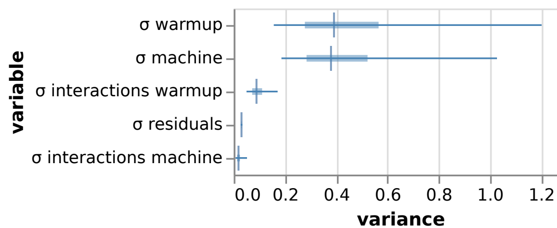


Figure 9: Posterior distributions of the σ values in the model, with medians marked. Horizontal lines show 50% (thick) and 95% (thin) uncertainty intervals.

4 Discussion

We presented a method to investigate benchmark results, and argue that Bayesian methods helped us in several ways:

More flexible: Bayesian inference allows us to start with a simple model, and extend it as we face more research questions. We started with a linear model, and extended it with interactions. ANOVA was not a post-processing step, but was always part of the model.

More explicit: Each of our models explicitly presented their statistical assumptions, there is no need to be aware of hidden, tool-specific assumptions.

More intuitive: We argue that focusing on probability distributions is easier than learning a vocabulary specific to some statistical technique. We illustrate this argument by comparing ANOVA and our hierarchical linear model. We also see posterior distributions as a more informative tool than confidence intervals, but leave the reader to judge.

One drawback of Bayesian inferences is speed. Classical statistical procedures are optimized for common use cases, while Bayesian tools rely on statistically rigorous sampling. Analyzing our model with interactions (Section 3) takes 1ms via frequentist ANOVA, and 353s via Bayesian inference.

We have shown how Bayesian methods agree with frequentist ones in in simple performance analysis tasks, but argue that they are better equipped to handle the challenges outlined by Blackburn et al. [Bla+16]: Bayesian methods support a variety of distributions (multimodal, long-tailed, …), are easy to adapt, and offer considerable flexibility. For instance, we can trivially index any coefficient: we can even partition noise (our σ), to model machine- or treatment-specific variation in noisiness.

Future Plans Bayesian inference offers the opportunity to increase the rigor and precision in how we communicate about performance analysis, and allows us to identify and exploit the statistical distributions behind the factors that impact real-life performance behavior. We expect that exploring the distributions of common factors will shed new light on the *how* and *why* of performance in modern systems.

References

[Bla+06a] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, Oct. 2006, pp. 169–190.

[Bla+16] Stephen M. Blackburn, Amer Diwan, Matthias Hauswirth, Peter F. Sweeney, José Nelson Amaral, Tim Brecht, Lubomír Bulej, Cliff Click, Lieven Eeckhout, Sebastian Fischmeister, Daniel Frampton, Laurie J. Hendren, Michael Hind, Antony L. Hosking, Richard E. Jones, Tomas Kalibera, Nathan Keynes, Nathaniel Nystrom, and Andreas Zeller. "The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations". In: *ACM Transactions on Programming Languages and Systems* 38.4 (Oct. 13, 2016), pp. 1–20.

[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.

[Dav10] Matthew J Davis. "Contrast coding in multiple regression analysis: Strengths, weaknesses, and utility of popular coding structures". In: *Journal of data science* 8.1 (2010), pp. 61–73.

[FFT19] Carlo A. Furia, Robert Feldt, and Richard Torkar. "Bayesian Data Analysis in Empirical Software Engineering Research". en. In: *IEEE Transactions on Software Engineering* (2019). arXiv:1811.05422 [cs, stat], pp. 1–1.

[FTF22] Carlo A. Furia, Richard Torkar, and Robert Feldt. "Applying Bayesian Analysis Guidelines to Empirical Software Engineering Data: The Case of Programming Languages and Code Quality". In: *ACM Transactions on Software Engineering and Methodology* 31.3 (July 2022), pp. 1–38.

[GXG18] Hong Ge, Kai Xu, and Zoubin Ghahramani. "Turing: a language for flexible probabilistic inference". In: *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain.* 2018, pp. 1682–1690.

[Gel05] Andrew Gelman. "Analysis of variance—why it is more important than ever". In: *The Annals of Statistics* 33.1 (Feb. 2005).

[GHV21] Andrew Gelman, Jennifer Hill, and Aki Vehtari. *Regression and other stories.* eng. Analytical methods for social research. Cambridge New York, NY Port Melbourne, VIC New Delhi Singapore: Cambridge University Press, 2021.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications.* OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 57–76.

[Jun+11b] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. "Brainy: Effective Selection of Data Structures". In: *SIGPLAN Not.* 46.6 (June 2011), pp. 86–97.

[McE20] Richard McElreath. *Statistical rethinking: a Bayesian course with examples in R and Stan.* 2nd ed. CRC texts in statistical science. Taylor and Francis, CRC Press, 2020.

[Pap+21] Alessandro Vittorio Papadopoulos, Laurens Versluis, Andre Bauer, Nikolas Herbst, Joakim von Kistowski, Ahmed Ali-Eldin, Cristina L. Abad, Jose Nelson Amaral, Petr Tuma, and Alexandru Iosup. "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing". In: *IEEE Transactions on Software Engineering* 47.8 (Aug. 2021), pp. 1528–1543.

[R C20] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing. Vienna, Austria, 2020.

[VK11] Jan Vitek and Tomas Kalibera. “Repeatability, reproducibility, and rigor in systems research”. In: *Proceedings of the ninth ACM international conference on Embedded software - EMSOFT ’11*. the ninth ACM international conference. ACM Press, 2011, p. 33.

Classification-based Collection Selection for Java: Effectiveness and Adaptability

1 Introduction

Most recent programming languages include a collection framework as part of their standard library (or runtime). For example, Java, C#, Python and Ruby all provide a collection framework. Collections typically implement an Abstract Data Type (ADT), which defines operations that the collection supports, and their semantics. In Java, the List, Map, and Set interfaces describe the associated ADTs, which are implemented in several classes. Collections that implement the same ADT are largely interchangeable.

However, even when collections are *semantically* interchangeable, their respective *performance* characteristics may vary. Liu and Rus [LR09] found that changing a single collection initialization parameter yielded a 17% run time reduction, while Jung et al. [Jun+11a] reduced one benchmark's run time by 77% by swapping one collection datastructure for another.

The performance impact of *collection selection*, i.e., the choice between equivalent collection datastructures, depends on a complex interplay of factors. Developers may be aware of some factors, e.g., data structure size, but others (e.g., cache size) can be hard to grasp. Jung et al. [Jun+11a] showed that CPU architecture had a strong influence on running time for C++ programs, so that the optimal collection(s) for a task would be different for different machines. For Java programs, two additional factors could matter: the JVM implementation [Our+21], and Just-In-Time (JIT) compilation and its "JVM warmup" [Bla+08] effect.

To decide which collection to use, existing approaches for Java use different models of collection performance. CoCo [Xu13] and Chameleon [SVY09] use expert specifications as their models, and observe collection usage (e.g., method call counts) to take decisions. CollectionSwitch [CA18] uses micro-benchmarking to learn the cost of each operation in relation to collection size.

However, if CPU architecture has a strong influence on which collection is best, as Jung et al. report [Jun+11a], our tools should *adapt* to the environment when making suggestions. Their Brainy approach is promising, as it automatically builds a model that factors in CPU behavior. However, the Brainy approach has so far only been realized for C++, and not for any languages whose implementations utilize just-in-time compilation, like Java.

In this paper, we report on our experience of using the Brainy approach for Java, focusing on the following research questions:

RQ$_1$ What are the technical challenges in porting the Brainy approach to Java?

RQ$_2$ How effective is the Brainy approach at optimizing Java programs?

RQ$_3$ How adaptive is the Brainy approach when applied to Java?

RQ$_4$ What obstacles impair the effectiveness of the Brainy approach in Java?

To find answers to these questions, we have ported Brainy to Java in an implementation we call Brainy4J. We have replicated the Brainy approach as far as possible, based on details available in the original paper [Jun+11a] and from correspondence with the authors. In some cases we had to fill in missing details ourselves, or to make larger adjustments to adapt the approach to the needs of the Java environment.

We evaluate our system on five benchmarks from the DaCapo benchmark suite [Bla+06b], on three different CPU architectures. We compare our implementation against a ground truth, constructed from a brute-force approach to collection selection. We find Brainy4J to be less effective at selecting collections than our ground truth. Brainy4J runs faster, but misses the most impactful optimization available. We note weaknesses in the Brainy approach that manifest in our setting, we highlight key challenges to overcoming them. We find Brainy4J highly adaptive to different environments, but find no strong evidence for the significance of CPU architecture details to Java collection selection.

The contributions of this paper are the following:

- Brainy4J, a port of the Brainy approach to Java.
- an evaluation of Brainy4J's effectiveness and adaptability.
- a list of challenges found when porting Brainy. along with an exploration of the design space for overcoming them.

2 Related work

Table 1 lists the most closely related tools. covering static collection replacement [SVY09; Bas+18; Jun+11a] and dynamic replacement via adaptive collections [Xu13; OL13; CA18]. We further divide approaches between those that use hand-written performance models (“Manual”), and those that construct models via machine learning (“Automatic”).

Static collection replacement. Shacham et al.'s 2009 Chameleon system [SVY09] uses traces, heap information, and hand-written rules to select the collections to use. Basios et al's 2018 Artemis [Bas+18] uses a genetic algorithm to optimize the program directly, without a cost model. Instead, it explores possible variants by executing them in the cloud. Our work is a direct port of Jung et al.'s 2011 Brainy [Jun+11a], which uses a machine learning model instead of hand-written heuristics (as in Chameleon) and running purely locally, with knowledge about the underlying architecture (unlike Artemis).

Adaptive collections. Xu [Xu13] and Österlund et al. [OL13] present collections that switch implementations adaptively, based on usage. Österlund et al. present lists that switch between array and hashmap representations based on a state machine that tracks method calls. CoCo [Xu13] minimizes copies by moving elements between different collections on demand. Costa et al.'s 2018 CollectionSwitch [CA18] builds on these tools to introduce smart constructors that select which type to instantiate. Similarly to Brainy, CollectionSwitch learns the relationship between the size of a data structure and its cost of operations via regression on micro-benchmarks. However, Brainy uses more complex models and suggests static changes.

Related approaches. The notion of re-usable software components dates to the early days of software engineering, with McIlroy proposing “catalogues of standard parts” that software engineers should be able to choose from [McI68]. This intuition sees components as units of functionality that should adhere to well-defined interfaces [Szy03; Par72; LZ74; GRS05]. For object-oriented programming, this idea matches Liskov's behavioural contracts for *substitutability* in subtype interfaces [LW94]. We here exploit that parts of the Java Collections Framework follow such contracts, but note that some aspects of their semantics are left to implementers, limiting substitutability.

At the language level, SETL [SSS81] entirely hides the choice of data structure from developers to allow its compiler to effect speedups. For model-based or

	Manual	**Automatic**	**Other**
Static	Chameleon [SVY09]	Brainy [Jun+11a]	Artemis/NSGA-II [Bas+18]
Dynamic	CoCo [Xu13]	CollectionSwitch [CA18]	

Table 1: Overview of collection selection assistance tools.

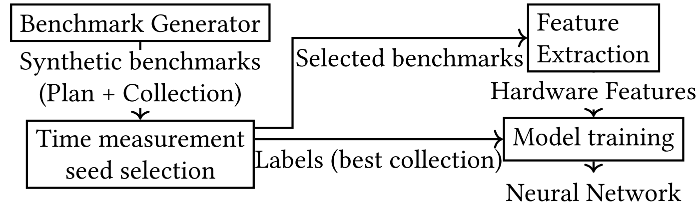


Figure 1: Overview of Brainy's classifier training process

program refinement techniques [Smi90; Kah99; Bau+85] and for domain-specific languages like Spiral [Püs+05], prior research has shown the effectiveness of data structures selection, up to library-level auto-tuning [SP14]. While some modern dynamically typed languages similarly hide data structure details from their users and could use similar techniques, statically typed languages like Java or C++ require users to utilize explicit abstractions, such as the factory APIs exposed by CollectionSwitch.

3 Brainy

We first discuss main aspects of Brainy's approach to data structure selection. Later, we discuss our adaptations to port the C++ implementation to Java (Section 4).

Brainy tries to answer the question: *"what data-structure should I use, when the original data-structure behaves a certain way?"* [Jun+11a]. Its premise is that developers provide Brainy with a program and source location, and it suggests the collection to use.

Brainy's centerpiece is a machine-learned classifier that takes information about (a) the current data structure, (b) collection usage, and (c) the CPU architecture, and proposes a replacement datastructure. To train this classifier (illustrated in Figure 1), Brainy collects training data from *synthetic micro-benchmarks* specific to each ADT. Brainy runs each micro-benchmark with each known datastructure for the ADT, both to determine execution time and to extract *collection usage features*, using instrumentation. To optimize a given target program, Brainy extracts the same types of features from one or more target program executions and asks the classifier for a recommendation.

3.1 Building Brainy's training set

For each ADT, Brainy generates micro-benchmarks to examine how different datastructures behave in different use cases. We identify each micro-benchmark with this ADT and with a *random seed*, from which Brainy synthesizes a **plan** — a sequence of operations to execute on one datastructure instance. Brainy fixes the length of all plans at 1000 methods. It first assigns a random weight to each

ADT operation, then selects the 1000 operations one at a time with a probability proportional to their weight.

Phase I: Time measurements and benchmark selection. For each datastructure that implements the ADT, Brainy measures how long the micro-benchmark takes to run, i.e., how well the datastructure performs for this specific plan. If one datastructure is at least 5% faster than all the alternatives, we say it *wins* the micro-benchmark. If a winner is found, Brainy includes the winner and the micro-benchmark in its training set. Brainy ensures that all datastructures win equally often: once Brainy has collected a fixed number of wins for a given datastructure (1000 in their evaluation), it discards any additional plans in which the same datastructure wins. We call this process *benchmark selection*. We hypothesize that the authors' motivation behind this step is to avoid the *class imbalance problem* [JS02], which in our context means that an imbalance in the number of wins will disproportionately penalize datastructures with fewer wins.

Phase II: Obtaining collection usage features. After Phase I has computed a set of benchmarks with clear "winner" datastructures, Phase II revisits each benchmark to extract collection usage features, i.e., information that Brainy can use to categorize *how* the micro-benchmark is using the datastructure. Brainy instruments the plan and re-executes it with each datastructure to collect features from four different categories: *hardware performance counter* features, specifically branch prediction and L1 cache miss rates, one *memory size* feature, the ratio between each collection element and the cache block size, *operation counts*, for each ADT operation, and three *cost* features, as we describe below.

Cost Features. Choosing the right data-structure requires knowing what operations we want to execute, and how much time these cost. Brainy therefore gathers the cost associated with *insertions*, *deletions*, *searches* (look-ups and iterations) during the benchmark. The authors measure this cost as follows: insertion and deletion (number of data elements moved forwards or backwards by the insertion), and search (number of data elements accessed before finding the element of interest).

Training data. At the end of Phase II, Brainy has collected training data in the form of micro-benchmarks, annotated with the winning data structure, and collection usage features for each datastructure. Brainy trains a family of artificial neural networks[1] with this data: one network per datastructure to replace.

3.2 Classification

For classification, Brainy obtains collection usage features and records the datastructure that it should replace, and queries the appropriate classifier. For example, if the user wants to replace a hash_set, Brainy will feed the features into a classifier specialized for hash_set replacement, the set ADT, and the user's current machine.

[1] We were unable to obtain details about the structure or size of these networks.

3.3 Evaluation of Brainy

The authors apply Brainy to nine datastructures (vector, list, deque, set, AVL_set, hash_set, map, AVL_map, and hash_map) and six ADTs (set, map, list, vector, and order-oblivious list and vector), and train it on two distinct Intel microarchitectures.

To test the training accuracy of Brainy, the authors generate a test set of 1000 benchmarks, one for each datastructure. They report an accuracy between 80% and 90% for the 2006 Intel Core microarchitecture, and an accuracy between 70% and 80% on the 2008 Intel Bonnell microarchitecture (Intel Atom).

The authors also compare Brainy's suggestions and their effect on four benchmark programs, each of them on three different workloads. On average, Brainy reduces execution time by 27% on the Core microarchitecture, and by 33% on the Bonnell (Atom) microarchitecture. All benchmarks report at least 10% execution time reduction, with a maximum of 77% in one case.

4 Porting Brainy to Java: Brainy4J

In this section we address $\mathbf{RQ_1}$ by describing our strategy for implementing Brainy4J (Brainy for Java) and highlighting our design decisions in this process.

4.1 Selection of Collections

Following prior work [SVY09; Xu13; CA18], we selected collections from the Java Collection Framework's List, Set, and Map ADTs. After initial experiments, we concluded that we could not reproduce several of the optimizations from prior work purely with datastructures from the standard library, and added data structures from CollectionSwitch [CA18].

For lists, we selected ArrayList, LinkedList and HashArrayList from CollectionSwitch. For sets, we selected HashSet, TreeSet, and ArraySet from FastUtil [Vig06], as used in CollectionSwitch (Chameleon [SVY09] also reports using an ArraySet). For maps, we selected HashMap, TreeMap and ArrayMap [Vig06].

These data structures are both the ones we consider as *targets* of a transformation, and as *sources*. We additionally configured LinkedHashSet and LinkedHashMap as sources (only). We did not explore transformations e.g. from custom user-defined map implementations to standard library map datastructures.

4.2 Datastructure Adaptability

We automatically replace data structures at the Java bytecode level. To enable replacement, we introduce suitable shared super-interfaces for all affected collections as needed, e.g. LinkedListInterface. Since two data structures may not offer the same different APIs even if they both implement the same ADT,

we also added "universal" adapter subclasses for each data structure that inherit the shared super-interfaces and offer suitable adapter functionality. For example, LinkedList exposes a method pop() that ArrayList does not offer. The adapter subclass ArrayListUniversal, which inherits from ArrayList and implements LikedListInterface, implements this feature. Other methods we expose through default methods in the super-interfaces. Our measurements show that these changes by themselves have no significant effect on execution time.

4.3 Feature Selection and Extraction

In their original study, Jung et al. selected one set of features for each collection using genetic algorithms. In our case, we used the features that they reported as being important for all collections.

To gather features we need to instrument benchmarks. We use our adaptability transformations as the foundation for our tracing framework. For each adapter interface, we synthesize wrapper classes that obtain call counts for each method, and the same hardware performance features used in Brainy (via JNI invocations to the PAPI C library [Ter+10]). The wrappers support concurrent operations via lock-free datastructures and discount recursive calls (e.g., calls from addAll to this.add). This instrumentation gives us two of the four feature categories used in Brainy, excluding the *memory size* and *cost* categories. We do not use the *memory size* feature, since Java's generic collections use boxed element representations. Our minimally invasive instrumentation strategy makes it challenging to obtain *cost* features. Instead, we aggregate cycle counts for four families of operations (insertions, deletions, iterations, look-ups) as proxy metrics. When tracing the target program, we average features over two replications of ten runs, and discard data for the first three iterations. To measure running time and hardware performance counters needed to train our models, we followed best practices in using the Java Microbenchmarking Harness (JMH) [Cos+19].

4.4 Benchmarking and Model Training

To keep a balanced training set, Brainy rejects new benchmarks if they do not add new information. If collection C wins for benchmark b, but Brainy already has enough benchmarks for which collection C wins, b is rejected. In our case, this process was too slow to be practical, so we do not reject benchmarks (Section 7). We therefore explored a fixed number of 1000 seeds per ADT, and restricted exploration to datastructure elements of type Integer.

In their original study, Jung et al. used neural networks. We used random forests with 100 decision trees, which were as effective as neural networks at classification, while making it easy to measure the importance of each feature.

4.5 Allocation Site Selection

In their original study, Jung et al. manually selected an interesting allocation site for evaluating Brainy. However, typical Java programs often have hundreds of relevant allocation sites. To use a systematic, reproducible approach, we selected the ones we expected to be most significant for overall performance. We selected the 10 "busiest" allocation sites. To measure busyness, we instrumented our benchmarks to count the number of operations on each datastructure (including constructor calls), and summarized those per allocation site, counting only the ADTs that we are tracking and excluding allocations within the Java Standard Library.

5 Effectiveness of Brainy4J

To address **RQ$_2$**, we examine how effective Brainy4J is at reducing the running time of Java programs. We structure this exploration around the following research questions:

RQ$_{2.1}$ What are the model characteristics of Brainy4J?

RQ$_{2.2}$ What are the costs in terms of time for using Brainy4J?

RQ$_{2.3}$ How effective is Brainy4J compared to the ground truth?

5.1 Experimental Setup

For our experiments, we need a selection of environments, a selection of Java benchmarks, a configuration of Brainy4J, and a ground truth to compare to.

Selection of Environments. As *environmental factors*, we considered the Java Virtual Machine and hardware configuration. We tested with various Java Virtual Machines but observed no significant differences. Therefore, all machines used OpenJDK 8.0.292 with a JVM heap size of 12 GB. To reduce noise in our measurements, we ran each system with CPU frequency scaling and hyperthreading disabled. Table 2 summarizes the systems we evaluated on.

Selection of Benchmarks and Allocation Sites. We evaluate the effectiveness of Brainy4J on the default workloads of five DaCapo [Bla+06b] benchmarks, selected based on use in prior collection replacement studies (shown in Table 1). For each benchmark we considered the ten busiest allocation sites (Section 4.5) for replacement. Figure 2 shows the distribution of datastructure method calls for the top ten sites for the selected benchmarks. The top ten sites comprise 64% of all calls for fop, 73.3% for bloat, and > 99.8% for avrora, lusearch, and chart. We validated this selection mechanism by collecting the number of CPU cycles spent per allocation site and observed no substantial difference.

System	CPU	Cores	CPU Freq.	Microarch.
Sandy	Intel i7-3820	4×2	3.6–3.8 GHz	Sandy Bridge
Cypress	Intel i7-11700K	8×2	3.6–4.9 GHz	Cypress Cove
Zen3	AMD EPYC 7713P	64×2	2.0–3.675 GHz	Zen 3

System	RAM	OS: Ubuntu	Kernel: Linux
Sandy	16 GiB DDR3-1600	18.04.6 LTS	5.4.0
Cypress	128 GiB DDR4-3200	22.04.01	5.15.0
Zen3	512 GiB DDR4-3200	22.04.01	5.15.0

Table 2: Our benchmarking environments.

Brainy4J Configuration. For *training*, we generated 1000 micro-benchmarks per ADT, with 1000 operations each. For execution time, we used 3 replications (independent JVM runs), 2 warmup iterations lasting 500 ms each, and 5 measurement iterations, lasting 500 ms. Each micro-benchmark took 5ms to run. To extract hardware features, we ran each micro-benchmark 10 times. We provide each sample to the classifier. For *classification*, we used a random forest with 100 trees. Our training data consisted of 9×1000 micro-benchmarks, iterated 10 times, for $90,000$ training samples with 35 features.

Benchmarking. We ran 20 in-process iterations of each benchmark. For fop, this number was not sufficient to reach steady-state, so we ran 150 iterations. We replicated this measurement 20 times. The last 10 runs are used to estimate steady-state performance.

Ground Truth: Greedy Search. To understand how Brainy4J compares against the "best possible solution" given our search space, we performed a limit study, i.e., we estimate the maximum improvement we could hope to obtain.

Since the effort for exploring all possible combinations of replacements is exponential in the number of allocation sites (e.g., 59049 variants for fop just for the top ten allocation sites, each of which would take several minutes to benchmark), we selected the top ten allocation sites (sorted after number of calls) and opt for a 'greedy' strategy: we optimized each allocation site independently, and merged the results to produce an "expected best" candidate (requiring only

		Tools			
		Artemis	Coco	CollectionSwitch	Chameleon
Benchmarks	avrora	x	x	x	
	bloat		x	x	x
	chart		x		
	fop	x	x	x	x
	lusearch		x	x	

Table 3: DaCapo benchmarks used in our work, and their use in prior studies on collection selection.

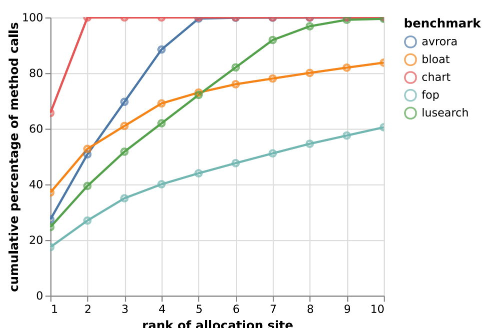


Figure 2: Cumulative collection operations and constructors aggregated for the top ten allocation site. For chart, two allocation sites account for more than 99% of the calls

21 variants per benchmark). To compare two variants, we measured the 95% confidence interval of the difference in running times, in steady-state. We picked the variant for which the confidence interval shows a significant improvement. If not, we keep the original collection. This greedy search would yield the same result as a full exhaustive search (for the top ten allocation sites) if the impact of all datastructure replacements were independent of each other.

5.2 Results

We present the results structured around the research questions presented at the beginning of this section.

$RQ_{2.1}$: Model characteristics Brainy4J. To evaluate our model, we measured accuracy on 10-fold of cross-validation, while making sure the test set contains data only about benchmarks which are not in the training set. The model learns effectively, with an accuracy of more than 85% for all collections and machines. However, this is partially explained by an imbalance problem in the training data. We come back to this issue in Section 7.

$RQ_{2.2}$: Cost of Brainy4J (time). Brainy4J's end-to-end optimization time is much shorter than that of greedy search. Once the classifier is trained, building a variant of a benchmark takes between 60 and 90 seconds, including tracing, classification, and building the optimized program. Greedy search takes much longer, since optimizing a program requires building roughly twenty variants of the program, and running each variant roughly 400 times (20 replications, 20 iterations until steady-state). As a result, greedy search takes between one (fop) and 14 hours (avrora) to optimize one benchmark.

$RQ_{2.3}$: Effectiveness Brainy4J vs. Greedy Search.

Figure 3 show the 95% confidence intervals of average speedup associated with different changes, as well as the speedup of the optimization by Brainy4J. We show

Shorthand	**Collection**
AL	ArrayList
LL	LinkedList
HAL	HashArrayList
HM	HashMap
TM	TreeMap
AM	ArrayMap
HS	HashSet
TS	TreeSet
FUAS	ArraySet

Table 4: The shorthand labels we use for collection names in our notation for changes in figure 3

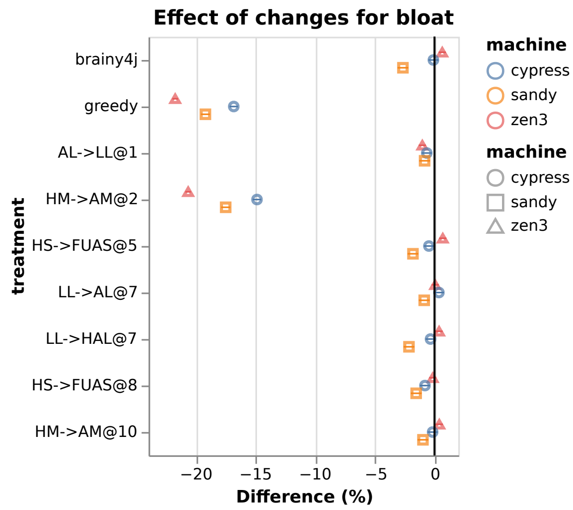
each variant in three environments: for each of our three environments (Table 2), for steady-state performance [Bla+08].

We denote single-change variants as ⟨*from*⟩ → ⟨*to*⟩@⟨*alloc-site*⟩ to identify the datastructures we transform *from* and *to*. For example, AL→LL@1 switches an ArrayList to a LinkedList at the first (highest-ranked) allocation site.

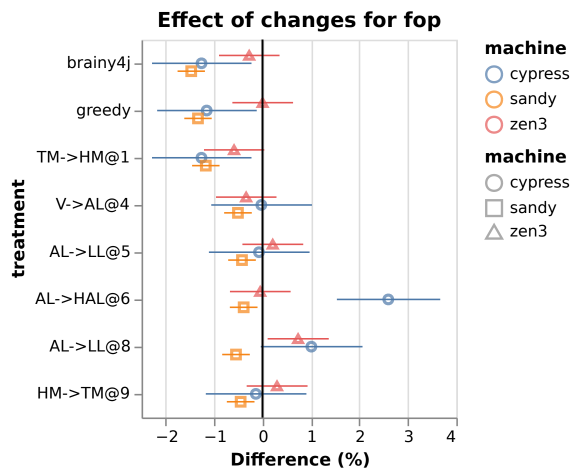
We see that for 3 out of 5 benchmarks, the original program was well optimized, in that neither Brainy4J nor greedy search managed to optimize the benchmark significantly. For chart and avrora, we observed statistically significant changes, but the effects were below 1%. For lusearch, we did not observe any statistically significant improvement. For fop, greedy search found an optimization which yielded between 1 and 2% of speedup on Zen3 and Sandy, and Brainy4J finds it as well. For bloat, greedy search finds one effective optimization (HM→AM@2) that improves the running time by approximately 20% in steady-state by changing a HashMap to an ArrayMap. However, Brainy4J did not suggest this optimization, it either suggests to keep the HashMap, or suggests to use a TreeMap instead. Using a TreeMap does not work as the elements stored in the map are not comparable, so Brainy4J discards that suggestion.

Our plots show the running times of the greedily and Brainy4J-optimized variants, but these include several changes. To understand the effect of one single change on performance, we took each change suggested by either greedy search or Brainy4J, and plotted the running time of a variant making that single change.

Using an ArrayMap is effective for steady-state performance, but has a detrimental effect on performance at startup, we discuss this in Section 6.2.



(a)



(b)

Figure 3: Effect of changes for bloat (a) and fop (b). Points denote the mean speedup, error bars denote the 95% confidence intervals. Greedy search finds an important optimization for bloat, but Brainy4J misses it. For fop, Greedy search and Brainy4J find a small optimization for Sandy and Zen3.

@	From	Brainy4J	M	Greedy	M
avrora					
1	HM	→ HM	S	→ TM	Z
2	LL	→ AL	C	→ AL	Z
2	LL	→ HAL	Z, S	→ AL	Z
3	LL	→ AL	C	→ AL	Z
3	LL	→ HAL	Z,S		S, C
5	LL	→ AL	all		all
6	HS		all	→ TS	Z
7	HM		all	→ AM	Z
8	HM	→ TM	S		all
9	HM		all	→ AM	Z
10	HM	→ TM	S		all
bloat					
1	AL			→ LL	all
2	HM			→ AM	all
5	HS			→ FUAS	C, S
7	LL	→ AL	C		all
7	LL	→ HAL	Z,S	→ HAL	S
8	HS			→ FUAS	C, S
10	HM			→ AM	S

@	From	Brainy4J	M	Greedy	M
chart					
3	AL		all	→ HAL	S
6	AL		all	→ LL	S
7	TM	→ HM	C	→ HM	S
8	AL		all	→ LL	S
9	TM	→ HM	C		all
fop					
1	TM	→ HM	C	→ HM	C, S
3	AL		all	→ LL	C
4	V		all	→ AL	S
5	AL		all	→ LL	S
5	AL		all	→ HAL	C
7	AL	→ LL	all	→ LL	C
8	AL		all	→ LL	S
9	HM	→ TM	S	→ TM	S
lusearch					
4	TM	→ HM	C		all
5	TM	→ HM	C		all
10	HM	→ TM	S		all

Table 5: Transformations used by Brainy4J/greedy search on different machines (**M**), with Z=Zen3, C=Cypress, S=Sandy

Examining the individually selected transformations (Figure 5), we find no matches between the suggestions by Brainy4J and suggestions by Greedy search. None of the changes suggested by Brainy4J has a significant impact on running time. We suspect that the difference between Brainy4J and Greedy search is due to our difficulties in generating a balanced training data-set. We come back to this issue in Section 7.

6 Adaptability of Brainy4J

To address **RQ_3**, we examine how adaptive Brainy4J is to changes in the environment (CPU, JVM) and the configuration (e.g., adding a new collection). We structure this exploration around the following research questions:

$RQ_{3.1}$ What is the effect of the CPU on recommendations?

$RQ_{3.2}$ What is the effect of the JVM on recommendations?

$RQ_{3.3}$ What is the cost of changing environment?

$RQ_{3.4}$ What is the cost of changing collections?

6.1 Experimental Setup

Understanding the effect of the CPU and JVM on recommendations hinges on two aspects: do the recommendations change, and if so, how does this change impact performance? In our case, we can look for recommendations in the training data, and in Brainy4J's recommendations for the DaCapo benchmarks.

To test if Brainy4J adapts to new CPU architectures, we ran the experiments from Section 5 on our three benchmarking machines. If CPU architecture matters, as in the original Brainy study, we should see different suggestions on different machines, both in the training data and in suggestions offered by greedy search.

To test if JVM implementation and warmup impacted collection suggestions, we ran greedy search on our different machines, with four different JVM implementations. To test if reaching steady state impacted the effect of changes, we measured the speedup of a greedy-search optimized variant in both startup and steady-state.

To evaluate the cost of changing environments (e.g. running Brainy4J on a new machine) and changing collections, we report on the time we spent performing such tasks.

6.2 Results

We present the results structured around the research questions.

$RQ_{3.1}$: Effect of the CPU. For our DaCapo benchmarks, we observe some differences between machines, and see some variation in greedy search's suggestions too, but they concern changes that had little effect on performance. For important changes, like for bloat, greedy search suggest the same things on all machines.

For Brainy4J, we observe that suggestions are not the same between different machines, but in practice the changes do not have significant effects. As far as training and classification are concerned, we observed many similarities between different machines.

Figure 4: Importance of different features to the classifier, per machine. BR_MSP_rate is not available on Zen3.

We compared the suggestions given in Brainy4J's training data for Cypress and Sandy, we see that they agree for 93.5% of benchmarks (n = 9000). This could show that CPU architecture does not matter, but it could also indicate that the architectures of both machines are quite similar. We compared on machines with two different CPU architectures: Cypress and Zen3 (one with an Intel CPU, one with an AMD CPU), and see that they also agree for 95.7% of benchmarks (n = 9000).

We looked at which features Brainy4J considers important for classifications, and also observed similarities between machines. Figure 4 shows the five most important features for classifiers specialized in each collection (x-axis), for all three machines. It displays what the classifier considers when analysing the behavior of the collection. The color denotes the importance of the feature, as reported by the random forest classifier. On the y axis, BR_MSP refers to branch mispredictions, while BR_MSP_rate refers to the ratio of misprediction per branch instructions. Notably, the classifier for ArrayList does not consider any feature, because the training data always suggests to keep the ArrayList. Several sets of features are important on all three machines. Cycles spent in methods that insert in the collection ("cycles in insertions" in the figure) and number of calls to Collection.addAll and the corresponding Map operation Map.putAll are important. The number of branch mispredictions is also an important feature for maps and sets. For lists, the number of cycles spent in insertions is the most important feature.

RQ$_{3.2}$: Effect of the JVM. We report all our results in this paper for OpenJDK 8.0.292. We validated the results on OpenJDK 18.0.1.1 and 11.0.12, and on GraalVM 22.1.0.r17 (excluding the bloat benchmark, which crashes[2] on the other JDKs), and observed no significant differences: both suggestions and running

[2]We suspect that it fails due to a problem in a custom class loader, likely due to changes in class loader semantics introduced in Java 9 or later.

startup	steady	interface	proportion
ArrayList	ArrayList	List	**99.4**
ArrayList	HashArrayList	List	0.6
HashMap	HashMap	Map	**94.3**
HashMap	TreeMap	Map	5.1
HashMap	ArrayMap	Map	0.6
HashSet	HashSet	Set	**79.1**
HashSet	TreeSet	Set	20.8
TreeSet	TreeSet	Set	0.01
TreeSet	ArraySet	Set	< 0.01

Table 6: Suggestions of best collection for training data, for startup and in steady state. When the tables agreed, the proportion is in bold.

times are similar.

To study the effect of the JVM, we thus put focus on the impact of JVM warmup on the choice of collections, we examined (a) training data and (b) benchmark variants under both hot and cold JVMs.

First, we used Brainy4J's training data, comparing the suggested collections when only considering startup running time measurements (cold JVM) against steady-state measurements (hot JVM), for all three machines. We found that the suggestions agreed in 90% of all micro-benchmarks. Table 6 shows the suggested data structures and how often each tuple was suggested by the training data. We see that the suggestions are more diverse after warmup, but in the majority of cases, the same collection would be suggested both at startup and after warmup.

Second, we compared the effect of a single collection change, for startup and steady-state. We find that warmup can significantly affects different collections differently. For bloat, one collection change (HM→AM@2) is detrimental at startup but reduces steady-state run time by around 20%.

We find that all four instances in which we introduced HashArrayLists (two in avrora, one each in bloat and lusearch) are reliably detrimental at startup, but the negative effect vanishes after warmup. HashArrayList is our only datastructure without lazy initialization and depends on the Eclipse Collections framework, through a nontrivial chain of delegation, so that we expect it to benefit from inlining optimizations more than other datastructures.

$RQ_{3.3}$: Cost of Changing Environment. Switching to a new machine or a new JVM does not require an expert to re-write the performance model, as it might be the case for tools like CoCo and Chameleon [Xu13; SVY09]. It is however necessary to re-train the classifiers. Training Brainy4J takes roughly 30-35 hours, for a training dataset of 9000 benchmarks (1000 seeds, 9 collections). Measuring the best collection for all benchmarks takes approximately 30 hours, and gathering the features takes between 45 and 90 minutes. Training the classifier

takes a few seconds. Measuring the running time of benchmarks takes most of the time, because we need to warm up the JVM before we can measure the running time [Bla+08].

$RQ_{3.4}$: Cost of Changing Collections. If Brainy4J can build a new model of collection performance, it should be relatively easy to add new collections to Brainy4J, comparatively to tools like CoCo, in which an expert would have to modify the performance model. We found that we need no more than a few hours to add a new collection to Brainy4J, and have partly automated this process. Re-running the micro-benchmark suite and training the model is then fully automatic and takes $\approx 30h$ for 1000 seeds. *Removing* a collection requires removing all training data from benchmarks that use this collection, and re-training the model which takes a few seconds.

7 Obstacles to Effectiveness

To address **RQ_4**, we examine interactions between Java and Brainy that may be obstacles to effectiveness. We structure this exploration around the following research questions:

$RQ_{4.1}$ What is the effect of JIT-compilation?

$RQ_{4.2}$ How effective is benchmark selection?

$RQ_{4.3}$ What is the balance of the training data?

$RQ_{4.4}$ Are our micro-benchmarks too short or too long?

$RQ_{4.5}$ Is the type of elements important?

$RQ_{4.1}$: Effect of JIT compilation. A key challenge in adapting Brainy to Java is the increased distance between application code and machine. Brainy depends on measuring dynamic program features, some of which (hardware performance counters and running time) can vary with external factors, like machine architecture, but are also influenced by JIT compilation. The JIT introduces an additional dimension to our models — how “hot” or “cold” the code that we are optimizing is — and substantially affects micro-benchmark evaluation.

First, even if we only focus on “hot” code (which is likely to dominate performance in long-running programs), we need to iterate the same benchmark many times ($3.5s$ per micro-benchmark). As a result, benchmark selection becomes prohibitively expensive. We have experimented with parallel execution but found that even on highly-parallel multicore systems with hyper-threading disabled, concurrent micro-benchmarking significantly altered our measurements. For Brainy, measuring a benchmark’s running time was *fast*, but obtaining features was *slow*, for Brainy4J, both benchmarking and tracing are slow.

Second, we found evidence that our instrumentation interferes with JIT compilation. With tracing enabled, we observed an overhead of anywhere between 4% (lusearch on a cold JVM) and 1300% (fop on a hot JVM, on Zen3). The effect on warmed-up JVMs was generally more pronounced. In three cases, the overhead was greater than 400%, with more than 70% attributable to our general instrumentation overhead (JNI + Java, not counting PAPI). We speculate that these interactions are due to JIT optimizations not triggering. In almost all other cases, the cost for PAPI calls dominates the overhead. While we expect that this overhead causes a nontrivial amount of perturbation, we found that hardware performance counter measurements during benchmark execution remained stable across multiple runs of the program.

$RQ_{4.2}$: Effectiveness of Benchmark Selection. Jung et al. used benchmark selection (Section 3.1) to gather the same amount of training data for all datastructures. In our case, benchmark selection is ineffective. We observed that ArrayList, HashMap and HashSet win far more often than the other collections, so benchmark selection struggles to find good benchmarks for some datastructures. This could indicate that some collections dominate others regardless of context, or that our micro-benchmarks are biased towards a subset of the collections.

To check if benchmark generation was causing the imbalance, we ran benchmark selection with three different benchmark generation schemes. Uniform (choose methods with a uniform distribution), Brainy (our best-effort approximation of Brainy's selection scheme, cf. Section 3.1), and Markov (first learning a Markov chain from traces on fop, lusearch, and avrora, and using it as generator).

Regardless of the benchmark generation scheme, benchmark selection did not find many benchmarks for which ArrayMap won, but greedy search chose ArrayMap as effective replacement twice, for bloat, including (HM→AM@2), the overall most impactful replacement. We suspect that benchmark selection is biased against ArrayMap, since ArrayMap works well on small maps, which we rarely create with a fixed length of 1000 method calls.

We found no synthetic benchmarks for which LinkedList won. This could suggest LinkedList is intrinsically inefficient, but Artemis [Bas+18] found 87 cases where LinkedList was more effective than ArrayList, and reports AL → LL as the most commonly proposed transformation. We assume that Brainy4J's benchmarks fail to show scenarios in which LinkedList shines and conclude that benchmark selection is ineffective in our setting.

We observe that even if benchmark selection were effective, it would be *inefficient*. If the most "unlucky" datastructure averaged one win per 100 micro-benchmarks (cf. zero wins for LinkedList), we would need 100k micro-benchmarks on average to observe 1000 wins for that datastructure. At ~3.5s of CPU time per datastructure and micro-benchmark, examining three datastructures would use over 11.5 days of CPU time, yet discard 97% of all measurements.

$RQ_{4.3}$: Balance of Training Data. Since we found benchmark selection ineffective in Brainy4J, we do not utilize it for benchmarking. As a consequence,

Brainy4J's training data is imbalanced: our training data favors some collections over others (Table 6); for example, LinkedList does not appear at all. Table 6 shows that the same datastructure wins for the vast majority of benchmarks: ArrayList and HashMap win in more than 90% of cases, HashSet wins in more than 70% of cases. Our micro-benchmarks did not find a single case in which LinkedList wins on any of our three machines. We see the synthetic benchmark generation algorithm, the length of benchmarks, and the elements that we store in the collections as possible causes.

RQ$_{4.4}$: Number of Method Calls. We traced how many methods are called for each collection object in the five selected benchmarks. Over two million method calls, 98% of objects have less than 10 method calls, 1% gets between 10 and 1000, and around 120 get much more (up to more than a million calls). Both short traces and long traces are interesting for Brainy4J. Some of the sites we selected allocate few collections that grow very big (long trace), while others allocate many collections that stay small (short trace per collection). Overall, we observe that the number of collection objects that receive 1000 or more method calls is negligible compared to the number of objects that receive ten or fewer calls. We note that this difference may reflect a difference in programming style between the C++ benchmarks that Brainy investigated and the Java benchmarks we examined.

RQ$_{4.5}$: Importance of Element Types. When training Brainy4J, our synthetic benchmarks only store integers in the collections they benchmark. Since we observed that HashArrayList's performance improved with warmup, we suspect that the cost of hashCode and compare (which might get cheaper with JIT compilation) plays an important role in the performance of sets and maps.

8 Threats to Validity

Internal Validity. Jung et al. used genetic algorithms for *feature selection*. We used the features that they reported as being important. It is possible that another selection could improve the results of Brainy4J. The *translation of a plan into a micro-benchmark* could introduce overhead with effects on performance. To prevent this, we generated bytecode for each micro-benchmark. All *benchmarks suffer from tracing*: the program runs slower and the benefit from JIT compilation is smaller. This may have affected the readings Brainy4J used to make a decision. To make our micro-benchmarks comparable with their real-world counterparts, we normalized the hardware features by the total number of cycles, and the software features (number of calls to methods) by the total number of method invocations. However, this *normalization* does not capture the size of the collection, which showed to be important for CollectionSwitch. *Various factors can affect runtime performance when running benchmarks.* To prevent noise from disturbing the measurements, we disabled frequency scaling and hyperthreading and ran 20

replications per run. For micro-benchmarking, we relied on JMH. *Classification accuracy* depends on the split between training and test data. We used 10-fold cross validation and reported statistics on how accuracy varied on different datasets. We also ensured that samples for the training and test sets never came from the same micro-benchmarks.

External Validity. We used three different machines and five benchmarks. Still, adding more machines and benchmarks would help to further generalize the results. Similarly, we focused on types of collections used in the related work [CA18], adding other collections could provide additional speedups.

9 Discussion

Here, we look back at our three research questions:

What are the technical challenges in porting the Brainy approach to Java? (RQ$_1$) In porting the Brainy approach to Java, the main technical challenge was tracing collection usage and CPU behavior during runs without disturbing the measurements. We needed to generate collection classes that support tracing, and devise a scheme to substitute collection classes by tracing collections without breaking the program. Despite our precautions we still observe that tracing has a significant overhead.

How effective is Brainy4J at optimizing Java programs? (**RQ$_2$**) Brainy sped up C++ programs by 10 to 77%. Our greedy search found statistically significant improvements for only one of our five benchmarks, driven by one high-impact replacement. Brainy4J's complex decision making mechanism failed to pick up the most important of these replacements. We expect that there is further potential for improvement: Artemis [Bas+18], another static selection tool, found statistically significant improvements (around 5% run time reduction) for both avrora and fop, though the authors do not report which changes they applied. Artemis considers three additional (concurrent) ADTs and several datastructures that we did not explore here, which may account for the differences. With Chameleon [SVY09], Shacham et al. found even more dramatic improvements, but their baseline is a modified version of the J9 JVM, which makes it difficult to compare their findings to ours. One of their key findings was the importance of lazy datastructure initialization, which has since been added to all datastructures we considered, as of OpenJDK 8, with the exception of HashArrayList. The CoCo [Xu13] and CollectionSwitch [CA18] approaches provide collections that can switch implementations at runtime. CoCo reports improvements on avrora by 11%, bloat by 4%, fop by 16%, and lusearch by 44%, albeit for Jikes RVM running on the Intel Nehalem microarchitecture. CollectionSwitch did not improve avrora's nor fop's running time, but improved bloat's running time by 22%, and lusearch's by 15%. Both CoCo and CollectionSwitch improved lusearch's running time by switching some instances of HashMap to ArrayMap, reducing memory usage for small

maps. Our greedy search confirmed one such optimization for bloat, but similar changes to lusearch did not have the same effect. We speculate that the difference may come from the tools' dynamic nature, or from a different selection of allocation sites.

How adaptive is the Brainy approach when applied to Java? (RQ$_3$) Brainy4J can *adapt to new CPU architectures* and *different JVMs* without manual work. However, we were unable to identify cases in which switching architectures or JVMs affected the optimal decisions of our greedy search in a significant way. By contrast, Brainy reported different optimizations for different architectures in their C++ benchmarks.

For *adapting Brainy4J's selection of datastructures*, we found that adding a new datastructure takes at most a few hours of work to implement adapter subclasses and default operations (Section 4.2). We have partly automated this process but expect that more automation is feasible.

We expect that Brainy4J could be effective at *adapting to hot vs. cold JVM usage*, using a separate model for the latter case. We did not explore this direction but note that it may be significant for Java programs with short run times: the most effective optimization we observed (in bloat) was ineffective on cold JVMs, even incurring a significant slowdown on one machine.

What obstacles impair the effectiveness of the Brainy approach in Java? (RQ$_4$) We found two challenges in adapting Brainy to Java: the composition of the synthetic benchmarks, and the role of JIT compilation in the JVM. From what we see in our investigations (Section 7), the generated synthetic benchmarks struggle to exercise the strengths of some of our data-structures, like LinkedList. We have identified several possible causes for this challenge: the size of the synthetic benchmarks, how we generate plans and method arguments, and our selection of elements to store in the datastructures. Brainy4J generated benchmarks with 1000 method calls, but we observed that real-world collection traces varied a lot in size. Brainy4J only stores integers in collections, and does not model the cost of methods like hashCode, equals or compareTo, though the cost of these methods may be crucial for deciding which collection to select. Lastly, we observed that JIT compilation plays a significant role in the effectiveness of collection changes, and found indications that Brainy4J's tracing instrumentation interferes with JIT optimization.

10 Conclusions

We were unable to find evidence that Brainy is effective when applied to Java. We observe two challenges caused by JIT compilation that reduce the effectiveness of the approach: First, the cost for benchmarking is higher than for C++, which prevented us from using benchmark selection, since it became too expensive to be practical. To adapt the Brainy approach to Java would require a different method

to generate the set of synthetic benchmarks. A better approach could be not to discard benchmarks (since benchmarking is expensive), and instead use the results. For example, by using regression to estimate the cost of running a benchmark.

Second, we suspect that Java JIT compilation is more sensitive to instrumentation than static compilation, especially since we must rely on JNI calls for gathering hardware performance counter data. This would reduce the accuracy of our models for reasoning about the performance of uninstrumented data structures when the JVM is hot. In summary, Brainy4J was not as effective as greedy search, and was less effective than dynamic tools, such as CoCo and CollectionSwitch.

Future Work. To make Brainy4J more effective, a possible approach would be to obtain more information from fewer benchmark runs. Currently, the benchmark generator does not use any feedback to build new benchmarks. For example, if it found a benchmark where LinkedList is very fast, it would not use this information to find other such benchmarks.

One possible direction for future work would be to allow the benchmark generator to take inspiration from existing benchmarks to build new ones. In addition, our collections only contained integers, while hash maps and tree-based maps make heavy use of methods of elements, like compare and hashCode. One possible extension of this work would be to test different types of data for the elements stored in the collections, and the relationship between the cost of hashCode and compare on these elements. Finally, we do not know how tracing interacts with JIT-compilation and this could also be explored further

11 Acknowledgements

This work was funded by Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the 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.

[Bau+85] F.L. Bauer, M. Broy, B. Möller, P. Pepper, M. Wirsing, et al. *The Munich Project CIP. Vol. I: The Wide Spectrum Language CIP-L*. Lecture Notes on Computer Science 183. Berlin: Springer Verlag, Berlin, Heidelberg, New York, 1985.

[Bla+06b] 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.

[Cos+19] 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.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. "Semantic essence of AsmL". In: *Theor. Comput. Sci.* 343.3 (2005), pp. 370–412.

[JS02] Nathalie Japkowicz and Shaju Stephen. "The class imbalance problem: A systematic study". In: *Intelligent data analysis* 6.5 (2002), pp. 429–449.

[Jun+11a] 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.

[Kah99] Wolfram Kahl. "The Term Graph Programming System HOPS". In: (1999), pp. 136–149.

[LZ74] Barbara Liskov and Stephen Zilles. "Programming with Abstract Data Types". In: *SIGPLAN Not.* 9.4 (Mar. 1974), pp. 50–59.

[LW94] Barbara H. Liskov and Jeannette M. Wing. "A Behavioral Notion of Subtyping". In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), pp. 1811–1841.

[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.

[McI68] M. D. McIlroy. “Mass-produced software components”. In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (1968).

[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.

[Our+21] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. “Evaluating the Impact of Java Virtual Machines on Energy Consumption”. In: *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’21. Bari, Italy: Association for Computing Machinery, 2021.

[Par72] David L Parnas. “On the criteria to be used in decomposing systems into modules”. In: *Pioneers and Their Contributions to Software Engineering*. Springer, 1972, pp. 479–498.

[Püs+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. “SPIRAL: Code Generation for DSP Transforms”. In: *Proceedings of the IEEE, special issue on “Program Generation, Optimization, and Adaptation”* 93.2 (2005), pp. 232–275.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. “An Automatic Technique for Selection of Data Representations in SETL Programs”. In: *ACM Trans. Program. Lang. Syst.* 3.2 (1981), pp. 126–143.

[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.

[Smi90] Douglas R. Smith. “KIDS: A Semi-Automatic Program Development System”. In: *Client Resources on the Internet, IEEE Multimedia Systems ’99*. 1990, pp. 302–307.

[SP14] Daniele G. Spampinato and Markus Püschel. “A Basic Linear Algebra Compiler”. In: *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’14. Orlando, FL, USA: ACM, 2014, 23:23–23:32.

[Szy03] Clemens Szyperski. “Component Technology: What, Where, and How?” In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE ’03. Portland, Oregon: IEEE Computer Society, 2003, pp. 684–693.

[Ter+10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. "Collecting performance data with PAPI-C". In: *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[Vig06] SEBASTIANO Vigna. "fastutil 5.0". In: (2006).

[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.

Automatic Collection Selection for Java: Comparing Static Approaches with Adaptive Collections

1 Introduction

Most recent programming languages include a collection framework. The Java Collections Framework, for example, provides a number of collection classes, which developers can use in their programs. To make it easier to swap between different classes, some of these classes implement a common abstract data type. For example, ArrayList and LinkedList both implement the interface List, and so a developer can easily swap one for the other if needed. Both ArrayList and LinkedList are a List, and they share some common functionality.

If they implement a common interface, different implementation classes can differ by their performance characteristics. An insertion of a key-value pair in a HashMap will have a different cost than the same operation on a TreeMap. For developers, these differences provide opportunities for optimizations: depending on how the collection is used, the developer can pick between different implementation classes, the one that minimizes the cost of running the program. Depending on the overall requirements, developers may choose to optimize for different cost metrics, such as execution time, memory usage, or energy consumption.

In practice, however, software developers do not always make optimal decisions. First, they might not know very well the pros and cons of each collection

implementation, and therefore tend to pick the ones they are most familiar with. Second, even if they do know the performance characteristics of each collection, they might not know in full detail how the collections are going to be used [MSS10].

There exist several tools for helping developers pick Java collections. Some suggest replacement at the source code level [Jun+11a; SVY09; Bas+18; Wan+22], we will call them *static collection selection tools.* Others provide smart constructors or collections which use dynamic information to switch between collections at runtime [CA18; Xu13], we will call them *adaptive collections.*

Both approaches have their pros and cons. On the one hand, we can expect dynamic collection replacements reduce execution time more effectively because they have access to runtime data (for example, the size of collection, which we know influences the complexity). On the other hand, static tooling can use any information necessary, and "think" for a longer time, while dynamic optimization tools must make measurements and take decision at run time, inducing additional costs.

So far, there has not been any comparison between the static approach and the dynamic approach. In this paper, we compare the performance improvements provided by implementations of the two approaches.

For the static approach, we implement a greedy collection selection approach, which will serve as our ground truth. Greedy search selects a number of allocation sites in the target program, and looks for the optimal collection for each site. It then combines the locally optimal solutions to produce the optimized program. Compared to the state of the art, it is very slow, but it has the advantage that it is unlikely to *miss* a promising optimization, since it tries every option to every site.

For the dynamic approach, we reproduce the experiments of CoCo [Xu13] and CollectionSwitch [CA18] approaches using their adaptive collections, and compare the performance improvements against greedy search.

Most of these tools have been tested on DaCapo benchmarks [Bla+06b]. Since some of these tools have been published a few years ago (CoCo is from 2013), the speedups that they reported might be invalid now, since the context has changed (e.g. the Java Collection Framework has new optimizations). We try to reproduce their experiments to see if their numbers are still valid, after changes in the Java standard library, CPU architectures and JVMs.

1.1 Research Questions

Our research questions are:

RQ1 What is the best possible improvement on DaCapo benchmarks?

RQ2 How much improvement can we obtain from a static replacement approach?

RQ3 How much improvement can we obtain from adaptive collections, such as CoCo and CollectionSwitch?

RQ4 What role do different program locations play in observed speedups?

1.2 Contributions

Our contributions are as follows:

- We estimate collection usage in six benchmarks from a widely used benchmark suite and show that two of them are unlikely to benefit from optimizations related to collection usage.
- We reproduce CoCo and CollectionSwitch, and report on several challenges to the replication.
- We demonstrate the use a Bayesian statistical model, which allows us to re-use the previously reported results in our statistical analysis.
- We compare the effect of applying CoCo and CollectionSwitch to our six benchmarks with a static approach to collection selection. We find that we cannot reproduce the results from CoCo and CollectionSwitch, while our greedy search approach finds one important optimization.

2 Related Work

2.1 Static Collection Selection

In 2009, Shacham et al. [SVY09] proposed Chameleon, which uses a modified IBM J9 JVM to suggest collection changes, based on tracing and heap information. Chameleon optimizes both running time and memory usage. Their tool can suggest changes, change collections before starting the program, or can switch between collections at runtime. They report on the offline results, and suggest that online replacements were as effective, with the notable exception of the PMD benchmark, which took 6x as long in online mode. We did not consider comparisons Chameleon for this study, since it uses a modified JVM.

Artemis [Bas+18] is a collection selection tool that makes static changes, and optimizes both memory usage and execution time. To choose between collections, it runs many variants of the target program, and compares their execution times.

Cres [Wan+22] is also a static collection selection tool, which operates at the source code level. It uses static program analysis to identify promising replacements, and program synthesis to suggest them. One defining feature of Cres is its ability to replace a collection with another that supports another interface. For example, Cres can replace an ArrayList with a HashSet.

2.2 Adaptive Collections

CoCo [Xu13] and CollectionSwitch [CA18] both provide adaptive collections that can switch implementation at runtime. CoCo trades memory usage for execution time, while CollectionSwitch can optimize either execution time or memory usage. CoCo's collections instantiate several collections at once, and lazily move elements between the different collection implementations. CollectionSwitch, on the other provides both adaptive collections and smart constructors which track the collections they have instantiated, and decide which type of collection should be created based on the behavior of previously created collections.

2.3 Benchmarking

Artemis, CoCo, CollectionSwitch and Chameleon have all been evaluated on benchmarks from the DaCapo benchmark suite [Bla+06b]. The DaCapo benchmark suite is a set of Java benchmarks which has been first released in 2006. Since then, there were two major releases, and the project is now hosted on GitHub.

We could find several different versions of the benchmark suite: As of March 2023, version 9.12-MR1 is the latest official release. It was released in 2018. It is similar to the version 9.12-bach, which was released in 2009 [1]. These versions do not include the benchmarks bloat and chart. We could find versions 2006-10-MR1 released in 2006, and 2006-10-MR2, released in 2007 [2]. Table 1 shows the list of tools and on which benchmarks they have been evaluated.

CoCo and CollectionSwitch do not specify what version of the benchmark they used. The two studies use benchmarks avrora, bloat, chart, fop, lusearch, and h2. avrora and h2 are included since the 9.12-bach version. bloat and chart are only included in the 2006-10-MR2 version. fop and lusearch are included in both 2006 and post-2009 versions. However, the two versions use different versions of fop. The old version takes approximately 800-900ms, while the new version takes 100-200ms. It is possible that CoCo and Chameleon used one of the 2006 versions, and that Artemis used one of the post-2009 versions. CollectionSwitch used benchmarks from both versions. We refer to the 2009 version as fop and the 2006 version as fop-2006.

2.4 Experimental Design and Statistical Methods

In this work, we use structural causal models (SCMs) [PGJ16] to describe the background and our experimental setup. A structural causal model is a directed acyclic graph for which each vertex represents a feature of the domain of study, and each arrow describes a causal relationship: $X \to Y$ means "X directly causes Y".

[1] https://dacapobench.sourceforge.net/news.html
[2] https://sourceforge.net/projects/dacapobench/files/archive/2006-10-MR2/

		Tools			
		Artemis	Coco	CollectionSwitch	Chameleon
Benchmarks	avrora	x	x	x	
	bloat**		x	x	x
	chart**		x		
	fop*	x	x	x	x
	lusearch*		x	x	
	h2			x	
	xalan*	x			
	sunflow	x			
	pmd*	x			x

Table 1: DaCapo benchmarks used in our work, and their use in prior studies on collection selection. We do not know which versions were used by the authors of previous studies. They do not report it explicitly, and they include benchmarks from both. Benchmarks marked with an asterisk were available in the version 2006-10-MR2. Benchmarks marked with two asterisks were only available in that version.

We analyze our experimental data using Bayesian inference. Bayesian reference uses Bayes' rule to make estimations based on a prior distribution and a likelihood. The likelihood specifies the probability of the observed data (also called evidence), given parameters, which we want to estimate. For each of those parameters, we give a prior distribution, which describes the probable values of the parameters, before any evidence is observed. The result of Bayesian inference is a posterior distribution, which represents the likely values of the parameters. It factors both the prior probability and the likelihood, and represents an update of the prior, in the light of the evidence.

In our case, we design our priors based on the numbers reported in the original studies. The prior distribution represents the expected effect of optimization before we see any experimental result. The likelihood will specify how well estimates of the speedups match the observed data.

For an introduction to Bayesian inference in software engineering research, we refer to Furia et al. [FFT19]. For a more detailed introduction to Bayesian inference with mention of structural causal models, we refer to McElreath's approachable textbook [McE20].

3 Background: What Influences Execution Time?

Figure 1 shows an SCM describing the influences at play in our experiments. We measure execution time, but this execution time is affected by features internal to the program (collection used, collection usage), but also external to the program (JVM used, maximum heap size, CPU architecture, etc).

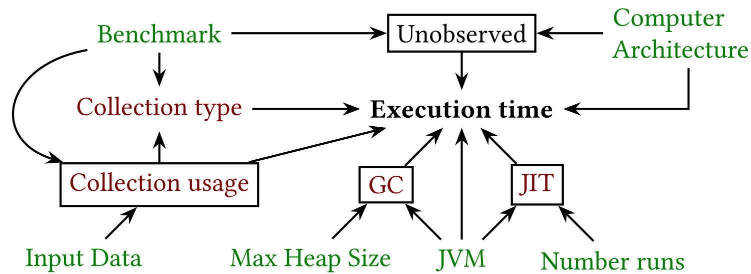


Figure 1: An SCM of various influences on execution time for Java programs. Independent variables are in green, and the experimenter selects them. Execution time is the dependent variable. Variables in red are influenced by other variables. Unobserved variables are variables that we cannot measure directly (at least not without disturbing the measurements), and are shown in boxes. "Unobserved" stands for the influences outside of the graph, which will influence our results nonetheless (noise). That includes thread scheduling effects, the content of caches, and the influence of other processes on execution time.

3.1 Benchmarks

CoCo and CollectionSwitch considered the benchmarks avrora, bloat, chart, fop, lusearch, and h2. For bloat and chart, we use DaCapo version 2006-10-MR1.

For avrora, fop, lusearch and h2, we needed to fork the original repository. To trace collection behavior, we needed to print the tracing results after the benchmark is finished, but the original implementation called System.exit() when the benchmark was finished. We forked the original repository [3] and removed that method call. The DaCapo benchmark suite can be built into one single JAR, but we build each benchmark in its own JAR, to make it easier to replace collection in a benchmark's dependencies.

3.2 Input Data

The DaCapo benchmark suite provides different workloads (small, default, large). For the fop benchmark, only the default workload is available. For the others, CoCo and CollectionSwitch both use the large workload, so we use the same workload, to make comparisons easier.

3.3 Computer Architecture

We will run our experiments on three machines, which are described in table 2. The running time of each benchmark is strongly influenced by the machine that

[3]The software is available at https://github.com/Noricc/dacapobench/tree/v042023

is used. For example, lusearch takes 162ms on average on Zen3, while it takes eleven times longer on Sandy.

The effect of computer architecture on the execution time interacts with the benchmark: If Zen3 is faster on average, it is not the fastest for all benchmarks. For example, avrora takes 14s on Zen3 (which has the most RAM and the highest number of cores), but takes about 8s on Cypress. As a result, we must consider this interaction in our statistical analysis of the results.

System	CPU	Cores	CPU Freq.	Microarch.
Sandy	Intel i7-3820	4 × 2	3.6–3.8 GHz	Sandy Bridge
Cypress	Intel i7-11700K	8 × 2	3.6–4.9 GHz	Cypress Cove
Zen3	AMD EPYC 7713P	64 × 2	2.0–3.675 GHz	Zen 3

System	RAM	OS: Ubuntu	Kernel: Linux	JDK
Sandy	16 GiB DDR3-1600	18.04.6 LTS	5.4.0	8.0.292-open
Cypress	128 GiB DDR4-3200	22.04.01	5.15.0	8.0.292-open
Zen3	512 GiB DDR4-3200	22.04.01	5.15.0	8.0.292-open

Table 2: Our benchmarking environments.

3.4 Java Virtual Machine

As far as we know, the choice of Java Virtual Machine can influence execution time through the garbage collection strategy [Len+17], and JIT-compilation [GBE07].

We suspected that the standard library implementation can also influence execution time. CoCo obtained significant speedups through the use of lazy initialization of collections. This lazy initialization was implemented in the standard library in OpenJDK 7. We could therefore expect lower execution times with OpenJDK 7.

To check this, we run bloat, chart, lusearch-2006 and fop-2006 with OpenJDK 6, 7 and 8, on Zen3[4]. We run 4 replications of 20 runs per benchmark, for each version. We keep the 10 last measurements as steady-state performance. We then measure the standard error as a percentage of the mean. Table 3 shows the standard error for each benchmark, we see that we could detect differences of about 1% in execution time, except for lusearch, which is more unstable. We observe no significant difference between replications (1 to 4), which indicates that our results are stable.

To measure the effect of each variant, we use a linear model where the execution time for sample i, the execution time t_i is the sum of the effect of the benchmark b_i and the effect of the JDK version used v_i, plus some noise ϵ.

$$\log(t_i) = \alpha_{b_i} + \beta_{v_i} + \epsilon_i$$

[4]Post-2009 versions of DaCapo require at least JDK 8 to work, so we use the 2006 versions

benchmark	mean	std	n	std error	std error (%)
bloat	1015.75	48.803	120.0	4.455	0.439
chart	1327.79	38.939	120.0	3.555	0.268
fop-2006	469.43	14.722	120.0	1.344	0.286
lusearch-2006	95.45	51.883	120.0	4.736	4.962

Table 3: Mean, standard deviation, and standard error for benchmark runs, the last column shows the standard error as a percentage of the mean.

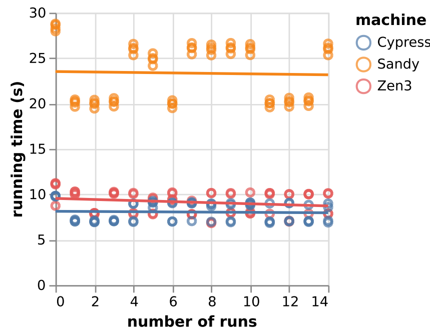


Figure 2: The running time of bloat relative to the number of consecutive runs, on three machines. bloat seems to oscillate between two modes, but the mean running time does not drop significantly over 15 runs.

Including the benchmark used is not absolutely necessary, however, we include it to get more precise estimates of the effect of each JDK version (narrower confidence intervals).

On average, JDK 7 speeds up programs by between 2.3% and 8.0% compared to JDK 6, while JDK 8 is between 7.8% and 13.1% faster than JDK 6. When we detail per benchmark, we see that chart benefits the least from the change of JDK (2.3-2.9% faster with JDK 7, 6.3-6.9% faster with JDK 8). lusearch-2006 benefits the most, although we get larger confidence intervals (2.1-22.8% faster with JDK7, 7.8-27.3% faster with JDK 8).

Just-in-Time Compilation

The effect of Just-in-Time compilation on execution time is extensively documented [GBE07; Tra+22; Bar+17; Xu13; CA18]. As we run the program for longer times, the JVM "warms-up", speeding up the program. Deciding exactly when a program has reached steady-state performance is tricky [Bar+17; Tra+22]. For example, for bloat, we observe that the benchmark does not seem to reach a stable steady-state. The execution times oscillate between two modes, particularly for Sandy. Figure

2 shows the effect. In this case, we choose to consider the program has reached steady-state when there is no significant downward trend.

In this work, we will use 15 runs of each benchmark. For fop and lusearch, we observe that 15 runs is not enough, and increase that number to 150 iterations for fop, and 50 for lusearch. We use the last 10 runs as steady-state performance.

Garbage Collection

Since Java is a garbage-collected language, the JVM pauses programs to re-claim memory, so if there is more garbage collection, programs take more time. Lengauer et al. [Len+17] report in their study of the effect of garbage collection on DaCapo benchmarks that avrora barely spent time doing garbage collection (0.7 to 1.5% of running time), while lusearch spent almost all its execution time performing garbage collection. They have not tested bloat nor chart, since these benchmarks were removed in 2009.

We checked how different DaCapo benchmarks use the heap by manual inspection with VisualVM. We run these benchmarks with OpenJDK version 8, which uses the G1 Garbage collector. We tested a max heap size of 12GB and a max heap size of 1GB.

We tried to run each benchmark with 12GB of heap size, but no benchmarks used more than 2GB. avrora uses approximately 10-30MB of RAM. bloat used a used maximum of 480MB, chart used 670MB, fop used a maximum of 100MB, h2 used 1.5 GB of heap, lusearch used maximum 1.1GB. In our experiments, we therefore use a max heap size of 2GB.

3.5 Collection Types

In this study, we focus on two groups of collections: popular collections from the Java Collections Framework, and collections which were used in existing works, such as CollectionSwitch and CoCo.

HashArrayList is a collection that was introduced by CollectionSwitch. It essentially works like a HashSet, but preserved the insertion order, and is wrapped in a list interface [CA18]. ArrayMap and ArraySet come from the FastUtils library, and we can therefore expect software developers to use them. They are used in CollectionSwitch [CA18]. CollectionSwitch also proposes so-called Adaptive lists, but we only use those in our reproduction of CollectionSwitch.

3.6 Collection Usage

Collections are created at many program locations, so selecting which sites to modify is important. We estimated collection usage by replacing constructor calls in each of our benchmarks (including its dependencies), by wrapper classes that count method invocations. This tracing infrastructure allowed us to count how many collections were created at each program location, and how many method

Interface	Classes
List	ArrayList*, LinkedList*, HashArrayList
Map	HashMap*, TreeMap*, LinkedHashMap*, ArrayMap
Set	HashSet*, TreeSet*, LinkedHashSet*, ArraySet

Table 4: Collections that we included in our greedy search for best collection. Classes marked with an asterisk are sources, which we can replace.

Benchmark	Number of allocation sites
avrora	41
bloat	178
chart	60
fop	653
lusearch	35
h2	18

Table 5: Number of allocation sites found in each DaCapo benchmark

calls were performed on these collections. We call these metrics the "business" of an allocation site, which we use to rank allocation sites by importance. Figure 3 displays the percentage of method invocations for collections, per allocation site, compared to the total for all locations that we traced.

CollectionSwitch

CollectionSwitch selects all allocation sites that generate more than 1000 collections. We implemented the same strategy, but our results show significant differences with the original study. Table 6 shows the difference in number of selected sites between the two studies. We contacted the authors of CollectionSwitch, and determined possible causes of the discrepancies.

Costa et al used JBoss' Byteman to trace collection usage [5], while we implemented our own bytecode processing tool. We inspected their tracing scripts, written in Byteman's domain specific language. They indicate that Costa et al. traced the usage of any class implementing the Collection interface. In our case, we only trace the source collections listed in Table 4. For example, Costa et al. probably trace collection usage for collections like CopyOnWriteArrayList, while we do not.

[5] https://byteman.jboss.org/

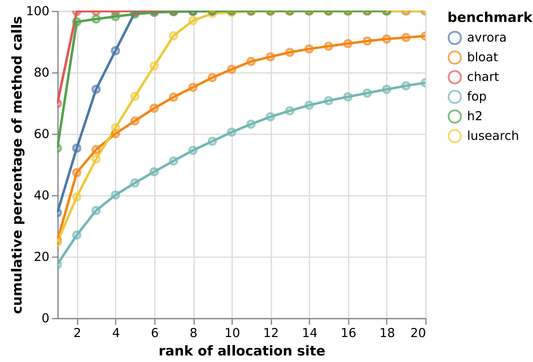


Figure 3: Percentage of method invocations for allocation sites for different benchmarks, sorted by rank. By the 20th allocation site, we have covered more than 80% of method calls on collections for most benchmarks.

Benchmark	CollectionSwitch	Our Method
avrora	7	1
bloat	17	36
fop	15	33
lusearch	12	4

Table 6: Number of sites which instantiate more than 1000 collections, for CollectionSwitch and for our approach. We would expect to find the same results, but they differ substantially.

CoCo

We could not find any indication of what methods were used to select allocation sites in CoCo. We decided to target every allocation site that was possible to replace with a collection from CoCo. To study the impact of our selection of sites, we also tried to compare variants where one single site was changed.

4 Methods

4.1 Experimental Setup

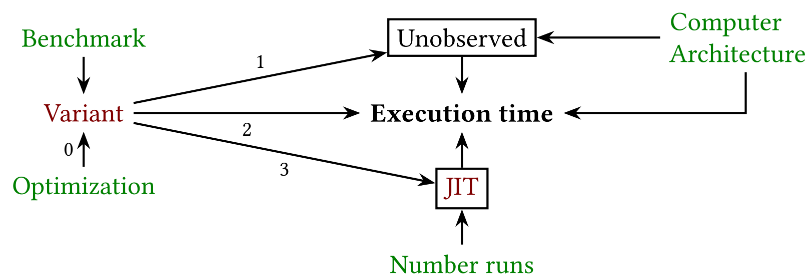


Figure 4: The SCM of influences on execution time in our experiments. "Computer Architecture" has the same meaning as in Figure 1. "Benchmark" now refers to the name of the benchmark. "Variant" refers to the result of applying the optimization to the benchmark and packaging it in a JAR. "Optimization" refers to the optimization applied to the program (e.g. "Original", "CoCo", "Greedy search", etc.). We account for JIT-compilation by running the program several times, which is reflected in the "Number runs" variable.

We describe our experimental design with an SCM, displayed in figure 4. "Benchmark" refers to the benchmark to use as a source program. We select the benchmarks avrora, bloat, chart, fop, lusearch, and h2. "Computer Architecture" refers to the machine used for experiments. We use three machines, described in Table 2. On each machine, we disabled frequency scaling.

In contrast to figure 1, the input data, the JVM and garbage collection are absent from this SCM, because we do not vary these variables. Input data is not included because we test only one workload per benchmark. The JVM and garbage collection are absent because we use OpenJDK version 8.0.292 and a heap size of 2GB for all our measurements.

We are interested in the causal paths between "Optimization" and "Execution time", through "Variant", numbered 0 to 3. Arrow 0 represents the whole process of applying an optimization (e.g. CollectionSwitch) to the benchmark and packaging it in an executable JAR. It includes selecting allocation sites to make replacements.

Arrow 1-3 represent the effects of running the variant and obtaining an execution time. Arrow 3 describes the effect of the program on JIT-compilation. To take JIT-compilation into account, we use 15 iterations of each benchmark, except for fop, where we use 150 iterations, and lusearch, where we use 50 iterations. We keep the 10 remaining at steady-state performance. Arrows 1 and 2 represent the systematic and stochastic effects that can affect execution time. To separate the two, we run 4 replications. For each, we randomize the order in which variants are run.

4.2 Bayesian Inference

To analyze our results for RQ2, RQ3 and RQ4, we will use a Bayesian linear model. Bayesian inference allows us to include information from the previous works of CoCo and CollectionSwitch into our statistical analysis.

Our analysis is implemented using Turing [GXG18]. When there are few alternatives, we will report the full posterior distribution, and when there are many (e.g. when comparing allocation sites for RQ4), we will use credible intervals. Credible intervals are the Bayesian equivalent of confidence intervals. For example, the 90% credible intervals are intervals in which the value of interest has a 90% probability to be.

4.3 RQ1: What is the best possible improvement on DaCapo benchmarks?

For each benchmark, we estimate the total time spent manipulating collections. We use our tracing framework measure the percentage of total time spent in collections. Our tracing framework tracks the time spent in method calls, but will not consider the time spent iterating over collections, since it doesn't track iterators.

4.4 RQ2: How much improvement can we obtain from a static replacement approach?

To study the impact of static collection replacements, we use a greedy approach to collection selection. Given a benchmark, we select several allocation sites, which are candidates for replacements. For each site, there can be several alternatives to the collection instantiated in the original program. Greedy search replaces the collection by each alternative, and compares the confidence intervals of running times. The suggestion is the collection yielding the lowest execution time for the program location. Once we gathered suggestions for each sites, we merge those changes to produce the result. To apply the changes, we use our own bytecode rewriting tool.

We select the 10 allocation sites with most method invocations, for each benchmark. Because the replacements applied depend of the benchmark used, we will run greedy search for each benchmark separately. We have tested with more (including with all allocation sites), but observed no significant improvement past the first 10 sites.

We do this for our three machines, but we will report the mean improvement, since we did not observe large variations between machines (a collection that improves execution time on one machine is likely to also work on another). We will report both the mean effect (across all benchmarks) and the detail for each benchmark.

4.5 RQ3: How much improvement can we obtain from adaptive collections, such as CoCo and CollectionSwitch?

Experiments

For each benchmark, we compare the original variant with a variant optimized by CoCo and CollectionSwitch.

Challenges to Reproduction

CoCo and CollectionSwitch lack information about which allocation sites were selected in benchmarks, what version of the DaCapo benchmarks were used, and what versions of the JVM they used.

The authors of CoCo and CollectionSwitch to not report on which versions of the JDK they used. We found a JAR file for CoCo, for which we checked the bytecode version, which indicates that they used JDK 6. For CollectionSwitch, the Maven file indicates that the authors used JDK version 8. So far, our class replacement tooling relies on Java 8-specific features, so we could not run programs with CoCo, and compare with Java 6.

They also do not report explicitly on which version of the DaCapo benchmark they used. Both tools were tested on avrora, which is only present in the 9.12-MR1 version, but also on bloat, which is only present in the 2006-10-MR1 and the 2006-10-MR2 versions.

4.6 RQ4: What role do different program locations play in observed speedups?

To study the impact of each program location on the performance of an optimization, we study the effect of collection changes to single sites. For each optimization and benchmark, we generate variants of the benchmark that only change one allocation site. We rank the selected allocation sites by "business", and pick the 20 busiest sites. For CoCo and Greedy search, business is defined as the total number of method invocations on collections instantiated at that site. In CollectionSwitch,

business is defined as the number of instantiated collections at that site, so we use the same definition.

For example, if we take the bloat benchmark, select the busiest allocation site, and insert a call to the appropriate CoCo collection at that site. This variant will be called bloat/coco@1. For CollectionSwitch, we apply the same strategy. It should be noted that bloat/coco@1 and bloat/collectionswitch@1 may target different allocation sites, because the definition of "business" is not the same if both cases.

Greedy search already tries several collections for each site, we reuse the experimental data used to generate the greedy search variant. In that case, the variant will be named after the change that was applied. Each change is noted <source>-><target>@<site-rank>. Where source and target are collection types. For compactness, we use shorthands for the names of collections, reported in Table 9. For example bloat/AL->LL@1 is a variant that replaces an ArrayList by a LinkedList at site 1.

Because there are many variants to consider, we focus on bloat and lusearch, and run this experiment on one single machine: Zen3. We chose lusearch because it is the benchmark for which CoCo and CollectionSwitch reported the highest speedups. bloat is the benchmark for which we found the most effective optimizations with greedy search, so we select it as well. We selected one single machine because we observed no major interactions between a machine and the effect of an optimization.

5 Statistical Analysis

5.1 RQ1: What is the best possible improvement on DaCapo benchmarks?

To compute the percentage of time spend in collections, we divide the time spent in collections by the total runtime of the tracing variant (which accounts for the tracing overhead). We use the geometric mean of two replications.

5.2 RQ2: How much improvement can we obtain from a static replacement approach?

Mean Effect of Greedy Search

This model measures the average effect of greedy search, for all benchmarks. We define our model for the mean effect of treatment as follows. The log of the execution time for sample i (noted $\log(t_i)$) is distributed according to a normal distribution, with a mean μ_i and a variance σ, which is common to all samples. μ_i is the sum of:

- The mean execution time of the benchmark b_i used for sample i, and the machine m_i used for sample i, noted $\alpha[b_i, m_i]$.
- The mean effect of the variant of the program v_i used for sample i, noted $\beta[v_i]$.

More formally, the model is:

$$\begin{aligned}\log(t_i) &\sim \text{Normal}(\mu_i, \sigma)\\ \mu_i &= \alpha[b_i, m_i] + \beta[v_i]\end{aligned}$$

α is a matrix of parameters, while β is an array, which is why we use the notation $\alpha[i, j]$. σ is a single coefficient. Because we use a Bayesian model, we need to assign priors to each parameter. We denote the set of all benchmarks as B, the set of machines as M.

$$\begin{aligned}\sigma &\sim \text{Exponential}(0.1)\\ \alpha[b, m] &\sim \text{Normal}(7.5, 2.5), \forall b \in B, m \in M\\ \beta[\text{greedy}] &\sim \text{Normal}(0.0, 0.2)\\ \beta[\text{original}] &\sim \text{Normal}(0.0, 0.2)\end{aligned}$$

We implement this model with Turing, and use four chains of 1000 samples from a NUTS sampler to estimate the posterior of α, β, and σ. The posterior is represented in Turing as a vector of samples. To measure the speedup of the greedily-optimized variant over the original, we use $\beta[\text{greedy}] - \beta[\text{original}]$.

Interaction Between Benchmarks and Greedy Search

In this section, we re-use the previous model, but instead of having one single effect per variant, we have one effect per benchmark and variant. β is not an array, but a matrix of coefficients.

$$\begin{aligned}\log(t_i) &\sim \text{Normal}(\mu_i, \sigma)\\ \mu_i &= \alpha[b_i, m_i] + \beta[v_i, b_i]\end{aligned}$$

We use the similar priors as earlier.

$$\begin{aligned}\sigma &\sim \text{Exponential}(0.1)\\ \alpha[b, m] &\sim \text{Normal}(7.5, 2.5), \forall b \in B, m \in M\\ \beta[\text{greedy}, b] &\sim \text{Normal}(0.0, 0.2)\ \forall b \in B\\ \beta[\text{original}, b] &\sim \text{Normal}(0.0, 0.2)\ \forall b \in B\end{aligned}$$

We implement this model with Turing, and use four chains of 1000 samples from a NUTS sampler to estimate the posterior distributions. To measure the speedup of the greedily-optimized variant over the original, we use $\beta[\text{greedy}, b] - \beta[\text{original}, b]\ \forall b \in B$.

5.3 RQ3: How much improvement can we obtain from adaptive collections, such as CoCo and CollectionSwitch?

We analyze the mean effect (averaged over machines and benchmarks), and the interaction between the benchmark and the adaptive collections used. The study of interactions allows us to see if there are differences in effectiveness, for different benchmarks.

Mean Effect of CoCo and CollectionSwitch

Because there is prior work on CoCo and CollectionSwitch, we can use this prior work to define our priors. We will start by estimating the mean effect of CoCo and CollectionSwitch across many benchmarks, using numbers from the original studies.

The prior work reports only the mean effects for different benchmarks. We write a model assuming they come from a common distribution, with an unknown mean and variance, which we try to estimate. Below is the model for estimating the mean effect of CoCo, based on the speedups observed on different benchmarks in the original study. The model for CollectionSwitch is the same, but with different numbers.

$$
\begin{aligned}
\log(0.96) &\sim \text{Normal}(\mu_{\text{coco}}, \sigma_{\text{coco}}) \\
\log(0.84) &\sim \text{Normal}(\mu_{\text{coco}}, \sigma_{\text{coco}}) \\
\log(0.91) &\sim \text{Normal}(\mu_{\text{coco}}, \sigma_{\text{coco}}) \\
\log(0.66) &\sim \text{Normal}(\mu_{\text{coco}}, \sigma_{\text{coco}}) \\
\log(0.89) &\sim \text{Normal}(\mu_{\text{coco}}, \sigma_{\text{coco}}) \\
\mu_{\text{coco}} &\sim \text{Normal}(0.0, 0.2) \\
\sigma_{\text{coco}} &\sim \text{Exponential}(0.1)
\end{aligned}
$$

To return point estimates, we take the mean of the posterior distributions for μ_{coco} and σ_{coco}, and do the same process for CollectionSwitch. For CoCo, our prior for the mean effect will be $\text{Normal}(-0.148, 0.158)$. For CollectionSwitch, our prior will be $\text{Normal}(-0.044, 0.085)$.

Mean Effect of Adaptive Collections

We use a similar model to the model defined in section 5.2.

$$\begin{aligned}\log(t_i) &\sim \text{Normal}(\mu_i, \sigma)\\ \mu_i &= \alpha[b_i, m_i] + \beta[v_i]\end{aligned}$$

We set the priors based on the results from section 5.3. We denote the set of all benchmarks B, the set of machines M, and the set of variants $V = \{\text{original}, \text{coco}, \text{collectionswitch}\}$.

$$\begin{aligned}\sigma &\sim \text{Exponential}(1.0)\\ \alpha[b, m] &\sim \text{Normal}(7.5, 2.5), \forall b \in B, m \in M\\ \beta[\text{coco}] &\sim \text{Normal}(-0.148, 0.158)\\ \beta[\text{collectionswitch}] &\sim \text{Normal}(-0.044, 0.085)\\ \beta[\text{original}] &\sim \text{Normal}(0.0, 0.01)\end{aligned}$$

We implement this model with Turing, and use two chains of 1000 samples from a NUTS sampler to estimate the posterior of α, β, and σ. The prior for CoCo and CollectionSwitch is not centered at zero, which means that the model expects CollectionSwitch to provide a light speedup, and CoCo a more important speedup.

Interactions Between Benchmark and Adaptive Collections

We re-use the previous model, but instead of one single effect per variant, we have one effect per benchmark and variant.

$$\begin{aligned}\log(t_i) &\sim \text{Normal}(\mu_i, \sigma)\\ \mu_i &= \alpha[b_i, m_i] + \beta[v_i, b_i]\end{aligned}$$

We use the similar priors as earlier.

$$\begin{aligned}\sigma &\sim \text{Exponential}(1.0)\\ \alpha[b, m] &\sim \text{Normal}(7.5, 2.5), \forall b \in B, m \in M\\ \beta[\text{coco}, b] &\sim \text{Normal}(-0.148, 0.158)\ \forall b \in B\\ \beta[\text{collectionswitch}, b] &\sim \text{Normal}(-0.044, 0.085)\ \forall b \in B\\ \beta[\text{original}, b] &\sim \text{Normal}(0.0, 0.01)\ \forall b \in B\end{aligned}$$

5.4 RQ4: What role do different program locations play in observed speedups?

To analyze the results, we also use a Bayesian model. The difference is that we instantiate one model per benchmark and tool (2×3). For each instance, the prior of the effect for each location is centered around the reported effect in the original study. For example, CoCo reported a 4% speedup on bloat, so our prior is Normal(log(0.96), 0.1). Table 7 shows the reported effects for each tool and benchmark.

Benchmark	Tool	Effect
lusearch	CoCo	0.96
bloat	CoCo	0.66
lusearch	CollectionSwitch	0.85
bloat	CollectionSwitch	1.00

Table 7: The effect of collection selection tools on bloat and lusearch, as reported in the prior work

The statistical model infers the effect of applying to tool c to the benchmark b on the centered log execution time. The value $e[b,c]$ is the effect of the collection selection tool c on the benchmark b, as reported in table 7.

$$
\begin{aligned}
\log(t_i) - \overline{\log(t_i)} &\sim \text{Normal}(\mu_i, \sigma) \\
\mu_i &= \beta[v_i] \\
\sigma &\sim \text{Exponential}(0.1) \\
\beta[\text{original}] &\sim \text{Normal}(0.0, 0.01), \forall b \in B \\
\beta[v_i] &\sim \text{Normal}(\log(e[b,c]), 0.1), \forall i \in \{1 \ldots 20\}
\end{aligned}
$$

6 Results

6.1 RQ1: What is the best possible improvement on DaCapo benchmarks?

Table 8 shows the results of the experiments. We see that for avrora and fop, the percentage of time spent manipulating collections is lower than 10%. bloat, lusearch and h2 are the benchmarks which spend the most significant amount of time manipulating collections.

benchmark	startup (%)	steady (%)
avrora	7	4
bloat	68	34
chart	28	17
fop	16	7
h2	36	20
lusearch	34	34

Table 8: Percentage of total running time spent in collections, for each DaCapo benchmark.

6.2 RQ2: How much improvement can we obtain from a static replacement approach?

We observe that the mean effect of greedy search is between 6% and 8% improvement. However, this effect is very dependent on the benchmark. The strongest effect is bloat, for which greedy search managed to obtain a mean speedup between 47-48% speedup. It has a small positive effect on fop, between 0.5% and 2% speedup. However, greedy search fails to obtain a significant speedup for avrora, chart, lusearch, and h2. Figure 5 shows the posterior distribution of the difference in percentages between the original program and the greedily optimized variant.

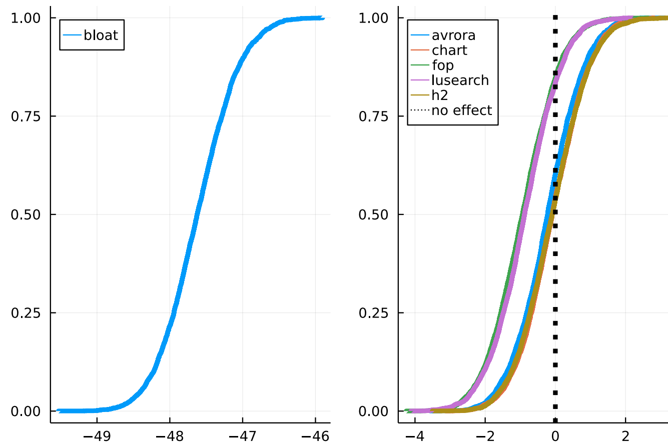


Figure 5: Posterior difference (in %) between the original and a greedily-optimized variant on different benchmarks, shown as a cumulative distribution function. Greedy search finds a major optimization for bloat, and some minor speedups for lusearch and fop.

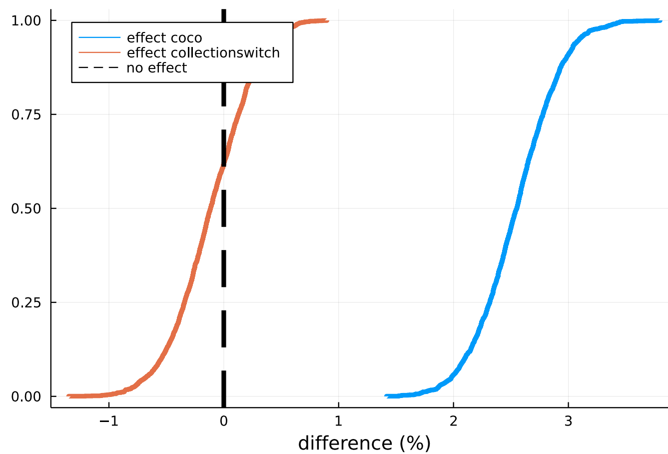


Figure 6: The posterior of the effect of mean effect for CoCo and CollectionSwitch, shown as a cumulative distribution function. CoCo has a effect of +1.5 to +3.5% on average, while CollectionSwitch has an effect of -1 to +1.5% on average.

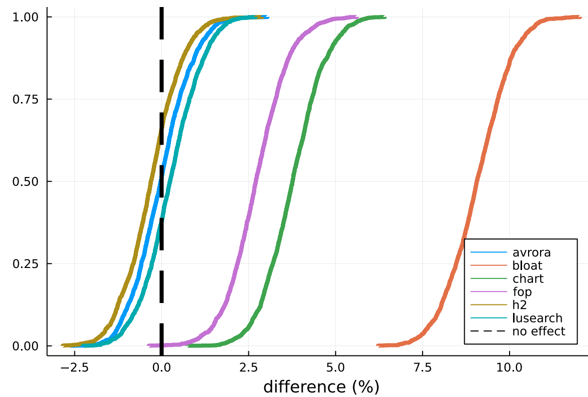
6.3 RQ3: How much improvement can we obtain from adaptive collections, such as CoCo and CollectionSwitch?

Mean Effect of Adaptive Collections

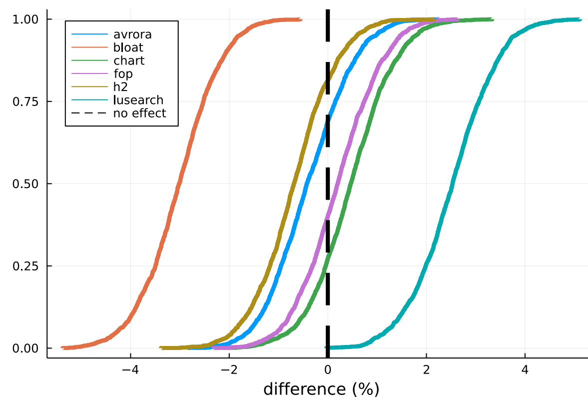
Figure 6 shows the posterior distribution of the mean speedup vs the original, for both CollectionSwitch and CoCo. Neither CollectionSwitch nor CoCo speed up all benchmarks on average. For CoCo, posterior mean effect is far from its prior, which shows that our data contradicts the original findings.

Effect of Adaptive Collection on Each Benchmark

Figure 7a and 7b show the posterior of the effect of adaptive collections on each benchmark, as a cumulative distribution function. When we detail by benchmark, we see that for CoCo significantly slowed down bloat by 5 to 8.5%. For the other benchmarks, we observe no significant effect. CollectionSwitch speeds up bloat by 2.5 to 5.5%, and slows down lusearch by 2.0 to 5.0%. For the other benchmarks, it has no significant effect.



(a) CoCo slows down bloat by 5.5 to 10.5%, and both chart and fop by 2 to 5%. For the other benchmarks, the posterior of the effect does not show strong effects.



(b) CollectionSwitch speeds up bloat by 1 to 5%, and slows downs lusearch by 0.5 to 5.0%.

Figure 7: The posterior effect of adaptive collections, shows as a cumulative distribution function.

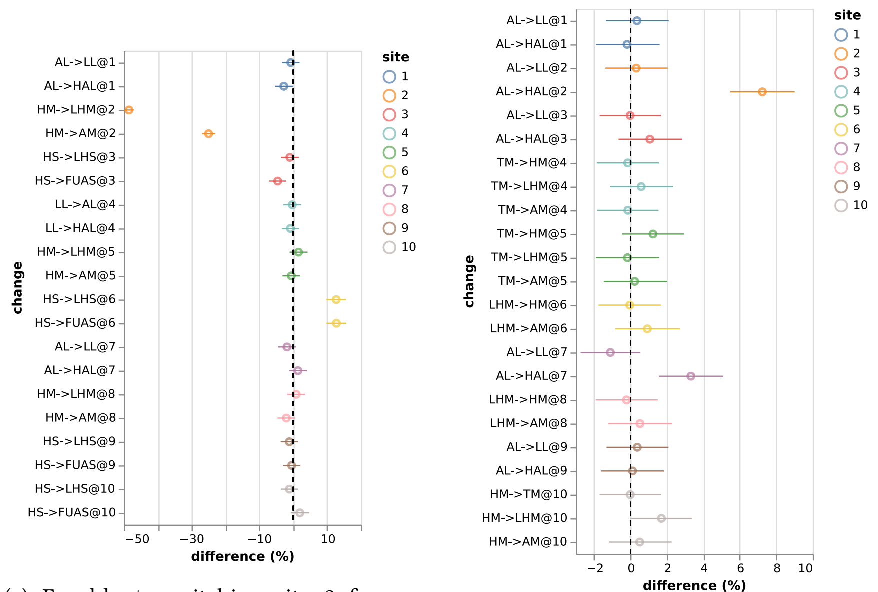
6.4 RQ4: What role do different program locations play in observed speedups?

Greedy Search

Figure 8a shows the effect of different changes applied to the 10 busiest allocation sites in bloat. Allocation site number 2 seems particularly important, since there are two changes to it that incur a major speedup. Figure 8b shows the effect of different changes applied to the 10 busiest allocation sites in lusearch. Greedy search could not find any change for lusearch that incurred any strong positive or negative effect.

Collection	Shorthand
ArrayList	AL
LinkedList	LL
HashArrayList	HAL
HashMap	HM
TreeMap	TM
LinkedHashMap	LHM
ArrayMap	AM
HashSet	HS
TreeSet	TS
LinkedHashSet	LHS
ArraySet	FUAS

Table 9: Shorthand notation for collections, used in figures 8a and 8b



(a) For bloat, switching site 2 from a HashMap to a LinkedHashMap speeds up the program by almost 50%.

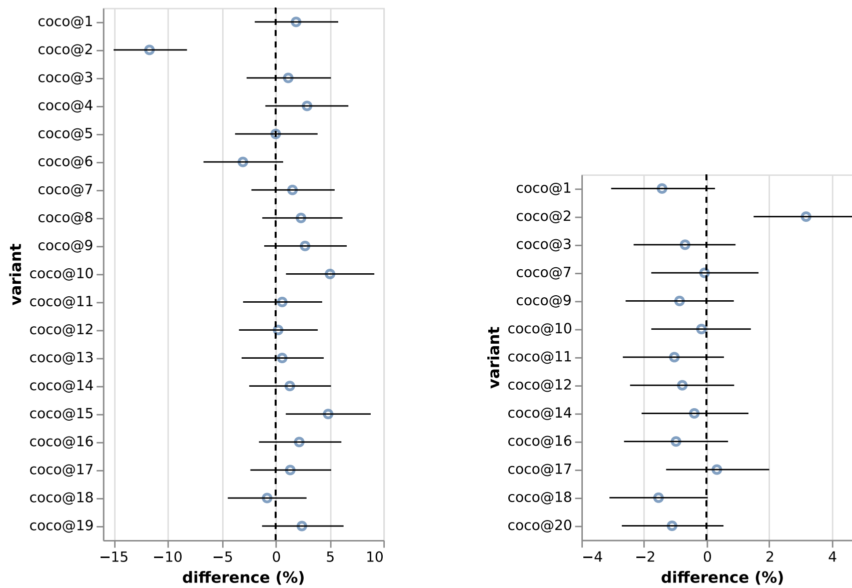
(b) For lusearch, we do not find any change with significant positive effects.

Figure 8: Mean difference and its 90% credible intervals for changes applied to allocation sites in lusearch and bloat.

CoCo

Figure 9a shows the 90% credible intervals for the difference in percentages, when CoCo is applied to different allocation sites of bloat. Among the 20 busies sites, some sites are not used because they were not replaceable by a CoCo collection. We see that CoCo has a positive effect of approximately 10% for allocation site 2, but has a detrimental effect for the several other sites.

Figure 9b shows the 90% credible intervals for the difference in percentages, when CoCo is applied to different allocation sites of lusearch. We see that CoCo has a small positive effect for one of the sites, and a detrimental for another. For most of the sites, we observed no effect.



(a) For bloat, CoCo has a positive effect for one allocation site (2), but is detrimental for at least two others, which might explain the overall negative effect.

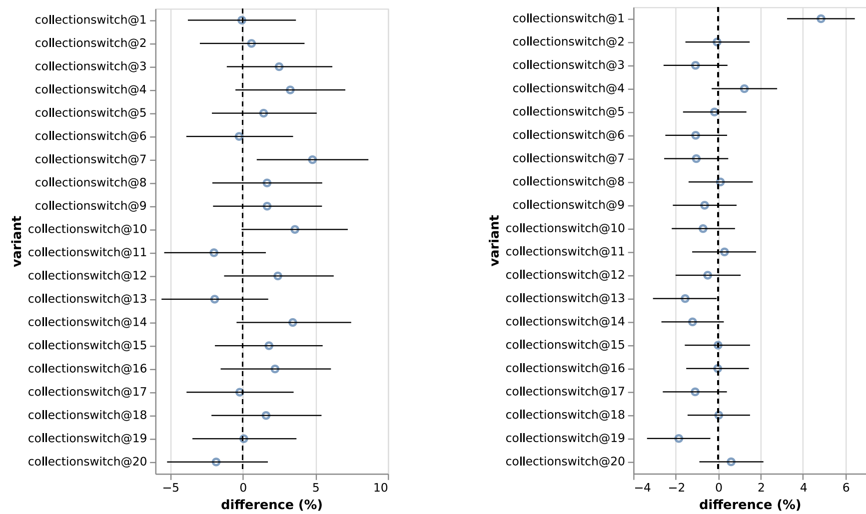
(b) For lusearch, We observed that CoCo was does not have a positive effect on most allocation sites.

Figure 9: Mean difference and its 90% credible intervals for CoCo applied to different allocation sites.

CollectionSwitch

Figure 10a shows the 90% credible intervals for the difference in percentages, when CollectionSwitch is applied to different allocation sites of bloat. We see that CollectionSwitch has no positive effect, when applied on individual sites.

Figure 10b shows the 90% credible intervals for the difference in percentages, when CollectionSwitch is applied to different allocation sites of lusearch. It has a small positive effect for site 19, and a detrimental for site 1. For most of the sites, we observed no effect.



(a) For bloat, we could not find a positive effect from CollectionSwitch, on individual sites.

(b) For lusearch, CollectionSwitch applied to site 19 has a small positive effect, but it is probably compensated by the stronger negative effect on allocation site 1.

Figure 10: Mean difference and its 90% credible intervals for CollectionSwitch applied to different allocation sites.

7 Discussion

7.1 RQ1: What is the best possible improvement on DaCapo benchmarks?

The time spent manipulating collections varies significantly between benchmarks. avrora and fop use collections for less than 10% of their running time. lusearch and bloat spend at least 34% of the time manipulating collections. h2 spends 20% of the time manipulating collections. bloat, h2 and lusearch are therefore more likely to benefit from collection selection than avrora and fop.

7.2 RQ2: How much improvement can we obtain from a static replacement approach?

For bloat, greedy search found a major optimization, speeding up the benchmark by almost 50%. For other benchmarks, greedy search could find optimizations with a small positive effect.

For most benchmarks, the effect of optimization is below the bounds reported in Table 8. For bloat, we expected 34% of execution time spent in collections, but we could speed bloat up by 46 to 48%. That indicates that our approach to estimate time spent in collections under-estimates it.

We think the discrepancy could be due to the fact that our approach to estimating the time spent in collection does not trace iterators. The most effective optimization for bloat switches a HashMap to a LinkedHashMap. Iterating on a LinkedHashMap is faster than for a HashMap [6].

7.3 RQ3: How much improvement can we obtain from adaptive collections, such as CoCo and CollectionSwitch?

We could not reproduce results from CoCo or CollectionSwitch. CoCo reported speedups between 4% for bloat and 44% for lusearch. In our experiments, CoCo slows down bloat by 5.5 to 10.5%, and has no effect on lusearch. CollectionSwitch reported a speedup of 6% for h2 and 15% for lusearch. In our case, we observe that CollectionSwitch slows down lusearch by 2.0 to 5.0%, but optimizes bloat by 2.5 to 5.5%. It does not seem to have an effect on h2.

Because greedy search was more effective that both CoCo and CollectionSwitch, we conclude that we could not find evidence that adaptive collections are more effective than static approaches at reducing execution time.

[6] https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html

7.4 RQ4: What role do different program locations play in observed speedups?

Because we could not reproduce the methods CoCo and CollectionSwitch use to select allocation sites, we made a detailed study on the effect of these collections for individual allocation sites. We see that for some allocation sites, adaptive collections did have a noticeable positive effect (e.g. CoCo could speed up bloat by 10%). For lusearch, CoCo was not effective for any of the sites that we tried, but greedy search also failed to find major optimizations.

CollectionSwitch causes a 5% speedup on bloat, but we have been unable to find which allocation sites were responsible for these speedups. We ran the same experiment with the 50 most important allocation sites, but found no sites which could explain the 5% speedup.

One possible explanation for this behavior is a positive interaction between several changes. It is possible that each change has no positive effect, but that several changes combined have a positive effect. For example, in the case where collections store collections as elements.

Another possible explanation is a positive effect that only happens for some runs. The distribution of execution times for bloat is bi-modal. In some cases, bloat runs 30% faster than usual. We suspected that frequency scaling was the cause of this behavior, but turning it off for our experiments did not have any impact. Since we do not have an explanation for the modality of the distribution, it is possible that small changes change the probability of falling in the fast or slow mode.

8 Threats to Validity

8.1 Internal Validity

Steady-state Performance Setting a precise threshold to determine steady-state performance is tricky [Bar+17; GBE07; Tra+22]. We manually inspected the execution times of benchmarks and selected the lowest threshold for which we saw that the benchmarks reached a plateau. It is possible that our benchmarks did not reach steady-state performance.

Statistical Power We used 10 measurements, but did not observe significant effects for most of our benchmarks. Using more benchmark runs will reduce the variance of the posterior distributions and therefore improve the precision of the estimates.

Variance per Benchmark Our statistical analysis assumes that unobserved factors (“noise”) influences different benchmarks in the same way. There is therefore one single σ term in each of our statistical models. We tried running the

statistical analysis with one variance term per benchmark. We observed that the estimated effects were more precise, but saw no evidence of bias in the results: the mean of the posterior distributions were approximately the same.

8.2 External Validity

JDK Version We confirmed that the version of JDK could have a significant impact on execution time. However, we could not reproduce CoCo with JDK 6, because our bytecode manipulation tool required default methods from Java 8, which we used to

Selected Allocation Sites The CoCo study did not report the allocation sites that were selected for their experiments. We tried to reproduce the strategy used for CollectionSwitch, but differences in implementation prevented us from selecting the same sites. For a more nuanced comparison, we showed a detail of the effect of CoCo, for benchmarks were we observed a strong effect, to evaluated how the choice of allocation site resulted in speedups.

Selected Benchmarks We selected six benchmarks from DaCapo, focusing on those used in CoCo and CollectionSwitch. We have not tested with xalan or pmd.

9 Conclusion and Future Work

In this work, we investigated how much improvement we could obtain by manipulating collections in six DaCapo benchmarks. We showed that for avrora and fop, we were unlikely to obtain large speedups, because they do not spend much time manipulating collections.

We use a greedy approach to optimize benchmarks, and observed a major speedup in only one benchmark: the execution time of bloat was reduced by almost 50%. For the other benchmarks, however, we did not observe such strong effects.

We tried to reproduce previous studies which used adaptive collections. For CoCo, we did not observe any speedup. However, in the case of bloat, CoCo could speed up the benchmark by 10% when used on a single allocation site.

We observed a 5% improvement by CollectionSwitch on bloat. For the other benchmarks, we did not observe significant effects. We could not find a specific allocation site in bloat that was responsible for the 5% speedup.

9.1 Future Work

We could not reproduce results from the original works, but we used a different selection of allocation sites, and different JDK version. Future work could try

to test if CoCo is more effective with JDK version 6 and if using the exact same allocation sites as those in CollectionSwitch helps.

We could not explain which allocation site was responsible for the 5% speedup observed when using CollectionSwitch on bloat. We suspect it could be due to an interaction effect, in which individual changes do not have an effect, but their combination does.

In our statistical analysis, we noticed that the variance of measurements differs significantly between benchmarks. Implementing a model with benchmark-specific variance terms is possible, and could be extended to variance specific to the benchmark and machine.

References

[Bar+17] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. “Virtual machine warmup blows hot and cold”. en. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–27.

[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+06b] 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.

[FFT19] Carlo A. Furia, Robert Feldt, and Richard Torkar. “Bayesian Data Analysis in Empirical Software Engineering Research”. en. In: *IEEE Transactions on Software Engineering* (2019). arXiv:1811.05422 [cs, stat], pp. 1–1.

[GXG18] Hong Ge, Kai Xu, and Zoubin Ghahramani. “Turing: a language for flexible probabilistic inference”. In: *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*. 2018, pp. 1682–1690.

[GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. "Statistically Rigorous Java Performance Evaluation". In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 57–76.

[Jun+11a] 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.

[Len+17] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008". en. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. L'Aquila Italy: ACM, Apr. 2017, pp. 3–14.

[McE20] Richard McElreath. *Statistical rethinking: a Bayesian course with examples in R and Stan*. 2nd ed. CRC texts in statistical science. Taylor and Francis, CRC Press, 2020.

[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.

[PGJ16] Judea Pearl, Madelyn Glymour, and Nicholas P. Jewell. *Causal inference in statistics: a primer*. Chichester, West Sussex: Wiley, 2016.

[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.

[Tra+22] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. "Towards effective assessment of steady state performance in Java software: are we there yet?" en. In: *Empirical Software Engineering* 28.1 (Nov. 2022), p. 13.

[Wan+22] Chengpeng Wang, Peisen Yao, Wensheng Tang, Qingkai Shi, and Charles Zhang. "Complexity-guided container replacement synthesis". en. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA1 (Apr. 2022), pp. 1–31.

[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.