Empirical Study of Usage and Performance of Java Collections

Diego Costa
Institute of Computer Science
Heidelberg University,
Germany
diego.costa@informatik.uni-heidelberg.de

Artur Andrzejak
Institute of Computer Science
Heidelberg University,
Germany
artur@uni-hd.de

Janos Seboek
Institute of Computer Science
Heidelberg University,
Germany
seboek@cl.uni-heidelberg.de

David Lo
School of Information Systems
Singapore Management University, Singapore
davidlo@smu.edu.sg

ABSTRACT

Collection data structures have a major impact on the performance of applications, especially in languages such as Java, C#, or C++. This requires a developer to select an appropriate collection from a large set of possibilities, including different abstractions (e.g. list, map, set, queue), and multiple implementations. In Java, the default implementation of collections is provided by the standard Java Collection Framework (JCF). However, there exist a large variety of less known third-party collection libraries which can provide substantial performance benefits with minimal code changes.

In this paper, we first study the popularity and usage patterns of collection implementations by mining a code corpus comprised of 10,986 Java projects. We use the results to evaluate and compare the performance of the six most popular alternative collection libraries in a large variety of scenarios. We found that for almost every scenario and JCF collection type there is an alternative implementation that greatly decreases memory consumption while offering comparable or even better execution time. Memory savings range from 60% to 88% thanks to reduced overhead and some operations execute 1.5x to 50x faster.

We present our results as a comprehensive guideline to help developers in identifying the scenarios in which an alternative implementation can provide a substantial performance improvement. Finally, we discuss how some coding patterns result in substantial performance differences of collections.

ICPE'17, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4404-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3030207.3030221

Keywords

empirical study, collections, performance, memory, execution time, java

1. INTRODUCTION

Collection libraries and frameworks offer programmers data structures for handling groups of objects in an abstract and potentially efficient way. The popularity and importance of such libraries and frameworks is indicated by the fact that programming languages such as Java, C#, Python, Ruby, or C++ include them as a part of the core language environment (STL is a de facto standard in case of C++.)

Typically in such libraries each of the major abstract collection types (e.g., lists, sets and maps) has multiple implementations adhering to the same API. Each implementation attempts to fulfill the requirements of different execution scenarios and thus provides different trade-offs. For example, Java's ArrayList has a smaller memory footprint than a Java LinkedList, yet it has a higher asymptotic complexity when it comes to inserting or deleting elements at random positions.

Given that modern programs use collections in thousands of program locations [23], selecting appropriate type and implementation is a crucial aspect of developing efficient applications. Choosing a wrong collection may result in performance bloat - the excessive use of memory and computational time for accomplishing simple tasks. Numerous studies have identified the inappropriate use of collections as the main cause of runtime bloat [5, 16, 27, 28]. Even in production systems, the memory overhead of individual collections can be as high as 90% [16].

The problem of selecting an appropriate collection becomes more complex when accounting for third party libraries. These allow for many more choices compared to the standard Java Collection Framework (JCF), from simple alternatives to existing JCF implementations to collections with extra features such as immutability and primitive-type support.

Despite its importance, we found a gap in experimental studies comparing execution and memory performance of non-JCF collections. Partial benchmarks, especially on the websites of the libraries, are common enough; they, however, do not give a performance comparison for different libraries, nor do they provide an evaluation under a large set of scenarios. This paper attempts to fill this gap in experimental studies and derive a set of guidelines for developers on how can they improve performance with little code refactoring.

To this aim (i) we study the usage of collections in real Java code via repository mining and (ii) evaluate the memory consumption and execution performance of collection classes offered by six most popular collection libraries. The key question to be answered by our study is: *"Can we improve performance of applications in typical scenarios by simply replacing collection implementations, and if yes, to which degree?"*. In particular, we explore alternatives to JCF implementations under the same collection abstraction.

Our results show a considerable variance in performance in different implementations of the same data structure. Moreover, the best alternative implementations outperform the collections provided by the standard JCF library across many workload profiles, making them a better choice in terms of memory consumption and runtime for most applications in many circumstances. Improvements range from 20% to 50 times in execution time, and saving up to 88% in memory.

The contributions of this paper are as follows:

- We analyze the popularity and usage patterns of collection libraries based on mining a dataset with 10,986 Java projects (Section 3).
- We propose a framework for systematic evaluation of collection performance. This framework can be further extended to cover new libraries, scenarios and implementations (Section 4).
- We evaluate the performance of JCF and six major alternative libraries in terms of execution and memory under a variety of scenarios. As a part of the study, we investigate the performance of primitive collections (Section 5).
- We provide a guideline for developers on replacing standard JCF collections by alternative implementations based on performance characteristics (Section 6).
- We explain the causes of performance differences of collection implementations by analyzing the source-code and benchmark indicators for some relevant scenarios (Section 6).

2. BACKGROUND

Collections are data structures that group multiple elements into a single unit. A collection uses metadata to track, access and manipulate its elements via specified APIs. Naturally, the metadata incurs extra memory consumption (*collection overhead*) in addition to the memory used by its elements (*element footprint*). Collections come in several *abstraction types*, primarily list, map, set, and queue.

The standard implementation of the collections in Java is the *Java Collections Framework*, or *JCF*. It provides implementations of all major abstraction types in several variants, e.g. ArrayList, HashMap, HashSet and others. JCF is implemented purely as *object collection*, i.e. the metadata contains only references to Java objects, but no data itself (Figure 1).

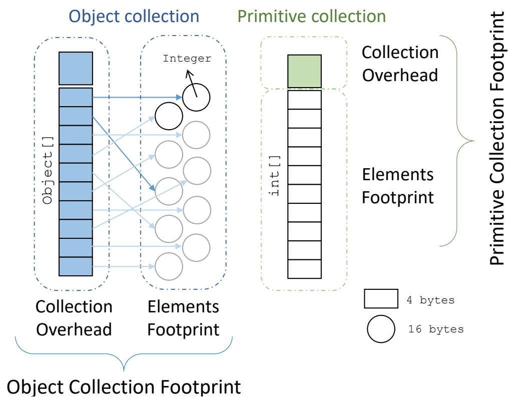


Figure 1: Conceptual view of object collections vs. primitive collections with the example of ArrayList, and terms related to memory usage.

The JCF offers programmers a stable and reliable collections framework. However, there exist many alternative collection libraries that can outperform standard JCF abstraction types, and additionally target features not supported by the JCF, such as immutable collections, multisets and multimaps.

If a collection contains simple objects like Integer, Long, Double, Character, etc., *primitive collections* can be used to reduce the memory footprint. The key difference is that a primitive collection stores data directly in an array of primitives (int, long, double, char), instead of using an array of references to data objects. Figure 1 illustrates an ArrayList of integers as an object collection (left) and a primitive collection (right). The primitive variant reduces the collection footprint in two ways. First, the primitive collection needs only a single reference to an array of primitives instead of an array of references. Second, each primitive data element (type int) requires only 4 bytes instead of 16 bytes of an object Integer. This example indicates that alternative collections can slash memory footprint of collections by a large factor.

3. ANALYSIS OF COLLECTION USAGE

In this section we describe the usage patterns of collections found in a large code corpus. Understanding such patterns helps us to identify the most popular collection types and implementations. This study will help us devise a benchmark presented later in the paper.

3.1 Data and Static Analysis

For our analysis we use the GitHub Java Corpus [1], a dataset from GitHub containing 10,986 Java projects, in total more than 268 millions lines of code. This corpus consists only of projects with more than one fork to eliminate small or inactive projects, which prevail at GitHub [12]. In addition, the creators of GitHub Java Corpus analyzed and pruned this dataset manually in order to eliminate project clones.

We categorize the whole data into two sets of repositories, namely all projects provided by Java GitHub Corpus (*FullSet*) and a subset of the 50 most popular projects (*Top50*). The popularity of the latter set is based on ranking of projects provided by GitHub corpus [1], and it is illustrated by the fact that *Top50* contains projects like Eclipse IDE[1], Clojure[2], and Elastic Search[3]. By analyzing both sets we can investigate whether there is an impact of project maturity (approximated by popularity) on the patterns of collection usage.

We need to extract from the code every declaration of a collection and every site of a collection instantiation. This is needed to count the usages of collection classes, record the types of held elements, and count how often the initial capacity is specified. We use JavaParser[4] to extract each variable declaration and instantiation site in the code. We then filter the collection instances by applying a heuristic as defined by the following regular expression:

$$.*List|.*Map|.*Set|.*Queue|.*Vector| .$$

This pattern finds all collection implementations of interest but also retrieves some false positives, e.g., `java.util.BitSet` is not a general purpose collection but is retrieved by our heuristic regardless. We rank the retrieved types and manually inspect and filter out false positive for the top 99% of the retrieved data, ranked by occurrence.

3.2 Collections Usage in Real Code

Collection Types. We analyze the instantiations sites and extract the most used collections in the code (see Figure 2). Unsurprisingly, list is the most commonly used collection abstraction, followed by map and finally by set. Our ranking is dominated by JCF as developers only rarely opted for others, not nearly often enough to make it into our charts.

Among the abstraction types, lists are the most common ($\approx$ 56%), followed by maps ($\approx$ 28%) and sets ($\approx$ 15%). Contrary to this, queues are rather rare (1.23% of usages). Among lists, ArrayList makes up the bulk of all collections usages, namely 47%. The LinkedList is surprisingly common, with $\approx$ 5% and is ranked as the fourth most common implementation. In maps, HashMap is most commonly used, representing $\approx$ 23% of all collections instantiation, followed by LinkedHashMap ($\approx$ 2%) and the TreeMap ($\approx$ 1%). Set instances follow a similar pattern to maps, HashSet is most used ($\approx$ 10%), followed by TreeSet and LinkedHashSet with both $\approx$ 1% of occurrences.

In summary, the top four most frequently used collections are JCF implementations: ArrayList, HashMap, HashSet and LinkedList. Together they account for approximately 83% of all collections declared in the repository. We found a similar distribution in *Top50*, which have a slightly higher usage of concurrent collections and a lower variety of types.

Element Types. By *element type* we mean the object types held by collections. Understanding their distribution helps us to cover in our benchmark the most frequent real scenarios. To simplify, we group the element types into four categories:

- Strings: String, String[], and String[][].
- Primitives: The primitive-wrappers such as Double, Float, Long, Short, Integer, as well as Boolean, Character, and their respective arrays.
- Collections: Collection types that can be identified by our heuristic from Section 3.1.
- Other: All the classes and data types not fitting any of the mentioned categories.

The result of our analysis, presented in Figure 3, shows a clear similarity between the results in *FullSet* and *Top50.* A pattern that emerges is the similar usage of primitive wrappers throughout the categories, always ranging from $\approx$ 5% to $\approx$ 8% of the declarations. This category is particularly interesting because it contains the collections that can be replaced by primitive collections with simple code refactoring.

The remaining categories show a distinct pattern for each of lists, maps and sets. Lists have a higher variability of element types, but hold strings $\approx$ 20% of the time. Sets hold strings more often than lists, $\approx$ 31% of the time. Maps also hold strings very often: $\approx$ 70% of all declared maps use the String as a key and $\approx$ 28% as a value. Map values are often a collection ($\approx$ 15%) pointing to a common usage of maps as a collection holder.

Initial Capacity Specification. Defining an appropriate initial capacity of a collection is a simple but effective method for optimizing runtime and memory. We sample 400 of the instantiation sites of ArrayList, HashSet and HashMap, which give us 5% confidence interval. Then we manually categorize whether a developer specifies an initial capacity, copies this instance from another collection, or uses default constructor values.

Figure 4 shows that programmers specify initial capacities for ArrayList only in $\approx$ 19% of the declarations. HashMap and HashSet have their capacity specified in $\approx$ 7% and $\approx$ 8% of collection declarations, respectively. Our analysis with the *Top50* provides very similar results, with a slightly higher percentage of initial capacity in ArrayList declarations. Interestingly, HashSet is the category more commonly created through copy instruction (in $\approx$ 11% of cases in *FullSet* and $\approx$ 14% in *Top50*).

Project Maturity and Collections Usage. We found similar results in our analysis of *FullSet* and *Top50.* This indicates that, regardless of the project maturity, programmers instantiate collections in a similar fashion. They mostly select standard collections types, and rarely specify the initial capacity in a collection instantiation.

4. EXPERIMENTAL DESIGN

In this section we describe the design of experiments for evaluating performance of collection implementations in terms of execution time and memory usage. In particular:

1. We identify a suitable set of collection libraries for this evaluation through ranking of libraries in terms of their popularity.
2. We describe a benchmark framework capable of measuring the steady-state performance of collections with high precision.
3. We design an experimental plan which covers a large set of usage scenarios based on the findings of usage patterns reported in Section 3.

[1] https://github.com/eclipse/eclipse.jdt/
[2] https://github.com/clojure/clojure
[3] https://github.com/elastic/elasticsearch
[4] https://github.com/javaparser/javaparser

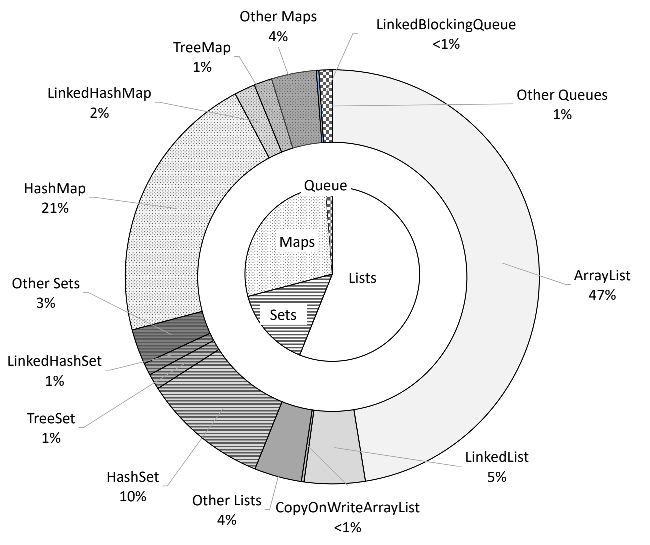


Figure 2: Distribution of instantiated collection types in *FullSet*. Every specific collection in this chart is from JCF, as standard implementations are the most prominent ones.

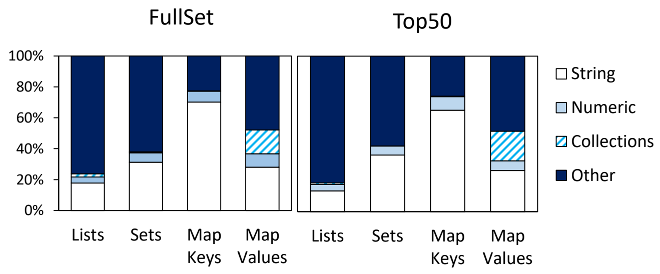


Figure 3: Distribution of element types by collection for both the *FullSet* and the *Top50* set.

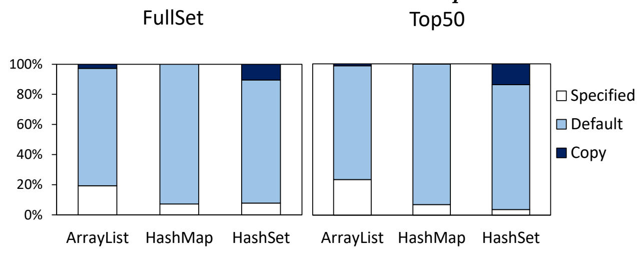


Figure 4: Statistics on the style of specifying initial capacity of collections for *FullSet* and *Top50*. "Specified" denotes that the developer explicitly set the initial capacity, and "Copy" that a copy constructor was used.

The results of rigorous statistical analysis of each evaluated scenario are presented then in Section 5.

Note that we compare the performance within the same abstraction type (i.e. list, map, set). Our experimental parameters include the number of elements held in a collection, and a particular usage scenario (Section 4.2).

4.1 Selection of Collection Libraries

We search the Web for alternative collection libraries implemented in Java and find a total of 14 libraries. From these, we use two different approaches to extract a suitable set for our experimental evaluation. First, we rank them by popularity of their project in GitHub, see Table 1. Second, we analyze how many times they have been included in existing benchmarks (Table 2).

The GitHub metrics such as number of stars and number of watches provide a simple but effective method for ranking software projects. Such metrics have also been used as a criterion in other studies [19, 22]. To account for libraries that are not on GitHub, we retrieve a set of collection benchmarksfrom the web, and count for each collection library in how many benchmarks it was evaluated.

Furthermore, we filter libraries that do not provide implementations that can serve as a replacement to JCF collections. This excludes Javaslang as it provides only immutable collections for lambda function usage. In the final step we select the top five libraries from each ranking and merge them into a single list. This yields seven unique libraries (three of them occur in both rankings) to be included in our experimental evaluation. For this study we use the JCF of Java 8 (jdk1.8.0_65.)

4.2 Benchmark Design

A managed runtime environment such as the Java Virtual Machine poses a challenge for performance measurement. Numerous factors can affect performance: the effects of the garbage collector, the Just in Time compilation (JIT), the heap size, the method sampling optimization, and many other unpredictable system effects [3, 4, 18]. To reduce the impact of these factors we follow the suggestions by Georges et al. [4] and implement a four-step methodology for evaluating a steady state performance of collection implementations:

Table 1: Ranking of collection libraries by GitHub popularity. We select the top 5 collections from this ranking. Data was obtained on 28 September 2015.

Rank	Library	# Stars	# Watches
1	Guava	5067	641
2	GS-Collection	1293	196
3	Koloboke	369	69
*	Javaslang	309	31
4	HPPC	189	31
5	Fastutil	69	6
6	HPPC-RT	8	3

S1 For each scenario we execute ten warm-up iterations to achieve a steady performance [5].

S2 We execute 30 iterations while measuring our response variables. Each iteration executes the same operation (sample) in an uninterrupted fashion for five seconds. After reaching the timeout, we calculate the average of all samples as a result. Each result also contains

[5]Our preliminary analysis showed that the execution time of the experiment converges after seven warm-up iterations

Table 2: Ranking of collection libraries by number of occurrences in benchmarks. We select the top 5 collections from this ranking.

Rank	Library	# Occurrences
1	JCF	13
2	Trove	10
3	HPPC	6
4	Koloboke	5
5	Fastutil	4
6	GS-Collection	4
7	Javolution	4
8	Guava	3
9	Mahout	3
10	Commons	2
11	Brownies	1
12	Colt	1
13	Javaslang	0
13	HPPC-RT	0

the experimental error of the iterations at a 99% confidence interval.

S3 We execute steps S1 and S2 twice to avoid circumstantial external influence.

S4 We analyze the results of 2 x 30 iterations using rigorous statistical methods. In detail, we analyze the variance with ANOVA [17] and compare multiple means accounting for their experimental errors.

We perform this experimental methodology on our benchmark suite called *CollectionsBench*. CollectionsBench is built upon JMH[6], a Java harness framework for building, running and pre-analyzing benchmarks. JMH provides support for benchmark forks, warm-up iterations, and can give the benchmark results with nanosecond precision.

To reliably measure performance using CollectionsBench, we perform the following steps: First, we guarantee a homogeneous benchmark behavior throughout different collection implementations. The Template design pattern helps us reuse the same test code in all JCF compliant libraries (see Table 3) and to delegate the collection creation to the library specific code. Second, we consume every non-void return to avoid undesired dead-code optimization. All these steps help to minimize sources of non-determinism in the JVM.

Finally, to only measure time and memory on the collections operations, we divide our experiment into two distinct *setup* and *benchmark* phases: In the *setup* phase we allocate all the elements and insert them into an instance of the evaluated collection. The values of each element are generated randomly through a uniform distribution, and we then reuse the random seed to ensure the same conditions in all tests. In the *benchmark* phase we execute the collection operations and measure the following response variables:

- *Execution time*: Time spent executing the benchmark in nanoseconds, using JMH native support.

- *Memory allocation*: The sum of memory requested during the benchmark execution. Since we allocate the elements in the *setup* phase, this variable shows only the memory requested for the collection overhead (including the collection object header and eventual memory padding), but does not consider the element footprint. We extract this information through the JMH GC profiler.

[6]http://openjdk.java.net/projects/code-tools/jmh/

We also collect some performance indicators through the Perf[7] profiler, such as the number of instructions executed, cache miss rate and branch misprediction rate. In Section 6 we analyze those indicators along with the source-code to understand the reasons for performance differences of implementations.

The memory allocated during the benchmark is sensitive to buffers and temporarily allocated objects. For instance, when expanding the ArrayList allocates a new and larger buffer to accommodate more elements. The previously allocated buffer and the new one will be measured by the *memory allocation* variable. Therefore, the memory allocation alone cannot provide an accurate view of the memory usage of a collection.

To account for this problem, we evaluate the memory usage of a collection by computing their collection overhead. We use the tool Java Object Layout (JOL)[8] to retrieve the collection overhead of each implementation. We then analyze both memory allocation and overhead together, to give a detailed perspective of a collection's memory consumption.CollectionsBench is open-source and is fully available online[9].

4.3 Experimental Planning

Our experimental plan takes into consideration the findings on collections usage presented in Section 3. First, we focus on the most frequently used collection types while investigating possible alternatives (from third-party libraries) to the JCF collections. Therefore we select the four most used collection types, ArrayList (AL), LinkedList (LL), HashMap (HM), HashSet (HS), which accounts for more than 84% of all declared collections.

Most of the libraries do not implement an alternative to LinkedList, so we opted to compare JCF LinkedList implementations against ArrayList alternatives, as they are interchangeable. We also profile primitive alternatives to the top three collection types, as they provide a small-footprint option to collections that hold primitive wrappers (see Table 3).

Second, we pay a special attention to collections holding the element type String. Strings are particularly interesting in Java due to their extensive use, and have been studied by various authors [13]. To expand on the results, we evaluate the collection implementations for Integer and Long objects as well. We opt for these two numeric objects as they are representative: primitive-wrappers are the second most common category identified. Moreover, they mainly represent objects with 32 bytes and 64 bytes respectively, a common object size.

Third, we do not specify the initial capacity in our benchmark because from our static analysis results, specifying the initial capacity is done rarely in real code. Since we aim to provide results that are useful to the widest range of programmers, we evaluate collections with their default initial capacity.

Fourth, given C as a collection populated with N elements

[7]https://perf.wiki.kernel.org/index.php/Main_Page
[8]http://openjdk.java.net/projects/code-tools/jol/
[9]https://gitlab.com/DiegoCosta/collections-bench

Table 3: The evaluated collection implementations. The primitive implementations are marked with (p). The column JCF indicates whether the libraries provide implementations compatible with the Java Collection Framework (only for object collections). In total we evaluate 33 implementations.

Library	Version	JCF	List(AL/LL)	HashMap (HM)	HashSet (HS)
JCF [20]	8.0_65	yes	ArrayList LinkedList	HashMap	HashSet
Guava (Gu) [8]	18.0	no		HashMultimap	HashMultiset
Fastutil (Fu) [24]	7.0.10	yes –	ObjectArrayList IntArrayList(p)	Object2ObjectOpenHashMap Int2IntOpenHashMap(p)	ObjectOpenHashSet IntOpenHashSet(p)
Koloboke (Ko) [2]	0.6.8	yes –		HashObjObjMap HashIntIntMap(p)	HashObjSet HashIntSet(p)
HPPC (HP) [21]	0.7.1	no –	ObjectArrayList IntArrayList(p)	ObjectObjectHashMap IntIntHashMap(p)	ObjectHashSet IntHashSet(p)
GSCollections (GS) [7]	6.2.0	yes –	FastList IntArrayList(p)	UnifiedMap IntIntHashMap(p)	UnifiedSet IntHashSet(p)
Trove (Tr) [6]	3.0.3	yes –	 TIntArrayList(p)	THashMap TIntIntHashMap(p)	THashSet TIntHashSet(p)

in the *setup* phase, we evaluate the collections under the following scenarios:

- *populate*: add N random elements into an empty collection.
- *iterate*: iterate through all elements of C.
- *contains*: check whether C contains an existing random element.
- *add*: add a random element to C. To keep C with the same size throughout the experiment, we remove the added element at the end of the scenario.
- *get*: get a random existing element from C.
- *remove:* find and remove a random element from C. Analogously to *add*, we add the removed element in C at the end of the scenario.
- *copy*: copy C into an empty collection using the copy constructor.

We could not obtain the size of collections through static code analysis. To compensate for this fact and to account for various scenarios, we run our benchmarks with multiple values for N ranging from 100 to 1M, with the interval set as a power of ten (five categories of size).

In summary, we perform a *factorial experiment* [17] with three element types and five collection sizes, in a total of 15 different configurations. This 15-configuration experiment is run through the seven scenarios for each of the 33 collection types (see Table 3). We execute a total of 3,465 experiments, where each experiment lasts five minutes including replications and forks. Running the full benchmark takes 12 days to finish.

We conduct our experiments on a machine with a E5-1660 3.3GHz CPU, 64GB RAM using Linux 3.16.0-53. We use 64-bits JVM HotSpot, and the jdk1.8.0_65 as our Java version. All tests were executed in a single-threaded environment, as our target collections were built for such settings.

5. RESULTS

Our goal is to find a *superior alternative* that can outperform JCF in terms of execution time and/or memory consumption in several scenarios, while introducing no or minimal penalties in others. To achieve this, we ask the following research questions:

RQ1. Does the type of elements stored in collections influence execution time?

RQ2. Are there superior alternatives to the most used JCF collections with regard to execution time?

RQ3. Do primitive collections perform better than JCF collections with regard to execution time?

RQ4. Are there superior alternatives to the most used JCF collections with regard to memory consumption?

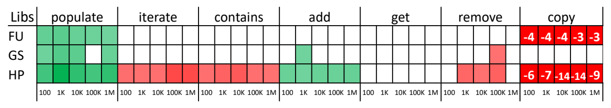
RQ1 addresses the impact of element type in the execution time of collections; RQ2 and RQ3 show the analysis of alternative object and primitive-collections on execution time; RQ4 reports superior object-collection alternatives on memory consumption. We omit the analysis of primitive-collections on memory consumption, as this question is rather well-explored and exemplified in Section 2.

RQ1. Does the type of elements stored in collections influence execution time?

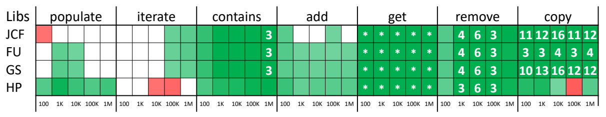
Key result: The element type has a significant influence in the execution time in all major collection abstractions. For lists, however, this impact is homogeneous throughout all implementations.

We analyze the impact of the element type in the object-collections execution time to verify whether our results can be generalized beyond the types evaluated. We use the ANOVA method [17], with a 95% confidence interval and analyze the influence of element type on lists, sets and maps.

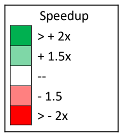
The element type influences execution time significantly in our three main categories ($p < 0.05$). For lists, however, the element type factor does not interact with the list implementation ($p = 0.76$), indicating that it influences the performance regardless of the list type. Thus, we expect that our results for lists can be generalized beyond the type used. We report the results for sets and maps only for the String element type as it represents the largest amount of collection instantiations.

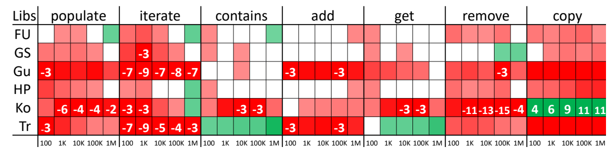


(a) ArrayList alternatives

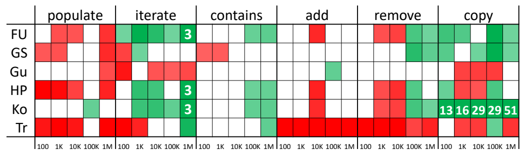


(c) ArrayList alternatives to LinkedList



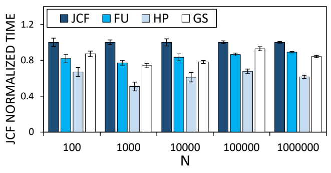


(b) HashMap alternatives (element type = String)

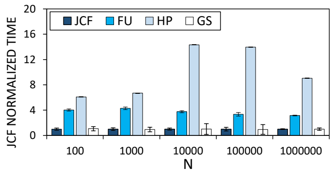


(d) HashSet alternatives (element type = String)

Figure 5: Heatmap showing speedup/slowdown (as defined in Equation 1) of alternative collections compared to JCF, per scenario and size. Green cells indicate a speedup, red cells represent a slowdown. Performance ratios larger than factor 2x are rounded to the nearest integer and printed inside the cell. LinkedList *get* ratios are substantially higher and not shown.

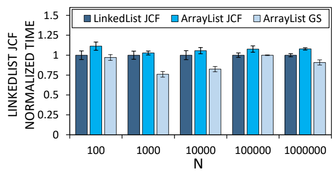


(a) populate

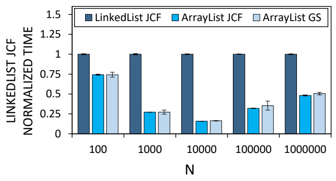


(b) copy

Figure 6: Execution time profiles of Array Lists for the *populate* and *copy* scenarios.



(a) populate



(b) remove

Figure 7: Execution time profiles of LinkedList for *populate* and *remove* scenarios.

RQ2. Are there superior alternatives to the most used JCF collections with regard to execution time?

Key result: GSCollections provides a superior alternative to JCF ArrayList. LinkedList is outperformed by any other ArrayList implementation, and both the HashSet of Koloboke and Fastutil are a solid improvement over the standard one. We found no superior alternative to the JCF HashMap, which provides a stable and fast implementation throughout all the scenarios.

We report our results as a comparison to the JCF implementation instead of the absolute performance. First, we extract only comparisons where the errors (99% confidence interval) do not overlap. Then we calculate the impact of an alternative collection over JCF using the following speedup/slowdown S definitions:

$$S = \begin{cases} \dfrac{T_{jcf}}{T_{alt}}, & \text{if } T_{jcf} > T_{alt} \\ -\dfrac{T_{alt}}{T_{jcf}}, & \text{otherwise} \end{cases} \quad (1)$$

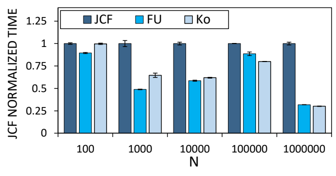
where T_{jcf} and T_{alt} are the time obtained with using the JCF implementation and the time using an alternative implementation respectively.

We present our results in a broad heatmap analysis in Figure 5, labeled by color. We make an in-depth analysis for each category in the remainder of the section.

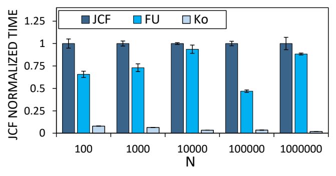
ArrayList. In the *populate* scenario occurrences of JCF ArrayList can be replaced by the implementation from GSCollections to achieve faster population without compromising the speed of any other operation. In fact, all evaluated alternatives are faster than JCF when populating the list (see Figure 6.) The HPPC implementation is, however, slower when iterating the elements in both the *iterate* and the *contains* scenario. In addition, both Fastutil and HPPC copy their lists from 3 to 14 times more slowly.

LinkedList. JCF LinkedList is outperformed by all ArrayList implementations by a large margin, even in scenarios where LinkedList has a theoretical advantage (Figure 7). This is the case for the *remove* scenario, where we search and remove the element through the `remove(Object)` method. Despite the asymptotic advantage, LinkedList was outperformed by a large margin for $N \geq 1k$. Furthermore, LinkedList has a comparable performance in the *populate* scenario, even without having to reallocate its buffer (as ArrayList does). The LinkedList is up to three times slower when searching for a random element ($N = 1M$), and it was also outperformed in the *iteration* scenario by some ArrayList variants, but only for $N > 10k$. Note that in our benchmark design the LinkedList is unfragmented as the elements are inserted without any removal. Hence, we consider these results a best-case scenario for LinkedList.

HashMap. JCF HashMap provides solid performance and cannot be easily replaced by any alternative in terms of ex-



(a) iterate



(b) copy

Figure 8: Execution time profiles of HashSets holding String elements for the *iterate* and *copy* scenarios.

ecution time improvement. As shown in Figure 5, some implementations have a comparable performance, like Fastutil, GSCollections and HPPC. Standard HashMap is outperformed only in the *copy* scenario, where Koloboke is able to copy up to 11x faster. However, Koloboke is substantially slower in most of the other remaining scenarios.

HashSet. Standard HashSet is outperformed by many alternatives, mostly in the *iterate* and *copy* scenarios (Figure 8). Koloboke is a superior alternative to JCF, outperforming JCF HashSet for large set iteration, and copying at least 10x faster. Fastutil and GSCollections can also be selected as good alternatives: both are slower when populating but faster in the *copy* and and *iterate* scenarios. The JCF implementation is faster than the majority of alternatives when adding elements to the set, but is often outperformed in the remaining scenarios for large workloads ($N > 100k$).

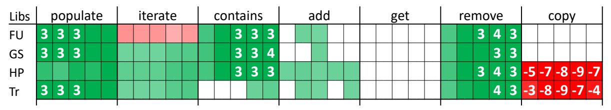
RQ3. Do primitive collections perform better than JCF collections with regard to execution time?

Key Result: For all three abstraction types (list, map, set) we found a set of primitive implementations that are superior to JCF implementations. GSCollections and Koloboke provide the fastest primitive-based alternatives, followed by Fastutil and HPPC. Trove is often outperformed by JCF implementations in this context.

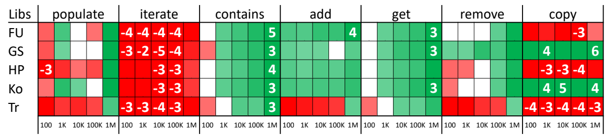
Our results for analyzing the experiments are summarized in a colored heatmap in Figure 9.

Primitive ArrayList. Primitive lists provide a superior alternative to JCF ArrayList (Figure 9a). The speedup in some cases reaches four times better than the baseline when checking or removing an element from the list. GSCollections consistently outperforms the JCF, followed by Fastutil, which is slightly slower when iterating through the list. HPPC and Trove are much slower when copying their lists, and should be avoided if the workload demands this operation often.

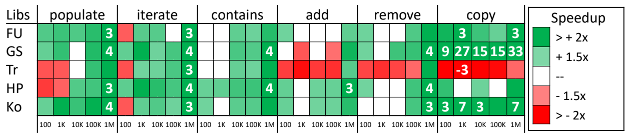
Primitive HashMap. As a consequence of the standard JCF HashMap's good performance, in some scenarios there is no underlying benefit gained from changing to a primitive-type map implementation (Figure 9b). Iterations are especially good with the standard HashMap, up to five times faster than the alternatives. For the *contains*, *add*, *get* and *remove* scenarios, most primitive implementations can provide an improvement. When also considering the *copy* scenario, GSCollections and Koloboke provide the highest performance gain among the alternatives.



(a) ArrayList primitive alternatives



(b) HashMap primitive alternatives



(c) HashSet primitive alternatives

Figure 9: Heatmap showing the speedup/slowdown (Equation 1) of primitive collections (int) compared to the JCF collections holding Integers, per scenario and workload.

Primitive HashSet. Primitive-based sets can be beneficial for the application's performance when it comes to a large number of elements (Figure 9c). In fact, the speedup commonly reaches four times as fast across many scenarios with $N = 1M$. GSCollections, Fastutil and Koloboke provide superior alternatives and can copy their instances up to 27 times faster. HPPC is also a good alternative but it does not provide the same copy operation speed gain as the other mentioned libraries. Trove's implementation was the only primitive-based HashSet outperformed by the standard HashSet in the *add*, *remove* and *copy* scenarios.

RQ4. Are there superior alternatives to the most used JCF collections with regard to memory consumption?

Key result: We found no superior alternative to the JCF ArrayList in terms of memory overhead, but Fastutil allocates less memory when populating its own list implementation. Numerous alternatives offer a superior alternative to the JCF HashMap and HashSet. LinkedList has a higher overhead than any considered ArrayList alternative.

In this section, we analyze the difference in the overhead of collections as the main goal for a superior collection replacement. As a supplementary analysis, we also looked at an aspect often neglected by micro-benchmarks: memory allocation. A collection that allocates more memory than indicated by its overhead requests memory for two reasons: (i) allocation as a buffer for future operations, (ii) allocation for temporary objects. The latter reason has a negative impact on performance and can be used as an indicator of how often a collection implementation can trigger the action

Table 4: Comparison of collection overhead and memory allocation of various implementations. The overhead is given in bytes in the form $\alpha \times N + \beta$, where N is the number of elements in the collection, and α and β are implementation-specific factors. The average allocated is given in bytes per element.

Category	Libs	Collection Overhead	Avg Allocated	
			populate	copy
Array Lists	JCF	$4 \times N + 24$	14.83	4.01
	Fu	$4 \times N + 24$	10.07	4.02
	GS	$4 \times N + 24$	13.56	4.01
	HP	$4 \times N + 48$	15.10	4.01
Linked List	JCF	$24 \times N + 32$	24.07	28.11
Hash Maps	JCF	$36 \times N + 48$	49.93	41.05
	Fu	$8 \times N + 64$	35.52	42.18
	GS	$8 \times N + 64$	59.87	24.62
	Gu	$96 \times N + 64$	96.90	132.36
	HP	$8 \times N + 96$	36.41	18.16
	Ko	$8 \times N + 232$	35.47	17.96
	Tr	$8 \times N + 72$	47.23	18.37
Hash Sets	JCF	$36 \times N + 80$	48.85	40.43
	Fu	$4 \times N + 40$	24.43	8.43
	GS	$4 \times N + 32$	34.44	14.21
	Gu	$52 \times N + 112$	64.77	58.25
	HP	$4 \times N + 88$	16.88	8.43
	Ko	$4 \times N + 208$	16.84	8.43
	Tr	$4 \times N + 64$	22.25	8.35

of the Garbage Collector (GC). Collections that require frequent intervention of the GC can reduce the performance of an application in a long run, a behavior difficult to observe with micro-benchmarks.

Since the elements are pre-allocated in the setup phase, memory allocation only comes into play when elements are added to the collection, namely by *populate* and *copy* operations.

ArrayList. All implementations have a similar overhead (see Table 4). Due to its buffer reallocation, each implementation allocates on average three times its own overhead in the *populate* scenario. Regarding only memory allocation, Fastutil can be a good alternative, saving 30% of allocations in the *populate* scenario.

LinkedList. We show in Table 4 that the standard LinkedList implementation has an overhead of 24 bytes per element, as opposed to the 4 bytes required in each ArrayList implementation. This is a consequence of the pointers to neighbor elements for each LinkedList entry. Essentially, an ArrayList saves 83% of the memory overhead. Despite the buffer expansion of ArrayList, LinkedList still allocates twice as much memory in the *populate* scenario, and it allocates five times more memory when copying from a previous instance.

HashMap. We can observe in Table 4 that the JCF implementation has a considerably higher overhead. Standard HashMap has an overhead of 36 bytes per each entry in the map while almost every other alternative consumes only 8 bytes. This difference occurs because JCF uses a Node object for each entry, a structure that contains three references (12 bytes) and a primitive (4 bytes), but, being an object, also has an overhead of 12 bytes of header and 4 bytes lost due to alignment. The alternatives do not define each entry as an object and instead use two arrays where each key and value pair are stored. As a consequence, they save 77% of memory overhead. Regarding allocations however, the difference of JCF and the alternatives are not quite as drastic as in the collection overhead. This means that even though the alternatives have five times less overhead, the memory involved in the map operations is of a similar size.

HashSet. Similarly to Maps, JCF HashSet is implemented with a larger overhead than almost any of the potential alternatives. In fact, it uses a HashMap internally to store the elements, causing the same additional overhead of 36 bytes per element, due to the HashMap's Node object. The alternatives, on the other hand, implement the HashSet as a simple single array. This allows them to save 88% of memory overhead compared to the JCF HashSet. Unlike HashMap, this difference holds in the memory allocation as well, where JCF allocates twice as much memory in the *populate* and five times as much memory in the *copy* scenario.

6. DISCUSSION

6.1 Guideline for Collection Replacement

Each additional alternative collection implementation used in a software project increases its complexity and maintenance effort. Consequently, developers should consider replacing collection implementations only if such a change will result in a significant benefit. In order to support programmers in such decisions, we present a guideline (Table 5) showing the potential benefits and drawbacks of using alternative implementations for several relevant scenarios and/or optimization objectives.

We illustrate the effect of JCF collection replacement on execution time for collections with one thousand (small) and one million elements (large). Note that in case of object collections the memory savings come solely from a reduced overhead as the elements themselves are not modified. In case of the primitive collections the programmer must replace the object element type by its respective primitive. However, the impact on memory savings is higher as the memory footprint of elements is reduced as well.

Table 5 presents seven recommendations for replacements leading to a substantial improvements. Recommendation R1 describes the replacement of a LinkedList by an ArrayList from one of three libraries, each offering a consistent improvement in time and memory. In R2 we recommend Koloboke's alternative to JCF HashSet, as it provides a substantial improvement on execution time and memory overhead. For HashMaps it is possible to aim for a memory overhead reduction with a small execution time penalty (R3), or to reduce memory and improve the execution time of copy operations (R4).

An important result is the universal superiority of primitive collections. It is possible to considerably improve both memory footprint (overhead + elements) and execution time for all three major collections: ArrayList (R5), HashSet (R6) and HashMap (R7). Note that in all cases the replacement is particularly beneficial for large collections as the savings of memory and execution time are, in general, bigger.

6.2 Reasons for Performance Differences

Our investigation of the source code of collections revealed some implementation patterns responsible for performance

Table 5: A guideline showing the impact of replacing JCF collections with alternative collection implementations for some relevant scenarios/optimization objectives. For primitive collections, the memory savings include overhead reduction as well as smaller element footprint (results assume replacing Integer objects by the primitive int).

	Overhead Savings	Speedup/Slowdown 1k elements	Speedup/Slowdown 1M elements
To reduce JCF LinkedList overhead and improve time performance			
R1) Replace it by JCF/Fasutils/GSCollections ArrayList	84%	▲ +2x contains ▲ +4x remove ▲ +12x/+3x/+13x copy	▲ +3x contains ▲ +2x remove ▲ +4x/+2x+4x copy
To reduce JCF HashSet overhead and improve time performance			
R2) Replace it by Koloboke HashSet	88%	▲ +1.5x iterate ▼ -1.5x remove ▲ +16x copy	▲ +3x iterate ▲ +1.5x contains ▲ +51x copy
To reduce JCF HashMap overhead with smallest time penalty			
R3) Replace it by GSCollections/Fastutils HashMap	78%	▼ -1.5x populate ▼ -1.5x/-3x iterate ▼ -1.5x copy	▼ -1.5x copy
To reduce JCF HashMap overhead and improve copy performance			
R4) Replace it by Koloboke HashMap	78%	▲ +6x copy ▼ -6x populate ▼ -11x remove	▲ +11x copy ▼ -2x populate ▼ -4x remove

	Footprint Savings	Speedup/Slowdown 1k elements	Speedup/Slowdown 1M elements
To reduce JCF collections footprint and improve time performance			
R5) Replace ArrayList by GSCollections primitive-collection	60%	▲ +3x populate ▲ +2x contains ▲ +2x remove	▲ +2x populate ▲ +4x contains ▲ +2x remove
R6) Replace HashMap by GSCollections/Koloboke primitive-collection	76%	▲ +1.5x/2x populate ▼ -2x iterate ▲ +2x remove	▲ +2x populate ▼ -2x iterate ▲ +6x/+4x copy
R7) Replace HashSet by GSCollections/Koloboke primitive-collection	84%	▲ +2x populate ▲ +2x/+1.5x iterate ▲ +27x/7x copy	▲ +4x populate ▲ +4x/+3x iterate ▲ +33x/+7x copy

differences in collections. We discuss these in the following.

Distinct API calls. The implementations often differ in usage of the API call for copying an array. Here two methods are used: `System.arraycopy()` and `Arrays.copyOf()`. The latter is just a wrapper for `System.arraycopy()` which requires only the original array, as opposed to the System version, where the target must be passed as a parameter. Our experiments showed that `System.arraycopy()` is 25% faster than `Arrays.copyOf()` for arrays up to 1 million elements. This is one of the reasons why GSCollections ArrayList (which calls `System.arraycopy()`) is faster than JCF counterpart, since *populate* scenario is dominated by the buffer expansion cost.

Add copy versus Memory Copy. Collections copy is the scenario with the largest discrepancy in performance of our experiments. We found that libraries implement two main approaches: they either add all elements one by one to a new created collection instance, or they perform a memory copy. Needless to say, the memory copy is faster as adding elements one by one has the burden of manipulating objects individually. This explains why Koloboke executes up to 50x faster than JCF when copying a set with 1 million elements.

Nevertheless, libraries often opt for adding the elements into the new instance for the simplicity of the workflow. The copy constructor receives a collection reference as a parameter and must be able to polymorphically handle all kinds of collection types. A memory copy is restricted only to copies from the same type. GSCollections and JCF use memory copy for ArrayLists but rely on addAll for HashSet and HashMap. Koloboke is the only library that use memory copy for HashSet/HashMap. It is important to address that copy is called 11% of the HashSet instantiations. Therefore applications can strongly benefit from memory copy implementation.

Sub-optimal primitive API. Iterating elements of FastList's ArrayList primitive collection is slower than JCF ArrayList, and far more time-consuming than any other primitive implementation. To understand this, note that each primitive library provides a `forEach(IntProcedure)` method to iterate and process the list. However, FastList provides a `forEach()` method defined by JCF `AbstractCollection`, which accepts an `Object` and implicitly converts each primitive to its wrapper, degrading the performance by a factor of 5x. This case illustrates the complexity of a collection API, which often provides multiple ways of performing the same task, but with distinct performance costs.

Sub-optimal loop implementation. The `contains()` method of Trove primitive ArrayList is 3 times slower than any other primitive implementation (see Fig 9a). The code inspection reveals that contrary to other primitive alternatives, Trove defined the array loop using an unconventional for loop, namely `for ( int i = offset; i- > 0; )`. This loop, albeit correct, produces three times more branches and 30% more branch misses than a backwards loop defined with the decrement in third position of the `for`-statement. In fact, after refactoring the `contains()`-method in an obvious way in our own extension of Trove's primitive ArrayList the performance bottleneck was fixed.

6.3 Threats to Validity

There are several issues that may affect the validity of our work. As a benchmark study, our results are subject to a number of external factors. Even though we mitigate their influence with our methodology (see Section 4), the hardware specs may have an impact on collections time performance [11]. In particular, the speedup/slowdown indicators might be biased towards our test machine configuration. The memory footprint results, however, can be generalized beyond the hardware specification.

We try to reuse the same code to provide a homogeneous benchmark for the different collection libraries. Some implementations are not JCF compliant though, e.g. HPPC implementations and most primitive collections. For those cases we carefully adapt our code to their respective API, introducing only subtle code differences. Still, this may impact time performance of evaluated collections.

Lastly, we do not consider workloads with a convoluted set of operations. For such workloads, the memory overhead and execution time can be impacted by complex interactions among operations. Such workloads are harder to generalize though. That is why we based our micro-benchmark on single operations so that a useful guideline can be derived from the results.

7. RELATED WORK

There is a substantial body of research devoted to understanding the impact of collections on code performance.

Empirical Studies of Java Collections. To the best of our knowledge our evaluation is the most comprehensive study of the space and time profiles of alternative collection implementations in Java. However, the benchmarks we evaluated partially cover some aspects featured in our experiments. In [25] the author evaluates different HashMaps libraries in more convoluted scenarios. Lewis [15] compares JCF implementations and suggests a new collection for a specific GUI update scenario. Another interesting paper [14] exemplifies the time and memory trade-offs of HashMaps. Our work differs from these in some key aspects. First, we evaluate a wider range of implementations, presenting the performance profile of lists, sets and maps, including primitive alternatives, while those works focus more commonly on HashMap alternatives. Second, we based our experimental planning on patterns found on real code (Section 3). This allows us to evaluate scenarios more commonly adopted by developers in real applications, and provide more representative results. Lastly, we investigate and report some patterns in real code that explain some performance differences, an aspect rarely explored in the benchmarks we have found.

More recently, some works have studied the collections selection problem in relation to energy consumption of applications [10]. In particular, Hasam [9] presented an inspiring work which experimentally explores the impact of collections on energy consumption and concludes that some implementations can increase the energy usage of an application up to 300%. Their work does not consider time/memory profile as we do, and we believe that time, memory and energy consumption profiles can be combined to give programmers a more complete view of collections performance.

Collection inefficiencies. Several authors have designed methods to detect collection inefficiencies. Xu et al, [28] use both static and dynamic analysis to detect overpopulated and underutilized collections. Yang et al, [29] tracks and records the containers and the flow of its elements, and uses an offline analysis to pinpoint some inefficiencies. Those works target performance problems related to collection without exploring the effect of different collection types. Thus the authors attempt to solve an orthogonal problem to collection selection.

Performance optimization by collection replacement. Recently, several studies have attempted to advise developers on selecting collection abstraction type which improves the performance in a context-dependent way. Shacham et al., [23] optimizes applications by monitoring and replacing the collections types using a user-defined model. Coco [26] also utilizes a user defined model to replace the collections at runtime. Both works differ from our approach as they deploy solely asymptotic analysis to select the most suitable collection type, without accounting for the real performance profile. Consequently, the authors do not consider alternative collection libraries, and focus only on JCF.

The approach of Brainy [11] is most closely related to our work. Brainy is a C++ framework which generates a large amount of synthetic applications, models their performance via machine learning, and suggests appropriate collection replacements. This framework does not consider alternative implementations, but rather, focuses on a set of specific cross-abstraction transformations which might be difficult to apply to real code. Our work targets understanding the performance of alternative libraries, and we provide a guideline for replacing collection implementations for scenarios with a high impact on performance.

8. CONCLUSIONS AND FUTURE WORK

We analyzed usage patterns of collections by mining a code corpus of 10,986 Java projects and found that even in high-quality code, the usage of alternative collections libraries is rare. To investigate the potential for performance improvement in Java applications with only moderate programming effort, we have conducted a rigorous performance evaluation study of the standard Java Collection Framework and six most popular alternative collection libraries. We found that such alternative libraries can offer a programmer a significant reduction of both execution time and memory consumption.

We summarized the impact of collection replacement on runtime improvements and memory savings for some relevant scenarios. The resulting guideline assists programmers in deciding whether in a given setting an alternative implementation can offer a substantial performance improvement, and if so, which implementation should be used. Some of these replacements are easy to automate and can improve the performance profile of collections through simple code refactoring.

As future work, we target implementation of such automated refactoring tools. We also intend to evaluate our recommendations in more complex scenarios, including a study of their performance impact in large, mature software projects.

9. REFERENCES

[1] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 207–216, Piscataway, NJ, USA, 2013. IEEE Press.

[2] Chronicle. Koloboke. http://chronicle.software/products/koloboke-collections/, Oct. 2015.

[3] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, Oct. 2007.

[4] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. *SIGPLAN Not.*, 43(10):367–384, Oct. 2008.

[5] J. Y. Gil and Y. Shimron. Smaller footprint for java collections. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '11, pages 191–192, New York, NY, USA, 2011. ACM.

[6] GNU Trove. Trove. http://trove.starlight-systems.com/, May 2015.

[7] Goldman Sachs Group, Inc. GS Collections. https://github.com/goldmansachs/gs-collections, June 2015.

[8] Google. Guava. https://github.com/google/guava, Aug. 2014.

[9] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 225–236, New York, NY, USA, 2016. ACM.

[10] N. Hunt, P. S. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *2011 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 63–70, Feb 2011.

[11] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: Effective selection of data structures. *SIGPLAN Not.*, 46(6):86–97, June 2011.

[12] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[13] K. Kawachiya, K. Ogata, and T. Onodera. Analysis and reduction of memory inefficiencies in java strings. *SIGPLAN Not.*, 43(10):385–402, Oct. 2008.

[14] R. Leventov. Time - Memory Tradeoff With the Example of Java Maps. https://dzone.com/articles/time-memory-tradeoff-example.

[15] L. Lewis. Java Collection Performance. https://dzone.com/articles/java-collection-performance, July 2011.

[16] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 245–260, New York, NY, USA, 2007. ACM.

[17] D. C. Montgomery. *Design and Analysis of Experiments*. Wiley, Hoboken, NJ, 8 edition edition, Apr. 2012.

[18] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of java profilers. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[19] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA, 2014. ACM.

[20] Oracle. Java development kit. https://www.oracle.com/java/index.html, Sept. 2015.

[21] S. Osiński and D. Weiss. HPPC: High Performance Primitive Collections for Java. http://labs.carrotsearch.com/hppc.html, Jan. 2015.

[22] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 155–165, New York, NY, USA, 2014. ACM.

[23] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 408–418, New York, NY, USA, 2009. ACM.

[24] S. Vigna. Fastutil. http://fastutil.di.unimi.it/l, Jan. 2016.

[25] M. Vorontsov. Large HashMap Overview. http://java-performance.info/hashmap-overview-jdk-fastutil-goldman-sachs-hppc-koloboke-trove-january-2015/, Feb. 2015.

[26] G. Xu. Coco: Sound and adaptive replacement of java collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 1–26, Berlin, Heidelberg, 2013. Springer-Verlag.

[27] G. Xu and A. Rountev. Precise memory leak detection for java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 151–160, New York, NY, USA, 2008. ACM.

[28] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 160–173, New York, NY, USA, 2010. ACM.

[29] S. Yang, D. Yan, G. Xu, and A. Rountev. Dynamic analysis of inefficiently-used containers. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*, WODA 2012, pages 30–35, New York, NY, USA, 2012. ACM.