CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection

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.andrzejak@informatik.uni-heidelberg.de

Abstract

Selecting collection data structures for a given application is a crucial aspect of the software development. Inefficient usage of collections has been credited as a major cause of performance bloat in applications written in Java, C++ and C#. Furthermore, a single implementation might not be optimal throughout the entire program execution. This demands an adaptive solution that adjusts at runtime the collection implementations to varying workloads.

We present CollectionSwitch, an application-level framework for efficient collection adaptation. It selects at runtime collection implementations in order to optimize the execution and memory performance of an application. Unlike previous works, we use workload data on the level of collection allocation sites to guide the optimization process. Our framework identifies allocation sites which instantiate suboptimal collection variants, and selects optimized variants for future instantiations. As a further contribution we propose adaptive collection implementations which switch their underlying data structures according to the size of the collection.

We implement this framework in Java, and demonstrate the improvements in terms of time and memory behavior across a range of benchmarks. To our knowledge, it is the first approach which is capable of runtime performance optimization of Java collections with very low overhead.

CCS Concepts • **Software and its engineering → Data types and structures**; **Software performance**; **Software libraries and repositories**;

Keywords data structure, performance, optimization, adaptive algorithms

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168825

ACM Reference Format:
Diego Costa and Artur Andrzejak. 2018. CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3168825

1 Introduction

Collections are data structures that group multiple elements into a single unit. Given that modern programs use collections in thousands of program locations [15], [22], selecting appropriate type and implementation is a crucial aspect of developing efficient applications. Choosing a wrong collection variant may result in performance bloat, i.e. unnecessary or even excessive use of memory and/or computational time. Numerous studies have identified the inappropriate use of collections as the main cause of performance bloat [10, 19]. Even in production systems, the memory overhead of individual collections can be as high as 90% [5].

Selecting a collection variant suitable for a given application and its workload depends on many factors. The most common method used to choose the collection type and its implementation is the asymptotic model. The asymptotic model offers a suitable approach for understanding the time complexities of different data structure operations, however it might lead to wrong conclusions in a real usage context.

In the realm of collections performance, constants matter. For instance, using map implementations (e.g. HashMap or TreeMap in JDK) offers constant or logarithm asymptotic search times, but if collections have only few elements, a linear search on an array might perform much better due to effects of caching and memory locality.

In reality, programmers often rely on standard libraries and select only a handful of implementations for the development of a program. An empirical study [6] of open-source Java projects showed that a majority of all declared collections use only three JDK implementations: ArrayList, HashSet and HashMap. Alternative implementations of the same collections are rarely used. However, such alternatives can provide substantial performance improvements under certain conditions.

Another challenge is the fact that the workload of a program is dynamic and might drastically change during the

execution. Many studies have shown that a real-world execution often consists of multiple phases, and workloads (as well as program behavior) can change many times during a single run [14, 23]. In many cases, it has proven to be impossible to find a single optimal solution for the whole program run [25].

While crucial for performance, selecting a suitable collection type and implementation might be a complex task, creating an additional problem to be faced by the developers. Consequently, this calls for automated solutions that can analyze the application workload, and decide which collection variant to use based on the current usage scenario.

We propose an approach and its implementation for dynamic (runtime) selection and optimization of collection variants. Particular attention was given to efficiency, resulting in a negligible overhead despite of runtime adaptation capabilities. Our approach works at two levels of granularity: at the level of a collection allocation site, and at the level of individual collection instances.

In the first case, our method modifies the allocation sites of collections in order to monitor the created instances, and to potentially change the implementation variants created at these sites. To this aim the approach analyzes the workloads of previous instances created by a specific allocation site. On basis of this data and user-defined performance rules it decides whether the considered allocation site should use other variants of collection implementations for future instantiations, and which variants.

To optimize on the level of individual collection instances, we introduce adaptive collection variants capable of changing the internal data structures depending on the collection size. An example is *AdaptiveSet* which changes the internal data structure from an array to a hash table above a certain number of elements. In this way they ensure low memory overhead without performance penalty when the number of elements increase.

Contributions. The contributions of this work are the following:

1. An approach for dynamic (runtime) selection of collection implementations. The choice of collection variants optimizes (for a specific allocation site) performance along multiple dimensions according to the workload profiles of monitored instances and according to configurable rules.
2. CollectionSwitch: a low-overhead, concurrent implementation of this approach as an application-level library.
3. An analysis and implementation of adaptive collections which dynamically switch their underlying data structure according to the size of the collection.
4. An empirical evaluation of both concepts and their implementation on synthetic benchmarks and real applications.

2 Background and Motivation

Terminology. In Java, collections are objects that group multiple elements into a single unit. A collection uses one or more data representation to efficiently store, manipulate and provide access to the held elements. For instance, an `ArrayList` is a list implemented with the array representation, while the `LinkedHashMap` uses both a hash-table and a linked-list to provide an ordered map with constant element access time. For this work, we refer to data representation as the *collection implementation*.

Each collection implements a set of operations defined by an abstract data type. The abstract data type, hereby denoted as the *collection abstraction*, binds a semantic contract for the collection, but leaves the implementation open for different approaches. Multiple *variants* of the same implementation can then be selected impacting the underlying performance of the program without compromising its functionality.

2.1 The Collection Selection Problem

Definition. Given a program and its usage scenario, the *collection selection problem* concerns finding a set of collection implementations which optimize some performance-related criteria, typically in context of the usage scenario. Multiple performance criteria can be considered for this problem, such as the execution time [22], memory footprint and allocation [6], and energy consumption [13]. In this work, we focus on a multi-dimension improvement of time and space, that is, we aim finding collection variants that minimize the execution time to perform the operations and/or reduce the space and allocation required to hold the elements.

Online Solutions. Online solutions propose to shift the responsibility of selecting an implementation from the developer to the runtime, realized by adaptive collections. As the name suggests, an adaptive collection adapts its own implementation to another *variant*, better suited for its current usage scenario. Apart of taking away this burden from the developer, the adaptive collection also tackles the problem of optimizing programs with multiple workload patterns. Several works [1, 7, 25] have shown that a single implementation and algorithm is often not optimal for the entire program execution, specially on large-scale softwares.

From the proposed adaptive approaches, two major strategies became dominant in recent studies:

1. Adaptivity with Instant Transition. The adaptive collection performs a complete transition (copy) to the most adequate variant triggered by workload analysis [21].
2. Adaptivity with Gradual Transition. The adaptive collection simultaneously uses multiple implementations, and gradually switches to the most adequate one when certain conditions apply [25].

Instant Transition. The first approach is more generally applicable as it can provide improvements for both memory

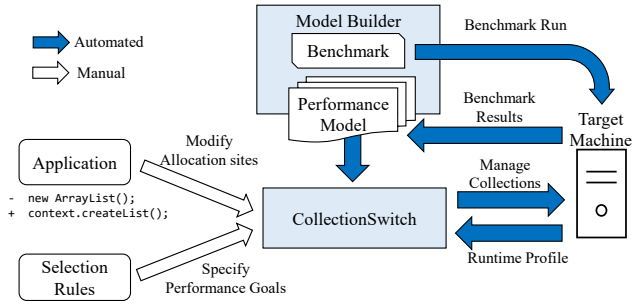


Figure 1. Overview of the processes and dependencies used in our approach.

and time. For example, it can achieve a reduction of the memory footprint for maps via the following strategy. Initially, a map is realized via an `ArrayMap`, a memory efficient variant of the map abstraction. For a small number of elements this incurs no penalty for lookups and can significantly reduce the footprint of an application. As the collection grows in size, it replaced its internal data structure by a traditional `HashMap`, thus avoiding the penalty of a linear search on larger sizes.

In some cases this approach might yield a substantial performance degradation. The H2 application in DaCapo benchmark [3] provides an example of such behavior, as few allocation sites generate the majority of collection instances in the large benchmark. In particular, the allocation site `IndexCursor:70`, instantiates +1 million objects in a few seconds. The above mentioned technique causes 12% of performance degradation without reducing the memory footprint of the program. This can be attributed to the fact that the majority of created instances were short-lived, and about half of the instances triggered a collection transition.

The instant transition approach has inherent shortcomings in terms of execution performance. The swap of the implementation costs both time and allocation and should only be performed when the analysis predicts that future usage will benefit from the alternative data structure.

Gradual Transition. In the second approach, the developer is actively trading memory for potential time improvement. Each collection maintains multiple implementations with a similar state (elements) and gradually changes the active implementation to a more optimal variant. This effectively reduces the cost of transitioning the implementation but cannot be used for memory improvements. Coco [25] proposes this method and evaluates it on the DaCapo benchmark.

3 Approach

The optimization of collection variants takes place at two levels of granularity: at the level of a collection allocation site, and at the level of the individual collection instances.

- **Allocation site-level adaptation.** We modify the allocation sites of collections in order to enable them for workload-aware selection of the variants created during future instantiations (Section 3.1). To this aim we monitor and analyze the behavior of previous instances created by a specific allocation site, and decide on switching to other variants according to user-defined performance rules. The overall approach is illustrated in Figure 1.
- **Instance-level adaptation.** We introduce adaptive collection variants capable of changing the internal data structures depending on the collection size (Section 3.2). Such variants are suitable if collection instances created by the same allocation sites can have both only few or a large number of elements.

In the first case the selection exploits previously computed performance models, and the runtime data characterizing the workloads of the deployed collection instances (Figure 1). Furthermore, users can influence the selection outcomes via configurable rules.

3.1 Adaptation on Allocation Site-Level

Technically, selected allocation sites are instrumented with a code layer called *allocation context* which creates, monitors and adapts the collections (see Figure 4 and description of the implementation in Section 4.3). Each allocation context initially instantiates default collections variants, as specified by the developer.

A sample of all created collection instances is instrumented and monitored in order to obtain the *workload profiles* of these instances. A workload profile comprises the number of executed *critical collection operations* (listed in Section 4.1.2, such as *populate*, *contains*, or *iterate*) and essentially the maximum size of a collection.

The allocation context periodically evaluates these profiles to decide whether future instantiations should use another collection variant than the current one (i.e. whether to "switch"), and if yes, which variant (Figure 2). After switching to a new variant a fraction of the instances is monitored to allow a continuous adaptation process.

3.1.1 Variant Selection Algorithm

The allocation context in our approach selects a collection variant by considering multiple *cost dimensions* such as execution time and memory overhead. While the interplay of these dimensions is described in Section 3.1.2, we assume in the following a fixed cost dimension D.

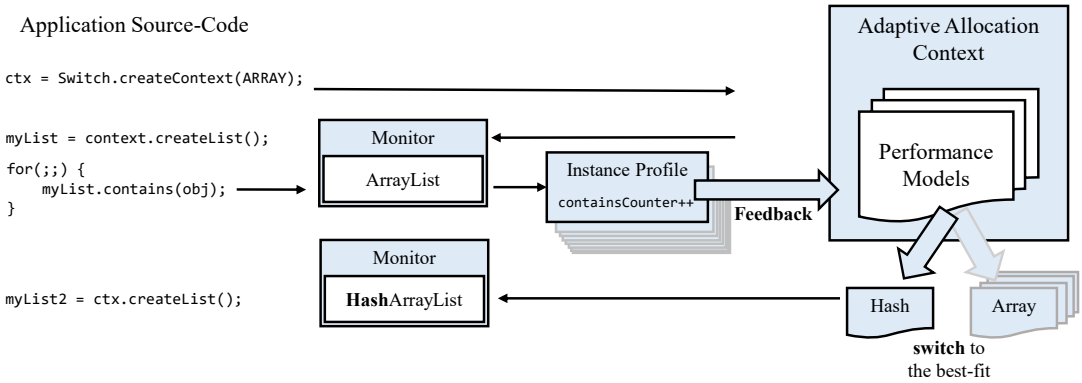


Figure 2. Selection of collection variants by allocation contexts based on the workload profiles on collection instances.

We compare the collection variants V (per cost dimension D) according to the *total cost* $tc(V)$ metric. This metric depends on the observed workload profiles W of the monitored collection instances. In particular, W comprises the numbers of executed critical operations N_{op} during the lifetime of a monitored collection instance, and the maximum size s of this instance. Moreover, tc depends on the performance models obtained for each variant (Section 4.1). These models yield the averaged costs $cost_{op,V}(s)$ of a critical operation op of the collection V depending on s, the maximum size of a collection.

With these preliminaries, we define $tc(V) = tc_W(V)$ as

$$tc_W(V) = \sum_{op} N_{op,W} * cost_{op,V}(s).$$

Note that $tc(V)$ only estimates the total cost of all operations under a workload, since we use the *maximum* collection size s as an argument to a performance model $cost_{op,V}(s)$, and not the actual collection size when a specific operation is executed. As the cost of an operation are typically larger with growing s, the value of $tc(V)$ is an overestimate.

The above description assumed workload data from only a single monitored collection instance. In reality we monitor multiple such instances (per allocation context). We exploit all this data by summing up the total cost over all monitored instances. These sums $TC_D(V)$ (per collection variant V and a cost dimension D) are used when applying the selection rules described below.

3.1.2 Configurable Selection Rules

Typically an improvement on one cost dimension might incur penalties on costs in other cost dimensions. For instance, an `ArrayMap` is memory efficient but has a linear time for access, as no structure is used to index the keys. Adding an index to reduce the access time (hashtable, AVL tree) comes at a cost of higher memory usage.

To account for such trade-offs, we introduce configurable *selection rules*. Such a rule R consists of one or more criteria (predicates) $C_1, C_2, \ldots$, each corresponding to a unique cost dimension (e.g. execution time, memory overhead, or energy usage). A collection variant is selected by R if all of the criteria are satisfied. A criterion C_i is satisfied if the ratio of the total cost $TC_D(V_{new})$ of a candidate collection variant V_{new} (i.e. a potential replacement) by the total cost $TC_D(V_{cur})$ of the current variant V_{cur} is not larger than a user-specified threshold T_D. In other words, C_i is satisfied if

$$\frac{TC_D(V_{new})}{TC_D(V_{cur})} \leq T_D.$$

Note that $T_D < 1$ enforces a cost reduction, while $T_D \geq 1$ expresses a maximum penalty incurred by the candidate variant. Table 4 shows examples of some typical selection rules.

During periodical evaluation of workload data for a given allocation context, we switch the collection variant if a selection rule finds a variant different from the currently used one. If multiple candidates satisfy all the criteria, we select a variant with a largest improvement on the first criterion C_1.

3.2 Adaptation on Instance-Level via Adaptive Collections

Adaptive collections are able to change their internal data structure depending on the current size of the collection (i.e. number of contained elements). The motivation for such data structures is that for small collection sizes, operations such as element search (or set `contains`) require comparable or even shorter time if it is implemented as linear search on an array as compared to a lookup in a proper hash table. However, using an array as the underlying data structure reduce significantly the memory footprint.

In this work we studied adaptive collections which change the underlying data structure from an array (lower memory overhead but linear search) to a hash table (higher memory

Table 1. Adaptive collection types studied in this work, their transition types, and the optimal transition thresholds (in terms of the collection size).

Col. Variant	Transition	Threshold
AdaptiveList	*array* → *hash*	80
AdaptiveSet	*array* → *openhash*	40
AdaptiveMap	*array* → *openhash*	50

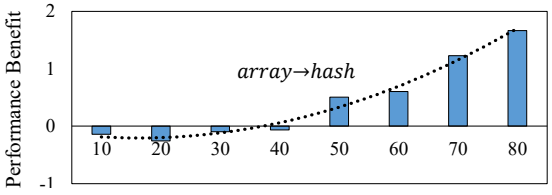


Figure 3. Transition threshold analysis of AdaptiveSet. The optimal threshold for transitioning from array to hash table is at collection size of 40.

overhead, constant lookup time). For the hash tables, *hash* denotes an implementation which creates a bag of keys in case of hash function conflicts (similarly to JDK's HashMap implementation). The *openhash* version solves hash function conflicts by shifting the key placement to the next free position in the underlying table. Table 2 lists the three adaptive data structures and their transition types.

In CollectionSwitch, such adaptive variants are considered as candidates for future instantiations only if the corresponding allocation context has identified that the previously created collection instances had widely ranging sizes.

Transition threshold of adaptive collections. The performance behavior of an adaptive collection is significantly influenced by the criteria for changing the internal representation. To create performance models of these data structures, we had to optimize and fix the transition thresholds for all studied variants (Table 2). We used the lookup search as the scenario for finding this threshold, since our adaptive collections attempt to optimize element search.

We calculate the transition threshold by finding the collection size for which the cost of transition to a hash table would be surpassed by the cost of calling the lookup operation for every collection element. Figure 3 illustrates this method for the AdaptiveSet. The optimal thresholds for each adaptive collection are shown in Table 1.

4 Implementation

Our framework is composed of two components: the performance model builder and a library. The performance builder creates performance models of collection variants via benchmarking. The CollectionSwitch library exploits these performance models at runtime for adaptive selection of collection variants (Section 3.1.1).

4.1 Performance Models via Benchmarking

The underlying hardware plays an important role in the collection selection problem. In the evaluation shown in [15], different hardware architectures yield distinct best data structures given the same applications. This substantiates the need for hardware-specific benchmarking and performance modeling as a prerequisite to optimization of collection selection. Another benefit of such benchmarking is uncovering the performance differences hidden by the asymptotic analysis.

4.1.1 Considered Collection Variants

In this paper we consider and benchmark multiple implementations of the most used collection abstraction types, namely Lists, Sets and Maps. The variants used in this study are shown in Table 2.

In order to have a compelling search space to explore, we select implementations from both JDK and alternative collection libraries. We consider implementations from Koloboke [16], EclipseCollections [8] and FastUtil [24] due to their good overall performance.

Additionally, we include implementations not provided by a collection library, such as the `ArraySet/ArrayMap` provided by Google HTTP Client [11] and Stanford NLP [12] Those variants have a narrow best-case scenario, but offer a substantial improvement when used in the right circumstances.

4.1.2 Computing the Performance Models

To compute the performance models, we run a set of benchmarks using a factorial experimental plan [20], designed to evaluate each collection variant in a wide scope of usage scenarios (see Table 3). Each usage scenario is composed of a single operation executed on a range of collections size from 1 to 10K. To reduce the time of the benchmark, we only evaluate *critical operations*, i.e., operations that have in at least one variant a linear or above asymptotic cost ($O(n)$). Consequently, we evaluate collections when adding elements to the collection (*populate*), searching for an element (*contains*), traversing (*iterate*) and adding/removing an element in the middle (*middle*), which is linear on array and linked implementations.

The empirical model was built considering only the `Integer` data type. Albeit having an impact on the performance of some operations, we believe this impact will be dwarfed by the differences of performances caused by the collection implementation. The data distribution, on the other hand, can impact hash structures as it has an influence on the collisions. We only consider uniform distribution in this model.

We build the benchmark using the support of JMH[1] framework, a Java harness framework for building, running and analyzing benchmarks. The JMH provides a native method for collecting the execution time, and we use the JMH GC

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

Table 2. Collection implementations identified as candidates for variants.

Collection Abstraction	Implementation	Implemented by	Description
Lists	ArrayList	JDK	Array-backed list
	LinkedList	JDK	Double-linked list
	HashArrayList	Switch	ArrayList + HashBag for faster lookups
	AdaptiveList	JDK → Switch	ArrayList on small sizes and HashArrayList for large sizes
Sets / Maps	HashSet / Map	JDK	Chained hash-backed set/map
	OpenHashSet / Map	Koloboke, Eclipse, Fastutil	Open-address Hash-backed set/map
	LinkedHashSet / Map	JDK	Chained hash-backed with double-linked entries
	ArraySet / Map	Fastutil, Google, NLP	Array backed set/map
	CompactHashSet / Map	VLSI	Byte-serialized map for high memory efficiency
	AdaptiveSet / Map	NLP/Google → Koloboke	Array-backed on small sizes and Hash-backed on large sizes

Table 3. Factors and levels adopted in the empirical evaluation of collections.

Factor	Levels/Categories
Collection Size	[10,50,100,150,..,1000]
Scenarios	populate, contains, iterate, middle
Data Type	Integer
Data Distribution	Uniform

profiler to retrieve the memory allocated and footprint required in each scenario. Each iteration runs for five seconds, executing the defined scenario continuously and returning the average of the measured performance indicators. We run 15 unmeasured iterations to achieve the steady-performance, and use the average results of 30 measured iterations in our performance model.

Modeling collection performance. We model the cost of each critical operation as a polynomial function of the collections size s:

$$cost_{op}(s) = \sum_{k=0}^{d} a_k s^k$$

The coefficients are calculated using the least squares polynomial fit on the results of the benchmark. For this work we use polynomials of third degree ($d = 3$), as this choice provided the small residuals, while polynomials of higher degree did not increase the least-square fit significantly.

4.2 The CollectionSwitch Library

We designed the CollectionSwitch as an application library as opposed to a customized Virtual Machine (VM). This design choice was based solely on facilitating the adoption of our framework by developer teams. A modified VM would incur on a harsh constraint, as to use CollectionSwitch applications would be required to deploy using our personalized VM. The CollectionSwitch is open-source library and is available online[2].

[2]https://github.com/DiegoEliasCosta/collectionSwitch

4.3 Adaptive Allocation Context

An allocation context is an instrumented version of a collection allocation site. It creates the collections and monitors a subset of the created instances to obtain workload data, characterizing the current usage scenario. When collections finish their life-cycle, the workload data is passed to the allocation context. As described in Section 3.1, this initiates the performance analysis of collection variants. If a better variant is found, the allocation context switches the current implementation for future collection instantiations and starts another monitoring round.

Specifying an Adaptive Context. The CollectionSwitch acts as middle layer between the application and the collection libraries, collecting information and switching the adaptive allocation sites to the appropriate variant. The allocation context is implemented as a Java object, instantiated before the creation of the collections. The context creation is specified by the programmer via API, or can be automatically generated via code parser.

By creating the context, the developer may choose to use a static or non-static context object. A static context is created as soon as the class is loaded in the class-loader and is kept alive until the end of the application. The usage of static context greatly reduces the potential overhead incurred by the framework, and it is closely related to the concept of tuning the allocation site. However, a developer could use a non-static context if the collections behavior is dependent on the instance that creates it.

We also provide an automated parser that rewrites the code of collection instantiation to the adaptive context required by our framework. The parser only identifies collections already declared as using the JCF interfaces and only uses the static context.

Monitoring the Collections Usage Each allocation context collects metrics on a subset of created instances to characterize their overall collections usage. Example of metrics include the maximum collection size and the amount of critical operation calls. We decide to analyze only a subset to

```
// Original allocation site
List<?> list = new ArrayList<>();
// Modified code with allocation context
static ListContext ctx =
    Switch.createContext(CollectionType.ARRAY);
List<?> list = ctx.createList();
```

Figure 4. Instrumenting collection allocation sites with allocation context.

avoid a potential overhead in case of a huge amount of collection instantiation in a short period of time. The size of this monitored subset is defined by the monitored *window size*, and it is parametrized by the developer. The monitored collections are created with an extra layer called *monitor*, a wrapper that logs the metrics in the context and forwards the collection logic to the proper implementation.

Analyzing the Collections Usage. A vital aspect of the CollectionSwitch implementation is when should the allocation context use the feedback to perform its transitions. In principle, a feedback should be used only when it provides a complete context, ie when the collections already ended its life-cycle. In practice, this delays the decision of the allocation context when collections are retained for too long in memory, hurting the context adaptivity. To address this we created the *finished ratio*, which defines the ratio of monitored collections that needs to be finished, before the context can take any decision. For instance, a *finished ratio* of 0.6 implies that the allocation context will only take action when at least 60% of the instances have finished their execution. It is important to note that we always use the whole set of metrics to analyze the collections usage, the ration only determines when the feedback should be analyzed.

To assess whether the collection object has finished its execution, the allocation context saves a `WeakReference` to the instance. As soon as the collection is eligible for garbage collection, this weak reference returns null, when asked for the referenced object. This method is more reliable and does not incur the substantial overhead caused by the `finalize` method [4].

We implement the analysis of the collections usage using a thread pool to analyze every collections context. A periodic task is scheduled at a parametrized fixed rate (*monitoring rate*). This thread pool can be assigned to a specific processor, to reduce the impact of the analysis on the monitored application time.

4.4 Limitations

The major limitations of the CollectionSwitch are: 1) errors in estimating the true cost of collection execution, and 2) potential increase of susceptibility to faults due to increased complexity.

Table 4. Selection rules R_{time} and R_{alloc}.

Rule	Improvement	Penalty
R_{time}	Time cost < 0.8	–
R_{alloc}	Alloc cost < 0.8	Time cost < 1.2

Estimation errors. Multiple factors contribute to this limitation. First, we use accumulated execution cost, which might hide the true behavior of collections in a real application. For example, short-lived instances and collections executed in parallel can have distinct impact on the application's performance. Also factors such as memory locality and branch misprediction are not considered in our performance model. On the other hand, CollectionSwitch only needs accuracy sufficient to expose the performance *differences* between collection implementations. Using a more or fully accurate model might increase the runtime overhead and thus limit the benefits of the approach.

Increased susceptibility to faults. CollectionSwitch inherently increases the complexity of the target application-code, potentially introducing new defects. Furthermore, the overall complexity of application behavior increases due to introduced adaptivity: different scenarios might yield different collection implementations, increasing the chance of functional bugs and raising the cost of diagnosis. We mitigate this by selecting well-tested implementations, and by providing a detailed log system for tracing framework events.

5 Evaluation

We evaluate the effectiveness of CollectionSwitch on two sets of benchmarks: the *CollectionsBench* [6], a set of micro-benchmarks specifically designed to evaluate collections performance, and *DaCapo*, a set of real-world application benchmarks [3]. We conduct our experiments on a machine with the i7-2760QM 2.40GHz CPU and 8GB RAM under Ubuntu Linux 3.16.0-53 (64 bits).

To optimize applications for time and space dimensions we use two rules shown in Table 4. They target optimizing execution time and memory allocation, respectively. Note that R_{alloc} comprises a maximum penalty allowed on the execution time, otherwise array-backed implementations would be prioritized due to their low memory footprint.

For the whole experiment we use a *window size* of 100 instances, a level that showed a good compromise between fast analysis and stable transitions. Furthermore, the *finished ratio* is set 0.6, and the *monitoring rate* is 50ms.

5.1 Micro-benchmarks

We extended the CollectionsBench benchmark to evaluate the CollectionSwitch through experiments covering scenarios dominated by a single collection operation (single-phased), and scenarios with the dominant operation varying over time (multi-phased). We follow the methodology [9] for evaluating the steady-state performance of Java programs. In

particular, we run each test scenario with 15 unmeasured iterations for warm-up, followed by 30 measured executions.

Single-Phased scenario. In this experiment, each scenario consists of creating and populating 100k collection instances, followed by 100 lookup searches (`contains()`). The lookup operations have different asymptotic complexities on array/hash implementations and showcase an interesting trade-off between time and memory consumption. Figure 5 shows the performance against JDK implementations (ArrayList, HashSet, HashMap) for varying collections size.

CollectionSwitch was able to select variants with better performance on all collection abstractions. In Figure 5a the performance is gained by switching to a `HashArrayList` implementation. On sets and maps (Figures 5b and 5c) this performance was achieved by switching to the Koloboke OpenHash implementation.

In case of R_{alloc}, CollectionSwitch switches multiple times for both sets and maps. On small collections ($size < 400$), FastUtil OpenHash implementation is selected (the most memory efficient variant). For medium size collections, the time penalty for using FastUtil lookup crosses the threshold established by R_{alloc}, and EclipseCollection is selected. A yet better implementation (Koloboke) is identified and used when the collection size reaches 700.

Multi-Phased scenarios. Each iteration of this experiment is comprised of the creation and population of 100k instances followed by an execution of 100 operations. Every five iterations we change the operation type resulting in the phases depicted in Figure 6.

In our experiments our framework switched to the expected best-fit implementation for all phases except for the phase "search and remove". Here the HashArrayList instead of the optimal ArrayList was used. We attribute this to a limitation in our performance model. In summary, our model assumes that cost of removing an element by index is identical on both variants. In reality, HashArrayList implementation is slower as it searches on both hash and array structures.

5.2 Evaluation on Real Applications

DaCapo is a set of application benchmarks widely used as an evaluation tool for a variety of scientific studies. The latest DaCapo version (dacapo-9.12) contains 14 benchmarks from which we use the following five: *avrora*, *fop*, *h2*, *lusearch* and *bloat* (2006 version). We select this subset as container inefficiencies were previously reported [21, 22, 25] on those projects.

CollectionSwitch requires modifications of existing code to use the adaptive allocation contexts. To limit this effort, we modify only allocation sites that yield at least one thousand instances and that comply with JCF interfaces (possibly with little refactoring effort), hereby denoted as *target allocation sites*. To this end, we first monitor a regular benchmark execution and rank the allocation sites by the number of generated instances.

To compare the effectiveness of the full framework against the benefits of a simple adaptivity at instance-level we use two modified versions of DaCapo. In the first version (*FullAdap*), each target allocation site is modified to use our allocation context, providing full framework capabilities. In the second version (*InstanceAdap*), the target allocation sites are simply changed to always instantiate an adaptive variant (e.g. ArrayList → AdaptiveList), i.e. without the capability of selecting the collection types through the analysis of previous instances.

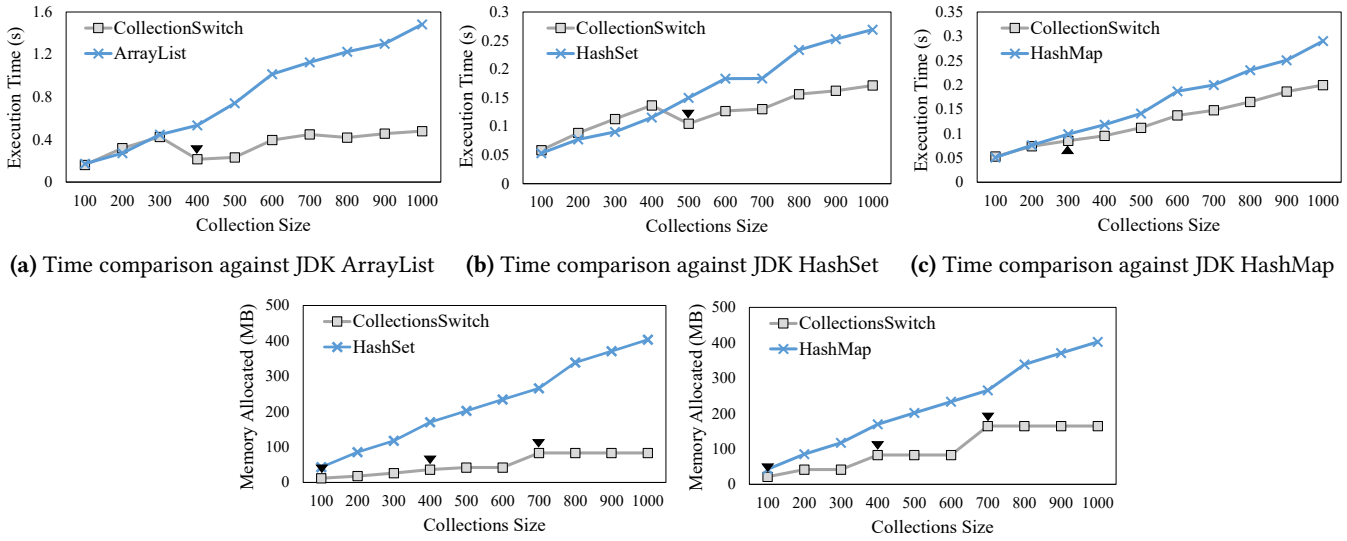
In this experiment we run the original and the two modified DaCapo benchmarks 35 times with a maximum heap size of 1 GB (first 5 runs are discared as warm-ups) .

Results summary. Table 5 shows that the time and memory improvement vary considerably by application and selection rule. Overall, in case of *FullAdap*, CollectionSwitch managed to positively impact the execution time and the peak of memory consumption in most of the evaluated applications.

Time Improvement (R_{time}) for *FullAdap*. The largest improvement of the execution time ($\approx$ 15%) was obtained for *lusearch*. The dominant transition in *lusearch* was performed on map variants, as most of its HashMap instances held less than 20 elements, and were replaced by AdaptiveMap and Koloboke OpenHashMap. Thus, as a side effect, CollectionSwitch also reduced the memory peak consumption of *lusearch* by $\approx$ 5%. Benchmarks *fop* and *h2* showed similar characteristics of collection transitions. Both of them have allocations sites that extensively instantiate lists exposed to large amount of lookup calls. CollectionSwitch has correctly transitioned them to AdaptiveList ($array \rightarrow hash$). This transition improved the execution time of *h2*, but provided no significant improvement for *fop*.

Memory Improvement (R_{alloc}) for *FullAdap*. In case of R_{alloc}, CollectionSwitch managed to reduce the peak of memory consumption of most of the applications. Typically, HashSet and HashMap were replaced by adaptive and openhash variants. Interestingly, the *bloat* benchmark had a lower execution time in this case (R_{alloc}) than when when aiming at the time reduction (R_{time}). We conjecture that a reduction of memory usage implied in a better cache utilization and lower Garbage Collection time.

Comparizon of *FullAdap* and *InstanceAdap*. Table 5 compares the effectiveness of both levels of optimization, *FullAdap* and *InstanceAdap*. The simpler version *InstanceAdap* featured comparable (but not better) improvement grades for memory usage as the full framework under the rule R_{alloc}. However, the *InstanceAdap* version failed to achieve any improvement on the execution time, especially in comparison with the full CollectionSwitch under the R_{time}. These results indicate that allocation-site adaptivity is essential for improvement of execution time Such mechanism help to consider multi-dimensional criteria (memory and *time*),



(a) Time comparison against JDK ArrayList **(b)** Time comparison against JDK HashSet **(c)** Time comparison against JDK HashMap

(d) Allocation comparison against HashSet **(e)** Allocation comparison against HashMap

Figure 5. CollectionSwitch performance on populating 100k collections and executing 100 random lookups (`contains`). CollectionSwitch was evaluated with R_{time} for execution time and R_{alloc} for allocation. The marker indicates a size where CollectionSwitch performed a transition to a different variant.

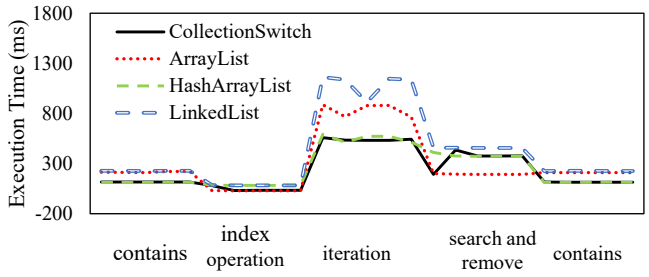


Figure 6. CollectionSwitch time performance on a multi-phased scenario, evaluated with R_{time}.

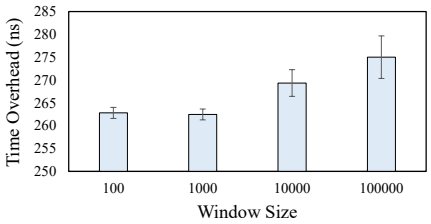


Figure 7. Overhead of analyzing the collections metrics by *window size*.

which prevents the uncontrolled performance degradation which might be introduced by adaptive collection.

Common Transitions. Table 6 shows the most frequently selected transitions (by application and by selection rule). Only 11 out of the 25 possible variants were used in our evaluation. This indicates that a small set of collection variants might be sufficient to cover most of the real cases found in the applications. Moreover, in most allocation sites our framework replaced the original implementation. We conjecture that our approach offers unexploited potential for improving performance in applications.

5.3 Overhead of the CollectionSwitch Framework

To evaluate the time and space overhead incurred by our approach on DaCapo benchmark, we compare the performance statistics of the unmodified benchmarks against such statistics under CollectionSwitch (*FullAdap*) with disabled optimization actions. We achieve the latter by setting an impossible selection rule (required 1000x time/space improvement for a transition). We found no significant difference (Tukey HSD) in the execution time in any DaCapo benchmark when using our standard configuration.

Additionally, we run a micro-benchmark to evaluate the cost of analyzing a set of collection metrics while varying the *window size* from 100 to 100k. Figure 7 confirms that the overhead is negligible ($< 285ns$), and can be easily amortized by a multi-threaded environment.

Regarding the memory overhead, each allocation context has a footprint of $\approx 1KB$. As the amount of allocation context is limited by the amount of allocation sites, we consider the memory overhead of the CollectionSwitch non-significant for real applications.

Table 5. Results of our approach on DaCapo. Section "Original Run" reports the execution time (T) and the peak of memory consumption (M) of the original Dacapo run. Section "Full CollectionSwitch" shows the results for the full optimization level (*FullAdap*) under both selection rules R_{time} and R_{alloc}. Section "*InstanceAdap* " shows the results for lower optimization level, using only adaptive instances. The column "# Target Alloc." states the number of modified target allocation sites. We present the gain/loss percentages for the significant differences (Tukey HSD test [20]) compared to the original run (positive values are better). Non-significant differences are reported as –.

Bench	Input Size	#Target. Alloc.	Original Run T(s)	Original Run M(MB)	Full CollectionSwitch (*FullAdap*) R_{time} $T_1(s)$		$M_1(MB)$		R_{alloc} $T_2(s)$		$M_2(MB)$		*InstanceAdap* $T_3(s)$		$M_3(MB)$	
avrora	large	7	4.1	72.4	4.2	–	72.1	–	4.4	-7%	65.4	+10%	4.4	-7%	64.9	+10%
bloat	large	17	30.3	96.7	28.9	–	96.9	–	26.6	+12%	89.4	+8%	29.5	–	89.6	+8%
fop	default	15	0.5	53.4	0.5	–	57.0	-7%	0.5	–	53.9	–	0.5	–	53.8	–
h2	large	10	40.1	509.0	38.3	+6%	508.7	–	44.6	-11%	470.1	+8%	44.9	-12%	493.2	+3%
lusearch	large	12	3.6	282.4	3.1	+15%	269.4	+5%	3.4	+6%	268	+5%	3.5	–	275.7	+2%

Table 6. Most commonly performed transitions (AL = ArrayList, LL = LinkedList, HS = HashSet, HM = HashMap).

Benchmark	R_{time}	R_{alloc}
avrora	*HS → OpenHashSet*	*HS → AdaptiveSet*
bloat	*LL → AL*	*HS → AdaptiveSet*
fop	*AL → AdaptiveList*	*AL → AdaptiveList*
h2	*AL → AdaptiveList*	*HS → ArraySet*
lusearch	*HM → OpenHashMap*	*HM → AdaptiveMap*

6 Related Work

Offline Collection Selection. A substantial body of work has been conducted on proposing approaches that monitor the application usage and report beneficial code transformations through an offline model. Liu and Rus [17] provide an analysis tool that identifies suboptimal patterns on C++ including patterns related with STL data structures. Shacham *et al.* [22] presented Chameleon, a tool that uses both collection traces and heap analysis to select the best collection variant, given a pre-defined set of rules. In 2011, Jung *et al.* proposed Brainy [15], an approach that relies on a machine-learning model to advise the developer. Similarly to our approach, Brainy uses a set of benchmarks to analyze the performance based on the specified hardware. Moreover, approaches such as SEED [18] use exhaustive search to reduce the energy consumption and Basios *et. al* [2] uses a genetic-algorithm to recommend collections on a multi-optimization problem.

Aside from tackling the same problem, our approach is fundamentally different than the above mentioned works due to two aspect. First, our framework is able to automatically apply the analyzed optimizations, going one step further towards the auto-tuning of applications. Second, we perform both monitoring and optimization at runtime, adapting the collections to the current application demand.

Adaptive Collection Selection. Xu [25] and Orsterlund *et. al* [21] describe approaches most closely related to ours. CoCo [25] was the first fully automated approach to tackle the collection selection in Java and served as an inspiration for CollectionSwitch. It monitors the application usage at the instance level, where each collection carries multiple data structure representations and gradually transitions to the most appropriate when it seems fit. Orsterlund *et. al* [21] uses context composition to identify scenarios in which transforming the collection would lead to performance benefits.

Our work extends the adaptivity proposed of the collection instance to the allocation context, allowing us to reduce the impact of the adaptive collections and propose an automated method for both time and memory improvement. Our framework can be combined with the adaptive instances proposed by Xu [25] and Osterlund [21], by including their adaptive implementations in our set of variant candidates.

Collections Inefficiencies. Researchers have explored multiple solutions to identify and fix collection inefficiencies, Xu et al [26] have proposed the use of static and dynamic analysis to identify underutilized and overpopulated collections. Other authors have identified inefficiencies within the standard Java implementation [10], and have proposed the use of alternative collections as a beneficial optimization technique [6]. We incorporate some of those works' findings when selecting our variants, but we have yet to explore the benefits of fine tuning collection parameters.

7 Conclusions and Future Work

In this work we presented CollectionSwitch, a framework for runtime selection of collection implementations for Java applications. Our framework monitors the workloads for instantiated collections and selects the collection variants used for future allocations according to the performance goals expressed by users in form of rules. While implemented for Java collections, the concept of exploring allocation-site regularities can be applied to different programming languages and domains.

Our future work will aim to expand the performance model of the CollectionSwitch to other cost dimensions such as energy usage. We will also address a wider set of candidate collections, including concurrent and sorted collections.

References

[1] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. 2009. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/1542476.1542481

[2] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2017. *Optimising Darwinian Data Structures on Google Guava*. Springer International Publishing, Cham, 161–167. https://doi.org/10.1007/978-3-319-66299-2_14

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, 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 Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[4] Joshua Bloch. 2008. *Effective Java (2Nd Edition) (The Java Series)* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[5] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of Memory Inefficiency. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 383–407. http://dl.acm.org/citation.cfm?id=2032497.2032523

[6] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 389–400. https://doi.org/10.1145/3030207.3030221

[7] Mattias De Wael, Stefan Marr, Joeri De Koster, Jennifer B. Sartor, and Wolfgang De Meuter. 2015. Just-in-time Data Structures. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!) (Onward! 2015)*. ACM, New York, NY, USA, 61–75. https://doi.org/10.1145/2814228.2814231

[8] Eclipse Foundation. 2016. Eclipse Collections. https://www.eclipse.org/collections/. (2016). [Online; accessed 10-September-2017].

[9] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. https://doi.org/10.1145/1297027.1297033

[10] Joseph Gil and Yuval Shimron. 2012. Smaller Footprint for Java Collections. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 356–382. https://doi.org/10.1007/978-3-642-31057-7_17

[11] Google. 2013. Google HTTP Client Library for Java. https://github.com/google/google-http-java-client. (2013). [Online; accessed 10-September-2017].

[12] Stanford NLP Group. 2013. Stanford CoreNLP. https://github.com/stanfordnlp/CoreNLP. (2013). [Online; accessed 10-September-2017].

[13] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy Profiles of Java Collections Classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 225–236. https://doi.org/10.1145/2884781.2884869

[14] Canturk Isci, Alper Buyuktosunoglu, and Margaret Martonosi. 2005. Long-Term Workload Phases: Duration Predictions and Applications to DVFS. *IEEE Micro* 25, 5 (Sept. 2005), 39–51. https://doi.org/10.1109/MM.2005.93

[15] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. 2011. Brainy: Effective Selection of Data Structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 86–97. https://doi.org/10.1145/1993498.1993509

[16] Roman Leventov. 2013. Koloboke. https://github.com/leventov/Koloboke. (2013). [Online; accessed 10-September-2017].

[17] Lixia Liu and Silvius Rus. 2009. Perflint: A Context Sensitive Performance Advisor for C++ Programs. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. IEEE Computer Society, Washington, DC, USA, 265–274. https://doi.org/10.1109/CGO.2009.36

[18] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-optimization Decision Support Framework. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 503–514. https://doi.org/10.1145/2568225.2568297

[19] Nick Mitchell and Gary Sevitsky. 2007. 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)*. ACM, New York, NY, USA, 245–260. https://doi.org/10.1145/1297027.1297046

[20] Douglas C. Montgomery. 2006. *Design and Analysis of Experiments*. John Wiley & Sons.

[21] Erik Österlund and Welf Löwe. 2013. Dynamically Transforming Data Structures. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 410–420. https://doi.org/10.1109/ASE.2013.6693099

[22] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. Chameleon: Adaptive Selection of Collections. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 408–418. https://doi.org/10.1145/1542476.1542522

[23] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. 2003. Discovering and Exploiting Program Phases. *IEEE Micro* 23, 6 (Nov. 2003), 84–93. https://doi.org/10.1109/MM.2003.1261391

[24] Sebastiano Vigna. 2016. fastutil: Fast and compact type-specific collections for Java. http://fastutil.di.unimi.it/. (2016). [Online; accessed 10-September-2017].

[25] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP'13)*. Springer-Verlag, Berlin, Heidelberg, 1–26. https://doi.org/10.1007/978-3-642-39038-8_1

[26] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 160–173. https://doi.org/10.1145/1806596.1806616