Science of Computer Programming 161 (2018) 34–56

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Reducing resource consumption of expandable collections: The Pharo case

Alexandre Bergel [a,*], Alejandro Infante [a], Sergio Maass [a], Juan Pablo Sandoval Alcocer [a,b]

[a] *Pleiad Lab, DCC, University of Chile, Chile*
[b] *Universidad Mayor de San Simón, Bolivia*

A R T I C L E   I N F O

Article history:
Received 30 January 2017
Received in revised form 11 December 2017
Accepted 19 December 2017
Available online 8 January 2018

Keywords:
Collection
Pharo
Lua
Profiling
Experiment

A B S T R A C T

Expandable collections are collections whose size may vary as elements are added and removed. Hash maps and ordered collections are popular expandable collections. Expandable collection classes offer an easy-to-use API, however this apparent simplicity is accompanied by a significant amount of wasted resources.

We describe some improvements of the collection library to reduce the amount of waste associated with collection expansions. We have designed two new collection libraries for the Pharo programming language that exhibit better resource management than the standard library. We improved the Pharo collection library using two complementary perspectives.

First, across a basket of 5 applications, our optimized collection library significantly reduces the memory footprint of the collections: (i) the amount of intermediary internal array storage by 73%, (ii) the number of allocated bytes by 67% and (iii) the number of unused bytes by 72%. This reduction of memory is accompanied by a speedup of about 3% for most of our benchmarks.

Second, we looked for an alternative to the classical expandable collection. The Lua programming language offers a unique abstract data type called *table*. We designed, implemented, and introduced this data type in the Pharo programming language and we ran a number of micro and macro-benchmarks. Overall, replacing the standard Pharo collection library by one inspired on Lua's table data type results in an execution speedup of up to 15% and a reduction of the memory consumption by up to 19%.

We analyzed the collection implementations of Java, C#, Scala, and Ruby: these implementations largely behave like Pharo's, therefore with the same limitations. Our results are thus likely to benefit designers of future programming languages and collection libraries.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

Creating and manipulating any arbitrary group of values is largely supported by today's programming languages and runtimes [1]. A programming environment typically offers a collection library that supports a large range of variations in the way collections of values are handled and manipulated. Collections exhibit a wide range of features [1–3], including

* Corresponding author.
E-mail address: abergel@dcc.uchile.cl (A. Bergel).

https://doi.org/10.1016/j.scico.2017.12.009
0167-6423/© 2017 Elsevier B.V. All rights reserved.

being expandable or not. An expandable collection is a collection whose size may vary as elements are added and removed. Expandable collections are highly popular among practitioners and have been the topic of a number of studies [4–7].

Expandable collections are typically implemented by wrapping a fixed-sized array. An operation on the collection is then translated into primitive operations on the array, such as copying the array, replacing the array with a larger one, inserting or removing a value at a given index.

Unfortunately, the simplicity of using expandable collections is counter-balanced by resource consumption when not adequately employed [4,5,8]. Consider the case of a simple ordered collection (e.g., ArrayList in Java and OrderedCollection in Pharo). Using the default constructor, the collection is created empty with an initial capacity of 10 elements. The 11th element added to it triggers an expansion of the collection by doubling its capacity. This brief description summarizes the behavior of most of the expandable collections in Java, C#, Scala, Ruby, and Pharo.

We have empirically determined that in Pharo a large portion of collections created by applications are empty. As a consequence, their internal arrays are simply unused. Moreover, only a portion of the internal array is used. After adding 11 elements to an ordered collection, 9 of the 20 slot arrays are left unused. Situations such as this one scale up as soon as millions of collections are involved in a computation.

We have selected the Pharo programming language for our study. Pharo[1] is an object-oriented dynamically typed programming language which offers a large and rich collection library [9]. Pharo is syntactically close to Ruby and Objective-C. Conducting our experiment in Pharo has a number of benefits. Firstly, Pharo offers an expressive reflective API which greatly reduces the engineering effort necessary to modify and replace the collection library. Secondly, the open source community that supports Pharo is friendly and is looking for contributions for improvement, which means that our results are to have a measurable impact across Pharo developers. In principle, our technique may be implemented in an highly optimized environment such as Java or .Net. However, we avoided a statically typed language for two reasons: (i) the runtime and the JIT depend on the Collection library,[2] as such our measurements would measure the implementation of the JIT, which would radically change the focus of the article; (ii) it is unclear whether Lua's tables may be implemented in a statically typed language (again, studying this question would change the focus on the present article). Appendix A briefly presents the syntax of Pharo.

This article is about measuring wasted resources in Pharo (memory and execution time) due to expandable collections. Improvements are then deduced and we measure their impact. We made two improvements to the Pharo collection library.

We improve first the way Pharo collection classes behave by using popular techniques: lazy object creation and recycling objects in a pool of frequently created objects. This article carefully evaluates the application of these well known techniques on the collection implementation. The analyses that this article describes focus on the profiling of over 6 million expandable collections produced by 15 different program executions. Note that our intent is not to prohibit a manual setting of the expansion strategy. Instead, we provide a simple mechanism to complement existing expandable collections.

Second, we looked for an alternative schema of the classical way expandable collections are implemented. Lua is a popular programming language that offers tables, a hybrid abstract data type combining features of sequential collections and dictionaries. We describe tables in Lua and compare their performance with the dictionary and the sequential ordered collection. Our experiments show that when replacing instances of the standard Pharo classes `OrderedCollection` and `Dictionary` by our implementation of Lua's tables, the memory allocation due to collections is decreased by up to 19% when executing long-running benchmarks that make extensive use of collections. It also has an impact on execution time, which is overall decreased, with a maximal reduction of 15%. Our novelty is about (i) porting tables to a language that contains a different library framework, (ii) adapting code using the Pharo standard library to use tables, and (iii) measuring the gain in terms of resource consumption.

The research questions we are pursuing are:

- A – *How to characterize the use of expandable collections in Pharo?* Understanding how expandable collections are used is highly important in identifying whether or not some resources are wasted. And if this is case, how such waste occurs.
- B – *Can the overhead associated with expandable collections in Pharo be measured?* Assuming the characterization of collection expansions revealed some waste of resources, measuring such waste is essential to properly benchmark improvements that are carried out either on the application or the collection library.
- C – *Can the overhead associated with expandable collections in Pharo be reduced?* Assuming that a benchmark to measure resource waste has been established, this question focuses on whether the resource waste accompanying the use of a collection library can be reduced without disrupting programmer habits.

Our results show the Pharo collection library can be significantly improved by (i) considering lazy array creation and recycling those arrays, and (ii) by using hybrid collections (OrderedCollection and Dictionary). The expandable collections of Java, Scala, Ruby and C# are very similar to those of Pharo, and therefore largely exhibit the same deficiencies, as described in Section 10. We therefore expect our recommendations to be beneficial to these languages as well.

[1] http://www.pharo-project.org.

[2] https://stackoverflow.com/questions/33317720/performance-of-collections-emptylist-and-empty-arraylist-with-jit-compiler.

This article is structured as follows: Section 2 describes the Pharo expandable collections and synthesizes their implementation. Section 3 describes a benchmark composed of 5 Pharo applications and a list of metrics. Section 4 details the use of expandable collections in Pharo, both from a static and dynamic point of view. Section 5 details the impact on our benchmark to have lazy array creation. Section 6 presents a technique to recycle arrays among different collections. Section 7 describes an approach to find missing collection initialization. Section 8 presents another technique to improve collections by investigating the use of Lua's tables in Pharo. Section 9 evaluates implementation of Lua's tables in Pharo. Section 10 discusses the case of other languages. Section 11 presents the work related to this article. Section 12 concludes and presents our future work.

This paper is an extended version of our IWST'14 paper.[3] Our IWST submission is not available to any Digital Library (IEEE or ACM).

2. Pharo's expandable collections

This section discusses expandable collections from the point of view of Pharo. However, the problematic situations we present here are found in most expandable collections of other languages (see Section 10 for a detailed comparison).

The collection library is a complex piece of code that exhibits different complex aspects [10]. One of these aspects is whether a collection created at runtime may be resized during the life time of the collection. We qualify a collection with a variable size as "expandable". An expandable collection is typically created empty, to be filled with elements later on. In Pharo 3.0, the collection library is modeled as a set of 77 classes, with each class being a direct or indirect subclass of the root `Collection` class. Out of the 77 classes that compose the Pharo collection library, 34 are expandable. Typical expandable collections include dictionaries (usually implemented with a hash table), lists, growable arrays in which elements may be added and removed during program execution. Interestingly, expandable collections in Pharo, C#, Ruby, Java, and Scala are designed to only expand. Although the internal array may be explicitly trimmed (by using `trimToSize()` in Java), removing elements from a collection does not trigger any shrinkage of the internal collection. We therefore only focus on element addition and not removal.[4]

Issues with expansions. Expandable collections are remarkable pieces of software: most expandable collections have a complex semantics hidden behind a simple-to-use interface. Consider the class `Dictionary`. The class employs sophisticated hashing tables to balance efficiency and resource consumption. Such complexity is hidden behind what may appear as trivial operations. The programmer has to simply address what to add or remove from the collection while the collection implementation takes care of managing the collection's inner storage accordingly.

Expandable collections commonly used in Pharo employ a fixed-sized array as an internal data structure for storage. Adding or removing elements from an expandable collection are translated into low-level operations on the internal storage, typically copying, setting or emptying a particular part of the array storage.

The creation of an expandable collection may be parametrized with an initial *capacity*. This capacity represents the initial size of the array's internal storage. The size of the collection corresponds to the number of elements actually stored in the collection. Adding elements to a collection increases its size and removing elements decreases it (but does not decrease the size of the inner storage array). When the size of the expandable collection reaches its capacity or close to it, the capacity of the collection is increased, leading to an expansion of the collection. A collection-specific threshold ratio *size/capacity* drives the collection expansion. The values 0.75 and 1.0 are commonly used thresholds (0.75 for collections operating with hashtags values and 1.0 for every other collection). These thresholds are used to trigger an expansion based on how much of the internal array is filled. Consider the class `OrderedCollection`, a frequently used expandable collection. Adding one element to an ordered collection of a given capacity c increases its size s by one. When $s = c$, then the collection is expanded to have a capacity of $2c$ elements.

Expanding a collection is a three-step operation summarized as follows:

1. *Creation of a larger new array* – the size of the collection having reached its capacity (i.e., the size of the internal data storage), a new array is created, typically twice as large as the original array.
2. *Copying the old array into the new one* – content of the old array is entirely copied into the first half of the new array.
3. *Using the new array as the collection's storage* – the expandable collection takes the new array as its internal storage, realized by simply making the storage variable point to the new array. The old array is garbage collected since it is not useful anymore.

Although efficient in many situations, expandable collections may result in wasted resources, as described below.

Expansion overhead. Expanding a collection involves creating and copying of possibly large internal array storage. Consider the following micro benchmark:

[3] http://www.esug.org/wiki/pier/Conferences/2014/IWST14.

[4] Intuitively, collection shrinking may have a significant impact on reducing resources. However, no collection offers this behavior.

```
c := OrderedCollection new.
[ 30000000 timesRepeat: [ c add: 42 ] ] timeToRun
  => 3375 milliseconds
```

This benchmark simply measures the time taken to add 30 million elements to an ordered collection. In our current execution setting, the micro benchmarks reported in this section have a variation of 7%.

The class OrderedCollection, when instantiated using the default constructor, as above, uses an initial capacity of 10 elements. An expansion of the collection occurs when adding the 11-th element. The capacity is then doubled. The size of the collection is 11 and its capacity is 20. When the 21st element is added to it, its capacity is 40.

Adding 30 million elements in a collection triggers $log_2(30\,000\,000 / 10) = 22$ expansions. Such expansions have heavy cost, both in terms of memory and CPU time. When the capacity is equal to or greater than the number of elements to be added:

```
c := OrderedCollection new: 30000000.
[ 30000000 timesRepeat: [ c add: 42 ] ] timeToRun  =>
  => 1356 milliseconds
```

in such a case, no expansion occurs, thus resulting in adding the elements without any expansion phases.

The case of LinkedList. In Pharo, all but one expandable collection use an array as internal storage. LinkedList uses linked elements instead. For this reason we have voluntarily excluded this class from our analysis. In addition, LinkedList is rarely used in Pharo applications (Section 4.1). LinkedList is used only once, in the Pharo thread scheduler. Since LinkedList is largely not used in Pharo, we could not define a reliable benchmark to measure our improvements.

In Java, all collection classes (with the exception of Linked−List, Tree, TreeSet, TreeMap, Queue) use an array as internal storage. These non-array-based collection classes are not free of problems. For example, a linked list used in place of an ArrayList may suffer from costly random accesses (e.g., LinkedList.get(int)). By not using an internal array, we leave these issues out of the scope of this article.

Copying of memory. At each expansion of the collection, the whole internal array content has to be copied into the newly created array. Consider the OrderedCollection in which 30 million elements are added to it. Since the collection is expanded 22 times, the internal array has been copied 21 times.

At the first expansion, when the internal storage grows from 10 to 20 slots, 10 slots are copied. Since each array slot is 4 bytes long, 40 bytes have been copied. In total, 80 bytes are copied for the second expansion. Since the internal array size increases exponentially, the number of bytes that are copied scale up easily. Adding 30 million elements produces 22 expansions, incurring $\sum_{i=0}^{21} 10 * 2^i = 41\text{M}$ slot copies. In total, $41 * 4 = 164$ Mb of memory that are copied between unnecessary arrays. Such copying could be reduced or avoided by giving a proper initial capacity to the collection.

Virtual memory. The memory of a virtual machine is divided into generations. Garbage collection happens by copying part of a generation into a clean generation. Such copying is likely to happen across memory pages [11], since the new array is likely to be in the young generation (i.e., part of the memory used for short lived objects and new object creations). In addition, the copying of arrays may activate part of the virtual memory stored on disk if the part of the memory containing the old array has been swapped to disk [11].

Collector pauses. Garbage collection copies and joins portions of memory to reduce memory fragmentation [7]. Copying and scanning a large portion of memory, such as collections, may cause large and unpredictable memory recollection pause times. The garbage collection pauses in proportion to array size [12].

Unnecessary slots. Expanding a collection doubles the size of the internal array representation. As a consequence, a collection having a size less than its capacity has unused slots.

For example, adding 30 million elements to a collection with the default initial capacity generates 22 expansions. After the 22nd expansion, the collection has a capacity of $10 * 2^{22} = 41{,}943{,}040$, large enough to contain the 30,000,000 elements. As a consequence, the collection has $41{,}943{,}040 - 30{,}000{,}000 = 11{,}943{,}040$ unused slots. Since each slot weighs 4 bytes, nearly 48 Mb of memory are unused after having added the 30M elements.

Note that the issue of having the unused portion of the array has already been mentioned (Pattern 1, 3, 4 in [13]). Our article reports the evolution of the amount of unused memory space against the improvement we have designed of the collection library. Our approach to address this issue is new and has not been considered before.

3. Benchmarking and metrics

To move away from micro-benchmarks and understand this phenomenon better in real applications, we pick a representative set of Pharo applications and profile their execution.

Table 1
Description of the benchmark (the #Ref column indicates the number of references to expandable collection in source code).

Index	Application	LOC	#Ref
1	AST	8,091	57
2	Nautilus	1,566	9
3	Petit	14,919	95
4	Regex	5,055	16
5	Roassal	19,844	133

3.1. Benchmark descriptions

We pick 5 open source software projects from the Pharo ecosystem stored on the Pharo forge[5]:

- *AST* is a framework to model, represent, and transform Abstract Syntax Trees of Pharo code. This framework makes intensive use of recursive trees. The benchmark we consider for AST is parsing a large amount of Pharo code.
- *Nautilus* is the Pharo standard system code browser: a programmer employs Nautilus to read and write Pharo code. The browser is a complex user-interface that contains many interconnected graphical widgets. The benchmark programmatically iterates over all the source code elements of Pharo (e.g., packages, classes, and methods). The browser is then subsequently updated at each selected entity.
- *Petit* is an advanced parsing framework using PEG (Parsing expression grammar) to build parsers by composing existing parsers. The benchmark builds a Pharo parser from existing parsers and parses the whole Pharo source code data base. In some sense, these benchmarks are similar to AST's, however the implementation is different (and thus, produce a different execution behavior).
- *Regex* is a regular expression framework. Regular expressions may be expressed and run over a textual content. The benchmarks we consider build parsers and run them over large textual contents.
- *Roassal* is a visualization engine that offers sophisticated visual and graphical representations of data structures. The benchmarks we consider for Roassal build large visualizations and programmatically simulate user interactions over it. Note that the first author of this article is involved in the development of Roassal.

These applications, listed in Table 1, have been selected for our study for three reasons:

- The benchmarks we consider for each application are representative of some application usage typically employed by Pharo developers;
- The applications are actively supported and represent relevant assets for the Pharo community. Therefore, our results are likely to raise interest from this community composed of several industries. These applications are used daily both in industries and academia;
- The community is friendly and interested in collaborating with researchers. As a result, developers are accessible and positive in answering our questions.

We employ the benchmark to approximate how expandable collections are used in general. The 5 applications we have picked are CPU intensive and the benchmarks are likely to reflect practical and representative execution scenarios. Each application comes with a set of benchmarks. We have arbitrarily picked 3 for each application. These benchmarks have been written by the authors of the considered application and represent a typical heavy usage of the application. In case that the application was shipped with less than three benchmarks, we kindly asked the authors to provide new additional benchmarks.

3.2. Metrics about the collection library

We propose a set of metrics to understand how expandable collections are used and what the amount is of resulting wasted resources. The metrics that we propose to characterize the use of expandable collections for a particular software execution are:

- *NC* – **N**umber of expandable **C**ollections – This metric corresponds to the number of expandable collections created during an execution. This metric is used to give relative numbers (i.e., percentages) for most of the metrics described below.
- *NNEC* – **N**umber of **N**on **E**mpty **C**ollections – Number of expandable collections that are not empty, even temporarily, during the execution.

[5] http://smalltalkhub.com.

- *NEC* – **N**umber of **E**mpty **C**ollections – Number of expandable collections to which no elements have been added during the execution. A collection for which elements have been added then removed are not counted by *NEC*.
- *NCE* – **N**umber of **C**ollection **E**xpansions – Number of collection expansions happening during the program execution.
- *NCB* – **N**umber of **C**opied **B**ytes due to expansions – Amount of memory space, in bytes, copied during the expansions of expandable collections.
- *NAC* – **N**umber of internal **A**rray **C**reations – Number of array objects created used as internal storage during the execution.
- *NOSM* – **N**umber of collections that are filled **O**nly in the **S**ame **M**ethods that have created the collections. A collection that is *both* created and filled within a method *m* is counted. A collection that is created in a particular method, and then passed to another in order to be filled is not counted.
- *NSM* – **N**umber of collections filled in the **S**ame **M**ethods that have created them. A collection that is created and filled in the same method *m* is counted, regardless if the collection escapes *m*.
- *NAB* – **N**umber of **A**llocated **B**ytes – Accumulated size of all the internal arrays created by a collection.
- *NUB* – **N**umber of **U**nused **B**ytes – Size of the unused portion of the internal array storage. For a given collection, this metric corresponds to the difference $capacity - size$.

These methods will be employed to characterize expandable collections from a perspective of unused allocated resources. To our knowledge, these methods are new and have not been proposed by any other research effort.

3.3. Computing the metrics

Measuring these metrics involves a dynamic analysis to obtain an execution blueprint for each collection. We have instrumented the set of expandable collections in Pharo to measure these metrics.

We measure only the collections that are directly created by an application. Computation carried out by the runtime is not counted. If we counted collections created by the runtime and the application, a residual amount would have to be determined since the same collections may be counted several times across different applications.

Collections are often converted thanks to some utility methods. For example, an ordered collection may be converted as a set by sending the message `asSet` to it. Converting an expandable collection into another expandable collection are considered in our measurements.

Our measurements, used to characterize the use of expanded collections and measure wasted resources associated with them, have to be based on representative application executions, close to what programmers are experiencing. Unfortunately, Pharo does not offer a standard benchmark for measuring performance in the same spirit as DaCapo [14] and SpecJVM. We have designed our benchmarks from performance scenarios of program executions.

The tables given at the end of the article show the results of our measurements. Table 7 gives the measurement of our benchmarks using the standard collection library of Pharo. This table is used as the baseline for our improvements of the library.

Minimizing measurement bias. Carefully considering measurement bias is important since an incorrect setup can easily lead to a performance analysis that yields incorrect conclusions. Despite numerous available methodologies, it is known that avoiding measurement bias is difficult [15–17]. An effective approach to minimize measurement bias is called *experimental setup randomization* [16], which consists in generating a large number of experimental settings by varying some parameters, each considered parameter being a potential source of measurement variation. Our measurements are programmatically triggered, meaning that multiple runs of our benchmark are easily automatized. We have considered the following parameters:

- *Hardware and OS* – We have used two hardwares running with OSX ((a) a MacBook Air, 1.3 GHz Intel Core I5, 4 Gb 1333 MHz DDR3, with a solid hard disk running OS X 10.10.2 and (b) iMac, Quad-core Intel Core i5, 8 Gb, running OS X 10.9). We avoided any user-defined processes beside Pharo and our measurement tool. The default processes launched by the operating systems are allowed to reflect a real and unmodified setting.
- *Heap size* – We run our experiments using different initial sizes of the heap (100 Mb, 500 Mb, 1000 Mb).
- *Repeated run* – For each execution of the complete benchmark, we averaged 5 runs, with a random pause between each run.
- *Randomized order* – The individual performance benchmarks are randomized at each complete benchmark run.
- *Reset caches* – Method caches located in the VM are emptied before each run.
- *GC* – The garbage collector has been activated several times before running each benchmark.

In total, we have considered 9 different experimental setups. We did not notice any significant variation between these experimental setups.

The measurements given in Appendix B are the result of an average of 9 different executions, each considering a different combination of the parameters given above.

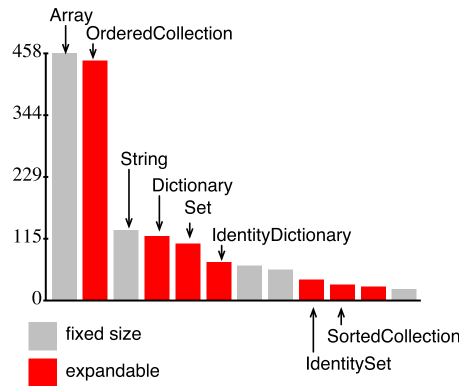

Fig. 1. References of collection classes in source code.

4. Use of expandable collections in Pharo applications

This section analyzes the use of expandable collections in Pharo applications. The results given in this section answer the research question *A*.

4.1. Use of expandable collections in source code

Not all the collection classes share the same popularity among programmers. Fig. 1 gives a distribution of class references in the Pharo applications we have studied. The histogram indicates the frequency of the most popular expandable and non-expandable collections in the benchmark.

The class `Array` represents fixed size collections and is the most popular collection class. There are 458 references of the class `Array` found in the 17 applications. This number takes into account explicit instantiations of the class. Use of immediate values (a.k.a. "literals" in the Pharo jargon) are not considered. For example, the expression `v := #(1 2 3)`, which assigns an array to the variable `v`, is not considered as referencing the class `Array`.

The class `OrderedCollection` is the second most popular collection. This class is expandable and gives a large range of facilities to insert, remove, sort and filter elements. The remaining expandable collections are used less frequently.

All expandable collections but one use an array as internal storage. `LinkedList` is the only expandable collection that does not use an array, but instead, a linked list of elements. `LinkedList` is rarely used however. As far as we are aware, `LinkedList` is solely used in the thread scheduler to model the roundtrip when scheduling. No occurrence of `LinkedList` has been found in the applications we have studied. We can therefore safely exclude `LinkedList` from our experiment.

4.2. Dynamic analysis

We have run our benchmarks and profiled their executions. The metrics given in Section 3.2 have been computed and reported in Table 7 for each of the applications' executions. The execution of the 15 performance benchmarks create 6,129,207 expandable collections. We use the pattern *bXY* to label each benchmark, where *X* represents an acronym of the full application name, and *Y* a digit ranging from 1 to 3.

Naturally, very few of these expandable collections live through the whole execution since the garbage collector regularly cleans the memory by removing unreferenced collections. In our measurements, we do not consider the action of the garbage collector on the collection themselves since garbage collection is orthogonal to the research questions that we focus on.

The number of created collections indicates large disparities between the analyzed applications. Benchmarks bReg1 and bReg2 involve a long and complex execution over a significant amount of data, indicated by the large number of created expandable collections. Benchmarks bN1, bN2, bN3 create a small number of collections.

Variation in the measurements. Two executions of the same code may not necessarily create the same number of collections, even if no input/output or random number generation is involved. Measurements vary little over multiple runs of the benchmarks. Values reported in the tables in the appendix have been obtained after multiple runs and have an average variation of 0.0095%. Although the applications we have selected for our case study do not make use of random number generation, the use of hash values can make non-deterministic behavior. A hash value is given by the virtual machine when the object is created. In the case of Pharo, such a hash value depends on an internal counter of the virtual machine. Consider the following code:

```
d := Dictionary new.
d at: key1 put: OrderedCollection new.
d at: key2 ifAbsentPut: [ OrderedCollection new ]
```

The class Dictionary uses the equality relation and hash values between keys to insert pairs. If we have the relation key1 = key2 and key1 hash = key2 hash, then the dictionary considers that the two keys are actually the same and we have only one instance of OrderedCollection. However, in case that the hash is not overridden but = is overridden, the relation key1 hash = key2 hash may be true only sporadically, thus triggering a non-deterministic behavior over multiple executions.[6]

Empty collections. Table 7 indicates a surprisingly high proportion of empty collections in our benchmarks. From over 6.1 million expandable collections created by our benchmarks, 4.4 million (73%) were created without having any element added to them. Only 26% of collections have at least one element added to them during their lifetime.

To understand this phenomenon better, we will take a closer look at the data we obtained. The number of empty collections created by our benchmark varies significantly across applications. Benchmark bReg2 creates a total of 2.1M of expandable collections, for which only 0.4M are non-empty. This application is a regular expression engine that applies pattern matching. The engine is complex due to the underlying optimized logic engine.

Cause of empty collections. We manually have inspected the applications and benchmarks that generate a high proportion of empty collections. A large proportion of the created empty collections is caused by the object initialization specified in the constructors. Consider the constructor of the class RBVariableEnvironment:

```
RBVariableEnvironment >> initialize
  super initialize.
  instanceVariables := Dictionary new.
  classVariables := Dictionary new.
  instanceVariableReaders := Dictionary new.
  instanceVariableWriters := Dictionary new
```

This constructor implies that each instance of RBVariableEnvironment comes with at least four instances of dictionaries. Most instances of RBVariableEnvironment actually have their dictionaries empty. This is not an isolated case. The 5 applications under study are composed of 1,713 classes. We have 375 of these 1,713 classes that explicitly define at least one constructor. We have also found that 144 of these 375 classes explicitly instantiate at least one expandable collection when being instantiated. Expandable collections created in the constructor is a prominent cause of unused collections.

Number of array creations. The standard collection library creates a new array at each collection expansion. Since instantiating a collection results in creating a new array, the number of created arrays (*NAC*) subtracted from the number of expansions (*NCE*) is equal to the number of collections (*NC*). We have roughly the following relation $NAC - NCE = NC$ in Table 7. Some differences may be noted due to rehashing operations on hash-based collections (e.g., HashSet, Dictionary) that may be triggered by an application. Such effects are marginal and have a little impact on the overall measurements, which is why we do not investigate such minor variations further.

Collection expansions. From the 6.1 million of collections (*NC* column in Table 7), 1,637,669 (26%) of the collections are expanded 980,792 times during the execution of the benchmark (*NCE* column). These expansions result in over 46.9 Mb of copies between these arrays (*NCB* column).

Unused memory. Summing up the memory consumed by all the internal arrays yields over 253 Mb (*NAB* column). More than 228 Mb of these 253 Mb are actually unused (*NUB* column) as a result of having expandable collections filled only a little on average (i.e., the size of the collection being much below its capacity).

4.3. Reducing the overhead incurred by collection expansions

The measurements given in the previous section reveal that the use of expandable collections may result in wasted CPU and memory consumption. We use the observations made above to reduce the overhead caused by expansions. We propose three heuristics to reduce the overhead incurred by expandable collections:

Creating the internal array storage on demand. Creating an internal array only when necessary, i.e., upon the first element addition. Since 76% of arrays are empty, lazily instantiating the internal array will be beneficial.

Reusing arrays when expanding. Expanding a collection involves creating an array larger than the previous one (usually twice the initial size). After copying, the original array is discarded by removing all references to it. The task to free the memory is then left to the garbage collector.

Instead of letting the garbage collector discard old arrays, arrays can be recycled: a collection expansion frees an array, which itself may be used when another collection expands.

Setting an initial capacity. About 10% of expandable collections are created and filled in the same method. These 10% of the collections have been created by 276 methods across our benchmark. There are 105 of these 276 methods that use the default construction with the default initial capacity.

Some of these methods may be refactored to create expandable collections with an adequate initial capacity.

[6] Redefining = without redefining hash is a classic defect in software programs and it is widely recognized as such. Unfortunately, this defect is frequent and inferring whether this defect is the cause of non-determinism is not trivial.

We have designed the *OptimizedCollection* library, a collection library for Pharo that exhibits better resource management than the standard set of collection classes. OptimizedCollection implements the design points made above. Section 5, Section 6 and Section 7 elaborate on each of these points.

5. Lazy internal array creation

For the programming languages we have studied, expandable collections have been implemented under the assumption that a collection will be filled with elements. This assumption unfortunately does not hold for the usage scenarios we are facing in our benchmark. Less than a third of the expandable collections are filled in practice. This suggests that creating the internal array only when elements are added is likely to be beneficial. We call this mechanism *lazy internal array creation*.

As far as we are aware of, lazy internal array creation for expandable collections has not been reported in the academic literature or in engineering notes. Lazy initialization is a well known technique to allocate memory only when necessary. Surprisingly, using lazy initialization to optimize expandable collections has been little considered (except for a very few exceptions in the Java and C# collections) despite the significant memory overhead collections may generate [5].[7]

This section first describes the design points of lazy internal array creation and reports measurements on our benchmarks.

5.1. Creating the array only when necessary

Introducing a lazy creation of the internal array is relatively easy to implement. Instead of creating the internal storage in the constructor, we defer its creation when adding an element to the collection. For this, we need to remember the capacity for the future creation of the array. Methods that add elements to the collection have to be updated accordingly.

This simple-to-implement improvement leads to a significant reduction in memory consumption. Using the default capacity, an empty ordered collection now occupies 20 bytes only (in comparison to the 64 bytes without supporting lazy internal array creation). After adding an element to the collection, the internal array is created, thus increasing the size of the collection to 64 bytes.

We have implemented the lazy internal array creation as described above in all the expandable collection classes. The following section describes the impact on our case studies.

5.2. Lazy creation on the benchmark

Table 8 gives the metric values of our benchmark when using the lazy internal array creation. Contrasting Table 7 (using the standard collection library, i.e., without lazy internal array creation) with Table 8 (lazy creation) shows a significant reduction of unused memory and number of created internal arrays. More specifically, we have:

- The number of array creation (*NAC*) has been significantly reduced as one would expect. It went from 6,205,920 to 1,874,940, representing a reduction of $(6{,}205{,}920 - 1{,}874{,}940)/6{,}205{,}920 = 69.78\%$ of array creation.
- The number of unused bytes (*NUB*) has also been significantly reduced from 228 Mb to 61 Mb, representing a reduction of $(228{,}171{,}448 - 61{,}393{,}008)/228{,}171{,}448 = 73.09\%$.

The lazy internal array creation has a slight positive impact on the execution time of the benchmark. By lazily creating the internal arrays, the execution time of all runs has been reduced by 2.38%.

6. Recycling internal arrays

A collection expansion is carried out with three sequential steps (Section 2): (i) creation of a larger array; (ii) copying the old array into the new one; (iii) replacing the collection's storage with the new array. The third step releases the unique reference of the array storage, entitling the array to be disposed by the garbage collector. This section is about recycling unused internal arrays and measures the benefits of recycling.

The general mechanism of recycling arrays along a program execution is not new. It has already been shown that for functional programming avoiding unnecessary array creation by recycling those arrays is beneficial [18]. Recycling arrays in a context of expandable collections is new and, as far as we are aware of, it has not been investigated.

6.1. Recycling arrays on the benchmark

Principle. Instead of releasing the unique reference of an array, the array is recycled by keeping it within a globally accessible pool. The array disposed after a collection expansion is inserted in the pool. The first step of expansion has now to check for a suitable array from the pool. If a suitable array is found, the array is removed from the pool and used as

[7] Note that this work by Gil et al. uses an older version of Java. Version 7 and 8 have learned from them and improved their collection.

Table 2
Effect of the different strategies for the unit test benchmarks (best performance is indicated in **bold**).

Metrics	*S1*	*S2*	*S3*
NC	6,127,788	6,127,788	6,127,788
NCE	980,792	**977,904**	980,805
NCB	46,953,084	**45,314,124**	47,140,001
NAC	**1,874,940**	1,875,235	1,876,520
NAB	86,510,132	90,120,012	**86,451,502**
NUB	**61,393,008**	70,825,135	61,851,892
#full GC	80	88	80
#incr GC	28,884	**28,846**	40,628

internal array storage in the expanded collection. If no array from the pool can be used as internal array storage for a particular collection expansion, a new array is created following the standard behavior.

When an array is inserted into the pool, the array has to be emptied so as to not keep unwanted references. Emptying an array is done by filling it with the nil value.

Need for different strategies. Consider the following example:

```
c1 := OrderedCollection new.
50 timesRepeat: [ c1 add: 42 ].
c2 := OrderedCollection new.
c3 := OrderedCollection new.
```

Filling c1 with 50 elements triggers three expansions, which increases the capacity from 10 to 20, from 20 to 40 and from 40 to 80. Having c1 of a capacity of 80 is sufficient to contain the 50 elements. The creation of the collection and these expansions has created and released three arrays sized 10, 20, 40, respectively. These arrays are inserted in a pool of arrays.

When c2 is created, an array of size 10 is needed for its internal array storage. The pool of arrays contains an array of size 10 (obtained from the expansion of c1). This array is therefore removed from the pool and used for the creation of c2.

Similarly, c3 requires an array of size 10. The pool contains two arrays, of size 20 and size 40. The creation of the ordered collection faces the following choice: either we instantiate a new array of size 10, or we use one of the two available arrays.

This simple example illustrates the possibility of having different strategies for picking an array from the pool. We propose three strategies and evaluate their impact over the benchmark:

S1: $requiredSize = size$ – Pick an array from the pool of exactly the same size that is requested
S2: $requiredSize <= size$ – Pick the first array with a size equal to or greater than what is requested
S3: $size * 0.9 < requiredSize < size * 1.1$ – Pick an array of a size within a range of what is requested.

The effect of the different strategies on the benchmarks is summarized in Table 2. We consider 8 metrics: *NC* (number of created expandable collections), *NCE* (number of collection expansions), *NCB* (number of copied bytes), *NAC* (number of internal array creations), *NAB* (number of allocated bytes), *NUB* (number of unused bytes), the number of full garbage collections and the number of incremental garbage collections.

S1 generates less unused array portions (*NUB*) than *S2* and *S3*. *S2* incurs less collection expansions than *S1* and *S3*, which also result in fewer copied bytes (*NCB*). Oddly, the number of incremental garbage collections is higher with *S3*. The results given in Table 9 use Strategy *S1* since this strategy is more effective than the two other regarding the number of unused bytes (*NUB* metric).

Effect on the benchmark. When supporting the lazy internal array creation without recycling arrays (Table 8), the number of unused bytes (*NUB* column) has increased by (61,420,484 − 61,393,008)/61,393,008 = 0.04%. The reduction of the number of created arrays (*NAC* column) is (1,798,578 − 1,874,940)/1,874,940 = 4%. In all, 35,063 collections have been recycled. More interestingly, the technique of reusing arrays has reduced the number of allocated bytes by 9.4% (column *NAB*: (86,510,132 − 78,373,588)/86,510,132 = 9.4%).

After profiling the benchmark, the number of collections left over in the pool is rather marginal. Only 216 collections are in the pool, totaling less than 89 kB.

Using the pool of arrays incurs a relatively small execution time penalty. This represents an increase of 5.8% of execution time when compared with the lazy array creation and an increase of 2.8% with the original library.

Recycled arrays. The techniques described in this section recycle arrays of different sizes. Fig. 2 shows the distribution of size of recycled arrays for Strategy *S*1. The vertical axis indicates the number of recycled arrays. The horizontal axis lists the size of arrays that are effectively recycled.

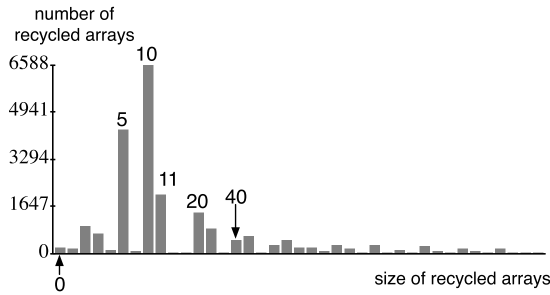

Fig. 2. Distribution of recycled arrays.

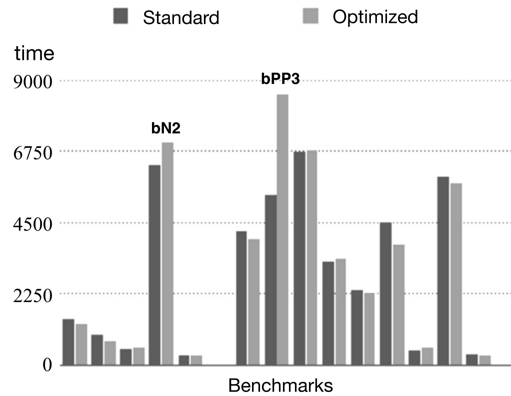

Fig. 3. Impact of execution time of the optimized collection library.

Arrays that are the most recycled have a size of 5 and 10. The standard Pharo library is designed as follows: 5 corresponds to the minimum capacity of hash-based collections, and 10 is the default size of non-hashed collections.[8] The value 20 corresponds to the size of the internal array of default collection after expansion. An array of size 40 is obtained after a second expansion.

Multi-threading. The pool of recycled internal arrays is globally accessible. Access to the pool needs to be adequately guarded by monitors to avoid concurrent addition or removal from the pool. Several of the applications included in our benchmark are multi-threaded. Previous work on pooling reusable collections [19] shows satisfactory performance in a multi-threaded setting.

6.2. Variation in time execution

If we consider the global figures, recycling arrays has a penalty of 3% of execution time in the average. However, if we have a close look at each individual benchmark, we see that most of the performance variation indicates that our optimized collection library performs slightly faster than the standard collection library (in addition to significantly reduce the memory consumption, as detailed in the previous sections).

Fig. 3 shows the variation of execution time of the performance benchmarks between the standard collection library and our optimized library. All but two benchmarks are slightly faster with our library. The execution of benchmarks *bN2* takes 6,738 seconds with the standard collection library and takes 6,789 with our library. Since this represents a variation of $(6{,}789 - 6{,}738)/6{,}738 = 0.7\%$, we consider this variation as insignificant.

Benchmark *bPP3* goes from 6,330 seconds with the standard library to 7,010 with our optimized library, which represents an increase of 9.7%. The reason for this drop in performance is not completely clear to us. This benchmark parses a massive amount of textual data. Private discussion with the authors of the considered application revealed the cause of this variation may be due to the heavy use of short methods on streams. Traditional sampling profiler does not identify the cause of the performance drop, indicating it stems from particularities of the virtual machine (for which its execution is not captured by the standard Pharo profiler). These short methods have an execution time close to the elementary operations performed by the virtual machine to lookup the message in method cache. Although we carefully designed our execution by emptying

[8] Note that we are not arguing whether 5 and 10 are the right default size. Other languages including Scala and Ruby use a different default capacity size. We are simply considering what the Pharo collection library offers to us.

different caches and multiply activating the garbage collection between each execution, the reason of the performance drop may be related to some particularities of the cache in the virtual machine.

It turns out that our optimized library performs better except for two benchmarks. By excluding the benchmark *bPP3*, our library performs 3.01% faster than with the standard collection library.

7. Setting initial capacities

A complementary approach to improving the collection library is to find optimization opportunities in the base application (which makes use of the collection library).

Example. We have noticed recurrent situations for which an expandable collection is filled in the same method that creates the collection. The following method, extracted from a case study, illustrates this:

```
ROView>>elementsToRender
  "Return the number of elements that will be rendered"
  | answer |
  answer := OrderedCollection new.
  self elementsToRenderDo: [ :el | answer add: el ].
  ^ answer
```

The method elementsToRender creates an instance of the class `OrderedCollection` and stores it in a temporary variable called answer. This collection is then filled by iterating over a set of elements.

The method elementsToRender uses the default constructor of the class `OrderedCollection`, which means a default capacity to the collection is given. As described in the previous sections, such a method is a possible source of wasted memory since a view may contain a high number of elements, thus recreating the situation we have seen with the micro-benchmark in Section 2.

By inspecting the definition of the method `elementsToRenderDo:`, we have noticed that the number of elements to render is known at that stage of the execution. The method may be rewritten as:

```
ROView>>elementsToRender
  "Return the number of elements that will be rendered"
  | answer |
  answer := OrderedCollection new: (self elements size).
  self elementsToRenderDo: [ :el | answer add: el ].
  ^ answer
```

This new version of elementsToRender initializes the ordered collection with an adequate capacity, meaning that no resource will be wasted due to the addition of elements in the collection referenced by answer.

Profiling. The metrics *NOSM* and *NSM* identify methods that create a collection and fill it. The instance of OrderedCollection created by the method elementsToRender is counted by *NSM* since the collection is created and filled in this method. The collection is also counted by *NOSM* in the case that no other methods add or remove elements from the result of elementsToRender.

We see that about 8% of the expandable collections are immediately filled after their creation. We also notice that slightly fewer collections are only filled in the same method in which they were created. We focus on these collections since they are likely easy to refactor without requiring a deep knowledge about the application internals.

The *NOSM* and *NSM* metrics are computed by instrumenting all the constructors of expandable collection classes and all the methods that add and remove elements.

Refactoring methods. The 670,064 collections (*NOSM* column) that are filled solely in the methods that have created them have been produced by exactly 276 methods. We have manually reviewed each of these methods. We have refactored 105 of the 276 methods to insert a proper initialization of the expandable collection. The remaining 171 methods were not obvious to improve. Since we did not author these applications and had a relatively low knowledge about the internals of the analyzed applications, we took a conservative approach: we have refactored only simple and trivial cases for which we had no doubt about the initial capacity, as in the example of elementsToRender given above. We use unit-test to make sure we did not break any invariant captured by the tests.

Impact on the benchmark. Table 10 details the profiling for the benchmark by lazily creating internal arrays, reusing these arrays and refactoring the applications. By comparing from Table 9 to Table 10, the reduction gain for the number of allocated bytes is 0.05% (column *NAB*, which goes from 78.37 Mb to 78.33 Mb). The amount of unused space was reduced by 0.06% (column *NUB*, which goes from 61.42 Mb down to 61.38 Mb). No variation in terms of execution time was found.

Setting the capacity. We ran the *modified version of our benchmark* with the *original collection library*, without the recycling and the lazy array creation. Gains are marginal. Only a reduction of 0.05% of the number of allocated bytes was measured. We conclude that the obtained gain by allocating a proper initial capacity is marginal.

8. Reducing resource consumption with Lua's table

As an application of our finding presented above, we investigate the use of another data structure to reduce the consumption of resources (both memory and CPU).

8.1. In a nutshell

Lua[9] is a highly popular language: it is within the top languages present on github[10] and is intensively used in the gaming industry. In Lua, a table is the unification of what is commonly perceived as a hashtable (e.g., Hashtable in Java and Dictionary in Pharo) and a sequential ordered collection (e.g., ArrayList in Java and OrderedCollection in Pharo).

A table is the unique abstract data-type provided by Lua to manipulate collections. Since Lua 5.0,[11] released in 2005, a table is a hybrid data structure, composed of a hash part and an array part. Despite the fact that Lua has a deserved reputation for performance, tables have not been carefully studied by the research community as far as we are aware.

Our experiments show that when replacing instances of the standard Pharo classes `OrderedCollection` and `Dictionary` by our implementation of Lua's tables, the memory allocation due to collections is decreased by up to 19% (the variation from our optimized version over the non-optimized ranges from 0.8% to −19.3%) when executing long-running benchmarks that make extensive use of collections. It also has an impact on execution time, which is overall decreased, with a maximal reduction of 15% (the variation ranges from 4.2% to −15.7%).

The following two subsequent sections make the following contributions:

- *Description of the Lua collection* – The internal representation of Lua's table is carefully presented and discussed. Currently, Lua's table is only superficially described in various textbooks about Lua (Section 8.2).
- *Comparing table against* `Dictionary` *and* `OrderedCollection` – After having implemented tables in Pharo, we carried out a number of measurements to assess the various aspects of our table implementation and compare its performances with two highly popular collections in Pharo (Section 9).

8.2. Lua's table

To handle collections of values Lua offers only one data-structure, called *table*. To cope with all the typical usages of collection [10] (e.g., filtering, sorting, transforming), a large set of functions are provided. A table is a particular data type that has two facets: a table is an efficient combination of both an *associative array* and an *expandable sequential collection*.

Two usages. Being an associative array refers to the fact that a table may be composed of a collection of *(key, value)* pairs, as any implementation of hashtable. For example, the expression `{ x = 10, y = 20 }` creates a table having two keys, `x` and `y`, bound to the values `10` and `20`, respectively.

A table is also an expandable sequential collection, referring to the fact that it may dynamically grow when data is added to it and shrink when data is removed. For example, the expression `{5, 6, 7}` creates a table with three sequential elements. Lua offers built-in functions to manipulate a table and dedicated iterators to traverse a table. The functions `insert` and `remove` add and remove elements for a table.

Internal structure. The internal structure of a table accommodates this radically two different usages. A table is implemented as a hybrid data structure: it contains a hash part and an array part. A table automatically and dynamically adapts its two parts according to their contents: the array part tries to store values corresponding to integer keys from 1 to some limit n. Values corresponding to non-integer keys or to integer keys outside the array range (i.e., $> n$) are stored in the hash part. Consider the table `a = {10, 20, x=40}`. Three keys are used, `1`, `2`, and `x`. The two first keys are stored in the array part, which means that the expression `a[4] = 1` stores the value `1` at the index 4 of the array. On this example, we have $n = 4$. The expression `a[1000] = 1` inserts the value `1` for the key `1000` in the hash part since the key is outside the allowed range.

Grow. During an `insert()` operation or a direct assignation as in `a[4] = 1`, a table grows if it cannot accommodate a key which is being assigned to a value. Either the hash part or the array part may grow. For example, consider the `a` table described previously. The expression `a[8] = 1` first tries insert the binding in the array part. The insertion in the array part fails since the array has a size of $n = 4$ and $8 > 4$. The table insertion algorithm then tries to insert the binding in the hash part. If the key cannot be accommodated in the hash part without growing then the grow algorithm is triggered. This algorithm considers all integer keys in both the array part and the hash part. It computes the maximal size n, which is the greatest power of `2` such that (i) at least 50% of the slots between 1 and the candidate n are used and (ii) there is at least one slot used between $\frac{n}{2} + 1$ and n. The algorithm assigns a size n to the array part and inserts in it all integer keys that are less or equal to n that were previously in the hash part. Subsequently, the hash part is then rehashed.

For example, consider the table produced by the Lua code `a = {10, 20, 4 = 30, x = 25}`. This table has a hash part holding the key `x` associated to the value `25` and an array part with capacity 4 but with the third slot unused (see Fig. 4). When applying the operation `insert(a, 40)` the array part grows to accommodate the additional value 40, and the new size for it will be twice the number of used slots (largest n such that at least half the slots between 1 and n are in use).

As a consequence, if a table is being used as an array, it performs as an array, as long as its integer keys are dense, i.e., at least 50% of the array slots are used. If keys are consecutively ordered, this is satisfied. In case only the array part is used,

[9] http://www.lua.org.

[10] http://githut.info.

[11] Lua is now at version 5.3.4.

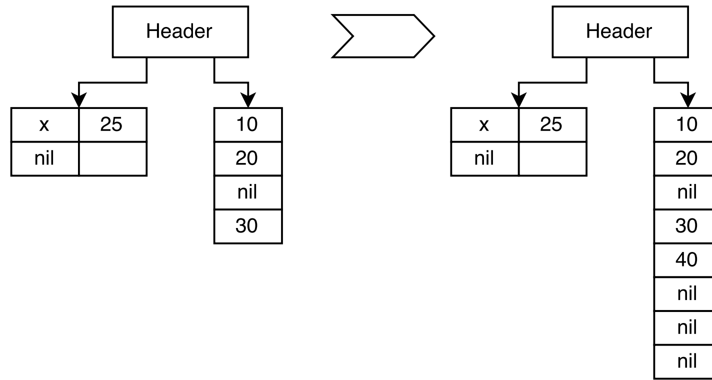

Fig. 4. Lua table `a = {10, 20, 4 = 30, x = 25}` before and after the operation `insert(a, 40)`.

the hash part does not incur any memory penalty since it is not even created. Conversely, if the table is being used as an associative array, and not as an array, then the array part is empty and will not incur any penalty.

9. Evaluating Lua's tables

9.1. Methodology

To evaluate the performance of Lua's tables to be used as a data structure to be employed in Pharo, we use the following 5-steps methodology:

S1- Determine frequently used expandable collections in Pharo. These collections will then be evaluated against Lua's table.
S2- Determine commonly employed features of the Pharo collections identified in Step S1.
S3- Implement Lua's table data structure in Pharo. Features offered by the table have to be accessible with the same name as the methods frequently used identified in Step S2.
S4- Performing micro-benchmarks to measure the execution time and memory cost of the features identified in S2 and individually measure the cost of these features.
S5- Performing macro-benchmarks using some representative Pharo applications. These applications will be automatically adapted to uses tables instead of the popular expandable collections identified in S1. Thanks to the polymorphism exhibited by our implementation of Lua's table data structure, references to collections can be automatically replaced with our new table while preserving the overall application invariant.

Ultimately, results obtained from Step S5 will position Lua's table against Pharo collections.

9.2. Step S1: identifying popular expandable collection in Pharo

We picked 124,292 methods found in a large Pharo source code base. Among these methods, 11,995 methods (9.64%) are using a collection. From these methods, we found:

- 2,281 methods (19%) are referencing the class `OrderedCollection`,
- 2,067 methods (17%) are referencing the class `String`,
- 1,967 methods (16%) are referencing the class `Array`,
- 774 methods (6%) are referencing the class `Dictionary`.

The remaining 4,906 methods are referencing at least one of the other 150 collection classes. Since the classes Array and String are non expandable collections, they cannot be replaced by Lua's table. As a result, `OrderedCollection` and `Dictionary` are the two most popular expandable collections in Pharo. Our measurements will therefore compare Lua table against these two Pharo collection classes.

9.3. Step S2: determine commonly employed features

The classes `OrderedCollection` and `Dictionary` define 73 methods and 82 methods, respectively. Some of these methods are private, or rarely used. We have profiled the usage of these two classes during the execution of 5 large Pharo applications. We have identified 7 non-private methods that are the most frequently used. We excluded private methods since they are not meant to be invoked by end-user applications.

Table 3
Micro-benchmark results (1000 iterations) for SLua and `OrderedCollection` (OC). Times are in milliseconds. Δ% is $100 * (t_{SLua} - t_{OC})/t_{OC}$.

Benchmark	OC	SLua	Δ%
addFirst:	30.56 ± 0.11	32.38 ± 0.17	5.9 ± 0.7
addLast:	34.5 ± 0.13	41.42 ± 0.14	20 ± 0.6
do:	9.2 ± 0.084	9.34 ± 0.088	1.5 ± 1.3
reduce:	199.35 ± 0.11	237.76 ± 0.1	19.3 ± 0.1
remove:	130.9 ± 0.26	105.86 ± 0.25	−19.2 ± 0.2

Table 4
Micro-benchmark results ($n = 1000$) for SLua and `Dictionary` (Dic). Times are in milliseconds. Δ% is $100 * (t_{SLua} - t_{Dic})/t_{Dic}$.

Benchmark	Dic	SLua	Δ%
at:ifAbsent:	21.83 ± 0.27	22.47 ± 0.24	2.9 ± 1.7
at:put:	30.92 ± 0.14	32.63 ± 0.28	5.5 ± 1

In the class `OrderedCollection`, the most frequently employed methods are `addFirst:` (add an element at the beginning of the collection), `addLast:` (add an element at the end), `do:` (iterate over the collection), `reduce:` (iterate and fold the collection), and `remove:` (remove a particular element). In the class `Dictionary`, the method `at:ifAbsent:` (get a value from a key and use a default value in case the key is not found) and `at:put:` (insert a value for the given key).

Since these methods are frequently used, our micro-benchmarks measure the cost of each of these.

9.4. Step S3: the SLua implementation

We then implemented Lua's table in Pharo. Our implementation, which we refer to as SLua, is based on the Lua documentation, source code of the table implementation in Lua, and the source code of the Lua virtual machine. SLua slightly differs from the Lua's table implementation for a number of reasons:

- Lua is a procedural language and Pharo is a class-based object-oriented language. Lua offers numerous functions to manipulate tables. These functions had to be implemented as methods in SLua.
- Services offered by SLua use the method names we have identified in Step S2. This will make the class `SLua` polymorphic to the class `OrderedCollection` and `Dictionary`. Doing so will ease the implementation of the macro-benchmarks, in Step S5.

Despite these two differences, we are confident that our implementation is semantically close to the original implementation of tables. While Lua's table is a structure that is manipulated by Lua's built-functions, `SLua` is a class whose instances are objects that are mutated by sending messages to them.

Methods defined on `SLua` that correspond to the methods of `OrderedCollection` will essentially affect the array part while methods that correspond to `Dictionary` will affect the hash part. Some essential methods, such as `at:put:`[12] may access both the array and the hash parts, as described in Section 8.2.

9.5. Step S4: micro-benchmarks

Micro-benchmarks allow us to compare each of SLua's methods with the original implementation offered by the classes `OrderedCollection` and `Dictionary`. The benchmarks have been defined using the SMark framework [20].

Table 3 compares the execution of individual methods in SLua and in plain Pharo. The first column lists the name of the micro-benchmarks. The second column gives the measurement of the micro-benchmarks for the class `OrderedCollection`. The third column gives the performance for SLua. The last column gives the performance variation using the Pharo class as base. A positive variation indicates an overhead for SLua, while a negative variation indicates a performance improvement of SLua compared with the Pharo class.

We have considered only the most used methods, as identified in Step S2. Performance exhibited by SLua is lower than the same methods than `OrderedCollection`. This result is not surprising since the algorithms related to element insertion are more complex in Lua than in `OrderedCollection`. One notable exception is `remove:`, which is faster in SLua than in Pharo classes. At this stage, we are still unsure the reasons of the difference. We will investigate it in the future.

Table 4 compares methods that are directly involving the hash part of SLua compared with `Dictionary`. SLua is slightly slower than the standard Pharo dictionary when inserting and retrieving bindings. Again, this is not surprising due to the complexity of SLua's growing algorithm.

[12] This method is equivalent to the Lua syntactic code `a[4] = 1`, written in Pharo `a at: 4 put: 1`.

Table 5
Execution time of benchmarks with normal Pharo collections (NC), lazy Pharo collections (LC) and SLua. NC is represented in seconds while LC and SLua are shown as a percentage of variation in relation to NC.

Benchmark	NC (s)	LC (%)	SLua (%)
CircularTreeMap	58.26 ± 5.08	−8.3 ± 9.1	−15.7 ± 7.4
ForceBasedLayout	73.02 ± 2.07	0.9 ± 3.7	−0.9 ± 5.4
Mondrian	29.2 ± 0.18	−7.1 ± 1	−11.8 ± 0.7
NameCloud	2.31 ± 0.09	1.7 ± 5.6	4.2 ± 30
Spectrograph	15.81 ± 0.13	2.2 ± 1.1	2.5 ± 1.2

9.6. Step S5: macro-benchmarks

We conducted a number of macro-benchmarks to measure SLua's performance in representative application executions. We compare the performance along two complementary aspects: execution time and memory consumption.

Benchmark executions. We consider five benchmarks that are data intensive. Each of these benchmarks produces and manipulates large data sets of a couple millions of entries. Carrying out significant benchmarks is know to be challenging [16]. We have taken care to (i) clean the memory by multiply triggering the garbage collector to clean memory, (ii) warming the just-in-time compiler (JIT) and caches with some a few initial benchmark executions, and (iii) multiply executing each benchmark.

Comparing SLua and standard collections. The five applications we selected designed using the standard Pharo collection, without using SLua before. Each benchmark is run twice, a first time using the standard Pharo collections and a second time using SLua. Since we have designed the class `SLua` to be polymorphic to `OrderedCollection` and `Dictionary`, we replaced all references of these two classes by `SLua`. As a consequence tables are produced instead of Pharo collections by the application when the benchmark is run.

The Pharo runtime has been left untouched: this means if an application *A* invokes an API offered by a standard Pharo library (e.g., socket), then the API may produce a standard Pharo collection. Having this behavior is critical for our experience since we do not wish to measure the effect of SLua on the standard Pharo libraries. The class replacement has been carried out using the Spy profiling framework [21].

The original Lua implementation, and therefore SLua, uses lazy internal array creation and initialization. It is known that lazy creation reduces the overhead when instantiating `OrderedCollection` and `Dictionary`. Comparing SLua (which uses lazy internal array creation) against the standard Pharo collections (which do not use a lazy internal array creation) will introduce a significant bias. We therefore added a version of the `OrderedCollection` and `Dictionary` that have a lazy internal array creation.

Results for execution time. Table 5 gives the result of the benchmark executions. Each benchmark is executed 10 times. The first column lists the macro-benchmarks. The second column gives the execution time, including its range, in seconds. The third column gives the variation of the execution time of the lazy version of the collections against the standard, non-lazy, version of the Pharo collection. The fourth column gives the execution times version of the benchmark using SLua, using the non-lazy Pharo collection as base. A negative variation indicates a benchmark execution time reduction and a positive variation indicates a longer execution time.

Table 5 shows that SLua offers better performance for 3 benchmarks than the standard and lazy Pharo collections. For the benchmarks CircularTreeMap and Mondrian, SLua is 15.7% and 11.8% faster than the standard Pharo collections, respectively. The table also indicates that these better performances are not solely due to the lazy support of SLua.

Differences in ForceBasedLayout and NameCloud benchmarks are not significant because of the high variance. It can be seen that the lazy collections (LC) increase performance in the cases of CircularTreeMap and Mondrian, so part of SLua's performance boost in those benchmarks (in relation to NC) is due to lazy initialization. It is also noted that LC produces a slowdown in the Spectrograph benchmark, which is then increased when using SLua. This suggests that lazy initialization can also slightly degrade performance in some cases.

Results for memory consumption. To measure the memory consumption of the macro-benchmarks we built a dedicated profiler that monitors all instances of `OrderedCollection`, `Dictionary`, and `SLua` during the benchmark execution. After the benchmark execution, our memory profiler outputs statistics regarding the memory consumed by these collections.

We measured the memory consumption for the five benchmarks. In each case, we computed the allocated bytes when using normal collections, lazy collections, and SLua. The results are shown in Table 6.

Table 6 shows that while lazy Pharo collections (LC column) consumes marginally less memory than NC, SLua consumes significantly less memory (around 19%) in four of our benchmarks. In one of our benchmark, SLua consumes marginally more (0.8%). There is no apparent correlation between this memory consumption reduction and the execution time improvement presented in Table 5.

Table 6
Total memory allocated by normal collections (NC), lazy collections (LC) and SLua. NC is represented in KB while LC and SLua are shown as a percentage of variation in relation to NC.

Benchmark	NC (KB)	LC (%)	SLua (%)
CircularTreeMap	3,373.4	−0.1	0.8
ForceBasedLayout	98.2	−0.1	−18.6
Mondrian	42,788.4	−0.4	−19.4
NameCloud	176.9	−0.1	−18.9
Spectrograph	205,965.7	−0.0001	−19.3

9.7. Summary

This section presented an evaluation of a Pharo implementation of Lua's table data structure. Since we designed the table to be polymorphic with OrderedCollection and Dictionary, we were able to automatically adapt a set of applications and run some benchmarks. In some cases, speedup is significant (15%) and memory consumption is significantly reduced (19%). For two of the five benchmarks, execution time is slightly degraded by Lua's table (−4.2% and −2.5%).

10. Other programming languages

This section reviews four programming languages (Java, C#, Scala, and Ruby) by briefly describing how collections are handled in them and how our results may be applied to them.

Java. The Java Collection Framework is composed of 10 generic interfaces implemented by 10 classes. In addition, the framework offers 5 interfaces for concurrent collections. We restrict our analysis to general purpose collections since concurrent collections are often slower due to their synchronization.

JDK 6 suffers from the same problems as the Pharo implementation of the collections. In JDK 7 and 8, the classes ArrayList, TreeMap, HashMap have been improved with the lazy internal array creation.

However, several classes suffer from the problem we have identified, even in JDK 8. For example, the classes Hashtable, Vector, and ArrayDeque create an internal array of size 10 when instantiated, therefore presenting the very same problem we have identified in Pharo.

C#. ArrayList is similar to its Java sibling and Pharo's OrderedCollection. The C# version of ArrayList initializes its internal array with an empty array, resulting in an implementation equivalent to the lazy internal array creation (Section 5). Similarly to ArrayList, Stack initializes its internal array storage with an empty list, thus triggering an expansion at the first element addition.

On the other hand, Hashtable, Dictionary, and Queue do not lazily create the internal array, making these classes suffer from the problems we have identified in this article.

Scala. Instead of simply wrapping Java collections as many languages do when running on top of the Java Virtual Machine, Scala offers a rich trait-based collection library that supports statically checked immutability [22] (which Java does not support). The implementation design of expandable collections in Scala is similar to Pharo.

However, Scala collections suffer from the very same problems we have identified in Pharo. For example, the class ArrayBuffer which is the equivalent of Java's ArrayList creates an empty array of default size 16. The array creation occurs in the ResizableArray superclass.[13] All classes deriving from ResizableArray face the problematic situation we have identified in this article.

Ruby. Ruby provides a complete implementation of array, the most used expandable collection in Ruby, in the virtual machine. All the arithmetic operations, copy, element addition and removing are carried out by the virtual machine. Ruby associates to each empty collection an array of size 16, thus recreating the problematic situations identified in Pharo.

VisualWorks. VisualWorks is another Smalltalk dialect. VisualWorks' collection class library is very similar to Pharo's. A notable difference is (i) OrderedCollection is initialized with an internal array storage with a size of 5, while in Pharo initializes a new ordered collection with a capacity of 10, and (ii) Dictionary is initialized with an internal array of size 3, while Pharo uses 5 as a default size. VisualWorks provides many collections that are related to text processing, and cannot be considered as generic containers.

Applicability of our results. In our experiment we have identified a significant amount of empty collections. Similar behavior was found in other situations. For example, when conducting the case studies in Java with Chameleon [4], a high proportion of empty collections were also identified.

The collection frameworks of Java, C#, Scala, and Ruby largely behave similarly to Pharo. We therefore expect our improvement on the Pharo library to have a positive and significant applicability on these collection libraries. As future work, we plan to verify our assumption by modifying the standard library and running established benchmarks: DaCapo [14] and

[13] http://bit.ly/ResizableArrayScala.

SPECjbb are commonly used benchmarks. Note that it has been shown that SPECjbb is a more demonstrative collection user than DaCapo [23].

11. Related work

Patterns of memory inefficiency. A set of recurrent memory patterns have been identified by Chis et al. [13]. Overheads in Java come from object headers, null pointers, and collections. Three of their 11 patterns (P1, P3, P4) are about unused portions of internal arrays of collections. The model *ContainerOrContained* has been proposed to detect occurrences of these patterns.

We have proposed the lazy internal array creation technique to efficiently address pattern *P1 – empty collections*. Addressing pattern *P3 – small collections* is unfortunately not easy. *P4 – Sparsely populated collections* is addressed by our SLua table implementation. Our collection profiler identifies the provenance of collections having an unnecessary large capacity. However refactoring the base application to properly set the capacity does not result in a significant impact (only a reduction of 0.13% of allocated bytes has been measured). As future work, we plan to verify whether some patterns, depending on the behavior of the application, may be identified (e.g., a method that always produces collections of a same size).

This paper gives our preliminary results about the effect of Lua's table implementation to reduce resources consumption. As future work, we plan to explore how tables are filled and expanded. Such future research will be key to understanding the gain shown in some of our measurements.

Storage strategies. Use of primitive types in Python may trigger a large number of boxing and unboxing operations. Storage strategies [6] significantly reduce the memory footprint of homogeneous collections. Each collection has a storage strategy that is dynamically chosen upon element additions. Homogeneous collections use a dedicated storage to optimize the resources consumed by the storage.

Our approach focuses on reducing the memory footprint of expandable collections, which is different, but complementary to the approach of Bolz, Diekmann and Tratt which focuses on the representation in memory of homogeneous collections.

Discontiguous arrays. Traditional implementations of memory models use a continuous storage. Associating a continuous memory portion to a collection is known to be a source of wasted space which leads to unpredictable performance due to garbage collection pauses [24]. *Discontiguous arrays* is a technique that consists in dividing arrays into indexed memory chunks [25,26,12,7]. Such techniques are particularly adequate for real-time and embedded systems.

Implementing these techniques in an existing virtual machine usually comes at a heavy cost. In particular, the garbage collector has to be aware of discontiguous arrays. A garbage collector is usually a complex and highly optimized piece of code, which makes it very delicate to modify. Bugs that may be inadvertently introduced when modifying it may result in severe and hard-to-trace crashes.

Our results show that a significant improvement may be carried out without any low-level modification in the virtual machine or in the executing platform. Many of our experiments about memory profiling in Pharo have been carried out having simultaneously multiple different versions of the collection library. Nevertheless, research results about discontinuous arrays, in particular Z-rays [12], may be beneficial to expandable collections. In the future, we plan to work on this.

Tables use a form of discontinuity since some data may be contained in the array part of the table and other in the hash part. On the other hand, tables use continuous arrays for the array part. Therefore, combining discontiguous arrays are likely to be beneficial. We will also investigate this as future work.

Dynamic adaptation. Choosing the most appropriate collection implementation is not simple. The two collections ArrayList and HashSet are often chosen because their behavior is well known, which makes them popular. Improperly chosen collection implementations may lead to unnecessary resource consumption. Xu [8] proposes an optimization technique to dynamically adapt a collection into the one that fits best according to its usage (e.g., replacing a LinkedList with an ArrayList).

Xu's approach is similar to the storage strategies mentioned above, which makes it complementary to our approach.

Adaptive selection of collections. In the same line as dynamic adaption, Shacham et al. [4] describe Chameleon, a profiler specific to collections which outputs a list of appropriate collection implementations. The correction can be either made automatically, or presented to the programmer for correction. A small domain-specific language is described to define rules to characterize the use of collections.

Chameleon assumes (i) a set of interchangeable implementation for every collection type and (ii) different implementations have the same logical behavior. For example, a set may be implemented using an array or a linked-list, while insuring that the collection has no duplicates.

Chameleon maintains a list of rules performed on dynamically gathered data. A rule may be an ArrayList on which many contains(...) operations are performed. If the condition expressed by the rules is found, then replacing the ArrayList by a LinkedHashSet is offered to the developer to avoid iterating over the collection at each contains(...) operation.

The effort of Chameleon is orthogonal to the technique presented in this article. Chameleon makes suggestions on a proper abstract data type, while our research is about optimizing the use of the internal storage array.

Recycling collections. The idea of recycling some collections classes has been investigated in the past. For example, functional languages create a new copy, at least in principle, at each element addition or removal. Avoiding such copies has been the topic of numerous research work [18,27].

Recycling collections when possible is known to be effective [28]. For example, *Java Performance Tuning* [19], Chapter 4, Page 79, mentions "Most container objects (e.g., Vectors, Hashtables) can be reused rather than created and thrown away." However, no evidence about the gain is given. In the case of Pharo, recycling internal arrays of expandable collections reduces the number of allocated bytes by 9.4%. The book also argues that recycling collections is effective in a multi-threaded setting. It supports the idea that programmers should make their collections reusable, whenever is possible. Our work embeds this notion of recycling arrays within the collection library itself.

The notion of unnecessary or redundant computation within loops has been the topic of some recent work [29–32]. An efficient model for reusing objects at the loop iteration level are provided. For example, reusing collections within a loop leads to a "20–40% reduction in object churn" and "the execution time improvements range between 6–20%." Object churn refers to the excessive generation of temporary objects. Our approach essentially embeds the improvement within the collection library, which has the advantage to not impact the programmer's habits. However, our performance improvements are smaller.

Adaptive collection. The Clojure programming language[14] offers persistent data structures. Such data structures have their implementation based on the usage of the internal array storage. For example, a PersistentArrayMap is promoted to a PersistentHashMap once the collection exceeds 16 entries. This behavior is therefore related to Lua's table data structure, except that some of Clojure's data structure automatically migrate their internal representation.

12. Conclusion and future work

Expandable collections are an important piece of the runtime. Although intensively used, expandable collections are a potential source of wasted memory space and CPU consumption. This work shows that the collection code libraries may be significantly improved regarding time and memory performance, without impacting the behavior of applications that use them.

We have improved the performance of expandable collections by using two different approaches. The first approach went through three different steps, as described in Section 5, Section 6 and Section 7. We have defined a total of 32 executions of 17 different applications, which generate over 6M of expandable collections. The execution blueprint of these collections obtained with the standard collection library is given in Table 7. We have developed OptimizedCollection, a collection library that supports lazy array creation and array recycling. The execution profile of the benchmark is given in Table 9. The positive effect of our collection is given by contrasting Table 9 against Table 7. OptimizedCollection has:

- reduced the number of created intermediary internal array storage by (6,205,920 − 1,798,380)/6,205,920 = 71.02% (column *NAC*)
- reduced the number of allocated bytes by (253,288,572 − 78,336,068)/253,288,572 = 69.07% (column *NAB*)
- reduced the number of unused bytes by (228,171,448 − 61,383,684)/228,171,448 = 73.09% (column *NUB*)

Recycling arrays incurs a time penalty during the execution. Our benchmark runs 3% faster for all but one performance benchmark.

As our second approach to improve the performance of collections, we have implemented Lua's table data structure in Pharo, polymorphic with OrderedCollection and Dictionary. We were able to run a number of benchmarks by replacing these two Pharo collections with our table implementation. In some cases, the obtained speedup is significant (15%) and the memory consumption is significantly reduced (19%). For two of the five benchmarks, execution time is slightly degraded by Lua's table (−4.2% and −2.5%).

Our results were obtained with representative and well-supported Pharo applications. Although these applications make use of threads and concurrency, they are not commonly considered as truly concurrent applications. We deliberately excluded concerns about concurrent aspects of our work by considering the abilities of the applications we have.

Our improvement of the Pharo library is available on http://smalltalkhub.com/#!/~abergel/MCollection. The Pharo implementation of Lua table is available on http://smalltalkhub.com/#!/~smaass/SmartCollection. Our implementation is available under the MIT license. The inclusion of our work into the Pharo distribution is currently under way. The Pharo community is concerned with reducing the memory footprint of its applications and it is widely recognized that improving the collection library has a global positive impact. Our new collections are backward compatible with the old collection.

Some future directions of this work are to consider other incrementation strategies rather than doubling the size of the internal array. It is likely that gain may be gained by considering an incrementation strategy per collection creation site. Another future work will consider the garbage collection activity when monitoring the memory consumption. Currently, we are monitoring the object creation without considering whether they are kept in memory and for how long. Interfacing our profiler to the garbage collector would add a new dimension to our effort.

We have carefully reviewed the collection implementations of Java (JDK6, JDK7, JDK8), C#, Ruby and Scala. These implementations suffer from the same symptoms found in Pharo. We hope this article will contribute to improving collection libraries across programming languages and serve as a guideline for future collection designers.

[14] http://clojure.org.

Acknowledgements

We thank Oscar Nierstrasz, Lukas Renggli, Eric Tanter, and Renato Cerro for their comments on an early draft of this article. We also thank Aleksandar Prokopec for his help with Scala collections.

We gratefully thank LAM Research for its financial support. Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PCHA/Doctorado Nacional para extranjeros/2013-63130199. Alejandro Infante is supported by CONICYT-PCHA/MagísterNacional/2015-22150809.

Appendix A. Pharo syntax

Pharo is a dynamically typed programming language, with a syntax relatively close to Ruby, Objective-C, and Smalltalk. Pharo's syntax is designed to that program text can be read aloud as though it is a simplified English. Consider the code given in Section 7:

```
1 ROView>>elementsToRender
2   "Return the number of elements that will be rendered"
3   | answer |
4   answer := OrderedCollection new: (self elements size).
5   self elementsToRenderDo: [ :el | answer add: el. ].
6   ^ answer
```

Line 1 indicates that the given source code defines the method `elementsToRender` on the class ROView. Line 2 is a method comment. Line 3 declares the answer temporary variable. Since Pharo is dynamically typed, no type annotation comes with the variable definition. Line 4 creates an instance of the class OrderedCollection, with an initial capacity. The capacity is the value of the expression self elements size. The pseudo variable self represents the object which has received the message elementsToRender. To that object, the message elements is sent, and size is sent to the result. Lines 5 takes a block closure as argument.

Appendix B. Application benchmark detail & measurement

Table 7
Original benchmark (baseline for all the other measurements).

Bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*
bAST1	210,000	38,000(18%)	172,000(81%)	0	0
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200
bN1	150	0(0%)	150(100%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000
bN3	240	240(100%)	0(0%)	60	9,000
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	436,800
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800
bPP3	546,710	398,420(72%)	148,290(27%)	52,860	6,475,120
bReg1	1,000	200(20%)	800(80%)	0	0
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,680
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600
bR2	2,530	1,583(62%)	947(37%)	117	15,608
bR3	79,456	53,259(67%)	26,197(32%)	4,809	686,196
total	**6,129,207**	**1,637,669(26%)**	**4,491,538(73%)**	**980,792**	**46,953,084**

Bench.	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
bAST1	210,000	38,000(18%)	38,000(18%)	6,752,000	6,468,000
bAST2	183,000	41,000(22%)	41,000(22%)	5,928,000	5,580,000
bAST3	431,220	87,570(20%)	87,570(20%)	13,795,440	13,212,720
bN1	150	0(0%)	0(0%)	3,000	3,000
bN2	240	120(66%)	120(66%)	22,440	7,680
bN3	300	180(75%)	180(75%)	22,680	7,560
bPP1	96,200	46,200(50%)	46,200(50%)	4,214,400	3,033,600
bPP2	84,600	44,800(57%)	44,800(57%)	3,790,400	2,571,200
bPP3	599,570	398,420(72%)	398,420(72%)	29,103,720	17,192,120
bReg1	1,000	100(10%)	100(10%)	34,400	33,600
bReg2	2,162,860	10(0%)	10(0%)	86,513,920	84,799,800
bReg3	1,950,010	10(0%)	10(0%)	78,001,720	76,093,720
bR1	400,055	0(0%)	3(0%)	17,263,480	13,023,236
bR2	2,642	289(11%)	299(11%)	141,056	99,404
bR3	84,073	13,365(16%)	13,454(16%)	7,701,916	6,045,808
total	**6,205,920**	**670,064(10%)**	**670,166(10%)**	**253,288,572**	**228,171,448**

Table 8
Lazy internal array creation.

Bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*
bAST1	210,000	38,000(18%)	172,000(81%)	0	0
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200
bN1	150	0(0%)	150(100%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000
bN3	240	240(100%)	0(0%)	60	9,000
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	436,800
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800
bPP3	546,710	398,420(72%)	148,290(27%)	52,860	6,475,120
bReg1	1,000	200(20%)	800(80%)	0	0
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,680
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600
bR2	2,422	1,583(65%)	839(34%)	117	15,608
bR3	78,145	53,259(68%)	24,886(31%)	4,809	686,196
total	**6,127,788**	**1,637,669(26%)**	**4,490,119(73%)**	**980,792**	**46,953,084**

Bench.	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
bAST1	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	53,000	41,000(22%)	41,000(22%)	1,016,000	668,000
bAST3	113,040	87,570(20%)	87,570(20%)	2,389,680	1,806,960
bN1	0	0(0%)	0(0%)	0	0
bN2	210	120(66%)	120(66%)	21,840	7,080
bN3	300	180(75%)	180(75%)	22,680	7,560
bPP1	78,000	46,200(50%)	46,200(50%)	3,490,400	2,309,600
bPP2	70,200	44,800(57%)	44,800(57%)	3,218,400	1,999,200
bPP3	543,770	398,420(72%)	398,420(72%)	26,952,320	15,040,720
bReg1	200	100(10%)	100(10%)	7,200	6,400
bReg2	428,000	10(0%)	10(0%)	17,120,000	15,405,880
bReg3	476,070	10(0%)	10(0%)	19,044,600	17,136,600
bR1	52	0(0%)	3(0%)	5,263,360	1,223,116
bR2	1,698	289(11%)	299(12%)	109,356	67,704
bR3	63,400	13,365(17%)	13,454(17%)	7,034,296	5,378,188
total	**1,874,940**	**670,064(10%)**	**670,166(10%)**	**86,510,132**	**61,393,008**

Table 9
Lazy internal array creation + reuse of array.

Bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*
bAST1	210,000	38,000(18%)	172,000(81%)	0	0
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200
bN1	150	0(0%)	150(100%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000
bN3	240	240(100%)	0(0%)	60	9,000
bPP1	91,000	46,400(50%)	44,600(49%)	5,600	437,600
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800
bPP3	546,710	398,420(72%)	148,290(27%)	52,170	6,449,480
bReg1	1,000	200(20%)	800(80%)	0	0
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,720
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600
bR2	2,422	1,583(65%)	839(34%)	117	15,608
bR3	78,145	53,259(68%)	24,886(31%)	4,872	699,036
total	**6,128,188**	**1,637,869(26%)**	**4,490,319(73%)**	**980,165**	**46,941,124**

Bench.	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
bAST1	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	49,002	41,000(22%)	41,000(22%)	992,012	668,000
bAST3	110,370	87,570(20%)	87,570(20%)	2,361,480	1,806,960
bN1	0	0(0%)	0(0%)	0	0
bN2	153	120(66%)	120(66%)	13,400	7,080
bN3	243	180(75%)	180(75%)	14,240	7,560
bPP1	72,603	46,400(50%)	46,400(50%)	3,058,196	2,312,800
bPP2	63,604	44,800(57%)	44,800(57%)	2,743,088	2,000,000
bPP3	490,915	398,420(72%)	398,420(72%)	20,488,808	15,051,560
bReg1	200	100(10%)	100(10%)	7,200	6,400
bReg2	427,970	10(0%)	10(0%)	17,119,200	15,405,880
bReg3	476,011	10(0%)	10(0%)	19,042,492	17,136,640
bR1	38	0(0%)	3(0%)	5,243,040	1,023,116
bR2	1,597	289(11%)	299(12%)	94,656	67,712
bR3	58,872	13,365(17%)	13,454(17%)	6,375,776	5,390,776
total	**1,798,578**	**670,264(10%)**	**670,366(10%)**	**78,373,588**	**61,420,484**

Table 10
Lazy internal array creation + reuse of array + code refactoring.

Bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*
bAST1	210,000	38,000(18%)	172,000(81%)	0	0
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200
bN1	150	0(0%)	150(100%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000
bN3	240	240(100%)	0(0%)	60	9,000
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	437,600
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800
bPP3	546,710	398,420(72%)	148,290(27%)	52,170	6,449,480
bReg1	1,000	200(20%)	800(80%)	0	0
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,720
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600
bR2	2,422	1,583(65%)	839(34%)	117	15,608
bR3	78,145	53,259(68%)	24,886(31%)	4,872	699,036
total	**6,127,788**	**1,637,669(26%)**	**4,490,119(73%)**	**980,165**	**46,941,124**

Bench.	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
bAST1	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	49,002	41,000(22%)	41,000(22%)	992,012	668,000
bAST3	110,370	87,570(20%)	87,570(20%)	2,329,080	1,774,560
bN1	0	0(0%)	0(0%)	0	0
bN2	154	120(66%)	120(66%)	11,280	4,920
bN3	244	180(75%)	180(75%)	12,120	5,400
bPP1	72,403	46,200(50%)	46,200(50%)	3,057,396	2,312,800
bPP2	63,604	44,800(57%)	44,800(57%)	2,743,088	2,000,000
bPP3	490,915	398,420(72%)	398,420(72%)	20,488,808	15,051,560
bReg1	200	100(10%)	100(10%)	7,200	6,400
bReg2	427,970	10(0%)	10(0%)	17,119,200	15,405,880
bReg3	476,011	10(0%)	10(0%)	19,042,492	17,136,640
bR1	38	0(0%)	3(0%)	5,243,040	1,023,116
bR2	1,597	289(11%)	299(12%)	94,616	67,672
bR3	58,872	13,365(17%)	13,454(17%)	6,375,736	5,390,736
total	**1,798,380**	**670,064(10%)**	**670,166(10%)**	**78,336,068**	**61,383,684**

References

[1] W.R. Cook, On understanding data abstraction, revisited, SIGPLAN Not. 44 (10) (2009) 557–572, https://doi.org/10.1145/1639949.1640133, http://www.cs.utexas.edu/~wcook/Drafts/2009/essay.pdf.

[2] K. Wolfmaier, R. Ramler, H. Dobler, Issues in testing collection class libraries, in: Proceedings of the 1st Workshop on Testing Object-Oriented Systems, ETOOS '10, ACM, New York, NY, USA, 2010, pp. 4:1–4:8, http://doi.acm.org/10.1145/1890692.1890696.

[3] S. Ducasse, D. Pollet, A. Bergel, D. Cassou, Reusing and composing tests with traits, in: Tools'09: Proceedings of the 47th International Conference on Objects, Models, Components, Patterns, Zurich, Switzerland, 2009, pp. 252–271, http://hal.archives-ouvertes.fr/docs/00/40/35/68/PDF/Reusing_Composing.pdf.

[4] O. Shacham, M. Vechev, E. Yahav, Chameleon: adaptive selection of collections, in: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, ACM, New York, NY, USA, 2009, pp. 408–418, http://doi.acm.org/10.1145/1542476.1542522.

[5] J.Y. Gil, Y. Shimron, Smaller footprint for Java collections, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH '11, ACM, New York, NY, USA, 2011, pp. 191–192, http://doi.acm.org/10.1145/2048147.2048201.

[6] C.F. Bolz, L. Diekmann, L. Tratt, Storage strategies for collections in dynamically typed languages, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, ACM, New York, NY, USA, 2013, pp. 167–182, http://doi.acm.org/10.1145/2509136.2509531.

[7] S. Joannou, R. Raman, An empirical evaluation of extendible arrays, in: Proceedings of the 10th International Conference on Experimental Algorithms, SEA'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 447–458, http://dl.acm.org/citation.cfm?id=2008623.2008663.

[8] G. Xu, 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, 2013, pp. 1–26, https://doi.org/10.1007/978-3-642-39038-8_1.

[9] A. Bergel, D. Cassou, S. Ducasse, J. Laval, Deep into Pharo, Square Bracket Associates, http://rmod.lille.inria.fr/pbe2/, 2013.

[10] D. Cassou, S. Ducasse, R. Wuyts, Traits at work: the design of a new trait-based stream library, Comput. Lang. Syst. Struct. 35 (1) (2009) 2–20, https://doi.org/10.1016/j.cl.2008.05.004, http://scg.unibe.ch/archive/papers/Cass08a-NileNewKernel-ComputerLanguages.pdf.

[11] S. Wilson, J. Kesselman, Java Platform Performance, Prentice Hall PTR, 2000, http://java.sun.com/docs/books/performance.

[12] J.B. Sartor, S.M. Blackburn, D. Frampton, M. Hirzel, K.S. McKinley, Z-rays: divide arrays and conquer speed and flexibility, in: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 471–482, http://doi.acm.org/10.1145/1806596.1806649.

[13] A.E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, J. Murphy, Patterns of memory inefficiency, in: Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 383–407, http://dl.acm.org/citation.cfm?id=2032497.2032523.

[14] S.M. Blackburn, R. Garner, C. Hoffmann, A.M. Khang, K.S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S.Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J.E.B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, 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, 2006, pp. 169–190, http://doi.acm.org/10.1145/1167473.1167488.

[15] A. Georges, D. Buytaert, L. Eeckhout, 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, 2007, pp. 57–76, http://doi.acm.org/10.1145/1297027.1297033.

[16] T. Mytkowicz, A. Diwan, M. Hauswirth, P.F. Sweeney, Producing wrong data without doing anything obviously wrong!, in: Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '09, ACM, New York, NY, USA, 2009, pp. 265–276.

[17] T. Kalibera, R. Jones, Rigorous benchmarking in reasonable time, in: Proceedings of the 2013 International Symposium on Memory Management, ISMM '13, ACM, New York, NY, USA, 2013, pp. 63–74, http://doi.acm.org/10.1145/2464157.2464160.

[18] A. Kagedal, S. Debray, A Practical Approach to Structure Reuse of Arrays in Single AssignmentLanguages, Tech. rep., Tucson, AZ, USA 1996.

[19] J. Shirazi, Java Performance Tuning, 2nd edition, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[20] C. Bruni, Optimizing Pinocchio, Master's thesis, University of Bern, Jan. 2011, http://scg.unibe.ch/archive/masters/Brun11a.pdf.

[21] A. Bergel, F. Bañados, R. Robbes, D. Röthlisberger, Spy: a flexible code profiling framework, Comput. Lang. Syst. Struct. 38 (1) (2012) 16–28, https://doi.org/10.1016/j.cl.2011.10.002, http://bergel.eu/download/papers/Berg10f-Spy.pdf.

[22] M. Odersky, A. Moors, Fighting bit rot with types (experience report: Scala collections), in: R. Kannan, K.N. Kumar (Eds.), IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2009, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 4, Schloss Dagstuhl–Leibniz-Zentrum Fuer Informatik, Dagstuhl, Germany, 2009, pp. 427–451, http://drops.dagstuhl.de/opus/volltexte/2009/2338.

[23] A. Potanin, M. Damitio, J. Noble, Are your incoming aliases really necessary? Counting the cost of object ownership, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 742–751, http://dl.acm.org/citation.cfm?id=2486788.2486886.

[24] P. Wilson, M. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: a survey and critical review, in: H. Baler (Ed.), Memory Management, in: Lecture Notes in Computer Science, vol. 986, Springer, Berlin, Heidelberg, 1995, pp. 1–116, https://doi.org/10.1007/3-540-60368-9_19.

[25] D.F. Bacon, P. Cheng, V.T. Rajan, A real-time garbage collector with low overhead and consistent utilization, in: Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '03, ACM, New York, NY, USA, 2003, pp. 285–298, http://doi.acm.org/10.1145/604131.604155.

[26] G. Chen, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, B. Mathiske, M. Wolczko, Heap compression for memory-constrained Java environments, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 282–301, http://doi.acm.org/10.1145/949305.949330.

[27] N. Mazur, P. Ross, G. Janssens, M. Bruynooghe, Practical aspects for a working compile time garbage collection system for mercury, in: P. Codognet (Ed.), Logic Programming, in: Lecture Notes in Computer Science, vol. 2237, Springer, Berlin, Heidelberg, 2001, pp. 105–119.

[28] G. Xu, Finding reusable data structures, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12, ACM, New York, NY, USA, 2012, pp. 1017–1034, http://doi.acm.org/10.1145/2384616.2384690.

[29] N. Mitchell, 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, ACM, New York, NY, USA, 2007, pp. 245–260, http://doi.acm.org/10.1145/1297027.1297046.

[30] S. Bhattacharya, M.G. Nanda, K. Gopinath, M. Gupta, Reuse, recycle to de-bloat software, in: Proceedings of the 25th European Conference on Object-Oriented Programming, ECOOP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 408–432, http://dl.acm.org/citation.cfm?id=2032497.2032524.

[31] G. Xu, D. Yan, A. Rountev, Static detection of loop-invariant data structures, in: Proceedings of the 26th European Conference on Object-Oriented Programming, ECOOP'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 738–763, https://doi.org/10.1007/978-3-642-31057-7_32.

[32] A. Nistor, L. Song, D. Marinov, S. Lu, Toddler: detecting performance problems via similar memory-access patterns, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 562–571, http://dl.acm.org/citation.cfm?id=2486788.2486862.