Reducing Waste in Expandable Collections: The Pharo Case*

Alexandre Bergel, Alejandro Infante, Juan Pablo Sandoval Alcocer

Pleiad Lab, DCC, University of Chile

ABSTRACT

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 a new collection library for the Pharo programming language that exhibits better resource management than the standard library. 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 with a speedup of about 3% for most of our benchmarks. We analyzed the collection implementations of Java, C#, Scala, and Ruby: these implementations largely behave as in Pharo's, therefore suffering from the very same limitations. Our result are likely to benefit designers of future programming languages and collection libraries.

1. INTRODUCTION

Creating and manipulating any arbitrary group of values is largely supported by today's programming languages and runtimes [Cook 2009]. 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 [Cook 2009, Wolfmaier et al. 2010, Ducasse et al. 2009], including 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 have been the topic of a number of studies [Gil and Shimron 2011, Bolz et al. 2013, Joannou and Raman 2011, Shacham et al. 2009].

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 [Gil and Shimron 2011, Xu 2013, Shacham et al. 2009]. 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 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. Several studies have been made to evaluate the amount of wasted memory resources [Gil and Shimron 2011, Xu 2013, Shacham et al. 2009], however, as far as we are aware of, we present here the first case study made on expandable collections.

We have selected the Pharo programming language for our study. Pharo[1] is a dynamically typed programming language which offers a large and rich collection library. 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 reduce 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 contribution for improvement, which means that our results are to have a measurable impact across Pharo developers.

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. The improvements we propose are 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

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

focus on the profiling of over 6M expandable collections produced by 15 different program executions. 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 shows the Pharo collection library can be significantly improved by considering lazy array creation and recycling those arrays. 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 8. We therefore expect our recommendations to be beneficial in these languages.

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 discusses the case of other languages. Section 9 presents the work related to this article. Section 10 concludes and presents our future work.

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 8 for a detailed comparison).

The collection library is a complex piece of code that exhibits different complex aspects [Cassou et al. 2009]. 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.

Issues with expansions. Expandable collections are remarkable pieces of software: most expandable collections have a complex semantic 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 decrease it (but do 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. A 0.75 and 1.0 are commonly used thresholds (0.75 for collections operating with hashtags values and 1.0 for every other collection). Consider the class `OrderedCollection`, a frequently used expandable collection. Consider an ordered collection of a given capacity c. Adding one element to the collection 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:

```
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 a linked element instead. For this reason we have voluntary excluded this class from our analysis.

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 M 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. 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 30M elements produces 22 expansions, incurring $\sum_{i=0}^{21} 10 * 2^i = 41$M slot copies. In total, $41 * 4 =$ 164Mb of memory 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 [Wilson and Kesselman 2000], 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 [Wilson and Kesselman 2000].

Collector pauses. Garbage collection copies and joins portions of memory to reduce memory fragmentation [Joannou and Raman 2011]. 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 [Sartor et al. 2010].

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 48Mb 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 [Chis et al. 2011]). 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 on real applications, we pick a representative set of Pharo applications and profile their execution.

3.1 Benchmark descriptions

We pick 5 open source software projects from the Pharo ecosystem stored on the Pharo forge[2]. These applications, listed in Table 1, have been selected for our study for two reasons:

- They are actively supported and represent relevant assets for the Pharo community. Therefore, our results are likely to raise interest from this community essentially composed of industrials. These applications are daily used both in industries and academia.
- The community is friendly and interested in collaborating with researchers. As a results, 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 benchmark 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:

[2] `http://smalltalkhub.com`

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

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

- *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.
- *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 equally 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 [Blackburn et al. 2006] and SpecJVM. We have designed our benchmark from performance scenarios of program executions.

The tables given at the end of the article show the results of our measurements. Table 3 gives the measurement of our benchmark 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 [Kalibera and Jones 2013, Mytkowicz et al. 2009, Georges et al. 2007]. An effective approach to minimize measurement bias is called *experimental setup randomization* [Mytkowicz et al. 2009], 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 is easily automatized. We have considered the following parameters:

- *Hardware and OS* – We have used two different hardwares and operating systems ((a) a MacBook Air, 1.3Ghz Intel Core I5, 4Gb 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).
- *Heap size* – We run our experiments using different initial size of the heap (100Mb, 500Mb, 1000Mb).
- *Repeated run* – For each execution of the complete benchmark, we have 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 cache located in the VM are emptied before each run.
- *GC* – 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 the appendix are the result of an average of 9 different executions, each considering a different combination of the parameters given above.

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 Dynamic analysis

We have run our two sets of our benchmark and profiled their executions. The metrics given in Section 3.2 have been computed and reported in Table 3 for each of the application's execution. The execution of the 15 performance benchmarks create 6,129,207 expandable collections.

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 we are focusing 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[3].

[3]Redefining `=` without redefining `hash` is a classic defect in software programs and it is widely recognized as such. Unfortunately, this defect is frequent.

Empty collections. Table 3 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%) have been 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.

Figure 1 shows the frequency distribution of the performance benchmarks.

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 `RBVariable Environment` comes with at least four instances of dictionaries. Most instances of `RBVariableEnvironment` actually have their dictionaries empty, which contributes to the 98% of the collections created by Benchmark 10 being left empty. This is not an isolated case. The 17 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 to the number of expansions (NCE) is equal to the number of collections (NC). We have roughly the following relation $NAC - NCE = NC$ in Table 3. Some differences may be noticed 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 6.1M of collections (NC column in Table 3), 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 46.9Mb of copies between these arrays (NCB column).

Unused memory. Summing up the memory consumed by all the internal arrays yields over 253Mb (NAB column). More than 228Mb of these 253Mb 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

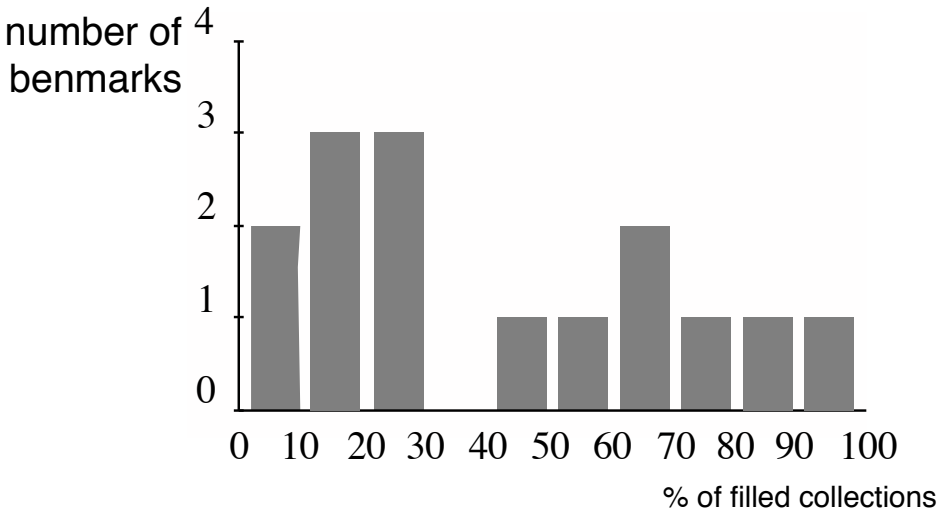

Figure 1: Frequency distribution of filled collections (*NNEC*)

much below its capacity).

4.2 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.,* at the first element added. 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.

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 collection has been little considered (except a very few exceptions in the Java and C# collections) despite the significant memory overhead collections may generate [Gil and Shimron 2011].

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 with 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 4 gives the metric values of our benchmark when

using the lazy internal array creation. Contrasting Table 3 (using the standard collection library, *i.e.,* without lazy internal array creation) with Table 4 (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 down 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. It went from 228Mb down to 61Mb, 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 [Kagedal and Debray 1996]. 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 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 which has a size within a range of 20% of what is requested.

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**	40628

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

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*. Result given in Table 5 uses 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 4), 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 89kB.

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.

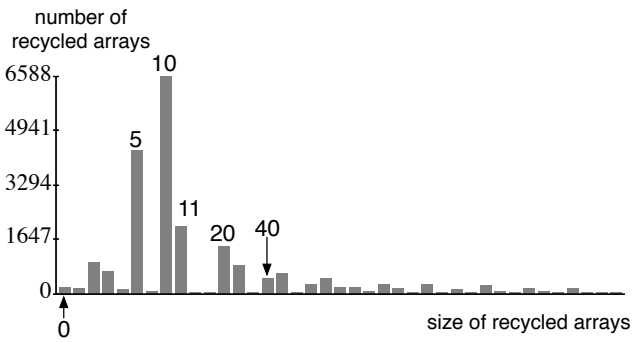

Figure 2: Distribution of recycled arrays

Recycled arrays. The techniques described in this section recycle arrays of different sizes. Figure 2 shows the distribution of size of recycled arrays for Strategy $S1$. The vertical axis indicates the number of recycled arrays. The horizontal axis lists the size of arrays that are effectively recycled.

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[4]. The value 20 corresponds to the size of the internal array of a default collection after expansion. An array of size 40 is obtained after a second expansion.

Multi-threading. The pool of recycled internal array is globally accessible. Accesses to the pool need 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 [Shirazi 2002] 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).

Figure 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

[4]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.

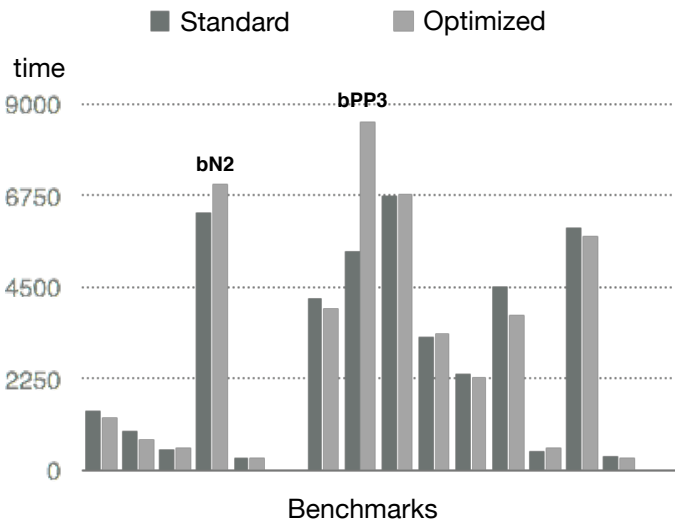

Figure 3: Impact of execution time of the optimized collection library

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 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.

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 are focusing 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 refactor. 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 6 details the profiling for the benchmark by lazily creating internal arrays, reusing these arrays and refactoring the applications. By comparing from Table 5 to Table 6, the reduction gain for the number of allocated bytes is 0.05% (column *NAB*, which goes from 78.37Mb to 78.33Mb). The amount of unused space has been reduced by 0.06% (column *NUB*, which goes from 61.42Mb down to 61.38Mb). No variation in terms of execution time has been found.

Setting the capacity. We have run 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 has been measured. We conclude that the obtained gain by allocating a proper initial capacity is marginal.

8. OTHER PROGRAMMING LANGUAGES

This section reviews four programming languages (Java, C#, Scala, and Ruby) by briefly describing how collections are handled in these 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 than the Pharo implementation of the collection. 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` creates 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 [Odersky and Moors 2009] (which Java does not support). The implementation design of expandable collections in Scala is similar to Pharo.

However, the Scala collection suffers 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 a default size 16. The array creation occurs in the `ResizableArray` superclass[5]. 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

[5] `http://bit.ly/ResizableArrayScala`

16, thus recreating the problematic situations identified in Pharo.

Applicability of our results. In our experiment we have identified a significant amount of empty collections. Similar behavior has been found in other situations. For example, when conducting the case studies in Java with Chameleon [Shacham et al. 2009], a high proportion of empty collections have also been 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 impact on these collection libraries. As future work, we plan to verify our assumption by modifying the standard library and running established benchmarks: DaCapo [Blackburn et al. 2006] and SPECjbb are commonly used benchmarks. Note it has been shown that SPECjbb is a more demonstrative collection user than DaCapo [Potanin et al. 2013].

9. RELATED WORK

Patterns of memory inefficiency. A set of recurrent memory patterns have been identified by Chis *et al.* [Chis et al. 2011]. Overheads in Java come from object headers, null pointers, and collections. Three of their 11 patterns (P1, P3, P4) are about unused portions 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. 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).

Storage strategies. Use of primitive types in Python may trigger a large number of boxing and unboxing operations. Storage strategies [Bolz et al. 2013] significantly reduces 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.

Storage strategies may be considered as a generalization of the lazy internal array creation described above. 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 homogenous collections.

Discontiguous arrays. Traditional implementation of memory model uses 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 [Wilson et al. 1995]. *Discontiguous arrays* is a technique that consists in dividing arrays into indexed memory chunks [Joannou and Raman 2011, Sartor et al. 2010, Bacon et al. 2003, Chen et al. 2003]. 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 [Sartor et al. 2010], may be beneficial to expandable collections. In the future, we plan to work on this.

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 implementation may lead to unnecessary resource consumption. Xu [Xu 2013] 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.* [Shacham et al. 2009] describe a profiler specific to collections which outputs a list of appropriate collection implementation. 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 use of collections.

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 [Kagedal and Debray 1996, Mazur et al. 2001].

Recycling collections when possible is known to be effective [Xu 2012]. For example, *Java Performance Tuning* [Shirazi 2002], 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%. This book chapter 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 [Mitchell and Sevitsky 2007, Bhattacharya et al. 2011, Xu et al. 2012, Nistor et al. 2013]. 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[6] 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.

10. 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.

Improving the performance of expandable collections 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 3. 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 5. The positive effect of our collection is given by contrasting Table 5 against Table 3. 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.

Some future directions of this work are to consider other incrementation strategies 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.

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 in improving collection libraries across programming languages and will serve as a guideline for future collection designers.

Acknowledgments. 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.

11. REFERENCES

[Bacon et al. 2003] David F. Bacon, Perry Cheng, and V. T. Rajan. 2003. 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, 285–298. DOI: http://dx.doi.org/10.1145/604131.604155

[Bhattacharya et al. 2011] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta. 2011. Reuse, recycle to de-bloat software. In *Proceedings of the 25th European conference on Object-oriented programming (ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 408–432. http://dl.acm.org/citation.cfm?id=2032497.2032524

[Blackburn et al. 2006] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. DOI: http://dx.doi.org/10.1145/1167473.1167488

[Bolz et al. 2013] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. 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, 167–182. DOI: http://dx.doi.org/10.1145/2509136.2509531

[Cassou et al. 2009] Damien Cassou, Stéphane Ducasse, and Roel Wuyts. 2009. Traits at Work: the design of a new trait-based stream library. *Journal of Computer Languages, Systems and Structures* 35, 1 (2009), 2–20. DOI:http://dx.doi.org/10.1016/j.cl.2008.05.004

[Chen et al. 2003] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, and M. Wolczko. 2003. 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, 282–301. DOI: http://dx.doi.org/10.1145/949305.949330

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

[Cook 2009] William R. Cook. 2009. On understanding data abstraction, revisited. *SIGPLAN Not.* 44, 10 (2009), 557–572. DOI: http://dx.doi.org/10.1145/1639949.1640133

[Ducasse et al. 2009] Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Damien Cassou. 2009. Reusing and Composing Tests with Traits. In *Tools'09: Proceedings of the 47th International Conference on Objects, Models, Components, Patterns*. Zurich,

[6] `http://clojure.org`

Switzerland, 252–271. http://hal.archives-ouvertes.fr/docs/00/40/35/68/PDF/Reusing_Composing.pdf

[Georges et al. 2007] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications (OOPSLA '07)*. ACM, New York, NY, USA, 57–76. DOI: http://dx.doi.org/10.1145/1297027.1297033

[Gil and Shimron 2011] Joseph (Yossi) Gil and Yuval Shimron. 2011. 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, 191–192. DOI: http://dx.doi.org/10.1145/2048147.2048201

[Joannou and Raman 2011] Stelios Joannou and Rajeev Raman. 2011. An Empirical Evaluation of Extendible Arrays. In *Proceedings of the 10th International Conference on Experimental Algorithms (SEA'11)*. Springer-Verlag, Berlin, Heidelberg, 447–458. http://dl.acm.org/citation.cfm?id=2008623.2008663

[Kagedal and Debray 1996] Andreas Kagedal and Saumya Debray. 1996. *A Practical Approach to Structure Reuse of Arrays in Single AssignmentLanguages*. Technical Report. Tucson, AZ, USA.

[Kalibera and Jones 2013] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 63–74. DOI: http://dx.doi.org/10.1145/2464157.2464160

[Mazur et al. 2001] Nancy Mazur, Peter Ross, Gerda Janssens, and Maurice Bruynooghe. 2001. Practical Aspects for a Working Compile Time Garbage Collection System for Mercury. In *Logic Programming*, Philippe Codognet (Ed.). Lecture Notes in Computer Science, Vol. 2237. Springer Berlin Heidelberg, 105–119. DOI:http://dx.doi.org/10.1007/3-540-45635-X_15

[Mitchell and Sevitsky 2007] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 245–260. DOI: http://dx.doi.org/10.1145/1297027.1297046

[Mytkowicz et al. 2009] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. 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, 265–276. DOI:http://dx.doi.org/10.1145/1508244.1508275

[Nistor et al. 2013] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. 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, 562–571. http://dl.acm.org/citation.cfm?id=2486788.2486862

[Odersky and Moors 2009] Martin Odersky and Adriaan Moors. 2009. Fighting bit Rot with Types (Experience Report: Scala Collections). In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2009) (Leibniz International Proceedings in Informatics (LIPIcs))*, Ravi Kannan and K Narayan Kumar (Eds.), Vol. 4. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 427–451. DOI:http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2009.2338

[Potanin et al. 2013] Alex Potanin, Monique Damitio, and James Noble. 2013. 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, 742–751. http://dl.acm.org/citation.cfm?id=2486788.2486886

[Sartor et al. 2010] Jennifer B. Sartor, Stephen M. Blackburn, Daniel Frampton, Martin Hirzel, and Kathryn S. McKinley. 2010. 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, 471–482. DOI: http://dx.doi.org/10.1145/1806596.1806649

[Shacham et al. 2009] Ohad Shacham, Martin Vechev, and Eran Yahav. 2009. 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, 408–418. DOI: http://dx.doi.org/10.1145/1542476.1542522

[Shirazi 2002] Jack Shirazi. 2002. *Java Performance Tuning* (2nd ed.). O'Reilly & Associates, Inc., Sebastopol, CA, USA.

[Wilson et al. 1995] PaulR. Wilson, MarkS. Johnstone, Michael Neely, and David Boles. 1995. Dynamic storage allocation: A survey and critical review. In *Memory Management*, HenryG. Baler (Ed.). Lecture Notes in Computer Science, Vol. 986. Springer Berlin Heidelberg, 1–116. DOI: http://dx.doi.org/10.1007/3-540-60368-9_19

[Wilson and Kesselman 2000] Steve Wilson and Jeff Kesselman. 2000. *Java Platform Performance*. Prentice Hall PTR. http://java.sun.com/docs/books/performance

[Wolfmaier et al. 2010] Klaus Wolfmaier, Rudolf Ramler, and Heinz Dobler. 2010. Issues in Testing Collection Class Libraries. In *Proceedings of the 1st Workshop on Testing Object-Oriented Systems (ETOOS '10)*. ACM, New York, NY, USA, Article 4, 8 pages. DOI: http://dx.doi.org/10.1145/1890692.1890696

[Xu 2012] Guoqing Xu. 2012. 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, 1017–1034. DOI: http://dx.doi.org/10.1145/2384616.2384690

[Xu 2013] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *Proceedings of the 27th European Conference on Object-Oriented*

Programming (ECOOP'13). Springer-Verlag, Berlin, Heidelberg, 1–26. DOI: http://dx.doi.org/10.1007/978-3-642-39038-8_1

[Xu et al. 2012] Guoqing Xu, Dacong Yan, and Atanas Rountev. 2012. Static Detection of Loop-invariant Data Structures. In *Proceedings of the 26th European Conference on Object-Oriented Programming (ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 738–763. DOI: http://dx.doi.org/10.1007/978-3-642-31057-7_32

APPENDIX

A. APPLICATION BENCHMARK DETAIL & MEASUREMENT

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 3: 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	2,422	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 4: 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	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 5: 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	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**

Table 6: Lazy internal array creation + reuse of array + code refactoring