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. In the Pharo programming language, expandable collection classes offer an easy-to-use API, however this apparent simplicity is accompanied by a significant amount of wasted resource.

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 Pharo that exhibits better resource management than the standard library. Across a basket of 17 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 further discuss the applicability of our findings to other languages, including Java, C#, Scala, and Ruby.

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 exhibits a wide range of features [1–3], including being expandable or not. An expandable collection is a collection whose size may vary as elements are added and removed. 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 [4–6]. Pharo[1] is a dynamically typed programming language which offers a large and rich collection library. Consider the case of a simple ordered collection (`OrderedCollection` in Pharo and `ArrayList` in Java). 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. 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.

This paper is about measuring wasted resources in Pharo (memory and execution time) due to expandable collections. Improvements are then deduced and we measure their impact.

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

OOPSLA '14, Month d–d, 20yy, City, ST, Country.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnnReprinted from OOPSLA '14, [Unknown Proceedings], Month d–d, 20yy, City, ST, Country, pp. 1–??.

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

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. We therefore expect our recommendations to be beneficial in these languages.

This paper is structured as follows: Section 2 describes the Pharo expandable collections and synthesizes their implementation. Section 3 describes a benchmark composed of 17 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 paper. Section 10 concludes and presents our future work.

2. Pharo's Expandable Collections

The collection library is a complex piece of code that exhibits different complex aspects [7]. 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. 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 are designed to only expand. Removing elements from a collection does not trigger any shrinkage of the internal collection. We therefore only focus on element addition and not removal.

2.1 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 growing or shrinking the collection 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 shrinks it. 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 collections). 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.

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, $41 * 4 = 164$Mb 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 [9], 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 [9].

Collector pauses. Garbage collection copies and joins portions of memory to reduce memory fragmentation [10]. Copying and scanning a large portion of memory, such as collections, may cause large and unpredictable collection pause times. The garbage collection pauses in proportion to array size [11].

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 unused portion of the array has already been mentioned (Pattern 1, 3, 4 in [12]). Our paper 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

Appendix A lists the 17 Pharo applications we consider in our benchmark. These applications are open source[2], thus easing a replication of our experiments. These applications are daily used both in industries and academia. They are furthermore supported by active communities.

We employ the benchmark to approximate how expandable collections are used in general.

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.
- *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.
- *NSM* – **N**umber of collections filled in the **S**ame **M**ethods that have created them.
- *NAB* – **N**umber of **A**llocated **B**ytes – Accumulated size of all the internal arrays created by a collection.

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

- *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*.

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 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 sums up 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 [13]. We have designed our benchmark from two different sets of program executions: (i) execution of unit tests and (ii) performance scenarios.

Unit-test benchmarks. Running unit tests is convenient in our setting since unit tests are likely to represent common usage and execution scenarios [14]. We execute the unit tests associated with each of the 17 applications.

The primarily purpose of a unit test is to validate correctness. A unit test typically represents a short execution scenario that is quick to run. We will profile the execution of unit tests to measure the creation of short-lived small collections. Executing unit tests is an action often performed by a programmer. Optimizing this action is therefore a valuable contribution. Note that we consider unit tests as part of the applications. This means that collections created within a unit test are counted in our measurement.

Performance benchmarks. We use 15 benchmarks that perform a computation on a large amount of input data. From the 17 applications, we consider 5 applications for which it makes actual sense to run a long execution. These applications are marked with an * and we have three benchmarks for each of these. These benchmarks have been written by the authors of the considered application and represent a typical heavy usage of the application.

Referring to the benchmarks. The tables given at the end of the papers show the result of our measurements. We refer to the execution of unit tests for application X as “Benchmark X”, X ranging from 1 to 17. The long executions are referred to as “Benchmark bAST*Y*, bN*Y*, bPP*Y*, bReg*Y*, and bR*Y*”, where Y ranges from 1 to 3.

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 [8, 15]. An effective approach to minimize measurement bias is called *experimental setup randomization* [15], which consists in generating a large number of experimental settings by varying some parameters, each considered parameters 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 and (b) iMac, Quad-core Intel Core i5, 8 Gb).
- *Heap size* – We run our experiment using different initial size of the heap (20Mb, 150Mb, 300Mb).
- *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 benchmarks (*i.e.,* a unit-test benchmark or a performance benchmark) 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 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 applications execution.

The execution of the 17 unit test benchmarks create a total of 2,474,499 expandable collections and the 15 performance benchmarks create 6,342,087 expandable collections. The analyses this paper describes focus on the profiling of nearly 9M expandable collections produced by 32 different program executions (17 unit test benchmarks + 15 performance benchmarks).

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 1, 10, 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 2, 6, 8, bN1, bN2, bN3 create a small number of collections, indicating short executions.

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 10 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].

Empty collections. Table 3 indicates a surprisingly high proportion of empty collections in our benchmarks. From over 8.6 million expandable collections created by our benchmarks, 6.6 million (76%) have been created without having any element added to them. Only 23% of collections have at least one element added to them during their lifetime.

To understand this phenomena 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. Consider the application 10 and its corresponding benchmark. Benchmark 10 creates a total of 1.4M of expandable collections, for which only 14,891 are non-empty. This application is a refactoring engine that applies pattern matching and rewriting rules on source code. The engine is complex due to the underlying optimized logic engine[4]. By excluding this application, the ratio of the number of not empty collections for the unit test benchmarks rises to 31.3% (*NC*= 1,043,634 and *NNEC*= 327,445): about one-third of collections created by the unit tests are left empty in the average.

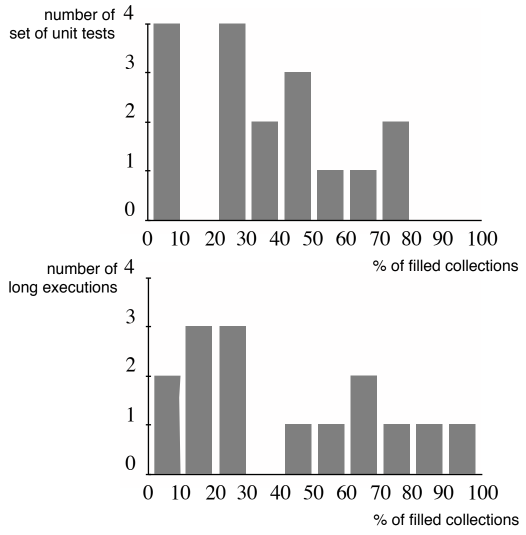

Figure 1: Frequency distribution of filled collections (*NNEC*)

Figure 1 shows the frequency distribution of the benchmark for both unit tests and performance benchmarks. There are four applications that have less than 10% of collections empty, and four applications that have between 20% and 30% of collections that are not empty. Performance benchmarks have a tendency to fill more expandable collections compared with unit test benchmarks. This highlight an important difference between two unit-test benchmarks and performance benchmarks. The first has tendency to create many empty collections over the latter. Benchmark bN3 generates no empty collection.

Cause of empty collections. We manually have inspected the applications and benchmarks that generate a high proportion

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

[4] Interestingly, PMD, a Java application similar to Refactoring, exhibits the very same problem [6].

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. The class `RBVariableEnvironment` is part of a code meta-model that belongs to the application `Refactoring`. 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 and lazy evaluation of variable are 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 8.6M of collections (*NC* column), only 0.56% of the collections are expanded 1,002,212 times during the execution of the benchmark (*NCE* column). These expansions result in over 62Mb of copies between these arrays (*NCB* column).

Unused memory. Summing up the memory consumed by all the internal arrays yields over 342Mb. More than 296Mb of these 342Mb are actually unused 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).

4.2 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 conceived 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

In Pharo, 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*.

This section describes the first design point, which is to support lazy internal array creation.

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 8,701,783 down to 2,437,083, representing a reduction of (8,605,147 − 2,437,083) / 8,605,147 = 71.67% of array creation.
- The number of unused bytes (*NUB*) has also been significantly reduced. It went from 296Mb down to 82Mb, representing a reduction of (296 − 82) / 296 = 72.29%.

Application 10 is producing a high number of expandable collections that remains empty during the overall execution. Using the original collection library, Application 10 created 1,438,380 internal arrays (*NAC* column in Table 3). Making the collection library support the lazy internal array creation makes this value goes to 68,098. A reduction of (1,438,380 − 68,132) / 1,438,380 = 95% of created arrays.

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 has been reduced by 2.38%.

6. Recycling Internal Arrays

A collection expansion is carried out with three sequential steps (Section 2.1): (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 [17]. 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.

The effects of the different strategies on the unit-test benchmark is summarized in Table 1. 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*,

metrics	*S1*	*S2*	*S3*
NC	2,475,658	2,475,670	2,475,708
NCE	21,134	**19,118**	21,139
NCB	15,519,656	**14,761,492**	15,544,560
NAC	**542,622**	542,757	542,863
NAB	31,467,464	36,983,740	**31,445,140**
NUB	**21,332,420**	26,808,248	21,334,960
#full GC	40	44	40
#incr GC	14,442	**14,423**	20,314

Table 1: Effect of the different strategies for the unit test benchmarks (best performance is indicated in **bold**)

which also result in fewer copied bytes (*NCB*). Oddly, the number of incremental garbage collections is higher with *S3*.

Effect on the benchmark. Table 5 details the use of strategy *S1* on the benchmark. When supporting the lazy internal array creation without recycling arrays (Table 4), the number of unused bytes has been reduced by $(82,927,120 - 82,752,904)/82,927,120 = 0.2\%$. The reduction of the number of created arrays is $(2,437,083 - 2,341,191)/2,437,083 = 4\%$. In all, 35,063 collections have been recycled. More interestingly, the technique of reusing arrays has reduced the number of allocated bytes by 14.6% (column *NAB* : $(128,678,564 - 109,840,556)/128,678,564 = 14.6\%$).

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.

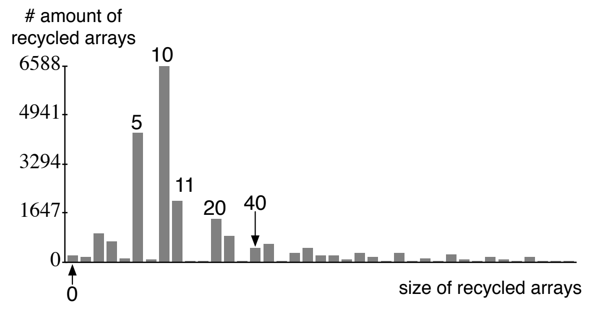

Figure 2: Distribution of recycled arrays

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[5]. 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. However, the execution scenarios we consider are not thread-intensive. Previous work on pooling reusable collections [18] 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).

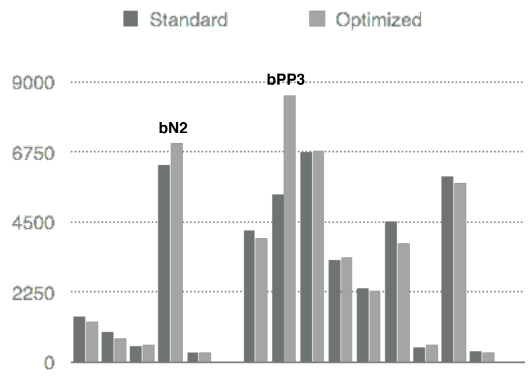

Figure 3: Impact of execution time of the optimized collection library

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 *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.

[5] 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.

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 massing 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, which indicates us that its 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. When considering this outlier, our performance benchmark runs 7.9% slower.

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
  | 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 2.1.

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 204,680 collections 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. The reduction gain for the number of allocated bytes is 0.11% (column *NAB*, which goes from 109.840Mb to 109.713Mb). The amount of unused space has been reduced by 0.12% (column *NUB*, which goes from 82.752Mb down to 82.651Mb). 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. Again, gains are marginal. Only a reduction of 0.13% 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 result 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, however.

Classes describing collections are very similar to Pharo's. For example[6], the class `ArrayList` uses an internal array to store elements, as `OrderedCollection` does. The class `HashSet` wraps an instance of `HashMap`. `HashMap` uses an array of `Entry` elements, each entry being an association *(key, value)*. The implementation of `HashSet` is again very similar to Pharo's `Dictionary`, with a 0.75 threshold to trigger an expansion. The class `TreeMap` does not have an equivalent in standard Pharo and uses an array to store a collection's elements.

In Pharo we did not consider the class `LinkedList` since this class is only used by the runtime and not by user-defined applications. However, in Java, `LinkedList` is used more and other collection classes are built on it, *e.g.,* `LinkedHashSet` and `LinkedHashMap`.

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).

`Hashtable` uses an internal array which is created with the proper capacity when the class is instantiated. `Hashtable` does not use an empty array as `ArrayList` does. The class `Dictionary` and `Queue` do not lazy initialize its internal array storage. Similarly to `ArrayList`, `Stack` initializes its internal array storage with an empty list, thus triggering an expansion at the first element addition.

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 [19] (which Java does not support). The design of expandable collections in Scala is similar to Java. `ArrayBuffer` which is the equivalent of Java's `ArrayList` creates an empty array of a default size 16.

`ArrayBuffer` extends `ResizeableArray`[7] utility class used by several other collections. The private array field in that class is called `array` and the logic of manipulating it is the same as with Java's `ArrayList`.

Ruby. Oddly, Ruby provides the 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.

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 [6], a high proportion of empty collections have also been identified.

The collection framework of Java[8], C#, Scala, and Ruby behave similarly to Pharo, except for the C# version of `ArrayList` and `Stack`. 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 assumption by modifying the standard library and running established benchmarks (e.g., DaCapo [13]).

9. Related Work

Patterns of memory inefficiency. A set of recurrent memory patterns have been identified by Chis *et al.* [12]. 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 produce 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 [20] 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.

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 continuous storage. Associating a continuous

[6] The source code of `ArrayList` is visible online on http://bit.ly/ArrayListOpenJDK6

[7] https://github.com/scala/scala/blob/ master/src/library/scala/collection/-mutable/ResizableArray.scala

[8] A private discussion with some developers at Oracle indicates that an updated version of the Collection library in JDK 7 will soon support lazy array creation.

memory portion to a collection is known to be a source of wasted space which leads to unpredictable performance due to garbage collection pauses [21]. *Discontiguous arrays* is a technique that consists in dividing arrays into indexed memory chunks [10, 11, 22, 23]. 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 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 [11], 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 [5] 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.* [6] 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 [17, 24].

Recycling collections when possible is known to be effective [25]. For example, *Java Performance Tuning* [18], 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 14.6%. This chapter also argues that recycling collections is effective in a multi-threaded setting. Although our benchmarks includes multi-threaded applications, our execution did not make an heavy use of threads. This book chapter supports the idea that programmers should make their collection reusable, whenever is possible. Our work embeds this notion of recycling arrays within the collection library itself.

The notion of redundant computation within loops has been the topic of some recent work [26–28]. Efficient model for reusing objects at loop iteration are provided. For example, reusing collections within 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 improvement are lesser.

Adaptive collection. The Clojure programming language[9] 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 nearly 9M 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 (8,701,783 − 2,341,191) / 8,701,783 = 73.09% (column *NAC*)
- reduced the number of allocated bytes by (342,818,892 − 109,840,556) / 342,818,892 = 67.95% (column *NAB*)
- reduced the number of unused bytes by (296,863,696 − 82,752,904) / 296,863,696 = 72.12% (column *NUB*)

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

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

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

References

[1] W. R. Cook, On understanding data abstraction, revisited, SIGPLAN Not. 44 (10) (2009) 557–572.

[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.

[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.

[4] J. Y. Gil, Y. Shimron, Smaller footprint for java collections, in: Proceedings of OOPSLA'11, pp. 191–192.

[5] G. Xu, Coco: Sound and adaptive replacement of java collections, in: Proceedings of ECOOP'13, pp. 1–26.

[6] O. Shacham, M. Vechev, E. Yahav, Chameleon: Adaptive selection of collections, in: Proceedings of PLDI '09, pp. 408–418.

[7] D. Cassou, S. Ducasse, R. Wuyts, Traits at work: the design of a new trait-based stream library, Journal of Computer Languages, Systems and Structures 35 (1) (2009) 2–20.

[8] 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.

[9] S. Wilson, J. Kesselman, Java Platform Performance, Prentice Hall PTR, 2000.

[10] 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.

[11] 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 PLDI '10, pp. 471–482.

[12] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, J. Murphy, Patterns of memory inefficiency, in: Proceedings of ECOOP'11, pp. 383–407.

[13] 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 OOPSLA '06, pp. 169–190.

[14] R. C. Martin, Agile Software Development. Principles, Patterns, and Practices, Prentice-Hall, 2002.

[15] 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.

[16] M. Dias, M. M. Peck, S. Ducasse, G. Arévalo, Fuel: a fast general purpose object graph serializer, Software: Practice and Experience 44 (4) (2014) 433–453.

index	Application	LOC	#Ref
*1	AST	8,091	57
2	Arki	627	6
3	Glamour	17,525	105
4	GraphET	2,757	10
5	Magritte	5,884	29
6	Manifest	2,864	11
7	NECompletion	3,446	33
*8	Nautilus	1,566	9
*9	Petit	14,919	95
10	Refactoring	21,328	125
*11	Regex	5,055	16
12	Ring	3,378	50
*13	Roassal	19,844	133
14	RoelTyper	2,003	85
15	Shout	3,290	10
16	SmallDude	3,805	66
17	Spec	10,212	37

Table 2: Description of the benchmark (the #Ref column indicates the number of references to expandable collection in source code)

[17] A. Kagedal, S. Debray, A practical approach to structure reuse of arrays in single assignmentlanguages, Tech. rep., Tucson, AZ, USA (1996).

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

[19] 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)

[20] C. F. Bolz, L. Diekmann, L. Tratt, Storage strategies for collections in dynamically typed languages, in: Proceedings of OOPSLA '13, pp. 167–182.

[21] P. Wilson, M. Johnstone, M. Neely, D. Boles, Dynamic storage allocation: A survey and critical review, in: H. Baler (Ed.), Memory Management, Vol. 986 of LNCS, 1995, pp. 1–116.

[22] D. F. Bacon, P. Cheng, V. T. Rajan, A real-time garbage collector with low overhead and consistent utilization, in: Proceedings of POPL '03, pp. 285–298.

[23] G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, B. Mathiske, M. Wolczko, Heap compression for memory-constrained java environments, in: Proceedings of OOPSLA '03, pp. 282–301.

[24] 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, Vol. 2237 of LNCS, 2001, pp. 105–119.

[25] G. Xu, Finding reusable data structures, in: Proceedings of OOPSLA '12, pp. 1017–1034.

[26] S. Bhattacharya, M. G. Nanda, K. Gopinath, M. Gupta, Reuse, recycle to de-bloat software, in: Proceedings of ECOOP'11, pp. 408–432.

[27] G. Xu, D. Yan, A. Rountev, Static detection of loop-invariant data structures, in: Proceedings of ECOOP'12, pp. 738–763.

[28] A. Nistor, L. Song, D. Marinov, S. Lu, Toddler: Detecting performance problems via similar memory-access patterns, in: Proceedings ICSE '13, pp. 562–571.

A. Application Benchmark Detail

B. Measurement

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,603	151,665(23%)	491,938(76%)	2,636	35,940	646,239	112,308(17%)	112,439(17%)	18,819,920	17,907,204
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	78,976	34,912(44%)	44,064(55%)	322	2,272	79,298	10,604(13%)	10,681(13%)	2,374,464	2,213,384
4	1,690	512(30%)	1,178(69%)	30	1,720	1,720	245(14%)	245(14%)	65,760	55,420
5	3,191	2,009(62%)	1,182(37%)	30	1,208	3,193	200(6%)	251(7%)	77,808	67,724
6	96	44(45%)	52(54%)	0	0	96	40(41%)	40(41%)	3,740	3,464
7	612	218(35%)	394(64%)	614	2,257,960	1,231	48(7%)	48(7%)	4,657,696	837,948
8	2	0(0%)	2(100%)	0	0	2	0(0%)	0(0%)	40	40
9	158,589	58,371(36%)	100,218(63%)	5,663	280,876	164,252	57,644(36%)	57,728(36%)	6,833,532	5,534,280
10	1,432,306	14,891(1%)	1,417,415(98%)	6,074	12,258,424	1,438,380	8,248(0%)	8,344(0%)	46,958,632	34,241,204
11	6,839	2,058(30%)	4,781(69%)	1,280	78,712	6,967	471(6%)	480(7%)	291,052	256,852
12	8,363	3,530(42%)	4,833(57%)	1,103	47,852	9,466	73(0%)	73(0%)	279,236	165,460
13	108,571	57,590(53%)	50,981(46%)	1,739	369,336	109,990	8,151(7%)	8,810(8%)	5,778,016	4,845,764
14	10,305	586(5%)	9,719(94%)	145	14,044	10,448	82(0%)	110(1%)	443,828	420,352
15	20,815	14,886(71%)	5,929(28%)	255	125,900	21,070	5,736(27%)	5,740(27%)	2,692,404	1,984,240
16	766	172(22%)	594(77%)	17	496	783	123(16%)	123(16%)	126,368	122,532
17	1,203	880(73%)	323(26%)	1,512	48,384	2,715	764(63%)	764(63%)	127,356	35,988
total	**2,475,940**	**342,336(13%)**	**2,133,604(86%)**	**21,420**	**15,523,124**	**2,495,863**	**204,738(8%)**	**205,877(8%)**	**89,530,320**	**68,692,248**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	210,000	38,000(18%)	38,000(18%)	6,752,000	6,468,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	183,000	41,000(22%)	41,000(22%)	5,928,000	5,580,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	431,220	87,570(20%)	87,570(20%)	13,795,440	13,212,720
bN1	150	0(0%)	150(100%)	0	0	150	0(0%)	0(0%)	3,000	3,000
bN2	180	150(83%)	30(16%)	60	9,000	240	120(66%)	120(66%)	22,440	7,680
bN3	240	240(100%)	0(0%)	60	9,000	300	180(75%)	180(75%)	22,680	7,560
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	436,800	96,200	46,200(50%)	46,200(50%)	4,214,400	3,033,600
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	84,600	44,800(57%)	44,800(57%)	3,790,400	2,571,200
bPP3	546,710	398,420(72%)	148,290(27%)	52,860	6,475,120	599,570	398,420(72%)	398,420(72%)	29,103,720	17,192,120
bReg1	1,000	200(20%)	800(80%)	0	0	1,000	100(10%)	100(10%)	34,400	33,600
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	2,162,860	10(0%)	10(0%)	86,513,920	84,799,800
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,680	1,950,010	10(0%)	10(0%)	78,001,720	76,093,720
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	400,055	0(0%)	3(0%)	17,263,480	13,023,236
bR2	2,530	1,583(62%)	947(37%)	117	15,608	2,642	289(11%)	299(11%)	141,056	99,404
bR3	79,456	53,259(67%)	26,197(32%)	4,809	686,196	84,073	13,365(16%)	13,454(16%)	7,701,916	6,045,808
total	**6,129,207**	**1,637,669(26%)**	**4,491,538(73%)**	**980,792**	**46,953,084**	**6,205,920**	**670,064(10%)**	**670,166(10%)**	**253,288,572**	**228,171,448**
Total	**8,605,147**	**1,980,005(23%)**	**6,625,142(76%)**	**1,002,212**	**62,476,208**	**8,701,783**	**874,802(10%)**	**876,043(10%)**	**342,818,892**	**296,863,696**

Table 3: Original benchmark (baseline for all the other measurements)

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,603	151,665(23%)	491,938(76%)	2,636	35,940	229,379	112,308(17%)	112,439(17%)	3,956,792	3,044,076
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	79,035	34,970(44%)	44,065(55%)	45	2,272	44,294	10,612(13%)	10,689(13%)	1,397,668	1,236,260
4	1,690	512(30%)	1,178(69%)	30	1,720	905	245(14%)	245(14%)	33,160	22,820
5	3,551	2,009(56%)	1,542(43%)	30	1,208	2,685	200(5%)	251(7%)	65,208	55,124
6	96	44(45%)	52(54%)	0	0	91	40(41%)	40(41%)	3,560	3,284
7	644	218(33%)	426(66%)	614	2,257,960	927	48(7%)	48(7%)	4,509,216	689,468
8	2	0(0%)	2(100%)	0	0	0	0(0%)	0(0%)	0	0
9	158,395	58,331(36%)	100,064(63%)	5,663	280,876	123,703	57,637(36%)	57,718(36%)	5,220,152	3,921,204
10	1,432,306	14,891(1%)	1,417,415(98%)	6,074	12,258,424	68,132	8,248(0%)	8,344(0%)	19,547,672	6,830,248
11	6,839	2,058(30%)	4,781(69%)	1,280	78,712	2,186	471(6%)	480(7%)	109,212	75,012
12	7,870	3,472(44%)	4,398(55%)	1,094	44,384	5,253	20(0%)	20(0%)	194,732	84,716
13	108,571	57,590(53%)	50,981(46%)	1,739	369,336	62,510	8,151(7%)	8,810(8%)	4,218,656	3,286,404
14	10,305	586(5%)	9,719(94%)	145	14,044	963	82(0%)	110(1%)	136,008	112,532
15	20,815	14,886(71%)	5,929(28%)	255	125,900	18,489	5,736(27%)	5,740(27%)	2,640,784	1,932,632
16	766	172(22%)	594(77%)	17	496	221	123(16%)	123(16%)	14,288	10,452
17	1,203	880(73%)	323(26%)	1,512	48,384	2,392	764(63%)	764(63%)	120,856	29,488
total	**2,475,704**	**342,296(13%)**	**2,133,408(86%)**	**21,134**	**15,519,656**	**562,143**	**204,686(8%)**	**205,822(8%)**	**42,168,432**	**21,334,112**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	53,000	41,000(22%)	41,000(22%)	1,016,000	668,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	113,040	87,570(20%)	87,570(20%)	2,389,680	1,806,960
bN1	150	0(0%)	150(100%)	0	0	0	0(0%)	0(0%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000	210	120(66%)	120(66%)	21,840	7,080
bN3	240	240(100%)	0(0%)	60	9,000	300	180(75%)	180(75%)	22,680	7,560
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	436,800	78,000	46,200(50%)	46,200(50%)	3,490,400	2,309,600
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	70,200	44,800(57%)	44,800(57%)	3,218,400	1,999,200
bPP3	546,710	398,420(72%)	148,290(27%)	52,860	6,475,120	543,770	398,420(72%)	398,420(72%)	26,952,320	15,040,720
bReg1	1,000	200(20%)	800(80%)	0	0	200	100(10%)	100(10%)	7,200	6,400
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	428,000	10(0%)	10(0%)	17,120,000	15,405,880
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,680	476,070	10(0%)	10(0%)	19,044,600	17,136,600
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	52	0(0%)	3(0%)	5,263,360	1,223,116
bR2	2,422	1,583(65%)	839(34%)	117	15,608	1,698	289(11%)	299(12%)	109,356	67,704
bR3	78,145	53,259(68%)	24,886(31%)	4,809	686,196	63,400	13,365(17%)	13,454(17%)	7,034,296	5,378,188
total	**6,127,788**	**1,637,669(26%)**	**4,490,119(73%)**	**980,792**	**46,953,084**	**1,874,940**	**670,064(10%)**	**670,166(10%)**	**86,510,132**	**61,393,008**
Total	**8,603,492**	**1,979,965(23%)**	**6,623,527(76%)**	**1,001,926**	**62,472,740**	**2,437,083**	**874,750(10%)**	**875,988(10%)**	**128,678,564**	**82,727,120**

Table 4: Lazy internal array creation

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,603	151,665(23%)	491,938(76%)	2,636	35,940	226,736	112,308(17%)	112,439(17%)	3,921,136	3,044,076
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	78,977	34,912(44%)	44,065(55%)	45	2,272	44,193	10,604(13%)	10,681(13%)	1,393,396	1,234,432
4	1,690	512(30%)	1,178(69%)	30	1,720	873	245(14%)	245(14%)	31,200	22,820
5	3,551	2,009(56%)	1,542(43%)	30	1,208	2,683	200(5%)	251(7%)	65,168	55,124
6	96	44(45%)	52(54%)	0	0	91	40(41%)	40(41%)	3,560	3,284
7	644	218(33%)	426(66%)	614	2,257,960	235	48(7%)	48(7%)	2,372,616	689,468
8	2	0(0%)	2(100%)	0	0	0	0(0%)	0(0%)	0	0
9	158,399	58,333(36%)	100,066(63%)	5,663	280,876	118,044	57,639(36%)	57,720(36%)	4,938,224	3,921,356
10	1,432,306	14,891(1%)	1,417,415(98%)	6,074	12,258,424	62,065	8,248(0%)	8,344(0%)	11,877,480	6,830,244
11	6,839	2,058(30%)	4,781(69%)	1,280	78,712	2,056	471(6%)	480(7%)	91,804	75,012
12	7,870	3,472(44%)	4,398(55%)	1,094	44,384	4,163	20(0%)	20(0%)	144,144	84,716
13	108,571	57,590(53%)	50,981(46%)	1,739	369,336	61,326	8,151(7%)	8,810(8%)	3,907,184	3,286,404
14	10,305	586(5%)	9,719(94%)	145	14,044	817	82(0%)	110(1%)	119,512	112,532
15	20,815	14,886(71%)	5,929(28%)	255	125,900	18,233	5,736(27%)	5,740(27%)	2,514,840	1,932,620
16	766	172(22%)	594(77%)	17	496	203	123(16%)	123(16%)	13,700	10,452
17	1,203	880(73%)	323(26%)	1,512	48,384	882	764(63%)	764(63%)	72,536	29,488
total	**2,475,650**	**342,240(13%)**	**2,133,410(86%)**	**21,134**	**15,519,656**	**542,613**	**204,680(8%)**	**205,816(8%)**	**31,466,968**	**21,332,420**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	49,002	41,000(22%)	41,000(22%)	992,012	668,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	110,370	87,570(20%)	87,570(20%)	2,361,480	1,806,960
bN1	150	0(0%)	150(100%)	0	0	0	0(0%)	0(0%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000	153	120(66%)	120(66%)	13,400	7,080
bN3	240	240(100%)	0(0%)	60	9,000	243	180(75%)	180(75%)	14,240	7,560
bPP1	91,000	46,400(50%)	44,600(49%)	5,600	437,600	72,603	46,400(50%)	46,400(50%)	3,058,196	2,312,800
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	63,604	44,800(57%)	44,800(57%)	2,743,088	2,000,000
bPP3	546,710	398,420(72%)	148,290(27%)	52,170	6,449,480	490,915	398,420(72%)	398,420(72%)	20,488,808	15,051,560
bReg1	1,000	200(20%)	800(80%)	0	0	200	100(10%)	100(10%)	7,200	6,400
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	427,970	10(0%)	10(0%)	17,119,200	15,405,880
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,720	476,011	10(0%)	10(0%)	19,042,492	17,136,640
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	38	0(0%)	3(0%)	5,243,040	1,023,116
bR2	2,422	1,583(65%)	839(34%)	117	15,608	1,597	289(11%)	299(12%)	94,656	67,712
bR3	78,145	53,259(68%)	24,886(31%)	4,872	699,036	58,872	13,365(17%)	13,454(17%)	6,375,776	5,390,776
total	**6,128,188**	**1,637,869(26%)**	**4,490,319(73%)**	**980,165**	**46,941,124**	**1,798,578**	**670,264(10%)**	**670,366(10%)**	**78,373,588**	**61,420,484**
Total	**8,603,838**	**1,980,109(13%)**	**6,623,729(86%)**	**1,001,299**	**62,460,780**	**2,341,191**	**874,944(8%)**	**876,182(8%)**	**109,840,556**	**82,752,904**

Table 5: Lazy internal array creation + reuse of array

bench.	*NC*	*NNEC*	*NEC*	*NCE*	*NCB*	*NAC*	*NOSM*	*NSM*	*NAB*	*NUB*
1	643,626	151,687(23%)	491,939(76%)	2,648	35,956	226,964	112,329(17%)	112,460(17%)	3,906,608	3,029,456
2	13	12(92%)	1(7%)	0	0	13	1(7%)	1(7%)	468	392
3	79,043	34,974(44%)	44,069(55%)	65	2,108	44,259	10,616(13%)	10,693(13%)	1,384,428	1,225,288
4	1,691	513(30%)	1,178(69%)	30	1,760	874	246(14%)	246(14%)	30,952	22,532
5	3,551	2,009(56%)	1,542(43%)	28	1,120	2,683	200(5%)	251(7%)	65,352	55,308
6	96	44(45%)	52(54%)	0	0	91	40(41%)	40(41%)	3,560	3,284
7	644	218(33%)	426(66%)	614	2,257,960	235	48(7%)	48(7%)	2,372,608	689,492
8	2	0(0%)	2(100%)	0	0	0	0(0%)	0(0%)	0	0
9	158,399	58,333(36%)	100,066(63%)	5,651	280,636	118,043	57,639(36%)	57,720(36%)	4,938,340	3,921,516
10	1,432,306	14,891(1%)	1,417,415(98%)	6,069	12,280,124	62,070	8,248(0%)	8,344(0%)	11,852,428	6,828,504
11	6,839	2,058(30%)	4,781(69%)	1,280	78,760	2,055	471(6%)	480(7%)	91,812	75,064
12	7,870	3,472(44%)	4,398(55%)	1,095	46,236	4,163	20(0%)	20(0%)	145,360	86,528
13	108,571	57,589(53%)	50,982(46%)	1,683	367,164	61,330	8,150(7%)	8,809(8%)	3,865,908	3,245,964
14	10,305	586(5%)	9,719(94%)	145	14,044	819	82(0%)	110(1%)	120,872	112,660
15	20,815	14,886(71%)	5,929(28%)	255	125,900	18,233	5,736(27%)	5,740(27%)	2,514,084	1,932,656
16	766	172(22%)	594(77%)	14	376	202	123(16%)	123(16%)	12,488	9,428
17	1,203	880(73%)	323(26%)	0	0	880	764(63%)	764(63%)	72,472	29,488
total	**2,475,740**	**342,324(13%)**	**2,133,416(86%)**	**19,577**	**15,492,144**	**542,914**	**204,713(8%)**	**205,849(8%)**	**31,377,740**	**21,267,560**
bAST1	210,000	38,000(18%)	172,000(81%)	0	0	47,000	38,000(18%)	38,000(18%)	820,000	536,000
bAST2	179,000	47,000(26%)	132,000(73%)	4,000	24,000	49,002	41,000(22%)	41,000(22%)	992,012	668,000
bAST3	428,550	103,830(24%)	324,720(75%)	2,670	28,200	110,370	87,570(20%)	87,570(20%)	2,329,080	1,774,560
bN1	150	0(0%)	150(100%)	0	0	0	0(0%)	0(0%)	0	0
bN2	180	150(83%)	30(16%)	60	9,000	154	120(66%)	120(66%)	11,280	4,920
bN3	240	240(100%)	0(0%)	60	9,000	244	180(75%)	180(75%)	12,120	5,400
bPP1	90,600	46,200(50%)	44,400(49%)	5,600	437,600	72,403	46,200(50%)	46,200(50%)	3,057,396	2,312,800
bPP2	78,000	44,800(57%)	33,200(42%)	6,600	476,800	63,604	44,800(57%)	44,800(57%)	2,743,088	2,000,000
bPP3	546,710	398,420(72%)	148,290(27%)	52,170	6,449,480	490,915	398,420(72%)	398,420(72%)	20,488,808	15,051,560
bReg1	1,000	200(20%)	800(80%)	0	0	200	100(10%)	100(10%)	7,200	6,400
bReg2	2,162,830	427,970(19%)	1,734,860(80%)	427,950	17,118,080	427,970	10(0%)	10(0%)	17,119,200	15,405,880
bReg3	1,949,950	476,010(24%)	1,473,940(75%)	476,020	19,042,720	476,011	10(0%)	10(0%)	19,042,492	17,136,640
bR1	400,011	7(0%)	400,004(99%)	46	2,631,600	38	0(0%)	3(0%)	5,243,040	1,023,116
bR2	2,422	1,583(65%)	839(34%)	117	15,608	1,597	289(11%)	299(12%)	94,616	67,672
bR3	78,145	53,259(68%)	24,886(31%)	4,872	699,036	58,872	13,365(17%)	13,454(17%)	6,375,736	5,390,736
total	**6,127,788**	**1,637,669(26%)**	**4,490,119(73%)**	**980,165**	**46,941,124**	**1,798,380**	**670,064(10%)**	**670,166(10%)**	**78,336,068**	**61,383,684**
Total	**8,603,528**	**1,979,993(23%)**	**6,623,535(76%)**	**999,742**	**62,433,268**	**2,341,294**	**874,777(10%)**	**876,015(10%)**	**109,713,808**	**82,651,244**

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