Transparent Memory Optimization using Slots

Pablo Tesone, Santiago Bragagnolo, Stéphane Ducasse, Marcus Denker

Transparent Memory Optimization using Slots

Pablo Tesone
Unité de Recherche Informatique et Automatique
IMT Lille Douai, Univ. Lille
Lille, France
Inria Lille-Nord Europe
Lille, France
pablo-adrian.tesone@imt-lille-douai.fr

Santiago Bragagnolo
Inria Lille-Nord Europe
Lille, France
santiago.bragagnolo@inria.fr

Marcus Denker
Inria Lille-Nord Europe
Lille, France
marcus.denker@inria.fr

Stéphane Ducasse
Inria Lille-Nord Europe
Lille, France
stephane.ducasse@inria.fr

Abstract

Memory size limits the number of instances available in memory at a single time. This limitation affects applications that require large graphs of objects. Moose is an example of such applications. Moose is a tool used in software analysis. It parses and models software code in an object graph. Later it performs multiple operations (*i.e.,* measuring, querying, mining and analysing the code) on such models. However, not all the information in the model is present, as the model is intended to be used with different applications and programming languages (not all applications or programming languages uses the same elements). Analysis of different models shows that between 30 and 50% of memory is wasted. Analysis models produced in an industrial context reveals that models composed of several millions of instances used up to 2Gb memory.

In this work, we propose new slots and their combination to transparently optimize memory consumption: NilAwareSlot optimizes automatically nils and LazyInitializationSlot handles the case where an empty collection is required and use by many clients. We show that performing a limited amount of changes, we improved the memory footprint of Moose models in around 30%. We also show that our solution has comparable performance with an ad hoc solution, but without the need for boilerplate code. To implement this solution, we leverage the existing Pharo support of slots, write barriers and efficient forwarders.

CCS Concepts • **Software and its engineering → General programming languages**;

Keywords Memory consumption, optimization, first class instance variables

ACM Reference Format:
Pablo Tesone, Santiago Bragagnolo, Marcus Denker, and Stéphane Ducasse. 2018. Transparent Memory Optimization using Slots. In *Proceedings of International Workshop on Smalltalk Technologies (IWST'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/

IWST'18, September 10–14th, 2018, Cagliari, Italy
2018.

1 Introduction

Complex object-oriented applications require large object graphs in memory. If instances in the graph have uninitialized (nil valued) instance variables or point to empty collections, memory is wasted. As soon as the graph gets bigger, the problem also increases.

Modeling tools such as model checking or software analysis exhibits such problems. Let us take Moose as an example. Moose[1] is an extensible reengineering environment designed to provide the necessary infrastructure for tool development and integration. Moose's core is a language-independent meta-model. It offers services such as grouping, querying, navigation, and advanced tool integration mechanism [6]. To perform different analyses, Moose requires software models to be loaded in memory. The model is represented in a graph of objects following the FAMIX [4, 5] metamodel.

Moose is a typical case since its models have to manipulate often several millions of entities, its models are represented as graphs and storing such graphs does not work well with traditional relational databases [3, 12]. These models do not fit in the relational model, as these databases are designed to query structured data without relational cycles. RDBMSs[2] are efficient unless the data contains many relationships and cycles requiring joins of large tables [14]. Also, by the nature of the model, it cannot be analysed in parts [6].

When analysing concrete Moose models, one can see that memory is wasted. Around 30% of the instance variables have *nil* as value, these instance variables occupy space but do not provide any meaningful information. Also, the model includes a large number of empty collections, that even being empty they take up space. For example, an empty OrderedCollection occupies 68 bytes including its backing array[3].

[1] http://www.moosetechnology.org/
[2] Relational Database Management System
[3] in 32-bits Pharo

This is amplified as the models for an analysed application contains several tenth of millions of objects. For example, the FAMIX model for Hadoop 3.1.0-RC1[4] includes 48 millions of instances. It has 25% of nil instance variables and 25 millions of empty collections. Typical memory waste is caused by nil instance variables and in empty collections (Section 2). Note that such problem is not linked to the analysed models but are artefacts of the modeling approach. Moose captures as much as possible as information to offer the possibility to perform different analyses. Industrial users of Moose, for example, Synectique[5] report than in a model with around 27 millions of instance variables, 10 millions contains nil and there are 5 millions of empty OrderedCollection instances.

This waste of memory may be solved during development using techniques such as lazy initialization, object splicing or storing instance variables in associated data structures (*e.g.,* HashMaps, Arrays, Linked Lists). All of these strategies require boilerplate code to be applied all over the application. However, when the application is already developed these modifications represent a reimplementation of the application with its associate effort.

The contribution of this paper is the design of new slots that implement a transparent memory optimization that does not require boilerplate code nor special VM support (Section 3).

Our solution is implemented in Pharo [2]. Our solution leverages existing Pharo infrastructure, namely Slot [13], Write Barrier [1] and Efficient Forwarders [9] (Section 4).

We validated our solution by using it in the new FAMIX implementation. We reduced memory use by around 30%, depending on the modelled application (Section 5). Even though our solution presents an overhead when compared to the default unoptimized version, our solution has comparable results with a custom solution (Section 6) with the benefit of not requiring boilerplate code. Moreover, these results could have been improved even more but we limited the changes to only perform a small amount of changes to an existing application (Section 7).

There are other solutions to this memory misuse, *e.g.,* Lazy initialization or instance reshaping [7, 10, 11]. However, they require boilerplate code, special Virtual Machine and language support, or recompilation of methods during execution (Section 8).

Finally, we present a conclusion and possible future works (Section 9).

2 Wasted Memory

The problem that we address is, in a nutshell, the waste of memory from unused objects and instance variable slots.

[4]http://hadoop.apache.org/

[5]A company providing software analysis solutions using Moose http://synectique.eu/

2.1 A Simple Example

To illustrate the problem, we propose a simplified example that shows the problem ubiquity. This example represents a small part of a movie database, inspired from the information available in IMDb[6] (Figure 1).

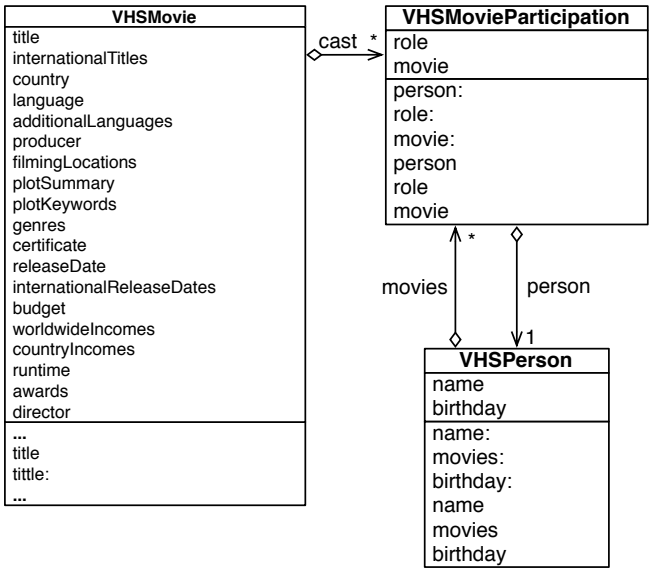

Figure 1. Example model

This example exposes a large amount of variables. Many of these variables will be used only on specific movies. For example, the variable income will be filled up once the movie is released. Meanwhile it will point to nil. The variables additionalLanguages, awards, internationalReleases, etc, will all point to empty collections, up to the moment that they get filled. Each of these pointed variables is susceptible to be used, as much as they may remain unused by the rest of the lifecycle of the object.

This amount of empty information, may not be a real problem for a small dataset. But in a large dataset, as IMDB, with more than 4,734,693 titles[7], this waste of memory become more obvious.

With having only one empty collection per title, we would have at least 4,734,693 idle collections. With having only one variable per object unused, we would have at least 4,734,693 idle slots. Each slot takes a word size for storing information, therefore, around 18mb wasted in a 32-bit architecture, 36mb on a 64-bit architecture.

[6]IMDb, also known as Internet Movie Database, is an online database of information related to world films, television programs, home videos and video games, and internet streams, including cast, production crew and personnel biographies, plot summaries, trivia, and fan reviews and ratings. An additional fan feature, message boards, was abandoned in February 2017. Originally a fan-operated website, the database is owned and operated by IMDb.com, Inc., a subsidiary of Amazon.

[7]01/06/2018, https://www.imdb.com/pressroom/stats/

Even though nowadays 18mb or 36mb is not a significant amount of wasted memory, this is a really small example. For illustrate the problem in industrial dimensions, we show some industrial case analysis.

2.2 Moose

Moose is a free and open source platform for software and data analysis built in Pharo. Moose offers multiple services ranging from importing and parsing data, to modelling, to measuring, querying, mining, and to building interactive and visual analysis tools.[8] Moose is used largely for software analysis. The software analysis is specifically supported through the FAMIX family of meta-models. The core of FAMIX is a language-independent meta-model that is similar to UML but it is focused on analysis. Furthermore, it provides a rich interface for querying models. [9]

A FAMIX model represents software entities as instances, each instance includes fields that holds all the properties and relationships of the entity. For example, as shown in Figure 2, the representation of a class (FmxNGClass) has 27 instance variables. These instance variables point to values, other single entities or to collections of entities and values. The attribute name is an example of an instance variable pointing as a value, as it points to a String. Examples of instance variables pointing to collections of entities are methods and superInheritance[10].

The described problem arises in FAMIX models because not all of the instance variables have values. For example, the instance variable usedTraits or subInheritances point to empty if the modelled language or entity do not use traits or have subclasses.

This problem affects industrial users of Moose. Table 1 presents the memory waste produced when an industrial application is modelled to be analysed by Moose. As the table shows that 35 % of the instance variables are unused (they point to nil), and there are five million empty collections.

Total instance variables	26,852,653
Unused instance variables	9,458,715
Empty collections	5,023,508

Table 1. Memory Waste in Industrial Application modelled as a FAMIX Model

To analyse more deeply this problem, we work with models of open-source projects. Table 2 presents a summary of the information of the FAMIX model for the full Hadoop repository (it includes Hadoop itself and all the related tools that are bundled with Hadoop). As the table shown, the

[8] http://www.moosetechnology.org/

[9] https://en.wikipedia.org/wiki/Moose_(analysis)

[10] It is a collection because FAMIX support to model multiple inheritance

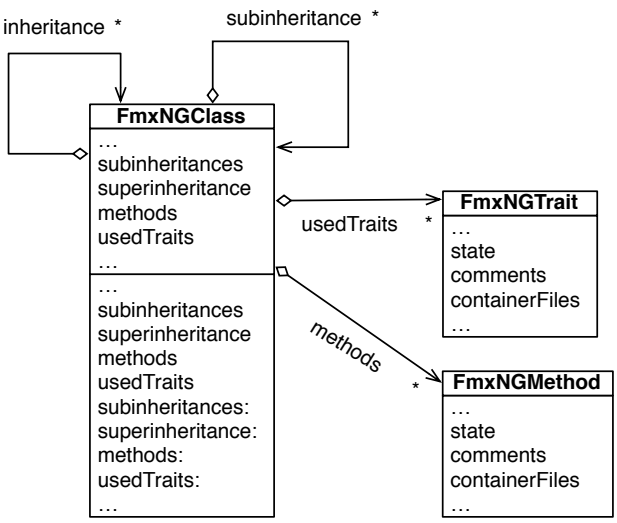

Figure 2. Simple moose entity model

amount of wasted memory takes more significance with bigger model. In comparison, this model includes 48 million entities.

	Size (MB)	% Waste
Total Instances	2,387.7	
Unused Instance Variables	497	20%
Empty Collections	1,227	51%

Table 2. Memory Waste in Industrial Application modelled as a FAMIX Model

Different solutions exit to solve memory waste. However, applying them to an already implemented solution requires the modification of a large amount of code base. An optimal solution should be as least intrusive as possible minimizing the number of modifications and required boilerplate code. It should be noted that as with any other optimization, it is difficult to detect the bottlenecks without having a running implementation.

In the following section, we present existing *ad-hoc* solutions and the problems that arise when applied them to an existing application. In Section 3, we present our proposed solution and how it solves the presented problem. Finally, Section 8 presents other techniques to solve this problem.

2.3 Ad hoc Solutions

There are some ways for solving this problem with more quotidian ad hoc techniques, and as we will show, these techniques bring unwanted side effects.

Lazy initialization. Lazy initialization of instance variables pointing to empty collections reduce the memory usage. This

solution seems simple as it only needs to redefine the getter method to the variable as shown in Listing 1. This implementation returns a new OrderedCollection when the instance variable is nil.

```
VHSMovie >> cast
  ^ cast ifNil: [ OrderedCollection new]
```

Listing 1. Lazy initialized getter for an instance variable holding a collection.

However, this naïve solution presents three problems. First, it requires a special method to modify the collection, as the OrderedCollection returned by the getter method is not referenced when the instance variable is nil. So, adding an element to an empty collection requires the use of a special message. Listing 2. Second, all accesses to the instance variable should be rewritten to use the getter and mutator messages. Last, if any client instance requires to store a reference to the collection, or this is passed as a parameter and later modified, the modified collection is not referenced. This requires a getter method that materializes the collection on access (Listing 3). The different clients and even the methods in the same class should use the correct messages depending on what they need to do with the collection.

```
VHSMovie >> addCast: anElement
  cast ifNil: [ cast := OrderedCollection new].
  cast add: anElement
```

Listing 2. Required collection mutator method

```
VHSMovie >> materializedCast
  ^ cast ifNil: [ cast := OrderedCollection new]
```

Listing 3. Lazy initialized getter for an instance variable holding a collection.

Applying such partial solution is possible using automated code rewriting. However, it requires the repetition of boilerplate code for each of the instance variables holding a collection. For each of the instance variables, it is required to have a proper getter method and a mutator for the collection. Second, it requires to analyse all the users of the modified classes to check that they are correctly using the proper messages, *i.e.,* checking that they are not modifying the collection returned by the lazily initialized getter, or keeping a reference to the collection.

Dictionary of Instance Variables. Using a dictionary to store the instance variables reduces the impact of having many uninitialized instance variables. This solution requires to define the class with only a single field, this field holds a reference to a dictionary, and all the instance variables that were previously defined in the class are stored in this dictionary. Listing 4 shows the implementation of the accessors for this model.

This solution requires the modification or creation of accessor methods for all the instance variables in the class. These accessors are boilerplate code across the modified classes.

```
"Definition of the class, with only one instance variable"
Object subclass: #VHSMovie
  instanceVariables: 'values'
  classVariableNames: ''
  package: 'SlottyExample-plain'

VHSMovie >> initialize
  super initialize.
  values := Dictionary new

VHSMovie >> cast
  ^ values at: #cast

VHSMovie >> cast: aValue
  ^ values at: #cast put: aValue
```

Listing 4. Accessors Methods using a Dictionary of Instance Variables

Sharing and privatizing. Sharing a default value among a set of instances is possible. The problem is then how to turn such shared value into a private instance specific one. Indeed, advanced smalltalk programmers may think that using become: [9] offers a solution. This is not the case because become: swaps pointers. Therefore all the objects sharing the value would share another value. There is no ready to use solution for such a problem.

Sharing by default. It is possible to share a value by default among a set of instances and to make it specific to an instance on demand. This requires to use a class variable and an instance variables. Listing 5 shows a possible ad hoc implementation.

```
Object subclass A
  instance: col
  class: Col

A class >> initialize
  Col := OrderedCollection new.

A >> initialize
  super initialize
  col := Col

A >> prepareChangingCol
  col := Col copy

A >> changingCol
  col add: 'something'
```

Listing 5. Ad hoc sharing of a default value.

To work such approach forces the programmer to invoke prepareChangingCol before any side effect on the share collection. In addition, the class variable should only be used during the instance creation. In addition, it forces to have both a class variables and an instance variables.

3 Slots to the rescue

Slots provide an extension point to customize the behaviour of the instance variables in a given class. They are a first-class representation of instance variables. Classes in Pharo includes a list of slots.

Every time an instance variable is accessed, this access is performed using the slot. Also when a method is compiled the slot is responsible to emit the set of bytecode to execute this operation. Slots allow us to modify the behaviour when an instance variable is read or written. As slots are first-class objects, they can be easily implemented and extended by the application developers. This flexibility allows the application developer to change behaviour that was previously reserved to language developers [13].

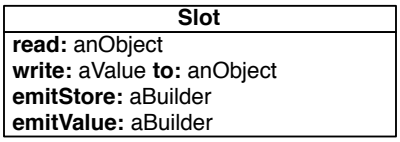

Figure 3. Slot API to implement by the new extensions

Figure 3 is the basic API implemented by slots:

- **read:** anObject reads the represented instance variable from anObject.
- **write:** aValue **to:**anObject writes aValue into the represented instance variable of the anObject.
- **emitStore:** aBuilder uses the method builder passed as parameter to generate the bytecode to be used when writing into an instance variable.
- **emitValue:** aBuilder uses the method builder passed as parameter to generate the bytecode in to be used when reading an instance variable.

The Slot class provides a default implementation of emitStore: and emitValue: that uses the read: and write:to: messages.

Using slots, we modify how the instance variables are stored in it. To minimize memory waste we propose two slots: LazyInitializationSlot and NilAwareSlot. LazyInitializationSlot implements a transparent lazy initialization of the instance variable and sharing of the default value. NilAwareSlot transparently implements an instance variable that does not occupy space if its nil. These slots then are used in the classes to transparently implement memory optimization.

3.1 Lazy Initialized Slot

LazyInitializationSlot implements transparently the following behaviour:

- Return a default value when the instance variable is nil.
- Share the default value between all the instances of the class using this slot.
- Detect when the shared value is modified and update all the references to it.
- Store the value only when it is neither nil nor the shared value.

This slot is used to implement a lazily initialized instance variable that shares the same default value for all the instances.

Following the example already used, VHSMovie class has the cast instance variable. This variable is initialized with an empty OrderedCollection. In the naïve implementation, all the instances of VHSMovie have their own OrderedCollection. An improvement is to share the empty collection, and only create a new one when the VHSMovie is modified to add a new element in the cast collection. LazyInitializationSlot implements this behaviour. Figure 4 shows the implemented slot.

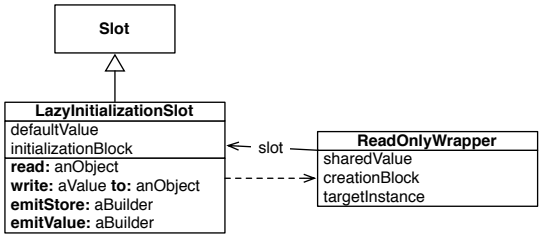

Figure 4. The LazyInitializationSlot slot

To perform this behaviour, when the cast instance variable is read the slot returns a ReadOnlyWrapper wrapping the default value.

This wrapper acts as a proxy, redirecting the messages it receives to the default value [8]. Once the wrapper receives a message that modifies the shared value, it intercepts the message and it creates a new instance using the creationBlock in the slot. When the new value is created the instance variable of the targetInstance is updated and all the references to the wrapper are updated to point to the new value of the instance variable. After the new value is created, the message that started the instance creation process is dispatched to the new instance.

If the slot is used to write a value in the instance variable, it also replaces all the references to the wrappers with the default value effectively removing the wrapper.

Figure 5 presents an example where two instances of VHSMovie have the default value of the cast instance variable. Both instances have nil as value in their instance variables. The client objects that access to the cast instance variable have a reference to ReadOnlyWrapper instances. Each client object has a reference to its ReadOnlyWrapper instance. If nobody keeps a reference to the value of cast, there is no ReadOnlyWrapper. It is only kept while it is needed to update the reference.

Figure 6 shows the example when the cast collection of the VHSMovie *aliens* is modified adding a new actor. To perform this, all the references to the ReadOnlyWrapper of aliens are

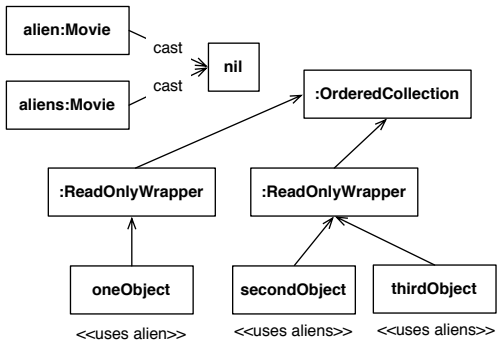

Figure 5. When the VHSMovie instances have empty collections

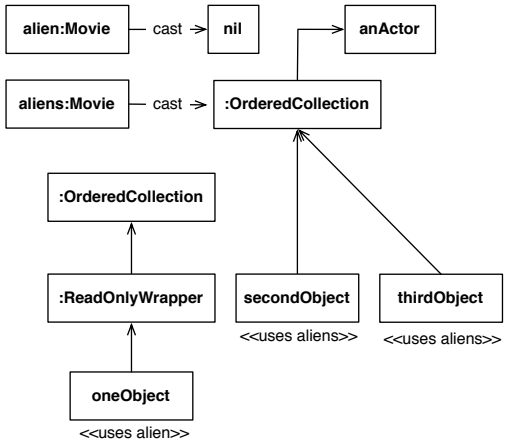

Figure 6. When the collection is modified the clients are updated

replaced with a reference to a new OrderedCollection and the Actor object is added to this newly created collection.

The ReadOnlyWrapper handling is needed to keep the clients updated, this is only needed if the client objects keep references to the original value of cast.

3.2 Nil Aware Slot

NilAwareSlot provides a transparent way of compacting instances with nil values. Compact instances only occupy the space that is required by their used instance variables.

The instance of the classes using NilAwareSlot have two instance variables. The first one holds an integer that is used as bit-mask marking the used variables; and the second is an array with the values that are not nil.

Figure 7 shows an example of how the information is stored. The logical model represents the traditional instance layout for VHSMovie instances, and the physical model shows how the information is stored in the memory using our solution.

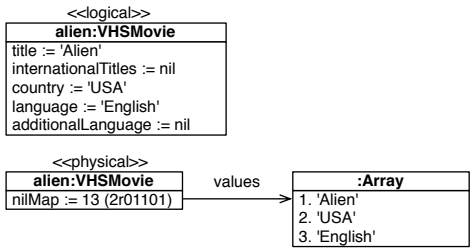

Figure 7. Comparison between the logical and physical instances.

The values that are not nil are stored in the associated array. So, the array does not contain any nil value. The array is compacted every time a value is removed to improve memory use, and it is enlarged when an instance variable got a non nil value. If the instance variable is nil, the bit-mask has a 0 for the slot index. All the NilAwareSlot have a non-conflicting index. Otherwise, if the instance variable has a value a 1 is stored in the bit-mask.

When the slot is used to access the instance variable, the bit-mask is read to check if the instance variable is nil or not. If the variable is nil, the read: method returns nil. If the variable is not nil, an index inside the array is calculated. This index is then used to retrieve the instance variable value from the associated array.

Each instance variable with a non nil value has a position in the array. This position is used to access the value. It is calculated from the slot index and the bit-mask. As the array does not contain nil values, the nil slots are skipped during the calculation of the index.

For example, if the bit-mask is 9 (2r1001)[11], the first and fourth instance variables are used. So the index[12] for access to the fourth instance variable is 2. As it is the smallest not used slot. If a non nil value for the third instance variable is stored, the array index for the third will be 2 and for the fourth will be 3; as the bit-mask new value is 13 (2r1101).

3.3 Combining the Slots

To completely solve the problem, we need the ability to use the two slots at the same time in the same class. We require to have lazily initialized slots that store their values in nil aware slots. So, we have added to the LazyInitializationSlot the base slot to use as storage. Then the LazyInitializationSlot delegates to the base slot the final read and write of the instance variable in the instance. Figure 8 shows the extended LazyInitializationSlot.

If the LazyInitializationSlot is used alone, the base slot it uses is BaseSlot. BaseSlot is a regular instance variable slot, however it is hidden from the definition. It stores the values in the instance between adding any special behaviour.

[11] Smalltalk base-2 number notation.

[12] In Smalltalk all the indexes are one-based.

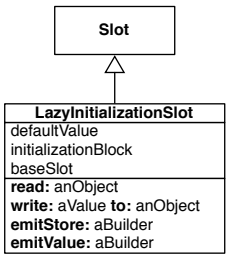

Figure 8. The LazyInitializationSlot extended to support the NilAwareSlot

When the LazyInitializationSlot is used with the NilAwareSlot, the LazyInitializationSlot uses a NilAwareSlot as base slot. So, the actual value of the lazy initialized value is stored using the technique implemented by the NilAwareSlot.

3.4 Using the Slots

Once the slots are implemented, we have to use them in the classes of the application. Using the slots is as easy of using any slot. The application includes the following definition of the VHSPerson class:

```
Object subclass: #VHSPerson
  instanceVariableNames: 'name movies birthday'
  classVariableNames: ''
  package: 'SlottyExample-plain'
```

We want to make all the slots NilAware, so the new definition is:

```
Object subclass: #VHSPerson
  slots: {
    #name => NilAwareSlot.
    #movies => NilAwareSlot.
    #birthday => NilAwareSlot }
classVariables: { }
package: 'SlottyExample-plain'
```

To use LazyInitializationSlot we have to provide some parameters in its creation. The slot requires to know: (1) a block to create the default value, (2) a block to create a value to be used when trying to mutate the default value, and (3) the class of the base slot. In this case, we will use the NilAwareSlot as the base slot. If we use LazyInitializationSlot to provide a default value for the movies instance variable, the class definition is the following.

```
Object subclass: #VHSPerson
  slots: {
    #name => NilAwareSlot.
    #movies => (LazyInitializedSlot
      default: [ OrderedCollection new ]
      initializationBlock: [ OrderedCollection new ]
      baseSlotClass: NilAwareSlot).
    #birthday => NilAwareSlot }
classVariables: { }
package: 'SlottyExample-plain'
```

As shown in the listings, the modification is straightforward and it is simpler than using an ad hoc solution.

4 Implementation

To implement the slots minimizing the overhead we have implemented them using different techniques existing in Pharo.

The LazyInitializationSlot slot requires to detect the methods that have side-effect on the shared instance. To do so, the slot uses the *Write Barrier* present in Pharo [1]. This write barrier permits us to detect the messages that try to change the shared default value.

The update of the references to ReadOnlyWrapper is performed using the *Efficient Forwarders* [9]. This technique present in the Pharo VM allows us to update the references without the need to scan the whole heap, it only requires marking the replaced instance. Later on, the references are updated only when they are used.

To be able to share the ReadOnlyWrapper, we use a weak structure. If the ReadOnlyWrapper instances are not referenced, they will be reclaimed by the garbage collector. Without leaking memory.

5 Results of Applying the solution

We validated our solution updating the FAMIX implementation to use the proposed slots. To use our solution the definition of the instance variables should be modified to use the new slots.

We did not modify all the instance variables, we started with an analysis of the memory usage. To minimize the required changes we used a tool called MemoryStats[13]. This tool allows us to identify the instances that have more unused instance variables and empty collections. After the analysis, we just performed the following changes.

- Modify MooseEntity to have NilAwareSlot s. This class is the superclass of all the classes in the FAMIX model.
- All the subinstances of MooseEntity includes an instance of MooseMemoryEfficientStore to store optional attributes and properties of the instance. The instances of MooseMemoryEfficientStore have two collections that are mostly empty, so we changed them to be lazily initialized.
- FAMIX models use two slots to represent relations between entities (FMOne and FMMAny), we modified them to be subclasses of NilAwareSlot.
- Many-to-one and Many-to-many relations are modelled using the class FMMultivalueLink. We modified this class to use lazily initialized collections.

Doing these reduced changes, we achieved our objective of reducing the memory footprint of FAMIX models, and also we proved that our solution only requires small changes

[13] https://github.com/tesonep/memoryStats

	Original				Using Our Solution				
	Instances	*Empty Collections*	*Nil Variables*	*Memory (MB)*	*Instances*	*Empty Collections*	*Nil Variables*	*Memory (MB)*	*Saving*
ArgoUML	4,962,653	2,352,843	14,966,375	180.5	2,826,212	921,420	3,819,773	106.4	41%
HSQLDB	5,171,400	2,309,958	13,292,098	192.7	3,081,508	776,616	3,862,947	125,9	34%
Hadoop	48,114,127	21,221,890	124,308,862	2,387.7	29,209,714	7,218,219	35,015,147	1,768,1	26%

Table 3. Results of using the solution

to gain a big improvement. We have performed the analysis and the changes in 6 hours of work. We reduced the memory footprint of FAMIX models in an average of 30 %.

To validate the impact of our solution, we loaded the models of 3 open source applications. We selected two medium applications (ArgoUML and HSQLDB) and one big application (Hadoop). For ArgoUML[14], we analysed the version 0.34. For HSQLDB[15], we generated the model from the version 2.4.1, and for Hadoop we used version 3.1.0-RC1. Table 3 shows more detailed results of the three analysed FAMIX models.

6 Benefit of using our solution

The presented solution performs a memory saving, although it includes a level an overhead in execution time. However, we in this section we present that the overhead is less or equals to an ad hoc solution.

To perform this benchmark we implemented the movies example in 3 flavours:

- A *Baseline* implementation that does not use any optimization in memory. It is the basic object-oriented design. All the instance variables are initialized in nil or with an empty ordered collection. As said, this baseline establishes the minimum execution time and the maximum memory usage.
- An *ad hoc* implementation. This implementation includes lazy initialization and dictionary backed fields as shown in Section 2.3. The implemented ad hoc solution presents a possible solution to this problem.
- Our solution implementation. This implementation uses the newly proposed set of slots.

For these benchmarks, we will be using a subclass of LazyInitializedSlot that does not perform the update of the references or the use of ReadOnlyWrapper, as this functionality is not implemented in the ad hoc solution.

All the source code needed to reproduce the benchmarks is available in Github[16]. Also in this repository, there are instructions to reproduce the benchmarks.

All the benchmarks were executed in a machine running OS X 10.12.6, with a 2,6 GHz Intel Core i7 processor and 8 GB 1600 MHz DDR3 of memory. We used Pharo 7-Alpha (build 915) 32-bits, using the Production Spur VM 201805090836.

[14] http://argouml.tigris.org
[15] http://hsqldb.org/
[16] https://github.com/tesonep/slotty

6.1 First Scenario: Single instance creation and modification

In the first scenario, we tested the speed of creating new instances, setting a nil valued instance variable to a value and adding an element to an empty collection. Listing 6 shows the code executed in the benchmark.

```
anInstance := VHSMovie new
  title: 'aTitleToSet';
  title;
  yourself.
anInstance addInternationalReleaseDates: (Date today)
```

Listing 6. First Scenario: Single instance creation and modification

This benchmark stresses the creation of instances and setting values to nil instance variables and lazy initialization of the collections. This analysis is important as our solution requires special execution when an instance variable is modified from nil to another value. Also, the lazy initialization of a collection requires special treatment. So, this scenario stresses our solution and the ad hoc solution.

Table 4 shows the results of this benchmark. As the results show, our solution provides an overhead comparable with the ad hoc solution, with the benefit of having less boilerplate code.

Implementation	Executions per second	Overhead
Baseline	245,257	0 %
Ad Hoc	188,623	30 %
Our Solution	191,873	28 %

Table 4. Results of first Scenario: Single instance creation and modification

6.2 Second Scenario: Memory Footprint

In this scenario we tested the memory impact of our solution, comparing it with the ad hoc solution and the baseline implementation. We created a million of instances and compared the memory impact of the three implementations. This is important to show that our solution presents a better memory

footprint and a comparable processing impact of the ad hoc solution. Listing 7 shows the benchmarked code.

```
instances := OrderedCollection new.
1 to: 1000000 do: [ :i | anInstance:= VHSMovie new
  title: 'aaa';
  title;
  yourself.
anInstance internationalReleaseDates add: (Date today).
instances add: x ]
```

Listing 7. Second Scenario: Memory Footprint

Table 5 shows the results of this benchmark. The results are comparable with the ones in our validation. Our solution provides a better memory footprint compared with the ad hoc solution.

Implementation	**Memory Size** (in MB)	**Savings**
Baseline	612	0 %
Ad Hoc	222	63 %
Our Solution	169	72.5 %

Table 5. Results of second Scenario: Memory Footprint

7 Discussion

From the benchmark, we observe that our proposed solution presents an execution time overhead compared with the baseline solution, although also we observer that the memory footprint has been reduced in an important amount. We have decided to implement a solution that gets the minimum memory footprint.

In the benchmarks that we presented, the ad hoc solution seems as an equivalent one. However, our solution presents a single point of improvement, modifying the implementation of the slots only requires to modify a class. Also, it does not require boilerplate code distributed in all the affected classes.

Our proposed solution allows us to implement other strategies, to balance the memory footprint and the execution time penalty. Having different implementations and comparing them is a subject of future work.

Our solution is designed to be applied to an application that is already built and present memory footprint problems. It requires to analyse the application and detect the bottlenecks where our solution provides better results. It is true that not all the applications will present the same reduction of memory footprint, even more, there are scenarios where our solution does not provide any optimization. However, as shown in the paper, it is applicable and provides a considerable optimization if the conditions presented in Section 1 and Section 2 are met. An example of a place where the optimization is not possible it is when the instances have less than 2 instance variables, applying NilAwareSlots to them does not provide any optimization.

Our benchmark only centred on showing that the use of our solution does not produce an execution penalty that proves it cannot be used. The execution penalty exists, although it showed that it does not affect the normal loading and use of the FAMIX models.

Also, we are aware that there are more performance differences between the ad hoc solutions that we are not correctly showing. As future work, we have the idea of comparing our solution with different sizes of objects, different distributions of nil instance variables and empty collections.

In the validation, we could have continued optimizing the application, although we decided to show which was the impact of limited changes in the correct spots.

The set of analysis techniques and the tools used are outside the scope of this work. Even more, the analysis of this tools and techniques is a subject of future work. This future work may include the automation of the suggested changes.

8 Related Works

The Pharo default image includes a set of slots implementing special behaviour, examples of them are BooleanSlot which stores boolean fields in a single value, and PropertySlot which uses a Dictionary to store the instance variables. However, these implementations are not used extensively in applications, and they are not implemented to minimize memory footprint as they are only a reference implementation.

Schmidt [10] describes a way of implementing dynamic slots in an object-oriented programming language. His motivation and the implementation techniques are similar, although its solution requires modification to the runtime system. He validates its solution with a limited example showing an improvement in the memory usage. He implements its slots as a linked list. Requiring a linear search when modifying a slot. We implemented the lookup using cached indexes to avoid this linear search.

Hölzle and Ungar [7] implemented a way of extending instances with new instance variables, but this technique requires support from the Virtual Machine. Our solution does not require support from the Virtual Machine as everything is resolved in the image.

Serrano [11] proposes a way of extending an instance by changing its class to a wider subclass. His solution requires to create a subclass of the original class. Also, it is not intended to be dynamically adapted while the instance variable is used or not. Our solution allows us to shrink the instance variable as soon as an instance variable is not used any more, without caring where in the hierarchy the instance variable is defined.

9 Conclusion

In this paper, we took a daily problem of the industrial users of Moose. After analysing it, we discover that this problem is

not only present in Moose, but Moose is an excellent subject to resolve it.

We presented a solution using Slots to improve the memory footprint of an already developed application minimizing the set of required changes.

Our solution presents an excellent result in reducing the memory footprint, although it presents an execution penalty. Even though this execution penalty does not affect the normal use of the application. Moreover, it allows us to load FAMIX models that previously were impossible because of its size.

As a future work, we plan to continue analysing different strategies to achieve similar solutions and different implementations of slots that improve the existing applications minimizing the need for boilerplate code.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. Also, we will like to thank the collaboration of Pavel Krivanek, Guillaume Larcheveque and Benoit Verhaeghe who provide us with MSE models and the descriptions of their experiences using Moose.

References

[1] Clément Béra. 2016. A low Overhead Per Object Write Barrier for the Cog VM. In *International Workshop on Smalltalk Technologies IWST'16*. Prague, Czech Republic. https://doi.org/10.1145/2991041.2991063

[2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. http://rmod.inria.fr/archives/books/Blac09a-PBE1-2013-07-29.pdf

[3] Paul Butterworth, Allen Otis, and Jacob Stein. 1991. The GemStone object database management system. *Commun. ACM* 34, 10 (1991), 64–77. https://doi.org/10.1145/125223.125254

[4] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. 2001. *FAMIX 2.1 — The FAMOOS Information Exchange Model.* Technical Report. University of Bern.

[5] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. 2011. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family.* Technical Report. RMod – INRIA Lille-Nord Europe. http://rmod.inria.fr/archives/reports/Duca11c-Cutter-deliverable22-MSE-FAMIX30.pdf

[6] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. 2005. Moose: a Collaborative and Extensible Reengineering Environment. In *Tools for Software Maintenance and Reengineering*. Franco Angeli, Milano, 55–71. http://scg.unibe.ch/archive/papers/Duca05aMooseBookChapter.pdf

[7] Urs Hölzle and David Ungar. 1994. A Third-generation SELF Implementation: Reconciling Responsiveness with Performance. *SIGPLAN Not.* 29, 10 (Oct. 1994), 229–243. https://doi.org/10.1145/191081.191116

[8] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. 2011. Efficient Proxies in Smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'11)*. Edinburgh, Scotland. https://doi.org/10.1145/2166929.2166937

[9] Eliot Miranda and Clément Béra. 2015. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In *International Symposium on Memory Management (ISMM '15)*. Portland, United States, 93–104. https://doi.org/10.1145/2754169.2754186

[10] R. W. Schmidt. 1997. Dynamically Extensible Objects in a Class-Based Language. In *Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems (TOOLS '97)*. IEEE Computer Society, Washington, DC, USA, 294–. http://dl.acm.org/citation.cfm?id=832250.832649

[11] Manuel Serrano. 1999. Wide classes. In *Proceedings ECOOP '99 (LNCS)*, R. Guerraoui (Ed.), Vol. 1628. Springer-Verlag, Lisbon, Portugal, 391–415. http://www.ifs.uni-linz.ac.at/~ecoop/cd/papers/1628/16280391.pdf

[12] Dave Thomas and Kent Johnson. 1988. Orwell — A Configuration Management System for Team Programming. In *Proceedings of the Object-Oriented Programming, Systems, Languages & Applications, ACM SIGPLAN Notices (OOPSLA'88)*, Vol. 23. 135–141.

[13] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. 2011. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 959–972. https://doi.org/10.1145/2048066.2048138

[14] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10)*. ACM, New York, NY, USA, Article 42, 6 pages. https://doi.org/10.1145/1900008.1900067