Language-independent Storage Strategies for Tracing-JIT-based Virtual Machines

Tobias Pape Tim Felgentreff
Robert Hirschfeld
Hasso Plattner Institute,
University of Potsdam, Germany
{firstname.lastname}@hpi.uni-potsdam.de

Anton Gulenko
Technische Universität Berlin,
Germany
anton.gulenko@tu-berlin.de

Carl Friedrich Bolz
King's College, London,
United Kingdom
cfbolz@gmx.de

Abstract

Storage strategies have been proposed as a run-time optimization for the PyPy Python implementation and have shown promising results for optimizing execution speed and memory requirements. However, it remained unclear whether the approach works equally well in other dynamic languages. Furthermore, while PyPy is based on RPython, a language to write VMs with reusable components such as a tracing just-in-time compiler and garbage collection, the strategies design itself was not generalized to be reusable across languages implemented using that same toolchain.

In this paper, we present a general design and implementation for storage strategies and show how they can be reused across different RPython-based languages. We evaluate the performance of our implementation for RSqueak, an RPython-based VM for Squeak/Smalltalk and show that storage strategies may indeed offer performance benefits for certain workloads in other dynamic programming languages. We furthermore evaluate the generality of our implementation by applying it to Topaz, a Ruby VM, and Pycket, a Racket implementation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments, code generation, incremental compilers, interpreters

Keywords Implementation, collection types, memory optimization, dynamic typing

DLS'15, October 27, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3690-1/15/10...
http://dx.doi.org/10.1145/2816707.2816716

1. Introduction

Virtual Machines for dynamic programming languages depend on run-time optimizations to achieve appropriate performance. One optimization that has been shown to work well in the PyPy VM for Python is called *storage strategies* [3].

Storage strategies optimize collection data types by transparently inlining element data instead of holding references to full objects on the heap. This avoids unnecessary allocations, which conserves memory and can have major benefits for the performance of the just-in-time compiled code. Avoiding allocations on the heap is a major optimization of many JIT compilers, and is usually done by determining that an object does not escape a compiled loop. In contrast, storage strategies are applicable to any objects, not only those that do not escape the loop that is being optimized by the JIT.

Certain conditions have to be met in order to apply storage strategies successfully. These conditions are hard to detect in a general way. RPython, the framework that powers the PyPy Python implementation, provides many optimizations that are automatically applied to all languages that are build using RPython, such as garbage collection and a tracing JIT compiler. However, even with the information available to the tracing JIT compiler, storage strategies are hard to apply in a general way, and have so far been implemented for PyPy only [3], but are not automatically provided to all languages built with RPython.

Storage strategies have been shown to be beneficial for the Python programming language [3]. Since Python offers one generic list data type, optimizing it can have a big impact on program performance. The initial paper on strategies suggested that storage strategies might be applicable to other dynamic languages, as well. We set out to test this hypothesis, and to provide a generic framework to use strategies in a variety of dynamic languages such as Smalltalk, Racket, or Ruby, which have other kinds of collections in their standard libraries. In this paper we investigate how we can make the storage strategy approach generic and gain its performance benefits in other contexts than the Python programming lan-

guage. Towards that goal, we make the following contributions:

- We present a generic, language-independent framework to include the storage strategies optimization which in any VM implemented using the RPython toolchain.
- We show that storage strategies can be beneficial for certain workloads in other dynamic languages.
- To determine how reusable our implementation is, we evaluate the application of storage strategies to three RPython-based virtual machines: RSqueak, a VM for Squeak/Smalltalk; Topaz, a Ruby VM; and Pycket, a Racket implementation.

2. Background: Storage Strategies in PyPy

The PyPy virtual machine is built using RPython, a restricted subset of Python that is amendable to static analysis and that can automatically generate a tracing just-in-time compiler. The most important optimization implemented by RPython's tracing JIT is an aggressive escape analysis [6]. When a loop is detected it is *traced*, that is, all operations in the loop are recorded. The recorded operations are then simplified using a form of partial evaluation [2]. The original trace is traversed and object allocation are replaced with *static objects* which are allocated on the stack. Only when an object is found to escape the trace it is actually allocated on the heap. This optimization is safe because the optimizer can prove that static objects don't affect the program state after exiting the loop.

The drawback of static objects is a rather complex deoptimization. For each assumption the JIT compiler made during optimization, a guard is left in the compiled code. An optimized JIT trace must be left when such a guard check fails. In this case the escape analysis performed as basis for the static object optimization does not hold anymore. Static objects allocated in the last loop iteration have to be allocated on the heap, and their contents have to be copied from the stack to the newly allocated objects. To do this, the JIT has to maintain information about the stack layout while executing an optimized trace; using a stack map, the stack can be walked in order to create non-static versions of the objects.

Since RPython's meta tracing JIT creates traces of VM operations, not only language level objects can be turned into static objects, but also VM level objects. Just like other optimizations in the RPython toolchain, this leads to cleaner code; short living objects can substantially increase the expressiveness of VM code.

Storage Strategies Usually, collections in dynamic languages are represented by allocating an array and filling it with pointers that point to the actual elements of the collection. This is because in a dynamic language, or even any language with polymorphism, objects of different types can be added to the same collection. Using an array of pointers is the generic way to represent such a dynamic collection.

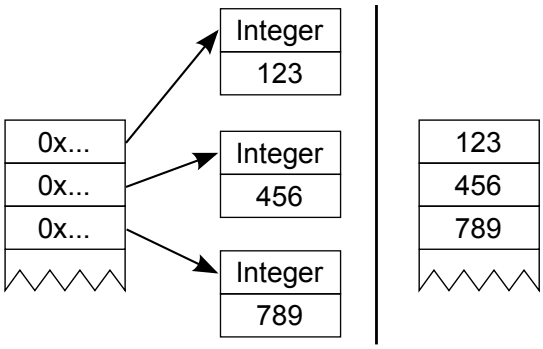

Figure 1. Example for a storage strategy. Left side: generic collection holding only boxed integer objects. Right size: Optimized collection, resulting in less object allocations and memory usage.

However, it also introduces an indirection between the collection and the actual elements. To access any value from the collection, the VM must first fetch the pointer address and dereference it, before fetching the actual value. This additional memory access is a big overhead compared to a C array of pure integer values, which can be traversed at full speed.

Another allocation removal technique called storage strategies, or *strategies* for short, can optimize dynamic collections [3]. Storage strategies are implemented on VM level and are based on the idea that the VM has multiple possible ways of representing collections internally. Based on the user program and the operations performed on a collection, the VM can switch between multiple strategies and choose the most efficient one. To do so, the collection contains a reference to the currently used strategy object, and all important operations performed on the collection are delegated to that object. Ideally the strategy will not be switched very often, so the second important idea behind storage strategies is that user programs often utilize collections in a predictable way.

For example, collections are often used in a homogeneous way. If the first few elements appended to a Python list are integers, then it is likely that the user program will keep storing only integer values in that list. This is enough information to optimize the list. In this case, the VM could allocate an array for integer values and store the values directly, without allocating box objects to represent these integers. Figure 1 illustrates this example by showing two possible representations for the same collection: the collection on the left side contains boxed integer objects, while the collection on the right side contains pure integer values.

In order to optimize the internal representation of a collection, the VM must make assumptions about the usage pattern of that collection. It cannot wait until a collection has been used for a while before optimizing it, otherwise it might end up not optimizing short living or small collections. Optimizing short living collections can be very important, for example when they are allocated frequently in a tight loop.

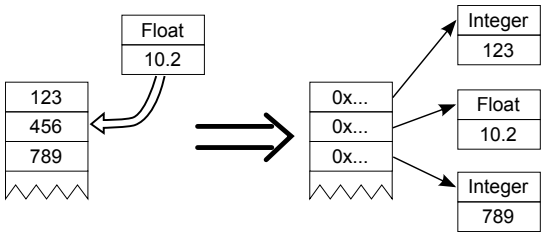

Figure 2. Deoptimization of an optimized integer collection. A general storage array has to be created, and Integer objects have to be allocated

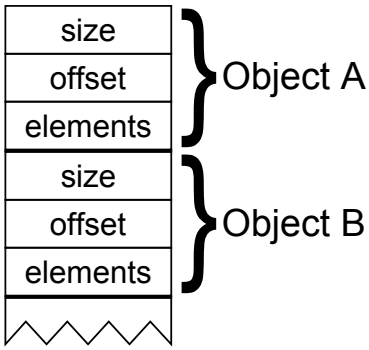

Figure 3. Inlining storage for composite objects with 3 fields

In order to optimize as many collections as possible, the VM must predict the usage pattern based on the very first operations performed on a collection.

Making predictions about usage patterns is hard and the success can vary depending on the situation. Therefore, the VM should make assumptions based on meaningful heuristics, which can be obtained by examining existing systems and deriving typical usage patterns. But even with these preconditions there will be cases where the VMs assumptions turn out to be wrong. In such cases, the VM must uphold the semantics of the programming language and perform a *deoptimization*. Most of the time this means allocating a new, more general collection and copying all elements from the old collection into the new one. Figure 2 shows an example of this procedure. The storage on the left is optimized to hold integer values, but the user program wants to add a float value to the list. The VM has to allocate a new, more general storage array which is able to hold generic objects. In this case, the deoptimization could have been avoided if the VM had implemented a storage strategy to hold both integer and float values. The deoptimization procedure must be tuned for highest performance, and should take place as rarely as possible.

Homogeneous Collections As indicated by the examples given, the most important optimization target of storage strategies are homogeneous collections. A collection is homogeneous when it only contains elements of the same type. Since the size of the datatype is known, the VM can allocate an array big enough to hold the data of many such objects in consecutive memory. The benefit of this optimization is two-fold: the VM can save the memory for the objects, and accessing the objects becomes faster due to the omitted indirection between collection and elements.

This optimization is not limited to numeric datatypes like integers. Even objects with multiple fields can be represented that way, as illustrated in figure 3. The fields of each objects just have to be placed sequentially in memory. Accessing any field in any object can happen without additional memory access, simply based on constant-time pointer arithmetics.

3. Language Independent Storage Strategies

In this section we present the design of our library for storage strategies called rstrategies. First we introduce the principles this library design is based on, then we describe different parts of the library and how a VM developer can utilize the library to create specific collection implementations.

In order to be reusable in the context of different programming languages, rstrategies must expose its functionality in a very flexible way. We use the concept of mixins [4, 5] to provide the required level of flexibility while allowing the VM to extend the provided functionality when needed.

In the context of RPython, any regular class can be used as a mixin. The programmer must simply include a `import_from_mixin(MixinClass)` clause into the body of the instantiating class. All attributes and methods will then be copied over from `MixinClass` to the target class, including all inherited methods. A mixin is not intended to be used on its own. It can require other mixins and fields or methods provided by the instantiating class itself. This way the code inside a mixin can be written in an abstract way to promote modularity.

rstrategies supports two types of collections: variable sized and fixed sized lists. These two abstract types of collections cover the needs of primitive collections for most programming languages. The reasons behind this design decision are further explained in section 6.

The rstrategies library provides a hierarchy of mixins that can be combined into actual collection classes. If required, parts of the provided functionality can instead be implemented in the VM specific collection classes. Any mixin class can also be extended in the VM specific code before being used. There is one main mixin hierarchy that provides the largest part of the functionality. At the top of this hierarchy is the class `AbstractStrategy`.

`AbstractStrategy` defines the interface implemented in its specific subclasses, in addition to a few internal methods. The following methods are used to interact with the various storage strategy implementations:

store(index, item) Stores the given `item` at the given `index` in the collection. The strategy will check internally if it

can store the value of `item` in an optimized way. If this check fails, the strategy will be automatically switched to a more general strategy that is able to hold both the current elements and the new `item`. The switching mechanism is explained in section 3.2. Optionally, the value of `index` is checked and an error is raised if it is out of bounds of the collections size. The value of `index` must be within the bounds of the collection, even if the collection is variable sized.

fetch(index) Returns the item stored at the given `index` in the collection. Depending on the underlying strategy, it might be necessary to box the value on the fly to represent it on the VM level. Because of this, the results of two identical calls to `fetch` might have different object identities, while still ayways containing equal values. The distinction between object identity and value equality is important. The same goes for a call to `fetch` after switching the underlying storage strategy: the results must contain equal values, but there are no guarantees about the object identity.

size() Returns the number of elements stored in the collection. In case of a variable sized collection, the result of this can change after an intermediate call to `insert` or `delete`.

insert(index, items) Inserts the given list of `items` at the given `index` in the collection. This increases the size of the collection by the number of inserted items and should therefore only be used in the context of variable sized collections.

delete(start, end) Deletes the items at indices `start` (inclusive) to `end` (exclusive) from the collection. All indices must be within the bounds of the collection. This decreases the size of the collection by *end* − *start*.

3.1 Strategy Optimizations

The interface of `AbstractStrategy` is designed in a way that any of its subclasses can be used both in a fixed sized or variable sized context. A collection becomes fixed sized by simply never invoking any of the `insert` or `delete` methods. RPython's optimizer will then determine that underlying storage arrays are never changed in size and accordingly optimize allocations and accesses of such collections.

The subclasses of `AbstractStrategy` represent different storage strategies, as described in section 2. Table 1 gives an overview over the supported strategies, which are further explained in the following paragraphs.

The first one is `EmptyStrategy`, which represents a collection with zero elements. The size of a collection using `EmptyStrategy` is always zero, the `fetch` operation always raises an error, and the `store` operation always switches the strategy to a different one. The `EmptyStrategy` can be used together with variable sized lists, like Pythons primitive `list` type. In such a context, the `EmptyStrategy` would be the starting point for every empty list. Then, depending on the first inserted element, the storage strategy would be switched appropriately. When all elements of a list are removed, the strategy can be optimized to the `EmptyStrategy` once again, in case it is reused with a different type of element.

Table 1. Summary of the strategies supported by rstrategies.

Empty	Size always zero
SingleValue	Every element is the same value
Generic	Can hold any element
WeakGeneric	Holds on weakly to its elements
SingleType	Stores unboxed values of a single type
Tagging	Stores unboxed values of a single type plus one special tag value

Another useful strategy mixin is `SingleValueStrategy`. This strategy can only store a single value and therefore does not require to allocate an entire storage array. Again, this strategy can save a lot of memory when used frequently. The `SingleValueStrategy` must allocate memory only for a single value: the size of the list. As soon as a different value is stored in this strategy, it must be deoptimized to a more general strategy. This is useful for programming languages with fixed sized primitive collections that are always instantiated filled with a default element. For example, a Java `Array` is initially always filled with `null` values, and a Smalltalk object is initially always filled with `nil` values. Here, the `SingleValueStrategy` can be used as initial strategy, before being deoptimized to a more general strategy.

`GenericStrategy` is the strategy that is able to handle any element. It corresponds to a simple, non-optimized list and is the fallback strategy when any other strategy has to be deoptimized. Depending on the context, there can be cases where a `GenericStrategy` is again optimized to another strategy. For example, if the last element is removed from a variable sized list using a `GenericStrategy`, it switches the strategy to `EmptyStrategy`. This way, it will again switch to another optimized strategy next time an element is added. Similar scenarios can be constructed when using a fixed sized list, but in that case additional information must be maintained about the elements. For example, keeping track of the number of `null` values would allow optimizing a generic fixed-sized array to `SingleValueStrategy`. The associated overhead of maintaining this information would most likely surpass the optimization benefits.

`SingleTypeStrategy` is able to store elements of a single type in an optimized storage array, as described in section 2. Certain conditions must be met in order to use this strategy. Namely, the identity of the element objects must be determined only through their value, and element objects must be immutable. These strict conditions are usually only met by primitive types like integer, float datatypes, or value classes, if supported. Therefore, the `SingleTypeStrategy` can be used to implement optimized storage for collections of primitive datatypes. The VM has to provide routines for *boxing* and *unboxing* the elements of this strategy. These rou-

tines form the bridge between the machine level and VM level representations of the element values. For example, assume an `IntegerStrategy` has been implemented based on `SingleTypeStrategy`. Now a boxed integer object is added to a collection with such a strategy. This object has to be unboxed in order to store its value into a machine level array of integers. When fetching a value from this strategy, the according boxed integer object has to be created again.

The `TaggingStrategy` is an extension of `SingleTypeStrategy`. In addition to optimized storage of a single element type, it uses one specific tag value in the value range of unboxed elements to represent some special boxed value. This is useful when fixed sized collections have a default value for their contents. For example, a fresh Java `Array` is filled with the default value `null`. When this array is filled with `Integer` instances, there will often be some slots in the array which still have the value `null`. Now if we were using the `SingleTypeStrategy`, we would already have to deoptimize to `GenericStrategy` in order to represent both `null` and `Integer` values in the same collection. With the help of the `TaggingStrategy`, we can avoid this early deoptimization by representing the `null` slots with special integer values, for example the maximum possible integer value.

There are two drawbacks to this strategy. First, it adds the overhead of comparing fetched elements with the tag value. This is the penalty for representing two different types in the same optimized storage. Second, the `TaggingStrategy` can not store the tag value itself. Therefore, the tag value should be chosen to occur very rarely in real situations. This can be hard for integer values, since even the maximum possible integer can very well be produced by the user program. For float values, the IEEE 754 Standard [7] allows for multiple representations of the `NaN` value, one of which can be used as tag value.

`WeakGenericStrategy` is a version of `GenericStrategy` that holds on weakly to its elements. This means that an element of a list using a `WeakGenericStrategy` can be garbage collected, if the list contains the last reference to this element. Many programming languages require support for weak collections or objects on VM level. Since this strategy serves a very special purpose, collections should never switch from or to this strategy. Instead, a collection should be instantiated with this strategy and keep it throughout its lifetime.

The described strategies can further be combined by the VM. For example, a Smalltalk object can consist of two parts: a fixed sized part containing instance variables, and an optional variable sized part, which is Smalltalks notion of arrays. Even though the two parts form a single object, they can have different requirements regarding storage strategies. In this case, the VM can use two instances of different strategies to represent a single object.

The VM can even implement strategies that contain objects with more than one data field. For example, a language might have a primitive type representing a *point*, consisting of two number fields. A collection of such *point* objects could be optimized by storing all the point numbers in one consecutive array. This can be implemented using `SingleTypeStrategy` and storing instances of RPython's `tuple` type into it. In this example, the VM would have to provide routines to convert between the tuple of raw data, and an actual *point* object.

3.2 Switching Strategies

An important part of rstrategies is the mechanism that allows a collection to switch between different storage strategies. The generic way to switch from one strategy to another requires two steps:

1. Allocate a new storage strategy of an appropriate type and with an appropriate size.
2. Invoke `fetch` on the old strategy instance and `store` on the new strategy instance for every element in the old instance.

While the first step can not be avoided, the second step can be largely optimized in many cases. For example, when switching from a `SingleValueStrategy` to a `GenericStrategy`, every `fetch` operation will yield the same result. Therefore, an optimized switching routine can be implemented, which allocates a `GenericStrategy` and fills its storage array with a fixed value, using a routine similar to `memset` of the C standard library.

4. Evaluation

In this section we evaluate the rstrategies library in two general directions: runtime performance and applicability to different programming languages. We evaluate the runtime performance by comparing the results of benchmarks executed with and without storage strategies. We show the generality of the library by demonstrating two programming languages of very different nature that have been successfully adapted to using rstrategies.

4.1 Performance

In this section we present the results of experiments measuring the execution time of benchmarks to evaluate the performance benefits of storage strategies. The experiments have been conducted on a 64-bit Windows 8.1 machine with 8 GB of RAM and an Intel Core i7-2620M CPU with two physical and four virtual cores, clocked at 3.40 GHz during single-core execution. During the experiments, no other user applications were running and the experiment process was given increased priority in order to minimize influences from operating system processes. We used the RSqueak VM at commit 7a0ce39abc12, PyPy at commit 35fdf446e439 and a Squeak 4.5 image to take the measurements.

To evaluate the runtime performance of rstrategies, all experiments were conducted twice. First, all specialized storage strategies have been disabled via a command line switch. In this setup, the `ListStrategy` is used by every object or

collection created during the benchmarks. In the second setup, specialized storage strategies were enabled. In the case of the RSqueak VM, these strategies are `AllNilStrategy` (a subclass of the abstract `SingleValueStrategy` that just defines the single value to be the RSqueak VM's `nil` object), `SmallIntegerOrNilStrategy` (a subclass of the `TaggingStrategy` which uses RSqueak small integers and tags `nil` as special value) and `FloatOrNilStrategy` (which does the same for floats). We compare the runtime performance of these two experimental setups to evaluate the impact of the specialized storage strategies.

The following list gives short explanations of the seven benchmarks executed during the experiments. These are all common benchmarks based on The Computer Language Benchmarks Game[1].

AStar The AStar path searching algorithm is used to find a solution through two predefined mazes. This benchmark creates a big graph of interconnected objects and generates lots of non-recursive messages. Two different mazes are solved, called AStar1 and AStar2; the second maze is larger.

BinaryTree Balanced binary trees of a given depth are constructed and then walked, summing up the elements of all leaf-nodes. This benchmark generates many recursive messages.

Blowfish The symmetric-key block cipher Blowfish is used to encrypt and decrypt fixed challenge data. This benchmark is heavy on basic number arithmetic, very few objects are created.

DeltaBlue DeltaBlue is a constraint solver benchmark. Different constraints are created for a number of variables, then the solver finds optimal solutions for the given constraints. This benchmark is heavy on message sends and conditional logic.

NBody The NBody algorithm creates a number of objects with an assigned movement vector and mass, then performs a round-based simulation of their movement and interactions using Newton's laws of motion. This benchmark balances basic arithmetics and message sends.

Richards The Richards benchmarks simulates the task scheduler of an operating system kernel. Multiple classes of tasks can be created with different priorities, then the tasks are scheduled until every task has finished. The main workload of this benchmark consists of message sends and conditional logic.

SplayTree This benchmark creates a splay tree of a fixed size and then performs a number of tree modifications by removing the greatest key node in the tree. This benchmark creates and modifies a big interconnected graph of objects.

[1] `http://benchmarksgame.alioth.debian.org/`

Table 2. Excluded and analysed measurements for the different benchmarks.

Benchmark	Excluded Measurements	Analysed Measurements
AStar1	14	36
AStar2	14	36
BinaryTree	3	47
BlowfishDecryption	13	37
BlowfishEncryption	16	34
DeltaBlue	8	42
NBody	3	47
Richards	10	40
SplayTree	12	38

For all experiments, entire benchmark suite is executed within the same VM instance. This creates a more realistic setup, since a real-world application is likely to be long-running and to perform multiple different tasks.

4.1.1 Results

The execution time of the benchmarks was measured by recording timestamps at the beginning and end of every benchmark. Every benchmark was executed 50 times to achieve a satisfactory significance level. From the 50 measurements taken for every benchmark, the first n are excluded from the analysis to eliminate the impact of the JIT warmup time. The number n is determined manually for every benchmark, since the time it takes to reach steady performance differs from one benchmark to another. Table 2 lists the number of excluded measurements for every benchmark.

The remaining measurements have been summarized by means of confidence intervals using a 5% significance level. The confidence intervals comparing the two VMs have been computed following the procedure proposed by Kalibera and Jones [8]. The confidence intervals summarizing the results of the individual VMs show the standard error of the population, assuming normal distribution. Table 3 summarizes the results, each line represents one of the benchmarks. The table is divided in three columns, each summarizing different results for the respective benchmark. The first column shows the results for the RSqueak VM *without* storage strategies, i.e. with the rstrategies library disabled. The second column shows the results for the RSqueak VM with storage strategies enabled. The numbers in the first two columns are given in milliseconds. The third column shows the relative performance change from the first to the second section. A positive number indicates increased performance, meaning that the benchmark was executed in less time.

The AStar1 and SplayTree benchmarks showed considerable increase in performance, while Blowfish and NBody showed a minor decrease. The other three benchmarks showed minor increases between 1% and 7%. Since the means and standard errors shown in the first two columns differ quite substantially, it would not be meaningful to construct

Table 3. Performance measurements for different benchmarks. From left to right: RSqueak VM without specialized storage strategies, RSqueak VM with specialized storage strategies, relative performance change between the first two. Measurements given as 95 % confidence intervals.

Benchmark	Without Strategies	With Strategies	Performance Change
AStar1	80.64 ms ±1.92	55.81 ms ±2.61	+46.93 % ±6.58
AStar2	351.17 ms ±8.15	288.81 ms ±2.68	+21.21 % ±2.25
BinaryTree	187.47 ms ±1.00	174.94 ms ±1.77	+7.26 % ±1.01
BlowfishDecryption	422.16 ms ±2.05	429.16 ms ±2.49	−1.62 % ±0.64
BlowfishEncryption	423.85 ms ±2.00	427.15 ms ±2.47	−0.76 % ±0.63
DeltaBlue	99.86 ms ±2.24	98.31 ms ±2.16	+1.57 % ±2.62
NBody	271.38 ms ±2.15	274.09 ms ±2.16	−0.98 % ±0.94
Richards	165.97 ms ±2.80	162.95 ms ±4.14	+2.11 % ±2.29
SplayTree	781.16 ms ±2.25	469.92 ms ±2.89	+66.28 % ±0.97

a single confidence interval from all benchmarks. Instead, we observe that storage strategies provide high performance benefits for certain problems, while the potential performance penalty remains very low. The average half-width of the confidence intervals is 1.99%, which is reasonably low for a significance level of 5%.

The two benchmarks that benefit most, AStar and SplayTree, both create large graphs of objects. Therefore, it is not surprising that they benefit a lot from an allocation removal optimization. The benchmarks with slight performance losses, Blowfish and NBody, are the two benchmarks with the highest focus on arithmetical operations. Since there are not many object allocations happening here, we can measure the overhead of storage strategies as a slight performance drop.

These results seem to indicate that the benefits of storage strategies very much depend on the workload—Squeak, although it is a very object-oriented environment, has relatively few objects that include only the primitive types for which we have strategies. Although the benefits for some algorithms do look good, a Squeak VM also has to run the entire development environment, including rendering to the screen and communicating over the network. The algorithms used there depend more on arithmetic performance and rarely use many homogeneous objects, diminishing the benefits of strategies. During development use, we thus have experienced little performance benefits of strategies.

4.2 Generality

An important goal of the rstrategies library was to be independence of the language executed by the VM. Neither the programming language nor indeed the class or family of programming languages should matter, and our work should be applicable to a range of dynamic language VMs. In order to evaluate this, we introduced rstrategies into three different RPython based VMs. As presented, in the RSqueak VM, nearly every object uses storage strategies. In the Topaz Ruby VM, we added strategies to the generic array datatype that is provided by the VM. Finally, in Pycket [1], we added strategies to the vector datatype. While it is not possible to prove that our design works for arbitrary programming languages, we believe that the three selected examples cover an interesting range of languages. Squeak and Ruby are both object-oriented and imperative languages; while Ruby is a scripting language, Squeak is image-based language and loads an entire live programming environment before executing the first bytecode. Racket (implemented by Pycket) belongs to the Lisp/Scheme programming language family. It is a multi-paradigm language including object-oriented features, but the syntax, control flow, and main data structures are typical for a functional programming language.

We were able to integrate strategies in these RPython based VMs in largely the same way. To extend this evaluation, we use the two VMs RSqueak and Topaz to demonstrate how rstrategies works in the context of different programming languages. Our strategies library includes a logging facility which can output a diagram of all strategy related operations such as the creation of collections or switching between different strategies. We used this to create transition graphs for RSqueak and Topaz while executing different benchmark suites. For RSqueak we reused the benchmark suite from the previous section, for Topaz we used a benchmark suite consisting of the 5 benchmarks BinaryTrees, Dhrystone, Mandelbrot, Revcomp, and Richards. We have not selected two equal benchmark suites since we are not conducting a performance comparison between the two VMs.

Figure 5 show the resulting transition diagram for the RSqueak VM. Objects created at image loading time are directly instantiated with the best-fitting strategy, while objects created later always start off with the `AllNilStrategy` or the `WeakListStrategy`. The `WeakListStrategy` is a special case: weak objects never transition to any other strategy after creation. Other objects transition to one of the specialized strategies, in the current implementation either `SmallIntegerOrNilStrategy` or `FloatOrNilStrategy`, or directly to the generic `ListStrategy`. The percentage of objects remaining in any of the specialized strategies is a

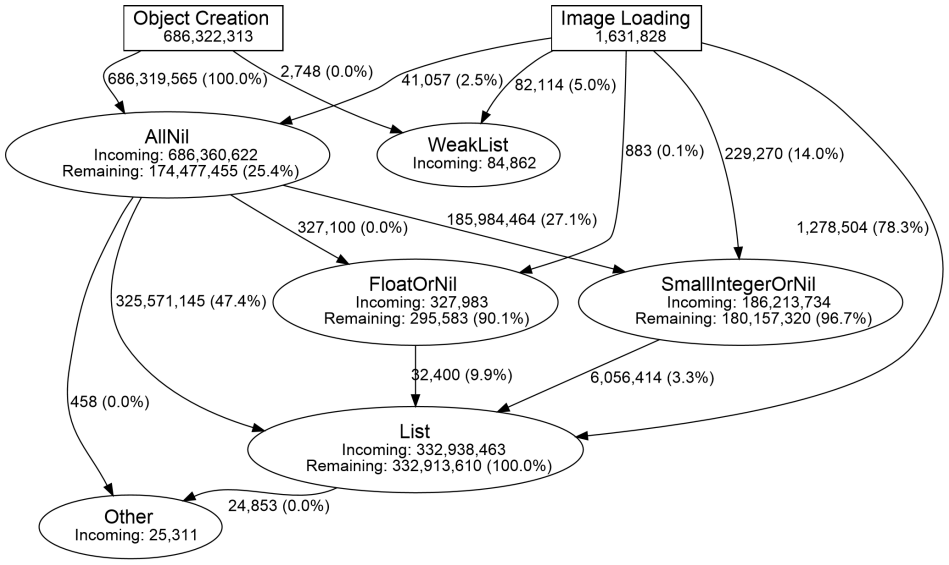

Figure 4. Strategy transitions of the RSqueak VM executing benchmarks

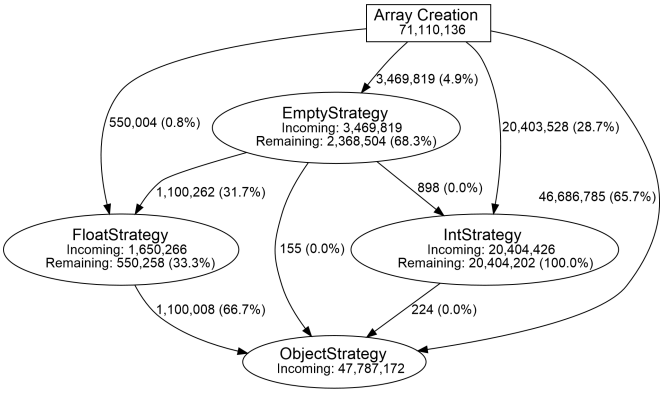

Figure 5. Strategy transitions of the Topaz VM executing benchmarks

good indicator for how well the storage strategies perform. In the case of RSqueak, approximately 25 % of all objects never leave the `AllNilStrategy`. These objects are mainly MethodContext objects which are created automatically upon message sends. 97 % and 90 % of all objects arriving at the `SmallIntegerOrNilStrategy` or `FloatOrNilStrategy`, respectively, actually stay in these specialized strategies. These numbers indicate a high success rate of the heuristics behind choosing the storage strategies. The node at the bottom labeled `Other` summarizes a few rarely used strategies which are special to RSqueak.

Figure 5 shows an example transition graph for the Topaz VM. It has fewer nodes than the RSqueak diagram because in Topaz, strategies have only been added to the array datatype, not to every single object. The `IntStrategy` has a near 100 % success rate, which we assume is due to the nature of the benchmarks executed while capturing this data. The `EmptyStrategy` also has a rather high success rate, while the `FloatStrategy` only works one out of three times. Compared to the RSqueak VM, these numbers look less conclusive and more artificial. The main reason is the different nature of the two VMs: RSqueak loads an entire graphical programming environment, which means the VM executes some unrelated code even while running these benchmarks. Topaz, on the other hand, evaluates pure Ruby scripts, which results in a more deterministic and uncluttered execution.

5. Related Work

The Graal/Truffle project is a VM building framework similar to RPython in its main design goal. An interpreter written in a statically typed programming language is used as the source artifact to automatically produce an efficient VM including a JIT. Graal [9] is an experimental extension to the HotSpot VM which exposes VM internal mechanisms to the executed program via a Java API. Namely, HotSpot's method based JIT compiler can be controlled from the client program. Truffle [11] is a generic, self-optimizing AST interpreter that makes use of the interface provided by Graal. In order to implement a VM using Graal/Truffle, the program code must be parsed into a syntax tree and passed to Truffle for execution. Truffle will dynamically modify the AST using techniques like partial evaluation and inlining to create compilation units for heavily used code paths, which are then converted to machine code by the Graal JIT. This approach effectively uses a method JIT compiler, but is similar to RPython's metatracing JIT.

The main difference between RPython and Graal/Truffle, aside from the underlying infrastructure, lies in the flexibility for the VM programmer. Graal/Truffle requires the VM implementor to use an AST, while the RPython toolchain accepts arbitrary Python programs as long as they satisfy the type inference.

The project JRuby+Truffle[2] is a Ruby VM written in Java which uses the Graal/Truffle infrastructure. JRuby+Truffle includes storage strategy optimizations [10], but not in form of a reusable library like rstrategies.

A common VM optimization is to use pointer tagging to store some immediate values, such as integers, floats, or characters directly but tagged. This reduces the number of bytes available for the immediate values, but often provides significant performance benefits, as specialized CPU instructions can be used to perform arithmetic. However, compared to storage strategies, this increases the complexity of the interpreter, as each pointer has to be checked for tags, and only a very limited set of tagged types can be provided.

6. Discussion

The collection types supported by rstrategies are fixed sized and variable sized lists. Of course there are may more collection types and algorithms used in modern programs. The collection framework in the Java standard library features 8 interfaces and 10 implementation classes. A modern Squeak image[3] has as many as 79 subclasses of `Collection`, because it includes collections with very special purposes. However, rstrategies is a library for optimizing collections on VM level, which means it is only meaningful to support collections that occur as *primitive types* in multiple programming languages. Even though Java supports collections like `ArrayList`, `TreeSet` and `HashMap`, the only primitive generic collection type is the `Array`. The same holds for Smalltalk and many other programming languages. And even though Smalltalk (as well as many other programming languages) have type-specific versions of collections for some primitive types (e.g. `ByteArray` and `WordArray`), these put the burden on the user of the programming language to choose the right VM-level representation.

It would be possible to take the approach of storage strategies even further and let a VM transparently optimize collection types from the standard library of the programming language. For example, since the interface and semantics of Javas `ArrayList` are well defined, the VM could try to handle instances of `ArrayList` by itself, without actually executing the Java bytecode in the methods of the `ArrayList` class. This might lead to improved performance when accessing elements of `ArrayList`, but this approach has several drawbacks. One important function of a standard library is to reduce the complexity of the VM. Moving parts of the library into the VM would only further increase its complexity. Another purpose of having a separate standard library is that it is able to evolve independently of the VM. It is very hard to exactly mirror the code of the standard library and the suggested approach would require releasing a new VM version after every substantial change to the standard library.

Therefore we chose to only optimize primitive collection types with rstrategies. The rest of this section lists and discusses several collection types not supported by rstrategies, although they do occur as primitive types in several modern programming languages.

[2] `http://www.chrisseaton.com/rubytruffle/`

[3] `http://www.squeak.org/`

Table 4. Primitive collection types in dynamic languages

Language	Fixed List	Variable List	Dictionary
JavaScript		Array	Object
PHP		array	array
Python	tuple, bytearray	list	dict
Racket	vector, cons, box		hash
Ruby		array	hash
Smalltalk	Array		

6.1 Non-List Collections

We examined the primitive datatypes in various dynamic programming languages. Table 4 summarizes the results by listing the primitive datatypes of each language and mapping them to abstract collection types which are explained in the following.

We distinguish three general kinds of collections: *fixed lists*, *variable lists* and *dictionaries*. Fixed and variable lists correspond to the collection types supported by rstrategies, using the mixin hierarchy described in section 3.1. Hence, all collection types listed in these two columns could be implemented and optimized using rstrategies.

Dictionary is a primitive collection type that is quite important for many languages. Storage strategies can be extended to include support for dictionaries. This has been done for the Python interpreter PyPy [3] and in the JRuby+Truffle project [10]. We have not yet implemented dictionaries as part of rstrategies yet, but intend to do so as part of our future work on this library.

Note that Python has several additional primitive types for special use cases: buffer, xrange, set, frozenset, iterator, generator. This variety of primitive collection types is special to Python and all these types can be mapped to the *Fixed List* or *Variable List* types by adding additional algorithms to the `store` and `fetch` routines. Therefore we chose not to support special datatypes like these.

6.2 Non-Contiguous Lists

Functional programming languages often feature a primitive datatype for linked lists. Lisp for example has the built in type `cons` which represents one link of a linked list, formed by two fields: the value (named `car`) and the pointer to the next link (named `cdr`). While these labels have historical reasons originating in the 1950s, the linked list is still a central data structure in modern Lisp programs. Since accessing arbitrary elements of a linked list requires walking the entire list up to that element, optimizing linked lists would be very beneficial. For example, all elements of the list could be placed in consecutive memory, allowing constant access to any element. However, such optimizations bring about further complications and are a separate (although related)

research topic to storage strategies. Therefore, we consider optimizations of linked lists out of scope for rstrategies.

7. Conclusion

We have presented a generic implementation of storage strategies that can be used by different virtual machines implemented using the RPython toolchain. We have shown that storage strategies can be applied to different dynamic languages and that they provide benefits for some workloads, while inducing only small overhead for others.

Since we have not experienced significant performance gains while using RSqueak VM for development, but only in select benchmarks, we can not conclusively recommend storage strategies for every dynamic language workload. We have left a command line switch to deactivate storage strategies in place for now, so that users can deactivate them depending on the workload. In addition, our results indicate that the added implementation complexity may not be worth the performance benefit if storage strategies have to be implemented separately for any VM implementation. Thus we feel that storage strategies should be a generic feature provided by VM toolchains such as RPython or Truffle, so that the tradeoff between performance and code complexity is not an issue.

Acknowledgments

We acknowledge the support of HPI's Research School and the Hasso Plattner Design Thinking Research Program. Carl Friedrich Bolz is supported by the EPSRC *Cooler* grant EP/K01790X/1.

References

[1] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Krilichev, T. Pape, J. Siek, and S. Tobin-Hochstadt. Pycket: A tracing JIT for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP '15, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. to appear.

[2] C. F. Bolz, A. Cuni, M. Fijłakowski, M. Leuschel, S. Pedroni, and A. Rigo. Allocation removal by partial evaluation in a tracing jit. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 43–52. ACM, 2011.

[3] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 167–182. ACM, 2013.

[4] G. Bracha and W. Cook. Mixin-based inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, 1990.

[5] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183. ACM, 1998.

[6] B. Goldberg and Y. G. Park. Higher order escape analysis: optimizing stack allocation in functional program implementations. In *ESOP'90*, pages 152–160. Springer, 1990.

[7] IEEE. Ieee 754: Standard for binary floating-point arithmetic, Aug. 2014. URL `http://grouper.ieee.org/groups/754`.

[8] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *ACM SIGPLAN Notices*, volume 48(11), pages 63–74. ACM, 2013.

[9] Oracle. OpenJDK: Graal project, Aug. 2014. URL `http://openjdk.java.net/projects/graal/`.

[10] C. Seaton. Optimising small data structures in jruby+truffle, Aug. 2014. URL `http://www.chrisseaton.com/rubytruffle/small-data-structures/`.

[11] C. Wimmer and T. Würthinger. Truffle: a self-optimizing runtime system. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 13–14. ACM, 2012.