JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Optimizations for Dynamic Collections

Joannah Nanjekye[*], **David Bremner**[*], **and Aleksandar Micic**[†]
[*]University of New Brunswick, Canada
[†]IBM, Canada

ABSTRACT Languages that support dynamically evolving collections present a range of challenges in modern virtual machines, and require sophisticated optimizations to address any efficiency gaps. Collections in such languages dynamically evolve in size as well as the types of the elements they contain, and thereby affect performance and memory efficiency due to the expensive operations required to support resizing and the heterogenous object types. This paper presents two collection optimizations to address this cost; 1) a novel memory layout, *type-based stores*, storing primitives of heterogenous collections unwrapped in contiguous storage areas; and 2) an allocation context aware collection presizing technique, which is a profile-guided optimization that uses feedback from allocation sites and infers the calling context from the stack height to determine an optimal collection size. We implement type-based stores as a collection optimization in PyPy, a Python implementation that is built using the RPython framework for meta-tracing-JIT based virtual machines. We also implement language-independent type-based stores in RPython and perform evaluation for Topaz, a Ruby implementation and Pycket, a Racket implementation. Our evaluation shows that while some individual Python collection operations see a noticeable slowdown, in PyPy benchmarks with more realistic mixes of operations, the overall performance is improved by an average of 11.4%. The language-independent type-based stores also improve performance for Ruby and Racket applications by as high as 5% and 14% respectively. We also apply an offline version of the allocation context aware presizing technique to PyPy, observing a performance improvement of 11% and memory savings as high as 16% on average for the best median strategy; while we introduce an overhead as high as 25% when accessing and applying the profiled data.

KEYWORDS Collections, Optimizations, Heterogeneity, Resizing, Dynamic Languages.

1. Introduction

Dynamic languages like Python and JavaScript continue to be widely used and for a large number of domains. Their main characteristic is that they use dynamic type systems to allow rapid prototyping, but also provide desired features like: easier syntax, automatic memory management, a robust ecosystem of standard libraries and third party packages, support for dynamic code generation and execution, interactive execution environments, and advanced introspection capabilities (Ilbeyi 2019).

These productivity-oriented features come at a cost compared to static languages. This is because dynamic languages degrade performance with dynamism due to the extra runtime tasks required for aspects like type checking and method binding among others.

One of the most important idioms in languages that support dynamic features is the use of collections, data structures that allow the programmer to store and retrieve data by location or by name. Compared to programming languages that do not support dynamic evolution, dynamic collections have important programmer conveniences including the ability to grow and shrink (resizing), and the ability to contain elements of multiple types (heterogeneity). Both of these features have a memory management related performance cost.

This paper makes two contributions to improving the performance of collections in dynamic languages. First, we present and evaluate a *type-based stores* technique that transparently

JOT reference format:
Joannah Nanjekye, David Bremner, and Aleksandar Micic. *Optimizations for Dynamic Collections*. Journal of Object Technology. Vol. 24, No. 01, 2025.

http://dx.doi.org/10.5381/jot.2025.24.01.a1

re-organizes heterogeneous collections in memory. Second, we use offline and lightweight feedback to accurately predict the final size of a collection while being sensitive to the allocation context.

Type-based Stores. We use the terms *heterogeneous* and *homogeneous* for collections in this paper.

Heterogeneous Collections are data structures where a single container can have members of dissimilar types. For instance, consider the pseudocode for a Python function in Listing 1, that computes the sum of select items of a list, `lst_het`, and defines another list, `lst_hom`.

```
1 def process_lst():
2     lst_het = [1, "foo", 3, "bar", 5, 6]
3     lst_het[0] + lst_het[2]
4     lst_het[0] + lst_het[1] # fails
5     str(lst_het[0]) + lst_het[1]
6     lst_hom = [1, 2, 3]
```

Listing 1 An Example of Data Structure Heterogeneity

Heterogeneous collections as the one shown on line 2 of Listing 1 are typically represented uniformly, with a general object type, using a technique known as boxing or wrapping. When an operation is performed on an item, the language implementation has to unwrap or unbox the memory, to check its concrete type as a way of verifying whether the operation is acceptable for that type of value or not. In this example, lines 3 and 5 will execute correctly while line 4 will throw an exception, as it is forbidden to perform an arithmetic operation between a string and an integer.

Homogeneous Collections on the other hand contain items of the same type, for instance, the list `lst_hom` on line 6 of Listing 1. The literature addresses the challenges regarding collection boxing of values with optimizations like *storage strategies* and *element kinds*, both of which only handle homogeneous data structures (Bolz et al. 2013; Clifford et al. 2015). PyPy supports storage strategies for collections with a mix of floats and integers but does not do so for any other combination of types.

PyPy's storage strategies assume that if any collections become heterogeneous, then they do so when the collection has few items. However, we show in Section 3.1 that for several Python workloads, storage strategies actually lead to worse performance. This is because, when a collection becomes heterogeneous, it is iterated, boxing any items that may have been unboxed before. Therefore storage strategies have two challenges, 1) they do not optimize heterogeneous collections and 2) they do not deal well with collections that become heterogeneous later in program execution.

Element kinds (Clifford et al. 2015) are similar to storage strategies and minimize dehomogenization through initializing collections from a given allocation site using a strategy of a previously allocated collection from the same site. This reduces the number of transitions a collection can have, but it also means that a list that becomes homogeneous later will not be optimized if it already transitioned to a heterogeneous collection prior. For large collections, that transition to a general kind also incurs a boxing overhead. Therefore element kinds also share most of the same challenges as storage strategies.

We propose *type-based stores*, a new memory layout to handle heterogeneous collections through *collection splitting*. The layout introduces multiple contiguous stores to match the types of items in the collection, storing objects of the same type in each store, and can use the *strategy pattern* to manage the stores, as well as unwrap the values.

We implemented the proposed mechanism in PyPy, a Python implementation, built using the RPython framework. We also demonstrate language-independent type-based stores in RPython and perform evaluation for Topaz, a Ruby implementation and Pycket, a Racket implementation, both RPython-based virtual machines.

Our evaluation shows that while some individual Python collection operations see a noticeable slowdown, in PyPy benchmarks with more realistic mixes of operations, the overall performance is improved by an average of 11.4%. The language-independent type-based stores also improve performance for Ruby and Racket applications by as high as 5% and 14% respectively.

The improvements are much lower in Ruby and Racket because we integrate and use a modular version of type-based stores for Ruby and Racket, implemented in the RPython-based Rstrategies library as *RStores*. The library uses metaprogramming to interface the runtimes but runtimes also have specific implementation considerations during integration, because Rstrategies does not have complete information about collection implementation details for the different runtimes.

The goal of the *type-based* stores technique is to find an efficient storage representation for collections containing values of a primitive type; based on occurring element types, for dynamic languages where there is a significant possibility that collections are heterogeneous. We aim to optimize collections of primitive types like strings in data structures where techniques like tagging are hard and complex to achieve. Tagging also has performance implications due to branch prediction. Type-based stores can be defined as contiguous storage areas created in memory, to hold items of the same type.

Context Aware Presizing. Data structure resizing allows for the expansion and shrinking of collections as shown in Listing 2 on lines 4 and 6. To support this flexibility, first, collections are internally over allocated, for example the lists `a` and `b` are allocated twice the slots needed in most virtual machines. This is a performance optimization and it is possible to naively resize by one on every insertion, but in many cases this performance optimization is used and over allocation is wasted memory for data structures that do not expand.

```
1  a = [1, 2, 3]
2  b = ["foo","bar", "zar", "zoo", "zeh"]
3  if condition:
4     a.extend(b)
5  else:
6     a.append(4)
7
```

Listing 2 An Example of Data Structure Resizing

Second, expansion requires expensive creation of larger internal structures and an extra copying cost to the larger structures. For example, when line 6 is executed, a larger internal array is

created, copying items from the older list to the new one, as well as deleting the older list.

Presizing is a technique used to predict an optimal data structure size to avoid the overhead involved with resizing operations but can also avoid over-allocation of internal slots. Existing work uses allocation site-based techniques to achieve presizing (Clifford et al. 2015), and canonical profiling (Henning et al. 2020), all of which do not account for the calling context and branches in the program.

Data structure resizing exists in both statically and dynamically typed languages; for example, like Python lists, C++ vectors are a related comparison for a statically typed language. Static languages have similar dynamism for some collections like many dynamic languages. The literature known to us does not mention automatic presizing in these contexts[1] . It is therefore possible to discuss presizing for static languages in a different context.

To address the presizing limitations in existing work, we show that it is possible to predict an optimal data structure size by combining the *context-identifier* of an *allocation site*, using *call-site analysis* with *collection size* profile data created either online or offline, thereby achieving allocation context-aware data structure presizing.

We apply an offline version of this allocation context-aware presizing methodology to PyPy, observing a performance improvement of 12% and memory savings of 16% on average for the best median strategy; while we introduce an overhead as high as 25% when accessing and applying the profiled data. In the rest of this paper, collection and data structure are used interchangeably.

The allocation context-aware data structure presizing optimization proposed is designed to predict the *size* of a data structure, based on its *allocation calling context*. This is achieved through matching data structure sizes to their calling contexts from profiling information.

2. Background

In this section, we briefly introduce the RPython framework and the different ways collections are optimized in dynamically typed languages in the context of types and resizing.

2.1. The RPython Framework

The RPython programming language was developed simultaneously with the PyPy Python implementation. In fact, earlier work on PyPy described the RPython framework and translation process as one of the components of PyPy, the standard Python interpreter being the other component [76, 83].

The framework is now maintained as a project of its own. Therefore, for the purposes of the discussion in this work, PyPy is defined as just the Python implementation written in RPython. RPython is a restricted and statically typed subset of the Python language. It is not limited in syntax but more in how it handles objects of various types. Languages developed with the framework are compiled to various supported environments and platforms, using the RPython translation toolchain. In addition to PyPy, we implement our work in two other languages implemented in RPython; namely Topaz (Gaynor 2013) and Pycket (Bauman et al. 2015).

[1] There are some manual presizing APIs in C++

2.2. Memory Layout of Types in Dynamic Languages

In languages where types are instead associated with runtime behavior, schemes to identify the types of values at runtime are required. This category of languages is said to support *dynamic typing*. The memory layout in these languages is such that there is a uniform representation of all objects regardless of type with a common *object* type.

In practice, there is a possibility of converting between specific types like integers, strings, etc., and the uniform *object* type, as well as determine what type of object is being represented. For this reason, Just-In-Time (JIT) compilers defer optimizations until run-time, when such languages can identify types of objects (Bolz et al. 2013; Gudeman 1993; Diekmann 2012). We discuss the different ways types are represented in dynamic languages.

2.2.1. Canonical Type Optimizations

Gudeman surveyed and evaluated several strategies for representing types in dynamic languages (Gudeman 1993). The canonical schemes can be classified as *tagged words, partitioned words, object pointers, large wrappers*, and *typed locations*. A combination of these techniques can be used for a hybrid representation.

Tagged Words use a *tag field* to represent the type of an object. The object is represented as a sequence of bits, with one or more tags storing the type of the object and the rest storing other object details (value). This scheme leads to compact memory layouts and relatively good access time, but not all unwrapped values can be represented as wrapped values.

Partitioned Words allocate each type to a certain subset of the available bit patterns. Each type is therefore restricted to representing those values that it can represent in the bit patterns allocated to it. The entire word in this strategy is legitimate as opposed to only the value field being legitimate for the tagged word representation. Object pointers use a machine pointer to a memory block containing all type information to represent each wrapped value. The object is therefore a structure that consists of the required information to identify what type of value the block represents.

Large Wrappers use more than a single word to represent both the type and a complete machine value. This scheme requires more registers, uses more memory, and incurs higher costs in loading and storing wrapped values than the single word schemes. Some optimizations can reduce this cost, and those optimizations are not possible for single word schemes.

Typed Locations allow for determining the type of the pointer based on where it is located on the stack, register, or any place in memory. In static languages, locations rather than values have types. Closely related, dynamic systems can also have a memory layout where types have type codes, but the value and the type code are located in separate places.

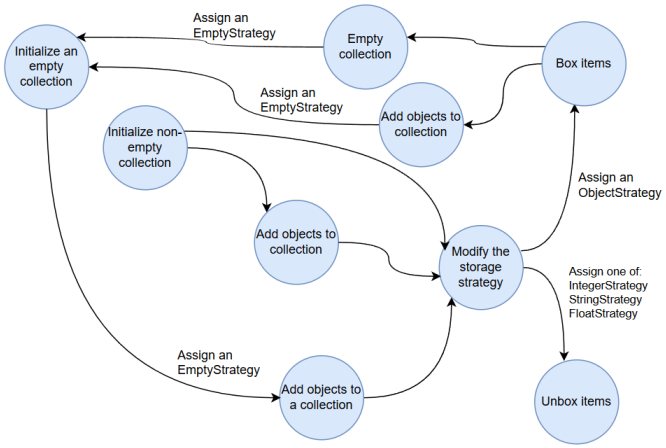


Figure 1 Storage Strategies (Bolz et al. 2013)

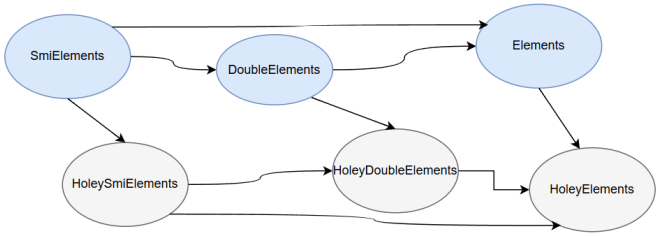


Figure 2 Element Kinds in V8 (Clifford et al. 2015)

2.2.2. Specialized Optimizations for Collections

An integral part of recent research on the representation of types for dynamic languages has focused on optimizations targeting the design of efficient collection libraries (Bolz et al. 2013; Clifford et al. 2015; Bergel et al. 2018; Daloze et al. 2018; Henning et al. 2020; Maas et al. 2020a). We discuss two of these prominent optimizations, namely *storage strategies* and *element kinds*.

Storage Strategies Aiming to solve the challenges of pointer tagging for type representation, storage strategies optimize the treatment of collections (Bolz et al. 2013). Storage strategies were proposed with two assumptions, 1) that homogeneous collections seldom de-homogenize and 2) when they de-homogenize, this happens when a collection has a small number of elements.

The design is such that each collection references a storage strategy and storage area in memory. Similar to the strategy design pattern, the storage strategy manages all operations related to a collection but also how data is laid out in the storage area.

Figure 1 shows that collections can evolve through several storage strategies during program execution starting with an *EmptyStrategy* for an empty collection. A collection gets a specific strategy when items are added to it. The specialized strategy unwraps the elements and stores them.

When an element of a different type is added to a collection, the collection is assigned a general *ObjectStrategy* and each element is boxed returning to the equivalent of the default representation of the collection in dynamic languages.

Storage strategies are generally more efficient than pointer tagging but for some corner cases, they actually perform worse compared to when they are not used. One of the cases is where a large homogeneous collection de-homogenizes. This results in the re-boxing of each element in the collection, an operation with high overhead. The changes in strategies are also a source of overhead. To solve this, the *V8's element kinds*, discussed next, limit the frequency of strategy transitions.

Element Kinds V8's element kinds were developed during the same time as storage strategies (Clifford et al. 2015). Both approaches share the same underlying object model where collections with elements of the same type are unboxed and stored in a designated area in memory. The representation of elements of a collection is called *ElementKinds* as shown in Figure 2.

Elements in a collection are represented in six ways depending on their type and whether the collection has holes. A spatial array is an example of a collection with holes, i.e., with missing items in some memory cells.

A list of integers will have a special kind of *SmiElements*, adding a float to the list will transition it to a *DoubleElements* kind. Similarly, when the list gets a hole, it will be transitioned to a respective *Holey* kind like *HoleySmiElements* for integers. The paper on ElementKinds discusses support for only *Integer* and *Double* types and it is not confirmed anywhere if other types like *Strings* are supported (Bynens 2017).

A shortcoming for both approaches is the fact that they both do not optimize heterogeneous collections. We propose an alternate approach to handle this in Section 3.

2.3. Collection Presizing Schemes for Dynamic Languages

The performance overhead and resource wastage due to shrinking and expanding collections in dynamic languages has been addressed using presizing together with other optimizations like pretenuring and pretransitioning (Bruno et al. 2019; Maas et al. 2020a; Henning et al. 2020; Marr & Daloze 2018a), with anecdotal details on implementation and evaluation.

Presizing is a technique in which profiling information is used to create an optimal internal size of a collection. We discuss some of the techniques from the literature on presizing, which include *mementos*, *recycling*, *lazy creation* and other language specific methods.

2.3.1. Mementos

Allocation mementos as proposed by Clifford et al. (Clifford et al. 2015), are temporary objects allocated next to an object they track for purposes of storing profiling information about the object to allow for further optimization. Mementos are created at either the object's call-site or allocation site and live for a short time typically, only surviving the closest garbage collection cycle.

For presizing, mementos are used to store the sizes of collections from the last run at a given allocation site or call site so that in future, collections from a given site are allocated an internal data structure with slots of an observed size to avoid the overhead of having to resize the collection.

Mementos do not take up any space when garbage collected in the object and are observed to have no engineering com-

plexity, having been achieved in roughly 150 lines of code for presizing in the V8 runtime but they have overhead associated with their operation. They are also tied to the garbage collector and so runtimes like Graal VM could not directly emulate or use them for presizing (Henning et al. 2020).

2.3.2. Recycling When a collection expands from a user request to a point where the internal structure has to be resized, the process involves creating a new larger collection, copying the items from the older collection and garbage collecting the old unused collection.

Bergel et al. propose not garbage collecting the unused collections (Bergel et al. 2018), and instead reusing them for future requests that fit the specifications of the now unused internal collection. This may avoid creating many internal collections but does not take away the overhead from copying of items from the old to the new internal collection.

2.3.3. Lazy Creation When a collection is initialized in dynamic languages, an internal collection is immediately created and over-allocated with the assumption that the collection will expand eventually, which is not true in some cases.

Lazy collection creation or *initialization* is a technique where the internal slots of the collection are not allocated until a request to add items to the collection is encountered (Bergel et al. 2018). This avoids extra slots that may have been allocated, which may have been more than required, hence wastage.

2.3.4. Other Language Specific Approaches In the GraalPython project, due to the garbage collector requirements of mementos, the creation and append operations of Python lists were intercepted, observing the sizes of collections, lists in this case, and using this information to compute the optimal sizes for all other collection instances using the same allocation site (Henning et al. 2020).

This profiling information is collected during interpretation to reduce performance overhead. This approach generally avoids most of the bookkeeping overhead of mementos but does not consider the allocation calling context, limiting its applicability. The calling context captures the control flow of the program in branches.

3. Type-based Stores for Heterogenous Collections

We start with the motivation in Section 3.1 before discussing our solution and its evaluation in Sections 3.2 and 3.3.

3.1. The Overhead of Heterogenous Collections

To motivate our study, we start with an analysis of the impact of storage strategies on Python applications, and also test the hypothetical case of a very large homogeneous collection becoming heterogeneous, common for applications with large collections that transition. The prevalence of strategy transitions at least suggests a potential for improvement.

We demonstrate the prevalence of heterogeneous collections in Table 2 and the overhead of an existing optimization, storage strategies, that only handles homogeneous data structures in Figure 3. The numbers in Table 2 are an approximation breakdown of how many collections transition from homogeneous integer, string and empty collections, to being heterogeneous. For this experiment we evaluate four collection intensive standard PyPy benchmarks (Contributors 2012) modeled after the official PyPerformance Python benchmark suite (Python 2016).

Benchmark	Description
Nqueens	The n-queen problem solver using different search algorithms
Pidigits-modified	A modified implementation that computes arbitrary digits of pi
Float	Heavy floating-point arithmetic
Richards	A Python implementation of the Martin Richards program
Delta Blue	Constraint solving problem
AI	An algorithm to exercise the performance of a simple AI system
Eparse	LXM parsing
Meteor-contest	An implementation of the Meteor puzzle board
Fannkuch	Indexed-access to tiny integer-sequence
Spectral-Norm	Calculating the spectral norm of an infinite matrix
Chaos	Fractals for the chaos game
Nbody	The original nbody problem solver as translated to Python
Telco	Measuring the performance of decimal calculations
Call_Simple	A trivial function call
Regex_Effbot	Working of Regex
Nbody-modified	The modified nbody problem solver as translated to Python
Unpack_Sequence	Unpacking a sequence
Fib	The fibonacci algorithm

Table 1 A Description of PyPy Benchmarks

The benchmarks are described in Table 1. Figure 3 also uses an artificial example, *list-example*, to create a homogeneous list of one million integers and transitions it to a heterogeneous collection by adding a string to it. The rest of the benchmarks are part of the standard Python benchmark suite. We reproduce the benchmark results of the storage strategies paper (Bolz et al. 2013), observing two key limitations.

First, we show that about as high as 50% of collections in our benchmarks are heterogeneous and thereby remain unoptimized by storage strategies as shown in Table 2. We observe that list-based collections transition the most by about 50% from being empty. Equally significant are list transitions from being integer-based, which impacts about 22% of the lists. Lists are followed by dictionaries where about 42% of integer-based dictionaries become heterogeneous. Transitions from being empty are also significant at about 100,000 lists, a point capable of introducing overhead. Sets experience the least transition to the general object/wrapped state, the highest being 4% transitions from being empty. We do not track the distribution of collection sizes when a transition happens but the speed overhead discussed next elaborates on the overhead storage strategies.

Secondly, optimizing for only homogeneous collections degrades performance for some workloads with large collection sizes due to the need to re-box items when a homogeneous collection becomes heterogeneous. When collections become heterogeneous, we observe a slowdown in speed when storage

	Total Number	Number of Transitions	Percentage (%)
Lists			
Integer	9,000,000	2,000,000	22.2
String	20,000,000	3,000,000	15
Empty	30,000,000	15,000,000	50†
Sets			
Integer	1,500,000	20,000	1.3
String	30,000,000	30	1.5
Empty	5,000,000	200,000	4
Dictionaries (Key Types)			
Integer	120,000	50,000	41.7
String	350,000	30	0.2
Empty	2,000,000	100,000	5

Table 2 Collection Strategy Transitions – an approximate breakdown of collection transitions to the object strategy during program execution while running the PyPy benchmark suite for benchmarks described in Table 1 . For some key container type combinations, as much as 40%–50% of collections become heterogeneous from a specific type like integer, string or empty. (†) We show the total number of transitions from the empty strategy to both homogeneous and heterogeneous collections here.

strategies are used. To our second observation, to understand this overhead, we ran the five benchmarks in Figure 3, against two PyPy versions, one with storage strategies turned off and the other with storage strategies turned on. We only exercise lists since they are the most used data structures but insight from the experiment is consistent with sets and dictionaries. The slow down is because every item in the list is revisited to box the items again at the point of becoming heterogeneous.

In Figure 3 the hypothetical example is interesting; when we transitioned a list of one million integers to the object strategy, the program took 25 seconds with storage strategies to run while it took less than five seconds without storage strategies. We also observe similar slow downs of about 4% by mean with the other four PyPy benchmarks, though not as dramatic as the hypothetical example. This shows that optimizing for heterogeneous collections is potentially beneficial for some Python applications and storage strategies instead worsen the performance of such applications.

The overhead is therefore more pronounced during the boxing operations when a collection becomes heterogenous, but in the transitions as well. Reducing the frequency of transitions using mementos and allocation-site feedback can improve performance but does not optimize heterogeneous collections, which is the main focus of our work.

3.2. The Type-based Memory Layout

We do not attempt to forward homogeneous collections to the type-based stores technique, as these are already handled by

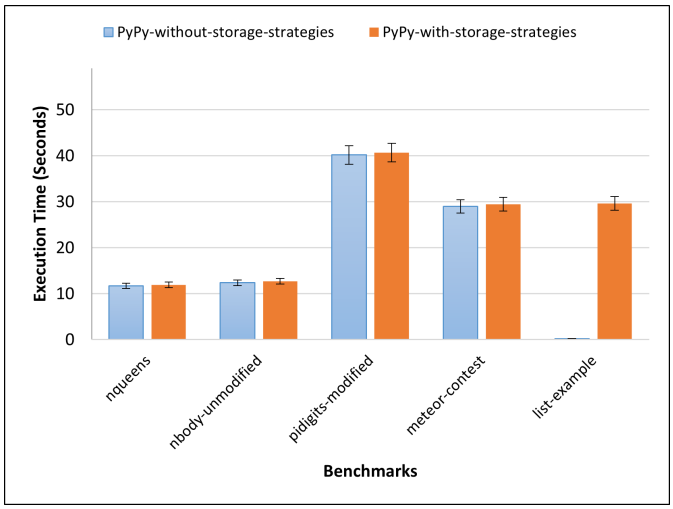
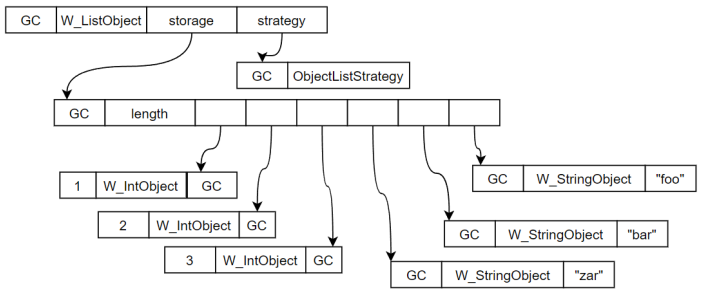
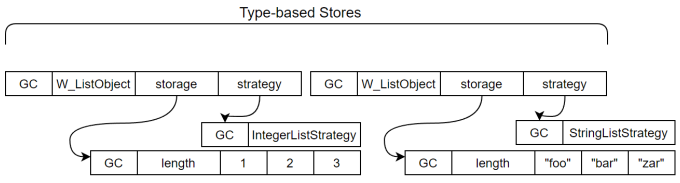


Figure 3 The Overhead of Storage Strategies – the unoptimized heterogeneous collections and the impact of strategy transitions slows down applications. This leads to an overhead of 4% by mean for the four benchmarks we ran from the PyPy benchmark suite described in Table 1. Transitioning a list of one million integers to the object strategy, takes the program 25 seconds with storage strategies while it takes less than five seconds without storage strategies

existing optimizations. Instead, the heterogeneous data structure is broken down by categorizing its members by type creating two stores in this example: one to hold the strings and the other for integers. We use contiguous storage areas that are expandable and capable of holding both sequential and hash-like data.



(a) Storage Strategies



(b) Type-based Stores

Figure 4 Type-based Stores vs. Storage Strategies – consider a list `lst = [1, "foo", 2, "bar", 3, "zar"]`, (a) shows the memory layout of the list after applying storage strategies; and (b) shows the memory layout of the list after applying type-based stores and storage strategies

For example, the heterogeneous collection `lst = [1, "foo", 2, "bar", 3, "zar"]`, can not be optimized with existing techniques like storage strategies. Instead, the items stay wrapped, with types `W_IntObject` for integers and `W_StringObject` for strings as shown in Figure 4 (a). However, with type-based stores, shown in Figure 4 (b), the numeric items `1, 2, 3` are stored in a separate contiguous area, while the textual items `foo, bar, zar` are stored in another adjacent area, each storage area managed by a storage strategy which unwraps the items.

The stores are created on demand, and there is a one-to-one mapping between a data structure and its respective internal set of stores. Therefore given a collection L, and an internal layout D, a mapping function exists, f, from D to L, for all d in D and l in L such that, $f(d) = l$. The associated stores of a collection are dynamic, to expand and contract, matching the size of the collection they represent. Therefore, at all times, $|L| = |D|$ as discussed further in Section 3.2.2.

For sequential data structures, the indexing in the sequential storage area is implied. The size `N` of each storage area in the sequential parts is computed dynamically, every time the internal layout has to resize as the largest power of two such that at least half the elements in the stores will be filled. A mapping is also needed between the source collection and the type based stores.

The goal of restructuring the collection to use stores is to allow for further optimization. Therefore, when the collections have been split into the different stores by type, we take advantage of the RPython methods that are based on the `rerased` RPython library module to unbox the items in each store.

The `rerased` library provides two main functions of interest to our work, namely; `erase` and `unerase`. The `erase` method wraps the object by erasing its type, returning a generic type which is the equivalent of a void pointer in C or `Object` in many languages while `unerase` unwraps the objects in the storage area to access their actual types. We mostly use `unerase` since unwrapping is the core of our technique and call `erase` to wrap objects for cases we do not support.

3.2.1. Memory Layout Analysis

Our technique determines how elements of a heterogeneous collection are allocated in memory using a restructure function that creates a set of stores for a collection, placing elements in one of the following relative positions:

$$\{colocated,\ close\}$$

Given an arbitrary collection, `lst = [`X_1`, `X_2`, ..., `X_n`]`, we define a *colocate* operation, $\sim$ to show that two objects are in the same store. For any given heterogeneous collection, a layout mapping exists with several stores as follows:

$$\text{layout(lst)} = \begin{cases} store\ 1 & X_1 \sim X_2 \sim ... \\ store\ ... & ... \sim ... \sim ... \\ store\ k & X_{n-2} \sim X_{n-1} \sim X_n \end{cases}$$

Any of the items will be colocated in the same store, for example, $X_1 \sim X_2$, if they are of the same type and *close* (X_1 and X_{n-1}) if not in the same store but in the same set of stores belonging to a given collection.

At the point of restructuring the collection, triggered by creation or modification, every item should be allocated in one of the stores. We optimize for collections containing a mix of primitive types like strings and integers but if a collection contains non-primitive types, optimization (splitting to stores) is not supported yet, so we do nothing in this case and delegate to wrapping. At all times, two collection primitive items of different types will be colocated only if they have not been mapped to any stores yet.

3.2.2. Internal Mapping after Restructure

The consequence of reorganizing the original data structure, for sequential collections, is that there has to be a way to track items in the stores and their positions in the original collection to help with access and modification by any operations. We maintain a non-trivial map, implemented efficiently as a hash, that is updated on collection creation/update and read when there is a read request to the items of the collection. This map is the same map referenced in the storage strategies paper, PyPy already also uses maps to handle certain structures, like instances.

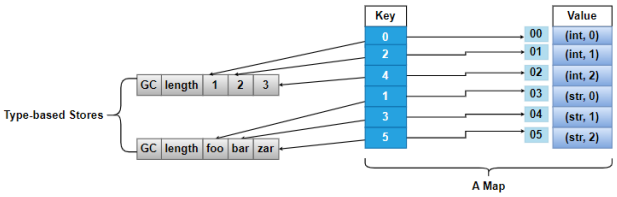


Figure 5 Internal Mapping to Support Operations – the map tracks positions of items in the source data structure to help with processing access operations. This map is useful for sequential collections

Figure 5 shows how items in a list, `lst = [1, "foo", 2, "bar", 3, "zar"]`, are laid out in memory, and a map used to track positions of items in the source data structure. For purposes of easier presentation in this figure, the map is rearranged to group values of the same type together. A typical access, `lst[i]`, first checks the map to acquire the position in the stores for an item at this index.

The values in the map, should have an indication of the type so that during the search, we can distinguish between two similar indices that may refer to different stores. This map introduces an extra step in collection access but as we show in the evaluation section, the benefits of unwrapping the items outweigh this overhead. The pseudocode below shows the steps to access `lst[i]` in a given representation.

```
store = lst.map[i] -> (int, 0)
for stor in lst.stores()
    type = store[0]
    if stor.type == type
        index = store[1]
        item = stor[index]
```

Listing 3 The Internal Map Access Algorithm

First, we search the internal map by index `i` as shown on line 1 of Listing 3, which returns a tuple containing the item type and its index in the store. On lines 2–4, we search the collection stores by the type, retrieving a store that matches the type of item we want to access. We use the index read from the map to access the item in the retrieved store on line 6.

3.2.3. Dictionaries Type-based stores for dictionaries can be applied for both keys and values, but we do not take this route of unwrapping each pairwise (key, value) primitive type combination. We instead optimize for hashing and comparing of keys by creating multiple stores to match the types of keys in the original data structure as shown in Figure 6.

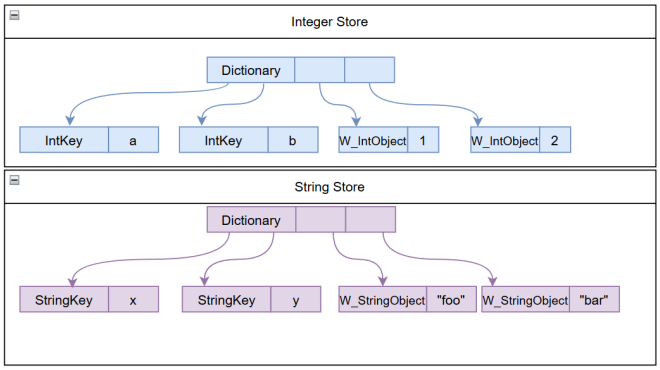


Figure 6 Type-based Stores for Dictionaries – we only optimize keys to reduce the cost of handling every possible key/value type combination when unwrapping

Consider a dictionary, `{a: 1, b: 2, x: "foo", y: "bar"}`, where `a` and `b` are integer keys while `x` and `y` are string keys. The keys are unwrapped while values remain wrapped. This is a good compromise because keys are more stable in many applications than values in terms of type transition (Bolz et al. 2013) but the many type combinations if we were to handle both keys and values can also easily become costly.

We also do not create a separate internal map for dictionaries, instead dictionary operations search the associated stores by key, since duplicate keys are not allowed. This reduces the bookkeeping for dictionaries.

3.2.4. Implementation The algorithm for the optimization described in this work is summarized in Algorithm 1 and impacts list, set, and dictionary data objects. We modify the PyPy3.9 branch, which is the latest version at the time of writing this paper. This algorithm is invoked at the point where a collection switches from homogeneous to heterogeneous.

We first read all the items in the heterogeneous collection (line 3), then transfer control to the layout function to allocate the required stores (lines 6 and 10). Depending on whether the collection is a list, set or dictionary, we create a new instance of storage areas, either create sequential storage areas for lists or hash-like storage areas for dictionaries, then split the objects, sorted by type into the storage areas, and finally update the map on line 7 for lists.

Algorithm 1 TypeBasedStores: layout(ds)

Input: Let ds be the heterogeneous collection.
Result: Layout and unbox heterogeneous collections.

```
initialization;
items = ds.getItems();
ds.discard();
if type(ds) == list then
    createSequentialStores(items);
    updateMap();
if type(ds) == dict or set then
    createHashStores(items);
```

To achieve the procedures in Algorithm 1, we implement a new internal data structure in PyPy. Splitting of the heterogeneous collections involves intercepting the implementation of the collections at the points where collections currently switch to the general object strategy or at initialization after checking if the collection is heterogeneous, thereby creating stores for each collection, and passing items by type to the stores.

Listing 4 is an extract of the implementation of the `List` data structure/collection in PyPy. Lines 5–9 consist of the method that currently switches any heterogeneous collections, while lines 10–13 have an implementation of the `length()` method for lists. For lists, the `switch_to_object_strategy` (line 5) routine and `from_storage_and_strategy` (not shown in this code), which is one of the other operations, have special conditions to handle heterogeneous collections by boxing.

```
1  class W_ListObject(W_Root):
2    strategy = None
3    storages = None
4    ....
5    def switch_to_object_strategy(self, w_item=
       None):
6      list_w = self.getitems()
7      self.clear(self.space)
8      if self.stores is None:
9          self.stores =  create_stores(self.space
       , list_w)
10   def length(self):
11     if self.stores:
12         return self.stores.size
13     return self.strategy.length(self)
```

Listing 4 Splitting Objects to Type-based Stores – pseudocode for the key implementation details of the type-based stores technique for the list data structure in PyPy

We therefore intercepted them to use or pass their data to the type-based stores mechanism. We also consequently modify existing functions like `length()` on line 10, etc., to instead read and/or modify the stores of the said collection (line 12). In PyPy and CPython, sets are handled as dictionaries with the actual items of the set as keys and corresponding values being `None`, so in our implementation sets are handled by the code for dictionaries.

3.2.5. Evaluation The goal of our evaluation is to compare the PyPy version based on our technique with a default PyPy

Property	Specification
CPU	Intel(R) Xeon(R) Gold 6248
Clock Speed	2.50GHz
Operating System	64-bit Debian 10.2.1
Cores	10
GCC Version	10.2.1

Table 3 Experimental Hardware Setup

baseline version that does not optimize heterogeneous data structures, i.e., the one based solely on storage strategies.

We use this baseline for two main reasons 1) PyPy, the Python implementation used, has storage strategies in its default upstream branch, which means all PyPy users currently use storage strategies making it the feasible baseline by virtue of widespread use and 2) turning off storage strategies completely would involve a major overhaul of the interpreter because all major data structures including classes are affected, which is unrealistic and out of scope of our research.

Optimizations that reduce the boxing overhead mostly impact execution time but also reduce the memory footprint. We therefore use both metrics in our experiments and attempt to answer the following questions:

1. *How much, if at all, does the technique improve the execution speed of Python applications?*
2. *How much, if at all, does the technique reduce the memory consumption of Python applications?*
3. *How much, if at all, does the technique introduce overhead to collection operations?*

Methodology We use a subset of the standard PyPy benchmark suite, mainly the unladen swallow benchmarks, based on the standard *Pyperformance* benchmark suite. Pyperformance benchmarks are designed to emulate real-world Python applications. To generate results exercising all supported collection operations, we used artificial workloads. We use the hardware setup detailed in Table 3. We run the benchmark programs 30 times, ignoring the first 5 runs; this is not aimed at determining accurate steady-state because it is impractical, but rather a large number of iterations increases the likelihood of getting close to steady-state. The acronyms TBS and NTBS, denote, type-based stores and no type-based stores respectively.

Execution Speed Figure 7 shows the execution time. To answer the first question, we observe that the type-based stores technique improves performance for 11 of the 16 benchmarks. Of these, five benchmarks (*float, richards, deltablue, ai, regex-effbot*) have speedups of 20% and above and the rest are 10%–19% or slightly below (*spectral-norm, call-simple, eparse, meteor-contest, fannkuch, nbody-modified*). Of these 11 benchmarks, storage strategies alone could not improve performance.

We estimate the magnitude of this speed to be 11.4% for the 16 benchmarks, a number we arrived at by calculating the geometric mean increase (positive speed ups) and slow down (negative percentages) for the benchmarks. For specifics, the speedup is between 2% to 40%. The slow downs are between 2% to 60%.

The benefits of our technique can be better appreciated by comparing the boosts and slow downs in the original homogeneous-only optimization in the storage strategies work (Bolz et al. 2013) with our results. Benchmarks like *fannkuch*, *meteor-contest*, *richards*, and *spectral-norm* performed worse when only homogeneous collections were optimized, but with our technique they are faster, which means heterogeneous collection optimization is beneficial to them.

On the other hand, looking closely at the benchmarks where our technique slows down performance, the slow down is observed for benchmarks like *telco* and *nbody* that ran faster with just the homogeneous collection optimization. This is because we perform extra checking in the algorithm before splitting, even though the algorithm does not optimize homogeneous collections; these extra checks are overhead. These benchmarks are fewer, and by default dynamic languages assume polymorphism, so we can optimize for language defaults. The *chaos* and *unpack-sequence* benchmarks are an outliers as they run slower for both the homogeneous and heterogeneous optimization.

Memory Consumption Figure 8 shows the memory consumption. To answer the second question, these results are mixed; our technique uses less memory in some benchmarks and more memory in other benchmarks. The benchmarks *ai* and *eparse* run faster, and still use about 3% and 1% less memory respectively; however, while the *float* and *fannkuch* benchmarks run faster, they use more memory, about 1% - 2% more respectively. The benchmarks *fib* and *unpack-sequence* run slower but use less memory.

Our technique generally has to reduce the size of objects, so we expect less memory consumption, but we introduce a memory layout with a map that may use more memory in some cases. For example the *richards* benchmark runs faster but uses more memory with type-based stores. The storage strategies paper (Bolz et al. 2013) also shows increased memory usage for some benchmarks, so we can expect that type-based stores will inherit the same behaviour because of using strategies in the stores.

The reason is that without storage strategies, a single object is allocated once on the heap, with multiple pointers to it from collections and elsewhere. With storage strategies, the same element can be unboxed in the storage strategy, and then, when pulled out of the list, reboxed to multiple distinct objects in the heap. This is because some items that are unboxed inside a list are later used outside and reboxed.

The Overhead of Type-based Stores We base our analysis on the results shown in Figure 9. To answer the third question, several operations slow down due to the extra checks we do before creating or accessing the stores of a collection while others are not affected.

Despite some extra checking and processing of stores, some list operations are faster, specifically `getitem()`, `sort()`, `count()`, and `create()` are 23%–60% faster, `remove()` runs

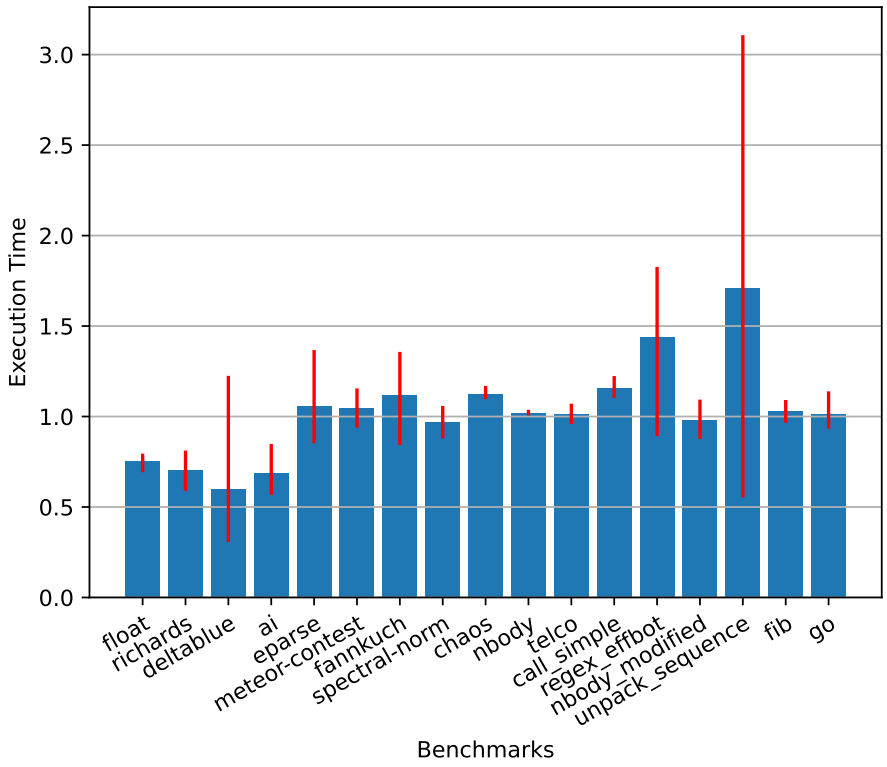


Figure 7 Speedup for PyPy – the results are normalized to the storage strategies baseline, lower is better, the type-based stores technique accelerates execution times for most of the benchmarks

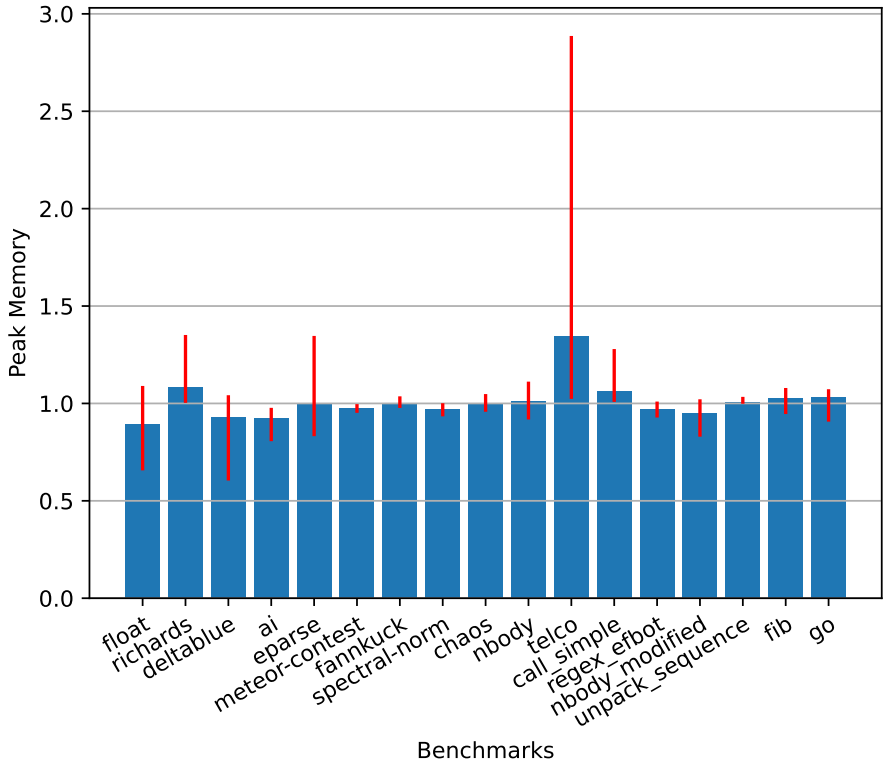
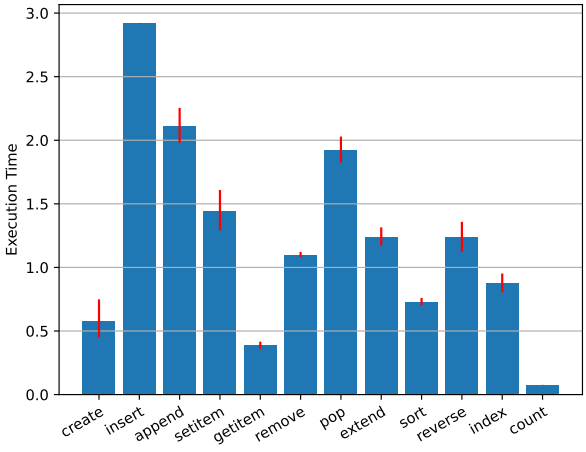
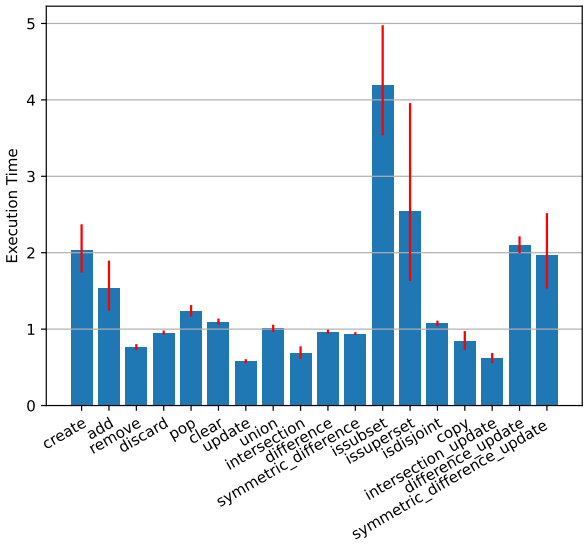


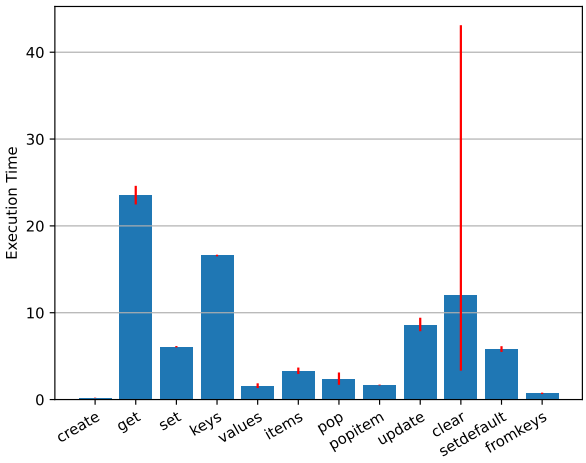
Figure 8 Memory Savings for PyPy – the results are normalized to the storage strategies baseline, lower is better, the type-based stores technique reduces the memory footprint for some benchmarks and increases the memory footprint for some others



(a) Lists



(b) Sets



(c) Dictionaries

Figure 9 The Overhead of Collection Operations – the results are normalized to the storage strategies baseline, lower is better

at almost similar speeds with overhead no more than 8%, `append()` and `insert()` operations cost about 100%–200% overhead due to the extra search and bookkeeping required for indices. Similarly the other operations like `setitem()`, `pop()`, `extend()`, and `reverse()` experience overhead between 25%–90%.

Similarly, most of the set operations run faster with our technique, for example, about 24% better for the `remove()` operation, 39% better for the `update()` operation, 30% better for the `intersection()` operation and 41% better for the `intersect_update()` operation. However, the `difference_update()`, `create()`, and `symmetric_difference_update()` operations are almost twice as slow due to extra checks and collection of items from the different stores. Sets are processed as dictionaries, so some of the overhead related to dictionaries discussed next, applies.

Due to not requiring any synchronization of indices between the original dictionary and the stores using the map, dictionary operations like `fromkeys()` and `create()` are less impacted and run at approximately the same speed. However, we experience about 500% to 800% slow-downs for the `set()` and `update()` operations because of extra checking, and creation of stores. Generally, most dictionary operations have overhead with our approach.

As noted above, there is an overhead observed for a few operations, where we added an extra step of creating the stores, but also where some checks are required to access certain items in the collection. However, as we discussed in Section 3.2.5, this overhead is insignificant and does not have much impact on the general speed for most applications, the unwrapping performance boosts outweigh the overhead in these few operations for most applications.

3.3. Language-independent Type-based Stores

Storage strategies were generalized for use by RPython virtual machines (VMs) in a library called *rstrategies* (Pape et al. 2015). The library was ported to several languages. This work also extends this library to optimize heterogeneous collections in a language-independent way. This section discusses our algorithm for providing type-based stores in rstrategies, along with an integration to two RPython VMs, Topaz and Pycket.

3.3.1. Overview of Language-independent Storage Strategies

The rstrategies library is based on the use of metaclasses and metaprogramming concepts rather than making normal API calls, due to the complexity involved in handling collection types in the generic manner required and the need for flexibility.

Integration with an RPython VM requires that a collection implementation adheres to the use of a central class structure or hierarchy. All RPython classes implemented in the VM can duplicate attributes and functionality from the library by importing `mixin` classes. The VMs we experiment with all follow this structure and the actual integration of the baseline rstrategies library involves three main steps.

The first step is to implement getter and setter methods in the VM collection class to expose the strategy and storage attributes required to optimize collections using storage strategies. These methods can be implemented manually but rstrategies also uses a method, `make_accessors(strategy='strategy',storage='storage')`, that can be called within the VM collection class to automate the implementation of these getter/setter methods.

Secondly, rstrategies provides mixins that should be used to create strategy classes, starting with a single root strategy class, `AbstractStrategy`. The `AbstractStrategy` class is an interface that should be subclassed to specific strategy classes, to implement the methods used to interact with various storage strategies. The specific strategy classes match the known strategies for the different object types like `IntegerStrategyClass`, etc. The strategy classes can use a decorator which signals to the strategy class what strategy to switch to in case the current collection cannot be assigned to the current strategy, `@rstrategies.strategy(generalize=alist)`. This usually happens when an incoming item to the collection is not of the same type as the items in the current collection.

Then thirdly and finally, VMs have to subclass a class with some modifications (`StrategyFactory`) to any methods responsible, like `instantiate_strategy` for strategy initialization, switch actions (`switch_strategy`), etc. This allows the VM collection operations to use the storage strategies mechanisms of unboxing.

3.3.2. Extending Rstrategies with Type-based Stores

In extending *rstrategies* with type-based stores, we modified the mechanism responsible for switching between storage strategies. The switch that we care about for this work is where a concrete strategy is switching to the general strategy.

Therefore, inside *rstrategies*, we intercept the step that is responsible for strategy classes (step two in Section 3.3.1 above), modifying how generalization works. We define generalization as a point where the current strategy class notices an incoming item of a different type and defers to a routine that assigns the `ObjectStrategy` instead, and boxing items as required.

We specifically provide a decorator `rstores` that can be applied to strategy classes in the VM, that invokes type-based stores instead of switching to the `ObjectStrategy` when a collection becomes heterogeneous. We therefore do not directly modify generalization, the type-based stores feature is optional, and if the `rstores` decorator is not applied, we default to assigning the `ObjectStrategy`.

The `ObjectStrategy` can also be assigned when you apply the `rstores` decorator for aspects we do not support yet, like instances. Therefore, anything the strategy class can not handle through a concrete strategy or `rstores` will default to generalization.

Listing 5 contains pseudocode for the routine that handles collections that are not homogeneous. We use an example where a new object `value` of a different type is appended to a homogeneous collection, `w_self`. On line 1, we override the generalization method, `generalize_for_value()`, `self` is the current instance of the collection class. We first process the type of the incoming item `value` on line 4. Then we invoke

```
1  def generalize_for_value(self, w_self, value):
2   ......
3   try:
4    strategy_type_value = self.get_stategy_type(
       value)
5    create_type_based_stores(self.
       strategy_factory(), w_self,
       strategy_type_value, value)
6     ......
7   except (ValueError, AttributeError):
8    pass
9   ......
10 def create_type_based_stores(self, collection,
     strategy, *args):
11   self.store_one = collection
12   list_w = collection.strategy.get_storage(
       collection)
13   self.update_map(self.store_one, list_w)
14   self.store_two = allocate_storage(value, size
       )
15   self.store_two.strategy.append(self.store_two
       , list(args))
16   self.update_map(self.store_two, list(args))
```

Listing 5 Language-independent Type-based Stores - – the method, `generalize_for_value` is called at the point where an incoming item does not match the items in the current collection, and thereby requires generalization to the `ObjectStrategy`

the creation of the required stores on line 5, the items in the current collection, `w_self`, the incoming item, `value`, and strategy type of this item, `strategy_type_new`, are taken as arguments.

When creating the stores, the first store (line 12) is for the existing homogeneous collection, `collection`, which was already allocated a storage strategy of appropriate type and size, all items unwrapped. The second store on the other hand is allocated a storage strategy on line 15, for the incoming item type, and line 16 stores the items, in the store in unwrapped form. For both stores, we have to update the map on lines 14 and 17, tracking positions of the items in the stores, to allow for future access of the collection items.

3.3.3. Evaluation We applied the language-independent type-based stores, implemented in the rstrategies library to both Topaz and Pycket with the new `rstores` decorator. To assess the benefits of the technique, we compare versions of the implementations, using just storage strategies as a baseline, in this case through rstrategies with a version that has the type-based stores extension.

We use this baseline because of similar reasons as PyPy 1) The language implementations use storage strategies by default and 2) turning off storage strategies through rstrategies requires modifications to each language implementation, and are impacted by the collection structure of the language, which is an overhaul that is out of scope of this work.

For Ruby, we use benchmarks from the YJit project by Shopify (Shopify 2023) and add a few more benchmarks from the standard computer language benchmarks game. For Racket, we use benchmarks from the computer language benchmarks game (Chakraborty 2023). All benchmarks are run on a machine with specifications described in Table 3. The numbers are a geometric mean of 30 invocations for each benchmark. We do not describe each benchmark, but the links point to the relevant documentation.

The goal of our evaluation here is to assess the impact of language-independent type-based stores on Ruby and Racket benchmarks for language runtimes using the RPython meta-tracing JIT-based framework. Table 4 has the results for Pycket and Topaz.

Pycket For Pycket, we provide full support for vectors and partial support for hash tables. Both mutable and immutable properties of vector operations are supported.

As shown in Table 4, the highest speed gains are from the *spectral norm* benchmark with about 17%. The lowest gains are from the CTAK benchmark with just about 2%. We can conclude that type-based stores highly impact applications with collections for Racket and still work well for applications that may not be heavily dependent on collections.

We also observe some overhead in some benchmarks; the highest slowdown is registered in the *dot* benchmark of about 20%. For benchmarks with mostly homogeneous collections, we incur overhead due to an extra check to optimize heterogeneous collections, so this is expected and minimal; the dot case is an outlier with the highest overhead here.

As shown in Table 4, the highest memory saving is from the *meteor* benchmark, about 32%. This means that type-based stores can improve both speed and memory usage. There are cases where optimization improves performance at the expense of using more memory, case in point, the *triangle* benchmark, which gains speed but uses 25% more memory. This can be due to the extra bookkeeping in maps for type-based stores to work.

The *dot* benchmark also uses more memory, which can be linked to the fact that compared to the storage strategies branch, it uses mostly homogeneous vectors that work well with an optimization that assumes homogeneity; therefore, the extra processing for type-based stores is just overhead for this benchmark without much benefit.

Topaz The integration with Topaz was more complete than that with Pycket, with all collection data structures and operations supported. Our optimization has the highest speed gain for the *getvar* benchmark, 13%, and the lowest gain for *GC array*, also showing that data structures can run with accelerated performance. We also register overhead sometimes as high as 16% for the *throw* benchmark.

A direct integration of type-based stores may accelerate speed gains if the overhead of calling type-based stores functionality through a library is eliminated. This applies to Pycket too but we do not quantify the overhead, because we considered it out of scope for now.

Other benchmarks that do not heavily rely on collections like *setvar* and *respond* also experience about 3% overhead. Like Pycket, memory is saved in some benchmarks, like *nbody*, and more memory is used in other cases, such as *sieve* benchmark. The memory savings and overhead are all negligible in the Topaz benchmarks, all barely 1% in absolute value.

Pycket			Topaz		
	Speed	Memory		Speed	Memory
nbody	1.046 (1.026, 1.066)	1.012 (1.012, 1.013)	**nbody**	0.982 (0.978, 1.039)	0.999 (0.997, 1.003)
meteor	0.937 (0.912, 0.963)	0.689 (0.689, 0.690)	**fibonacci**	0.938 (0.929, 1.100)	0.999 (0.996, 1.002)
spectral	0.834 (0.723, 0.963)	0.999 (0.997, 1.002)	**binary tree**	1.033 (1.010, 1.075)	1.000 (0.999, 1.001)
triangle	0.944 (0.924, 0.967)	1.253 (1.252, 1.254)	**throw**	1.159 (1.112, 1.239)	1.003 (1.001, 1.005)
vector	1.025 (1.016, 1.035)	0.988 (0.987, 0.988)	**getvar**	0.869 (0.851, 1.070)	0.999 (0.999, 1.000)
bubble	1.015 (0.982, 1.048)	1.269 (1.268, 1.270)	**setvar**	0.968 (0.963, 1.013)	1.001 (1.000, 1.002)
nqueens	1.010 (1.000, 1.018)	0.996 (0.996, 0.997)	**respond**	0.969 (0.961, 1.020)	0.997 (0.996, 1.002)
puzzle	0.868 (0.849, 0.888)	1.259 (1.258, 1.260)	**keywords var**	0.914 (0.905, 1.050)	1.000 (1.000, 1.000)
fannkuch	1.030 (0.996, 1.092)	0.999 (0.997, 1.002)	**gc array**	0.983 (0.979, 1.006)	0.999 (0.997, 1.000)
hash table	0.992 (0.713, 1.178)	0.999 (0.998, 1.001)	**sieve**	0.945 (0.927, 1.056)	1.002 (1.000, 1.005)
CTAK	0.979 (0.862, 1.113)	0.997 (0.997, 1.210)	**spectral**	0.946 (0.844, 1.186)	1.000 (0.998, 1.001)
dot	1.202 (1.171, 1.234)	1.004 (1.004, 1.004)	**fannkuch**	1.033 (1.005, 1.060)	1.001 (0.999, 1.005)
min	0.834	0.689	**min**	0.869	0.997
max	1.202	1.269	**max**	1.159	1.003
geomean	0.986	1.027	**geomean**	0.976	0.999

Table 4 Language-independent Benefits of Type-based Stores – lower values are better, all results are shown as ratios normalized to the baselines. The language-independent type-based stores improve performance for both Ruby and Racket applications

4. Context Aware Presizing

We begin with the presizing challenges in Section 4.1 then delve into a motivating example in Section 4.2. Sections 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, and 4.9 further detail the approach, while Section 4.10 discusses its evaluation.

4.1. Optimal Presize Prediction Challenges

Presizing addresses the over-allocation and resizing overhead of data structures by hypothesizing that data structures from a given allocation site exhibit similar behaviour in regards to size. An observed size through allocation-site-based profiling can be used to compute the optimal size of a data structure. The key benefit for presizing is that it reduces and/or eliminates the overhead due to expansion and shrinking operations when a data structure resizes. Memory consumption is also significantly reduced because with optimal internal slot allocation, there is no need to allocate extra slots for memory that is likely not to be used. Optimal size prediction of data structures is believed to mostly be beneficial if used in hot sections of the code, and impacts large data structures (Henning et al. 2020).

The literature as noted earlier achieves presizing using allocation-site-based profiling, storing the information with garbage collection (GC) assisted approaches (Clifford et al. 2015) by Clifford et al. The GC assisted approach was adopted with some modifications by Henning et al., for GraalPython due to the limited level of access to the GC (Henning et al. 2020) in this virtual machine. Presizing has been attempted in only these two articles known to us, and in both, the technique is discussed with anecdotal evidence and evaluation is performed for one benchmark, *DeltaBlue*. Most significantly both implementations do not solve the key presizing challenges. Optimal data structure presize prediction in existing work does not support the following aspects, namely, branches and allocation calling context.

4.1.1. Branches Setting sizes without accounting for the control flow of a program causes less optimal estimations and leads to the same resizing overhead we seek to avoid with presizing. For example, in Listing 2, the optimal size of list a

depends on whether the code follows the `if` path or the `else` path. Finding an optimal size with this instability is hard and remains an open question in all the existing work on presizing, because making the optimal size 8 could lead to allocation of 4 extra slots if the `else` path is taken, while making the size 4 assuming the `else` path, will lead to costly expansion operations if the `if` path is taken. A more accurate solution must account for the program path in addition to the list size, growth factor, frequency of growth, and allocation site of a data structure.

4.1.2. Allocation Calling Context Similar to the branching problem, the allocation calling context is part of the application call graph, important for profiling real-world applications. Allocation site profiling alone, as demonstrated in existing presizing work, is not sufficient for abstractions built on top of resizable data structures that provide a different interface as observed and pointed out in the GraalPython study (Henning et al. 2020). This is because a high-level collection library, for example, can be seen to have the same allocation site for data structure instantiation at different locations in the program. Profiling the allocation context of these allocation sites is therefore required to facilitate better accuracy in size estimation.

4.2. Our Approach

We use listings 6 and 7 as a non-trivial example of collection resizing to motivate our work. Listing 6 defines several classes and a method, all of which use collections. The main program Listing 7 contains the call sites of interest to our profiling.

In Listing 7, allocation site-based presizing will identify the five allocation sites 2, 3, 5, 6 and 13 initialize the same abstraction, the allocation sites appear the same at both locations. Also, presizing for allocation site 5 is difficult without understanding the context determined by the flag on line 7.

```
1  import collections
2  class AbstractCollection:
3      def allocate(self, num_slots):
4          self.slots = [None] * num_slots
5  class List(AbstractCollection):
6      def init(self):
7          super.allocate(16)
8  class Map(AbstractCollection):
9      def init(self):
10         super.allocate(2*16)
11 def initialize_map():
12     return collections.OrderedDict()
```

Listing 6 Collection Resizing Class and Method Definitions

For better accuracy, context aware profiling profiles call sites in addition to the allocation site. Figure 10 shows the calling contexts, denoted by arrows, for Listing 7, which are the sequence of calls associated with an allocation site and call site. We use $C_{@n}$ and $A_{@n}$ to represent call sites and allocation sites respectively.

In Figure 10, for example, the allocation on line 4 in Listing 6 can have two calling contexts, $main() \rightarrow newList() \rightarrow AbstractCollection()$ and $main() \rightarrow newMap() \rightarrow AbstractCollection()$. Similarly, `list_ex` has two contexts, $main() \rightarrow IF \rightarrow range() \rightarrow append()$ and $main() \rightarrow ELSE \rightarrow append()$.

```
1  def main():
2      R = List()
3      K = Map()
4
5      list_ex = []
6      d1 = initialize_map()
7      if flag:
8          for i in range(0, 10000):
9              list_ex.append(i)
10     else:
11         list_ex.append(1)
12
13     d2 = initialize_map()
14
15 if __name__ == '__main__':
16     main()
```

Listing 7 Example Code Exercising Collections

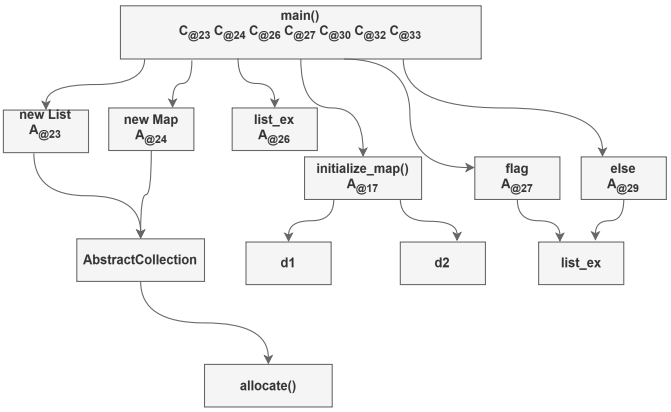


Figure 10 Call Graph for Listing 7 – collections are indicated at the respective call sites. Arrows indicate the calling context which is a chain of calls with origins from the main method

Our approach does not store the whole call chain, we instead infer the context using the (call symbol, stack height) pair and allocation site, with the assumption that the stack height is unique for a given call path or chain of calls. The context identifier, stack height value for the current calling context and call site are computed from the stack pointer (SP) and program counter (PC). This way, even on the different paths for `list_ex`, for example, it is possible to observe a unique stack height for the different `append()` call sites.

This assumption is not always true as the a call path can have the same call site symbol and stack height, like lines 6 and 13, which can cause uncertainty. In these cases, we use existing disambiguation approaches to reduce the duplication of the mappings between call paths and the stack height. We discuss the three cases we are able to disambiguate in Section 4.6. For any cases we can not disambiguate, then our approach will not accurately perform presizing but as acknowledged in existing work (Mytkowicz et al. 2009) and our findings in Section 4.6, this assumption holds for most cases with disambiguation.

4.3. Approach Overview

Answering the presizing question stated at the start of this section is not straightforward, and like many optimizations, ad-

dressing every use case is aspirational. Figure 11 shows the techniques used to answer the question and the following novel profile-guided methodology is used to arrive at an ideal solution:

1. The first step is to gather profiling data about collection sizes, allocation sites and their allocation contexts. This can be done either offline or online.

2. The profiling data is initially stored in a log file but we provide some structure by storing it in a hash map, we call a context map, to guide the presizing optimization. We also install the allocation context in the object header, that is matched later. We discuss the impact in size to the object header in Section 4.7.1.

3. Before presizing, we access the object header and the map to match the size using the calling context as key.

4. Then the data structure is allocated slots using an exact size, estimated and read from the map to avoid waste.

We define the *size* of a data structure as the number of elements it contains. We estimate this size using profiling information that tracks the maximum, median and mean sizes of collections along with the number of times they grow and the growth factor when their allocated size is exhausted.

On the other hand, the allocation calling context is a tuple containing (1) the allocation site identifier, which is the allocation location in the source code for the collection object; and (2) the context identifier, which is a pair containing the symbolized name of the currently executing function and the stack height. The context identifier infers the stack state or trace during object allocation as discussed in the next section.

4.4. Object Size Profiling

The branching and calling context challenges discussed earlier are addressed by our approach by enhancing presizing to work with profiling data that is based on both allocation sites and the calling context. We infer the calling context by building on existing assumptions in the literature about the *context identifier* (Maas et al. 2020b; Mytkowicz et al. 2009). A context identifier is a pair consisting of the *call site symbol* and *stack height* while a call path is the nested sequence of calls invoked at runtime.

We therefore complement allocation site profiling with call site analysis, and inferring the calling context through a context identifier that is based on the stack height. By using the stack height, we avoid the significant overhead of capturing the calling stack using instrumentation to track the stack epilogue and prologue. We therefore hypothesize that:

> *Knowing the context identifier characterized by the currently executing function, and stack height in bytes mostly identifies the calling context we need to precisely predict the size of data structures.*

4.5. Inferring the Calling Context

Inferring the call chain using the stack height is discussed in the context of C/C++ in the literature with no evaluation for dynamic languages (Mytkowicz et al. 2009; Maas et al. 2020b). Therefore, to use the stack height hypothesis for a dynamic language, we implemented a tool to profile Python applications, logging the program's context identifier and call paths, first to prove the assumption, all done without annotating the program. For further profiling we do not include the call paths, we only use a context identifier that is based on the call site symbol and stack height.

Listing 8 shows an example of logs generated from profiling a hypothetical target program, while Table 5 shows profiling statistics. We capture the program log number, file name, line number of a call site, the function name at the call site, the stack height, the call path and a tuple with information about any data structures associated with the call site. The tuple of data structure details contains the collection type, the variable name of the collection, the size and allocation site. Call sites that do not involve any data structure use, will not have the data structure fields in their log.

```
1, target.py:45, add, 4, <module> -> <module> ->
    add -> trace_function,
    ([Tuple, add:4, 4])
2, target.py:6, tul, 5, <module> -> <module> ->
    add -> tul ->
    trace_function, ([List, tul_list:22, 4])
3, target.py:21, calculate_factorial, 6, <module>
     -> <module> -> add ->
    tul -> calculate_factorial
4, target.py:31, n_queens, 7, <module> -> <module
    > -> add -> tul ->
    calculate_factorial -> n_queens ->
    trace_function,
    ([Tuple, my_tuple:35, 4], [List, item_list
    :36, 4])
5, target.py:7, mul, 5, <module> -> <module> ->
    add -> mul ->
    trace_function
6, target.py:46, sub, 4, <module> -> <module> ->
    sub -> trace_function,
    ([Tuple, sub:11, 4])
```

Listing 8 Logs Generated from the Profiler

The stack height in this case refers to the distance between the current stack pointer and the top of stack in bytes. By combining the stack height with the current return address (i.e., local call site) and the data structure size, we are able to get a fingerprint of a particular stack trace. This corresponds to the current number of frames on the call stack in Python, calculated by inspecting code objects of a frame.

For the call paths, we build the Python interpreter with frame pointers, accessing the frames to read the calls in a file. To log collections in the body of a function call, the first challenge we encountered was that the `exec()` function produces the source code of a function in the form of text if we inspect the source lines of the frames instead of code objects, which makes it hard to identify any nodes of interest for our profiling. We instead through some form of static analysis parse the source code, generating an abstract syntax tree (AST), then walking the AST to identify list, set and dictionary nodes.

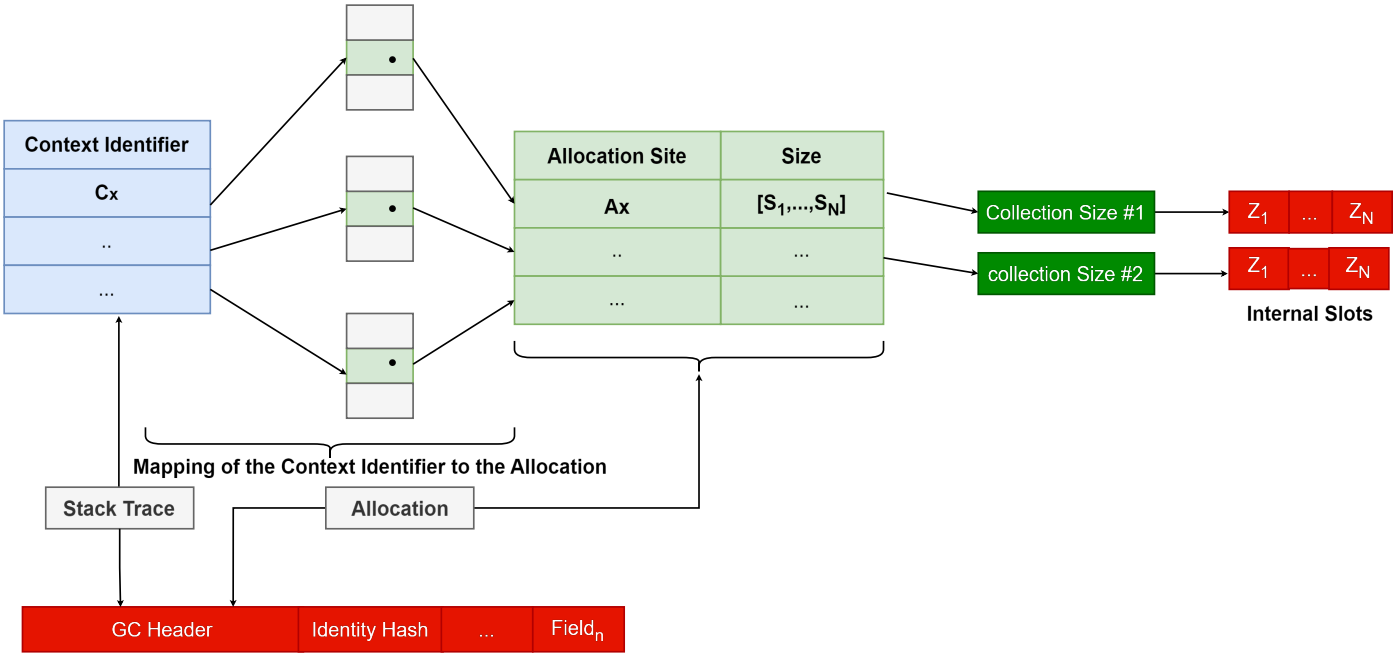


Figure 11 Context Aware Collection Size Profiling

	stmts	stmts cov.	fns	fns cov.	calls
ai	98	6	3	3	6
call_simple	198	84	6	5	84
html5lib	59	5	5	5	5
nbody	143	8	4	3	8
pickle	350	9	6	4	9
regex_compile	92	5	2	2	5
richards	427	57	19	17	57
unpack_sequence	458	7	4	4	7
regex_effbot	147	47	4	4	47

Table 5 Call Site Analysis for Python – statistics generated from the profiling tool

4.6. Accuracy of Context Identifiers

The generated logs are not completely accurate and thereby require disambiguation in some cases because of the lack of precision in mapping between context identifiers and call paths, i.e., multiple distinct call paths mapping to the same (function_name, stack height) pair. We used the methods of disambiguation discussed in the literature which include: *active record resizing (ARR)*, *call site wrapping (CSW)* and *function cloning (FC)* (Mytkowicz et al. 2009). The following patterns of ambiguity are handled in our benchmarks:

$$X \to Y \to Z \qquad X \to W \to Z \tag{1}$$

$$X \to Y \to Z \qquad Y \to X \to Z \tag{2}$$

$$X \to Y \to X \to X \to Z \qquad X \to X \to Y \to X \to Z \tag{3}$$

The first pattern involves two profile items with the same (function_name, stack height) but the call paths have a unique call, i.e., Y in the first call path and W in the other. To disambiguate this case, the intermediate operand of the instruction is modified, changing the function's active record size (ARR) to account for function's local variables on the stack, which changes the stack height of the path, making it precise.

The second pattern is harder, since the same (function_name, stack height) pair matches the same call paths, and active record resizing does not help with this. This case is managed by replacing a call on one of the edges with a wrapper function that then calls the original function using the call site wrapping technique. The wrapper function adds its own active record, thereby changing the stack height.

The third case contains more duplicate calls and can not be handled by call site wrapping; instead, it is handled by replacing a call to a duplicate function with a call to a copy of the function. The function clone contains disambiguation, in this example, clone $X \to X'$ so that `X'` wraps its call to `Y`. The additional disambiguation in the clone changes the stack height.

Figure 12 shows the degree of precision before and after the disambiguation, for the three disambiguation methodologies. We used nine benchmarks from the standard PyPy benchmarks described in Table 1. The profiler uses these benchmarks as target programs and generates four logs in this experiment, one without disambiguation, one after applying only active record resizing, another after applying only function cloning and call site wrapping, and the last log file is the one after applying all the three techniques. The hardware specifications for this experiment are detailed in Table 3. We use the geometric mean for all mean computations of the results.

Before disambiguation, we observe the precision of 43. 3% by means of the benchmarks we used. The highest level of precision is 60% seen in four benchmarks, namely *html5lib*, *regex_compile*, *unpack_sequence* and *regex_effbot*. The lowest

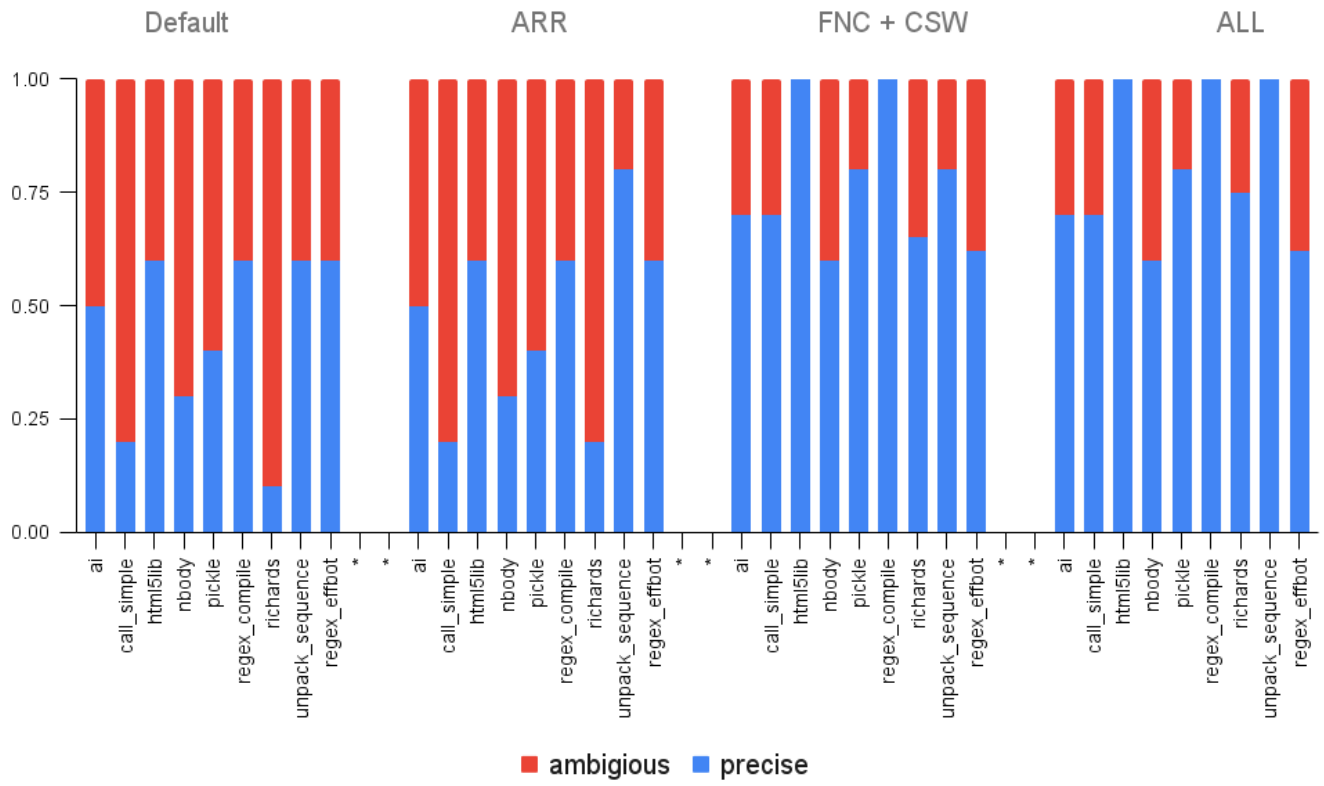


Figure 12 Proving the Stack Height Hypothesis – the profiler generates four logs in this experiment, one without disambiguation, one after applying only active record resizing, another after applying only function cloning and call site wrapping and the last log file is the one after applying all the three techniques. The configurations correspond to the disambiguation methods; *active record resizing (ARR)* : the intermediate operand of the instruction is modified, changing the function's active record size; *call site wrapping (CSW)*: replacing a call on one of the edges with a wrapper function that then calls the original function; and *function cloning (FC)*: replacing a call to a duplicate function with a call to a copy of the function. We observe that the context identifier can uniquely identify a call path with a 79.9% accuracy level by mean.

level of precision is observed for the *richards* and *call_simple* benchmarks, at a degree of 10% and 20%, respectively. The general observation is that the more complex the workload, the higher the chances of ambiguity; as an example, the *richards* benchmark is more complex compared to the rest of the benchmarks.

Active record resizing was only beneficial for *richards* and *unpack_sequence* benchmarks with improvements in precision of 10% and 20%, respectively. Call site wrapping was only beneficial after function cloning, improving the precision of the rest of the benchmarks by 2%–100%. The benchmarks *html5lib* and *regex_compile* had 100% precision and the lowest degree of precision was 60% for the *nbody* benchmark after complete disambiguation.

After applying all the disambiguation techniques, we observe that by mean the context identifier (executing function and stack height) can uniquely identify a call path with a 79.9% accuracy level for the Python workloads we used. An 80% mean precision level justifies our use of the stack hypothesis assumptions in inferring the calling context for the presizing optimization.

4.7. Optimal Size Prediction

The profiling information collected is stored on a globally accessible context map. The context identifier and the allocation site for each object are also installed in the object header to help with prediction (this is not needed if all prediction were to be done offline). In the RPython framework, the object layout is modified to store the allocation context. We store both the context identifier and the allocation site.

4.7.1. Installing the Context in the Object Header

The structure of the RPython object as shown in Figure 11 consists of one word for the GC header and vtable pointer, the identity hash field and the rest of the bits are used for static data in classes. In practice, there are a few spare bits on 64-bit machines, which we can use for the default *incminimark* garbage collector, and we mostly reuse some bits from the header to complement the spare bits. However, other times, we use extra bits in the object header if we cannot reuse the hash field bits. We maintain one more bit as a flag to signal whether profile is set or not, as shown in Figure 13.

Python objects are usually assigned a hash field in the

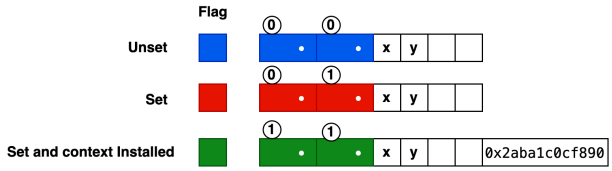


Figure 13 Installing the Context in the RPython Object Header – when a collection object is allocated, memory is allocated and reserved for the context identifier. The set and unset is for the hash field

CPython and earlier versions of PyPy interpreters, since these objects could be stored in a dictionary by the program. PyPy modified this feature later to only create a hash field when the object's identity hash is used, but once set, it can not be reset, so it is possible that some objects whose identity hash is not taken still have a hash field assigned to them.

We work on the assumption that all objects in Python are assigned a hash field as supported by the reference CPython implementation, and reuse the bits of the hash field to store the context identifier, if the object does not have its identity hash taken. This is a common case in Python, where just a fraction of Python objects have their identity hash taken. When the object later requires its hash field, the profile information is overwritten. This allows us to not enlarge the object header for most objects, but for objects whose hash-field is taken, we use any spare bits and only enlarge the object header by a few bits, introducing a memory trade-off in those cases.

The garbage collector is thus modified upon allocation to install context data in the hash field if the object does not use its hash field. We maintain an extra flag as part of the GC header to signal if the object has profiling data installed. The new version of the object has its flag for the profile set, and is extended, allocating space for the profile information that is to be written at the end of the object header, as shown in Figure 13. Reusing bits allows us to not use more memory in the object header for most objects, but for dictionaries, we may use some extra bits to store this information.

4.7.2. Reading and Writing to the Context Map

Upon allocation, data from a collection object x' on the number of collections, the allocation site, A_x, the context identifier, C_x, and any observed sizes, S_N, are recorded in the context map as shown in Figure 11. The *context map* presented in Figure 11 contains the number of objects organized by allocation context and item size. For example, if an object is allocated in the allocation context $C_x + t$ with size, S_0, where t is a constant, the context map is updated to increment the number in the cell corresponding to the row $C_x + t$ and the column S_1 (one more object with size S_1). An allocation site can have several observed sizes for a given context, $S_0, .., S_N$.

4.8. Optimal Size Estimation

The optimal size of the collection is determined by reading the context map to access the maximum, mean, or median size of the collection size that matches an object's calling context. We thus support three strategies, *max*, *mean* and *median* for the maximum, mean, and median sizes, respectively.

Max Strategy: This strategy uses the latest maximum sizes of an allocation site to determine the presize of a collection. A threshold is maintained P_{th}, where if the current presize exceeds this threshold, the new maximum is set as a presize. It suffers from overestimation should most collections end up being smaller. In the worst case, unused slots may be used, but this strategy has the advantage of still reducing the overhead of expansion operations. In our analysis, we do not shrink the internal slots in case the collection is smaller, so there is no shrinking overhead.

Mean Strategy: Rather than naively taking the maximum value of observed sizes from profiles, this strategy uses the geometric mean value of the sizes of the observed list for an allocation site. The presize value is also likely to be high like in the max strategy, hence unused slots for smaller collections, but the impact may be less. This strategy can suffer from the overhead of expansion operations for larger collections, but the impact is less than if there was no pre-sizing at all. If anything, the performance impact of this strategy is closer to the max strategy.

Median Strategy: The presize using this strategy is calculated from the median of the sizes observed for an allocation site. The presize is smaller than in the max and mean strategies, hence fewer cases of unused slots in the best case scenario. However, because of the underestimation, it is likely to suffer from the overhead of expansion operations for larger collections. The strategy produces the best results for most experiments discussed in Section 4.10.

4.9. Implementation

We modify the PyPy3.9 branch, which was the latest version at the time of writing this paper. The default RPython *incminimark* GC is modified to install profiling information on allocation, and to read from the global context map to presize data structures. The *incminimark* GC is a generational GC, so we naturally identify collection objects in nursery allocation when they are first allocated, putting a flag on these objects asking for the `id` or `identityhash`. This flag is different from the profiling flag, which is set to signal if presize profiling is on or off.

The memory for these objects is initialized and reserved at the time of object creation. The nursery objects with this flag are tracked and assigned destination locations during the closest nursery collection cycle. The *incminimark* GC supports resizing of arrays in the nursery so that they consume less memory when moved to tenure, but objects flagged to use the `id` for our case are not resized with this feature, as we can lose pre-allocated memory in these objects.

When moving objects, we also first visit all the objects we marked as susceptible for our profiling and have the flag that signals using the `id` field, this traversal has an eminent cost quantified later in Section 4.10. We then copy over the objects while overwriting the relevant header fields with the allocation

and context information. We access the global context map to allocate the optimal size. The hash value is not static and changes the next time the object needs to move again. Our implementation relies on generational and moving collectors, but it is possible to have an alternate implementation of our methodology in other contexts.

4.10. Evaluation

The evaluation discussion in this section has two main goals — we present results on execution time and peak memory usage — as a way of demonstrating the benefits of our technique. We also explore the allocation sites in the benchmarks used. Profiling has overhead, so we focus our evaluation on profile guided optimization, where we use information from a previous run, a use case that is useful for when the optimization can be used during deployment. There is overhead involved with reading the off-line information to perform the actual profiling but it is not significant; we have left this overhead investigation for our future work when we have fully explored the online option.

4.10.1. Methodology The default PyPy *Incminimark* incremental garbage collector is used and four pre-sizing configurations are compared in this evaluation; the baseline is the main upstream PyPy branch without pre-sizing optimization; the *max*, *mean* and *median* settings correspond to the heuristics for choosing the presize as either *maximum*, *mean* or *median* depending on the several sizes observed at an allocation site during profiling. Even in the baseline, we assume that all collection objects are assigned a hash field; if we used the default settings where the hash field is assigned to a few objects, then the results would be different. All benchmarks are run on a machine with specifications described in Table 3.

4.10.2. Workload Description The official Python benchmark suite used in Section 4.4 was not sufficient for this evaluation. We instead use workloads recommended and open sourced for cross language benchmarking (Marr 2016), to demonstrate the significance of the approach. The workloads described next are chosen because they use both a small and a large number of collections compared to the official Python benchmark suite; of these, four benchmarks are macro while two of them are micro.

The *CD* (macro workload) benchmark aims to simulate airplane collision detection and was designed to evaluate real-time JVMs. We use the Python implementation of the workload with a setting of two planes and 5000 iterations. *DeltaBlue* (macro workload) is an algorithm that implements a constraint solver and runs for 300 iterations. *Havlak* (macro workload) contains a loop recognition algorithm with a number of collections that serves as a good use case for our approach, we run it for 20 iterations. The *JSON* (macro workload), *Permute* (micro workload), and *List* (micro workload) implement JSON string parsing, permutations and list operations, respectively. These benchmarks are all run for 5000 iterations.

4.10.3. Observed Allocation Sites Figure 14 shows a breakdown of the allocation sites and the number of collections observed at these sites. To understand the plots in this figure, it is worth noting that the line dividing the box into two represents the median value and shows that 50% of the data lies on the left-hand side of the median value and 50% lies on the right-hand side. The left and right edges of the box represent the lower and upper quartiles. The lower quartile shows the value at which the first 25% of the data falls. The upper quartile shows that 25% of the data is to the right of the upper quartile value. The values at the ends of the horizontal lines are the upper and lower values of the data, while single points on the diagram show any outliers.

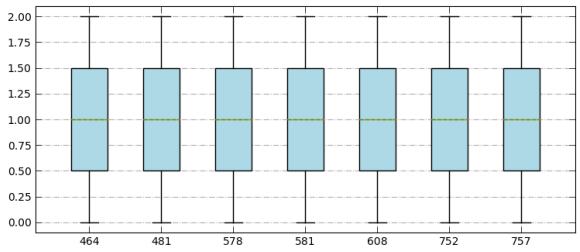
The allocation sites correspond to line numbers in the source code. We show the maximum, mean, and median values for the allocation site as well. We do not show the types of collections at the allocation sites, but for a breakdown, the benchmarks *DeltaBlue*, *JSON*, *Lists* and *Permute* only use lists. The *CD* benchmark uses only vectors. The *Havlak* benchmark uses a list at allocation site 318, a set at allocation site 160, a dictionary at allocation site 316, and vectors at allocation sites 315, 314 and 404.

The values without bars are mostly fixed size collections and smaller collection numbers, especially for the *Lists* and *Permute* workloads. The macro benchmarks *CD*, *DeltaBlue*, *Havlak*, and *JSON* contribute the highest number of collections, since they are macro workloads. The different presize strategies generally benefit different workloads depending on the distribution of sizes observed at the allocation sites.

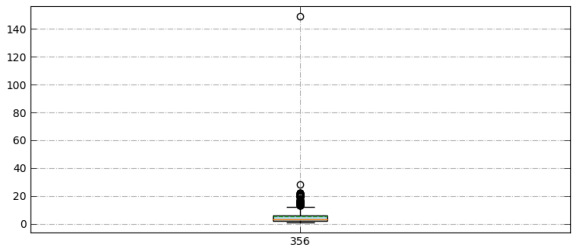
4.10.4. Discussion of Results In terms of *execution time* as shown in Table 6, the max configuration achieves the best speed-up in wall clock time of 4% for workload *CD* followed by 2% and 1% for the median and mean strategies compared to baseline. This is the only workload where choosing the maximum observed size at an allocation site has the best results in execution time. The *CD* workload is stable across call sites, mainly depending on the number of planes involved in the simulation. For the rest of the workloads, we observe that the median setting works best due to more distribution and variation in the sizes observed at the allocation sites, which means that using the median value is a good compromise. For example, the *JSON*, *Havlak*, *List*, and *Permute* workloads have the best speed-ups of 6%, 3%, 3%, and 12%, respectively, compared to the baseline with the median strategy. The *DeltaBlue* benchmark experiences the highest gains of 35% overall also with the median strategy because it is collection intensive.

Memory-wise, when we reuse some header bits to store the allocation context data, our approach can reduce memory consumption due to a reduction in memory overallocation from the presizing in objects. As shown in Table 6, the best memory savings are with the median strategy across all benchmarks. The memory results show the memory savings of the technique without accounting for the header overhead. Profiling information does not have to be stored in the header; existing techniques such as memorabilia can be used instead.

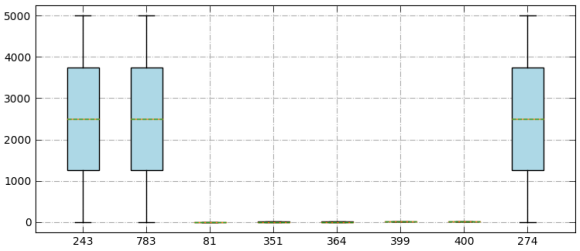
4.10.5. The Overhead of using the Profiles Table 7 shows the overhead of accessing the object header and the context map while presizing; for workloads from the standard PyPy benchmark suite. We use standard benchmarks here with a



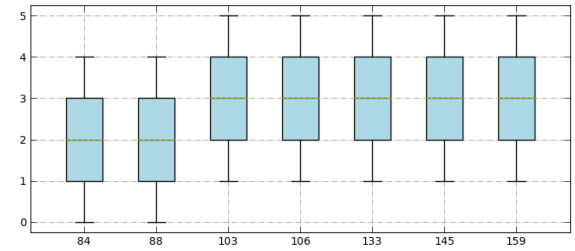
(a) Aircraft Collision Detection (CD)



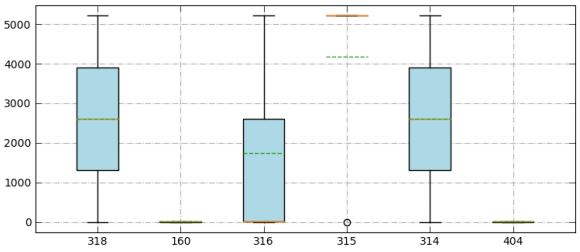
(b) JSON



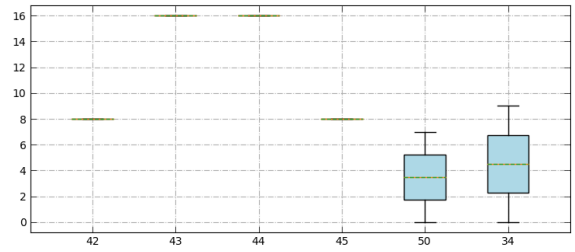
(c) DeltaBlue



(d) Lists



(e) Havlak



(f) Permute

Figure 14 Number of Observed List Sizes per Line Number of Allocation Sites – the Y (collection count) and X (line number) axes show the size of collections and allocation sites respectively, while the orange and green lines show the mean and median

	Strategy		
	Max	**Mean**	**Median**
Execution Speed			
collision detection	0.965 (0.954, 0.971)	0.985 (0.973, 0.999)	0.981 (0.978, 1.014)
json	0.963 (0.958, 0.970)	0.962 (0.953, 1.013)	0.943 (0.939, 0.976)
havlak	0.983 (0.977, 0.987)	0.979 (0.968, 0.997)	0.975 (0.975, 0.987)
deltablue	0.987 (0.965, 0.996)	0.916 (0.908, 0.930)	0.652 (0.650, 0.820)
lists	0.987 (0.978, 1.000)	0.980 (0.973, 1.001)	0.973 (0.964, 1.000)
permute	0.998 (0.996, 1.002)	0.998 (0.995, 1.001)	0.882 (0.874, 0.906)
min	0.963	0.916	0.652
max	0.998	0.998	0.981
geomean	0.980	0.969	0.892
Peak Memory			
collision detection	0.919 (0.919, 1.175)	0.903 (0.850, 1.177)	0.962 (0.850, 1.037)
json	0.973 (0.969, 1.311)	0.972 (0.963, 1.322)	0.973 (0.943, 1.032)
havlak	0.685 (0.320, 1.003)	0.984 (0.975, 1.006)	0.991 (0.975, 1.001)
deltablue	0.972 (0.960 ,1.060)	0.943 (0.932, 1.378)	0.978 (0.951, 1.030)
lists	0.915 (0.868, 1.086)	0.910 (0.785, 1.136)	1.004 (0.973, 1.023)
permute	0.916 (0.890, 1.144)	0.902 (0.886, 1.148)	1.006 (0.902, 1.143)
min	0.685	0.902	0.962
max	0.973	0.984	1.006
geomean	0.891	0.935	0.985

Table 6 Performance Gains and Memory Savings for Context Aware Presizing – lower values are better, all results are shown as ratios normalized to the baselines and aggregated based on the geometric mean across 30 invocations. The error values correspond to 95% confidence intervals; for details see Appendix A. There is a 11% performance improvement for the best median strategy and memory savings of 11% for the best max strategy by geometric mean.

mix of small and large collections. The goal is to observe the overhead of the operations that access profile data.

The numbers presented in Table 7 do not directly compare to the evaluation in the previous Section 4.10.4 due to the cost of instrumentation used, but we also mostly consider installation of profiles in the object header an offline task in our evaluation.

We present results for both reading and writing to the object header, as well as reading from the context map. We do not show the overhead of writing to the context map.

Our approach spends the most overhead on reading from the context map (maximum of 7.7%). Reading the object header is the least costly (maximum overhead of 4.9%). The total overhead of the three operations can be as high as 25% and 6% by mean. Due to our evaluation methodology, the overhead presented here may include overhead of the instrumentation used.

Our main contribution is the use of the stack height to capture the calling context required for pre-sizing accuracy. The way profiling information is stored can change. We do not therefore consider the overhead presented here as a bottleneck to the underlying technique, as alternative approaches can be used instead of the object header and the context map.

5. Related Work

We present related work on presizing and type optimizations for heterogeneous collections in this section to contextualize our contributions relative to existing work.

5.1. Type Optimizations for Collections

Much work exists that uses techniques like tagging, and new frameworks (Chambers et al. 1989; Bigot & Debray 1997; Kataoka et al. 2018; Holkner & Harland 2009) to represent the types of data in dynamic languages (Ureche et al. 2015), but regardless of these efforts, we have not been able, as a field, to completely avoid boxing in dynamic languages; efforts to date have targeted specific use cases. This work extends existing work to handle type representation of heterogeneous collections, which the literature has not solved.

Type optimization for collections has been researched with various levels of success, the most recent, in Graal (Henning et al. 2020; De Wael et al. 2015); storage strategies were studied to aid list presizing, the Pharo language was modified with similar optimizations to reduce resource usage (Bergel et al. 2018; Marr & Daloze 2018b), even parallelization algorithms exist for collections in dynamic languages (Daloze et al. 2018) but none of these optimize heterogeneous collections.

Storage strategies (Bolz et al. 2013) and element kinds (Clifford et al. 2015) discussed in Section 2.2.2 are similar optimizations to the ones we describe in this paper. We extend their use with modifications to support cases they do not support by efficiently handling heterogeneous collections. Storage strategies were made language-agnostic by Pape et al. (Hudson et al. 1991) in their work on the RPython translation toolchain; we also support the type-based stores mechanism in the RPython framework.

Memory restructuring is also not new; Aleksandros et al. (Tasos et al. 2020) reshape the layout for objects to address locality and the Collection Skeleton technique (Franke et al. 2022) also provides flexibility in handling collections, but neither handle unboxing. D'Souza et al. solve parametric polymorphism (D'Souza et al. 2023) instead.

5.2. Presizing

Clifford et al., use allocation-site-based profiling, storing the information with garbage collection assisted approaches in one paper (Clifford et al. 2015) where a temporary object called a *memento* is allocated close to the object being profiled, created at the object's allocation site, and survives long enough just to survive the next GC cycle. The concept of mementos was demonstrated in V8 but has no further indication of being used upstream.

Mementos were adopted by Henning et al., with some modifications in GraalPython due to the limited level of access to the GC (Henning et al. 2020) in the runtime. We solve the limitations acknowledged in these papers by inferring the calling context based on the stack height in this work.

6. Conclusion

In summary, we propose two techniques that optimize collections in dynamic languages. This section summarizes the key contributions for the explored topics of research.

6.1. Type Optimizations for Collections

The *type-based stores* technique described in Section 3 restructures the layout of heterogeneous collections, splitting the collection into contiguous memory partitions and unboxing the items in the partitions.

Storage strategies will benefit workloads with heavy use of homogeneous collections better than our technique. This is because we incur overhead due to extra checking before and during splitting. Our technique is especially beneficial for large heterogeneous collections or where a large homogeneous collection becomes heterogeneous.

As shown in the evaluation, we guarantee gains for workloads with large heterogeneous collections (11 of the 16 benchmarks in our experimentation). Type-based stores can complement type speculation optimizations, because objects are stored in their concrete types. Type speculation benefits from our technique by being able to know the types of the objects involved.

In general, applications with heterogeneous data structures with primitives will benefit from this optimization. The number of data types may have less impact since we support integers, floats, and strings. In fact integers and floats share a store, in which case, by default only two stores will ever be created for any heterogeneous collection, hence the number of data types is unimportant.

Storing objects in an unboxed form benefits even non-JIT interpreters like CPython in both speed and memory consumption. However, JIT-based interpreters gain more because JIT compilers benefit further from using the type information for further optimization, like type speculation mentioned in V8 earlier.

	Header			Map	All
Benchmark	**WR**	**RD**	**WRD**	**RD**	**Total**
ai	1.017 (1.007, 1.029)	1.014 (1.006, 1.023)	1.030 (1.014, 1.046)	1.010 (0.999, 1.022)	1.070 (1.031, 1.109)
call simple	0.992 (0.971, 1.018)	0.985 (0.967, 1.006)	0.976 (0.944, 1.013)	0.984 (0.962, 1.011)	0.929 (0.854, 1.020)
nbody	0.999 (0.995, 1.003)	1.004 (1.000, 1.007)	1.003 (0.996, 1.009)	1.001 (0.997, 1.006)	1.007 (0.990, 1.019)
pickle	1.010 (1.002, 1.019)	1.018 (1.008, 1.028)	1.028 (1.014, 1.043)	1.009 (1.001, 1.018)	1.064 (1.035, 1.097)
richards	1.012 (1.000, 1.022)	1.004 (0.993, 1.014)	1.016 (1.003, 1.028)	0.996 (0.988, 1.002)	1.026 (1.001, 1.053)
threading	1.009 (0.994, 1.022)	1.005 (0.990, 1.021)	1.013 (0.991, 1.036)	1.007 (0.994, 1.018)	1.031 (0.981, 1.080)
unpack sequence	1.029 (1.016, 1.042)	1.045 (1.025, 1.058)	1.073 (1.041, 1.096)	1.045 (1.025, 1.058)	1.189 (1.118, 1.241)
pystone	0.980 (0.940, 1.026)	1.016 (0.990, 1.049)	0.995 (0.941, 1.056)	0.990 (0.949, 1.028)	0.968 (0.855, 1.108)
binary tree	1.013 (1.010, 1.016)	1.009 (1.006, 1.012)	1.022 (1.018, 1.026)	1.010 (1.008, 1.012)	1.054 (1.047, 1.061)
chaos	1.009 (1.003, 1.014)	1.012 (1.005, 1.019)	1.020 (1.012, 1.029)	1.011 (1.005, 1.016)	1.051 (1.033, 1.073)
delta	1.018 (1.009, 1.029)	1.012 (1.006, 1.018)	1.030 (1.020, 1.040)	1.005 (1.000, 1.015)	1.064 (1.043, 1.088)
float	1.018 (1.014, 1.022)	1.035 (1.025, 1.047)	1.055 (1.039, 1.067)	1.036 (1.021, 1.047)	1.144 (1.107, 1.166)
go	1.017 (1.011, 1.023)	1.018 (1.013, 1.022)	1.035 (1.027, 1.042)	1.077 (1.010, 1.301)	1.147 (1.073, 1.382)
meteor contest	1.011 (1.000, 1.021)	1.010 (1.000, 1.020)	1.021 (1.006, 1.034)	1.020 (1.009, 1.029)	1.061 (1.026, 1.092)
pidits	1.001 (1.000, 1.001)	1.010 (1.009, 1.011)	1.010 (1.010, 1.011)	1.015 (1.014, 1.015)	1.035 (1.035, 1.036)
spectral norm	1.053 (1.048, 1.060)	1.049 (1.043, 1.053)	1.102 (1.092, 1.110)	1.051 (1.047, 1.054)	1.254 (1.232, 1.274)
telco	1.008 (1.001, 1.014)	1.002 (0.993, 1.009)	1.010 (0.997, 1.020)	1.005 (0.997, 1.010)	1.024 (0.996, 1.047)
min	0.980	0.985	0.976	0.984	0.929
max	1.053	1.049	1.073	1.077	1.254
geomean	1.011	1.014	1.025	1.016	1.063

Table 7 The Overhead of using the Profiles – lower values are better, values presented as ratios relative to the PyPy baseline within the given error margins. The columns (WR) and (RD) refer to write and read respectively. (WRD) refers to both read and write. We also present the total overhead for all operations, all values are normalized to the baseline, aggregated across 30 invocations

Regarding fragmentation, when objects are stored in an unboxed form, primitives are now not individually heap allocated objects; hence less GC pressure, which is why we observe speedups for most benchmarks. In addition to creating several objects through splitting, our technique does not manage GC allocation; fragmentation should be handled by the underlying GC allocator like any other object. Splitting in our type-based stores technique is high-level, at least in the garbage collection sense.

6.2. Presizing

In Section 4, we propose a technique that optimizes data structure resizing in dynamic languages. It addresses the branching and allocation context challenges for data structure pre-sizing, by inferring the allocation context through a profiled call site symbol and stack height, along with a profiled allocation site and list size information.

The stack height assumption is not always true, as the same call path can have the same call site symbol and stack height, which is a threat to validity in our work. We remedy this with existing disambiguation approaches to reduce the duplication of the mappings between call paths and the stack height.

Our approach will not accurately perform presizing for any cases we cannot disambiguate but as acknowledged in existing work (Mytkowicz et al. 2009) and our findings in Section 4.6, this assumption is not always true.

We observed between 4%—30% performance gains and an average 16% of memory savings while incurring profiling overhead that is as high as 25% for Python applications when context-aware presizing is applied to the PyPy implementation.

We implemented an offline version of the technique and the results did not reflect the full profile overhead. Our main contribution is the use of the stack height to capture the calling context required for presizing accuracy. The way profiling information is stored can change. We do not therefore consider the overhead presented here as a bottleneck to the underlying technique, as alternative approaches can be used instead of the object header and the context map.

7. Future Work

We discuss and propose potential future directions for the type-based stores and context aware presizing techniques in this section.

7.1. Type-based Stores

The language-independent type-based stores and storage strategies techniques can be extended to a library that is not based on the RPython toolchain for wider adoption by other languages, where using the toolchain is not an option.

7.2. Context Aware Presizing

The presizing technique needs to address the profiling overhead since such optimizations are more beneficial if they are online and dynamic.

We also believe that there is a relationship between stack height and live size, a key insight in driving garbage collection optimizations. For example, the square-root rule algorithm (Kirisame et al. 2022) proposed by Kirisame et al. can use the height of the stack to automatically sizing the heap instead of the live size, because the live size cannot be measured until after a garbage collection cycle. Optimizations of this nature are best evaluated on realistic workloads to observe their impact on pause time, cache behavior, etc.

Acknowledgments

This research was conducted within the IBM Centre for Advanced Studies–Atlantic, Faculty of Computer Science, University of New Brunswick. The authors are grateful for the colleagues and facilities of CAS–Atlantic in supporting our research. The authors would like to acknowledge the funding support provided by the Atlantic Canada Opportunities Agency (ACOA) through the Atlantic Innovation Fund (AIF) program. Furthermore, we would also like to thank the New Brunswick Innovation Foundation for contributing to this project.

References

Bauman, S., Bolz, C. F., Hirschfeld, R., Kirilichev, V., Pape, T., Siek, J. G., & Tobin-Hochstadt, S. (2015). Pycket: a tracing jit for a functional language. In *Proceedings of the 20th acm sigplan international conference on functional programming* (p. 22–34). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2784731.2784740 doi: 10.1145/2784731.2784740

Bergel, A., Infante, A., Maass, S., & Alcocer, J. P. S. (2018). Reducing resource consumption of expandable collections: The pharo case. *Science of Computer Programming*, *161*, 34–56. Retrieved from https://www.sciencedirect.com/science/article/pii/S0167642317302940 doi: https://doi.org/10.1016/j.scico.2017.12.009

Bigot, P. A., & Debray, S. K. (1997). A simple approach to supporting untagged objects in dynamically typed languages. *The Journal of Logic Programming*, *32*(1), 25–47.

Bolz, C. F., Diekmann, L., & Tratt, L. (2013). Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 acm sigplan international conference on object oriented programming systems languages & applications* (p. 167–182). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2509136.2509531 doi: 10.1145/2509136.2509531

Bruno, R., Patricio, D., Simão, J., Veiga, L., & Ferreira, P. (2019). Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the fourteenth eurosys conference 2019*. New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3302424.3303988 doi: 10.1145/3302424.3303988

Bynens, M. (2017). *Elements kinds in v8.* https://v8.dev/blog/elements-kinds. (Accessed: 2021-09-10)

Chakraborty, S. (2023). *The computer language benchmarks game.* https://sschakraborty.github.io/benchmark/index.html. (Accessed: 2023-03-14)

Chambers, C., Ungar, D., & Lee, E. (1989). An efficient implementation of self a dynamically-typed object-oriented

language based on prototypes. *ACM Sigplan Notices*, *24*(10), 49–70.

Chernick, M. R. (2011). *Bootstrap methods: A guide for practitioners and researchers*. John Wiley & Sons.

Clifford, D., Payer, H., Stanton, M., & Titzer, B. L. (2015). Memento mori: Dynamic allocation-site-based optimizations. In *Proceedings of the 2015 international symposium on memory management* (p. 105–117). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2754169.2754181 doi: 10.1145/2754169.2754181

Contributors, P. (2012). *Pypy benchmarks.* https://github.com/hg-mirrors/pypy_benchmarks. (Accessed: 2019-06-14)

Daloze, B., Tal, A., Marr, S., Mössenböck, H., & Petrank, E. (2018, October). Parallelization of dynamic languages: Synchronizing built-in collections. *Proc. ACM Program. Lang.*, *2*(OOPSLA). Retrieved from https://doi.org/10.1145/3276478 doi: 10.1145/3276478

De Wael, M., Marr, S., De Koster, J., Sartor, J. B., & De Meuter, W. (2015). Just-in-time data structures. In *2015 acm international symposium on new ideas, new paradigms, and reflections on programming and software (onward!)* (pp. 61–75).

Diekmann, L. (2012). Memory optimizations for data types in dynamic languages. *Masterarbeit of Institut Fur Informatik, Softwaretechnik und Programmiersprachen, Heinrich Heine Universitat Dusseldorf*, 1–51.

D'Souza, M., You, J., Lhoták, O., & Prokopec, A. (2023). Tastytruffle: Just-in-time specialization of parametric polymorphism. *Proceedings of the ACM on Programming Languages*, *7*(OOPSLA2), 1561–1588.

Franke, B., Li, Z., Morton, M., & Steuwer, M. (2022). Collection skeletons: Declarative abstractions for data collections. In *Proceedings of the 15th acm sigplan international conference on software language engineering* (p. 189–201). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3567512.3567528 doi: 10.1145/3567512.3567528

Gaynor, A. (2013). *A high performance ruby, written in rpython.* https://github.com/topazproject/topaz. (Accessed: 2021-09-10)

Gudeman, D. A. (1993). Representing type information in dynamically typed languages..

Henning, J., Felgentreff, T., Niephaus, F., & Hirschfeld, R. (2020). Toward presizing and pretransitioning strategies for graalpython. In *Conference companion of the 4th international conference on art, science, and engineering of programming* (p. 41–45). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3397537.3397564 doi: 10.1145/3397537.3397564

Holkner, A., & Harland, J. (2009). Evaluating the dynamic behaviour of python applications. In *Proceedings of the thirty-second australasian conference on computer science-volume 91* (pp. 19–28).

Hudson, R. L., Moss, J., Diwan, A., & Weight, C. F. (1991). A language-independent garbage collector toolkit. *University of Massachusetts*.

Ilbeyi, B. (2019). *Co-optimizing hardware design and meta-tracing just-in-time compilation* (Unpublished doctoral dissertation). Cornell University.

Kataoka, T., Ugawa, T., & Iwasaki, H. (2018). A framework for constructing javascript virtual machines with customized datatype representations. In *Proceedings of the 33rd annual acm symposium on applied computing* (p. 1238–1247). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3167132.3167266 doi: 10.1145/3167132.3167266

Kirisame, M., Shenoy, P., & Panchekha, P. (2022). Optimal Heap Limits for Reducing Browser Memory Use. *Proc. ACM Program. Lang.*, *6*(OOPSLA2), 986–1006. Retrieved from https://doi.org/10.1145/3563323 doi: 10.1145/3563323

Kirkwood, T. B. L. (1979). Geometric means and measures of dispersion. *Biometrics*, *35*(4), 908–909. Retrieved 2024-09-14, from http://www.jstor.org/stable/2530139

Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., & Raffel, C. (2020a). Learning-based memory allocation for c++ server workloads. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems* (p. 541–556). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3373376.3378525 doi: 10.1145/3373376.3378525

Maas, M., Andersen, D. G., Isard, M., Javanmard, M. M., McKinley, K. S., & Raffel, C. (2020b). Learning-based memory allocation for c++ server workloads. In *Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems* (p. 541–556). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3373376.3378525 doi: 10.1145/3373376.3378525

Marr, S. (2016). *Are we fast yet? comparing language implementations with objects, closures, and arrays.* https://github.com/smarr/are-we-fast-yet. (Accessed: 2019-06-14)

Marr, S., & Daloze, B. (2018a). Few versatile vs. many specialized collections: How to design a collection library for exploratory programming? In *Companion proceedings of the 2nd international conference on the art, science, and engineering of programming* (p. 135–143). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3191697.3214334 doi: 10.1145/3191697.3214334

Marr, S., & Daloze, B. (2018b). Few versatile vs. many specialized collections: how to design a collection library for exploratory programming? In *Companion proceedings of the 2nd international conference on the art, science, and engineering of programming* (pp. 135–143).

Mytkowicz, T., Coughlin, D., & Diwan, A. (2009, oct). Inferred call path profiling. *SIGPLAN Not.*, *44*(10), 175–190. Retrieved from https://doi.org/10.1145/1639949.1640102 doi: 10.1145/1639949.1640102

Pape, T., Felgentreff, T., Hirschfeld, R., Gulenko, A., & Bolz, C. F. (2015, October). Language-independent storage strategies for tracing-jit-based virtual machines. *SIGPLAN Not.*, *51*(2), 104–113. Retrieved from https://doi.org/10.1145/

2936313.2816716 doi: 10.1145/2936313.2816716
Python. (2016). *Python performance benchmark suite.* https://github.com/python/pyperformance. (Accessed: 2019-05-10)
Shopify. (2023). *Set of benchmarks for the yjit cruby jit compiler and other ruby implementations.* https://github.com/Shopify/yjit-bench. (Accessed: 2023-03-14)
Tasos, A., Franco, J., Drossopoulou, S., Wrigstad, T., & Eisenbach, S. (2020). Reshape Your Layouts, Not Your Programs: A Safe Language Extension for Better Cache Locality (SCICO Journal-first). In *34th european conference on object-oriented programming (ecoop 2020)* (Vol. 166, pp. 31:1–31:3). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik. Retrieved from https://drops.dagstuhl.de/opus/volltexte/2020/13188 doi: 10.4230/LIPIcs.ECOOP.2020.31
Ureche, V., Biboudis, A., Smaragdakis, Y., & Odersky, M. (2015). Automating ad hoc data representation transformations. In *Proceedings of the 2015 acm sigplan international conference on object-oriented programming, systems, languages, and applications* (pp. 801–820).

About the authors

Joannah Nanjekye is a PhD student in an IBM CAS lab at the University of New Brunswick. Her research spans programming language design, optimization, and garbage collection. She is the author of the book Python 2 and 3 Compatibility, published by Apress, but has also authored several peer-reviewed papers on programming language implementation and optimization. She is a former director of the Python Software Foundation and a Python core developer whose interests are on the C API and subinterpreters.You can contact the editor at jnanjeky@unb.ca or visit http://cs.unb.ca/~jnanjeky/.

David Bremner holds three degrees in Computer Science, a B.Sc. Hons. from the University of Calgary (1990), an M.Sc. from Simon Fraser University (1993), and a Ph.D. from McGill University (1997). David spent two years as an NSERC postdoctoral fellow in the Department of Mathematics at the University of Washington from 1997 to 1999. Since 2000 David has been a faculty member at the University of New Brunswick and is currently a Professor of Computer Science with a Cross Appointment to the Department of Mathematics and Statistics. You can contact the editor at bremner@unb.ca or visit https://www.cs.unb.ca/~bremner.

Aleksandar Micic is a senior engineer on the runtimes team at IBM Canada in Ottawa. He has led several research projects at the IBM Center for Advanced Studies for about a decade. You can contact the editor at aleksandar_micic@ca.ibm.com.

A. Calculation of confidence intervals for normalized values

The well known *geometric mean* is defined for a series X of positive numbers as

$$\mathrm{GM}(X) := \sqrt[n]{\Pi_i X_i}$$

This has the nice property that for two equal length series X and Y of positive numbers,

$$\mathrm{GM}(X_i / Y_i) = \mathrm{GM}(X) / \mathrm{GM}(Y)$$

In particular this statistic is invariant under reordering X and Y. Unfortunately, the same does not hold for the standard deviation (nor for the geometric version defined by Kirkwood (Kirkwood 1979)). In less abstract terms, when calculating speedups, it matters which new run is compared with which baseline run, and in our experimental setup there is no obvious bijection. In order to account for different possible bijections, we calculate error bars via *bootstrapping* (Chernick 2011). More specifically, we use the `scipy.stats.bootstrap` function with parameter `confidence_level` equal to 0.95; for each of the default 9999 resamples (with replacement) (X', Y'), the method calculates $\mathrm{GM}(X') / \mathrm{GM}(Y')$, and reports a 95% confidence interval for the resulting values.