Toward Presizing and Pretransitioning Strategies for GraalPython

Johannes Henning
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
johannes.henning@hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany
tim.felgentreff@oracle.com

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

Presizing and pretransitioning are run-time optimizations that reduce reallocations of lists. These two optimizations have previously been implemented (together with pretenuring) using Mementos in the V8 Javascript engine. The design of Mementos, however, relies on the support of the garbage collector (GC) of the V8 runtime system.

In contrast to V8, dynamic language runtimes written for the GraalVM do not have access to the GC. Thus, the prior work cannot be applied directly. Instead, an alternative implementation approach without reliance on the GC is needed and poses different challenges.

In this paper we explore and analyze an approach for implementing these two optimizations in the context of GraalVM, using the Python implementation for GraalVM as an example. We substantiate these thoughts with rough performance numbers taken from our prototype on which we tested different presizing strategies.

CCS CONCEPTS

• **Software and its engineering → Just-in-time compilers**; **Object oriented languages**; **Virtual machines**.

KEYWORDS

storage strategies, run-time optimizations, Python, GraalVM

ACM Reference Format:
Johannes Henning, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. 2020. Toward Presizing and Pretransitioning Strategies for GraalPython. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3397537.3397564

1 INTRODUCTION

From the programmer's perspective, one of the biggest differences between managed and unmanaged languages is the level of control over when and how memory for an object is allocated and structured. In contrast to unmanaged languages, many dynamic programming languages offer only high-level, resizable, and untyped list-like data structures.

The optimization of such data structures can make up a significant portion of the work of a just-in-time (JIT) compiler and the performance benefits provided by it [5]. One well-known optimization applied to lists in multiple virtual machines (VMs) for dynamic languages are storage strategies [2, 6, 9], which "dynamically optimise collections whose elements are instances of the same primitive type"[2] by avoiding boxing these elements.

In this paper we focus on two additional optimizations that aim to improve the efficiency of these data structures further: *presizing* and *pretransitioning*. As our current GraalPython prototype only works for lists, we mainly discuss lists and not arrays, vectors, or other similar structures. But we believe our findings to be applicable for other collection types also.

Presizing. As the size of lists that the VM manages for the programmer changes over time, part of the execution time is spent growing or shrinking these structures according to their usage.

To avoid some of this overhead, some list implementations allow programmers to control their initial size. Appropriate size values, however, are often unknown at development time and are usually only used when resizing is a significant performance overhead. Therefore, automatic resizing is a trade-off between ease of programming and run-time performance.

Presizing aims to lessen this negative impact on the performance by propagating the observed list-sizes back to their original positions in the program (allocation sites), which we will expand on in section 3.

Pretransitioning. The size of a list is not the only property that can change over time, the types of objects stored inside it can also vary. Similarly to automatic memory management, the incurred lookup overhead is a performance disadvantage which, in turn, improves the programming experience by allowing collections to contain dynamically typed objects. To reduce the footprint and improve operations on lists of primitive values, many dynamic, object-oriented runtimes use the aforementioned storage strategies that store values without boxing, thereby increasing run-time performance by reducing the number of indirections incurred when performing computations on them.

The downside of storage strategies is, that they only work as long as all objects in a list are supported by the active strategy. If a floating-point number is added to a list of integers, the storage strategy has to be changed and the entire list has to be converted to a different format. Such a conversion is called *transition* and can be a costly operation on large lists. Pretransitioning aims to avoid these transitions, as laid out in section 4.

2 ALLOCATION-SITE-BASED OPTIMIZATIONS

The closest related work we are aware of was done by Clifford et al. [3] on the V8 VM. They extended memory management with *Mementos*, short-lived objects used to associate lists with their allocation sites, i.e. the original position in the program where they were declared. The information gathered this way would then be used to optimize future allocations of lists from the same allocation site. Their main focus was on another optimization: *pretenuring*. This optimization reduces garbage collector (GC) times by using the Memento information to speculate on objects that were likely to survive the next GC cycles. Such objects are then directly allocated in the older generation.

Pretenuring is a GC-specific optimization and GraalVM languages [10] do not have this level of access to the GC. Therefore, we do not consider pretenuring in this work.

The other two optimizations discussed by Clifford et al. are the ones we target in this paper. While pretransitioning was discussed and evaluated thoroughly, presizing was only discussed briefly and the evaluation is only anecdotal. The paper mentions that their prototype "was able to learn the optimum allocation size for the important sites in the very sensitive DeltaBlue benchmark without prior knowledge, and preliminary measurements indicated a slight decrease in the maximum overall heap size in Octane, with no other performance loss"[3]. We also found no further indication that presizing was used in V8.

3 PRESIZING

Presizing relies on the assumption that lists from the same allocation site will have similar resource consumption over time. Hence, when a new list is created, the runtime environment does not allocate a predetermined static number, but instead uses the previously observed sizes of the list to compute the optimal size to allocate.

While this optimization should have little to no effect on lists that are used infrequently and stay small, it can save a significant number of resizing operations for lists that grow multiple times and are used in a *hot* portion of the code. For example, a list allocated with size 4 that grows by a factor of 2 whenever it hits its limit will grow 12 times if 10.000 elements are added to it. With presizing, the number of operations needed to reach the optimal size can be reduced significantly or even eliminated entirely.

To achieve this we need to answer how, given the history of all sizes a specific list had during the execution of a program, do we determine the optimal *presize*? There is no straightforward answer that works in every scenario.

Presizing works well for lists at allocation sites that have stable list-sizes over time because the presize can approximate the maximum size and eventually eliminate all resizing operations.

The challenge in determining a reasonable presize value comes with lists that do not reach a stable size. Consider Listing 1 as an example. Whether the appropriate presize is closer to 1 or 10000 depends on the likelihood of `flag` being true. If we choose to not modify the presize, for each time `flag` is true `a` could grow over 10 times using the example from above. Whereas if we increase the presize, for each time `flag` is false we would unnecessarily allocate $(presize - 1) * size(1)$ bytes. While finding an optimal solution remains an open question, we discuss what we have found so far in section 6.

But, even if a theoretically optimal solution could be found, the applicability also depend on the profiling information available. We believe that the variables relevant for profiling are:

- The number of times the list grows
- The number of elements the list contains at most, at least, and on average
- The size of the list when it is garbage collected, including how much of it was empty

Whereas the most important ways the performance could be influenced to approach an optimal solution are:

- Presize, i.e. initially allocated size
- Growth factor, i.e. by how much does the list grow when its allocated size is exhausted

```
var a = []
if flag:
    for i in range(0, 10000):
        a.append(1)
else:
    a.append(1)
```

Listing 1: Simple example of branches impacting list-size for the same allocation site depending on external factors

4 PRETRANSITIONING

Pretransitioning relies on the same speculation as presizing: lists from the same allocation site will likely contain the same types of objects over time. For example, Listing 2 shows the unoptimized storage strategy transitions an list might go through in GraalPython. In this example, as the list will always end up in the generic object strategy, *pretransitioning*, i.e. initializing it with the object strategy instead of empty, would avoid 3 transitions, with the last 2 in particular being expensive, as large lists need to be copied in both cases. While this example is straightforward, the question remains of how to handle cases where a list only transitions *sometimes*, but stays homogeneous in other cases. Which variables should be weighted to make this decision?

While pretransitioning is straightforward when no branches are involved, a solution that does not consider them will have an

```
var a = []       # Initialized with Empty strategy
for i in range(0, 10000):
    a.append(1)  # Transition to 32-bit Integer strategy
                 # on first execution.
a.append(1.0)    # Transition to Double&Integer strategy
a.append('1')    # Transition to Object/Generic strategy
```

Listing 2: Storage strategy transitions in GraalPython

```
var a = []
if flag:         # While flag is true
    a.append(1)  # a will use an Integer strategy
else:            # If flag is false once a will use
                 # the Object strategy from then on
    a.append(Object())
if flag:
    sum(a)       # Will run fine on the Integer strategy
                 # but will be slow on the Object strategy
```

Listing 3: Example of pretransitioning adversely affecting performance due to branching

adverse effect on performance, as illustrated in Listing 3. One way to avoid this scenario could be to consider branching information collected during interpretation and pretransitioning to the more likely branch. This is speculative however, as this has so far proved to be too complex to implement in our prototype.

Another answer to this problem might be to consider this example as unlikely and thus accept the decrease in performance.

5 PROTOTYPE FOR GRAALPYTHON

Building on the idea of Mementos, we implemented allocation-site tracking in GraalPython[1] allowing us to link the information of a list resizing operation back to its allocation site. Based on this information, we conducted preliminary experiments with different strategies for determining the presize. Our code and benchmarks are available on Github[2].

In this section, we will discuss the difference in implementation strategy between our approach and the one of Clifford et al. as well as the limitations of our proposal.

5.1 Tracking Allocation-sites

In contrast to the V8 VM, we are working within the GraalVM ecosystem, which relies on a Java VM for many of its features, in particular for memory management and garbage collection. Subsequently, where Clifford et al. implemented their allocation-site tracking by storing information in Mementos that were managed in part by the GC, we do not have this level of access to the GC and cannot easily implement this strategy.

Instead, we decided to instrument the list creation and append operations of Python instead. We can thereby observe the size of a list on every append that is called on it and use this information to compute the presize for all other instances sharing its allocation site.

While Clifford et al. paid particular attention to the performance implications of the book-keeping added by Mementos through their allocation strategy and GC cycles, our approach eliminates its book-keeping cost by only doing it during interpretation. We do no book-keeping in compiled code and assume that list-sizes are stable by the time the program is compiled, which generally happens after a code segment has be executed 1000 times.

[1]https://github.com/graalvm/graalpython
[2]https://github.com/hpi-swa-lab/graalpython/tree/morevms/2020-presizing

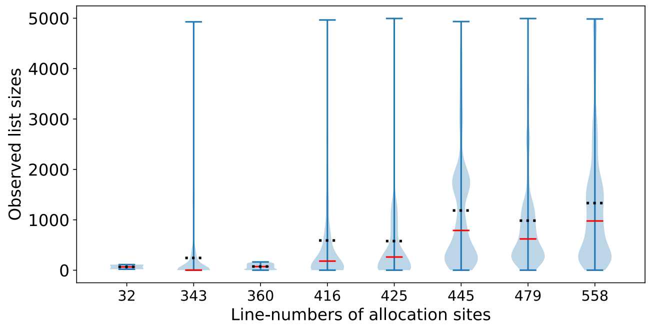

Figure 1: Observed list sizes at DeltaBlue allocation sites, excluding list with fewer than 5 elements. Median in red and mean as black dots.

5.2 Limitations of this Approach

As we only track the allocation site and not the allocation call chain, our approach does not work for abstractions built on top of resizable lists that provide a different interface. For example, Smalltalk's `OrderedCollection` type is written in Smalltalk itself and holds primitive arrays in a field, so the same allocation site would be seen for all arrays used in OrderedCollections, regardless of where the OrderedCollection was instantiated. This means that high-level collection libraries cannot benefit from our approach. This limits the applicability of our solution to other languages, as we realized when we tried to implement presizing for GraalSqueak [8].

Secondly, our current approach cannot observe branches in the control flow of the program and therefore cannot avoid performance degradation like the one exemplified in Listing 1. This, however, is also a limitation of the prior work that we also have yet to solve.

6 PERFORMANCE ANALYSIS

The first indications on the performance impact of presizing and pretransitioning look promising. In order to not rely on microbenchmarks, we chose the DeltaBlue benchmark, which tests the performance impact for small and large lists simultaneously.

6.1 DeltaBlue

DeltaBlue is an incremental constraint hierarchy solver [4] originally developed for Smalltalk, which has been implemented in several languages and thus can be useful for cross-languages benchmarking, as suggested by Marr et al. [7].

6.2 Allocation Sites in DeltaBlue

As the solver relies on lists for storing and evaluating its constraints, there are several performance relevant allocation sites in the source code. Allocation sites are identified by their line number in the benchmark source code[3]. Figure 1 illustrates the list

[3]Available here: https://git.io/Jve2G

sizes observed through the instrumentation of our prototype when running DeltaBlue 30 times with an input size of 5000.

Figure 1 provides several insights and illustrates, why there isn't an obvious answer to what the optimal size for presizing might be. There are several observations we would like to make:

- Presizing works fine for lists that have stable sizes, like lines 32 and 360.
- All other allocations sites have a bottom-heavy distribution, meaning that there are many small lists and some very large ones.
- In particular lines 445 and 558 have local maxima, where there are many large lists that are still smaller than the maximum size.
- All lists other than lines 32 and 360 approximately reach the maximum size given as the benchmark input (5000).

As expected, the optimization is straightforward and useful for lists that reach a stable size. But lists that vary widely present an interesting trade-off: choosing a small presize limits the usefulness of the optimizations, but choosing too large a presize can explode the memory usage of the program. In fact, at the time of writing, our version aborts with an out-of-memory error, when we run the benchmark with input sizes larger than 8000. We will explain which strategies we tested so far for weighing this trade-off below.

6.3 Optimization Potential

It is important to point out that transitions as well as resizing involve copying the list, which entails performance overhead proportional to the list size. Hence, the performance gains of presizing and pretransitioning are proportional to the size of the list and the number of times it would have been resized or transitioned.

In addition, avoiding resizing and transitioning can help other optimizations to be more effective. For example, allocation removal [1] aims to avoid allocations of objects that are only short-lived and do not escape outside the scope of the compilation unit. The Graal compiler is able to perform allocation removal on small lists, but this is limited by the kind and number of operations performed on them. Any list larger than the default initial size will require resizing and potentially transition operations. These operations make it less likely that the list will be allocation-removed, because they will create branches in the resulting compiler graph. With presizing and pretransitioning, however, we are able to avoid these operations, which removes these branches, which in turn increases the probability that some lists can be allocation-removed.

6.4 Tested Presizing Strategies

Naive strategy. The easiest solution for testing the achievable performance without regard for the increase in memory consumption is to take each newly observed maximum size as the new presize. Our naive strategy works similarly, except that we inhibit the growth of presize: we only increase the presize if the current size has been exceeded on 8 occasions. The resulting presizes per allocation site are equivalent to the maximum values in Figure 1.

Promisingly, we observed a significant impact on performance in the DeltaBlue benchmark with the size configuration at 5000. The average run-time speed-up was by a factor for 1.37, the maximum speedup observed (shortest run-time with optimizations versus shortest run-time without) was a factor of 3.7. This was at the cost of memory consumption (increased by a factor of 2.5) and slower warm-up by a factor of 0.8.

Naive strategy with shrinking. Without shrinking, our naive strategy approximated the maximum list-size over time and never went back. Our next step was to shrink the estimate when the list was observed to be smaller, again inhibited by a factor of 8. This did not produce satisfying results, most likely because we have no way of observing the final size of the list, but as discussed are notified of every append operation performed on it. Smaller list-sizes are thus over-represented in the history of all sizes.

We experimented with different growth and shrinking factors, but did not find any way to balance the presize with this approach.

Average size strategy. When only saving the presize for each allocation site and reducing or increasing it did not produce balanced results, we decided to record all sizes encountered at each allocation site to determine the mean size. While Figure 1 was created from this data, we have yet to measure the performance characteristics of this strategy. To our best knowledge though, it seems like that the mean would still be too large, but the median size looks promising and will be the subject of our next experiments.

Summary of our current insights. As observed with the naive strategy, presizing can have a positive effect on performance, but needs to be considered carefully to find the sweet-spot, where we maximize the performance benefits while minimizing the impact on memory consumption.

Further research is needed for any definite answer in this regard.

7 CONCLUSION & FUTURE WORK

We presented preliminary evidence that pretransitioning and presizing can be implemented in GraalVM languages using our approach and provide performance benefits for certain applications. Our naive solution shows a speedup of 37% in the DeltaBlue benchmark, at the cost of a 150% increase in memory consumption and 20% longer warm-up times. Micro benchmarks for the presented code snippets also showed smaller compiler graphs with fewer allocations.

While these initial findings are promising, more work is needed in order to find an optimal strategy for increasing performance while decreasing the additional memory consumption. In particular, we want to leverage more of the variables discussed in subsection 5.1 and different heuristics for determining the presize. Increasing the mentioned growth factor could be a good way to avoid exploding memory consumption while still reducing the number of resizing operations needed for lists that infrequently get very large.

Finally, the impact of our optimizations needs to be evaluated with additional benchmarks.

ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of HPI's Research School[4] and the Hasso Plattner Design Thinking Research Program[5].

[4]https://hpi.de/en/research/research-school.html
[5]https://hpi.de/en/dtrp/

REFERENCES

[1] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Allocation Removal by Partial Evaluation in a Tracing JIT. In *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 43–52.

[2] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 167–182.

[3] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *ACM SIGPLAN Notices*, Vol. 50. ACM, 105–117.

[4] Bjorn N Freeman-Benson and John Maloney. 1989. The DeltaBlue Algorithm: An Incremental Constraint Hierarchy Solver. In *Eighth Annual International Phoenix Conference on Computers and Communications. 1989 Conference Proceedings*. IEEE, 538–542.

[5] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *ACM Sigplan Notices*, Vol. 44. ACM, 465–478.

[6] Matthias Grimmer, Stefan Marr, Mario Kahlhofer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2017. Applying Optimizations for Dynamically-typed Languages to Java. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*. ACM, 12–22.

[7] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *ACM SIGPLAN Notices*, Vol. 52. ACM, 120–131.

[8] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 14–26. https://doi.org/10.1145/3357390.3361024

[9] Tobias Pape, Tim Felgentreff, Robert Hirschfeld, Anton Gulenko, and Carl Friedrich Bolz. 2015. Language-independent Storage Strategies for Tracing-JIT-based Virtual Machines. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 104–113.

[10] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. One VM to Rule Them All. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM, 187–204.