Adaptive Just-in-time Value Class Optimization

Transparent Data Structure Inlining for Fast Execution

Tobias Pape
Hasso Plattner Institute
Universtiy of Potsdam
tobias.pape@hpi.de

Carl Friedrich Bolz
Software Development Team
King's College London
cfbolz@gmx.de

Robert Hirschfeld
Hasso Plattner Institute
Universtiy of Potsdam
hirschfeld@hpi.de

ABSTRACT

The performance of value classes is highly dependent on how they are represented in the virtual machine. Value class instances are immutable, have no identity, and can only refer to other value classes or primitives and since they should be very lightweight and fast, it is important to optimize them well. In this paper we present a technique to detect and compress commonly occurring patterns of value class usage to improve memory usage and performance. micro-benchmarks show two to ten-fold speedup of a small prototypical implementation over the implementation of value classes in other object-oriented language implementations.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Data types and structures, Dynamic storage management; D.3.4 [**Processors**]: Code generation, Compilers

Keywords

Meta-tracing, JIT, Data Structure Optimization, Value Classes

1. INTRODUCTION

Objects are at the heart of object-oriented languages and the choice of how to represent them in memory is crucial for the performance of a language implementation [3, 27]. The standard way of representing objects involves an indirection for references to other objects, e.g. by using direct pointers or object tables. Typical best practices of object-oriented modeling and design—such as delegation or the composite design pattern—have an influence on performance with such representations. Every additional indirection between delegators and delegates or composites and their parts has the overhead of a new object. This includes memory consumption, but also execution time to navigate the referenced objects.

In this paper we propose an object layout that stores nested object groups in a compacted, linearized fashion. To simplify the problem, we restrict ourselves to optimizing *value classes* [2]. Value classes are immutable classes without identity that only store other value classes or primitive data. Value classes are available in Java [31], Scala [28], and .Net [25], to name a few.

Composite structures involving value classes have certain usage patterns. This is obvious in linked lists (a single list element probably references another list element and so forth) or trees (a tree node references a number of other nodes or a value). Data structures with such patterns can be transformed into lower-level structures more suitable to the machine model. While simple variants of such patterns can be statically inferred, many become apparent only at run-time. In particular, recursive structures often exhibit patterns that are opaque to static inference, e.g. while a tree apparently may have sub-nodes, it is statically unknown whether trees are used as deep trees or rather flat ones, or which kind of values are stored in the tree in practice. However, each case could be optimized differently. Hence our optimization approach works at run-time in conjunction with a just-in-time (JIT) compiler.

In this paper, we describe the following contributions:

- We propose an approach for finding patterns in value class usage at run-time.
- We present a compressed layout for value classes that makes use of the patterns to store value classes more efficiently.
- We report on the performance of micro-benchmarks for a small prototype language.

The paper is structured as follows: section 2 gives brief introductions to tracing JIT compilers [6]. In section 3, we present our approach to just-in-time optimization of data structures. A proof-of-concept implementation is briefly presented in section 4 and its performance is evaluated in section 5. Our approach is put into context in section 6 and we conclude in section 7.

2. TRACING JUST-IN-TIME COMPILERS

Just-in-time (JIT) compilation has become a mainstream technique for, among other reasons, speeding up the execution of programs at run-time. After its first application to Lisp in the 1960s, many other language implementations have benefitted from JIT compilers—from APL, Fortran, or Smalltalk and Self [1] to today's popular languages such as Java [29] or JavaScript [24].

One approach to writing JIT compilers is using *tracing*. A tracing JIT compiler records the steps an interpreter takes to obtain an instruction sequence called *trace* and then uses this trace instead of the interpreter to execute the same part of that program [26] at higher speed. Tracing produces specialized instruction sequences, e.g. for one path in if–then–else constructs; missed branches still use the interpreter. Tracing JIT compilers have been successfully used for optimizing native code [4] and also for efficiently executing object-oriented programs [19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SAC'15 April 13–17, 2015, Salamanca, Spain.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3196-8/15/04...$15.00.
https://dx.doi.org/10.1145/2695664.2695837

Meta-tracing takes this approach one step further by observing the execution of the interpreter instead of the execution of the application program. Hence, a resulting trace is not specific to a particular application but the underlying interpreter [10, 13]. Therefore, it is not necessary for a language implementer to program a optimized language-specific JIT compiler but rather to provide a straightforward language-specific interpreter in RPython, a subset of Python that allows type inference. *Hints* to the meta-tracing JIT enable fine-tuning of the resulting JIT compiler [9]. RPython's tracing JIT also contains a very powerful escape analysis [7], which is an important building block for the optimization described in this paper.

3. OPTIMIZATION APPROACH

Our optimization detects common patterns of how instances of value classes (*value objects* for short) reference each other. It then introduces short forms for these patterns, which we call *shapes*, that make it possible to represent these patterns more efficiently in memory. This physical representation of the data structures is separated from the interface visible to programmers.

A straightforward value object representation would be a chunk of memory that stores a pointer to the class of the value object, followed by its contents that typically consist of pointers to all the fields of that value object. We call the contents the *storage* of the value object. The class describes the storage, e.g. how many fields there are and how they are to be interpreted. This representation corresponds very closely to the programmers' view on value classes.

In our approach, the storage area remains, but the pointer to the class is replaced by a pointer to the shape. As with the regular representation, the shape determines the class of the value object and hence the meaning of its contents. Two examples for this separation can be found in Figure 1. For every value class there is a *default shape* that has no additional information compared with directly storing the class. If the default shape was always used, the representation would be completely equivalent to the straightforward one.

The difference from the straightforward representation is that a shape does not necessarily describe only the class of the value object. Rather, a shape can additionally describe the shape of referenced value objects, recursively. If a referenced value object's shape is not specified in the referencing object's shape, it is stored as a reference in the storage. If the shape is specified, that value object's content is inlined into the referencing value object's storage. That way, referenced values are stored *compactly*, i.e. in spatial proximity without overhead such as headers or a pointer between them, and hence with less memory consuption compared to storage without compaction. This process can be applied recursively.

To actually save memory, a shape has to be shared by significant number of value objects. Indeed, if every shape were used by only one object, the memory use would not be improved. Therefore, a new shape should only be introduced after run-time profiling ensures that it occurs often enough.

To understand the rest of the system, we now need to look at (a) how structure patterns are recognized, (b) howthe construction of values ensures the proper usage of shapes, and (c) how the field reading of inlined fields is implemented.

3.1 Shapes and their recognition

A *shape* (▭ in all figures) describes the abstract, structural representation of composite value objects and is shared between all identically structured value objects of the same value class, denoted by its name[1].

[1]We refer a value class by its name and the arity of its type in a Prolog style, e.g. Node/2 for binary node objects.

Shapes can be recursive; they consist of sub-shapes for each field in a value object's storage. A special, non-recursive type of shapes denotes unaltered access to object content (*direct access*, ▼ in all figures) and termination of shape recursion. Value objects with these shapes are treated as black boxes, e.g. scalar data or unoptimized objects that are stored directly. This is depicted in the bottom part of Figure 1; all three nodes in the list share the same shape, which denotes that each node consists of two references with *direct access* shapes. The same holds for the nodes of the tree in that figure, but with three references.

Storing the shape of value objects may seem redundant given that the shape matches what it tries to describe. This only holds as long as no optimization has taken place. In this case, a value object's shape is the *default shape* of its value class and solely consist of *direct access* sub-shapes. The shapes in Figure 1 are the default shapes for their value classes.

Further, a mapping of replacement options for inlining (the *transformation rules*), and profiling data built up during object creation to aid the creation of new transformation rules (the *history*) are supplementary structures that we use to aid the inlining process.

3.1.1 History

The immutability of value objects demands that all to-be-referenced value objects that will constitute the content of a new value object have already been constructed beforehand. Hence, their shapes will be available at construction time and we can count occurrences of *sub-shapes* at specific positions in the value object. That way, we obtain a histogram of all possible shapes a referenced value object can have. In Figure 2, e.g. for shape s_1 at position 1, the shape s_1 itself has been encountered 17 times as sub-shape, while shape s_2 has been encountered 3 times as sub-shape in that position.

When a certain threshold of encounters has been reached, we generate a new transformation rule.

3.1.2 Transformation rules and recognition

The transformation rules are mappings $Shape \times Position \times Shape \rightarrow Shape$ that drive the inlining process. When constructing a new value object, they are consulted by the inlining algorithm. These mappings can be specified prior to program execution or inferred dynamically based on shape history.

Upon value object creation, just after updating the shape history, we check whether the sub-shape counters hit a certain threshold, and if so, proceed to create a new shape that combines the value object's current shape with the sub-shape that hit the threshold. In this new shape, we replace the *direct access* sub-shape at the position of the threshold hit with the sub-shape found in the history entry. The position of the hit, the sub-shape at that position, and the newly created shape are then recorded as new rule in the transformations table. Considering Figure 2 as example, shape s_2 would be the result of turning the history entry (s_1, 1, s_1, 17) into the transformation rule $(s_1, 1, s_1) \mapsto s_2$. The structure of shape s_2 is the structure of shape s_1 but with another s_1 structure in the final position.

We call the process of recording the shape history and inferring transformation rules *shape recognition*.

3.2 Compaction though inlining

Since value objects are immutable, compaction is required only when creating new ones. With this premise, our optimization technique works by inlining the to-be-referenced value objects into the to-be-created value object upon its creation.

3.2.1 Inlining

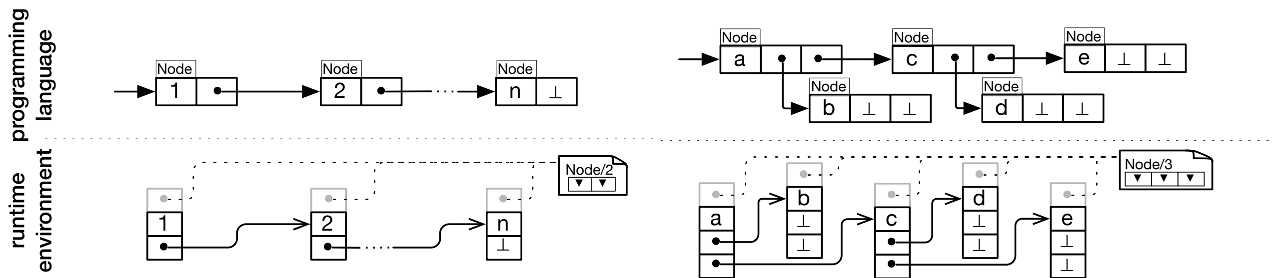

Figure 1: Value class representation for a linked list and a tree. Top: the language view; bottom: runtime environment view with *storage* and *shape*

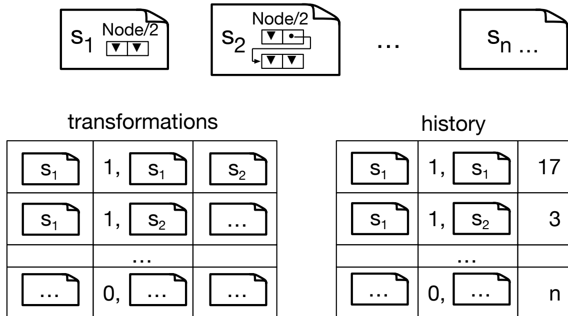

Figure 2: Shapes and supplementary data structures: transformations describe replacement rules for inlining; history provides a histogram of all encountered sub-shapes at a certain position

When a new value object is created, we handle the default shape s for the type and the value object's new content c as specified in the algorithm in Figure 4. We iterate over the given new content and for each to-be-referenced value object o_i at position i and its sub-shape s_o, we look up a replacement shape in the transformations table. If the table contains a mapping, the replacement shape s' is assumed as the shape of the to-be-created value object. At that point, the *storage* of the current value object c_i is *spliced* into the current content c instead of the current value object c_i itself; the value object c_i is now *inlined*. After a successful inlining, the new shape s' becomes the to-be-created value object's shape s and the current content is reiterated from start to allow for possible other transformations due to the shape change. That way, transition chains are possible that may quickly lead to shapes of deeply nested structures. Once no further transitions are found, the value object's shape s and the current content c are returned as the shape and storage of the new value object.

The effect of this process is shown simplified with the example in Figure 3: creating a new node consisting of "1" and a rest list as in the figure. We start with a list of "1" and the rest list as initial content for the new value object and shape s_1 as the initial default shape. We iterate over the list and encounter "1" at position 0. For this example, we assume that the transformation table does not contain a mapping for "1" at position 0, thus s' will be s and we continue with the next position. At position 1, we find the rest list with the sub-shape s_1. In the transformation table, the entry for $(\mathsf{s}_1, 1, \mathsf{s}_1)$ holds a replacement shape, s_2. Thus, we inline the current value object's storage into the current content as c', which now has three elements. Note that it is not the shape of the rest list s_c that is changed but rather the shape s of the to-be-created value object. The content, now c, is reiterated but no further transformations are found. The resulting value object is that to the right in Figure 3.

The shape of thusly optimized value objects are themselves subject to the shape-recognition process and eventually, transition rules to more optimized shapes can be created in the default shapes for the value classes. Thus, more specific shapes are directly available for the inlining process. Value objects can be more directly transitioned into the most optimized shape compared to working off a long transition chain.

This inlining technique has two main advantages. First and foremost, inlined value objects take up less space than individual, inter-referenced value objects. But even more, the shape of a value object provides structural information in a manner the meta-tracing JIT compiler can speculate on. This is crucial for optimizing the access to references of a value object.

3.3 Transparent field access

While optimization of data structures takes place during construction, we have to apply the reverse during deconstruction, i.e. when accessing a value object referenced by another. This is no longer trivial, as several (formerly referenced) value objects may have been inlined into their referencing value objects. Therefore, we construct new value objects whenever a reference is navigated, essentially reifying it. We use the information a value object's shape provides to identify which parts of the value object's storage comprise the value object to be reified. The structural information allows a direct mapping from the language view of the data structure to the actually stored elements. In Figure 5, the structural information in the shape of the leftmost list allow the reasoning that the first element of the storage is equivalent to the head of the language level node value object and the remaining three storage elements are equivalent to the tail of that value object, as recored in the shape. Hence the middle view in that figure; both the element "1" and the rest list have been reified. The same goes for the rightmost view.

Note that this reification is completely transparent to the programmers. Taking, e.g. the tail of a node value object or accessing the third element of a ternary tree repeatedly, the operations remain the same on the language level, no matter what is the inlining status of the value objects on the implementation level.

3.4 Benefits

With the inlining approach, fewer value objects need to be created for long living data structures, since the references to the now-inlined value objects are elided. Combining this with the reification and the shape recognition, more memory is saved the longer a program runs; the shapes will be tailored to fit the specific application running. That said, there may be cases where no memory can be saved, especially in programs that only work on primitive data, non-composite data structures, or with a high amount of sharing between data structures.

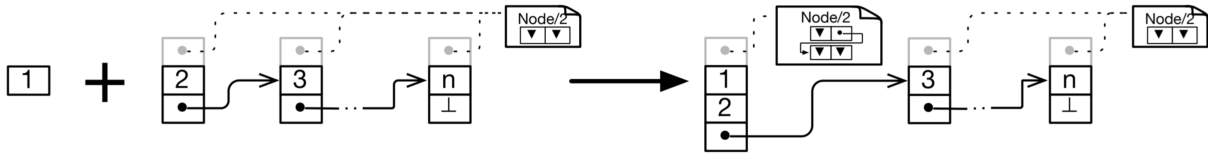

Figure 3: When creating a new node value object that should contain "1" and the list as shown, a new value object that merges the "1" with the "2" object and a different shape is created instead.

Input: $s : Shape, c : [Object]$
$i \leftarrow 0$
while $i < |c|$ **do**
  $o \leftarrow c_i$
  $s_o \leftarrow o_{\{shape\}}$
  $s' \leftarrow transformations_{s,i,s_o}$ or s
  if $s' \neq s$ **then**
    $c' \leftarrow [c_{0,\dots,i-1}, o_{\{storage\}}, c_{i+1,\dots,|c|}]$
    $s \leftarrow s'$
    // rewind over new storage:
    $i \leftarrow 0, c \leftarrow c'$
  else
    $i \leftarrow i + 1$
  end
end
return s, o

Figure 4: Merging composite objects based on shape

4. IMPLEMENTATION IN RPYTHON WITH A TRACING JIT COMPILER

We implemented our optimization approach in a simple execution model[2]. It provides a λ-calculus with pattern matching as the sole control structure and is implemented as a direct application of the CEK-machine [17]. The only structured data types currently available are value classes. We used the RPython tool chain to incorporate its meta-tracing JIT compiler [6]. The implementation has been carefullly unit-tested during development to make sure that various complex substitutions and compactions work correctly.

The presence of the JIT compiler is necessary to begin with, because the approach just presented, i.e. shape recognition, inlining, and reified reference access combined, does not yield a performance increase on its own. In fact, implementing the approach naïvely yields significantly worse performance than leaving it out altogether, due to the constant check of the transformation rules every time a new value object is created. Additionally, reading inlined fields of compacted value objects results in the allocation of intermediate data structures. This is of course not the case in the naïve representation.

To improve performance, the JIT compiler needs to reduce the overhead of these operations. The first step is to treat the transformation tables as constant when a function is compiled. This allows the JIT compiler to compile value object creation down to a series of type checks for the types of the referenced value objects. While the transformation tables are not constant *per se*, we instruct the JIT compiler to treat them as such for all practical purposes.

Second, we have to avoid the otherwise necessary reification of referenced value objects when it is being read from a value object it has been inlined into. For that, the observation that most of these intermediate value objects are actually short-lived is crucial; most value object are created just to be either immediately discarded or consumed in another, typically larger data structure. As a concrete example, typical linked list operations deconstruct the list they are working on. Hence, if the tail is read off a linked list node which has the tail inlined (as the transition from left to middle in Figure 5) and needs to be reified, that tail is usually soon deconstructed itself into its head and tail components (as the transition from middle to right in the same figure). This allows the tracing JIT compiler to optimize the reading of fields that need reification. Since the value objects allocated when reifying a field are short-lived, the built-in escape analysis [8] will fully remove their allocation and thus remove the overhead of reification.

[2]availiable at `https://bitbucket.org/krono/lamb`

5. RESULTS

We report the performance of five micro-benchmarks, i.e. their execution time and their maximal memory consumption (resident set size). The benchmarks chosen are *append*, *filter*, *map*, and *reverse* on very long linked lists and the creation and complete prefix traversal of a binary *tree*. Due to the limited feature scope of our prototype, more sophisticated applications are currently not available for benchmarking.

In the left part of Figure 6, the execution time of all benchmarks is reported. Our implementation, labeled *prototype*, is significantly faster—from two to ten times faster—for all but the tree benchmark, where our implementation is second to just the ahead-of-time (AOT) compiled OCaml version. However, the other two RPython-based implementations are likewise significantly slower than expected; the Pycket interpreter uses the same CEK execution model as our implementation. It is possible that not the value class implementation but the interpreter style is responsible for most of the execution time. Nevertheless, our implementation is still significantly faster than both RPython-based implementations. For memory consumption, shown in the right part of Figure 6, our implementation always uses significantly less memory than the other implementations.

One key reason for our prototype's performance is the interaction between escape analysis and the compacted storage. The benchmarks exhibit a certain usage pattern, in particular, the access to a list element is typically followed by inserting this element into a new list, with possibly processing it. The tracing JIT compiler and its escape analysis can infer that no reification of the actual value object is necessary and, furthermore, that a certain number of such operations occur consecutively. Hence, operations can happen *en bloc*, e.g. for a list inlined n levels deep, reverse can operate in n-chunks of items.

Our approach makes use of three parameters that may influence performance:

Maximum object size Only value objects up to this size are considered for inlinig. Setting this to to zero disables our optimization, setting it to a very high number might result in very large value object at runtime, which might be undesirable. We used a maximum size of 7 fields for our measurements.

Figure 5: Referenced value object reification. Accessing the second item 2 of the list $l \leftarrow \textsf{Node/2}[1, \textsf{Node/2}[2, \textsf{Node/2}[3, \ldots]]]$ by two operations *head*(*tail*(l)) results in two differently reified rest lists to be created. Note how the shapes of the rest lists differ.

Maximum shape depth The number recursive shape occurences per value object is bounded by this parameter. Setting this to a low value may not catch all optimizable object shapes, setting it to a very high number may lead to an excessive number of shapes at runtime should there be a lot of value objects with no fields at all. We used a maximum depth of 7 shapes for our measurements.

Substitution threshold The threshold for transformation rule creation (as in section 3.1.2), when set to a zeor or a very low value can lead to excessive transformation rule creation for value object combinations that are only rarely used. A very high number might inhibit the creation of such rules at all and practically disables our optimization. We used a threshold of 17 shape occurences for our measurements.

The results suggest that our approach is viable and warrants application to other programming languages.

Setup.

Hardware The processor used was an Intel Xeon E5410 (Harpertown) clocked at 2.33 GHz with 2×6 MB cache; 16 GB of RAM were available. All runs are un-parallelized, hence the number of cores (four) was irrelevant to the experiment. Although virtualized on Xen, the machine was dedicated to the benchmarks.

Software The machine ran Ubuntu 12.04.4 LTS with a 64 bit Linux 3.2.0. The benchmarking framework *ReBench*[3] was used to carry out all execution of the benchmarks and collection of measurements. RPython as of revision 5a423327c96a served for translation.

Compared Implementations We included OCaml △, Racket ◻, Pycket +, and PyPy ⊠ in the comparison. For all these, value classes or equivalent means supporting immutable data are available. OCaml provides a concept similar to value classes with its algebraic data types and its execution model bears similarities to our implementation. OCaml produces native binaries. Racket's *cons* cells, *structs* and classes can act as value classes. Racket acts as virtual machine with hand-written JIT compiler. Pycket [12] is a RPython implementation of Racket and provides a meta-tracing JIT compiler. PyPy is the RPython implementation of Python and has a meta-tracing JIT compiler. While Python has no actual concept of value classes, we used regular classes without mutating them. PyPy is then able to handle them as value classes. We intended to also include the standard Python (CPython) but it was too slow and would have rendered the comparison meaningless.

Methodology Every benchmark was run ten times uninterrupted at highest priority, in a new process. The execution time (*total time*) was measured *in-system* and, hence, does not include start-up; however, warm-up was not separated, so JIT compiler execution time is included in the numbers. The maximal memory consumption (*resident set size*) was measured *out-of-system* and may hence include set-up costs. We report the arithmetic mean of the ten runs. The error was negligible. We provide all numbers in the appendix. Our benchmarking code and infrastructure are publicly available.[4]

6. RELATED WORK

From a data structure optimization point of view, value classes are similar to *algebraic data types* as found in languages in the ML family. Hence, optimizations done to this category of data structures are relevant to value classes, too.

Wimmer has proposed object inlining [37] as a general data structure optimization for structured objects in Java. Superficially, this approach is similar to this work, yet object inlining is restricted to statically typed object oriented languages like Java, as the approach needs full knowledge of all class layouts until just before JIT compilation. Moreover, object inlining is bound to create one "optimized path" per class.

Language-level optimization.

The idea to improve data structures to gain execution speed was proposed especially to improve operations on linked lists in functional languages, e. g. by unrolling [32]. Typically, those optimizations are restricted to linked lists of cons-cells.

One of the key effects in our optimization is avoiding to allocate intermediate data structures. In that respect, *hash consing* [16, 18, 23], as used in functional languages for a long time, is related to this work. However, hash consing typically works at the language level using libraries, coding conventions, or source-to-source transformations. It is not adaptable at run-time.

Ahead-of-time optimization.

Deforestation [21, 34, 36] has the aim to eliminate intermediate data structures and is in this respect related to our approach. However, deforestation deliberately works through program transformation and does not incorporate dynamic usage information. It is typically only available to statically typed functional languages, such as ML.

Just-in-time compilers.

Compiling to native code at run-time, i. e. JIT compilation, is a prevalent and extensively studied technique, found in several different, but chiefly object-oriented, dynamically-typed languages [1]. Prominent examples include the Smalltalk-80 bytecode-to-native-code compiler by Deutsch and Schiffman [15], and the optimizing JIT compiler of Self, with type specialization and speculative inlining [14]. These concepts were later used in the HotSpot JIT compiler [29] for Java.

The prevalence of web browsers has made JIT compilation an important topic for JavaScript implementations, e. g. the int V8 JavaScript implementation [24].

Nota bene: The map transitions for hidden classes used in V8 [22] and inspired by Self [14], are in principle similar to our notion of transformation rules. As well as objects in V8 start with a default hidden class and follow map transitions to their most optimal hidden class, the transformation rules in our approach change the shape of a value object from its default shape to its most optimized one during the value object's creation.

[3] https://github.com/smarr/ReBench

[4] https://bitbucket.org/krono/lamb-bench

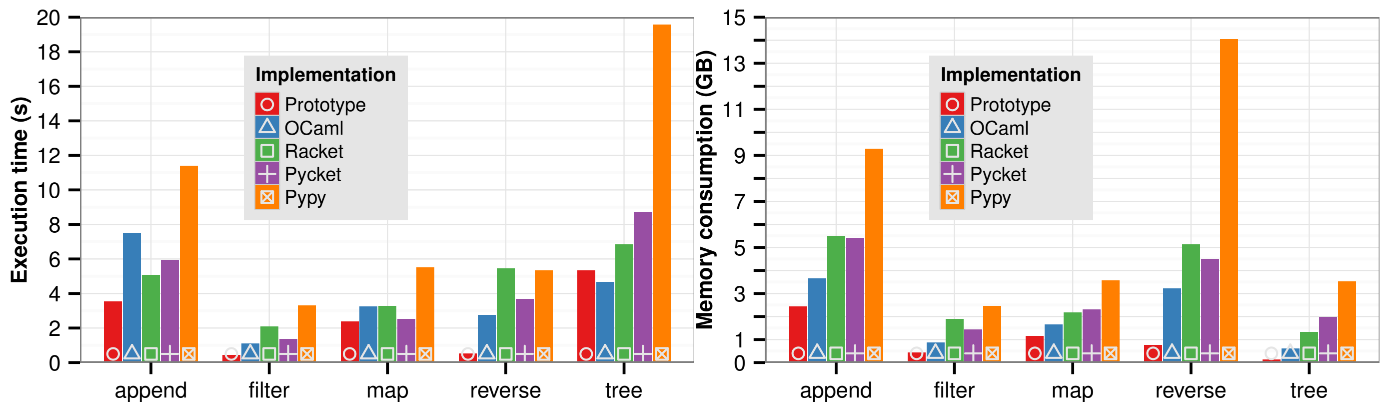

Figure 6: Benchmarking results. Each bar shows the arithmetic mean of ten runs for execution time (left) and memory consumption (right). Lower is better.

Tracing JIT compilers as introduced by Mitchell [26] have seen implementations for Java [19], JavaScript [20], or Lua[5], to name a few. In the context of a JavaScript implementation, the SPUR project [5] provided a tracing JIT compiler for Microsoft's Common Intermediate Language (CIL).

Tracing an *interpreter* that runs a program instead of tracing the program itself it the core idea of meta-tracing JIT compilers, pioneered in the DynamoRIO project [33]. PyPy [10, 30] is a meta-circular Python implementation that uses a meta-tracing JIT compiler. Provided through the RPython tool chain, other language implementations can benefit from a meta-tracing, e.g. Smalltalk [11], Haskell [35], PHP[6], or R[7].The meta-tracing JIT used in this work is provided by RPython, as well.

7. CONCLUSION AND FUTURE WORK

Our approach to just-in-time optimization of value classes provides very good initial results both for execution time and memory consumption for a small prototype implementation on selected micro-benchmarks. They are promising and motivate us to investigate the matter further.

Immediate next steps include the integration of our approach into existing programming language implementations. Here, languages that already have an implementation with a meta-tracing JIT compiler would be obvious candidates. Then, larger and more real-world benchmarks can be tackled.

Our aim is then to broaden the scope of our approach beyond value classes. We want to support objects that have identity as well as mutable objects. Yet, in the context of our optimization, these need more in-depth investigation.

Acknowledgments

We gratefully acknowledge the financial support of HPI's Research School and the Hasso Plattner Design Thinking Research Program (HPDTRP). Carl Friedrich Bolz is supported by the EPSRC *Cooler* grant EP/K01790X/1. We thank Alan Borning for comments on a draft version of this paper.

References

[1] J. Aycock. "A Brief History of Just-in-time". In: *ACM Computing Surveys* 35.2 (June 2003), pp. 97–113.

[2] D. F. Bacon. "Kava: a Java dialect with a uniform object model for lightweight classes". In: *Concurrency and Computation: Practice and Experience* 15.3-5 (Feb. 12, 2003), pp. 185–206.

[3] D. Bacon, S. Fink, and D. Grove. "Space- and Time-Efficient Implementation of the Java Object Model". In: *ECOOP 2002 Object-Oriented Programming*. Ed. by B. Magnusson. Vol. 2374. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, May 29, 2002, pp. 13–27.

[4] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: A Transparent Dynamic Optimization System". In: *ACM SIGPLAN Notices* 35.5 (2000), pp. 1–12.

[5] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. "SPUR: A Trace-based JIT Compiler for CIL". In: *SIGPLAN Notices* 45.10 (Oct. 2010), pp. 708–725.

[6] C. F. Bolz. "Meta-tracing just-in-time compilation for RPython". PhD thesis. Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.

[7] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. "Allocation Removal by Partial Evaluation in a Tracing JIT". In: *Proc. PEPM* (2011), pp. 43–52.

[8] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. "Allocation Removal by Partial Evaluation in a Tracing JIT". In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM '11. Austin, Texas, USA: ACM, 2011, pp. 43–52.

[9] C. F. Bolz, A. Cuni, M. Fijalkowski, M. Leuschel, S. Pedroni, and A. Rigo. "Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages". In: *Proc. ICOOOLPS*. 2011, 9:1–9:8.

[10] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. "Tracing the Meta-level: PyPy's Tracing JIT Compiler". In: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*. ICOOOLPS '09. Genova, Italy: ACM, 2009, pp. 18–25.

[5] `http://luajit.org`
[6] `http://hippyvm.com/`
[7] `https://bitbucket.org/roy_andrew/rapydo`

[11] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. "Back to the Future in One Week Implementing a Smalltalk VM in PyPy". In: *Self-Sustaining Systems*. Vol. 5146. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 123–139.

[12] C. F. Bolz, T. Pape, J. Siek, and S. Tobin-Hochstadt. "Meta-tracing makes a fast Racket". In: Dyla'14. Edinburgh, United Kingdom, June 2014.

[13] C. F. Bolz and L. Tratt. "The impact of meta-tracing on VM design and implementation". In: *Science of Computer Programming* (2013).

[14] C. Chambers, D. Ungar, and E. Lee. "An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes". In: *SIGPLAN Notices* 24.10 (Sept. 1989), pp. 49–70.

[15] L. P. Deutsch and A. M. Schiffman. "Efficient Implementation of the Smalltalk-80 System". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pp. 297–302.

[16] A. P. Ershov. "On Programming of Arithmetic Operations". In: *Communications of the ACM* 1.8 (Aug. 1958), pp. 3–6.

[17] M. Felleisen and D. P. Friedman. "Control operators, the SECD-machine and the -calculus". In: *Proceedings of the 2nd Working Conference on Formal Description of Programming Concepts - III*. Ed. by M. Wirsing. Elsevier, 1987, pp. 193–217.

[18] J.-C. Filliâtre and S. Conchon. "Type-safe Modular Hash-consing". In: *Proceedings of the 2006 Workshop on ML*. ML '06. Portland, Oregon, USA: ACM, 2006, pp. 12–19.

[19] A. Gal, C. W. Probst, and M. Franz. "HotpathVM: An Effective JIT Compiler for Resource-Constrained Devices". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. Ottawa, Ontario, Canada: ACM, June 14, 2006, pp. 144–153.

[20] A. Gal et al. "Trace-based Just-in-time Type Specialization for Dynamic Languages". In: *SIGPLAN Notices* 44.6 (June 2009), pp. 465–478.

[21] A. Gill, J. Launchbury, and S. L. Peyton Jones. "A Short Cut to Deforestation". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: ACM, 1993, pp. 223–232.

[22] Google, Inc. *Chrome V8 Documentation: Design Elements*. Sept. 17, 2012. URL: `https://developers.google.com/v8/design` (visited on 09/11/2014).

[23] E. Goto. *Monocopy and Associative Algorithms in Extended Lisp*. Technical Report TR-74-03. University of Tokyo, Japan, 1974.

[24] M. Hölttä. *Crankshafting from the ground up*. Tech. rep. Google, Aug. 2013.

[25] Microsoft Developer Network. *Common Type System*. Aug. 22, 2014. URL: `http://msdn.microsoft.com/en-us/library/zcx1eb1e(d=default,l=en-us,v=vs.110).aspx` (visited on 09/15/2014).

[26] J. G. Mitchell. "The Design and Construction of Flexible and Efficient Interactive Programming Systems". PhD thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 1970.

[27] M. E. Noth. "Exploding Java Objects for Performance". PhD thesis. University of Washington, 2003.

[28] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. *An overview of the Scala programming language*. Tech. rep. LAP-REPORT-2006-0001. Lausanne, Switzerland: EFPL, 2006.

[29] M. Paleczny, C. A. Vick, and C. Click. "The Java HotSpot Server Compiler". In: *Proceedings of the 2001 Symposium on Java Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Monterey, California: USENIX Association, Apr. 24, 2001.

[30] A. Rigo and S. Pedroni. "PyPy's approach to virtual machine construction". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 944–953.

[31] J. Rose. *JEP 169: Value Objects*. July 10, 2014. URL: `http://openjdk.java.net/jeps/169` (visited on 09/15/2014).

[32] Z. Shao, J. H. Reppy, and A. W. Appel. "Unrolling lists". In: *SIGPLAN Lisp Pointers* VII.3 (July 1994), pp. 185–195.

[33] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. "Dynamic Native Optimization of Interpreters". In: *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*. IVME '03. San Diego, California: ACM, June 8, 2003, pp. 50–57.

[34] A. Takano and E. Meijer. "Shortcut Deforestation in Calculational Form". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. FPCA '95. La Jolla, California, USA: ACM, 1995, pp. 306–313.

[35] E. W. Thomassen. "Trace-based just-in-time compiler for Haskell with RPython". MA thesis. Norwegian University of Science and Technology Trondheim, 2013.

[36] P. Wadler. "Deforestation: transforming programs to eliminate trees". In: *Theoretical Computer Science* 73.2 (1990), pp. 231–248.

[37] C. Wimmer. "Automatic object inlining in a Java virtual machine". PhD thesis. Linz, Austria: Johannes Kepler Universität, 2008.

APPENDIX

Table 1: All benchmark results. We give means of execution time and memory consumption along with the confidence interval showing the 95 % confidence level.

Benchmark	Prototype		OCaml		Racket		Pycket		Pypy	
	time	memory	time	memory	time	memory	time	memory	time	memory
append	3538±51 ms	2387 MB ±5 kB	7502±12 ms	3576 MB ±5 kB	5078±22 ms	5378 MB ±188 kB	5954±160 ms	5287 MB±129 kB	11408±87 ms	9074 MB ±5 kB
filter	439 ±8 ms	433 MB ±5 kB	1106±15 ms	850 MB ±8 kB	2094±11 ms	1848 MB ±195 kB	1352 ±8 ms	1398 MB ±96 kB	3313±32 ms	2399 MB ±10 kB
map	2365±32 ms	1134 MB ±5 kB	3241 ±7 ms	1607 MB ±7 kB	3276±17 ms	2116 MB ±131 kB	2526 ±35 ms	2249 MB±105 kB	5495±85 ms	3490 MB ±9 kB
reverse	530±15 ms	743 MB ±5 kB	2765±34 ms	3136 MB ±6 kB	5448±48 ms	5010 MB ±35 kB	3685 ±40 ms	4405 MB ±95 kB	5333±23 ms	13726 MB ±18 kB
tree	5324±57 ms	134 MB±694 kB	4670±42 ms	595 MB±10 kB	6852±28 ms	1285 MB±10503 kB	8739 ±68 ms	1939 MB ±20 kB	19576±77 ms	3446 MB±6842 kB