# Compiler-Assisted Object Inlining with *Value Fields*

Rodrigo Bruno
Oracle Labs
Switzerland
rodrigo.b.bruno@oracle.com

Vojin Jovanovic
Oracle Labs
Switzerland
vojin.jovanovic@oracle.com

Christian Wimmer
Oracle Labs
USA
christian.wimmer@oracle.com

Gustavo Alonso
Systems Group, Dept. of Computer Science, ETH Zurich
Switzerland
alonso@inf.ethz.ch

## Abstract

Object Oriented Programming has flourished in many areas ranging from web-oriented microservices, data processing, to databases. However, while representing domain entities as *objects* is appealing to developers, it leads to data fragmentation, resulting in high memory footprint and poor locality.

To improve memory footprint and memory locality, embedding the payload of an object into another (*object inlining*) has been proposed, however, with severe limitations. We argue that object inlining is mostly useful to optimize objects in the application data-path and that such objects have *value* semantics, unlocking great potential for inlining objects.

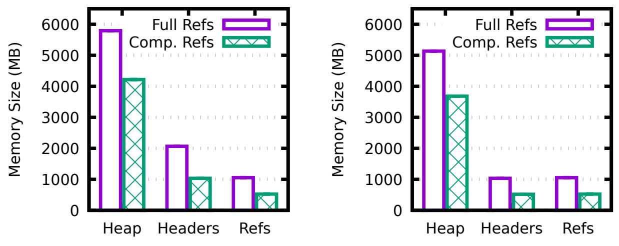
We propose *value fields*, an abstraction which allows fields to be marked as having value semantics. We take advantage of the closed-world assumption provided by GraalVM Native Image to implement Object inlining. Results show that using *value fields* requires minimal to no effort from developers and leads to improvements in throughput of up to 3×, memory footprint of up to 40%, and GC pause times of up to 35%.

***CCS Concepts:*** • **Software and its engineering** → **Compilers**; **Runtime environments**.

***Keywords:*** Object Inlining, GraalVM, Native Image

**ACM Reference Format:**
Rodrigo Bruno, Vojin Jovanovic, Christian Wimmer, and Gustavo Alonso. 2021. Compiler-Assisted Object Inlining with *Value Fields*. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3453483.3454034

ACM ISBN 978-1-4503-8391-2/21/06...$15.00
https://doi.org/10.1145/3453483.3454034



a) OpenJDK 11 HotSpot VM    b) GraalVM 20.3 Native Image

**Figure 1.** Memory usage to load the IMDB movie collection dataset (plain text size on disk = 854 MB, 6.3M entries).

## 1 Introduction

Object-oriented programming (OOP) languages such as Java, Python, and JavaScript are among the most popular programming languages used to date. However, by allowing developers to easily express domain concepts as *objects*, OOP languages promote partitioning of application data into many data objects, resulting in increased memory footprint and poor memory locality. This overhead is further aggravated in managed languages that tend to i) promote generalized *objectification* (everything is an object), and ii) embed metadata into object headers to help with language runtime tasks.

In managed languages, objects are typically represented in memory in two components: header, and payload (contents of the object). The object header contains the type of the object, as well as some additional information used for garbage collection, synchronization, hashing, etc. Object headers can account for up to 16 bytes in current production Java Virtual Machine (JVM) implementations such as OpenJDK HotSpot when references are not compressed (heaps larger than 32 GB cannot take advantage of compressed references). In many scenarios, such boxed primitives in Java, the object header corresponds to a large proportion of the total memory occupied by the object. Different runtimes have different object header sizes but, in overall, headers largely contribute to a higher memory consumption.

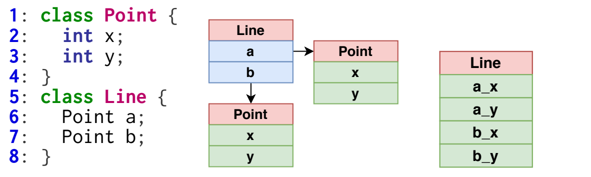
Figure 1 shows the amount of memory used by object headers and object references required to load a movie collection database (IMDB dataset [11]) into memory. Two VMs are analysed: the Java HotSpot VM of OpenJDK 11, and GraalVM 20.3 Native Image. For each VM, two variants are considered, with and without reference compression[36]. Results show that, out of the 5794 MB and 5133 MB required to load the dataset, 36% and 20% of the memory is dedicated to object headers, for HotSpot and Native Image, respectively. Object references also take significant amounts of space, 18% and 21% of the memory. The combined effect of headers and references accounts for up to 54% and 41% of the total memory required for the dataset. Enabling compressed references leads to an approximate reduction of 50% of the space used for object headers and references, but the remaining overhead is still significant as headers and references are still in place. In summary, partitioning data into large collections of domain objects has a high memory cost.

The overhead of OOP is particularly noticeable in applications/frameworks that handle massive amounts of objects in memory. Examples include in-memory caches [10, 23], data analytics [4, 8, 41], databases [7, 10, 14], among others. To mitigate the inefficiencies introduced by splitting application data into many data objects, we propose the use of object inlining [19, 20], a technique that reverts data separation by aggregating multiple objects into a single one. This idea is supported by our first key insight: **data objects are *confined***, i.e., object sub-graphs rooted by data objects are disjoint. Using object inlining, it is then possible to aggregate each of these data sub-graphs into a single object.

Aggregating multiple objects into a single one can lead to reduced memory bloat and improved memory locality. However, it introduces two main challenges that derive from relaxing the properties associated with objects: i) loss of object identity and ii) loss of atomic field access (more details in §2). To overcome these challenges, we rely on our second key insight: **data objects have value semantics**, i.e., data objects are used to carry values so neither identity nor atomic field access are required for these objects.

To take advantage of these insights, we propose *value fields*, a simple abstraction that enables fields to be marked as having value semantics, allowing the compiler to inline the marked fields. This new abstraction hides all the complexity of object inlining and offers a solution to have better control over the memory layout of application data, thereby reducing the memory footprint and improving memory locality. Fields marked as value fields are *inlined* upon field store, and copied into a newly allocated object upon field load. Compiler optimizations help reducing the pressure on the garbage collector by removing allocations of objects that can be escape analyzed. We show that *value fields* can be used in real-world frameworks to reduce memory footprint and improve throughput with minimal to no user effort.

We implement *value fields* as a compilation phase in the GraalVM Native Image builder [40]. We take advantage of the closed-world assumption and static analysis capabilities, that provide us with enough information to make inlining



```
class Point {
  int x;
  int y;
}
class Line {
  Point a;
  Point b;
}
```

a) Language Layout  b) Original Memory  c) Inlined Memory

**Figure 2.** Object inlining example.

decisions at image build-time. In addition, it allows easy integration with other compiler optimizations such as escape analysis [34], build-time initialization [40], and method inlining, that amortizes the costs of accessing inlined objects. We show that, for a variety of realistic use-cases, *value fields* can be used to reduce memory footprint and improve throughput with minimal or even no developer effort (more details in §6). Results show that memory footprint is reduced by up to 40% for data analytics using Apache Spark [41], throughput is improved by up to 3× for graph database requests using OrientDB, and garbage collection pauses are reduced by up to 35% for microservice requests for both Micronaut and Spring Boot.

In summary, this paper contributes with the following:

- It revisits the topic of object inlining, presenting the challenges that prevent it from being a generally applicable optimization;
- It proposes *value fields*, a simple abstraction which, when applied in a closed-world environment, unlocks significant potential for object layout optimizations (*object inlining*) resulting in reduced memory footprint and improved memory locality;
- It integrates *value fields* into GraalVM Native Image, a production system targeting Java applications;
- It evaluates the proposed technique using platforms and workloads inspired by real use-cases, showing when and how it can be effective for improving performance with little to no developer effort.

## 2 Object Inlining

Object inlining [19, 20, 37] is a technique that optimizes the memory layout of a set of objects. As described in previous work [19, 37], object inlining is applicable to two objects that are in a parent-child relationship. Parent-child relationships are one-to-many, meaning that one child has one parent but a parent may have multiple children. Object inlining can be applied multiple times over the same object graph until no more parent-child relationships exist. In the scope of this work, object inlining is used to replace a *parent field* by a set of *children fields*. Figure 2 presents a simple example of object inlining where `Line`, the *parent type*, has two fields of `Point` type, the *child type*, which will be inlined. After inlining is finished, the *children fields* (`Point.x` and `Point.y`) replace the original *parent fields* (`Line.a` and `Line.b`). The memory layouts before (center) and after (right) object inlining show that both the headers of the `Point` objects and the references (*parent fields*) were removed.

Object inlining produces a compact memory representation for object graphs at the expense of additional complexity to load and store parent fields. Using the example from Figure 2, a field store to `Line.a` is converted into a copy of `Point.x` and `Point.y` into `Line.a_x` and `Line.a_y`, respectively. A field load from `Line.a` is converted into the allocation of a new object of `Point` type followed by its initialization using the values of the `Line.a_x` and `Line.a_y`.

Finally, because object inlining rearranges the layout of types, type polymorphism in the *child type* is not allowed. Therefore, to enable inlining `Line.a` and `Line.b`, the `Point` type needs to be final, i.e., there can be no sub-types of `Point`.

### 2.1 Data Layout Optimizations in a Closed World

Data layout optimizations that involve changing type layouts (such as object inlining) are particularly hard to apply in language runtimes such as JVMs because once objects are allocated with their optimized layout, the optimization cannot easily be invalidated and reverted since changes have been committed to memory. Type layout deoptimization would require a complete memory re-write, converting all objects to their original memory layout, something we consider infeasible in terms of performance overhead. To avoid doing so, it is required that all type optimizations are proven to be applicable before the optimization is applied, and thus, speculative optimizations are often not possible or severely restricted. For example, the parent field (`Line.a`) cannot be inlined unless it is proven that all instances of `Point` have exactly two `int` fields. Language runtimes that allow dynamic class loading, for example, render this particular inlining candidate unviable as new sub-types of `Point` could be loaded with different type layouts.

To realistically apply type optimizations we argue that a closed-world environment is particularly important as it guarantees that all the application code is known at compile-time. Such an environment offers strong static analysis that significantly increases the chances of successfully applying type transformations such as object inlining. Therefore, to improve the applicability of the type transformations proposed in this work, we take advantage of the Native Image builder, provided as part of GraalVM. A closed-world environment is now feasible [40] and has been shown to work for a variety of real-world use-cases such as microservice frameworks like Spring Boot [17] and Micronaut [5].

### 2.2 Detaching Memory and Language Data Layouts

While the closed-world environment maximizes the number of potential candidates for object inlining, it still does not provide enough guarantees to automatically apply object inlining. There are two reasons to this: i) non-atomic parent field load/store; and ii) loss of object identity during inlining. The first issue emerges from the fact that a single parent field is replaced by a set of children fields and therefore, a single field access is now converted into multiple accesses (one for each child field). For example, using Java code to represent the before and after transformation logic, the following code

```
Point p = line.a;
```

will be converted into

```
Point p = new Point();
p.a = line.a_x;
p.y = line.a_y;
```

Since multiple field read and write operations are not guaranteed to be executed atomically, data races are possible. Solutions involving locks require expensive operations and would lead to additional memory to keep the lock state. Wide read and write operations could be a possible solution but these are often differently supported in different CPUs/architectures and require complex cache alignments in order to achieve an atomic operation.

The second issue stems from the fact that loading the parent field will result in the allocation of a new object which is not guaranteed to have the same identity as the original object stored into the parent field. For example, the following code would not succeed if `Line.a` is inlined:

```
line.a = p;
assert(line.a == p);
```

Maintaining object identity would require extra memory space to keep a reference to the original object, defeating the purpose of using object inlining to reduce memory footprint. Returning a copy of the object stored in the parent field also raises an additional problem with aliasing. For example, updates to the object returned by a parent field access will not be propagated back the original object and would be lost. This issue however, only applies if the objects are mutable, i.e., if the returned copy of the parent field can be modified.

Proving non-atomic access or loss of identity is difficult as objects often escape the scope of allocation (for example, when objects are inserted into a data structure). These two issues (non-atomic parent field access and the loss of object identity) prevent object inlining from being an automatic optimization technique since applications can potentially detect side-effects. We claim that to unlock type layout optimizations such as object inlining, new abstractions are needed to detach the language-level data layout from the memory layout. To this end, we propose *value fields*.

## 3 *Value Fields*

Often, big data and data science applications handle many objects with value semantics. Such objects carry data that needs to be processed but do not benefit from having an identity nor atomic field access. However, neither the compiler nor the language runtime can easily detect that such objects have value semantics and therefore, optimizations such as object inlining are severely restricted. To unlock memory

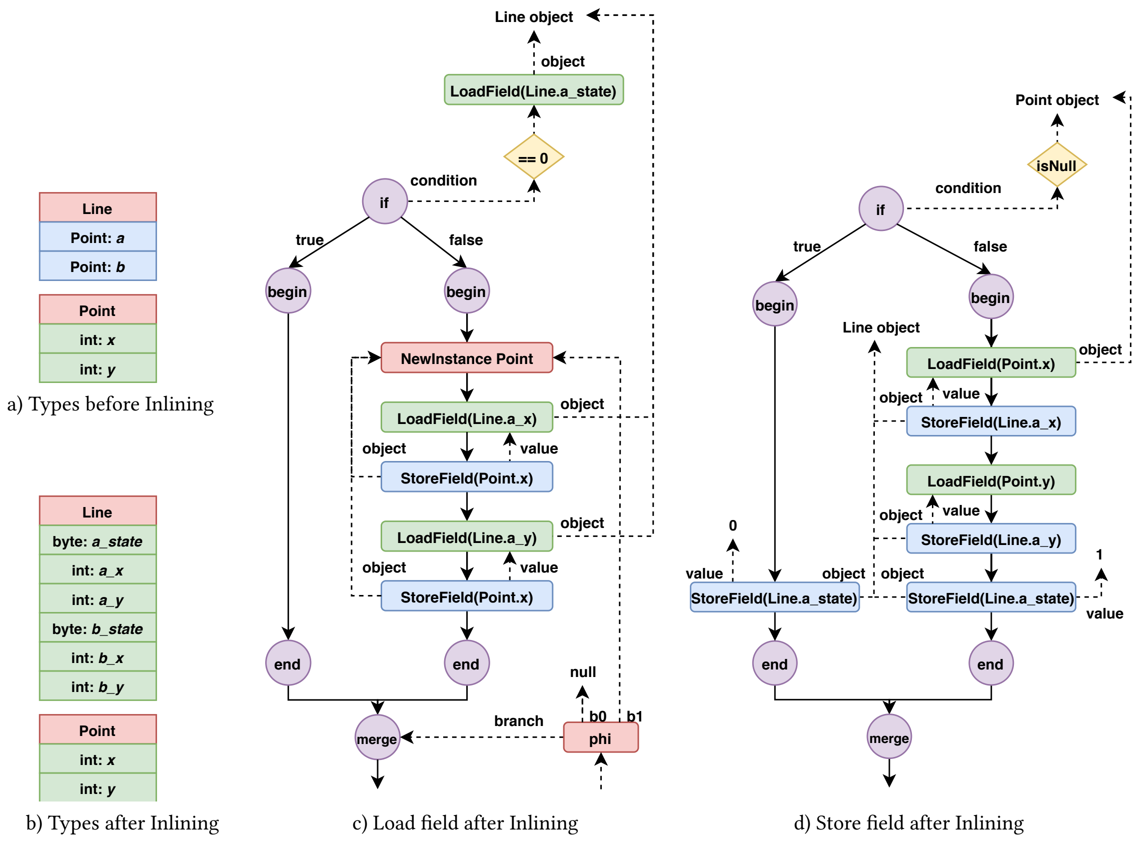


**Figure 3.** Type and field access transformations to inline `Line.a`.

layout optimizations, we propose *value fields*, a simple abstraction that allows fields to be marked as data carriers, i.e., as having value semantics. Fields marked as *value fields* will be selected by the compiler to be subject to type transformations and code transformations (update how to access inlined fields).

## 3.1 Type Transformations

Fields marked as *value fields* will be inlined at Native Image build-time. We continue using the initial example from Figure 2 and, in Figures 3.a and 3.b, we illustrate the type transformations for inlining `Line.a` and `Line.b`. This example is simple but yet representative of the transformations required during object inlining.

Type transformations use the following procedure. For each field marked as a *value field* (parent fields, `Line.a` and `Line.b`), remove it from the parent type (`Line`) and replace it by the respective children fields (`Point.x` and `Point.y`). Finally, a state field is also added (`Line.a_state` and `Line.b_state`) to keep track of whether the field is initialized or not.

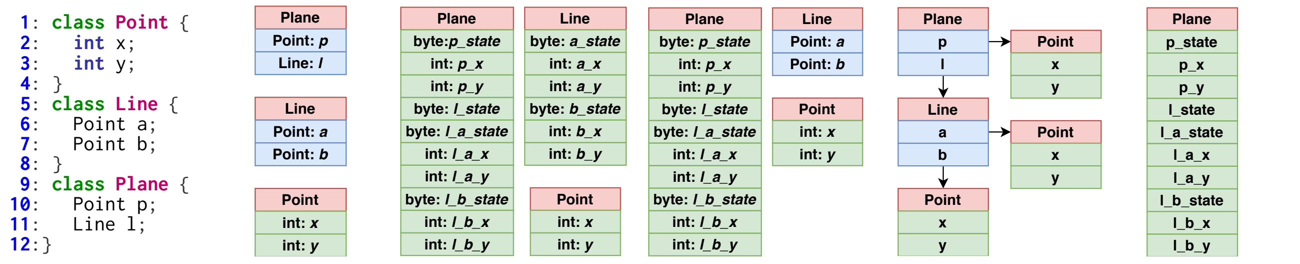
All types besides the parent type (`Line`) remain unchanged, thus limiting changes to the fields marked as *value fields*. No additional types are created.

## 3.2 Field Access Transformations

To cope with the type transformations just described, field loads and field stores to the parent field (field marked as *value field*) need to be updated. Figure 3.c and 3.d present a simplified version of the Graal compiler Intermediate Representation (IR) [22] graph after the field access transformations are applied. Solid arrows denote control flow while dashed arrows represent data dependencies.

**Load Field** To load a parent field (`Line.a` in this example), a single `LoadField` IR node is converted into the IR sub-graph presented in 3.c. In this sub-graph, an `if` node is utilized to separate the execution depending on whether the parent field is initialized or not. If it is initialized, then a new instance of the child type (`Point`) is allocated and all children fields are copied from the `Line` object into the newly allocated instance. If, on the other hand, the parent field is not initialized, a `null` value is passed down as a result. Depending on the branch taken at run-time, the `phi` node will provide the resulting value which replaces the value returned by the original `LoadField` (before the transformation).

**Store Field** A store to a parent field (`Line.a` in this example) is converted into the IR sub-graph presented in Figure 3.d. In this sub-graph, an `if` node is utilized to separate the execution depending on whether the value being passed for



```
class Point {
  int x;
  int y;
}
class Line {
  Point a;
  Point b;
}
class Plane {
  Point p;
  Line l;
}
```

**Figure 4.** Object graph inlining example.

the field store is null or not. If the value is non-null, all children fields are copied from the Point object into the Line object. The state field (Line.a_state) is set. If, on the other hand, the value being passed to the field store is null, all reference fields and the state field must be reset by storing a null value in reference fields and 0 in the state field. Resetting all reference fields avoids memory leaks as these references could never be accessed by the application again but the garbage collector would not be able to collect the objects referenced by them.

### 3.3 Type Layout Optimizations

For performance reasons, we allow extra information to be passed to the compiler to indicate specific properties of children fields used to optimize the layout of the parent type. In particular, we allow two properties to be defined: a) children fields that have a non-null value, and b) children fields that can be recomputed if needed. The former (non-null fields) can be used as a replacement for the state field as it will only have a null value if the parent field is not initialized. The latter can be used to ignore particular children fields that can be discarded during inlining. We evidence the usability of these properties using the String type as an example.

String objects often represent a large portion of application data objects and, in many cases, String objects have value semantics (i.e., the object is only used as a data carrier). Strings are wrappers for a byte array which stores the String's content. For a given String, the byte array (from here on called String.value) is always initialized upon the initialization of the String object. Taking advantage of this fact, this field is marked as a non-null field and therefore no state field is required and all checks are performed directly on the String.value field. This optimization further reduces the memory footprint (no state field) and also avoids both the set and unset operations on the value field (required for inlined field stores). Strings also contain a hash field which caches the result of hashing the String's content. This particular field can be recomputed if necessary. To save extra memory space, we skip this field during inlining.

## 4 Object Graph Inlining

Object inlining is not limited to one-level inlining but instead, it supports multi-level inlining or, in other words, object graph inlining. Figure 4.a shows a simple extension of the example presented in Figure 2. In this example, a single instance of Plane is the root for an object graph containing one Line instance and three Point instances (Figure 4.e). Using object graph inlining, it is possible to compact all five objects into a single object as shown in Figure 4.f. We present two variants for object graph inlining.

***Value fields*** can be used to inline entire object graphs into a single object. For instance, it is possible to mark as *value fields* all non-primitive fields in the presented example (Line.a, Line.b, Plane.p, and Plane.l). This will result in type transformations not only in Line, but also in Plane, as depicted in Figure 4.c. Loading Place.l will return an object of Line type which inlines both Point fields.

***Value graphs***, a different inlining primitive, can also be utilized to inline object graphs. Fields marked as *value graphs* will inline the entire object graph but type transformations will be limited to the parent type. For example, if both Plane.p and Plane.l are marked as *value graphs*, and no other fields are marked as *value fields* or *value graphs*, only the Plane type will be transformed (see Figure 4.d). This object graph inlining variant is beneficial when changing children types is not possible.

Both variants of object graph inlining produce the same inlined memory layout for an instance of Plane type (see Figure 4.f). The algorithm used for our inliner is depicted in Algorithm 1. In the first phase (lines 2-9) all possible parent fields are considered. If a particular parent field contains children fields that are also parents to other fields (line 6), then this parent field is deferred for later inlining (line 7). Otherwise, the field is inlined (line 9).

After the first phase is finished, all one-level inlining is finished and all the remaining parent fields will be inlined in the second phase (lines 10-17). The idea behind the second phase is to inline fields from the bottom to the top, i.e., all parent fields whose children fields are not parent fields to other children, are inlined first. The algorithm converges after all inlineable fields have been inlined. For simplicity,

**Algorithm 1** Object graph inlining.

```
1: $queue \leftarrow []$
2: for $parent\_type$ in $known\_types$ do
3:   for $parent\_field$ in $fields(type)$ do
4:     if $is\_inlineable(parent\_field)$ then
5:       $child\_type \leftarrow type(parent\_field)$
6:       if $has\_inlineable\_fields(child\_type)$ then
7:         $queue.push(field)$
8:       else
9:         $inline(parent\_type, parent\_field)$
10: while $not\_empty(queue)$ do
11:   $parent\_field \leftarrow queue.pop()$
12:   if $is\_inlineable(parent\_field)$ then
13:     $child\_type \leftarrow type(parent\_field)$
14:     if $has\_inlineable\_fields(child\_type)$ then
15:       $queue.push(parent\_field)$
16:     else
17:       $inline(parent\_type, parent\_field)$
```

several methods are left out. In particular, `is_inlineable` checks if the field is marked as a *value field* or *value graph*, and if the type of the field is monomorphic. Arrays, primitive fields, and fields marked as `volatile` are also not considered for inlining. Cyclic data structures are also automatically ignored. The `inline` method internally updates the compiler data structures to accommodate the changes in the parent type (which depend on the variant of object inlining).

## 5 Using *Value Fields*

*Value fields* combine semantics from value types and reference types. When using *value fields*, deciding if a particular object is passed by reference or value does not depend on the type, but rather on the operation in which the object is being utilized. From the previous example, instances of `Point` are always passed by reference except when being loaded/stored from/to a field marked as a *value field*.

Fields can be marked as *value fields* either through a Java field annotation (`@ValueField` or `@ValueGraph`), or through a configuration file (JSON file which contains a list of Java value graph/fields). By default, only *value fields* of immutable child type or *value graphs* of immutable child type hierarchy are inlined. This restriction prevents lost updates resulting from the lack of aliasing between the object returned by a parent field load and the inlined field. For example, if a parent field of a mutable type is inlined, a store to an object returned by a parent field load will not be propagated to the inlined field. This problem does not occur when inlining is restricted to immutable child types as no updates are possible.

Our experience using *value fields* to inline objects processed by large frameworks such as Spring Boot, Micronaut, or Spark, suggests that identifying candidate fields for inlining is a simple task, taking no more than a few minutes per application. To further simplify this task, we developed a JVMTI-based agent that can be used for profiling when running the same application on the HotSpot VM. The agent periodically traces the entire Java heap and tracks fields referencing confined object graphs, i.e., disjoint object graphs that have a single incoming reference. Such fields are reported to developers as candidates for inlining. By reporting fields referencing confined object graphs, the profiler helps reducing potential memory overheads resulting from inlining the same object in multiple locations. The profiler, however, does not guarantee that the application semantics won't be impacted due to the loss of object identity or non-atomic inlined field access.

We also noted that in all the frameworks we analyzed so far, most data objects are immutable and neither object identity nor atomic field access are necessary. On the one hand, object identity is often used to implement a fast-path for the `equals` method but it does not compromise correctness. On the other hand, synchronization among multiple worker threads is usually done at a much coarser grain to avoid inter-worker synchronization overhead and is commonly provided at the data structure entry level.

## 6 Evaluation

We evaluate different aspects of a set of applications we use to test *value fields*. Our analysis is focused on three main metrics: footprint reduction, throughput improvement, and effort to integrate into existing applications/frameworks. While the first two metrics are easy to measure experimentally, the third required us to utilize and deploy different applications and try to assess the extent of changes required.

Object inlining is implemented as a compilation phase of the GraalVM 20.3 Native Image builder. The new compilation phase is executed early in the compilation pipeline (right after generating the Graal IR [22]) so that the transformed code can benefit from all the existing compiler optimizations such as method inlining and escape analysis to optimize the code produced by the inlining transformation.

Experiments run in isolation for at least 10 iterations (more iterations are used if the results take longer to stabilize). The last 5 iterations are utilized to create average values. The standard deviation resulting from measurements is low in most experiments and therefore we only include it in our plots if it is above 5%. *Value fields* produce no measurable footprint overhead and less than 1% increase in compilation time during Native Image building. The default Native Image Garbage Collector (GC) is utilized in all experiments (using the recently added Garbage First Native Image GC did not affect the benefits of *value fields*). Experiments run in a single cluster node running Debian 10 (Linux kernel 4.19.0-10) equipped with an Intel(R) Xeon(R) CPU E3-1225 v6 @ 3.30GHz, and 32GB of DDR4 DRAM. CPU frequency scaling and hyper-threading are disabled.

The remainder of this section is divided into sub-sections, each exploring a specific use-case. We picked different use-cases from different areas ranging from data analytics (Apache

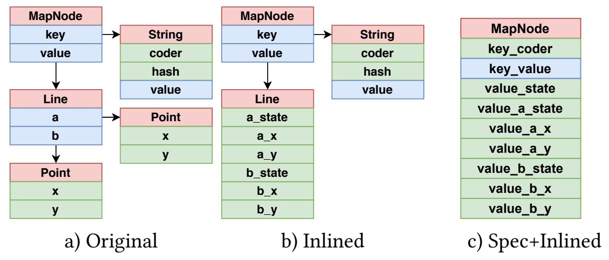


**Figure 5.** Combining type specialization and inlining.

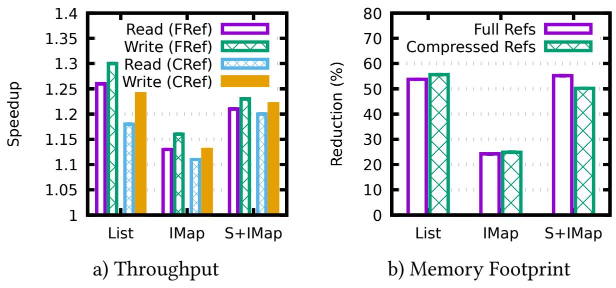


**Figure 6.** Performance of data structures with *value fields*.

Spark), Microservices (Spring Boot and Micronaut), to Graph Databases (OrientBD) to illustrate the wide applicability of *value fields*. We also benchmark the effect of object inlining on widely used Java data structures and take advantage of DaCapo [18] and Renaissance [32] to study the applicability of object inlining on a wider spectrum of applications.

### 6.1 Optimizing Java Generic Data Structures

We start by demonstrating how *value fields* can be used to improve both the memory footprint and throughput of Java generic data structures. To this end, we select two of the most widely used data structures in the Java Development Kit (JDK): `ArrayList<V>` and `HashMap<K,V>`. We parametrize both data structures using `Line` (as value) and `String` (as key, only for `HashMap`). These two data structures are selected as representative of other JDK generic data structures.

Internally, an `ArrayList` contains an `Object` array which keeps references to the objects inserted into the data structure. In this section, we use *value fields* to inline the fields of the `Line` type and compare to a version of the same data structure with no inlining (see example in Figure 2). Similarly, `HashMaps` also keep references to map entries inside an array of `MapNode`. Each `MapNode` contains a reference to a key and a value (Figure 5.a). The resulting type layout of using *value fields* to inline the fields into `Line` is depicted in Figure 5.b. To maximize throughput and reduce memory, we further inline both the `key` and `value` fields in `MapNode` using a technique called Type Specialization (described below). Figure 5.c represents the final layout of `MapNodes`. By combining inlining with specialization, it is possible to reduce by 3× the number of objects utilized in `HashMaps`.

Type Specialization [21, 33, 35] is a technique that allows the specialization of generic data structures by allowing the creation of specialized instances of such data structure. As opposed to the regular utilization of generic Java data structures, which are subject to type erasure during compilation and rely on artificial type casts introduced by the (Java) compiler to complement the data structure implementation, specialized data structures keep their type information until run-time and therefore unlock inlining opportunities. For example, specialization assigns a concrete type to `MapNode.key` (`String`) and `MapNode.value` (`Line`) whereas in the original generic version both fields are of `Object` type.

Specialized data structures are offered through a factory provided as a library to applications. Developers simply need to replace their regular generic data structure allocation

```
Map<String,Line> map = new HashMap<>();
```

by

```
Map<String,Line> map =
    newHashMap(String.class,Line.class);
```

During Native Image building, calls to the factory methods are statically analyzed and all data structure specializations are created to accommodate all calls to factory methods. At run-time, upon calling the factory method `newHashMap`, a specialized instance is returned. We currently implement specializations for a variety of the most widely used generic Java data structures.

To evaluate the proposed data structures, we utilize a simple micro-benchmark which performs random read and write operations. Results (Figure 6) show that inlining leads to both read and write speedups in all three data structures variations: `ArrayList` (List), Inlined `HashMap` (IMap), and Specialized and Inlined `HashMap` (S+IMap). Speedups are more pronounced when using full references (FRefs) as the locality is significantly improved by inlining objects. Specialization also has a positive performance impact as it also improves locality by avoiding one extra memory indirection to access both the `key` and `value` fields in `MapNode`. Memory footprint reduction ranges from 25% for IMap, and up to 55% for List and S+IMap. Benefits come from reducing the number of object headers and object references. In sum, generic Java data structures can greatly benefit from object inlining with minimal user involvement. Results show speedups of up to 30% and memory reductions of up to 55%.

### 6.2 Value Inlining in Microservice Caches

Microservices [28] have recently received a lot of attention from the software industry as a way to split monolithic applications into smaller, more maintainable, isolated, and easier to deploy services. Large companies such as Netflix [3], Amazon [2], Ebay [6], and Uber [1] have transitioned several of their services/applications into microservice architectures.

As a result many Microservice frameworks are now available to help users build, manage, and deploy microservices

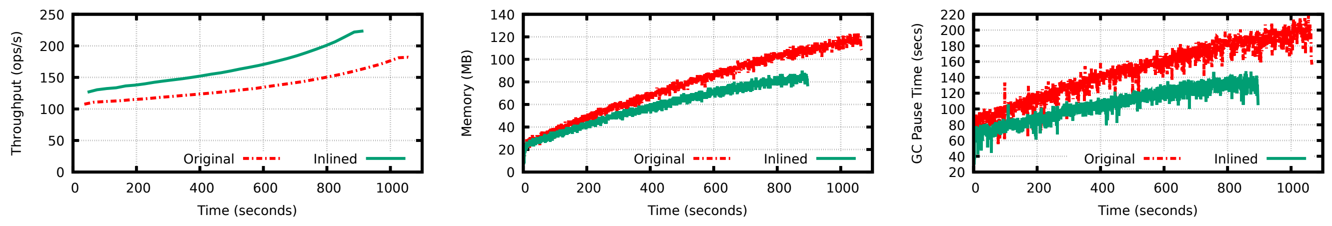


**Figure 7.** PetClinic Throughput (left), Memory Footprint (center), GC Pause Time (right).

more easily. Popular frameworks include, for example, Spring Boot [17] and Micronaut [5]. Among many of the functionalities provided by these frameworks, caching of requests (through a `@Cacheable` annotation) is a common built-in feature. Cacheable requests usually keep in memory the result of a database request, therefore improving throughput, but at the expense of higher memory footprint. In this section we evaluate *value fields* for reducing the footprint and improving the throughput of two popular microservice frameworks, Spring Boot and Micronaut.

**6.2.1 Spring Boot PetClininc.** To benchmark Spring Boot with *value fields* we take advantage of a popular demo application, PetClinic [1]. The setup includes: i) a MySQL Server 8 installation that keeps a database with all the state; ii) an instance of the PetClinic application; and iii) JMeter [9] that produces load based on a realistic dataset (which includes names of people, addresses, etc). Requests are issued using a combination of the services provided by the microservice and try to emulate real users using the website.

To trigger object inlining, we look at the domain types in the PetClinic application and create an object inlining configuration file with all the fields that should be inlined. In total, 12 fields are marked for inlining across 6 different domain types as can be seen in following configuration file. The configuration file is loaded by the Native Image builder and thus no changes to the application source code are required.

```
{
  "value_fields" : {
    "petclinic.model.BaseEntity" : ["id"],
    "petclinic.model.Person" : ["firstName", "lastName"],
    "petclinic.model.NamedEntity" : ["name"],
    "petclinic.owner.Owner" : ["address", "city",  "phone"],
    "petclinic.owner.Pet" : ["birthDate", "type"],
    "petclinic.visit.Visit" : ["date", "desc", "petId"]
  }
}
```

Figure 7 shows the experimental results for PetClinic's throughput, memory footprint, and GC latency, respectively. All plots compare the original deployment of PetClinic (Original) with the version using *value fields* (Inlined). Results clearly indicate that throughput increases as time goes on (this is a side-effect of more requests being served directly from the in-memory caches) but the Inlined deployment is always superior in terms of requests per second. After the initial warmup, the Inlined deployment of PetClinic has 23% higher throughput. At this point, requests are both being served from the database and from the cache showing that, in both situations, *value fields* makes request handling faster.

Memory footprint and GC latency follow the same trend. After the initial warmup, the memory footprint of PetClinic is reduced by 33% and the GC latency, important for long tail latencies of application requests, drops by 35%.

**6.2.2 Micronaut ShopCart.** The same approach of preparing a configuration file for object inlining that was used for Spring Boot could also be applied to Micronaut. However, since Micronaut performs most of its framework setup logic at (Java) compilation-time (during annotation processing to be specific), we extended Micronaut's annotation processing engine to automatically configure object inlining for cached objects. With such extension, no user involvement is required and applications that use framework-based caching automatically benefit from *value fields*.

To benchmark Micronaut with *value fields*, we developed a simple application called ShopCart, which has similar operations when compared to PetClinic, but in a different domain (online shopping). One of the domain types used in the application and returned in a `@Cacheable` request is `Product`. The following code shows the Java representation of the generated type whose instances are saved inside the microservice cache instead of the original `Product`:

```
class Value$Product {
  @ValueGraph Product p;
  public void inline(Product p) { this.p = p; }
  public Product deinline() { return this.p; }
}
```

Our extension of Micronaut uses `inline` and `deinline` when inserting and retrieving into/from the cache, respectively. Note that, in this use-case, inlining is used to compress the memory layout only when objects are stored inside the cache. This is made possible by creating a wrapper type (`Value$Product` in this example) that inlines the original `Product` object. At run-time, Micronaut automatically intercepts cache accesses and calls `inline` and `deinline` when inserting and retrieving objects from the cache (respectively).

Similarly to the PetClinic experiments, JMeter is utilized to produce load on the microservice by issuing a combination of requests that emulate user requests on the website. In this

[1]https://github.com/spring-projects-experimental/spring-graalvm-native/tree/master/spring-graalvm-native-samples/petclinic-jdbc

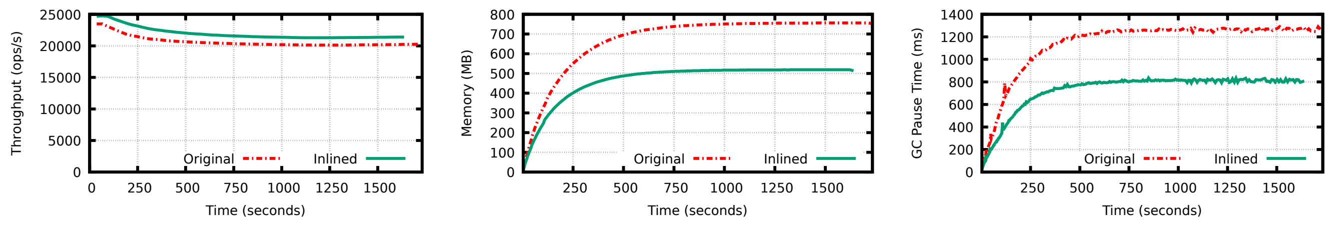


**Figure 8.** ShopCart Throughput (left), Memory Footprint (center), GC Pause Time (right).

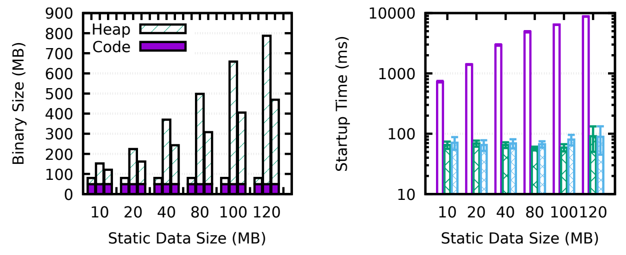


**Figure 9.** ShopCart static data initialization at run-time (left), build-time (middle), using inlining at build-time (right).

specific use-case, we do not use a backing database and all information is kept inside the microservice caches.

Results are reported in Figure 8. After an initial warmup, results stabilize for both the deployment with (Inlined) and without (Original) object inlining. Throughput of the Inlined deployment shows an improvement of approximately 7%. This improvement is less significant compared to PetClinic because objects are only inlined while stored inside the cache. When objects are retrieved form the cache, the original layout is utilized. A throughput improvement means that the overhead of restoring the original objects (during de-inlining) is more than compensated by having fewer objects in memory (thus reducing the pressure on the runtime).

Memory and GC latency show significant improvements. After the initial stabilization period, memory is reduced by 32% and GC latency is reduced by 35%. These performance benefits come with zero user involvement as all object inlining setup is performed at compilation-time using Micronaut.

We also analyze the tradeoff between run-time and build-time initialization and how *value fields* can reduce the size of binaries produced by the Native Image builder. We deploy ShopCart with a static table of product prices and descriptions. This data can be loaded into the application a) at run-time, in which case the loading time is included into the startup time of the microservice or, b) at build-time, reducing startup time but increasing the binary size generated by the Native Image builder. Figure 9 shows how *value fields* can be used to reduce the size of static data structures initialized at build-time.

Results show that through object inlining, the binary size can be reduced by up to 40%, leading to a total package size (binary size plus static data size) increase of 1.3-2.16× compared to run-time initialized. In exchange for the increased binary size, startup time (time until the microservice is ready to serve requests) is reduced by up to 98.3×. This great reduction in startup time is the result of pushing dataset loading time to built-time. Loading the static data and inserting all entries into an in-memory table (`HashMap`) takes up to 8.7 seconds for a 120 MB dataset with 10M entries.

**Table 1.** Spark RDD queries.

| Query | Description |
|---|---|
| Q1 | Number of movies released in a year by genre. |
| Q2 | Movies ordered by movie rating. |
| Q3 | Average age of a movie's actors. |
| Q4 | Actors ordered by number of roles. |
| Q5 | Year with more average rating vores. |
| Q6 | Actors ordered by number of roles in highly rated movies. |

## 6.3 Data Analytics with Spark

We now look at how object inlining can be used to optimize data analytics using Apache Spark [41]. We take advantage of a public movie dataset with 6.3M entries [11] which is loaded into a `SparkRDD` and used to execute a number of queries (see Table 1). Queries are implemented using a mix the most common SparkRDD operations (map, filter, reduce, flatMap, etc).

The Spark application contains only two domain types: `Movie` and `Actor` and all non-array and non-primitive fields are marked using the `@ValueField` annotation (6 annotated fields across 2 domain types):

```
class Movie {
  Actor[] actors;
  @ValField String genre;
  @ValField String name;
  @ValField Date release;
  int votes;
  float rating;
}
```

```
class Actor {
  @ValField Date birth;
  @ValField Date death;
  @ValField String name;
}
```

We run Spark in a single node using 8 threads and 16GB of memory. We anticipate that, in a cluster setting, most benefits and conclusions are similar as the proposed optimizations have effect on memory consumption and query processing time, and not on data distribution over the network. Since Spark is not yet supported by the Native Image

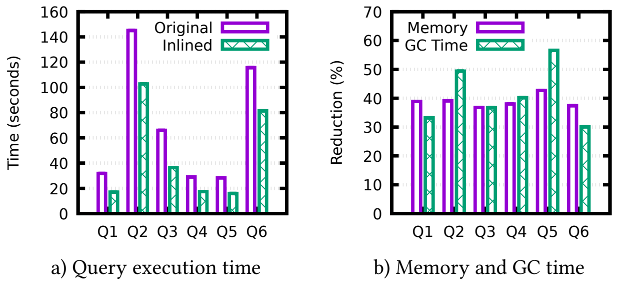


**Figure 10.** Spark performance using *value fields*.

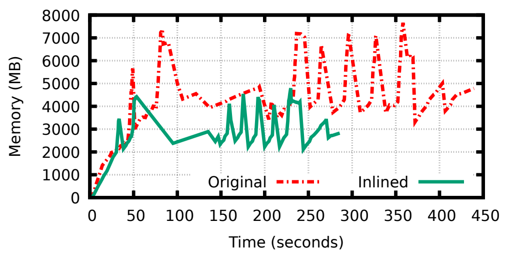


**Figure 11.** Spark memory footprint.

builder, we perform all inlining transformations by hand. No other changes to the application (query implementation) or to Spark platform are required.

Results for query execution time show an average speedup of 1.35× compared to the original Spark deployment (see Figure 10.a). This improvement shows that since objects are more compact in memory, Spark engines can process data faster. Improvements can also be measured for memory footprint (up to 40%) and GC latency (up to 58%). Both memory consumption and GC latency are a direct benefit of optimizing the memory layout of objects to reduce the number of headers and object references.

Figure 11 shows the memory utilization trace throughout an entire run of loading the dataset, followed by a single execution of each query. Comparing both deployments, one can see that both curves share the same number of peaks and relative duration but the inlined deployment presents lower and earlier peaks, showing that each query used less memory and executed faster. In sum, using *value fields* yields not only memory footprint reductions, but also throughput improvements.

## 6.4 Graph Processing with OrientDB

We now look into how inlining can be helpful to optimize graph/object databases. We take advantage of OrientDB [14], an opensource graph/object database which we use to store a subset of S2ORC [27], a public collection of research articles. Once the database is loaded, we perform a number of popular queries such as fetching all citations of a paper/researcher, or calculating the journal impact factor (see Table 2).

**Table 2.** OrientDB queries.

| Query | Description |
|---|---|
| Q1 | Get all citations of a paper. |
| Q2 | Get all citations of a researcher. |
| Q3 | Calculate the hIndex for a researcher. |
| Q4 | Calculate the i10Index for a researcher. |
| Q5 | Calculate the Impact Factor for a journal. |

Queries are implemented in Java using a few domain types. In total, 5 fields are annotated across two domain types:

```
class Paper {
  @ValField String id;
  @ValField String title;
  @ValField Publication pub;
}
```

```
class Researcher {
  @ValField String id;
  @ValField String name;
}
```

OrientDB's object API accepts object graphs which are then serialized and merged into its internal graph representation. The database data is stored in off-heap memory (memory which is not managed by the garbage collector). OrientDB serializes object graphs to off-heap memory using their database format. Since OrientDB is not supported yet by the Native Image builder, we perform all inlining transformations by hand. No other modifications to the application or to OrientDB database are performed.

Throughput (see Figure 12.a) shows a significant speedup ranging from 2.5x to 3x for queries and 2x for writing new article entries into the database. *Value fields* leads to higher read and write rates compared to the original deployment, a direct consequence of reducing the number of objects involved in serialization/deserialization to/from off-heap. Footprint improvements are depicted in Figure 12.b. For simplicity, we show the memory traces for a single execution for a workload that starts by loading the dataset (the initial increase in memory utilization), followed by a query execution phase. During query processing time, memory utilization does not increase as no new entries are inserted into the database. Memory utilization is positively impacted by object inlining, showing an 18% reduction compared to the original deployment. It is relevant to note that this example shows that *value fields* can effectively reduce the size of serialized objects. Results for GC latency are not depicted as most of the memory is allocated off-heap and therefore, GC latency is negligible as only a few objects reside inside the heap.

## 6.5 DaCapo and Renaissance Benchmark Suites

DaCapo [18] and Renaissance [32] are popular benchmark suites that represent a wide spectrum of applications. In this section, we use benchmarks from both suites to measure the improvement of *value fields*. From the benchmarks included in both suites, we exclude benchmarks that are currently not supported by Native Image (due to missing Native Image build configuration, required for applications using dynamic features such as serialization and `Unsafe` memory access).

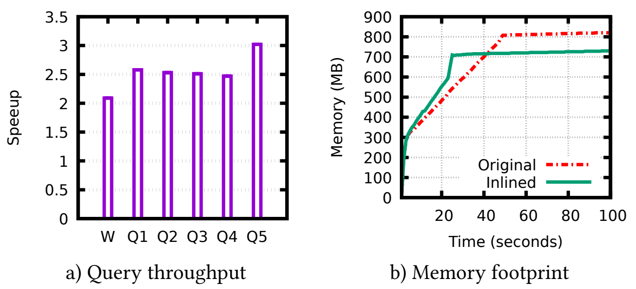


**Figure 12.** Performance of OrientDB with *value fields*.

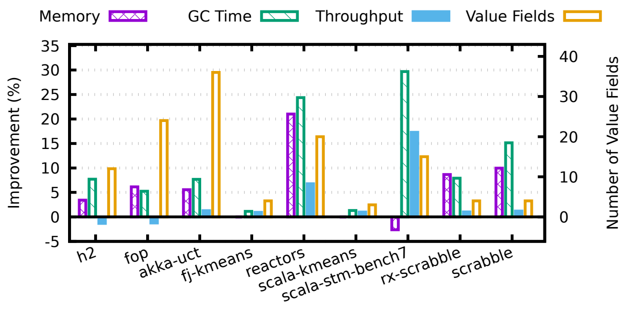


**Figure 13.** DaCapo and Renaissance with *value fields*.

In addition, all applications with low memory footprint (less than 5 MB) are also excluded.

For each application, we create a configuration file which contains the fields that should be considered for inlining. To determine such fields, we first use the JVMTI profiler (see Section 5) to create an initial set of fields to inline. From this set, we eliminate all fields whose type is mutable. Finally, we analyze a heap dump of the running application looking for objects with high retained memory size (data structures) and we limit the inlining candidates to fields of objects inside the data structure.

Normalized improvement percentage results for Memory, GC Pause Time, Throughput are included in Figure 13. The plot also includes the number of fields selected for inlining (right axis). Results show that *value fields* improve the performance across of most metrics in all applications. Only three benchmarks report degradations in one of the three metrics. In `scala-stm-bench7`, a 2% memory footprint degradation results from some objects escaping the scope of allocation (inlined field loads) but it compensated by 17% improvement in throughput and 30% reduction in GC pause time. In `h2` and `fop`, a 1% throughput reduction is compensated by 4% and 6% memory footprint reduction, respectively. All other benchmarks achieve improvements across all three metrics. Benchmarks with low memory footprint (less than 5 MB) are not presented as no fields are selected for inlining and therefore no improvement or degradation is reported.

## 7 Related Work

**Value types, fields, and objects** (or structures, as in C# and Swift) offer value semantics and therefore do not require identity and do not provide atomic access, making them good candidates for type optimizations which minimize memory footprint and improve data locality. However, the dichotomy between value types and reference types forces developers to have completely separate types (and semantics) for data and control, increasing the complexity and code maintainability effort. Languages such as Java, Python, and JavaScript opted for a uniform type system where all composite types are reference types leading, however, to performance issues.

Recent efforts have been trying to bridge the performance gap between value and reference types. Project Valhalla [13] is an ambitious experimental project which aims at bringing value types and data structure specialization into Java. It does so by allowing objects to be marked as immutable values at run-time through a new method introduced into `Object`, the root of the Java type system. Objects marked as immutable values can no longer be used in identity exposing operations (such as reference equality) and cannot be modified. Just-In-Time (JIT) compilers are then free to optimistically optimize code that handles objects marked as immutable values. Deoptimization checks are also automatically introduced by the JIT compiler, and exceptions are thrown if the application tries to mutate an already immutable object.

Compared to *value fields*, project Valhalla requires significant changes to core language libraries and core runtime components such as the interpreter and JIT compilers, increasing the complexity of the design and implementation. We also argue that handling the conversion from value to reference at the field access level, compared to explicitly calling a method on the object, is not only a simpler abstraction for developers, but also limits the number of cases in which both reference types and value types are involved, possibly reducing the number of complex and ambiguous scenarios.

Records [16] were recently introduced as a new data type in the Java type system. Records are specifically designed to represent immutable data objects. The goal of this new proposal is to reduce boilerplate code and not to improve performance. Records still have *object* properties such as identity and atomic access, and therefore are not candidates for object inlining.

Scala Value Classes [26] provide a mechanism for types extending `AnyVal` to be automatically inlined. However, these types have many restrictions such as being limited to having a single field. In practice, value classes can be used as a tool for type aliasing. We argue, however, that declaring types as values creates a type dichotomy which increases complexity and prevents already existing types which may have value semantics in particular scenarios, from being inlined.

**Object inlining** is not a new idea [19, 20]. Wimmer et al. [38] proposed an automatic feedback-oriented object inlining technique for the Java HotSpot VM. The goal was not to reduce the memory overhead but to optimize the performance of field accesses instead. This is done by co-allocating objects with their fields consecutively in memory and replacing field accesses by address arithmetic. In order to find potential candidates for this optimization, profiling code is installed during Just-In-Time (JIT) compilation.

The two preconditions for this optimization are i) that the object and its fields must be allocated together, i.e., the field store must occur right after the allocation of both objects, and ii) the field should not be modified to reference other objects. The profiling code installed in the class loader and reflection system makes sure no newly loaded code or reflection call could violate the preconditions. If the latter occurs, code deoptimization must be triggered to remove the optimized field accesses.

The idea of optimizing field accesses was further extended to arrays [39]. Inlining arrays required the same preconditions as for inlining regular objects. Inlining of objects inside arrays is not handled due to the missing type information in Java bytecodes, preventing a safe optimization.

Haubl et al. [25] propose an optimized version of *String* for the Java HotSpot VM based on inlining the character array (previously, a field of the *String* type) directly into the *String* type. According to the authors, the motivation for this work was that the vast majority of *String* objects do not share their character array with other *String* objects and therefore, the character array could be directly inlined into *String*. Unlike the previously discussed work, this approach proposes that the header of the character array should be removed when inlining it into *String*.

Pape et al. [31] also proposed taking advantage of the JIT compiler to identify specific types that contain fields that are either primitives or object references not modified after initialization (in other words, immutable). A prototype implementation of this technique was presented for RPython. Gope et al. [24] presented a technique for inlining hashtable keys to provide faster lookup access. This technique was also based on a profiler that detects frequent accesses to specific hashtables. Once the optimization is triggered, the hashtable keys are moved into an array so that faster key lookup is possible. This optimization also relies on the assumption that the number of keys is constant.

Compared to previous object inlining techniques, our proposal improves the state of the art in three ways. First, taking advantage of the closed-world environment, which has been made a viable approach in recent years, greatly improves the range of applicability of object inlining as all types are known statically and inlineable fields can easily be identified. Second, *value field* further unlocks the potential of object inlining by allowing the compiler to treat objects as values. This abstraction allows us to overcome the barriers imposed by proving that identity of inlining candidates is not exposed and that concurrent field access is not possible. Both conditions are hard to prove, especially if objects escape the scope of allocation, therefore imposing severe restrictions on the candidate fields for inlining. Third, the proposed object inlining design aims at exploiting all existing compiler optimizations. To this end, inlining is performed early in the compilation pipeline so that other optimizations such as method inlining and escape analysis can optimize the code produced to perform inlined field accesses.

**Object Serialization** is a widely used technique that converts object graphs into a compact serialized format (byte streams), reducing memory consumption. Serialization is, however, a complex and slow process as it requires runtime introspection (reflection), or developer-provided code-snippets on how to serialize objects [12, 15]. Compared to object serialization, object inlining is a compiler-transparent technique meaning that objects are not required to be copied out from their inlined format if the value does not escape the current scope. Using serialization, every object access requires deserialization. In addition, we also show that inlining can be combined with serialization to reduce the overhead imposed by the serialization process, and also to reduce the memory footprint of the serialized format (see §6.4).

Previous works [29, 30] have also studied techniques to process data in its native/serialized format. These techniques have been shown to work in the context of Big Data platforms such as Spark [41], Hadoop [4], and Flink [8]. However, unlike *value fields*, these techniques will not help improving the performance of caches and databases since objects stored in such data stores tend to always escape the scope of allocation.

## 8 Conclusions

This paper revisits the topic of object inlining and starts by addressing the challenges that prevent it from being a generally applicable technique in languages that contain identity exposing operations. We demonstrate through experiments with real-world platforms and workloads that there is a significant potential for performance improvements of throughput and memory footprint that could be attained by allowing such type layout optimizations. To unlock object inlining, we propose using the closed-world environment combined with *value fields*, an abstraction that allows the compiler to optimize data layouts by relaxing object identity and atomic field access guarantees. Applying *value fields* to the application data-path is shown to be possible with minimal user effort or even with no effort at all if incorporated directly into frameworks. Looking forward, we see this work as a tool that framework and library developers can use to optimize applications and to develop memory-efficient data structures and algorithms which can take advantage of object inlining with minimal to no user involvement.

## References

[1] 2015. *Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow*. Retrieved November 10, 2020 from https://eng.uber.com/service-oriented-architecture/

[2] 2015. *What Led Amazon to its Own Microservices Architecture*. Retrieved November 10, 2020 from https://thenewstack.io/led-amazon-microservices-architecture

[3] 2016. *Mastering Chaos - A Netflix Guide to Microservices*. Retrieved November 10, 2020 from https://www.infoq.com/presentations/netflix-chaos-microservices/

[4] 2019. *Apache Hadoop*. Retrieved November 19, 2020 from https://hadoop.apache.org/

[5] 2019. *Micronaut - A modern, JVM-based, full-stack framework for building modular, easily testable microservice and serverless applications*. Retrieved November 10, 2020 from https://micronaut.io/

[6] 2019. *Microservices at eBay, Part 2: Sharing Modules Across Experience Services and Multi-Screen Applications*. Retrieved November 10, 2020 from https://dzone.com/articles/microservices-at-ebay-part-2-sharing-modules-acros

[7] 2019. *Neo4j*. Retrieved November 10, 2020 from https://neo4j.com/

[8] 2020. *Apache Flink - Stateful Computations over Data Streams*. Retrieved November 19, 2020 from https://flink.apache.org/

[9] 2020. *Apache JMeter*. Retrieved November 10, 2020 from https://jmeter.apache.org/

[10] 2020. *H2 Database Engine*. Retrieved November 10, 2020 from https://www.h2database.com/html/main.html

[11] 2020. *IMDb Datasets*. Retrieved November 10, 2020 from https://www.imdb.com/interfaces/

[12] 2020. *Kryo*. Retrieved November 10, 2020 from https://github.com/EsotericSoftware/kryo

[13] 2020. *OpenJDK - Valhalla*. Retrieved November 10, 2020 from https://wiki.openjdk.java.net/display/valhalla/Main

[14] 2020. *OrientDB*. Retrieved November 10, 2020 from https://orientdb.org/

[15] 2020. *Protocol Buffers - Google's data interchange format*. Retrieved November 10, 2020 from https://github.com/protocolbuffers/protobuf

[16] 2020. *Records*. Retrieved November 10, 2020 from https://docs.oracle.com/en/java/javase/14/language/records.html

[17] 2020. *Spring Boot*. Retrieved November 10, 2020 from https://spring.io/projects/spring-boot

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[19] Julian Dolby. 1997. Automatic Inline Allocation of Objects. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. Association for Computing Machinery, New York, NY, USA, 7–17. https://doi.org/10.1145/258915.258918

[20] Julian Dolby and Andrew Chien. 2000. An Automatic Object Inlining Optimization and Its Evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 345–357. https://doi.org/10.1145/349299.349344

[21] Iulian Dragos and Martin Odersky. 2009. Compiling Generics through User-Directed Type Specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 42–47. https://doi.org/10.1145/1565824.1565830

[22] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages* (Indianapolis, Indiana, USA) *(VMIL '13)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/2542142.2542143

[23] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. 2018. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 94–106. https://doi.org/10.1145/3274808.3274816

[24] Dibakar Gope and Mikko H. Lipasti. 2016. Hash Map Inlining. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation* (Haifa, Israel) *(PACT '16)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/2967938.2967949

[25] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. 2008. Optimized Strings for the Java HotSpot&Trade; Virtual Machine. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java* (Modena, Italy) *(PPPJ '08)*. ACM, New York, NY, USA, 105–114. https://doi.org/10.1145/1411732.1411747

[26] John Hunt. 2014. *Value Classes*. Springer International Publishing, Cham, 147–150. https://doi.org/10.1007/978-3-319-06776-6_15

[27] Kyle Lo, Lucy Lu Wang, Mark Neumann, Rodney Kinney, and Daniel Weld. 2020. S2ORC: The Semantic Scholar Open Research Corpus. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 4969–4983. https://doi.org/10.18653/v1/2020.acl-main.447

[28] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.".

[29] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: Thin Computation over Big Native Data Using Speculative Program Transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 538–553. https://doi.org/10.1145/3341301.3359643

[30] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) *(ASPLOS '18)*. Association for Computing Machinery, New York, NY, USA, 56–69. https://doi.org/10.1145/3173162.3173200

[31] Tobias Pape, Carl Friedrich Bolz, and Robert Hirschfeld. 2015. Adaptive Just-in-time Value Class Optimization: Transparent Data Structure Inlining for Fast Execution. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. ACM, New York, NY, USA, 1970–1977. https://doi.org/10.1145/2695664.2695837

[32] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[33] Olivier Sallenave and Roland Ducournau. 2012. Lightweight Generics in Embedded Systems through Static Analysis. *SIGPLAN Not.* 47, 5 (June 2012), 11–20. https://doi.org/10.1145/2345141.2248421

[34] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Orlando, FL, USA) *(CGO '14)*. Association for Computing

Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2544137.2544157

[35] Vlad Ureche, Milos Stojanovic, Romain Beguet, Nicolas Stucki, and Martin Odersky. 2015. Improving the Interoperation between Generics Translations *(PPPJ '15)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/2807426.2807436

[36] Kris Venstermans, Lieven Eeckhout, and Koen De Bosschere. 2007. Object-Relative Addressing: Compressed Pointers in 64-Bit Java Virtual Machines. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–100.

[37] Christian Wimmer. 2008. *Automatic Object Inlining in a Java Virtual Machine.* Trauner.

[38] Christian Wimmer and Hanspeter Mössenböck. 2007. Automatic Feedback-directed Object Inlining in the Java Hotspot™Virtual Machine. In *Proceedings of the 3rd International Conference on Virtual Execution Environments* (San Diego, California, USA) *(VEE '07)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/1254810.1254813

[39] Christian Wimmer and Hanspeter Mössenböck. 2008. Automatic Array Inlining in Java Virtual Machines. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Boston, MA, USA) *(CGO '08)*. ACM, New York, NY, USA, 14–23. https://doi.org/10.1145/1356058.1356061

[40] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360610

[41] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing* (Boston, MA) *(HotCloud'10)*. USENIX Association, USA, 10.