HeapBuffers: Why Not Just Using a Binary Serialization Format for Your Managed Memory?

DANIELE BONETTA, Vrije Universiteit (VU) Amsterdam, The Netherlands
JÚNIOR LÖFF, Università della Svizzera italiana (USI), Switzerland
MATTEO BASSO, Università della Svizzera italiana (USI), Switzerland
WALTER BINDER, Università della Svizzera italiana (USI), Switzerland

In modern cloud applications, data serialization and deserialization (DSD) are common yet expensive operations. Languages such as Java incur significant overhead from DSD operations because of their managed memory, which uses an IO-incompatible, sparse, platform-dependent data layout for storing objects.

Can language VMs bypass costly DSD operations by directly operating on an *IO-ready* binary format? In this paper, we address this question by presenting a new DSD library for Java called HeapBuffers. HeapBuffers maintains a Java-friendly programming model through managed memory, yet it operates on data stored using a dense, platform-independent, memory format. In this way, HeapBuffers data can be used to perform IO operations without the need to convert objects from/to a VM-internal memory representation, thus eliminating the need for expensive DSD operations. Compiler optimizations combined with specialized garbage collection ensure that accessing HeapBuffers data is nearly as efficient as handling regular Java objects.

The answer to our question is largely positive: our experiments show that HeapBuffers data can be accessed at high performance and can be used *as is* to perform IO, making the cost of DSD effectively negligible. IO performance of applications using HeapBuffers is often orders of magnitude faster than that obtained using the prevailing DSD libraries.

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

Additional Key Words and Phrases: Data serialization and deserialization, Managed language runtimes, Garbage collection, FlatBuffers

ACM Reference Format:
Daniele Bonetta, Júnior Löff, Matteo Basso, and Walter Binder. 2025. HeapBuffers: Why Not Just Using a Binary Serialization Format for Your Managed Memory?. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 397 (October 2025), 25 pages. https://doi.org/10.1145/3763175

1 Introduction

Data Serialization and Deserialization (DSD) is a very common but expensive operation in modern cloud computing. In 2022 Google reported [Karandikar et al. 2021] that almost 10% of the total CPU time of its data centers is spent performing Protobuf [Google Inc. 2018] DSD operations, and all cloud providers experience similar overheads [Sriraman and Dhanotia 2020]. With projected growth in networking performance [IEEE Standards Association 2023; NVIDIA Corporation 2023] and an increasing dependence on cloud technologies, the impact of DSD on data centers is poised to increase even further.

Authors' Contact Information: Daniele Bonetta, Vrije Universiteit (VU) Amsterdam, Amsterdam, The Netherlands, d.bonetta@vu.nl; Júnior Löff, Università della Svizzera italiana (USI), Lugano, Switzerland, loeffj@usi.ch; Matteo Basso, Università della Svizzera italiana (USI), Lugano, Switzerland, matteo.basso@usi.ch; Walter Binder, Università della Svizzera italiana (USI), Lugano, Switzerland, walter.binder@usi.ch.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART397
https://doi.org/10.1145/3763175

DSD operations are often considered an unavoidable cost (sometimes called a *Tax* [Kanev et al. 2015]). This is particularly true in the context of *managed programming languages* such as Java, Python, or JavaScript. The in-memory representation of objects in language VMs, such as the Java Virtual Machine (JVM) or the JavaScript V8 engine, is designed to optimize *sparse* memory accesses and *large* collections of objects representing *all* application data. On the other hand, messages to be exchanged in cloud applications (for example, between serverless functions [Amazon Web Services 2015; Google Inc. 2017]) are often *small* and must be encoded using a *dense* representation to minimize data-transfer costs. Popular binary serialization formats such as Protobuf [Google Inc. 2018], FlatBuffers [Google Inc. 2016], and others [Apache Software Foundation 2023; Caucho Technology 2004; Esoteric Software 2025; Protostuff 2011], are explicitly designed to optimize message passing; their layout minimizes data size and prioritizes on-wire transfer performance.

The discrepancy between object layout in language VMs and wire-optimized binary formats is accentuated by the nature of the language VM's Garbage Collector (GC), which can freely relocate objects in memory. In most language VMs, the GC directly manages all application objects, without differentiating between objects solely for transfer and those whose lifespan depends on their usage in the application. This discrepancy is reflected in the actual performance of data access and serialization. Accessing data from an in-memory Java object is significantly faster than manipulating the data directly in a binary format such as FlatBuffers (see Figure 1). Due to this performance gap, applications often deserialize data by *copying* the objects from a binary format directly into the VM's heap memory upon message receipt.

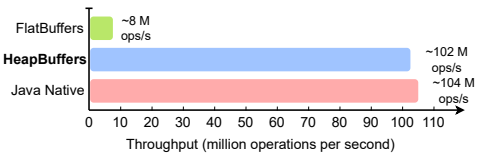


Fig. 1. Read performance of Java objects vs FlatBuffers data. HB have performance comparable to Java, while being stored in an IO-ready binary format.

In this paper, we propose to store objects used for data transfer in an IO-ready format *already in the VM memory space*. To this end, we introduce the HeapBuffers (HB) library and show how it can be supported natively in the state-of-the-art GraalVM Native Image [Oracle Corporation 2019] Java VM. In our approach, objects intended for IO operations are defined using a *serialization schema*, which serves as a template, similar to libraries like Protobuf or FlatBuffers. Unlike traditional serialization libraries, HB objects resemble regular Java objects within a managed heap, since they also benefit from automatic memory management. Furthermore, VM-level optimizations enable HB objects to be accessed with performance comparable to Java objects. Being in an IO-ready format, they completely avoid DSD operations. This paper makes the following contributions.

- We present the design of the HB DSD library and its API. Like regular Java objects, HB objects are garbage-collected and benefit from automatic memory management. Unlike regular Java objects, they are stored in a dedicated memory region using a dense, platform-independent binary format that can be used to perform IO operations directly. HB uses the same schema language of the popular FlatBuffers [Google Inc. 2016] DSD library, and is binary compatible with FlatBuffers messages. The HB design achieves full migration compatibility for applications already using FlatBuffers.
- We describe how language VMs can implement *native* support for HB objects. In particular, we demonstrate how HB can leverage code generation and compiler optimizations to enhance object-access performance. Thanks to our optimizations, applications can use HB objects with performance comparable to Java objects.

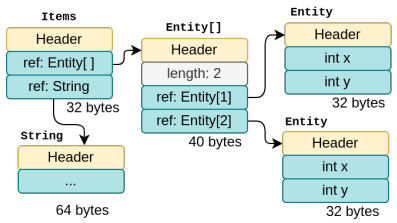
- We natively integrate HB into the GraalVM Native Image Java VM, and we evaluate HB on serialization benchmarks and on cloud microservices. Compared to the best existing DSD libraries, HB is orders of magnitude faster at performing DSD operations.

The remainder of this paper is structured as follows. In Section 2 we discuss popular binary IO serialization formats and their limitations in the context of managed programming languages, focusing on the popular FlatBuffers format. In Section 3 we introduce HB and provide a detailed description of the programming model and its main features. In Section 4 we discuss how the library is implemented and how it interacts with the language VM. Section 5 compares HB performance against existing alternatives. Section 6 discusses related work and Section 7 concludes.

2 Background: Data Serialization and Managed Language Runtimes

Language runtimes such as the JVM store objects in a managed heap memory space that is entirely controlled by the runtime. Automatic memory management via Garbage Collection (GC) ensures that memory is reclaimed when no longer needed, and the virtual machine has complete freedom as to where and how to organize data in memory. Although the general concept of a managed memory space has proven extremely successful (and is considered one of the great achievements of programming languages research [Jones et al. 2011]), existing general-purpose approaches to automatic memory management come with considerable downsides when it comes to message-based applications, such as those found in all modern cloud and web applications.

The first problem is that the memory organization of language virtual machines such as the JVM is optimized for access to *sparse* data, with all objects organized in memory forming a large object graph. This is depicted in Figure 2, where the typical in-memory representation of Java objects is presented. As the figure shows, objects are organized using an object graph, with references (e.g., compressed pointers [Oracle Corporation 2023a]) used to connect different instances. The complete freedom that the VM has to place objects in memory makes it impossible for objects in the heap to be directly used in IO operations, which mandate all data to be represented in a contiguous region (e.g., a buffer). A second issue with the way modern VMs organize heap memory is the opaque and platform-specific nature of the VM memory management, which makes the data non-portable to other processes: All references in Figure 2 are valid only in the specific virtual memory space of the VM process and cannot be transferred to another process or machine without translation. Finally, data objects stored in the JVM memory space are inherently implementation-dependent. For example, Java objects typically have an object header that consumes space and is used to store metadata that is not relevant for languages other than Java. Interoperability between different programming languages is increasingly important in cloud applications which tend to be developed using multiple loosely-coupled microservices.




```
class Items {
    final Entity[] entities;
    final String name;
}

class Entity {
    private int x;
    private int y;
}
```

Fig. 2. Typical memory organization for common Java objects.

2.1 Binary Serialization Formats and the JVM

The need to share data across various programming languages and the limitations of VM-specific binary formats have resulted in the emergence of numerous data serialization formats. Managed languages typically serialize data prior to IO operations and deserialize it after receiving a network message or reading from a file. Generally, serialization transforms VM-specific data into a platform-independent format. Both serialization and deserialization represent significant performance bottlenecks due to the need to perform multiple copies within the VM memory. With the growing centrality of cloud computing, data serialization formats that specifically target message-heavy applications, such as microservices, have emerged. Popular examples in this space are Protobuf [Google Inc. 2018], FlatBuffers [Google Inc. 2016], Cap'n Proto [Sandstorm Development Group 2012], and more. Regardless of the differences in the way they encode data, all such formats represent data using a *dense* format where all elements of an object graph are encoded in a contiguous memory space, with platform-independent references (typically an offset relative to the buffer head), and support for multiple programming languages. An example of the data in Figure 2 stored in the popular FlatBuffers storage format is depicted in Figure 3. As the figure shows, data are stored using a compact representation, starting from a *root object* that corresponds to the root of the object graph being serialized. The efficient representation of FlatBuffers allows applications to save considerable amounts of memory (≈ 80 bytes in the example figure, compared to more than 300 bytes for the equivalent Java in-memory representation). In summary, using modern binary serialization formats such as FlatBuffers makes data transfers more efficient and language-independent. Nevertheless, in managed programming languages, the benefits of FlatBuffers are accompanied by non-negligible DSD costs.

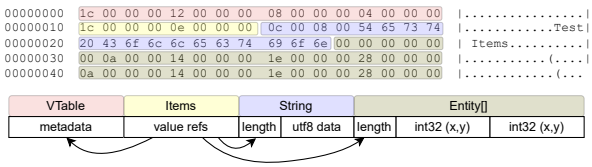


Fig. 3. FlatBuffers binary encoding for the Java objects in Figure 2.

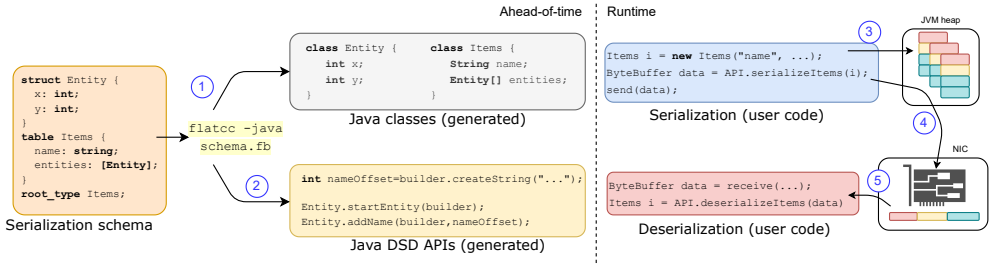


Fig. 4. Typical DSD workflow of a Java application using the FlatBuffers format.

2.2 The Issues with Binary Serialization in the JVM

Converting data from/to the Java memory format of Figure 2 to/from an IO-friendly data format such as FlatBuffers presents two main problems in the context of managed programming languages like Java, namely performance and API usability. Concerning performance, Java objects can be accessed with high efficiency by the JVM. In contrast, accessing data directly from a FlatBuffers

```java
// Initialize DSD library (estimated buffer size)
FlatBufferBuilder fb = new FlatBufferBuilder(1024);
// Create Entity structs
int e1 = Entity.createEntity(fb, 10, 20);
int e2 = Entity.createEntity(fb, 30, 40);
int[] entities = new int[]{e1, e2};
int entities = Items.createItemsVector(fb, entities);
int name = fb.createString("String name");
// Create the Items table (root object)
Items.startItems(fb);
Items.addName(fb, name);
Items.addEntities(fb, entities);
int items = Items.endItems(fb);
// Finalize and get serialized data
fb.finish(items);
byte[] data = fb.sizedByteArray();
```

Fig. 5. FlatBuffers serialization API example usage.

object is significantly slower due to the VM's inability to perform runtime optimizations on the raw data: from the perspective of the JVM's optimizing compiler, a FlatBuffers message is just an opaque piece of memory (a buffer); since compilation cannot optimize data access without knowledge of the FlatBuffers schema, the VM simply treats the data as *unmanaged* raw data (e.g., a `byte[]` array). Operating directly on the serialized FlatBuffers data is possible, but incurs high overhead as anticipated in Figure 1. To overcome the high cost of operating on serialized data, FlatBuffers data must be converted (deserialized) to Java heap objects. Obviously, serialization and deserialization are inherently costly processes. Moreover, unlike regular Java objects, binary serialized data cannot be altered directly. When a serialized FlatBuffers message is received, it is normally treated as immutable. To modify its content, applications must first convert (deserialize) the message to Java objects, make changes, and then serialize it again to create a new FlatBuffers message.

A second less intuitive limitation of binary data formats in the context of Java is their API and programming model. Figure 4 illustrates the standard workflow for using FlatBuffers in Java. First, being a platform- and language-independent data format, a *serialization schema* (1) must be defined to map concrete classes in the Java type system to abstract FlatBuffers types such as `table` or `struct`. Once defined, a code-generation tool (`flatcc`) is used to generate the Java classes corresponding to the FlatBuffers types specified in the schema (2). Analogously, the schema is also used to generate tailored *serialization APIs* that control how data can be converted from/to its binary format into equivalent Java types. Leveraging the generated Java classes and serialization APIs, user code can finally perform serialization and deserialization operations. Using the serialization APIs to convert Java objects to an equivalent FlatBuffers message is however a non-trivial operation. In particular, since FlatBuffers objects are not stored in managed memory, Java developers have to *manually control* the allocation of each object in the target binary serialization buffer. This is a form of manual memory management that is in open contradiction with the automatic memory management provided by Java's GC. As such, it is often the source of subtle bugs. Consider the code in Figure 5, which is based on the Java types and serialization schema of Figure 4. First, users must explicitly specify the size of the expected buffer (`1024` in the example). This is a first estimate, as the user does not know the exact size of the final serialized object graph, and must be over-sized to avoid running out of space. Then, all object allocations must be explicitly controlled using the `create()` and `finish()` methods, which are used to compute the object offsets. Moreover, the offset of each object must be explicitly tracked by the user and passed as arguments to allocate structured types (e.g., in the call to `createItemsVector()` which expects the offsets of each element of an array to be supplied explicitly to the generated constructor). Not only is the API cumbersome, it is also error-prone: objects must be created in a specific order which corresponds to a depth-first traversal

of the object graph specified in the schema (starting from the root type). Changing this order (e.g., by creating `e2` prior to `items` in the example) leads to runtime errors. This programming style has obvious drawbacks compared to the equivalent code that a Java developer has to write when allocating Java objects on the heap memory space managed by the Java GC.

3 HeapBuffers

HeapBuffers (HB) is a DSD library designed to provide a *best of both worlds* solution for managed programming languages, focused on Java. From a user's perspective, HB exposes a managed API built on a familiar Java programming model based on interfaces and lambda functions. In contrast to serialization APIs like FlatBuffers, working with HB data is akin to manipulating Java objects. When constructing an HB message, objects can be created and modified freely and in any order, avoiding the issues described in Section 2.2. Notably, HB leverages GC, relieving developers from manual memory management and data placement tasks. Concerning performance, Java applications can access HB data *directly*, entirely bypassing DSD operations. Consequently, HB data access is almost as fast as access to standard Java objects. Moreover, since data access does not involve DSD operations, HB data can be altered *in place*, similar to standard Java objects. This contrasts popular DSD libraries, as highlighted in Section 2.2.

HB data are stored in a dense, IO-ready format. In this way, they can be directly used to perform IO operations. The data for a message are stored in the same memory as the Java VM process running the application; however, they are not stored in the global garbage-collected JVM heap memory where all Java objects are allocated. Instead, they are placed in a separate dedicated heap memory space, called *Transferable Heap*. The HB runtime guarantees that data in transferable heaps are stored in an IO-ready format. This is maintained by HB GC, which excludes unreachable data from messages before IO operations and ensures that data are well-structured after GC.

Essentially, HB GC not only handles the disposal of unreachable references, but also prepares HB data for IO. Messages in transferable heaps are always IO-ready after a GC cycle, enabling Java applications to execute zero-copy IO operations, such as memory mapping a transferable heap to an IO device. The binary serialization format used by HB is compatible with the FlatBuffers [Google Inc. 2016] binary format: messages that are encoded using FlatBuffers (in any programming language) can be consumed using HB, thus improving interoperability with existing applications. Likewise, data produced using HB can be consumed by existing FlatBuffers libraries. Moreover, HeapBuffers adopts the FlatBuffers schema language. By adopting the schema language and binary format of FlatBuffers, the HB design achieves migration compatibility and facilitates adoption. In the rest of this section, we discuss how HB is used in Java applications and describe how the HB runtime, VM-level optimizations, and dedicated GC enable high-performance data access.

3.1 Programming Model and API

HB adopts the well-established workflow of the FlatBuffers serialization library introduced in Figure 4: users specify a serialization schema and use a tool to generate Java APIs to be used to produce and consume data. The FlatBuffers schema language enforces a tree-like structure, where a single root object exists, and all other objects follow a strictly hierarchical organization free of cycles. HB employs the same schema language as FlatBuffers [Google Inc. 2014], allowing users to adopt HB while maintaining existing FlatBuffers data exchanges between applications. Unlike FlatBuffers, HB does not generate Java classes. Instead, it generates a number of Java *sealed* interfaces [Oracle Corporation 2023b] corresponding to all types specified in the schema. We chose interfaces over classes because they provide a clear abstraction mechanism and allow us to enforce restrictions that prevent users from extending or modifying the generated code. Since HB does not

require method code generation, interfaces alone are sufficient to support the HB programming model. The generated interfaces are the only mechanism to interact with the HB data in Java.

Figure 6 presents an example of the interfaces generated for the schema of Figure 4. As the figure shows, the interfaces provide accessor methods to read and modify all values in a message, with a direct mapping to Java primitive types (e.g., `int`) and HB-specific types. In particular, HB strings are not exposed to users as Java built-in strings (i.e., `java.lang.String`) but instead use a specialized type (the `HBString` interface). The decision to expose strings with a dedicated type is motivated by two main reasons. First, using a dedicated HB interface type enables high-performance access to data stored within the HB heap. This approach allows string operations supported by the `HBString` type to be executed with low overhead, as it eliminates the need to copy the byte array to the Java heap. To increase compatibility, `HBString` implements Java's `CharSequence` interface and supports common functionalities of regular Java strings. Moreover, when Java strings are needed, `HBString` objects can be converted using `asString()`, which performs a memory copy of the underlying HB data and allocates a Java string in the JVM heap [1]. A second motivation is that FlatBuffers strings are encoded using `utf8`, while strings in Java are `utf16` [2]. Similarly to strings, array types are exposed to users with a dedicated interface (generated from the input schema) instead of using plain Java arrays. This is, for instance, the case of `EntityArray` in our example, which is a sealed interface that extends the `HBIterable` interface. Importantly, all objects stored in a HB transferable heap have *Value-type* semantics, as we will discuss later in Section 3.4.1.

```
sealed interface Entity
    extends HBObject {
  int getX(); void setX(int value);
  int getY(); void setY(int value);
}

sealed interface Items
    extends HBRoot {
  EntityArray getEntities();
  void setEntities(EntityArray value);
  HBString getName();
  void setName(HBString value);
}

sealed interface EntityArray
    extends HBArray, HBIterable<Entity> {
  int getSize();
  Entity get(int index);
  void set(int index, Entity e);
  // Iterable methods ...
}
```

Fig. 6. HB generated interfaces for the data serialization schema of Figure 5.

3.2 Serializing Data

Figure 7 shows an example of the use of HB to serialize data. The code is based on the schema introduced earlier in Figure 4 and on the corresponding interfaces in Figure 6. First, a new message must be created using `HBMessage.allocateHeap()`, which initializes the underlying HB runtime and allows the creation of HB objects with types defined in the schema. Each `HBMessage` is specifically bound to an HB object designated as the root type in the schema. Once the message is created, it can be accessed and modified using the `open()` method, which provides access to the message's transferable heap. The `open()` method accepts a lambda function that grants access to the HB runtime (denoted as `hb` in the code examples). Through this runtime, HB objects can be allocated and modified as necessary. Users can call `open()` to gain access to the transferable heap as many times as needed, but operations on HB objects outside of the `open()` call are not allowed and have unspecified behavior. Once a transferable heap is accessible, objects can be allocated in it using the HB runtime. This can be achieved by means of *factory methods* that allocate opaque objects implementing the interfaces specified in the serialization schema. For example, `hb.newEntity()` can be used to create a new HB object that implements the `Entity` type of Figure 4. Similarly to interfaces, HB generates code for all factory methods based on a specified serialization schema.

[1] The copy can be performed efficiently when the string is `latin1`.

[2] In practice, most of the strings found in regular Java applications are stored internally using `latin1` [Oracle Corporation 2014]. When this is the case, `latin1` strings are stored identically in Java and in the HB string's `utf8` encoding.

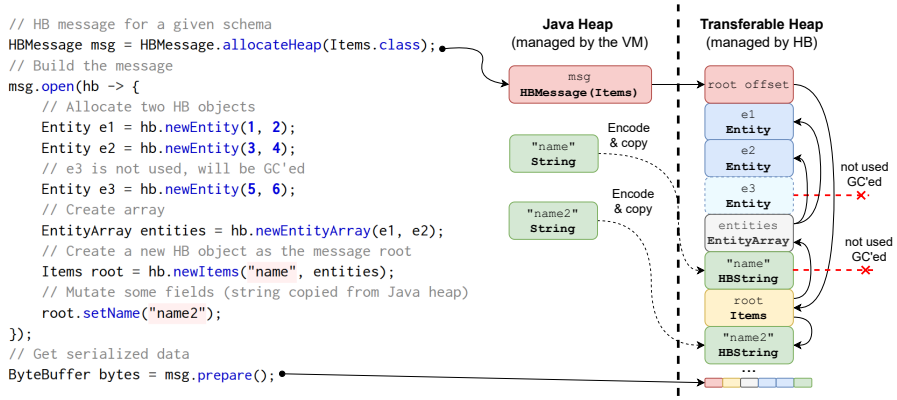


Fig. 7. Example of object allocation and serialization using HB.

After creation, HB objects can be accessed using their getter and setter methods. Once again, calls to such getters and setters are allowed only from within the boundaries of the `open()` call. Allocations in `open()` occur on the HB transferable heap, which the HB runtime manages with its own GC. This is shown in Figure 7, where an `Entity` object (`e3`) is instantiated without being assigned elsewhere, resulting in its exclusion from the transferable heap after GC. Finally, when the contents of an HB message are ready for transfer, the `prepare()` method can be invoked. This triggers the GC to prepare the transferable heap, ensuring that the data are IO-ready, and returns a *direct* `ByteBuffer` [Oracle Corporation 2024a] for use in IO operations. As the content of a transferable heap is already serialized, preparing an HB message for IO is a zero-copy operation, because the underlying data are already stored in native memory.

Using `open()`, the HB library implements a form of dynamic scoping: HB objects can be accessed only from within the provided lambda function, and other Java code cannot instantiate or manipulate HB objects from another scope. This is an intentional API decision with the goal of making it explicit to users that HB interfaces belong to a separated memory space. The `open()` API takes inspiration from future Java API proposals and employs dynamic scoping similar to JEP 446 Scoped Values [Oracle Corporation 2023c]. Like Scoped Values, HB objects are confined to a specific scope (inside the lambda function). While the Scoped Values API focuses on concurrency, HB aims at data serialization. Moreover, using `open()` also disciplines concurrent access: transferable heaps cannot be accessed concurrently, and calls to `open()` can be considered a form of lock acquisition, where only one thread at a time is allowed to operate on a transferable heap with exclusive ownership.

3.3 Deserializing Data and In-Place Mutations

Another advantage of having garbage-collected managed memory for HB transferable heaps is that it enables efficient *object mutability*. This is exemplified in Figure 8 which shows how IO data can be modified in place. As the figure shows, when a message is received (e.g., via a socket), its raw data can be *mounted* into an HB transferable heap. After that, the HB heap can be accessed using the `open()` method, and the received data can be used by the Java application. Since HB objects are managed, they can be freely modified. For example, the value of a field of primitive type can be changed using `setX()`. Changing a field value in this way is equivalent to setting a new value in a regular Java object, and does not require any special handling or DSD operations. Since the message is already stored in the HB transferable heap, its content could be used again for

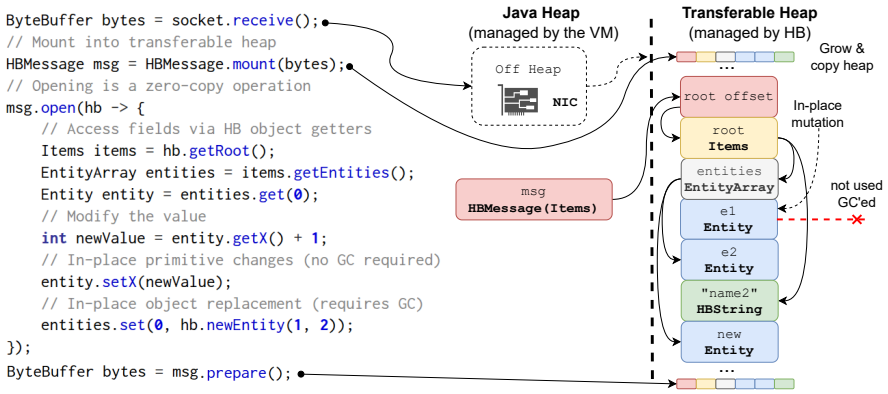


Fig. 8. Example of data deserialization and in-place mutation.

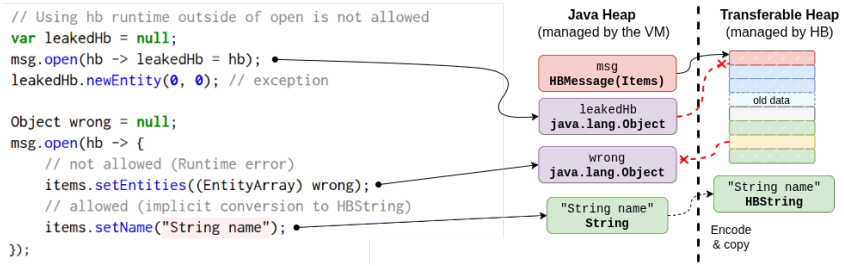


Fig. 9. Main limitations of the HB API and basic usage of `HBRef` reference types.

another IO operation right after a field modification. The same applies to HB fields of non-primitive type. For example, user code is free to modify object references with new objects (as done with `entities.set()` in Figure 8). The transferable heap will grow accordingly, and new objects will be placed in the new transferable heap. Old objects (i.e., no longer reachable from the root) will be garbage collected when `prepare()` is called, excluding them from future IO operations. This is in contrast with other serialization libraries such as Protobuf or FlatBuffers, where full deserialization is normally needed before modifying object fields, and where in-place modifications may require manual handling of object placements and memory operations, as discussed in Section 2.2.

3.4 Heap Separation and its Semantics

Conceptually, most serialization libraries, such as FlatBuffers, store their data in a memory space that is separate from the JVM heap memory. However, the allocated data are often a buffer of unmanaged raw memory (e.g., a `ByteBuffer` mapped to an off-heap memory region allocated via `malloc` or another allocator). Unmanaged memory results in the problematic and error-prone APIs discussed in Section 2.2. In contrast to unmanaged memory, HB stores its binary data in *managed* memory regions (transferable heaps), which are isolated and independent of the global JVM heap where all regular Java objects are allocated. However, being managed memory, HB objects can be manipulated conveniently and with more freedom by HB users. By adopting this design, the HB runtime is free to organize the content of an HB message in a contiguous IO-ready layout. Each transferable heap is exclusive to one instance of a `HBMessage`. Thus, it contains only one root

object (with the type specified in the schema via `root_type`). Transferable heaps do not support concurrent access. Multiple instances of `HBMessage` are possible, for example, to support different schemas or to perform multiple concurrent serializations for the same schema (e.g., using one thread-local instance of `HBMessage` per thread). Consequently, multiple independent transferable heaps may be active in a JVM simultaneously. These heaps are intended for IO and message passing, making their sizes much smaller than the larger general-purpose Java heap. At runtime, the JVM process defines the HB transferable heap size, typically on the order of a few KB or MB, as each heap carries just one message at a time.

As discussed in the previous section, dynamic scoping and the `open()` API enforce that operations on HB data occur only after a transferable heap has been opened for access. It is an error to allocate HB data outside of the `open()` scope, as it would mean allocating an object in the Java heap. Separation between the Java heap and the HB transferable heaps is also enforced by the type system: Java applications cannot create classes that extend the sealed HB interfaces, and any attempt to cast arbitrary objects to them will fail. Likewise, HB enforces (at compilation time) that its sealed interfaces cannot be stored in plain Java `Object` references, as we discuss in more detail in Section 3.4.1. This is depicted in Figure 9 which also shows other limitations of the API. In general, trying to store a HB object in the main Java heap will raise an error (e.g., storing HB objects in fields of Java classes). Likewise, Java objects cannot be stored in the HB heap. The only exception is with `String` objects, which are automatically converted to the `HBString` type when passed as arguments to HB methods (e.g., `setName()` in the figure).

3.4.1 Value-Type Semantics and Equality. From a Java semantics point of view, HB objects have the same semantics as the upcoming Java Value Types proposed in JEP 401 *Project Valhalla* [Oracle Corporation 2024b]. They do not extend the `Object` class, and an attempt to cast them to `Object` will result in an error. Like the upcoming Java value types, they do not have identity, only value equality. Moreover, they do not support Java's built-in synchronization (i.e., they cannot be used in `synchronized` blocks) and do not provide default methods other than what is specified in their serialization schema. Finally, like the upcoming Java value types, they are non-nullable and cannot contain null values. Compared to JEP 401 value types, HB objects have two significant differences. First, as discussed, their in-memory layout is IO-ready, platform independent, and compatible with the FlatBuffers binary format. Second, HB objects are not immutable by design and rather support mutability as a key feature that enables their efficient usage in IO applications. Figure 10 provides an overview of the type separation of HB objects compared to regular Java types, with the corresponding heap allocation semantics. As depicted in the figure, value types reside in the transferable heap and are only accessible by Java objects through two dedicated mechanisms, which we will discuss next.

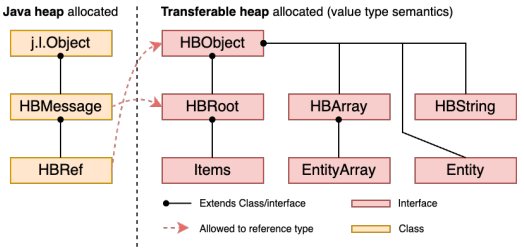


Fig. 10. HB type hierarchy and heap allocation semantics.

3.4.2 Java References to HB Data. The clear separation between the HB transferable heap and the Java heap ensures that all data within a transferable heap are IO-ready. Although it is not possible to store references to Java objects in a HB transferable heap, HB offers two mechanisms for referencing HB objects from the Java heap. First, all instances of `HBMessage` reside in the Java heap as standard Java objects. These objects hold a reference to the HB root object, allowing users to access its contents via `open()`. When a reference to an HB object other than the root object is

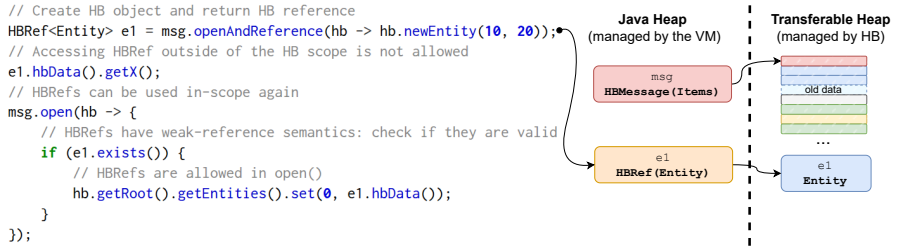


Fig. 11. Basic usage of HBRef types.

required, HB provides a dedicated API for direct object access, namely HBRef. An example of such API usage is shown in Figure 11.

As the figure shows, references to HB objects can be stored in the Java heap using dedicated HBRef<> objects. HBRef objects serve as clearable proxies for transferable heap references and can be used to (temporarily) store references to HB objects in the Java heap (e.g., within a Java Collection). HBRef objects have weak-reference semantics, meaning they do not prevent the HB GC from reclaiming referenced HB objects in the transferable heap. If an HBRef is cleared by the HB GC, any subsequent access to it (via hbData()) will result in an exception. These references remain valid as long as the referenced HB object is part of an HB object graph, and their content can be freely accessed from within the scope of open(). However, they can be invalidated in the following situations: (1) when the referenced HB object is no longer part of a message; (2) when the HBMessage containing the referenced HB object has been reclaimed by the Java GC. In general, HBRef objects are not intended as the primary mechanism for storing HB references in the Java heap. Instead, they are provided as a utility mechanism for (uncommon) scenarios where a reference to a HB object must be shared with Java code. In the common case, users should maintain a Java reference to the HB message's root object via HBMessage, which serves as the primary mechanism to interact with the transferable heap.

3.4.3 Garbage-Collection Semantics. Objects in the transferable heap are garbage collected by the HB runtime. Unlike the default Java heap GC, the transferable heap GC used by the HB runtime serves two purposes: (1) it reclaims objects that are no longer used, and (2) it takes care of *arranging* objects in a way that they can be directly used in IO operations. Specifically, the HB GC ensures that data placement follows the IO-friendly HB binary format, making the transferable heap immediately ready for transfer after each GC cycle. In standard Java applications, GC can occur at any time. In contrast, the HB runtime provides additional guarantees regarding GC timing. In particular, it is guaranteed that GC will happen when prepare() is called on a message. In this case, GC is synchronously executed by the calling thread. This is different from common (asynchronous, concurrent) garbage collectors, and guarantees that the content of a transferable heap never contains unreachable objects when it is about to be used in an IO operation. Moreover, performing GC immediately before IO operations ensures that data in the transferable heap remains compatible with the FlatBuffers format, as discussed next.

4 Implementation in GraalVM Native Image

In modern language runtimes such as the JVM, fast access to heap-allocated objects is achieved thanks to key optimizations implemented by the language VM's optimizing compiler. Achieving similar performance on HB objects requires the language VM to be able to perform common

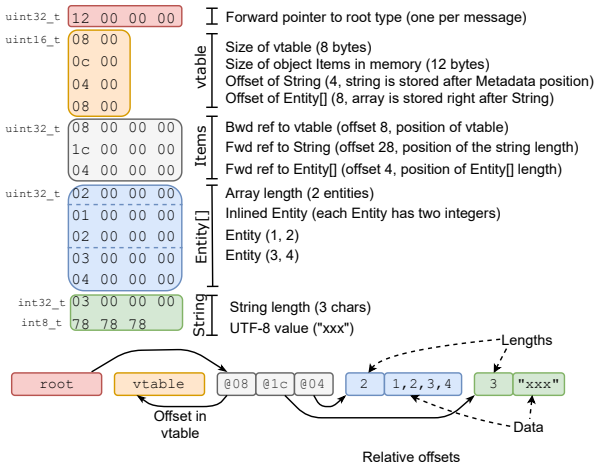


Fig. 12. Example of HB data encoding using the FlatBuffers binary format (based on the schema in Figure 4).

optimizations on HB objects as well. Similarly, garbage collection for HB objects needs to be performed with high efficiency, to prevent it from becoming a source of performance degradation. In this section, we discuss how compiler optimizations and automatic memory management are implemented in the context of HB objects. To demonstrate the feasibility of our proposed model, we have implemented the HB runtime and compiler optimizations in GraalVM Native Image [Oracle Corporation 2019], a state-of-the-art Java runtime that has gained significant adoption over the past decade. Given its very fast startup, Native Image is the de-facto choice for serverless and cloud-based Java applications, making it a very good fit for the message-based cloud applications targeted by HB. The changes in GraalVM Native Image to support HB objects are well confined (less than 1000 LOC) and are available open-source within our artifact [Bonetta et al. 2025]. Although our prototype targets GraalVM Native Image, our approach for operating directly on an IO-ready binary format and optimizing binary data access is not tied to Native Image and could also be ported to the Graal JIT and possibly to other VMs.

4.1 Binary Format and Transferable-Heap Organization

The HB binary format is compatible with FlatBuffers and has the following characteristics:

- Values are scalars or references. Scalars have various sizes and are all stored aligned (i.e., using padding to align them to fit 2, 4, or 8 bytes). Each scalar is always represented in little-endian format, which is most common on modern CPUs.[3]
- Floating-point numbers use the `IEEE-754` format.
- Signed integers use a *two-complement* representation.
- Endianness is the same for floating-point and integers.

Besides these analogies, the HB in-memory layout differs from that of FlatBuffers in one aspect: while references in FlatBuffers are always unsigned positive integers (i.e., relative offsets always forward pointing), in a HB transferable heap, references are signed and can contain negative values (i.e., they can be offsets pointing backwards). This subtle difference allows HB to store objects

[3]Like for FlatBuffers, HB could also work on big-endian machines, with some overhead to perform byte conversions.

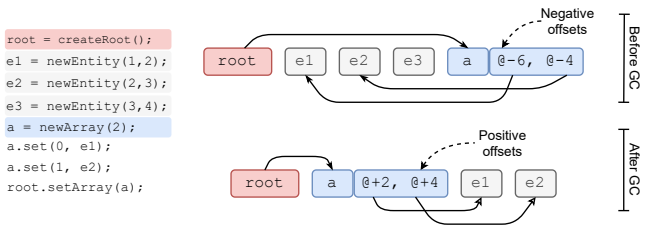


Fig. 13. Effect of GC on offsets for an example sequence of HB allocations. While negative offsets are allowed in the HB memory, all offsets are converted to positive integer values after GC (i.e., forward looking offsets).

in memory in the order they are allocated by user code, without having to (potentially) have to re-organize the content of the managed transferable heap after each object allocation. As a consequence, the total addressable memory with HB is smaller than what can be addressed with FlatBuffers. As offsets are stored in 32-bit unsigned integers, the maximum offset addressable with HB is $\approx 2GB$ ($2^31 - 1$), in contrast to the $\approx 4GB$ maximum addressable space of regular FlatBuffers. In practice, we consider the maximum size of a HB transferable heap to be reasonable at 2GB, given that messages exchanged in cloud applications are often of the order of only a few MB [Dean and Barroso 2013]. An example of data stored using the HB format is provided in Figure 12. The data in the image correspond to a serialized message created using the schema in Figure 2. While HB can have negative offsets in its transferable heap, it is important to note that after GC, all offsets in the heap will be transformed to positive integers, therefore ensuring that the content of a transferable heap is a valid FlatBuffers binary message after GC (see Figure 13).

4.2 Garbage Collection and Code Generation

Managed memory with automatic GC is a key feature to enable a familiar Java programming model and is a key enabling factor to support in-place mutation of messages.

Transferable heaps possess two main features that enable an efficient *specialized* GC implementation. First, each heap has only one valid root object, since it contains only a single message at a time. Second, the types and the type hierarchies in the heap are dictated by the serialization schema and are known ahead of time. In the object hierarchy established by the serialization schema, no cycles are possible. Such type information can be used to determine the *exact* order in which the garbage collector needs to traverse an object graph. As the actual structure of the object graph is known, it is possible to implement GC as a tree traversal of the HB object graph starting from its root. Moreover, during GC, objects can be re-organized in memory to ensure that all references will be positive integers, as discussed in the previous section.

GC can then be implemented as a traditional semispace copying GC [Jones et al. 2011], where memory is divided into two spaces (*from* and *to* spaces), and the GC is performed with a compacting copy of objects from one space to the other. Semi-space copying GC is a natural solution in the context of HB, and has the advantage of ensuring that data in a semi-space are always dense after GC. Moreover, as discussed, the garbage collector ensures that after each GC cycle, only positive offsets are used, making the content of the transferable heap a valid FlatBuffer message. GC can benefit from another key property of the HB model. Since the types are already known ahead of time, *the entire garbage collector can be generated ahead of time*. The core idea is that the typing information in the serialization schema can identify a deterministic exploration path in any HB object graph. Since each transferable heap has only one root object, code for traversing an entire

```
@HbType
public static sealed interface Entity extends HBStorage {
    short X_OFFSET = 0;
    short Y_OFFSET = 4;
    short STRUCT_SIZE = 8;

    @HbFieldAccess(kind=GET, offset=X_OFFSET, type=i32)
    int getX();
    @HbFieldAccess(kind=SET, offset=X_OFFSET, type=i32)
    void setX(int value);

    @HbFieldAccess(kind=GET, offset=Y_OFFSET, type=i32)
    int getY();
    @HbFieldAccess(kind=SET, offset=Y_OFFSET, type=i32)
    void setY(int value);
}
```

Fig. 14. Example of annotations placed by HB on the generated interfaces for a given serialization type. They are used by the Graal compiler to optimize memory accesses.

HB message graph can therefore be generated before execution. Pre-generating GC code allows it to be *specialized* for the specific types in a serialization schema. For example, the GC can *hard-code* field-reference offsets and perform fixed-size lookups and/or copies. Without static code generation, a conventional GC must dynamically track all class metadata, such as reference field positions in objects. Generating the GC ahead of time by leveraging the HB message schema also fits perfectly in the context of GraalVM Native Image, where key optimizations are executed at build time.

4.3 Compiler Optimizations and Code Generation

In addition to GC, ahead-of-time code generation using the types defined in the serialization schema significantly contributes to HB data-access performance through compiler optimizations. HB leverages the static information contained within the serialization schema to generate appropriate Java interfaces according to the types specified in the schema, and to pre-compute tailored metadata. The Graal compiler can use the static metadata to perform compiler optimizations, allowing it to apply common optimizations to read and write operations on HB objects in a way similar to the ones applied to Java standard objects.

An example of such interfaces annotated with relevant metadata is depicted in Figure 14. As the figure shows, the interface for the simple `Entity` HB type previously introduced in Figure 6 contains special annotations that include specific metadata about the in-memory layout of any HB object of that type. Such metadata is used by Graal to perform compiler optimizations. First, all interfaces annotated with `@HbType` inform the Graal compiler that a given type should be treated as a HB object. Among other things, this implies that the compiler will know that the type is not allocated in the JVM heap, but in a managed transferable heap. Secondly, the getters and setters are annotated with a `HbFieldAccess` annotation. This annotation signals to the compiler that calls to those interfaces in user- code should not be treated like regular method calls (potentially leading to a virtual method dispatch), but rather should be replaced with direct read/write operations, to be performed on the HB object instance corresponding to the original function calls. Note that Graal can very efficiently perform aggressive optimizations on such *special* interfaces, as the compiler knows that they are generated from the user-provided serialization schema.

Although the `Element` struct type in Figure 14 is a simple type that contains only primitive values, the strategy for optimizing complex types with references or arrays does not change: each generated interface will contain a number of annotations that the Graal compiler will interpret at compilation time to replace interface method calls with optimized direct memory accesses.

Table 1. Characterization of the schemas utilized in the experimental evaluation.

Schema	Fields	Objects	Primitives	Strings	Enums	Arrays	Nesting Depth	Total Elements	Raw Data (bytes)
gRPC [Google Inc. 2025]	8	4	1	3	-	1	3	5	87
Matrix	2	1	1	-	-	2	2	2	4
Media [Esoteric Software 2025]	19	3	8	7	1	3	3	5	271
Monster [Google Inc. 2025]	15	5	7	2	1	4	3	5	76
Sample [Esoteric Software 2025]	23	1	21	1	-	8	2	2	410
Schedule [Viotti and Kinderkhedia 2022]	10	3	3	3	1	5	4	5	143

Importantly, the compiler will handle reads and writes to HB fields similarly to regular Java field accesses. This implies that the compiler will be able to perform most of the optimizations it can perform on Java fields on HB objects as well. For example, the JVM optimizing compiler may perform inlining of methods accessing a HB object, it may *float* reads and writes to such fields, etc. In this way, HB code generation effectively provides compilation hints to the JVM compiler, which will then use them when generating machine code.

5 Performance Evaluation

In this section, we evaluate HB performance. To conduct the experiments, we designed a set of benchmarks and applications aimed at evaluating the performance of HB compared to state-of-the-art serialization libraries. The evaluation uses representative schemas extracted from various relevant domains as workloads. Our implementation of HB comprises sources for code generation, the runtime system, and compiler optimizations within GraalVM Native Image.

5.1 Experimental Setup and Datasets

All experiments are executed on a server-class machine equipped with a 16-core Intel(R) Xeon(R) Gold 6326 CPU (2.90GHz), 256GB of DRAM (3200MHz), and 1TB NVMe. Hyper-threading and turbo boost are disabled. The operating system is Ubuntu 22.04 LTS, Linux kernel 5.15.0-25-generic. The HB library uses the open-source GraalVM Native Image Java VM [Oracle Corporation 2019] version 21.0.2, based on OpenJDK 64-Bit Server VM 21.0.2-dev+13.1 (build 21.0.2+13-jvmci-23.1-b33).

All of our experiments involve DSD operations using a serialization *schema*. For the read-write performance evaluation (Section 5.2), we selected nine schemas that vary in nesting depth, number of fields, and field types (primitives and objects). The remaining experiments use the schemas presented in Table 1: *Media* and *Sample* are from Kryo's benchmarking suite [Esoteric Software 2025]; *gRPC* and *Monster* are from Google's FlatBuffers repository [Google Inc. 2025]; *Schedule* originates from prior research [Viotti and Kinderkhedia 2022]; and *Matrix* is a widely used data structure in many computational tasks.

The schemas exhibit diverse characteristics in nesting depth, number of fields, and data types. Some primarily consist of primitive types, while others have a higher number of variable-length data structures, such as strings and arrays. Table 1 also reports the raw data size for *scale* size 1, which corresponds to the uncompressed workload size without the additional metadata required for data access. For instance, storing three contiguous raw strings alone would be insufficient without metadata specifying their respective start and end positions, or special bytes indicating the string end. Each DSD library employs its own binary format, and we provide a more detailed discussion later in the paper by analyzing the message sizes. We will also explain the strategy used to scale the messages. The source code used in our experimental evaluation is included in the accompanying artifact [Bonetta et al. 2025].

5.2 In-Memory Data Access

To be competitive, read and write performance for HB objects should closely follow that of native Java objects, particularly for reads. If performance is significantly lower, it would be better to copy HB objects to Java objects, incurring a one-time data-copy cost but enabling faster access. In a first experiment, we focus on evaluating the access performance of HB objects compared against Java objects. To this end, we have designed a set of micro-benchmarks based on the JMH [Oracle Corporation 2013] experiments used to assess field-access performance of Java objects in OpenJDK. In all experiments, we evaluate the performance of *accessing* an object after the object has been allocated. For Java and HB, this involves field read-write operations. In binary serialization formats like FlatBuffers, it pertains to raw byte-buffer data access, which is typically slow and impractical. The performance difference was already highlighted in Figure 1, showing FlatBuffers read performance can be over 10× slower than native Java objects. In fact, the read performance shown in Figure 1 is the arithmetic mean across all schemas in Figure 15, which we discuss next.

5.2.1 Read Performance. Figure 15 presents the throughput when reading HB objects compared to Java objects. As shown, HB achieves read performance comparable to native Java objects. In three schemas, HB outperforms the JVM baseline. One reason is its dense data representation, which improves cache locality (e.g., `struct` types). For instance, in *bench05*—a simple array of three objects, each storing an integer and a string—HB achieves nearly 2× the performance by storing all objects contiguously, whereas the JVM does not guarantee where objects will be placed in memory. Finally, the JVM baseline performs better in the remaining schemas: the main reason is that in some cases, the JVM applies different inlining decisions than HB.

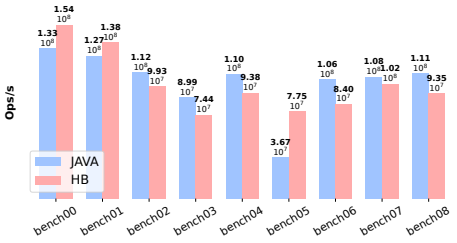


Fig. 15. Read performance.

5.2.2 Write Performance. Figure 16 shows the write performance for the same schemas. As depicted, HB generally achieves lower performance than Java for write operations. The main reason is that the JVM can often eliminate allocations through optimizations such as those enabled by Partial Escape Analysis [Stadler et al. 2014] (e.g., stack allocation and scalar replacements), while HB must always allocate data in the transferable heap. The JVM baseline achieves almost peak write performance in all schemas except for *bench05*, which is the largest in bytes. The arithmetic mean of the Java-object write throughput is around 45M ops/s, whereas HB objects reach on average 20M ops/s. For comparison, FlatBuffers is 26× slower than the JVM baseline, while HB is only 2.25× slower. Additionally, HB performs better on schemas that do not use strings. The reason is that strings residing in the Java heap are copied to the HB transferable heap as discussed in Section 3.1, incurring extra overhead.

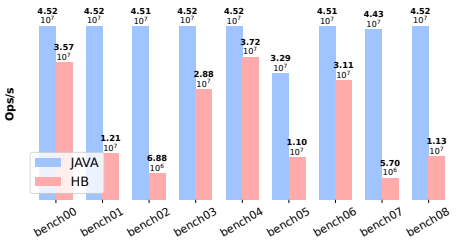


Fig. 16. Write performance.

5.2.3 GC Performance. To evaluate the impact of GC, we re-executed the write-intensive microbenchmarks, enforcing a full GC cycle within the HB runtime whenever a new message is created. The results are presented in Figure 17 and show that the GC overhead varies significantly between different schemas. In the HB model, a single GC cycle is required to arrange and prepare the transferable heap for transmission. The impact is generally higher in workloads that reach higher-throughput, where the HB semispace copying GC further increases the memory write load. Additionally, performance degradation is influenced by the number of objects that must be copied. In contrast, dense HB objects can be copied in larger contiguous chunks, reducing this cost.

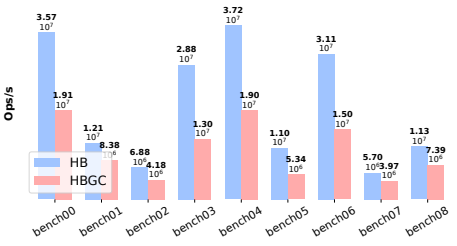


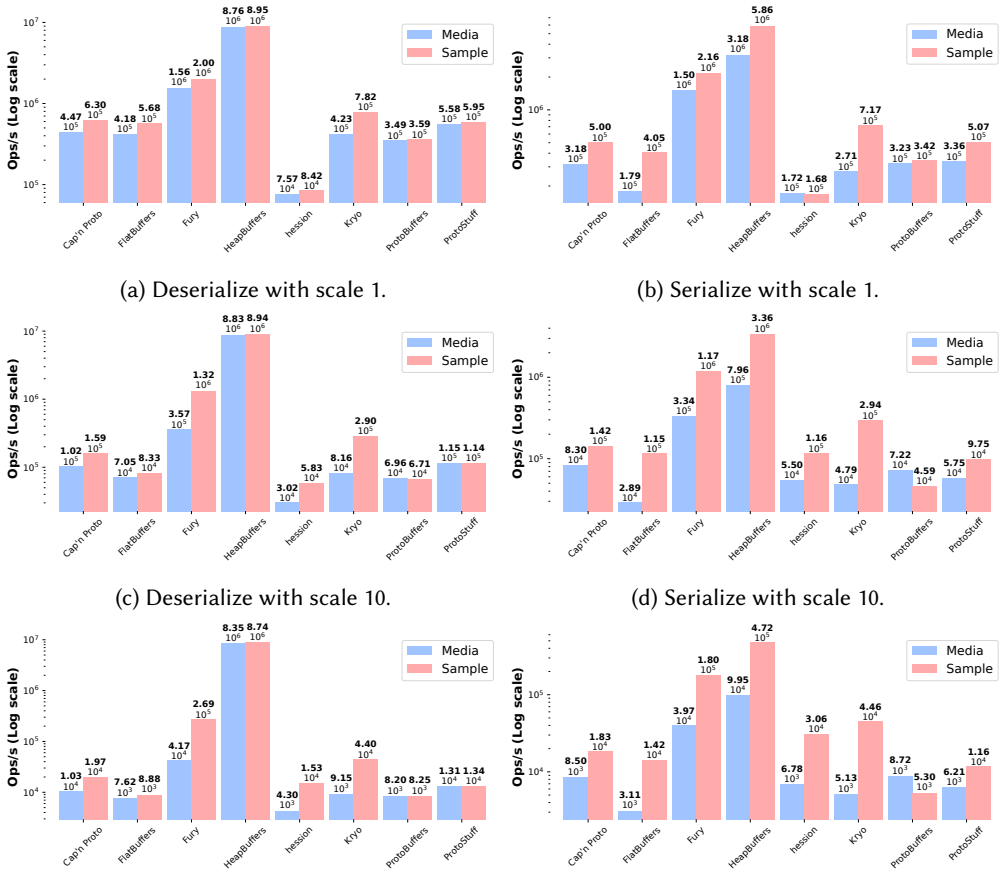
Fig. 17. Write and GC performance.

5.3 Serialization and Deserialization Performance

The main goal of HB is to make DSD operations more efficient. In this section, we present performance experiments comparing HB against state-of-the-art DSD solutions: FlatBuffers [Google Inc. 2016], Apache Fury [Apache Software Foundation 2023], Kryo [Esoteric Software 2025], Hessian [Caucho Technology 2004], Protobuf [Google Inc. 2018], Protostuff [Protostuff 2011], and Cap'n Proto [Sandstorm Development Group 2012]. We adapt an existing benchmark [Apache Software Foundation 2023; Esoteric Software 2025] and implement additional DSD libraries to ensure a comprehensive evaluation. Moreover, we modified the benchmarks to support Native-Image technology [Oracle Corporation 2019]. The design of this benchmark, along with subsequent evaluations, is an additional contribution of our work, and we make the source code available.

5.3.1 Deserialization. Figure 18 (left-hand side) presents the deserialization performance for scales 1, 10, and 100. Note that the results are presented using a logarithmic scale. The input consists of binary data, and the benchmark measures the time required for the libraries to transform these binary data into a safe, readable object. Figure 18 reveals that HB outperforms all other libraries. As can be seen, HB maintains consistent performance across all scales. The explanation is that the deserialization process is decoupled from the message size, which is not the case for the other libraries. Instead of reconstructing objects, HB provides zero-copy deserialization, as the received raw data can be mapped into its runtime transferable heap. Subsequently, the data can be used regularly via the provided getters and setters, as if they were allocated locally. The measured overhead arises from loading the necessary memory pages that contain the actual data. Among the other libraries, Fury exhibits the best deserialization performance and is the only library with competitive performance against HB. For example, it reaches the same magnitude of deserialization performance as HB in Figures 18a and 18c. However, it remains at least four times slower overall.

5.3.2 Serialization. Figures 18 (right-hand side) present serialization performance results. HB significantly outperforms all DSD libraries. The closest library to HB in terms of serialization efficiency is Fury, which shows a slowdown of approximately 2×. In this experiment, HB triggers a GC cycle during each serialization call to ensure that the data is dense and ready for transmission. Despite this, HB maintains high performance and outperforms the other libraries. Regarding schema differences, *Media* scales with arrays of objects, while *Sample* scales with primitive arrays, explaining why Sample achieves higher throughput than Media. Overall, the results confirm that the HB implements efficient DSD operations.



(a) Deserialize with scale 1.

(b) Serialize with scale 1.

(c) Deserialize with scale 10.

(d) Serialize with scale 10.

(e) Deserialize with scale 100.

(f) Serialize with scale 100.

Fig. 18. Performance comparison of DSD operations between HB and many popular libraries.

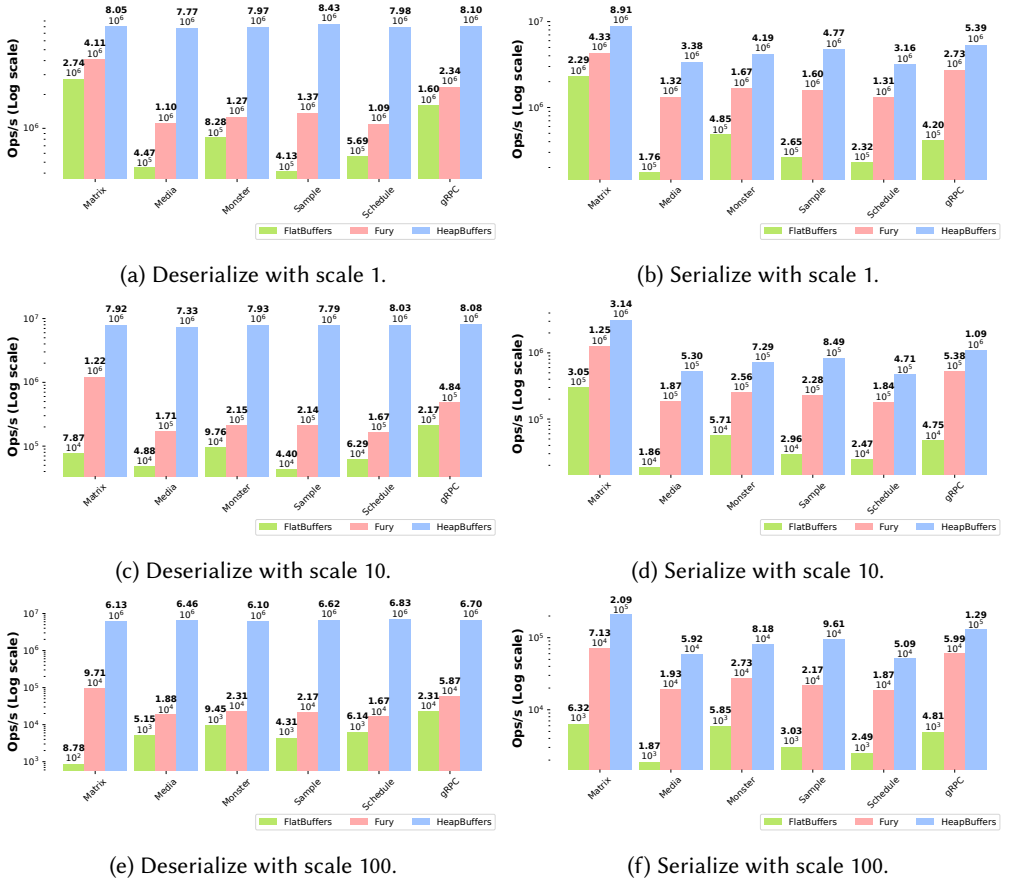
5.4 Serialization and Deserialization Performance on Different Schemas

Here, we extend the experiments from the previous section by introducing additional schemas and a refined methodology. We have developed a new benchmarking suite to compare the serialization and deserialization performance of HB, FlatBuffers, and Apache Fury. FlatBuffers was included because HB adopts the same binary representation and is compatible, making HB a potential in-place replacement with a more efficient implementation. Fury was selected as a baseline because of its good performance in the previous experiment.

Table 2. Message size (bytes) for different scales across different protocols.

Protocol	gRPC			Matrix			Media			Monster			Sample			Schedule		
	1	10	100	1	10	100	1	10	100	1	10	100	1	10	100	1	10	100
FlatBuffers	188	1520	14840	44	584	41624	504	4392	42912	224	1736	16856	584	5160	50880	384	3368	33248
Fury	112	1048	10409	17	458	40609	301	2938	29309	139	1318	13109	445	4378	43709	198	1908	19009
HB	195	1482	14352	48	588	41628	469	4060	39970	234	1872	18252	552	4890	48270	347	3002	29552

Previously, we presented and discussed Table 1, which characterizes the schemas. From this point onward, all experiments will use these schemas with the specified amount of raw data. In



(a) Deserialize with scale 1.

(b) Serialize with scale 1.

(c) Deserialize with scale 10.

(d) Serialize with scale 10.

(e) Deserialize with scale 100.

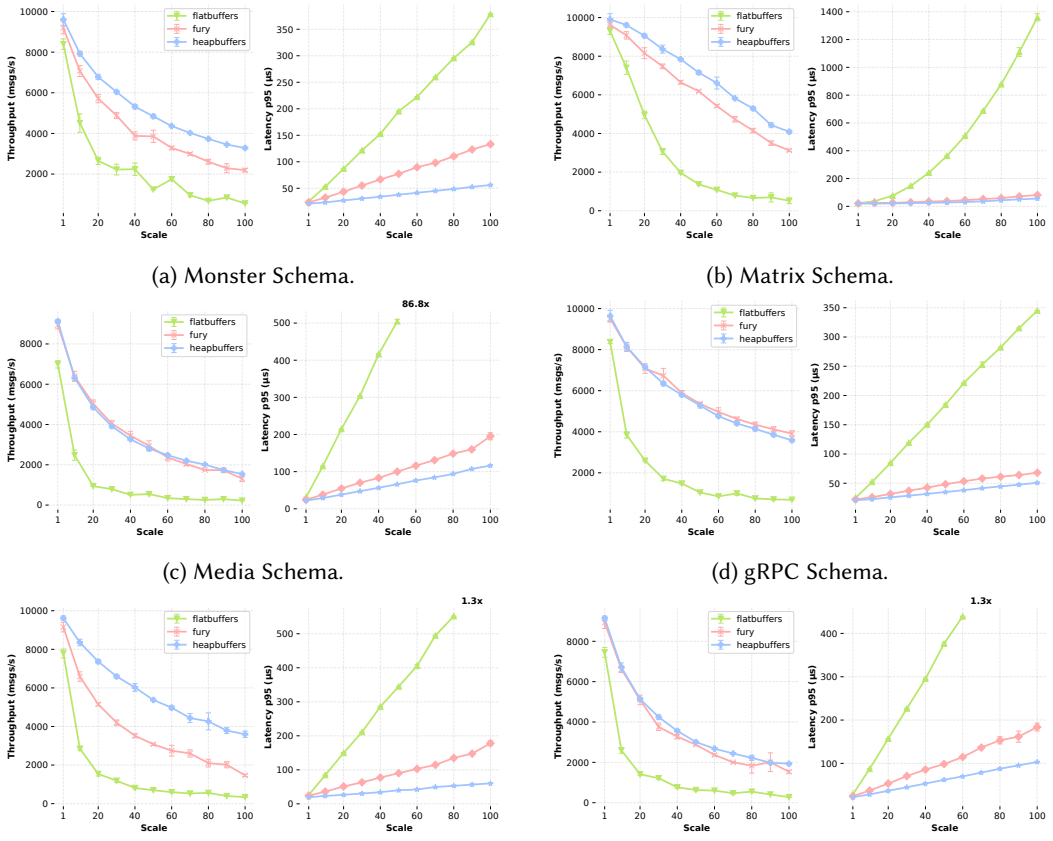
(f) Serialize with scale 100.

Fig. 19. Extended performance comparison of DSD operations for the schemas presented in Table 1.

addition, we have revised the scaling strategy. The prior experiment scaled specific arrays within the schema, emphasizing certain data types. Instead, we scale the schema itself, where each root object is an array of that schema type. The Matrix schema is an exception, as scaling is based on the dimensions of the square matrix. Table 2 illustrates how messages scale in different schemas and binary formats. Note that the increase in size is not linear, as only the schema objects scale, while additional metadata remains constant. Also, Fury presents the smallest data sizes, while HB and FlatBuffers have similar sizes given their binary compatibility [4].

Figure 19 presents the deserialization performance of three libraries in different schemas. HB achieves zero-copy deserialization, while FlatBuffers and Fury exhibit a decrease in throughput as the scale increases. A small slowdown is seen in HB as the message increases (e.g., Figure 19e), which comes from the JMH harness that is not entirely isolated from the benchmark-under-test and adds memory pressure. This memory contention similarly affects the other two libraries. Fury and FlatBuffers show linear scaling, where throughput decreases approximately in an inverse proportion to the scaling factor, making their performance highly predictable. However, this trend does not hold for the Matrix schema, where scaling follows a quadratic pattern.

[4]Small differences between the two can be explained by different ordering of objects, which leads to different padding.



(a) Monster Schema.

(b) Matrix Schema.

(c) Media Schema.

(d) gRPC Schema.

(e) Sample Schema.

(f) Schedule Schema.

Fig. 20. Throughput and latency results for different schemas and message scales on Microservice A.

For serialization, HB outperforms both libraries across all schemas and scales. Compared to Fury, HB achieves a throughput increase ranging from 2× to 5×. The performance difference is larger when compared to FlatBuffers. Moreover, FlatBuffers is the only library where throughput decrease consistently follows the inverse of the scaling factor. This behavior is because FlatBuffers has to copy the data to Java objects. In contrast, both HB and Fury experience a smaller impact on throughput as the message size increases.

The results in this experiment confirm that HB provides efficient DSD operations on a wide range of schemas and scales, achieving better performance than state-of-the-art solutions.

5.5 End-to-End Performance Evaluation on Microservices

The previous experiments demonstrate that HB significantly outperforms other serialization libraries, although they primarily measured raw DSD performance. In contrast, the next experiment evaluates the efficiency of DSD operations within realistic cloud-application settings. We designed two end-to-end microservice benchmarks since we did not find existing open-source benchmarks predominantly composed of DSD operations. These microservices mimic realistic communication patterns by exercising the complete Linux and Java networking stacks using Java sockets

(`SocketChannel`). Client and server are co-located on the same machine and communicate over the loopback network interface to minimize I/O nondeterminism and ensure reproducibility.

The experiments simulate a client-server microservice setup: (1) In Microservice A, the client sends a message to the server, which reads the entire message, computes its checksum, and returns the result to the client. (2) In Microservice B, the client sends a message to the server, which updates a field at a random position and returns the modified message to the client, which then computes the checksum. The performance measurements represent the total time required to create, serialize, transmit, deserialize, and validate.

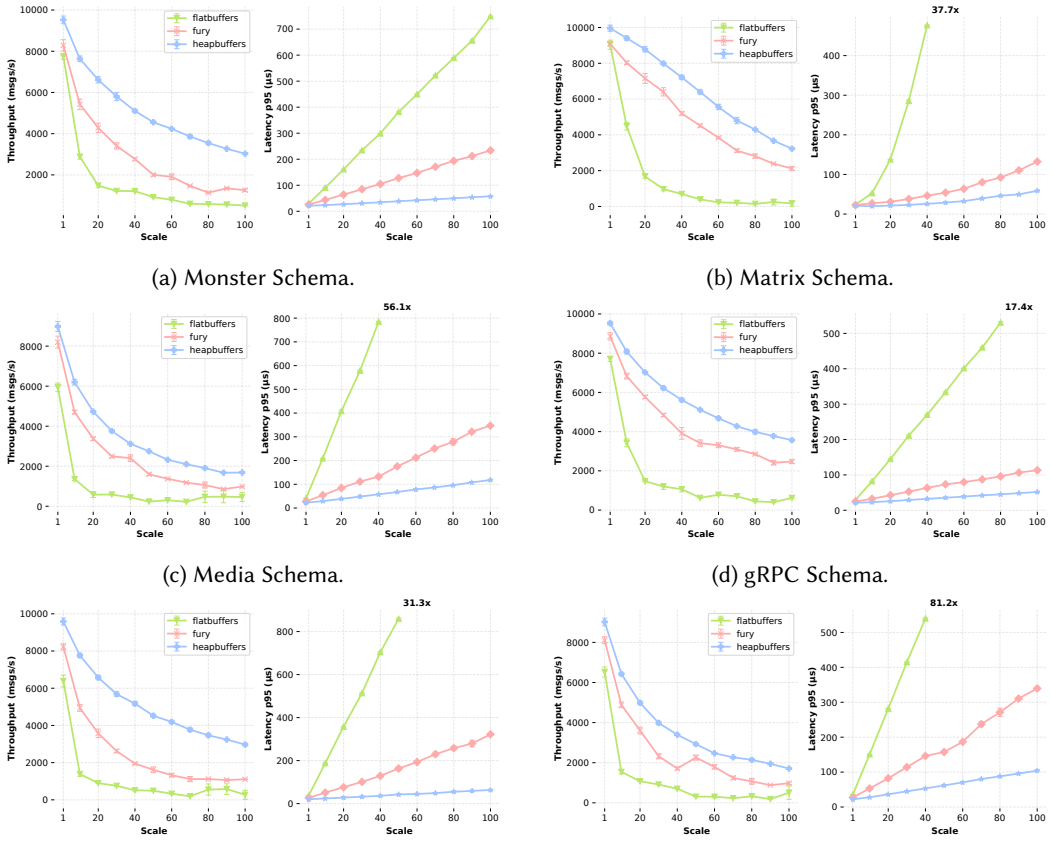
5.5.1 Microservice A: Round-Trip Performance. Figure 20 presents the performance results for Microservice A, reporting throughput in messages per second and 95th percentile latency. All measurements are obtained from the client's perspective, with each data point representing the arithmetic mean and standard deviation over five executions. For HB and FlatBuffers, read and write operations are performed directly on their respective objects (e.g., HB objects or FlatBuffers objects), eliminating the overhead associated with intermediate Java objects, as the application only accesses and manipulates data in a local context.

As shown in Figure 20, HB achieves higher throughput in many schemas (Figures 20a, 20b, and 20e) and better latency in all schemas. HB and Fury achieve similar throughput in Figures 20c, 20d, and 20f. One reason is that these schemas use many string fields. In these cases, HB must encode and copy the data from the Java heap to the transferable heap, introducing additional overhead and increasing memory write load. Another reason is that Fury generates smaller messages (shown in the previous table 2), minimizing the amount of data transmitted over the socket channels.

Still, on these very same schemas, HB also shows much better latency values. For example, at scale 100 (Figure 20c), by decomposing the message total latency, we observe that HB has a response time of 16 μs, while Fury has 113 μs. The difference is because HB employs zero-copy deserialization, while Fury has to recreate the entire object first and only then can access its fields. Refer to previous Figure 19 for recalling the difference between the libraries on deserialization operations. Finally, FlatBuffers consistently achieves the worst performance, with the lowest throughput and highest latency among the evaluated libraries.

5.5.2 Microservice B: In-Place Mutations. Figure 21 presents the results for Microservice B. HB outperforms the other two libraries in all schemas and scales in both throughput and 95th percentile latency. In this experiment, the server receives a message, modifies a field at a random position, and sends the updated message as a response. This microservice model aligns well with one of HB's key advantages: it enables efficient object *mutability*. The deserialized object can be accessed and modified in place (refer to Figure 8 for a code example), and since the data layout and remaining raw data remains unchanged, it can be immediately retransmitted to the client. In contrast, both FlatBuffers and Fury require full deserialization of the received data, modification of the target field, and subsequent re-serialization before transmission. This additional processing introduces overhead, resulting in higher latency and lower throughput compared to HB.

These findings are consistent with the discussion in Section 3.3 about object mutation. HB natively supports in-place mutations by allowing raw data received via sockets to be directly mapped into the HB runtime as a new transferable heap. Once mapped, the data can be manipulated using HB objects. Primitive in-place mutations (e.g., modifying an integer or a double) do not alter the data layout, enabling the data structure used for receiving raw data (e.g., `ByteBuffer`) to be sent back with the updated value. Moreover, HB also allows new object allocations or modifications that alter the schema, such as adding new strings. However, in these cases, the data must be serialized again. Hence, HB GC guarantees an IO-ready binary format.



(a) Monster Schema.

(b) Matrix Schema.

(c) Media Schema.

(d) gRPC Schema.

(e) Sample Schema.

(f) Schedule Schema.

Fig. 21. Throughput and latency results for different schemas and message scales on Microservice B.

These end-to-end microservice experiments reveal that HB can outperform state-of-the-art solutions. The results indicate that it consistently achieves the lowest latency across all workloads while delivering superior or, in some cases, comparable throughput. However, when microservices require in-place modifications, HB significantly outperforms both Fury and FlatBuffers.

6 Related Work

The cost of DSD operations is a well-known issue in modern cloud computing [Bharti et al. 2022; Lei et al. 2023; Li et al. 2022] and in many other data-intensive applications [Jang et al. 2020]. Generally speaking, two main directions have been explored to reduce or remove DSD operations' overheads.

First, a significant area of research focuses on eliminating the overheads of serialization and deserialization by using the unmodified internal object representation of languages like Java or Python in data transfers. In this approach, minimal DSD costs have to be paid as objects need not to be converted from their in-memory representation to the actual binary format that will be transferred on the wire. Initially introduced in different flavors [Nguyen et al. 2018], this approach has since been applied in various ways. Recently, it was combined with RDMA networking to further reduce the latency of microservice applications [Gienieczko et al. 2024; Lu et al. 2024; Taranov et al. 2021]. A main limitation of all these approaches is that objects stored in the VM

memory space are not platform independent and are not optimized for IO (e.g., they are not dense and still contain object headers). Not having platform-independent objects means that references need to be re-constructed when a message is received, since they may not be valid in the memory space of the receiving application. An unoptimized IO format means data transfer is less efficient than formats like Protobuf or FlatBuffers. HB shares with these approaches the key idea that no serialization should happen and that objects should be transferred from the VM memory without modifications. Unlike such approaches, HB relies on an IO-optimized format.

A second broad line of research has tried to speed up and improve DSD operations instead of completely removing them. In this approach, compiler optimizations or hardware acceleration are used to speed up (de)serialization. Examples in this context include speculative parsing [Bonetta and Brantner 2017; Li et al. 2017], using SIMD instructions [Langdale and Lemire 2019], and leveraging FGPAs to implement hardware-accelerated DSD operations [Dann et al. 2022]. Within JVM languages, just-in-time compilation can be used to optimize costly reflective operations and accelerate read/write access to Java objects. Apache Fury, a highly efficient serialization library for Java [Apache Software Foundation 2023], uses this method. While serialization methods utilizing these techniques can greatly enhance performance, they are fundamentally constrained by the necessity to access data from GC-managed memory and transform it into a specific binary format.

Finally, two other lines of research are closely related to HB. Mostly serialized heaps [Koparkar et al. 2024] are a model of organizing the memory space of a programming language using a platform-independent format. In this model, the entire memory space of the language is stored using relative offsets instead of references and can therefore be considered as binary serialized data. The approach shares with HB the intuition that data in the language memory space should already be represented in a platform-independent way. The key difference with the HB approach is that our model explicitly targets IO messaging and as such relies on a popular binary format that is optimized for small messages (FlatBuffers). Moreover, HB operates in the context of managed languages such as Java, where it would not be possible to represent the entire heap of a JVM using the techniques described in [Koparkar et al. 2024], given the peculiarities of the Java language (e.g., the need to keep object metadata in each object header). Another research area that shares a vision similar to that of HB is the Cornflakes serialization library [Raghavan et al. 2023]. Cornflakes shares with HB the idea that scatter/gather vectorized IO can be used to drive object serialization. Unlike Cornflakes, however, HB operates within a fully managed programming language with automatic memory management. Cornflakes, instead, has been designed to operate within native languages such as Rust and is therefore more aligned with traditional libraries such as FlatBuffers or Protobuf, inheriting all the limitations of their approach when used in the context of a language like Java.

7 Conclusion

In this paper, we have introduced the HB library, which offers a Java-friendly programming model and is designed to reduce the overhead of data serialization in managed languages, particularly Java and the JVM. HB allows objects to remain in a binary, IO-ready format directly within the JVM process, eliminating the need for traditional serialization. By integrating a custom garbage collection mechanism, HB significantly optimizes memory management and enhances performance for IO-heavy applications. HB completely eliminates the need to perform DSD operations, resulting in improved throughput and lower latency compared to existing binary formats. In future work, we plan to expand the HeapBuffers model to other programming languages and explore how managed language runtimes can use HB in combination with advanced high-performance networking stacks (e.g., RDMA). We also intend to explore the application of the HB model in areas beyond networking and cloud applications, such as high-performance storage systems.

Data-Availability Statement

Our HeapBuffers implementation, the benchmarks, and the raw data from our evaluation are available within our artifact on Zenodo [Bonetta et al. 2025].

Acknowledgments

This work was supported by Oracle and by the Dutch National Growth Fund 6G FNS.

References

Amazon Web Services. 2015. AWS Lambda: Run Code Without Thinking About Servers. https://aws.amazon.com/lambda/ Accessed: 2025-03-01.

Apache Software Foundation. 2023. Apache Fury: A blazing-fast cross-language serialization framework powered by just-in-time compilation and zero-copy. https://fury.apache.org Accessed: 2025-03-01.

Urmil Bharti, Anita Goel, and S. C. Gupta. 2022. A Scalable Design Approach for State Propagation in Serverless Workflow. In *2022 IEEE 3rd Global Conference for Advancement in Technology (GCAT)*. 1–7. doi:10.1109/GCAT55367.2022.9972158

Daniele Bonetta and Matthias Brantner. 2017. FAD.js: fast JSON data access using JIT-based speculative optimizations. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1778–1789. doi:10.14778/3137765.3137782

Daniele Bonetta, Júnior Löff, Matteo Basso, and Walter Binder. 2025. HeapBuffers: Why not just using a binary serialization format for your managed memory? (Artifact). doi:10.5281/zenodo.15751987 Accessed: 2025-08-01.

Caucho Technology. 2004. Hessian: Binary Web Service Protocol. http://hessian.caucho.com Accessed: 2025-03-01.

Jonas Dann, Royden Wagner, Daniel Ritter, Christian Faerber, and Holger Froening. 2022. PipeJSON: Parsing JSON at Line Speed on FPGAs. In *Proceedings of the 18th International Workshop on Data Management on New Hardware* (Philadelphia, PA, USA) *(DaMoN '22)*. Association for Computing Machinery, New York, NY, USA, Article 3, 7 pages. doi:10.1145/3533737.3535094

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794

Esoteric Software. 2025. Kryo: A fast and efficient binary object graph serialization framework for Java. https://github.com/EsotericSoftware/kryo Accessed: 2025-03-01.

Mateusz Gienieczko, Filip Murlak, and Charles Paperman. 2024. Supporting Descendants in SIMD-Accelerated JSONPath. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 338–361. doi:10.1145/3623278.3624754

Google Inc. 2014. FlatBuffers Schema Language. https://google.github.io/flatbuffers/flatbuffers_guide_writing_schema.html Accessed: 2025-03-01.

Google Inc. 2016. FlatBuffers: Memory Efficient Serialization. https://google.github.io/flatbuffers/ Accessed: 2025-03-01.

Google Inc. 2017. Google Cloud Functions. https://cloud.google.com/functions Accessed: 2025-03-01.

Google Inc. 2018. Protocol Buffers: Google's Data Interchange Format. https://developers.google.com/protocol-buffers Accessed: 2025-03-01.

Google Inc. 2025. FlatBuffers: A cross platform serialization library architected for maximum memory efficiency. https://github.com/google/flatbuffers Accessed: 2025-03-01.

IEEE Standards Association. 2023. IEEE P802.3df™ Defines Architecture Holistically to Achieve 800 Gb/s and 1.6 Tb/s Ethernet. https://standards.ieee.org/ieee/802.3df/ Accessed: 2025-03-01.

Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A specialized architecture for object serialization with applications to big data analytics. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 322–334. doi:10.1109/ISCA45697.2020.00036

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* CRC Press, Boca Raton, FL.

Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 158–169. doi:10.1145/2872887.2750392

Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 462–478. doi:10.1145/3466752.3480051

Chaitanya S. Koparkar, Vidush Singhal, Aditya Gupta, Mike Rainey, Michael Vollmer, Artem Pelenitsyn, Sam Tobin-Hochstadt, Milind Kulkarni, and Ryan R. Newton. 2024. Garbage Collection for Mostly Serialized Heaps. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management* (Copenhagen, Denmark) *(ISMM 2024)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3652024.3665512

Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.

Zhengyu Lei, Xiao Shi, Cunchi Lv, Xiaobing Yu, and Xiaofang Zhao. 2023. Chitu: Accelerating Serverless Workflows with Asynchronous State Replication Pipelines. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 597–610. doi:10.1145/3620678.3624794

Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *Proc. VLDB Endow.* 10, 10 (June 2017), 1118–1129. doi:10.14778/3115404.3115416

Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. 2022. FaaSFlow: enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 782–796. doi:10.1145/3503222.3507717

Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. 2024. Serialization/Deserialization-free State Transfer in Serverless Workflows. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 132–147. doi:10.1145/3627703.3629568

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. doi:10.1145/3173162.3173200

NVIDIA Corporation. 2023. ConnectX-7 Ethernet Datasheet. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/ethernet-adapters/connectx-7-datasheet-Final.pdf Accessed: 2025-03-01.

Oracle Corporation. 2013. Java Microbenchmark Harness (JMH). https://openjdk.java.net/projects/code-tools/jmh/ Accessed: 2025-03-01.

Oracle Corporation. 2014. JEP 254: Compact Strings. https://openjdk.org/jeps/254 Accessed: 2025-03-01.

Oracle Corporation. 2019. GraalVM Native Image: Ahead-of-Time Compilation for Java. https://www.graalvm.org/reference-manual/native-image/ Accessed: 2025-03-01.

Oracle Corporation. 2023a. Java Vectorized I/O: Performance Enhancements for Input/Output Operations. https://openjdk.org/jeps/384 Accessed: 2025-03-01.

Oracle Corporation. 2023b. JEP 409: Sealed Classes. https://openjdk.org/jeps/409 Accessed: 2025-03-01.

Oracle Corporation. 2023c. JEP 446: Scoped Values (Preview). https://openjdk.org/jeps/446 Accessed: 2025-03-01.

Oracle Corporation. 2024a. Direct ByteBuffer API Documentation. https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/nio/ByteBuffer.html#isDirect() Accessed: 2025-03-01.

Oracle Corporation. 2024b. Project Valhalla: JVM Support for Value Types and Specialized Generics. https://openjdk.org/projects/valhalla/ Accessed: 2025-03-01.

Protostuff. 2011. Protostuff: Protocol Buffers and Serialization Library for Java. https://github.com/protostuff/protostuff Accessed: 2025-03-01.

Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. 2023. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 200–215. doi:10.1145/3600006.3613137

Sandstorm Development Group. 2012. Cap'n Proto. https://capnproto.org/ Accessed: 2025-03-01.

Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 733–750. doi:10.1145/3373376.3378450

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. doi:10.1145/2581122.2544157

Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free {rdma} networking in java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 1–14.

Juan Cruz Viotti and Mital Kinderkhedia. 2022. A survey of JSON-compatible binary serialization specifications. *arXiv preprint arXiv:2201.02089* (2022).

Received 2025-03-25; accepted 2025-08-12