Zero-Change Object Transmission for Distributed Big Data Analytics

Mingyu Wu, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University and Shanghai AI Laboratory;* Shuaiwei Wang, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University;* Haibo Chen and Binyu Zang, *Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University and Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China*

https://www.usenix.org/conference/atc22/presentation/wu

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

NetApp®

Zero-Change Object Transmission for Distributed Big Data Analytics

Mingyu Wu[1,2], Shuaiwei Wang[1], Haibo Chen[1,3], and Binyu Zang[1,3]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Shanghai AI Laboratory
[3]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Distributed big-data analytics heavily rely on high-level languages like Java and Scala for their reliability and versatility. However, those high-level languages also create obstacles for data exchange. To transfer data across managed runtimes like Java Virtual Machines (JVMs), objects should be transformed into byte arrays by the sender (*serialization*) and transformed back into objects by the receiver (*deserialization*). The object serialization and deserialization (OSD) phase introduces considerable performance overhead. Prior efforts mainly focus on optimizing some phases in OSD, so object transformation is still inevitable. Furthermore, they require extra programming efforts to integrate with existing applications, and their transformation also leads to duplicated object transmission. This work proposes Zero-Change Object Transmission (ZCOT), where objects are directly copied among JVMs without any transformations. ZCOT can be used in existing applications with minimal effort, and its object-based transmission can be used for deduplication. The evaluation on state-of-the-art data analytics frameworks indicates that ZCOT can greatly boost the performance of data exchange and thus improve the application performance by up to 23.6%.

1 Introduction

High-level languages like Java and Scala are welcomed in areas like big-data analytics thanks to their reliable and versatile managed runtime environment. However, the abstraction provided by the managed runtime also introduces performance overhead, especially for data exchange. Since managed runtimes like Java Virtual Machines (JVMs) store data in an opaque object-based format, they have to transform objects into interpretable binary streams before exchanging. The transformation contains two phases: a *serialization* phase transforming objects into a byte array, and a *deserialization* phase transforming the byte array back into objects. The object serialization/deserialization (OSD) mechanism introduces considerable transformation overhead and has become a significant performance bottleneck in distributed object transmission, especially for applications demanding large-scale data exchange through network [3, 6, 13, 15, 44].

Prior work has recognized the performance problem in OSD and proposed different approaches, both in software [26, 38, 39] and hardware [16, 32, 40, 46], to mitigate its effect. However, those approaches mainly focus on optimizing specific phases in OSD, and the data transformation is still inevitable. Furthermore, although they can boost the performance of OSD, many of them require extra programming efforts to annotate serialization points or change the original inter-JVM communication model. Last but not least, they treat the transferred data as a monolithic byte array instead of individual objects, which makes it difficult to identify duplicated transmission and misses optimization opportunities.

Instead of optimizing OSD, this work aims at directly eliminating the whole OSD process. To this end, this work proposes Zero-Change Object Transmission (ZCOT), which provides an ideal data exchange mechanism where objects are transferred among JVMs through **direct object copying**. When a JVM receives objects from others, it can directly process them without any modifications (*Zero-Change*). ZCOT removes the demands for object transformation and thus improves the performance of data exchange.

However, it is non-trivial to achieve zero-change communication given each JVM manages objects in a process-specific and opaque format. To this end, ZCOT first introduces a globally shared abstraction named *exchange space*, a part of the Java heap space accessible for multiple JVMs in a distributed environment. ZCOT further adopts its distributed class-data sharing (DCDS) mechanism, which provides a unified object format to make objects in the exchange space interpretable for all JVMs. To remain compatible with traditional OSD-based applications, ZCOT proposes a two-level transmission mechanism to bridge the gap between object-based copying and traditional byte-based transmission.

As ZCOT introduces a globally shared exchange space, it is responsible to manage objects shared among multiple JVMs. By introducing a metadata server, ZCOT memorizes

the stored location for objects and helps build data transmission channels between JVMs. Since objects in big-data analytics are usually exchanged as a whole dataset, ZCOT embraces *group-based object management*, which organizes objects in groups and greatly reduces the traffic between the metadata server and JVMs. Furthermore, ZCOT also integrates with the garbage collections (GC) triggered in individual JVMs and reduces the GC pause time.

ZCOT sends objects instead of byte arrays during transmission, which makes it object-conscious and easier to identify duplicated objects. This work thus proposes a data deduplication mechanism to further optimize the data transmission. The deduplication module in ZCOT leverages the exchange space abstraction to memorize which objects have been sent and avoids unnecessary object transmission in the future. Nevertheless, deduplication may introduce references (or dependencies) among different datasets. To this end, ZCOT extends its distributed memory management module to consider inter-group dependencies.

This work implements ZCOT in the HotSpot JVM of OpenJDK 11, the long-time-support version for OpenJDK. ZCOT is well-integrated with existing features in OpenJDK (like APPCDS [30]) to remain friendly to Java developers. We evaluate ZCOT against state-of-the-art OSD libraries and optimizations with both the micro-benchmark and macro-benchmark. The micro-benchmark contains both basic and complicated data structures for data transmission, while the macro-benchmark contains two big-data analytics frameworks (Spark and Flink). The result for micro-benchmark shows that ZCOT outperforms other OSD libraries, especially for complicated data structures, and reaches up to 4.35× speedup compared with Naos [39], a state-of-the-art optimization on OSD. As for macrobenchmark, ZCOT outperforms the default OSD libraries in both Spark and Flink and thus boosts the application time by up to 23.6% and 22.2%, respectively.

To summarize, the contribution of ZCOT includes:

- A distributed shared abstraction named *exchange space* to enable zero-change object transmission among JVMs while remaining compatible with traditional OSD-based applications.
- A memory management mechanism on the globally shared space integrated with GC in individual JVMs.
- A data deduplication module to identify and eliminate unnecessary object transmission for further performance improvement.
- Experiments on communication-intensive workload to show the performance improvement of ZCOT over existing OSD libraries.

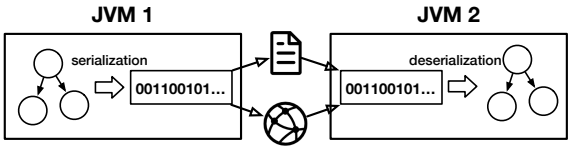


Figure 1: The workflow of OSD

2 OSD Background

2.1 OSD

Language runtimes provide a high-level abstraction for platform-independent code execution. As for user objects, runtimes store them with an opaque format, which maintains object data together with corresponding metadata (type information, synchronization, memory management, etc.). Taking Java as an example, JVMs maintain a *header* for each object to store its metadata.

However, when data exchange among JVMs is required, objects must go beyond the runtime scope. For example, objects might be persisted into disks and reused by other JVMs later; they can also be sent and received through network. To support those scenarios, objects have to be interpretable even when leaving JVMs. Therefore, JVMs embrace the object serialization/deserialization (OSD) mechanism, which transforms Java objects into a generalized data format (serialization) and transforms back when reusing in JVMs (deserialization). The Java system library (JSL) already provides a built-in OSD library for applications. Figure 1 shows the workflow of JSL's OSD. As for the serialization part, objects are transformed into a byte array that follows a data format agreed among JVMs. The byte array will be written into disks or sent through network. When another JVM receives the byte array, it transforms the byte array back into objects through deserialization.

The OSD mechanism has two major advantages. First, the library provides a general-purpose data format so that Java objects can be transformed among JVMs with different versions and configurations. Second, the serialized data is compressed and induces smaller footprints in both disks and network.

2.2 Limitations and opportunities

The major disadvantage for OSD is its performance penalty. The performance problem of OSD in big-data analytics is three-fold.

Transformation overhead. OSD introduces extra phases for object persistence and transmission. To serialize an object, OSD should traverse all its reachable objects and store their type information. As for deserialization, OSD should scan serialized data and reconstruct objects.

Memory footprint. OSD generates a considerable number of temporary objects during data transformation. As shown

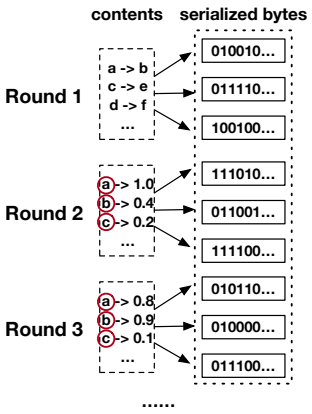


Figure 2: Duplicated data transmission in the page-rank application

in Figure 1, byte arrays are generated during serialization and become useless after sending out. Those temporary objects can increase the memory pressure and cause more frequent GCs.

Duplicated transmission. Big-data analytics leverage OSDs in many rounds of communication and duplicated objects may be repetitively transformed and exchanged in each round. Figure 2 shows a concrete example in Spark [44], which calculates the popularity for each URL (simplified as letters) with the page-rank algorithm [31]. Since the algorithm executes for multiple iterations, the data transmission is also conducted in many rounds. In the first round, the URL-based network topology is sent through network, which consists of many string pairs to indicate the point-to relationships among URLs. In the later rounds, the rank value for each URL is iteratively propagated, which is organized as key-value pairs. Note that the strings in the key-value pairs are URLs that have been sent in the first round. Unfortunately, since all objects have been transformed and merged into byte arrays, JVMs cannot tell that some objects have been received before. They have to receive all objects as a monolithic byte array, which leads to unnecessary network transmission and OSD phases. In the Spark page-rank application, over 60% of transferred objects are duplicated.

Furthermore, the advantages of OSD also fade with advances in hardware technologies. For example, the bandwidth of off-the-shelf network devices can reach 100Gb/s or larger, which makes network transmission time less important, so OSD may become a more significant bottleneck. On the other hand, the general data format is not always required. Therefore, many optimizations have been proposed to reduce the performance overhead of OSD, both in software [26,38,39] and hardware [16,32,40,46]. Since hardware-based approaches require building customized hardware accelerators to improve OSD, this work mainly focuses on software-based approaches with off-the-shelf hardware.

2.3 State-of-the-art optimizations

The basic idea behind OSD is to achieve an *agreement* on object representation among JVMs. Therefore, optimizations on OSD should consider how to create the agreement so that objects can cross JVMs' boundaries. Besides, they also need to consider issues like compatibility with existing applications.

Kryo. Kryo [38] is a fast OSD library for Java. Compared to JSL's OSD, Kryo refines the binary data format to achieve a smaller serialized data size and better performance. Applications like Spark have leveraged Kryo as its default serializer. Nevertheless, Kryo does not eliminate any phases in OSD; objects still need to be transformed back and forth.

Skyway. Skyway [26] proposes to directly send object graphs instead of serialized bytes. With Skyway, the serialization phase is nearly removed as objects are no longer transformed to a binary format. Although Skyway has simplified phases in OSD, modifications on objects are still required. First, it needs to transform the type information in the headers to a globally-agreed ID so that it can be identified by all JVMs. Second, it needs to fix references after copying, as objects have been moved to different addresses. Moreover, Skyway also requires programmers to mark the point where the serialization phase starts manually.

Naos. Naos [39] is a network-specific data transmission mechanism. Similar to Skyway, Naos also employs a global service to reach agreements for types, but it relies on RDMA technology to achieve rapid zero-copy object transmission. However, Naos still requires modifications on both object headers and references. Besides, it only supports network-based transmission, and existing applications need significant modifications to leverage Naos.

2.4 Summary

Prior optimizations have proposed different solutions to reduce the overhead of OSD. However, they cannot eliminate the whole OSD process. Table 1 compares the built-in OSD in JSL with other optimizations. Although recent work like Naos eliminates the serialization phase, a deserialization phase is still required to fix the type information and the references. Besides, none of them has considered the duplicated data transmission problem.

This work proposes ZCOT (short for Zero-Change Object Transmission), which aims to eliminate the whole OSD process during data exchange. In ZCOT, object transmission is conducted in the most straightforward way: the sender JVM copies objects and the receiver can directly use them without any modifications. ZCOT also considers the duplicated transmission problem and provides a deduplication module. Finally, ZCOT is not bound to specific network technologies (like RDMA) and provides easy-to-integrate interfaces for existing applications.

Data transmission mechanisms	Serialization	Deserialization	Ease of Integration	Data deduplication
JSL	Slow	Slow	Yes	No
Kryo	Medium	Medium	Yes	No
Skyway	Fast (removed)	Medium	Medium	No
Naos	Fast (removed)	Medium	No	No
ZCOT (this work)	**Fast (removed)**	**Fast (removed)**	**Yes**	**Yes**

Table 1: Comparisons on existing OSD optimizations against our work ZCOT

3 Design of ZCOT

3.1 Overview

The core idea of ZCOT is to build a distributed-shared-memory (DSM)-like abstraction for JVMs running on different machines. Figure 3 illustrates the architecture of ZCOT. In a ZCOT-enabled system, the heap for each JVM consists of two parts: its private space (the original Java heap) and a globally shared *exchange space*. Objects are originally managed in the private space. When they require to be sent through network or persisted into disks, they will be copied to the exchange space. The exchange space is an abstraction available for all JVMs; each JVM can directly access objects therein. Therefore, object transmission can be achieved with direct copying to the exchange space, and the whole OSD process can be eliminated.

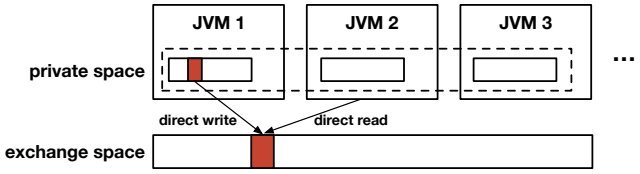


Figure 3: The architecture of ZCOT

The idea for building a DSM-like abstraction is well-known and has been studied for decades [2, 7, 9, 14, 20, 21, 25, 33–35, 41, 42, 45]. Although our exchange space shares similar wisdom with DSM, it is only used for data exchange and does not need to tackle complicated issues like coherence. It also assumes objects in the exchange space are immutable, which usually holds for big-data analytics like Spark and Flink. If a write operation occurs on objects in the exchange space, ZCOT creates a copy for it on the JVM's private heap. Nevertheless, combining the DSM concept with data transmission in high-level languages is still not trivial. To enable efficient and easy-to-use object transmission, ZCOT has to resolve the following challenges.

- How to build a shared exchange space so that all JVMs can access it freely? (Section 3.2)
- How to leverage the exchange space abstraction to support OSD-based applications? (Section 3.3)
- How to manage objects in the exchange space in the presence of garbage collections in individual JVMs? (Section 4)
- How to resolve the duplicated transmission problem? (Section 5)

3.2 Distributed class-data sharing

ZCOT relies on its distributed class-data sharing (DCDS) mechanism to build a globally accessible shared space. DCDS guarantees that class-related metadata will be mapped into the same virtual memory address for all JVMs. This helps JVMs to achieve an agreement on the class metadata, so no type-related modifications (e.g., identifiers) are required.

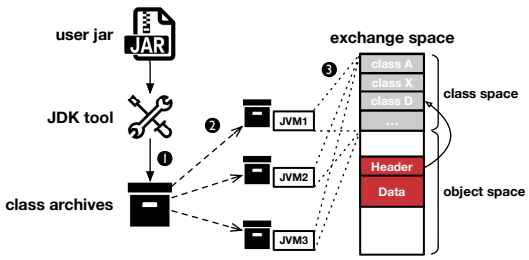
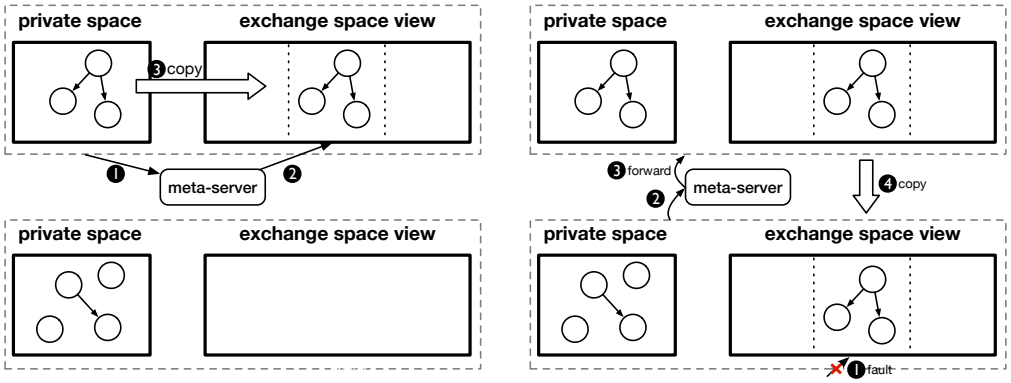


Figure 4: The workflow of distributed class-data sharing

Figure 4 elaborates the workflow of DCDS. First, the cluster manager should prepare a shared class archive for all JVMs. The class archive should contain all classes whose corresponding object instances would be shared during inter-JVM communication. ZCOT relies on the tools provided by OpenJDK to generate such class archives [30]. Afterward, the archive will be used during JVM startup, and classes in the archive will be mapped to a given virtual address. The virtual address range is also memorized and marked as a part of the exchange space (*class space* in Figure 4). This step assures that JVMs share the same view on the classes. As shown in Figure 4, an object in the exchange space stores a reference to its class-related metadata. Since the reference points to the class space, the object's class information is interpretable for all JVMs. Although DCDS requires the data types of applications should be known in advance, mainstream big-data analytics frameworks usually guarantee this by sending a fat jar file for execution.

Figure 5 shows how ZCOT transfers objects through network with its DCDS support. First, the sender JVM applies for an available memory chunk in the exchange space for object copying. This is achieved by communicating with an external



(a) Sender: deep copy to a given address

(b) Receiver: trigger a page fault and fetch data from the sender

Figure 5: The workflow of ZCOT

metadata server (details in Section 4). Second, the sender JVM copies objects to the chunk's memory address. This step is similar to a deep copy in a normal Java application. To detect cycles and avoid repeated copying on the same object, we add a marker word in each object header to store its new address if it has been copied.

The copied objects are kept on the sender machine and lazily retrieved by receivers. When a receiver JVM tries to access this part of data (Figure 5b), it encounters a page fault since the data is unavailable on its machine. We have registered the page fault handler in ZCOT-enabled JVMs so that they can request the metadata server for faulted pages. The metadata server has tracked the ownership of memory addresses in the exchange space, so it forwards the request to the data owner. Afterward, the sender builds a connection with the receiver and puts the requested objects to the desired address. Now the receiver can directly access those objects for further processing, with neither metadata updating nor reference fixing (namely *zero-change*).

3.3 Supporting OSD-based scenarios

Thanks to the exchange space abstraction, a JVM can directly access received objects without any modifications. However, this mechanism is not compatible with traditional OSD-based applications, which usually adopt byte arrays for communication. To this end, ZCOT should provide user-friendly interfaces to integrate easily with applications.

Programming interfaces. JSL provides stream-based classes for OSD implementation. The `ObjectOutputStream` class provides the `writeObject` method to serialize an object into a stream (usually files or network). Similarly, the `ObjectInputStream` class provides the `readObject` method to deserialize data into objects. Therefore, prior OSD optimizations like Skyway implement new serializers/deserializers by inheriting those two classes for ease of integration. ZCOT adopts a similar strategy and Figure 6 shows its basic classes: `ZCObjectOutputStream` and `ZCObjectInputStream`, which are subclasses of `ObjectOutputStream` and `ObjectInputStream`, respectively. Compared with `ObjectOutputStream`, `ZCObjectOutputStream` slightly modifies the interface for `writeObject` to support different OSD-based scenarios (discussed later). To use ZCOT-based communications, applications only need to replace the original stream classes with ours. In contrast, prior work requires developers to modify the original communication model or annotate the serialization points [26, 39].

OSD-compatibility. To remain compatible with OSD interfaces (`writeObject` and `readObject`), ZCOT should also transfer data with byte arrays. To this end, ZCOT adopts a two-level transmission mechanism. As illustrated in Figure 7, ZCOT transfers data via both *frontend* and *backend*. The frontend transmission is compatible with OSD interfaces, but it only sends metadata, including the object's start address and the data length. When `ZCObjectInputStream` receives the metadata through `readObject`, it directly accesses the corresponding address and fetches objects through backend transmission if a page fault is triggered (as mentioned in Figure 5b). ZCOT will launch dedicated VM threads in both sender and receiver JVMs to transfer the requested objects. This two-level design fills the gap between the byte-based OSD interfaces and the object-based transmission in ZCOT.

Supporting different OSD scenarios. In OSD libraries, objects are serialized and written into a stream (e.g., the *out* variable defined on Line 3 in Figure 6) when invoking `writeObject`, which are usually redirected into files or network. To support both scenarios, ZCOT adds a parameter *volatile* in the constructor of `ZCObjectOutputStream` (Line 6). When *volatile* is set to *false*, the copied objects will be written into a file, and the memory pages can be soon reclaimed

```
// Output class
class ZCObjectOutputStream extends ObjectOutputStream {
    private OutputStream out; // Private output stream

    // Constructor
    public ZCObjectOutputStream(OutputStream out,
         boolean volatile /* Mode */)
         throws IOException {...}

    // Compatible with the serialization interface
    public void writeObject(Object obj)
         throws IOException {...}
    ...
}

// Input class
class ZCObjectInputStream extends ObjectInputStream {
    private InputStream in;  // Private input stream

    // Constructor
    public ZCObjectInputStream(InputStream in)
         throws IOException {...}

    // Compatible with the deserialization interface
    public Object readObject()
         throws IOException{...}
    ...
}
```

Figure 6: Basic classes in ZCOT

through GC (details in Section 4). Nevertheless, those objects still reserve a corresponding virtual address in the exchange space. When the object data is read by other JVMs, the metadata server asks the sender to pass the file so the receiver can map it to the corresponding memory address. The case is simpler when *volatile* is *true*, which indicates a network-based transmission. In this scenario, objects are only kept in memory and can be reclaimed only if they have been read by others.

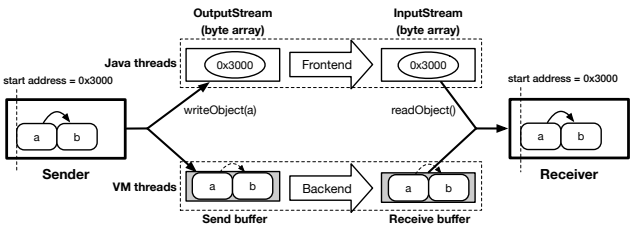


Figure 7: The two-level data transmission mechanism in ZCOT

Assumptions. Note that ZCOT is mainly designed to improve the data exchange phase for big data analytics, so it makes several assumptions about the transferred data. First, all classes related to communication should be known in advance so that they can be packed into the class archive. Second, the transferred objects are read-mostly, otherwise copy-on-write operations would be triggered for modification operations. Lastly, objects are managed in large groups and share similar life cycles, so they can be efficiently managed in the exchange space. Since representative big-data analytics systems like Spark conform to the above assumptions, ZCOT works well for them.

4 Memory Management

Since the global exchange space is built atop a DSM-like abstraction, ZCOT should manage objects distributed to different machines. Furthermore, the managed runtimes complicate the scenario as they introduce their own memory management strategy: garbage collections (GC). This section will introduce how ZCOT manages the distributed exchange space while remaining harmonized with GC in JVMs.

4.1 Group-based management

Unlike traditional DSM-based systems, ZCOT introduces *group*, a semantic-aware notion for distributed memory management. As analyzed in Section 2, big-data analytics frameworks treat serialized objects as a whole dataset (monolithic byte array) and retrieve them together. Therefore, ZCOT puts all objects copied in the same `writeObject` invocation to a group so that they are managed together. When a receiver encounters a page fault, ZCOT will send all related data pages belonging to the same group to the receiver and avoid future faults. This mechanism, namely *group-based prefetching*, leverages the semantics in the OSD scenario to mitigate the page-based management overhead in traditional DSM.

4.2 Metadata server

The metadata server is the core module for ZCOT's memory management. JVMs communicate with the metadata server through remote procedure calls (RPCs) to acquire or release memory resources in the exchange space. Figure 8 illustrates the core data structures in the metadata server. The metadata server is agnostic to groups; groups are only managed by individual JVMs. It partitions the shared exchange space into equal-sized memory chunks (256MB by default) for memory allocation and deallocation. It also maintains an *allocation bitmap* to mark if a chunk has been allocated. Each chunk is assigned with an integer ID, which is calculated by its relative offset compared with the exchange space's start address. To track the stored locations of chunks, the metadata server maintains a *copy set* for each chunk, which is stored in a *chunk mapping table*. The copy set contains JVMs storing a copy of the corresponding chunk (in memory or disk), which are also represented with integer IDs. The mapping between a JVM's ID and information (e.g., IP address) is stored in a separated *member table*.

Since each JVM needs to communicate with the metadata server, its reliability becomes considerable. To tolerate failures on the metadata server, we can introduce replicas for it, and the overhead would be acceptable given the low frequency of communications between the metadata server and worker JVMs (several times in a data-processing stage lasting for seconds).

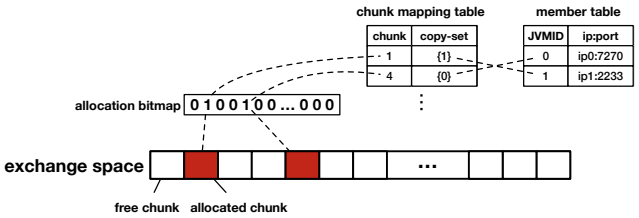


Figure 8: Important data structures in the metadata server.

4.3 RPC interfaces

The metadata server provides four important RPC interfaces listed below.

```
int register(std::string ip, int port);

Chunk* acquire();

Chunk* get_remote(Address addr);

int release(Chunk* chunk);
```

register. `register` is only invoked when a JVM is launched. ZCOT has provided a JVM option *-XX:+UseZCOT*, and a JVM enabling this option automatically spawns an RPC thread and sends a `register` RPC to the metadata server with its IP address and listening port. After receiving the RPC, the metadata server saves the IP and port number to the member table, generates an integer as the JVM's ID, and returns with the ID. For subsequent RPCs, JVMs should always attach the returned ID to help the metadata server maintain the stored locations of objects (omitted in the interfaces above).

acquire. When a JVM runs out of allocated memory from the exchange space, it should send `acquire` RPCs for more memory resources. After receiving an `acquire` request, the metadata server scans its bitmap to allocate an available chunk. Afterward, the metadata server memorizes the relationship between the allocated chunk and the JVM's ID and returns the chunk. To reduce the overhead of bitmap scanning, ZCOT memorizes the address of the last successfully allocated chunk and starts scanning there. If the scanned address reaches the end of the exchange space, ZCOT will continue scanning from the beginning. To handle simultaneous `acquire` requests, ZCOT introduces a bitmap lock to ensure the bitmap is exclusively accessed.

get_remote. The `get_remote` interface is used by JVMs encountering a page fault when accessing a virtual address. Since a page fault indicates the requested objects are not stored locally, the JVM sends `get_remote` to fetch the corresponding chunk. After receiving `get_remote`, the metadata server gets the corresponding chunk containing the address and finds which JVMs store the chunk by scanning the chunk mapping table. As illustrated in Section 3.2, the metadata server forwards the request to the corresponding JVM for actual data transmission. Since the size of a chunk is relatively large, sending chunks may introduce considerable performance overhead. To reduce the transferred data size, the sender JVM only sends used pages in the chunk, which are represented as the length of data in the frontend transmission (Figure 7). Due to ZCOT's group-based prefetching mechanism, the sender may directly send multiple chunks in the same group to the receiver. In this case, the receiver is responsible for sending an auxiliary RPC to update the copy set in the metadata server.

release. The `release` interface is relatively simple. When a JVM finds that objects in a chunk are no longer used, it sends `release` to give up this chunk. After receiving `release`, the metadata server removes the JVM's ID from the corresponding copy set in the chunk mapping table. If no JVM stores this chunk, the metadata server will reclaim it by marking the corresponding bit as *free* in the bitmap.

4.4 Garbage collection

JVMs have already implemented their garbage collection (GC) algorithms to automatically reclaim unused memory. When GC is triggered, JVMs track all live objects and reclaim memory consumed by dead ones. Since objects in the exchange space are reachable from individual JVMs, they will also be affected by GC. To this end, ZCOT has integrated its memory management strategy with G1, the default GC algorithm in OpenJDK, to ensure the correctness of distributed memory management and reduce GC overhead.

G1 basics. G1GC (short for Garbage-first Garbage Collection [8] is the default garbage collector after OpenJDK 9 [29]. G1 divides the Java heap into equal-sized *regions* for ease of management. It also maintains per-region metadata named *remember sets* to memorize all references pointing to objects in the same region. The remember set is updated by instrumenting all write operations in Java code (also known as *write barriers*). The G1 algorithm is mostly stop-the-world, which means that application threads should be paused until GC ends. During GC[1], each selected region is processed simultaneously: a dedicated GC thread scans the remember set of a region, finds all reachable objects, and copies them to an empty region (named *survivor region*).

Integrated with G1. ZCOT extends the region-based design of G1 to support the exchange space. It proposes *ZCRe-*

[1]For simplicity, we only discuss the young GC and mixed GC in G1

gion, a new kind of region allocated from the metadata server. Compared with regions in G1, the size of ZCRegion is not fixed. Each ZCRegion corresponds to a group in the exchange space, and all objects therein are expected to have the same life cycle. Since objects in ZCRegions have different behaviors compared with those in other regions, G1 should treat them specially. First, we modify the behavior of write barriers to consider ZCRegions. When a reference points to objects in a ZCRegion, we do not memorize this reference but only mark the ZCRegion as used. This is because objects in a ZCRegion are only collected when no references point to any of them. Similarly, GC threads do not need to scan ZCRegions during GC because all objects are treated as alive if there exists any reference pointing to the region. When GC ends, the JVM will scan all ZCRegions and find those containing no incoming references. For those ZCRegions, the JVM invokes the `release` RPC to reclaim corresponding chunks. If objects in a group are written into disks, the corresponding ZCRegion can also be reclaimed by GC, but the JVM does not invoke `release` since the virtual address is still reserved by the group.

In summary, our design successfully integrates the memory management of the exchange space with G1GC. When GC ends, the memory resource in the exchange space is automatically reclaimed by following the reachability-based algorithm. Furthermore, by specially handling regions in the exchange space, we avoid unnecessary metadata tracking and object scanning. In some cases, this design can even reduce GC pause time (as shown in Section 6.3).

5 Transmission Deduplication

Since ZCOT sends objects instead of byte arrays during transmission, it would be much easier to track transmitted objects and conduct deduplication. This section introduces the data deduplication module in ZCOT based on its object-centric transmission mechanism.

5.1 Overview

Figure 9 shows the effect of ZCOT's data deduplication module in the aforementioned page-rank example (compared against Figure 2). When sending the URL-based network topology in the first round, the sender has copied all URL string pairs (together with two string objects) into their corresponding addresses. In the next few rounds, the application sends key-value pairs to update rank values for each URL. Since all key-value pairs are sent as objects, it is much simpler for ZCOT to find that all URL objects have been sent. Therefore, the sender can directly update the references in those key-value pairs with the addresses in the exchange space and thus remove duplicated transmission on URL objects.

ZCOT runtime should be further extended to achieve data deduplication. First, ZCOT should track copied objects to rapidly find duplicated transmission. Second, ZCOT should manage dependencies among object groups for safe memory reclamation.

5.2 Duplication detection

A straw-man design for duplication detection would be scanning all objects in the exchange space. However, this design would induce considerable overhead given the large number of objects. ZCOT instead follows a simple detection criterion: if an object is in the exchange space, an attempt to copy it is a duplicated one.

We still use page-rank as an example to explain ZCOT's duplication detection. Suppose a JVM receives the network topology in round 1 (consider Figure 9); it reads URL objects from the exchange space and uses them in the following rounds. Therefore, when it propagates updated rank values to other JVMs, it still uses the URL objects received from others. When copying the URL-rank pairs in the next few rounds, ZCOT checks each object's address and thus avoids copying those URL objects already in the exchange space.

5.3 Dependency management

Although data deduplication in ZCOT can reduce the network overhead by avoiding repeated copying on the same object, it also complicates memory management by introducing inter-group references. As mentioned in Section 4.1, each invocation to `writeObject` creates a new group for object management, and each group is separately used by calling `readObject`. After deduplicating objects from different groups, objects in a group can hold references to those in another group, which should be correctly handled especially when a group is being garbage collected. To this end, ZCOT has managed those references as *dependencies* among groups.

Due to the large number of inter-group references, ZCOT does not maintain reference-level dependencies. When a group holds a reference to any objects in other groups, ZCOT marks the group as dependent on others. The dependency tracking is still achieved by extending the write barriers. To memorize all dependencies, ZCOT extends the chunk mapping table in the metadata server to contain a *dependency set* for each chunk, which stores all other chunks it relies on. When a JVM finds that its group relies on another group after deduplication, it sends a new RPC `add_dependency` to the metadata server. Since the metadata server is not aware of groups, the RPC should specify all chunk IDs owned by the group it relies on. Those chunk IDs will be added to the corresponding dependency set by the metadata server.

Figure 10 uses an example to illustrate how ZCOT leverages dependencies during object copying. After encountering a page fault on chunk 4, the receiver JVM sends a `get_remote` RPC to the metadata server. By fetching the dependency set in the chunk mapping table, the metadata server finds all chunks that chunk 4 depends on. Afterward,

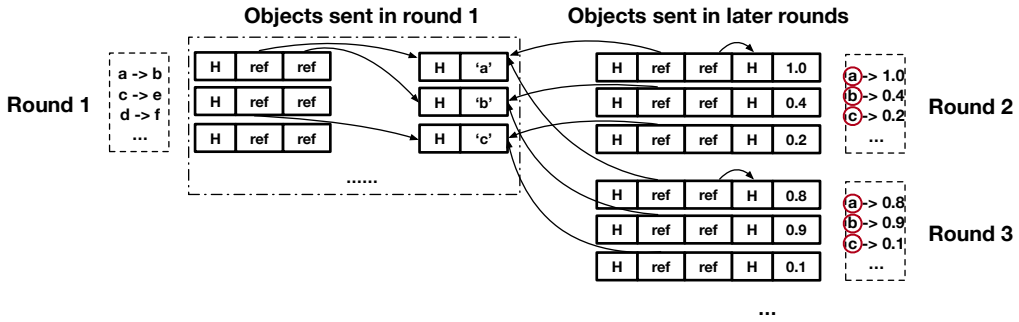


Figure 9: ZCOT avoids duplicated object transmission in page-rank

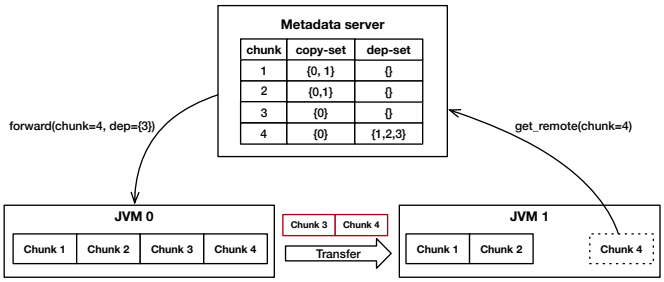


Figure 10: ZCOT avoids sending duplicated data with dependency tracking

ZCOT checks the copy set for each chunk to find if the receiver JVM already has a copy. If the receiver does not have a copy, ZCOT adds the corresponding chunk ID in a message, which will be forwarded to the sender JVM later for real data transmission. In our example, since the receiver JVM already has copied chunk 0 and 1, it only needs to receive chunk 4 (requested) and chunk 3 (dependent). This example indicates that ZCOT can avoid duplicated data submission with slight modifications on the metadata server.

5.4 Garbage collection

Adding dependencies also complicates GC for individual JVMs. Since a group (represented as ZCRegions) can be referenced by others stored in remote JVMs, local GC cannot determine if a group can be safely reclaimed. For example, suppose JVM 0 stores a group (chunk 0) that contains a reference to another group (chunk 1) stored on JVM 1. Although JVM 1 no longer contains references to chunk 1, the chunk should not be collected because JVM 0 may access it through references in chunk 0. To this end, we extend G1GC to consider remote inter-group references.

In our refined GC algorithm, once a JVM detects a ZCRegion has incoming references from other ZCRegions (through write barriers), it marks the region as *pinned* and thus cannot be reclaimed. It also sends the dependency relationship to the metadata server through RPCs. When GC ends, the JVM skips all pinned ZCRegions and only collects those with no incoming references. A pinned ZCRegion can be reclaimed when the metadata server finds that all chunks relying on it have been released. In this case, the metadata server will send a `canRelease` message to all JVMs in the corresponding copy set, and those JVMs will mark the ZCRegion as *unpinned* to safely reclaim it in later GC cycles.

5.5 Internalization

Big-data analytics usually generate a large number of objects with simple types, such as Integer, String, Double, etc. OpenJDK has provided an *internalization* mechanism to merge those objects with the same content together. For example, Integer objects whose values are between -128 and 127 would be merged into one if their values are equal. ZCOT also embraces this mechanism for deduplication, but in its distributed exchange space. It extends DCDS so that all JVMs allocate a small region at the same virtual address during start-up to contain globally-shared Integer objects. Thanks to this optimization, the number of transferred Integers can be greatly reduced.

6 Evaluation

6.1 Experimental setup

ZCOT is implemented atop the HotSpot JVM in OpenJDK 11.0.8-GA, with 8,327 lines of C code and 654 lines of Java code. We leverage the following workloads to evaluate ZCOT.

Microbenchmark. The microbenchmark contains four different data types used in prior work [26, 39]: 2-dimension points, key-value pairs, hashmaps, and media objects. To simulate big-data scenarios, we transfer them in large arrays whose length is 65536. Since some baselines crashed for large arrays of media objects, we reduced the length to 16384 for this data structure.

Spark. Spark (v3.0.0) is a data analytics engine that requires massive data transmission among JVMs.

Flink. Apache Flink [6] (v1.14) is a distributed data processing engine for both batch and streaming workloads.

As for baselines, we compare ZCOT with two commonly-used OSD libraries (JSL and Kryo) and two state-of-the-art OSD optimizations (Naos and Skyway[2]).

Our test environment includes a cluster with four nodes connected by 100 Gbit/s Mellanox ConnectX-5 NICs. Each node contains dual Xeon E5-2650 CPUs and 128GB DRAM.

6.2 Microbenchmark

To directly compare ZCOT with state-of-the-art OSD optimizations, we leverage the *microperf* tester in the Naos' open-source repository for evaluation. The tester involves a sender and a receiver deployed on two separate machines and reports the communication time with different type of data objects. The heap size for all workloads is 16GB.

Figure 11 shows the results for ZCOT and other baselines, which are the average of 1000 times of repetitive execution. ZCOT achieves the best performance of all except for 2-dimensional points. The average speedup is 2.28×, 1.94×, 2.19×, 3.95× compared with Naos, Skyway, Kryo, and JSL, respectively. The result also suggests that ZCOT performs better for complicated data structures. The media class from the Java serialization benchmark set (JSBS) [37] is the most complicated one, so the improvement is the largest especially against Naos (4.35×). This is because the computation overhead increases when the data structure becomes more complex. For simple data structures like points, ZCOT's reduction on data transformation is offset by larger network overhead, so it performs slightly worse than Naos and Skyway.

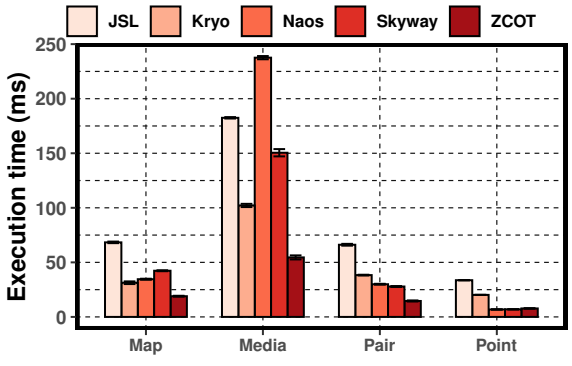


Figure 11: The evaluation results for microbenchmark

6.3 Spark

Ease of integration. To adopt ZCOT in Spark, we need to implement a new data serializer `ZCSerializer` to replace the default `KryoSerializer`. Although the name seems to involve OSD phases, it is only for compatibility considerations and still remains zero-change during transmission. `ZCSerializer` contains 70 lines of code, and most of them are inherited from the JSL serializer. Furthermore, we replace the original stream classes from JSL with ours. If a Spark user wants to enable ZCOT, she only needs to (1) configure the *spark.serializer* to `ZCSerializer` and (2) add *-XX:+UseZCOT* to the launch option of all JVMs, which is quite simple.

[2]Both implemented by Naos' authors

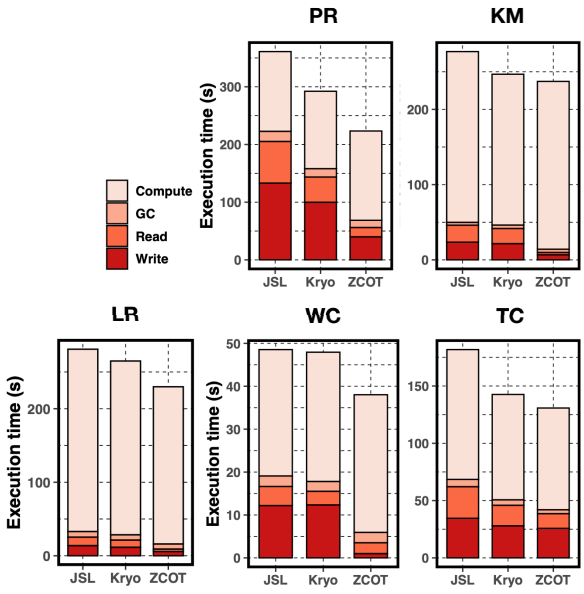


Figure 12: The performance of Spark applications

Evaluation results. We leverage five applications in the example directory of Spark for evaluation. Their descriptions and evaluated datasets are shown in Table 2. We configure one node as the metadata server and Spark master while the other three servers as Spark workers. The Java heap size for each node is set to 80GB.

Figure 12 shows the results for all applications. The results indicate that ZCOT can improve the performance by 13.9% and 24.1% on average compared with Kryo and JSL, respectively. Although Kryo has optimized the OSD performance over JSL, our evaluation shows that the data transmission can be further improved.

Application	Dataset
PageRank (PR)	LiveJournal [4]
Word Count (WC)	LiveJournal
KMeans (KM)	USCensus1990 [10]
Transitive Closure (TC)	Blogs [1, 17]
Logistic Regression (LR)	SUSY [5]

Table 2: Evaluated applications and datasets for Spark

We have further broken the results into four different phases: write (serialization), read (deserialization), compu-

tation, and garbage collection (GC). Since the four phases are not overlapped in Spark (the GC phase only contains stop-the-world time), the accumulated time is equal to the overall execution time. Figure 12 indicates that the performance mainly comes from the improvement in OSD-related parts. Since OSD occupies a considerable portion in pagerank execution, ZCOT can reach its best improvement (23.6% and 38.1% w.r.t. Kryo and JSL). Averaged across all applications, ZCOT can reach 4.19× speedup in the write part and 2.95× in the read part over the default Kryo serializer (4.52× and 3.81× speedup for write and read part in JSL). As for GC, ZCOT shows comparable pause time with others. In PR, LR, and TC, the GC time is even shorter than JSL and Kryo. Although ZCOT needs to manage the copied groups (ZCRegions), its coarse-grained collection strategy avoids scanning objects inside ZCRegions. Moreover, ZCOT avoids generating monolithic byte arrays by eliminating the serialization phase, which can mitigate the memory pressure and introduce less frequent GC.

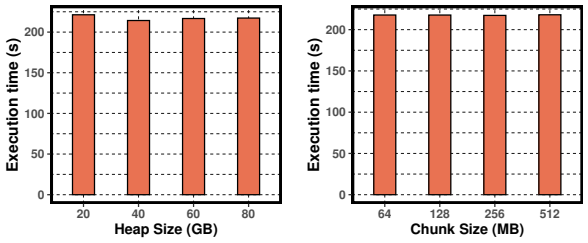
Note that the computation time in ZCOT is somewhat larger than that in JSL and Kryo. This can be explained by two reasons. First, since ZCOT does not compress the object contents during transmission to achieve zero-change, the transferred data size is larger than JSL and Kryo, which leads to larger network overhead (included in the computation part). Second, the data deduplication module makes objects in the same dataset scattered into different virtual address ranges, which may lead to more random memory accesses and cache misses. Nevertheless, the overall performance improvement is satisfying.

Results for deduplication. We have also studied the effect of our data deduplication module. As shown in Table 3, ZCOT can reduce the transferred data size for all four applications, ranging from 8.1% to 53.8%. Even for the non-iterative application (WC), ZCOT is also helpful thanks to its internalization optimization technique. Meanwhile, LR and KM receive smaller savings because they generate many different Double objects in each iteration, which cannot be reused and deduplicated. The result indicates that duplicated transmission is common in data analytics and ZCOT's optimizations are helpful. Note that the number of transferred bytes after deduplication is still much larger than that in Kryo and JSL, since both of them coverts objects in a compact format before transmission. Therefore, it is still preferred to use ZCOT with larger network bandwidth.

	PR	WC	TC	KM	LR
dedup	15.25	4.13	5.03	5.37	5.55
no-dedup	31.64	5.50	10.88	5.86	6.04

Table 3: Average transferred bytes (GB) for Spark executors

Various settings and overhead analysis. We evaluate the performance of ZCOT with various settings on the heap size and the chunk size by using PR as an example. The results in Figure 13 show that ZCOT is not sensitive to different settings and reaches similar performance. We have also studied the overhead of write barriers by running Spark applications atop ZCOT's JVMs (with Kryo serializers) and comparing the performance against vanilla JVMs. The average overhead among all applications is 2.73%, which is much smaller compared with the improvement brought by ZCOT. We also find the average communication overhead with the metadata server is only several milliseconds for each data-processing iteration, which usually lasts for seconds.



(a) Maximum heap size (b) Chunk size

Figure 13: Results for PR under various settings

6.4 Flink

Ease of integration. We have also integrated ZCOT to Flink, another big-data analytics framework. Although Flink adopts its built-in serializer and deserializer for OSD, the integration is not complicated since we only need to replace them with ZCOT's OSD-compatible interfaces and streams.

Evaluation results. We leverage four representative SQL queries in the TPC-H benchmark for evaluation (Q1, Q3, Q6, and Q10) and rely on its built-in generator to create input data (10GB). The configuration is similar to Spark: we launch three workers on different machines for evaluation, but the Java heap for each node is 20GB. Since the read and write phases are overlapped in Flink, we do not break the execution time into parts. The results in Figure 14 show that ZCOT outperforms the built-in serializer in Flink for three out of four queries and leads to 2.3%-22.2% improvement in query execution time. ZCOT does not improve Q6 since it does not involve a reduce operator and the amount of transferred data is limited. It performs the best for Q10 (22.2%) since it reaches 4.40× improvement for the write part and 1.44× for the read part. The speedup is smaller compared with Spark since Flink's built-in serializers are manually optimized for specific data structures (like tuples). Nevertheless, ZCOT still shows better performance than the vanilla version of Flink, which suggests the importance of zero-change transmission mechanism.

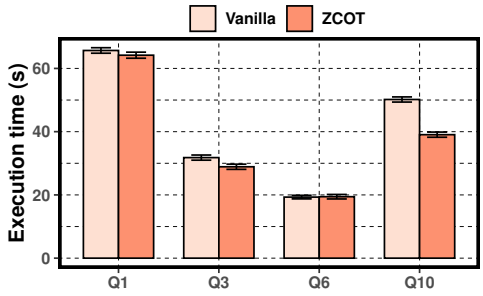


Figure 14: The performance of Flink applications

7 Related work

7.1 OSD optimizations

OSD has become a considerable performance bottleneck especially for large-scale communication-intensive applications. To optimize the time-consuming phases in OSD, prior work such as Kryo [38], Skyway [26], and Naos [39] has refined the transmission data format or leveraged the advances in network hardware technologies. ZCOT instead aims at eliminating the whole OSD process. Apart from software-based techniques, another line of work adopts hardware-based approaches to reduce OSD overhead. Optimus Prime [32] builds a data transformation accelerator (DTA) to improve the OSD throughput for microservices. Cereal [16] co-designs the data transmission format with hardware accelerators to improve the performance and energy efficiency of Spark applications. Morpheus [40] moves the deserialization phase into smart SSDs, while Hgum [46] leverages FPGAs to handle OSD tasks. ZCOT is based on off-the-shelf hardware and thus orthogonal to those hardware-based optimizations.

7.2 Distributed language runtimes

The idea for building a distributed language runtime (e.g., distributed JVMs) has been explored for decades. Java/DSM [43] builds a distributed JVM atop DSM for heterogeneous computing. JESSICA [21, 47] provides a single global thread space and transparently migrates Java threads for load balance. Comet [14] builds a DSM-abstraction for JVMs running on both mobile devices and the cloud and relies on its memory model to achieve effective code offloading. Semeru [41] proposes a universal Java heap abstraction so that a Java application can freely access all memory resources in a memory-disaggregated architecture. Those systems leverage a shared heap to synchronize data among different endpoints, but they do not consider the performance overhead of inter-JVM communication for large applications. XMem [42] enables efficient type-safe object sharing among multiple JVMs on the same physical machine, but it does not consider distributed environments. ZCOT also proposes a distributed runtime design, but it mainly focuses on boosting data transmission among multiple JVMs.

7.3 Runtime optimizations for Java

High-level languages like Java are intensively used in large-scale, distributed applications, which stimulates research interests in runtime optimizations for performance improvement. ITask [11] makes data processing tasks interruptible when facing large memory pressure, which leads to better performance and fewer out-of-memory errors. Yak [27] divides the application execution into epochs and triggers GC when an epoch ends. Broom [12] embraces a region-based design and puts objects with the same lifecycle into the same region for fast reclamation. ScissorGC [18, 19] proposes shadow regions to improve the scalability of full GC phase. Taurus [22, 23] coordinates GC from different JVMs to reach better performance or smaller tail latency. Facade [28] and Deca [36] store massive data objects in off-heap memory to reduce GC pressure, while Gerenuk [24] enables speculative execution on serialized data to reduce both memory footprint and GC overhead. ZCOT focuses on eliminating the OSD process and duplicated object transmission, and it also collects objects by coordinating with the metadata server.

8 Conclusion

This work introduces ZCOT, which aims to eliminate the object serialization/deserialization phase in data exchange among language runtimes (like JVMs). ZCOT provides an exchange space where objects are interpretable for all JVMs, which removes the need for any data transformation during object transmission. It also uncovers the duplicated object transmission problem and provides a corresponding deduplication mechanism. The evaluation shows that ZCOT can significantly improve the performance of object transmission.

9 Acknowledgement

We sincerely thank our anonymous shepherd and reviewers for their insightful suggestions. This work is supported in part by the National Natural Science Foundation of China (No. 62172272, 61925206, 62132014). Binyu Zang (byzang@sjtu.edu.cn) is the corresponding author.

A Artifact Appendix

Abstract

ZCOT, or Zero-Change Object Transmission, is proposed to optimize data exchange among multiple Java virtual machines (JVMs) in a distributed environment. Instead of sending and

receiving data with the costly object serialization/deserialization (OSD) phase, ZCOT allows JVMs to directly communicate with Java objects, which significantly improves the data exchange time, especially for applications like big data analytics.

Scope

This artifact (including binaries, source code, documents, and scripts) is used to conduct the main experiments in ZCOT, which consists of the following two parts:

- **Micro-benchmark performance**. The result should show that ZCOT outperforms recent OSD optimizations (Skyway [26] and Naos [39]) and state-of-the-art OSD libraries (Kryo [38] and JSL) for most data structures used in Naos' microbenchmark.
- **Spark performance**. The result should show that ZCOT outperforms Kryo and JSL-based Spark applications in both data exchange and task execution.

Note that we only report numbers evaluated on our machines, so the results might be different with various hardware configurations.

Contents

We pack all related files into a zipped one, which contains the following contents.

- **README.** A file containing instructions for artifact evaluation.
- **ZCOT-jdk.** The source code of a modified OpenJDK to support ZCOT.
- **Meta-server.** The source code of the metadata server used in ZCOT.
- **Micro.** Scripts and jars used for the micro-benchmark.
- **Spark.** Since the code size of Spark is quite large, we provide an executable binary for Spark, which is slightly modified to evaluate ZCOT.
- **Naos-jdk.** A slightly modified version of Naos' OpenJDK to compare with ZCOT.

Hosting

Currently our code is not ready for open-source. Nevertheless, you can contact us via mingyuwu@sjtu.edu.cn to obtain the artifact.

Requirements

Hardware requirements. We evaluate ZCOT on four nodes connected by 100 Gbit/s Mellanox ConnectX-5 NICs. The NIC bandwidth has a significant impact on ZCOT's performance.

Software requirements. The operating system used in our machines is Ubuntu 16.04.2, but higher versions are also acceptable. Note that huge pages should be enabled to run ZCOT. Dependencies for installing OpenJDK have been listed in the README file.

References

[1] Lada A Adamic and Natalie Glance. The Political Blogosphere and the 2004 US Election: Divided they Blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, pages 36–43. ACM, 2005.

[2] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[3] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[4] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 44–54, 2006.

[5] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5(1):1–9, 2014.

[6] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[7] John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and performance of munin. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 152–164, 1991.

[8] David Detlefs, Christine H. Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, pages 37–48. ACM, 2004.

[9] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[10] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[11] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 394–409, 2015.

[12] Ionel Gog, Jana Giceva, Malte Schwarzkopf, Kapil Vaswani, Dimitrios Vytiniotis, Ganesan Ramalingam, Manuel Costa, Derek G Murray, Steven Hand, and Michael Isard. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[13] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 599–613, 2014.

[14] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. {COMET}: Code offload by migrating execution transparently. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 93–106, 2012.

[15] Apache Hadoop. Hadoop, 2009.

[16] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W Lee. A specialized architecture for object serialization with applications to big data analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 322–334. IEEE, 2020.

[17] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Statistical properties of community structure in large social and information networks. In *Proc. Int. World Wide Web Conf.*, pages 695–704, 2008.

[18] Haoyu Li, Mingyu Wu, and Haibo Chen. Analysis and optimizations of java full garbage collection. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, pages 1–7, 2018.

[19] Haoyu Li, Mingyu Wu, Binyu Zang, and Haibo Chen. Scissorgc: scalable and efficient compaction for java full garbage collection. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 108–121, 2019.

[20] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[21] Matchy JM Ma, Cho-Li Wang, and Francis CM Lau. Jessica: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.

[22] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. *Acm SIGPLAN Notices*, 51(4):457–471, 2016.

[23] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS {XV})*, 2015.

[24] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. Gerenuk: thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 538–553, 2019.

[25] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 291–305, 2015.

[26] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 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, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pages 56–69. ACM, 2018.

[27] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 349–365, 2016.

[28] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. *ACM SIGARCH Computer Architecture News*, 43(1):675–690, 2015.

[29] OpenJDK. Jep 248: Make g1 the default garbage collector, 2017.

[30] OpenJDK. Jep 310: Application class-data sharing, 2018.

[31] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[32] Arash Pourhabibi, Siddharth Gupta, Hussein Kassir, Mark Sutherland, Zilu Tian, Mario Paulo Drumond, Babak Falsafi, and Christoph Koch. Optimus prime: Accelerating data transformation in servers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1203–1216, 2020.

[33] Daniel J Scales and Monica S Lam. The design and evaluation of a shared object system for distributed memory machines. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 9–es, 1994.

[34] Ioannis Schoinas, Babak Falsafi, Alvin R Lebeck, Steven K Reinhardt, James R Larus, and David A Wood. Fine-grain access control for distributed shared memory. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 297–306, 1994.

[35] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 323–337, 2017.

[36] Xuanhua Shi, Zhixiang Ke, Yongluan Zhou, Hai Jin, Lu Lu, Xiong Zhang, Ligang He, Zhenyu Hu, and Fei Wang. Deca: a garbage collection optimizer for in-memory data processing. *ACM Transactions on Computer Systems (TOCS)*, 36(1):1–47, 2019.

[37] Eishay Smith. Jvm-serializers, 2020.

[38] Esoteric Software. Kryo, 2021.

[39] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. Naos: Serialization-free RDMA networking in java. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 1–14. USENIX Association, July 2021.

[40] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogeneous computing. *ACM SIGARCH Computer Architecture News*, 44(3):53–65, 2016.

[41] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 261–280, 2020.

[42] Michal Wegiel and Chandra Krintz. Xmem: type-safe, transparent, shared memory for cross-runtime communication and coordination. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 327–338, 2008.

[43] Weimin Yu and Alan Cox. Java/dsm: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.

[44] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10, Boston, MA, USA, June 22, 2010*, page 95. USENIX Association, 2010.

[45] Matthew J Zekauskas, Wayne A Sawdon, and Brian N Bershad. Software write detection for a distributed shared memory. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, pages 8–es, 1994.

[46] Sizhuo Zhang, Hari Angepat, and Derek Chiou. Hgum: Messaging framework for hardware accelerators. In *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8. IEEE, 2017.

[47] Wenzhang Zhu, Cho-Li Wang, and Francis CM Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 381–388. IEEE, 2002.