A Specialized Architecture for Object Serialization with Applications to Big Data Analytics

Jaeyoung Jang† Sung Jun Jung‡ Sunmin Jeong‡ Jun Heo‡
Hoon Shin‡ Tae Jun Ham‡ Jae W. Lee‡
†Sungkyunkwan University ‡Seoul National University
jaey86@skku.edu {miguel92, sunnyday0208, j.heo, zmqp, taejunham, jaewlee}@snu.ac.kr

***Abstract*—Object serialization and deserialization (S/D) is an essential feature for efficient communication between distributed computing nodes with potentially non-uniform execution environments. S/D operations are widely used in big data analytics frameworks for remote procedure calls and massive data transfers like shuffles. However, frequent S/D operations incur significant performance and energy overheads as they must traverse and process a large object graph. Prior approaches improve S/D throughput by effectively hiding disk or network I/O latency with computation, increasing compression ratio, and/or application-specific customization. However, inherent dependencies in the existing (de)serialization formats and algorithms eventually become the major performance bottleneck. Thus, we propose *Cereal*, a specialized hardware accelerator for memory object serialization. By co-designing the serialization format with hardware architecture, Cereal effectively utilizes abundant parallelism in the S/D process to deliver high throughput. Cereal also employs an efficient object packing scheme to compress metadata such as object reference offsets and a space-efficient bitmap representation for the object layout. Our evaluation of Cereal using both a cycle-level simulator and synthesizable Chisel RTL demonstrates that Cereal delivers 43.4× higher average S/D throughput than 88 other S/D libraries on Java Serialization Benchmark Suite. For six Spark applications Cereal achieves 7.97× and 4.81× speedups on average for S/D operations over Java built-in serializer and Kryo, respectively, while saving S/D energy by 227.75× and 136.28×.**

***Index Terms*—Object serialization, Domain-specific architecture, Data analytics, Apache Spark, Hardware-software co-design**

I. Introduction

Object serialization and deserialization (S/D) is a fundamental operation in modern big data analytics for portable, loseless communication between distributed computing nodes. For efficient inter-node data transfers, the sender node first *serializes* a sea of objects into a stream of bytes; the receiver node then reconstructs the objects from the serialized byte stream. Massive data manipulation operations in today's big data analytics frameworks [3]–[6], [11], [55], such as map/reduce, shuffle, and join, heavily utilize S/D operations.

S/D operations incur substantial performance overhead to modern data analytics frameworks running on managed runtime environments like Java and Scala. Recent studies report that serialization overhead accounts for some 30% of the total execution time in popular big data analytics frameworks [40], [41]. According to Google, serialization, along with other low-level operations like memory allocation, compression, and network stack processing, consumes 20-25% of total CPU cycles as "datacenter tax" [9].

Unfortunately, existing S/D libraries still have relatively low throughput for various reasons. For example, the Java built-in serializer (Java S/D) embeds type strings (e.g., class and field names) in a string format to require expensive string matching operations for type resolution during deserialization and bloat the serialized stream. Other S/D libraries improve throughput by adopting more compact representations for types (i.e., integer class numbering) [34], [41], [51], hand-optimized serialization functions [34], [48], compilation-based approach to obviate the need for extracting field information at runtime [45], direct operations at backing array to reduce the encoding cost [12], and sending a raw object graph while overlapping computation with communication [41].

Although these proposals address some of the inefficiencies, there is still substantial room for improvement. For example, our experiments demonstrate that S/D operations still take about 28% of total execution time on average (and up to 83.4%) for six Spark applications even if we use Kryo [34], a highly optimized S/D library for Java (details in Section III).

A serialization operation requires a recursive traversal of object graph from the top-level object being serialized as all the referenced child objects should be serialized together. Furthermore, it invokes a large number of function calls to extract individual fields for each object (e.g., reflection [30]). Existing S/D libraries are inefficient when handling those operations, partly due to their limited use of parallelism within the S/D process. Even with highly parallelized S/D algorithms, their performance on general-purpose CPUs is bottlenecked by limited microarchitectural resources such as instruction window and load-store queues. Even worse, S/D operations are characterized by frequent indirect loads, random memory accesses, large-volume memory copies, and little data reuse, which make it difficult to handle them efficiently on CPUs.

To address these limitations, we propose Cereal, a hardware accelerator for S/D operations. We carefully co-design the serialization format with the hardware architecture to effectively leverage multiple levels of parallelism during the S/D process. This multi-level parallelism harnesses massive memory-level

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00036

parallelism (MLP) for Cereal to achieve high memory bandwidth utilization, and hence high S/D throughput. To keep the space overhead for metadata low, we also propose an efficient object packing scheme to compress object reference offsets by preserving only significant bits (e.g., discarding leading zeros) and a bitmap representation for the object layout. In summary, this paper makes the following contributions:

- We propose a new serialization format to better expose coarse-grained parallelism in the S/D process, while maintaining compact representation of metadata via efficient object packing.
- We architect and implement Cereal, a specialized architecture for S/D operations, which is co-designed with the new serialization format.
- We integrate Cereal with Apache Spark on HotSpot JVM to validate its functionality and evaluate its performance using real-world data analytics workloads.
- We demonstrate that Cereal achieves significant speedups and energy savings over state-of-the-art software S/D implementations on Java Serialization Benchmark Suite and Spark applications.

II. Background

Object Serialization. Serialization is a process of converting the objects into a stream of bytes and deserialization is a reverse process of reconstructing the original objects from the stream of bytes. S/D is a fundamental operation in data analytics frameworks for massive data communication and manipulation (e.g., for join and shuffle), I/O management in a distributed file system, and messaging for remote procedure calls [5], [18], [55]. There are a variety of S/D implementations depending on use case, data types, and necessity for supporting references (pointer objects) [34], [45]. The serialization process becomes more complicated with references as a recursive object graph traversal is required to identify all child objects transitively pointed to by the top-level object. Indirect loads to fetch information about the layout of an object make this process even slower. Recently, S/D has been identified as a major performance bottleneck in data analytics frameworks [39], [41], [52]. Thus, this paper focuses on optimizing S/D operations in Java as many such frameworks build on JVM-based execution environments.

Java Object Layout. Figure 1(a) illustrates the memory layout of Java objects in HotSpot [27], the most widely used production JVM, using an example code snippet. An object has a fixed-length header, followed by fields that hold the values and references of the object. The header is 16B in length and composed of a mark word (8B) and a klass pointer (8B). The mark word includes an identity hash code (31 bits), a synchronization state (3 bits), GC state bits (6 bits), and 25 unused bits. A klass pointer points to type metadata, called *type descriptor*, which contains the object layout as well as the total object size. The object layout includes the offsets of all the references in the object, which a serializer utilizes to locate them.

Java built-in Serializer. Java built-in serializer [28] (Java S/D) is the baseline serializer provided by Java. When an object

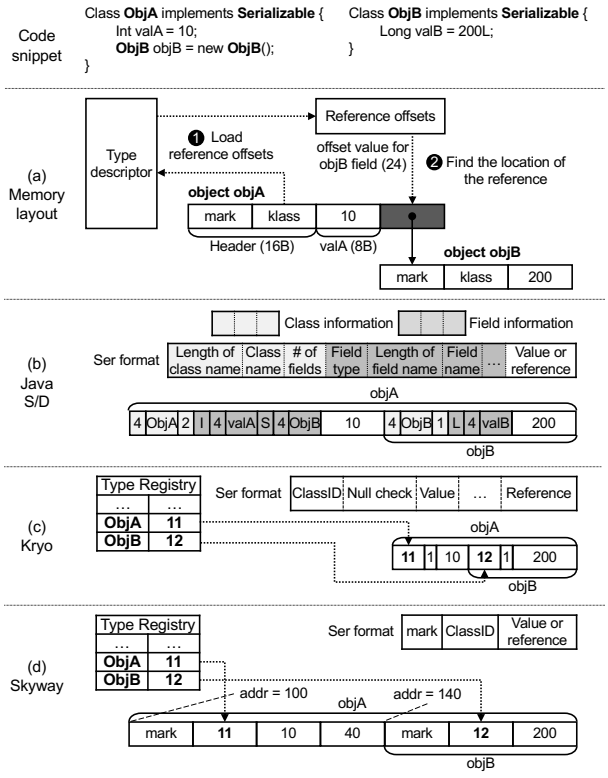

Fig. 1. Serialization format of three Java serializers

is serialized, the serialized stream contains the field data of the object and metadata for its class and the classes of all the objects it recursively references to. Figure 1(b) shows an example of a serialized stream of Java S/D. Java S/D stores the names of all classes and fields as string types. In addition, Java S/D stores the length of the name, the metadata of the fields (number of fields, field types), and other metadata associated with serialization. As a result, the serialized stream contains a large amount of metadata. For reference fields, Java S/D uses `Package java.lang.reflect` [30] to get the content of each reference field and insert it to the serialized stream. For example, `Field getField (String name)` in the package returns a `Field` object of the class specified by `name` and `void set (Object obj, Object value)` sets the target field with the input value. Generally, the methods in this package are a well-known source of computational overhead in Java S/D as they perform string lookups with no given type information.

Kryo Serializer. Kryo [34] is one of the most popular third-party serialization libraries for Java. Kryo addresses the limitations of Java S/D with manual registration of classes. Figure 1(c) shows a serialized stream in Kryo. Unlike Java S/D, Kryo represents all classes by just using a 4B ClassID (integer class numbering). All field types and primitive types (e.g., Int, Long) are also registered to get assigned distinct ClassIDs. By registering classes and types (type registration), Kryo reduces the overhead of storing type names as strings (plus additional metadata) in Java S/D. And, there is a Null check (1B) field to mark an object with no fields. Overall,

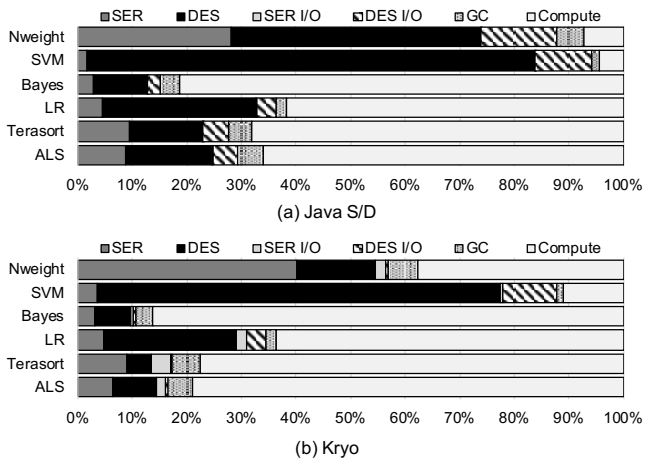

Fig. 2. S/D overhead in Spark applications with different serializer libraries

there is much less metadata than Java S/D as Kryo only stores the actual data. Kryo also uses an optimized library [49] to get class/field information (serialization) or set field values (deserialization), instead of the original reflect package in Java S/D. These optimizations reduce the time and space overhead drastically, but it requires users to manually register the class of every single object being serialized. The same type registry must be used for deserialization. Therefore, Kryo requires additional user effort for performance.

Skyway Serializer. Skyway [41] is a state-of-the-art serializer specialized for data shuffling between nodes in a distributed system. Skyway proposes a way to transfer an object by a simple memory copy to reduce the computational overhead of disassembling and reassembling it. This eliminates the cost of accessing any field or type imposed on existing serializers. Skyway also uses an integer type ID (like Kryo) for identifying the class of the object instead of type string. Skyway also keeps a global *type registry*, which maps every type string to its unique type ID (like Kryo). Unlike Kryo, however, Skyway can reduce manual type registration effort by utilizing an automatic type registration system. During serialization, Skyway converts the absolute address of the object to a relative address, eliminating the overhead of invoking methods in the reflect package for adjusting references. Skyway reduces the user effort as well as computational overhead, to report a 16% speedup over Kryo on average [34]. However, the sequential reference adjustment of objects at the receiver is still inefficient. Also, the size of the serialized byte is larger than Kryo's size as the object is serialized as is including reference fields and headers.

III. Analysis and Motivation

S/D Overhead. To quantify the S/D overhead in data analytics applications, we analyze the performance of six data analytics applications on Apache Spark running on HotSpot JVM [27]. Spark extensively utilizes S/D operations for 1) input/output management in a distributed file system (e.g., HDFS [50]), 2) communication between the master and worker nodes, 3) data manipulation across the nodes such as Shuffle, and 4) software caching and data spill for efficient memory management. These use cases are common in other large-scale data analytics frameworks. We select six S/D-intensive applications from Intel HiBench [24] and execute them using two Spark executor instances on Intel i7-7820X Processors [25]. More detailed experimental setup is available in Section VI-A. We breakdown the total application time into 1) computation time, 2) GC time, 3) I/O time, and 4) S/D time.

Figure 2 shows a runtime breakdown of the six applications using Java S/D (Figure 2(a)) and Kryo (Figure 2(b)), which are used by Spark. Our analysis shows that the overhead of S/D operations is substantial, accounting for an average of 39.5% of total execution time for Java S/D (and up to 90.9% with SVM), and 28.3% for Kryo (and up to 83.4% with SVM). As discussed in Section II, Java S/D embeds a string for every class and field into the serialized stream, which leads to substantial I/O overhead (up to 13.9% for NWeight). Even worse, the excessive use of reflection functions for type resolution, which involve expensive string match operations, makes S/D process very slow. In contrast, Kryo employs integer class numbering, minimizing type metadata overhead. Also, Kryo does not need to use reflection for type resolution but utilizes a custom reflection library for (re)storing values, to yield substantially lower runtime overhead than Java S/D. However, even with Kryo, the S/D overhead is still substantial, taking nearly 30% of total execution time on average.

To understand the S/D performance better, we use three data structure benchmarks commonly used in many applications, Tree, List, and Graph, and measure the IPC, cache miss rate, and bandwidth utilization of S/D process using Linux Perf tool [14]. For each benchmark, we scale the number of children nodes and depth (Tree), the size (List), and the number of connected edges (Graph). More details of these benchmarks are covered in Section VI-A.

There are several observations with the results. First, Figure 3(a) shows relatively low average IPCs for both Java S/D (1.01) and Kryo (0.96). Second, Figure 3(b) demonstrates that the degree of (temporal) locality is pretty low (i.e., cache miss rate is high). Finally, even with a high cache miss rate, both fail to fully utilize memory bandwidth due to limited parallelism in the S/D process (Figure 3(c)).

Motivation for Specialized Hardware. In an abstract form, the key operations in the S/D process include (i) object graph traversal for serialization and (ii) copying the contents of the visited object into another space. These operations are handled poorly in general-purpose CPUs for the following reasons. First, object graph traversal requires many indirect loads to visit its child objects. Such indirect loads inherently generate lots of random memory accesses, which significantly degrade the cache performance. A copying operation, which touches large memory regions, evicts cache lines to increase cache miss rate. Moreover, the object graph has a low degree of parallelism without exploiting intra-object parallelism, and CPUs cannot fully utilize memory-level parallelism (MLP) due to limited instruction window and load/store queue size, hence yielding low memory bandwidth utilization (Figure 3(c)). As a result, Kryo improves the Java S/D by removing several

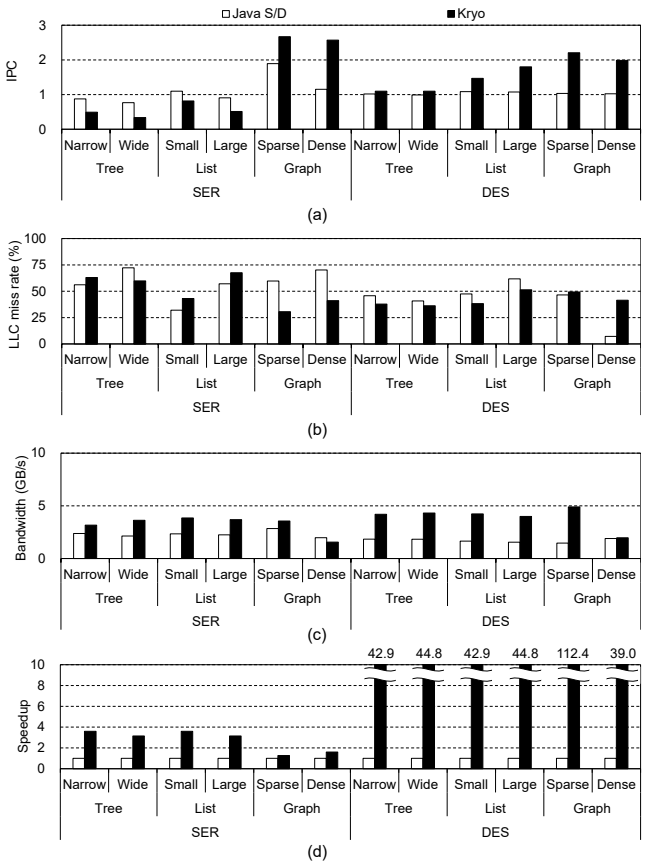

Fig. 3. S/D process analysis with microbenchmark (a) IPC (b) LLC miss rate (c) bandwidth (d) speedup

inefficiencies like size reduction and reducing the use of expensive reflective functions, but the performance gain is marginal (Figure 3(d)). Moreover, many big data applications are compute-intensive [41], [44], [54] for user computation to contend with S/D operations for CPU cycles. Thus, a specialized architecture designed to extract parallelism within the S/D process can substantially improve the S/D performance and reduce the CPU load.

IV. Serialization Format for Cereal

A. Serialization Stream Layout

Serilization Format. Figure 4 presents the baseline serialization format of Cereal (before object packing) with an illustrative example. Figure 4(a) shows an object graph with four objects with `objA` as the root object. It also shows how the layout bitmap marks the location of each reference field. Since all fields in a JVM object are 8B aligned, one bit of the layout bitmap corresponds to an 8B in the heap. If the object contains a reference, the bit at that location is set to 1, and the size of the object can be obtained by multiplying the layout bitmap size in bits by 8B. Figure 4(b) illustrates the serialized format of Cereal composed of three structures: value array, reference array, and layout bitmaps (including object graph size). These structures are fed into the object packer (Section IV-B) to

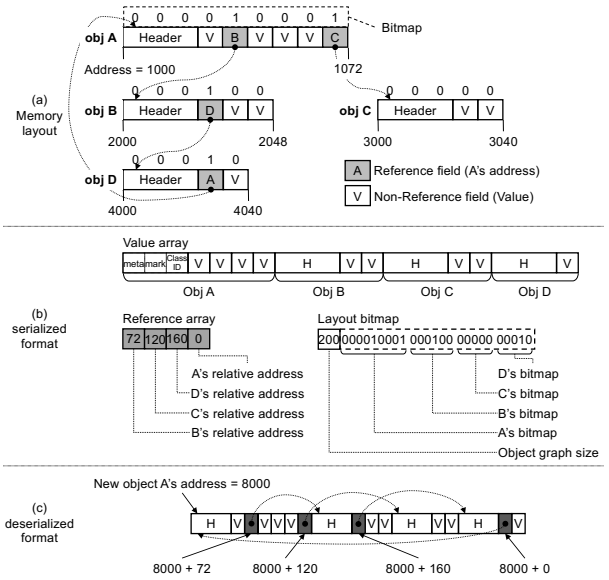

Fig. 4. Illustration of Cereal's baseline serialization format: (a) original memory layout (b) serialized format (c) deserialized format

reduce the size of the serialized stream. Figure 4(c) shows how the four objects are reconstructed at the base address 8000 from the serialized stream.

A notable difference of Cereal from Skyway is the way to store references and values and the aforementioned object packing scheme. During deserialization, Skyway needs to adjust the references sequentially using relative addresses. However, because Cereal stores values and references separately, it can handle value copying and reference adjustment in parallel. Since the layout bitmap contains the location of each reference, the operation on the value and the reference can be performed independently. This enables Cereal to exploit a greater degree of parallelism, hence achieving higher throughput using specialized hardware.

Space Overhead Analysis of the Baseline Format. The overhead of the layout bitmap itself is proportional to the object size. Since one bit of the layout bitmap is responsible for 8B, the size of the layout bitmap is 1.56% of the object size. Also, when the layout bitmap is stored as a byte stream, it is necessary to tell the boundary between two adjacent objects in the layout bitmap. The easiest way to distinguish the layout bitmap boundaries would be to store the layout bitmap length. However, since the space overhead of layout bitmap length is 8B per each object, if the object size is small, the layout bitmap length can be much larger than the data size stored. Alternatively, a layout bitmap could be stored with padding using fixed bucket sizes (4B, 8B, and so on), but, depending on the range of layout bitmap values, the space overhead due to padding can be significant. For references, the value (relative address) is the difference between the top-level object address and the referenced object address in the deserialized stream. Typically, it takes much fewer bits than 8B (long type) to represent the relative address. Finally, there is a 4B overhead representing the sum of object sizes (before serialization) that

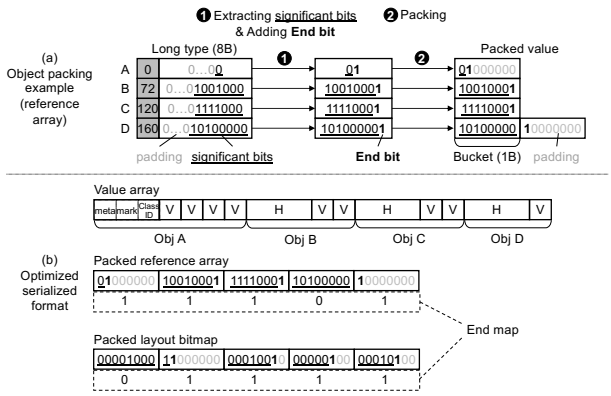

Fig. 5. The optimized serialized format in Cereal (a) object packing example for the reference array (b) the optimized serialized format

are included in this serialized format. Using the serialization format in this section as is can result in large space overhead than the existing schemes. To compensate for this overhead, we propose an *object packing scheme* to compress both the layout bitmap and references in the following section.

B. Object Packing Scheme

Cereal uses an optimized packing scheme to eliminate the redundant metadata (layout bitmap length and excessive padding) and reduce the overall size of serialized bytes. Figure 5 shows the optimized serialization format of Cereal with object packing. Figure 5(a) illustrates a step-by-step process of packing four references. In Step 1, the object packer extracts only significant bits from the reference bits (i.e., dropping leading zeros) and add an end bit (**1** in bold). In Step 2, it puts the bit string (with optional padding zeros) to 1B buckets to make it byte-aligned. Using the end map to indicate the boundaries of the packed references (shown in Figure 5(b)) occupies much less space than storing the layout bitmap length for each object or using a static bucket size. When deserializing the serialized stream Cereal can process multiple values and references by inspecting the end map. Thus, we apply this object packing scheme to both the layout bitmap and references. In contrast, the value array (with headers and values) is stored as in the baseline format (Section IV-A) without applying this scheme.

V. Cereal Architecture

A. Overview

Figure 6 shows an overview of Cereal architecture. The host issues a serialization or deserialization request through the simple software interface (explained in the paragraph below). This request is then buffered in Cereal's command queue. The request scheduler inspects the request at the head of the queue and finds the available serialization unit (SU) or deserialization unit (DU). If found, it forwards a request to the corresponding unit, which then starts the execution. Our Cereal interacts with the memory system directly without going through the cache hierarchy. Instead, Cereal has its own memory access interface

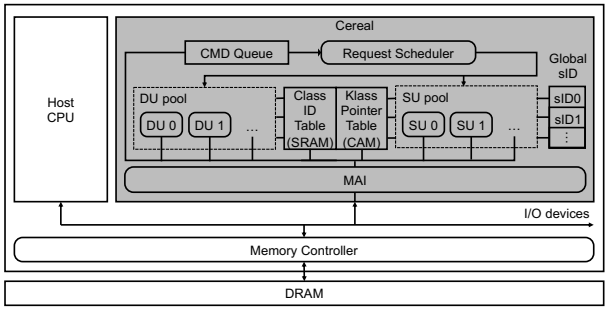

Fig. 6. Cereal architecture overview

(MAI), which performs basic functionalities such as requesting coalescing (as in conventional MSHRs).

Software Interface. Cereal shares the similar serialization, deserialization interfaces with other popular serializers such as Kryo [34] and Skyway [41], which essentially makes replacing a conventional serializer/deserializer to a Cereal serializer/deserializer a trivial work. `Initialize` is simply called at the beginning of the application to secure a certain amount of memory region for Cereal's S/D process. `RegisterClass(Class Type)` registers the specified class so that the class can be serialized and deserialized with Cereal. This function should be called once for every type that needs to be serialized or deserialized. Note that this function is effectively identical to the Kryo serializer's function having the same name. To serialize an object `obj`, the user needs to use `WriteObject(ObjectOutputStream oos, Object obj)`. Here, the `ObjectOutputStream oos` is often connected to the `FileStream` for the output file. To deserialize an object, the user should use `ReadObject(ObjectInputStream ois)` which returns the deserialized object. Here, `ObjectInputStream ois` provides the raw byte sequences from the associated `FileStream`.

Memory Access Interface (MAI). MAI is a memory access interface for our accelerator. The main role of the MAI is to correctly return the response to requests from various modules of Cereal. The MAI contains an associative memory with 64 entries. Each entry has the memory address for this outstanding request, the list of modules that it should send the responses to, and the associated metadata related to this memory request. In addition, the MAI also contains multiple reorder buffer which reorders the unordered memory responses so that the requester can receive these responses in order of the original requests. Finally, the MAI also supports an atomic read-modify-write (within an accelerator) so that a module within this accelerator can safely update a value without a race. For this atomic operation, the MAI contains another associative memory structure, which buffers outstanding read-modify-write operations.

B. Serialization Unit (SU)

Serialization Flow. Figure 7 shows the block diagram of our serialization unit (SU). At a high level, this unit consists of four components: header manager, object metadata manager, object handler, and reference array writer. Each of these

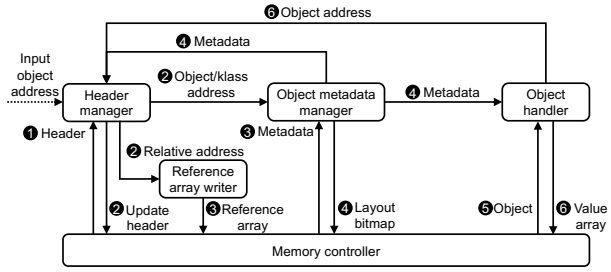

Fig. 7. The data flow of Serialization Unit

components performs a different functionality. Specifically, the header manager is responsible for inspecting object headers and updating the headers. The object metadata manager receives the object header from the header manager, fetches all the necessary object metadata from memory, and then generates a packed layout bitmap in the Cereal serialization format (Figure 4 and 5). The object handler receives the object layout information from the object metadata manager and fetches the object from memory. Using this layout information, the object handler can distinguish the references and values within this object. Then, it sends the references (i.e., addresses of the referenced objects) to the header manager, and updates the value array in the memory. Lastly, the reference array writer receives object relative addresses from the header manager, performs packing (Section IV-B), and outputs the packed reference array.

Serialization starts when the object enters the header manager. ❶ Every time an object in the object graph is visited, the header manager needs to check its header to determine if it is visited. ❷ If the object has not been visited, the header manager sets the object's relative address as the sum of the already serialized object sizes, sends it to the reference array writer, and updates the header as visited by storing the relative address in the object header. At the same time, the header manager also sends the object/`klass` address to the object metadata manager. If the object is already visited, which implies that the object is already serialized, the header manager just gets the relative address from the object header, sends it to the reference array writer, and then gets the new object address. Once the object metadata manager gets the object/`klass` address from the header manager, serialization of the current object starts. ❸ The object metadata manager fetches the object's metadata to identify which of 8-byte fields following the object header are references/values. ❹ Once the metadata is loaded, the object metadata manager generates the layout bitmaps (see Figure 4) and stores them in memory. At the same time, this metadata passed to the object handler. ❺ The object handler loads the object from memory and ❻ gathers value fields to update the value array in memory or ❻ passes the references (i.e., addresses of the referenced objects) to the header manager. Below, we discuss the operations of each component in detail.

Header Manager. Header manager takes object addresses as its inputs. For each object provided, it first reads the header of the object and then inspects it. By inspecting the header, the header manager identifies whether this object was already visited or not. If it is not, it means that this object needs to be serialized. In this case, the loaded header is passed to the object metadata manager, which will then utilize this header to fetch various object metadata. Another job of the header manager is to update the object header. First, it needs to mark that this object is visited if this is the first time to trace this object. Second, the header manager needs to record the relative address of the current object to the header (see Figure 4) . This value equals the total serialized object size so far and indicates the relative address of the currently serialized objects in the deserialized format. When the object metadata manager returns the object size information, the header updates a counter that tracks this value. The last responsibility of the header manager is to pass the relative addresses of the current object (i.e., the relative address in the deserialized format) to the reference array writer. As explained, if the object was not visited previously, this address is equal to the total serialized objects size. In case the object was previously visited, the relative address is already recorded in the header and thus can be extracted from it. Note that the header manager cannot process another object until it receives the object size from the object metadata manager and update its counter.

Object Metadata Manager. Object metadata manager retrieves the object header from the header manager and fetches relevant object metadata (e.g., object layout, object size) from memory. Then, it passes the object layout fetched from memory to the object handler. In addition, it generates the packed layout bitmap utilizing the object layout information. Finally, it also passes the fetched object size information to the header manager so that the header manager can update the total size of the objects serialized so far.

Object Handler. Object handler receives the object metadata (including object layout) from the object metadata manager. Once receiving the object metadata, the object handler loads the object from memory. When the object is loaded, the values (including the header) or the references are extracted. If a reference is extracted, the object handler passes it to the header manager. Here, it is important to pass the references within the object to the header manager in the original order. To guarantee that the memory accesses return in the requested order, we utilize the reorder buffers in the memory access interface (MAI). When a value is extracted, the object handler first checks if it is a header. If so, the `klass` address field is translated to the class ID by performing a lookup on its content-addressable memory (CAM) structure (Klass Pointer Table, 4KB), whose contents are filled when `RegisterClass` API is called in the software. These headers and values are buffered and then stored in memory at 64B granularity when the buffer becomes full. The stored data in memory is the value array of Cereal serialization format (Figure 4).

Reference Array Writer. Reference array writer is a module that outputs the packed reference array. It receives the object relative addresses from each object handled by the header manager and then performs the packing to generate the packed reference array.

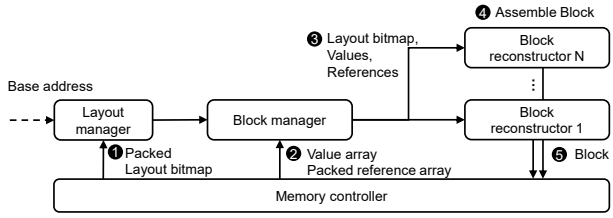

Fig. 8. The data flow of Deserialization Unit

C. Deserialization Unit (DU)

Deserialization Flow. Figure 8 shows the block diagram of our deserialization unit (DU). This unit consists of a layout manager, a block manager, and multiple block reconstructors. ❶ Here, the layout manager is a module that fetches the layout bitmap, unpacks it, and counts the number of 0s and 1s in the layout bitmap and then identifies how many values/headers and references that a single block (64B) has. Then, it sends the 8-bit layout bitmap corresponding to a single block along with the number of 1s and 0s to the block manager. ❷ The block manager is a module that internally prefetches i) the value array, and ii) the packed reference array. Once fetched, the packed reference array is unpacked immediately. When the block manager receives the 8-bit layout bitmap chunk, along with the number of value/headers and references, it retrieves i) the exact amount of values (and headers) in the block from its internal value array loader and ii) the exact amount of references in the block from its internal reference array loader. ❸ This block manager dispatches this loaded layout bitmap, values, and references for a single block to one of the available block reconstructors. ❹ Finally, the block reconstructor scans the 8-bit layout bitmap and reconstructs them to 64-byte output according to the layout bitmap. ❺ After reconstruction, block reconstructor writes the output to its destination address.

Layout Manager. Layout manager module eagerly fetches the packed layout bitmap, unpacks it, and counts the number of 0s and 1s in a single 64B block. It has a hardware module named layout bitmap loader, which eagerly prefetches the packed layout bitmap. It maintains a set amount of internal buffer and eagerly issues a load request to the memory whenever this buffer is empty. This loaded layout bitmap is then fed to the custom unpacking module, which unpacks the layout bitmap. Finally, the custom logic counts the number of 0s and 1s in a single cycle and passes this information and the unpacked layout for a single block to the next module, block manager.

Block Manager. Block manager module also includes two modules that eagerly prefetch the value array and the packed reference array, respectively. As in the layout manager, the loaded packed reference array is unpacked with the custom hardware module, and the unpacked result is stored in the internal buffer. Whenever the layout manager passes the 8-bit layout bitmap and the number of 0s and 1s, it retrieves the number of values (i.e., equal to the number of 0s) from the value array loader and retrieves the number of references (i.e., equal to the number of 1s) from the reference array loader. Then, it sends i) 8-bit layout bitmap, ii) retrieved values, and iii) retrieved references to one of the available block constructors. In addition to those three values, it also maintains the internal counter that counts the number of 64B blocks that it processed to generate the destination address where the block reconstructor should write its outputs.

Block Reconstructor. Block reconstructor can start reconstructing the output when the block manager assigns the block to it. During the time it is reconstructing a block, it marks itself busy and this prevents the block manager from issuing another work to this block reconstructor. With the 8-bit layout bitmaps, the block reconstructor simply puts the values to where the corresponding layout bitmap value is zero while putting the references to where the corresponding layout bitmap value is one. Additionally, the block reconstructor identifies whether it has a header field that represents the class ID using a 8-bit layout bitmap chunk along with its corresponding end map. If so, it needs to translate that class ID to a `klass` address by checking the appropriate index from a SRAM structure (Class ID Table, 2KB) which has class ID to `klass` address mapping. As in the case of serialization, this SRAM is populated when the `RegisterClass` API is called in the software. Once finished, it sends the write request of the resulting 64B block to the provided destination address and marks it not busy.

D. Parallelism in Cereal

Cereal exploits three different forms of parallelism.

- *Operation-level parallelism*: This is the parallelism across S/D operations. With multiple Cereal SUs and DUs, multiple S/D operations can be executed in parallel. This form of parallelism is essentially the task-level parallelism exploited by the multi-threaded execution of S/D operations on multiple CPU cores.
- *Object-level parallelism*: This is the parallelism across objects within an object graph. Our SU exploits pipeline parallelism to process multiple objects within the same object graph. Each block in Figure 7 processes different objects within the object graph.
- *Block-level parallelism*: The is the parallelism across multiple unpacked blocks in DU. Decoupled value/reference array and layout bitmap enable deserialization to be processed in parallel with multiple block reconstructors.

Existing libraries such as Java S/D and Kryo also utilize operation-level parallelism with multiple CPU cores (i.e., multi-threaded execution). However, those libraries cannot easily exploit neither object-level parallelism nor block-level parallelism. This is because conventional CPU lacks the ability to extract such parallelism due to the limited resources (i.e., instruction window, load/store queue size). Furthermore, the existing S/D libraries' serialization format also limits the amount of parallelism. In contrast, Cereal leverages hardware specialization with tightly-coupled software format optimization to fully exploit abundant parallelism within these operations.

E. Implementation Details

Header Extension. Cereal requires a certain amount of per-object metadata for its serialization process. However, assigning a separate memory address space for these per-object metadata leads to a performance degradation since the Cereal serializer cannot easily retrieve the address for the specific object's metadata. To avoid this performance degradation, Cereal extends the JVM so that all potentially serializable objects (whose type implements Java Serializable interface) allocate an additional 8B in its header so that the Cereal serializer can utilize that space for metadata necessary during the serialization. By doing so, these metadata can easily be retrieved with a simple relative address calculation on an object address. These metadata include i) metadata to track visited objects, ii) metadata to support shared objects, and iii) metadata to record relative address for the already serialized objects.

Tracking Visited (Traversed) Objects. A conventional way of tracking the visited object for the object graph traversal is maintaining a single bit on the object header, then mark it when it is visited. However, that approach requires resetting the visited bit to zero every time the traversal ends. In our case, this means that all object's visited bits need to be cleared at the end of the serialization. However, such overhead can potentially negate the benefits of the Cereal. Thus, we assign a certain number of bits (e.g., 16 bits in our case) in the object header for this purpose. Every time a serialization happens during the application runtime, a per-unit serialization counter is incremented. Then, when the serializer visits an object during a traversal, it compares its serialization counter with the value in the reserved field of the object header. If these two values are equal, this means that this object has been visited during this serialization. If not, the current serialization counter is recorded to mark that it is visited. This counter has a chance to overflow; however, we clear this metadata during the Java garbage collection so that the overflow is not likely to happen. If a serialization counter is about to overflow, it is also possible to force the garbage collection by invoking `System.gc()`.

Supports for Shared Objects. During the serialization, we record the relative address of the object for already serialized objects. However, since there is only a limited space reserved for this purpose, when multiple threads happen to perform serialization on a shared object, only one thread can record the value for this reserved space. In our implementation, we let the very first thread to serialize this object reserve this header area by writing down its unit ID to another reserved field. Then, when other unit accesses this header and then checks the unit ID field, it will find out that this object's header area is reserved for a different unit. In this case, Cereal cannot perform the serialization and should fall back to the software-handled serialization, which utilizes a thread-local hash table to record the relative addresses for serialized objects. This can potentially reduce the performance benefits of the Cereal; however, we observe that real-world big data analytics workload essentially does not serialize shared objects. Note that this unit ID field can also be regularly cleared during a garbage collection.

TABLE I
ARCHITECTURAL PARAMETERS FOR EVALUATION

Host Processor	
Core	Intel (R) Core (TM) i7-7820X CPU 8-core @ 3.60GHz
L1 $	32KB I-cache, 32KB D-cache
L2 $	1MB (Private, Unified)
L3 $	11MB (Shared, Unified)
DDR4 Memory System	
Organization	DDR4-2400, 4 channels, 128GB
Bandwidth Latency	76.8 GB/s (19.2 GB/s per channel), Zero-load latency 40ns
Cereal Configuration	
Cereal Unit	8 Serializer Unit, 8 Deserializer Unit
MAI / TLB	4KB, 32B block size, 64 entries / 128 entries

Limitations on the Number of Class Types. Since Cereal utilizes hardware structures (CAM for the serialization; SRAM for the deserialization) for a `klass` address to the class ID translation (during serialization) and a class ID to the `klass` address translation, Cereal can only support serializations or deserializations which involves less than the certain number of serializable class types. However, in practice, our 4K entries for this CAM and RAM is more than enough to run various real-world applications. In fact, we found that our evaluated real-world big data analytics applications running on Spark has at most a few hundreds of serializable class types.

Address Translation and Cache Coherence. Cereal assumes 1GB huge pages, which are popular in a system with large physical memory size. Our prototype system has 128GB physical memory, and thus we assume no TLB miss with our 128-entry TLB. However, on a system with larger physical memory, a TLB miss may happen in Cereal. Since Cereal Deserialization Unit (DU) has sequential memory access pattern, the cost of missing TLB can be amortized. On the other hand, Cereal Serialization Unit (SU) includes random memory accesses, and therefore TLB misses can potentially become a performance bottleneck. Improving TLB effectiveness for accelerators is an active research area, and we can leverage recent proposals such as identity mapping with coarse-grained protection [20], flattened/region-based page tables [23], coalesced/shared MMU caches for a large working set [10], and speculative address translation using huge pages [8]. For cache coherence Cereal sends `get` coherence messages as in [37] to fetch the up-to-date copy from either the on-chip cache or off-chip memory. As Cereal is directly connected to the on-chip bus, participating in the on-chip coherence domain is relatively straightforward. A potential increase in memory latency due to coherence operations can be effectively tolerated by Cereal's pipelined execution.

VI. EVALUATION

A. Methodology

Evaluation Model. We evaluate Cereal using a custom cycle-level simulator integrated with a DRAM model similar to what is used in popular architectural simulators [21], [47]. We integrated the simulator to the production-grade JVM (OpenJDK 1.8.0.60 [43]) so that the simulator can access the complex JVM internal functions, such as class, object layout,

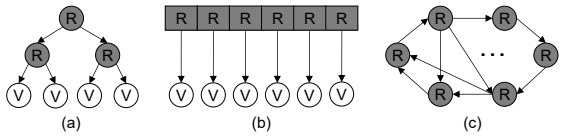

Fig. 9. Simplified layouts of microbenchmarks (a) Tree (b) List (c) Graph

TABLE II
MICROBENCHMARK CONFIGURATION

Tree	narrow(leaf: 2, node: 2,097,150) / wide(leaf: 8, node: 19,173,960)
List	small(length: 524,288) / large(length: 2,097,152)
Graph	sparse(node: 4,096, edge: 1) / dense(node: 4,096, edge: 4,095)

TABLE III
SPARK WORKLOADS

Workload	Type	Input size
NWeight	Graph	156MB
Support Vector Machine (SVM)	Machine learning	1740MB
Bayesian Classification (Bayes)	Machine learning	1126MB
Logistic Regression (LR)	Machine learning	1945MB
Terasort	Sort	3072MB
Alternating Least Squares (ALS)	Machine learning	1331MB

and address/constant acquisition. We functionally validated our simulator and checked that our simulator can produce the functional outcome for the S/D during real-world workloads, including Spark applications. We compare Cereal with two software serializers; Java S/D and Kryo. For the Kryo, we use Kryo version 4.0. Table I summarizes the system parameters we use to evaluate the software serializers and Cereal. For the power and area estimation, we implement Cereal modules in RTL using Chisel3 [13], and then synthesize Cereal RTL using Synopsys Design Compiler with TSMC 40nm standard cell library. Throughout the evaluation, we assumed eight S/D units. Each deserialization unit utilizes four-block reconstructors to process multiple blocks in parallel.

Workloads. To evaluate Cereal, we use the three sets of workloads; microbenchmarks, Java Serialization Benchmark Suite (JSBS) [29], and real-world Spark applications. First, we evaluate a set of microbenchmarks that serializes objects whose object graph has a tree, list, or random graph shape. Figure 9 shows the example layouts of our microbenchmark set. Table II shows our microbenchmark configuration.

To compare Cereal with the performance of other S/D libraries, we evaluate Cereal on JSBS as well. JSBS contains a set of different serializers and repeatedly executes the S/D process of each serializer using several pre-defined objects. We execute all of the serializers in JSBS for 1,000 times and compare average S/D speedup and the size of the serialized object with Cereal.

Lastly, we evaluate Apache Spark 2.4.1 with Intel HiBench 7.0 [24]. We carefully select a different set of benchmarks that heavily use the S/D process. Table III summarizes workloads with the corresponding input size and type. we set a region-of-interest (ROI) for the S/D event only and use sufficiently large heap size (64GB) with the default heap size ratio (Young:Old Generation = 1:2) to minimize the impact of GC.

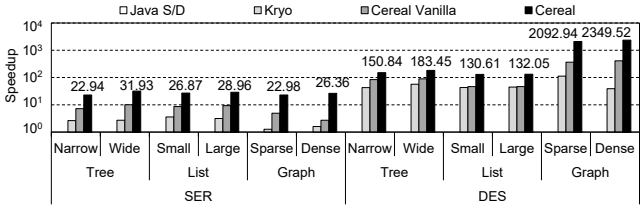

Fig. 10. Performance of software S/D implementations and Cereal for microbenchmarks (in log scale)

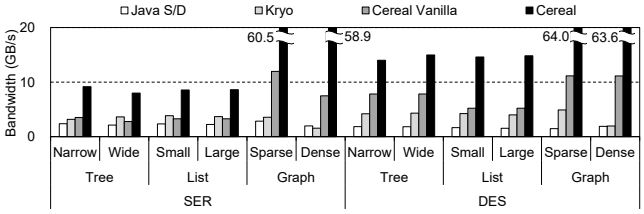

Fig. 11. Bandwidth utilization of software S/D implementations and Cereal implementation for microbenchmarks

B. Microbenchmark Analysis

Figure 10 shows the overall speedup of Kryo and Cereal over Java S/D. Throughout all benchmarks, Kryo achieves significant speedups of 2.30× (serialization) and 52.3× (deserialization) from its integer class numbering and tightly optimized reflection functions. On the other hand, Cereal achieves speedups of 26.5× (serialization) and 364.5× (deserialization) over Java S/D, which are much more significant than those of Kryo.

Cereal's significant speedups come from the utilization of multi-level parallelism that existing S/D libraries such as Java S/D and Kryo fails to exploit. The third bar from each group in Figure 10 (labeled "Cereal Vanilla") demonstrates this point. In the figure, the third bar from each group represents the hypothetical performance of Cereal without any parallelism (i.e., no pipelining, single block reconstructor for DU) except for the operation-level parallelism (i.e., still assumes the multiple SUs and DUs can run in parallel). Comparing those bars to the third bars from each group (i.e., real Cereal performance), we can identify that substantial amount of performance gains are indeed attributed to the fine-grained parallelism.

Part of this parallelism, especially for DUs, is newly exposed by efficient S/D format. Specifically, our serialization format that decouples value array, reference array, and layout bitmap allows the DUs to execute the deserialization process in parallel at the 64B block granularity regardless of the object layout. This enables a single DU to have multiple block reconstructor modules to further improve the throughput. Also, our proposed serialization format lets Cereal DU to handle deserialization with sequential memory accesses and maximize the on-chip data reuse. As a result, Cereal achieves higher speedups on deserialization than serialization.

Figure 11 shows the bandwidth utilization during the serialization and deserialization. For both Java S/D and Kryo executed on CPU, the system utilizes only a small fraction of the memory bandwidth it can use. For example, Java serializer and Kryo utilizes an average of 2.71% and 4.12% of total

TABLE IV
SERIALIZED OBJECTS SIZE ACROSS VARIOUS MICROBENCHMARKS

Unit (MB)	Tree		List		Graph	
	Narrow	Wide	Small	Large	Sparse	Dense
Java S/D	23.0	148.6	8.0	59.4	22.1	115.5
Kryo	12.0	48.0	2.5	10.0	10.8	51.1
Cereal	16.1	80.0	16.0	47.8	2.4	2.4

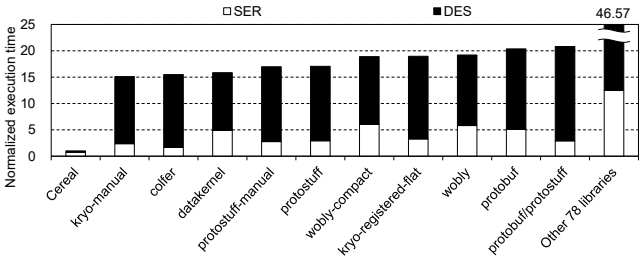

Fig. 12. Performance comparisons of different S/D libraries

DRAM bandwidth (i.e., 76.8 GB/s with four DDR4 memory channels), respectively. This indicates that there is ample room for a hardware accelerator to achieve a substantially higher throughput without being limited by the memory bandwidth. In fact, Cereal utilizes 20.9% of the peak DRAM bandwidth on average (up to 74.5%), which leads to significant speedups presented in Figure 10. Similar patterns are observed in deserialization. Cereal utilizes 31.1% of the DRAM bandwidth on average (up to 83.3%), while Java deserializer and Kryo only use 3.48% and 4.50%, respectively.

Table IV compares the size overhead of microbenchmarks. The size of our format is similar or a little larger than Java S/D for Tree and List benchmarks. As our serialization format focuses more on efficient computation, the Cereal serialization format incurs modest size overhead mostly coming from additional metadata and reference offsets that Java S/D and Kryo do not include. However, Cereal provides the object packing scheme to represent our metadata. As a result, the size of Graph benchmark which has objects with many references can be serialized much more efficiently than other formats.

C. JVM Serializer Benchmark Set

Figure 12 compares the normalized speedups of different libraries for S/D. Cereal shows significantly high speedups compared to the other 88 libraries, which achieves the 43.4× speedups on average. Kryo-manual is the fastest library among all serializers in our experiment. As an optimization, Kryo-manual utilizes the integer class numbering to reduce size and reflection overhead, optimized reflection functions in deserializer, and manually optimized serialization functions. Even with aggressive optimizations of Kryo-manual, Cereal still outperforms the Kryo-manual by 15.1× due to better computation efficiency that comes from utilizing multiple levels of compute parallelism and memory-level parallelism. While Cereal significantly outperforms others, the size of Cereal is comparable to the others. Note that the size of Cereal is smaller than the average size of all the benchmarks by 46% with its packing scheme for the references and bitmap metadata.

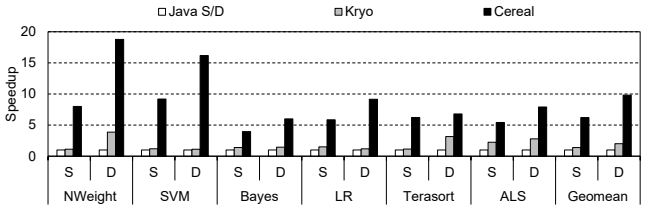

Fig. 13. S/D speedups for Spark applications

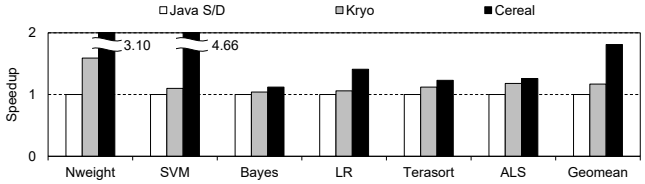

Fig. 14. Program speedups for Spark applications

D. Apache Spark

Figure 13 shows the S/D speedup of Cereal over software implementations on Spark applications. Compared to the Java S/D, Kryo only achieves 1.67× speedup. On the other hand, Cereal achieves 7.97× and 4.81× speedup over Java S/D and Kryo throughout all the applications. This is because Cereal utilizes specialized hardware to exploit abundant parallelism within S/D operations exposed by the specialized serialization format. By accelerating the S/D process, overall application performance is improved by 1.81× (up to 4.66×) and 1.69× (up to 4.53×) over Java S/D and Kryo, respectively. Figure 14 shows overall application performance improvements.

Figure 15 shows the bandwidth utilization of software S/D and Cereal. The overall trends are similar to those of the microbenchmarks and JSBS. Cereal utilizes substantially more memory bandwidth than software schemes, and deserialization significantly more than serialization.

We carefully co-design the serialization format to enable specialized hardware to execute more efficiently. Consequently, the serialization format of Cereal requires such additional metadata, such as bitmap and references. As Java S/D and Kryo do not include references in their serialized format, Cereal has relatively large size overhead than Java S/D and Kryo when the objects have a large number of references. In our experiments, the size of Cereal is larger than Java S/D and Kryo by 4.3% and 21.7%, respectively. However, Cereal focuses on computation efficiency through its unique serialization format and specialized hardware designed for the format.

Figure 16 shows the compression ratio of Cereal across varying applications. Note that our packing scheme only reduces the size of reference offsets and layout bitmaps. Naturally, the scheme is not very effective for the objects with a few references. Cereal can further reduce the size of the serialized format by stripping markword (8B) in the header that are often not necessary after the serialization (i.e., Header Strip in Figure 16). However, there are cases where a certain field of the header (e.g., `hashcode`) needs to be re-constructed during the deserialization [41], incurring additional

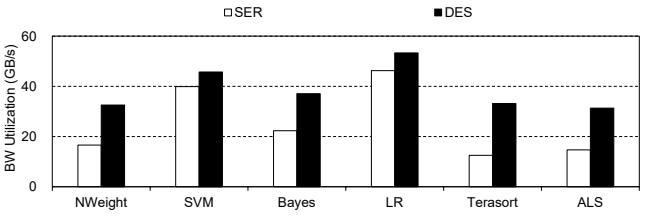

Fig. 15. Bandwidth utilization of Cereal on Spark applications

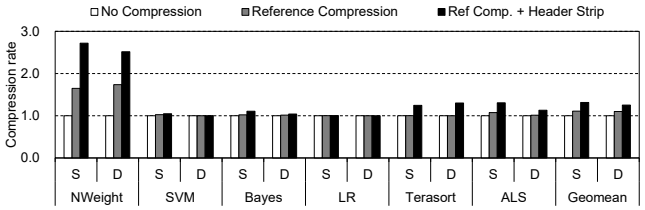

Fig. 16. Compression rate of Cereal object packing scheme

performance overheads. While our packing scheme achieves a moderate compression ratio (28.3% on average), the packing scheme is very effective on some workloads such as NWeight, which includes a very large number of references. For several workloads like SVM, Bayes, and LR, most objects have just a few references. Therefore, both our packing scheme and markword stripping have little impact on the compression rate.

E. Area, Power, and Energy Analysis

Area. Table V shows the breakdown of the area in Cereal. Total area of Cereal with 8 serialization and deserialization units is 3.857mm^2. Note that the die size of our baseline CPU, Intel i7-7820X Processors with 14nm process, is 2362.5mm^2 [25]. Cereal occupies 612.5 times less area than our baseline CPU. Considering that we synthesize Cereal using 40nm technology, the effective area difference becomes even larger. A single Serializer Unit occupies 0.058mm^2, so the total area for eight serializer units is 0.464mm^2. A single deserializer unit occupies 0.281mm^2, so the total area for eight deserializer units is 2.248mm^2. The others are system-wide components such as TLB and MAI which consume 1.145mm^2.

Power. Table V also shows the average power breakdown of Cereal. Compared to the Host CPU power, whose TDP is 140W, Cereal consumes 113.7× less power. This indicates that general-purpose CPU is inefficient in performing S/D operations, which mainly consists of very simple arithmetic operations and many off-chip memory accesses. The use of specialized architecture for S/D operations achieves both performance improvement and power savings at the same time.

Energy. Figure 17 shows the energy consumption of Java S/D, Kryo, and Cereal in Spark applications that are normalized to numbers of Java S/D. Kryo, which runs on the host CPU like Java S/D does, consumes geomean 1.39× less energy compared to Java S/D during serialization, and geomean 2.01× energy compared to Java S/D during deserialization. Since both serializers run on the same device, the energy efficiency of Kryo results from its speedup. On the other hand, Cereal consumes geomean 313.6× less energy compared to Java S/D during serialization, and geomean 165.4× less energy compared to Java S/D during deserialization. Also, Cereal consumes geomean 225.5× less energy compared to Kryo during serialization, and geomean 82.3× less energy compared to Kryo during deserialization. Cereal saves total 227.75× and 136.28× S/D energy over Java built-in serializer and Kryo respectively. The superior energy efficiency of Cereal is attributed to both its high speedups and specialized hardware.

TABLE V
TOTAL AREA/POWER USAGE OF CEREAL

Unit	Area (mm^2)	Power (mW)	# of unit	Total Area (mm^2)	Total Power (mW)
Serializer					
Header manager	0.003	1.3	8	0.024	10.4
Reference array writer	0.013	5.8	8	0.104	46.4
Object metadata manager	0.014	7.6	8	0.112	60.8
Object handler	0.028	18.4	8	0.224	147.2
Serializer Area: 0.464mm^2 / Average Power : 264.8mW					
Deserializer					
Layout manager	0.020	10.9	8	0.16	87.2
Block manager	0.217	81.1	8	1.736	648.8
Block reconstructor	0.011	6.9	32	0.352	220.8
Deserializer Area: 2.248mm^2 / Average Power : 956.8mW					
TLB	0.282	2.7	1	0.282	2.7
MAI	0.161	0.8	1	0.161	0.8
Class ID Table (2KB)	0.230	1.2	1	0.230	1.2
Klass Pointer Table (4KB)	0.472	5.3	1	0.473	5.3
Total Area: 3.857mm^2 / Average Power : 1231.6mW					

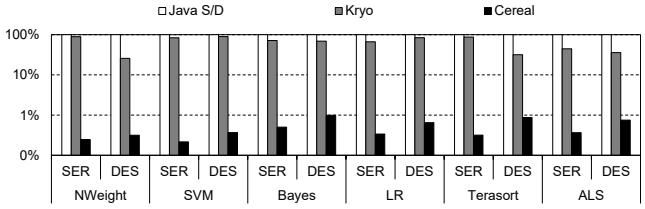

Fig. 17. Normalized energy consumption

VII. RELATED WORK

Software-based S/D Optimization. Many S/D libraries have been proposed [28], [32], [34], [38], [39], [41]. Instant pickles [38] serializes Scala objects fast by using statically generated S/D code. However, it still falls back to the reflection function at runtime when a type cannot be settled at compile time. Skyway [41] focuses on reducing the inefficiencies of using reflective functions and user effort for manual type registration. However, Skyway achieves only marginal average performance gains over Kryo and can produce much more inflated serialized streams. While many off-heap approaches [7], [17], [36], [42] are proposed as an alternative way to cope with several problems with data representation as an object, they heavily relies on the user-level refactorization [42] or heavy engineering effort both on dedicated compiler/runtime systems [39].

Hardware-based S/D Optimization. Morpheus [52] aim to minimize CPU burden by offloading deserialization to the embedded core in the SSD. However, since the SSD controller cores often be overloaded with various tasks such as FTL, garbage collection, and flash control functions, adding more

burden to the cores leads to I/O performance degradation. As an alternative approach, HODS [35] exploits separate FPGA module to offload deserialization which works in parallel with all storage operations. However, they have relatively narrow applicability as they cannot deal with objects with references in their fields inside the SSD. On the other hand, several studies proposed to accelerate the serialization with FPGA-based accelerators [46], [56], but they show relatively marginal gains due to limited utilization of object-level parallelism.

Domain-specific Architectures for Big Data Applications. Recent studies proposed various domain-specific architectures to accelerate data-intensive workloads such as graph processing [1], [2], [19], garbage collection [26], [31], [37], and database applications [15], [16], [22], [33], [53]. Many of these proposals overcome the limitations of the convention CPU by i) specializing hardware for primitive data-intensive or computational operations within the target algorithms, ii) exploiting the abundant data-level parallelism within the algorithm, iii) utilizing memory-level parallelism, iv) maximizing the re-use of on-chip memory to minimize the off-chip communication, and v) adopting emerging technologies such as near-data processing. Cereal also exploits such techniques to design efficient hardware for serialization and deserialization. Moreover, our proposal jointly optimizes the serialization format along with the hardware design to achieve greater efficiency.

VIII. Conclusion

This paper presents Cereal, a specialized architecture for S/D operations, which are widely used in big data analytics frameworks. With careful co-design of the serialization format and hardware architecture, Cereal effectively exploits massive memory-level parallelism (MLP) by processing in parallel multiple references and values in an object as well as multiple objects in the S/D process. Cereal also introduces an object packing scheme to keep the metadata for a serialized stream compact. Cereal demonstrates substantial performance and energy efficiency gains over the state-of-the-art S/D libraries running on a CPU for both synthetic benchmarks and real-world data analytics applications on Apache Spark.

Acknowledgments

This work was partly supported by Institute for Information & communications Technology Promotion (IITP) grant funded by Korea government (MSIT) (2014-0-00035, Research on High Performance and Scalable Manycore Operating System), Nano-Material Technology Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (2016M3A7B4909668), and a research grant from Samsung Electronics. We also acknowledge support from the IC Design Education Center (IDEC) Korea for EDA tools and the ICT at Seoul National University for research facilities. Jae W. Lee is the corresponding author.

References

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, "Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 336–348.

[2] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, p. 105–117.

[3] "Apache Flink," http://flink.apache.org.

[4] "Apache Giraph," http://giraph.apache.org.

[5] "Apache Hadoop," http://hadoop.apache.org.

[6] "Apache Ignite," https://ignite.apache.org.

[7] "Tungsten - Databricks," https://databricks.com/glossary/tungsten.

[8] T. W. Barr, A. L. Cox, and S. Rixner, "Spectlb: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, p. 307–318.

[9] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," *Commun. ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[10] A. Bhattacharjee, "Large-reach memory management unit caches," in *Proceedings of the 46th International Symposium on Microarchitecture*, 2013, p. 383–394.

[11] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica, "Hyracks: A flexible and extensible foundation for data-intensive computing," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 1151–1162.

[12] "Cap'n Proto," https://capnproto.org.

[13] "Chisel3," https://github.com/freechipsproject/chisel3.

[14] A. C. De Melo, "The new linux'perf'tools," in *Slides from Linux Kongress*, vol. 18, 2010.

[15] M. Drumond, A. Daglis, N. Mirzadeh, D. Ustiugov, J. Picorel, B. Falsafi, B. Grot, and D. Pnevmatikatos, "The mondrian data engine," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017, p. 639–651.

[16] M. Gao, G. Ayers, and C. Kozyrakis, "Practical near-data processing for in-memory analytics frameworks," in *2015 International Conference on Parallel Architecture and Compilation*, 2015, pp. 113–124.

[17] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingam, M. Costa, D. G. Murray, S. Hand, and M. Isard, "Broom: Sweeping out garbage collection from big data systems," in *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, 2015.

[18] "gRPC," https://grpc.io.

[19] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proceedings of the 49th International Symposium on Microarchitecture*, 2016, pp. 1–13.

[20] S. Haria, M. D. Hill, and M. M. Swift, "Devirtualizing memory in heterogeneous systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, p. 637–650.

[21] W. Heirman, T. Carlson, and L. Eeckhout, "Sniper: scalable and accurate parallel multi-core simulation," in *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*, 2012, pp. 91–94.

[22] J. Heo, J. Won, Y. Lee, S. Bharuka, J. Jang, T. J. Ham, and J. W. Lee, "IIU: Specialized architecture for inverted index search," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, p. 1233–1245.

[23] K. Hsieh, S. Khan, N. Vijaykumar, K. K. Chang, A. Boroumand, S. Ghose, and O. Mutlu, "Accelerating pointer chasing in 3d-stacked memory: Challenges, mechanisms, evaluation," in *2016 IEEE 34th International Conference on Computer Design*, 2016, pp. 25–32.

[24] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *2010 IEEE 26th International Conference on Data Engineering Workshops*, 2010, pp. 41–51.

[25] Intel, "Intel® core™ i7-7820x x-series processor," https://ark.intel.com/content/www/us/en/ark/products/123767/intel-core-i7-7820x-x-series-processor-11m-cache-up-to-4-30-ghz.html, 2017.

[26] J. Jang, J. Heo, Y. Lee, J. Won, S. Kim, S. J. Jung, H. Jang, T. J. Ham, and J. W. Lee, "Charon: Specialized near-memory processing

architecture for clearing dead objects in memory," in *Proceedings of the 52nd International Symposium on Microarchitecture*, 2019, p. 726–739.
[27] "Java HotSpot Virtual Machine," http://openjdk.java.net/groups/hotspot.
[28] "Java object serialization specification," https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html.
[29] "Java Serializer Benchmark Suite," https://github.com/eishay/jvm-serializers/wiki.
[30] "java.lang.reflect," https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/package-summary.html.
[31] J. A. Joao, O. Mutlu, and Y. N. Patt, "Flexible reference-counting-based hardware acceleration for garbage collection," in *Proceedings of the 36th International Symposium on Computer Architecture*, 2009, p. 418–428.
[32] A. Khrabrov and E. de Lara, "Accelerating complex data transfer for cluster computing," in *8th USENIX Workshop on Hot Topics in Cloud Computing*, 2016.
[33] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan, "Meet the walkers accelerating index traversals for in-memory databases," in *Proceedings of the 46th International Symposium on Microarchitecture*, 2013, pp. 468–479.
[34] "Kryo," https://github.com/EsotericSoftware/kryo.
[35] D. Li, F. Wu, Y. Weng, Q. Yang, and C. Xie, "Hods: Hardware object deserialization inside ssd storage," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2018, pp. 157–164.
[36] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 936–947, 2016.
[37] M. Maas, K. Asanović, and J. Kubiatowicz, "A hardware accelerator for tracing garbage collection," in *Proceedings of the 45th International Symposium on Computer Architecture*, 2018, pp. 138–151.
[38] H. Miller, P. Haller, E. Burmako, and M. Odersky, "Instant pickles: Generating object-oriented pickler combinators for fast and extensible serialization," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2013, pp. 183–202.
[39] C. Navasca, C. Cai, K. Nguyen, B. Demsky, S. Lu, M. Kim, and G. H. Xu, "Gerenuk: Thin computation over big native data using speculative program transformation," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 538–553.
[40] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *2015 USENIX Annual Technical Conference*, 2015, pp. 291–305.
[41] K. Nguyen, L. Fang, C. Navasca, G. Xu, B. Demsky, and S. 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*, 2018, pp. 56–69.
[42] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu, "Facade: A compiler and runtime for (almost) object-bounded big data applications," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 675–690.
[43] "OpenJDK8," http://openjdk.java.net/projects/jdk8.
[44] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun, "Making sense of performance in data analytics frameworks," in *12th USENIX Symposium on Networked Systems Design and Implementation*, 2015, pp. 293–307.
[45] "Protocol Buffers," https://developers.google.com/protocol-buffers/docs/javatutorial.
[46] S. Rheindt, A. Fried, O. Lenke, L. Nolte, T. Wild, and A. Herkersdorf, "Nemesys: near-memory graph copy enhanced system-software," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 3–18.
[47] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," in *Proceedings of the 40th International Symposium on Computer Architecture*, 2013, p. 475–486.
[48] "SerDe," https://serde.rs.
[49] "ReflectAsm," https://github.com/EsotericSoftware/reflectasm.
[50] K. Shvachko, H. Kuang, S. Radia, R. Chansler *et al.*, "The hadoop distributed file system." in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, vol. 10, 2010, pp. 1–10.
[51] "The Colfer Serializer," https://go.libhunt.com/project/colfer.
[52] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating application objects efficiently for heterogeneous computing," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, 2016.
[53] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture and design of a database processing unit," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, 2014.
[54] M. Zaharia, "What is changing in big data," 2016.
[55] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, 2010, pp. 10–10.
[56] S. Zhang, H. Angepat, and D. Chiou, "Hgum: Messaging framework for hardware accelerators," in *2017 International Conference on ReConFigurable Computing and FPGAs*, 2017.