

A Deserialization Architecture for GC Languages

Ethan Wu
Viansa Schmulbach

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-268

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-268.html>

December 12, 2023



Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Thank you to Sagar Karandikar who provided invaluable ideas and guidance on this project. Additionally, this project was submitted as part of CS264. Thank you to Professor Sen and Shangyin Tan, who gave very helpful advice on this project.

A Deserialization Architecture for GC Languages

Ethan Wu

ethanwu10@berkeley.edu
University of California, Berkeley
USA

Viansa Schmulbach

ansa@berkeley.edu
University of California, Berkeley
USA

Abstract

Numerous works have tackled the problem of offloading deserialization workloads to specialized hardware accelerators, however few target interoperability with “managed” languages which employ a garbage collector and other advanced runtime features. We develop a deserialization accelerator that can directly create “native” Java objects that have no additional software overhead for Java code to interact with, as if they had been created by Java code. In the process, we explore how a hardware accelerator’s memory allocation and object creation functionality can be integrated with the HotSpot Java Virtual Machine, without sacrificing accelerator performance.

CCS Concepts: • **Hardware** → **Hardware accelerators**; • **Software and its engineering** → **Garbage collection**.

Keywords: deserialization, warehouse-scale computing, garbage collection

ACM Reference Format:

Ethan Wu and Viansa Schmulbach. 2023. A Deserialization Architecture for GC Languages. In *Proceedings of Programming Language Design and Implementation (PLDI '24)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

1.1 Motivation

The rise of warehouse scale compute has sparked interest in the discussion of datacenter-specific costs, dubbed the “datacenter tax” [7] which includes serialization/deserialization, RPCs, and compression. In Google’s Warehouse Scale Computers, an estimated 5% of cycles were spent serializing and deserializing objects. This provided the motivation for Karandikar et. al.’s accelerator for Protobuf [8], Google’s serialization framework. This accelerator offloads cycles from the CPU by directly serializing and deserializing C++ objects in hardware and placing the result in an accelerator-owned arena. The deserialization accelerator was shown to have a 6.9x increase in performance over the Xeon-based system, and the serialization accelerator had a 4.5x increase in performance over the Xeon. However, the accelerator is limited to only working with C++ objects. We would like to extend support of this accelerator to a garbage-collected language, specifically Java, while maintaining comparable

performance, as well as performance for C++. That is, we would like for the hardware to write the native object to be used by the software directly into memory, without the software needing to perform further manipulation of the deserialized objects. Additionally, we would like the accelerator to remain interoperable with existing software such that applications using the accelerator can talk to other applications using the standard software Protobuf implementation.

1.2 Background

1.2.1 The Garbage-First Garbage Collector. The HotSpot Java Virtual Machine implementation contains many garbage collectors. We target the current default, Garbage First (G1GC), which is a parallel, generational, concurrent-marking garbage collector [2]. Like all other garbage collectors in HotSpot, G1GC compacts the heap by moving live objects into a “survivor region” when performing a collection—this process is called *evacuation*. As a result, HotSpot serves the majority of its allocations via a simple bump allocator, since the due to compaction the heap has large contiguous chunks of free space from which to allocate. Thus, the hardware accelerator can also use a bump allocator instead of needing to understand more complex structures like freelists. Additionally, G1GC performs its evacuation step in a stop-the-world pause, not concurrently; this fact greatly simplifies integration with the hardware accelerator’s allocations.

Internally, G1GC partitions the heap into multiple regions, each of which can be designated as Eden (new objects), Survivor (young generation), and Old. Collections evacuate entire regions at a time into new regions of either Survivor or Old type; the original regions are then freed.

1.2.2 The Protobuf Wire Format. Protocol buffers are a structured serialization format, where each field is encoded on the wire as a field identifier containing type information, followed by the actual field data. The Protobuf wire encoding heavily uses the *varint* encoding for integers, which uses fewer bytes for smaller integers [3]. This format is particularly amenable to hardware acceleration, since it is significantly slower to decode on CPU compared to other field data types such as bytes (which simply places the raw bytes as-is, and is decoded with a *memcpy*) [8].

1.2.3 Java Object Format. Each Java object in the HotSpot JVM contains a header (usually 12 bytes) at the beginning of the object that contains metadata that must be populated for the object to be correctly recognized by the JVM. On 64-bit

systems, the first 8 bytes are a *mark word* containing generic metadata and GC state—for freshly-created objects, this is always a constant bit-pattern. The next 4 bytes contain the *klass word*, which identifies the type of the object (analogous to a vtable pointer in C++). For each object type, the klass word is a constant value for the lifetime of the JVM.

Fields within a Java object are laid out as the JVM sees fit, and are often rearranged from their declared order; the JVM packs them to reduce waste due to alignment as much as possible. The layout may not be stable between different invocations of the JVM (depending on VM parameters), however fields are never rearranged while the JVM is running.

1.3 Prior Work

Karandikar et. al. [8] describe the original Protobuf accelerator which is to be extended during this project. As explained in the background section, the accelerator in this project only supported C++ as a host language. We would like to extend this accelerator to support multiple languages, including garbage collected languages.

The Cereal paper [5] describes a hardware accelerator for serialization which implements a specialized serialization format for Java objects, but does not explain how the GC becomes aware of the objects created by the accelerator. Additionally, the paper does not publish any artifacts, so it is hard to determine if the paper edited the GC. Finally, the paper only supports one language, while our proposed design will support multiple target languages.

The Skyway paper [10] discusses how to share heap data between multiple machines without undergoing deserialization and serialization, thus solving a similar problem of objects “appearing” on the heap without software knowledge. This project targets moving objects around a distributed system, without regard to interoperability with existing formats; it also tackles the problem with pure software, without investigating hardware offload. Skyway modifies the JVM and garbage collector, updating GC data structures whenever a new object is allocated so that the object is reachable by the GC. Additionally, Skyway targets sharing static data, which is a very different role from Protobuf messages, which are used for active communication. Thus, some of the design decisions (such as allocating into the old generation) are unsuitable for warehouse-scale workloads.

The Breakfast of Champions paper [11] uses an NIC based accelerator which implements a zero-copy serialization technique. However, it does not address deserialization and the garbage collector and allocator integration it entails, but calls this out as future work. Their accelerator also does not work with multiple languages. In addition, they generate custom types for Protobuf messages that do not match those generated by the standard C++ Protobuf compiler.

1.4 Contributions

For the native language, we chose to focus on Java for a few reasons: (1) Java is commonly used by the users of Protobuf, ie. there is more demand for a Java deserialization accelerator (2) the complexity of the JVM and HotSpot and lack of pointers make Java likely the most challenging target language, and thus raises some interesting research questions and (3) the compacting garbage collector (as compared to Go, which uses free lists) allows us to reserve a large chunk of memory and have our objects be naturally adopted by the JVM when they are collected and moved outside of our region.

To the best of our knowledge, no other work has created a deserialization accelerator in a mainstream serialization format such as Protobuf (ie. not custom format) which also allows for GC language support, and is the only paper implementing a deserialization accelerator which explicitly addresses garbage collection support. Additionally, this work will propose the first hardware deserialization architecture which supports multiple languages with a standardized wire format.

2 Software Stack and Integration with VM

In order to interface with the accelerator, we used the Java Native Interface (JNI), which allows for “native” C++ methods to be called from Java code, which can then call into the JVM through a special `JNIEnv*` object passed to the method.

2.1 Background: The Java Native Interface

All JNI methods are passed in as arguments the `JNIEnv*`, which facilitates calling into the JVM, as well as the `jobject`, a reference to the current object calling the native method. For instance, the JNI Function header for the the `deserialize` method is as follows:

```
JNIEXPORT jobject JNICALL Protoacc_deserialize
(JNIEnv *, jobject, jclass, jbyteArray);
```

In addition to the `JNIEnv*` and `jobject`, our `deserialize` function requires a reference to the class which is being deserialized into, as well as a byte array which holds the bytes of the serialized object.

Of important note is the fact that the `jobject` is not a direct pointer to a Java object, but rather a pointer to a “handle” which contains a pointer to the Java object. In addition to requiring all objects to be wrapped in a JNI handle, this also means that Java will update the JNI handles of objects whenever they are moved by garbage collection.

2.2 Generations and Barrier Interaction

Since Protobuf is often used for workloads like RPC (via gRPC [4]) where deserialized objects are short-lived, we allocate objects into the young generation of the heap, where objects are cheap to create and expected to mostly die young.

To keep the young generation fast, in G1GC, there are no barriers present in reading or writing to young-generation objects except during concurrent marking. The concurrent marking write barrier serves to ensure that during marking, a mutator cannot remove an object from the graph visible to the marking process [2]. However, a deserialization accelerator will never overwrite pointers to existing objects since it will only create new objects; thus, this write barrier can be safely ignored.

The second barrier present in G1GC is the remembered set write barrier, triggered when writing a pointer to the young generation into an old generation object [2]. However, since the accelerator never writes to any objects other than the ones it creates, it does not interact with this barrier either.

2.3 Allocating Java Heap Memory

In order to implement a zero-copy accelerator, our accelerator places objects directly on the Java heap, as the GC will reject any object placed outside the Java heap. However, if we simply pass a pointer on the Java heap to the accelerator, the JVM will write over the objects created as that heap space is unallocated according to the JVM. Because there is not a straightforward method of requesting a portion of heap space in Java, we accomplished this by requesting a Java byte array through the JVM, so that the JVM would reserve the space inside the array. Then, we pass a pointer from within our newly-allocated array to the accelerator, which then allocates the object within the array. Once the accelerator completes, a new JNI Handle for the deserialized object is created in C++; a pointer to this handle is then returned to the Java caller.

Figure 1 shows the Java heap after the accelerator has deserialized an object. On the left, the accelerator has newly deserialized the object into the Java byte buffer. The JVM considers all JNI handles to be roots when marking live objects during a garbage collection cycle. Thus, when the GC is evacuating a region for collection, it will find the accelerator-created objects via the JNI handle pointing to the root message object, and proceed to copy the entire object hierarchy out into a survivor region. After this collection (shown on the right of Figure 1), the newly-allocated objects are exactly the same as any other object on the Java heap.

As an optimization, we opt to not retain a JNI handle to the byte array, and instead make a handle to a sentinel object at the beginning of the array. This way, the Java heap remains parseable from the perspective of the JVM (which assumes the heap is packed with contiguous objects), but the byte array does not need to get unnecessarily copied. We detect an evacuation of the region the byte array was in by observing when the sentinel object moves; when this happens, we must obtain a new byte array to allocate into. This also ensures that our allocation remains in an Eden heap region, and does not get moved to old space.

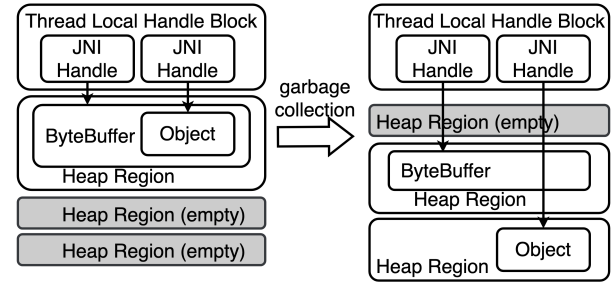


Figure 1. Java Heap, Before and After GC

2.4 Pausing the Garbage Collector

If garbage collection occurs while the accelerator is deserializing the object, this could result in the byte array being moved in memory and the accelerator writing into newly freed space. Therefore, while the accelerator is in the process of deserializing, garbage collection must be paused. To accomplish this, we add a new JNI method, `ForceThreadSafepointUnsafe` to pause garbage collection.

In G1GC, all threads must be at a safepoint before the evacuation phase of garbage collection can occur. However, all native threads are considered to be at a safepoint until they call into the JVM, as it is assumed that native threads are not editing Java objects, and therefore their execution will not interfere with the JVM. We add an additional bit to each Java thread’s state to indicate whether an accelerator is currently deserializing in that thread. Additionally, we update the safepoint checking mechanism to check this field, and return false if this thread currently has a hardware deserialization in progress. The implementation of `ForceThreadSafepointUnsafe` therefore simply sets this bit in the current thread.

3 Accelerator Design

3.1 Software Stack

3.1.1 Background: Protobuf Generated Java Object Format. The Java object graph for the Protobuf in-memory representation contains many common Java classes (such as `java.util.List`) and singleton objects. For instance, integer repeated fields are deserialized into `IntArrayList`, a custom Protobuf implementation of the `java.util.List` interface. The object layout of an `IntArrayList` is shown in Figure 2. Additionally, each user-defined message contains some internal fields which need initializing, such as memoized values that must be set to -1, and an `unknownFields` object which must point to a singleton empty list. Furthermore, the object contains a bitfield (referred to as the “hasbits” field) which holds information about which of the fields in the message are populated. The object layout of a Protobuf Java message `CustomMessage` is shown in Figure 2.


```

class IntArrayList extends AbstractList {
    boolean isMutable;
    int size;
    int[] array;
}
class CustomMessage {
    int memoizedSize = -1;
    int bitField0_;
    UnknownFieldSet unknownFields;
    byte memoizedIsInitialized = -1;
    /* Other fields here */
}

```

Figure 2. Object structures of types involved in Protobuf messages

3.1.2 Accelerator Descriptor Table. In order for our accelerator to be able to deserialize a message object, it needs some additional information about the layout of the object in Java. To implement this, we generate an *Accelerator Descriptor Table* for each type of message sent, expanding upon the ADT developed by Karandikar, et al [8]. Our deserialize method constructs the ADT lazily the first time a deserialization of this message type is requested, and is reused on subsequent deserializations.

Figure 3 shows all the fields in the ADT (labeled DescriptorTable), all of which are constant throughout the execution of the Java program. These fields include (1) the Java `klass_word` of the object (2) the offset of multiple fields, as discussed in 3.1.1 (3) the size of the current object, so the accelerator knows the amount of space to allocate, and (4) the minimum field number used (used to look up items in the descriptor table).

Additionally, for each field, there is an entry in the descriptor table (labeled `DescriptorTableFieldEntry` in Figure 3) which includes the offset of that field in bytes within the Java object, (2) the Java type of this field, (3) whether or not this field is repeated, and (4) if this field is a submessage, a pointer to the ADT of that message.

3.1.3 Custom Accelerator Instructions. Since the accelerator is located near the core [8], software interfaces with the accelerator through custom instructions, implemented using the RoCC interface of the Rocket Chip framework [1].

The accelerator contains a set of new instructions for managing information about the current JVM’s runtime environment that is used during deserialization. Since the classes and singleton instances are not specific to any particular message, information about these objects is stored on registers within the accelerator. This reduces the number of memory reads that need to be performed during deserialization. This data (klass words for the classes and object instance addresses for singletons) is loaded into the accelerator via a set of RoCC instructions. In particular, since singletons may move upon

```

struct DescriptorTableFieldEntry {
    uint64_t offset : 58;
    uint64_t type : 5;
    uint64_t is_repeated : 1;
    DescriptorTable *nested_descriptor;
};

struct DescriptorTable {
    uint32_t klass_word;
    uint16_t unknown_fields_offset;
    uint16_t memoized_size_offset;
    uint32_t object_size;
    uint32_t memoized_is_initialized_offset;
    uint64_t hasbits_offset;
    uint32_t min_field_num;

    DescriptorTableFieldEntry entries[];
};

```

Figure 3. Accelerator Descriptor Table Fields

GC, the addresses of singleton objects is sent to the accelerator before each deserialization operation after garbage collection has been paused; since these are implemented as custom instructions, these operations complete very quickly.

3.2 Hardware Updates

The Protobuf accelerator RTL implemented by Karandikar et. al. was fairly language-agnostic. We aimed to reuse as much RTL as possible between the two accelerators for a few reasons: (1) to cut down on area and power consumption, (2) to show that the accelerator could be easily expanded to multiple languages with a similar object format beyond Java and C++, and (3) ease of engineering.

3.2.1 Background: C++ Accelerator. In Karandikar et. al’s Protobuf accelerator implementation, there are two allocation arenas passed to the accelerator via RoCC instructions: a flexible arena, where arrays are allocated, and a fixed arena, where all other objects are allocated. This is due to the fact that the size of array objects are not known up front, so the hardware must therefore continue writing to this array in the flexible arena while allocating objects in the fixed arena until the array is closed out.

Field handling logic is rather language specific. Scalar fields (i.e. those with Java primitive types) are handled exactly the same in C++ and Java, however more complex types such as strings and nested messages are represented differently between the two languages, as C++ tends to place more data inline within objects while Java places every object behind a pointer.

3.2.2 Strings, Byte Buffers, and Nested Objects. For representing string and bytes types, the Protobuf Java library uses a `ByteString` object which contains a byte array. These objects have fixed layouts and known-upfront sizes, and thus are populated with memory writes as fast

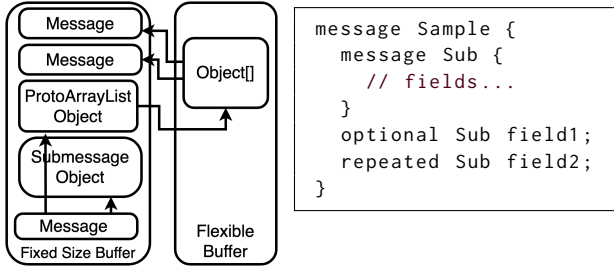


Figure 4. Message with repeated nested submessage

as the data-cache can accept them—all header values are retrieved from internal registers (3.1.3).

Nested objects are allocated and populated in a similar fashion to C++, except with more fields to populate. Since the layout and class word depends on the message, nested messages require loading data from the descriptor table and result in more, smaller scattered writes.

3.2.3 Repeated Fields. Repeated fields are backed by an array containing the elements of the field. Since the length of this array is not known up front, it is allocated into the flexible region, where new elements are written as they are encountered. Additionally, Java has strongly-typed arrays, and does not support polymorphism over primitive types. As a result, for a repeated field of each primitive type, the Protobuf Java library utilizes a different type of primitive array and wrapper ArrayList object. Thus, the field type dictates which of the classes (stored in local registers) is used for the created objects.

Since most of the objects involved in creating the list for a repeated field include length fields, the length fields of the objects are all written (in separate requests) once the repeated field is closed, either by the end of message or by the start of a new repeated field; when possible, additional fields are initialized at the same time as this final write to save a write operation.

4 Evaluation

4.1 Correctness

Since the majority of the correctness tests are architecture-agnostic and do not depend on exact accelerator details, these were implemented through JNI code performing memory accesses like an accelerator would. They were run against the modified JVM on ARM64, and also verified on RISC-V under QEMU emulation—the results are identical.

4.1.1 Verify JVM Object Adoption. To verify that our approach of allocating new objects into an existing byte array on the Java heap does not cause unforeseen problems, we created a test that allocates large numbers of objects and verifies that their internal structure is still valid. On each iteration, the test fills an array with newly allocated objects,

and then reads the fields of all of these objects after the array is populated. Thus, we verify that the messages are not corrupted by the GC and are still usable later during program execution. Finally, the amount of data allocated by this test loop far exceeds the maximum heap size that the JVM is allowed to use, forcing all the created objects to be garbage-collected. If the newly created objects were not properly being collected, we would see that the JVM would run out of memory. Since this test completes successfully, we see that the objects are properly being integrated with the JVM’s heap.

4.1.2 Verifying GC Pause. To confirm the effectiveness of the safepointing implementation, we implemented a simple test where one thread marks itself as not at a safepoint and sleeps, while another thread triggers garbage collection. By enabling garbage-collection and safepoint logs in the JVM, we can observe that the JVM does not reach a safepoint until after the sleeping thread wakes up again and clears our safepoint override bit. Similarly, from the logs we observe that the GC evacuation does not occur until after the safepoint is reached, confirming the validity of our approach.

4.1.3 Benchmarking Stress Tests. Finally, our main benchmarks (described in detail in the next section) also serve as a stress-test for general robustness. The benchmarking harness loads and executes code from many other classes before and after benchmarking runs, and from multiple threads. Thus, during certain phases of the benchmark, the accelerator will operate alongside other activity in the JVM; also, the benchmark runner creates heap pressure as many objects are allocated as fast as possible.

4.2 Benchmarking

4.2.1 Setup. To benchmark accelerator performance, we leveraged HyperProtoBench, a set of benchmarks containing messages that are representative of those seen in workloads at Google [8]. We ported these benchmarks to Java by exporting serialized data from the original C++ benchmarks, and then loading and deserializing them in Java. The benchmark runs measure only the time spent in deserialization, however the JVM also outputs additional debugging information about GC events such as pause times. Each message from the benchmarks in HyperProtoBench are measured independently because the variation in individual message composition provides interesting performance insight. The measurements are performed using OpenJDK’s JMH harness [6], configured to run 5 iterations of measurement lasting 5 seconds each. For each test, the accelerator is compared against the pure Java Protobuf library implementation, running on the same CPU and system.

The tests are run on HotSpot JVM 21 Server for RISC-V, using G1GC, and with pointer compression disabled and a maximum heap of 512MB. The heap size was chosen to be artificially small to bring out the impact of the accelerator

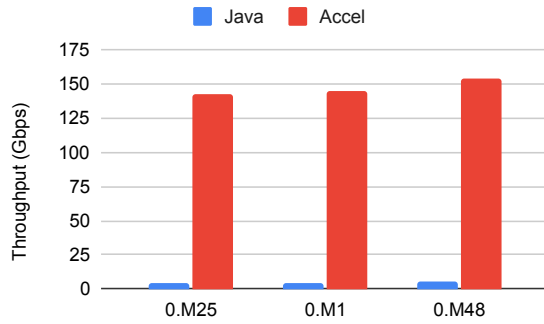


Figure 5. Accelerator throughput

architecture on garbage collection performance. To evaluate the hardware design, we employ FireSim [9], a cycle-accurate simulator with accurately modeled memory timings, booting Linux to run the Java workloads. The accelerator is attached to a BOOMv3 core, an OoO superscalar RISC-V core comparable to ARM A72-like cores [12]; the core runs at 3.0GHz.

Due to the cycle-accurate simulated nature of the system and low noise within Linux (since the image was based on buildroot with no other significant userspace processes running), benchmark timings are extremely stable—all standard deviations were measured to be less than 2% of the measured average value.

4.2.2 Throughput Results. Unlike in the pure Java deserialization baselines, with the hardware accelerator, we observe deserialization times that correlate very strongly with message size, showing that the hardware accelerator is much closer to being completely bottlenecked by system memory bandwidth. In terms of peak performance, we see that the accelerator is between 20 and 35 times faster in comparison to the pure Java baseline measurements.

Observing the scatter plot of time against message size as seen in Figure 6, we can see that the time for a 0-byte message approximately extrapolates to around 1 μ s; this roughly lines up (within error) with our measured overhead of approximately 500 ns when performing all of the setup but not any actual deserialization. This overhead originates from Java code calling into JNI, which then calls back into the JVM; these state transitions have a more significant impact on time than the JVM operations actually being performed. However, compared to the deserialization time for most messages from Java code, this shows that the message size threshold at which accelerated deserialization saves time is quite small.

4.2.3 Pause Times. Pause times during benchmarks were measured to be in the 10 ms range, well above the time necessary to perform a deserialization. Thus, in terms of overall impact on pause time, the behavior of pausing GC until a deserialization has completed has minimal impact on overall responsiveness. However, pauses were infrequent enough

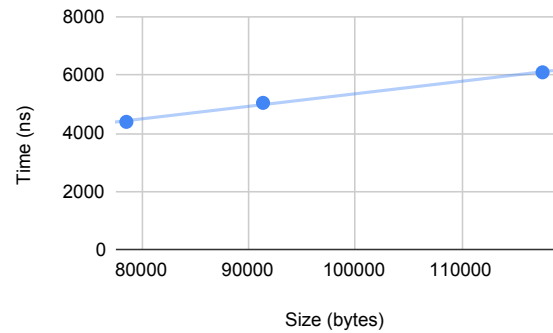


Figure 6. Accelerator deserialization time vs. message size

and deserialization times short enough that we were unable to directly observe the impact of the GC requesting a stop-the-world pause while a deserialization operation is running—this demonstrates that even under heavy synthetic load, the hardware accelerated deserialization is highly unlikely to impact pause times at all.

5 Conclusion and Future Work

In conclusion, we have demonstrated that hardware acceleration of deserialization workloads can be extremely effective in a managed language runtime such as Java. We have validated a method for interoperating with a current production-grade garbage collector (G1).

Future work to be done includes extending the hardware design further to support reconfiguring language-specific parameters (such as the structure of the object graph for a particular Protobuf field type) at runtime, so that new languages can be supported without changes to the hardware. In particular, a model must be developed for describing the object graph and fields to initialize in a way that can be processed without needing excessive memory reads while deserializing.

An additional avenue of improvement is obtaining tighter integration with the JVM; significant portions of the overhead in our approach are due to the JNI interface and the necessary book-keeping for transitioning between “native” and “VM”. If the accelerator were to be directly integrated in the core JVM, much of this can be skipped.

References

- [1] Krste Asanovic et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4, 6–2.
- [2] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management (ISMM '04)*. Association for Computing Machinery, Vancouver, BC, Canada, 37–48. ISBN: 1581139454. DOI: [10.1145/1029873.1029879](https://doi.org/10.1145/1029873.1029879).
- [3] [n. d.] Encoding | protocol buffers documentation. (). <https://protobuf.dev/programming-guides/encoding/>.

- [4] [n. d.] gRPC. (). <https://grpc.io>.
- [5] Jaeyoung Jang, Sung Jun Jung, Sunmin Jeong, Jun Heo, Hoon Shin, Tae Jun Ham, and Jae W. Lee. 2020. A specialized architecture for object serialization with applications to big data analytics. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 322–334. doi: [10.1109/ISCA45697.2020.00036](https://doi.org/10.1109/ISCA45697.2020.00036).
- [6] [n. d.] Jmh. (). <https://openjdk.org/projects/code-tools/jmh/>.
- [7] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43, 3S, (June 2015), 158–169. doi: [10.1145/2872887.2750392](https://doi.org/10.1145/2872887.2750392).
- [8] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A hardware accelerator for protocol buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, Virtual Event, Greece, 462–478. ISBN: 9781450385572. doi: [10.1145/3466752.3480051](https://doi.org/10.1145/3466752.3480051).
- [9] Sagar Karandikar et al. 2018. Firesim: fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 29–42. doi: [10.1109/ISCA.2018.00014](https://doi.org/10.1109/ISCA.2018.00014).
- [10] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: connecting managed heaps in distributed big data systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. Association for Computing Machinery, Williamsburg, VA, USA, 56–69. ISBN: 9781450349116. doi: [10.1145/3173162.3173200](https://doi.org/10.1145/3173162.3173200).
- [11] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of champions: towards zero-copy serialization with nic scatter-gather. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, Ann Arbor, Michigan, 199–205. ISBN: 9781450384384. doi: [10.1145/3458336.3465287](https://doi.org/10.1145/3458336.3465287).
- [12] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. Sonicboom: the 3rd generation berkeley out-of-order machine, (May 2020).