Hardware Deserialization Accelerator with GC Language Support

Ethan Wu, Viansa Schmulbach

JVM Integration

- We extend the Java Native Interface (JNI) to allow manipulation of JVM internals
- In HW, use simple bump allocator from large contiguous on-heap region obtained from the GC's allocator
  - Implemented as a `byte` array allocated via JNI—can detect collections by observing when the array is moved
  - Optimization—can be implemented via direct allocations, which avoids having the GC copy unnecessary data, but GC must inform code of when the region has been freed
- Hook JVM's **safepointing** mechanism to allow the accelerator to mark its thread as not being at a safepoint—causing the GC to wait for it to finish before performing stop-the-world pauses (in particular, evacuations cannot happen concurrently)
- Entire object graph of deserialized objects made visible to GC in one "atomic" operation by creating a thread-local **JNI handle** pointing to the root message.
  - Barrier integration not necessary—G1GC's barriers are for pointer writes to old-space and during concurrent marking, but HW never performs these writes

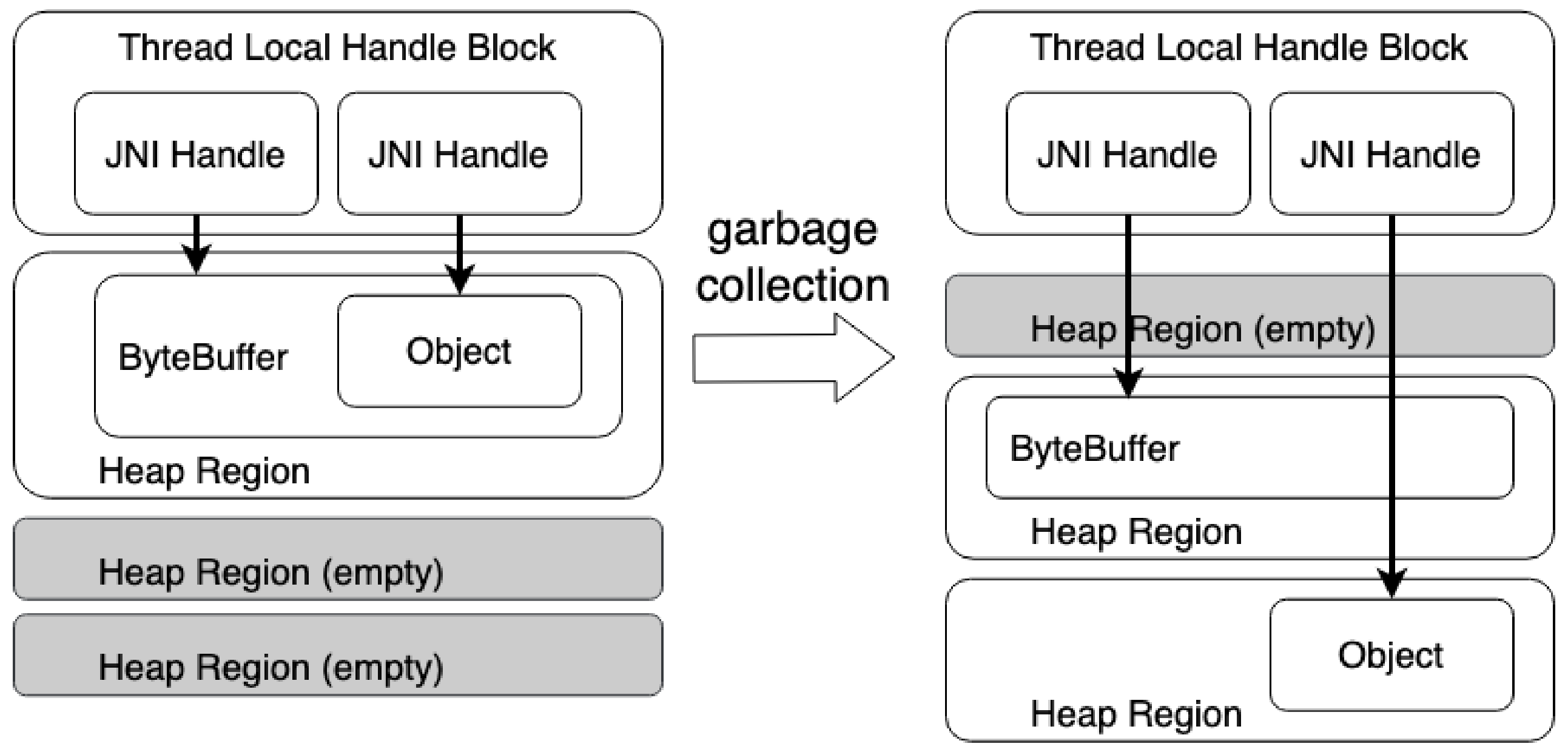

Figure 1: JNI Object Layout on Java Heap

Dynamically Sized Fields and Pointers

- All objects that have a known size when the field is first encountered are allocated into the "fixed size buffer"; nested messages are represented as pointer fields, and require the accelerator to allocate and initialize a new object
- `repeated` fields are represented as arrays but length is not known up front—accelerator keeps separate "flexible buffer" for allocating these array buffers
- Compared to C++, there is an additional intermediate (fixed-sized) object (`ArrayList`) that needs to be allocated for all fields backed by arrays
  - Allocated in the same manner as normal submessage objects

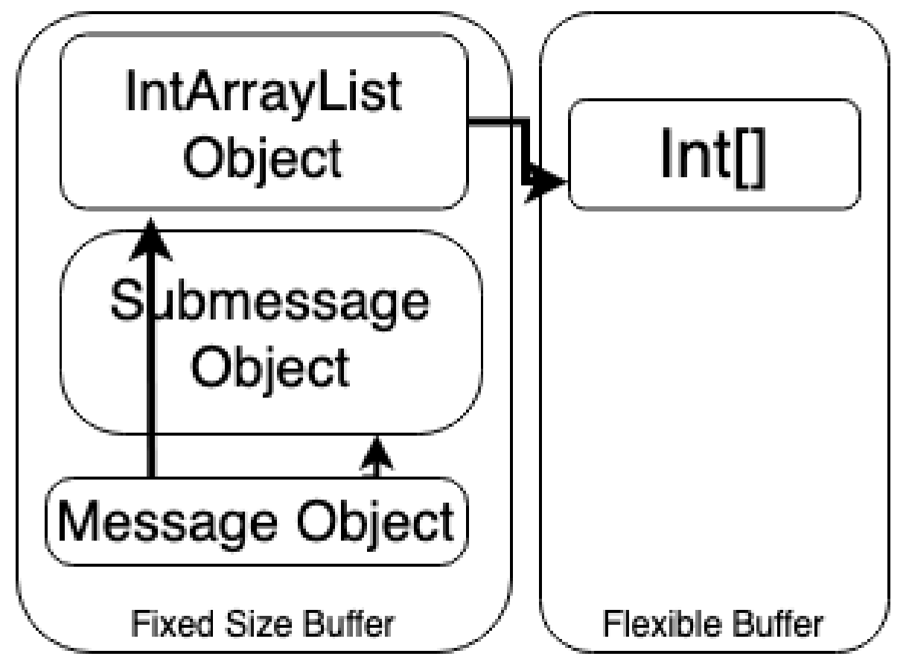

Figure 2: Example Deserialized Protobuf Object with Repeated and Submessage

Overview

- Serialization and deserialization contribute a significant overhead to datacenter workloads—5–6% among workloads at Google and Facebook
- We aim to accelerate deserialization for languages running on the Java Virtual Machine building upon previous hardware accelerators targeting C++
  - Deserialized objects must be **source-compatible** with the original Protobuf libraries—field types must remain as "native" Java objects (e.g. `java.util.List`)
  - We would like to expand upon this accelerator in a way that both preserves the original C++ compatibility and can accommodate more languages in the future
  - Aimed to reuse large pieces of hardware when possible. RTL updates are in blue (Figure 3)

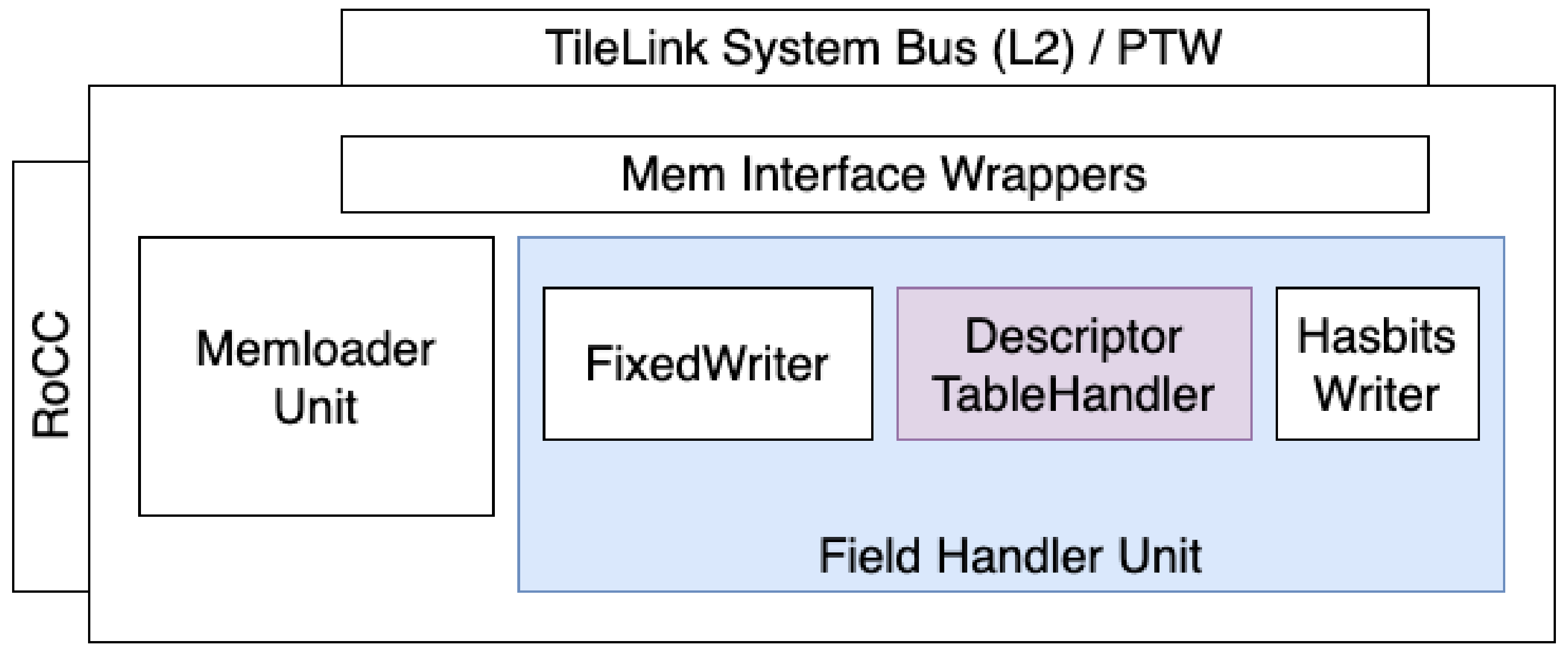

Figure 3: Accelerator Block Diagram

Accelerator Updates

- Many more fields need to be initialized in Java objects compared to C++—implemented generic constant-value-writer in **Accelerator Descriptor Table** handling to write arbitrary values that do not depend on the content of the incoming messages
  - Subobject pointers must be valid at all times (even when fields not present) because the GC will trace through them
  - Must be initialized to point to empty singleton objects or `null` to ensure the heap is not corrupt
- Object sizes and field offsets are not known at compile-time—perform one-time ADT construction at runtime for each message type being deserialized, via JNI calls to probe the JVM for object layout information
- Klass words (equivalent of C++ vtable pointers) are fixed throughout the lifetime of the JVM, but addresses of singleton objects are not (they are affected by GC)—must update these elements of tables whenever a collection occurs
- Object allocation state machines modified for the object format expected by Java (see left)

```
                             {
                              .object_size = sizeof(M1),
                              .min_field_no = 1,
message M1 {                  .fields = {
  optional M2 f1 = 1;           [0] = {.offset = offsetof(M1, f1_), .type = MESSAGE, .is_repeated = false,
  repeated int64 f2 = 2;               .submessage = &M2_descriptor},
}                               [1] = {.offset = offsetof(M1, f2_), .type = INT64,   .is_repeated = true,
                                       .submessage = nullptr},
                              },
                             }
```

Figure 4: Simplified Accelerator Descriptor Table Example

Verification

- Allocated 128MB of objects, forced an 8MB heap, to ensure that garbage collection was happening
  - Even after garbage collection happened, verified that was still able to change and use all fields
- Ensure that old → new pointers did not break (JVM keeps card table structure for keeping track of these)
- Ensure that GC is actually paused between safepoints by creating another thread which allocates a lot of memory, seeing if allocations hang

Benchmarks

- Evaluated using FireSim on a BOOM core running OpenJDK on Linux
- Messages for deserialization are representative samples of Google workloads
- Measured JNI overhead very small compared to most messages (leftmost bar, in Figure 6)
- Software deserializing of large messages is limited by CPU memory bandwidth (Figure 5)—accelerator has wider memory interface and can achieve greater peak bandwidth

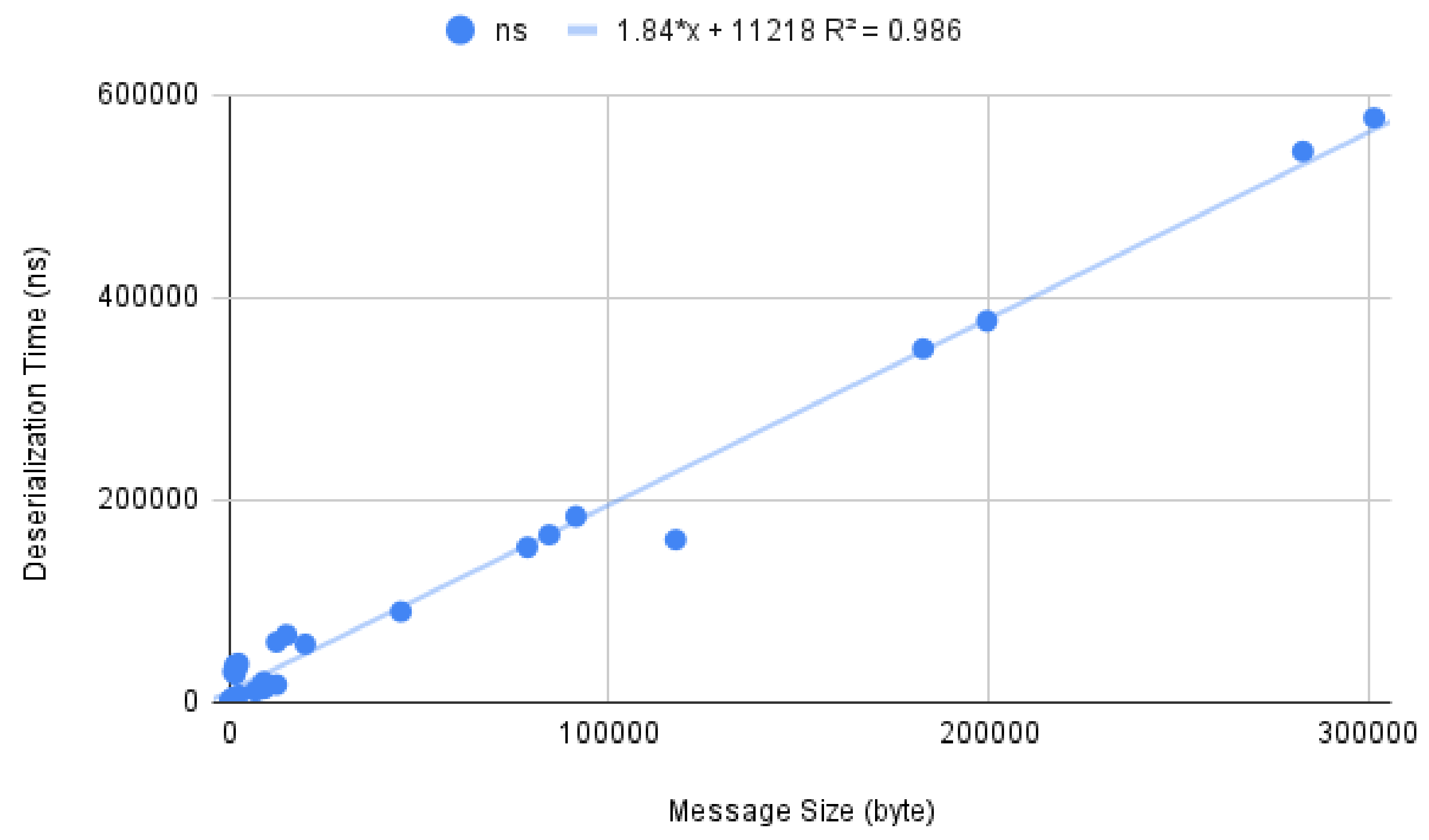

Figure 5: Baseline Deserialization Rate

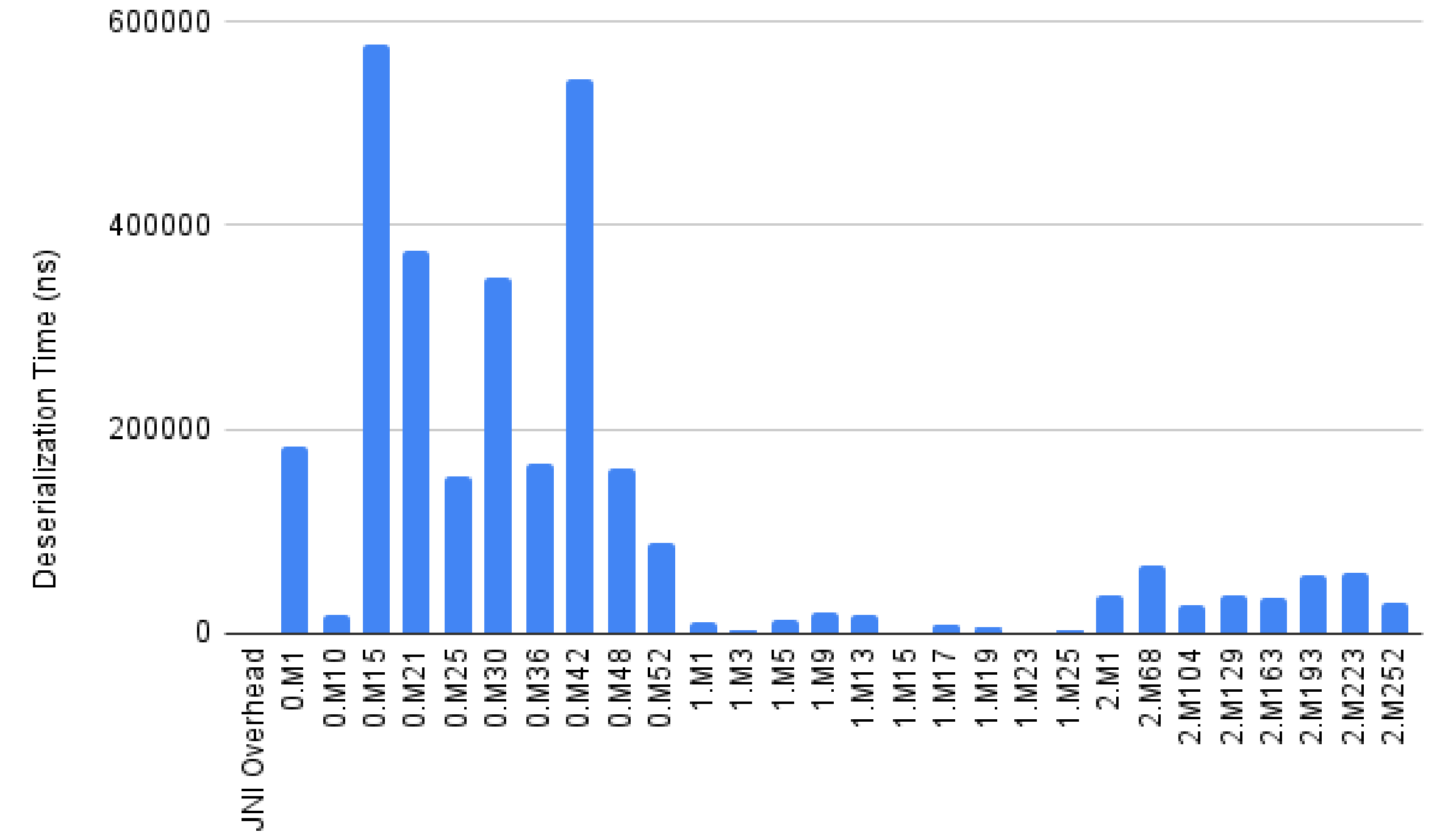

Figure 6: JNI Overhead vs. Java Deserialization for each message

University of California, Berkeley

Department of Electrical Engineering & Computer Sciences[1]