PackedObjects vs. MVT

April 2017

Dan Heidinga, Bjørn Vårdal

PackedObjects was a tech-preview introduced in IBM's J9 VM in the early releases of Java 8. The design of Packed was based on the following goals:
* Improve data locality by flattening ("packing") data structures,
* Allow "zero copy" access, particularly for off-heap data, and
* Apply a consistent approach to accessing both on and off heap data

When a PackedObject is created, all packed fields of the PackedObject were flattened (or "inlined") into a single data payload. This essentially removes the header for inlined fields and co-locates the data into a single object which improves locality by removing pointer chasing. PackedArrays are also treated in the same way - one header to describe the shape of the elements and all array elements inlined / flattened into the array body.

One of the key things Packed recognized is that the entity used to access a data payload (today's object header) didn't need to be attached to that payload (the fields). Instead, Packed took a page from sun.misc.Unsafe's addressing mode and used a base plus offset model for accessing all Packed entities.

Ironically, Packed got its improved data locality and zero copy access by increasing the size of a PackedObject header to include the base and offset. When a PackedObject is created, the base pointer points to itself and the offset points to the start of the data. When accessing an embedded PackedObject, a new header is created (!) with its base pointing to the owning (top level) object and the offset to the start of the embedded entities data. When the base is null, the offset is treated as a raw pointer into off-heap memory. See previous JVMLS talks for details [1]

This design means a packed reference still looks like a normal reference and can be used anywhere a normal reference would be. It just might have its data located anywhere in the address space.

Local variables and parameters are "references" regardless of their declared type. This also means that no change in behaviour is required at the astore/aload, getstatic/putstatic and call/return bytecodes in the JVM. Although MVT introduces new types and new bytecodes, a straightforward interpreter implementation will take a very similar approach of always working with boxes (or buffers) and leave the JIT to clean up hot paths.

PackedObjects allowed mutability as immutability prevents important use cases such as read/write of native memory structures. Users were free to opt-into immutability, but it wasn't mandatory. MVT requires immutability so it punts on off heap access, leaving that to the domain of Project Panama.

If users wanted the equals, hashCode, etc based solely on field values (ie: traditional value types behaviour), then in Packed, they could code that themselves. MVT takes the same approach though it leaves the door open to generating these methods in the future.

The semantics of common operations on traditional object references like identity, hashCode, synchronization, finalization, etc. are not well defined for packed object references because there may not be a reference pointer to each and every inlined instance. MVT has similar restrictions on the validity of using Object methods (some enforced, some not) and deals with the same concerns about accidental identity in the Box (VCC).

One of the areas Packed struggled with was making the assignment-by-value clear to the programmer. We prototyped using ':=' as a value copy operator which helped address the mismatch between the what the naive user expected and what they got. This was especially evident when dealing with Packed arrays as users were often confused about whether they were operating on a header that modified the array's backing storage or a copy of the element's data. In MVT, everything is a copy. We'll have to guard against the opposite user problem of forgetting to write their values back to the array storage / fields.

In Packed, there wasn't a good way to express nested field initialization in source code. When a Packed Object with embedded Packed objects was initially created, it was difficult to initialize the the fields of the embedded packed object without explicitly setting each individual field. The obvious answer is a model similar to C++'s placement new using an "init" method in place of a constructor call for the embedded packed fields as the storage has already been created in the enclosing object. Unfortunately, this area of investigation was never completed. We're unclear on whether MVT supports values embedded at values at this point.

Summary comparison:

	PackedObject	**MVT**
Identity	Accidental	Accidental
Mutability	Yes	No
Layout	Recursively flattened Static layout for off-heap data Arrays are inlined (flattened)	Flattened No layout requirements Arrays are references (Flattened arrays may be supported)

Off-heap / On-heap	Both	On-heap
Sharing	Explicit copy operations required Cursor to array	CoW CoR from arrays
Type safety	Yes	Yes
Type hierarchy	Yes, but either abstract or final	No
Language support	Minimal	Yes (eventually)

(Many thanks to John Duimovich as parts of this response were based on his blog post: https://duimovich.blogspot.ca/2012/11/packed-objects-in-java.html)

[1] = http://www.oracle.com/technetwork/java/jvmls2013sciam-2013525.pdf