Heap flattening for JEP 401

Heap flattening before JEP 401

From the beginning of the Valhalla project up to LW4, heap flattening was based on four properties of values types:

- lack of identity
- immutability
- non-atomicity (value tearing was allowed)
- null-freeness

The design and the optimizations around flat fields was relying heavily on those four properties.

The first main principle of the initial flat field design was the single layout. Fields of a value types were laid out by the `FieldLayoutBuilder` in a way to optimize their footprint and makes their inclusion in containers easy. This layout was then use in all containers: as a flat field embedded in another object, as the element of a flat array or as the content of a standalone heap allocated instance (or heap buffer) of this value type. Whatever container it is in, this "payload" of the value type has always the same size and the same alignment constraints. If the beginning of the payload is placed accordingly to its alignment constraint, it is guaranteed that all fields inside the payload are correctly aligned too.

The second principle of the initial flat field design was that content of the field could be accessed in any way. The interpreter and the runtime used to copy the whole payload from one container to the other. C1 used to copy either the whole payload or only one section of the payload from one container to the other. C2 was doing individual embedded field accesses, mostly ignoring the notion of payload as an entity, only seeing it as the succession of individual fields.

Changes introduced by JEP 401

JEP 401 still defines value class instances as being identity-free and immutable (past the construction phase). However, to be aligned with the philosophy that Java must be safe by default, value tearing is not allowed anymore (unless explicitly opt-out by the user). And null-freeness is not part of the JEP 401, which means that fields declared with a value class type are nullable and any flat encoding of such field must accommodate for the encoding of null in addition to the value itself.

Those new specifications add a lot of constraints to heap flattening.

Impact of atomicity

Preventing value tearing means all field accesses, reads and writes, must be performed atomically. Experiments done earlier in the Valhalla project showed that, currently, the only viable solution for atomic accesses is to use the instruction that are naturally atomic: 8, 16, 32 and 64 bits load and store on correctly aligned addresses. This has three effect on flat fields. First, the VM won't be able to flatten values bigger than 64 bits. Second, the space allocated to store a flat value has to be rounded up to the size of an atomic read/write instruction. For instance, even if a value containing an int and a boolean needs only 5 bytes of data, it requires an 8 bytes space in its container. The 3 padding bytes at the end cannot be used to store another field because it would create a dependency between those two fields, meaning accesses of those two fields would have to be synchronized somehow, to prevent resurrecting an old value (this issue would not exist for truly final fields, but such fields are not optimized to this point yet for other reasons). Third, when placed in to a container, flat values might now have a stronger alignment constraint. For instance, in the previous flattening model, a value made of three bytes had an alignment constraint of 1 (byte alignment) , meaning it could be placed anywhere 3 contiguous bytes were unused. With the atomicity requirement, the space required to store this value has to be increased to 4 bytes, and it has to be aligned on a 4 bytes boundary, just like an int, otherwise there's no guarantees that read and write will be atomic.

Impact of nullability

Supporting nullability means adding one bit of information to the content of a flat field, and in many case in means increasing the size of the flat representation of the field by one byte (the smaller piece of data the VM can address in a field layout with its current implementation). This increase in size can quickly become problematic considering the tight constraints cited in the previous section. A value class Point made of two ints would require 65 bits of data, rounded up to 72, to be represented in a flat form, which is currently not possible considering the size of the atomic operations the VM can use.

Trying to reuse a spare bit in one of the fields of the value to encode the null marker could be possible, but the cost and complexity of such encoding are still to be evaluated.

Old and new fields layouts

Field flattening is a VM decision, and the VM still has the choice to use a reference for a value field. It is called the `REFERENCE` layout.

Almost all our tests are using the old LW4 flattening model, where the size of the flat field is not limited by the size of atomic operations, where value tearing is allowed, and there's no need to encode null. This is also the kind of heap flattening providing the best performance on benchmarks (flat arrays of Point). The VM still supports LW4 style flattening, but the Friends&Family APIs are required to enable it. It is now called the `NON_ATOMIC` layout.

The first new layout is called `ATOMIC_FLAT`. It guarantees that the flat value can be read or updated atomically, but still requires the field to be null-free by be annotated with `@NullRestricted`. The maximum size of an atomic flat field is limited by the size of the atomic instructions supported on the platform.

The second new layout, and the one that can be used directly from JEP 401 without any addition, is the `NULLABLE_ATOMIC_FLAT` layout, which guarantees that the flat value can be read or updated atomically and has support for nullability. The maximum size of a nullable flat field (including its null marker) is limited by the size of the atomic instructions supported on the platform.

There's a third new layout, but it is a VM internal one and it is caller `PAYLOAD`. This is the layout used in standalone heap allocated instances (heap buffer). In order to simplify the implementation of the different flat fields access code, the `PAYLOAD` characteristics are the same as the ones of the most constraining layout supported by the class. This property allows direct copying between any flat layout and a heap buffered value.

All flat layouts share the same organization of fields: a given field is always at the same offset relatively to the first field offset. The differences between the layouts are related to the size, the alignment constraint and the presence of a null marker.

Generation and selection

When a value class is loaded, the VM determines which layouts the class can support based on multiple parameters from the class itself (size, annotations) and the VM configuration (VM flags, platform configuration). Many value classes won't be able to support all layouts described above. Availability of each layout, and layouts characteristics are stored in the `InlineKlass` instance of the class and can be seen with `-XX:+PrintInlineLayout`.

For instance, a class `Point` defined like this:

```
value class Point {
    int x = 0;
    int y = 0;
}
```

will have the following layout information:

```
Layout of class Point@0x7f03e8363760 extends java/lang/Object@0x7f03e804eb30
Instance fields:
 @0 RESERVED 12/-
 @12 PADDING 4/1
 @16 REGULAR 4/4 "x" I
 @20 REGULAR 4/4 "y" I
Static fields:
 @0 RESERVED 112/-
 @112 REGULAR 4/4 ".default" Ljava/lang/Object;
 @116 REGULAR 4/4 ".reset" Ljava/lang/Object;
Instance size = 24 bytes
First field offset = 16
Payload layout: 8/8
Non atomic flat layout: -/-
Atomic flat layout: 8/8
Nullable flat layout: -/-
---
```

Class `Point` doesn't support a non-atomic flat layout, it has an atomic flat layout with a size of 8 bytes and an alignment constraint of 8 bytes, and doesn't has a nullable flat layout. The characteristic of the payload layout are identical to the characteristics of the atomic flat layout because this is the only layout being supported.

When a container class declares a field with a value class type, the VM checks the requirements of the field (qualifiers, annotations), then selects among the layouts supported by the value class, the best layout fitting those requirements.

For instance, let's consider the value class `Value1` below and the container class `Container1` which declares 3 fields of type `Value1` with different requirements:

```
@ImplicitlyConstructible
@LooselyConsistentValue
value class Value1 {
    byte b0 = 0;
    byte b1 = 0;
    byte b2 = 0;
}

class Container1 {
    Value1 v0;
    @NullRestricted
    volatile Value1 v1;
    @NullRestricted
    Value1 v2;
}
```

This configuration will produce the following layouts:

```
Layout of class Value1@0x75f17859d360 extends java/lang/Object@0x75f1781357a0
Instance fields:
 @0 RESERVED 12/-
 @12 REGULAR 1/1 "b0" B
 @13 REGULAR 1/1 "b1" B
 @14 REGULAR 1/1 "b2" B
 @15 NULL_MARKER 1/1
Static fields:
 @0 RESERVED 112/-
 @112 REGULAR 4/4 ".default" Ljava/lang/Object;
 @116 REGULAR 4/4 ".reset" Ljava/lang/Object;
Instance size = 16 bytes
First field offset = 12
Payload layout: 4/4
Non atomic flat layout: 3/1
Atomic flat layout: 4/4
Nullable flat layout: 4/4
Null marker offset = 15
---

Layout of class Container1@0x75f17859d360 extends java/lang/Object@0x75f1781357a0
Instance fields:
 @0 RESERVED 12/-
 @12 FLAT 4/4 "v0" LValue1; Value1@0x75f17859d360 NULLABLE_ATOMIC_FLAT
 @16 FLAT 4/4 "v1" LValue1; Value1@0x75f17859d360 ATOMIC_FLAT
 @20 FLAT 3/1 "v2" LValue1; Value1@0x75f17859d360 NON_ATOMIC_FLAT
Static fields:
 @0 RESERVED 112/-
Instance size = 24 bytes
---
```

All `Container1`'s fields have the same type, but each one has a different layout.

Not just fields layouts

Computing fields layouts satisfying the new specifications is only one part of the feature. The second part is the code accessing those flat fields, and this is where the current Valhalla prototype requires a lot of additional work. LW4 had a single kind of flat field, but now the VM needs to be able to handle 3 different flat encodings as demonstrated above. And with each kind of flat layout, there's a precise semantic to be respected (atomicity, null marker). Those changes have to be implemented across all codes performing field accesses: interpreter, runtime, C1, C2, JNI, Unsafe, Graal.

Layout information encoding

There are two ways to retrieve the layout information for a given field.

The first one is to use the `FieldInfo` stream. If the field is marked as flat (`field_flags().is_flat()` is true), then it is possible to call the method `layout_kind()` to get the layout used for this field. The layout kind is expressed using the following enum:

```
enum LayoutKind {
 REFERENCE, // indirection to a heap allocated instance
 PAYLOAD, // layout used in heap allocated standalone instances, probably temporary for the transition
 NON_ATOMIC_FLAT, // flat, no guarantee of atomic updates, no null marker
 ATOMIC_FLAT, // flat, size compatible with atomic updates, alignment requirement is equal to the size
 NULLABLE_ATOMIC_FLAT, // flat, include a null marker, plus same properties as ATOMIC layout
 UNKNOWN // used for uninitialized fields of type LayoutKind
};
```

If the field has the NULLABLE_ATOMIC_FLAT layout, it is then possible to get the offset of the null marker by calling the `null_marker_offset()` method. The null marker offset is expressed as a regular field offset, meaning that it is relative to the beginning of the object when the value is buffered in the Java heap. In order to get the offset inside a flat field (relatively to the beginning of the flat field inside a container), the offset of the value's first field (`first_field_offset()`) must be subtracted.

The `FieldInfo` stream stores information in a very compact form, but it is costly to decode, making access to field information expensive when the VM needs it. The interpreter and the runtime often have to access field layout information in order to be able to perform the right access semantic. This is why flat field layout information are also stored in a second data structure, easier to access.

There was already a similar data structure in LW4. To perform access to a LW4 flat field, the VM needed the `InlineKlass` of the type of the field. Because flat field are stripped from their object header, this information was stored in a dedicated array attached to the `InstanceKlass` declaring this field: `_inline_type_field_klasses` . This array of `InlineKlass*` was accessed by the field's index and provided fast access to the methods handling value copying.

For JEP 401, this array has been replaced with an array of a more complete data struct: `InlineLayoutInfo` :

```
class InlineLayoutInfo : public MetaspaceObj {
 InlineKlass* _klass;
 LayoutKind _kind;
 int _null_marker_offset; // null marker offset for this field, relative to the beginning of the current container
}
```

`InlineLayoutInfo` provides a pointer to the `InlineKlass` of the field, but also the kind of the layout as well as a precomputed null marker offset for this field, relative to the beginning of the container object.

The new array, `_inline_layout_info_array`, is still accessed using the field's index. However, JEP 401 introduced field inheritance for value classes. So, in LW4 it was valid to simply use the class of the container object (the receiver of the `getfield` or `putfield` bytecode) to retrieve the array, because flat fields could only be declared in the container class. This assumption is not valid anymore because of field inheritance. The array containing the layout information about a field must be retrieved from the class that declared this field (holder class), which can be quickly found in the `ResolvedFieldEntry` of the field.

Access protocols

Each layout has its own access protocol that is detailed below.

- `REFERENCE` : this layout uses a pointer to a heap allocated instance (no flattening). When used, `field_flags().is_flat()` is `false` . The field can be nullable or null-restricted, in the later case, `field_flags().is_null_free_inline_type()` is true. In case of a null-restricted field, `putfield` and `putstatic` must perform a null-check before writing a new value. Still for null-restricted fields, if `getfield` reads a `null` pointer from the receiver, it means that the field was not initialized yet, and `getfield` must substitute the `null` reference with the default value of the field's class.
- `NON_ATOMIC_FLAT` : this layout is the former LW4 flattening. Any field embedded inside the flat field can be accessed independently. The field is null-restricted, meaning `putfield` must perform a null-check before performing a field update.
- `ATOMIC_FLAT` : this new flat layout is designed for atomic updates, with size and alignment that make use of atomic instructions possible. All accesses, reads and writes, must be performed atomically. The field is null-restricted, meaning `putfield` must perform a null-check before performing a field update.
- `NULLABLE_ATOMIC_FLAT` : this is the new flat layout designed for JEP 401. It is designed for atomic updates, with size and alignment that make use of atomic instructions possible. All accesses, reads and writes, must be performed atomically. The layout includes a null marker which indicates if the field's value must be considered as `null` or not. The null marker is a `byte`, with the value zero meaning the field's value is `null`, and a non-zero value meaning the field's value is not `null`. A `getfield` must check the value of the null marker before returning a value. If the null marker is zero, `getfield` must return the `null` reference, otherwise it returns the field's value read from the receiver. When a `putfield` writes a non-null value to such field, the update, including the field's value and the null marker, must be performed in a single atomic operation. If the source of the value is a heap allocated instance of the field's class, it is allowed to set the null marker to non-zero in the heap allocated instance before copying the value to the receiver (the `PAYLOAD` layout used in heap allocated values guarantees that the space for the null marker is included, but has no meaning for the heap allocated instance which is always non-null, and that the whole payload is correctly aligned for atomic operations). When a `putfield` writes `null` to such field, the null marker must be set to zero. However, if the field contains oops, those oops must be cleared too in order to prevent memory leaks. In order to simplify such operation, value classes supporting a `NULLABLE_ATOMIC_FLAT` layout have a pre-allocated `reset` value instance, filled with zeros, which can be used to simply overwrite the whole flat field and reset everything (oops and null marker). The `reset` value instance is needed because the VM needs an instance guaranteed to always be filled with zeros, and the `default` value could have its null marker set to non-zero if it is used as a source to update a `NULLABLE_ATOMIC_FLAT` field.

Transition

By default, only the `REFERENCE` and the `NON_ATOMIC_FLAT` layouts are enabled, because most of the VM lacks the code to correctly handle the other layouts (only the interpreter does in PR #1275). The `ATOMIC_FLAT` layout can be enabled with `-XX:+AtomicFieldFlattening`. The `NULLABLE_ATOMIC_FLAT` layout can be enabled with `-XX:+NullableFieldFlattening`.

Be aware that enabling the `NULLABLE_ATOMIC_FLAT` layout is going to trigger flattening in many classes of the JDK because of the migrated classes, and this can cause issues when some parts of the VM, like Unsafe, are not able to handle this layout yet.

Unsafe

In LW4, there were two ways to access a flat field with Unsafe. One way was to compute the offsets of individual primitive or reference fields embedded inside the flat field, and then use the regular Unsafe methods to read or write fields. This was allowed because flat value were not guaranteed to be atomic in LW4. The other way was to use two new Unsafe methods: `putValue()` and `getValue()` which take an additional parameter, compared to to other get and put methods, to pass the class of the flat field.

Those Unsafe ways to access flat fields don't work well with the new layouts introduced for JEP 401. The issue with the first method is that it breaks the atomicity of values that is supposed to be guaranteed for `ATOMIC_FLAT` fields and `NULLABLE_ATOMIC_FLAT` fields. The issue with the second method is that the arguments passed to the methods are the receiver, the field offset and the field class, and retrieving the flat field's layout kind (in order to apply the correct semantic) from those arguments is a very expensive operation (linear search for offset match across fields meta-data).

VarHandle

One outstanding issue between VarHandles and flat fields in LW4 was the support of the atomic operations (`getAndSet()` methods). It was not possible to guarantee the semantic of those methods while allowing non-atomic updates from the Unsafe API.

The new layouts in JEP 401 could help resolving this issue by restricting those atomic operations to flat fields layouts with atomicity guarantees. Practically, a VarHandle associated with an `ATOMIC_FLAT or a NULLABLE_ATOMIC_FLAT` field would support `getAndSet()` methods, while a VarHandle associated with a `NON_ATOMIC_FLAT` field would not.

Future layouts

It is possible that more layouts are added in the future. Notably, in order to optimize strict final nullable instance fields, a `NULLABLE_NON_ATOMIC_FLAT` layout might be needed, but because such layout comes with a lot of caveats, this work is not part of the initial implementation.