Atomic VarHandles

Paul Sandoz
Oracle

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.

It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Atomic VarHandles

- A ref to an underlying variable to safely perform atomic ops on
- Motivated by JEP 193 Enhanced Volatiles
- Inspired by MethodHandles

Current situation

- Some fields are **`volatile`**; loads/stores automatically fenced
- **`sun.misc.Unsafe`** methods
  - Fenced loads/stores
  - Atomic updates (CAS)
- **`j.u.c.atomic.Atomic*`** classes

The bad and the ugly

- **`Atomic*`** classes have overhead
  - Not used in **`j.u.concurrent`** classes
- **`sun.misc.Unsafe`** is...
  unsafe, not "portable", and going away
  - CAS is too important to relegate
- No unified/safe model for accessing on and off-heap

The unsafe situation

http://cr.openjdk.java.net/~psandoz/dv14-uk-paul-sandoz-unsafe-the-situation.pdf

- **`sun.misc.Unsafe`** ~~is going away~~ will be made in accessible
- A number of vertical use-cases
  - JEP 193 Enhanced Volatiles tackles one such use-case

JEP 193
Enhanced Volatiles

- **Safe**, **performant**, enhanced atomic access to field and array elements
- Replace nearly all usages of **`Unsafe`** in **`java.util.concurrent`** classes
- Depends on JEP 188 Java Memory Model Update

JEP 193
Success Metrics

- API should be at least as good as **`Unsafe`**
- Performance results should be close to **`Unsafe`**, and faster than **`Atomic*`** classes

Strawman Proposal

- Enable access to corresponding methods for fields using the **`.volatile`** prefix

```
class Usage {
  volatile int count;

  int incrementCount() {
    return count.volatile.
        incrementAndGet();
  }
}
```

Strawman Proposal

- The **`.volatile`** prefix introduces scope for atomic ops on "L-values"

- Scoped to interfaces with methods, **`VolatileRef`**, **`VolatileInt`** etc.

- Requires language changes

Strawman Proposal

- What should be the implementation?
- Would it be sufficient on its own?
- Is there a way do this without such language changes?

Alternative Proposals

- MethodHandles
- VarHandles

MethodHandles

- A **`MethodHandle`** is a reference to an underlying method, constructor, or field
  - Supports volatile access to a field
  - Invocations inline surprisingly well
- With some tweaks can support other forms of access

MethodHandles

- MethodHandles are cunning!

  "So cunning you could put a tail on it and call it a weasel"

- VarHandles are cunning too

  Marching with ill-deserved confidence in the direction of this presentation

Deconstructing MethodHandles

https://wiki.openjdk.java.net/display/HotSpot/Deconstructing+MethodHandles

MethodHandle

- Invocation is signature *polymorphic*
- Has a *method type descriptor*, that is compared to the *symbolic type descriptor*
- Has a *lambda form* that holds a *member name* characterizing the method to invoke

MethodHandle.invokeExact

```
@ 45   jvmls.varhandles.MHandles$Receiver::setValue

  @ 5    java.lang.invoke.LambdaForm$MH/<N>::invokeExact_MT
    @ 2    java.lang.invoke.Invokers::checkExactType
      @ 11   java.lang.invoke.MethodHandle::type

    @ 11 java.lang.invoke.LambdaForm$MH/<M>::putObjectVolatileFieldCast
      @ 1    java.lang.invoke.DirectMethodHandle::fieldOffset
      @ 6    java.lang.invoke.DirectMethodHandle::checkBase
        @ 1    java.lang.Object::getClass
      @ 13   java.lang.invoke.DirectMethodHandle::checkCast
        @ 5    java.lang.invoke.DirectMethodHandle$Accessor::checkCast
          @ 5    java.lang.Class::cast
            @ 6    java.lang.Class::isInstance
      @ 28   sun.misc.Unsafe::putObjectVolatile
```

Code in ForkJoinPool

```
final ForkJoinTask<?> poll() {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (((a.length - 1) & b) << ASHIFT) + ABASE;
        t = (ForkJoinTask<?>)U.getObjectVolatile(a, j);
        if (t != null) {
            if (U.compareAndSwapObject(a, j, t, null)) {
                U.putOrderedInt(this, QBASE, b + 1);
                return t;
            }
        }
        else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```

Code in ForkJoinPool

```
final ForkJoinTask<?> poll() throws Throwable {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (a.length - 1) & b;
        t = (ForkJoinTask<?>)ABASE_getVolatile.invokeExact(a, j);
        if (t != null) {
            if ((boolean) ABASE_compareAndSet.invokeExact(
                                a, j, t, (ForkJoinTask) null)) {
                QBASE_setRelease.invokeExact(this, b + 1);
                return t;
            }
        }
        else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```

MethodHandles: Success Metrics

✗ API should be at least as good as **`Unsafe`**, preferably better

✓ Performance results should be close to **`Unsafe`**, and faster than **`Atomic*`** classes

VarHandle

- Abstraction of **safe** access to a memory location
- Leveraging MethodHandle invoke intrinsics and **`sun.misc.Unsafe`**
  - "MethodHandles for data"

VarHandle: One abstract class

- Many *access-modes*, one per method
  - Fenced, atomic (CAS)
- Many *access-kinds*, one per instance
  - Static/instance field, array, off-heap
- Many *value-kinds*, one per instance
  - Object ref, primitive, "composite" value

Patches to langtools & hotspot

```
langtools $ hg diff --stat -r qparent
 src/share/classes/com/sun/tools/javac/code/Symtab.java  |    6 +
 src/share/classes/com/sun/tools/javac/code/Types.java   |   11 +-
 src/share/classes/com/sun/tools/javac/comp/Infer.java   |  152 ++++++++++++++--
 src/share/classes/com/sun/tools/javac/comp/Resolve.java |   18 +-
 4 files changed, 151 insertions(+), 36 deletions(-)
```

```
hotspot $ hg diff --stat -r qparent
 src/share/vm/ci/ciEnv.cpp                       |   5 ++-
 src/share/vm/classfile/systemDictionary.cpp     |   6 +-
 src/share/vm/classfile/systemDictionary.hpp     |   6 ++-
 src/share/vm/classfile/vmSymbols.hpp            |   3 +
 src/share/vm/interpreter/linkResolver.cpp       |  25 ++++++++++--
 src/share/vm/interpreter/linkResolver.hpp       |   1 +
 src/share/vm/interpreter/rewriter.cpp           |  23 ++++++++++-
 src/share/vm/prims/methodHandles.cpp            |  56 ++++++++++++++++++++++++++---
 src/share/vm/prims/methodHandles.hpp            |   5 ++-
 9 files changed, 111 insertions(+), 19 deletions(-)
```

VarHandles

```
public abstract class VarHandle extends BaseVarHandle {
...
    public final native
    @MethodHandle.PolymorphicSignature
    Object getVolatile(Object... args);
...
    public final native
    @MethodHandle.PolymorphicSignature
    boolean compareAndSet(Object... args);
...
}
```

Deconstructing VarHandles

http://cr.openjdk.java.net/~psandoz/varhandles/VarHandle-0.1.md

VarHandle

- Has methods for atomic operations
- Each operation is signature *polymorphic*
- Has a *method descriptor type* for each operation, that is compared to the *symbolic descriptor type*
- Has a *var form* that holds a *member name* for each operation characterizing the method to invoke

Code in ForkJoinPool

```
final ForkJoinTask<?> poll() {
    ForkJoinTask<?>[] a; int b; ForkJoinTask<?> t;
    while ((b = base) - top < 0 && (a = array) != null) {
        int j = (a.length - 1) & b;
        t = (ForkJoinTask<?>)ABASE.getVolatile(a, j);
        if (t != null) {
            if (ABASE.compareAndSet(a, j, t, (ForkJoinTask) null)) {
                QBASE.setRelease(this, b + 1);
                return t;
            }
        }
        else if (base == b) {
            if (b + 1 == top)
                break;
            Thread.yield(); // wait for lagging update (very rare)
        }
    }
    return null;
}
```

Performance

volatile-set then volatile-get

Benchmark	ns/op
ARFU_sub_types	17.613
ARFU_exact_types	15.108
field_getputfield	15.057
field_reflection	24.661
MH_invoke	15.069
unsafe	15.065
varHandle	15.079

http://cr.openjdk.java.net/~psandoz/varhandles/VarHandle-0.1.md

Performance
volatile-set then volatile-get

Benchmark	Insns per cycle	Stalled cycles per insn
ARFU_exact_types	1.76	0.26
unsafe	0.73	0.97

Performance
release-set then volatile-get

Benchmark	ns/op
ARFU_sub_types	10.088
ARFU_exact_types	7.064
unsafe	2.714
varHandle	3.230

http://cr.openjdk.java.net/~psandoz/varhandles/VarHandle-0.1.md

Performance
compare-and-set

Benchmark	ns/op
ARFU_sub_types	15.111
ARFU_exact_types	15.074
unsafe	11.560
varHandle	11.581

http://cr.openjdk.java.net/~psandoz/varhandles/VarHandle-0.1.md

VarHandles: Success Metrics

- ✓ API should be at least as good as **`Unsafe`**, preferably better
- ✓ Performance results should be close to **`Unsafe`**, and faster than **`Atomic*`** classes

VarHandles: Success Metrics

✓ API should be at least as good as **`Unsafe`**, preferably better

✗ No static type checking

✗ Casts required for **`null`** values

✗ Cast required for returned value type

Reified polymorphic signature methods

- One interface per *access-kind* shape

```
abstract class FieldHandle<R, V> {
...
    public final native
    @MethodHandle.PolymorphicSignature
    boolean compareAndSet(R r, V e, V a);
...
}

abstract class ArrayHandle<R, I, V> {
...
    public final native
    @MethodHandle.PolymorphicSignature
    boolean compareAndSet(A r, I i, V e, V a);
...
}
```

A cunning primitive plan?

- What about primitive value types?
- A boxed type parameter is associated with its primitive type at the call site
  - Compiler infers `int` to `Integer`
  - Atomic operations on refs to boxed type instances are likely rare

A cunning primitive plan?

```
FieldHandle<Receiver, Integer> FH_I =
    FieldHandles.findFieldHandle(Receiver.class, "i", int.class);


public boolean compareAndSetI(int actual, int expected) {
    // No casting required to boolean
    return FH_I.compareAndSet(this, actual, expected);
}


public compareAndSetI(II)Z
 L0
  LINENUMBER 83 L0
  GETSTATIC .../FHandles$Receiver.FH_I : Lj/l/i/FieldHandle;
  ALOAD 0
  ILOAD 1
  ILOAD 2
  INVOKEVIRTUAL j/l/i/FieldHandle.compareAndSet (L.../
FHandles$Receiver;II)Z
  IRETURN
```

VarHandles in Valhalla

- Is the cunning primitive plan too weaselly?
- Do this in Valhalla instead?

```
FieldHandle<R, any V>

ArrayHandle<R, any I, any V>
```

VarHandle Summary

- Safe access to a memory location
- Performance looks good
  - When C2 kicks-in
  - More measurements required

VarHandle Summary

- API improvements possible with some cunning
- Connections/synergies with
  - Project Valhalla
  - Project Panama
  - Arrays 2.0