Shenandoah is a low pause time garbage collector developed for OpenJDK.

Motivation:

All of the current OpenJDK garbage collectors have the limitation that they only compact the heap while the java threads are stopped. This is a fundamental limitation for interactive and high availability applications. These applications require faster response times then a stop the world evacuating garbage collector can achieve. One benchmark which addresses this concern is the SpecJBB critical JOps metric.

Goals:

1. Fit as seemlessly as possible into the OpenJDK source code base. We want Shenandoah to be a runtime choice just like the other garbage collectors.
2. Perform evacuation phases concurrently, thereby making pause times proportional to the size of the thread stacks, not the amount of live data.
3. Make our read/write barriers as efficient and unobtrusive as possible.

Heap Layout:

Heap layout is the same as G1s, we divide the heap into independently collectable regions.

Phases:

1. Init marking: Scans root set. (SATB protocol the same as G1s)
2. Concurrent marking: Mark the live objects.
3. Final marking: Rescan the root set and initiate concurrent evacuation.
4. Concurrent compaction: Evacuation of live objects from targeted regions.
5. Init marking:
6. Concurrent Marking + update references to point to new copies of object.
7. Final Marking + free now unused regions
8. Goto 4

Init marking, Conc Marking, and Final Marking are all shared with G1.

Concurrent Compaction:

There's no free lunch. Evacuating objects while the Java threads are running requires synchronization between the gc threads and the java threads so that they agree on the address of the active copy of an object. As with most hard problems in computer science this issue may be solved by adding a level of indirection. All references to an object go through an indirection pointer. Thus when an object moves, simply by updating the indirection pointer, all references to the object will be updated atomically. Shenandoah adds this indirection pointer immediately preceding an object in memory. This allows us to run with the same object layout as the other garbage collectors. It also allows us to dereference the forwarding pointer simply by reading the address before the address of the object we are interested in.

All object uses go through this indirection pointer. In most cases it is an L1 or L2 cache hit and is far enough removed from the actual use of the object that it's very inexpensive.

When the garbage collector copies an object it follows the same protocol as the g1 and parallel collector. It makes a speculative copy of the object in a to-region, and then performs an atomic compare and swap (CAS) instruction to update the indirection pointer to point to the speculative copy. A successful CAS marks the speculative copy as the now official copy and all uses of the object will use the new copy.

Reading a Field:

Here's a snippet of assembly code for reading a field:

```
0x00007fffe1102cd1: mov    0x10(%rsi),%rsi    ;*getfield value
                                               ; - java.lang.String::equals@20 (line 982)
```

Here's what the snippet looks like with Shenandoah:

```
0x00007fffe1102ccd: mov    -0x8(%rsi),%rsi             read the contents of the indirection pointer
                                                        for the address contained in register
                                                        rsi back into rsi.
0x00007fffe1102cd1: mov    0x10(%rsi),%rsi    ;*getfield value
                                               ; - java.lang.String::equals@20 (line 982)
```

Writing a Field:

If the GC copies an object while the Java threads are updating there is the potential to miss an update. To avoid this, Shenandoah requires an invarient that all writes occur on to-space copies of objects. If a Java thread writes to an object which is in a region targeted for compaction, then it must first copy the object out of the targeted region and then write to the new copy. Our write barriers consist of a test to see if evacuation is in progress and then a call out to a shared stub that implements the fast path of the write barrier inline, and only calls out to the runtime in slow cases (i.e. full gclab etc). You can see an earlier version which always calls out to the runtime_call write barrier in this snippet of code:

```
0x00007fffe1110318: movabs $0x7fffec0b92c0,%rax
0x00007fffe1110322: mov    (%rax,%rbx,1),%al
0x00007fffe1110325: test   $0x1,%al                          ← evacuation in progress?
0x00007fffe1110328: je     0x00007fffe1110339                ← if not jump to putfield
0x00007fffe111032e: xchg   %rdi,%rax
0x00007fffe1110331: callq  0x00007fffe10ffd20  ;  {runtime_call} ← else make a call out to the
                                                                     runtime to copy the object to
                                                                     an evacuation region.
0x00007fffe1110336: xchg   %rax,%rdi
0x00007fffe1110339: mov    %esi,0x10(%rdi)    ;*putfield count
                                      ; - java.util.Hashtable::addEntry@83 (line 436)
```

These specialized code snippets are only emitted when you are running with -XX:+UseShenandoahGC!!!

Implementation Costs:

This indirection pointer has a cost in both space and time. The space cost is obviously one word/object. The time cost is more complicated. Not every use of an object requires a read of the indirection pointer. For example if you read two fields of an object, the second one will not require resolving the indirection pointer. Even when our optimization passes don't eliminate read barriers, the C2 code generator is smart enough to emit them such that the code execution is rarely waiting for the resolution of an indirection pointer.

Source Code Changes:

The bulk of the Shenandoah code is in a separate GC implementation directory. This follows the same convention as all of the other OpenJDK collectors. We've moved some of the G1 code out of the G1 directory and into the GC shared directory so that we may make use of it (SATBMarkQueue, CMBitMap, ParallelCleaning and in the future maybe StringDedup).

The indirection pointers require adding new barriers to ithe interpreter, c1, and c2. We've tried to make these changes as cleanly as possible. Note: our changes will make it easier in the future to experiment with other collectors which require more sophisticated barriers than are currently available in hotspot. You could also imagine using a read barrier to move objects closer to their executing thread in a NUMA enviornment, or to implement a software transactional memory system.

We have ammended the BarrierSet class to include the following methods:

```
  virtual oop read_barrier(oop src) ;
  virtual oop write_barrier(oop src);
  virtual bool obj_equals(oop obj1, oop obj2);
  virtual bool obj_equals(narrowOop obj1, narrowOop obj2);
  virtual void interpreter_read_barrier(MacroAssembler* masm, Register dst);
  virtual void interpreter_read_barrier_not_null(MacroAssembler* masm, Register dst) ;
  virtual void interpreter_write_barrier(MacroAssembler* masm, Register dst);
  virtual void asm_acmp_barrier(MacroAssembler* masm, Register op1, Register op2) ;
```

The read and write barriers implement the protocol described above. The obj_equals method ensures that the comparison occurs on the to-space copies of objects. The interpreter_read_barrier methods are used in the template interpreter. The acmp barrier is similar to the obj_equals barrier ensuring that the acmp occurs on to-space copies of objects.

These methods are used in many places throughout the vm to ensure that our barrier paradigm is enforced. They are no-ops when running with other collectors. We have measured their execution cost and have found so significant overheap when running G1 with and without Shenandoah code.

Interpreter read barriers:

Interpreter read barriers are emitted through a call out to the barrier_set.

From templateTable_x86.cpp

```
void TemplateTable::getfield_or_static(int byte_no, bool is_static, RewriteControl rc) {
...
  if (!is_static) pop_and_check_object(obj);
  oopDesc::bs()->interpreter_read_barrier_not_null(_masm, obj);
  const Address field(obj, off, Address::times_1, 0*wordSize);
…
```

The code in red does nothing unless you are running Shenandoah.

C1 Read Barriers:

C1 Read Barriers simply are emitted through a method call.

```
void LIRGenerator::do_LoadField(LoadField* x) {
  bool needs_patching = x->needs_patching();
  bool is_volatile = x->field()->is_volatile();
  BasicType field_type = x->field_type();
...
  obj = shenandoah_read_barrier(obj, info, x->needs_null_check() && x->explicit_null_check() !=
NULL);
…
```

Where shenandoah_read_barrier equals

```
LIR_Opr LIRGenerator::shenandoah_read_barrier(LIR_Opr obj, CodeEmitInfo* info, bool
need_null_check) {
  if (UseShenandoahGC) {

    LabelObj* done = new LabelObj();
    LIR_Opr result = new_register(T_OBJECT);
    __ move(obj, result);
    if (need_null_check) {
      __ cmp(lir_cond_equal, result, LIR_OprFact::oopConst(NULL));
      __ branch(lir_cond_equal, T_LONG, done->label());
    }
    LIR_Address* brooks_ptr_address = generate_address(result, BrooksPointer::BYTE_OFFSET,
T_ADDRESS);
    __ load(brooks_ptr_address, result, info ? new CodeEmitInfo(info) : NULL, lir_patch_none);

    __ branch_destination(done->label());
    return result;
  } else {
    return obj;
  }
}
```

C2 Read Barriers:

We added two new ideal graph nodes to C2: ShenandoahReadBarrierNode and

ShenandoahWriteBarrierNode.

The ShenandoahReadBarrierNode gets inserted at parse time.

```
void Parse::do_get_xxx(Node* obj, ciField* field, bool is_field) {
  // Does this field have a constant value?  If so, just push the value.
...

  // Insert read barrier for Shenandoah.
  if (! ShenandoahOptimizeFinals || (! field->is_final() && ! field->is_stable())) {
    obj = shenandoah_read_barrier(obj);
  }

  Node *adr = basic_plus_adr(obj, obj, offset);
...
```

By adding our own nodes to the ideal graph we can perform analysis in later compiler phases which allows us to elide some of the barriers inserted at parse time.

We have currently implemented code generation for X86 and started on AARCH64. For X86 we include the following read barrier code generation for x86_64.ad:

```
instruct shenandoahRB(rRegP dst, rRegP src, rFlagsReg cr) %{
  match(Set dst (ShenandoahReadBarrier src));
  effect(DEF dst, USE src);
  ins_cost(125); // XXX
  format %{ "shenandoah_rb $dst, $src" %}
  ins_encode %{
    Register d = $dst$$Register;
    Register s = $src$$Register;
    __ movptr(d, Address(s, -8));
  %}
  ins_pipe(ialu_reg_mem);
%}
```

Runtime system read barrier example:

There are several places in the runtime where the vm accesses objects in the heap. These accesses must be protected by read barriers as well. Here is an example:

```
ClassLoaderData* java_lang_ClassLoader::loader_data(oop loader) {
  loader = oopDesc::bs()->read_barrier(loader);
  return *java_lang_ClassLoader::loader_data_addr(loader);
}
```