Memory Allocation in JikesRVM

The way that objects are allocated in JikesRVM can be difficult to grasp for someone new to the code base. This document provides a detailed look at some of the paths through the JikesRVM - MMTk interface code to help bootstrap understanding of the process. The process and code illustrated below is current as of March 2011, svn revision 16052 (between JikesRVM 3.1.1 and 3.1.2).

Memory Manager Interface

The best starting place to understand the allocation sequence is in the class org.jikesrvm.mm.mminterface.MemoryManager, which is a facade class for the MMTk allocators. MMTk provides a variety of memory management plans which are designed to be independent of the actual language being implemented. The MemoryManager class orchestrates the services of MMTk to allocate memory, and adds the structure necessary to make the allocated memory into Java objects.

The method *allocateScalar* is where all scalar (ie non-array) objects are allocated. The parameters of this method specify the object to be allocated in sufficient detail that when this method is compiled by the opt compiler, all of the parameters are compile-time constants, allowing maximum optimization. Working through the body of the method,

```
Selected.Mutator mutator = Selected.Mutator.get();
```

As mentioned above, MMTk provides many different memory management plans, one of which is selected at build time. This call acquires a pointer to the thread-local per-mutator component of MMTk. Much of MMTk's performance comes from providing unsynchronized thread-local data structures for the frequently used operations, so rather than provide a single interface object, it provides a per-thread interface object for both mutator and collector threads.

```
allocator = mutator.checkAllocator(org.jikesrvm.runtime.Memory.alignUp(size, MIN_ALIGNMENT), align,
allocator);
```

An MMTk plan in general provides several spaces where objects can be allocated, each with their own characteristics. JikesRVM is free to request allocation in any of these spaces, but sometimes there are constraints only available on a per-allocation basis that might force MMTk to override JikesRVM's request. For example, JikesRVM may specify that objects allocated by a particular class are allocated in MMTk's non-moving space. At execution time, one such object may turn out to be too large for allocation in the general non-moving space provided by that particular plan, and so MMTk needs to promote the object to the Large Object Space (LOS), which is also non-moving, but has high space overheads. This call will generally compile down to 0 or a small handful of instructions.

```
Address region = allocateSpace(mutator, size, align, offset, allocator, site);
```

This calls a method of MemoryManager, common to all allocation methods (for Arrays and other special objects), that calls

```
Address region = mutator.alloc(bytes, align, offset, allocator, site);
```

to actually allocate memory from the current MMTk plan.

```
Object result = ObjectModel.initializeScalar(region, tib, size);
```

Now we call the JikesRVM object model to initialize the allocated region as a scalar object, and then

```
mutator.postAlloc(ObjectReference.fromObject(result), ObjectReference.fromObject(tib), size, allocator);
```

we call MMTk's *postAlloc* method to perform initialization that can only be performed after an object has been initialized by the virtual machine.

Compiler integration

The *allocateScalar* method discussed above is only actually called from one place, the method *resolvedNewScalar(int ...)* in the class org.jikesrvm.runtime.RuntimeEntrypoints. This class provides methods that are accessed directly by the compilers, via fields in the org.jikesrvm.runtime.Entrypoints class. The 'resolved' part of the method name indicates that the class of object being allocated is resolved at compile time (recall that the Java Language Spec requires that classes are only loaded, resolved etc when they are needed - sometimes it's necessary to compile code that performs classloading and then allocate the object).

RuntimeEntrypoints also contains an overload, *resolvedNewScalar(RVMClass)*, that is used by the reflection API to allocate objects. It's instructive to look at this method, as it performs essentially the same operations as the compiler when compiling the call to *resolvedNewScalar(int ...)*.

Working backwards from this point requires delving into the individual compilers.

Baseline Compiler

There is a different baseline compiler for each architecture. The relevant code in the baseline compiler for the ia32 architecture is in the class org.jikesrvm.compilers.baseline.ia32.BaselineCompilerImpl. The method *emit_resolved_new(RVMClass)* is responsible for generating code to

execute the 'new' bytecode when the target class is already resolved. Looking at this method, you can see it does essentially what the *resolvedNewScalar(RVMClass)* method in RuntimeEntrypoints does, then generates Intel machine code to perform the call to the resolvedNewScalar entrypoint. Note how the work of calculating the size, alignment etc of the object is performed *by the compiler*, ***at compile time***.

Similar code exists in the PPC baseline compiler.

Optimizing Compiler

The optimizing compiler is paradoxically somewhat simpler than the baseline compiler, in that injection of the call to the entrypoint is done in an architecture independent level of compiler IR. (An overview of the JikesRVM optimizing compiler can be found in [1]).

In HIR (the high-level Intermediate Representation), allocation is expressed as a 'new' opcode. During the translation from HIR to LIR (Low-level IR), this and other opcodes are translated into instructions by the class org.jikesrvm.compilers.opt.hir2lir.ExpandRuntimeServices. The method *perform(IR)* performs this translation, selecting particular operations via a large switch statement. The NEW_opcode case performs the task we're interested in, doing essentially the same job as the baseline compiler, but generating IR rather than machine instructions. The compiler generates a 'call' operation, and then (if the compilation policy decides it's required) inlines it.

At this point in code generation, all the methods called by RuntimeEntrypoints.resolvedNewScalar(int...) which are annotated @Inline are also inlined into the current method. This inlining extends through to the MMTk code so that the allocation sequence can be optimized down to a handful of instructions.

It can be instructive to look at the various levels of IR generated for object allocation using a simple test program and the OptTestHarness utility described elsewhere in the user guide.

[1] The Jalapeño Dynamic Optimizing Compiler for Java

- Michael Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio Serrano, V.C. Sreedhar, and Harini Srinivasan.
- "1999 ACM Java Grande Conference", San Francisco, June 12-14, 1999.
- "'(Source code available as of version 2.0.0 of Jikes RVM.)'"

The class