PerformanceTechniques

Performance techniques used in the Hotspot JVM

What code shapes does the JVM optimize best? Here is a list.

Knowing these optimizations may help language implementors generate bytecodes that run faster. Basic information about bytecodes is in Chapter 7 of the JVM Spec..

In the context of this discussion, "compiler" = "JIT", and usually more specifically refers to the server (or final tier) JIT.

Getting solid performance measurements is surprisingly tricky. Take extra care with micro-benchmarks.

See below for a list of relevant presentations and papers.

Here is another slice on the same information, from another angle: PerformanceTacticIndex.

Constants

- Use constants when you can.
- It's OK to store them in local variables; the SSA representation tends to keep them visible.
- It's OK to store them in static final fields, too.
- Static final fields can hold non-primitive, non-string constants.
- The class of a static final field value is also constant, as is the array length (if it's an array).

Low-level safety checks

- Null checks are cheap. They usually fold straight into a related memory access instruction, and use the CPU bus logic to catch nulls. (Deoptimization follows, with regenerated code containing an explicit check.)
- User-written null checks are in most cases functionally identical to those inserted by the JVM.
- Null checks can be hoisted manually, and suppress implicit null checks in dominated blocks.
- Similar points can be made about other simple predicates, like class checks and range checks.
- All such checks, whether implicit or manually written, are aggressively folded to constants.

Loops

- The server compiler likes a loop with an int counter (int i = 0), a constant stride (i++), and loop-invariant limit (i <= n).
- Loops over arrays work especially well when the compiler can relate the counter limit to the length of the array(s).
- For long loops over arrays, the majority of iterations are free of individual range checks.
- Loops are typically peeled by one iteration, to "shake out" tests which are loop invariant but execute only on a non-zero tripcount. Null checks are the key example.
- If a loop contains a call, it is best if that call is inlined, so that loop can be optimized as a whole.
- A loop can have multiple exits. Any deoptimization point counts as a loop exit.
- If your loop has a rare exceptional condition, consider exiting to another (slower) loop when it happens.

Profiling

Profiling is performed at the bytecode level in the interpreter and tier one compiler. The compiler leans heavily on profile data to motivate optimistic optimizations.

- Every null check site has a record of whether a null was ever seen.
- Similar points can be made about other low-level checks.
- Every call site with a receiver has a record of which types were encountered (up to 2-3 types).
- There is also a type profile for every checkcast, instanceof, and aastore. (Helps with generics.)
- Every call site and branch point has a record of execution counts.

Deoptimization

Deoptimization is the process of changing an optimized stack frame to an unoptimized one. With respect to compiled methods, it is also the process of throwing away code with invalid optimistic optimizations, and replacing it by less-optimized, more robust code. A method may in principle be deoptimized dozens of times.

- The compiler may stub out an untaken branch and deoptimize if it is ever taken.
- Similarly for low-level safety checks that have historically never failed.
- If a call site or cast encounters an unexpected type, the compiler deoptimizes.
- If a class is loaded that invalidates an earlier class hierarchy analysis, any affected method activations, in any thread, are forced to a safepoint and deoptimized.
- Such indirect deoptimization is mediated by the dependency system. If the compiler makes an unchecked assumption, it must register a checkable dependency. (E.g., that class Foo has no subclasses, or method Foo.bar is has no overrides.)

Methods

- Methods are often inlined. This increases the compiler's "horizon" of optimization.
- Static, private, final, and/or "special" invocations are easy to inline.
- Virtual (and interface) invocations are often demoted to "special" invocations, if the class hierarchy permits it. A dependency is registered in case further class loading spoils things.

- Virtual (and interface) invocations with a lopsided type profile are compiled with an optimistic check in favor of the historically common type (or two types).
- Depending on the profile, a failure of the optimistic check will either deoptimize or run through a (slow) vtable/itable call.
- On the fast path of an optimistically typed call, inlining is common. The best case is a de facto monomorphic call which is inlined. Such calls, if back-to-back, will perform the receiver type check only once.
- In the absence of strong profiling information, a virtual (or interface) call site will be compiled in an agnostic state, waiting for the first execution to provide a provisional monomorphic receiver. (This is called an "inline cache".)
- An inline cache will flip to a monomorphic state at the first call, and stay in that state as long as the exact receiver type (not a subtype) is repeated every time.
- An inline cache will flip to a "megamorphic" state if a second receiver type is encountered.
- Megamorphic calls use assembly-coded vtable and itable stubs, patched in by the JVM. The compiler does not need to manage them.

Intrinsics

There are lots of intrinsic methods. See library_call.cpp and vmSymbols.hpp.

- Object.getClass is one or two instructions.
- Class.isInstance and Class.isAssignableFrom are as cheap as instanceof bytecodes when the operands are constants, and otherwise no more expensive than aastore type checks.
- Most single-bit Class queries are cheap and even constant-foldable.
- Reflective array creation is about as cheap as newarray or anewarray instructions.
- Object.clone is cheap and shares code with Arrays.copyOf (only recently!).
- *...Need much more here...*

Miscellanous

- Use a disassembler if available to inspect the generated code.
- Switches are profiled but the profile information is poorly used. For now, consider building an initial decision tree if you know one or two cases are really common.
- Exception throwing compiles to a goto, *if* the thrower and catcher inline together. For such uses, rely on preallocated or cloned exceptions, or override the **fillInStackTrace** part of exception creation, which is an expensive, reflective native call.
- Do not use jsr/ret. Just clone your finally code if you have to.
- If you are compiling a non-Java language, consider using standard mangling conventions.
- If you are generating almost the same class many times in a row, with small variations, factor out the repeating parts into a superclass or static helper class.
- For small variations in the remaining part, consider using a single named class as a template and loading it multiple times as an anonymous class with constant pool edits. Anonymous classes load and unload faster than named ones.

Presentations and Papers

- Buckley & Rose, Towards a Universal VM, Devoxx 2009. Includes summary of HotSpot optimizations.
- Schwaighofer, Tail Call Optimization in the Java HotSpot VM, Master's thesis, Johannes Kepler University Linz, 2009. Pages 23-35 are a good introduction to HotSpot architecture.