CS143 Handout #34
Autumn 2000

Some thoughts on Java

I keep yakking on about Java in class, I figured I owed you a little written info on some of those Java features I keep referring to. Since SOOP is based on C++/Java, the syntax and design is pretty similar to what you have been implementing this quarter. Java is a hybrid procedural/object-oriented language using a combination of static and dynamic typing, and supporting single inheritance where methods default to dynamic dispatch. Because of its nature for use in distributed programs over networks, Java has a very aggressive model toward safety and security and portability, often having to sacrifice some efficiency to ensure those goals.

Just in time code generation

When Java was first designed, it was designed to be interpreted by the Java Virtual Machine on every platform. However, performance suffers under this approach. Java's most significant impedance to performance can be found in its virtual machine model. Java uses a stack machine, therefore the bytecodes are stack instructions. As an intermediate representation, stack instructions are very compact, and therefor nice to send over a network. However, they prevent numerous optimizations that would otherwise be available on to a modern optimizing compiler, such as:

Instruction scheduling refers to reordering instructions to make more efficient use of the machine's resources. Such as reordering loads and stores to hide their latency. For example, on a typical RISC architecture, a load takes 3 cycles before the value it loads can be used. So ideally several loads are put in a row to avoid waiting for the first load's result to become available. But Java bytecode instructions cannot be reordered like this. Why not?

Operator strength reduction. The Java bytecode specification doesn't contain a bit-shift operation, so multiplies cannot be reduced to a shift. The addition of a bit-shift operation to Java might cause problems that aren't normally encountered when adding features to another language, however.

Register coloring A stack machine has no registers, therefore no register allocation can be done on the bytecode. Register allocation is probably the single most important optimization a compiler can perform. Without good register allocation that high-performance Alpha or UltraSparc3 has no chance to blow your socks off.

Further, the secure Java runtime model creates performance barriers that cannot be optimized away prior to the program's actual execution. For example, unlike C, Java requires all array references to bounds checked, to make sure that the array reference doesn't access memory that it shouldn't. However, consider the following code:

```
int a[10];

for (int i = 0; i < 10; i++)
   a[i] = 1;
```

It is clear that there are never any illegal array references here. However, the decision to eliminate the checks for illegal references cannot be made by the compiler translating from Java source to java bytecode because the verifier will not allow it.

Another run-time check that must be performed is checking for illegal access to a private member variable, as in the following code:

```
class Foo {
   private int bar = 4;
   public int barValue() { return bar; }
}

Foo someFoo = new Foo();
int z = someFoo.barValue();
```

Setting z to bar involves a function call, which is comparatively slow. Wouldn’t it be faster to just write:

```
int z = someFoo.bar;
```

But the bar value has been declared private and thus is not directly accessible! Translating the code to an inline version that access the variable directly will be flagged as an error by the verifier.

There are several methods for addressing Java performance problems. All have their advantages and disadvantages. The first is to compile Java to a native executable and distribute that rather than the Java bytecode version. What are the advantages and disadvantages of that approach? What about having the destination machine compile the program completely before executing it instead?

The browser you use today probably uses a just-in-time code generation (these are often called just-in-time compilers, but to be precise, it is really *code generation* that is being performed on the fly). “Just in time code generation” translates Java byte codes to target machine code before/during execution, rather than interpreting each instruction one by one.

Typical JIT compilers don’t compile the entire program or file as it is downloaded. They compile the program function by function, as the function is executed. So it is conceivable that only half of the program is ever compiled for your local machine. This provides one major time-benefit. The second is that having the program compiled into native machine code gives the compiler a chance to optimize on an IR that isn’t as restrictive as the bytecode.

Just-in-time code generation provides an interesting environment for optimization, because the calculation of “what is faster” changes dramatically. Consider the following scenario: Turning on optimization for program X will give it a 10% speed up, but will require an extra 25% of compilation time. Is turning on optimization a good move if we are talking about a C compiler? What about a Java JIT code generator?

Optimizations that take a long time to perform for proportionally smaller benefits cannot be performed and still have a Java application “feel” fast. Many JIT compilers do only a quick and dirty job and likely only consider local optimizations—optimizations within basic blocks—because data-flow analysis is time-consuming. However, this restriction prohibits some pretty important optimizations we have seen.

Optimizing with profiling information

The latest of the Java JITs take an interesting approach to code generation and optimization. Rather than generating code for everything (quickly and without much effort toward optimization), the VM first gathers runtime profiling information. Once it is more clear what code passages are heavily traveled, the JIT can invest time in aggressively optimizing those passages and ignoring the others. Thus concentrating the efforts on those areas most likely to pay off.

This technique (of using profiling information to assist optimizaion) also applies to other compilers, not just Java. Here is how it works: As program is compiled, the compiler inserts code to report certain things, such as:

• How often is this boolean expression true?

• How long does this loop typically take?

• Does this load hit in the cache more often than not, and so on.

The program is then recompiled and optimized using this information, which can result in much faster code than optimizing without profiling information. This works by making the common case fast. Here is a classic example. Consider the following code:

```
class Parent {
   virtual void Binky() {//do something};
}
class Child1 : public Parent{
   virtual void Binky() {//do something else};
}

class Child2 : public Parent{
   virtual void Binky() {//do something else};
}

Parent objs[100000]; .
..//a bunch of code that sets some if a to parents and some to children
for (int i = 0; i < 100000; i++)
   objs[i].Binky();
```

Because Binky is virtual, the compiler code must treat all calls to Binky as dynamically bound. Which means the vtable look up and slow going. Furthermore, Binky cannot be inlined. However, if the compiler knows that 90% of the time the objects in the array are of type Parent and not the children classes, it can insert code to special case the situation which looks something like this:

```
for (int i = 0; i < 1000000; i++)
{
  if (objs[i] is a parent object)
     Parent::Binky without virtual dispatch, possibly even inline
  else
     objs[i].Binky() with virtual dispatch
}
```

Even with the extra if statement, the resulting code can be much faster. It is possible to think of many other situations that can be special cased to make the normal case fast, if only the compiler knows what the common case is. Can you think of others? However, use of profiling information for optimization is not a panacea for all performance issues. What drawbacks are there to these techniques? Why might you not want to code this way by hand?

Bibliography

B. Venners, Inside the Java Virtual Machine, Berkeley, CA, Mc-Graw-Hill, 1999.