Java Implementation and Performance

Java Compiler Structure

Compile classes in .java files -- produce bytecode in .class files

Bytecode

A compiled class stored in a .class files or a .jar file
Represent a computation in a portable way -- as PDF is to an image

Java Virtual Machine

The Java Virtual Machine (JVM) is an abstract stack machine -- the bytecode is written to run on the JVM
The JVM is also the name of the (written in C) program that loads and runs the bytecode. The JVM interprets the bytecode to "run" the program.
The JVM runs the code with the various robustness/safety checks in place -- robustness vs. performance tradeoff.

Verifier

The JVM also has a "verifier" that checks that the bytecode is well-formed (e.g. an int is never directly used as a pointer, ...). This is a step in making Java virus proof.
A bad guy can try to feed a "bad" program to the JVM, but the verifier will catch in places where the program tries to get around the runtime type, array, etc. checks.
Usually, you don't see verifier errors, since the compiler will only generate "correct" bytecode, but the verifier is still needed, in case a bad guy hand crafts some bad bytecode.

ByteCode Example

You can use the "javap" command to print out the bytecode for a class. Normally it prints a summary. The -c switch causes it to print the actual bytecode.

Bytecode Strategy

The bytecode is just a description of the computation that does not have a particular limit on the number of registers.
HotSpot can translate the bytecode into real register machine code for the particular architecture.
The Bytecode is structured to be portable, and so the verifier can work.

Bytecode Primer

The byte code executes against a stack machine -- adding 1 + 2 like this

```
    iload 1;            // push a 1 onto the stack
    iload 2;            // push a 2 onto the stack
    add;                // add the two numbers on the stack
                // leaving the answer on the stack
```

"load" means push a value onto the stack
aload_0 = push address of slot 0 -- slot 0 is the "this" pointer
iload_1 = push an int from slot 1 (a parameter)
getfield -- using the pointer on the stack, load an ivar
putfield -- as above, but store to the ivar

Student Bytecode

```
nick% javap -c Student
Compiled from Student.java
public class Student extends java.lang.Object {
    protected int units;
    public static final int MAX_UNITS;
    public static final int DEFAULT_UNITS;
    public Student(int);
    public Student();
    public int getUnits();
    public void setUnits(int);
    public int getStress();
    public boolean dropClass(int);
    public static void main(java.lang.String[]);
}

<snip>

Method int getUnits()
   0 aload_0
   1 getfield #20 <Field int units>
   4 ireturn

Method void setUnits(int)
   0 iload_1
   1 iflt 10
   4 iload_1
   5 bipush 20
   7 if_icmple 11
  10 return
  11 aload_0
  12 iload_1
  13 putfield #20 <Field int units>
  16 return

Method int getStress()
   0 aload_0
   1 getfield #20 <Field int units>
   4 bipush 10
   6 imul
   7 ireturn
```

JITs and Hotspot

Just In Time compiler -- the JVM may compile the bytecode to native code at runtime (with the robustness checks still in). (This is one reason why java programs have slow startup times.)

The "hotspot" project tries to do a sophisticated job of which parts of the program to compile. In some cases, hotspot can do a better job of optimization than a C++ compiler, since hotpsot is playing with the code at runtime and so has more information.

Future

Maybe cache the compiled version, to speed class loading

Think of bytecode as a **distribution** format, while at runtime something more native is happening.

Optimization

Optimization Quotes

> Rules of Optimization:
> Rule 1: Don't do it.
> Rule 2 (for experts only): Don't do it yet.
> - M.A. Jackson

> "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity." - W.A. Wulf

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." - Donald Knuth

Optimization 101

Reality

- Hard to predict where the bottlenecks are
- It's not so hard to use tools to measure what the code is doing once it is written.
- Therefore, write the code you way you want to be **correct** and **finished** first, then worry about optimization.

"Premature Optimization" = evil

- Classic advice from Don Knuth
- Write the code to be straightforward and correct first
- Maybe it's fast enough already
- If not, measure to find the bottleneck
- Focus optimization there. Use CS161 type optimal algorithms + use language techniques as below

Data Structures

Your data structure will have a profound influence on performance.

This is one bit of "early" design where you might want to think about performance a little.

The choice of data structure (what you store, who has pointers to whom) can be very constraining on the possible algorithms later on.

Proportionality To Caller

Suppose we write a foo() utility in a way which is easy to code but naive -- it currently costs 1 millisecond, but could be sped up drastically. foo() is only called in one place by the bar() method. (If foo() is called multiple times, just add them all up to get the total foo() cost.)

How do you know if this matters?

The key question: **how costly is bar()**? If bar() takes 20 milliseconds, then foo() just doesn't matter. The smart strategy is to leave foo() in it's slow/naive/correct implementation -- find something else to fix.

If bar() takes 2 milliseconds, then foo() makes a huge difference and should be fixed.

1-1 User Event Rule

If something happens some fixed number of times like 1 or 3, for each single user event, such as a button push, then performance is not too important for that operation.

Watch for operations that happen 100's or thousands of times in relation to each user event.

User events happen very slowly from the computer's point of view.

e.g. use reflection to do a single method lookup when a button is pressed. Reflection is slow, but the true bottleneck will certainly be some other operation that happens many times per press.

Java Tips

Using the right data structure and algorithm is the most important. After that we have language feature rules...

1. 1-10-100 Rule

assignment (=) : 1 unit of time

method call : 10 units of time

similar overhead to C

new object or array : 100 units of time

In reality, it is very hard to say what the cost of "new" is -- it depends on how expensive the object ctor is and how long lived the object is and how much burden it adds to the GC system over the object's lifetime.

This has also been known as the 1-10-1000 rule, but modern GC has improved so much that 1-10-100 is probably closer. In any case, you should think of "new" as being rather expensive.

With modern GC systems, it's probably a bad idea to try to maintain your own large "free" list of objects for re-use. If the list is large, it interferes with GC doing its job. Techniques that were speedups with old GC systems are no longer speedups.

2. int getWidth() vs. Dimension getSize()

getSize() requires a heap allocated object

getWidth() and getHeight() may just be inlined to move the two ints right into the local vars of the caller code.

With HotSpot, short lived objects are a case where the new GC dos very well, so this is less of a concern than it once was.

3. Locals Faster Than iVars

Local (stack) variables are faster than member variables of any object (the receiver or some other object). Locals are also easier for the optimizer to work with for a variety of optimizations.

A .width variable in this object or in some other object pointer is slower than a local stack variable.

Inside loops, pull needed values into local variables (int i;).

Suppose we are in a for loop...

1. Slow -- message send
   ...i < piece.getWidth()
2. Medium -- instance variable -- with a good JIT, this case and (1) above are essentially the same.
   ...i < piece.width
   -or-
   ...i< width (suppose the code is executing against the receiver)
3. Fast -- pull the state into a local (stack) variable, and then use it. This makes it easier for the JIT to pull the value into a native register. If the value is in an ivar, the runtime may need to retrieve it from memory every time it is used. It's hard for the runtime to deduce that .width is not being changed, so it reloads it from memory. Whereas it's easy for it to deduce that localWidth is not being changed, so it can just put it in a register and use that value the whole time. (Note theme for the future: we're sensitive to generating memory traffic.)

```
int localWidth = piece.getWidth(); // or width if we are the receiver
... i<localWidth...

-or-

// make it even more clear for the JIT...
final int localWidth = piece.getWidth();
```

4. Avoid Synchronized (Vector)

Synchronized has a moderate runtime cost -- although this has been greatly reduced as of Java 1.3

Can have synch and unsynch versions of the same method, and switch between the two based on some other flag.

Use "immutable" (unchangeable) objects to finesse synchronization problems.

As usual, this matters if...

The routine is called many times
What the routine does is cheap, so the synch overhead is significant

The old Vector class is synchronized for everything, which is often a needless cost. Use the new Collections ArrayList instead.

If you can get away with a plain array, even better -- that's the fastest

5. StringBuffer

Use StringBuffer for multiple append operations -- change to String only once it's not going to change.

Automatic

This case the compiler optimizes for you -- appending together a bunch of strings at one moment into one immutable string.

```
String s = "a string" + foo.toString() + "some other string";
```

No

```
String record;                  // ivar

void transaction(String id) {
  record = record + " " + id;                   // NO, chews through memory
}
```

YES

```
StringBuffer record;
void transaction(String id) {
  record.append(" ");
  record.append(id); + id;
}
```

6 Don't Parse

Obvious but slow strategy: read in XML, ASCII, etc. -- build big data structure

Fast: read it into memory, but leave it as just chars. Do the search, etc. in the chars -- just parse/build the sub-part you need on the fly.

7. Avoid Weird Code

The whole suite of JVM optimizations added over time will be oriented towards common looking code -- write your code in the most obvious, common way, not some weird way. Ironically, weird code often gets written in the pursuit of optimization.

e.g. the write obvious form: for (int i = 0; i<bound; i++) {...}

Also, realize that obvious method implementations like getWidth() {return(width);} will certainly be targeted by HotSpot, so don't worry about the method overhead.

8. Threading / GUI Threading

Use separate threads so the GUI remains responsive. This "feels" fast. (snappy). This is always a good idea -- java makes it pretty easy to do right.

Divide the problem in to sub-problems that can be performed by multiple threads simultaneously -- a performance win on multiple processors.

Pro: in the future of parallel hardware, this may be the best way to get more performance.

Con: the software is harder to write, and the bugs can be subtle

Con: the locking/unlocking imposes a cost, so the parallel speedup needs to be great enough to cover that extra cost.

This may be the performance technique of the future -- explicit parallelism in the software

9. Inlining Methods/Classes

JVM optimizers, and hot spot in particular, make aggressive use of inlining -- pasting called code into the caller code.

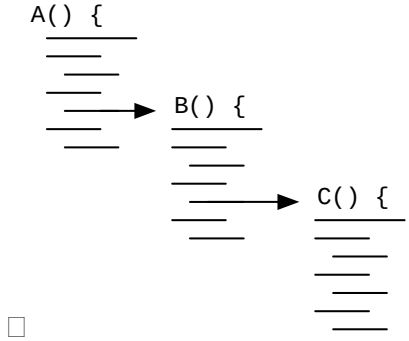
Inlining enables many other optimizations.

The "final" keyword for a class means it will not be subclassed, and a "final" method will not be overriden.

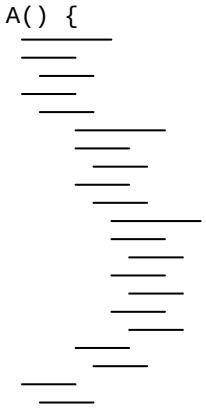
These assumptions can help HotSpot figure out message-method mappings before they happen.

As Hotspot gets smarter, it can figure things out even without "final".

Not Inlined



Inlined



Advantages

Data Flow
- Values in A() are passed to parameters in B(), passed to C(), where they are used.
- Now, the flow of that value through the whole A/B/C sequence can be analyzed -- the value can just live in one variable/register for the whole computation
- This saves on memory traffic, which is just what we need

Propagation of analysis
- Suppose A() is running a for(i=0; i<array.len; i++) loop, and calls B() and C().
- Down in the C() code, a statemene like array[i] would need to be checked for i<0 and i>=len normally.
- But now Hotspot can see the value of i from start to finish, show that it's always in range, and so remove the cost if the aray-bounds check.
- Similar optimizations work for example: checking if pointers are null, checking instanceof on a pointer.

10. Think About Memory Traffic

Old: CPU bound
- Think about how many operations you do.

New: Memory bound
- CPU operations are getting cheaper all the time as the CPU gets faster than the memory system.
- Think about how much "traffic" your algorithm must read and write. Once it's in the cache, it's cheap, so reading the same things multiple times is cheap. Reading consecutive addresses is also cheap.

Cache
- All access is through the cache hierarchy, so reading or writing the 4 bytes at at address x, actually loads 16 bytes or so around that address. This is why touching consecutive memory locations is cheap.

Linked List Example

What is the cost of iterating through all the elements of a linked list?
Need to load each linked list element: data + next field
Bad: the next field itself is just added overhead
Bad: the next linked list element will somewhere else in memory. Cache systems do best when you accessing contiguous stretches of memory.

LinkedList vs. ChunkList

Imagine a ChunkList implementation where each linked list element contains a small array of elements.
The ChunkList is faster, purely because it makes better use of cache lines.
The linked list probably gets just one client element per cache line loaded.
Because the ChunkList stores the client elements adjacent to each other, it is able to load several all in one cache line.
The point: think of an algorithm in terms of the memory touch pattern of an algorithm. The unit of cost is the number of cache lines pulled in from main memory.

Array of Objects

Suppose you have an array of objects, with ivars x, y, and z. in each object.
What is the cost of adding up all the y's -- think in terms of cache lines.
Suppose we re-organize the data into 3 "parallel" arrays -- one of all the x's, one of all the y's, and one of all the z's -- now what is the cost?
The parallel arrays design is not OOP, and it's just generally bad, but it could be a lot faster -- a technique to have in mind if you're truly desparate. NEVER do something like this for the original design -- fall back to it in desperation.