Re: Why is java considered a language for "web" or "internet" programming?

Source: http://coding.derkeiler.com/Archive/Java/comp.lang.java.help/2006−10/msg00381.html

- *From*: Mishagam <noemail@xxxxxxxxxxxx>
- *Date*: Sun, 22 Oct 2006 05:20:16 GMT

Tom Forsmo wrote:

> Mishagam wrote:
>
> > I just used my little bit C / Asm / Knuth education, and my very very limited experience writing interpreters / compilers. I don't remember reading exact things about Perl interpreter, for example. I assume that for operation i = i+2 on Java after JIT (or on C / C++ after compiler) you have do something like (asm instructions for native CPU, r − register):
> >
> > load r, i
> > add r, #2
> > store r, i
> >
> > For perl, assuming what should be done, and taking into account what you wrote, and using higher level operation here, you have to do:
> >
> > load and parse bytecodes.
> > load i ptr
> > load i type bits
> > check i type
> > go to correct procedure for integers
> > load i integer value
> > add 2
> > store i value
> > return to start of next bytecode.
>
> This is where you are mistaken, because many of the steps you mention here are automaticaly performed by the processor and some dont need to be any more elaborate than it is for c or java. The only thing that takes more times is performing the bit test. Loading pointer and type bits are predone, so is loading integer value along with i+2.

I don't understand what "predone" means here. I think, integer in Perl or Python or other scripting languages is some structure, with at the least field for type and value, and pointer to this structure you hold in 'i' variable. So you have to load i, and then load i type and value. (and also may be / probably reference counter and so on).

The actual

> decision on whether to use integer procedure or not is branch−predicted, so are the instructions for each branch. In addition to all this, much of the bytecode is hotspotted or cached so the conversion to binary does not need to happen again.

I again don't understand what "hotspotted" means here. I now think, that you can convert bytecode to sequence of branch instructions, through it appears that you have to have rather large branch table (to jump depending from i type) so you cannot put these tables inline for each command.
In my sequence I already assumed that conversion of '2' to binary and checking of '2' type is done in (Perl) compiler.
So I assume shortened sequence can be:

jump to summing procedure
load i ptr
load i type
switch to procedure/case for integer addition
load i value
add 2
store i value
break // switch
return to next command

Which looks only about 3 times longer than Java version, takes more space and more jumps.
As you can see below, this is definitely not how things are done in Python.

So the only real difference is the type

> checking of the bits which is done in a couple of cycles.
> So the end result is pretty much the same as teh above example, pluss a couple of cycles as penalty, which does not amount to much.

Finally I decided to run real benchmarks. I wrote very short, but not trivial integer calculations (so compiler will not optimize it away):
On VC++:

```
const int NRUNS = 1000;
const int NNUM = 1000000;
const int A = 23491;
const int B = 789175;
.....
clock_t t0 = clock();
for (int i=0; i<NRUNS; i++) {
int k = 5;

for (int j = 0; j<NNUM; j++) {
k = k*A + B;
kk = kk + k + j*j;
}
}
clock_t t1 = clock();
```

```
int rep = (NRUNS*NNUM);
double time = ((t1−t0)*1.0)/CLOCKS_PER_SEC;
printf("Speed test ended, kk = %d rep = %d, time = %9.3f\n", k, rep, time);
```

On Java:

```
final static int NRUNS = 1000;
final static int NNUM = 1000000;
final static int A = 23491;
final static int B = 789175;
.....
long t0 = System.currentTimeMillis();
for (int i=0; i<NRUNS; i++) {
int k = 5;

for (int j = 0; j<NNUM; j++) {
k = k*A + B;
kk = kk + k + j*j;
}
}
long t1 = System.currentTimeMillis();
int rep = (NRUNS*NNUM);
double time = (t1 − t0)/1000.0;
System.out.println("Speed test ended, kk = " + kk +
" rep = " + rep + ", time = " + time);
```

On Python:

```
NRUNS = 1000
NNUM = 1000
A = 23491
B = 789175
MAXI = (1 << 32)

import time
print "Speed Test Started"
t0 = time.time()
i = 0
kk = 0
while i < NRUNS:
k = 5
j = 0
while j < NNUM:
k = (k*A + B) % MAXI;
kk = (kk + k + j*j) % MAXI;
j=j+1
i=i+1

t1 = time.time()
rep = (NRUNS*NNUM)
tm = (t1 − t0)/1.0
print "Speed test ended, kk = " + str(kk) + " rep = " + str(rep) + ", time = " + str(tm)
```

Results :
VC++/Release − 2 sec
VC++/Debug − 4 sec
Java − 4 sec

Python − 2.5 sec − but you can see, that C / Java make 10 ^ 9 repetitions, and python makes 10^6 repetitions, so Python is 1000 times slower − just what I expected !!
I agree, that we compare different things here, first in Python I used while cycle, because 'for' has different semantics, (I don't think this is very important here) and second − I had to make % MAXI for Python, because C and Java ignored integer overflows, but Python tried to make exact computations using unlimited precision. Again, apparently Python doesn't make optimizations based on fact that k is integer − but I think this is design choice if you use typeless scripting language.

I didn't test Perl, because I hate Perl and it's syntax and I never seen uglier language, so I choose Python as modern scripting language that I like and assumed it is representative.
Do you really think that Perl would work substantially better in this case?

> Of course, such a micro operation, is going to take a little more time because the language is dynamically typed and interpreted, but that is the point of knuths assertion, dont: "dont pre−optimise".
>
> > I generally like you comments, but I usually really hate when something makes advices against micro−optimization,
>
> I like to be referred to as someone, not something, unless you think my brain is such a super computer :)

Please excuse me for my typing error. I am sorry.

.