Java HotSpot VM
« Subtil java »

« Ou comment je suis passé
de 12 à 2 jours de calcul... »

Pierre Fernique – Infusion – 18 oct 2013

Compil – recompil - decompil

- **JIT+OSR** = Compilation native (en plus du bytecode), en tâche de fond, pour certaines méthodes du code, avant exécution, ou pendant l'exécution, éventuellement plusieurs fois...
- Instrumentation du code interprété => **optimisation adaptative**
- Permet d'obtenir des gains de performances considérables (code natif 10 à 20x plus rapide que du code interprété, et éventuellement plus rapide que du code compilé traditionnellement car mieux optimisé)
- **-Xint** : force l'interprétation (juste pour comparer)
- **-server | -client** : change le comportement de la VM (server = optimisation maximale quitte à perdre du temps au début, et de la place = instrumentation)... *choisit tout seul en 1.7, tjrs -server sous Windows*
- **-XX:+PrintCompilation** : affichage en live de ce que fait la VM.

Test en live

```
static void testTrigo() throws Exception {

   double z=0;
   long t1,t2,t=System.nanoTime();
   long n=1000000;

   t=System.nanoTime();
   for( int j=0; j<n; j++ )
      for( double i=0; i<2*Math.PI; i+=Math.PI/180 ) z += Math.cos(i);
   t1 = System.nanoTime();

   z=0;
   for( int j=0; j<n; j++ )
      for( double i=0; i<2*Math.PI; i+=Math.PI/180 ) z += FastMath.cos(i);
   t2 = System.nanoTime();

   double d1 = (t1-t)/1000000.;
   double d2 = (t2-t1)/1000000.;
   System.out.println("n="+n+" Math: "
      +d1+" FastMath: "+d2+" => "+(d2/d1)+"% z="+z);
}
```

Déclenchements de la compilation

- 1 -30 itérations:

```
rien
```

- 30-35 itérations:

```
    70    1     healpix.newcore.FastMath::cos (93 bytes)
    70    2     healpix.newcore.FastMath::sincoshelper (98 bytes)
```

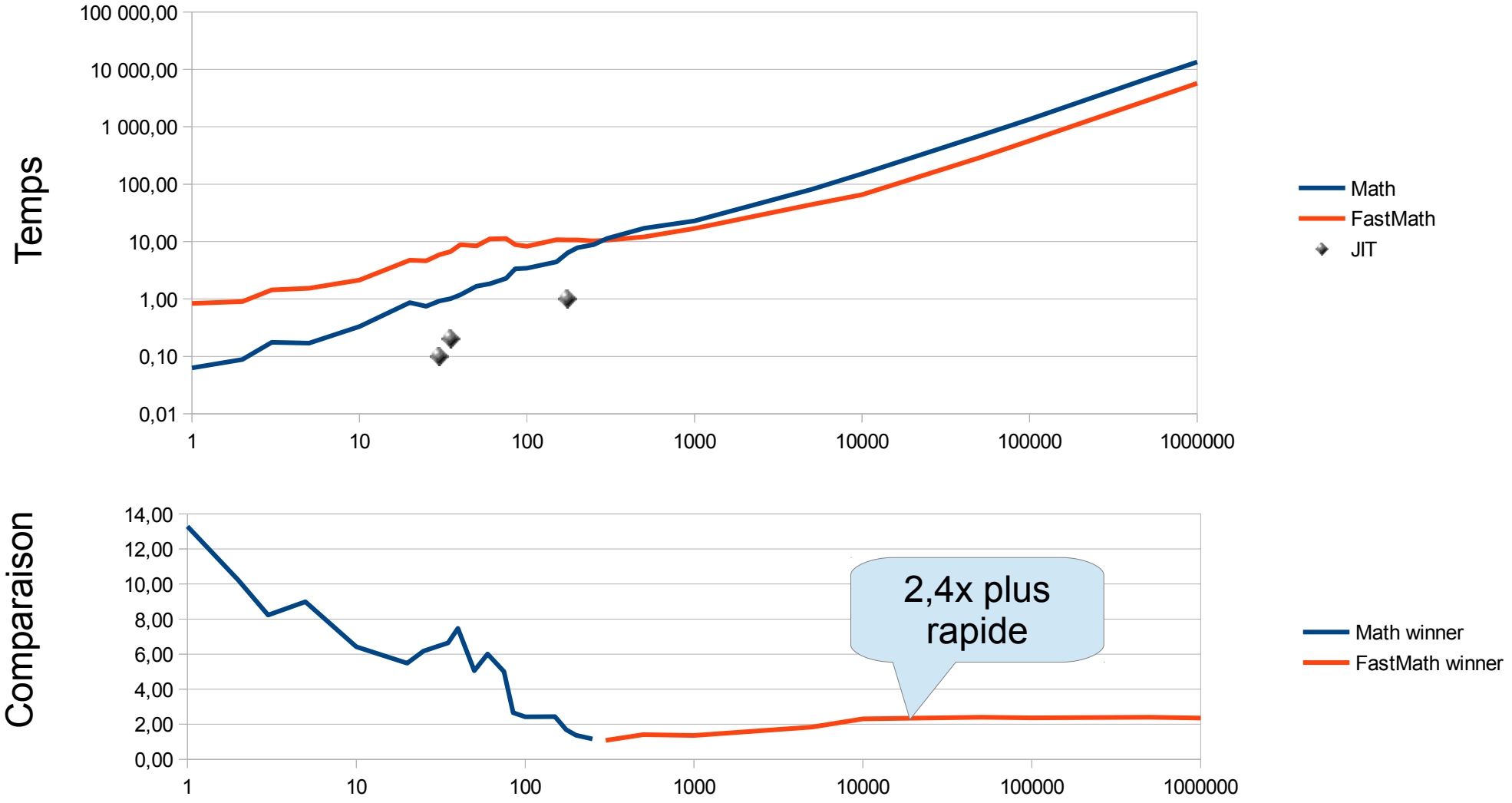
- 35-150 itérations:

```
    64    1%    cds.moc.examples.MocTest::testTrigo @ 29 (215 bytes)
    74    1     healpix.newcore.FastMath::cos (93 bytes)
    75    2     healpix.newcore.FastMath::sincoshelper (98 bytes)
    75    2%    cds.moc.examples.MocTest::testTrigo @ 82 (215 bytes)
```

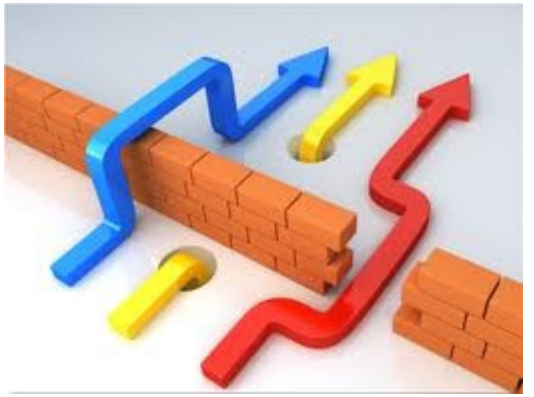
- >150 itérations:

```
    60    1%    cds.moc.examples.MocTest::testTrigo @ 29 (215 bytes)
   751    1%    cds...MocTest::testTrigo @ -2 (215 bytes) made not entrant
   756    1     healpix.newcore.FastMath::cos (93 bytes)
   757    2     healpix.newcore.FastMath::sincoshelper (98 bytes)
   757    2%    cds.moc.examples.MocTest::testTrigo @ 82 (215 bytes)
  1040    2%    cds...MocTest::testTrigo @ -2 (215 bytes) made not entrant
```

Comparaison des temps d'exécution



Qu'est-ce que la VM essaye de supprimer ?



- **Méthodes** inutiles (ex : getter/setter)
- **Synchronisations** inutiles (ex : un seul thread manipule)
- **Tests** inutiles dans les boucles plutôt qu'à l'extérieur (for {switch(bitpix)... }, tests des paramètres en entrée de méthode, elle-même appelée par une boucle...), tests indices de tableaux...
- **Polymorphismes** inutiles : Plus le code est abstrait (Object, List, ...), plus il sera lent

Optimisations les plus courantes

- **Inlining** : supprime l'appel de la méthode en copiant directement le contenu dans la méthode appelante
- **Unrolling** : supprime la boucle en répétant plusieurs fois le contenu
- **RangeCheck, nullCheck elimination** : supprime ces tests lorsqu'ils ne sont pas nécessaires.
- **Lock coarsening/eliding** : « factorise » ou supprime les synchronisations
- **Escape analyzing** : supprime les polymorphismes, voire les objets inutiles

Ce qui aide la VM (1)...

- Utiliser : Static, private, final, des constante
- Eviter les synchros si possible (StringBuilder, ArrayList...)
- Utiliser : Les types primitifs (un Vector de Integer est une bêtise)
- Appel de méthode dans une boucle : oui...
  mais de préférence si elle peut être inlinée.
- Boucle sur tableaux de 0 à la taille du tableau
- Sortir les cas rares de la boucle
- Eviter quand c'est possible les invocations virtuelles
  (monomorphisme yes, bimorphisme mouais, multimorphisme ga

Ce qui aide la VM (2)...

- Ne pas trop morceler son code (gros blocs plus faciles à optimiser => horizon de l'optimiseur)
- Eviter si possible la réflexion (CORBA, RMI,...) qui ne s'optimise pas du tout
- Eviter le chargement de code dynamique
- Eviter le « volatile » : force la synchronisation des cache CPU avec RAM => variable non optimisable
- Eviter les arraycopy overlappants (non optimisés)
- *Code java pure plutôt que « native » ? JNI overhead ? Pas d'optimisation possible ?*

Bref... si on doit aller vite... ne pas trop programmer « objet »

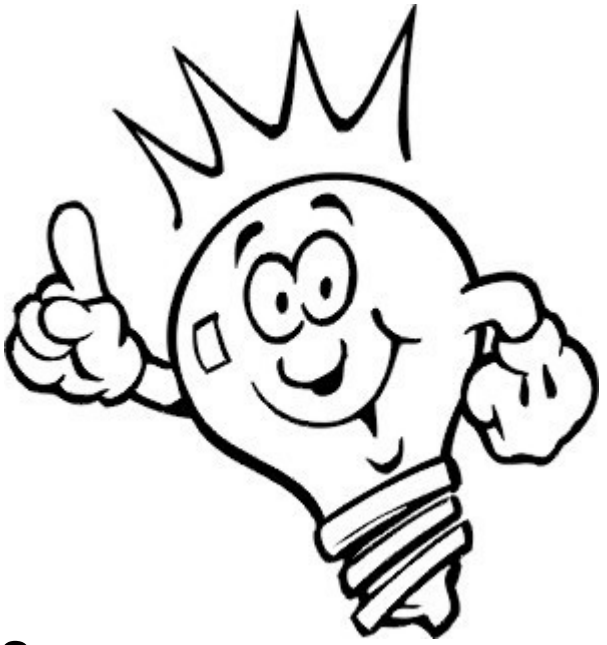
Dans le détail (VM 1.7)

```
  60     1%     cds.moc.examples.MocTest::testTrigo @ 29 (215 bytes)
 751     1%     cds...MocTest::testTrigo @ -2 (215 bytes) made not entrant
 756     1      healpix.newcore.FastMath::cos (93 bytes)
 757     2      healpix.newcore.FastMath::sincoshelper (98 bytes)
 757     2%     cds.moc.examples.MocTest::testTrigo @ 82 (215 bytes)
1040     2%     cds...MocTest::testTrigo @ -2 (215 bytes) made not entrant
```

Date IDDrapeaux Méthode Paramètres TailleOriginale

- « % » : OSR – On Stack Replacement (remplacement du code à la volée (sans attendre le prochain appel de la méthode)
- « ! » : Exception à gérer
- « s » : Synchronisation à gérer
- « n » : appel à du code natif (librairie pré-compilé)
- @29 : OSR_BCI => identificateur du code compilé
- Made not entrant : compilation agressive (=> le code ne peut être ré-appelé)
- Made zombie : code invalidé (supposition fausse)

Mais avant tout cela...



- Eviter d'allouer inutilement
  idée : passer les références de vos objets
- Eviter de déplacer ou recopier inutilement
  idée : passer les tableaux dans sa globalité, en indiquant l'offset et la taille de la section à travailler
- Eviter de faire plusieurs fois la même chose
  idée : cache 1 case, très facile à implanter

Pourquoi me fatiguer,
et écrire comme un geek-C,
puisque la VM va optimiser
mon code ?

Comparaison (10 millions)

```
int [] decompte(StringBuffer a) {
   int [] lettre = new int[256];
   for( int j=0; j<a.length(); j++ ) {
      char ch = a.charAt(j);
      int i = (int)ch;
      lettre[i]++;
   }
   return lettre;
}
```

```
int [] decompte(int [] lettre, String a) {
     for(int i=0; i<lettre.length; i++ )
        lettre[i]=0;
     char [] b = a.toCharArray();
     for( int j=0; j<b.length; j++ ) {
         lettre[ (int)b[j] ]++;
     }
     return lettre;
 }
```

```
  59    1  s          java.lang.StringBuffer::length (5 bytes)
  60    2             cds.moc.examples.MocTest::decompte (41 bytes)
  65    3  s          java.lang.StringBuffer::charAt (28 bytes)
  83    4             java.lang.String::length (6 bytes)
  95    5             java.lang.Object::<init> (1 bytes)
  96    6             java.lang.AbstractStringBuilder::ensureCapacityInternal (16 bytes)
  97    7     n       java.lang.System::arraycopy (0 bytes)   (static)
  99    8             java.lang.String::getChars (62 bytes)
  99    9             java.lang.AbstractStringBuilder::append (48 bytes)
 100   10             java.lang.AbstractStringBuilder::<init> (12 bytes)
 101   11  s          java.lang.StringBuffer::append (8 bytes)
 101   12             java.lang.StringBuffer::<init> (18 bytes)
 112    1 %           cds.moc.examples.MocTest::testBis @ 16 (84 bytes)
6572    1 %           cds.moc.examples.MocTest::testBis @ -2 (84 bytes)   made not entrant
```

C'est termine en 6510ms

```
  67    1             cds.moc.examples.MocTest::decompte1 (48 bytes)
  79    1 %           cds.moc.examples.MocTest::testBis1 @ 20 (82 bytes)
 891    1 %           cds.moc.examples.MocTest::testBis1 @ -2 (82 bytes)   made not entrant
```

C'est termine en 830ms

De la lecture...

- Optimisation Java :
  http://www.oracle.com/technetwork/java/6-performance-137236.html
  http://www.oracle.com/technetwork/java/whitepaper-135217.html
- Documents 2007 !
  => Oracle n'y met plus l'énergie.
  => OpenJDK ?

- *À suivre .... « gestion du GC »...*