Java performance

Reducing time and space consumption

Peter Sestoft (sestoft@dina.kvl.dk)

Royal Veterinary and Agricultural University, Copenhagen, Denmark
and
IT University of Copenhagen, Denmark

Version 2 of 2005-04-13

Abstract: We give some advice on improving the execution of Java programs by reducing their time and space consumption. There are no magic tricks, just advice on common problems to avoid.

1 Reducing time consumption

1.1 Standard code optimizations

Do not expect the Java compiler (such as `javac` or `jikes`) to perform many clever optimizations. Due to Java's rather strict sequencing and thread semantics there is little the compiler can safely do to improve a Java program, in contrast to compilers for less strictly defined languages such as C or Fortran. But you can improve your Java source code yourself.

- Move loop-invariant computations out of loops. For example, avoid repeatedly computing the loop bound in a `for`-loop, like this:

  ```
  for (int i=0; i<size()*2; i++) { ... }
  ```

  Instead, compute the loop bound only once and bind it to a local variable, like this:

  ```
  for (int i=0, stop=size()*2; i<stop; i++) { ... }
  ```

- Do not compute the same subexpression twice:

  ```
  if (birds.elementAt(i).isGrower()) ...
  if (birds.elementAt(i).isPullet()) ...
  ```

  Instead, compute the subexpression once, bind the result to a variable, and reuse it:

  ```
  Bird bird = birds.elementAt(i);
  if (bird.isGrower()) ...
  if (bird.isPullet()) ...
  ```

- Every array access requires an index check, so it is worth-while to reduce the number of array accesses. Moreover, usually the Java compiler cannot automatically optimize indexing into multidimensional arrays. For instance, every iteration of the inner (`j`) loop below recomputes the indexing `rowsum[i]` as well as the indexing `arr[i]` into the first dimension of `arr`:

```
double[] rowsum = new double[n];
for (int i=0; i<n; i++)
  for (int j=0; j<m; j++)
    rowsum[i] += arr[i][j];
```

  Instead, compute these indexings only once for each iteration of the outer loop:

```
double[] rowsum = new double[n];
for (int i=0; i<n; i++) {
  double[] arri = arr[i];
  double sum = 0.0;
  for (int j=0; j<m; j++)
    sum += arri[j];
  rowsum[i] = sum;
}
```

  Note that the initialization `arri = arr[i]` does not copy row `i` of the array; it simply assigns an array reference (four bytes) to `arri`.

- Declare constant fields as `final static` so that the compiler can inline them and precompute constant expressions.

- Declare constant variables as `final` so that the compiler can inline them and precompute constant expressions.

- Replace a long `if-else-if` chain by a `switch` if possible; this is much faster.

- If a long `if-else-if` chain cannot be replaced by a `switch` (because it tests a `String`, for instance), and if it is executed many times, it is often worthwhile to replace it by a `final static HashMap` or similar.

- Nothing (except obscurity) is achieved by using 'clever' C idioms such as performing the entire computation of a while-loop in the loop condition:

```
int year = 0;
double sum = 200.0;
double[] balance = new double[100];
while ((balance[year++] = sum *= 1.05) < 1000.0);
```

1.2 Fields and variables

- Access to local variables and parameters in a method is much faster than access to static or instance fields. For a field accessed in a loop, it may be worthwhile to copy the field's value to a local variable before the loop, and refer only to the local variable inside the loop.

- There is no runtime overhead for declaring variables inside nested blocks or loops in a method. It usually improves clarity to declare variables as locally as possible (with as small a scope as possible), and this may even help the compiler improve your program.

1.3 String manipulation

- Do not build strings by repeated string concatenation. The loop below takes time quadratic in the number of iterations and most likely causes heap fragmentation as well (see Section 2):

```
String s = "";
for (int i=0; i<n; i++) {
   s += "#" + i;
}
```

  Instead, use a `StringBuilder` object and its `append` method. This takes time linear in the number of iterations, and may be several orders of magnitude faster:

```
StringBuilder sbuf = new StringBuilder();
for (int i=0; i<n; i++) {
  sbuf.append("#").append(i);
}
String s = sbuf.toString();
```

- On the other hand, an expression containing a sequence of string concatenations automatically gets compiled to use `StringBuilder.append(...)`, so this is OK:

```
String s = "(" + x + ", " + y + ")";
```

- Do not process strings by repeatedly searching or modifying a `String` or `StringBuilder`. Repeated use of methods `substring` and `index` from `String` may be legitimate but should be looked upon with suspicion.

1.4 Storing tables of constants in arrays

- Declaring an initialized array variable inside a method causes a new array to be allocated at every execution of the method:

```
public static int monthdays(int y, int m) {
  int[] monthlengths =
     { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
  return m == 2 && leapyear(y) ? 29 : monthlengths[m-1];
}
```

  Instead, an initialized array variable or similar table should be declared and allocated once and for all as a `final static` field in the enclosing class:

```
private final static int[] monthlengths =
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

public static int monthdays(int y, int m) {
  return m == 2 && leapyear(y) ? 29 : monthlengths[m-1];
}
```

- More complicated initializations can use a static initializer block `static { ... }` to precompute the contents of an array like this:

```
private final static double[] logFac = new double[100];
static {
  double logRes = 0.0;
  for (int i=1, stop=logFac.length; i<stop; i++)
    logFac[i] = logRes += Math.log(i);
}

public static double logBinom(int n, int k) {
  return logFac[n] - logFac[n-k] - logFac[k];
}
```

The static initializer is executed when the enclosing class is loaded. In this example it precomputes a table `logFac` of logarithms of the factorial function $n! = 1 \cdot 2 \cdots (n-1) \cdot n$, so that method `logBinom(n,k)` can efficiently compute the logarithm of a binomial coefficient. For instance, the number of ways to choose 7 cards out of 52 is `Math.exp(logBinom(52, 7))` which equals 133 784 560.

1.5 Methods

- Declaring a method as `private`, `final`, or `static` makes calls to it faster. Of course, you should only do this when it makes sense in the application.
- For instance, often an accessor method such as `getSize` can reasonably be made `final` in a class, when there would be no point in overriding it in a subclass:

```
class Foo {
  private int size;
  ...
  public final int getSize() {
    return size;
  }
}
```

  This can make a call `o.getSize()` just as fast as a direct access to a `public` field `o.size`. There need not be any performance penalty for proper encapsulation (making fields `private`).
- Virtual method calls (to instance methods) are fast and should be used instead of `instanceof` tests and casts.
- In modern Java Virtual Machine implementations, such as Sun's HotSpot JVM and IBM's JVM, interface method calls are just as fast as virtual method calls to instance methods. Hence there is no performance penalty for maintenance-friendly programming, using interfaces instead of their implementing classes for method parameters and so on.

1.6 Sorting and searching

- Never use selection sort, bubblesort or insertion sort, except on very short arrays or lists. Use heapsort (for arrays) or mergesort (for doubly linked lists) or quicksort (for arrays; but you must make a good choice of pivot element).
- Even better, use the built-in sorting routines, which are guaranteed to be fast: $O(n \log(n))$ time for n elements, and sometimes faster if the data are nearly sorted already:

For arrays, use `java.util.Arrays.sort`, which is an improved quicksort; it uses no additional memory, but is not *stable* (does not preserve the order of equal elements). There are overloaded versions for all primitive types and for objects.

For `ArrayList<T>` and `LinkedList<T>`, implementing interface `java.util.List<T>`, use `java.util.Collections.sort`, which is *stable* (preserves the order of equal elements) and *smooth* (near-linear time for nearly sorted lists) but uses additional memory.

- Avoid linear search in arrays and lists, except when you know that they are very short. If your program needs to look up something frequently, use one of these approaches:

  – *Binary search on sorted data*:

  For arrays, use `java.util.Arrays.binarySearch`. The array must be sorted, as if by `java.util.Arrays.sort`. There are overloaded versions for all primitive types and for objects.

  For `ArrayList<T>`, use `java.util.Collections.binarySearch`. The array list must be sorted, as if by `java.util.Collections.sort`.

  If you need also to insert or remove elements from the set or map, use one of the approaches below instead.

  – *Hashing*: Use `HashSet<T>` or `HashMap<K,V>` from package `java.util` if your key objects have a good hash function `hashCode`. This is the case for `String` and the wrapper classes `Integer`, `Double`, ..., for the primitive types.

  – *Binary search trees*: Use `TreeSet<T>` or `TreeMap<K,V>` from package `java.util` if your key objects have a good comparison function `compareTo`. This is the case for `String` and the wrapper classes `Integer`, `Double`, ..., for the primitive types.

1.7 Exceptions

- The creation `new Exception(...)` of an exception object builds a stack trace, which is costly in time and space, and especially so in deeply recursive method calls. The creation of an object of class `Exception` or a subclass of `Exception` may be between 30 and 100 times slower than creation of an ordinary object. On the other hand, using a `try-catch` block or throwing an exception is fast.

- You can prevent the generation of this stack trace by overriding method `fillInStackTrace` in subclasses of Exception, as shown below. This makes creation exception instances roughly 10 times faster.

```
class MyException extends Exception {
  public Throwable fillInStackTrace() {
    return this;
  }
}
```

- Thus you should create an exception object only if you actually intend to throw it. Also, do not use exceptions to implement control flow (end of data, termination of loops); use exceptions only to signal errors and exceptional circumstances (file not found, illegal input format, and so on). If your program *does* need to throw exceptions very frequently, reuse a single pre-created exception object.

1.8 Collection classes

Java's collection classes in package `java.util.*` are well-designed and well-implemented. Using these classes can improve the speed of your program considerably, but you must beware of a few pitfalls.

- If you use `HashSet<T>` or `HashMap<K,V>`, make sure that your key objects have a good (uniform) and fast `hashCode` method, and that it agrees with the objects' `equals` method.
- If you use `TreeSet<T>` or `TreeMap<K,V>`, make sure that your key objects have a good and fast `compareTo` method; or provide a `Comparator<T>` resp. `Comparator<K>` object explicitly when creating the `TreeSet<T>` or `TreeMap<K,V>`.
- Beware that indexing into a `LinkedList<T>` is not a constant-time operation. Hence the loop below takes time quadratic in the size of the list `lst` if `lst` is a `LinkedList<T>`, and should not be used:

  ```
  int size = lst.size();
  for (int i=0; i<size; i++)
    System.out.println(lst.get(i));
  ```

  Instead, use the enhanced `for` statement to iterate over the elements. It implicitly uses the collection's iterator, so the traversal takes linear time:

  ```
  for (T x : lst)
    System.out.println(x);
  ```

- Repeated calls to `remove(Object o)` on `LinkedList<T>` or `ArrayList<T>` should be avoided; it performs a linear search.
- Repeated calls to `add(int i, T x)` or `remove(int i)` on `LinkedList<T>` should be avoided, except when `i` is at the *end or beginning* of the linked list; both perform a linear traversal to get to the `i`'th element.
- Repeated calls to `add(int i, T x)` or `remove(int i)` on `ArrayList<T>` should be avoided, except when `i` is at the *end* of the `ArrayList<T>`; it needs to move all elements after `i`.
- Preferably avoid the legacy collection classes `Vector`, `Hashtable` and `Stack` in which all methods are `synchronized`, and every method call has a runtime overhead for obtaining a lock on the collection. If you do need a synchronized collection, use `synchronizedCollection` and similar methods from class `java.util.Collection` to create it.
- The collection classes can store only reference type data, so a value of primitive type such as `int`, `double`, ...must be wrapped as an `Integer`, `Double`, ... object before it can be stored or used as a key in a collection. This takes time and space and may be unacceptable in memory-constrained embedded applications. Note that strings and arrays are reference type data and need not be wrapped.

  If you need to use collections that have primitive type elements or keys, consider using the Trove library, which provides special-case collections such as hash set of `int` and so on. As a result it is faster and uses less memory than the general Java collection classes. Trove can be found at <http://trove4j.sourceforge.net/>.

1.9 Input and output

- Using buffered input and output (`BufferedReader`, `BufferedWriter`, `BufferedInputStream`, `BufferedOutputStream` from package `java.io`) can speed up input/output by a factor of 20.
- Using the compressed streams `ZipInputStream` and `ZipOutputStream` or `GZIPInputStream` and `GZIPOutputStream` from package `java.util.zip` may speed up the input and output of verbose data formats such as XML. Compression and decompression takes CPU time, but the compressed data may be so much smaller than the uncompressed data that it saves time anyway, because less data must be read from disk or network. Also, it saves space on disk.

1.10 Space and object creation

- If your program uses too much space (memory), it will also use too much time: Object allocation and garbage collection take time, and using too much memory leads to poor cache utilization and possibly even the need to use virtual memory (disk instead of RAM). Moreover, depending on the JVM's garbage collector, using much memory may lead to long collection pauses, which can be irritating in interactive systems and catastrophic in real-time applications.
- Object creation takes time (allocation, initialization, garbage collection), so do not unnecessarily create objects. However, do not introduce object pools (in factory classes) unless absolutely necessary. Most likely, you will just add code and maintenance problems, and your object pool may introduce subtle errors by recycling an object in the pool although it is still being referred to and modified from other parts of the program.
- Be careful that you do not create objects that are never used. For instance, it is a common mistake to build an error message string that is never actually used, because the exception in which the message is embedded gets caught by a `try-catch` that ignores the message.
- GUI components (created by AWT or Swing) may claim much space and may not be deallocated aggressively enough. Do not create GUI components that you do not necessarily need.

1.11 Bulk array operations

There are special methods for performing bulk operations on arrays. They are usually much faster than equivalent `for` loops, in part because they need to perform only a single bounds check.

- `static void java.lang.System.arrayCopy(src, si, dst, di, n)` copies elements from array segment `src[si..si+n-1]` to array segment `dst[di..di+n-1]`.
- `static bool java.util.Arrays.equals(arr1, arr2)` returns true if the arrays `arr1` and `arr2` have the same length and their elements are pairwise equal. There are overloads of this method for arguments of type `boolean[]`, `byte[]`, `char[]`, `double[]`, `float[]`, `int[]`, `long[]`, `Object[]` and `short[]`.
- `static void java.util.Arrays.fill(arr, x)` sets all elements of array `arr` to `x`. This method has the same overloads as `Arrays.equals`.
- `static void java.util.Arrays.fill(arr, i, j, x)` sets elements `arr[i..j-1]` to `x`. This method has the same overloads as `Arrays.equals`.
- `static int java.util.Arrays.hashcode(arr)` returns a hashcode for the array computed from the hashcodes of its elements. This method has the same overloads as `Arrays.equals`.

1.12 Scientific computing

If you are doing scientific computing in Java, the Colt open source library provides many high performance and high quality routines for linear algebra, sparse and dense matrices, statistical tools for data analysis, random number generators, array algorithms, mathematical functions and complex numbers. Don't write a new inefficient and imprecise numerical routine if the one you need is here already. Colt can be found at <http://hoschek.home.cern.ch/hoschek/colt/>

1.13 Reflection

- A reflective method call, reflective field access, and reflective object creation (using package `java.lang.reflect`) are far slower than ordinary method call, field access, and object creation.
- Access checks may further slow down such reflective calls; some of this cost may be avoided by declaring the class of the called method to be `public`. This has been seen to speed up reflective calls by a factor of 8.

1.14 Compiler and execution platform

- As mentioned above, a Java compiler cannot perform many of the optimizations that a C or Fortran compiler can. On the other hand, a just-in-time (JIT) compiler in the Java Virtual Machine (JVM) that executes the bytecode can perform many optimizations that a traditional compiler cannot perform.
- For example, a test `(x instanceof C)` conditionally followed by a cast `(C)x` may be optimized by a JVM so that at most one test is performed. Hence it is not worth the trouble to rewrite your program to avoid either the `instanceof` test or the cast.
- There are many different Java Virtual Machines (JVMs) with very different characteristics:
  - Sun's HotSpot Client JVM performs some optimizations, but generally prioritizes fast startup over aggressive optimizations.
  - Sun's HotSpot Server JVM (option `-server`, not available for Microsoft Windows) performs very aggressive optimizations at the expense of a longer startup delay.
  - IBM's JVM performs very aggressive optimizations, comparable to Sun's HotSpot Server JVM.
  - The JVMs in implementations of J2ME (mobile phones) and PersonalJava (some PDAs) do not include JIT compilation and probably perform no optimizations at all. Hence in this case it is even more important that you do as many optimizations as possible in the Java code yourself.
  - I do not know the optimization characteristics of Oracle's JVM, the Kaffe JVM, Intel's Open Runtime Platform, IBM's Jikes RVM, ...

  You can see what JVM you are using by typing `java -version` at a command-line prompt.

1.15 Profiling

If a Java program appears to be too slow, try to *profile* some runs of the program. Assume that the example that performs repeated string concatenation in Section 1.3 is in file `MyExample.java`. Then one can compile and profile it using Sun's HotSpot JVM as follows:

```
javac -g MyExample.java
java -Xprof MyExample 10000
```

The result of the profiling is shown on standard output (the console):

```
Flat profile of 19.00 secs (223 total ticks): main

  Interpreted + native   Method
  1.3%     1  +     0    java.lang.AbstractStringBuilder.append
  1.3%     1  +     0    java.lang.String.<init>
  2.6%     2  +     0    Total interpreted

     Compiled + native   Method
 51.3%     0  +    40    java.lang.AbstractStringBuilder.expandCapacity
 29.5%    23  +     0    java.lang.AbstractStringBuilder.append
 10.3%     8  +     0    java.lang.StringBuilder.toString
  6.4%     0  +     5    java.lang.String.<init>
 97.4%    31  +    45    Total compiled

  Thread-local ticks:
 65.0%   145             Blocked (of total)


Flat profile of 0.01 secs (1 total ticks): DestroyJavaVM

  Thread-local ticks:
100.0%     1             Blocked (of total)


Global summary of 19.01 seconds:
100.0%   929             Received ticks
 74.6%   693             Received GC ticks
  0.8%     7             Other VM operations
```

It says that 51.3% per cent of the *computation time* was spent in native method `expandCapacity` and a further 29.5% was spent in method `append`, both from class `AbstractStringBuilder`. This makes it plausible that the culprits are `+` and `+=` on `String`, which are compiled into `append` calls.

But what is even more significant is the bottom section, which says that 74.6% of the total time was spent in garbage collection, and hence less than 25% was spent in actual computation. This indicates a serious problem with allocation of too much data that almost immediately becomes garbage.

2 Reducing space consumption

- In a JVM, data are allocated on a call stack (for method parameters and local variables) and on a heap (for objects, including strings and arrays). There is a separate stack for every thread of execution, and a joint heap for all the threads. The stack of a thread grows and shrinks with the depth of method calls. Object, strings and arrays are allocated in the heap by the executing threads; they are deallocated (garbage-collected) by an autonomous garbage collector.

- Three important aspects of space usage are allocation rate, retention and fragmentation:

  – *Allocation rate* is the rate at which your program creates new objects, strings, and arrays. A high allocation rate costs time (for allocation, object initialization, and deallocation) and space (because the garbage collector may set aside more memory for efficiency reasons) even when the allocated data has a very short lifetime.

  – *Retention* is the amount of live heap data, that is, the heap data transitively reachable from the call stacks at any point in time. A high retention costs space (obviously) and time (the garbage collector must perform more administrative work both for allocation and deallocation).

  – *Fragmentation* is the creation of fragments: small unusable chunks of memory. Allocation of increasingly larger objects, such as increasingly longer strings or arrays, may cause memory fragmentation, leaving many small memory fragments that cannot be used. Such fragmentation costs time (to search for a sufficiently large hole at allocation) and space (because the fragments go unused). Most garbage collectors take care to avoid fragmentation, but that itself may cost time and space, and may not be done in embedded JVM implementations.

- A *space leak* is unwanted or unexpected retention, which usually causes memory consumption to grow linearly with execution time. A space leak is caused by objects, strings or arrays being reachable from live variables although those objects will actually never be used again. For instance, this may happen if you cache computation results in a `HashMap`: the results remain reachable from the `HashMap` even if you will never need them again. This can be avoided by using a `WeakHashMap` instead.

- A space leak may be caused by a deeply tail-recursive method that should have been written as a loop. A Java compiler does not automatically optimize a tail-recursive method to a loop, so all data reachable from the execution stack will be retained until the method returns.

- The kind of garbage collector (generational, mark-sweep, reference counting, two-space, incremental, compacting, ...) strongly influences the time and space effects of allocation rate, retention, and fragmentation. However, a functioning garbage collector will never in itself cause a space leak. Space leaks are caused by mistakes in your program.

- Make sure that constant fields shared among all objects of a class are `static`, so that only one field is ever created. When all `Car` objects have the same icon, do not do this:

```
public class Car {
  ImageIcon symbol = new ImageIcon("porsche.gif");
  ...
}
```

  Instead, do this:

```
public class Car {
  final static ImageIcon symbol = new ImageIcon("porsche.gif");
  ...
}
```

- When you are not sure that an object will actually be needed, then allocate it lazily: postpone its allocation until needed, but allocate it only once. This will unconditionally create a `Button` for every `Car` object, although the `Button` may never be requested by a call to the `getButton` method:

  ```
  public class Car {
    private Button button = new JButton();

    public Car() {
      ... initialize button ...
    }

    public final JButton getButton() {
      return button;
    }
  }
  ```

  Instead, you can allocate the `Button` lazily in `getButton`:

  ```
  public class Car {
    private Button button = null;

    public Car() { ... }

    public final JButton getButton() {
      if (button == null) { // button not yet created, so create it
        button = new JButton();
        ... initialize button ...
      }
      return button;
    }
  }
  ```

  This saves space (for the `Button` object) as well as time (for allocating and initializing it). On the other hand, if the button is known to be needed, it is more efficient to allocate and initialize it early and avoid the test in `getButton`.

3 Other resources

The book J. Noble and C. Weir: *Small Memory Software*, Addison-Wesley 2001, presents a number of design patterns for systems with limited memory. Not all of the advice is applicable to Java (for instance, because it requires pointer arithmetics), but most of it is useful albeit somewhat marred by pattern-speak.

4 Acknowledgements

Thanks to Morten Larsen, Jyrki Katajainen and Eirik Maus for useful suggestions.