Increasing Script Performance, Part III

It is quite interesting to compare the speed of MetaCard with HyperCard on the one hand and lower level languages like Pascal or C on the other hand. Consider a simple statement like (assuming that x is appropriately initialized):

```
put x + 1 into x
```

How do the different languages compare? Because you cannot use MetaBench for this comparison, here are the (rough) results.

Pascal: x := x+1 --> 1
C++: x = x+1 --> 1
MetaCard: add 1 to x --> 80
MetaCard: put x + 1 into x --> 130
HyperCard: add 1 to x --> 2800
HyperCard: put x + 1 into x --> 3600
Empty extension: --> 3500

The relative speeds might be different for different computers or operating systems of course. Nevertheless, it can easily be seen that HyperCard is much slower than MetaCard. MetaCard however is much slower than Pascal or C. This difference may suggest that writing a simple extension that adds up numbers might considerably enhance the speed of calculations. Unfortunately this is not true. The overhead of calling an extension and sending the result back to MC outweighs the gain. In the table above you can see that calling an "empty" extension (an extension that just does nothing) would take 3500 time units. Extensions can enhance the speed, but only if the total amount of calculations performed by the extension outweighs calling the extension and sending the result back. It is easy to see that if your extension performs an adding operation 100 times, you have a significant speed gain (about 3600 versus 8000 units). More about extensions in a forthcoming part. Hence we need different approaches to enhance the speed of calculations. Don't expect to learn how to add up numbers faster. But you can expect how to prevent that scripts manipulating numbers may run 10 or even 50 times (!) or so slower than necessary. First some fairly simple things to get the most out of calculations.

You most likely know that

```
add 1 to x
```

executes faster then:

```
put x + 1 into x
```

The reason is that in 'add 1 to x', the variable x is accessed only once, whereas in 'put x + 1 into x' it is accessed two times. The computer doesn't know that the result of 'x + 1' should be put in the same place x and has to figure out twice where in memory x resides; execution of the statement is similar to the execution of 'put x + 1 into y'.

Probably you already preferred to write 'add 1 to x' instead of 'put x + 1 into x': you have less characters to type. Be honest, do you also always write 'subtract 1 from x' instead of 'put x - 1 into x'? The speed gain is the same, for the very same reason. And this also holds of course for the multiply and divide operators; or for x^2 versus x*x.

Accessing variables as least as possible is a general principle to enhance the speed. Suppose you have a complex calculation:

script A:

```
put (Ncases*covXY-sumX*sumY)/ sqrt((Ncases*sum2X-(sumX)^2)*(Ncases*sum2Y-(sumY)^2)) into r
```

The statement (the formula for the Pearson Correlation, if you really want to know) is hardly readable and probably you run into trouble in putting the parentheses at the right places. So the next script looks much better:

script A:

```
put Ncases * covXY - sumX * sumY into c
put Ncases * sum2X - (sumX)^2 into r1
put Ncases * sum2Y - (sumY)^2 into r2
put c / sqrt(r1 * r2) into r
```

Much nicer, but the script is 30% slower because of the overhead of putting and getting the variables c, r1 and r2. The general rule is: minimize accessing variables, either to get them, or to put something in them.

The div and mod operators are often used for particular transformations of numbers. For example, to transform number 0 to 7 into 7, 8 to 15 into 15, 16 to 23 into 23, etc. you may use:

```
get (i div x + 1) * x - 1
```

(i is the number to transform, x signifies the range, 8 in this case) To transform 0 to 7 into 0 to 7, 8 to 15 into 0 to 7, 16 to 23 into 0 to 7, etc. you may use:

```
(i mod x + 1) - 1
```

For particular numbers (i.e. 2, 4, 8, 16, etc.) it is faster to use the bitwise operators:

```
i bitor (x-1)
```

respectively

```
i bitand (x-1)
```

to obtain the same result, but more then 50 % faster.

To determine whether a number is odd or even, you can use:

```
(x bitand 2) = 0
```

(true if even, false if odd)

If you wonder what bitwise operators are (the documentation in MC is minimal), see the text at the end of this contribution.

Now for something completely different. Suppose you have read a real number x, e.g. "0.345678" from a field. Compare these two scripts:

script C:

```
get x/(1-x)
```

script D:

```
add 0 to x
get x/(1-x)
```

Believe it or not, but script D will run considerably faster. If the content of a field (or a file) is read, MC hasn't the faintest idea that it's meant to be a number, and stores it as a string of characters. Thus it has to convert the string into a number, including a check whether or not x is a number before it actually starts calculating. In executing script C, this conversion takes place two times. In script D however, the conversion takes place only once. If 'add 0 to x' is executed, the string x is converted to a number, and the number is stored into x: now x no longer contains the string "0.345678" but (a binary representation of) the number 0.345678. Thus it saves a lot of time in performing calculations on it. The more x's in the equation, e.g. x*(1+x)/(1-x), the more you gain. It's a bit tricky to perform a speed test on script D, to compare it with C. Consider script E: **script E:**

```
put fld "inputFld" into x
repeat 100000
  add 0 to x
  get x/(1-x)
end repeat
```

After the first loop x is converted to a number. Hence the conversion takes place only once, instead of 100000 times. You should be very well aware of such peculiarities in performing speed tests. But you can safely try:

script F:

```
put fld "inputFld" into x
add 0 to x -- convert it to a real number
repeat 100000
  get x/(1-x)
end repeat
```

It will run nearly 10 times as fast compared to a script without 'add 0 to x', excluding the overhead of the repeat loop.

Generally, if you store a variable, and MC has become aware that this variable is a number, it will be stored as a binary number: no conversion is needed if it is used for further calculations. You may note that if your are doing speed tests on numbers, and test numbers are read from a field, results will be completely unreliable, if you assume that the numbers read, are binary numbers.

If a variable is stored as a binary number, and you use statements like 'get length (x)' or 'get char 3 of x', it has to be converted to a string, before the statement can be executed. For some reason, converting a number to a string is much slower then converting a string to a number. Converting an integer to a string takes about 3 times as much as the other way around, whereas converting a real number is even slower, because of the more complex formatting (by the way, MC does not distinguish internally between integers and real numbers; all numbers are stored as reals). Suppose variable x is read from a field, containing "0.345678". Compare:

script G:

```
add 0 to x -- convert it to a real number
repeat 100000
  get last char of x
end repeat
```

script H:

```
repeat 100000
  get last char of x
end repeat
```

Script G may take about 50 times as long as script H...

If variable x is binary stored, and you are going to perform a number of string operations on it (like if char 1 of x = "2"; get offSet (".", x); put length (x) into s), you better copy it into another variable and convert the copy to a string first, for example:

```
put x into s
put empty after s
```

You should leave the original variable x untouched. Compare script I and J:

script I:

```
put 1/3 into x
put last char of x into last char of x
put 3*x
```

script J:

```
put 1/3 into x
put 3*x
```

Script I yields 0.999999 (if the local numberFormat has the default value), script J yields 1. Because you converted the number to a string, you lost precision. To illustrate things, here is a nice example from Xavier, in a reaction to Part 1 of How to speed up MC. He wrote:

```
> Another 2 things to consider in your loops:
> - set cursor to busy
>   will slow down scripts considerably.
>
> - If you have a "progress" bar, dont change it at every loop,
>    do so for each percent change or at every 100th...
>    this will also save major time...
>
> for example
>   repeat with x = 1 to 100000
>     if char -3 of x is not oldx then
>       set cursor to busy
>       showstatus
>      put char -3 of x into oldx
>   end if
```

Although his observations are correct, his solution is not (sorry Xavier, but I love your example because it illustrates that we really can gain a lot). The variable x is a binary one, not a string one (it's produced by MC itself). So each time 'if char -3 of x is not oldx then' is executed, x is converted to a string, to be able to get character -3: it takes an awful lot of time. To be precise, x itself remains intact, the converted copy is put in some internal temporary variable. The script takes 370 ticks on my computer, neglecting 'showStatus' of course. Alternatively, take a look at script J:

```
put the ticks into myTestTime
repeat with x = 1 to 1000000
  if the ticks - myTestTime > 5 then
    put the ticks into myTestTime
    set cursor to busy
  end if
  -- do something
end repeat
```

It takes 160 ticks. Not that much difference? Please note the difference in the number of repeats: hundred and thousand versus one million! This script has some of other advantages too:

- The rotation of the busy cursor is independent of the time 'do something' takes (unless it takes more then 5 ticks, in which case that the script would take a day or more).
- You can set a flag, becoming true the first time the ticks - myTestTime exceeds 5 ticks. Based on the value of x at that moment and the total number of repeats, you can decide whether or not a progress bar is shown (assuming that the 'do something's' during the first 5 ticks are representative for the remaining ones). For example, if the total time is expected to exceed 60 ticks, show a progress bar, else stick to the busy ball.

For your convenience, here is a more complete scheme.

```
put 60 into myDelayPref
-- but better let user decide on whether progress bar will be
-- shown if analysis is expected to take more then 1 second,
-- 2 seconds, 5 seconds or whatever
put 0 into myProgresStatus
put the number of lines of myData into nLines
put the ticks into myTestTime
repeat with x = 1 to nLines
  if the ticks - myTestTime > 5 then
    switch myProgresStatus
    case 2 -- analysis takes more then one second
      myAdjustProgressBar -- adjusts the progress bar
      break
    case 1 -- analysis will take less then 1 second
      set cursor to busy
      break
    case 0 -- not checked yet
      if (5*nLines)/x > myDelayPref then -- take 5 ticks as criterion for updating
        myShowProgressBar -- show a progress bar
        put 2 into myProgresStatus
      else
        put 1 into myProgresStatus
      end if
      break
    end switch
    put the ticks into myTestTime
  end if
  -- do something
end repeat
```

If analyses take more then 10 ticks, case 0 will occur the least, hence this case is the last one; if the analysis takes less then 10 ticks -- well, why bother? The five ticks delay before a progress bar appears will not be observed by the user. Take the switch part (not the ticks check!) out of the script to create a function handler, returning myProgresStatus; this will hardly add time, because it is only called each five ticks.

Happy programming,

Wil Dijkstra

Posted on 4/14/03 by Will Dijkstra to the MetaCard list (See the complete post/thread)

Print this tip

News and Rumors Products Services Developer Resources Contact STS About STS