Clément Béra ~ Smalltalk, Tips 'n Tricks

Massive String Comparison Performance Boost Up-coming in the Cog VM

17 - *Saturday* - *Feb 2018*

Posted by Clement Bera in Cog

≈ 2 Comments

Hi all,

These days my student Sophie Kaleba is working on the optimizations of String operations in the Cog VM and I'm spending a few hours here and there helping her. The main idea is to add a numbered primitive for String comparison that would lead, I believed based on my experience with Sista, to up to 10x performance boost in String comparison. Some applications, like parsers which use extensively String>>#= to test tokens, will benefit greatly from this optimization.

Primitive specification

The existing Misc plugin primitive was specified as follow:

```
ByteString class >> compare: string1 with: string2 collated: order
"Return 1, 2 or 3, if string1 is string2, with the collating order of
characters given by the order array."
```

There were multiple things we wanted to change:

- Move away from the class-side primitive non-sense (Due to SmartSyntax thingy)
- Answer -1, 0 ,1 instead or 1, 2 or 3
- Make the last parameter optional, since this primitive is mainly used for String = which use ASCIIOrder, not making any difference in String comparison

The final primitive we implemented is specified as follow (two versions since last parameter is optional):

```
ByteString >> compareWith: anotherString
ByteString >> compareWith: anotherString collated: order
"Return a negative Smi, 0 or a positive Smi if self is anotherString, with the
collating order of characters given optionnally by the order array."
```

We've moved away from class-side to instance-side. Based on our Java friends advises we decided not to answer -1, 0 or 1 but a negative Smi, 0 or positive Smi since it improves a lot performance (we'll talk about that in next section). The last parameter is optional.

Primitive implementation

Basically we wrote all cases in pure Slang using direct SpurObjectMemory APIs since it's a numbered primitive, then we re-wrote the common case (no order) in the JIT's RTL opcodes.

I'm not going to show all the Slang code here since it's horrible to read without syntax coloring, but we've written it in quite a fancy way. Here are the 2 calls (depending if the optional order parameter is present) the the main comparison loop and the loop itself :

```
order
  ifNil:
    [self
      rawCompare: string1
      length: strLength1
      with: string2
      length: strLength2
      accessBlock: [:str :index | objectMemory fetchByte: index ofObject: str
]]
   ifNotNil:
    [self
      rawCompare: string1
      length: strLength1
      with: string2
      length: strLength2
      accessBlock: [:str :index | objectMemory fetchByte: (objectMemory
fetchByte: index ofObject: str) +1 ofObject: order ]]

rawCompare: string1 length: strLength1 with: string2 length: strLength2
accessBlock: accessBlock
  | c1 c2 min |
  "needs to be forced else slang does not inline it by default"
  min := strLength1 min: strLength2.
  0 to: min-1 do:
    [:i | c1 := accessBlock value: string1 value: i.
    c2 := accessBlock value: string2 value: i.
    c1 = c2 ifFalse: [^c1 - c2]].
  ^strLength1 - strLength2
```

We note two interesting things:

- We were able to extract the comparison loop in an external function. We had to manually force inlining to make it work, but it allows to write once the loop and generate in C two loops, an

optimized loop for the common case (no order) and the normal one without too much code duplication.
- Remember when we said answering a negative or positive value instead or -1 or 1 was valuable ? In the loop function, one can see that we return "strLength1 – strLength2" and "c1 – c2" instead of many branches, leading to better performance (and in the jitted version, much simpler code).

In Cog's JIT, we rewrote only the version without the order parameter (Still trying mainly to optimize String>>=). We wrote multiple versions to evaluate performance. We've not trying pointer-size vectorization yet, but so far one of the simplest version seems to be the fastest, here is the 7-instructions comparison loop:

```
instr := cogit MoveXbr: TempReg R: string1Reg R: string1CharOrByteSizeReg.
cogit MoveXbr: TempReg R: string2Reg R: string2CharOrByteSizeReg.
cogit CmpR: string1CharOrByteSizeReg R: string2CharOrByteSizeReg.
jumpMidFailure := cogit JumpNonZero: 0.
cogit AddCq: 1 R: TempReg.
cogit CmpR: TempReg R: minSizeReg.
cogit JumpAboveOrEqual: instr.
```

Preliminary performance evaluation & C compiler non-sense

Once everything was ready, we evaluated the performance of the original Misc primitive, the new primitive with Slang version only and the new primitive with Slang & JIT version. We made the evaluation in the case where there is no order since we're focusing on String>#=, so it's a bit unfair for the old Misc primitive which has the indirection, but that's the code that's going to be run.

The micro-bench I am going to show is the following (it's not that well written, hence the "preliminary" result):

```
{[ByteString compare: var with: var collated: asciiOrder] benchFor: 5 second.
[ var compareSlangPrim: var] benchFor: 5 second.
[ var compareSlangAndJITPrim: var] benchFor: 5 second}
```

We ran it twice, once with a byteString of size 3 and once with a byteString of size 1000.

For the byteString of size 3, we got something like that:

```
#('15,700,000 per second. 63.8 nanoseconds per run.'
'41,600,000 per second. 24 nanoseconds per run.'
'78,600,000 per second. 12.7 nanoseconds per run.')
```

which leads to the Slang prim being 2.6x faster than the Misc primitive, and the jitted code 5x faster than the Misc primitive. Small string evaluation is about testing the performance of all the non-loop comparison code: the subtraction at the end in our case instead of the multiple branches makes a difference here, quicker checks at the beginning of the primitive too. The removal of the order indirection makes a minor difference since the loop is used only 3 times. The jitted version is way faster than the Slang version since there's no jit-to-C stack switch dance.

For the byteString of size 1000, interestingly, I got:

```
#('841,000 per second. 1.19 microseconds per run.'
'1,500,000 per second. 666 nanoseconds per run.'
'2,190,000 per second. 457 nanoseconds per run.')
```

while Sophie got:

```
#('761,000 per second. 1.31 microseconds per run.'
'1,950,000 per second. 513 nanoseconds per run.'
'1,970,000 per second. 508 nanoseconds per run.')
```

See the difference ? In my case, the jitted code is faster than Slang code on large String, while for Sophie it isn't. Obviously we discovered that at 1 am yesterday and none of us could consider going to sleep before figuring out this non-sense. Fortunately, our first educated guess was the correct one, it's a C compiler difference. On Mac OS X I have a LLVM compiled VM while on Linux Sophie has a GCC-compiled VM. We disassembled all the code to make sure our guess was correct.

Here are the three different versions of the tight comparison loop (no order):

LLVM

```
0001f644 movl -0x14(%ebp), %esi
0001f647 movzbl 0x8(%edx,%esi), %esi
0001f64c movl -0x18(%ebp), %edi
0001f64f movzbl 0x8(%edx,%edi), %edi
0001f654 subl %edi, %esi
0001f656 jne 0x1f660
0001f658 incl %edx
0001f659 cmpl %ebx, %edx
0001f65b jl 0x1f644
```

GCC

```
0x080623b0 : movzbl 0x8(%eax,%edx,1),%ebp
0x080623b5 : movzbl 0x8(%eax,%ebx,1),%ecx
0x080623ba : cmp %ecx,%ebp
0x080623bc : jne 0x80624b0
0x080623c2 : add $0x1,%eax
0x080623c5 : cmp %eax,%edi
0x080623c7 : jne 0x80623b0
```

Cog's JIT

```
000018b2: movzbl %ds:(%edx,%eax,1), %edi
000018b6: movzbl %ds:(%esi,%eax,1), %ecx
000018ba: cmpl %edi, %ecx
000018bc: jnz .+0x0000003d
```

```
000018be: addl $0x00000001, %eax
000018c1: cmpl %eax, %ebx
000018c3: jnb .+0xfffffed
```

In the case of Sophie, the performance is the same because GCC and Cog's JIT generate almost exactly the same loop of 7 instructions. The main difference is that we use "jnb" instead of "jne", it's a micro-optimisation that we could integrate in the JIT in a further version (but that's I believe the kind of <1% performance optimisation). There's also this difference that GCC uses %ebp as a general purpose register (I checked GCC pushes %ebp on stack and pops it before returning), which is something we could consider doing too since we're lacking 1 register to optimize more this primitive (but we've tried multiple things and that version seems to be the best).

In my case, the LLVM code is simply horrible. LLVM does not succeed in keeping everything in registers leading to 2 spills that need to be reload at each iteration of the loop. The spills lead to two L1 memory reads and are the reason why the Slang primitive is slower than the jitted code on Mac. However, LLVM does this smart thing of using (subl, jne) while the other versions use (cmp, jne) and jump to the sub instruction if the branch is taken. Again we could integrate that in the JIT in a further version to earn <1% performance. LLVM also use the incl instruction instead of add $0x1, which is kind of fun but it does not make sense to do that in the JIT (we limit the number of instructions the RTL support to limit back-end complexity, incl is not part of the RTL instructions, and given that gcc does not use it, it most likely does not bring significant performance).

Retrospective & Considerations

Looking back at the byteString of size 3, I noticed that my results were also a little bit different that Sophie's for pure Slang primitive. I did not notice it at first but it's there and it's significant. Anyway there's an implementation in the JIT now which solves that problem.

Looking ahead at the loop version using order, we can see that the GCC version is also far better:

GCC

```
0x08062450 : movzbl 0x8(%ecx,%edx,1),%edi
0x08062455 : movzbl 0x1(%eax,%edi,1),%ebp
0x0806245a : movzbl 0x8(%ecx,%ebx,1),%edi
0x0806245f : movzbl 0x1(%eax,%edi,1),%edi
0x08062464 : cmp %edi,%ebp
0x08062466 : jne 0x80624a0
0x08062468 : add $0x1,%ecx
0x0806246b : cmp %ecx,%esi
0x0806246d : jne 0x8062450
```

LLVM

```
0001f5d4 movl %ebx, %edx
0001f5d6 movl -0x14(%ebp), %esi
0001f5d9 movzbl 0x8(%edi,%esi), %esi
0001f5de movl -0x10(%ebp), %eax
0001f5e1 movzbl 0x9(%esi,%eax), %esi
```

```
0001f5e6 movl -0x18(%ebp), %ebx
0001f5e9 movzbl 0x8(%edi,%ebx), %ebx
0001f5ee movzbl 0x9(%ebx,%eax), %ebx
0001f5f3 subl %ebx, %esi
0001f5f5 jne 0x1f665
0001f5f7 incl %edi
0001f5f8 movl %edx, %ebx
0001f5fa cmpl %ebx, %edi
0001f5fc jl 0x1f5d4
```

which makes me wonder… Should we write a jitted version for this case to avoid the LLVM slow-down on Mac? Should we report this somehow to the LLVM people so they may be able to fix it? The problem if we write this case in the JIT is that normally we don't do this hack that GCC does of using %ebp as a general purpose register, so we will lack a register on x86 to make it efficient. I mean we could do it, but it has to be done in some kind of platform-independent way which may not be obvious… Maybe we just have 1 read instead of the LLVM 3 reads on x86, and for ARMv5 and x64 we use one of the extra general purpose register? Or maybe we just ignore this case since it's not that common and pray the LLVM god to make this better?

Conclusion

We've just demonstrated through preliminary benchmarks that we've sped-up the String comparison operation by a factor ranging from 2.5x to 5x.

In the case of parsers, most string comparisons happen on small strings to test tokens, so parsers will benefit more from a 5x speed-boost than a 2.5x on String equality. I've got a JSON parser benchmark which is written in the dumbest way possible (it mainly uses strings over streams). In the Sista VM where string equality is rewritten at runtime as raw pointer-size vectorized byte comparison, the JSON parser bench is 10x faster (mostly because of the String equality optimization), I can't wait to see what are the results with this primitive. Although most likely you'll have to wait for the IWST paper to know about that ;-).

thoughts on "Massive String Comparison Performance Boost Up-coming in the Cog VM"

1. ***said:*****Stéphane Ducasse**

   February 17, 2018 at 10:58 am

   Great post! I like the smi trick. It is so simple :).
   I imagine that this is were nativeBoost approach was better because you could control the assembly generated but at the expense of not being platform independent. May be resurrecting the virtualCPU project would make sense.

REPLY

- ***said:*** **Clement Bera**

  February 17, 2018 at 11:36 am

  I am a bit confused, we can control the assembly code generated by Cogit, but we cannot really control the code generated by the C compiler. The main advantage of NativeBoost was to implement the generation of machine code in Smalltalk and not in Slang. There is nothing you can do in NativeBoost that you cannot do in Cogit and the other way around. To remove the Slang/C primitive we would need to remove entirely the interpreter and rewrite it on top of Cogit's back-end.

  I think reviving VirtualCPU to be able to write assembly code in Pharo would make sense. However I would revive it using the existing JIT as a back-end (through extended bytecode set instructions) instead of NativeBoost (so we have one back-end to maintain and not 2, VCPU code can be executed in the machine code zone and managed by Cogit infrastructure, etc.).

  REPLY

2