Gmail **Sophie Kaleba <sophie.kaleba@gmail.com>**

ICOOOLPS18 notification for paper 4

1 message

ICOOOLPS18 <icooolps18@easychair.org> 12 juin 2018 à 17:12
À : Sophie Kaleba <sophie.kaleba@gmail.com>

Dear Sophie,

I am pleased to inform you that your ICOOOLPS 2018 submission has been accepted.

The reviews are below, and will also be available shortly at the submission web site.

Please consider the reviews carefully in preparing the final version of the paper. If you have questions about the reviews, please send them to me (email below) and I will involve the programme committee as appropriate. Our common goal from this point is to make your paper and presentation as good as it can be.

The deadline for the final version is Sunday July 1, 2018.

The paper will be posted to the ICOOOLPS web site by the date of the workshop and will appear in the ACM DL afterwards. If you don't want you paper to appear in the ACM DL, please let me know.

Details about the workshop agenda, duration of your presentation, etc., will be forthcoming.

Thank you again for your submission, and congratulations.

Tim Felgentreff
Program Co-Chair
tim.felgentreff@oracle.com

---------- REVIEW 1 ----------
PAPER: 4
TITLE: Assessing primitives performance on multi-stage execution
AUTHORS: Sophie Kaleba, Clement Bera and Stéphane Ducasse

Overall evaluation: 2 (accept)

---------- Overall evaluation ----------
This is a nice and easy to read paper.
The paper compares the performance and implementation effort of primitives in the Smalltalk Cog VM. Specifically, the paper evaluates the string comparison primitive, when implemented in pure Smalltalk, as a VM Plugin, in the VM itself as Slang code and finally in the JIT backend.

An interesting and very relevant discussion which doesn't seem mentioned here is profiling. This can be hard for a primitive written in C in the VM, but is usually already available for client code. As such, client code might perform better than a VM primitive, due to the extra knowledge from the profiling information (branches, types, values, inline caches for callbacks, etc).

I would have liked to see a more general discussion about primitives in other systems. The paper discusses RPython to some extent, but doesn't generalize.
For instance, I think Truffle languages don't have the issue to have to implement a primitive multiple times: either it's in client code, or it's in Java.
In both cases, the implementation can be optimized by the Graal JIT.
I think RPython is in a similar situation where primitives written in RPython or client code can be used by both the interpreter and JITed code efficiently.
So it seems in all generality that if the JIT can take as input (or via automated conversion) some code

implementing the primitive that can also be used in the interpreter (e.g. If the Cog JIT could take Slang as input), there is no need to write multiple implementations of a primitive. Efficiency is still a question though, especially if the JIT is producing rather inefficient code.
The paper also notes that these approaches affect startup/warmup and that is still relevant of course.

Another benefit to write primitives in client code is they can be shared between multiple VMs (as mentioned by "Reuse" in [FPW+15]).
This is quite valuable to avoid bugs which seems likely if code is complex and duplicated multiple times like primitives.

In footnote 3, `(length1 min: length2)` is anyway executed only once as it is an argument. I don't understand the footnote in this context.

In 4.3, the paper mentions "due to the old GNU C extensions we use in the interpreter code present in the same file to force variables into specific registers,".
Is it really in the same file, and are those annotations used or not for the code of the primitive?
It seems quite surprising a compiler would optimize less well a function because another function in the same file uses weird annotations.

In 5.1, I think "interpreter performance" would better describe rather than "start-up performance".

I noticed a few typos:
* Introduction: "As VMs *have*", "overall runtime performance"
* Introduction->Measurements: "We implemented a first version of the optional prim-
itive in *client code (Smalltalk)*." (hard to follow and why is Pharo relevant here?)
* 2.1 For example, accesses to object headers are dependent on the memory representation
* 4: I would expect some mention of the 1st "Baseline" configuration here.
* 4.3: "both the compilers": what does "both" refer to? GCC & Clang?
* 4.3: bolded in figure *4*
* 5.1: the need of primitive*s*
* 5.3: "the interpreter to have *the* same register convention *as* the"
* Conclusion: "the string primitives" => "the string comparison primitive"

---------- REVIEW 2 ----------
PAPER: 4
TITLE: Assessing primitives performance on multi-stage execution
AUTHORS: Sophie Kaleba, Clement Bera and Stéphane Ducasse

Overall evaluation: 0 (borderline paper)

---------- Overall evaluation ----------
The paper presents an evaluation of implementing certain operations at different levels in the ecosystem of a virtual machine for dynamic object-oriented languages using a string-compare as showcase. The different chosen levels exhibit different performance characteristics and possibly indicate different maintainability overhead.

Novelty (of the contribution)

(+) The paper gives insight into the performance gains possible for primitive implementations with an approach like RTL.

(-) Although the paper explicitly mentions the Algorithmic-Primitives paper, it is not made explicit _what more_ to expect here.

* The related work spells out that the AP paper does not deal with warm up and latencies but rather peak performance, which is correct. However, this paper does itself not deal with "non-peak" performance beyond indicating that it is "no problem" as second invocation of a method will already be optimized. This is not sufficient to create increased insight here.
* Moreover, this paper only distinguishes between "essential" and "optional" primitives, with an interpretation that matches Smalltalk's. It is however unclear why this is a good grouping in relation to this paper. The AP-paper presents a different grouping, including the "algorithmic primitives" group, where the

presented operations surely would fall into. Note that it is not necessarily the case here that the AP-paper grouping would be better, but rather that the grouping should be clearly motivated and fitting. It is hence not clear whether multiplication is an essential primitive (it is for the VM here, but by contract, not by this paper's definition). It can surely be written as looping additions, and even addition can be mapped to function call, if necessary. It is clear that performance would be abysmal here, but the primitive for that would be de facto dispensable.

Interest (to the community)

(+) Possible different coarse-granular levels of optimization are separated and clearly discerned. These might map to similar levels in other runtimes and the results could be used as starting points to identify similar optimization potentials.

(+) The paper shows that an approach that avoids stack-switching with a technique like RTL can actually improve performance at apparently reasonable maintenance costs.

(-) The paper fails short in generalizing the requirements, approach, and results beyond Smalltalk/Squeak /Pharo/Newspeak and their peculiar (although versatile) interpretation of primitives (fallback code, plugins, optionality). It its in _this form_ hardly imaginable to project the findings on, eg, JavaScript VMs or Ruby VMs.

Evidence (in support of the claims)

(+) Performance measurements and implementation descriptions are given and show differences in all approaches.

(-) The manageability argument that shines through certain statements (p10, "Conculsion": "the quickest to execute the primitive is, the harder it is to maintain") is only weakly supported by the numbers given and only briefly discussed.

(-) The main message here seems to be "the closer to the platform, the faster we get". This is not at all a surprising, new, or otherwise noteworthy insight. (Note: this does not mean that this is uninteresting per se, but rather that this is probably not the claim you want to make?)

(-) There are two motivating claims made in the introduction that lack support, eg proof, argument, or citation of other evidence:

  * p2, "High performance relies on a JIT with speculative optimizations, which is hard to implement, maintain and evolve."
  * p2, "There is a big performance gap between the baseline performance and the peak performance."

(-) The paper claims much performance is lost due to "stack-swiching" (p2, "World border cost"), wich is evident with the chosen VM but not shown universally. If this is an universal problem of VMs for dynamic languages, this must be shown more thoroughly (which, in turn would make the papers claim much stronger!).

Clarity (of presentation)

(+) The paper is written in an easy to follow style and good to read.

(+) The organization is good to follow.

(-) The title mentions "multi-stage execution", however this idea is never mentioned anywhere in the paper. It is also not self-evident what is meant by stages or "multi-stage"; the title should probably be adjusted.

(-) Figure 2 names the JIT's stack the "Client stack", which might be slightly ambiguous; Smalltalk as the actual client has a notion of its stack as a linked list of contexts. Arguably, _this_ it the "client stack".

Maybe just name it "JIT stack"?

(-) The graphs of Figure 3 are hard to compare due to the changing y-axis and missing intermediate marks. Also the error bars are hard to make out.

(-) Table 2 is never references in the paper.

(-) Figure 4 seems superfluous, as the simple explanation of the different generated code in the text ("cannot keep live", "additional memory loads") is completely sufficient and convincing.

(-) Since the "Slang" and even "Slang-RTL" implementations are not too long, they could have been included for comparison. Also, the paper says "We did not use high-level construct and chose to implement the Smalltalk version the most optimized way Smalltalk allows". It would have been _very_ interesting to also include the most idiomatic version into the comparison.

(-) It is not made clear, why the requirements of the string comparison are laid-out as is. If it would just follow C's `strcmp` requirements, the Smalltalk code could short-circuit even more:

```
baselineCompareWith: aString
| c1 c2 length1 length2 |
((length1 := self size) = (length2 := aString size))
    ifTrue: [1 to: (length1 min: length2) do:
            [ :i |(c1 := self basicAt: i) = (c2 := aString basicAt: i) ifFalse: [ ^c1 − c2 ] ].
^ length1 − length2
```

However, it seems that lexicographic ordering is a requirement here, so that should maybe made explicit?

Correctness (of the approach/method/argumentation)

(+) For the performance, warump and JIT timings have been taken into account.

(+) The setup is explained in detail, including versions for compilers.

(-) The setup is missing versions for the uses operating systems and maybe informations on the CPUs performance governors. Also, under "Processor", it claims using a 32bit VM. This is a bit confusing as both processors listed are Intel Core i5, which are surely 64bit x86_64 processors. Maybe rename the paragraph?

(-) 3.2 Comparison states that lines-of-code "arguably can be used to evaluate the complexity level of each implementation." This should be based on evidence.

(-) The performance results bear error measurements (bars in Figure 3, standard error in Table 2), but present "relative speed-up". It is not explained how this speed-up is calculated, whether by using an estimator (eg, Kalibera/Jones "Rigorous Benchmarking in Reasonable Time") or dividing the execution times by the baseline's results. In the latter case, it must be specified how the error was determined, as simply dividing leads to incorrect results and a bootstrapping of the error is required.

(-) It is said that ByteStrings are encoded in "the extended ASCII standard". This is incorrect, as there is no such thing as "the extended ASCII standard". ByteStrings by definition encode the first 255 code points of Unicode, which happens to match the Latin-1 (ISO 5589-1) encoding. Also, later uses of the word ASCII are not correct in this regards (even though the VM and image actually use the term ASCII in such places, but this is due to historic reasons).

Further questions

* To my knowledge, the RTL has not been described before. Putting this in the center of this paper might give a point-of-view with a more defendable claim.

Further comments

* Self has a special way of avoiding such things as the stack overhead: their JIT has knowledge of how the primitives work and can potentially inline primitives during JIT compilation. This should be regarded,

even though Cog does not seem to inline during JIT compilation.

* The template and reference style uses are not the one prescribed by the workshop and should be updated.