3 Measuring the Performance of the Linklet System On Pycket

Measure the performance of your changes to the Pycket JIT on the existing benchmarks. For at least 3 performance differences, propose a hypothesis that explains the difference.

In this section, we start by demonstrating and discussing the differences between the old and the new Pycket utilizing the linklets, and move on to measuring the performance of the new Pycket on the existing benchmarks, and finally we talk about the behaviors we observe in the experiment results by explaining some of the performance issues that are caused by the new linklet implementation on Pycket.

3.1 A Step Back : Old Pycket vs New Pycket

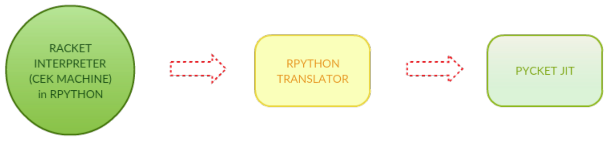

Fig. 18. Pycket

Pycket is a tracing JIT compiler that is generated by the RPython meta-tracing framework (originally designed for PyPy) that automatically generates tracing JIT compilers from interpreters written in RPython. [2][3] Before moving on to the new Pycket version that utilizes the linklets, let's recall how the Pycket works in its original design. As shown in the Figure 19, given a Racket program source, Pycket runs the Racket binary to expand the given program into Racket core forms (see fully expanded programs), and then uses its own expander (*expand.py*) to turn fully-expanded Racket code into a Pycket AST and evaluates it. [1]

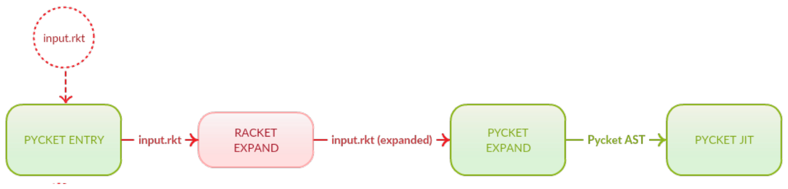

Fig. 19. Old Pycket

As we discussed a bit in the Section 1.1, a couple of fundamental changes have happened in the Racket's core that made the Racket developers to consider moving onto a new runtime, namely the Chez Scheme. To make this transition easier (and correspondingly to make Racket more portable) Racket implements a layer called the *"rumble"*, which gives to any runtime that implements it the capability of hosting other layers that are written in Racket. [2] The way that it works is, that Racket exports four layers in the form of serialized independent linklets that are used to bootstrap Racket using the *rumble* implementation of the runtime system. These *bootstrapping* linklets implement the *expander*, *thread*, *io*, and *regexp* layers. Therefore, any VM that implements *rumble* can import these linklets and get the Racket's module system, macro system and many more for free.

Having most of the *rumble* layer already implemented within, implementing the linklets enabled the Pycket to exploit this change in the Racket and use the exported bootstrapping linklets. Currently, the new version of Pycket loads the *"expander"* linklet into its runtime at boot, thereby gets (among others) the entire module system, the functions *read*, *expand* and *eval* for free. It uses Racket's own module system to *load*, *expand* and *eval* Racket modules (e.g. *racket/base*), consequently running entirely independent from the Racket binary. With this setup, Pycket even has a jitted Racket REPL, as shown in Figure 20 below.

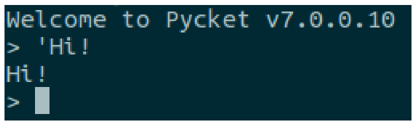
```
Welcome to Pycket v7.0.0.10
> 'Hi!
Hi!
> 
```

Fig. 20. A jitted Racket REPL on Pycket

3.2 Performance

The new implementation with the linklets fundamentally changed how the evaluation works in Pycket. In the old version, Pycket used to handle resolving, expanding and evaluating the modules by itself. The new Pycket, on the other hand, uses Racket's module system through the *expander* linklet. Therefore on the high level, the entire execution is in the continuation of the evaluation (instantiation) of the expander linklet. In the low level, because that the expander represents everything in terms of linklets, the only way to start an evaluation is by instantiating a linklet as we discussed in Section 1.2,

Although the use of linklets provided us with lots of cool Racket features for free, such as REPL, it came with the cost of losing most of the control in evaluating core Racket library modules in general. Pycket has to blindly follow the implementation provided in the bootstrap linklets. This is good from the perspective of performance comparison between Pycket and Racket, since both systems evaluating exactly the same implementation of the module system, macro system, and all the others that are exported with the bootstrap linklet. However, since Pycket doesn't have any static access to the Racket codes, it is hard (if not impossible) to communicate hints to the JIT for the execution of any Racket function that we received from the exported linklet (e.g. using decorators like @jit.unroll_safe). Because of this, and because the JIT is automatically generated by the RPython framework, it is not a trivial task to customize and specialize the JIT specifically for Racket modules implemented on the linklets at the high level (e.g. the macro system or the module system).

Because of the implementation differences between the old Pycket and the new Pycket, and also the developments on Racket, currently the old Pycket is not able to work with the latest version of Racket (7.0.0.10), and the new Pycket is not able to work with the old Racket (6.12) that the old Pycket was working on. Therefore, in this section, we're going to separately introduce with our benchmarks

- the difference between the old Racket (6.12) and the new Racket (7.0.0.10), and
- the difference between the old Pycket and the new Pycket.

to identify interesting points where the new Pycket might got slower against the old Pycket, while the new Racket got faster against the old one.

Setup We're basing our experiments on Pycket's 2015 ICFP paper [1], in which the old Pycket was being compared to the Racket 6.1. Here we're using Racket 6.12, because that some of the developments on Racket between 6.1 and 6.12 and the corresponding developments on Pycket rendered the old Pycket unable to run on 6.1.

We use the *Larceny Cross-platform Benchmarks*, each in *#lang racket/base*. We performed the experiments on IU's Karst cluster, where each node is an IBM NeXtScale nx360 M4 server equipped with two Intel Xeon E5-2650 v2 8-core processors, each node having 32 GB of RAM, running Red Hat Enterprise Linux (RHEL) 6 with kernel 2.6.32-754.2.1.el6.x86_64.

Every benchmark was run 100 times uninterrupted at highest priority in a new process. The execution time was measured in the program and, hence, does not include start-up (neither loading and expanding the *racket/base*); however, warm-up was not separated, so all times include JIT compilation. Analysis is performed using the ReBench's analysis tool. [2]

All of the benchmark sources, including the script generators for the Karst cluster can be found in the https://github.com/cderici/pycket-performance

[2] https://github.com/smarr/ReBench

3.3 Experiment Results

We start this section by showing the results for the old Pycket vs Racket 6.12, as a sanity check. Since the Pycket's ICFP paper runs Pycket against the Racket 6.1, we wanted to see if there is a major change in Pycket's performance between Racket 6.1 and 6.12. As can be seen in Figure 21, nothing major has changed, as the results are pretty much aligned with what we see in the paper. [1]

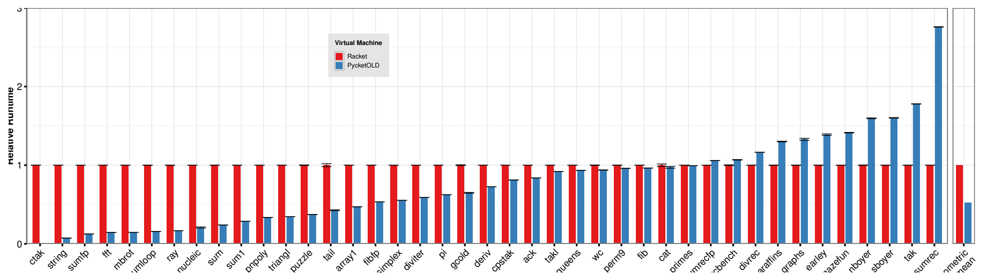

Fig. 21. Old Pycket vs Racket 6.12 - normalized to Racket, lower is better

Next, we show the results of Racket 6.12 vs. Racket 7.0.0.10 in Figure 22 below. The first thing to notice is, while for almost all the other benchmarks the performance is more or less the same, for the benchmark *sum1*, we observe a 3x slowdown in the new Racket. This is because that the *sum1* benchmark calls *read* at each iteration, and in Racket 6.12 the *read* is implemented in C, while in Racket 7.0.0.10 it is in Racket. All the other benchmarks are below the point of being an interesting difference between the two Rackets. However, while we evaluate Pyckets' performances we use the information here in a binary fashion, where we look for benchmarks for which the Racket 7.0.0.10 got (with any difference) faster and the new Pycket got slower.

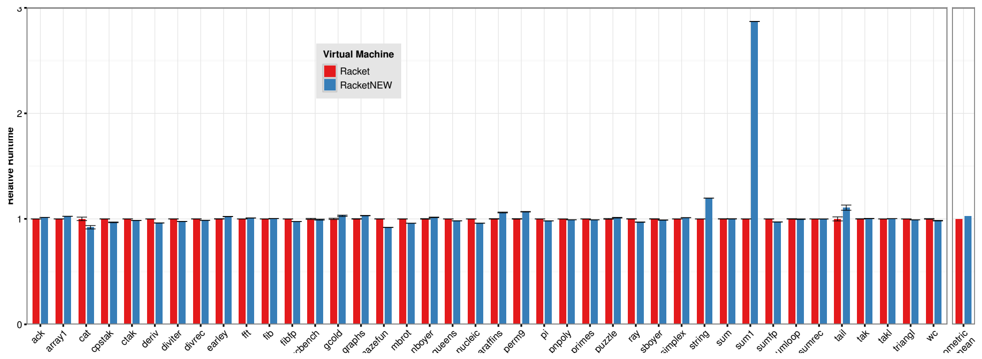

Fig. 22. Racket 6.12 vs. Racket 7.0.0.10 - normalized to Racket 6.12, lower is better

Finally, we show the results of the old Pycket vs. the new Pycket in Figure 23. Note that the behavior we see for *sum1* in Figure 22 is observable here in the new Pycket too. This is precisely why we're looking for a result where the new Racket got faster and the new Pycket got slower, to try to isolate and easily identify the performance slowdowns caused only by the implementation of linklets on Pycket.

After an examination of the results, we identify *mazefun*, *deriv*, *nqueens*, *nucleic*, *sumrec* and *triangl* as our cases of study, where the new Racket got faster, but the new Pycket got slower.

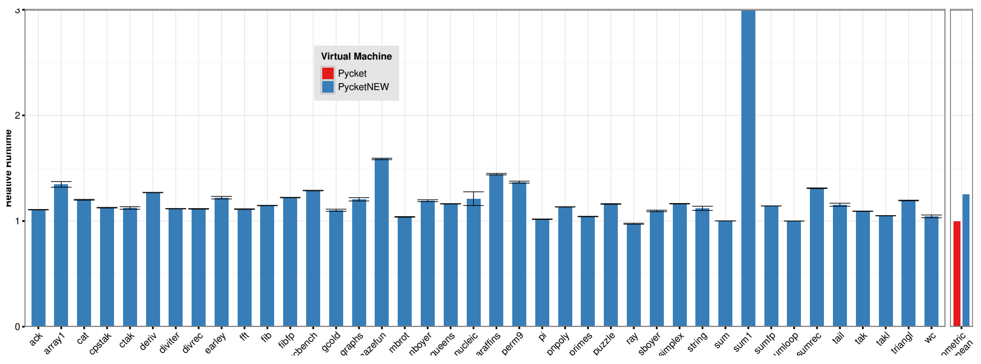

Fig. 23. Old Pycket vs. New Pycket - normalized to Old Pycket, lower is better

3.4 Discussion on the Results

To evaluate the results, for all the benchmarks we extracted the JIT back-end logs and looked at individual loops that are traced during the execution of the benchmarks for both Pyckets, focusing on the several benchmarks we identified before as interesting.

One of the issues of looking at the traces of old and new Pyckets is, now that the expansion of the modules (including the *racket/base*) are part of the JIT, we have three orders of magnitude many more traces in the new Pycket than the old one.[3] Recall that the old Pycket is running the Racket binary ahead of time to expand the program and then start executing. Table 1 demonstrates the effect of including the expansion in the runtime, for the mazefun benchmark. The "Tracing time" shows the time spent on tracing and "Backend time" shows the time spent on compiling the traces (generating Assembly for the loop). As we can see in the table, the number of loops that are traces are substantially different in old and new Pycket. This makes it a lot harder to identify the loop inside the trace log that corresponds to the interesting loop of the benchmark being

[3] Note that the timing of the benchmarks are done in the program itself, so they don't include the time spent on neither loading the Racket library modules nor the expansion of the benchmark being run.

	Old Pycket	**New Pycket**
Tracing time:	0.262688	28.287384
Backend time:	0.188901	5.581448
Total # of loops:	43	893
Total # of bridges:	250	6985
Total # of guards:	21401	3603638

Table 1. Differences in JIT summaries between Old Pycket and New Pycket for the *mazefun* benchmark

run. The get around this issue and identify the interesting loop, for each benchmark we generated the JIT back-end counts that includes the information on how many times a trace has been used. Looking at the mostly used trace for each benchmark showed us the interesting loop we are interested in.

Looking at the individual traces for all the benchmarks and identifying the interesting loops, we observed that the traces are mostly the same in both versions of Pycket, with some minor but essential differences that made the difference in performance results. This enabled us to clearly identify the additional overheads caused by the linklet implementation, and form some hypotheses on the behavior of our benchmarks.

Recall that, as we discussed in the Redex model in Section 1.3, the interpreter needs to have a handle to the current target instance to lookup for any variable that couldn't be found in the top level environment (i.e. not defined by the linklet). For this reason, we put the current linklet (target) instance to the environment to communicate it with the interpreter. Also recall that in the model (as well as in the Pycket implementation) we put Cells in the environment instead of just the values, in other words we put a pointer to the linklet variable. That's because any modifications on the variable (i.e. via a set! or target overwrite) need to be reflected on any instance that has that variable (as well as, for example, any closure that captured an environment that includes the variable).

Another interesting point is, in Racket (and Chez Scheme) implementation of linklets, the variables have *constance* information that's being used in various optimizations. Although Pycket's linklet variables have the *constance* information, we use it only as a check when a linklet variable is being modified. If a "constant" linklet variable is being modified, we signal an error.

These points about keeping the "current linklet instance" in the environment, using cells, and the linklet variables' constance play a major role in the overheads we observe in the traces.

3.5 Hypotheses for the Current Performance Overheads

The Figures 24 and 25 below demonstrate a one to one difference between typical traces we observe in the old and new Pycket across all benchmarks. The example is a cleared and simplified loop traced for the *sumrec* benchmark. In the Figure 25, additional lines within the inner loop are marked with ++. Here are the main observations and hypotheses to explain the current performance behaviors of Pycket with linklets, based on the individual traces we examined for almost every benchmark:

One observation that's visible in all the traces is, because of the additional *current_linklet_instance* field in our environment, we have an additional 8 bits (pointer) on each environment cell we create.

As discussed above, there are two ways to lookup a variable, **i)** top level environment, and *ii)* the *current_linklet_instance*. This indirection has a couple of effects on the traces:

- Whenever a variable is got from the *current_linklet_instance*, JIT has to put a guard to check if the instance and the value is received non-null. This is visible in all of the benchmarks.
- The inner loops have additional parameters for the *current_linklet_instance* value and (for some unclear reason) the top level environment.
- For any variable that's passed to the inner loop which is originally received from the *current_linklet_instance*, there's a guard within the inner loop to check whether the variable is null. Because of a currently unclear reason, the optimizer is not able to optimize away those guards, leaving the loop with an extra check in each iteration. We hypothesize that this is the primary reason for the benchmarks that have a lot of indirections in them, such as mazefun, since we observe many interconnected inner loops with additional parameters being checked at each iteration.
- Because of the extra field in the environment for the *current_linklet_instance*, whenever we create an environment cell there is an extra setfield_gc operation.

The handling of the linklet variables also causes some negative effects that are clearly visible on the traces. First of all, as we discussed above, because that we're not communicating the *constance* information with the JIT, the optimizer is not able to remove the guards that are clearly removed from the corresponding trace for the old Pycket. This is observed on the traces for *nqueens*, *deriv*, *sumrec*, *triangl* benchmarks.

The combination of these issues/hypotheses are visible in all the benchmark traces, and enough to explain the performance behaviors, as these are the *only* differences between the traces of the two versions of Pycket. Aside from these observations the most used traces of both systems are identical.

References

1. S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. *Pycket: A tracing JIT for a functional language*. In Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ACM, 22–34, 2015.

2. C. F. Bolz, A. Cuni, M. Fijakowski, and A. Rigo. *Tracing the meta-level: PyPy's tracing JIT compiler*. In Proc. ICOOOLPS, pages 18–25, 2009.
3. C. F. Bolz and L. Tratt. *The impact of meta-tracing on VM design and implementation*. Science of Computer Programming, 2013.

Outer Loop
```
   i11 = getfield_gc_i(p0, ConsEnvSize1Fixed.inst_vals_fixed_0)
```
Inner Loop
```
  1.  label(i11, p3, p0, p1)
  2.  guard_not_invalidated()
  3.  i18 = int_lt(i11, 0)
```
termination check
```
  4.  guard_false(i18)
  5.  i20 = int_sub(i11, 1)
```
construct a new environment cell
```
  6.  p21 = new_with_vtable()
  7.  p22 = new_with_vtable()
  8.  setfield_gc(p21, i11)
  9.  setfield_gc(p22, p3)
  10. setfield_gc(p21, ConstPtr(ptr24))
  11. setfield_gc(p22, p0)
  12. setfield_gc(p22, p1)
  13. setfield_gc(p22, ConstPtr(ptr25))
```
jump back to loop header
```
  14. jump(i20, p3, p21, p22)
```

Fig. 24. Optimized trace for `sumrec` inner loop on old Pycket

Outer Loop
```
    p11 = getfield_gc_r(p1, Cont.inst_env 16 pure>)
    guard_class(p11)
    p30 = getfield_gc_r(p11, ConsEnv.inst_current_linklet_instance)
    guard_nonnull(p30)
    p31 = getfield_gc_r(ConstPtr(ptr29), W_Cell.inst_w_value)
    guard_nonnull_class(p31, W_CellIntegerStrategy)
    i11 = getfield_gc_i(p31)
    guard_value(i11)
```
Inner Loop
```
  1.  label(p11, p30, i11, p3, p0, p1)
  2.  guard_not_invalidated()
  3.  i18 = int_lt(i11, 0)
  ++  guard_value(i11)
```
termination check
```
  4.  guard_false(i18)
  5.  i20 = int_sub(i11, 1)
```
construct a new environment cell
```
  6.  p21 = new_with_vtable()
  7.  p22 = new_with_vtable()
  8.  setfield_gc(p21, i11)
  9.  setfield_gc(p22, p3)
  ++  setfield_gc(p21, ConstPtr(ptr30))
  10. setfield_gc(p21, ConstPtr(ptr24))
  11. setfield_gc(p22, p0)
  12. setfield_gc(p22, p1)
  13. setfield_gc(p22, ConstPtr(ptr25))
```
jump back to loop header
```
  14. jump(p11, p30, i20, p3, p0, p1)
```

Fig. 25. Optimized trace for `sumrec` inner loop on new Pycket