Some Experiments in Pyrlang with RPython

Simple Introduction

Pyrlang is an Erlang's BEAM bytecode interpreter written in RPython. It implemented about 25% BEAM instructions and can support integer calculation (without bigint however), closures, exception handling, some operators to atom, list and tuple, user modules, and multi-process in single core. Pyrlang is still in development.
There are some differences between BEAM and the VM of PyPy:

- BEAM is a register based VM, different from the stack based VM in PyPy.
- There is no traditional call-stack in BEAM, the Y register in BEAM is like call-stack, but Y register can also store some variables at sometime.
- There is no typical language level threads and OS level threads in BEAM but only language level process, whose behavior is very similar with the actor model.

As for bytecode dispatch loop, Pyrlang uses a while loop to fetch instructions and operands, call the function corresponding to every instruction, and jump back to the head of the while loop. Due to the differences between RPython' call-stack and BEAM's Y register, we determined to implement and manage the Y register by hand. On the other hand, PyPy use RPython's call-stack to implement Python's call-stack, so as a result the function for dispatch loop in PyPy will call itself recursively but it does not happen in Pyrlang.

Usually the bytecode instructions for function invocation are distinguished by the Erlang compiler (erlc) into CALL (for normal invocation) and CALL_ONLY (for tail recursive invocation), and people can use a trampoline semantic to to implement it, that is:

- For CALL instruction, the VM will push current instruction pointer (or called program counter in PyPy) to Y register, and jump to destination label. When encountering a RETURN instruction, the VM will pop the instruction pointer from the Y register and return to the place of instruction pointer to continue executing outer function.
- For CALL_ONLY, the VM will simply jump to destination label, without any operations to Y register. As a result, the tail recursive invocation will never increase Y register.

In current implementation, we only insert the JIT hint of can_enter_jit following the CALL_ONLY instruction, it means JIT will only trace the tail-recursive invocation in Erlang code, which has a very similar semantic with the loop in most imperative programming languages including Python.
We have also written a single scheduler to implement the language level process in single core. There is a run able queue in the scheduler, and for each time, the scheduler will pop one element, which is a process object with dispatch loop, from the queue, and

execute the dispatch loop of it. In the dispatch loop, however, there is a counter call "reduction" inside the dispatch loop, the reduction will decrease during the execution of loop, and when reduction becomes 0, the dispatch loop will break. Then, the scheduler will push that element into the run able queue again, and pop next element for the queue, and so on.
We are also planning to implement multi-process in multi-core CPU, which will require multiple schedulers and even multiple run able queues for each core, but it will be another story :-).

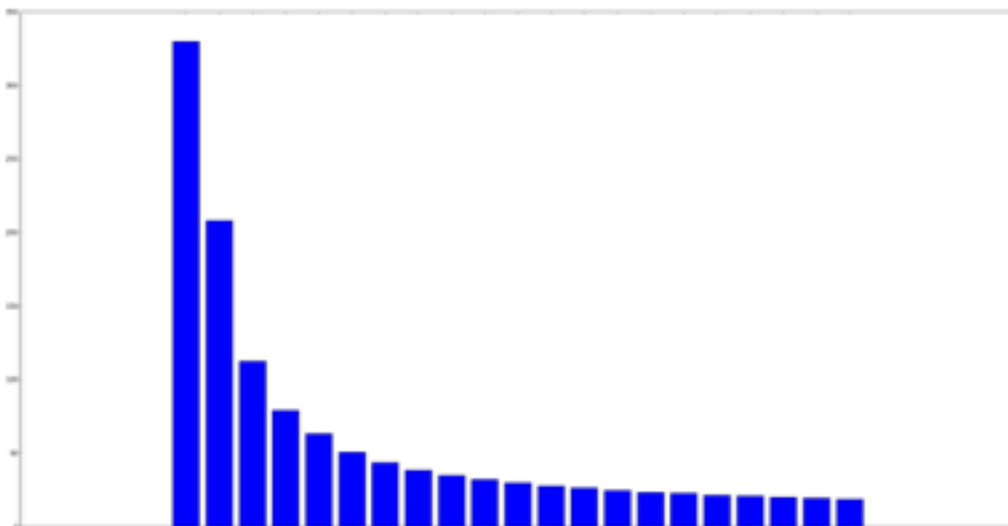
Some Experiments

We firstly wrote two benchmark programs of Erlang:

- FACT: a benchmark to calculate the factorial with tail-recursive style, but due to we haven't implemented big int, we therefore do a remainder calculation to the argument for next iteration, so the number will never overflow.
- REVERSE : the benchmark will firstly create a reversed list of number, such as [20000, 19999, 19998, …], and apply a bubble sort to it.

The Value of Reduction

We used REVERSE to evaluate the JIT in different value of reduction:



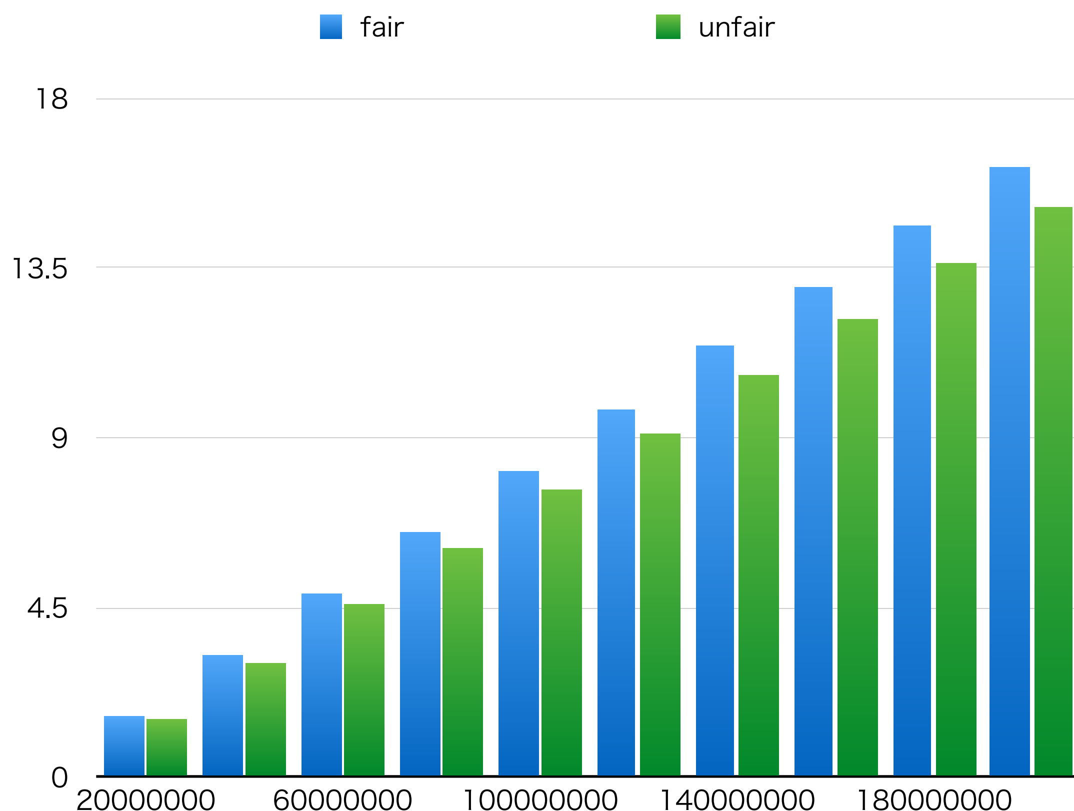
The X axis is the value of reduction, and the Y axis is the execution time (by second).
It seems that, when the value of reduction is small, the reduction will influent the performance significantly, but when reduction become larger, it will only increase the speed very slightly. In fact, we use 2000 as the default reduction value (as well as the reduction value in official Erlang interpreter).
Surprisingly, the trace will always be generated even when the reduction is very small, such as 0, which means the dispatch loop can only run for a very limited iterations, and

the language level process will execute fewer instructions than an entire loop in one switch of scheduler). And the generated trace are almost the same regardless of different reduction values.
Actually, the RPython JIT only cares what code it meets, but does not care who executes it, thus the JIT will always be generated at the experiment above. The trace even can be shared among different threads if they execute the same code.
The overhead at low reduction value may be from the scheduler, which switch from different processes too frequently, and from the switch between bytecode interpreter and native code too frequently, but not from JIT itself.
There is more explanations from Armin Rigo:

> the JIT works well because you're using a scheme where some counter is decremented (and the soft-thread interrupted when it reaches zero) only once in each app-level loop. The soft-thread switch is done by returning to some scheduler, which will resume a different soft-thread by calling it. It means the JIT can still compile each of the loops as usual, with the generated machine code containing the decrease-and-check-for-zero operation which, when true, exits the assembler.

Fair Process Switching vs. Unfair Process Switching

We are also concerned about the timing for decreasing reduction value. In our initial version of Pyrlang, we decrease reduction value at every local function invocation, module function invocation and BIF (built-in function) invocation, as what official Erlang



interpreter do. However, since the JIT in RPython basically traces target language loop, which is the tail recursive invocation in Pyrlang, it is typically better to keep the loop entire during one switch of language level process. So we modified Pyrlang, made the reduction decrement only occur after CALL_ONLY, which is actually the loop boundary of target language.

Of course, this strategy may cause an "unfair" execution among language level processes, for example if one process has only long sequence code, it will be executed until the end of the code; on the other hand, if a process has some very short loop, it may be executed by very limited steps then be switched out by the scheduler. However, in real world, this "unfairness" usually is thought acceptable, and be used in may VM implementations including PyPy for improving the whole performance.

We compared these two versions of Pyrlang in FACT, because there are some BIF invocations inside the loop, so the reduction decrement is quite different. In old version the process may be suspended at loop boundaries or other function invocation, but in new version, only loop boundaries.

It showed that the strategy is effective, which removed around 7% overhead. We have also compared it in reverse, but since there is not extra invocations inside the trace, it cannot provide any performance improvement. In real world, we believe there will usually be more than one extra invocations inside one loop, so this strategy will accounts at most cases.

Comparison with Default Erlang and HiPE

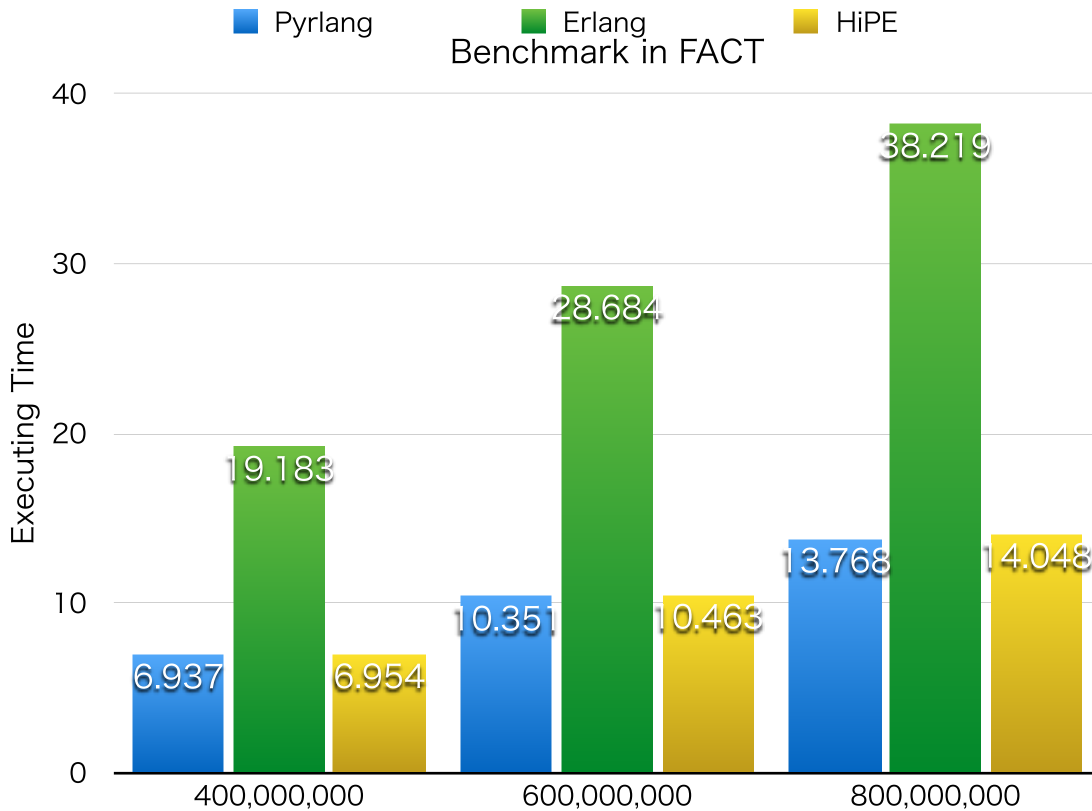
Finally, we tried to compare the performance with default Erlang interpreter, and the HiPE (High Performance Erlang) complier. HiPE is an official Erlang compiler which can compile Erlang source code to native code, so the speed of Erlang programs will be improved obviously but lost the probability instead.

Please note that Pyrlang is still in development, so at some situations it will do less work than the default erlang interpreter, such as checking integer overflow when dealing with big integer, checking and adding lock when accessing message queues in language level process, and therefore be faster. The final version of Pyrlang may be slower at this view.

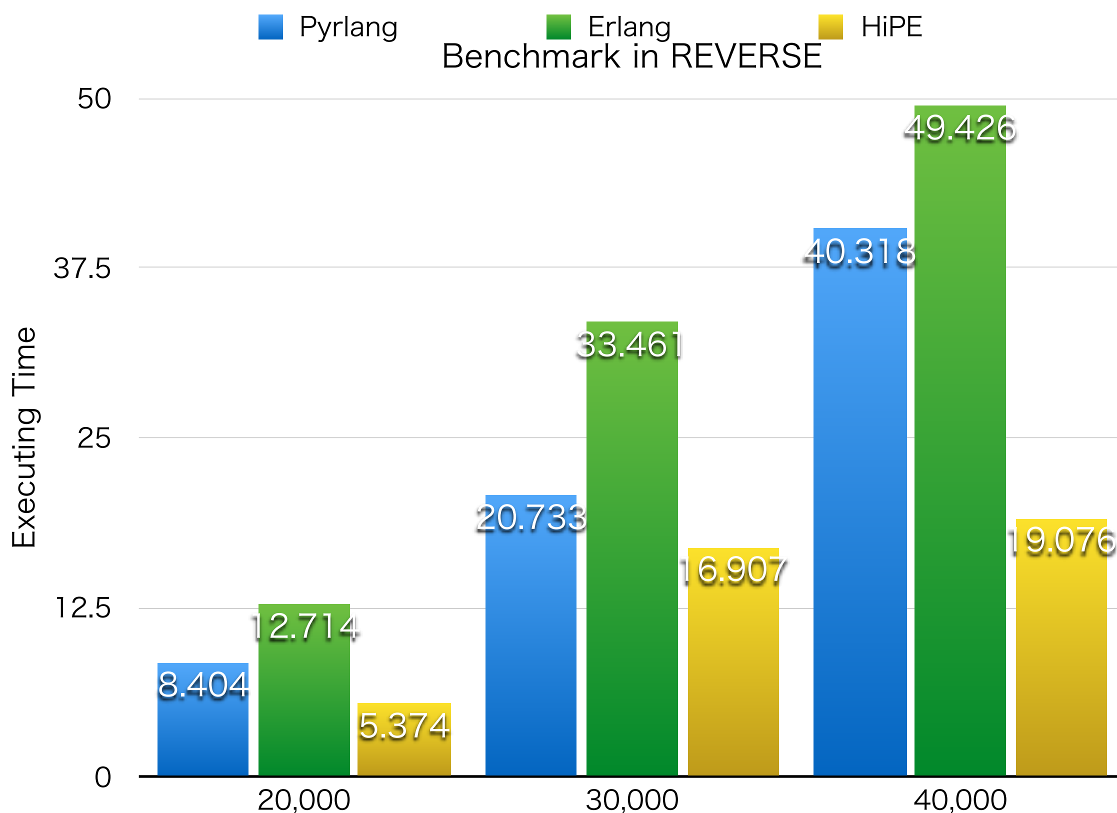
We use both two benchmark programs above, and make sure both of them will be executed more that 5 seconds to cover the JIT warming time for RPython. The experiment environment is in a OS X 10.10 machine with 3.5GHZ 6-Core Intel Xeon E5 CPU and 14GB 1866 MHz DDR3 ECC memory.

Firstly let's see the result of FACT, the graph shows that Pyrlang will run 177.41% faster on average than Erlang, and has almost the same speed as HiPE. However, since we haven't implemented big integer in Pyrlang, the arithmetical operators will not do any extra overflow checking, so it's reasonable that the final version for Pyrlang will be slower then the version now and the HiPE.

As for REVERSE, the graph shows that Pyrlang will run 45.09% faster than Erlang, but 63.45% slower than HiPE on average. We think it's reasonable because there are only



few arithmetical operators in this benchmark so the speeds of these three implementations are closer. However, we observed that, at the scale of 40,000, the speed of Pyrlang slowed down significantly (111.35% slower than HiPE) compared with other two scales (56.38% and 22.63% slower than HiPE).
Until now we cannot conclude a convincing reason why Pyrlang slows down at that point. We guess that the overhead might be from GC. This is because the BEAM bytecode will provide some GC hints to help the default Erlang to perform some GC behaviors immediately. For example, using GC_BIF instead of BIF instruction to tell VM that here may be a GC chance, and tell the VM how many live variables there should be



around one instruction. In Pyrlang we never use these kind of hints but rely on RPython's GC totally, so when there are huge number of objects during runtime, (as for REVERSE, it should be the Erlang list object) the speed therefore slows down.