Experiments in Pyrlang with RPython

Pyrlang is an Erlang BEAM bytecode interpreter written in RPython.

It implements approximately 25% of BEAM instructions. It can support integer calculations (but not bigint), 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, whereas the VM in PyPy is stack-based.
- There is no traditional call-stack in BEAM. The Y register in BEAM is similar to a call-stack, but the Y register can sometimes store some variables.
- There are no typical language-level threads and OS-level threads in BEAM; only language-level processes, whose behavior is very similar to the actor model.

Regarding 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 the RPython call-stack and BEAM's Y register, we decided to implement and manage the Y register by hand. On the other hand, PyPy uses RPython's call stack to implement Python's call stack. As a result, the function for the dispatch loop in PyPy calls itself recursively. This does not happen in Pyrlang.

The Erlang compiler (erlc) usually compiles the bytecode instructions for function invocation into CALL (for normal invocation) and CALL_ONLY (for tail recursive invocation). You can use a trampoline semantic to implement it:

- CALL instruction: The VM pushes the current instruction pointer (or called-program counter in PyPy) to the Y register, and jumps to the destination label. When encountering a RETURN instruction, the VM pops the instruction pointer from the Y register and returns to the location of the instruction pointer to continue executing the outer function.
- CALL_ONLY instruction: The VM simply jumps to the destination label, without any modification of the Y register. As a result, the tail recursive invocation never increases the Y register.

The current implementation only inserts the JIT hint of can_enter_jit following the CALL_ONLY instruction. This means that the JIT only traces the tail-recursive invocation in Erlang code, which has a very similar semantic to the loop in imperative programming languages like Python.

We have also written a single scheduler to implement the language level process in a single core. There is a runable queue in the scheduler. On each iteration, the scheduler pops one element (which is a process object with dispatch loop) from the queue, and executes the dispatch loop of the process object. In the dispatch loop, however, there is a counter-call "reduction"einside the dispatch loop. The reduction decrements during the execution of the loop, and when the reduction becomes 0, the dispatch loop terminates. Then the scheduler pushes that element into the runable queue again, and pops the next element for the queue, and so on.

We are planning to implement a multi-process scheduler for multi-core CPUs, which will require multiple schedulers and even multiple runable queues for each core, but that will be another story. :-)

Methods

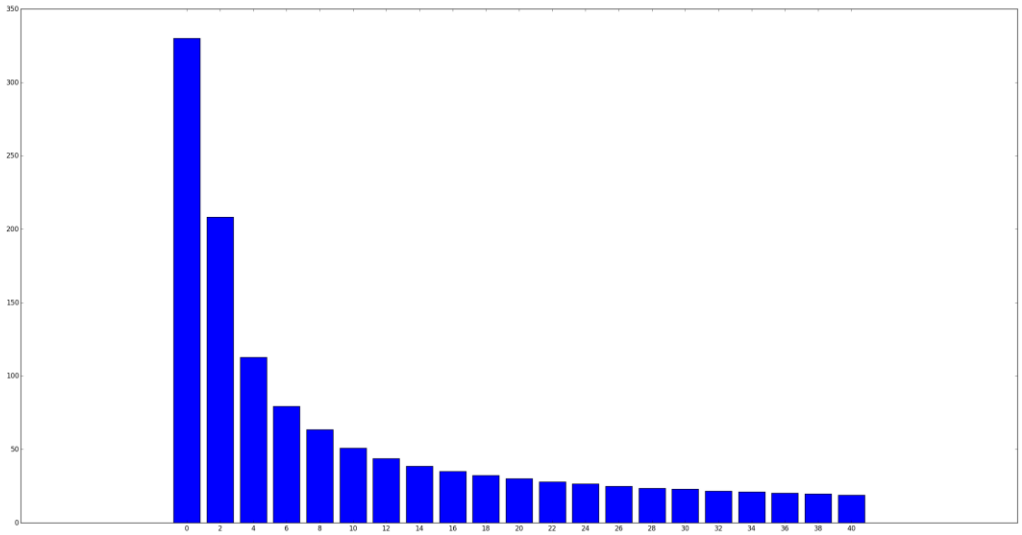
We wrote two benchmark programs of Erlang:

- FACT: A benchmark to calculate the factorial in a tail-recursive style, but because we haven't implemented big int, we do a remainder calculation to the argument for the next iteration, so the number never overflows.
- REVERSE: The benchmark creates a reversed list of numbers, such as [20000, 19999, 19998, …], and applies a bubble sort to it.

Results

The Value of Reduction

We used REVERSE to evaluate the JIT with different values 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 influences the performance significantly, but when reduction becomes larger, it only increases the speed very slightly. In fact, we use 2000 as the default reduction value (as well as the reduction value in the official Erlang interpreter).

Surprisingly, the trace is always generated even when the reduction is very small, such as 0, which means the dispatch loop can only run for a very limited number of iterations, and the language level process executes fewer instructions than an entire loop in one switch of the scheduler). The generated trace is 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 always generates the results above. The trace even can be shared among different threads if they execute the same code.

The overhead at low reduction value may be due to the scheduler, which switches from different processes too frequently, or from the too-frequent switching between bytecode interpreter and native code, but not from JIT itself.

Here is more explanation 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."

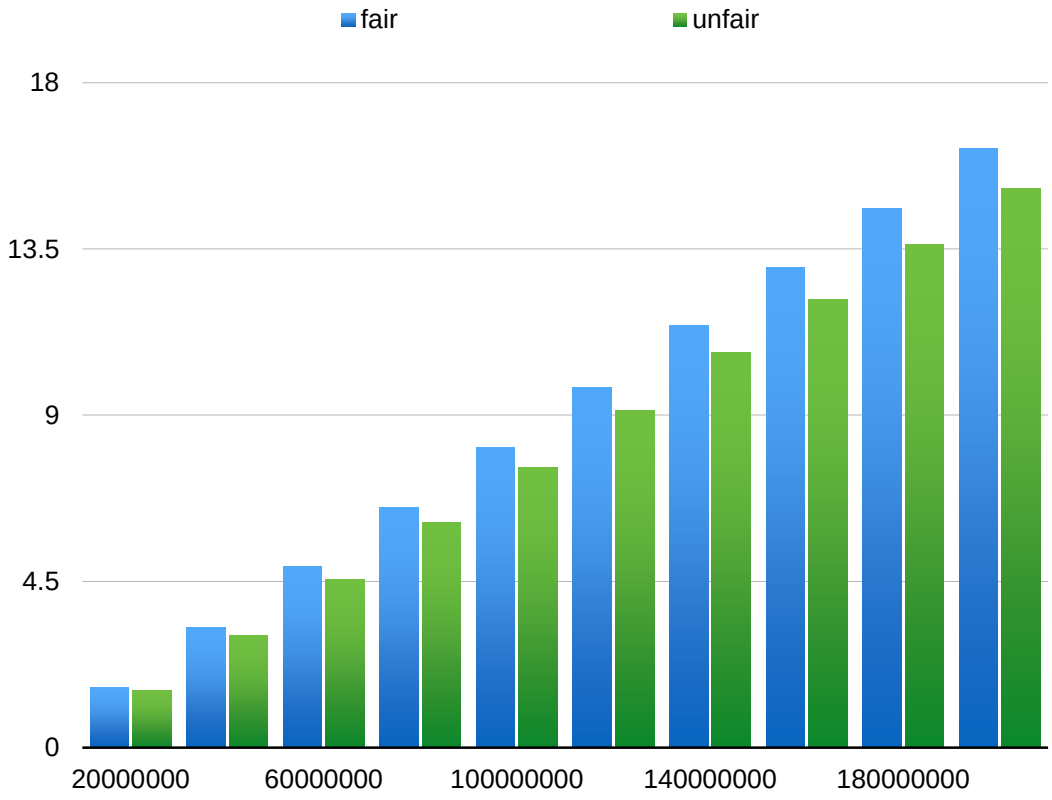
Fair Process Switching vs. Unfair Process Switching

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, since this is what the official Erlang interpreter does. However, since the JIT in RPython basically traces the target language loop (which is the tail recursive invocation in Pyrlang) it is typically better to keep the loop whole during a switch of the language level process. We modified Pyrlang, and made the reduction decrement only occur after CALL_ONLY, which is actually the loop boundary of the target language.

Of course, this strategy may cause an “unfair”nexecution among language level processes. For example, if one process has only a single long-sequence code, it executes until the end of the code. On the other hand, if a process has a very short loop, it may be executed by very limited steps then be switched out by the scheduler. However, in the real world, this “unfairness”nis usually considered acceptable, and is used in many VM implementations including PyPy for improving the overall performance.

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

We show that the strategy is effective, removing around 7% of the overhead. We have also compared it in



REVERSE, but since there are no extra invocations inside the trace, it cannot provide any performance improvement. In the real world, we believe there is usually more than one extra invocation inside a single loop, so this strategy is effective for most cases.

Comparison with Default Erlang and HiPE

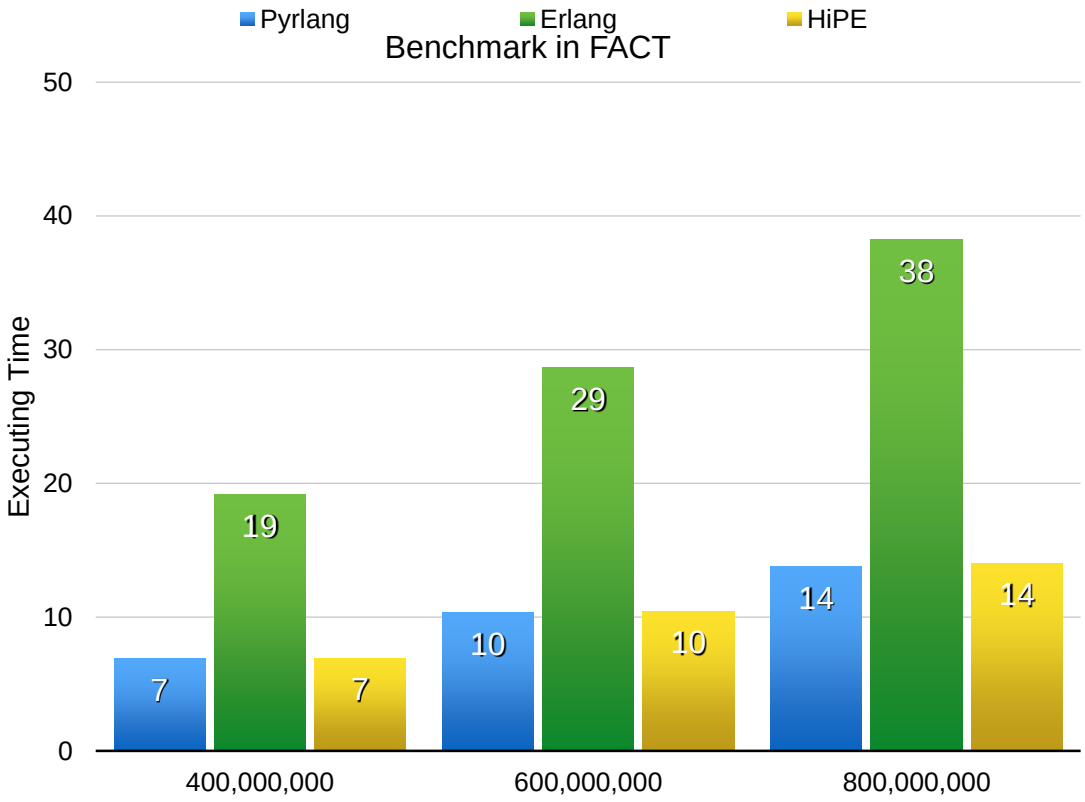
We compared the performance of Pyrlang with the default Erlang interpreter and the HiPE (High Performance Erlang) complier. HiPE is an official Erlang compiler that can compile Erlang source code to native code. The speed of Erlang programs obviously improves but loses its generality instead.

Please note that Pyrlang is still in development, so in some situations it does less work than the default Erlang interpreter, such as not checking integer overflow when dealing with big integer, and not checking and adding locks when accessing message queues in the language-level process, so is therefore faster. The final version of Pyrlang may be slower.

We used the two benchmark programs above, and made sure both of them are executed for more than five seconds to cover the JIT warm-up time for RPython. The experiment environment is a OS X 10.10 machine with 3.5GHZ 6-core Intel Xeon E5 CPU and 14GB 1866 MHz DDR3 ECC memory.

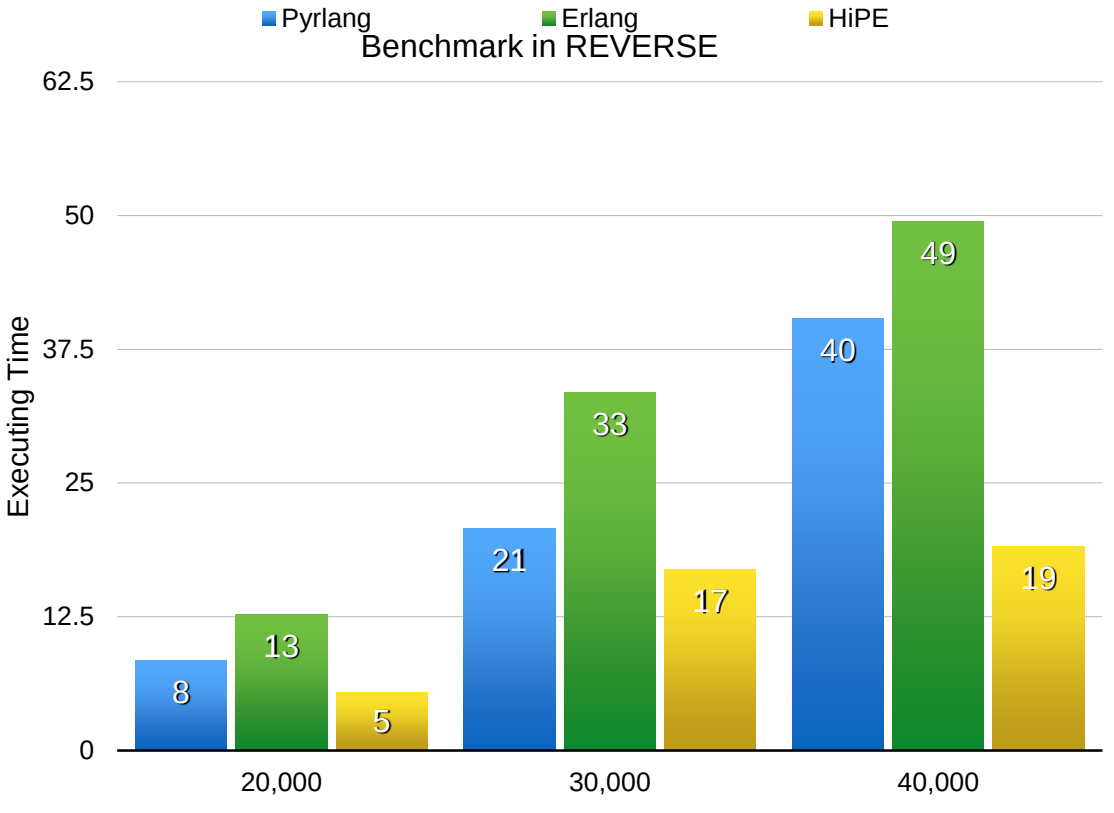
Let's look at the result of FACT. The graph shows that Pyrlang runs 177.41% faster on average than Erlang, and runs at almost the same speed as HiPE. However, since we haven't implemented big integer in Pyrlang, the arithmetical operators do not do any extra overflow checking. It is reasonable that the final version for Pyrlang will be slower than the current version and HiPE.

As for REVERSE, the graph shows that Pyrlang runs 45.09% faster than Erlang, but 63.45% slower than



HiPE on average. We think this is 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 the other two scales (56.38% and 22.63% slower than HiPE).

Until now we can only hypothesize why Pyrlang slows down at that scale. We guess that the overhead might be from GC. This is because the BEAM bytecode provides some GC hints to help the default Erlang compiler to perform some GC operations immediately. For example, using GC_BIF instead of a BIF instruction tells the VM that there may be a GC opportunity, and tells the VM how many live variables should be around one instruction. In Pyrlang we do not use these kinds of hints but rely on RPython's GC



totally. When there are a huge number of objects during runtime, (as for REVERSE, it should be the Erlang list object) the speed therefore slows down.