Orca: GC and Type System Co-Design for Actor Languages

The artifact is the implementation of ORCA in Pony, which is what we describe in the paper Sections 3, 4 and 5. Because Pony is actively developed, we provide a VM with a snapshot of the version that we used for the paper.

As examples of programs that use ORCA via Pony, we provide the benchmarks that were used in Section 6 of the paper, along with instructions how to run them and interpret the results. Reproducing the results of the paper requires a big machine (we scale up to 64 cores), and additionally there will be jitter stemming from running inside a virtual machine. For future researchers who wants to build on, not just the implementation of ORCA in Pony, but make performance comparisons, we do include the source files for the Erlang and Akka versions of the benchmark programs we run. We do not include Erlang or Java -- in particular licensing constraints on C4 (that we compare against in the paper) does not permit us to do so. Instead, we focus on comparing Pony with ORCA against Pony without ORCA to more clearly highlight differences stemming from garbage collection.

The VM includes the Pony compiler and runtime system. We provide the source code for that on GitHub plus a VM with different runnable versions of Pony -- the ones used in the paper. The source code of Pony used for our evaluation can be found at:
https://github.com/jupvfranco/ponyc.
You can easily verify that the commit fd7796b7f7f633c6764f2296c236042004454f74 is before the artefact deadline. A reviewer who wishes to take a more recent version of Pony for a spin will find instructions for installation at:
https://github.com/ponylang/ponyc
Since this is an active Open Source project, this will keep moving after the deadline. However, **the artefact comes with all codes included**, so the above links are mostly about pedigree.

Again: note that all results will be heavily dependent on the resources available to the VM. For example, when running Pony with the setup used for 64 cores on a 4 core machine, Pony will suffer heavily due to oversubscription of each core.

Getting Started Guide

Requirements:

- VirtualBox

Installation Instructions:

1. Download the zip containing a pre-built VM (2.07G).
2. Unzip it.
3. Double-click in the .vbox file, and press the green start arrow of the VirtualBox's User Interface. *(If your setup works in another way, e.g., is headless, we assume you know how to get the VM into VirtualBox and get started.)*
4. It will ask you for a login and password. Use vagrant in both.
5. A terminal should now be available, where the artefact can be evaluated.
6. Listing (ls) all the files should show three directories:
    a. ponyc: containing the code of the pony compiler and runtime.
    b. benchmarks: containing the benchmarks code
    c. run: containing scripts to run different tests

The Pony compiler (ponyc): In this directory, you can find the pony sources, built in three different modes:

- *release*, which can be used to run a benchmark without telemetry overhead
- *release-telemetry*, which can be used to run a benchmark and to print telemetry information
- *release-nogc*, which can be used to run a benchmark without garbage collection (it also ignores scanning upon message receiving and runtime messages related to garbage collection)

Note: referred to as <pony_release> in the Step-by-Step Instructions.

Benchmarks directory (benchmarks): In this directory, one can find the code for the benchmarks used in the Evaluation of ORCA.

- trees <N>
- trees2 <N>
- heavyRing <N> <M> <K>
- rings <N> <M> <K>
- mailbox <N> <M>
- serverSimulation <N> <M> <K> <J>

The description of each benchmark can be found in Section 6.1 of the paper.
Although we focus on Pony and ORCA in this artifact, we also include the code used in Pony, Erlang and Akka, for comparison purposes only.
Note: referred to as <benchmark_name> in the Step-by-Step Instructions.

Each benchmark takes inputs with different formats, which we now describe:

- trees: takes the depth of trees created (a number <N>) -- the deeper the trees, the more garbage created (same for all benchmarks).
- trees2: takes the depth of trees created (a number <N>) .
- heavyRing: takes the number of actors in the ring, the depth of the tree being passed around and the number of rounds (three numbers, <N> <M> <K>) -- the higher number of rounds, the more scanning overhead on send and receive is induced which is a Pony pain point.
- rings: takes the number of actors in each ring, the number of rings created, and the number of messages sent in each ring (three numbers <N> <M> <K>).

- mailbox: takes the number of sending actors and the number of messages to pass among them (two numbers <N> <M>).
- serverSimulation: takes the number of servers (actors) spawned, the number of requests sent to each server, the minimal depth and the maximal depth of trees being created (four numbers <N> <M> <K> <J>). These last two arguments were the same when running benchmarks.

Note: when running one of the scripts described below, if running any benchmark that takes more than one argument (e.g. mailbox), it is required to pass the input with quotes (e.g. "10 100" for the mailbox).

Scripts directory (run): In this directory, one can find the bash scripts used to run Pony. We describe how to run these in the step-by-step instructions below.

Playing around: If a reviewer wants to modify our programs, or write her own, we refer to the Pony tutorial here https://tutorial.ponylang.org/. There are some small starting programs in Figures 2, 3 and 4 in the paper as starting points in addition to the benchmark programs.

Finding the ORCA-related files

Given the size of the Pony run-time (about 13 KLOC), for the researcher wanting to study the full code of the implementation of ORCA, the following files are the key files related to ORCA. Inside ponyc folder:

- try_gc in src/libponyrt/actor/actor.c is the entry point for starting a GC cycle.
- handle_message in src/libponyrt/actor/actor.c shows how Orca related messages are processed.
- ponyint_gc_sendobject in ~/github/ponyc/src/libponyrt/gc/gc.c is for tracing on sending objects across actors; similar tracing code for receiving objects, sending & receiving actors, etc. can be found in the same file as well.
- src/libponyrt/mem/heap.[h|c] contain actor-local heap.

Step by Step Instructions for Running Benchmarks

We now go over the step-by-step instructions for how to evaluate the artefact. We note that since we are running in a VM, and on a different machine (the Bulldozer machine used in the paper is a NUMA machine with 64 cores, for comparison), the results will in all likelihood be different from those in our paper.

We provide scripts to run five different tests, which we now describe. For each step, we provide a command that runs a test corresponding to a part of the evaluation. The parameters to that command can be tweaked. We also provide an example command that should give expected results in a reasonable amount of time. (Note that the steps can be carried out in any order.)

These benchmarks can be used by someone interested in ORCA (esp. In its Pony implementation) to understand its performance, and especially possible pathologies, by running Pony programs that stress ORCA in various ways. As starting points, we provide the benchmarks from the paper, plus the necessary infrastructure to run these with different input and compare the output.

All the output files can be found in the same directory of the Pony source code for the benchmark executed.

STEP 1: Scalability: used to evaluate scalability, such as described in Section 6.2 of the paper.

- Command:
  bash run/run-scalability.sh <pony_release> <benchmark_name> <input> <max_cores>
- Examples:
  bash run/run-scalability.sh release-telemetry trees 10 64
  bash run/run-scalability.sh release-nogc trees 10 64
  bash run/run-scalability.sh release trees 10 64
  … and similarly for trees2, heavyRing, rings, etc ...
- Parsing and reading the output:
  The file benchmarks/<bechmark_name>/pony/orca.scalability.<pony_release>.txt contains the output produced by the benchmark itself.
  The file benchmarks/<benchmark_name>/pony/orca.scalability.<pony_release>.log contains a sequence of lines of the form <number of cores used,iteration id, wall-clock time for benchmark execution>.
- Understanding the output:
  For this benchmark, the output is run-time measured in wall-clock time. As is visible in Figure 16, when running non-virtualized on a big machine, the run-time of the program decreases as the number of cores increase. This is a strong scalability benchmark, meaning input size does not vary.

STEP 2: Responsiveness: used to measure responsiveness, as explained in Section 6.3. of the paper. All the other tests run any given benchmark. This one only runs the serverSimulation benchmark, which is a benchmark designed to test responsiveness.

- Command:
  bash run/run-responsiveness.sh <pony_release> <input>
- Example:
  bash run/run-responsiveness.sh release-telemetry “20 200 5 5”
  bash run/run-responsiveness.sh release-nogc “20 200 5 5”
  bash run/run-responsiveness.sh release “20 200 5 5”
  Do not forget to put the argument in quotation marks
- Parsing and reading the output:
  The file benchmarks/serverSimulation/pony/orca.responsiveness.<pony_release>.log contains the differences between response times, as reported by running the benchmark.
- Understanding the output:
  This benchmark measures differences in processing times for requests to a server.

The jitter should be minimal as there are no global GC events. Compare with Figure 20 in the paper.

Note: the script is not prepared to run this rest with release-telemetry.

STEP 3: Overhead: used to measure overhead in terms of time spent on scanning upon message sending and receiving and additional runtime messages due to ORCA. It also reports the number of GC cycles and the number of application messages received, such as described in Section 6.5. All the other tests run with any given release. This one runs only with release-telemetry, which is the only one that outputs all the information needed.

- Command:
  ```
  bash run/run-overhead.sh <benchmark_name> <input>
  ```
- Example:
  ```
  bash run/run-overhead.sh trees 10
  ```
- Parsing, reading and understanding the output:
  The file benchmarks/<benchmark_name>/pony/orca.overhead.release-telemetry.log contains several blocks of text containing the following information per single execution: iteration id, number of garbage collection cycles, number of actors spawned, number of application messages, number of increment messages, number of decrement messages, number of CPU cycles spent on behaviour execution, number of CPU cycles spent on garbage collection, number of CPU cycles spent on tracing up message sending, number of CPU cycles spent on tracing up message receiving and finally the total number of CPU cycles spent by the benchmark.

STEP 4: Footprint: used to evaluate memory footprint, such as described in Section 6.6 of the paper.

- Command:
  ```
  bash run/run-footprint.sh <pony_release> <benchmark_name><input>
  ```
- Example:
  ```
  bash run/run-footprint.sh release-telemetry trees 10
  bash run/run-footprint.sh release-nogc trees 10
  bash run/run-footprint.sh release trees 10
  ```
- Reading, parsing and understanding the output:
  The file benchmarks/<bechmark_name>/pony/orca.footprint.<pony_release>.txt contains the output produced by the benchmark itself.
  The file benchmarks/<benchmark_name>/pony/orca.footprint.<pony_release>.log contains a sequence of lines containing iteration id and memory footprint in KB per execution. Compare with Figure 22 in the paper.

STEP 5: CPU usage test:

- Command:
  ```
  bash run/run-cpu-usage.sh <benchmark_name> <input> <cores>
  ```
- Example:
  ```
  bash run/run-overhead.sh trees 10
  ```
- Reading, parsing and understanding the output:
  The file benchmarks/<benchmark_name>/pony/orca.cpuusage.release-telemetry.log contains information about starting and finishing of garbage collection cycles, tracing

upon sending and tracing upon receiving, and behaviour executions. It also contains the total number of CPU cycles used by the benchmark execution.

This information is used in Figure 21 to understand the overhead of scanning on message sends -- a possible pain point for ORCA. Thus, the important numbers here is the number of INC and DEC messages (as they could theoretically flood the normal message channels), and time spent in GC, tracing on send and receive in relation to time spent in behaviour (i.e., the application logic). The information is in terms of *intervals*: `(starting time in CPU cycles --- finishing time in CPU cycles)`. *Note that there is a bug in the output of "programs execution used X CPU cycles". We are working to fix this bug. We have been unable to reproduce this bug outside of the VM. Ultimately, we felt that the simplicity of a VM-based distribution outweighed the importance of this one metric.*