Artifact description

The artifact consists of two parts:

1. The open source implementation of our system, which has been contributed to the OpenJ9 project:https://github.com/eclipse-openj9/openj9. Our system is implemented in ~25 KLOC of C++; the code is integrated into the rest of the OpenJ9 code base.

The OpenJ9 code base is split across 3 separate repositories:

- OpenJ9: https://github.com/eclipse-openj9/openj9. A stable fork for artifact evaluation with some not yet upstreamed minor changes required by our benchmarking platform can be found here: https://github.com/AlexeyKhrabrov/openj9/tree/atc22ae.
- OMR: https://github.com/eclipse/omr. A stable fork for artifact evaluation can be found here: https://github.com/AlexeyKhrabrov/omr/tree/atc22ae.
- OpenJDK extensions for OpenJ9: https://github.com/ibmruntimes/openj9-openjdk-jdk8 (we used JDK 8 in our experiments; newer JDK versions are also available). A stable fork for artifact evaluation can be found here: https://github.com/AlexeyKhrabrov/openj9-openjdk-jdk8/tree/atc22ae.

Our automated benchmarking platform (see below) is set up to build the system from source. OpenJ9 builds (including JITServer) are also available from e.g. AdoptOpenJDK: https://adoptopenjdk.net/.

2. A set of scripts (Python, shell scripts, and Docker files) that automate the benchmark runs used in our evaluation and generate the resulting graphs. This part or the artifact is available at https://github.com/AlexeyKhrabrov/jitserver-benchmarks.

We used the following open source benchmark applications in our evaluation (their pre-built jars are included in the repository linked above):

- AcmeAir: https://github.com/blueperf/acmeair-monolithic-java.
- DayTrader7: https://github.com/WASdev/sample.daytrader7.
- Spring PetClinic: https://github.com/spring-projects/spring-petclinic.

Running the largest experiments reported in the paper requires an equivalent of a cluster of 11 machines with 16 CPU cores (32 hyperthreads) each, connected with a 10 Gbit/s network with round-trip latency between machines in the low hundreds of microseconds or less. We will provide remote access to our cluster via ssh (please reach out to the authors via HotCRP for instructions and credentials). Alternatively, the experiments can be run in a public cloud such as AWS on a cluster of virtual instances with roughly equivalent resources, or on a similar set of physical machines. Our setup supports Ubuntu 18.04.

Our artifact can be used to validate the main claims in our paper, most importantly:

- JITServer can reduce application start time and warmup time and system-wide CPU and memory usage for Java applications running in containers with limited resources (which are common in the cloud.)
- Caching dynamically compiled code at JITServer is necessary to fully achieve the reduction in overall CPU usage and application start times.

Note that the experimental results are expected to be slightly different from the ones reported in the paper (even on the same hardware setup) since the OpenJ9 implementation has evolved since we conducted those experiments. However, the main conclusions should still hold.

Getting started

A minimal setup to check the basic JITServer functionality involves cloning the source code repositories, building the JDK with OpenJ9, and running a minimalistic Java application such as `java -version` with JITServer. It should take about 30 minutes, depending on how fast the machine is. The scripts assume Ubuntu 18.04. The steps are as follows.

1. Clone the benchmarks repository.

```
$ git clone https://github.com/AlexeyKhrabrov/jitserver-benchmarks
$ cd jitserver-benchmarks/
```

2. Install prerequisite packages.

```
$ sudo scripts/openj9_prereqs.sh
```

3. Fetch the JDK, OpenJ9, and OMR source code (into `./jdk/`, `./openj9/`, and `./omr/`).

```
$ ./get_sources.sh
```

4. Build the JDK (into `./jdk/build/release/`).

```
$ cd scripts/
$ ./openj9_build.sh ../jdk/ 8
```

5. Start a local JITServer instance.

```
$ ./run_jitserver.sh ../jdk/ 8 -s -c
```

6. In a separate terminal, run `java -version` (which is a small Java application) using the local JITServer instance.

```
$ cd jitserver-benchmarks/scripts/
$ ./run_jitclient.sh ../jdk/ 8 localhost -s -c -p -version
```

Both the JITServer and the client JVM should produce output (JIT compiler verbose log) to `stderr` that describes remote compilation requests generated by the client and processed by the server. Additional client instances can be launched to use the same server, in which

case they can take advantage of the cache of compiled methods at the server (which should be reflected in the verbose log messages). The server can be stopped with SIGINT (Ctrl+C).

The shell scripts support a variety of additional options. Please consult their help text and source code for the description of the arguments.

JITServer documentation can be found at https://www.eclipse.org/openj9/docs/jitserver/ and https://github.com/eclipse-openj9/openj9/tree/master/doc/compiler/jitserver. Note that caching compiled code at JITServer is not yet documented in the official documentation.

Benchmarking cluster setup overview

Our benchmarking platform assumes that the cluster consists of:

- The *control node* where the user runs the scripts to set up and run the experiments, the logs produced by the experiments are accumulated, and the results are generated from those logs.
- A set of *worker nodes* where the components of the actual benchmarks (JITServer, application JVMs, application database instances, workload generators) are run. Note that one can run the “control plane” on one of the worker nodes since its overhead is relatively small.

The benchmark “driver” scripts that run on the control node launch tasks on the worker nodes via ssh, and transfer files using rsync. For optimal performance, we highly recommend setting up ssh connection caching/multiplexing (see the comment at the top of the file `jitserver-benchmarks/benchmarks/remote.py`). A set of worker nodes is defined in a `.hosts` file that lists their host names (one per line) and other optional parameters (see the comment in the function `load_hosts()` in `remote.py`).

Setting up JITServer and benchmarks on the cluster

Our setup assumes Ubuntu 18.04 as the OS. It should be possible (but not necessarily easy) to tweak it to work on other Linux distributions. Newer Ubuntu versions might require downgrading to an older gcc version (OpenJ9 currently officially supports gcc-7, but should also support gcc-10). Different Linux distributions will need more tweaks, namely different ways of installing prerequisite packages.

The setup assumes the same credentials on all the worker nodes and requires sudo permissions on all the nodes (including the control node). The required amount of storage space is approximately 25-30 GB on each node. This includes the JDK sources and build (~16GB; can be deleted except for the JDK image if necessary), the Docker container images (~5 GB on each worker node), and the logs generated by running the experiments (up to ~8GB on the control node).

The setup steps are as follows. All the commands should be run on the control node, after completing the initial setup described above. Some of the `.py` commands will prompt for the remote user password. Use the `-v` option to enable verbose output - the trace or local and

remote (via ssh) commands invoked by the Python scripts. Most of the setup scripts generate logs (output of remote commands) under the `./logs/` directory.

1. Install prerequisite packages on the control node.

```
$ cd jitserver-benchmarks/benchmarks/
$ sudo ./prereqs.sh
$ ./python_prereqs.sh
```

2. Setup ssh key authentication on the worker nodes.

```
$ ./host_setup.py all_workers.hosts
```

3. Build the JDK with OpenJ9 on the worker nodes. This should take about the same time as the local JDK build during initial setup.

```
$ ./openj9_setup.py all_workers.hosts ../jdk/ 8 -p
```

4. Setup individual benchmarks. This will copy the relevant scripts to the worker nodes and build all the Docker container images. This should take about 40 minutes in total, depending on the machine speed.

```
$ ./acmeair_setup.py all_workers.hosts -p
$ ./daytrader_setup.py all_workers.hosts -p -d
$ ./petclinic_setup.py all_workers.hosts -p
```

Running experiments

The following script can be used to run the whole set of experiments.

```
$ cd jitserver-benchmarks/benchmarks/
$ ./all_experiments.sh all.hosts main.hosts
```

This script invokes various `run_*.py` scripts that implement different experiments. It also describes (in the comments) how long each experiment is expected to take. To run a subset of experiments, simply comment out the unused parts of the code in `all_experiments.sh`.

The mapping of the experiment names in the scripts to the sections and figures in the paper is as follows:

- `run_single.py`: Section 4.1, Figures 3-5;
- `run_density.py`: Section 4.2, Figures 6-7;
- `run_scale.py`: Section 4.3, Figure 8;
- `run_latency.py`: Section 4.4, Figure 9.

All the scripts assume a setup with 11 machines (specified by the `all_hosts_file` parameter in `all_experiments.sh`), the first 8 of which (specified by `main_hosts_file`) have 16 cores each, and last 3 can be a bit smaller (they are only used for the workload generator which is not a bottleneck).

Running the experiments on a different number of nodes and/or CPU cores per node might require resizing the experiments (changing the numbers of instances and their assignment to hosts). Please reach out to the authors to discuss this if modifying the scripts for a different hardware configuration is not straightforward.

Workload durations specified in the scripts are based on estimated warmup time, which depends on single core speed. Durations might need to be adjusted for a different hardware setup, e.g. increased if application instances do not reach peak throughput.

Running the whole set of experiments as reported in the paper takes about 10 days of machine time. This time can be reduced as follows if necessary.

- Reduce the number of repetitions from the default (5 or 3 for “density” experiments) using the `-n` option, e.g. `$ ./run_density.py acmeair all.hosts -n 1`.
- Reduce the duration (number of application instance invocations) in “density” experiments (which take the longest time). The number of invocations is specified at the top of `run_density.py` in the `configurations` variable (the 4th tuple element).

Generating results

To generate the results based on the logs produced by the experiments, run the following script. To generate results for a subset of experiments, simply comment out unused code in `all_results.sh`. Note that the `-n` arguments passed to the `run_*.py` scripts specified in `all_experiments.sh` must match in `all_results.sh`.

```
$ cd jitserver-benchmarks/benchmarks/
$ ./all_results.sh
```

This should take up to a few minutes, and will produce the main graphs under `./results/plots/` as well as a larger and more detailed additional set of graphs and `summary.txt` files (containing e.g. percentage differences in various performance metrics between) for each experiment under `./results/`.

Environment used in our evaluation

We performed the evaluation described in our paper in a private cluster of 11 machines as described below. We will provide remote access to the cluster via ssh. Please reach out to the authors via HotCRP for access instructions and credentials. We also provide an archive with the logs from running the full set of experiments reported in the paper (`logs.tar.xz` in the `jitserver-benchmarks` repository).

Our cluster consists of 8 machines of type A (see below), and 3 additional less powerful machines of type B that were used to run the workload generator (which could be run on the same hardware as the other components without affecting the results). The description of the performance experiments and their setup (e.g. what components run on what machines) can be found in the paper.

Type A machines hardware details:

- CPU: 16-core (32 hyperthreads) AMD EPYC 7302P
- memory: 256 GB DDR4 2666 MHz
- motherboard: TYAN S8021
- storage: 2x Samsung NVMe SSD SM981/PM981 in RAID0
- NIC: Intel 10G X550T

Type B machines hardware details:
- CPU: 14-core (28 hyperthreads) Intel Xeon E5-2680
- memory: 128 GB DDR4 2400 MHz
- motherboard: ASUS X99-E-10G WS
- storage: Samsung SSD 850 EVO
- NIC: Intel 10G X550T

The machines are connected with a 10 Gbit/s Ethernet network with RTT latency of ~45 microseconds. We used the `netem` module in the Linux kernel to emulate additional latency in the range of up to ~8 milliseconds in some of the experiments. We also used a 100 Gbit/s Infiniband network (with Mellanox MT27800 ConnectX-5 NICs) for an additional latency data point of ~15 microseconds (which is not very important for the results, thus the use of Infiniband is not necessary).

Relevant software versions (including benchmarks) are as follows: Ubuntu 18.04.2; Linux kernel 4.19.49; Docker 19.03.6; JDK 1.8.0_292; JMeter 3.3; OpenLiberty 19.0; AcmeAir 2.0; MongoDB 4.4.6; DayTrader EE7; DB2 11.5; PetClinic 2.3.0; Spring Boot 2.3.3;