A. Artifact Appendix

A.1 Abstract

This section details the steps to reproduce the experimental results presented in STARNUMA. *The process consists of tracing, memory access profiling, and timing model simulation. We also provide scripts for post-processing and plotting the data. A high-level description of the evaluation methodology is also described in Section IV of the main manuscript.*

A.2 Artifact check-list (meta-information)

- **Program:** Pintool, modified champsim, custom c++ programs and python scripts.
- **Compilation:** g++, cMake
- **Model:** Memory access profiling model, Timing simulation model
- **Run-time environment:** Linux (tested on Ubuntu 20 and 22)
- **Hardware:** Server environment and abundant storage space required. Hardware supporting 64+ threads preferred.
- **Run-time state:** Could run in background. Using screen session or the like recommended
- **Output:** Simulation output and ultimately bar plots.
- **How much disk space required (approximately)?:** Instruction and memory traces could take up to hundreds of GB per application. Traces for all the benchmarks could take **15TB** or more.
- **How much time is needed to prepare workflow (approximately)?:** Setting up should take 1-2 hours.
- **How much time is needed to complete experiments (approximately)?:**

  Generating traces takes between 6˜19 hours. Memory access profiling takes between 1˜8 hours. Timing simulation takes between 4˜30 hours.

  Worst case for completing the full experimental pipeline of a single benchmark took 40 hours with sufficient HW (appendix A.3.2).
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Will add after the artifact evaluation is complete.

A.3 Description

A.3.1 How to access

Github repository with relevant components is available at the following link:

`https://github.com/albertycho/StarNUMA_Artifact/`

In linux environment, you can clone the repo with

```
git clone git@github.com:albertycho/StarNUMA_Artifact.git
```

To accommodate for any changes suggested during the artifact evaluation process, we delay adding the DOI link to once the process is complete.

A.3.2 Hardware dependencies

- x86 (to support Intel Pintool)
- (preferably) capability to run 64 or more threads simultaneously.
- 50GB˜100GB memory (preferably more) for instruction and memory tracing.
- The traces from STARNUMA consume extensive storage space. Traces for some applications could take up to 700GB. Traces for all benchmarks could add up to **15TB** or more.

A.3.3 Software dependencies

- Recent linux distribution (we used Ubuntu20 and 22)
- Python3 (with packages pandas, matplotlib, etc)
- gcc/g++ (we used 11.4)
- omp library support

A.4 Installation

For all installation and execution, set environment variable STARNUMA_ARTIFACT_PATH to the cloned repo with **export STARNUMA_ARTIFACT_PATH=your_path**.

Tracer

- Intel Pintool: We include Pin 3.30 inside the repository. If this version is not compatible with your environment, binary and instructions can be found on the Intel Pintool website.
- Set environment variable **PIN_ROOT** to the pin directory (`export PIN_ROOT=path_to/StarNUMA_Artifacts/pin-3.30-98830-g1d7b601b3-gcc-linux`).
- Inside **StarNUMA_Artifacts/tracer/pin_tracer** directory, compile with makefile.

  (`make obj-intel64/combined_tracer.so`)

Benchmarks Benchmarks are under the **benchmarks** directory.

- **Graph Benchmark Suite** - Can be found in **gapbs** directory.
  - Calling **make** will generate the binary executables for the benchmarks.
  - Generating the graph: You can call

    `make bench-graphs`

    or call

    ```
    mkdir benchmark/graphs
    ./converter -g28 -k32 -b benchmark/graphs/kron.sg
    ```

  For SSSP, you need to generate another graph kron.wsg by replacing kron.sg with kron.wsg in the above command (included in make bench-graphs call). Building the graph requires 200GB of memory.

  * While we used g28 k32 graph (where g is log2 of number of vertices and k is degree), generating a large graph could cause a crash depending on your environment. In such case, using a smaller graph (by changing KRON_ARGS in bench.mk or in the cmdline) should be adequate to reproduce the high level performance trend.
- **FMI/POA** - We use fmi and poa from Genomicsbench. Due to size constraints, we do not include the whole genomicbench in our package. Instead, we provide the modified **fmi.cpp** and **msa_spoa_omp.cpp** (for ROI flags) that one can replace into the full repo to compile. The rest of the files for Genomicsbench can be found on `https://github.com/arun-sub/genomicsbench`, which you can get via

  ```
  git clone --recursive https://github.com/arun-sub/genomicsbench.git
  wget https://genomicsbench.eecs.umich.edu/input-datasets.tar.gz
  ```

  Once the modified source files are replaced in fmi and poa directories, benchmakrs could be compiled with **make** command. Please refer to README in benchmarks/genomicsbench.
- **TPCC** - we use a version of available tpcc benchmark. Please refer to README.
- **MASSTREE** - We use masstree from Tailbench. Please refer to README.

Memory Access Profiling

Source code is under **mem_profile directory**. Compile the programs to generate page allocation mapping using

/StarNUMA_Artifacts/mem_profile/scripts_packaged/setup_package.py.

In the paper, we ran each benchmark for all migration limits and took the best performing limit for each benchmark and system. To save time and resources, in the reproduction effort we use no migration for baseline and max migration (256k 4KB pages) for StarNUMA, which respectively gave the best performance for most applications. From this simplification, CC and FMI speedup may differ from the paper.

StarNUMA_Artifacts/mem_profile/compile_all.py is provided to generate all required binaries and scripts in one call.

Timing Simulation

- DRAMSim3 (libdramsim3.so) must be compiled (Makefile included). Path to the directory must be added via
  `export LD_LIBRARY_PATH=/PATH_TO/DRAMsim3/:$LD_LIBRARY_PATH`
- we provide a script to build all the required configs (**build_all_bin.py**). Run it with:

```
python3 build_all_bin.py
```

The main evaluation uses 4C_16S for both StarNUMA and baseline, as StarNUMA without any CXL pool access is the same as baseline (baseline will not have any memory allocated to the pool at the page allocation mapping stage).

A.5 Experiment workflow

A.5.1 Tracing

Before tracing any benchmark, OMP_NUM_THREADS must be set to 64:

```
export OMP_NUM_THREADS=64
```

Python script to launch the tracer is provided for each benchmark in the **tracer** directory. Tracers will be launched inside StarNUMA_Artifact/tracer/TRACES/each_benchmark. You need to set a few environment variables prior to using the scripts - please refer to README in tracer directory. Descreption of the generated traces are also included in README.

The tracer will run either until a program's completion, or until any of the threads reaches 40 billion instructions.

A.5.2 Memory Profiling

After compiling the memory profilers for the three configuraitons described in appendix A.4, go to each benchmark's directory with memory traces, and call the **run_simulation.py** script from all three configurations compiled. For example:

```
python3 PATH_TO_REPO/mem_profile/
scripts_packaged/pp_512_256_ca_5/run_simulation.py
```

After completion, the process should result in **pagemaps_*** directory in the traces directory where the program was called, with memory allocation mapping for each phase in the directory.

Static Allocation (fig 9):
Static allocation processing requires a separate process. To generate page mapping in experiments for figure 9, you need compile the source files in **/mem_profile/static_placement** directory. Compile command is given in the directory. For static allocation, we don't use a python script but just run the generated **static_placement** binary in the trace directory.

A.5.3 Timing Simulation

We provide **setup_smarts_runs.py** script that leverages the provided **EX_DIR_HIER** directory to set up the run scripts.

Running the setup script will setup the run_smarts.py scripts for each benchmark in each configuration. We leave running each run_smarts.py script to you, as launching everything at once in a single script call may put too much burden on the server, depending on the server capability.

For reference, configurations used for each figures are included in README in timing_runscripts directory.

A.5.4 Collecting Results and Plotting

Scripts to collect experiment results are under the post_processing directory.

- **extract_smarts_stat_per_benchmark.py** : iterates through the simulations for all phases of a benchmark and computes the average stats.
- **collect_smart_stats_top.py** : this script coalesces output from extract_smarts_stat_per_benchmark.py.

Run extract_smarts_stat_per_benchmark.py for all benchmarks under all configurations. Script collect_smart_stats_top.py should be called in the configuration directory (for example, inside baseline_main). We provide script **post_process_2in1.py** to do this in one run, which you can run if you know all the runs have finished. In the script, you must update the path to the top directory (i.e. your EX_DIR_HIER) and path to the above two scripts.

In **plotting** directory, We give the data collecting script for each plot, to be called only after extract_smarts_stat_per_benchmark.py and collect_smart_stats_top.py complete for all benchmarks and configurations. (only works with the directory structure provided by EX_DIR_HIER) Before using any of the scripts, you have to set environment variable `STARNUMA_RUNS_TOP_DIR` to the top directory of your runs (i.e. your EX_DIR_HIER).

```
export STARNUMA_RUNS_TOP_DIR=YOUR_EX_DIR_HIER
```

Call each of the collect_data scripts inside the directory you want the plots to be generated (need separate directory for each plot). Each script collects and coalesces data for each figure.

Scripts for plotting are also under **plotting** directory. While each collect_data script should automatically generate ready to go input files for the plotting scripts, sample input data are provided in **sample_collected_results** directory for input format reference.

For plots in the main evaluation (Figure 8), you must run two scripts, plot_fig8_1.py and then plot_fig8_2.py where you have the collected the input files for figure 8.

The rest of the plots for the sensitivity studies each require one call to their respective plotting script.

A.6 Evaluation and expected results

Completing the full pipeline should reproduce Figures 8 to 11 presented in the main manuscript of StarNUMA.