ECE 721 Benchmarks and How to Run 721sim

Version 1.0 (created Jan. 13, 2022)

ECE 721: Advanced Microarchitecture
Prof. Rotenberg

- READ THIS ENTIRE DOCUMENT.

1. Benchmarks

In ECE 721, we use benchmarks from the SPEC CPU 2006 and SPEC CPU 2017 benchmark suites.[1] Learn more about these suites and each benchmark at the following links (for informational purposes only):

- SPEC CPU 2006: https://www.spec.org/cpu2006/, https://www.spec.org/cpu2006/CINT2006/, https://www.spec.org/cpu2006/CFP2006/
- SPEC CPU 2017: https://www.spec.org/cpu2017/ , https://www.spec.org/cpu2017/Docs/overview.html#benchmarks

NCSU has licenses for both suites.

1.1. 721sim's ISA

The SPEC suites come with source code (most are C/C++, some are Fortran) and input datasets. We (Rotenberg's group) compiled the source code to RISCV executables: 721sim's superscalar processor model implements the RISCV ISA.

1.2. Benchmark = benchmark+input

SPEC provides multiple input datasets for each benchmark. Thus, when we refer to "a given benchmark", we really mean a benchmark+input pair. For example, astar from SPEC CPU 2006 comes with two reference inputs, BigLakes and Rivers. Thus, we really have two different benchmarks that both use the astar executable: astar+BigLakes and astar+Rivers.

Continuing with the astar example, see below for how we named benchmarks:
473.astar_biglakes_ref
473.astar_rivers_ref

The "473.astar" is the name given by the SPEC organization. Apparently, every benchmark is preceded by a number that they use and it just happens to be 473 for astar.

The "biglakes" and "rivers" refer to the input datasets.

[1] For Project 4, we may also use microbenchmarks (small hand-crafted benchmarks for gaining insight, showcasing performance-enhancements, *etc.*) and other benchmarks as needed for specialized projects.

The "ref" indicates that "biglakes" and "rivers" are input datasets from the *reference inputs*, which commercial vendors are supposed to use for their result reporting (to compare different commercial CPUs). In addition to reference inputs, the SPEC suite includes *test inputs* (small inputs for testing correct functionality of the compiled benchmark executable) and *training inputs* (intended to be used by profile-guided optimizing compilers, *i.e.*, compilers that use runs to refine their compilation; good science and engineering dictates using different training datasets during profile-guided compilation and reference datasets when measuring performance). Note that we use only the reference inputs.

1.3. SimPoints

Each SPEC benchmark yields dynamic instruction streams that are trillions of instructions long or many trillions of instructions long. Some take days to execute natively on real hardware. This means it would take one or more weeks to just functionally simulate the benchmark, *i.e.*, to just simulate the functionality of dynamic instructions without modeling a pipeline. Thus, it could take months for 721sim to simulate the benchmark because it does detailed cycle-level modeling of a superscalar processor.

To be practical, we simulate and measure performance of only a small sample of the benchmark's dynamic instruction stream while trying to be representative of performance of the entire dynamic instruction stream. True statistical sampling uses many small samples that are randomly distributed over the entire dynamic instruction stream. "SimPoints" is a different strategy (more practical but it may not be statistically rigorous) based on frequency of basic blocks in the dynamic instruction stream [1][2]. The way it works is:

1. We run a full functional-only simulation of the benchmark and collect basic block frequency information to be used by the SimPoint tool. This could take a week for a given benchmark, but it only needs to be done once (unless you recompile the benchmark using different optimizations, then the benchmark has changed and the process must be repeated).
2. We feed the basic block frequency information to the SimPoint tool and ask it for 10 of the most representative 100-million instruction chunks in the dynamic instruction stream (the 10 and 100-million are typical examples). The tool considers every 100-million instruction chunk. It tells you which are the 10 most representative chunks and tells you their *weights*. The weight of a chunk is how important that chunk is in terms of its representative contribution to the entire dynamic instruction stream. The weights should sum close to one. If they do not sum close to one, more than 10 chunks are needed for more comprehensive representation of the full dynamic instruction stream. Conversely, the SimPoint tool indicates fewer than 10 chunks if their weights sum close to one without needing any additional chunks.

So the output of the SimPoint tool is a simple list of "SimPoints" and their weights. A "SimPoint" is a starting point in the dynamic instruction stream of the corresponding representative 100-million instruction chunk. The way that the SimPoint tool expresses the "SimPoint" is in terms of *which* 100-million (10^8) instruction chunk. For example, if the "SimPoint" is 252, the chunk starts at $252*10^8$ instructions – 25.2 billion instructions – into the dynamic instruction stream.

1.4. Checkpoints

One way to exploit SimPoints for faster simulation is to run functional-only simulation from the start of the benchmark until the SimPoint and then simulate 100-million instructions on the detailed

superscalar processor model. Unfortunately, even the fast-forwarding part can take a long time, especially for SimPoints that are very far into the dynamic instruction stream.

Instead of fast-forwarding, we use checkpoints. A checkpoint is a snapshot of the precise architectural register and memory state of the running benchmark at a specific point in the dynamic instruction stream. Once we know the SimPoints, we can generate their corresponding checkpoints by doing functional-only simulation of the benchmark and saving the precise architectural register state and memory state at each SimPoint. Checkpoints are quite large and are implemented as .gz compressed files.

721sim can run benchmarks from the start or from a checkpoint. It does the latter by loading the checkpoint to initialize the committed register state of the processor and the committed memory state of memory, after which it looks like we are starting from somewhere in the middle of the benchmark at the representative region of interest.

Going back to the astar example, the SimPoint tool gave us 9 SimPoints for the benchmark 473.astar_rivers_ref. We generated 9 corresponding checkpoints and named them as follows:

473.astar_rivers_ref.6476.0.04.gz
473.astar_rivers_ref.1612.0.07.gz
473.astar_rivers_ref.4348.0.07.gz
473.astar_rivers_ref.4295.0.24.gz
473.astar_rivers_ref.6929.0.07.gz
473.astar_rivers_ref.6123.0.17.gz
473.astar_rivers_ref.6353.0.05.gz
473.astar_rivers_ref.252.0.28.gz
473.astar_rivers_ref.3.0.00.gz

Each checkpoint has the benchmark name as its prefix: 473.astar_rivers_ref (which we explained in Section 1.2). The prefix is followed by the SimPoint number and its weight. For example, consider this checkpoint:
473.astar_rivers_ref.252.0.28.gz
This checkpoint is a snapshot just prior to SimPoint 252, which means it is a snapshot 25.2 billion instructions into the dynamic instruction stream. Its weight is 0.28, making it the highest-weighted SimPoint among the 9 SimPoints.

Notice that the sum of the weights is 0.99 and not 1. This is due to truncating the weights in the filename. For example, SimPoint 3 actually has a non-zero weight (0.00191319) but it needs more digits to be represented. If we fixed the number of *significant digits* in the filename weights, rather than the total number of digits, I suspect the sum would always be 1 even with rounding of the weights.

1.5. Aggregating a benchmark's SimPoints

The instructions-per-cycle (IPC) measurement of one SimPoint of a benchmark is interesting in and of itself. It does not represent IPC of the whole benchmark, however. You can project IPC of the whole benchmark by using the weighted harmonic mean of SimPoint IPCs:

$$\overline{IPC} = \frac{\sum_{i=1}^{N} w_i}{\sum_{i=1}^{N} w_i \cdot \left(\frac{1}{IPC_i}\right)}$$

N is the number of SimPoints for the benchmark.
w_i is the weight of SimPoint i.
IPC_i is the IPC of SimPoint i.

You can use weighted means for other measurements, too. Just make sure to use the correct mean. Recall from ECE 563, the correct mean for quantities (*e.g.*, time (s), cycles, CPI, energy (J)) is arithmetic mean, the correct mean for rates (*e.g.*, IPS (1/s), IPC (1/cycle), power (J/s)) is harmonic mean, and the correct mean for ratios is geometric mean.

2. How to Run 721sim

2.1. Grendel

The benchmark checkpoints are hosted in a NFS volume accessible from grendel.ece.ncsu.edu. Thus, to run the benchmarks on 721sim, log in to grendel.ece.ncsu.edu.[2]

2.2. What to do after logging into grendel.ece.ncsu.edu

Each time you login to grendel.ece.ncsu.edu:

```
source /mnt/designkits/spec_2006_2017/O2_fno_bbreorder/activate.bash
```

You only have to issue this command once when you first login. It equips you to use a tool for correctly setting up run directories.

2.3. Listing benchmarks and their checkpoints

Issue the following command to get a list of benchmarks:

```
atool-simenv list
```

Here's what you should see (truncated for space):

[2] I am exploring the use of NCSU's HPC system for Project 4.

```
atool[ericro@grendel28 build-run]$ atool-simenv list
All available app:
        000.helloworld (2 checkpoints available)
        400.perlbench_checkspam_ref (7 checkpoints available)
        400.perlbench_diffmail_ref (5 checkpoints available)
        400.perlbench_splitmail_ref (7 checkpoints available)
        401.bzip2_chicken_ref (9 checkpoints available)
        401.bzip2_combined_ref (8 checkpoints available)
        401.bzip2_liberty_ref (8 checkpoints available)
        401.bzip2_program_ref (9 checkpoints available)
        401.bzip2_source_ref (8 checkpoints available)
        401.bzip2_text_ref (8 checkpoints available)
        403.gcc_ref (9 checkpoints available)
        410.bwaves_ref (10 checkpoints available)
        429.mcf_ref (8 checkpoints available)
        434.zeusmp_ref (9 checkpoints available)
```

Issue the following command to get a list of checkpoints for a given benchmark:
`atool-simenv list [benchmark name]`
For example, to get a list of checkpoints for the benchmark 400.perlbench_checkspam_ref:
`atool-simenv list 400.perlbench_checkspam_ref`
Here's what you should see:

```
atool[ericro@grendel28 build-run]$ atool-simenv list 400.perlbench_checkspam_ref
Available checkpoints for app 400.perlbench_checkspam_ref:
   400.perlbench_checkspam_ref.11032.0.18.gz
   400.perlbench_checkspam_ref.4898.0.03.gz
   400.perlbench_checkspam_ref.7944.0.17.gz
   400.perlbench_checkspam_ref.3912.0.13.gz
   400.perlbench_checkspam_ref.1877.0.24.gz
   400.perlbench_checkspam_ref.59.0.00.gz
   400.perlbench_checkspam_ref.3762.0.25.gz
```

2.4. Required workflow

1. **Dedicated run directory for each checkpoint:** The required workflow is to set up a dedicated run directory for each checkpoint that you want to run. Section 2.5 explains how to set up a run directory and launch a 721sim job from within that directory.
2. **One-job-at-a-time per run directory:** For the safest simulation possible, you must not launch more than one 721sim job at a time from within the same run directory. If you want to launch multiple 721sim jobs at the same time (either on the same or different machines), these jobs must be launched from within different run directories (one job per run directory). If we are talking about multiple jobs, each using a different checkpoint, then we are in any case ok because workflow requirement #1 says that there should be a dedicated run directory per checkpoint. *But, if you want to run multiple jobs at the same time using the same checkpoint (e.g., varying the 721sim superscalar processor configuration while keeping the checkpoint fixed), then you must have multiple run directories each set up with the same checkpoint.*
3. **Naming convention for run directories:** A run directory should be named by the full checkpoint name plus an instance number (starting at 1) appended at the end. The instance number distinguishes multiple run directories with the same checkpoint, if you want to launch multiple jobs at the same time using the same checkpoint.

Example run directories:
400.perlbench_checkspam_ref.3762.0.25.gz-1/
473.astar_rivers_ref.252.0.28.gz-1/
473.astar_rivers_ref.252.0.28.gz-2/
473.astar_rivers_ref.252.0.28.gz-3/

The example shows four run directories. The first is set up for running the checkpoint 400.perlbench_checkspam_ref.3762.0.25.gz. The other three run directories are all set up for running the checkpoint 473.astar_rivers_ref.252.0.28.gz. The only reason for having three run directories set up the same way, is for launching multiple jobs at the same time with the same checkpoint.

So, in the example, the user can safely launch four 721sim jobs at the same time: one 721sim job using the 400.perlbench_checkspam_ref.3762.0.25.gz checkpoint and three 721sim jobs using the 473.astar_rivers_ref.252.0.28.gz checkpoint.

It should be noted that if you submit too many 721sim jobs at the same time on the same machine, all jobs might run (much) slower than if you ran them one at a time. In short, if you want to exploit the parallelism of the host machine (multiple cores and multiple threads per core), make sure you account for how many cores the machine has and how many other users are running compute and/or memory intensive jobs on that machine.

2.5. Setting up a run directory and launching a 721sim job

Suppose we want to set up a run directory for checkpoint 400.perlbench_checkspam_ref.3762.0.25.gz of the benchmark 400.perlbench_checkspam_ref:

1. `mkdir 400.perlbench_checkspam_ref.3762.0.25.gz-1`
2. `cd 400.perlbench_checkspam_ref.3762.0.25.gz-1`
3. Symbolically link to the proxy kernel:
   `ln -s /mnt/designkits/spec_2006_2017/O2_fno_bbreorder/app_storage/pk .`
4. Ensure your 721sim executable is in this directory, either as a copy or a symbolic link to it. For Project 1, the 721sim executable will be available for download under the Project 1 section of Moodle. For Projects 2-4, the 721sim executable will be in your build directory and you can copy it or symbolically link to it (the latter is more efficient).
5. Generate a Makefile that equips you for launching 721sim jobs with the desired checkpoint:
   `atool-simenv mkgen 400.perlbench_checkspam_ref --checkpoint 400.perlbench_checkspam_ref.3762.0.25.gz`
6. Launch a 721sim job by issuing this command:
   `make cleanrun SIM_FLAGS_EXTRA='[`*`extra simulator flags`*`]'`
   For example:
   `make cleanrun SIM_FLAGS_EXTRA='-e100000000 --disambig=0,1,0 --perf=0,0,0,0 --fq=64 --cp=32 --al=256 --lsq=128 --iq=64 --iqnp=4 --fw=4 --dw=4 --iw=8 --rw=4'`
   The above extra simulator flags configure 721sim to simulate for 100,000,000 instructions, and the superscalar processor is configured with speculative memory disambiguation, real branch prediction and instruction/data caches, a 256-entry Active List, a 128-entry LQ and 128-entry SQ, a 64-entry IQ, fetch/dispatch/retire widths of 4, and issue width of 8.
   The Makefile that was generated in step 5, above, ensures that 721sim uses the specified checkpoint.

You may repeat step 6 as many times as you like with different extra simulator flags specified using SIM_FLAGS_EXTRA. You can manually repeat step 6 or write a Python script to do so.

Repeat steps 1-6 for setting up additional run directories and launching jobs within them. When you do so, continue to follow the naming convention for run directories, which was explained in Section 2.4.

When you launch a 721sim job using "make cleanrun", various things are printed to the console. Some of these print-outs are from the Makefile and the atool-simenv commands that the Makefile issues. Some of these print-outs are from the 721sim simulator. And, some of these print-outs are from the simulated benchmark. The perl example, above, has a busy console initially, because there are many benchmark print-outs that are replayed when loading the checkpoint. If you want to see more of the print-outs from the Makefile and simulator, do steps 1-6 for a checkpoint from a different benchmark (*e.g.*, checkpoint 473.astar_rivers_ref.252.0.28.gz from benchmark 473.astar_rivers_ref).

2.6. Output from 721sim

The final output from 721sim is in a file named "stats.[*timestamp*]". The output file is timestamped so that every output gets a unique name. The output file shows the superscalar processor configuration and some measurements, including the final IPC (ipc_rate) for this processor configuration and checkpoint.

If the benchmark is supposed to have generated any output files from the beginning of the benchmark to the end of the simulated SimPoint, they are isolated under the simenv/ subdirectory that the Makefile creates for you underneath the covers. Note that the reason for the "one-job-at-a-time per run directory" workflow requirement (discussed in Section 2.4), is to prevent output file conflicts among multiple simulated benchmark instances (which could potentially modify behavior and corrupt simulation results). Also note that "make cleanrun" resets the simenv/ subdirectory (it deletes it and regenerates it) so that each simulation starts with a clean initial file system for the benchmark (how files would be before running the benchmark). When 721sim loads a checkpoint, it not only restores architectural register and memory state, it also replays all file I/O up to the SimPoint.

3. References

[1] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. Proceedings of the *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, October 2002.

[2] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. Proceedings of the *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.