Homework3 Report

Team-25
Husan-An Weng Lin (A20450355)
Kevin Tchouate Mouofo (A20454613)
Ji Tin Justin Li (A20423037)

Contribute:

- Writing part of MyDiskBench: Hsuan-An
- Reading part of MyDiskBench: Justin
- IOzone experiments: Kevin
- Cloud machine(Chameleon) Configuration: Kevin
- Result graphs making: Kevin

The Design of MyDiskBench:

1. Overall

We simply separate the benchmark design into two parts: writing and reading. Two of us take care of each part. The main file will combine the two parts by calling the writing function and reading function.

2. Writing Files (Hsuan-An)

2.1. Design detail: For the I/O, I'm using the fstream library in C++ which is very easy to use. However, it turns out that it might not disable the buffer thoroughly in some situations due to the fact that it is a high-level interface so it might not have full control of the buffer. In this case, I think the low-level C interface such as fopen, fclose, fread, fwrite will be better to use. For threading,I'm using the pthread library which I think is more compatible with all platforms and I think pthread just does its job. I didn't observe any big performance problems. For timing, I use chrono library to calculate the time and I think it does its job pretty well too.

2.2. Possible Improvements and Extensions: Just like what I mentioned in the previous part, I think for I/O operation, using a low-level C interface will be more ideal. Due to the time constraint, I didn't organize the output data well. I believe that it should be collected in a folder or auto delete after running the tests.

3. Reading Files (Justin)

3.1. ***Design detail:*** There are 4 access functions in readFile.cpp that are intended to be externally called: ***workload_init(), workload_clean(), rs_rr_test(), and iops_rr_test()***. workload_init() calls to a bash script that creates necessary data files in the filesystem that are ready to be read. workload_clean() cleans up the created data folder when called. rs_rr_test() runs the test across the created workload files (10G), and creates 2 csv output files with the calculated throughput results. iops_rr_test() uses the 1G workload files and creates 1 csv output file with the calculated ops/s results.
Used libraries: The Disk I/O is accessed using posix system calls(open, read and lseek), which allows direct IO with disk using O_DIRECT flag. Threading is done using the std::thread library available in c++. Timing is done with std::chrono.

3.2. Possible Improvements and Extensions: The current threading mechanism frankly is quite a mess. The number of threads are hardcoded into their respective function calls, which is very inflexible. It is possible to be improved using dynamic thread creation by calling the thread constructor, or using a thread pool library.

4. Overall Review

Unfortunately due to time constraints, we are unable to mimic the command line interface of iozone in our program.

IOZONE benchmark:

1. Overview

Originally made by William Norcott, Iozone filesystem benchmark is a powerful tool used to generate and measure a variety of file operations. We have used this benchmark tool to measure the I/O file performance of our instance on Chameleon cloud. The tests were focused on sequential read, sequential writing, random read, and random write, for different numbers of thread, and record size.

2. Sequential I/O

Iozone has numerous commands to do I/O file system benchmarks. To do the sequential read and write, we have used the following command:

-T -I -i 0 -i 1 -s $FileSize -t $NumberOfThreads -r $RecordSize.

Let's take a look at the differents configurations:

- We have used -T to use our POSIX pthreads as we were running on Linux 18.04. Afterwards -t was used to give the number of threads we wanted to use.
- We have used -I to tell our Chameleon instance to use DIRECT I/O for all file operations in order to bypass the buffer cache and go directly to our disk.
- -s was used to give the size of the file we wanted to use to carry the test, and we have used -r to give the size of our record.
- We have used -i 0 to write on the test file and -i 1 to read our test file.

  N.B: We can’t carry the read operation without a previous writing operation.

The result of this test was in KB/s and we converted it in MB/s.

3. Random I/O

In this part of the assignment, we had to do two tests:

- Random read/write and throughput result: we have used the same command as for the sequential I/O test except that we have replaced -i 1 by -i 2 to do the random write and read. We have also converted the throughput result in MB/s.

- Random read/write and OPS/s result: here we have used the following the command line: -T -I -i 0 -i 2 -s $FileSize -t $NumberOfThreads -r 4k -O.
  - -O was used to indicate that we were interested in the number of operations per second.

  N.B: We have used 1GB file as we were running on a haswell instance. We could not run this tests multiple time because each of them took a lot of time

4. Code

We have created 3 bash scripts source code for Iozone benchmark. Iozone_test.sh is a code dedicated to the professor. This script asks the user to enter some parameters and uses them to give the result of one configuration.

IOPS.sh is the bash script we used to obtain the results for random read/write in OPS/s. Throughput.sh was used to obtain the results of all configurations in KB/s.

Comparison of the two benchmarks:

Results analysis and Discussion:

For the writing part, MyDiskBench doesn't share the exact same trend as the IOZone. We think it is probably because we are still using buffers in some situations so the throughput rises obviously when using more threads. And compared to the sequential access and random access, both MyDiskBench and IOZone shows that sequential access is more efficient which I think is tallying with the theoretical trend. For the IOPS tests, we are getting way much faster than the theoretical value especially in MyDiskBench. I think it may because of the buffer or we still need to improve the way we implement random access.

For sequential read and random read, the throughput does not scale well with the increasing number of threads. The plots from iozone and MyDiskBench both share similar trends, where the throughput increases slightly at 2 threads, then decreases and flattens as more threads are introduced. This is to be expected, as there is only one single disk device. From the results itself we can infer that 2 threads are sufficient to fully saturate the single disk device. More than 2 threads will only introduce higher concurrency control overhead and thus decreases overall performance.
Comparing between sequential and random access on the disk, it is clear that sequential access yields much better performance compared to random access. Looking into the node details of our registered machine, we can see that the physical mounted drive is a seagate HDD (model: ST9250610NS). HDD has low random access performance since the disk head and platter needs to physically move to the data location to perform operations to the data.

Node configuration details:
https://www.chameleoncloud.org/hardware/node/sites/uc/clusters/chameleon/nodes/4b8ef8b4-ef7d-40ae-add6-ab5f72d73cd4/
Specification for seagate hdd, model ST9250610NS:
https://www.disctech.com/Seagate-Barracuda-ES-ST3250620NS-SATA-Hard-Disk-Drive

-end-