Group 13 Performance Evaluation Paper
Dr. Ngo
CSC 496
Francis Convery, Brandon Fonticoba, Colin Lesser, Colin Pigeon

For the course project, we chose the Performance Evaluation option. In this option, we are provided a paper written by IBM that details nine different benchmarks that a group of employees used to test Docker and KVM. Our task was to choose four of the nine benchmarks and attempt to perform them on a CloudLab profile that either runs Docker or KVM on it. We chose to create a CloudLab profile with Docker already running on it. We created a GitHub to have our CloudLab profile to run off of. In this GitHub, we had a docker branch that when instantiated automatically downloads and installs docker on the CloudLab profile. With Docker installed the next step was to run the benchmarks. The four benchmarks we chose to run are Linpack, Stream, Redis, and Randomaccess. The following paragraphs in this paper break down our progress on testing each benchmark. Along with that, it explains what each benchmark is, what the benchmark tests for, and how to test the benchmark.

In order to start testing the benchmark, we needed to create our CloudLab profile. When creating our CloudLab experiment we based it off of a GitHub we created. Our GitHub had two branches within it, the first branch was the master branch and the second branch was the Docker branch. Our master branch has our objectives and technical papers on it while our Docker branch has all of our work on it. Our Docker branch was set up so that when instantiated it will automatically download and install Docker on the CloudLab profile. Getting the CloudLab profile to accept our GitHub repository, and instantiate the Docker branch was a success and easy for our group. We had no problems with this part. Even though when we were sshd into our CloudLab profile we couldn't access our files on our GitHub we were able to solve this problem easily. We would clone our repository. This way our repository was on our profile and we had access to all the files we needed to run our benchmarks. The setup to run the benchmarks was simple and resulted in a single problem for our group that we were able to solve.

The Linpack benchmark is a test problem used to rate the performance of a system on a simple linear algebra problem. The equation is A*X=B. Matrix A in our case is said to be a 200x200 array. We used C code to run the actual benchmark and used the routines DGEA and DGSL. The former performs partial pivoting, in other words, row permutations only. The latter uses that decomposition to solve the given system of linear equations. To put it simply in layman's terms, his is essentially a measure of a system's floating-point computing power. It can be said that this benchmark is a simplification of the problem since no single equation alone can reflect the overall effectiveness of a computer.

For the Linpack benchmark, it seems to have reached what I expected in my research. Of course, I needed to convert from KFLOPS to GFLOPS in order to compare accordingly. Luckily, the process to run the Linpack benchmark was automated, I only had to build the Docker image and run it. I did not run into any issues automating the process, which allowed for a streamlined experience running the benchmark. Over the course of researching this benchmark, it has been favored due to its ability to scale. The biggest roadblock that I found that we ran into was making sense of what we expected to see in our benchmarks. Another issue that was encountered was that when the step came to SSH into the CloudLab server, it did not recognize Docker commands. After many trials and errors, it was recognized that the wrong branch was being instantiated from the GitHub repository that was shared among the group.

I mentioned earlier that there are two main aspects to the Linpack benchmark, DGEFA, and DGSL. The former standing for, "Double precision GEneral matrix FActor." The latter standing for

“Double precision GEneral matrix SoLve.” It is expected that DGEFA has the most time spent on it, in most cases around 77-78%. It then moves onto DGESL to find the solution.

In my experience trying to run this benchmark I did not run into too many issues, in fact, it was quite a streamlined process, the biggest issue I ran into was trying to visualize and understand what exactly I was trying to test. I initially built a CloudLab profile using the GitHub repository to create a head and two worker nodes. Next, I SSH into the head node. Once that was completed, I cloned another GitHub repository to get the C code to run the test. I built a Docker image in the Linpack directory that I cloned. From there I ran the Docker image, and the benchmark then ran. In my experience, the code ran, equated the results in KFLOPS (kilo floating-point operations per second). From my research, it is typically given in GFLOPS (giga floating-point operations per second).

The next benchmark that we chose to test on our CloudLab profile was Redis. Redis is an open-source, in-memory data structure store, used as a database, cache, and message broker. What makes Redis special is that it’s capable of storing large values, has its own hashing mechanism and it caches data at a rapid pace creating very little latency. The Redis benchmark on the paper is tested to see the latency of the server with an increasing number of parallel clients but a constant amount of requests and bytes per payload. In order to do this, the benchmark sends a set amount of commands (around 20) to the server and records the amount of time it takes for the server to complete the commands. Each command is completed individually and completed one after another. To test the benchmark on our CloudLab profile I did the following; After sshing into the head node of our profile I created a container with Ubuntu running within it. Once this container is created, I updated and installed the apt-get command to make sure that it is capable of installing the server with no errors. Once the apt-get command is finished updating and upgrading I installed the Redis Server using that command. Following that, I would run the server within the container using the command Redis-server. This creates my own Redis Server within the container and takes over the container by constantly running in that node. Once the server is up and running it takes up the whole head node so I created a new terminal and sshed back into the head node. I reentered the container that is running our Redis Server within it and confirmed that it was running properly by using the command redis-cli ping. Once I received the expected response I would perform the benchmark with the command Redis-benchmark. If using this command with no flags or restrictions it will run a set benchmark test using 100,000 requests on 50 parallel clients with 3 bytes per payload. This benchmark takes a long time and isn’t what we are looking to test. In order to test the benchmark similar to the paper, I used the flags -n, -c, and -d which restrict the requests, parallel clients, and bytes per payload. I tested the benchmark with a constant of 100 requests and 15 bytes per payload. I started testing with zero parallel clients and increased them by 10 every test up to 120 similar to how the paper tested it. I recorded the time it took to complete all of the requests.

For deliverable two, I believe that the project was successful in testing and performing this benchmark. When done manually there are little to no problems and each command works properly and the server runs without a problem. When trying to automate the process the server tends to be created in a temporary image or a hidden container and it is a struggle to access the image or container and run the tests. Sometimes, when I automate the process and the server successfully gets set up the profile struggles to realize if the server is set up or where and will not understand the Redis commands. Since it doesn’t understand the commands it asks you to download the server manually ruining the automation of the process. I will continue to work on automating the entire process but I believe that it will not be possible with Redis due to the previous issues.

For deliverable three, I believe that the Redis benchmark was a success. Even if it was only manually, we were able to run the benchmark and perform the required tests at the given speeds. Unfortunately, we were still unable to automate the process of installing Redis and running the benchmark. The biggest problem we encountered here was when you built and ran the dockerfile the container that it created would not show up when using the command docker ps. Another problem that we

were encountering is when we built and ran the dockerfile with different commands in it it would create the container and run Redis in it automatically. When you entered this container and tried to run the benchmark or any Redis commands the container did not know what the commands were and would ask you to install or download additional utilities. This defeats the purpose of automating the process so I considered this a failure. In order to get the benchmark to run manually, you just have to create a container and run four to five commands to get Redis installed and running. The process is very simple and not complicated. This is the same when testing the benchmark. All you have to do to modify the benchmark testing is to edit one of the flags you enter with the benchmark command. In doing so the benchmark will run with the newly entered number. In the end, I believe that we were able to successfully run the benchmark up to the paper's and our own expectations.

For our next benchmark, we chose the Stream Memory Bandwidth Benchmark. Configuring this benchmark proved to be a challenging process. The first crucial step of making the benchmark run successfully was to fully understand how Stream worked. The paper resource described the benchmark as a simple test to measure sustainable memory bandwidth by performing different operations on different vectors. The test ran on four different main vector kernels: Copy, Scale, Add, and Triad. Memory bandwidth is essentially the rate at which data can be read from or stored into memory by a processor. Memory bandwidth is usually measured in megabytes per second. So, the test will work to show average time, minimum time, maximum time, and the best rate in MB/s for data being read and stored in memory on those four different vectors.

After understanding the Stream benchmark, the next step was to comprehend how to configure the benchmark to run in Docker. The GitHub that was provided by the paper resource proved to be a great foundation to build an understanding of how the benchmark and Docker can work together. After sifting through files and doing research, our team was able to successfully build a docker image with the stream benchmark files configured, and then we were able to run the created Image which ran the benchmark 20 times and collected the data results in a separate log file. The data collected clearly indicated which vector kernel was being measured and presented the best rate in MB/s, the average time processed, the minimum time, and the maximum time.

The main driving force behind the Stream benchmark is the 'stream.c' file. This code is able to test and execute the benchmark using the four-vector kernels. In addition to the stream.c file, our team created a custom Dockerfile that was able to configure a docker image using the stream.c file coupled with docker commands to run the image. The final files created were some simple bash scripts to be able to automate the process and run the dockerfile as well as the benchmark with one command, and then collect the data.

The stream benchmark test is important because CPUs are becoming faster than computer memory systems. This is notable because this means that more and more programs are going to be limited through their performance by the memory bandwidth of the system instead of the physical hardware component CPU. Some systems are running operations on their kernels at only a 4-5% performance rate meaning that 95-96% performance is used staying idle. The Stream benchmark works with datasets that are much larger than cache on the system so that when the test is run the results can be scaled easily to help understand the performance of memory bandwidth on your system.

For Deliverable 3, our team feels that the Stream benchmark meets all the requirements. The expected outcome for the stream benchmark was to measure the memory bandwidth on four different kernel vectors in MB/s. Our team was able to configure scripts and a Dockerfile to run this benchmark 20 times with each of the four-vector kernels measured out in MB/s. When we first started out, it was a difficult task to be able to run the stream benchmark with the expected outcome. Building our scripts off of the provided GitHub was helpful, but we needed to debug a few errors here and there to be able to run the stream benchmark. As soon as we were able to run our stream benchmark locally one time, we then tackled the task of running the benchmark through Cloud Lab. After more research, we were able to run

the benchmark on Cloud Lab 20 times with all the expected outputs. Our team also configured a docker.log file to collect all the results in one convenient location. The last step was to automate this process. Our team wrapped the Dockerfile and the docker commands with a simple bash script that helped to automate the implementation. Anyone would be able to clone our GitHub and run the Stream benchmark 20 times with just one command. Our team views the Stream Benchmark as a successful implementation.

The fourth and final benchmark that we chose was the Random Access Memory benchmark. According to IBM's 2015 Research Report, "The benchmark initializes a large section of memory as its working set, that is orders of magnitude larger than the reach of the caches or the TLB. Random 8-byte words in this memory section are read, modified (through a simple XOR operation), and written back." In simple terms, our goal was to time how long it took the one socket and two-socket system to change an address's attached value.

Our group attempted to follow the structure of their test in CloudLab, but some aspects that were unrelated to Docker were ultimately commented out or removed. The doit.sh is what initiates the long string of automated commands needed to run this benchmark. This file calls docker.sh twenty different times with the combination of two different for loops. These loops are dedicated to simulating one socket and two-socket systems that require their own unique Docker file. Within docker.sh, the image and container are built based on the correct Docker file and gups.exe is created. Gups.exe is an executable version of gups.c and is named after the unit of measurement used for this test which is, giga-updates per second. Within this image, gups.exe is launched to run the necessary assessments such as CPU time, real-time, table size, amount of updates, and the number of errors that occurred. The end results are supposed to be recorded in docker.log which is located in a folder that sits inside the directory where the test was originally initiated.

This benchmark continues to be a challenge. After some minor adjustments to the code to fit our CloudLab environment, our group can successfully build the image through an automated process. However, memory allocation is an obstacle that we have yet to traverse over and this is currently impeding us from launching a container. The exact errors we receive state that local allocation and set mempolicy are not permitted. We believe that a hidden Docker security setting could be intervening or Docker's memory allocation process needs to be modified to accommodate our needs. To combat this, we are researching if a certain flag can enable a numactl command to bypass any interruptions. Another option we are considering is altering the config files or our profile.py so that enough space is made available to properly construct the container.

We aimed to have all the benchmarks running manually without any existing errors by the due date. Because of the previously mentioned challenges that we have yet to overcome, our original goal was not completely met. In hindsight, our group may have struggled to run the Random Access Benchmark, because we were trying to learn the intended structure of the author's process while simultaneously trying to automate it. We did not have our goal as a forethought during the development process for this aspect of our project. It was assumed that manually running the code for the benchmarks would be an easier way to troubleshoot for the other benchmarks, but Random Access was an exception in our minds. We were attempting to do too much at once and it may have contributed to our confusion. If we were to do this project over, we would readjust our focus and simplify our approach. Throughout my college career, I have noticed a pattern of multitasking ultimately hindering the quality of final drafts for my other classes. I think if we attempted to run and troubleshoot the code without automating it, we may have had an easier time pinpointing where our issue was coming from.