1. Testbed

Here is some information about the testbed we are providing. It is not possible for two reviewers to do evaluation at the same time on this testbed since our artifacts are kernel modules, and it is not possible to load two with the same name. Once a reviewer is done evaluating, they should terminate the kernel API remoting (explained later) or the next reviewer will not be able to use it, unless they forcibly unload and kill it.

Some experiments, mostly the latency prediction, when executed with a wrong configuration can cause the machine to fully hang, or cause kernel errors/deadlocks that can not be fixed without a reboot. This machine takes between 3-5 minutes to fully reboot. Depending on the error, the reboot can also get stuck. If the machine gets into a state where the reviewer can not fix it (which will likely happen), post a comment to us on hotcrp and we will reboot it through our BMC.

The machine already has our modified kernel installed, but the reviewers are welcome to compile it again and reboot the machine. Note that installing a new kernel requires rebooting, or module building will fail and reinstalling the Nvidia driver. The NVMes are at */dev/nvme0n1*, */dev/nvme1n1* and */dev/nvme2n1*. There is no data on them. Please do not set any of the filesystem or latency prediction experiments to the root drives (*/dev/sda*) as this can corrupt the filesystem (as we learned).

The users added for the reviewers will not require a password to run sudo to make things easier. Our home directory is mounted on NFS and has limited space. We will create a directory on */disk* and set the bash configuration to change to that directory on login.

The code in our repository has most, if not all, arguments and variables set to work on this machine, but we ask that the reviewers double check.

Loading the kernel API remoting system is done, as explained below, by running *sudo ./load.sh* at *src/kapi*. Killing this process with *ctrl+c* should unload and kill every process, but that does not happen on this machine. To kill the API remoting, run *sudo ./unload.sh*.

2. Installation

The experiments described in this document require some additions to the requirements of the initial basic test's requirements. For the initial test, we disabled the I/O latency predictor hooks from our kernel. Since this is part of the artifact evaluation, it requires pulling the most recent version of the repository, recompiling and installing it. To do this steps you need to:

1. *cd* into the linux-6.0 directory with the repository
2. Run *git pull –rebase*
3. Run *full_compilation.sh*, which will compile and install the kernel.
4. Reboot your machine. Since the kernel name did not change, no extra step to make this the default kernel booted by GRUB if you have already done it previously.

Reinstalling the kernel breaks the Nvidia drivers, so repeat the steps for installing it. Install the CUDA runtime (version 11.7) and an Nvidia driver for the GPU. We provide the link to download both in the *README*, but the reviewer is welcome to navigate Nvidia's website and manually download the CUDA *runfile*, which contains the runtime and driver. The driver installation from the CUDA runfil will probably fail. So we recommend executing the runfile and installing only the CUDA runtime. Then, download and execute the Nvidia driver only. The command *nvidia-smi* and *nvcc –version* **must** run successfully at this step. The latter can be fixed by add the CUDA installation path to your PATH (*/usr/local/cuda/bin* by default). For example, append the following to */.bashrc* or */.zshrc*: *export PATH=/usr/local/cuda/bin:$PATH* One of the workload also require cublabs which needs to be added to the path. Append the following to */.bashrc* or */.zshrc*:

```
export PATH=/usr/local/cuda-11.8/bin
${PATH:+:${PATH}}
export LD_LIBRARY_PATH=
/usr/local/cuda-11.8/lib64:
${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Reinstalling the kernel also makes every module that was compiled with the old kernel not be loaded. What this means is essentially that the kernel API remoting (the top left pane in all of our tmux scripts) needs to recompiled, so before executing the *load.sh*, you need to run *make*.

We require python3, the development headers and several libraries. To install the headers, run *sudo apt install python3-dev* We provide a script that creates a virtual environment to create necessary packages. To make sure you have venv installed by running *sudo apt install python3-venv* Go to the root of the LAKE directory and run *./create_venv*. This will install a venv in a directory called *lakevenv*. Activate it now with the command *source lakevenv/bin/activate*.

To reduce noise from cpu, we use a script that disables hyperthreading cpus and sets the cpus to performance mode. This script is provided at *LAKE/scripts/setup_machine.sh*. This step is optional, but recommended in order to collect data as close as possible to ours.

3. Reproducing Experiments

Every step for reproducing our experiments is made easier through tmux, to coordinate the moving pieces of our system. We provide a crash course below and are happy to help if the reviewer asks. Since our work relies heavily on kernel changes and kernel modules, most commands require sudo. If a command failed, make sure it ran with root permission. Some commands, if done in the wrong order can cause segmentation faults in the kernel or even hangs. In that case a reboot is necessary. We did not find any kernel hang on the version provided to artifact evaluation, but in case it happens, cycling the computer's power might be necessary; it's optimal if you have remote access to a PDU or BMC.

3.1 Crossover Experiments

Go to the root of LAKE and open the tmux session for these experiments by running *./xover_tmux.sh*. There will be three panes. The bottom left will print kernel messages, the top left will have the kernel API remoting system running and the right pane is where the experiments will be executed by hand. The bottom left pane might require your password. The top left pane will already have the commands require typed in.

All of our experiments **require** that the kernel API remoting system is running. In case it is not, experiments will fail.

For these experiments, the reviewer will run each workload separately. Output of the experiments will be written to the kernel log. Results will be compared visually through plotted graphs. Each workload has scripts to run it (by inserting and removing kernel modules) and a script that parses the kernel log and plot the graph. The script requires matplotlib, so make sure you have the provided python virtual environment installed and activated (§2). The plotting scripts are in the *ae_plot* subdirectory of every workload. The script that plots graphs is *plot.py*, for each workload the reviewer must manually call this script. Executing these scripts with python will output a graph inside the *ae_plot* directory. For most of the workloads both our results and the results gotten from running the experiment will be plotted to facilitate comparison.

All the experiments have a similar workflow: go into the workload directory, build it with *make*, run with the provided script (*sudo ./run.sh*), go in the *ae_plot* directory and run *python3 plot.py*. The pdf plotted will be in this directory.

3.1.1 LinnOs

Make sure the kernel API remoting is enabled on the top left pane. To run the latency prediction crossover experiment, to into *src/linnos*, type *make* and *sudo ./run.sh*. To plot the graph, cd into *ae_plot* and run *plot.py*. This is the only workload we do not plot our collected data and the generated data because there would be too many data in one graph. The other workloads plot both collected data into one graph. Instead, we provide our graph in pdf format in the *ae_plot* directory. Please compare the trends of the two graphs.

3.1.2 KML

To run the filesystem prefetching workload, do the same steps as above but at the *src/kernel-ml* directory.

3.1.3 MLLB

To run the load balancing workload, do the same steps as above but at the *src/load_balancing* directory.

3.1.4 kNN

To run the kNN workload, do the same steps as above but at the *src/knn* directory.

3.1.5 Kleio

Kleio requires tensorflow to run, which is installed in the virtual environment provided. It assumes that tensorflow is installed there so that our kernel api can import it. The path search if based on your python version, which is automatically detected. For example, with our python3.10, the path searched for is *lakevenv/lib/python3.10/site-packages/*.

If the reviewer face any problem with this workload, which could happen for different python versions, we will be glad to assist.

To run this workload, do the same steps as KML and MLLB: go to *src/kleio*, type *make* and *sudo ./run.sh*. Plotting script is in *ae_plot*.

3.2 eCryptfs

The accelerated eCryptfs evaluation is more complex than the previous. This experiment requires building three parts: our modified eCryptfs module, our GCM crypto module and building the file benchmark application.

Make sure the kernel API remoting is enabled on the top left pane. To start, go into *src/ecryptfs* and type *make*. This should build all three pieces.

We provide a script, explained below, that loads, execute experiments and unloads repeatedly for every data point. This script requires a directory to create the encrypted filesystem. On our testbed, this can be done by running:

```
sudo mkfs.ext4 /dev/nvme1n1p1
sudo mkdir -p /mnt/nvme1
sudo mount /dev/nvme1n1p1 /mnt/nvme1
sudo mkdir /mnt/nvme1/crypto
```

If the reviewer is interested in manually enabling eCryptfs and do its own testing, the crypto engine must be loaded first (*src/ecryptfs/crypto*). Then the eCryptfs module at *src/ecryptfs/ecryptfs*. The encrypted directory must be mounted with the extra flag *ecryptfs_cipher_mode=gcm*.

First, go into *benchmarks/ecryptfs*. The script to run the benchmark is *run.py*, but do not run it yet. This benchmark requires a directory to create encrypted directories and benchmark file operations. We provide instructions on doing this on our testbed below. We recommend a non-root drive formatted with ext4. After choosing a directory, the reviewer needs to find what drive this directory is in. The command *sudo lsblk* helps to find it. For example, if the directory chosen is at the user's home, the disk mounted as */home* is the one required to know the name. Open *run.py* and edit the *DRIVE* and *ROOT_DIR* variables. *DRIVE* should be just the suffix part of the drive, **without** the */dev/* prefix. *ROOT_DIR* should be the absolute path to the chosen directory that is mount within *DRIVE*. *DRIVE* is required so the script can correctly modify the readahead size, which is within */sys/block/DRIVE*. On our testbed and following the filesystem we created above, *ROOT_DIR* would be */mnt/nvme1/crypto* and *DRIVE* would be *nvme1n1*.

Run the script by executing *sudo -E python3 run.py*. The script will test the four comparison points by loading and unloading the correct modules and, for each, test different block sizes. These can be changed by modifying the dictionaries in *run.py* called *sizes* and *tests*.

If running this script returns an error similar to:

```
Unable to link the KEY_SPEC_USER_KEYRING into
  the KEY_SPEC_SESSION_KEYRING
```

a eCryptfs bug was hit per this bug report. This bug is related to the current executing environment. If the reviewer is running the experiments on tmux, we suggest detaching from the tmux session and running the script on the regular shell.

If the reviewer faces issues like the benchmark hanging (it does take a few minutes to complete), it can try to run less tests at a time. For example, the reviewer could comment all the *tests* variables in the script except the one it wants to run and collect its data. The data output by the script is in a readable csv format.

We do not provide automatic plotting of results, but we provide the plotting script and instructions. The reviewer shall copy the results from the script and replace the contents of *tput.csv* (which contains our collected data) with the data collected by the reviewer in the same format present in the csv. Running the *tput.py* script will plot *tput.pdf*.

3.3 Resource utilization

This experiment is similar to the previous and its script has the same requirements: providing a drive and a directory to mount.

To run and plot the graph for resource utilization, the reviewer goes to *benchmarks/utilization* and runs *make*. Then runs the script that does all tests with *sudo -E python3 run.py*. Note that this requires that the python virtual environment is activated or that the current python version has numpy and matplotlib. Note that the *-E* is important; without it the script will not find the required python packages if the option is not present. The plot will be at ecryptfs_util.pdf.

3.4 Contention

This experiment requires the LinnOS module to be compiled. To do so, go to *src/linnos* and type *make*. Similar to the experiment above, to reproduce the contention experiment, go to *benchmarks/contention*, run *make* and *sudo -E run.py*. This requires executing an extra script to plot: *python3 plot.py*. The plot will be at contention.pdf.

3.5 Latency Prediction

This is the most complicated workload to reproduce since it requires training and inference on traces running across three NVMes and is sensitive to many things. The first step is to generate the traces.

Go to *src/linnos/trace/tools* and run the following three scripts: *./gen_azure.sh*, *./gen_bing_i.sh* and *./gen_cosmos.sh*. For generating the "Mix+" workload later, there are these same scripts with an appended *3x*.

Now we will train. First, go into *src/linnos/io_replayer* and type *make* so that we can replay traces. Then, go to *src/linnos/training*. The drives that will be used to train are hardcoded in each of the train scripts (line 15) and correct for the testbed. For clarity, *3 /dev/nvme0n1-/dev/nvme1n1-/dev/nvme2n1* means we are going to use three NVMes, followed by their paths. With the correct drives set (be careful not to set any of them to devices with data because this process **will** corrupt it), we can now train. Train using the neural network from LinnOS by running the following script.

```
./train.sh ../trace_tools/azure/azure1.trace \
  ../trace_tools/bing_i/bing_i1.trace \
  ../trace_tools/cosmos/cosmos1.trace 90 mix
```

where the second to last number is the percentile of IOs considered late and the last is the name of this workload, used to save the weights. If you have already enabled LinnOS before, make sure it is not loaded into the kernel or the training script will fail with many IO errors. The generated weights will be at

```
src/linnos/kernel_hook/weights_header/mix
```

where the last directory is the name of the workload we set when training. Now these weights need to be loaded into our kernel hook, which will insert the weights into the kernel and enable/disable latency prediction. Go to *src/linnos/kernel_hook* and open the *main.c* file. Weights are chosen and added through includes. There can only be one weight per NVMe. Which NVMe a weight is for is the suffix of the weight. For the example above, the include needs to be, **with all other includes from *weights_header* commented out**:

```
#include "weights_header/mix/w_Trace_nvme0n1.h"
#include "weights_header/mix/w_Trace_nvme1n1.h"
#include "weights_header/mix/w_Trace_nvme2n1.h"
```

To use NNs with more layers, another part of the source code needs to be changed. The *long *weights[][8]* = {} variable contains how many weights we are inserting. The original NN uses four, NN+1 six and NN+2 eight. To use the regular layer we need these lines uncommented (they are by default):

```
//NN
{weight_0_T_nvme0n1, weight_1_T_nvme0n1,
 bias_0_nvme0n1, bias_1_nvme0n1 ,0,0,0,0},
{weight_0_T_nvme1n1, weight_1_T_nvme1n1,
 bias_0_nvme1n1, bias_1_nvme1n1 ,0,0,0,0},
{weight_0_T_nvme2n1, weight_1_T_nvme2n1,
 bias_0_nvme2n1, bias_1_nvme2n1 ,0,0,0,0},
```

This file is also where you set the three devices to be used. This is correct for our testbed.

```
  static const char *devices[] = {
"/dev/nvme0n1",
"/dev/nvme1n1",
"/dev/nvme2n1",
0
};
```

With these variables set, run *make*, which will build the kernel hook with the weights we just generated.

To enable I/O prediction in the kernel, there are helper scripts:

```
disable_linnos.sh
enable_linnos_cpu.sh
enable_linnos_cpu+1.sh
enable_linnos_cpu+2.sh
enable_linnos_gpu.sh
enable_linnos_gpu+1.sh
enable_linnos_gpu+2.sh
```

The first one disables whichever model is enabled at that moment. The others enable either the cpu or GPU version. The suffix +1 and +2 are for enabling the models with an extra NN layer. Be aware that **enabling a predictor with mismatching information (headers and weights array) in main.c will cause an error that requires a reboot**. For example, enabling a +2 model but only having the +1 headers and weights in *main.c* will cause many errors that may be unrecoverable. More details on how to run these below.

After enabling a version, we need to replay traces so that we can measure their latency. We recommend an extra tmux pane now, one for the kernel hook directory and one for the *io_replayer* directory.

The same string of drives that was set for training needs to be set in *run_3ssds.sh* which is in the *io_replayer* directory. This is already set correctly for the testbed. An example of the command sequence to run the cpu version would be:

```
sudo ./enable_linnos_cpu.sh
cd ../io_replayer
./run_3ssds.sh failover
  ../trace_tools/azure/azure1.trace \
  ../trace_tools/bing_i/bing_i1.trace \
  ../trace_tools/cosmos/cosmos1.trace \
python3 stats.py 3ssds_failover.data
```

The mode *failover* is the linnos mode that reissues I/Os that are classified as slow. We recommend running the replayer twice before running the stats.py script to warm up. Because the replayer is multi threaded and the order in which I/Os are handled in the kernel is stochastic, high variance can be observed. The kernel log may show some I/O errors, these are due to how I/Os are rejected/skipped in the kernel and can be ignored.

The stats.py python script will print statistics about the IOs. The main piece of information is the **Average read latency**. This value (or all the lines output by the python scripts) should be saved somewhere as the data will be overwritten the next time the replayer is executed. We also generate a cdf of read latencies at *3ssds_failover.data_cdf.pdf* if the reviewer is curious.

After the stats are generated, we need to disable linnos.

```
cd ../kernel_hook
sudo ./disable_linnos.sh
```

Disabling the hook for the GPU predictor will print information about the execution of the workload, including how many times the GPU was used and the histogram of batch sizes. These are not reported in the paper.

The experiment can now be repeated using a GPU by replacing *sudo ./enable_linnos_cpu.sh* with *sudo ./enable_linnos_gpu.sh* and repeating the steps.

Running the baseline, with no IO prediction is simpler. First, make sure no predictor is running by disabling it (running twice has no effect)

```
sudo ./disable_linnos.sh
cd ../io_replayer
./run_3ssds.sh baseline  \
  ../trace_tools/azure/azure2.trace \
  ../trace_tools/bing_i/bing_i2.trace \
  ../trace_tools/cosmos/cosmos2.trace
python3 stats.py 3ssds_baseline.data
```

Notice that two words have changed for the baseline, from *failover* to *baseline*.

3.5.1 Additional layers

For running the NNs with additional layers, the steps above must be repeated but with the correct scripts. For training, there are three scripts, with the suffix of how many extra layers:

```
train.sh
train+1.sh
train+2.sh
```

The same name (the last argument) can be used. A *+1* or *+2*, depending on how many extra layers there are, will be appended to the directory.

```
./train+1.sh ../trace_tools/azure/azure1.trace \
  ../trace_tools/bing_i/bing_i1.trace \
  ../trace_tools/cosmos/cosmos1.trace 90 mix
```

Then in main.c at kernel hook, comment all weight includes and add:

```
#include "weights_header/mix+1/w_Trace_nvme0n1.h"
#include "weights_header/mix+1/w_Trace_nvme1n1.h"
#include "weights_header/mix+1/w_Trace_nvme2n1.h"
```

Also change the weights array, comment out every line and uncomment:

```
// NN+1
{weight_0_T_nvme0n1, weight_2_T_nvme0n1,
  bias_0_nvme0n1, bias_2_nvme0n1,
  weight_1_T_nvme0n1, bias_1_nvme0n1 ,0,0},
{weight_0_T_nvme1n1, weight_2_T_nvme1n1,
  bias_0_nvme1n1, bias_2_nvme1n1,
  weight_1_T_nvme1n1, bias_1_nvme1n1 ,0,0},
{weight_0_T_nvme2n1, weight_2_T_nvme2n1,
  bias_0_nvme2n1, bias_2_nvme2n1,
  weight_1_T_nvme2n1, bias_1_nvme2n1 ,0,0},
```

Then to enable it, use either of:

```
enable_linnos_cpu+1.sh
enable_linnos_gpu+1.sh
```

The commands to replay, read statistics and disable the module are the same. For the two extra layers, same steps but replace +1 with +2, and in main.c uncomment the lines under *//NN+2* to use eight weights.

3.5.2 Additional experiments

With the instructions presented, the reviewer can train any combination of traces, neural networks and engine (cpu or gpu). To reproduce the *Mix 3x* of our paper, where the traces are sped up by 3x instead of 2x, scripts that generate these traces are provided at the *trace_tools* directory. Aside from the traces we provide, customized trace can be generated using *trace_tools/gen.py*.

We do not provide automated plotting for this experiment, but we do provide the script used to plot the graph in the paper at *src/linnos/ae_plot/bars.py*.

4. Crash course on tmux

The optional manual basic test and our main experiments rely on tmux to avoid having to have multiple terminals connected to a single machine, since our system contains many pieces. Mainly we need a pane for monitoring the kernel's logs, a pane for the kernel API remoting and a pane for running the experiments.

Tmux is a terminal multiplexer that allows the user to create multiple panes/windows and the option to leave them open and running in case the user disconnects. Tmux commands have a prefix key sequence, followed by a key to do the command. By default, the prefix sequence is **ctrl + b**. For example, if the user wants to change the active pane (the pane where the user can currently type commands, which is outlined in green), it will press **ctrl + b** then, once the keys are released, press an arrow key in the direction of the pane it wants to become active. Below we have a list of useful commands for our system. For more commands, we recommend this cheatsheet.

- *ctrl + b*, any arrow key: change active pane in the direction of the arrow key pressed.
- *ctrl + b*, z: toggle between display of all panes of fullscreen current pane.
- *ctrl + b*, pageup: start scroll mode. Press pageup to scroll up and pagedown to scroll down. Once done scrolling, press the *q* key to exit scroll mode.
- *ctrl + b*, d: detach from current tmux session, leaving it running on the background. Once the user wants to attach to the session again, run *tmux a*.
- *ctrl + d*: logs out of the current terminal. If ran from within a pane, that pane is closed.