MING HSEIH DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING, UNIVERSITY OF SOUTHERN CALIFORNIA

Optimized Load Balancing: Leveraging Hardware Awareness for Performance Enhancement

Jasneet Singh

May11, 2023

1. ABSTRACT

In this age of latest and greatest hardware, we carry out all sorts of measure to extract last bit of performance from our machines. While performing multi-tasking across consumer and enterprise use-cases, operating system has a crucial role to play, that is of balancing the load across all the available resources so that a given set of tasks can be executed in the as little clock time (execution time) as possible balancing constraints of energy. The currently running load balancing algorithm in the Linux kernel implicitly assume that by efficiently dividing the cpu time, all other resources are equally divided and hence, CPU division is the key criteria to avoid task contention and avoid the scenario of tasks unbalancing. However, the current approach overlooks other factors that contribute to contention of resources leading to sub-optimal division of load/threads (tasks are represented by threads in the Linux kernel) across hardware resources and longer clock execution time. In the first phase, this work first tries to match the performance of native load scheduling algorithm by proposing to replace the algorithm inside the kernel by the machine learning model and aims to measure the accuracy and other side-effects of deploying the model.

2. INTRODUCTION

2.1. MOTIVATION

Improving cpu load by efficiently managing the hardware resources can result in faster execution of task leading to either faster execution time for same number of tasks or executing more tasks (threads) given same amount of time, leading to savings in costs and energy.

2.2. SUMMARY OF EXISTING APPROCHES

In [1], Chen et al. extracted the parameters used by current Linux Scheduler (Completely Fair Scheduler) to make the load-balancing[1] decisions. They trained (in user space) a simple MLP model to achieve same accuracy as the default Linux load balancer and then deployed (in inference mode) the trained model in kernel to check for real-time accuracy and implications of introducing ML model in kernel. They developed two models, one based on floating point calculations and another on fixed-point calculations. During testing, they measured the accuracy as well as latency (extra run-time [clock time] to execute the can_migrate_tasks() function and load_balance() function)

[1] Load balancing is carried-out by Linux kernel to ensure that there is adequate load distribution across various computing units i.e. in order to avoid a scenario where a CPU is sitting idle and another cpu is over-loaded (i.e. has a long runqueue) Linux kernel periodically carries-out load-balancing

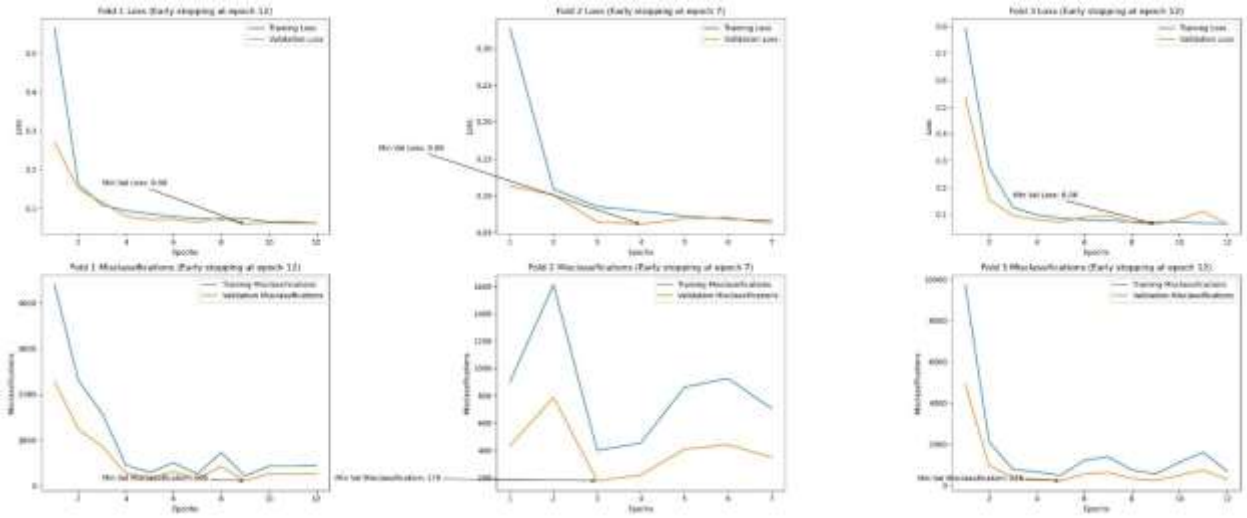
brought by in-kernel ML deployment), maximum difference between the length of runqueue[2] and actual execution time of real-world workloads.

Similarly, in order to reduce the latency introduced in kernel by introducing floating point calculations, fixed-point representation of weights was developed and deployed resulting in 2x improvement in latency yet delivering similar accuracy.

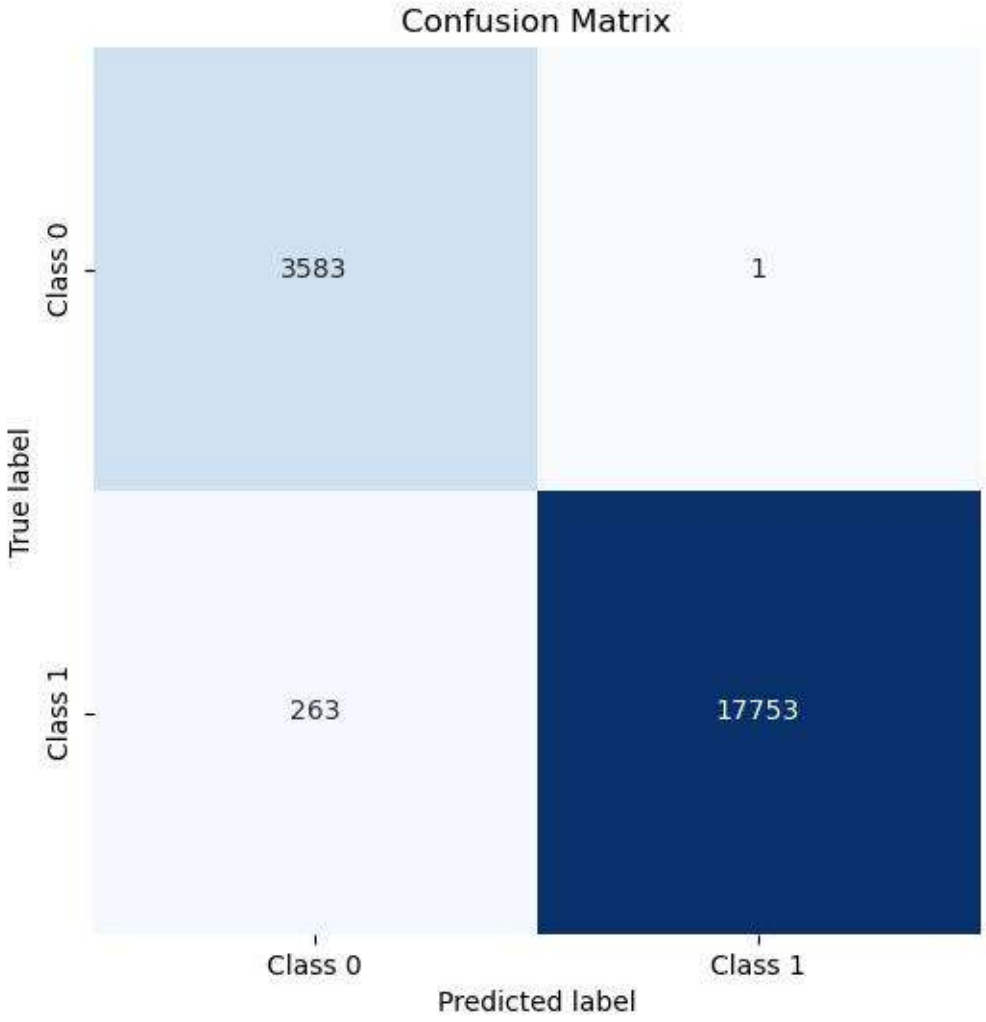
2.3 SUMMARY OF OBJECTIVES AND RESULTS

Achieved 99% test accuracy with 577 parameters. Extracted raw data by modifying the Linux kernel.

Testing Set had 21600 data points and minimum misclassification across runs is 106.



[2] Linux kernel maintain separate requeues for each CPU core. The idea of load balancing is to ensure that the difference between the length of runqueues across computing units should be as less as possible. By measuring the maximum difference of the runqueues with in-kernel ML implementation and measuring against the current kernel algorithm, the effectiveness of in-kernel ml algorithm can be judged



While the authors had extracted data using linux kernel 4.15, the report presented tried extracting the data on the linux kernel 5.15, in a way, charted unexplored territories.

3. PROBLEM FORMULATION

3.1 DATASET

Original intended data extraction involved two steps. First one is to attach existing code (eBPF functions developed using BCC library in Python available on GitHub) and Ubuntu 20.04 LTS Kernel and fetching all the parameters passed to the function in the function call. Secondly, in order to extract hard condition tests[3], kernel has to be

[3] eBPF programs can only access limited kernel functions, the hard condition tests in can_migrate_task() cannot be reproduced correctly in the eBPF probes. To resolve the inaccuracies brought by the uncaptured conditions, we modified the kernel and added a test flag to indicate whether the task has passed all the hard condition tests, as we only need cases for which migration is possible.

modified (again code available on GitHub.) Post this, the extracted fields (9 of them) have to be converted to about 15 fields (depending on the number of cores available.)

However, after following these steps, I could extract only 6 out of 15 fields, hence, I reverted to distillation to generate the dataset.

field	Range	distribution	remarks
src_non_pref_nr	Binary	random	
delta_hot	Binary	random	
cpu_idle	binary	random	
cpu_not_idle	binary	logic-based	
cpu_newly_idle	binary	logic-based	
same_node	binary	random	
prefer_src	binary	random	
prefer_dst	binary	random	
src_len	0~55[4]	guassian	mean = 0, std_dev = 3
src_load		exponential	
dst_load		exponential	
dst_len	0~55	guassian	mean = 0, std_dev = 3
delta_faults	0~10	guassian	
extra_fails	binary	random	
buddy_hot	binary	random	

[4] The range is taken 0~55, i.e. upper bound of 56, because the authors produced their results on Intel Xeon 4 machines that had 2 NUMA Nodes, 14 cores on each NUMA node and 2 SMT threads per Core, resulting in 56 cores (as classified by Linux kernel) and resulting in maximum load of 56

3.2. MODELS

MLP with 3 hidden layers was applied. The number of nodes in the three layers was 15, 12 and 10 and activation was ReLU. Output layer was the binary classification layer with loss function as Binary Cross-Entropy. Furthermore Stratified K-fold was selected as the method for cross-validation to avoid the overfitting.

Stratified K-Fold ensure that if the input data is unbalanced, then the training and validation sets carry similar weights for the different classes.

Model	3 Hidden layer with ReLU activation in hidden layer	
	577 parameters	
	L2 regularizer	0.001 regularization weight
	Weighted class (balancing)	
Optimizer	Adam	Default learning rate
Loss function	Binary cross entropy	Loss function
Metrics	Accuracy	Metrics
Validation	Stratified K-fold	num_folds = 3
	Early stopping callback	
EPOCHS	500	
BATCH_SIZE	64	

3.3. TRAINING METHODS

We trained the model and introduced early stopping if the change for three consecutive epochs is less than a pre-defined value. Using Adam optimizer, with default learning rate, the weight training was explored. Regularizer of 0.001 was chosen as an input to Adam Optimizer.

4. RESULTS

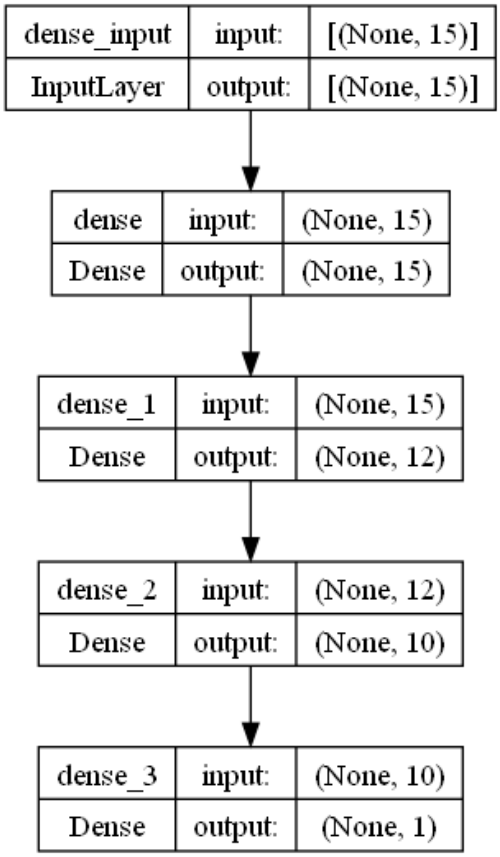
4.1. DATA INSPECTION

The data generated was 86,400 data points that was 15 dimensional. The output class was unbalanced with class 0 representing 18% weightage and class 1 representing 82% weightage.

4.2. MODELS

MLP Model with three hidden layers, Relu activation. Output layer is binary classification (sigmoid) with binary cross-entropy loss function.
Stratified KFold was carried with number of folds equal to 3.

Model	3 Hidden layer with ReLU activation in hidden layer	
	577 parameters	
	L2 regularizer	0.001 regularization weight
	Weighted class (balancing)	
Optimizer	Adam	Default learning rate
Loss function	Binary cross entropy	Loss function
Metrics	Accuracy	Metrics
Validation	Stratified K-fold	num_folds = 3
	Early stopping callback	
EPOCHS	500	
BATCH_SIZE	64	



4.3. TRAINING AND HYPER-PARAMETER OPTIMIZATION

The model was intended to be trained with 500 EPOCHS and BATCH-SIZE of 64.

5. INTREPRETATION AND RECOMMENDATION

5.1. ANALYSIS OF EXPLORED MODELS

After training an MLP model with 3 hidden layer leading to 573 parameters and comparing against the MLP model with 1 hidden layer and 177 parameters, we found that testing accuracy remans the same and validation accuracy and loss across different folds were also in the similar range. Final Model selected was the one that gave lowest miscalculations across the various validation folds.

5.2. RUN-TIME COMPLEXITY

Since, I was working on numerical data, hence, run-time complexity for the purpose of training was trivial.

5.3. RECOMMENDED SYSTEM SOLUTION

Since the trained model is going to be deployed in the kernel where complexity is forsaken, hence, the model must be able to attain accuracy but at the same time shouldn't hurt the latency (why would kernel be allowed to consume precious CPU/hardware resources, that were meant for running user applications.) Therefore, while lesser parameters are preferred, but simpler parameter representation i.e. fixed-point or integer weights are preferred to carry out the task so that load balancing decisions could be made quickly.

6. CONCLUSION

6.1. LEARNING OUTCOMES

This work is to be looked at as the first attempt in this vast and relatively new field of deploying ML models in environments where latency is as important as accuracy, thus, calling for models that deliver results (measurement metrics) yet not at the cost of increased latency.

Promising work can be carried out if one attains through understanding of the kernel and underlying architecture to decide probes to be used for fetching key metrics form the kernel.

Ability to design an efficient (less parameters) shall also be helpful in building a design that doesn't impacts latency.

Literature survey of current load balancing statistics across wide class of machines (data centers and consumer machines) can point towards the urgency of the problem and invigorate interest.

6.2. SUMMARY

This project opened avenues of further research in order to present the current state of affairs i.e. why kernel execution is still devoid of any widely known adoption of machine learning methods to solve underlying problems. It is known in literature that part of the reason for lack of deployment of ML models in kernel code also has to do with limitation of current kernel code to be only executed on CPU and lack of GPU/Accelerator interfaced available to carry out kernel's own's applications, thus hindering deployment of fine-tuned weights.

On one hand the work carried out used consumer-class machine, but load balancing has a bigger use-case in the data centers where every ounce of extra performance gains results in lower costs of services to end consumers or higher profitability for the data centers.

7. REFERENCES

[1] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. Machine Learning for Load Balancing in the Linux Kernel. In 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409963.3410492

[2] Henrique Fingler, Isha Tarte, Hangchen Yu, Ariel Szekely, Bodun Hu, Aditya Akella, and Christopher J. Rossbach. 2023. Towards a Machine Learning-Assisted Kernel with LAKE. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada. ACM, NewYork, NY, USA, 16 pages. https://doi.org/10.1145/3575693.3575697

[3] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI2: CPU Performance Isolation for Shared Compute Clusters. In Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13). Association for Computing Machinery, New York, NY, USA, 379–391. https://doi.org/10.1145/2465351.2465388

[4] Load balancing Code Walkthrough 10 Load Balancing on SMP Systems · Linux Process Schedule (gitbooks.io)

[5] Fixed Point numbers: Introduction to Fixed Point Number Representation (berkeley.edu)

APPENDIX

APPENDIX A1: MEANING OF EACH DATA POINT EXTRACTED

- **ts**: Timestamp at which the event was captured.
- **curr_pid**: Process ID of the currently running task.
- **pid**: Process ID of the task being considered for migration.
- **src_cpu**: Source CPU from which the task is being migrated.
- **dst_cpu**: Destination CPU to which the task is being migrated.
- **imbalance**: Load imbalance between the source and destination CPUs.
- **src_len**: Number of running tasks on the source CPU.
- **src_numa_len**: represents the number of tasks running on the source run queue (src_rq) that are running on their preferred NUMA nodes. In the context of NUMA systems, each task has a preferred NUMA node based on its memory access patterns. Running tasks on their preferred NUMA nodes can help in reducing NUMA faults and improving performance. The nr_numa_running field helps the load balancing algorithm to consider the number of tasks already running on their preferred NUMA nodes when making migration decisions.
- **src_preferred_len**: Number of running tasks on the source CPU that are preferred.
- **delta**: This variable represents the difference between the current task's runtime and the time it started executing on the source CPU. It is calculated as the difference between the task clock of the source runqueue (rq_clock_task(env->src_rq)) and the task's execution start time (p->se.exec_start).
- **cpu_idle**: Indicator if the CPU is in the idle state.
- **cpu_not_idle**: Indicator if the CPU is not in the idle state.
- **cpu_newly_idle**: Indicator if the CPU has just transitioned to the idle state.
- **same_node**: Indicator if the source and destination CPUs are in the same NUMA node.
- **prefer_src**: Indicator if the task prefers the source CPU.
- **prefer_dst**: Indicator if the task prefers the destination CPU.
- **delta_faults**: Difference in the number of faults between the source and destination CPUs. This variable represents the difference in the number of NUMA (Non-Uniform Memory Access) faults between the source and destination nodes. It is calculated as the difference between dst_faults and src_faults:

```
src_faults = event.p_numa_faults[src_nid]

dst_faults = event.p_numa_faults[dst_nid]

row['delta_faults'] = dst_faults - src_faults
```

Here, p.numa_faults represents the number of NUMA (Non-Uniform Memory Access) faults for a task on each node. NUMA is a memory architecture used in multiprocessor systems, where memory access time depends on the memory location relative to the processor. In NUMA systems, processors have their local memory and can access memory of other processors, but at a higher latency.

A NUMA fault occurs when a processor accesses memory that is not local to it, i.e., when it accesses memory from a different node in the NUMA architecture. This non-local memory access is slower compared to accessing local memory, which can impact the performance of a task running on the processor.

p.numa_faults is an array that stores the number of NUMA faults for a task on each node in the system. It is used to understand the memory access patterns of a task and helps in making informed decisions about task migration in a NUMA-aware load balancing approach. By considering the NUMA faults information, the load balancer can make better decisions to minimize the impact of non-local memory access on task performance.

- **total_faults**: Total number of faults for the task. This variable represents the total number of NUMA faults for a task. It is assigned from the event.total_numa_faults attribute
- **dst_len**: Number of running tasks on the destination CPU.
- **src_load**: Load average on the source CPU.
- **dst_load**: Load average on the destination CPU.
- **nr_fails**: Number of failed balance attempts.
- **cache_nice_tries**: `cache_nice_tries` is a variable used in the Linux kernel's load balancing algorithm. It is specifically related to the load balancing behavior when considering cache affinity.
  Cache affinity refers to the preference of a task to be executed on a CPU where its data is already present in the cache. In load balancing, sometimes the kernel may decide to migrate a task from one CPU to another to balance the load. However, migrating a task can cause a performance penalty due to loss of cache affinity, as the task's data needs to be loaded into the cache of the new CPU.
  The `cache_nice_tries` variable is used to control how many times the load balancer should attempt to migrate tasks while still considering cache affinity. When the load balancer tries to find tasks to migrate, it first considers tasks that have a lower cache affinity. If it doesn't find suitable tasks, it increments the `cache_nice_tries` counter and tries again, potentially considering tasks with higher cache affinity. The process is repeated until a suitable task is found or the maximum number of tries is reached.
- **buddy_hot**: Indicator if the task's buddy is running on the destination CPU.
- **throttled**: Indicator if the load balancing is throttled.
- **p_running**: Indicator if the task is currently running.
- **test_aggressive**: is set to '1' if all the following conditions are met. i.e. the thread is a suitable candidate for migration. The main motivation behind this research is to address the challenges posed by the complex nature of modern workloads and heterogeneous computing environments. Traditional heuristics-based load balancing mechanisms might not be able to efficiently adapt to these diverse scenarios. By concentrating on cases where migration is possible, the authors can train and validate their machine learning model on scenarios where it has the potential to improve the load balancing decisions. By doing so, they can demonstrate the effectiveness of their approach in optimizing resource utilization and enhancing the overall performance of the system.
  We don't want to consider those cases for training where any of this condition is true, essentially meaning that task cannot be migrated
  - *Throttled load balancing pair*: This condition checks if the load balancing pair (source and destination CPUs) is throttled, meaning that load balancing is temporarily restricted between the two. If the pair is throttled, the task cannot be migrated.
  - *Per-CPU kthreads*[5]: This condition checks if the task is a per-CPU kthread. If it is, the task cannot be migrated because it is where it needs to be.
  - *CPU affinity*: This condition checks if the task is allowed to run on the destination CPU based on the task's cpus_ptr (updated from cpus_allowed in the previous version). If the destination CPU is not in the task's cpus_ptr set, the task cannot be migrated.
  - *Task running state*: This condition checks if the task is currently running on the source CPU. If the task is running, it cannot be migrated.
- **can_migrate**: Indicator if the task can be migrated.
- **pc_0 and pc_1** respectively represent the performance counter available in the cpu.
  - pc_0 (PERF_COUNT_HW_CPU_CYCLES) counter represents the total number of CPU cycles elapsed during the task's execution. It gives an indication of how much time the CPU has spent executing the task.

[5] Per-CPU kthreads are kernel threads that are specifically bound to a particular CPU. They are designed to perform tasks on a single CPU and are not intended to be migrated. These threads often manage per-CPU data structures, and migrating them can result in additional overhead from accessing data structures on a different CPU.

- pc_1 (PERF_COUNT_HW_CPU_CYCLES) counter represents the total number of CPU cycles elapsed during the task's execution. It gives an indication of how much time the CPU has spent executing the task.

APPENDIX A2: INTREPRETATION OFEACH DATA POINT PROCESSED

- Only select rows that have **p_running equal to 0**, implying that we only want to consider tasks that are not running
- Only consider rows that have met **test_agressive of 1** implying we only want to train on the data points that are valid candidates for migration.
- Make a new column named '**delta_hot**' and assign a value of '1' if delta is less than '500000' and 0 otherwise. The reason to set delta_hot to 1 for delta values less than 500,000 is to categorize tasks as "hot" or "cold" based on their recent CPU usage.
- Create a new column named '**src_non_pref_nr**' which indicates whether the number of tasks running on the source CPU is greater than the number of tasks preferred to run on the source CPU (src_preferred_len). This could help the model identify if the source CPU has more tasks than it should ideally have, which could be a sign of potential load imbalance.
- Create a new column named '**src_non_numa_nr**' which represents the difference between the total number of tasks running on the source CPU (src_len) and the number of tasks running on the source CPU that are part of the same NUMA node (src_numa_len). This metric can help the model understand the distribution of tasks across the CPU and the NUMA node, which could be useful in making better load balancing decisions.
- Create a new column '**extra_fails**' which indicates whether the number of failed migrations (`nr_fails`) is greater than `cache_nice_tries`. This could help the machine learning model identify situations where the migration attempts have failed more often than expected, possibly indicating issues with the load balancing decisions. If the number of failed migrations (nr_fails) is greater than the cache nice tries (cache_nice_tries), set extra_fails to 1, otherwise, set it to 0.
- '**df.src_len**' = df.src_len - 2: Subtract 2 from the src_len column values.
- '**df.src_load**' = df.src_load / 1000: Divide the src_load column values by 1000.
- '**df.dst_load**' = df.dst_load / 1000: Divide the dst_load column values by 1000.
- '**df.delta_faults**' = df.delta_faults / df.total_faults.mask(df.total_faults.eq(0), 1): Calculate the ratio of delta_faults to total_faults. If total_faults is 0, use 1 instead to avoid division by zero.