THE UNIVERSITY OF CHICAGO

INTEGRATING MACHINE LEARNING INTO STORAGE PERFORMANCE SOLUTIONS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
ANNE MARIE FARRELL

CHICAGO, ILLINOIS
DECEMBER 2020

TABLE OF CONTENTS

LIST OF FIGURES iv

LIST OF TABLES v

ACKNOWLEDGMENTS vi

ABSTRACT viii

1 INTRODUCTION 1

2 RELATED WORK 4
- 2.1 Analyzing Performance on SSDs 4
- 2.2 Predicting Performance of SSDs 4
- 2.3 Improving Performance on SSDs 5
- 2.4 Applying Machine Learning to Systems 7

3 DECIDING WHEN TO FAIL OVER 9
- 3.1 Design of the Learner 9
- 3.2 The Cost of Failing Over 12
- 3.3 Biasing the Loss Function 14
- 3.4 Changing the Failover Threshold 15
- 3.5 Choosing a Bias Value or Failover Threshold Formulaically 16
  - 3.5.1 Failover Threshold 16
  - 3.5.2 Bias Value 16
  - 3.5.3 Minimizing Expected Latency 18
- 3.6 Choosing a Bias Value or Failover Threshold Experimentally 19

4 EVALUATION 22
- 4.1 Experimental Setup 22
- 4.2 Choosing the Bias Value and Failover Threshold 24
- 4.3 Effects of Bias on Predictions and Latency 26
  - 4.3.1 Classification 26
  - 4.3.2 Regression 28
- 4.4 Effects of Changing the Failover Threshold 29
- 4.5 Classification vs. Regression 30
- 4.6 Combining the Bias Value and Failover Threshold 31
- 4.7 Effects of Number of Replicas and Failover Overhead 31
- 4.8 Summary of Results 35

5 FUTURE WORK 37

6 CONCLUSION 38

REFERENCES 39

LIST OF FIGURES

3.1 Overview of ML-MittOS 10
3.2 Inputs and Outputs to Neural Network. 11
3.3 Neural Network Architecture 12
3.4 Actual vs. Predicted Latencies with No Bias and High Bias for Three Workloads 21

4.1 User-Level Simulation for Evaluating Overall Performance 22
4.2 Speedup of All Workloads using Regression with 30μs Failover Cost, 3 Replicas, Thresholds from Table 4.1 25
4.3 Breakdown of Classification Results with a Biased Loss Function and a Small 30μs Failover Overhead for AzureStorage Workload on Drive0 with Threshold from Table 4.1 27
4.4 Breakdown of Regression Results with a Biased Loss Function and a Small 30μs Failover Overhead for AzureStorage Workload on Drive0 with Threshold from Table 4.1 28
4.5 Breakdown of Results with a Varied Failover Threshold and a Small 30μs Failover Overhead for AzureStorage Workload on Drive0 29
4.6 Classification vs. Regression 30
4.7 Speedup of All Workloads with 30μs Failover Cost, 3 Replicas, Thresholds from Equation 31
4.8 Effects of Number of Replicas and Failover Cost 32
4.9 Effects of Number of Replicas, Broken Down by Workload, Threshold Equation Only 33
4.10 How the Accuracy of Predictions and Training Data Affect Performance of the Threshold Equation 34

LIST OF TABLES

4.1 Thresholds 24

ACKNOWLEDGMENTS

I'd first like to thank my advisor, Hank Hoffmann, whose experience and compassion has enabled me to make it though grad school. I'd also like to acknowledge my other committee members, Haryadi Gunawi and Janos Simon, for their time and advice on this thesis. Mingzhe Hao was also a huge help on this project with both his knowledge of MittOS and his feedback on earlier drafts of the proposal that turned into this dissertation.

I'd also like to thank all of my friends at the University of Chicago who helped me make it through grad school. In our research group, I'd like to thank Connor, Will, Ivana, Saeid, Yulie, Yi, Ahsan, Tejas, Nikita, Harper, Zhixuan, Bernard, Aji, and Gökalp for discussing research with me and helping me solve whatever problems I encountered. Thanks for always listening and providing helpful feedback, both in group meetings and in casual office chats. I'd also like to thank Ivana again for giving me invaluable advice on improving my CV and presentations.

Thank you to Joseph (Hing Yin), Will, Kavon, Joe, Samira, Steven, Pramod, and Rida, who all helped maintain my sanity during my first year of grad school (espcially Joseph, without whose patient explanations I might not have passed Discrete Math). I'd like to especially thank everyone who organized social events that helped with both my mental health and with understanding others' research. Thank you to Jean and Yi for starting the Graduate Women in CS social events, and to Ashka and Sophia for continuing them; without you, I would have many fewer female friends. I'd also like to thank all of the Ministers of Fun and Caffeine who have organized social events and tea times: Francesca, Suhail, Pranav, Bruno, Tushant, Ryan, and Casey (the honorary Virtual Minister who has kept virtual tea time running during the pandemic). Thanks, Chris, for restarting the pizza seminar and prodding other students to present their work, and for feeding both my brain and stomach. Thanks to Jesse and Brian for bringing your board games to play with everyone. Also thank you to Kartik, for helping found the Ministry and creating the Slack workspace that enabled

all of these events to happen.

Finally, I'd like to thank all of my other friends who discussed research with me, played Super Smash Bros., and generally helped me make it through grad school: Alex, Jonathan, Andrew (both the Andrew who regularly attended tea time and the Andrew who gave visualization advice), Valerie, Bogdan, Konstantinos, Reza, Ricardo, Akshima, Jenna, Weijia, Jasmine, Chengcheng, Michelle, Kate, Mikael, Nick, Tiago, Aritra, Tyler, Jas, and everyone else who I'll realize that I've forgotten to include here the day after the dissertation deadline.

ABSTRACT

The widespread adoption of SSDs has made ensuring stable performance difficult due to their high tail latencies, which are amplified in large systems. A promising approach to improving tail tolerance involves using machine learning to predict the latency of each request and using this information to decide whether to serve the request or fail over to a replica. Deciding whether to fail over to a replica is not as straightforward as setting a deadline for requests and then failing over if the latency is predicted to exceed the deadline (and not just because the predictions are sometimes inaccurate). To achieve the best performance, we need to consider all aspects of the system (including how many replicas are available, the cost of failing over, and the latency distribution of the workload). With this information, we can determine the expected latency of failing over to a replica and use it to decide when failing over is likely to improve performance. We incorporate this information into the system in two ways: by biasing the loss function of the learner to increase or decrease the predicted latencies (or for a classification problem, to increase or decrease the rate of positive or negative predictions), and by changing the threshold used to classify which requests are too slow. We find that both methods can achieve comparable performance improvements, but there is little performance benefit to combining them, suggesting that attempting to break down the problem and improve the machine learning predictions separately from tuning other aspects of the system may increase complexity unnecessarily compared to optimizing the whole system together. A holisitic approach and a deep understanding of the system is necessary to providing the best performance while minimizing system complexity.

CHAPTER 1

INTRODUCTION

Ensuring stable performance is becoming more difficult with the widespread adoption of solid-state drives (SSDs) [16, 27, 28, 35, 44, 53, 58, 68]. Previous systems that used hard disk drives (HDDs) had a latency baseline of 10s or 100s of milliseconds, but now SSDs offer an order of magnitude better performance, with latencies in the microsecond range. With this, a 1ms tail that might have previously seemed small begins to seem excessively slow compared to a 50μs regular access baseline, a slowdown of 20x! Furthermore, a single request can be composed of multiple input/outputs (I/Os), amplifying the effects of long tails. Consider a request involving ten I/Os with a p95 of 1ms. While the odds of any one I/O exceeding 1ms are only 5%, the odds of all ten I/Os in the request each finishing within 1ms is only around 60%. This becomes especially problematic in large systems containing many SSDs, each fulfilling a large quantity of I/O requests, where normally rare slowdowns occur regularly [3, 16].

Tail tolerance is growing worse. Increasingly short response times make existing tail tolerance strategies that involve passively waiting for a request to finish and then failing over to a replica when some timeout is reached very expensive and inefficient [6, 56, 70]. If we decide that a request that normally takes 50μs should timeout after the 95th percentile of, say, 500μs, then the slowest requests will still be 10x slower than average. Finding an appropriate timeout threshold that is general across different workloads is effectively impossible. If the timeout is too high, then it yields little benefit. A short timeout results in an increased workload from duplicating more I/O requests. In the most extreme case, setting the timeout to 0 and being blindly proactive [6, 66] would double the workload, which may require more resources and cause new problems. Since neither passively waiting nor being blindly proactive offer good results, the solution is to be selectively proactive and only retry requests that are predicted to be too slow.

MittOS [26] fulfills this principle of being selectively proactive by offering an interface for users to input their service-level objectives (SLOs). When MittOS receives an I/O request, it examines the metadata available in the operating system stack from the block device layer and SSD drivers and predicts whether the request will meet the SLO. When MittOS predicts that a request will be unable to meet the SLO, it returns an EBUSY signal to allow applications to failover to a faster node instead of waiting. With both OpenChannel SSDs [9] and with HDDs, which use a simpler mechanism compared to SSDs and an open-source I/O scheduler [1], MittOS achieves over 96% accuracy and can reduce overall I/O completion time by up to 35% compared to passively waiting.

One of the major drawbacks of MittOS is that it relies on white-box OpenChannel SSDs [9] to achieve reasonable accuracy. Unfortunately, software-defined flash storage [50] like OpenChannel SSDs [4] are still far from gaining widespread adoption and require a great deal of effort and expertise to successfully develop predictions for—predictions that do not translate across different devices. Most commercially-available SSDs are black-boxes that rely on proprietary algorithms that can vary significantly between devices. In order to cover real-world deployment scenarios, there is ongoing work to extend MittOS with a lightweight machine learning framework to predict when to send the EBUSY signal, which we hereafter refer to as ML-MittOS.

While it may sound straightforward to predict whether an I/O will meet the deadline and then use this prediction to decide whether to fail over to a faster replica, choosing when to fail over is more interesting than it may at first appear. Consider a request that is predicted to be a little bit faster than the threshold, so it will not fail over. If the threshold was set based on the amount of latency that is considered unacceptable (perhaps setting it to the p90 latency), then this request might still be fairly slow, and failing over may improve the latency. Since the neural network cannot make predictions with perfect accuracy, it is also possible that this request will take longer than predicted and will actually exceed the

threshold. This suggests that it may be advantageous to fail over sometimes even when a request is expected to meet the SLO. Of course, failing over more often also increases the risk that a fast I/O will fail over to a slower replica. To further complicate things, a real system may have three or more replicas, giving an I/O several chances to fail over and try to improve its latency, but each time adding a bit of latency from the network cost.

In this thesis, we:

1. Explain the relationship between the expected latency and several aspects of the system and workload

2. Propose two methods to influence how often the system decides to fail over: modifying the loss function of the neural network to more heavily penalize certain types of mispredictions, and changing the threshold used to determine when to fail over

3. Discuss when to use each of these strategies

We discuss related work in Chapter 2. In Chapter 3, we explore how to modify how often the system fails over by biasing the loss function of the neural network or changing the failover threshold. We discuss the results of our experiments in Chapter 4 and future work in Chapter 5. Finally, Chapter 6 summarizes our conclusions.

CHAPTER 2

RELATED WORK

Previous work focused on several areas: Some papers analyze the causes of tail latencies on solid-state drives and quantify the extent of the tail tolerance problem [3, 16, 24, 27, 28, 41]. Others focus on predicting latencies on SSDs more accurately [11, 30, 34, 36, 42]. Using these predictions and other insights, other work attempts to improve SSD performance [4, 6, 9, 17, 26, 31, 33, 35, 44, 50, 53, 56, 66, 68, 70]. Besides using machine learning to improve SSD performance, there are a number of papers discussing how to improve other systems outcomes using machine learning, such as [7, 8, 10, 12, 14, 15, 18–23, 32, 37–40, 43, 45–48, 54, 57, 59, 63, 65, 69, 71, 72].

2.1 Analyzing Performance on SSDs

There have been many studies quantifying tail latencies and analyzing their causes on SSDs. A summary of some of the difficulties associated with tail latencies on SSDs can be found in [3]. Similarly, Dean and Jeffrey discuss the many causes of tail latency in [16]. Li et al. provide an exploration of a variety of sources of tail latency across the hardware, OS, and application levels [41]. In [28], Harris summarizes many of the performance issues of SSDs. Hao et al. quantify the tail latencies of SSDs on a large number of drives and explore the root causes of slowdowns [27]. Gunawi et al. provide an analysis of the possible causes of fail-slow faults on various types of hardware [24]. These papers provide useful motivation for work like ours that focuses on improving tail tolerance.

2.2 Predicting Performance of SSDs

There is a variety of existing work on predicting performance of solid-state drives [30, 42]. Most prior work focuses on aggregate data like predicting the average latency over a period of

time, based on the aggregate request rate, I/O sizes, etc. Li and Huang evaluate the efficacy of black-box models to predict latency, bandwidth and throughput on SSDs [42]. Huang et al. further explore black-box models and how much more accurate they are on SSDs than HDDs [30]. These approaches can achieve good accuracy with simpler machine learning mechanisms like decision trees or random forests, but they are unable to make accurate predictions on a short enough time scale to use for making decisions about individual I/Os. Latency spikes occur at a millisecond scale [26], so more fine-grained predictions are necessary to accommodate the fast pace of modern storage.

Some work focuses on reverse-engineering and predicting the behavior of specific proprietary elements of commercial SSDs. Kim et al. propose a methodology for extracting several parameters that impact performance on SSDs [34]. In [11], Chen et al. discuss the potential performance gains from exploiting the internal parallelism of SSDs. Kim et al. develop a model for SSD performance by extracting information about the behavior of the write buffer and garbage collection [36]. Rather than try to reverse-engineer proprietary SSDs, we use machine learning to predict performance on black-box devices.

2.3 Improving Performance on SSDs

Since the original MapReduce paper [17], there have been a number of papers focusing on mitigating tail latencies in large systems. Zaharia et al. design a new scheduling algorithm for Hadoop that performs better in heterogeneous environments like vitualized data centers [70]. Pisces [53] provides weighted fair sharing and performance isolation on shared storage services. Dolly [6] clones small jobs to mitigate slow stragglers, using delay assignment to avoid contention in the intermediate data transfer. The CosTLO framework [66] issues requests with multiple forms of redundancy to meet a latency variance goal while minimizing configuration cost. Yang et al. offers a framework for dividing I/O scheduling across the storage stack to improve performance and reduce tail latencies [68]. PBSE [56] addresses tail

tolerance on networks experiencing node-level network throughput degradation by improving speculative execution. These approaches all consider different techniques for scheduling I/Os and dealing with stragglers. We attempt to provide a simpler and more general approach by using machine learning to decide whether to serve a request on a particular device. Our approach does not necessarily replace the performance insights presented by these papers; rather, it offers another technique for improving tail latencies.

Several papers analyze a specific aspect of SSD performance and offer solutions. Mesnier et al. propose classifying I/O to enable different categories of data to use different storage system policies [44]. In [35], Kim et al. show how SSDs fail to meet SLOs because of garbage collection; they propose a scheme to isolate flash memory blocks from different VMs so they no longer interfere with other VMs during garbage collection. FlashBlox [31] improves tail latencies by taking advantage of different channels and dies to better isolate virtual SSDs. Yan et al. address garbage collection using various strategies to avoid GC-induced slowdowns [67]. Kang et al. also attempt to reduce tail latencies caused by garbage collection using reinforcement learning [33]. These techniques improve different aspects of SSD performance than our work and can be implemented independently to potentially improve performance even further.

Rather than try to reverse-engineer proprietary SSDs, some work attempts to improve visibility and control over the inner workings of SSDs. In [50], Ouyang et al. propose software-defined flash that allows the host software to have direct access to flash channels on the SSD. Open-Channel SSDs [9] like LightNVM [4] can offer more predictable latency than traditional black-box devices. Using machine learning to predict performance may be unnecessary on these types of devices, but our work offers a solution for the proprietary SSDs that are still popular.

2.4 Applying Machine Learning to Systems

Besides improving SSD performance, there are a variety of other papers that apply machine learning to solving systems problems [7, 8, 10, 12, 14, 15, 18–23, 32, 37–40, 43, 45–48, 54, 57, 59, 63, 65, 69, 71, 72]. The most similar to our work is probably [21]. Ding et al. apply machine learning to choosing a system configuration that minimizes energy consumption while meeting a latency constraint and find that incorporating knowledge of the systems problem is more effective in improving results than improving the accuracy of the learner. Similar to our work, they use a biased learner to improve the systems outcome. Unlike our work, Ding et al. explore biasing matrix completion algorithms and Bayesian models using multi-phase sampling, while we bias a neural network by modifying the loss function or changing the threshold used for classifying inputs. Another key difference is that they try to minimize energy consumption under a latency constraint, while we are only interested in performance.

There is also similar work utilizing machine learning to predict response times on hard disk drives. Wang et al. use CART models to make black-box predictions of response times [63]. Crume et al. use machine learning to predict per-request response times on hard drives and employ Fourier analysis to deal with periodic behavior [15]. While one might expect that these ideas can be easily applied to other types of storage devices, the internal workings of SSDs are sufficiently different from HDDs that many of the insights into HDD performance are not applicable to SSDs.

A number of other works also use machine learning to improve systems outcomes like performance or energy-efficiency. For example, the Proteus framework combines control theory, machine learning, and programming language tools to enable developers to easily create adaptive applications [8]. CALOREE combines machine learning to model the effects of complex resource interactions on latency with control theory to meet a performance goal with minimal energy usage while responding dynamically to changes in the environment

[45]. In [32], Imes et al. use a machine learning classifier to choose the most energy-efficient resource configuration for an HPC workload. Memory Cocktail Therapy provides a learning-based framework for balancing the lifetime, performance, and energy efficiency of non-volatile memories [20]. ESP predicts interference between applications running on the same hardware to improve performance [46]. LEO uses probabilistic graphical models to estimate power and performance for a running application [47]. While these papers solve different problems and use different machine learning techniques than our work, they have the same goal of applying machine learning to improving systems outcomes, rather than the popular reverse of improving systems for performing machine learning.

CHAPTER 3

DECIDING WHEN TO FAIL OVER

Next, we explain how the learner works and how we can bias it to fail over more (or less) often by either modifying the loss function or changing the threshold for classifying which requests should fail over. We also explain how failing over affects the expected latency and use this knowledge to choose how to bias the learner and by how much. It turns out to be much easier to select a good threshold than to select the right amount of bias for the loss function.

3.1 Design of the Learner

ML-MittOS is an extension of MittOS [26] that uses a lightweight machine learning framework to predict whether an I/O will fail to meet the SLO (EBUSY) or will complete on time (not EBUSY). As shown in Figure 3.1, ML-MittOS interacts with the block-device layer and SSD driver to pull metadata like the list of previous latencies and number of pending I/Os in order to make predictions. When it decides that a request should trigger an EBUSY signal, ML-MittOS rejects the I/O and returns to the user application, allowing the application to retry the request on a faster node.

There are two ways to design the learner to decide when to fail over. The first is to perform classification to label each request as EBUSY or not EBUSY. The second is to perform regression to predict the latency of each request and then compare the predicted latency to a threshold to decide whether to fail over. Classification sounds like a good fit for the problem and is typically much easier to achieve high accuracy than regression, but we find that regression is surprisingly useful in this case. It is more flexible than classification because it allows us to easily change the failover threshold without having to retrain on a re-labeled training set. Additionally, because this is fundamentally a classification problem,

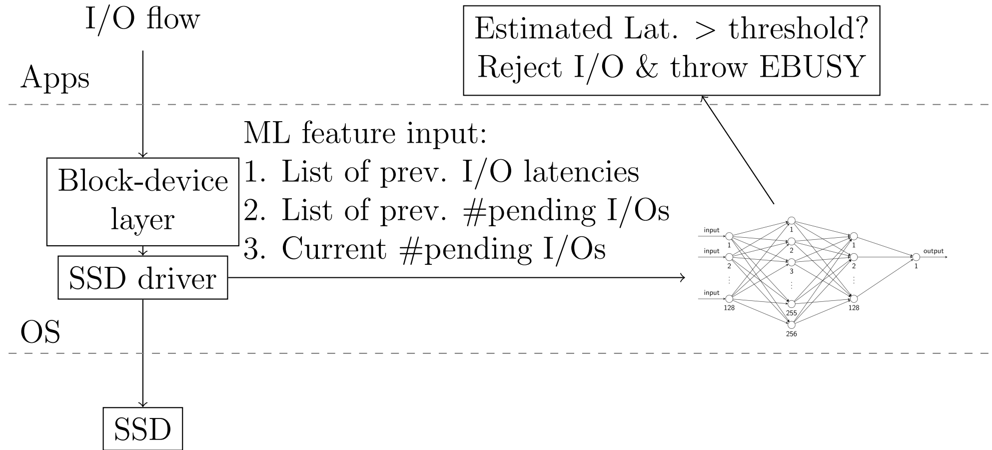


Figure 3.1: Overview of ML-MittOS
ML-MittOS interacts with the block-device layer and SSD driver to pull metadata like the list of previous latencies and number of pending I/Os in order to predict EBUSY/not EBUSY. When it decides that a request should trigger an EBUSY signal, ML-MittOS rejects the I/O and returns to the user application, allowing the application to retry the request on a faster node. This figure is based on [25].

the predicted latencies do not necessarily need to be very accurate as long as they are useful for deciding whether to fail over. In fact, because our main goal is classification in both cases, we can use a very similar setup for both classification and regression and achieve similar predictions for both.

Figure 3.2 shows an example of the input and output features to the learner used by ML-MittOS. The input consists of historical data about the latency and number of page I/Os in the queue for the most recent I/Os, plus the number of page I/Os pending in the current request (which has unknown latency). When performing classification, the output is EBUSY/not EBUSY encoded as a 1 or 0, respectively. For regression, the inputs are the same, but instead of an output of 1 (EBUSY) or 0 (not EBUSY), the output is an integer latency prediction in microseconds, which can then be converted to an EBUSY/not EBUSY prediction based on whether it exceeds the failover threshold.

While there are a number of possible ways to implement the learner, we use a simple neural network. The neural network is implemented in TensorFlow [5] using the Keras API [13] as a Sequential model with 4 Dense layers, as shown in Figure 3.3. The first 3 layers

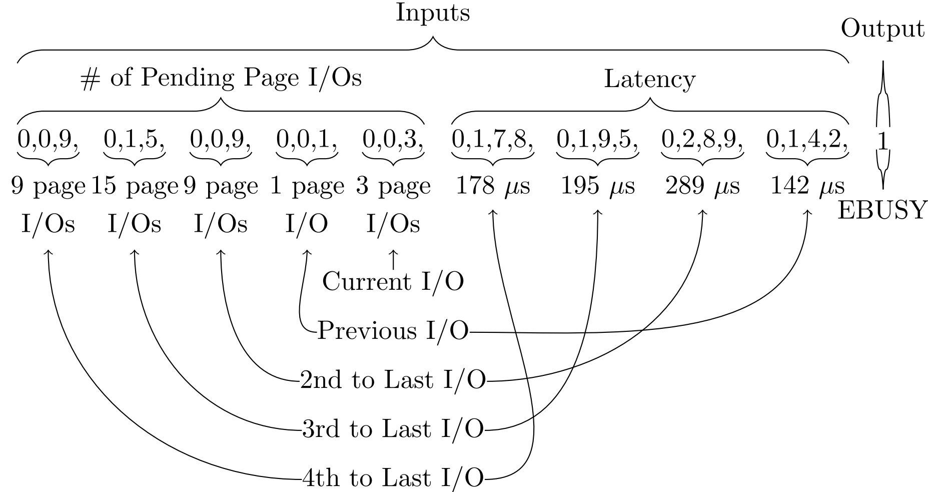


Figure 3.2: Inputs and Outputs to Neural Network.
In this example, the target I/O has 3 pending page I/Os in front of it in the queue. The previous request took 142μs with 1 page I/O in the queue. There are four previous requests used in this example to predict whether the current I/O should be EBUSY or not EBUSY. The output from the neural network in this case indicates that the request is EBUSY, so it should be rejected and retried on a different node in order to maintain acceptable performance. For regression, the inputs are the same, but instead of an output of 1 (EBUSY) or 0 (not EBUSY), the output is an integer latency prediction in microseconds, which can then be converted to an EBUSY/not EBUSY prediction based on the failover threshold.

have 128 or 256 neurons each and use ReLU activation functions, while the 1-neuron output layer uses either a sigmoid (for classification) or linear activation function (for regression). When performing classification, it is important to set the bias_initializer (which is different from the bias values described in the rest of this document) correctly to the natural log of the ratio of EBUSY requests to not EBUSY requests, as described in [2]. Failing to do so will sometimes randomly result in the neural network not learning the desired result when changing the loss function as described in Section 3.3. When using the network to perform regression, there is also a separate post-processing step to convert the predicted latency output into an EBUSY/not EBUSY decision.

We only make a few changes to get the neural network designed for classification to instead perform regression:

1. We change the activation function of the output layer from sigmoid to linear to allow

outputing latencies greater than 1.

2. We must modify the loss function from binary crossentropy to mean squared error, as that is more appropriate for regression.

3. Because the output is now numerical latency predictions, it is necessary to add a post-processing step to compare it to the failover threshold and convert it to EBUSY or not EBUSY.

4. The bias_initializer is not necessary for regression, only for classification with imbalanced classes.

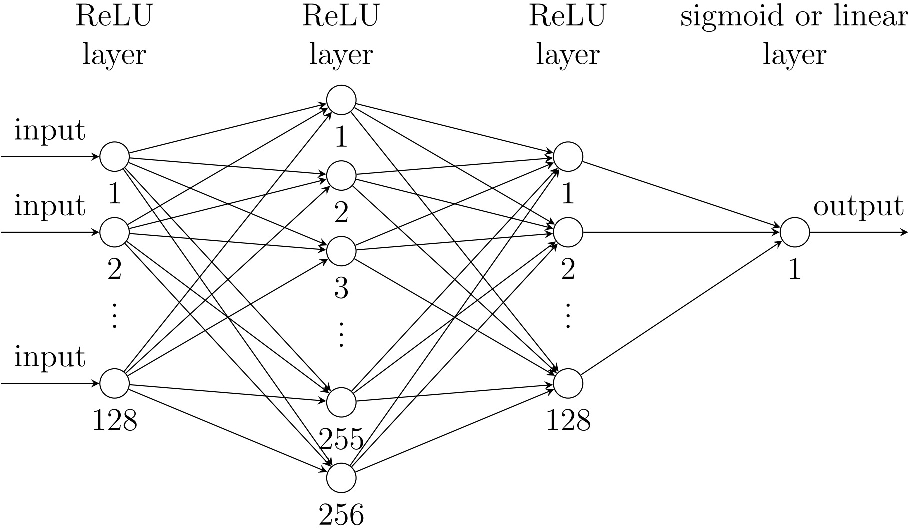


Figure 3.3: Neural Network Architecture
ML-MittOS uses a lightweight neural network consisting of 3 linear layers with ReLU activation functions, with a simple 1 neuron output layer that uses either a sigmoid (for classification) or linear activation function (for regression). When using the network to perform regression, there is also a separate post-processing step to convert the predicted latency output into an EBUSY/not EBUSY decision.

3.2 The Cost of Failing Over

To create a system that minimizes the latency, we need to understand how various system parameters affect the expected latency. Consider a system with r replicas with an average

failover cost of f. Let μ_t be the expected time it takes to complete a request. We will denote the mean latency of the sample data as $\bar{t}$, and let $\bar{t_n}$ and $\bar{t_e}$ be the mean completion times of requests that are not EBUSY and EBUSY, respectively, and p be the probability of a request being EBUSY.

Since all requests are either EBUSY or not EBUSY, the expected time it takes to complete a request can be given by

$$\mu_t = (1-p) * \bar{t_n} + p * \bar{t_e} \tag{3.1}$$

We can compute most of these values except $\bar{t_e}$ from the training data needed for the neural network by using NumPy [49, 60] to count which requests are EBUSY or not EBUSY for a given threshold and the mean latency of those requests. To find the average time $\bar{t_e}$ of a request that is EBUSY, we must consider μ_t to be a recursive function dependent on the number of replicas r. When $r = 1$, it is impossible to fail over, so $\mu_t(1)$ is just the average request time $\bar{t}$. For $r > 1$, an EBUSY request will failover to a replica for a failover cost f, so we have

$$\mu_t(r) = (1-p) * \bar{t_n} + p * (\mu_t(r-1) + f) \tag{3.2}$$

This equation is sufficient to write a function that can estimate the expected request latency given a sample of request latencies, an EBUSY threshold, the number of replicas, and the network cost of failing over. To improve performance when there are many replicas, we can solve the non-homogeneous recurrence relation by finding the roots of the characteristic equation and the particular solution to get

$$\mu_t = \frac{(f - \bar{t_n} + \bar{t}) * p^r + (\bar{t_n} - \bar{t}) * p^{r-1} + (\bar{t_n} - f) * p - \bar{t_n}}{p-1} \tag{3.3}$$

This equation depends on five variables. The number of replicas r and average failover cost f are known values that depend on the system configuration. The mean latency $\bar{t}$ is determined by the latency distribution of the workload and can be measured on the same

data used to train the neural network. The probability of predicting EBUSY p and the mean latency when not EBUSY $\bar{t_n}$ can also be measured on the training data as long as we can determine which requests are EBUSY/not EBUSY. This is easy if the neural network is already trained, since we can simply run it and get the predictions, but trying to predict how changing the neural network will affect the predictions is more challenging and will be discussed in the following sections.

There are two major ways we consider to change when the system decides to fail over to influence p and $\bar{t_n}$. The first is to change the loss function of the neural network to more heavily penalize a particular type of misprediction. For classification, this takes the form of penalizing false negatives more harshly than false positives; whereas for regression, this means increasing the penalty on under-estimates of the latency and reducing the penalty on over-estimates. This is discussed in detail in Section 3.3. The other way is to simply change the failover threshold so that more requests will fail over. In effect, the neural network remains the same, and we simply change which requests are labeled as ebusy and not ebusy. We discuss changing the threshold in Section 3.4.

3.3 Biasing the Loss Function

One way to influence when the system fails over is to change the loss function to more heavily penalize certain types of mispredictions. For classification, this is simple if we use the class_weight parameter when fitting the model in Keras. Class weights are typically used when the training data has an unbalanced number of examples for each class, such as when 95% of the training examples are zeros, but it is important to ensure that the network also learns to predict ones [2]. Here, we use class weights to integrating the relative costs of mispredicting a 1 as a 0 and vice versa into the classifier. When performing classification, the cost of predicting false positives and false negatives is different. Accepting a tail request is typically much more expensive than over-rejection, so we increase the penalty on false

negatives by setting the class weight for 1 (EBUSY) to a bias value $b > 1$, and we keep the penalty for false positives the same by setting the class weight for 0 (not EBUSY) to 1. The bias value b is unrelated to the bias_initializer that is needed to ensure that the starting values of the neural network are close enough to converge to the correct values.

For regression, it is also possible to bias the loss function to more heavily penalize underestimates. Inspired by [29] and [55], we modify the mean squared error as follows: Let y_i be the i-th latency value we want to predict and $\hat{y}_i$ be the corresponding prediction. Then, for n samples, the mean squared error (MSE) is defined as $MSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$. We define the asymmetric mean squared error (AMSE) to be:

$$AMSE(y, \hat{y}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 * (sign(y_i - \hat{y}_i) + b)^2 \tag{3.4}$$

, where $0 \leq b \leq 1$ is the amount of bias added. When $b = 0$, there is no bias, and the loss reduces to normal MSE. Higher values of b will penalize underpredictions (when $y_i < \hat{y}_i$) more heavily, while also reducing the penalty on overpredictions (when $y_i > \hat{y}_i$). It is also possible to use a negative value for b to increase the penalty on overpredictions, but we do not explore this possibility because it is rare for the system to be set up in a way such that failing over less often would be advantageous.

3.4 Changing the Failover Threshold

Biasing the loss function to change the penalties on over- and underestimates is not the only way to influence how often the system predicts EBUSY and fails over to a replica. Another possibility is to change the latency threshold for a request to be EBUSY. With classification, this would require changing the class labels and retraining the neural network on the new input. For regression, however, it is easy to raise or lower the EBUSY threshold without retraining. Compared to the biased loss function described in Section 3.3, changing the

EBUSY threshold has a very straightforward effect on which requests are predicted to be EBUSY. When changing the threshold, the predictions from the neural network remain the same and only need to be compared to a different constant value. This makes it much easier to predict how changing the threshold will affect the total latency of all the requests in the failover simulation.

3.5 Choosing a Bias Value or Failover Threshold Formulaically

It is not difficult to modify how often the system fails over by modifying the loss function of the neural network or changing the failover threshold, but the real challenge lies in choosing the correct bias value and failover threshold without having to retrain the neural network to try every possibility. As we saw in Section 3.2, if we can understand how the bias value and failover threshold affect the probability of a request being EBUSY p and the mean latency of requests that are not EBUSY $\bar{t_n}$, then we can use Eqn. 3.3 to compute the expected latency.

3.5.1 Failover Threshold

When changing the failover threshold, we do not need to retrain the neural network when performing regression, so it is easy to measure the necessary values using the predictions from the neural network on the training data. We use NumPy [49, 60] to count how many requests are EBUSY or not EBUSY for a given threshold and compute the mean latency of those requests.

3.5.2 Bias Value

It seems like it should be possible to predict the expected latency for a particular bias value b in a similar manner, but it turns out to be much more difficult to predict how biasing the loss function affects the predictions. In this subsection, we explain our efforts to find a

formula mapping a bias value b to an expected latency that holds for all workloads, but the best fitting equation we find turns out to not be very useful for choosing a bias value, as we see in Section 4.2. We discuss how we might try to further improve our results in Chapter 5.

We began by experimentally fitting a curve that mapped the bias value b to the EBUSY probability p when the threshold is given by Table 4.1 for each of the workloads, for both the classifier and regressor. We could find a sigmoid function for each workload that fit well, but the coefficients differed significantly for each of the workloads, so this did not allow us to generalize about the effects of using a particular bias value on an unseen workload. Because the threshold chosen affects p, and all the workloads have different latency distributions, we also tried fitting a curve from a bias value to p where the threshold is set to the p95 latency for each workload, but we still found that the coefficients varied wildly among different workloads. To further reduce the effects of choosing the threshold, we tried mapping b to the quantile of the latency distribution that corresponds to a particular value of p for the thresholds in Table 4.1 and the p95 latency, but still found that the coefficients for each workload were too different to be useful. Finally, to attempt to separate out the effects of each workload having a different latency distribution, we tried mapping b to the average change in latency of each request between the biased and unbiased regressors. We find that the coefficients for each workload are similar enough that the average change in latency Δt between a biased and unbiased neural network for all of the workloads can be approximated by

$$\Delta t = -ln(1 - b) * \bar{t} \tag{3.5}$$

With this equation, we can compute Δt for a given bias value b and add it to the predicted latency of each request computed by the unbiased regressor. Then, these latency values are used to estimate the probability of a request being EBUSY p and the mean latency of requests that are not EBUSY t_n. Unfortunately, the average change in latency between the unbiased and biased predictions is not a good predictor of the actual change in latency of

any particular I/O, making it difficult to find a useful formula for choosing a good bias value. Additionally, this formula does not work for the biased classifier, only the regressor.

To explain why quantifying the relationship between the bias value and its effect on predictions is so difficult, we can look at Figure 3.4. Figure 3.4 plots the actual latencies against the predictions made by an unbiased regressor on the left and a biased regressor with a high bias value of 0.9 on the right for three different workloads. The color of each point indicates by how much the prediction of that specific request differs between the biased and unbiased neural networks. The predictions of the biased neural network can be as much as 60,000 or even 130,000μs higher than those of the unbiased neural network, depending on the workload, but they have not shifted in a predictable manner. One might expect all the points to simply shift upward, or for the points farthest below the diagonal (where the latency is being underestimated) to shift the most, but this does not appear to be the case. The BingIndex and BingSelect workloads shown appear to have horizontal bands of requests that are predicted to be the same latency (despite having different actual latencies) and that move together when bias is applied, but the AzureStorage workload does not. For AzureStorage and BingIndex, many of the darkest points that moved the farthest started in the middle of the unbiased chart but ended up on the upper left of the biased chart. With such interesting behavior, it appears that the average change in latency is not a good means of predicting the change in latency of any particular request. We include the results of choosing a bias using the average change in latency equation in Chapter 4 to verify that it does not work well, but we do not recommend using it in practice.

3.5.3 Minimizing Expected Latency

Once we have a function for computing the expected latency for a particular bias value or failover threshold, we use the minimize_scalar function from SciPy [62] to find the bias value or failover threshold that minimizes the expected latency. We use the Bounded method and

specify that the threshold should be bounded by the minimum and maximum latency values of the training data, since it does not make sense to search for an optimal threshold outside of that range.

3.6 Choosing a Bias Value or Failover Threshold Experimentally

The previous section offered formulas for selecting a good bias value or failover threshold, but we also consider several experimental methods for making this decision by testing many different values. Several of these are practical ways to choose the bias value or failover threshold without having to test all of the infinite possibilities (namely, the "greedy" and "greedy skip" algorithms), while others are intended to show bounds on the best results achievable ("oracle" and "global").

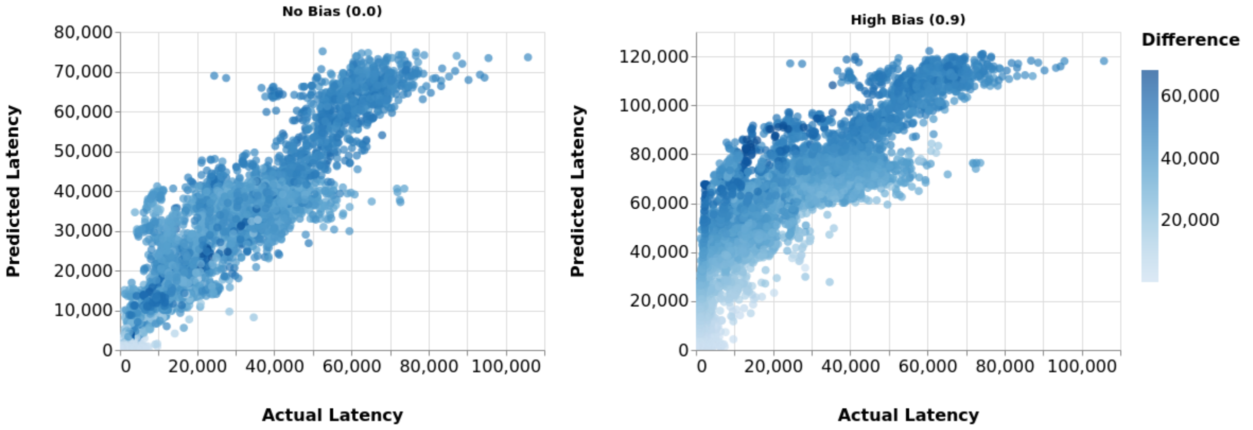
1. **Oracle:** The oracle represents what happens if we choose the bias value or failover threshold with the best latency from the testing data.

2. **Global:** The global strategy is achieved by choosing the bias value or failover threshold that does best on the training data. This should usually be pretty close to the oracle, but in some cases it chooses a suboptimal value. The biased neural network usually behaves similarly on the training and testing sets, but the difference between the global and oracle strategies reveals that there are some workloads for which the testing data differs somewhat from the training data.

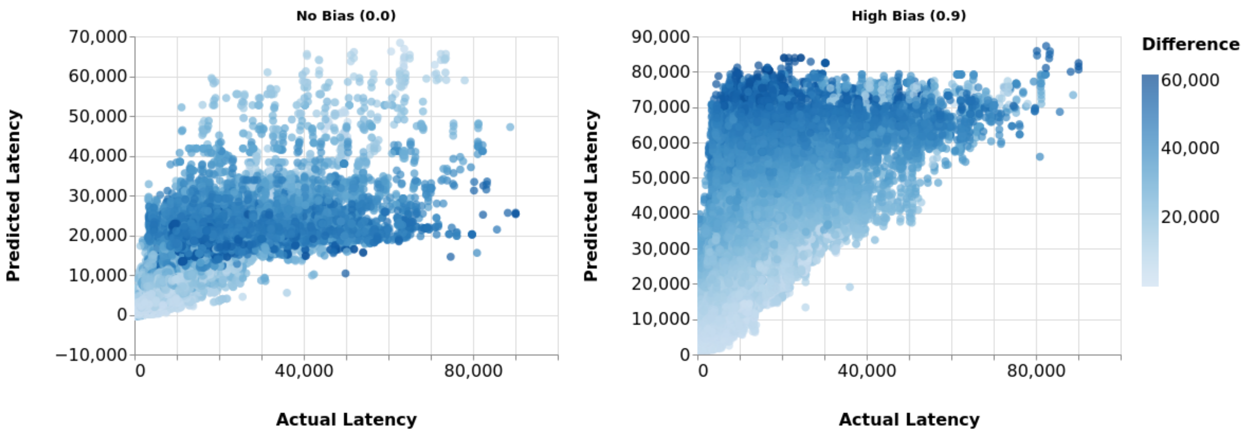
3. **Greedy:** This is a simple greedy algorithm for choosing the bias value, where we simply keep increasing the bias value until we see the latency start to get worse.

4. **Greedy Skip:** To prevent the greedy algorithm from stopping too early when the data is noisy, we consider a modification to the greedy algorithm called "greedy skip". This works like the greedy algorithm, except that we compare the predictions for all

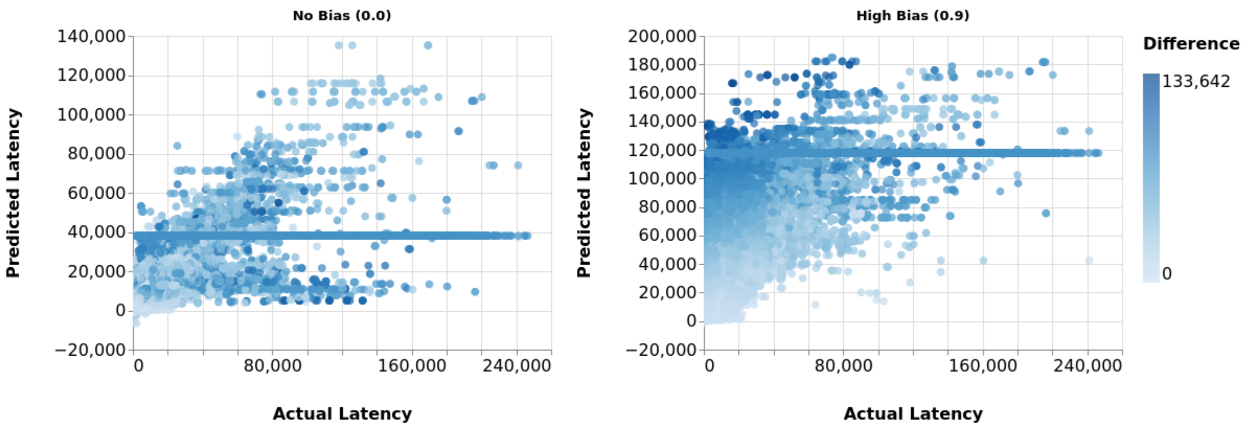
the requests to the predictions from the unbiased neural network. If fewer than 4% of the requests are predicted differently with bias than without, then we skip this bias value and move on to the next one. When comparing whether the latency is increasing or decreasing, we ignore the skipped bias values and only look at bias values that are sufficiently different from unbiased predictions. The 4% threshold was originally chosen to identify when the neural network failed to learn the bias value due to being poorly initialized, but after fixing that issue, it turns out to still be useful.



(a) AzureStorage on Drive0



(b) BingIndex on Drive3



(c) BingSelect on Drive3

Figure 3.4: Actual vs. Predicted Latencies with No Bias and High Bias for Three Workloads The graphs on the left show the the predicted vs. actual latencies when there is no bias (or a bias value of 0), while the charts on the right show a high bias value of 0.9 for three different workloads. Notice how there is no clear pattern as to which points move the most when increasing the bias value, and each workload shows very different behavior.

CHAPTER 4

EVALUATION

4.1 Experimental Setup

To evaluate overall system performance, we use a user-level simulation of a deployment scenario depicted in Figure 4.1 with a client, several servers, and a failover overhead of 30μs. In the example shown, a client sends a request for 100k reads with a p90 deadline. Servers

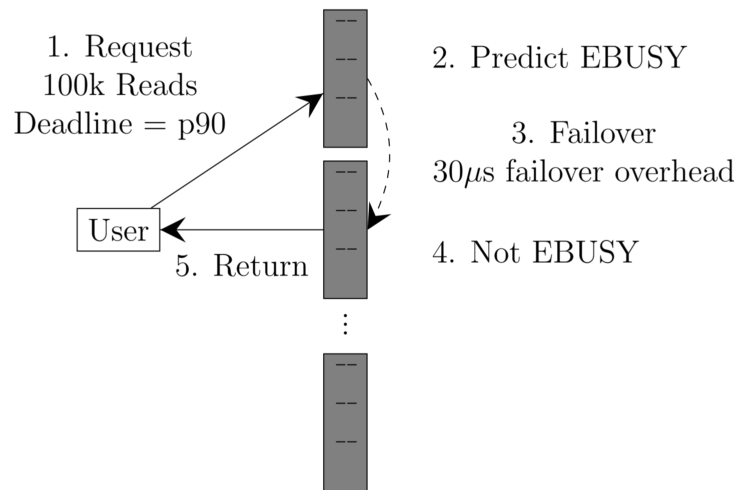


Figure 4.1: User-Level Simulation for Evaluating Overall Performance
In this example, a client sends 100k read requests for with a p90 deadline. When the server predicts a request will meet the p90 deadline, it serves the request as usual; otherwise, it will failover to the next server. This continues until a server predicts that the request is not EBUSY or it reaches the last replica, where the I/O request is completed and returned to the user. This figure is based on [25].

handle I/Os according to the latency distribution collected from actual workloads. When the server predicts a request will meet the p90 deadline, it serves the request as usual; otherwise, it will failover to the next server. This continues until a server predicts that the request is not EBUSY or it reaches the last replica. The important metric here is end-to-end latency, which is the sum of the I/O latency plus any applicable failover penalty. For example, say there are three servers running industry workloads. A client sends a request to the first server with a deadline of 500μs, which is p90 in this case. The first server returns EBUSY

and fails over to the second server with a network cost of 30μs. The second server predicts that the request will meet the latency goal and completes the I/O in 200μs. The overall latency in this scenario is $30 + 200 = 230\mu$s.

To simulate this scenario, we iterate through a list that contains the latency and EBUSY/not EBUSY predictions for each request. When a request is predicted to be not EBUSY, its latency is simply the latency associated with that request. When a request is EBUSY, three things happen:

1. First, we must compare the number of replicas available to the number of failovers so far. If there are no replicas left to simulate failing over, then we return the current total latency.

2. If there is another replica available, then we simulate failing over by skipping to a random request in the list, incrementing the number of failovers, and adding the network failover cost to the overall latency total.

3. We repeat the process of checking if the request is EBUSY and failing over until we reach a request that is not EBUSY or run out of replicas.

We test our system using 12 different traces from industrial cloud workloads with different latency characteristics. AzureStorage is the general storage for the Azure platform. BingIndex is the indexing server for the Bing search engine, while BingSelect is the selection algorithm. Cosmos is the Azure Cosmos DB database service from Microsoft. All four workloads are read-write, and each had data collected from three different drives on industrial machines. All twelve of these traces have been run on SM951 consumer-level SSDs to produce the data that we use to evaluate our system. For all the traces, we simulate failovers by jumping to a different request in the list of requests. We divide the data into training and testing sets deterministically based on the location in the list, where the first 80% of the array is used as the training set, and the remaining 20% is used for testing and validation.

Workload	Threshold	Percentile
AzureStorage 0	500	90
AzureStorage 1	500	72
AzureStorage 2	500	85
BingIndex 2	4849	59
BingIndex 3	4662	90
BingIndex 4	3604	98
BingSelect 2	2058	29
BingSelect 3	3420	69
BingSelect 5	2336	49
Cosmos 1	2852	97
Cosmos 2	2550	81
Cosmos 4	2788	90

Table 4.1: Thresholds

This table lists the failover thresholds we use for most of our experiments, except as noted, as well as what percentile of the training data is equal to or less than the threshold.

Table 4.1 shows the failover thresholds we use for all of our experiments, except as noted, as well as what percentile of the training data is equal to or less than the threshold. These thresholds were chosen by a human in an ad hoc manner based on what worked well in their experience. We find that they are reasonably good, but there is still room for improvement by choosing a good bias value or better threshold.

To create the charts in this section, we used Altair [61], based on Vega-Lite [52]. Data processing to produce the charts was done with pandas [51, 64].

4.2 Choosing the Bias Value and Failover Threshold

Figure 4.2 shows the speedups achievable by choosing the bias value and failover threshold according to the equations described in Section 3.5 and the various algorithms detailed in Section 3.6, as well as an unbiased perfect predictor, relative to an unbiased regressor with thresholds from Table 4.1. There are a few things to notice in this figure:

1. The speedups achievable by biasing the loss function and changing the failover threshold are comparable, with the bias oracle achieving an average speedup of 1.15x and the

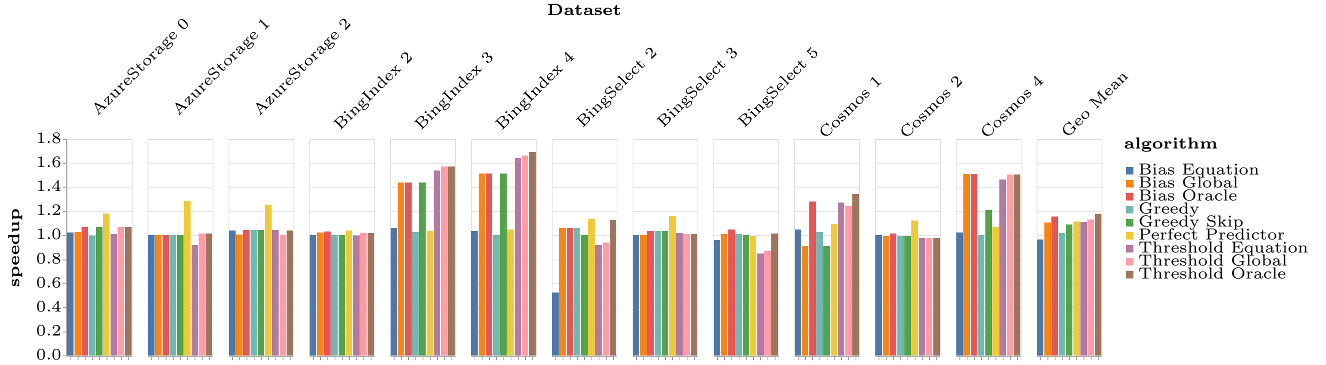


Figure 4.2: Speedup of All Workloads using Regression with 30μs Failover Cost, 3 Replicas, Thresholds from Table 4.1

This figure shows the speedups achievable by choosing the bias value and failover threshold according to the equation as described in Section 3.5 and the various algorithms detailed in Section 3.6

threshold oracle achieving 1.17x, so we can choose either solution based on whichever is easiest to implement.

2. The equation for choosing the bias is effectively useless since it cannot achieve much speedup for any workload, and on average yields 0.96x times the performance of the unbiased regressor.

3. On the other hand, the equation for choosing the threshold works very well in most cases, and achieves 1.10x speedup on average. This is close to the global strategy that averages 1.13x speedup.

4. Although the equation selects a bias value poorly, the "greedy skip" strategy performs close to the global strategy: 1.09x speedup on average using greedy skip, compared to 1.11x for global. This means it is possible to choose a bias value without increasing the latency indefinitely to determine whether an increase in latency is a result of exceeding the optimal value or just noise.

4.3 Effects of Bias on Predictions and Latency

4.3.1 Classification

Next, we examine the effects of bias on the predictions and overall latency for the biased classifier. Figure 4.3 show predictions made on data from AzureStorage on Drive 0 with a failover overhead of 30μs. It has four charts associated with it: the results on the right show what happens when we run the classifier on the training data, while the left shows the testing results. The bottom charts show a breakdown of how many requests are predicted to be false positives, true positives, etc., as well as whether the prediction of each request is the same or different from how the system predicts that same request without bias as the color of the bars. For example, the orange requests labeled "different false positive" indicate that the biased system is predicting a false positive, but the unbiased neural network predicted the same request correctly (as a true negative). The top charts show the sum of the latency of all requests, also broken down to show how much latency is contributed by each of the prediction categories. A bias value of 1 is the baseline with no bias (i.e., both classes have a weight of 1).

Looking at the bottom charts showing the breakdown of positive and negative predictions, as we go to the right and increase the bias amount, the number of false positives in orange increases, and the number of negatives being predicted decreases; both false negatives in green, and more obviously true negatives in purple. From this, we can see that the biased loss function does alter the predictions as expected.

Looking at the latency results, we can see the overall latency sum of all the requests from the height of the stacked bars. There is a fairly clear trend where increasing the bias causes the overall latency to decrease until it reaches a minimum, where it starts increasing again, and may eventually become even worse than no bias at all in both testing and training. This can be seen clearly in Figure 4.3, where the overall latency is minimized at around a bias

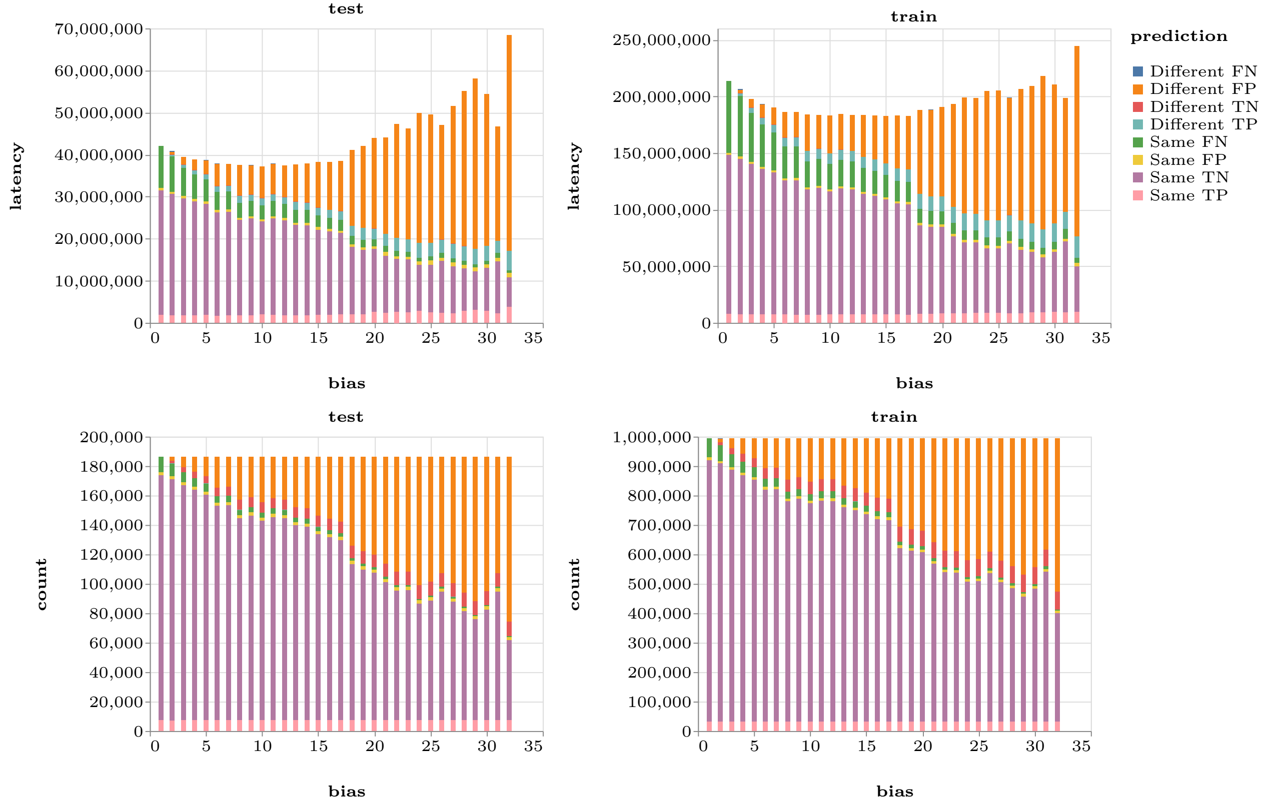


Figure 4.3: Breakdown of Classification Results with a Biased Loss Function and a Small 30μs Failover Overhead for AzureStorage Workload on Drive0 with Threshold from Table 4.1 This figure shows predictions made on data from AzureStorage on Drive0 on with a failover overhead of 30μs. There are four subplots: the results on the right show what happens when we run the classifier on the training data, while the left shows the testing results. The bottom charts show a breakdown of how many requests are predicted to be false positives, true positives, etc., as well as whether the prediction of each request is the same or different from how the system predicts that same request without bias as the color of the bars. For example, the orange requests labeled "different false positive" indicate that the biased system is predicting a false positive, but the unbiased neural network predicted the same request correctly (as a true negative). The top charts show the sum of the latency of all requests, also broken down to show how much latency is contributed by each of the prediction categories.

value of 10, and a bias more than 17 begins to significantly increase the latency. We can see that as we increase the bias value, the latency caused by false positives (in orange) increases, while the latency caused by false negatives (green and blue) and true negatives (purple and red) shrinks, as expected. This makes sense; we can get an improvement by making slow tail requests more likely to failover to a faster replica, but if we have too many requests fail

over, the risk that fast requests will fail over to a slower replica increases, so choosing the right bias value is important.

4.3.2 Regression

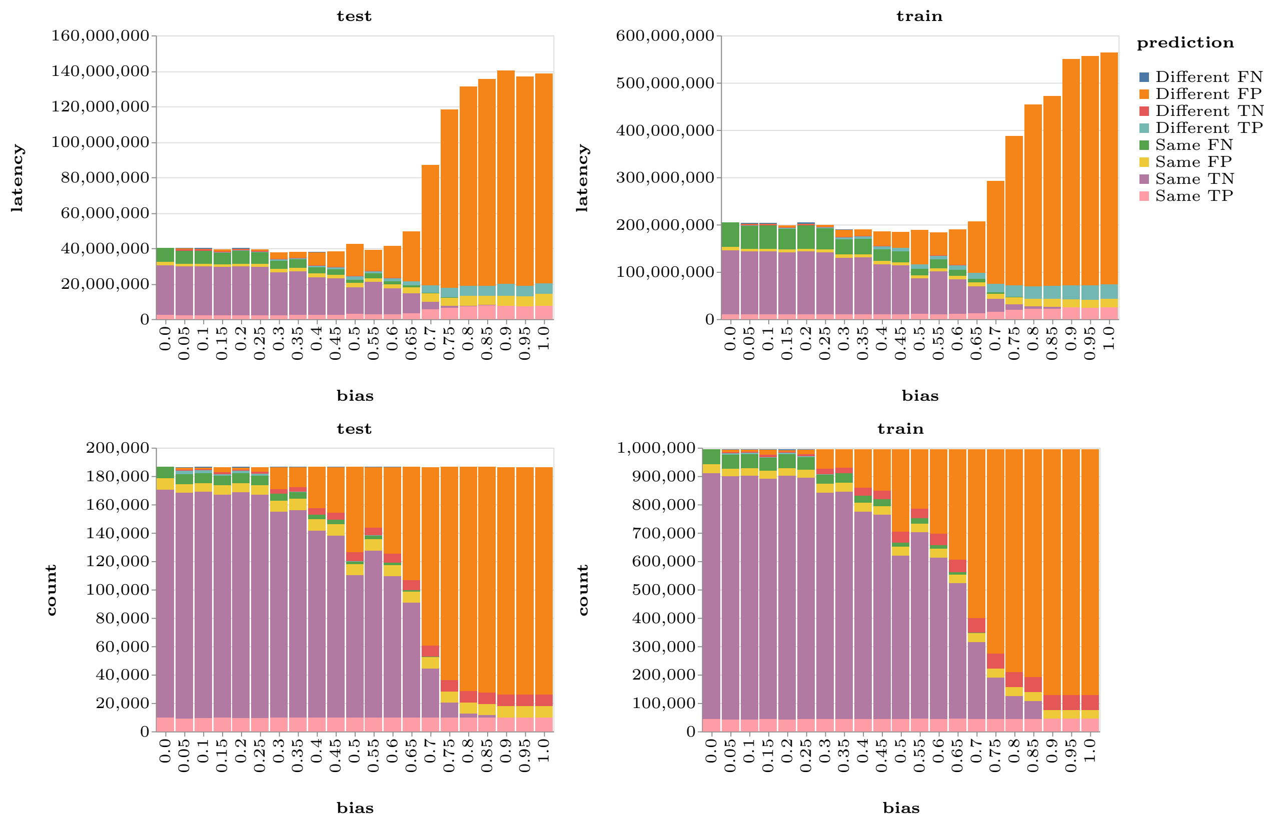


Figure 4.4: Breakdown of Regression Results with a Biased Loss Function and a Small 30μs Failover Overhead for AzureStorage Workload on Drive0 with Threshold from Table 4.1
This set of charts shows results similar to Figure 4.3 but for regression instead of classification.

Figure 4.4 shows similar results for regression. We can see that here, too, increasing the bias value increases the rate of failovers and can decrease the total overall latency sum until it passes the optimal value and starts increasing.

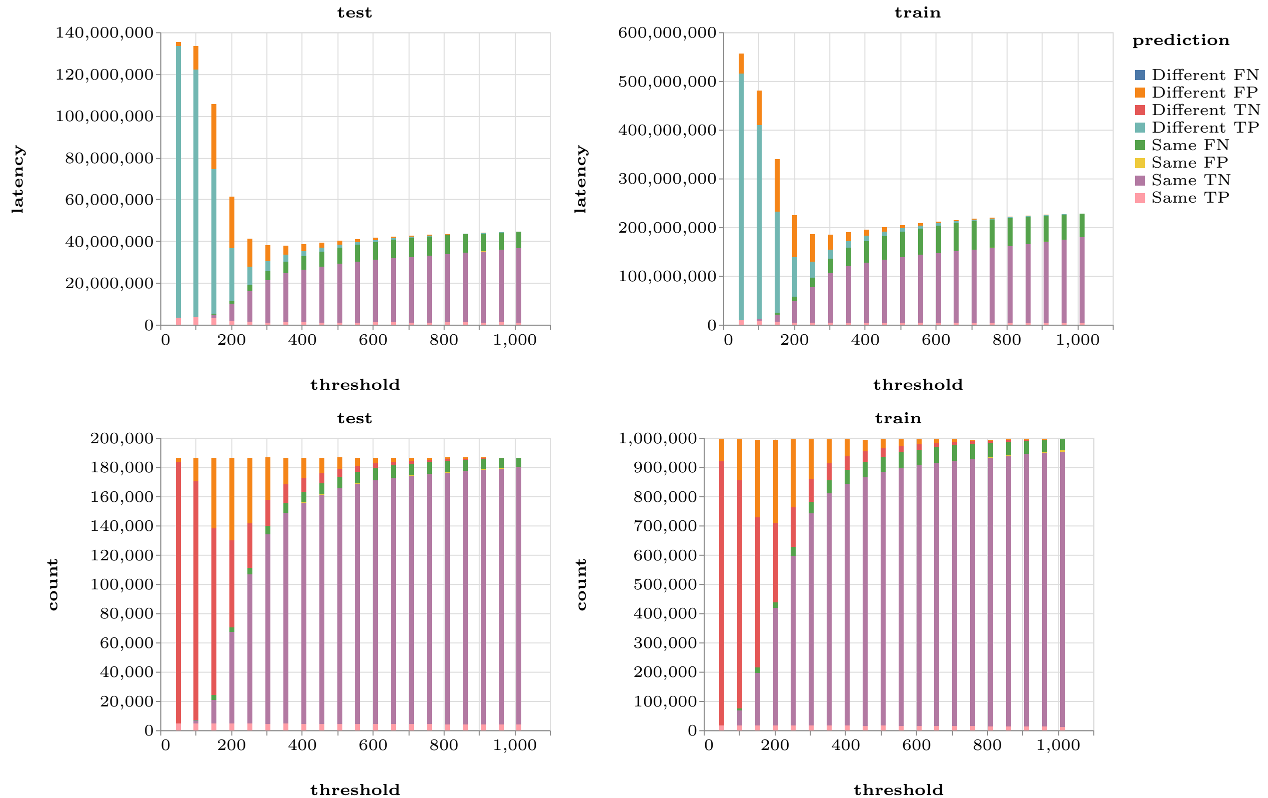


Figure 4.5: Breakdown of Results with a Varied Failover Threshold and a Small 30μs Failover Overhead for AzureStorage Workload on Drive0

This set of charts shows results similar to Figure 4.4 but shows what happens when we vary the failover threshold instead of biasing the loss function.

4.4 Effects of Changing the Failover Threshold

Figure 4.5 shows how changing the failover threshold affects the failover decision and total latency of all requests using an unbiased regressor. For this Azure workload, we were previously using a threshold of 500μs, so it looks like there is room for improvement by choosing a better failover threshold of around 300μs. On the other hand, choosing the wrong threshold can make the latency significantly worse, just like when biasing the loss function. Using a threshold of 100μs would more than triple the total latency sum compared to a threshold of 500μs.

4.5 Classification vs. Regression

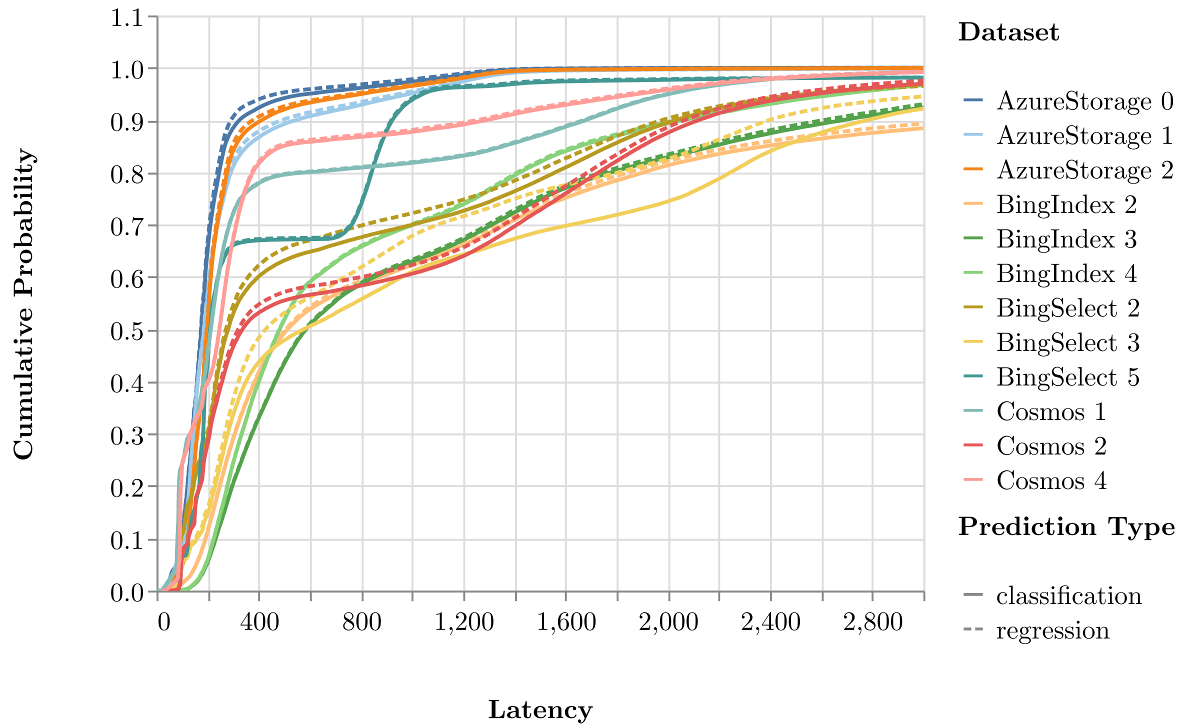


Figure 4.6: Classification vs. Regression
This is a CDF plot of how simulating failovers using the predictions from the unbiased classifier and regressor on all 12 workloads affects the latencies using thresholds from Table 4.1. Notice how close the solid and dashed lines are.

Figure 4.6 shows a CDF plot of how simulating failovers using the predictions from the unbiased classifier and regressor on all 12 workloads affects the latencies. The solid lines show the results we get when running the classifier, and the dashed lines are for regression. The thing to notice here is that the solid and dashed lines of each color are pretty close to each other, meaning that we can achieve similar results from regression as we did for classification. Since both have similar performance, the increased flexibility of regression makes it a better choice in many cases.

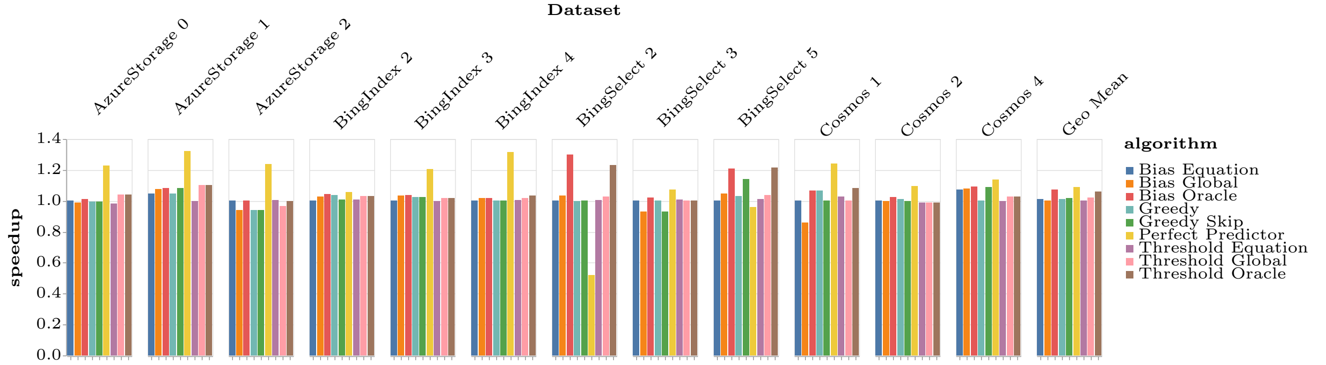


Figure 4.7: Speedup of All Workloads with 30μs Failover Cost, 3 Replicas, Thresholds from Equation

This figure shows speedups like in Figure 4.2, but the baseline for computing the speedups uses failover thresholds chosen by the threshold equation.

4.6 Combining the Bias Value and Failover Threshold

Figure 4.7 shows speedups similar to Figure 4.2, but this time the baseline uses failover thresholds chosen by the threshold equation. We can see that there is little benefit to both biasing the loss function and changing the failover threshold, with nearly all the algorithms achieving speedups very close to 1x for all of the workloads, with the exception of the oracles and perfect predictor. The oracles only achieve notable speedups on the workloads whose training data is not a good predictor of the testing data, like BingSelect on Drive2. The only algorithm that offers any significant speedup when the failover threshold is close to optimal is the unbiased perfect predictor. This makes sense; once we incorporate the relative costs of predicting false positives and false negatives into the threshold, the only way to further improve is to improve prediction accuracy (and to ensure that the training data is representative).

4.7 Effects of Number of Replicas and Failover Overhead

Figure 4.8 shows the geometric mean of speedups across all the workloads that can be obtained with each of the algorithms for various numbers of replicas (which is one more than

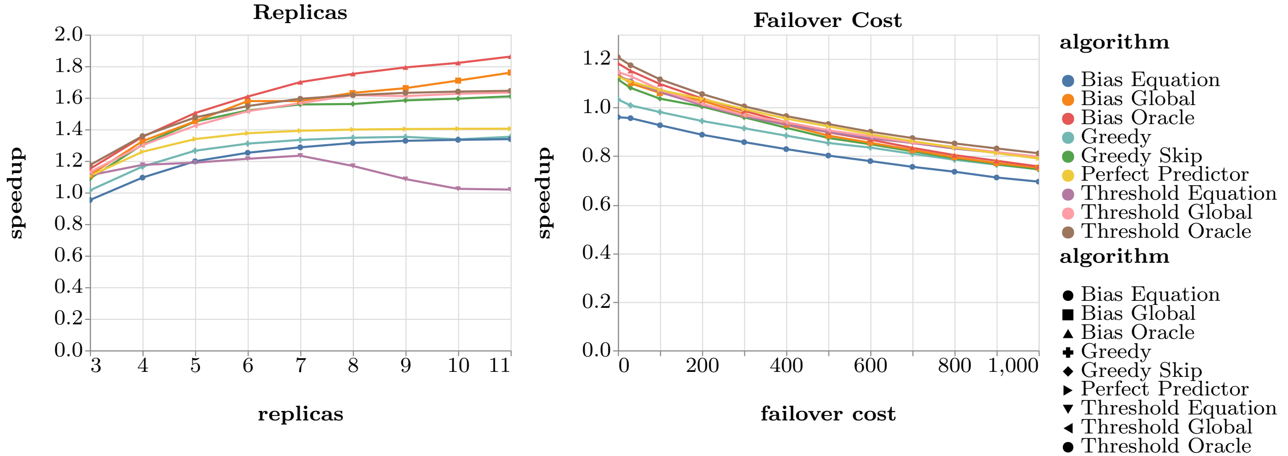


Figure 4.8: Effects of Number of Replicas and Failover Cost
This figure shows the geometric mean of speedups across all the workloads that can be obtained with each of the algorithms for various numbers of replicas and amounts of failover overhead compared to unbiased predictions.

the maximum number of failovers) and amounts of failover overhead compared to unbiased predictions using the thresholds given in Table 4.1. We can see that having more replicas and a lower failover cost generally increases the achievable speedup, as expected, with one exception: The purple line showing what happens when choosing the threshold based on the equation from Section 3.5 is the only algorithm that does worse with more replicas. To explain this, we start by looking at the speedups achieved by using the threshold equation broken down by workload in Figure 4.9.

We can see that several workloads have a positive slope (e.g. BingSelect on Drive 5), indicating that more replicas allows for more opportunities to fail over and speed up, as expected, but several workloads like BingIndex on Drive 4 and Cosmos on Drive 1 have a negative slope and experience slowdowns when there are many replicas. A few workloads show even stranger behavior, such as BingSelect on Drive 3, which shows increasing speedups up to 6 replicas, but with more replicas starts to experience worse performance.

There are two major factors causing the threshold equation to choose poor values when there are many replicas. The first is the differences between the training and testing data.

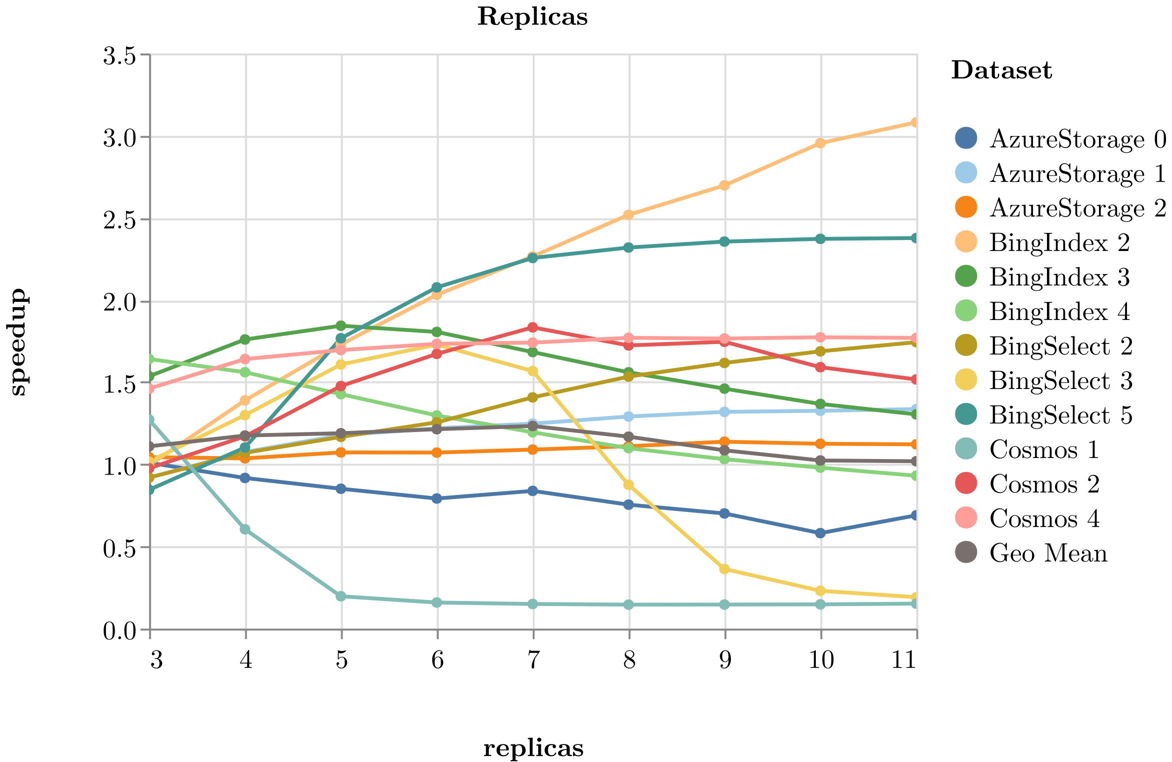
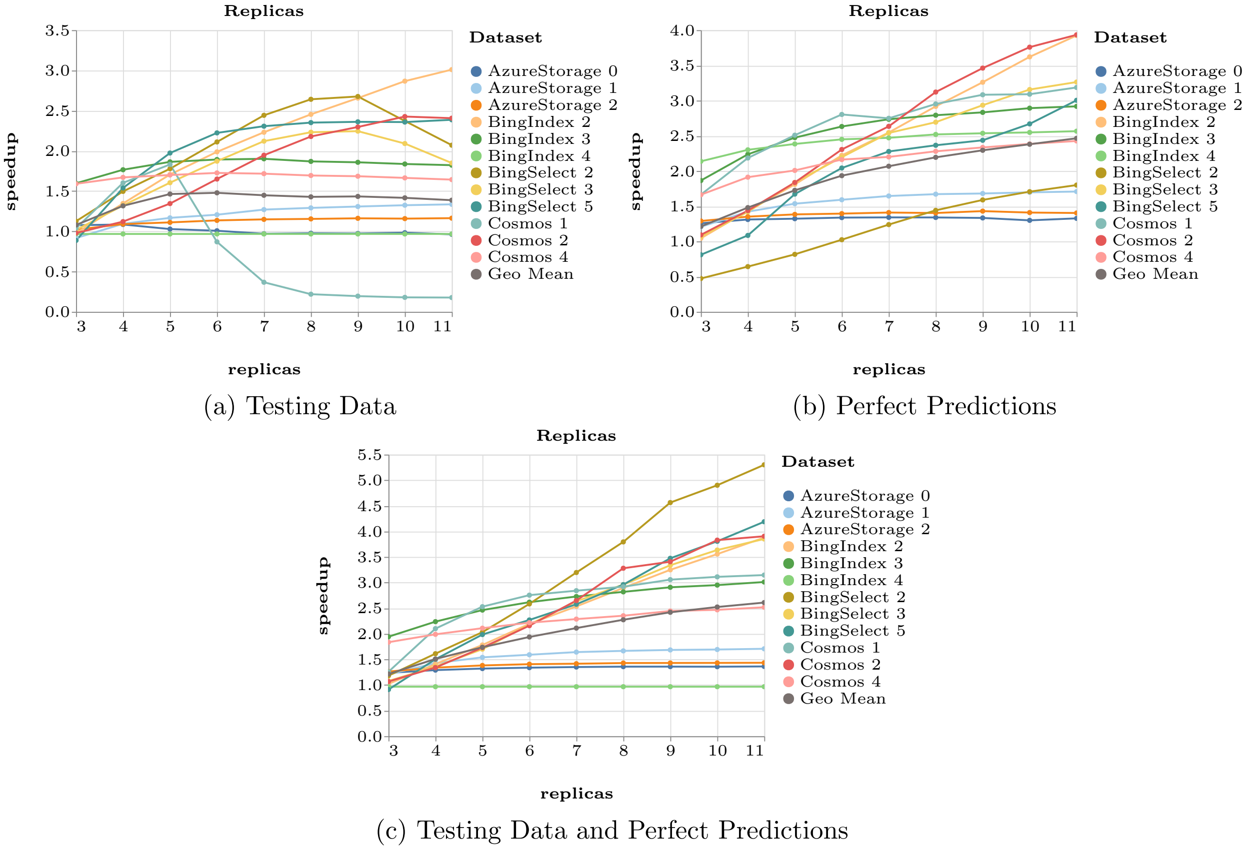


Figure 4.9: Effects of Number of Replicas, Broken Down by Workload, Threshold Equation Only

This figure shows speedups from using the threshold equation for different numbers of replicas in Figure 4.8 broken down by workload. We can see that the geometric mean of the speedups decreasing for larger numbers of replicas can be attributed to particular workloads, like BingIndex on Drive 4 and Cosmos on Drive 1.

In Figure 4.10a, we can see the speedups achievable when choosing the threshold based on the same data we use for testing. Several of the workloads show the same behavior in Figure 4.10a as in Figure 4.9 because the training data is representative of the testing data, such as BingSelect on Drive 5, but other workloads show significant improvements. In fact, the only workload that does not consistently achieve a speedup greater than or equal to 1x in Figure 4.10a is Cosmos on Drive 1, so simply having better training data improves our results significantly. However, there are still several workloads like BingSelect on Drive 2 and BingSelect on Drive 3 that start seeing reduced speedups after 9 replicas (in addition to Cosmos on Drive 1 that experiences reduced speedups when there are more than 5 replicas).



(a) Testing Data

(b) Perfect Predictions

(c) Testing Data and Perfect Predictions

Figure 4.10: How the Accuracy of Predictions and Training Data Affect Performance of the Threshold Equation

This figure shows how choosing the threshold based on the testing data instead of training data (4.10a), having perfect prediction accuracy (4.10b), or both (4.10c) affects the speedups broken down by workload. Compared to Figure 4.9, we see that these scenarios yield significantly better performance with large numbers of replicas.

This brings us to the second reason for reduced speedups: inaccurate predictions. Figure 4.10b shows what would happen if we chose the thresholds using the training data as normal, but we could perfectly predict whether a request will exceed the threshold. We can see that increasing the number of replicas never reduces the speedups achievable for any of the workloads, even Cosmos on Drive 1, which experienced slowdowns on both of the previous graphs. Having more replicas available to fail over to should typically improve the performance; however, it also multiplies the effects of failing over to a slower replica as a result of poor prediction accuracy and causes some workloads to have poor performance in

Figure 4.9 and Figure 4.10a when there are many replicas.

One thing we might notice in Figure 4.10b is that some workloads experience slowdowns (speedups less than 1x) when there are only 3 replicas, namely BingSelect on Drive 2 and BingSelect on Drive 5. To explain this, we can look at Figure 4.10c, which shows the speedups when both choosing the threshold based on the testing data and having perfect prediction accuracy. We see that none of the workloads ever experience less than 1x speedup in Figure 4.10c, suggesting that perfect prediction accuracy alone does not guarantee the best performance; choosing a good threshold based on training data that is representative of the deployment environment is also vital.

4.8 Summary of Results

We started by biasing the loss function of the neural network and found that it successfully changes the rate of EBUSY predictions and can improve latencies similar to the other approach of changing the failover threshold. However, it is difficult to understand how biasing the loss function affects the predictions, and we need to consider the failover threshold anyway, so it may be better to just change the failover threshold and leave the neural network alone. There is little benefit to changing both, and it is much easier to choose a good threshold since we have a formula we can use to do so. A neural network that performs classification or regression can achieve similar performance benefits, but changing the failover threshold on a neural network designed to perform regression is much easier to implement because it does not require retraining.

In general, all of this suggests that performing unbiased regression and changing the failover threshold is likely to be the simplest way to achieve the best results. As we saw in Section 4.7, though, the equation for choosing the threshold is sensitive to the representativeness of the training data and the prediction accuracy when there are many replicas, so there may be scenarios where biasing the loss function may prove to be more useful. This shows

how important it is to consider the entire system when attempting to integrate machine learning, rather than blindly attempting to improve a specific sub-problem, like the prediction accuracy, which may or may not be the most effective way to improve performance. We initially just wanted to make the outputs from the neural network more useful by taking into account certain aspects of the system like the failover threshold, but we can achieve a better solution by considering the system as a whole and deciding how to set all the parameters together, instead of trying to optimize the predictions and the failover threshold separately.

CHAPTER 5

FUTURE WORK

There are a number of things that could be improved with more time. One thing we would like to improve is how we deal with the testing data differing from the training data. As we can see in Figure 4.5, underestimating the failover threshold increases the latency much more drastically than overestimating it. If we suspect the training data may not be representative of the testing data, it seems like adding a buffer to the threshold to reduce the risk of an underestimate may be worthwhile. How much of a buffer is necessary is likely to depend on how confident we are that the training data represents the worse case latencies.

We would also like to try changing the way we set the bias for classification. Instead of leaving the class weight for 0 (not EBUSY) predictions as 1 and setting the class weight for 1 (EBUSY) to a bias value $b > 1$, it might be better to set the weight for 0 to $0 < b < 1$ and leave the class weight for 1 set to 1. By constraining the bias value to be between 0 and 1, we could more easily find a good bias value experimentally with a binary search, instead of having to keep increasing the bias value to a potentially very large number to find the optimal value.

It may also be interesting to try different machine learning models. Because prediction accuracy is not the goal for this problem, it is possible that a different model could achieve similar predictions with bias but be faster to train or have a lower inference time. A different model may also make it easier to understand the effect of bias on the predictions. We were unable to find a satisfactory formula for computing the optimal bias value, but exploring machine learning models that have outputs that are easier to interpret than the neural networks we used could make finding a formula more feasible.

CHAPTER 6

CONCLUSION

Tail tolerance is growing in importance as devices become faster. We started off trying to bias a neural network to predict more false positives and fewer false negatives, but discovered that it is impossible to separate the machine learning problem from the larger systems problem and we must also consider factors like the failover threshold. Not only does changing the failover threshold affect the results of biasing the neural network, it can even be a simpler alternative that offers similar performance benefits. In general, we find that we cannot break down the problem and improve the machine learning predictions separately from tuning other aspects of the system if we want the best results. A holisitic approach and a deep understanding of the system is necessary to providing the best performance while minimizing system complexity.

REFERENCES

[1] CFQ (Complete Fairness Queueing). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[2] Classification on imbalanced data | TensorFlow Core. https://www.tensorflow.org/tutorials/structured_data/imbalanced_data.

[3] Google: Taming the Long Latency Tail - When More Machines Equals Worse Results - High Scalability -. http://highscalability.com/blog/2012/3/12/google-taming-the-long-latency-tail-when-more-machines-equal.html.

[4] LightNVM. http://lightnvm.io/.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[6] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. pages 185–198, 2013.

[7] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '12, pages 91–100, New York, NY, USA, October 2012. Association for Computing Machinery.

[8] Saeid Barati, Ferenc A. Bartha, Swarnendu Biswas, Robert Cartwright, Adam Duracz, Donald Fussell, Henry Hoffmann, Connor Imes, Jason Miller, Nikita Mishra, Arvind, Dung Nguyen, Krishna V. Palem, Yan Pei, Keshav Pingali, Ryuichi Sai, Andrew Wright, Yao-Hsiang Yang, and Sizhuo Zhang. Proteus: Language and Runtime Support for Self-Adaptive Software Development. *IEEE Software*, 36(2):73–82, March 2019. Conference Name: IEEE Software.

[9] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel {SSD} Subsystem. pages 359–374, 2017.

[10] Chi-Ou Chen, Ye-Qi Zhuo, Chao-Chun Yeh, Che-Min Lin, and Shih-Wei Liao. Machine Learning-Based Configuration Parameter Tuning on Hadoop System. In *2015 IEEE International Congress on Big Data*, pages 386–392, June 2015. ISSN: 2379-7703.

[11] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277, February 2011. ISSN: 1530-0897.

[12] Seungryul Choi and D. Yeung. Learning-Based SMT Processor Resource Distribution via Hill-Climbing. In *33rd International Symposium on Computer Architecture (ISCA'06)*, pages 239–251, June 2006. ISSN: 1063-6897.

[13] François Chollet et al. Keras. https://keras.io, 2015.

[14] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack Cap: Adaptive DVFS and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, December 2011.

[15] Adam Crume, Carlos Maltzahn, Lee Ward, Thomas Kroeger, Matthew Curry, and Ron Oldfield. Fourier-assisted Machine Learning of Hard Disk Drive Access Time Models. In *Proceedings of the 8th Parallel Data Storage Workshop*, PDSW '13, pages 45–51, New York, NY, USA, 2013. ACM. event-place: Denver, Colorado.

[16] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, February 2013.

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107, January 2008.

[18] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-Aware Cluster Management. page 17.

[19] Christina Delimitrou and Christos Kozyrakis. QoS-Aware scheduling in heterogeneous datacenters with paragon. *ACM Transactions on Computer Systems*, 31(4):12:1–12:34, December 2013.

[20] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. Memory cocktail therapy: a general learning-based framework to optimize dynamic tradeoffs in NVMs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 232–244, New York, NY, USA, October 2017. Association for Computing Machinery.

[21] Yi Ding, Nikita Mishra, and Henry Hoffmann. Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, pages 39–52, New York, NY, USA, June 2019. Association for Computing Machinery.

[22] Christophe Dubach, Timothy M. Jones, Edwin V. Bonilla, and Michael F.P. O'Boyle. A Predictive Model for Dynamic Microarchitectural Adaptivity Control. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 485–496, December 2010. ISSN: 2379-3155.

[23] Archana Ganapathi, Kaushik Datta, Armando Fox, and David Patterson. A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, page 1, USA, March 2009. USENIX Association.

[24] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Deepthi Srinivasan, Biswaranjan Panda, Andrew Baptist, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. *ACM Trans. Storage*, 14(3):23:1–23:26, October 2018.

[25] Mingzhe Hao. Personal Communication, December 2019.

[26] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 168–183, New York, NY, USA, 2017. ACM. event-place: Shanghai, China.

[27] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and {SSD} Deployments. pages 263–276, 2016.

[28] Robin Harris. The case against SSDs. https://www.zdnet.com/article/the-case-against-ssds/, July 2015.

[29] Emre (https://datascience.stackexchange.com/users/381/emre). machine learning - Linear regression with non-symmetric cost function? https://datascience.stackexchange.com/a/10474, March 2016.

[30] H. Howie Huang, Shan Li, Alex Szalay, and Andreas Terzis. Performance modeling and analysis of flash-based storage devices. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, May 2011. ISSN: 2160-1968.

[31] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. pages 375–390, 2017.

[32] Connor Imes, Steven Hofmeyr, and Henry Hoffmann. Energy-efficient Application Resource Scheduling using Machine Learning Classifiers. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 1–11, New York, NY, USA, August 2018. Association for Computing Machinery.

[33] Wonkyung Kang, Dongkun Shin, and Sungjoo Yoo. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Transactions on Embedded Computing Systems*, 16(5s):134:1–134:20, September 2017.

[34] Jae-Hong Kim, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. A methodology for extracting performance parameters in solid state disks (SSDs). In *2009 IEEE International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, pages 1–10, September 2009. ISSN: 2375-0227.

[35] Jaeho Kim, Donghee Lee, and Sam H. Noh. Towards {SLO} Complying SSDs Through {OPS} Isolation. pages 183–189, 2015.

[36] Jihun Kim, Joonsung Kim, Pyeongsu Park, Jong Kim, and Jangwoo Kim. SSD Performance Modeling Using Bottleneck Analysis. *IEEE Computer Architecture Letters*, 17(1):80–83, January 2018.

[37] Benjamin C. Lee and David Brooks. Applied inference: Case studies in microarchitectural design. *ACM Transactions on Architecture and Code Optimization*, 7(2):8:1–8:37, October 2010.

[38] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ACM SIGOPS Operating Systems Review*, 40(5):185–194, October 2006.

[39] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 249–258, New York, NY, USA, March 2007. Association for Computing Machinery.

[40] Benjamin C. Lee, Jamison Collins, Hong Wang, and David Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 270–281, November 2008. ISSN: 2379-3155.

[41] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 1–14, Seattle, WA, USA, November 2014. Association for Computing Machinery.

[42] Shan Li and H. Howie Huang. Black-Box Performance Modeling for Solid-State Drives. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 391–393, August 2010. ISSN: 1526-7539.

[43] Jose F. Martinez and Engin Ipek. Dynamic Multicore Resource Management: A Machine Learning Approach. *IEEE Micro*, 29(5):8–17, September 2009. Conference Name: IEEE Micro.

[44] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, Cascais, Portugal, October 2011. Association for Computing Machinery.

[45] Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. CALOREE: Learning Control for Predictable Latency and Low Energy. *ACM SIGPLAN Notices*, 53(2):184–198, March 2018.

[46] Nikita Mishra, John D. Lafferty, and Henry Hoffmann. ESP: A Machine Learning Approach to Predicting Application Interference. In *2017 IEEE International Conference on Autonomic Computing (ICAC)*, pages 125–134, July 2017. ISSN: 2474-0756.

[47] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints. *ACM SIGPLAN Notices*, 50(4):267–281, March 2015.

[48] Adam J. Oliner, Anand P. Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 1–14, New York, NY, USA, November 2013. Association for Computing Machinery.

[49] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[50] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, ASPLOS '14, pages 471–484, Salt Lake City, Utah, USA, February 2014. Association for Computing Machinery.

[51] Jeff Reback, Wes McKinney, jbrockmendel, Joris Van den Bossche, Tom Augspurger, Phillip Cloud, gfyoung, Sinhrks, Adam Klein, Matthew Roeschke, Jeff Tratner, Chang She, Simon Hawkins, William Ayd, Terji Petersen, Jeremy Schendel, Andy Hayden, Marc Garcia, MomIsBestFriend, Vytautas Jancauskas, Pietro Battiston, Skipper Seabold, chris b1, h vetinari, Stephan Hoyer, Wouter Overmeire, alimcmaster1, Mortada Mehyar, Kaiqi Dong, and Christopher Whelan. pandas-dev/pandas: Pandas 1.0.1, February 2020.

[52] Arvind Satyanarayan, Dominik Moritz, Kanit Wongsuphasawat, and Jeffrey Heer. Vega-lite: A grammar of interactive graphics. *IEEE transactions on visualization and computer graphics*, 23(1):341–350, 2017.

[53] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. pages 349–362, 2012.

[54] David C. Snowdon, Etienne Le Sueur, Stefan M. Petters, and Gernot Heiser. Koala: a platform for OS-level power management. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 289–302, New York, NY, USA, April 2009. Association for Computing Machinery.

[55] Len Strnad. Asymmetric Loss Functions: How and Why in TensorFlow. http://www.lenstrnad.com/blog/2018/09/asymmetric_loss_function, September 2018.

[56] Riza O. Suminto, Cesar A. Stuardo, Alexandra Clark, Huan Ke, Tanakorn Leesatapornwongsa, Bo Fu, Daniar H. Kurniawan, Vincentius Martin, Maheswara Rao G. Uma, and Haryadi S. Gunawi. PBSE: a robust path-based speculative execution for degraded-network tail tolerance in data-parallel frameworks. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 295–308, Santa Clara, California, September 2017. Association for Computing Machinery.

[57] Gerald Tesauro. Reinforcement Learning in Autonomic Computing: A Manifesto and Case Studies. *IEEE Internet Computing*, 11(1):22–30, January 2007. Conference Name: IEEE Internet Computing.

[58] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, Farminton, Pennsylvania, November 2013. Association for Computing Machinery.

[59] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1009–1024, New York, NY, USA, May 2017. Association for Computing Machinery.

[60] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22, 2011.

[61] Jacob VanderPlas, Brian Granger, Jeffrey Heer, Dominik Moritz, Kanit Wongsuphasawat, Arvind Satyanarayan, Eitan Lees, Ilia Timofeev, Ben Welsh, and Scott Sievert. Altair: Interactive statistical visualizations for python. *Journal of Open Source Software*, 3(32):1057, 2018.

[62] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[63] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G.R. Ganger. Storage device performance prediction with CART models. In *The IEEE Computer Society's*

12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings., pages 588–595, October 2004. ISSN: 1526-7539.

[64] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[65] Weidan Wu and Benjamin C. Lee. Inferred Models for Dynamic and Sparse Hardware-Software Spaces. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–424, December 2012. ISSN: 2379-3155.

[66] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. pages 543–557, 2015.

[67] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Transactions on Storage*, 13(3):22:1–22:26, October 2017.

[68] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level I/O scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 474–489, Monterey, California, October 2015. Association for Computing Machinery.

[69] Nezih Yigitbasi, Theodore L. Willke, Guangdeng Liao, and Dick Epema. Towards Machine Learning-Based Auto-tuning of MapReduce. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 11–20, August 2013. ISSN: 2375-0227.

[70] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 29–42, San Diego, California, December 2008. USENIX Association.

[71] Xiao Zhang, Rongrong Zhong, Sandhya Dwarkadas, and Kai Shen. A Flexible Framework for Throttling-Enabled Multicore Management (TEMM). In *2012 41st International Conference on Parallel Processing*, pages 389–398, Pittsburgh, PA, USA, September 2012. IEEE.

[72] Yuhao Zhu and Vijay Janapa Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, February 2013. ISSN: 1530-0897.