

# Accurate Generation of I/O Workloads Using Generative Adversarial Networks

Heyu Zhang<sup>1</sup>, Zhen Yang<sup>1</sup>, Yulai Xie<sup>1,\*</sup>, Yafeng Wu<sup>1</sup>, Jiakun Li<sup>1</sup>, Dan Feng<sup>1</sup>, Avani Wildani<sup>2</sup>, Darrell Long<sup>3</sup>

<sup>1</sup> Huazhong University of Science & Technology, Wuhan, China

<sup>2</sup> Emory University, Atlanta, GA, USA

<sup>3</sup> University of California, Santa Cruz, CA, USA

\* Corresponding Author

**Abstract**—It is essential to utilize a large number of I/O workloads to analyze commodity system performance or simulate scientific phenomena in high-performance scientific computing. I/O traces are often unavailable at scale due to trace storage overhead, privacy concerns, and the performance impact of trace instrumentation. We study how to generate sufficiently representative I/O workloads using Generative Adversarial Networks (GANs). The best GAN architecture can generate I/O workloads with maximum mean discrepancy (MMD) as low as 0.015–0.05, which implies the synthetic I/O workloads have successfully learned the potential distribution of real I/O traces. We demonstrate that the performance similarity between the original I/O trace and the generated I/O workload through trace replay can be 90.36%–97.32%.

**Index Terms**—I/O trace, Generative Adversarial Networks, Long Short-Term Memory, workload generation

## I. INTRODUCTION

An I/O trace can be regarded as a kind of time series showing the actual workload and performance of the system for a period of time [1]. I/O traces collected from most parallel systems can be analyzed to find the workload characteristics or root cause of the inefficiency of I/O performance. In addition, I/O trace analysis is critical for reacting to failures and security events as well as projecting future storage infrastructure needs [2]. Suppose we develop a high performance computing (HPC) system and cannot directly test it with I/O traces from the production environment due to privacy and security requirements, we need to generate I/O traces similar to those in the real environment for testing. However, the issue with mathematically derived traces is that modeling a complex, noisy, dynamic real-life environment with a simple set of distributions for solving generic storage provisioning [3] is a challenging set of problems.

Generative Adversarial Networks (GANs) [4] have achieved excellent results for data generation in many fields [5–7]. Using GAN to generate I/O workloads has the following three advantages: 1) GANs excel at constructing spatially correlated matrices and thus are pre-optimized for reconstructing windowed I/O traces. 2) GAN architectures optimized for time-series analysis can learn the temporal relationships in the I/O traces. 3) GANs can be layered, allowing for a hierarchical series of networks to represent an I/O trace at several levels of granularity.

However, applying GAN to generating the I/O workloads faces two major challenges. On the one hand, in order to design and implement GAN architectures suitable for I/O workload generation, we need to find the neural network that is suitable for processing sequence data. On the other hand, determining whether the output of a GAN is representative of the original workload is non-obvious. We have to present a metric between workloads that is experimentally verified to represent the workload similarity with respect to performance impact within a small range.

We first analyze the meaning and numerical range of different fields in a variety of I/O traces with different formats and use a preprocessing algorithm to process I/O traces into vectors that contain real-valued sequence data. Secondly, we analyze the generator and discriminator of the GAN architecture for sequence data generation. Then, we select the neural network that can be used to generate sequence data such as I/O workload, and decide to use Long Short-Term Memory (LSTM) [8] as generator or discriminator, CNN as the discriminator, cross-entropy function or Wasserstein distance [9] as loss function, and then combine them into four GAN architectures. Next, we evaluate the generation effect by using maximum mean discrepancy (MMD) [10] and I/O trace replay technology. We conduct experiments by using different GAN architectures on four kinds of traces and get the best architecture by analyzing the I/O workload generation effect. Afterward, we make the hyperparameter sensitivity analysis to figure out how each hyperparameter affects the generation effect. At last, we analyze the overheads when using the GAN technology to generate I/O traces.

Our contributions are as follows:

1. We convert I/O traces into vectors acceptable to the neural network and use both the MMD and I/O replay to measure the similarity between the generated data and the real data.
2. We experimentally determine that the best GAN architecture for I/O generation is a recurrent (LSTM) network, and the best loss function is cross-entropy loss.
3. Using the developed GAN with LSTM and cross-entropy loss function, we experimentally demonstrate that we can generate I/O traces with up to 90.36% – 97.32% similarity to source traces. Moreover, the I/O traces generated by the

developed GAN have the similar cache hit rate to the source traces in downstream tasks, indicating its effectiveness in practice.

## II. BACKGROUND AND MOTIVATION

### A. I/O trace acquisition technology

Methods of acquiring I/O workload are mainly divided into two types: I/O workload collection technology and I/O workload synthesis technology.

I/O trace collection methods can be divided into application, file, network, and block levels. Among them, the file-level and block-level acquisition methods are more commonly used.

For file-level traces (e.g., strace [11], ScalaIOTrace [12] and Tracefs [13]), the system call interface is generally used for acquisition. Still, this method has poor portability and generally cannot collect traces across platforms. For example, ScalaIOTrace [12] intercepts traces of communication events called by MPI-IO; Tracefs [13] captures traces at different granularities without changing the target file system.

For block-level traces, Blktrace [14] collects disk I/O information after reading and writing to the block device layer. Iosnoop [15] traces disk I/O with details by reading the event data from ftrace [16] and processing them in the user space. However, collecting and storing traces at scale using these tools can consume sizeable time and space overheads and impact the system's performance.

### B. I/O workload synthesis methods

I/O workload synthesis methods typically generate a trace through a pre-established model simulation [17], trace feature generation [18] or workload integration [19]. For example, Filebench [17] can simulate the workload of an actual application to establish a load model and also supports users to customize load models according to WML load model language to generate trace datasets. Mathematical calculation methods mainly include TraceRAR [20] and ScalaIOExtrap [21]. TraceRAR calculates the linear correlation between the two metrics (e.g., the fraction of R/W, the internal arrival time, the disk name, the locality, the sequential rate and the request size). If the correlation between the metrics is strong, a linear equation between the two metrics is established for prediction; otherwise, the interval probability of a single metric is calculated separately to generate a trace. ScalaIOExtrap first collects a set of scalable lossless I/O traces in a relatively small number of nodes through ScalaIOTrace [12]. Then, it analyzes the trace datasets according to the different formats of different files, establishes specific equations to calculate the I/O trace of multi-node and large-scale systems, and uses I/O replay to verify the accuracy of the I/O trace.

These methods focus more on the comprehensive correlation of various indicators to analyze the overall trace characteristics but do not adequately explore the contextual and temporal relationships required to analyze the dynamic patterns of individual metrics over different periods. As a result, they are not able to accurately generate large-scale records.

TABLE I  
AN EXAMPLE OF SPC TRACE FILE CONTENT

| ASU | LBA    | Size(b) | Opcode(r/w) | Timestamp(s) |
|-----|--------|---------|-------------|--------------|
| 1   | 55596  | 3072    | w           | 0.117964     |
| 0   | 303588 | 3584    | w           | 0.530841     |
| 0   | 303595 | 3584    | w           | 0.550502     |

### C. Advantages of Generative Adversarial Network

Generative Adversarial Networks (GANs) [4] can address the above issues by learning from the characteristics of the existing data to generate large-scale data. GAN consists of two neural networks: a generator and a discriminator. The goal of the generator is to learn the intrinsic distribution of the input values and then simulate the distribution to generate fake data. The goal of the discriminator is to estimate the probability that the input value is real or fake. A loss function is used to calculate the error between the generated sample and the real sample, and then the error is backpropagated to guide the network parameter update. For GAN, the generator and discriminator can be represented by any different neural network. Due to many neural networks, determining the generator, discriminator, and loss function to generate I/O traces accurately involves examining a large search space.

In contrast to file-level or block-level trace generation methods, GANs do not depend on any specific file system or platform. Compared to previous I/O workload synthesis methods or mathematical calculation methods, GANs excel at constructing spatially correlated matrices and allow for a hierarchical series of networks to be pre-optimized for reconstructing windowed I/O traces at several levels of granularity, thus improving the applicability, compatibility, fault tolerance and efficiency of mathematical calculation methods. The characteristics of I/O traces are complex and diverse. In actual situations, GANs contain a certain degree of randomness and diversity in adversarial training methods, which can be more consistent with real I/O workloads.

Compared with large language models (LLMs), GANs only require a small number of real samples for training and can generate large-scale data sets. However, LLMs require powerful computing devices to support their complex model structures and massive parameter processing, resulting in a very high cost for generating traces.

## III. DESIGN AND IMPLEMENTATION

We first describe the preprocessing method for I/O traces and then elaborate the details of GAN architecture design.

### A. I/O trace analysis and preprocessing

Due to the various hardware platforms, operating systems, and data collection technologies, I/O traces have a variety of formats. As SPC trace file format [22] has good readability and portability, we take it as an example to analyze. Each row in the trace file represents an I/O record, as shown in Table I.

*Application Specific Unit (ASU)* indicates a disk drive where the I/O was sent. The first record in the trace file does not need ASU equal to zero. However, unit zero must exist within the

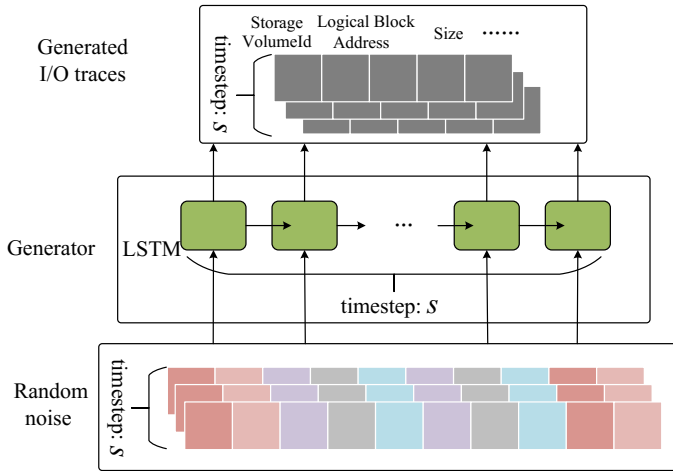


Fig. 1. The LSTM generator in GAN

trace file. If  $n$  units are recorded in a trace file, the trace file must contain at least one record for each unit from 0 to  $n - 1$ .

*Logical Block Address (LBA)* is a positive integer that describes the ASU block offset of data transmission for this record. The value of this offset can be between 0 and  $n - 1$ , where  $n$  is the capacity of the ASU block.

Similarly, for other traces, some fields have a similar meaning as the fields above. For instance, the storage volume ID in K5cloud trace [23] and the Lun index in Systor trace [24] used in the experiments of this paper are similar to ASU and thus are preprocessed in the same way. For the fields that are different from the above fields (e.g., the Response time in Systor trace), we normalize them to  $[-1, 1]$ . Specifically, for the strings (e.g., username), we map them to numbers using a dictionary encoding method and then scale them to  $[-1, 1]$ .

### B. Design and implementation of GAN architecture

A GAN is a bicameral data structure composed of a generator network, originally seeded with random noise, which suggests a candidate I/O matrix, and a discriminator network trained on real I/O matrices that determines whether the candidate matrix is “real enough” or not.

**(1) Generator.** We choose Long-Short Term Memory (LSTM) networks for both the generator and discriminator, as LSTM is suited to modeling time series data and is well-established across various domains [25, 26]. The main challenges when using LSTM as the generator are: 1) selecting the proper random noise dimension in the input layer to fit the real data distribution, and 2) accurately converting the output vector to match the I/O trace size.

Since I/O traces typically have 5-8 columns, the random noise dimension should be slightly larger. For instance, selecting a dimension of 10 ensures diversity in the generated data without excessively increasing training time or memory consumption.

As shown in Figure 1, for LSTM layer and output layer, we first initialize LSTM, then input random noise into LSTM for training, and finally convert the dimension of the output vector to the dimension of the I/O trace by multiplying the

output value by the weight. In addition, the vector must be added with a bias to make the synthetic trace more accurate. For example, the vector dimension of the SPC trace needs to be converted to 5. Then, the output value is activated with the  $\tanh$  function. The training steps are as follows: in the first LSTM unit, the input data is random noise  $z$ , and it generates the first record of the I/O trace  $t^1$ . In the  $n$ -th unit, the input data is random noise  $z$  and the previous record  $t^{(n-1)}$ , and the output is  $n$ -th record  $t^n$ . Finally, the output is  $s$  records,  $T = \{t^1, t^2, \dots, t^s\}$ , where  $s$  is equal to the timestep of the LSTM, which is the length of the time series processed by the LSTM, and the dimension of  $t$  is the number of columns of I/O traces through the conversion of the output layer. In addition, in the output layer, we need to binarize the opcode field to ensure the accuracy of the column.

The specific training process is shown in Algorithm 1. We first set the appropriate hyperparameters and generate noise (i.e., random number) using standard normal distribution. In the LSTM layer, we initialize the LSTM cell (i.e., a basic unit in LSTM to process and transfer information), input noise to train the LSTM cell, and get the output value  $c\_output$ . The training process mainly updates the weight and bias to fit the data distribution. In the output layer, we first initialize the weight and bias using standard normal distribution. Then, we use weight and bias to convert the dimension of the  $c\_output$  into the specified dimension. At last, we activate  $c\_output$  to normalize its value into  $[-1, 1]$  and output the synthetic traces.

#### Algorithm 1 Training process for Generator

---

**Input:** Noise,  $G\_rounds$   
 /// Noise is generated using standard normal distribution.  
 ///  $G\_rounds$  is the times that generator updates parameters in one epoch, explained in Section IV-C.

**Output:**  $g\_output$   
 ///  $g\_output$  is a vector with a numerical range of  $(-1, 1)$ .

- 1: **for**  $g$  in  $0 : G\_rounds$  **do**
- 2:   Generate Noise using the standard normal distribution
- 3:   Initialize  $lstm\_cell$
- 4:   Train  $lstm\_cell$  using Noise, and get the output value  $c\_output$
- 5:   Initialize  $weight$  and  $bias$  using standard normal distribution
- 6:    $c\_output = weight * c\_output + bias$ ;
- 7:   Get  $g\_output$  using activation function  $\tanh$  to activate  $c\_output$
- 8: **end for**

---

**(2) Discriminator.** The discriminator distinguishes the generated data from the real data. As CNN and LSTM have both acted as discriminators in the GAN for sequence data generation [27, 28], and have a good ability of binary classification, we evaluate both CNN and LSTM as potential discriminators to distinguish the authenticity of I/O traces.

As shown in Figure 2, the LSTM discriminator consists of two layers: an LSTM layer and an output layer, which echoes the architecture of the generator network. For LSTM layer,

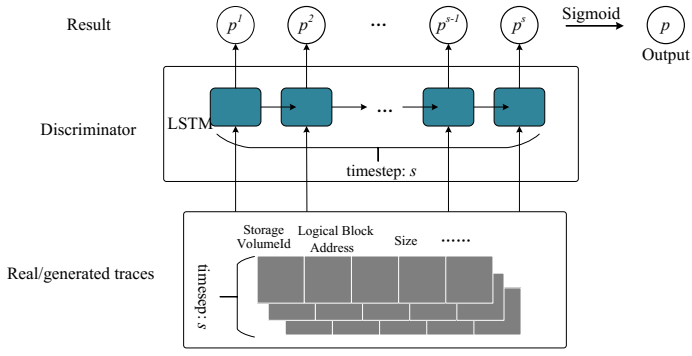


Fig. 2. The LSTM discriminator in GAN

because the timestep of the LSTM determines the length of the time series that the LSTM can process, and the vector received by the discriminator is generated by the LSTM generator, the timestep of the LSTM discriminator should be consistent with that of the LSTM generator. Suppose we input I/O traces as  $T = \{t^1, t^2, \dots, t^s\}$ . In the first LSTM cell, the input is  $t^1$ . The output is the probability that  $t^1$  is the real I/O trace and we denote it as  $p^1$ . In the  $n$ -th cell, the inputs are  $t^n$  and the probability value of the previous unit  $p^{(n-1)}$ . The output is  $p^n$ . Finally, the output of LSTM layer is a set of probability values indicating whether the input I/O traces are the real I/O traces, and we denote them as  $P = \{p^1, p^2, \dots, p^s\}$ . For the output layer, we multiply  $P$  by the vector of the specified size, convert it to a probability value, and then map the value to a probability  $p$  within the range of  $(0, 1)$  through the Sigmoid function. Thus  $p$  is the probability value that the input data are the real I/O traces.

#### Algorithm 2 Training process for CNN Discriminator

**Input:**  $d\_xs$ ,  $D\_rounds$   
 ///  $d\_xs$  is the generated data or real data.  
 ///  $D\_rounds$  is the times that discriminator updates parameters in one epoch.  
**Output:**  $p$   
 ///  $p$  is the probability of input values as real I/O traces.

- 1: **for**  $d$  in  $0 : D\_rounds$  **do**
- 2: Initialize convolution kernel  $conv\_weight$  using normal distribution
- 3: Train discriminator using  $d\_xs$
- 4: Get output value  $d\_conv\_out$  of convolution layer by using activation function  $leaky\_relu$  to activate the sum of the convolution result and the bias of the convolution operation
- 5: Initialize  $weight$  and  $bias$  using normal distribution
- 6:  $p = Sigmoid(weight * d\_conv\_out + bias)$   
 /// Sigmoid is a function to generate a probability value between 0 and 1.
- 7: **end for**

We train the CNN discriminator for the specified number of rounds in each training epoch as shown in Algorithm 2. In each round of training, we input the synthetic I/O traces

and real I/O traces into the convolution layer first and then output a probability value through the fully connected layer and output layer. As we scale I/O records to  $[-1, 1]$ , the output of activation function  $relu$  will always be zero. This will make the neural network unable to update parameters and stop learning. While,  $leaky\_relu$  solves this problem by introducing a small non-zero slope for the negative input, so that even if the input is negative, the neuron can still be activated and participate in the gradient update. Thus we use the activation function  $leaky\_relu$  to activate the sum of the convolution result and the bias of the convolution operation. Note that the output of the LSTM generator is a sequence of vectors whose dimensions are the number of columns. These data can be divided into a series of batches. The number of data items of each batch can be represented as  $timestep * number\ of\ I/O\ trace\ columns$ , which can be taken as the  $height * width$  of a matrix that can be fed into the CNN discriminator input to perform convolution.

**(3) Selection of loss function.** In the training process of GAN, the discriminator is trained to distinguish between real and generated data, and then the generator is trained to cheat the discriminator. This process continues until it reaches a state of equilibrium, when the discriminator can't distinguish the real data from the generated data, but the generator can generate enough realistic data. The loss function is a key factor in defining the adversarial dynamics between the generator and discriminator.

Considering I/O traces are discrete sequence data, the cross-entropy function performs well when processing sequence data [28], so we first choose the cross-entropy function as the loss function. In addition, as the Wasserstein distance loss function [9] can effectively solve the problem of unstable training of discrete data, we also choose Wasserstein distance as the potential loss function.

The binary cross-entropy loss function returns a value that ranges from 0 to 1. If I/O traces generated by the generator are good enough, so that the discriminator cannot distinguish whether the input value is real I/O traces or synthetic traces, then the discriminator loss function value will reach 0.5. The loss functions of the generator and discriminator are calculated as shown in Formulas (1) and (2):

$$G = -\log[D(G(z))], \quad (1)$$

$$D = -\log[D(x)] - \log[1 - D(G(z))], \quad (2)$$

where  $z$  is the input noise variable,  $G$  is the generator,  $D$  is the discriminator,  $G(z)$  is synthetic I/O traces, and  $D(x)$  is the probability of input data as real I/O traces or synthetic I/O traces. The cross-entropy loss function in GAN is to measure the distance between the generated traces and realistic traces. By minimizing this distance, GAN can learn to generate data highly similar to the real data. The minimax optimization in GAN is shown in Formula (3).

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))], \quad (3)$$

TABLE II  
THE COLUMN MEANINGS OF FOUR I/O TRACES

| Column | OLTP Trace                | K5cloud Trace          | Systor Trace          | WebSearch Trace           |
|--------|---------------------------|------------------------|-----------------------|---------------------------|
| 1      | Application Specific Unit | Storage VolumeId       | Timestamp             | Application Specific Unit |
| 2      | Logical Block Address     | Timestamp              | Response Time         | Logical Block Address     |
| 3      | Size, multiple of 512     | Opcode, 'W' or 'R'     | Opcode, 'W' or 'R'    | Size, multiple of 512     |
| 4      | Opcode, 'w' or 'r'        | Offset                 | LUN index             | Opcode, 'w' or 'r'        |
| 5      | Timestamp                 | Size, multiple of 1024 | Offset                | Timestamp                 |
| 6      | None                      | None                   | Size, multiple of 512 | None                      |

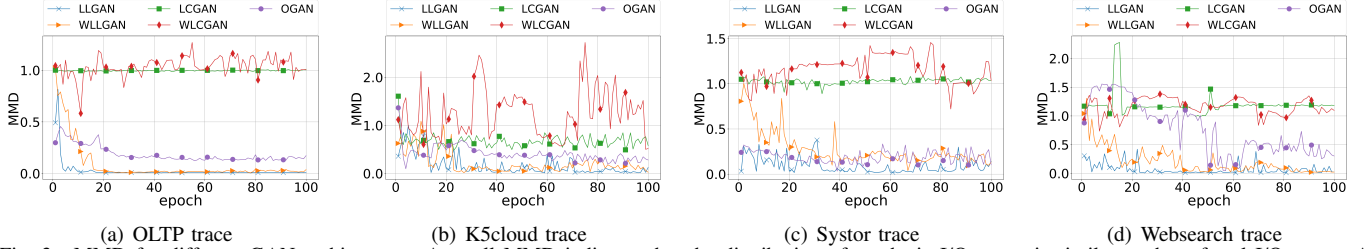


Fig. 3. MMD for different GAN architectures. A small MMD indicates that the distribution of synthetic I/O traces is similar to that of real I/O traces. A fluctuating MMD indicates that the training process is unstable. Epoch is a hyperparameter that refers to sending all data sets into the neural network to complete a forward calculation and backpropagation. The MMD of LLGAN is the smallest among all the GAN frameworks when the epoch becomes bigger.

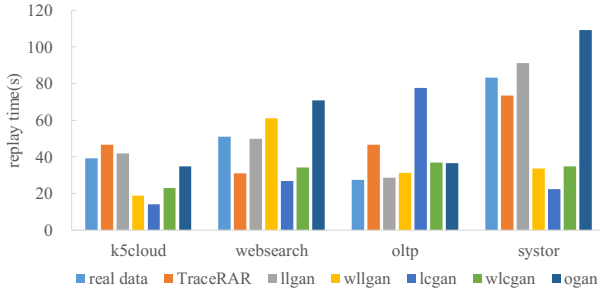


Fig. 4. Total replay time for four I/O traces

where  $V(D, G)$  is the function to be optimized,  $p_{\text{data}}$  is the probability distribution of the real data, and  $p_z$  is the probability score of the input noise of the generator.

Wasserstein distance is used by WGAN [9] as a loss function. WGAN makes four changes in the loss function of the original GAN [4]: 1) the discriminator does not use Sigmoid function as an activation function; 2) the loss values of the generator and discriminator do not take logarithm; 3) the updated discriminator parameter value is truncated to the specified range; and 4) the momentum-based optimizer is not used.

#### IV. EXPERIMENTAL EVALUATION

We first describe the testbed and datasets. Then we use Maximum Mean Discrepancy (MMD) and I/O trace replay to evaluate the designed architectures. Next, we make hyperparameter sensitivity analysis. At last, we measure the overhead of using GAN to generate I/O traces.

##### A. Experimental setup

The host installs a CentOS Linux release 8.2.2004 operating system, with four Intel(R) Core(TM) i7-6700 CPU@3.40 GHz, a 2 GB NVIDIA Quadro K620 GPU, a 16 GB DDR4 2400 MHz RAM, and a 256 GB Samsung 850 PRO SSD.

We use four different I/O trace datasets and the meanings of columns for each dataset are shown in Table II.

TABLE III  
THE GAN ARCHITECTURES

| Framework | Generator | Discriminator | Loss function |
|-----------|-----------|---------------|---------------|
| LLGAN     | LSTM      | LSTM          | Cross-entropy |
| WLLGAN    | LSTM      | LSTM          | Wasserstein   |
| LCGAN     | LSTM      | CNN           | Cross-entropy |
| WLCGAN    | LSTM      | CNN           | Wasserstein   |
| OGAN      | FC        | FC            | Cross-entropy |

The OLTP dataset [22] is the I/O trace collected from On-Line Transaction Processing (OLTP) system run by two large financial institutions. The K5cloud dataset [23] is the I/O trace collected from Fujitsu K5cloud cloud service based on OpenStack platform. The Systor dataset [24] is the I/O trace collected from Enterprise Virtual Desktop Infrastructure (VDI). The WebSearch dataset [29] is the I/O trace collected from a popular search engine. Because our goal is to generate a large number of traces using a small-scale trace, we use the first 60,000 trace records from a file of each dataset to generate the data. As the original K5cloud dataset only contains 800 thousand I/O records, we generate 800 thousand I/O records for comparison. For all other traces, we generate 1 million I/O records. According to the data preprocessing in Section III-A, we preprocess the four datasets into the vectors acceptable to the GAN architecture, and then split them into training sets and verification sets.

##### B. I/O Trace Generation

**(1) MMD.** Maximum Mean Discrepancy (MMD) [10] is widely used to measure similarity between data distributions and time series datasets. Lower MMD values indicate greater similarity between synthetic and real I/O traces, reflecting successful GAN architecture.

According to the analysis of the choices of the generator, discriminator, and loss function in Section III-B, we compare the generation effect of the five architectures as shown in Table III, using the original GAN (OGAN) [4] as a baseline.

TABLE IV  
READ/WRITE RATIO DURING REPLAY.

|                 | real        | LLGAN       | TraceRAR    | LCGAN | OGAN        | WLCGAN      | WLLGAN      |
|-----------------|-------------|-------------|-------------|-------|-------------|-------------|-------------|
| OLTP (R/W)      | 0.59 / 0.41 | 0.59 / 0.41 | 0.59 / 0.41 | 0 / 1 | 0.37 / 0.63 | 0.72 / 0.28 | 0.44 / 0.56 |
| K5Cloud (R/W)   | 0.75 / 0.25 | 0.75 / 0.25 | 0.75 / 0.25 | 1 / 0 | 0.53 / 0.47 | 1.00 / 0.00 | 0.98 / 0.02 |
| Systor (R/W)    | 0.67 / 0.33 | 0.67 / 0.33 | 0.67 / 0.33 | 0 / 1 | 0.71 / 0.29 | 0.87 / 0.13 | 0.98 / 0.02 |
| Websearch (R/W) | 1.00 / 0.00 | 1 / 0       | 1 / 0       | 1 / 0 | 1 / 0       | 1 / 0       | 1.00 / 0.00 |

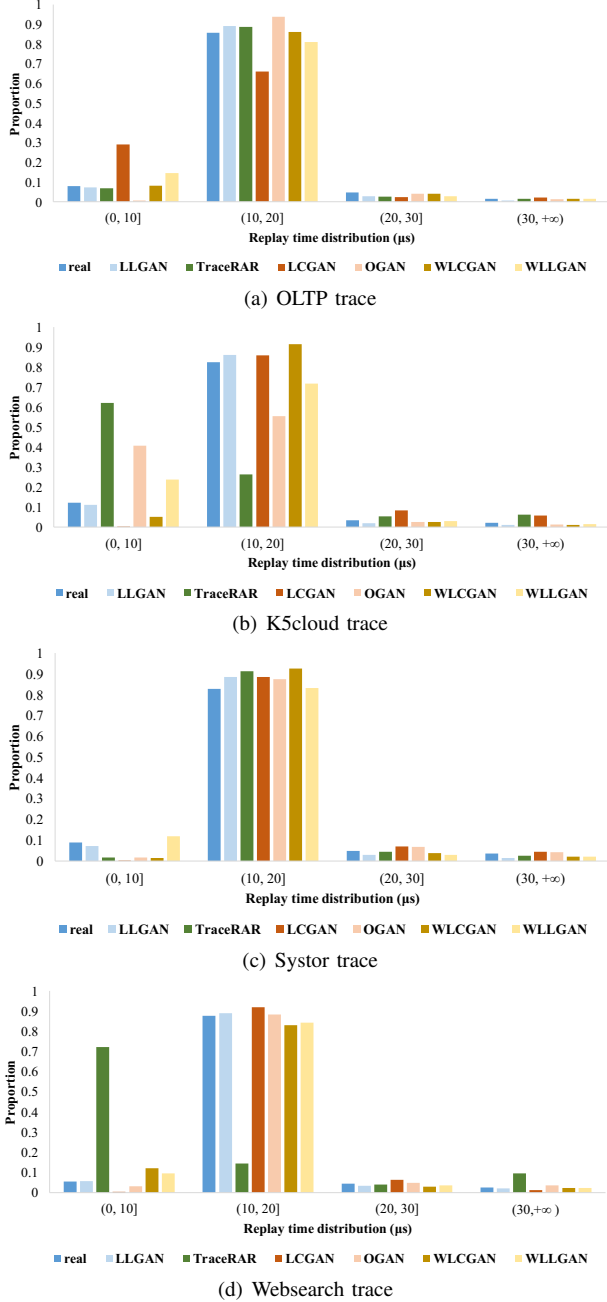
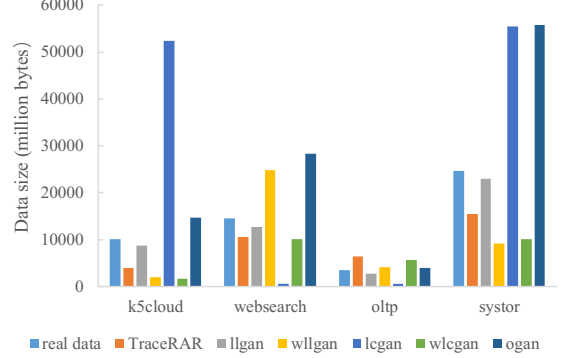


Fig. 5. Replay time distribution for different I/O trace generation methods.

The experimental MMDs between the verification sets of real I/O traces and the synthetic I/O traces are shown in Figure 3. After training, the MMD of LLGAN stabilizes at 0.015 – 0.05, showing the best performance on the four datasets. This indicates LLGAN can exploit the inherent time sequence information in the I/O workload to guide the opti-



mization of the generator and the discriminator, thus improving the accuracy of synthetic I/O traces. WLLGAN performs similarly to LLGAN but requires longer training due to weight parameter truncation and slowing gradient decline. LCGAN and WLCGAN perform worse than the baseline across all I/O traces because the CNN discriminator inefficiencies in extracting time sequence features for LSTM generator training prevent mutual inhibition and common optimization.

**(2) I/O trace replay.** For I/O trace replay, we compare the I/O performance between the synthetic traces and the real traces which include both training sets and verification sets to evaluate the accuracy of the synthetic traces. We use the breplay [30] tool to assess the similarity between generated and real samples in evaluating GAN performance.

**Replay Time.** Figure 4 shows the total replay time for each I/O trace. The replay time accuracy of LLGAN is 90.36%–97.32%, which is significantly higher than the replay time accuracy of other methods including TraceRAR. Figure 5 further shows the distribution of replay time. LLGAN still has comparable or better replay accuracy than TraceRAR for all the datasets in different replay time areas.

Take WebSearch trace as an example. The trace is collected in search engine applications. There are obvious differences in the number of visits to such applications in different time periods. When the scale of the trace is large, it contains more time periods, so that the trace may show different characteristics in different time periods. TraceRAR only uses mathematical methods such as correlation coefficients to analyze the overall characteristics of the metrics, and cannot explore different rules for different time periods, which affects the generation effect. LLGAN uses LSTM for training, and can predict the current time record through the characteristics of the previous time record. It can learn the characteristics displayed at different time periods, so as to avoid the limitations of mathematical methods such as correlation coefficient used in TraceRAR.

TABLE V  
IOPS ACQUIRED VIA REPLAYING DIFFERENT DATASETS.

| GAN      | OLTP    |           |       | K5cloud |           |        | Systor  |           |        | WebSearch |           |       |
|----------|---------|-----------|-------|---------|-----------|--------|---------|-----------|--------|-----------|-----------|-------|
|          | Real    | Generated | Diff  | Real    | Generated | Diff   | Real    | Generated | Diff   | Real      | Generated | Diff  |
| LCGAN    | 36469.7 | 12878.3   | 64.7% | 20371.8 | 57020.7   | 179.9% | 12019.2 | 44583.1   | 270.9% | 19558.0   | 37509.4   | 91.8% |
| WLCGAN   | 36469.7 | 27092.9   | 25.7% | 20371.8 | 34843.2   | 71.0%  | 12019.2 | 29360.0   | 139.4% | 19558.0   | 29360.0   | 50.1% |
| OGAN     | 36469.7 | 27262.8   | 25.3% | 20371.8 | 23054.8   | 13.2%  | 12019.2 | 9147.5    | 23.89% | 19558.0   | 14148.3   | 27.7% |
| WLLGAN   | 36469.7 | 31918.3   | 12.5% | 20371.8 | 42238.6   | 107.3% | 12019.2 | 29673.6   | 146.9% | 19558.0   | 16361.3   | 16.3% |
| LLGAN    | 36469.7 | 35014.0   | 3.99% | 20371.8 | 19175.5   | 5.87%  | 12019.2 | 10962.5   | 8.79%  | 19558.0   | 20096.5   | 2.75% |
| TraceRAR | 36469.7 | 21514.6   | 41.0% | 20371.8 | 17226.5   | 15.44% | 12019.2 | 13635.1   | 13.44% | 19558.0   | 32383.4   | 65.6% |

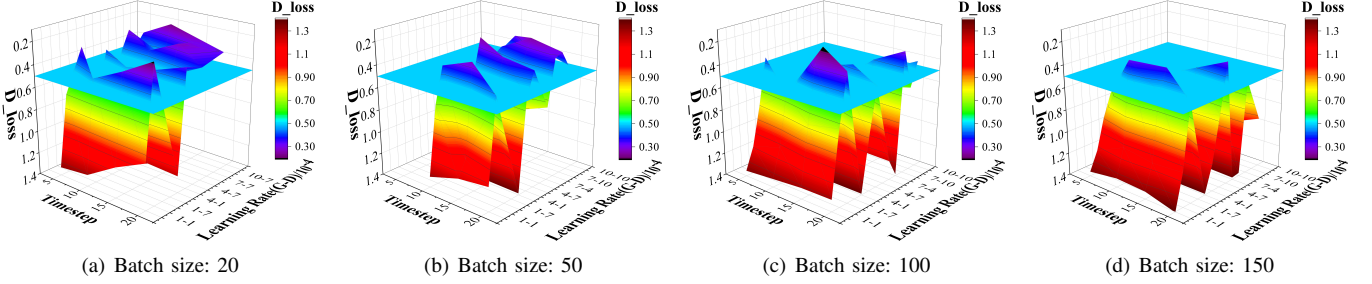


Fig. 7. Multiple parameter analysis in different batch sizes.  $G$  represents the generator learning rate and  $D$  represents the discriminator learning rate.

**Read and Write Ratio.** As shown in Table IV, the read/write ratio of LLGAN and TraceRAR is consistent and close to that of the original data. For LLGAN, it learns the read/write ratio of the whole original data and applies it to the generated trace. TraceRAR also considers the read-and-write ratio of raw data. It analyzes and sends the probability histogram of read-write probability to the Trace generator. All other GAN architectures are trained according to the read/write ratio in each batch of the original data. As different GAN architectures have different batches of data, when data is finally generated, different read-write ratios will appear in different GAN generation results.

**IOPS and Data Size.** Table V shows the IOPS of the real trace and the generated trace using btreplay, respectively. The error of using LLGAN to generate traces is as low as 2.75% – 8.79%. Similarly, as shown in Figure 6, LLGAN outperforms all other methods by accurately analyzing the size of the data read and written in each I/O trace.

LLGAN performs the best as it fully exploits the time sequence characteristics in the I/O traces. TraceRAR focuses on record relationships, failing to model the data size precisely. WLLGAN has the problems of difficulty in training and slow convergence speed because WLLGAN uses weight clipping to calculate the Wasserstein distance, which will cause the parameters of the discriminator to concentrate on the maximum and minimum values, thus converting the discriminator into a binary neural network that cannot give play to the fitting ability of a deep neural network. In addition, the architecture with CNN as the discriminator cannot give full play to local feature extraction for generated trace. Thus WLCGAN and LCGAN perform worse than LLGAN.

### C. Hyperparameter sensitivity analysis

**Number of hidden layers.** We adjust the number of hidden layers of generator and discriminator to see how it impacts the generation effect. We use “LSTMxy” to indicate x generator hidden layers and y discriminator hidden layers. During train-

ing time, the MMDs of LSTM11 and LSTM12 decline steadily and finally stabilize at a similar value, while the MMDs of LSTM22, LSTM44, and LSTM88 eventually stabilize at a larger value. This may be due to over-fitting. In summary, more hidden layers require more computing resources, but cannot achieve smaller MMD, so the optimal number of hidden layers for both the generator and discriminator is chosen to be one.

**Training times for generator and discriminator.** As the dataset usually has multiple columns of features, the generator has to output many eigenvalues, while the discriminator only outputs one eigenvalue. So the discriminator is initially trained once and the generator is trained several times. When the number of training times of the generator is equal to or greater than 3, MMD is stable and there is not much difference. More training times can result in additional calculations and overfitting risks, so the training time we choose is 3.

**Multiple parameters analysis.** Figure 7 shows the trend of discriminator loss function value with the change of four kinds of hyperparameters during OLTP trace training, including batch size, timestep, the learning rate of generator and the learning rate of discriminator. This figure can help us find the optimal hyperparameter range. In each subfigure, the x-axis is the timestep; y-axis is the combination of generator learning rate ( $G$ ) and discriminator learning rate ( $D$ ), and the learning rate increases in step of 0.0003; and z-axis is the loss function value of discriminator when training 50 epochs.

When the loss function value of the discriminator is stable at 0.5, it indicates that the generator and the discriminator have reached the Nash equilibrium state. So we draw a plane with  $D\_loss = 0.5$ . Each intersection point of the plane and the loss function graph corresponds to a good hyperparameter combination. When the batch size is greater than 100, there are fewer parts above the plane, which indicates that most of the loss function values of the discriminator are large and it is difficult to reduce to 0.5. When the batch size is too small, there are more parts above the plane, indicating that

TABLE VI

CACHE HIT RATE OF I/O TRACE GENERATED BY MODELS. HIT\_RATE\_O, HIT\_RATE\_LLGAN AND HIT\_RATE\_TRACERAR REPRESENT THE CACHE HIT RATE OF THE TRACE DATA IN THE ORIGINAL DATASET, THE CACHE HIT RATE OF THE TRACE DATA GENERATED BY THE GENERATED ADVERSARIAL NETWORK AND THE CACHE HIT RATE OF THE TRACE GENERATED BY TRACERAR, RESPECTIVELY.

|           | hit_rate_O | hit_rate_LLGAN | hit_rate_TraceRAR |
|-----------|------------|----------------|-------------------|
| K5_Cloud  | 80.8%      | 85.8%          | 99.9%             |
| OLTP      | 41.9%      | 43.1%          | 99.9%             |
| Systor    | 77.8%      | 77.5%          | 99.9%             |
| WebSearch | 0.00%      | 0.00%          | 99.9%             |

the loss function value of the discriminator drops too fast and lasts a long time. Therefore, the optimal batch size range is [50, 100]. When the batch size is 100, only when the timestep is in [10, 15], the loss function value of the discriminator can be 0.5. Thus the optimal range of timestep is in [10, 15].

#### D. Overhead analysis

For training, the time overheads of five GANs range from 20 to 50 seconds. OGAN uses FC as both the generator and discriminator so the GPU utilization is much smaller than the other four cases. Conversely, for the replay phase, the GPU utilization of five GANs is remarkably lower (1% – 5%) because it only requires simpler operations to generate data using the trained model parameters. Furthermore, the GPU memory consumption during both the training and replay phases is consistently high, occupying approximately 75.8% to 80.9% of the total memory. This high consumption is due to the necessity of storing intermediate calculation results and parameters, including weights and biases, throughout the process.

#### V. APPLICATION CASE

Our method is to use a small part of the I/O trace to generate a large number of I/O traces. So there may be similar or repetitive I/O records in the synthetic trace that can be used for cache. To further verify the effectiveness of the generated I/O trace, we measure the cache hit rate using LLGAN and TraceRAR as shown in Table VI.

The hit rate of the generated I/O trace using LLGAN is not much different from the original dataset, with an error from 0% – 6.19%. For K5\_Cloud and Systor, after sorting the generated data by timestamp, the distance between the block addresses of several consecutive trace records is not large, but the address offset has a large change. The address offset determines the size of the data in the cache. Therefore, one access with a large address offset can make subsequent accesses probably achieve cache hit. So the hit rate could be around 80%. For OLTP, when observing the generated trace data, it can be found that there is a certain distance between the block addresses of different trace records. The data required for different trace records when accessing the cache may not be in it, and the probability of a cache miss is high. As a result, the cache hit rate is lower than 50%. For WebSearch, the dataset was collected from a popular search engine application, and there is little correlation between different trace records.

Therefore, the cache hit rate of the original dataset is 0%, and the data distribution is easy to learn. After the generated trace records are sorted by timestamp, the block address difference between the previous and subsequent data is quite large, and cache hits are rarely achieved when actually accessing the cache, so the hit rate is basically 0%. Comparatively, the hit rate of data generated by TraceRAR is close to 100%. The reason is that only part of the original data is generated, and the synthetic trace data are repeatedly stacked by the generated data.

In total, the trace data generated by our method can be more accurate than the data generated using the traditional TraceRAR method in cache hit rate. Therefore, this further demonstrates that the trace generated by our approach is highly similar to the original trace.

#### VI. DISCUSSION

For I/O trace preprocessing, we vectorize the I/O trace and refer to the CGAN [31] framework to label data by dimensions such as process or time range. This transforms the unsupervised model into a supervised one, supporting the generation of specific trace data.

The generator's output trace value range and feature dimensions are set according to the input trace, allowing customization without being limited by input dimensions. MMD gives equal weight to every trace field, but some fields, like r/w and LBA, are more important for specific storage types [32]. Thus, more weight should be given to r/w fields in these cases.

Our method can be used for cache, and trace correlations might aid prefetching, but further analysis is needed. Generating many traces could cause wear leveling and lock contention issues, as multiple write records in the same block may increase wear pressure and require lock contention mechanisms.

Currently, our method reproduces existing trace characteristics. Future work may modify the generator to produce workloads with different characteristics, such as scaling SPC traces to larger disks. Additionally, to protect application privacy, we can anonymize fields with sensitive information using a one-way secure hash function with key [33] or apply differential privacy by adding random noise. By enhancing these aspects, we aim to improve the accuracy and applicability of our trace generation method.

#### VII. CONCLUSION

Accurately generating I/O traces has been a great challenge. We design and implement four GANs to generate I/O traces, including LLGAN, WLLGAN, LCGAN, and WLCGAN. We find that for the generator, considering that the number of eigenvalues of I/O traces is small, the number of LSTM layers should not be too large; for discriminators, as I/O traces have only a few eigenvalues and the values are relatively discrete, we do not recommend using CNN for feature extraction; in the training process, the convergence effect is better when the number of training for the generator is slightly greater than that of the discriminator. We use both MMD and I/O trace replay to measure the accuracy of the synthetic I/O traces.

The results show that the proposed LLGAN can generate I/O traces with the accuracy of 90.36% – 97.32%, achieving the best results among the four GANs. Therefore, LLGAN is the preferred method for generating I/O traces.

#### ACKNOWLEDGMENTS

This work is funded by the National Key Research and Development Program (No.2022YFB4501300)

#### REFERENCES

- [1] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley, “24/7 characterization of petascale I/O workloads,” in *Proc. of the Workshop on Interfaces and Abstractions for Scientific Data Storage*, 2009.
- [2] T. Patel, S. Byna, G. K. Lockwood, and D. Tiwari, “Revisiting i/o behavior in large-scale storage systems: the expected and the unexpected,” in *Proc. of International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 65:1–65:13.
- [3] L. Sun, S. Gong, T. Zhang, F. Jiang, Z. Zhao, J. Chen, and X. Zhang, “SUFS: A generic storage usage forecasting service through adaptive ensemble learning,” in *Proc. of 39th IEEE International Conference on Data Engineering*, 2023, pp. 3168–3181.
- [4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, “Generative adversarial nets,” in *Proc. of Advances in neural information processing systems*, 2014.
- [5] K. Neela and V. Kavitha, “Blockchain based chaotic deep gan encryption scheme for securing medical images in a cloud environment,” *Appl. Intell.*, pp. 4733–4747, 2023.
- [6] S. Gupta, A. Keshari, and S. Das, “Rv-gan: Recurrent gan for unconditional video generation,” in *Proc. of IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, 2022, pp. 2023–2032.
- [7] Y. Zhang, Z. Gan, K. Fan, Z. Chen, R. Henao, D. Shen, and L. Carin, “Adversarial feature matching for text generation,” in *Proc. of the 34th International Conference on Machine Learning*, 2017, pp. 4006–4015.
- [8] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, pp. 1735–1780, 1997.
- [9] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein GAN,” *arXiv:1701.0787*, 2017.
- [10] A. Gretton, K. M. Borgwardt, M. J. Rasch, B. Schölkopf, and A. J. Smola, “A kernel method for the two-sample problem,” in *Proc. of Advances in neural information processing systems*, 2006, pp. 513–520.
- [11] “strace,” <https://github.com/strace/strace>.
- [12] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth, “Scalable I/O tracing and analysis,” in *Workshop on Petascale Data Storage*, 2009.
- [13] A. A. W. C. P. and Z. E., “Tracefs: A file system to trace them all,” in *Proc. of USENIX conference on File and Storage Technologies*, 2004, pp. 129–145.
- [14] A. D. Brunelle, “Block I/O layer tracing: blktrace,” *HP, Gelato-Cupertino, CA, USA*, 2006.
- [15] “Iosnoop tool,” <https://github.com/brendangregg/perf-tools/blob/master/iosnoop>.
- [16] “ftrace,” <https://kernel.org/doc/Documentation/trace>.
- [17] “Filebench tool,” <https://github.com/filebench/filebench>.
- [18] A. K. Paul, J. Y. Choi, A. M. Karimi, and F. Wang, “Machine learning assisted HPC workload trace generation for leadership scale storage systems,” in *Proc. of International Symposium on High Performance Distributed Computing*, 2022, pp. 199–212.
- [19] J. Bhimani, A. Maruf, N. Mi, R. Pandurangan, and V. Balakrishnan, “Auto-tuning parameters for emerging multi-stream flash-based storage drives through new I/O pattern generations,” *IEEE Trans. Comput.*, pp. 309–322, 2022.
- [20] B. Li, F. Toussi, C. Anderson, D. J. Lilja, and D. H. Du, “TraceRAR: An I/O performance evaluation tool for replaying, analyzing, and regenerating traces,” in *Proc. of International Conference on Networking, Architecture, and Storage*, 2017, pp. 1–10.
- [21] X. Luo, F. Mueller, P. H. Carns, J. Jenkins, R. Latham, R. B. Ross, and S. Snyder, “ScalaIOExtrap: Elastic I/O tracing and extrapolation,” in *Proc. of IEEE International Parallel and Distributed Processing Symposium*, 2017.
- [22] “SPC Trace dataset,” <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- [23] “K5cloud Trace,” <http://iota.snia.org/traces/26663>.
- [24] “Systor’17 Trace,” <http://iota.snia.org/traces/4928>.
- [25] X. Han, X. Yu, T. Pasquier, D. Li, and H. Chen, “SIGL: Securing software installations through deep graph learning,” in *Proc. of USENIX Security*, 2021.
- [26] P.-J. Hoedt, F. Kratzert, D. Klotz, C. Halmich, M. Holzleitner, G. S. Nearing, S. Hochreiter, and G. Klambauer, “MC-LSTM: mass-conserving LSTM,” in *Proc. of 38th International Conference on Machine Learning*, 2021.
- [27] L. Yu, W. Zhang, J. Wang, and Y. Yu, “SeqGAN: Sequence generative adversarial nets with policy gradient,” in *Proc. of AAAI Conference on Artificial Intelligence*, 2017, pp. 2852–2858.
- [28] C. Esteban, S. L. Hyland, and G. Rätsch, “Real-valued (medical) time series generation with recurrent conditional gans,” *CoRR*, p. abs/1706.02633, 2017.
- [29] “WEBSEARCH Trace dataset,” <http://skulldata.cs.umass.edu/traces/storage/WebSearch1.spc.bz2>.
- [30] “Btrecord and btplay user guide,” <http://www.fis.unipr.it/doc/blktrace-1.0.1/btplay.pdf>.
- [31] M. Mirza and S. Osindero, “Conditional generative adversarial nets,” *Comput. Sci.*, pp. 2672–2680, 2014.
- [32] S. Jaffer, S. Maneas, A. Hwang, and B. Schroeder, “Evaluating file system reliability on solid state drives,” in *Proc. of USENIX Annual Technical Conference*, 2019.
- [33] A. G. Miklas, S. Saroiu, A. Wolman, and A. D. Brown, “Bunker: A privacy-oriented platform for network tracing,” in *Proc. of USENIX Symposium on Networked Systems Design and Implementation*, 2009, pp. 29–42.