Journal of Parallel and Distributed Computing 133 (2019) 63–76

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Preserving SSD lifetime in deep learning applications with delta snapshots

Zhu Wang [a], Jalil Boukhobza [b], Zili Shao [c,*]

[a] Department of Computing, Hong Kong Polytechnic University, Hong Kong, China
[b] Univ Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
[c] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China

HIGHLIGHTS

- Exploits the similarity between snapshots to reduce the overall written data volume.
- Two schemes with different snapshot size and disk space are proposed.
- The method is implemented in DiskSim with SSD extension to evaluate its performance.
- Experiments showed that it can achieve smaller snapshot size than other methods.

ARTICLE INFO

Article history:
Received 12 December 2018
Received in revised form 17 April 2019
Accepted 18 June 2019
Available online 25 June 2019

Keywords:
Solid state disk
Deep learning
Snapshots

ABSTRACT

In large-scale deep learning applications, SSDs (Solid State Drives) have been widely adopted to speed up the training. However, a snapshot process is periodically performed in a deep learning application, by which a great number of training parameters (at the TB level) need to be written to SSDs; thus, it poses serious challenges for the lifetime of SSDs. In this paper, we for the first time propose a mechanism, called delta snapshot, that can effectively reduce the overall amount of data written to an SSD during the training phase so as to preserve its lifetime. Delta snapshot exploits the redundant information between snapshots. Specifically, we observe that the exponent part and the most significant bits of the mantissa change very little between two consecutive snapshots. Based on this, we develop effective mechanism to compress the redundant bits of snapshots to reduce the size of the written data. Experimental results showed that our technique can reduce the overall amount of written data by 31% and the erase operations by 27%, with a negligible time overhead in the training phase.

1. Introduction

We are witnessing the era of big data, with exponential increases in the amounts of data being generated. For big data applications, it is necessary to design systems providing low-latency services and real-time data analytics (e.g., real-time deep learning applications [2], advertising, and social gaming [47]). Existing HDD-based storage systems cannot achieve a timely response because of the high access latency. SSDs have been widely adopted to improve the access latency and energy efficiency of storage systems. For instance, due to the high level of parallellism exposed in GPUs, the used storage device should present a high I/O efficiency, to feed the processing elements with data [33].

Deep learning has received a great deal of attention in both academia and industry. These applications rely on a training phase that runs in a very long period to allow some weight values to be computed iteratively. A large training model may generate a huge number of intermediate training parameters, which can be as large as 10^{12} [28], achieving TBs of data. Therefore, if any system failures occur during this phase, the whole process will need to be restarted. To prevent against such losses of data, intermediate training parameters are checkpointed to a file (namely the snapshot) stored in the storage system. This is the case in most common frameworks like Caffe [23], TensorFlow [1], and Project Adam [13]. As a consequence, huge amounts of data are saved periodically to the SSD-based storage system. This may cause severe lifetime degradation. Therefore, it is critical to examine this issue.

In existing studies [31], attempts have been made to address a similar issue. Their objective has been to update only those parameters of a snapshot that has been modified, in order to reduce the writing overhead on a traditional disk-based storage system. Unfortunately, this solution does not work in the case of flash memory based storage systems. Due to its out-of-place updating feature [6], in an SSD, a whole page needs to be updated on the flash memory even when a single variable is updated. In addition,

* Corresponding author.
E-mail address: zilishao@cuhk.edu.hk (Z. Shao).

https://doi.org/10.1016/j.jpdc.2019.06.011

Flash memory blocks obey the erase-before-write rule, meaning that a given block of data can only be updated if it is first erased. Generally, a flash memory block contains some valid data that need to be copied elsewhere before the erase operation can happen. This generates a so-called write amplification phenomenon. Write amplification is provoked when the number of device level write operations is greater than the number of application write operations. Due to these flash memory characteristics, solutions based on fine-grained partial updates of the snapshot are not effective on SSD-based storage systems.

Attempts have been made in other studies to tackle this problem for other applications such as large-scale simulations by using incremental checkpoints [15,34,45], in which, however, the rel[illegible]e was on a coarse-grained update optimization. In [illegible]t, blocks of data that were not modified were not written to the new snapshot. As we will show later in this paper, in deep learning frameworks most variables are frequently updated, which leads to very poor efficiency on the part of these coarse-grained approaches. In scientific data compression area, some solutions [4,32,38] proposed to compress single numbers. However, these methods are also coarse-grained. In high performance computing systems, Run Length Encoding (RLE) [5] is a well-known technique that exploits the similarity of data produced closely in time or location for compression sake. However, it does not consider the increasing similarity between adjacent snapshots when training goes to converge, and it also does not compress the first baseline snapshot. The experimental results showed that our method archived [illegible]er written volume than it.

In this paper, we propose a fine-grained method that can effectively reduce the volume of written data to the flash-based storage system. Our method, called **delta snapshot**, is based on the property of the convergence of the weight parameters in deep learning. We observed that parameters slowly and continuously converge towards the searched values. Based on this property, we observed that for floating point variables [1,13,23], the exponent [illegible] and the most significant bits of the mantissa change very little between two consecutive snapshots. As a consequence, we designed a technique that makes it possible to compress the redundant bits of snapshots to reduce the size of the written data. To the best of our knowledge, this is the first work to consider this observation.

In delta snapshot, we store a baseline snapshot. Then, we only update the snapshot by writing the delta (the updated [illegible] the previous one in another file. In addition, we also compressed the baseline snapshot. As a consequence, delta snapshot stores more data on SSD (baseline + delta snapshots) than other snapshot schemes, but it decreases the overall write load for the checkpointing process, thus preserving the lifetime of the SSD. We implemented the delta snapshot method in the DiskSim simulator with an SSD extension. Then we compared its performance with two compressing solutions GZIP and RLE [5]. We found that the delta snapshot scheme can reduce the total snapshot size by [illegible] with a negligible overhead on the time spent in saving and restoring the snapshot.

The main contributions of this paper are as follows:

- We propose a new delta snapshot mechanism for SSDs that deduplicates the common bits in a floating point numbers of different snapshots in order to reduce the overall volume of data written during the training phase of deep learning applications.
- We propose a method that can seamlessly and fully integrate the new mechanism into a training framework. This method defines how to store the snapshots, how to recover from failures, and how to delete the snapshots. We implemented the proposed snapshot mechanism in DiskSim and evaluated its efficiency.

The paper is organized as follows: Section 2 presents the background to the study and our motivation for conducting it. Section 3 describes the main idea behind the design of the delta snapshot. Section 4 illustrated the program interfaces of the delta snapshot. Section 5 gives an evaluation of the delta snapshot mechanism. Section 6 summarizes state-of-the-art work and Section 7 concludes the paper.

2. Background and motivation

In this section, we first present some basic knowledge on neural networks, then we illustrate the checkpoint mechanism. Finally, we will illustrate how the checkpoint mechanism of the big-scale deep learning framework impacts the lifetime of an SSD, and describe the motivations for our work.

2.1. Deep learning framework

2.1.1. Neural network basics

Fig. 1(a) presents a simple illustration of a neural network unit. A neural unit is a computation unit which has input data such as x_1, x_2 and x_3, that are respectively associated with weights such as w_1, w_2 and w_3. The result of the computation, $h_W(x)$, can be obtained through a given formula. Fig. 1(b) presents a simple illustration of a neural network. A neural network is composed of layers, each of which is composed of a set of neural units. Input data are processed and transferred from layer to layer to finally give the prediction result in the output layer. The process is called forward propagation. The "loss value" is obtained through comparison of the output with the expected result. The weights are iteratively adjusted from the last layer to the first layer. This process is called "backward propagation". This iterative process may be time consuming to obtain accurate weight values.

2.1.2. Checkpoint in deep learning systems

Training a deep learning model may take several hours or days, even when using a large number of computing resources (physical or virtual machines) [1]. In general, a long running job is likely to experience a failure or to be preempted. Some large-scale training systems such as Project Adam [13] conducted training in a cloud environment where machines may be highly unreliable. In addition, the training may be performed on non-dedicated machines with low availability which increases the training duration [1]. All these factors increase the failure rate of the training phase, as a consequence, some fault tolerance mechanism is required to avoid such failures to be fatal. Many deep learning systems adopt the checkpointing mechanism.

Fig. 2 shows a simple illustration of the workflow of TensorFlow which is a very popular deep learning framework in the industry. As shown in the figure, the input data are read and pre-processed, and then pushed into the training iterations. Weight parameters are updated continuously in the training phase. To protect the system against a possible failure or crash, a user-level checkpoint mechanism is used. A background process runs periodically to produce checkpoints during the training phase. The learning tool Caffe [23] also uses periodical checkpointing. Some training tools such as Project Adam [13] and MXNet [10] use parameter server [28] for large scale deep learning. In these tools, persistent storage is used as a write back cache, parameters are flushed asynchronously to the storage devices.

Huge amount of parameters are produced for very large training models [1], and they may occupy terabytes [14] in storage systems. Periodical checkpoint process applies a high pressure on the underlying storage media. In case of SSD, it may significantly degrade its lifetime. So, it is necessary to consider how to reduce the written volume to the persistent storage.

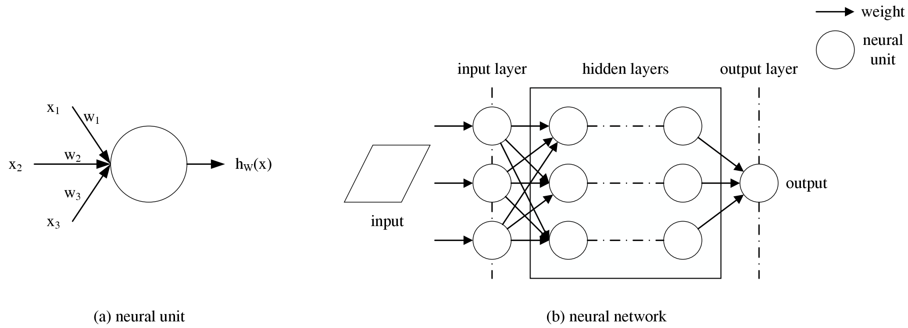


Fig. 1. Neural network.

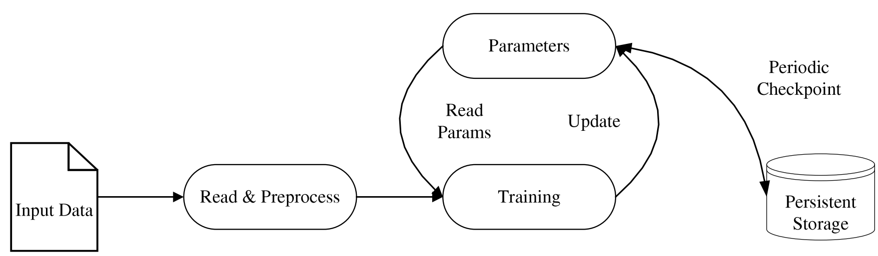


Fig. 2. The workflow of TensorFlow.

2.2. SSD basics

NAND flash memory based SSD has been widely adopted due to its high performance and energy efficiency. Each NAND flash memory cell can only sustain a limited number of write operations. So a huge number of writes will decrease its lifetime. Different from traditional magnetic disks, a NAND flash memory cell cannot be updated directly. It needs to be erased before new data can be written (the erase-before-write property). Before erasing a block, the valid data that it contains needs to be moved (read than written) to another location, generating more write operations than requested. This write amplification (**WA**) is harmful to the lifetime of an SSD. It can be caused by factors such as write updates and garbage collection.

The huge number of parameter updates issued by the checkpointing mechanism in large-scale deep learning tasks may incur significant write amplification. Therefore, it is critical to reduce the total written volume of the snapshot to increase the lifetime of the SSD. State-of-the-art studies present the snapshot method to update only the modified parameters to reduce the snapshot overhead [31]. It is not effective with SSDs for the above-mentioned reasons. Fig. 3 gives a simple illustration about this issue.

WA caused by saving full snapshot: Fig. 3(a) shows an example of saving a full snapshot "AEFD" to a NAND flash memory where the old snapshot "ABCD" is saved in pages 0 and 1. The second and third variables are changed between the two snapshots. The second variable was changed from "B" to "E", and the third one from "C" to "F". In this case, the flash memory cell cannot be updated directly. So the full snapshot is saved in two new pages: pages 2 and 3, and pages 0 and 1 are marked as invalid. In the delta scheme shown in Fig. 3(b), we only need to save the XOR result between the two new variables and the two old ones into page 2. Compared to the full snapshot, the delta snapshot reduced write load by one page and increased the storage space by one page also (as no invalid pages were generated).

This is just an illustrative example and the real case is not that simple. We discuss the algorithm in detail in the next section.

2.3. Motivation

Our main objective is to reduce the overall amount of data written caused by checkpointing during the training phase of large-scale deep learning applications. We observed that the parameters (or weights) between successive snapshots are similar to each other. As the training continues, parameters converge to their final value. This feature can be exploited to reduce the whole written volume. In this way, the total written volume can periodically be reduced in an effective manner. Next, we discuss the reason behind the similarity between the parameters, and then present some experiments conducted to support the motivation.

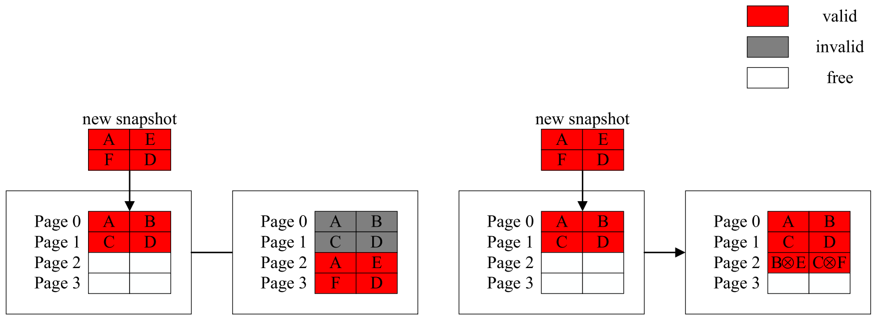
The weight values of a snapshot file in a neural network are described as floating point numbers. In this paper, single precision is used as an illustrative example (although our method can also be applied to double precision floating numbers). A single precision floating point number consists of three parts: a sign bit, an 8-bit exponent part, and a 23-bit significant or mantissa part.

A stochastic gradient descent algorithm [41] is an algorithm that is commonly used to compute the weight values through backward propagation. Its objective is to find the appropriate weight value W that can minimize the loss function (see [41] for more details). In each training iteration, the weight value W is updated as follows:

$$W = W - \alpha\beta(W) \quad (1)$$

In the above equation, α denotes the learning rate, which indicates how aggressive the change in weight values can be. It is usually a constant value or a variable that can cause the convergence to happen more quickly. $\beta(W)$ is a function of W, which is linearly correlated with W. It can be analyzed from that formula in which W will converge to its final value from one iteration to another. The longer the training continues, the less its value will change, as illustrated in Fig. 4, and the more we have similar bits between parameters in adjacent snapshots.

From the aforementioned property of the weight values, we conclude that the exponents of the weights will vary little between consecutive snapshots, especially during the final iterations.



(a) full snapshot (b) delta snapshot

Fig. 3. Write amplification caused by saving full snapshot.

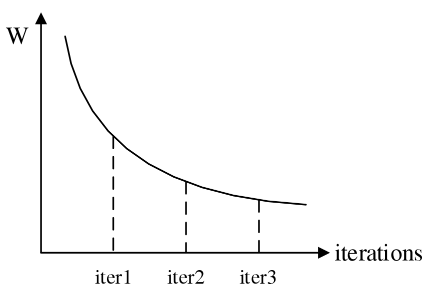


Fig. 4. Weight value *W* changes with iterations.

Table 1
The ratio of same heading bits (shb) of **weights** according to the number of considered bits in the CaffeNet benchmark. The column is the same heading bits between two parameters, the row is the snapshot iteration.

shb	10 000	15 000	20 000	25 000	30 000	35 000	40 000	45 000	50 000
12	17.14%	25.76%	33.23%	87.40%	90.59%	91.20%	91.82%	98.77%	98.81%
11	28.70%	40.42%	49.38%	92.59%	94.60%	94.98%	95.36%	99.33%	99.35%
10	43.20%	56.10%	64.62%	95.72%	96.94%	97.17%	97.40%	99.64%	99.65%
9	58.25%	69.95%	76.80%	97.57%	98.29%	98.42%	98.55%	99.81%	99.82%
8	71.48%	80.55%	85.43%	98.62%	99.04%	99.12%	99.19%	99.90%	99.90%
7	81.88%	88.05%	91.11%	99.21%	99.45%	99.49%	99.54%	99.94%	99.94%
6	87.13%	92.45%	94.74%	99.51%	99.65%	99.67%	99.70%	99.97%	99.97%
5	92.29%	95.47%	96.85%	99.66%	99.76%	99.78%	99.80%	99.97%	99.97%
4	92.75%	95.52%	97.29%	99.85%	99.91%	99.92%	99.93%	99.99%	99.99%

To illustrate this property, we conducted a set of experiments to check the similarity between the weight values in the different snapshots. We trained a benchmark CaffeNet [25] for 50 000 iterations and checkpointed a snapshot every 5000 iterations. We compared the weight values for all successive snapshots that were produced during the training phase. In the experiments, parameters are single precision floating point numbers with 4 bytes (hence it is 32 bits). The similarity between two parameters is represented by the number of same heading bits (notice that the heading bits for a floating number are first the sign, the exponent and then the most significant bit of the mantissa). For example, if two weight values are respectively “01111110” and “01111101”, then there are six same heading bits. Table 1 shows the results of the comparison. The column represents the number of same heading bits used as a base to calculate similarity ratios. The row is the training iteration number. The data in the table cells represent the ratios of the number of parameters with the “shb” same heading bits. This table shows the percentage of parameters with the same heading bits from 4 to 12.

The high ratios of similarity shown in this table motivated us to design a deduplication scheme for the heading bits of the weights in deep learning algorithms. By doing so, we reduced the size of the snapshot and hence decreased the write pressure to the storage system.

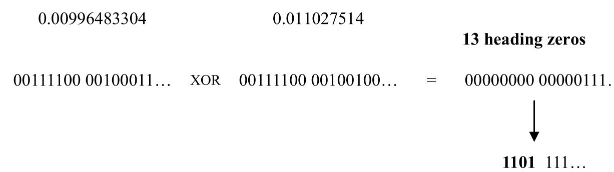


Fig. 5. Run length encoding method.

3. Delta snapshot design

In this section, we first present the basic idea behind delta snapshot. Two alternative schemes for delta snapshot are then illustrated. We will discuss the impact of each scheme on reducing the size of the snapshot and its overhead. Finally, we will detail the main management operations, such as storing snapshots, recovering from failures, and deleting snapshots.

3.1. Delta snapshot

In delta snapshot, rather than storing the whole weight value in each snapshot as it was done in the previous snapshot, we only store the modified bits (delta).

For each weight parameter in the snapshot, we store the XOR result with the previous weight value instead of the whole value in the new snapshot. We called this new snapshot a delta snapshot. To do so, we need to fully save one snapshot, namely the baseline snapshot, with complete weights to use it as a base for XOR calculations.

As previously shown in Table 1, there are some same heading bits between weight parameters in consecutive snapshots. As a result, the XOR operation will generate some ‘0’s for these heading bits in the delta snapshot (sign and exponent part and eventually some significant bits of the mantissa).

We used the Run Length Encoding (RLE) technique [5] for the delta snapshot compression as it can effectively reduce the heading zeros of the XOR results. RLE uses the first ‘x’ bits to denote the number of heading zeros. Fig. 5 illustrates this method simply. There are two single precision floating point numbers in the example. They are both translated in 32 bits. Then, they are XORed to get the delta value. This value has 13 heading zeros. In this example, RLE technique uses 4 bits to describe the heading zeros. So it replaces the 13 heading zeros with the code “1101”, hence saving 9 bits (from the 32).

One issue in such a mechanism is choosing the right number of bits to represent the XOR value. This number should neither be too large, in order to save space, nor too small, to be able to

Table 2
Notations.

Term	Description
$param_{cur}$	Weight parameters in the current snapshot.
$param_{prev}$	Weight parameters in the previous snapshot.
$param_{\Delta}$	The delta snapshot obtained through XORing $param_{cur}$ and $param_{prev}$.
$heading\_bits$	Number of bits to encode the heading zeros of the delta snapshot.
$size_i$	The size of delta snapshot when it used i heading bits.
$heading\_zeros(p)$	Number of heading zeros in the parameter p.

represent all heading zeros. It is better to choose bigger number of bits for delta snapshots with more heading zeros, in which way these zeros can be replaced by less bits. However for delta snapshots with less heading zeros, more heading bits only reduce the bits of the delta value by little or even increase the bits. In this case, less heading bits are preferred. We can observe from Table 1 that the number of similar heading bits between consecutive snapshots increases as the training approaches from convergence. It means that the appropriate number of heading bits to represent the XOR values for different delta snapshot should be different. For this sake, we designed an algorithm called DynRLE to calculate dynamically the number of heading bits to represent the XOR results for each delta snapshot when it is produced.

Before the algorithm is introduced, we firstly present some notations used in the algorithm in Table 2. The terms $param_{cur}$ and $param_{prev}$ respectively represent the weight parameters in the current and previous snapshots. $heading\_bits$ represents the number of bits used to denote the heading zeros of the weight parameters in the delta snapshot. The term $size_i$ denotes the size of the delta snapshot when its heading zeros are encoded using i bits. $heading\_zeros(p)$ gives the number of heading zeros in the parameter p.

Algorithm 1: DynRLE

```
   Input: param_cur , param_prev
   Output: heading_bits
1  param_Δ = param_cur ⊕ param_prev;
2  size_i = 0, (i ∈ [0, 5]);
3  for each parameter p in param_Δ do
4  |  for i = 0, 1 . . . , 5 do
5  |  |  size_i += 32 + i − Min(2^i − 1, heading_zeros(p));
6  if size_i = Min(size_0, . . . , size_5) then
7  |  heading_bits = i;
```

Algorithm 1 shows the procedure to calculate the heading bits for each delta snapshot. First, we obtain the delta snapshot through XORing the parameters of the current snapshot and the previous one (Line 1). Then, we calculate the size of the delta snapshot when the parameters are described using i heading bits (Lines 2–5). In this paper, we assume that parameters are single precision floating point numbers which are denoted using 32 bits. So the heading zeros are denoted using at most 5 bits (it can describe at most $2^5 - 1 = 31$ heading zeros). i heading bits can denote $[0, 2^i - 1]$ heading zeros. If the number of heading zeros of parameter p is smaller than $2^i - 1$, then only $heading\_zeros(p)$ bits can be reduced by the heading code. So $Min(2^i - 1, heading\_zeros(p))$ bits can be reduced by the code, while the code adds i bits, so $size_i$ should be added $32+i-Min(2^i-1, heading\_zeros(p))$ for the parameter p. Finally, we choose the heading bits which minimize the size of the delta snapshot (Lines 6–7).

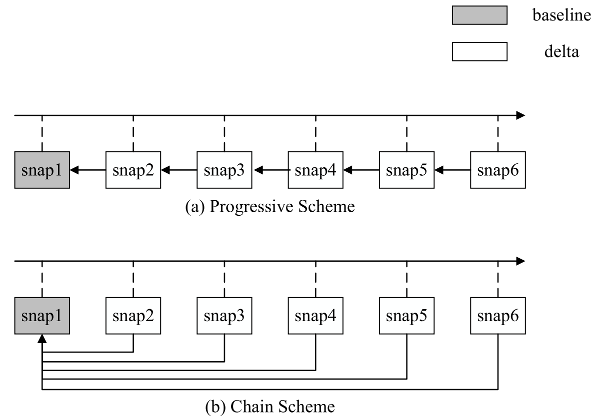


Fig. 6. Comparison of two schemes.

3.2. Two snapshot schemes

The delta snapshot introduced above is computed through XORing two adjacent snapshots. Each snapshot depends on the one just before it in case this latter is a baseline snapshot (containing full weights). We designed two ways of creating snapshots:

- **Progressive scheme:** In this scheme a snapshot is always built based on the previous one. If the latter is a baseline snapshot, then the delta values are directly computed. If the previous snapshot is a delta snapshot, then the full snapshot needs to be reconstructed so as to compute the new delta.
- **Chain scheme**: In order to avoid overhead from reconstructing the previous snapshot, we designed a chain scheme. The idea is to calculate the new delta snapshot based on the nearest baseline snapshot.

Fig. 6 shows the two schemes. Six snapshots are produced during the execution of the program, among which snap1 is the baseline snapshot and the other five are delta snapshots. In the progressive scheme shown in Fig. 6(a), each snapshot depends on the one just before it. To compute snap6, the previous snap5 should be rebuilt, which needs rebuilding snap4, and so on. At the end, all snapshots need to be rebuilt in order to compute the new delta values. Basically, the farther the baseline snapshot is, the more overhead is generated. Indeed, all six snapshots should be kept in the storage system, which consumes too much space. If the chain scheme is adopted as shown in Fig. 6(b), each snapshot only depends on the baseline snapshot. Then, only snap1 and snap6 are stored once snap6 is built. The overhead in terms of memory operations and storage space is much lower.

Due to the gradual change in weight values, the difference between the weights of adjacent snapshots is always smaller than that between discontinuous snapshots. As a consequence, the progressive scheme can always achieve smaller delta snapshots than the chain scheme. On the other hand, the recovering latency and storage space overhead of the chain scheme is much lower than that of the progressive scheme. The weight parameters vary aggressively during the first iterations of the training stage; as a consequence, much smaller snapshots can be obtained with the progressive scheme than with the chain scheme at the cost of more intrusive delta snapshot building operations and storage space. However, for the last training stages, the two schemes obtain snapshots of roughly similar size. We will compare these two schemes in detail in the evaluation part.

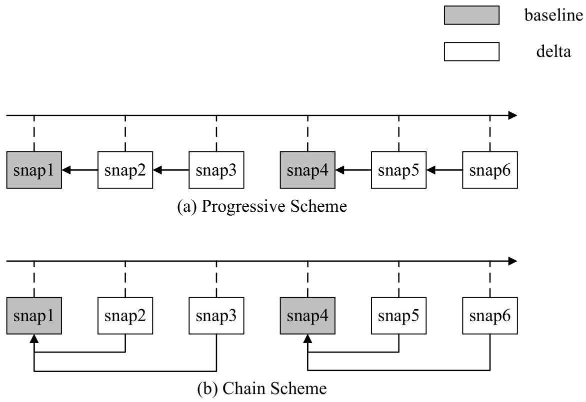


Fig. 7. Snapshots produced in the training when $snap\_dis = 3$.

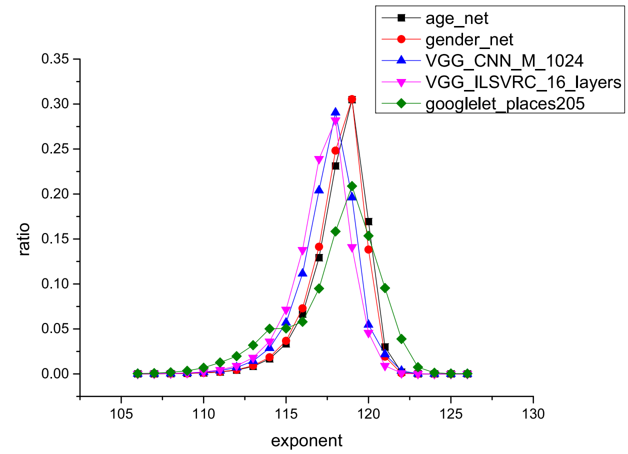


Fig. 8. Exponent number distribution of five benchmarks.

3.3. Baseline snapshot

In this subsection, we first introduce the use of multiple baseline snapshots to reduce the snapshot operation overhead. Then, the compression technique used for the baseline snapshot is introduced.

As training goes by, the distance between the last delta snapshot and the baseline snapshot increases. For the progressive scheme, this induces a significant increase of the building and reconstruction overhead of the delta snapshot, so as the storage space overhead. For the chain scheme, this induces a larger difference in weight values between the baseline snapshot and the last delta snapshot, which means a higher size of the delta snapshot. To avoid this problem, we adopted a multiple baseline snapshot scheme. Baseline snapshots are produced periodically during the training phase. With this method, more than one baseline snapshot will be generated during the training phase. In order to decrease the storage pressure, we also propose a compressing solution to reduce the size of the baseline snapshot, this is described in Section 3.3.2.

3.3.1. Multiple baseline snapshots

For the progressive scheme, having multiple baseline snapshots makes it possible to reduce the overhead of reconstruction operations when a new delta snapshot is built. Indeed, the longer the dependency chain between the snapshots, the higher the overhead. Therefore, one could consider periodically building a new baseline snapshot to reduce the dependency chain. Concerning the chain snapshot, as the delta is built exclusively on the basis of the baseline snapshot, if the latter is generated too early in the training phase, the delta values will be very large. As a consequence, one could consider generating new baseline snapshots to reduce the size of the delta snapshots. We introduce a new variable called *snap_dis*, which denotes the distance between two baseline snapshots.

Fig. 7 extends the example in Fig. 6 with two baseline snapshots, in which $snap\_dis = 3$. Therefore, snap1 and snap4 are both baseline snapshots. Suppose that we want to recover from snap6 with the progressive scheme; in this case, only three snapshots, snap4 to snap6, need to be rebuilt. The operation overhead and storage space are both reduced significantly. For this same example with the chain snapshot, snap6 is rebuilt from snap4 (one snapshot less compared to the progressive scheme).

How to choose the value of *snap_dis* to optimize the performance of the delta snapshot is yet another issue. It can be observed from Table 1 that the parameters change more in the initial training phase than in the last ones. So, for the initial training phase, the value *snap_dis* should be small. When training approaches from convergence and the parameters change less, the value *snap_dis* can be higher since the difference between parameters in different snapshots remains small. We will show how different *snap_dis* values affect the total written volume in the evaluation part.

3.3.2. Compressing the baseline snapshot

Since several baseline snapshots may be produced during the training phase, especially when *snap_dis* is low, it is important to reduce the total written volume. We have conducted a small experiment that underlined the similarity between exponent parts of the parameters. This low entropy rate in the exponent parts can be exploited to compress the baseline snapshot.

We have already extracted the parameters from the model used in several benchmarks. These benchmarks are found in the Caffe Model Zoo which included many open source experimental neural network models. Namely, the Caffe model for age and gender classification [27] trained on the Adience-OUI dataset (*age_net*, *gender_net*), the CNN (Convolutional Neural Network) models trained on the ILSVRC-2012 dataset [9] (*VGG_CNN_M_1024*), the 16-layer CNN models presented by [40] (*VGG_ILSVRC_16_layers*) and the CNN models trained for scene recognition [48].

Fig. 8 shows the distribution of the exponent parts in the Caffe models the aforementioned benchmarks. The weight value is a single precision floating point number and has an 8 bit exponent. We can observe that most exponents fall into a small interval (from 115 to 121). This fact can be exploited to reduce the total snapshot size. We divided the original snapshot into two parts: the first one consists of exponent numbers, and the second one consists of other two parts (sign bit and mantissa part). We developed a compression method to reduce the size of the first part. For the other parts, the entropy rate was very high, so they are not taken into consideration in our contribution.

We developed a compression algorithm based on the Huffman coding method. First, each exponent number is considered as a leaf node, and its weight is its ratio. Then the Huffman tree is created through continuously combining two nodes with the smallest weights. The procedure is repeated until there remains only one node, namely the root node. Then the Huffman code for each leaf node can be obtained through traversing the Huffman tree.

3.4. Snapshot management

In this subsection, we present the snapshot management operations: saving snapshots and restoring them through the reconstruction of snapshots. For simplicity, we use the same example shown in Fig. 7 for the purpose of illustration.

Saving snapshots: We use the example of saving snapshot snap6 to illustrate the process of saving snapshots.

- In the progressive scheme there are two steps: (1) complete snapshot snap5 is rebuilt based on snap4 ; (2) XOR operations are then performed with current weight values to build out the delta snapshot snap6. Snapshots snap4 to snap6 are all kept in the storage system.
- In the chain scheme the reconstruction operation is not needed and delta snapshot snap6 can be obtained through XORing the baseline snapshot snap4 and snap6. So the I/O operations and computation overheads are smaller than that of the progressive scheme. Only snap6 and snap4 (the baseline) are stored; snap5 is not needed and so it is deleted.

Note that in both previous cases, the baseline snapshot snap4 needs to be uncompressed beforehand. The snapshot saving overhead of the progressive scheme increases as *snap_dis* increases, while that of the chain scheme does not increase with *snap_dis*.

Restoring from failure: We use the same example of restoring from failure using snap6 to illustrate the process.

- For the progressive scheme, three snapshots snap4, snap5, and snap6 are loaded and merged to get the full snapshot. First a full snapshot is created from snap5. Then, the full snapshot is created from snap6 (based on the full snapshot of snap5).
- For the chain scheme only snap4 and snap6 are needed to fully reconstruct the snapshot to recover from. A full snapshot of snap6 is built based on snap4.

Thus, the I/O operations and computation overheads of the progressive scheme are higher than those of the chain scheme. In addition, the recovery overhead of the progressive scheme increases with *snap_dis*, while the overhead of the chain scheme is not impacted. However, one can intuitively conclude that the progressive scheme will generate fewer write operations, as the delta snapshots should be smaller. A discussion on the trade-offs between these parameters is given in the experimental evaluation part.

Storage space: Generally, only the weight parameters of the latest snapshot are used in the restoration from a training failure. Therefore, those snapshots that are useful for building the latest snapshot must be saved in the storage system. The other snapshots can be deleted to free up storage space. Suppose that snap6 is the newest snapshot; snap4, snap5, and snap6 should be stored when adopting the progressive scheme. The storage space that is occupied will increase with *snap_dis*. By contrast, in the chain scheme, only snap4 and snap6 need to be saved, and the needed storage space will not increase with *snap_dis*. With this scheme, a maximum of two snapshots is stored.

While the idea behind our method is to compose a complete snapshot using multiple snapshots, there is a dependency between the baseline and delta snapshots, and snapshots cannot be deleted as in state-of-the-art work. For the progressive scheme, once a baseline snapshot is obtained, all previous snapshots can be deleted to free up storage space. For the chain scheme, two snapshots (the latest delta snapshot and the latest baseline snapshot) at most need to be kept in storage, while the others can be deleted to free up storage space. As a consequence, our scheme is about trying to generate fewer write operations by storing more snapshot data.

Table 3
Delta snapshot programming interfaces.

Item	Description
Snap_dis	Number of snapshots among which a baseline snapshot is presented.
Delete (mode, iter, snap_dis, unit)	Delete the delta snapshots with iteration count *iter*. The parameter *mode* is used to denote the delta mode (progressive or chain). The *snap_dis* is shown as above. The *unit* parameter is the number of iterations between two continuous snapshot savings.
Save (para, mode, iter, snap_dis, unit)	Save the delta snapshot whose iteration count is *iter*. The address of saving parameters is given by *para*. The meaning of other parameters is the same as the previous function.
Restore (para, mode, iter, snap_dis, unit)	Restore the parameters from the delta snapshot whose iteration count is *iter*. The saving address of restoring parameters is given by *para*. The other parameters are the same as the previous functions.

4. Implementation

In this section, we present three program interfaces for the delta snapshot, namely delete, save and restore operations. The first function **delete** is used to delete a delta snapshot, the second and the third ones are used respectively to save and restore a delta snapshot.

Table 3 presents a brief description of the three programming interfaces. The meaning of *snap_dis* is the number of snapshots among which a baseline snapshot is presented. Each function has four common parameters *mode*, *iter*, *snap_dis* and *unit*. The first parameter *mode* denotes the delta snapshot mode (progressive or chain), *iter* is the iteration count of the training performed, and *unit* is the number of iterations between two continuous snapshots.

The first function is used to delete a delta snapshot whose iteration count is *iter*. It behaves differently with different modes. The function *save* saves parameters pointed to by *para* from memory to SSD, while the third function *restore* restores parameters from SSD to the memory address given by *para*. The parameter *para* in the second and third functions is a pointer to the parameters of the delta snapshot. Actually, only the last two functions can be called by the user. The function *delete* is a sub-routine of the function *save*.

Algorithm 2 presents how the three functions in Table 3 work. Let us consider the function *delete*, it is a sub-routine of the function *save*. If the progressive mode is used, all the last *snap_dis* snapshots are deleted, as the delete operation in that case only happens when a new baseline snapshot is created. At this time, the last *snap_dis* snapshots are useless and should be deleted (Line 3). For the chain mode, once a new snapshot is obtained, the previous delta snapshot should be deleted (Line 5) as it is not useful anymore. The condition $(iter/unit)\%snap\_dis = 0$ indicates that if it is a baseline snapshot, then the previously stored baseline snapshot $iter - (snap\_dis - 1) * unit$ should be deleted (Line 7).

The *save* function (Lines 8–22) computes the new snapshot and deletes the useless ones. If the snapshot *iter* is a baseline snapshot (Line 9), then useless snapshots should be deleted (Line 10) and a full snapshot should be saved into a new baseline snapshot *iter* (Line 11). Otherwise if the snapshot is a delta snapshot, then we should also consider both cases: the progressive mode (Lines 13–16) and the chain mode (Lines 17–22). If the progressive mode is adopted, then all snapshots from the last baseline snapshot $iter-((iter/unit-1)\%snap\_dis)$ to the previous snapshot

Algorithm 2: Functions

Input: para,mode,iter,snap_dis,unit
Output: NULL
1 **delete(mode,iter,snap_dis,unit)**;
2 **if** $mode = progressive$ **then**
3 Delete snapshots iterates from $iter - (snap\_dis - 1) * unit$ to $iter$;
4 **else**
5 Delete the current delta snapshot $iter$;
6 **if** $(iter/unit)\%snap\_dis == 0$ **then**
7 Delete the baseline snapshot $iter - (snap\_dis - 1) * unit$;
8 **save(para,mode,iter,snap_dis,unit)**;
9 **if** $(iter/unit - 1)\%snap\_dis = 0$ **then**
10 $delete(mode, iter - unit, snap\_dis, unit)$;
11 Save parameters from $para$ into a baseline snapshot $iter$;
12 **else**
13 **if** $mode = progressive$ **then**
14 Load snapshots iterates from $iter - ((iter/unit - 1)\%snap\_dis)$ to $iter - unit$ and merge them into a full snapshot;
15 Call $DynRLE$ to compute the heading bits through comparing parameters from $para$ and that from the full snapshot;
16 Save delta parameters into a new delta snapshot $iter$;
17 **else**
18 Load snapshots $iter - ((iter/unit - 1)\%snap\_dis)$ and $iter - unit$ and merge them into a full snapshot;
19 Call $DynRLE$ to compute the heading bits through comparing parameters from $para$ and that from the full snapshot;
20 Save delta parameters into a new delta snapshot $iter$;
21 **if** $(iter/unit - 2)\%snap\_dis \neq 0$ **then**
22 $delete(mode, iter - unit, snap\_dis, unit)$;
23 **restore(para,mode,iter,snap_dis,unit)**;
24 **if** $mode = progressive$ **then**
25 Load snapshots from $iter - ((iter/unit - 1)\%snap\_dis)$ to $iter$ and merge them into a full snapshot;
26 Extract parameters from the full snapshot and save them into address $para$;
27 **else**
28 Load snapshots $iter - ((iter/unit - 1)\%snap\_dis)$ and $iter$ and merge them into a full snapshot;
29 Extract parameters from the full snapshot and save them into address $para$;

$iter - unit$ should be loaded and merged into a full snapshot $iter$ (Line 14). After that, the function $DynRLE$ is called to compute the heading bits (Line 15) and save a new delta snapshot $iter$ (Line 16). If the chain mode is adopted, only the last baseline snapshot $iter - ((iter/unit - 1)\%snap\_dis)$ and the previous delta snapshot $iter - unit$ are loaded and merged into a full snapshot (Line 18). If the condition $iter - ((iter/unit - 1)\%snap\_dis) = iter - unit$ is satisfied, then the merge operation in Lines 14 and 18 can be omitted. After that, $DynRLE$ is called to calculate the heading bits and obtain the delta snapshot (Lines 19–20). If the previous snapshot is not a baseline snapshot, then we can delete it (Lines 21–22).

The function *restore* (Lines 23–29) is used to restore the parameters from the SSD. If the progressive mode is adopted, then all snapshots from the last baseline snapshot $iter - ((iter/unit - 1)\%snap\_dis)$ to the snapshot $iter$ are loaded and merged into a full snapshot, then it is saved to the memory address $para$ (Lines 24–26). The operation is nearly the same for the chain mode, the difference is that only the last baseline snapshot $iter - ((iter/unit - 1)\%snap\_dis)$ and the snapshot $iter$ are loaded and merged into a full snapshot (Line 28).

5. Evaluation of delta snapshot

We evaluated the delta snapshot in the widely adopted deep learning framework Caffe [23]. The method can also be used in other deep learning frameworks. In Caffe, the user can specify the execution mode (CPU or GPU) of a program, the number of training iterations, and the snapshot interval iterations. Caffe periodically checkpoints the parameters to the snapshot. We used Caffe in our experiments to get the original snapshots with complete parameters, and then these snapshots are used by the proposed technique to obtain the delta snapshots.

The used metric to evaluate the lifetime enhancement of our approach is the number of write operations difference with the traditional approach. Indeed, as stated in [43], minimizing write amplification makes it possible to improve the flash-memory storage devices' lifetime. This is because the number of erase operations, that tend to reduce the lifetime of the flash media, is correlated to the number of write operations. We present the related results in Sections 5.4 and 5.5.

Table 4
Disksim configuration.

Item	Description
Channel	8
Chip per Channel	1
Blocks per chip	16 384
Pages per block	64
Byte Transfer Latency	0.000025 ms
Page Read Latency	0.025 ms
Page Write Latency	0.2 ms
Block Erase Latency	1.5 ms

5.1. Platform and benchmarks

We implemented the three functions in Algorithm 2 on the Disksim simulator with the SSD extension. It can simulate the SSD behavior (read, write and erase) and analyze the access time of disk I/Os. Contrary to real experiments on commercial SSDs, simulation makes it easier to obtain low-level statistics (valid/invalid blocks etc.). We collected the snapshots obtained by Caffe and fed them to the algorithm. Finally, we could get the useful storage space and save/restore time by running the algorithm on Disksim.

Table 4 shows the Disksim configuration used in our experiment. There are 8 channels which can be accessed at the same time. There is only one chip in each channel, so no I/O contention exists between chips. A total of 8 chips can be accessed in parallel, but only one chip can send data to, or receive data from the host at a certain time. The page size is 4 KB in the evaluation, a block is composed of 64 pages. Data transfer rate is given by the "Byte Transfer Latency". In addition, the page read/write and block erase latencies are given in the table.

The benchmarks considered in the experiments are listed in Table 5. We evaluated two benchmarks: CaffeNet [25] (modified AlexNet) trained on the Oxford 102 category flower dataset [35], and GoogleNet [39] trained on the NABirds dataset [42]. The characteristics of both benchmarks are given in Table 5. The table lists the total size of all of the snapshots.

Both benchmarks were trained for 50 000 iterations. For the first benchmark CaffeNet, the snapshot period was set to 5000 iterations, so there were a total of 10 snapshots (checkpoint period is about 25 min.). For the second benchmark GoogleNet, the snapshot period was set to 2000 iterations, so there were a total of 25 snapshots (checkpoint period is around 20 min).

We implemented both progressive scheme and chain scheme for DynRLE. For comparison, we implemented GZIP and RLE algorithm to compress the delta snapshots. We evaluated the total written volume of all snapshots after using these methods.

Table 5
Benchmark characteristics.

Benchmark	Written volume (Bytes)	Description
CaffeNet	2,291,463,260	A neural network model of modified AlexNet to classify high-resolution images into different classes (102 classes).
GoogleNet	1,251,369,400	A neural network model to recognize different species of birds (555 classes, 48 562 images)).

5.2. Evaluation configuration and metrics

5.2.1. Configuration

We evaluated how the metric *snap_dis* impacts the size of snapshots. We tested three values for *snap_dis*: 3, 5, and 10. Choosing a different *snap_dis* produces different numbers of delta and baseline snapshots. If the progressive method is adopted, a higher *snap_dis* value will result in more snapshots being stored in the SSD, thus consuming more storage space. For the chain method, choosing different *snap_dis* will also impact the size of the snapshot, since the size of the delta snapshot may increase when the distance with the baseline snapshot increases.

5.2.2. Evaluation metrics

Our method mainly focuses on reducing the total traffic of written snapshots on SSD, thus enhancing its lifetime. We subdivided the evaluation part into two steps:

- Our first objective is to evaluate by how much this method can reduce the written volume. We then compared the result with that obtained using GZIP (directly applied on the snapshot or on the delta snapshot) and RLE algorithm.
- Then, the number of erase operations is evaluated and compared to several other mechanisms. For this evaluation to be relevant, we have warmed up the SSD in order to have a representative state that generates garbage collections.
- Finally, we measured the overhead of the delta snapshot in terms of useful storage space, snapshot saving time and loading time. The useful storage space is the space occupied by all of the useful snapshots. For example, in the progressive scheme shown in Fig. 7(a), the storage space occupied by snap4, snap5, and snap6 is useful for snap6. We should also evaluate the saving and loading times of the delta snapshots since the training time may be impacted by the merge operations induced by the delta mechanism.

5.3. Discussion about delta granularity

Before going through the evaluation of the delta snapshot, we first show why a larger granularity for incremental checkpoints in a deep learning framework does not work. Some incremental methods adopt snapshots that only save modified parameters [31] or modified pages [15,34,45]. Tables 6 and 7 respectively show the number of similar pages and equal parameters between adjacent snapshots. Each parameter is a single precision floating point number with 4 bytes. A page is 4 K bytes, so there are 1024 parameters in a page. The overall number of parameters saved in a single snapshot is 57 286 118, thus, it can be observed that the number of similar pages or parameters is very low between two snapshots. Hence, these methods are not effective at reducing the total written volume of the snapshots.

Table 6
The number of similar pages between adjacent snapshots in the benchmark CaffeNet.

Snapshots	Same pages
5 000,10 000	0
10 000,15 000	0
15 000,20 000	0
20 000,25 000	0
25 000,30 000	0
30 000,35 000	0
35 000,40 000	0
40 000,45 000	1
45 000,50 000	2

Table 7
The number of similar parameters between adjacent snapshots in the benchmark CaffeNet.

Snapshots	Same parameters
5 000,10 000	12
10 000,15 000	14
15 000,20 000	27
20 000,25 000	350
25 000,30 000	481
30 000,35 000	532
35 000,40 000	585
40 000,45 000	9 361
45 000,50 000	101 066

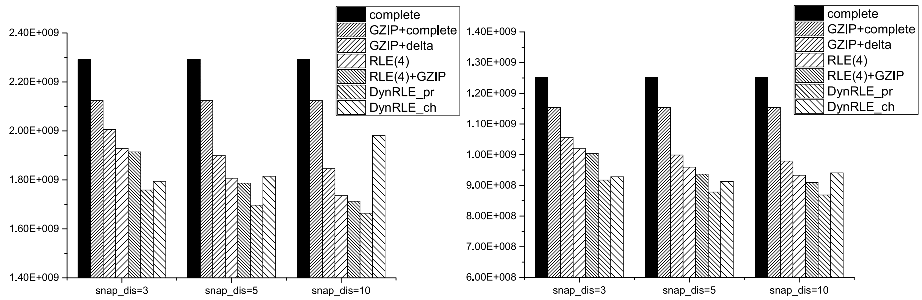
5.4. Reduction of the write volume

In this section, we first evaluate the reduction in the total snapshot size and compare those figures with the results obtained using GZIP and tuned RLE method (adapted from. [5]). Then, we analyzed the obtained results with regard to the *snap_dis* parameter.

Fig. 9 shows the generated write traffic due to snapshot saving using 7 different mechanisms. The first is the complete snapshot without any compression. The second mechanism, denoted "GZIP+complete", compresses the complete snapshot using the tool gzip. The third one, denoted "GZIP+delta", uses the progressive delta snapshot and compresses the snapshots using gzip. The fourth one, denoted RLE(4), uses a static RLE algorithm with 4 heading bits which is the best static configuration for the tested workloads. RLE was applied on a progressive snapshot scheme. Both RLE and "GZIP+delta" were configured with similar *snap_dis* values as our schemes. The fifth method (RLE(4)+GZIP) compressed the result obtained by RLE using the GZIP algorithm. The sixth evaluated mechanism is the DynRLE algorithm with the progressive scheme ($DynRLE_{pr}$), and the final method uses the chain scheme ($DynRLE_{ch}$).

From the figure, we can observe that the dynamic method can better reduce the write traffic issued to the SSD as compared to the other solutions, this is specifically the case for the progressive scheme. It can achieve down to 69% of the write traffic generated by Caffe without optimization (thus reducing the write traffic by 31%) when $snap\_dis = 10$ (for GoogleNet application). GZIP+delta, RLE(4) and RLE(4)+GZIP respectively achieve 80%, 75% and 73% (giving 20%, 25% and 27% write traffic reduction respectively). DynRLE methods generally work better as they adjust the heading bits dynamically during the training phase. In addition, compressing the baseline snapshots also contributes in the write traffic reduction.

One may also observe that the chaining scheme does not behave as good as the progressive one, especially for high values of *snap_dis*. This is because the higher the *snap_dis* value, the larger the difference between the delta and the baseline snapshot. In these particular cases, and for the CaffeNet applications, RLE



(a) CaffeNet (b) GoogleNet

Fig. 9. Size of total written volume of different methods based on a full snapshot (unit: bytes), namely (from left to right): complete snapshot, complete snapshot compressed by GZIP (GZIP+complete), delta snapshot compressed by GZIP (GZIP+delta), Run Length Encoding with 4 heading bits (RLE(4)), RLE(4) results compressed by GZIP (RLE(4)+GZIP), progressive DynRLE method and chain DynRLE method.

Table 8
Warm-up SSD configuration.

Parameter	Value
Valid blocks	111 432
Invalid blocks	13 048
Free blocks	6592
Over provisioning space	15%
GC threshold	5%

Table 9
Useful storage space in pages and snapshot save and restore time in seconds for CaffeNet and $snap\_dis = 5$.

Snapshot	Progressive			Chain		
	Useful storage	Save (s)	Restore (s)	Useful storage	Save (s)	Restore (s)
5 000	46 581	1.945	0.760	46 581	1.945	0.760
10 000	94 354	2.757	1.540	94 354	2.757	1.540
15 000	140 551	3.469	2.294	94 968	2.782	1.550
20 000	185 760	4.182	3.032	95 254	2.793	1.555
25 000	224 782	4.662	3.669	95 265	2.793	1.555
30 000	46 588	1.945	0.760	46 588	1.945	0.760
35 000	85 059	2.367	1.388	85 059	2.367	1.388
40 000	123 436	2.992	2.015	85 850	2.401	1.401
45 000	156 529	3.396	2.555	85 912	2.403	1.402
50 000	189 525	3.933	3.094	85 975	2.406	1.403

behaved better than the chaining scheme. This is because it was based on the progressive scheme and as a matter of fact, the delta snapshots compressed were smaller.

The delta snapshot size obtained by the progressive scheme decreases when *snap_dis* increases. This is quite intuitive since the number of delta snapshots increases and the number of the baseline snapshots decreases. The delta snapshot size obtained by the progressive scheme is not impacted by the value *snap_dis*, it is obtained by XORing the current snapshot with the one before it. On the other hand, for the chain scheme, the snapshot size increases when *snap_dis* decreases, since the size of the delta snapshot increases when *snap_dis* increases, because the distance between the delta snapshot and the baseline snapshot increases also. The progressive scheme produces a lower write traffic but it consumes storage space, while chain scheme does the contrary.

5.5. Erase operations

The objective of this contribution is to reduce the write pressure of the snapshot mechanism which leads to an increase in the SSD lifetime and a better performance during the checkpoint-process. In this section, we evaluate the ability of the delta snapshot to reduce the number erase operations.

Garbage collection happens when the free pages of SSD are below a certain threshold. So, in order to have a realistic initial state for our experiments, we have warmed up the evaluated SSD before launching the deep learning application. Data blocks in SSD are divided into: valid, invalid and free blocks. The over provisioning space is the disk space which should be reserved for GC purpose. Warm-up SSD configuration is presented in Table Table 8. The value in the table is presented as a proportion of the total SSD space. The GC threshold is a lower bound to trigger the GC operation.

Fig. 10 presents the erase operations generated by saving the snapshots. One may observe that the delta snapshot reduced the number of erase operations more than the other techniques. The obtained result is consistent with Fig. 9. Larger written volume would cause more erase operations since more GC operations are triggered. Also the *DynRLE_pr* obtains the least number of erase operations when *snap_dis* equals 10. For example for CaffeNet, the number of erase operations is 6448 which is 25% lower than the GZIP method, while compared with the default snapshot method with no compression, our method can reduce the erase operations by about 27%. Hence it can reduce the wear brought by the large written volumes of data during the training phase of deep learning applications.

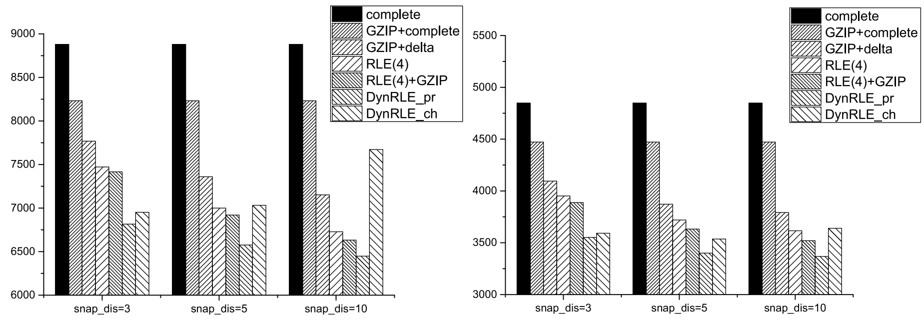
5.6. Useful storage space and saving/loading time

In our method, a complete snapshot is composed of several snapshots (a baseline snapshot and some other delta snapshots). All snapshots that need to build the latest complete snapshot should be kept in the storage system. The detailed procedure can be found in Algorithm 2. We called the storage space used by these snapshots "useful storage". When the amount of parameters is huge, the useful storage space may also be very large. So, it is critical to evaluate the used storage space of delta snapshot. We have also evaluated the save/restore time for the two proposed methods.

5.6.1. Useful storage space

Table 9 shows the useful storage space and the saving and restoring times for the two methods when $snap\_dis = 5$. The useful storage space is given in pages since it is the read/write unit of SSD (a page is 4 KB in size). One can observe that the useful storage space of the progressive scheme is much larger than that of the chain scheme. When *snap_dis* increases, the difference is even higher.

The progressive scheme restores the latest snapshot by merging all of the snapshots from the latest baseline snapshot (if the latest snapshot is a baseline snapshot, it does not need to undergo a merge operation). In the worst case, *snap_dis* snapshots are merged to get the complete information of the latest snapshot.



(a) CaffeNet (b) GoogleNet

Fig. 10. Erase operations brought by the written snapshots.

On the other hand, the chain method only needs to merge two snapshots at the most to get the latest snapshot (if the latest snapshot is a baseline snapshot, then there is no merge operation). Therefore, the chain scheme generates less useful storage space as compared to the progressive scheme.

5.6.2. Saving/restoring time

The save/restore time of the delta snapshot is obtained through implementing the method in DiskSim with SSD extension. Resulted save/restore time is shown in Table 9. The baseline snapshot save/restore time of two methods is the same. As training goes on, the last snapshots' save/restore time of the progressive method becomes larger than that of the chain method. Since for the progressive method, more and more snapshots need to be loaded and merged as the target snapshot went away from the baseline snapshot.

Overall, the save/restore time of both methods is less than one minute in the worst case. However, the training usually lasted for several hours. It can even last for hundreds of hours in case of large deep learning applications [28]. So this metric is not an important metric to be considered.

5.6.3. Computing time and memory usage

The saving/restoring time shown in Table 9 only presents the I/O time obtained from DiskSim. The CPU time to compute the delta snapshot or restore a complete snapshot cannot be obtained from the I/O simulator, so, we have measured this overhead on laptop Thinkpad X1. The highest overhead to compute one delta snapshot or restore a complete snapshot was measured to last 4.1 s. This is negligible compared to the overall training which lasted for 4 h.

When using our method, many delta snapshots need be loaded into memory and merged in order to form a complete snapshot. This loading may push data of other processes from memory to the storage. One may think this could counter balance the I/O reduction brought by the delta snapshots. Actually, this is not an issue. We have divided the computing process into iterations. The memory usage of each iteration is small. There is an example illustrated in Fig. 11. There is a baseline snapshot and a delta snapshot which are both divided into n partitions. There are two buffers 1 and 2 in the memory to store the temporary data used in one iteration. In the first iteration, the data parts b_1 and d_1 are loaded into memory and merged into a complete snapshot part c_1. In the second iteration, the buffers are loaded with b_2 and d_2 and c_2 are figured out. The compute process goes on until the complete snapshot is computed. The process to compute a new delta snapshot is similar. So we only used two more buffers compared with traditional snapshot method. The larger n is, the smaller the buffer is, and more load operations there are. Actually, the computing overhead of the load operation is negligible, so n can be set very large. In the experiment, the two buffers only used several megabytes. So our method only used several megabytes more memory space than complete snapshot method.

5.7. Summary

5.7.1. The most important metric

We have discussed various metrics in the paper: reduced written volume, the number of erase operations, useful storage space, saving/restoring time. Which is the most important metric to be considered? Since the objective of this paper is to improve the lifetime of SSD. It has been stated in reference [43] that the lifetime of SSD is mainly related to the erase operations of SSD. Less erases meant longer lifetime. And the erases are triggered by writes. So the most important metric to be considered is the written volume.

It can be seen from Table 9 that progressive scheme's useful storage space is larger than that of chain scheme. The useful storage space in large deep learning applications as [28] may exceed the storage capacity. If the storage capacity is large enough to hold all delta snapshots, then the progressive scheme should be chosen, or the chain scheme should be chosen.

The saving/restoring time is not an issue to be considered, it has been explained in Section 5.6.2 that it is very short compared to the total training time.

5.7.2. The issue of snap_dis

How to get the optimal *snap_dis* for the application? Table 10 presented a guidance to choose the optimal *snap_dis* for both schemes. For the progressive scheme, larger *snap_dis* means smaller written volume. So the largest *snap_dis* which satisfied storage space constraint should be chosen. For the chain scheme, the optimal *snap_dis* is achieved through a linear search from smallest value to the largest value. It can be seen from Table 9, the useful storage space of chain scheme is usually smaller than that of the progressive scheme. The optimal *snap_dis* with smallest written volume which satisfied the storage constraint is chosen.

5.7.3. Performance

The performance/overhead of the method is nearly linearly related to the number of parameters. For the two benchmarks in this paper, a single snapshot in CaffeNet is 229 146 326 bytes, the time to compute a snapshot is around 4.1 s, a snapshot in GoogleNet is 50 054 776 bytes, the time to compute a snapshot is around 0.94 s. We can observe that we have the same ratio between the snapshot sizes and the time for snapshot creation. This tends to confirm the linear correlation between snapshot computation time and snapshot size (number of parameters). However, as stated in Section 5.6.2, this time is negligible compared to the total training time.

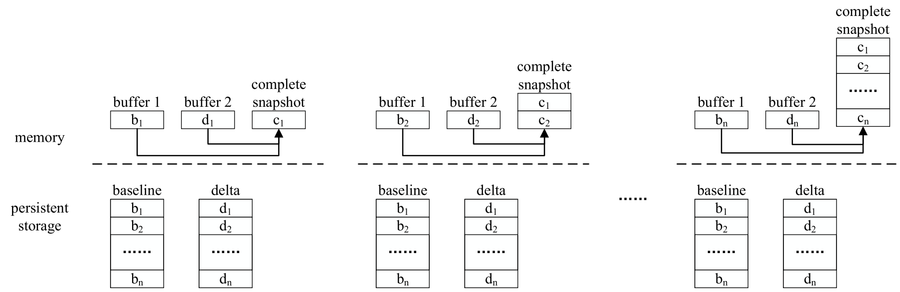


Fig. 11. Divide the snapshots merging into iterations.

Table 10
How to choose the optimal *snap_dis* value.

	With storage constraint	Without storage constraint
Progressive	Largest *snap_dis* which satisfy the storage constraint	Largest *snap_dis*
Chain	The *snap_dis* with smallest written volume and satisfied storage constraint	The *snap_dis* with smallest written volume

6. Related work

There are various studies investigating the following three areas: improvements in the endurance of SSDs, incremental checkpoints, and snapshot size reductions for deep learning frameworks. In this section, we will briefly discuss each area.

Endurance improvement: Nowadays, NAND flash-based solid-state disks (SSDs) are widely used in both personal computers and embedded systems [6,18,20,22]. Thanks to the continuous process of scaling semiconductors, the price gap between SSDs and HDDs has narrowed. Unfortunately, the limited endurance of NAND flash memory is a major barrier to its widespread adoption. In effect, a flash memory block becomes worn out if its erase count reaches a certain limit. For example, in the case of the Samsung K9F1G08U0C Single-Level Cell (SLC), the number of erase counts is about 100 K, and it is even smaller for the Multi-Level Cell (MLC) NAND flash. Many state-of-the-art studies have been conducted with the aim of improving the endurance of NAND flash memory such as [3,7,8,11,12,16,17,21,24,26,29,30,36,37,44,46,49]. These methods exploit the features of NAND flash to increase the lifetimes of SSDs; however, they do not target the data features of the deep learning applications. Our method exploits the similarity between snapshots to reduce the total write volume of the deep learning framework; hence, it is orthogonal to the above methods.

Compression in the deep learning applications: GraphLab [31] stored the modified parameters to reduce the snapshot size. However, the it can be seen from Table 7 we can see that number of same total parameters in the benchmark. So the methods in [31] are not effective to compress the snapshots.

Compression in scientific data: Many high-performance computing applications have proposed to use incremental checkpoints [15,34,45] to reduce the redundant information between snapshots. Instead of keeping the complete information in the full snapshot, incremental snapshot mechanisms only save the modified pages (4 KB) compared with the last snapshot. However, from Table 6 we can see there are nearly no same pages between snapshots in deep learning training phase; hence, this coarse-grained method nearly does not reduce the size of the snapshots. Many advanced compression methods for scientific data have been proposed [4,32,38]. The method in [38] proposed a byte-wise compression for floating data, and [4] used this method as a delta scheme for floating point scientific data. In [32], the authors used it for the deep learning snapshot deltas. However, these methods are byte-wise which are also too coarse-grained, our method is bit-wise so our method is better than theirs. The solution in [5] is the closer to ours, but it cannot adjust the heading bits dynamically in the runtime and it also does not present any scheme for reducing the baseline snapshot size.

Lossy Snapshots: it has been proposed by many papers [1,19] that deep learning models tolerate low-precisions, it means that a lossless compression scheme is not always necessary. Modern deep learning framework such as Tensorflow [1] used lossy compression (truncating high order bytes) for floating numbers while sending weight parameters to other workers in distributed training. It is proposed in [19] that using less bits for floating number nearly has no impact on the accuracy of the neural network. While the method in this paper only considered the lossless condition, it can also be complemented with lossy compression techniques.

7. Conclusion

The training of large-scale deep learning frameworks produces very large snapshot files that are periodically written to the storage system. When such training is conducted on SSDs, large volumes of written data might severely impact their endurance. In this paper we presented delta snapshots, a mechanism aiming at reducing the overall size of written data to reduce the impact on the lifetime of SSDs. Delta snapshot reduces the overall write traffic by storing only the difference between snapshots. Our mechanism relies on similarities between significant parts of floating point numbers to reduce the volume of data written to the storage media. The experimental results show that our method can reduce the overall write traffic by more than 30% at the expense of some reasonable overhead for saving and recovery.

Acknowledgements

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15222315, GRF 15273616, GRF 15206617, GRF 15224918), and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055096). We thank all the reviewers for their comments and feedback.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al., Tensorflow: A system for large-scale machine learning, in: OSDI, Vol. 16, 2016, pp. 265–283.

[2] A. Angelova, A. Krizhevsky, V. Vanhoucke, A.S. Ogale, D. Ferguson, Real-time pedestrian detection with deep network Cascades, in: BMVC, Vol. 2, 2015, p. 4.

[3] A. Badam, V.S. Pai, Ssdalloc: hybrid ssd/ram memory management made easy, in: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2011, pp. 211–224.

[4] S. Bhattacherjee, A. Deshpande, A. Sussman, Pstore: an efficient storage framework for managing scientific data, in: Proceedings of the 26th International Conference on Scientific and Statistical Database Management, ACM, 2014, p. 25.

[5] T. Bicer, J. Yin, D. Chiu, G. Agrawal, K. Schuchardt, Integrating online compression to accelerate large-scale data analytics applications, in: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on, IEEE, 2013, pp. 1205–1216.

[6] J. Boukhobza, P. Olivier, Flash Memory Integration, first ed., Elsevier, 2017.

[7] J. Boukhobza, P. Olivier, S. Rubini, L. Lemarchand, Y. Hadjadj-Aoul, A. Laga, Macach: An adaptive cache-aware hybrid FTL mapping scheme using feedback control for efficient page-mapped space management, J. Syst. Archit. 61 (3) (2015) 157–171.

[8] L.-P. Chang, Y.-S. Liu, W.-H. Lin, Stable greedy: Adaptive garbage collection for durable page-mapping multichannel SSDs, ACM Trans. Embedded Comput. Syst. (TECS) 15 (1) (2016) 13.

[9] K. Chatfield, K. Simonyan, A. Vedaldi, A. Zisserman, Return of the devil in the details: Delving deep into convolutional nets, in: British Machine Vision Conference, 2014.

[10] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, arXiv preprint arXiv:1512.01274 2015.

[11] R. Chen, Z. Qin, Y. Wang, D. Liu, Z. Shao, Y. Guan, On-demand block-level address mapping in large-scale nand flash storage systems, IEEE Trans. Comput. 64 (6) (2015) 1729–1741.

[12] R. Chen, Y. Wang, D. Liu, Z. Shao, S. Jiang, Heating dispersal for self-healing NAND flash memory, IEEE Trans. Comput. 66 (2) (2017) 361–367.

[13] T.M. Chilimbi, Y. Suzue, J. Apacible, K. Kalyanaraman, Project adam: Building an efficient and scalable deep learning training system., in: OSDI, Vol. 14, 2014, pp. 571–582.

[14] A.M. Dai, C. Olah, Q.V. Le, Document embedding with paragraph vectors, arXiv preprint arXiv:1507.07998 2015.

[15] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, C. Engelmann, Combining partial redundancy and checkpointing for hpc, in: Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on, IEEE, 2012, pp. 615–626.

[16] Y. Guan, G. Wang, C. Ma, R. Chen, Y. Wang, Z. Shao, A block-level log-block management scheme for MLC NAND flash memory storage systems, IEEE Trans. Comput. (2017).

[17] Y. Guan, G. Wang, Y. Wang, R. Chen, Z. Shao, Blog: block-level log-block management for nand flash memorystorage systems, ACM SIGPLAN Not. 48 (5) (2013) 111–120.

[18] J. Guo, C. Min, T. Cai, Y. Chen, A design to reduce write amplification in object-based NAND flash devices, in: Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2016 International Conference on, IEEE, 2016, pp. 1–10.

[19] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, in: International Conference on Machine Learning, 2015, pp. 1737–1746.

[20] S. He, Y. Wang, X.-H. Sun, C. Huang, C. Xu, Heterogeneity-aware collective i/o for parallel i/o systems with hybrid HDD/ssd servers, IEEE Trans. Comput. 66 (6) (2017) 1091–1098.

[21] M. Huang, Z. Liu, L. Qiao, Y. Wang, Z. Shao, An endurance-aware metadata allocation strategy for MLC NAND flash memory storage systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 35 (4) (2016) 691–694.

[22] J. Jeong, Y. Song, S.S. Hahn, S. Lee, J. Kim, Dynamic erase voltage and time scaling for extending lifetime of nand flash-based ssds, IEEE Trans. Comput. 66 (4) (2017) 616–630.

[23] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional Architecture for Fast Feature Embedding, arXiv preprint arXiv:1408.5093 2014.

[24] D. Jung, J.-s. Kim, S.-y. Park, J.-U. Kang, J. Lee, Fass: A flash-aware swap system, in: Proc. of International Workshop on Software Support for Portable Storage (IWSSPS), 2005.

[25] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Advances in Neural Information Processing Systems, 2012, pp. 1097–1105.

[26] T.-W. Kuo, Y.-H. Chang, P.-C. Huang, C.-W. Chang, Special issues in flash, in: Computer-Aided Design, 2008. ICCAD 2008. IEEE/ACM International Conference on, IEEE, 2008, pp. 821–826.

[27] G. Levi, T. Hassner, Age and gender classification using convolutional neural networks, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2015, pp. 34–42.

[28] M. Li, D.G. Andersen, J.W. Park, A.J. Smola, A. Ahmed, V. Josifovski, J. Long, E.J. Shekita, B.-Y. Su, Scaling distributed machine learning with the parameter server, in: OSDI, Vol. 14, 2014, pp. 583–598.

[29] D. Liu, K. Zhong, T. Wang, Y. Wang, Z. Shao, E.H.-M. Sha, J. Xue, Durable address translation in PCM-based flash storage systems, IEEE Trans. Parallel Distrib. Syst. 28 (2) (2017) 475–490.

[30] D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, Z. Shao, Non-volatile memory based page swapping for building high-performance mobile devices, IEEE Trans. Comput. 66 (11) (2017) 1918–1931.

[31] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J.M. Hellerstein, Distributed graphlab: a framework for machine learning and data mining in the cloud, Proc. VLDB Endow. 5 (8) (2012) 716–727.

[32] H. Miao, A. Li, L.S. Davis, A. Deshpande, Towards unified data and lifecycle management for deep learning, in: Data Engineering (ICDE), 2017 IEEE 33rd International Conference on, IEEE, 2017, pp. 571–582.

[33] T.I. Murphy, Storage Performance Basics for Deep Learning, http://timmurphy.org/2009/07/22/line-spacing-in-latex-documents/ (Accessed 04.04.10).

[34] B. Nicolae, F. Cappello, Blobcr: efficient checkpoint-restart for HPC applications on iaas clouds using virtual disk image snapshots, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 34.

[35] M.-E. Nilsback, A. Zisserman, Automated flower classification over a large number of classes, in: Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on, IEEE, 2008, pp. 722–729.

[36] Z. Qin, Y. Wang, D. Liu, Z. Shao, Y. Guan, MNFTL: An efficient flash translation layer for MLC NAND flash memory storage systems, in: Proceedings of the 48th Design Automation Conference, ACM, 2011, pp. 17–22.

[37] M. Saxena, M.M. Swift, Flashvm: Revisiting the virtual memory hierarchy., in: HotOS, 2009, p. 13.

[38] E.R. Schendel, Y. Jin, N. Shah, J. Chen, C.-S. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, et al., Isobar preconditioner for effective and high-throughput lossless data compression, in: Data Engineering (ICDE), 2012 IEEE 28th International Conference on, IEEE, 2012, pp. 138–149.

[39] M. Simon, E. Rodner, Neural activation constellations: Unsupervised part model discovery with convolutional networks, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1143–1151.

[40] K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, CoRR, abs/1409.1556, 2014.

[41] UFLDL Tutorial, Website, http://ufldl.stanford.edu/wiki/index.php/UFLDL_Tutorial 2013.

[42] G. Van Horn, S. Branson, R. Farrell, S. Haber, J. Barry, P. Ipeirotis, P. Perona, S. Belongie, Building a bird recognition app and large scale dataset with citizen scientists: The fine print in fine-grained dataset collection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 595–604.

[43] W.-L. Wang, T.-Y. Chen, Y.-H. Chang, H.-W. Wei, W.-K. Shih, Minimizing write amplification to enhance lifetime of large-page flash-memory storage devices, in: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), IEEE, 2018, pp. 1–6.

[44] Y. Wang, D. Liu, Z. Qin, Z. Shao, An endurance-enhanced flash translation layer via reuse for NAND flash memory storage systems, in: Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011, IEEE, 2011, pp. 1–6.

[45] C. Wang, F. Mueller, C. Engelmann, S.L. Scott, Hybrid full/incremental checkpoint/restart for mpi jobs in hpc environments, in: International Conference on Parallel and Distributed Systems, 2011.
[46] Y. Wang, Z. Qin, R. Chen, Z. Shao, L.T. Yang, An adaptive demand-based caching mechanism for nand flash memory storage systems, ACM Trans. Des. Autom. Electron. Syst. (TODAES) 22 (1) (2016) 18.
[47] H. Zhang, G. Chen, B.C. Ooi, K.-L. Tan, M. Zhang, In-memory big data management and processing: A survey, IEEE Trans. Knowl. Data Eng. 27 (7) (2015) 1920–1948.
[48] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, A. Oliva, Learning deep features for scene recognition using places database, in: Advances in Neural Information Processing Systems, 2014, pp. 487–495.
[49] P. Zhou, B. Zhao, J. Yang, Y. Zhang, A durable and energy efficient main memory using phase change memory technology, ACM SIGARCH Comput. Archit. News 37 (3) (2009) 14–23.

Zhu Wang received the B.E. degree in software engineering from the Wuhan University of China at Wuhan, China, in 2009, and the Ph.D. degree from the Department of Computer Science, Zhejiang University at Zhejiang, China in 2015. He has been a postdoc in the Department of Computing, Hong Kong Polytechnic University, since he graduated from Zhejiang University. His research interests include embedded software and systems.

Jalil Boukhobza received the electrical engineering (with Hons.) degree from the Institut Nationale d'Electricitè et d'èlectronique (I.N.E.L.E.C), Boumerdès, Algeria, in 1999, and the M.Sc. and Ph.D. degrees in computer science from the University of Versailles, France, in 2000 and 2004, respectively. He was a research fellow at the PRiSM Laboratory (University of Versailles) from 2004 to 2006. He has been an associate professor with the University Bretagne Occidentale, Brest, France, since 2006 and is a member of Lab-STICC. His main research interests include flash-based storage system design, performance evaluation and energy optimization, and operating system design. He works on different application domains such as embedded systems, cloud computing, and database systems.

Zili Shao received the B.E. degree in electronic mechanics from the University of Electronic Science and Technology of China, Sichuan, China, in 1995, and the M.S. and Ph.D. degrees from the Department of Computer Science, University of Texas at Dallas, in 2003 and 2005, respectively. He has been an associate professor in the Department of Computer Science and Engineering, The Chinese University of Hong Kong, since 2018. His research interests include big data systems, storage systems, embedded systems/software research.

Preserving SSD lifetime in deep learning applications with delta snapshots

Wang, Zhu; Boukhobza, Jalil; Shao, Zili

01 斯凌 杨 Page 1

31/8/2019 5:00

02 chen shuaiben Page 1

23/8/2019 9:16

03 chen shuaiben Page 1

23/8/2019 9:16

04 chen shuaiben Page 1

23/8/2019 9:16

05 chen shuaiben Page 1

23/8/2019 9:16

06 斯凌 杨 Page 1

31/8/2019 5:00

07 chen shuaiben Page 1

23/8/2019 9:16

08 chen shuaiben Page 1

23/8/2019 9:16

09 斯凌 杨 Page 1

31/8/2019 5:00

10 斯凌 杨 Page 1

31/8/2019 5:00

11 斯凌 杨 Page 1

31/8/2019 5:00

12 斯凌 杨 Page 1

31/8/2019 5:00

13 chen shuaiben Page 1

23/8/2019 9:16

14 chen shuaiben Page 1

23/8/2019 9:16

15 chen shuaiben Page 1

23/8/2019 9:16

16 chen shuaiben Page 1

23/8/2019 9:16

17 chen shuaiben Page 1

23/8/2019 9:16

18 chen shuaiben Page 1

23/8/2019 9:16

19 chen shuaiben Page 1

23/8/2019 9:16

20 chen shuaiben Page 1

23/8/2019 9:16

21 chen shuaiben Page 1

23/8/2019 9:16

22 chen shuaiben Page 1

23/8/2019 9:16

23 chen shuaiben Page 1

23/8/2019 9:16

24 chen shuaiben Page 1

23/8/2019 9:16

25 斯凌 杨 Page 1

31/8/2019 5:00

26 斯凌 杨 Page 1

31/8/2019 5:00

27 斯凌 杨 Page 1

31/8/2019 5:00

28 chen shuaiben Page 1

23/8/2019 9:16

29 chen shuaiben Page 1

23/8/2019 9:16

30 chen shuaiben Page 1

23/8/2019 9:16

31 chen shuaiben Page 2

23/8/2019 9:16

32 chen shuaiben Page 2

23/8/2019 9:16

33 chen shuaiben Page 2

23/8/2019 9:16

34 chen shuaiben Page 2

23/8/2019 9:16

35 chen shuaiben Page 2

23/8/2019 9:16

36 chen shuaiben Page 2

23/8/2019 9:16

37 斯凌 杨 Page 2

31/8/2019 5:00

38 斯凌 杨 Page 2

31/8/2019 5:00

39 斯凌 杨 Page 2

31/8/2019 5:00

40 斯凌 杨 Page 2

31/8/2019 5:00

41 斯凌 杨 Page 2

31/8/2019 5:00

42 斯凌 杨 Page 2

31/8/2019 5:00

43 斯凌 杨 Page 2

31/8/2019 5:00

44 斯凌 杨 Page 2

31/8/2019 5:00

coarse-grained update optimization

45 斯凌 杨 Page 2

31/8/2019 5:00
(Comment from 斯凌 杨) coarse-grained update optimization

46 斯凌 杨 Page 2

31/8/2019 5:00
(Comment from 斯凌 杨) coarse-grained update optimization

47 斯凌 杨 Page 2

31/8/2019 5:00

48 斯凌 杨 Page 2

31/8/2019 5:00

49 chen shuaiben Page 2

23/8/2019 9:16

50 chen shuaiben Page 2

23/8/2019 9:16

51 chen shuaiben Page 2

23/8/2019 9:16

52 斯凌 杨 Page 2

31/8/2019 5:00

53 斯凌 杨 Page 2

31/8/2019 5:00

54 斯凌 杨 Page 2

31/8/2019 5:00

55 斯凌 杨 Page 2

31/8/2019 5:00

56 斯凌 杨 Page 2

31/8/2019 5:00

57 斯凌 杨 Page 2

31/8/2019 5:00

58 chen shuaiben Page 2

23/8/2019 9:16

59 chen shuaiben Page 2

23/8/2019 9:16

60 chen shuaiben Page 2

23/8/2019 9:16

61 chen shuaiben Page 2

23/8/2019 9:16

62 chen shuaiben Page 2

23/8/2019 9:16

63 chen shuaiben Page 2

23/8/2019 9:16

64 斯凌 杨 Page 2

31/8/2019 5:00

65 斯凌 杨 Page 2

31/8/2019 5:00

66 斯凌 杨 Page 2

31/8/2019 5:00

67 chen shuaiben Page 2

23/8/2019 9:16

68 斯凌 杨 Page 2

31/8/2019 5:00

69 斯凌 杨 Page 2

31/8/2019 5:00

70 斯凌 杨 Page 2

31/8/2019 5:00

71 chen shuaiben Page 2

23/8/2019 9:16

72 chen shuaiben Page 2

23/8/2019 9:16

73 chen shuaiben Page 2

23/8/2019 9:16

74 chen shuaiben Page 2

23/8/2019 9:16

75 chen shuaiben Page 2

23/8/2019 9:16

76 斯凌 杨 Page 2

31/8/2019 5:00

77 斯凌 杨 Page 2

31/8/2019 5:00

78 斯凌 杨 Page 2

31/8/2019 5:00

79 斯凌 杨 Page 2

31/8/2019 5:00

80 斯凌 杨 Page 2

31/8/2019 5:00

81 斯凌 杨 Page 2

31/8/2019 5:00

82 chen shuaiben Page 2

23/8/2019 9:16

83 chen shuaiben Page 2

23/8/2019 9:16

84 chen shuaiben Page 2

23/8/2019 9:16

85 斯凌 杨 Page 2

31/8/2019 5:00

86 斯凌 杨 Page 2

31/8/2019 5:00

87 斯凌 杨 Page 2

31/8/2019 5:00

88 斯凌 杨 Page 2

31/8/2019 5:00

89 斯凌 杨 Page 2

31/8/2019 5:00

90 斯凌 杨 Page 2

31/8/2019 5:00

91 chen shuaiben Page 3

23/8/2019 9:16

92 斯凌 杨 Page 3

31/8/2019 5:00

93 chen shuaiben Page 3

23/8/2019 9:16

94 斯凌 杨 Page 3

31/8/2019 5:00

95 斯凌 杨 Page 3

31/8/2019 5:00

96 chen shuaiben Page 3

23/8/2019 9:16

97 斯凌 杨 Page 3

31/8/2019 5:00

98 chen shuaiben Page 3

23/8/2019 9:16

99 斯凌 杨 Page 3

31/8/2019 5:00

100 chen shuaiben Page 3

23/8/2019 9:16

101 chen shuaiben Page 3

23/8/2019 9:16

102 斯凌 杨 Page 3

31/8/2019 5:00

103 chen shuaiben Page 3

23/8/2019 9:16

104 chen shuaiben Page 3

23/8/2019 9:16

105 chen shuaiben Page 3

23/8/2019 9:16

106 斯凌 杨 Page 4

31/8/2019 5:00

107 斯凌 杨 Page 4

31/8/2019 5:00

108 斯凌 杨 Page 4

31/8/2019 5:00

109 斯凌 杨 Page 5

31/8/2019 5:00

110 chen shuaiben Page 5

23/8/2019 9:16

111 chen shuaiben Page 5

23/8/2019 9:16

112 chen shuaiben Page 5

23/8/2019 9:16

113 斯凌 杨 Page 5

31/8/2019 5:00

114 斯凌 杨 Page 5

31/8/2019 5:00

115 斯凌 杨 Page 5

31/8/2019 5:00

116 斯凌 杨 Page 5

31/8/2019 5:00

117 斯凌 杨 Page 5

31/8/2019 5:00

118 斯凌 杨 Page 5

31/8/2019 5:00

119 斯凌 杨 Page 5

31/8/2019 5:00

120 斯凌 杨 Page 5

31/8/2019 5:00

121 斯凌 杨 Page 5

18/8/2019 0:35

122 斯凌 杨 Page 5

31/8/2019 5:00

123 斯凌 杨 Page 5

18/8/2019 0:35

124 斯凌 杨 Page 5

18/8/2019 0:35

125 斯凌 杨 Page 5

18/8/2019 0:36

126 斯凌 杨 Page 5

31/8/2019 5:00

127 斯凌 杨 Page 5

31/8/2019 5:00

128 斯凌 杨 Page 5

31/8/2019 5:00

129 斯凌 杨 Page 5

18/8/2019 0:36

130 斯凌 杨 Page 5

18/8/2019 0:36

131 斯凌 杨 Page 5

31/8/2019 5:00

132 斯凌 杨 Page 5

31/8/2019 5:00

133 chen shuaiben Page 5

23/8/2019 9:16

134 chen shuaiben Page 5

23/8/2019 9:16

135 chen shuaiben Page 5

23/8/2019 9:16

136 chen shuaiben Page 6

23/8/2019 9:16

137 chen shuaiben Page 6

23/8/2019 9:16

138 chen shuaiben Page 6

23/8/2019 9:16

139 chen shuaiben Page 6

23/8/2019 9:16

140 chen shuaiben Page 6

23/8/2019 9:16

141 chen shuaiben Page 6

23/8/2019 9:16

142 chen shuaiben Page 6

23/8/2019 9:16

143 chen shuaiben Page 6

23/8/2019 9:16

144 chen shuaiben Page 6

23/8/2019 9:16

145 chen shuaiben Page 6

23/8/2019 9:16

146 chen shuaiben Page 6

23/8/2019 9:16

147 chen shuaiben Page 6

23/8/2019 9:16

148 斯凌 杨 Page 6

31/8/2019 5:00

149 斯凌 杨 Page 6

31/8/2019 5:00

150 斯凌 杨 Page 6

31/8/2019 5:00

151 chen shuaiben Page 6

23/8/2019 9:16

152 chen shuaiben Page 6

23/8/2019 9:16

153 chen shuaiben Page 6

23/8/2019 9:16

154 chen shuaiben Page 6

23/8/2019 9:16

155 chen shuaiben Page 6

23/8/2019 9:16

156 chen shuaiben Page 6

23/8/2019 9:16

157 斯凌 杨 Page 7

18/8/2019 0:35

158 斯凌 杨 Page 7

18/8/2019 0:35

159 斯凌 杨 Page 7

18/8/2019 0:35

160 斯凌 杨 Page 7

18/8/2019 0:35

161 斯凌 杨 Page 7

18/8/2019 0:35

162 斯凌 杨 Page 7

18/8/2019 0:35

163 chen shuaiben Page 7

23/8/2019 9:16

164 chen shuaiben Page 7

23/8/2019 9:16

165 chen shuaiben Page 7

23/8/2019 9:16

166 斯凌 杨 Page 7

18/8/2019 0:35

167 斯凌 杨 Page 7

18/8/2019 0:35

168 斯凌 杨 Page 7

18/8/2019 0:35

169 斯凌 杨 Page 7

18/8/2019 0:35

170 斯凌 杨 Page 7

18/8/2019 0:35

171 斯凌 杨 Page 7

18/8/2019 0:35

172 斯凌 杨 Page 7

18/8/2019 0:35

173 斯凌 杨 Page 7

18/8/2019 0:35

174 斯凌 杨 Page 7

18/8/2019 0:35

175 斯凌 杨 Page 7

18/8/2019 0:35

176 斯凌 杨 Page 7

18/8/2019 0:35

177 斯凌 杨 Page 7

18/8/2019 0:35

178 斯凌 杨 Page 7

18/8/2019 0:36

179 斯凌 杨 Page 7

18/8/2019 0:36

180 斯凌 杨 Page 7

18/8/2019 0:36

181 chen shuaiben Page 8

23/8/2019 9:16

182 chen shuaiben Page 8

23/8/2019 9:16

183 chen shuaiben Page 8

23/8/2019 9:16

184 chen shuaiben Page 8

23/8/2019 9:16

185 chen shuaiben Page 8

23/8/2019 9:16

186 chen shuaiben Page 8

23/8/2019 9:16

187 斯凌 杨 Page 8

18/8/2019 0:36

188 斯凌 杨 Page 8

18/8/2019 0:36

189 斯凌 杨 Page 8

18/8/2019 0:36

190 chen shuaiben Page 8

23/8/2019 9:16

191 chen shuaiben Page 8

23/8/2019 9:16

192 chen shuaiben Page 8

23/8/2019 9:16

193 chen shuaiben Page 9

23/8/2019 9:16

194 chen shuaiben Page 9

23/8/2019 9:16

195 chen shuaiben Page 9

23/8/2019 9:16

196 斯凌 杨 Page 9

18/8/2019 0:35

197 斯凌 杨 Page 9

18/8/2019 0:35

198 斯凌 杨 Page 9

18/8/2019 0:35

199 斯凌 杨 Page 9

18/8/2019 0:35

200 斯凌 杨 Page 9

18/8/2019 0:35

201 斯凌 杨 Page 9

18/8/2019 0:35

202 斯凌 杨 Page 9

18/8/2019 0:35

203 斯凌 杨 Page 9

18/8/2019 0:35

204 斯凌 杨 Page 9

18/8/2019 0:35

205 斯凌 杨 Page 9

18/8/2019 0:36

206 斯凌 杨 Page 9

18/8/2019 0:36

207 斯凌 杨 Page 9

18/8/2019 0:36

208 chen shuaiben Page 9

23/8/2019 9:16

209 chen shuaiben Page 9

23/8/2019 9:16

210 chen shuaiben Page 9

23/8/2019 9:16

211 chen shuaiben Page 9

23/8/2019 9:16

212 chen shuaiben Page 9

23/8/2019 9:16

213 chen shuaiben Page 9

23/8/2019 9:16

214 chen shuaiben Page 10

23/8/2019 9:16

215 chen shuaiben Page 10

23/8/2019 9:16

216 chen shuaiben Page 10

23/8/2019 9:16

217 chen shuaiben Page 10

23/8/2019 9:16

218 chen shuaiben Page 10

23/8/2019 9:16

219 chen shuaiben Page 10

23/8/2019 9:16

220 chen shuaiben Page 10

23/8/2019 9:16

221 chen shuaiben Page 10

23/8/2019 9:16

222 chen shuaiben Page 10

23/8/2019 9:16

223 chen shuaiben Page 11

23/8/2019 9:16

224 chen shuaiben Page 11

23/8/2019 9:16

225 chen shuaiben Page 11

23/8/2019 9:16

226 chen shuaiben Page 11

23/8/2019 9:16

227 chen shuaiben Page 11

23/8/2019 9:16

228 chen shuaiben Page 11

23/8/2019 9:16

229 chen shuaiben Page 11

23/8/2019 9:16

230 chen shuaiben Page 11

23/8/2019 9:16

231 chen shuaiben Page 11

23/8/2019 9:16

232 斯凌 杨 Page 11

22/8/2019 0:50

233 chen shuaiben Page 11

23/8/2019 9:16

234 chen shuaiben Page 11

23/8/2019 9:16

235 chen shuaiben Page 11

23/8/2019 9:16

236 斯凌 杨 Page 11

22/8/2019 0:50

237 斯凌 杨 Page 11

22/8/2019 0:50

238 chen shuaiben Page 11

23/8/2019 9:16

239 chen shuaiben Page 11

23/8/2019 9:16

240 chen shuaiben Page 11

23/8/2019 9:16

241 chen shuaiben Page 11

23/8/2019 9:16

242 chen shuaiben Page 11

23/8/2019 9:16

243 chen shuaiben Page 11

23/8/2019 9:16

244 chen shuaiben Page 11

23/8/2019 9:16

245 chen shuaiben Page 11

23/8/2019 9:16

246 chen shuaiben Page 11

23/8/2019 9:16

247 chen shuaiben Page 11

23/8/2019 9:16

248 斯凌 杨 Page 11

22/8/2019 0:50

249 chen shuaiben Page 11

23/8/2019 9:16

250 chen shuaiben Page 11

23/8/2019 9:16

251 斯凌 杨 Page 11

22/8/2019 0:50

252 斯凌 杨 Page 11

22/8/2019 0:50

253 chen shuaiben Page 11

23/8/2019 9:16

254 chen shuaiben Page 11

23/8/2019 9:16

255 chen shuaiben Page 11

23/8/2019 9:16

256 斯凌 杨 Page 11

22/8/2019 0:50

257 斯凌 杨 Page 11

22/8/2019 0:50

258 斯凌 杨 Page 11

22/8/2019 0:50

259 chen ping Page 12

15/9/2019 3:40

260 chen ping Page 12

15/9/2019 3:40

261 chen ping Page 12

15/9/2019 3:40

262 斯凌 杨 Page 12

22/8/2019 0:50

263 斯凌 杨 Page 12

22/8/2019 0:50

264 斯凌 杨 Page 12

22/8/2019 0:50

265 chen shuaiben Page 12

23/8/2019 9:16

266 chen shuaiben Page 12

23/8/2019 9:16

267 chen shuaiben Page 12

23/8/2019 9:16

268 chen ping Page 12

15/9/2019 3:43

269 chen ping Page 12

15/9/2019 3:43

270 chen ping Page 12

15/9/2019 3:43

271 斯凌 杨 Page 12

22/8/2019 0:50

272 chen shuaiben Page 12

23/8/2019 9:16

273 斯凌 杨 Page 12

22/8/2019 0:50

274 斯凌 杨 Page 12

22/8/2019 0:50

275 chen shuaiben Page 12

23/8/2019 9:16

276 chen shuaiben Page 12

23/8/2019 9:16

277 chen ping Page 12

15/9/2019 3:39

278 chen ping Page 12

15/9/2019 3:39

279 chen ping Page 12

15/9/2019 3:39