MLCache: A Space-Efficient Cache Scheme based on Reuse Distance and Machine Learning for NVMe SSDs

Weiguang Liu, Jinhua Cui*
Junwei Liu, Laurence T. Yang
Huazhong University of Science and Technology
China

ABSTRACT

Non-volatile memory express (NVMe) solid-state drives (SSDs) have been widely adopted in emerging storage systems, which can provide multiple I/O queues and high-speed bus to maximize high data transfer rate. NVMe SSD use streams (also called "Multi-Queue") to store related data in associated locations or for other performance enhancements. The on-board DRAM cache inside NVMe SSDs can efficiently reduce the disk accesses and extend the lifetime of SSDs, thus improving the overall efficiency of the storage systems. However, in previous studies, such SSD cache has been only used as a shared cache for all streams or a statically partitioned cache for each stream, which may seriously degrade the performance-per-stream and underutilize the valuable cache resources.

In this paper, we present MLCache, a space-efficient shared cache management scheme for NVMe SSDs, which maximizes the write hit ratios, as well as enhances the SSD lifetime. We formulate cache space allocation as a machine learning problem. By learning the impact of reuse distance on cache allocation, we build a workload specific neural network model. At runtime, MLCache continuously monitors the reuse distance distribution for the neural network module to obtain space-efficient allocation decisions. Experimental results show MLCache improves the write hit ratio of the SSD by 24% compared to baseline, and achieves response time reduction by 13.36% when compared with baseline. MLCache is 96% similar to the ideal model.

KEYWORDS

NVMe SSDs, Cache partition, Reuse distance, Neural network, Machine learning

ACM Reference Format:
Weiguang Liu, Jinhua Cui and Junwei Liu, Laurence T. Yang. 2020. MLCache: A Space-Efficient Cache Scheme based on Reuse Distance and Machine Learning for NVMe SSDs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415652

*Jinhua Cui is the corresponding author (cjhnicole@gmail.com)

1 INTRODUCTION

NAND flash memory-based solid-state drives (SSDs) are widely used in the modern computer systems, due to their high throughput, low response time, and energy consumption advantages over traditional magnetic hard disk drives (HDDs). With the continuing semiconductor process scaling and the bit density improvement [12], the SSD capacity has dramatically increased in a cost-effective fashion. As a result, SSDs become a promising solution to replace HDDs in storage medium.

Over the past decade, to meet the increasing performance demand of modern applications, one of the most prospective advances in flash memory industry is the non-volatile memory express (NVMe) interface used in modern SSDs to further enhance the SSD performance. NVMe SSDs provide the high-speed peripheral component interconnect express (PCIe) bus to enable high throughput (e.g., 126.03 GB/s). Furthermore, NVMe SSDs support the streams (multi-queue [20]), which enables low latency for I/O requests by executing multiple I/O requests received from the host side in parallel[3]. Therefore, NVMe protocol becomes a promising solution to replace the conventional serial advanced technology attachment (SATA) protocol in modern SSDs. A fundamental difference of a modern NVMe SSD from a conventional SSD is its use of multiple queues that directly expose the device controller to applications[13, 24].

In order to bridge the performance gap between host-side memory and the device-side SSD, the on-board volatile DRAM is widely employed as a buffer cache layer (simply "cache" henceforth) for modern SSDs in storage systems. Cache is used to exploit the access locality of the user I/O traffic, optimizing the I/O sequence and reducing the I/O accesses to the backend devices. Considerable research efforts have been devoted to the design of the appropriate cache replacement policy, such as LRU, PT-LRU [11]. However, modern NVMe SSDs provide the diverse multiple streams (e.g., QoS-oriented, capacity-oriented, or latency-sensitive) on the cache. In previous studies, such cache is roughly used as either a global shared cache or a statically partitioned cache for each stream, which have several shortcomings. Although the global shared cache scheme is simple and convenient, it fails to deal with the phenomenon of *performance interference.* This is due to the contention of the cache for the multiple streams and each stream can cause the exhaustion of the entire cache space, which will unintentionally affect the performance of the other streams. In contrast, the latter statically partitioned cache scheme statically divides the independent cache space between multi-queue, avoiding performance interference. However, the valuable cache resources may be underutilized when some request streams have the low locality of the user I/O traffic. Furthermore, in the statically partitioned cache scheme, the partitioned cache space is inflexible and it could not be adapted to the dynamic cache demand changes. This is due to the request data possess a great deal of variation in real access patterns, i.e., the locality and frequency of the request data could change during runtime.

In this paper, we present MLCache, a space-efficient shared cache management scheme for NVMe SSDs, maximizing the write hit ratios, as well as enhancing the SSD lifetime. To achieve this goal, several approaches are proposed. First, MLCache allocates efficient cache space for each stream independently according to the reuse distance distribution and relative request frequency. Secondly, MLCache uses artificial neural network techniques to adaptively learn and predict the optimal cache space allocation scheme for the multiple streams. We summarize our contributions as follows.

- We use both the recent reuse distance distribution of each stream and its relative frequency to allocate cache space for multiple streams in NVMe SSDs. We predict the future optimal cache allocation through appropriate historical observations. We approximate the reuse distance function distribution with the frequency of the reuse interval section.
- MLCache formulates the dynamic cache space allocation as a machine learning problem. By learning the impact of the reuse distance and the its relative frequency on the cache space allocation, MLCache builds a workload-specific neural network model. MLCache continuously monitors the reuse distance distribution to dynamically obtain the adaptive space-efficient allocation decisions.
- We evaluate MLCache and compare it to the baseline and the ideal cache allocating model. Experimental results show MLCache improves the write hit ratio of the SSD by 24% compared to baseline, and achieves response time reduction by 13.36% when compared with baseline. MLCache basically approximates the ideal model. MLCache is 96% similar to the ideal model.

The rest of this paper is organized as follows: Section 2 discusses the background. Section 3 provides the design details of of MLCache, and the experimental results are analyzed in Section 4. Section 5 present more related work. Finally, Section 6 concludes this paper.

2 BACKGROUND

2.1 Cache in NVMe SSDs

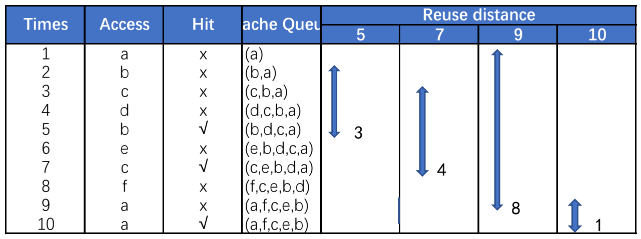
Cache is a significant component of modern SSDs, which can reduce the I/O response time and extend the lifetime of SSDs. The read and write speeds of SSDs vary greatly and write operations are more time-consuming than read operations. Furthermore, write operations may cause background activities, e.g., garbage collection and refresh. Thus the access latency will increase significantly. In order to use the cache space more effectively, the MLCache caching solution adopts a write request caching strategy.

Modern NVMe SSDs provide the diverse multiple I/O queues. As a result, how different streams share limited cache capacity is critical to the performance and longevity of the NVMe SSDs. The researchers explored statically independent partitioning and globally shared cache allocation strategies, each with its advantages and disadvantages. The global sharing strategy can cause interference between streams, and high-frequency requests for some streams may cause the exhaustion of cache resources. The static division strategy can not be well distributed according to the needs of the stream. When multiple streams are running at the same time, different streams have different requirements for the cache. Some streams have high cache requirements and high hit rates, and some have low cache efficiency and low hit rates. The static independent partitioning scheme will not only reduce the relative global shared capacity but also cannot reasonably use the cache based on dynamic requirements. The stream that has a large demand for cache resources and the stream that hardly needs to use the cache may be allocated the same cache size. In this way, the cache is unreasonably allocated, wasting precious cache resources. It is a problem to efficiently and dynamically allocate cache space on demand.

Different from the above strategies, this paper designs a solution to dynamically divide the cache space. The proposed MLCache scheme divides different independent spaces for each stream according to the run-time requirements. In this way, it combines the performance isolation of static independent division strategies and the advantages of high capacity and dynamic requirements of global sharing strategies.

2.2 Reuse distance

The locality of the data request is directly related to the cache hit rate. Reuse distance is an important indicator of request localities and is also a widely used data request locality modeling. The cache hits mainly depend on the relationship between the requested reuse distance and the cache size.



Times	Access	Hit	ache Queu	Reuse distance			
				5	7	9	10
1	a	x	(a)				
2	b	x	(b,a)				
3	c	x	(c,b,a)				
4	d	x	(d,c,b,a)				
5	b	√	(b,d,c,a)	3			
6	e	x	(e,b,d,c,a)				
7	c	√	(c,e,b,d,a)		4		
8	f	x	(f,c,e,b,d)				
9	a	x	(a,f,c,e,b)			8	
10	a	√	(a,f,c,e,b)				1

Figure 1: Relationship between reuse distance and cache hit,when cache queue length is set to 5.

Figure 1 uses a simple trace to show the relationship between reuse distance and cache hits. The trace length is 10, and 6 different data are referenced. The cache queue length is set to 5. The reuse distance can specifically quantify the locality of the data request, and provide an important reference value for the cache behavior. In this example, only the reuse distances of requests No. 5, No.7, No. 9, and No. 10 are finite integers, and the reuse distances of others are infinite. We use the LRU algorithm as a replacement algorithm. It can be observed that, when the reuse distance of a request is less than or equal to the cache capacity, the request will hit; otherwise, it will not hit. Therefore, the reuse distance can effectively predict the cache behavior.

When counting the reuse distance, we record T as the number of cache lines. We use T as the upper limit of reuse distance, and ignore requests with reuse distance greater than T. Because the cache space is not so large, there is no need to waste time and space to count the reuse distance greater than T. The biggest factor affecting cache behavior is the locality of the request. When multiple streams run together, in addition to the locality of the data request, the relative request frequency of the stream also affects the overall cache hit. Because when multiple streams compete together, the relatively high frequency of requests means more use of caching. Reuse distance can effectively predict the optimal distribution scheme of the stream. We can observe the reuse pattern of data requests from the change in reuse distance, thereby further optimizing cache allocation.

When we solve these two sub-problems, our cache partitions optimization problem will be solved.1) How to obtain the reuse distance and relative frequency of each stream request. 2) How to build a model based on quantified locality and frequency to obtain the optimal solution.

2.3 Artificial neural networks

The artificial neural network (ANN) is one of the most powerful machine learning models, which can automatically learn to approximate the target model. Artificial neural networks have been widely used in many complex practical problems. Artificial neural networks can handle inaccurate, fuzzy, noisy, and probabilistic information, and can also generalize learned tasks to unknown tasks[7]. Two important functions of artificial neural networks are modeling and prediction. Multi-layer neural networks can be considered as general approximators, which can approximate any function to any precision [7], which is suitable for the precision that is not available. The theoretical model that is difficult to analyze and calculate using the data observed in the experiment. The prediction ability of the artificial neural network includes ANN training using time series samples of a certain phenomenon in a given scene to predict the temporal behavior after other scenes. Artificial neural networks have been widely used in SSD problems[10, 23]. In view of the advantages of artificial neural networks, we use artificial neural networks to model cache allocation based on reuse distance.

3 MLCACHE DESIGN

3.1 MLCache architecture

The core of MLCache consists of two parts: Reuse distance monitoring module and neural network module. Reuse distance module is used to continuously monitor each stream's request address and then analyze the reuse distance and relative frequency of each stream. The function of the neural network module is to build the target model. When the reuse distance and relative frequency of different streams are input in the model, the output is the allocated space of each stream. The neural network can get a space-efficient cache scheme.

Figure 2 shows the location of the MLCache cache module in the NVMe SSD. The reuse distance detection module and the neural network module are the core of MLCache. When the space division is not changed, the two modules will occupy a small amount of resources and will not add the delay of the SSD process. When the operating system initiates a cache request, the request address will be passed to the reuse distance detection module in parallel. The reused distance monitoring module continuously collects the request addresses sent from the interface of the operating system.

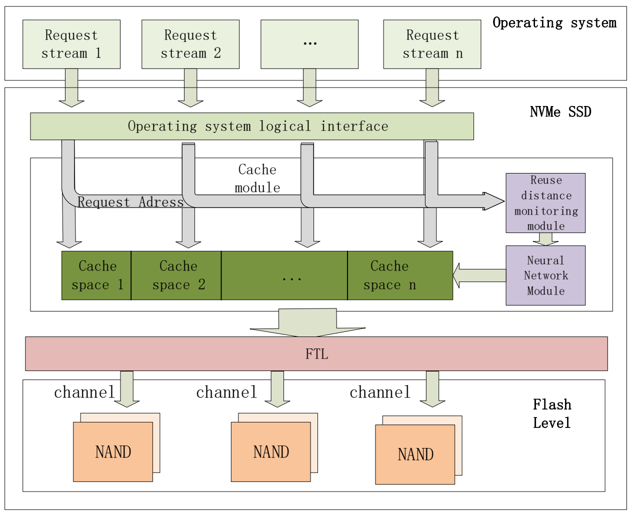


Figure 2: System overview with MLCache.

At intervals, the reused distance monitoring module transmits the reused distance distribution and relative frequency to the neural network module. The trained neural network uses the distance distribution and relative frequency as input to obtain the predicted optimal allocation plan. The allocation plan is processed and submitted to the cache control module for the reallocation of the cache space. The SSD cache space can be dynamically allocated by the cache control module according to the instructions of the neural network allocation module.

3.2 MLCache

For SSD cache, the locality of the request and the relative request frequency are important indicators. We balance locality and request frequency to divide space for different streams. MLCache solves two problems of the cache division. The first problem is how to describe the reused distance distribution of the stream to form input data. The second problem is how to predict the optimal allocation scheme of the multi-stream cache based on the data obtained in the first problem.

3.2.1 Monitor. MLCache continuously monitors the requested address to obtain reused distance information. MLCache uses PARDA to calculate the reuse distance of each stream request and records it in the queue[18]. When the NVMe SSD is running, the reuse distance of requests varies greatly. To avoid the violent response of the cache allocation module caused by great short-term changes, we use the nearest reuse distance as a reference. MLCache uses an additional queue to store the reused distance of the last N requests, which is called Recent Queue. The larger the value of N, the more historical requests to observe, and the less short-term interference. After counting the Recent Queue, we get the distribution of the reused distance of recent requests.

The stream identification and reuse the distance are collected in Recent Queue. We count the number of requests on different reuse distance intervals on each stream and use interval statistics to approximate the distribution function of N request reuse distances. The value of N needs to be set reasonably. If the value of N is too large, MLCache cannot timely sense the change of the request reuse distance, causing a delay in obtaining information, so that the current request mode is limited by historical requests. If the value of N is too small, the reuse distance observed by MLCache is easily affected by short-term request fluctuations, and finally a short-term judgment is made. The appropriate value of N can make MLCache adaptively perceive the reuse distance of each stream. N should be selected based on the longest fluctuation period of the flow.

MLCache counts the Recent Queue at the end of each time interval to obtain the frequency of k streams in all reused distance intervals. We will evenly divide the cache space into t values, denoted as $X_1, X_2, X_3 \dots X_t$. MLCache uses arrays $A_{1-k}[]$ to record the frequency of k streams reuse distance intervals. $A_i[v]$ records the number of requests in Recent Queue that belong to S_i and the reuse distance is less than X_v. In this way, $A_{1-k}[]$ records the frequency of each stream in each reused distance interval segment. $A_{1-k}[]$ can reflect both the locality of each stream and the relative frequency of multi-stream requests.

Dividing the cache space into t discrete interval segments is an approximate sampling. MLCache uses statistics of discrete interval segments to approximate continuous reused distance distribution functions to facilitate modeling. The larger t is set, the finer the cache reuse interval is divided, and the statistics of the reuse interval segment is closer to the original reuse distance distribution function. But the value of t should not be set too large, because the input of the model is proportional to the t value. The larger the t value, the higher the accuracy of the model, and at the same time, it makes the model more difficult to train. So it is significant to choose an appropriate t value.

The algorithm flow of the monitoring module is as follows:

a. First of all, MLCache initializes Recent Queue and $A_{1-k}[]$, and load the reuse distance calculation module (PARDA algorithm[18]).

b. When the SSD request arrives, MLCache checks whether it is a write request. If it is a write request, monitor the reused distance. When the reused distance exceeds the upper limit, it will not be added to the queue. If it is a read request, it is ignored. When the reuse distance is less than the upper limit, add the requested (including stream ID, reuse distance) to the Recent Queue.

c. When starting a new action selection interval, MLCache counts the number of requests in the reused distance intervals of each stream in the queue to obtain array $A_{1-k}[]$.

d. MLCache transfers $A_{1-k}[]$ to the artificial neural network module.

The pseudocode for monitoring and counting is shown in Algorithm 1 and Algorithm 2.

Algorithm 1 Reuse distance monitoring module

```
Require: Stream ID of the request : ($R_{id}$), Address of the request : ($R_{addr}$), Kind of the request : ($R_{kind}$)
Ensure: Initial the latest request queue :($Q$); Load the trained neural network : ($NetFile$)
function MONITOR($R_{id}, R_{addr}, R_{kind}$)
    if $R_{kind}$ = Read then
        return
    end if
    if $R_{kind} \neq$ Read then
        ReuseDistance = $CalculateReuseDistance(R_{id}, R_{addr})$
        if Queuelength(Q) > N then
            $Dequeue(Q)$
        end if
        $Enqueue(Q, (R_{id}, ReuseDistance))$
        return
    end if
    return
end function
```

Algorithm 2 Reuse distance counting module

```
function ALLOCATECACHE()
    Initial Distribution array of reused distance intervals for each stream : ($A_{[1...K]}[\,]$)
    Max = Queuelength($Q$)
    i = 0
    /* $k$ is the number of intervals, $CacheSize$ is the size of cache , $w$ is the width of the intervals, it equals $CacheSize/k$ */
    while $i++ < Max$ do
        ($R_{id}, ReuseDistance$) = Q[i]
        A$_{[Rid]}[floor(ReuseDistance/w)]$++
    end while
    return $A_{[1...K]}[\,]$
end function

```

Algorithm 3 Neural network module

```
function NEURALNETWORK($A_{[1...K]}[\,]$)
    Load the NetFile to the neural network module Net()
    $Result[\,]$ = Net($A_{[1...K]}[\,]$)
    ConstraintChecking($Result$)
    return $Result[\,]$
end function
```

3.2.2 *Artificial neural networks.*

The neural network is a widely used machine learning model. Although the neural network cannot completely fit the target optimal model, it can approximate the target optimal when the training data and training volume are large enough. As mentioned in the previous section, we can model the locality of data requests and the frequency of requests. When running multiple streams in parallel, the possible combinations of locality and frequency in real-world operations are almost infinite, so we need a model to predict a large number of possible inputs. Therefore, we transform the model into an artificial neural network model. We train the neural network by inputting the locality and frequency of the data requests and the corresponding optimal allocations. The trained model can quickly determine the high-performance distribution method in a dynamically changing operating environment. The pseudocode for the neural network module is shown in Algorithm 3.

Artificial neural networks can perform complex modeling well. The number of neurons and the number of layers in each layer affect the fitting ability of artificial neural networks. The more the number of neurons and the number of layers, the more complicated the model that the neural network can simulate.

Since flash memory is a hardware device, which requires low access latency, we use the neural network model in an offline mode. MLCache trains the neural network well in advance and uses the trained neural network directly. In order to reduce the access latency caused by the neural network, MLCache does not adopt a deep neural network with lots of deep neurons. Because the MLCache input layer only contains the reuse distance and frequency of multiple streams, it is relatively easy to model through learning. We take $A_{1-k}[]$ as input in the previous section, and then use the real optimal distribution solution as a label for training. We use the simulator to simulate cache allocation to get the real optimal distribution solution. The output layer is the size of the cache space for each stream, and the loss is set as the squared difference between the predicted value and the true label value.

In MLCache, insufficient training samples may cause underfitting. MLCache simulates a large number of different traces combinations during training and then saves the training data and the optimal solution together.

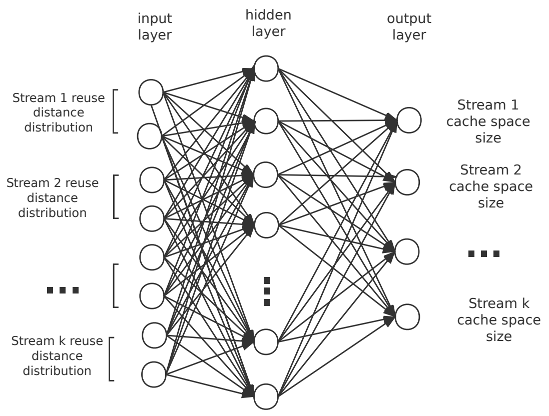

input layer
hidden layer
output layer
Stream 1 reuse distance distribution
Stream 2 reuse distance distribution
...
Stream k reuse distance distribution
Stream 1 cache space size
Stream 2 cache space size
...
Stream k cache space size

Figure 3: Artificial neural network when the number of streams is n.

Disrupting the order of the streams in $A_{1-k}[]$ can train the model more effectively, thus avoiding the influence of the order on the neural network. Since there are no abnormal samples in this model, there is basically no overfitting. During training, we need a large number of traces combinations to obtain training data. These combinations should include combinations of workloads with different characteristics. MLCache uses $A_{1-k}[]$ as training samples and their corresponding optimal distribution solutions as labels. We randomly select a part of the massive training data to train the neural network.

Figure 3 shows a schematic diagram of the neural network used when the number of streams is n. The neural network is composed of an input layer, a hidden layer, and an output layer. The input data of the input layer is the frequency of each stream in the reused distance interval. The hidden layer is a certain number of neurons that are fully connected to the input layer and the output layer. The output layer is the cache space allocated for each stream.

The training loss is continually decreasing and gradually converging to a lower value, indicating that the training model tends to fit the target model, and that the network model can approximate the optimal allocation scheme. The lower the loss, the more fits our expectations.

3.2.3 MLCache in NVMe SSDs. Whenever a write request from the host arrives, the reuse distance module calculates the reuse distance of the request. When it is within the statistical range, the queue ID and reuse distance are queued, and the elements at the tail of the Recent Queue are dequeued. MLCache will compile statistics $A_{1-k}[]$ every other time period, and input them into the neural network module to decide the optimal allocation scheme. The cache control module changes the space size according to the optimal allocation scheme. When the cache allocation scheme changes, the cache control module increases or decreases the cache proportion for each stream accordingly. The flow of MLCache is shown in the Figure 4. Each stream is faced with the three options of expanding, reducing and keeping the cache space unchanged. For the expanded cache size stream, the capacity of the stream can be directly set to the allocated size. The way to reduce the cache is to use the classic LRU method to evict some cached elements. MLCache evict one cache page at a time until the cache size of the stream reaches the specified cache size.

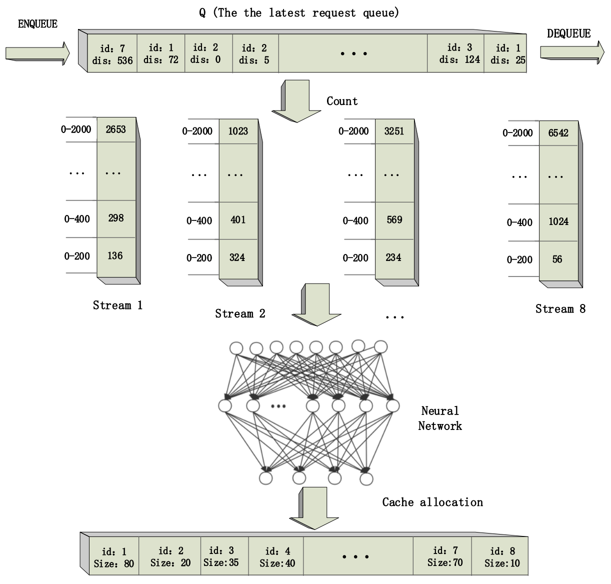


Figure 4: The flow of MLCache

Since the output layer of the neural network is not constrained, we add a penalty factor to the loss function during neural network training to control the degree of penalty for exceeding the constraints. In addition, we perform constraint checking on the values obtained from the neural network at the NVMe master control level, and allocate abnormal space (exceeding the cache capacity and non-positive values) according to the equal space distribution, which enhances the robustness of the scheme.

3.2.4 Overhead Analysis. 1)Space overhead. The space overhead of MLCache includes the space used for the distance calculation, the space occupied by the Recent Queue and the reused distance distribution array $A_{1-k}[]$. For the reuse distance calculation, we use PARDA for the calculation, and the maximum space complexity is O (M). The reused distance calculation only stores the requested address. The smallest unit of cache is a page, and the page capacity/page address is the proportion of extra overhead. Suppose that in a 64-bit operating system, the page capacity is 8192 bytes, the address length is 8 bytes, and the extra space is only $8/8192 = 0.0977\% \approx 0.1\%$. The Recent Queue takes up less space. We use 1 byte to store the stream id and 2 bytes to store the reuse distance. When the length of Recent Queue reaches 20,000, the occupied space is only 58.59KB, which can be ignored compared to hundreds of MB of cache space. The space occupied by the reused distance distribution of each stream is $k * t$ integers, which cannot exceed 2KB at most and can be ignored. The network model used for neural network training uses less than 100KB of memory, which also can be ignored.

2)Computation overhead. Most flash memory masters base on NVMe1.3 and above use Cortex R5 based chips for design. Take the consumer-level master Phison E12 produced in the fourth quarter of 2018 as an example. It has 2 Cortex R5 cores, and its average operating speed is above 300MHZ. Cortex R5 performance can reach 1.67 / 2.02 / 2.45 DMIPS / MHz. The PARDA algorithm calculates the reuse distance of M different references. The time complexity of the reuse distance is only O (log M). For M with a length of 50,000, we set it as the upper limit of our reuse distance. It only takes 12 searches to calculate it. With the addition operation, the time of its extra overhead is also below the μs level, so it can be ignored. Assuming that the maximum number of streams is 16, each stream is divided into 10 reused distance intervals. We design a neural network with an input of 160, two layers containing 256 hidden layers of neurons, and an output unit of 16. The computational cost is 110592 floating-point operations; the time required is $0.220743ms$, which is controlled within 1ms. The neural network is trained offline, and the neural network is used for calculation at the end of a period of time, so the delay caused by the overhead is extremely small and can be ignored.

4 EXPERIMENT AND ANALYSIS

In this section, we provide comprehensive experiments to evaluate the effectiveness of the MLCache.

4.1 Experimental Methodology

We evaluate MLCache on a trace-driven SSD simulator, MQSim[20]. We simulated a 256GB capacity NVMe flash-based SSD. The simulated NVMe SSD supports up to 8 streams for simultaneous reading and writing. Cache resources are scarce. The fewer shared cache resources, the better the superiority of the cache allocation algorithm. We use different cache capacities to simulate scenarios where different cache resources are scarce. We set the cache capacity to 260 pages in the 4-streams and 680 pages in the 8-streams.

Table 1: The combination of traces used in the experiment

Trace1	hm_0 hm_1	Trace6	src2_2 rsrch_0
Trace2	hm_0 src2_2	Trace7	proj_0 proj_1
Trace3	hm_1 src2_2	Trace8	usr_0 usr_1
Trace4	hm_1 rsrch_0	Trace9	usr_2 ts_0
Trace5	mds_0 proj_0	Trace10	Synthetic trace
Trace11	src2_1 proj_1 usr_0 ts_0		
Trace12	mds_0 prn_0 usr_0 ts_0		
Trace13	hm_1 src2_1 rsrch_2 stg_0		
Trace14	mds_0 rsrch_0 proj_0 proj_2		
Trace15	prn_0 rsrch_2 rsrch_0 usr_0		
Trace16	prn_0 src2_2 proj_0 usr_2		
Trace17	src2_2 rsrch_1 proj_0 ts_0		
Trace18	rsrch_2 rsrch_0 proj_1 stg_0		
Trace19	src2_2 rsrch_2 proj_3 usr_2		
Trace20	Synthetic trace		
Trace21	hm_0 mds_0 rsrch_2 rsrch_1 proj_0 usr_0 ts_0 stg_0		
Trace22	mds_0 src2_1 rsrch_2 rsrch_1 proj_0 proj_3 usr_0 ts_0		
Trace23	hm_1 mds_0 prn_0 rsrch_0 proj_1 proj_2 usr_0 usr_1		
Trace24	hm_0 src2_2 proj_0 proj_1 usr_0 usr_2 ts_0 stg_0		
Trace25	hm_1 mds_0 prn_0 src2_2 rsrch_1 proj_0 proj_1 stg_0		
Trace26	hm_1 prn_0 src2_2 src2_1 rsrch_0 usr_0 usr_2 ts_0		
Trace27	hm_0 hm_1 mds_0 rsrch_2 proj_0 usr_1 ts_0 stg_0		
Trace28	hm_0 prn_0 src2_2 src2_1 rsrch_0 proj_1 usr_0 stg_0		
Trace29	src2_2 src2_1 rsrch_2 rsrch_0 usr_0 usr_1 usr_2 ts_0		
Trace30	Synthetic trace		

The workloads. We using 16 real enterprise server traces at Microsoft Research Cambridge to evaluate the proposed scheme. In order to test the validity of the model, we randomly select the trace and combine it into multiple streams. We used 2 streams, 4 streams, and 8 streams for testing. We selected 1/4 of all real traces as test traces, and added synthetic traces to support the validity of the model. The trace combination used in the experiment is shown in Table 1. Note that the training set and the test set are different.

Schemes for comparison. Since there is no research on multi-stream cache allocation in NVMe SSD, most of the current cache allocation research is at the level of CPU multi-level cache allocation or virtual machine cache allocation[5, 9, 14, 17, 19]. Therefore, we experimentally compare the basic static evenly algorithm and the static optimal solution in the discrete time period. It is worth mentioning that the static optimal solution in the discrete-time period is the optimal solution of all algorithms for static cache allocation in the discrete-time period. Most of the current cache allocation uses static cache allocation in discrete time periods. We calculated the optimal solution in the discrete-time period through the simulator. The closer MLCache is to the optimal solution, the more successful MLCache is. In the subsequent experiments, the following schemes are implemented:

- **Evenly partition.** Evenly partition is the current NVMe SSD using a statically partitioned cache scheme. We adopt this scheme as the baseline scheme. All the experimental results were normalized to evenly partition.

- **MLCache.** MLCache is a space-efficient cache scheme based on reuse distance and machine learning for NVMe SSDs.
- **Discrete optimal partition.** Most current cache dynamic allocation schemes divide the running time into discrete time periods and allocate the cache in each discrete time period. In order to better reflect the optimization degree of the scheme, here we provide the optimal solution in discrete time periods, that is, the ideal target model. The ideal target model is to use the simulator to simulate the optimal allocation result obtained by all allocation combinations.

4.2 Experimental Results

Hit ratio: The total number of hits and the hit rate are the most important indicators of the cache. The improvement of the cache hit rate directly indicates the pros and cons of the cache allocation algorithm. Figure 8 a) compares the total number of hits of different schemes when NVMe SSD has 2 streams and cache capacity is 220 pages. The broken line indicates the hit ratio improvement rate of MLCache compared to evenly partition. Figure 8 b) shows that MLCache is suitable for different capacities. It also shows that when the cache capacity is scarce, the effect of improving the hit rate is more obvious. The improvement of the cache hit rate comes from: 1) Request local changes. 2) Changes in the size of the cache space. NVMe SSD Firmware controls cache allocation, and an excellent cache allocation algorithm can greatly utilize space.

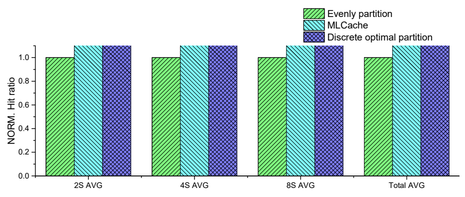


Figure 5: Comparing NORM. Hit ratio under different flow numbers.

Figure 8 compares the total number of hits and percentage increase in cache writing hit ratio on different trace combinations. It can be clearly seen that all traces have a larger improvement in hit rate compared to baseline, and all are close to discrete optimal partition. Therefore MLCache can be universally applied in a multi-stream environment, and the optimization effect is comparable to the optimal solution.

As can be seen from Figure 5, compared with evenly partition, MLCache improves the hit rate by about 24% on average, while the ideal Discrete optimal partition is optimized by about 26%. It can be seen that MLCache is very close to the ideal optimization model, and the optimization effect is significant. Figure 5 also shows that the smaller the number of streams, the more the MLCache approaches the optimal solution. This is because the increase in the number of streams will increase the difficulty of neural network training. The higher the number of streams, the more difficult it is to train the neural network, and the loss will gradually increase. But even in the case of 8 streams, MLCache is only 5% lower in hit ratio than the ideal target model.

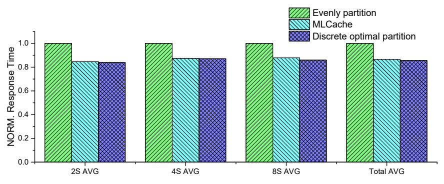


Figure 6: The average response time comparison

response time: To verify that MLCache does reduce the program latency effectively, Figure 6 compares the average response time. From Fig. 6, MLCache achieves 13.36% average response time reduction compared to Evenly partition.

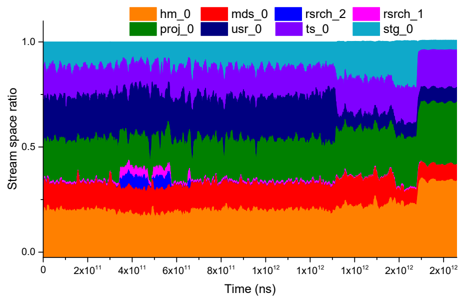


Figure 7: Percentage of cache space allocated to each stream while Trace21 is running

Figure 7 shows the percentage of cache space for each stream when Trace21 is running. The stream of hm_0, which has a lot of cache requirements, is allocated more cache space. The stream of rsrch_1, which has low demand for write cache, only allocates less cache space for a period of time. Figure 7 shows that MLCache can effectively dynamically allocate the optimal cache space for each stream.

5 RELATED WORK

In recent years, research on cache partitioning is not limited to SSD caching. The most important ones include CPU multi-level cache resource partitioning and virtual machine cache space partitioning. The cache characteristics are similar between various scenes, with certain contrast and reference.

5.1 Global cache and static cache allocation

Such as EMC VFCache, Fusion-io and vFRM[1, 2, 15]. The global sharing mode allows various cache competitors to freely compete for cache resources. The more requests a competitor initiates, the larger the cache resources will occupy, which is equivalent to allocating cache space according to the request frequency. This is likely to cause a situation where a few competitors will run out of competition for cache resources. However, static resource allocation cannot dynamically adjust the size of the cache space in real-time according to the change of the data request mode and requires manual effort to manually set it to a better state.

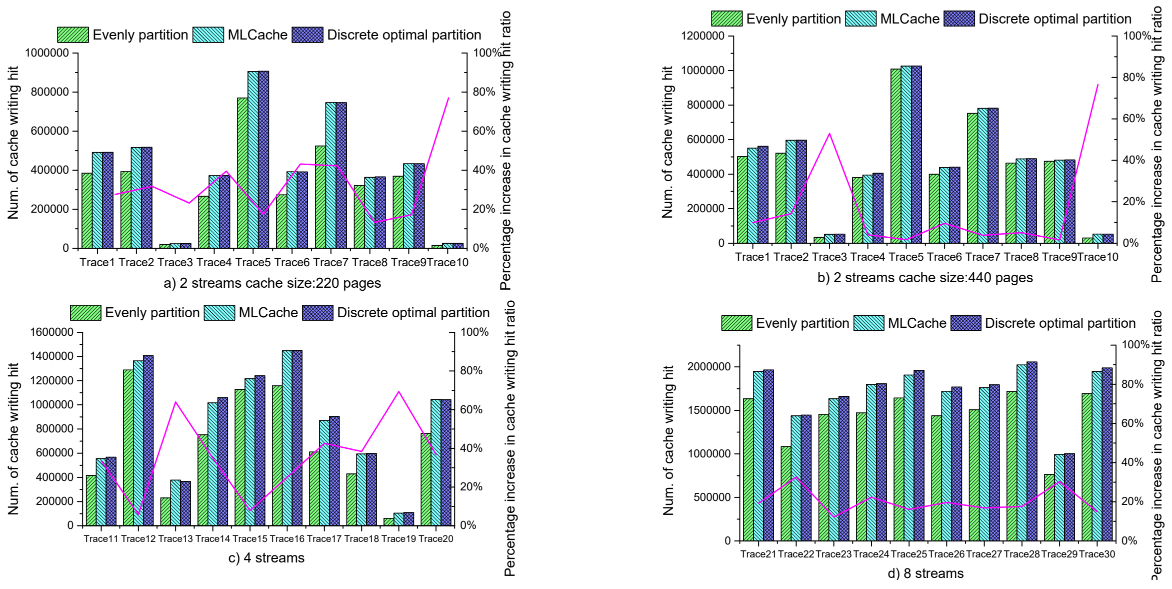


Figure 8: Comparing the total number of hits and percentage increase in cache writing hit ratio under different trace combinations

5.2 Relevant research on reuse distance

Reuse Distance as a metric for cache behavior proposes the key impact of reuse distance on cache behavior[8]. Reuse distance can help optimize cache behavior at the compiler level, hardware level, and algorithm level. Reuse Distance-Based Cache Hint Selection proposes to select the cache hint based on the data position of the instruction[9].

5.3 Dynamic cache allocation scheme based on locality

Argon proposed that multiple services use the cache dynamic partition according to the service access mode[21]. The scheme configures the efficiency of each service to maintain the efficiency achieved when it has a storage server as the score R. Argon sets the space allocated to each service to R is worth the smallest amount. Simultaneously use online simulation cache to determine the minimum cache space required.

S-CAVE is an SSD caching tool based on a hypervisor, which can effectively manage shared SSD caching devices of multiple VMs[16]. Its definition rECS is the ratio of the effective cache space used by the VM to the total cache space allocated to the VM.

CloudCache is a cache demand model scheme based on reused working set[6]. It uses workload reuse intensity to estimate cache space requirements and reduces the number of writes by capturing time locality.

ECI-Cache is a scheme for dividing cache resources for virtualization platforms[4]. It mainly uses reuse distance as the basis for division, and also considers the type of cache request, and defines the reuse distance in a specific request mode as the effective reuse distance.

The Animial class algorithm roughly divides the program into turtle (low cache usage), sheep (low cache miss rate space)[22]... The type of division is relatively rough, but it is simple and easy to use.

To sum up, CloudCache and ECI-Cache are based on the reuse distance to a certain extent. The objects they face are virtual machine resource division. The method of establishing the model is to use the mathematical model of the distance according to the reuse distance. The future situation is predicted without considering the relative strength of resource competitors. Combining reuse distance, relative request strength, and prediction of the future stage, we propose an MLCache scheme based on reuse distance, a scheme more suitable for partitioning SSD cache resources.

6 CONCLUSION

This work proposed MLCache, a space-efficient cache scheme based on reuse distance and machine learning for NVMe SSDs. MLCache allocates efficient cache space for each I/O queue independently according to the reuse distance distribution and relative request frequency. MLCache uses trained neural networks to model through training, effectively approximating the ideal optimal cache allocation scheme. Experimental results show MLCache improves the write hit ratio of the SSD by 24% compared to baseline and achieves response time reduction by 13.36% when compared with baseline. MLCache is 96% similar to the ideal model.

ACKNOWLEDGMENTS

This work is supported in part by NSFC (61902136 and 61932010), National Key R&D Program of China (2019YFB1705903), Hubei Provincial Natural Science Foundation of China (2019CFB119), Fundamental Research Funds for the Central Universities (2019kfyXJJS090).

REFERENCES

[1] 2005. Fusionio ioTurbine. http://web.sandisk.com..

[2] 2012. Squinting through the Glare of Project Lightning. http://wikibon.org/wiki/v/Squinting_through_the_Glare_of_Project_Lightning.

[3] 2020. NVMe 1.4a Specification. https://nvmexpress.org/developers/nvme-specification/.

[4] Saba Ahmadian, Onur Mutlu, and Hossein Asadi. 2018. ECI-Cache: A High-Endurance and Cost-Efficient I/O Caching Scheme for Virtualized Platforms. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2 (04 2018), 1–34. https://doi.org/10.1145/3179412

[5] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, and Eric Schrock. 2013. Janus: Optimal Flash Provisioning for Cloud Storage Workloads. In *Usenix Technical Conference*.

[6] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. 2016. CloudCache: On-demand Flash Cache Management for Cloud Computing. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 355–369. https://www.usenix.org/conference/fast16/technical-sessions/presentation/arteaga

[7] IA Basheer and M Hajmeer. 2000. Artificial neural networks: fundamentals, computing, design, and application. *Journal of Microbiological Methods* 43 (2000), 3–31.

[8] Kristof Beyls and Erik H. D'Hollander. 2001. Reuse Distance as a Metric for Cache Behavior. In *Iasted Conference on Parallel & Distributed Computing & Systems*.

[9] Kristof Beyls and Erik H. D'Hollander. 2002. *Reuse Distance-Based Cache Hint Selection*. Springer Berlin Heidelberg.

[10] Chandranil Chakraborttii, Vikas Sinha, and Heiner Litz. 2018. SSD QoS Improvements through Machine Learning. 511–511. https://doi.org/10.1145/3267809.3275453

[11] Jinhua Cui, Weiguo Wu, Yinfeng Wang, and Zhangfeng Duan. 2014. PT-LRU: A Probabilistic Page Replacement Algorithm for NAND Flash-based Consumer Electronics. *IEEE Transactions on Consumer Electronics* 60, 4 (2014), 614–622. https://doi.org/10.1109/TCE.2014.7027334

[12] Jinhua Cui, Youtao Zhang, Weiguo Wu, Jun Yang, Yinfeng Wang, and Jianhang Huang. 2018. DLV: Exploiting Device Level Latency Variations for Performance Improvement on Flash Memory Storage Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 8 (2018), 1546–1559. https://doi.org/10.1109/TCAD.2017.2766156

[13] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[14] Ricardo Koller, Ali Jose Mashtizadeh, and Raju Rangaswami. 2015. Centaur: Host-side SSD Caching for Storage Performance Control. In *2015 IEEE International Conference on Autonomic Computing (ICAC)*.

[15] D. Liu, N. Mi, J. Tai, X. Zhu, and J. Lo. 2014. VFRM: Flash Resource Manager in VMware ESX Server. In *2014 IEEE Network Operations and Management Symposium (NOMS)*. 1–7.

[16] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. 2013. S-CAVE: Effective SSD caching to improve virtual machine storage performance. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*. 103–112.

[17] Fei Meng, Li Zhou, Xiaosong Ma, Sandeep Uttamchandani, and Deng Liu. 2014. vCacheShare: Automated Server Flash Cache Space Management in a Virtualization Environment. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 133–144. https://www.usenix.org/conference/atc14/technical-sessions/presentation/meng

[18] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. 2012. PARDA: A fast parallel reuse distance analysis algorithm. In *IEEE International Parallel & Distributed Processing Symposium*.

[19] G Suh. 2004. Dynamic Cache Partitioning for CMP/SMT Systems. *The Journal of Supercomputing - TJS* (01 2004).

[20] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 49–66. https://www.usenix.org/conference/fast18/presentation/tavakkol

[21] Matthew Wachs, Michael Abdelmalek, Eno Thereska, and Gregory R Ganger. 2007. Argon: Performance Insulation for Shared Storage Servers (CMU-PDL-06-106). (2007).

[22] Yuejian Xie and Gabriel Loh. 2008. Dynamic classification of program memory behaviors in CMPs. (01 2008).

[23] H. Xu and F. Mueller. 2018. Work-in-Progress: Making Machine Learning Real-Time Predictable. In *2018 IEEE Real-Time Systems Symposium (RTSS)*. 157–160.

[24] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, and Vijay Balakrishnan. 2015. Performance Analysis of NVMe SSDs and their Implication on Real World Databases. In *the 8th ACM International Systems and Storage Conference*.