An Efficient Resource-Optimized Learning Prefetcher for Solid State Drives

Rui Xu, Xi Jin, Linfeng Tao, Shuaizhi Guo, Zikun Xiang, Teng Tian
Key Laboratory of Strongly-Coupled Quantum Matter Physics, Chinese Academy of Sciences, School of Physical Sciences, University of Science and Technology of China. Hefei, Anhui, China
{xray, jinxi, tlf, dybjxmg, xzk372, tianteng}@mail.ustc.edu.cn

***Abstract*—In recent years, solid-state drives (SSDs) have been widely deployed in modern storage systems. To increase the performance of SSDs, prefetchers for SSDs have been designed both at operating system (OS) layer and flash translation layer (FTL). Prefetchers in FTL have many advantages like OS-independence, easy-using, and compatibility. However, due to the limitation of computing capabilities and memory resources, existing prefetchers in FTL merely employ simple sequential prefetching which may incur high penalty cost for I/O access stream with complex patterns. In this paper, an efficient learning prefetcher implemented in FTL is proposed. Considering the resource limitation of SSDs, a learning algorithm based on Markov chains is employed and optimized so that high hit ratio and low penalty cost can be achieved even for complex access patterns. To validate our design, a simulator with the prefetcher is designed and implemented based on Flashsim. The TPC-H benchmark and an application launch trace are tested on the simulator. According to experimental results of the TPC-H benchmark, more than 90% of memory cost can be saved in comparison with a previous design at OS layer. The hit ratio can be increased by 24.1% and the number of times of misprefetching can be reduced by 95.8% in comparison with the simple sequential prefetching strategy.**

I. INTRODUCTION

In recent years, flash-based solid-state drives (SSDs) have occupied a large share of the digital storage device market and been widely used in laptop, desktop and enterprise server markets[1]. Compared to traditional hard disk drives (HDDs), SSDs have many advantages on density, input/output operations per second (IOPS), etc. Consequently, SSDs are regarded as the next-generation storage devices.

In an SSD, a microcontroller and a DDR memory, in addition to a NAND flash array, a flash controller, and a host interface, are used. The flash translation layer (FTL) is usually implemented within the microcontroller in the form of firmware[2]. The memory size and processing power are limited for the sake of low cost and low power consumption.

SSDs are faster than HDDs but much slower than main memory. To narrow the huge performance gap between main memory and storage devices, prefetching mechanisms are introduced. When a specific pattern in an I/O access stream is detected, the prefetcher assumes that the pattern will continue, and data predicted by this pattern are loaded into main memory for subsequent use. If the prediction is correct, data will be ready in the main memory when needed. Thus, the performance gap is narrowed. However, prefetching has some disadvantages. If the prediction is incorrect, unnecessary data are fetched (hereinafter, *misprefetching*). In addition to extra memory cost and power consumption, an increase in memory traffic can lead to an increase in memory latency. To reduce the high penalty cost of misprefetching, more prudent prefetching is required.

In this paper, an efficient resource-optimized learning prefetcher implemented in FTL is proposed, which aims at improving the hit ratio of prefetching, reducing the penalty of misprefetching, and improving read performance with the resource limitation of SSDs considered. The key idea of our design is to reduce the amount of resource usage of Markov chain prediction and implement a learning prefetcher based on the optimized algorithm in FTL. With a better prediction of access patterns, the hit ratio can be increased and the penalty of misprefetching can be reduced. Considering limited resources in SSDs, the proposed prefetcher is optimized in terms of memory consumption, computing complexity, and code size, with nearly no loss of performance.

The rest of this paper is organized as follows. In Section II, related work of our design is presented. The design of our optimized prefetcher is detailed in Section III. Experimental results are presented in Section IV. And this paper is concluded in Section V.

II. RELATED WORK

A. Prefetchers and Caches in FTL

Researches have been conducted on prefetchers and caches in FTL. DFTL[3] introduces cache mechanics for page-level mapping table to reduce memory cost. Cab-FTL[4] improves the performance of SSDs by reducing the size of the mapping table and uploading the mapping table to a DRAM. Ref. [3, 4] both aim at lowering memory consumption of the mapping table and improving the performance of SSDs. But actual data are left in the flash memory with no cache or prefetcher for it. P3Stor[5] implements an SSD for an enterprise-scale storage system. A prefetcher employs sequential prefetching is implemented in its FTL.

B. Prefetchers Based on Markov Chains

Prefetching based on Markov chains used for web data[6], cache[7] and storage[8, 9] has been researched for many years. Prefetchers based on Markov chains are always organized according to files or other structures in specific applications. However, information about these structures generally cannot be acquired in FTL except for a file-system-aware FTL which is costly. Besides, accesses across files or structures cannot be predicted. Specifically, Lynx[9] implements a prefetcher for SSDs based on Markov chains at OS layer in Linux kernel. However, memory consumption is too large even for hosts, making it impractical to implement Lynx in FTL.

III. Design and Implementation

A. Prefetching based on Markov Chains

In probability theory and related fields, a Markov process is a special type of stochastic process distinguished by a certain Markov property; a Markov chain is a Markov process with a denumerable number of states[10]. By associating each page with a state, an I/O request sequence can be modeled as a Markov model. A Markov chain can be fully described by a transition matrix, namely, a table where rows are labeled with states and columns are labeled with their next states, which is easy to represent in computers.

For a Markov model of an I/O request sequence, the probabilities of state transformations can be acquired from statistical analysis of the I/O request sequence. A prefetcher based on Markov chains has two phases, a *training phase* and a *predicting phase*. During the training phase, the transition matrix is constructed by recording and analyzing a current read request and a previous read request continuously. During the predicting phase, the prefetcher retrieves the transition matrix and prefetches a page with the highest probability of transition from the currently accessed page.

Markov chains have already been adopted for prefetching for SSDs in Lynx[9], as mentioned above. Lynx is implemented at OS layer and the direct use of Markov chains makes the memory consumption too large even for host systems. If the total size of the dataset is N, the space complexity of a prefetcher based on Markov chains is $O(N^2)$, which means memory consumption will increase faster and faster as N scales up. Furthermore, during the predicting phase, the time complexity of finding the page with the highest transition probability is $O(N)$. Both the space complexity and the time complexity of Lynx are not acceptable even for host systems, let alone for the microcontroller and the DDR memory in SSDs. Thus, a resource-optimized design is necessary for a prefetcher in FTL.

B. Resource-optimized Prefetcher

As described in Section I, the penalty of misprefetching is significantly high. For prefetchers based on Markov chains, only pages with the highest probabilities are prefetched during the predicting phase. But for transition probability distribution that does not have a page with *absolute advantage*, the ratio of misprefetching will be extremely high. Therefore, our first optimization is that only pages with absolute advantage will be prefetched. Only addresses of the pages with absolute advantage are useful for the prefetcher. Our second optimization aims at a resource-optimized algorithm for finding the pages with absolute advantage, which is shown in Algorithm 1. In this algorithm, a predicting record (PR) will be recorded for each page p_i presented in the I/O request sequence of SSDs. Each PR comprises four variables, respectively, page number p, size n, score s and the total access number N of p_i. Note that the *conditions* don't represent requests in the I/O request sequence, but the I/O requests immediately after the request accessing p_i. And the *condition sequence* is a sequence of conditions for a specific page. There is a common constant, *accepting threshold T*, which is used for all pages. T is set according to the definition of absolute advantage. Taking absolute advantage as a transition probability no less than P, T will be a difference between the transition probability of the page with absolute advantage and the total transition probability of the rest of the pages. If p is the page with absolute advantage throughout and its transition probability is P_p, there will be $(1-P_p)N_2$ subtractions and PN_2 additions, and thus s_x/N_2 will be $P_p-(1-P_p)$. It is true for all cases except for *accidental subtractions*. An accidental subtraction means the case that p is accidentally occupied by a page without absolute advantage while another page without absolute advantage comes, leading to a subtraction. When accidental subtractions occur, s_x/N_2 will be greater than $P_p-(1-P_p)$. For example, if T is set to 20% (i.e., P is 60%) and the condition sequence is as follows: $\{p_1, p_2, p_2, p_3, p_1, p_1\}$, at the end of the condition sequence, p will be p_1 and s_x/N_2 will be 33.3%. In this case, p_1 will be accepted although its transition probability is 50% which is less than P, that is, an accidental subtraction happened between p_2 and p_3. Accidental subtractions make this algorithm theoretically incorrect. There will be a transition probability range called *uncertain region*, which should not be accepted according to the definition of absolute advantage, however, whether it is accepted or not is uncertain in this algorithm. But, the probability of accepting a page decreases rapidly as P decreases. Therefore, the uncertain region is merely quite a narrow region. Our original intention for reducing the penalty of misprefetching will not be affected since the penalty incurred by the uncertain region will be low enough to neglect when the uncertain region is narrow enough. This will be validated in Section IV. Taking the considerable advantages into account, this algorithm is feasible for our design.

For the resource-optimized algorithm, to realize a function similar to that of a previous Markov chain, a *predicting unit* (PU) consisted of N PRs is needed, and its space complexity is $O(N)$. As a result, monitoring the whole disk and recording cross-file transitions becomes possible and this is highly

Algorithm 1. Resource-optimized algorithm

```
Input: a condition sequence $CS$ for a specific page
1:   $pr.p \leftarrow p_1, pr.n \leftarrow 0, pr.s \leftarrow 0, pr.N \leftarrow 0$
2:   for $c_i$ in $CS$
3:     $pr.N \leftarrow pr.N+1$
4:     if $pr.p = c_i.p$
5:       $pr.s \leftarrow pr.s+1, pr.n \leftarrow c_i.n$
6:     else $pr.s > 0$ then
7:       $pr.s \leftarrow pr.s-1$
8:     else
9:       $pr.p \leftarrow c_i.p, pr.n \leftarrow c_i.n, pr.s \leftarrow 1$
10:    end if
11:  end for
12:  if $pr.s/pr.N \geq T$
13:    return $pr$   // accept
14:  else
15:    return NULL // reject
16:  endif
```

conducive to reducing the miss ratio. Compared to previous prefetcher design whose space complexity is *$O(N^2)$*, memory consumption of the optimized design is affordable for SSDs. For reducing memory cost, a *super PU* monitoring the whole disk is divided into some PUs and each PU manages a fixed number of pages with successive logical addresses (hereafter, *PU blocks*). The third optimization is that a PU will be created only when its PU block is accessed. As the address of recorded transition is not limited, this scheme is functionally identical to the super PU. Besides, previous design needs to find the page with the highest transition probability and the time complexity is *$O(N)$*, while for the optimized design, no computation is needed during the predicting phase. Simple additions and subtractions during the training phase are also an affordable amount of computation for SSDs.

Because total access number N is recorded, our design can distinguish cold and hot data without extra cost. If 1% of the whole disk space is set as hot data region, for the size of the dataset is 100GB, theoretically speaking, the needed memory space will be no more than 1.25MB even if all PUs are created, and this memory space is negligible even for the DDR memory inside an SSD.

C. System Architecture

Fig. 1 shows the system architecture of an SSD with a resource-optimized prefetcher implemented in its FTL. Rectangles connected through solid lines represent hardware components and the FTL is a software/firmware component. Colored rectangles represent components of the prefetcher. The prefetcher comprises a *sequential pattern detector (SPD)* and a *learning unit (LU)*. Only accesses without a sequential pattern can be passed to the LU. The SPD is organized as a bitmap and the LU is managed as an array of pointers of PUs. If a PU has not been created, the corresponding pointer will be *NULL*. Each PU comprises an array of PRs. If a PR has no record or has been rejected, it will be *invalid*. A simulator of the system described above is implemented based on Flashsim[11], a widely used open-source SSD simulator.

IV. Experimental Results

A. Validation of Resource-Optimized Algorithm

In this paper, absolute advantage is defined as having a transition probability greater than 75%, i.e., P is set to 75% and T is set to 50%. There are another four test conditions, that is, the number of pages presented in the condition sequence

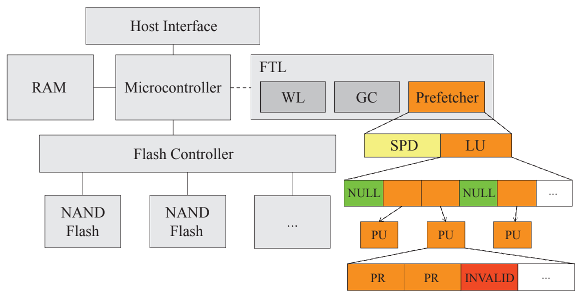

Fig. 1. System architecture of an SSD with a resource-optimized prefetcher implemented in its FTL

N_p, the highest transition probability of the pages P_h, the length of the condition sequence L, and the number of repetition times N_r. The transition probabilities of pages except for the page with absolute advantage are randomly set and the sum of transition probabilities of all pages is 1. Then the condition sequence is randomly generated according to the transition probabilities. Different condition sequences are processed based on the algorithm and results of testcases are probabilities that pages are accepted when P_h is less than P. All the results are listed in Table I and Table II.

According to the results, we can see that the probabilities decrease exponentially as P_h decreases. This means that the uncertain region is quite narrow. Note that the minimal interval of transition probability is 10% when L equals to 10. The probability increases slowly as N_p increases. Results are sufficient to demonstrate that the resource-optimized algorithm is feasible for our design.

B. Evaluation Setup

Available specifications for commercial SSDs are so insufficient that we cannot get all information needed for building an accurate model. Therefore, we configure our simulator based on assumptions and available information. The page size for actual data is 4KB and the number of pages in a block is 256. The time for reading a page from flash memory is set to 20us. The time for reading a page from the DDR memory is 800ns, 25 times faster than flash memory. The size of a PU block is set to 1024 pages, and the size of the cache is 128KB. For simplicity, page-level mapping design is chosen for testing the TPC-H benchmark[12] and application launch. The size of the dataset of the TPC-H benchmark is 100GB, and for application launch testing, MATLAB R2017a is selected. Traces of both tests are acquired by Blktrace 2.0.0.

C. Evaluation Results for TPC-H Benchmark

The results for TPC-H benchmark are shown in Fig. 2, Fig. 3 and Fig. 4. In Fig. 2, for Read-ahead, hit ratios of TPC-H queries are about 75% and the average hit ratio is 74.1%, while for our design, most of the hit ratios of TPC-H queries are greater than 99.9% and the average hit ratio is 98.2%. The hit ratio is increased by 24.1% on average and the miss ratio is reduced by 93%. This increase of hit ratios mainly benefits

TABLE I. Verification Results for the Algorithm (part one)

Testcase[a]		P_h				
		75%	*74%*	*73%*	*72%*	*71%*
N_p	*3*	1	7.5405%	0.7523%	0.0750%	0.0081%
	6	1	10.1206%	1.1224%	0.1181%	0.0138%
	20	1	11.1049%	1.2612%	0.0161%	0.0012%
	50	1	11.2601%	0.1473%	0.0150%	0.0018%

a. $L = 100$, $P = 75\%$, $T = 50\%$, $N_r = 1{,}000{,}000$

TABLE II. Verification Results for the Algorithm (part two)

Testcase[a]		P_h			
		80%	*70%*	*60%*	*50%*
N_p	*3*	1	4.6049%	0.1735%	0
	6	1	6.9620%	0.3604%	0

a. $L = 10$, $P = 75\%$, $T = 50\%$, $N_r = 1{,}000{,}000$

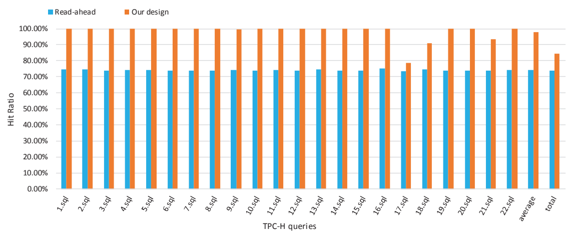

Fig. 2. Hit ratio for TPC-H benchmark

from the powerful predicting capability of our design for complex patterns. The hit ratio of our design for query 17 and query 18 is much less than that for other queries. The reason is that many pages without absolute advantage are ignored by our design but are prefetched by Read-ahead, which can be confirmed by results in Fig. 3. Apparently, our design is much more efficient. On average, the number of times of misprefetching of our design is 8,365, merely 4.2% of 199,993 for Read-ahead, which means that the penalty of misprefetching is reduced significantly. As for the storage accessing time shown in Fig. 4, the efficiency of our design can be convinced as the time cost by our design is less than that of Read-ahead for nearly all queries. On average, the storage accessing time is saved by 13.8%.

To test the performance of our design, we use our prefetcher for the whole SSD instead of only for hot data. The memory space used for TPC-H benchmark with 100GB of data is 92MB during the training phase and thereafter, it can be reduced to 57MB. Compared with Lynx[9] whose memory consumption is 92MB for 10GB of data, according to the space complexity *O(N*) of our design, more than 90% of memory space is saved. If the hot data region is set as 1% of the whole disk space, the memory used will be about 570KB.

D. *Quick Application Launch*

A trace of launching MATLAB is tested. Experimental results show that storage accessing time without a prefetcher is 1.73s. For Read-ahead, the time is 2.28s, while our design costs only 0.90s, which is merely 52.0% of traditional design without a prefetcher and 39.5% of Read-ahead. We can see that for complex I/O access patterns, sequential prefetching is incapable of prefetching data correctly, and moreover, because of the awful penalty of misprefetching, SSDs with sequential prefetching are even much slower than SSDs without a prefetcher. On the contrary, our design can work very well, and the improvement is considerable.

V. CONCLUSION

In this paper, we proposed an efficient resource-optimized prefetcher which is implemented in FTL of SSDs. Memory consumption, computing complexity, and code size are all optimized in our design to implement the prefetcher within an SSD with limited resources. If the prefetcher is used for only hot data, the DDR memory space costed by our design can be as less as 570KB for 100GB of data. There are only some additions and subtractions in our design, which can be carried out easily even by a microcontroller. The code size of our design is 1740 bytes for ARMv7 instruction set, an instruction set often used in industrial and commercial application

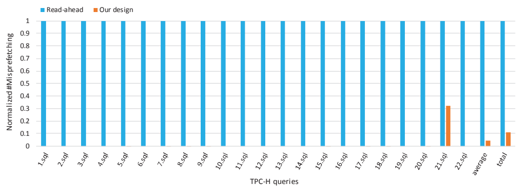

Fig. 3. Number of misprefetching for TPC-H benchmark

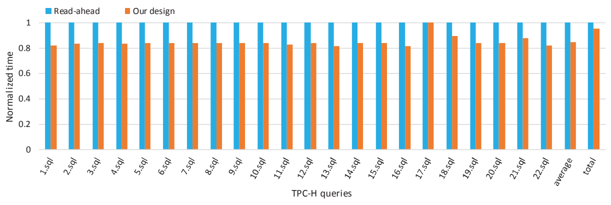

Fig. 4. Time of accessing storage for the TPC-H benchmark

scenarios of microcontrollers, and this is small enough so that our design can be implemented in FTL which is usually implemented as firmware in SSDs. These optimizations do not lead to low performance. Employing optimized Markov chain prediction, much higher performance can be achieved in our design even for I/O requests with complex patterns. Time spent on storage accessing is saved by 13.8% for the TPC-H benchmark and by 60.5% for application launching. Furthermore, our design also has other advantages, such as OS-independence, ease of deployment and use, and compatibility with general SSDs, etc.

REFERENCES

[1] G. Wong, "SSD market overview," in *Inside Solid State Drives (SSDs)*Dordrecht: Springer Netherlands, 2013, pp. 1-17.

[2] R. Micheloni, *Solid-State-Drives (SSDs) modeling*. Springer, 2017.

[3] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," *SIGARCH Comput. Archit. News,* vol. 37, no. 1, pp. 229-240, 2009.

[4] S. J. Kwon, "A cache-based flash translation layer for TLC-based multimedia storage devices," *ACM Trans. Embed. Comput. Syst.,* vol. 15, no. 1, pp. 1-28, 2016.

[5] N. Xiao, Z. Chen, F. Liu, M. Lai, and L. An, "P3Stor: A parallel, durable flash-based SSD for enterprise-scale storage systems," *Science China Information Sciences,* journal article vol. 54, no. 6, pp. 1129-1141, June 01 2011.

[6] M. B. Pal and D. C. Jain, "Web service enhancement using web pre-fetching by applying markov model," in *Communication Systems and Network Technologies (CSNT), 2014 Fourth International Conference on*, 2014, pp. 393-397: IEEE.

[7] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," in *Software, IEE Proceedings-*, 2004, pp. 96-96.

[8] R. Li, R. Guo, Z. Xu, and W. Feng, "A prefetching model based on access popularity for geospatial data in a cluster-based caching system," *International Journal of Geographical Information Science,* vol. 26, no. 10, pp. 1831-1844, 2012/10/01 2012.

[9] A. Laga, J. Boukhobza, M. Koskas, and F. Singhoff, "Lynx: a learning Linux prefetching mechanism for SSD performance model," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2016, pp. 1-6.

[10] K. L. Chung, *Markov chains*. Springer, 1967.

[11] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "FlashSim: A simulator for NAND flash-based solid-state drives," in *2009 First International Conference on Advances in System Simulation*, 2009, pp. 125-131.

[12] T. P. P. Council, "TPC-H benchmark specification," *Published at http://www. tcp. org/hspec. html,* vol. 21, pp. 592-603, 2008.