Article

A Cache Policy Based on Request Association Analysis for Reliable NAND-Based Storage Systems

Chi-Hsiu Su ^{†,‡} and Chin-Hsien Wu ^{*,†,‡}

Department of Electronic and Computer Engineering, National Taiwan University of Science and Technology
Taipei 106, Taiwan; m10702144@mail.ntust.edu.tw

* Correspondence: chwu@mail.ntust.edu.tw; Tel.: +886-2-2730-3274

† Current address: No. 43, Keelung Rd., Sec. 4, Da'an Dist., Taipei 106335, Taiwan.

‡ These authors contributed equally to this work.

Abstract: Compared with the traditional hard-disk drives (HDDs), solid-state drives (SSDs) have adopted NAND flash memory and become the current popular storage devices. However, when the free space in NAND flash memory is not enough, the garbage collection will be triggered to recycle the free space. The activities of the garbage collection include a large amount of data written and time-consuming erase operations that can reduce the performance of NAND flash memory. Therefore, DRAM is usually added to NAND flash memory as cache to store frequently used data. The typical cache methods mainly utilize the data characteristics of temporal locality and spatial locality to keep the frequently used data in the cache as much as possible. In addition, we find that there are not only temporal/spatial locality, but also certain associations between the accessed data. Therefore, we suggest that a cache policy should not only consider the temporal/spatial locality but also consider the association relationship between the accessed data to improve the cache hit ratio. In the paper, we will propose a cache policy based on request association analysis for reliable NAND-based storage systems. According to the experimental results, the cache hit ratio of the proposed method can be increased significantly when compared with the typical cache methods.

Keywords: cache; association analysis; NAND flash memory; solid-state drive (SSD)

Citation: Su, C.-H.; Wu, C.-H. A Cache Policy Based on Request Association Analysis for Reliable NAND-Based Storage Systems. *Appl. Sci.* **2021**, *11*, 6623. <https://doi.org/10.3390/app11146623>

Received: 17 May 2021

Accepted: 5 July 2021

Published: 19 July 2021

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Compared with the traditional hard-disk drives (HDDs), solid-state drives (SSDs) adopt NAND flash memory as the storage medium that has the advantages of non-volatility, fast speed, shock resistance, low power consumption, and small size. A read/write unit in NAND flash memory is a page that cannot be overwritten, and a block in NAND flash memory contains a fixed number of pages and is an erase unit. When a page needs to be updated, the updated content has to be written on another free page, where the original page will become an invalid page and the latest written page will be a valid page. When the number of free pages in NAND flash memory is not enough, the activities of garbage collection will first select a victim block, copy the victim block's valid pages to other free pages and then erase the victim block to generate a free block. Furthermore, the activities of garbage collection can reduce the performance of NAND flash memory. Therefore, DRAM can be added to SSDs as a cache to store frequently used data because DRAM has a higher access speed and better reliability than NAND flash memory. Caches in SSDs can speed up data reading and directly absorb a large amount of data written to improve the performance of SSDs. However, when the limited cache space is full, it is still an important research topic to select the appropriate victims to be replaced.

The typical cache methods (such as two-level LRU [1,2], ARC [3] and VBBMS [4]) mainly utilize the data characteristics of temporal locality and spatial locality to keep the frequently used data in the cache as much as possible. However, we observe that some specific functions may call each other and some specific data may be accessed at the same

time when users are running many different programs. There is not only temporal/spatial locality, but also certain associations between the accessed data. Therefore, we suggest that a cache policy should consider the temporal/spatial locality and the association between the accessed data to improve the cache hit ratio. In the paper, we will propose a cache policy based on request association analysis for reliable NAND-based storage systems. According to the experimental results, the cache hit ratio of the proposed method can be increased by about 14.09~35.08% when compared with the typical cache methods.

The rest of paper is organized as follows. Section 2 introduces the related work. Section 3 explains the motivation. In Section 4, we describe a cache policy based on request association analysis for reliable NAND-based storage systems. Section 5 provides a performance evaluation. Section 6 concludes the paper and suggests future work.

2. Related Work

2.1. Cache Management Methods

The current mainstream storage architectures [5–7] (e.g., SSDs or NVDIMM) usually adopt NAND flash memory as the storage medium. A read/write unit in NAND flash memory [8,9] is a page and cannot be overwritten unless its residing block is erased first. When the number of free pages in NAND flash memory is not enough, the activities of garbage collection will first select a victim block, copy the victim block's valid pages to other free pages and then erase the victim block to generate a free block. In the process of garbage collection, when a large number of page writes and erase operations are concentrated on a specific block, the block will be quickly worn out to damage the lifetime of NAND flash memory [10–13]. Therefore, DRAM is usually added to NAND flash memory as cache to store the frequently accessed data, because DRAM has a higher access speed and better reliability than NAND flash memory. So, a cache can speed up the performance of NAND flash memory by reducing the read and write requests. However, the cache capacity is limited. When the cache space is full, how to select the appropriate victims to be replaced has always been an important research topic. In the following, we will introduce some typical cache management methods. Different from the previous cache management methods that mainly consider the temporal/spatial locality, we will modify the ARC method (that will be described later) to both consider the temporal/spatial locality and the request associations between the accessed data to improve the cache hit ratio.

1. *LRU*: A least-recently-used (LRU) method [1] is a common replacement strategy based on temporal locality. Pages that have been frequently and recently accessed will have a high chance of being accessed again after a while. Therefore, when the cache space is full, an LRU method will select the least-recently-used (LRU) page as a victim and remove it. An LRU method generally uses a linked list. The newly inserted page will be at the head of the list. When any page is hit in the cache, the page will be moved to the head of the list. When the cache space is full, the least-recently-used (LRU) page at the tail of the list will be selected as a victim. However, the hot data stored in the cache could be easily evicted to cause performance degradation due to a lot of sequential accessing.
2. *Two-Level LRU*: As shown in Figure 1, a two-level LRU method [2] uses two LRU linked lists. The newly inserted page that does not exist in the cache will be placed at the head of the first list. When any page is hit in the cache, the page will be moved to the head of the second list. If the second list is full, the page at the tail of the second list will be moved to the head of the first list. If the first list is full, the page at the tail of the first list will be evicted from the list. Compared with the LRU method, when encountering a lot of sequential accesses, it will only affect the first list and will not affect the hot data in the second list. So, the two-level LRU method can effectively improve the cache hit ratio. However, different length settings of the two LRU lists will affect its performance, and how to dynamically and efficiently adjust the lengths of the two LRU lists for different workloads will be described later (e.g., ARC).

Figure 1. Two-Level LRU.

- ARC:** As shown in Figure 2, an adaptive replacement cache (ARC) [3] also uses two LRU linked lists: L1 and L2. L1 stores the pages that appear only once, and L2 stores the pages that appear two or more times. L1 is divided into T1 and B1, and L2 is divided into T2 and B2. T1 and T2 store the contents of the pages in the cache and the related metadata in the cache directory, but B1 and B2 (e.g., ghost buffers) only store the related metadata evicted from T1 and T2, respectively. Assuming that the cache can buffer c pages, the length limits of ARC are $0 \leq |L1| \leq c$, $0 \leq |L1| + |L2| \leq 2c$, $0 \leq |T1| + |T2| \leq c$, $0 \leq |B1| + |B2| \leq c$. If a newly inserted page does not exist in L1 and L2, it will be placed at the head of T1; otherwise, it will be placed at the head of T2. The space of T1 is decided by a target size p . When a new page is inserted to the cache that is full, we have to remove one victim page from T1 or T2 according to the following condition—if $|T1| \geq p$, the LRU page of T1 will be evicted and then its related metadata will be inserted to B1; otherwise, if $|T1| < p$, the LRU page of T2 will be evicted and then its related metadata will be inserted to B2. If a hit occurs in B1, the target size p of T1 will be increased by the maximum of $\{|B2|/|B1|, 1\}$. If a hit occurs in B2, the target size p of T1 will be decreased by the maximum of $\{|B1|/|B2|, 1\}$. ARC can dynamically and efficiently adjust the lengths of L1 and L2 for different workloads to achieve the best performance.

Figure 2. ARC.

- VBBMS:** As shown in Figure 3, VBBMS [4] is a typical flash-aware buffer method to combine consecutive pages in the same virtual block. When the cache space is insufficient, VBBMS will evict the entire virtual block so that the data written to flash memory can be more sequential. By the eviction of the entire virtual block, VBBMS can easily make a block to have more invalid pages and reduce the overhead of the garbage collection. In addition, because random requests and sequential requests

have completely different characteristics, VBBMS uses an LRU list to manage random requests and a FIFO queue to manage sequential requests. In addition, the parameter setting (such as the size of a virtual block and the distributions of an LRU list and a FIFO queue) could have a large impact on the results.

Figure 3. VBBMS.

Because VBBMS [4] shows that it has better hit ratio than the flash-aware cache replacement policies such as CFLRU [14], BPLRU [15], AD-LRU [16], PR-LRU [17] and CLOCK-DNV [18], we compared the proposed method with VBBMS in the experiments. CFLRU is a page-based cache replacement policy and can take the asymmetry of read/write cost into consideration. CFLRU splits the cache into working region and clean-first region, where the LRU clean pages in the clean-first region are evicted first, otherwise the LRU dirty pages in the clean-first region are evicted. BPLRU is a block-level cache replacement policy and maintains some clusters by the LRU mechanism, where a cluster contains a series of consecutive pages that belong to a data block. AD-LRU is a page-based cache replacement policy and uses an adaptive mechanism to buffer pages in the hot and cold LRU queues. AD-LRU considers the frequency and recency of page references and also takes the asymmetry of read/write cost into consideration when replacing pages. PR-LRU is a page-based cache replacement policy and uses a probability of reference (PR) to predict whether a page may be referenced in the future. CLOCK-DNV proposes a hybrid write buffer architecture that consists of DRAM and non-volatile memory. CLOCK-DNV is managed differently to DRAM and non-volatile memory: CLOCK-D for DRAM at a page granularity and CLOCK-NV for non-volatile memory at a block granularity.

2.2. Association Analysis

Association analysis is one of the commonly used methods in the field of data mining [19,20], and has been used for pattern-based address mapping in SSDs [21]. Assume that A and B appear together in several groups in the database due to the associations between A and B. Therefore, we can use association analysis to analyze the data in the data groups and gain benefits. For example, in the business field, if customers often buy product B while purchasing product A, the relationship (between product A and product B) can be used for product promotion or recommendation in the market.

Association analysis contains two steps. The first step is to find out frequent item sets that appear at the same time. The second step is to filter out strong association rules from the frequent item sets. The strong association rule is decided by two parameters of support and confidence. Assume that A and B are in the data group, and we want to

determine the association rule (i.e., $A \rightarrow B$) to represent that when A occurs, B will also occur. Support represents the probability of A and B appearing at the same time, that is, $P(A \cup B)$. Confidence is the conditional probability $P(B|A)$, that is, when A occurs, B also occurs. If the association rule can meet the minimum support and confidence, it will be called the strong association rule. Therefore, how to find out the frequent item sets is a key step when executing association analysis. Apriori [19] and FP-Growth [20] are two commonly used algorithms for searching frequent item sets. As FP-Growth can alleviate the huge time and space overhead caused by Apriori, we adopt FP-Growth to perform the association analysis in the paper.

As shown in Figure 4, there are five groups and the five groups are (f,a,c,d,g,i,m,p), (a,b,c,f,l,m,o), (b,f,h,j,o), (b,c,k,s,p) and (a,f,c,e,l,p,m,n). Assume that the minimum support is 3, the five groups will sort their items by their supports in all groups and remove those items whose support is less than 3 and become (f,c,a,m,p), (f,c,a,b,m), (f,b), (c,b,p) and (f,c,a,m,p), as shown in Figure 4. During the building process of the FP tree, if an item is encountered, its count will be increased by 1; otherwise, the item will be branched if it does not exist in the FP tree. For example, (f:4) in the FP tree denotes that f is encountered four times. Finally, we can use a header table to link the same item in the FP tree by the dotted line. Based on the FP tree, we can find the frequent item sets. For example, according to the header table, we can find that m is related to two paths in the FP tree (e.g., <f, c, a, m> and <f, c, a, b, m>) and the number of occurrences with m in the FP tree are that (f,m) appears three times, (c,m) appears three times, (a,m) appears three times and (b,m) appears one time. As the number of occurrences of (b,m) is 1 and less than the minimum support (e.g., 3), we can remove b and obtain three frequent item sets such as (f,m), (c,m) and (a,m). Therefore, we can also obtain three association rules such as (f,m), (c,m) and (a,m).

Figure 4. An example of FP-Growth.

3. Motivation

To improve the performance of NAND-based storage systems, DRAM is usually added to NAND-based storage systems as cache to speed up data reading and directly absorb a large amount of data written, because DRAM supports in-place updates and has higher access speed and better reliability than NAND flash memory. When the limited cache space is full, how to select the appropriate victims to be removed from the cache has a great impact on the performance of NAND-based storage systems. The typical cache methods (such as LRU, ARC and VBBMS) are mainly to distinguish the important data

based on temporal locality or spatial locality. However, we observe that there are certain associations between the accessed data for some specific workloads in the experiments. Therefore, we suggest that a cache policy should consider the temporal/spatial locality and the association rules between the accessed data to improve the cache hit ratio. The above observations are the motivation behind the paper.

4. A Cache Policy Based on Request Association Analysis for Reliable NAND-Based Storage Systems

4.1. Overview

In the paper, we will propose a cache policy based on request association analysis for reliable NAND-based storage systems. The proposed method contains two components: request association analyzer and request association cache, as shown in Figure 5. The request association analyzer is to find out strong and weak requests after performing request association analysis. With the request association analyzer, the request association cache can dynamically separate strong and weak request data and then keep the strong request data in the cache as much as possible to improve the cache hit ratio.

Figure 5. System architecture.

4.2. Request Association Analyzer

The purpose of the request association analyzer is to gain strong requests which are suitable to be placed in the data cache. The request association analyzer contains three parts: request sampling, request association analysis and request association pairs management, as shown in Figure 5.

4.2.1. Request Sampling

Request sampling is to use the starting LPN (logical page number) of each request as the sampling data, where different starting LPNs of requests are considered as different sampling data. During the request sampling is performed, a hot/cold request filter can be implemented to first filter out cold requests from the sampling data and then put the hot requests into a certain group for the future association analysis. When the amount of data in all groups meets the requirement, the request association analysis will be performed to obtain the associations between the data. After the request association analysis is completed, the oldest group will be removed for the newly generated hot requests.

4.2.2. Request Association Analysis

Request association analysis is to find out the association pairs by using the FP-Growth algorithm mentioned in Section 2.2. Generally speaking, an association rule is defined as directional such as $A \rightarrow B$ and is a conditional relationship that causes event B to occur due to the occurrence of event A. Moreover, based on our observations about workloads, we find that the order of occurrence of the related requests could be not directional. Therefore, different from the traditional FP-Growth algorithm, we only use the frequency of occurrences for determining an association rule. Therefore, we define that an association rule contains two requests' starting with LPNs to reduce the complexity of association analysis. We also call an association rule an association pair.

4.2.3. Request Association Pairs Management

Request association pairs management is to manage association pairs and select the hot association pairs from them. Request association pair management uses a two-level LRU to maintain the association pairs, where the first LRU list stores association pairs that appear once, and the second LRU list stores association pairs that appear more than two times and are considered as the hot association pairs, as shown in Figure 6. As the hot association pairs appear more than two times in the second LRU list, they usually conform to the temporal locality and can be useful in the future.

Figure 6. A two-level LRU list to maintain the hot association pairs.

After the hot association pairs are selected, we can extract each request's starting LPN from the hot association pairs and count the number of occurrences of each starting LPN in all association pairs. For example, Figure 7 shows the number of occurrences of each starting LPN in all association pairs such as (100,101), (100,200), (100,201), (101,200), (101,300), (200,201) and (200,400). Then, we can set a threshold to separate the strong requests according to the number of occurrences of each request's starting LPN in all association pairs. As shown in Figure 7, assume that the threshold is 50%, the first 50% of all requests' starting LPNs (sorted by the number of occurrences) are considered as strong requests (e.g., 200, 100 and 101) and are maintained in a strong request list. Other requests are considered as weak requests (e.g., 201, 300 and 400) and are maintained in a weak request list. Later, we will propose how to dynamically adjust the threshold (e.g., 50%) according to the current workload to improve the accuracy and performance in Section 4.3.2.

Figure 7. Strong/weak request lists.

4.3. Request Association Cache

The purpose of the request association cache is to dynamically separate strong and weak request data, and then keep the strong request data in the cache as much as possible to improve the cache hit ratio. The request association cache contains two parts: strong/weak request data management and request strength threshold to separate strong and weak request data, as shown in Figure 5.

4.3.1. Strong/Weak Request Data Management

When a request will enter the cache from the host, we will first compare whether the request's starting LPN is in the strong request list. If the request's starting LPN is in the strong request list, it is a strong request and its content can be considered as the strong request data; otherwise, the request's starting LPN is in the weak request list and its content is considered as the weak request data. Note that the strong and weak request lists are according to the results of request association pairs management and we use the starting LPN of each request to decide the strong or weak request. If the starting LPN is strong, the entire content contained in the request will be seen as the strong request data. If the starting LPN is weak, the entire content contained in the request will be seen as the weak request data. Therefore, we can modify the original ARC design to consider the strong and weak requests to improve the cache hit ratio, as shown in Figure 8. The proposed cache policy consists of two LRU lists which are a weak request data list (WL1) and a strong request data list (SL2). WL1 can be divided to two parts: WT1 and WB1, where WT1 stores the content (i.e., weak request data) and WB1 (like a ghost buffer) only stores the evicted information from WT1, not to store the content. SL2 can be also divided to two parts: ST2 and SB2, where ST2 stores the content (i.e., strong request data) and SB2 (like a ghost buffer) only stores the evicted information from ST2, not to store the content. As the total size of WT1 and ST2 is the same as the cache, if the size of WT1 is decreased, the size of ST2 will be increased, and vice versa.

Figure 8. The proposed cache policy.

When any new weak/strong request data will be input to WT1/ST2, if WT1/ST2 is full, the (LRU) weak/strong request data will be removed from WT1/ST2 and its related evicted information will be checked for existence in WB1/SB2, respectively. If the evicted information exists in WB1/SB2 (i.e., a hit in WB1/a hit in SB2), it means that the removed weak/strong request data could be a miss in WT1/ST2 so that the size of WT1/ST2 is increased by 1; otherwise, the evicted information will be insert to WB1/SB2, respectively. If WB1/SB2 is full, the LRU request information can be removed from WB1/SB2, respectively. There are two differences from the original ARC design. The first difference is that we distinguish between WL1 and SL2 based on the strong and weak request data management, not based on one or more occurrences like the original ARC design. The second difference is that the length of WL1 and SL2 is fixed because we can dynamically adjust the threshold to accurately separate the strong and weak request data according to the current workload. Note that how to dynamically adjust the threshold will be described in the following section.

4.3.2. Request Strength Threshold to Separate Strong and Weak Request Data

We propose a request strength threshold to dynamically separate strong and weak request data. We have two situations to adjust the request strength threshold. The first situation is to increase the request strength threshold. When the LRU request information will be removed from SB2, we will check whether the request's starting LPN exists in the strong request list. If the request's starting LPN is in the strong request list, it means that the request is considered as a strong request but it is removed because it has not been used for a long time. The reason is that the request strength threshold is so low that the request is easy to be regarded as the strong request. Therefore, we think that the request strength threshold for being considered the strong request should be increased. In the first situation, a raising factor T_{up} will be increased by 1.

On the other hand, the second situation is to decrease the request strength threshold. When a weak request's content is hit in WT1, a reduction factor T_{down} will be increased by 1. The reason is that the request strength threshold is so high that the request is easy to be regarded as the weak request but it might be considered as the strong request better. In the second situation, we think that the request strength threshold being considered the strong request should be decreased.

When the request association analyzer collects the hot association pairs and will separate the strong and weak request lists, we can compare T_{up} and T_{down} to dynamically determine the request strength threshold. As shown in Figure 9, the request strength threshold will be adjusted based on $Threshold = Threshold - p \times 5\%$, where p (whose initial value is 0) is a variable as the adjustment weight and the initial $Threshold$ is 100%. If T_{up} is greater than T_{down} , p is increased by 1, and will be reset to 0 once if T_{down} is greater than T_{up} . Similarly, if T_{down} is greater than T_{up} each time, p is decreased by 1, and will be reset to 0 once if T_{up} is greater than T_{down} .

Figure 9. Request strength threshold: $Threshold = Threshold - p \times 5\%$.

4.4. Execution Delay of Request Association Analyzer

As the request association analyzer could take time to perform association analysis, we tried to delay the execution of the request association analyzer as much as possible without

reducing the cache hit ratio too much. After the request sampling finished, we compared the number of hits of WT1 (in the weak request data list) and ST2 (in the strong request data list). Only when the number of hits of WT1 was larger than that of ST2 did we use the request association analyzer. If the number of hits of ST2 is larger than that of WT1, it means that the current request strength threshold and the strong request data list (SL2) are working well so that we do not need to execute the request association analyzer to reduce the additional overhead caused by association analysis. According to the experimental results, we can extend the time interval of performing association analysis by 2 times to 34 times without affecting the cache hit ratio too much.

5. Performance Evaluation

5.1. Experimental Setup and Performance Metrics

In the experiments, we used NVIDIA Jetson TX2 to implement a NAND-based storage simulator [22] with different cache sizes (i.e., 4, 8, 16, 32 and 64 MB) to run eight workload traces. Table 1 shows the specification of NVIDIA Jetson TX2. Table 2 shows the characteristics of eight workload traces which can be obtained from the website of the Global Network Storage Industry Association (SNIA). In Table 2, we can find that most workload traces contain a lot of read and write requests with different average request sizes so that the proposed method should both consider the read and write requests. MSR-usr_0 is from the Microsoft Research Cambridge enterprise server [23], which collects the I/O requests when users access data on the server. Systor'17-LUN_0, Systor'17-LUN_1, Systor'17-LUN_2, Systor'17-LUN_3, Systor'17-LUN_4, Systor'17-LUN_6 [24] collect the I/O requests from six virtual drives when users access data on the enterprise virtual desktop architecture. Financial is a collection of the I/O requests in the server when a large financial institution conducts online transactions and applications. For the association analysis, we set 10 groups to collect hot requests during the request sampling, where each group contains a set of 40 hot requests. When the amount of data in 10 groups is full, an association analysis will be performed on the data to obtain the associations between the data. After the association analysis is completed, the oldest group will be removed for the newly generated hot requests.

Table 1. Specification of NVIDIA Jetson TX2.

CPU	Dual-Core NVIDIA Denver 2 64-Bit CPU Quad-Core ARM Cortex-A57 MPCore
RAM	8 GB 128-bit LPDDR4 Memory
Storage	32 GB eMMC 5.1
OS	Ubuntu 16.04

Table 2. Characteristics of 8 workload traces.

Workload	Number of Requests	Write Ratio (%)	Average Read Request Size (KB)	Average Write Request Size (KB)
MSR-usr_0	2,237,889	59.58	40.92	10.28
Systor'17-LUN_0	1,768,140	22.24	27.54	8.84
Systor'17-LUN_1	1,390,948	27.66	20.33	16.47
Systor'17-LUN_2	3,615,901	12.38	19	16.7
Systor'17-LUN_3	2,161,973	20.49	19.5	14.38
Systor'17-LUN_4	2,496,148	17.66	28.73	22.98
Systor'17-LUN_6	2,146,801	17.09	21.76	16.55
Financial	5,334,987	76.84	2.25	3.73

In the experiments, we compared the proposed method based on request association analysis (i.e., RAA), the proposed method with execution delay of request association

analyzer (i.e., RAA-Delay), LRU, ARC and VBBMS. We measured the reduction ratio of the read and write requests to obtain the cache hit ratio under different cache sizes (i.e., 4, 8, 16, 32 and 64 MB). As the cache hit ratio is a key indicator to evaluate the performance, the higher cache hit ratio also means the better performance. Note that the proposed RAA and RAA-Delay are different from the typical cache methods (such as LRU, ARC and VBBMS) because they not only consider the temporal/spatial locality and but also the association rules between the accessed data.

5.2. Experimental Results: Improvement of the Average Hit Ratio

For MSR-usr_0, Figure 10a shows that RAA makes 42.25%, 28.77% and 35.27% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 28.76%, 17.09% and 22.75% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. For Systor'17-LUN_0, Figure 10b shows that RAA makes 60.45%, 27.68% and 40.94% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 55.80%, 24.29% and 37.04% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. For Systor'17-LUN_1, Figure 10c shows that RAA makes 39.99%, 17.99% and 27.85% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 34.60%, 13.62% and 23.01% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. For Systor'17-LUN_2, Figure 10d shows that RAA makes 29.46%, 21.65% and 24.20% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 27.98%, 20.29% and 22.81% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively.

For Systor'17-LUN_3, Figure 10e shows that RAA makes 43.07%, 18.52% and 29.26% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 33.75%, 10.89% and 20.88% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. For Systor'17-LUN_4, Figure 10f shows that RAA makes 28.68%, 14.73% and 21.39% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 24.38%, 11.05% and 17.41% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. For Systor'17-LUN_6, Figure 10g shows that RAA makes 33.18%, 16.15% and 24.82% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 30.16%, 13.59% and 22.01% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. For Financial, Figure 10h shows that RAA makes 3.58%, 2% and 4.6% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively. Furthermore, RAA-Delay makes 3.48%, 1.91% and 4.5% improvements of the average hit ratio when compared with LRU, ARC, and VBBMS, respectively.

Figure 10. Improvement of the average hit ratio.

We summarize the improvement of the average hit ratio in Table 3. For MSR-usr_0, Systor'17-LUN_0, Systor'17-LUN_1, Systor'17-LUN_2, Systor'17-LUN_3, Systor'17-LUN_4 and Systor'17-LUN_6, RAA and RAA-Delay can significantly improve the average

hit ratio under different cache sizes (i.e., 4, 8, 16, 32 and 64 MB), when compared with LRU, ARC, and VBBMS. This is because the seven workload traces contain more certain associations between the accessed data so that RAA and RAA-Delay can find the association rules and keep them as the strong request data in the cache as much as possible. For Financial, RAA and RAA-Delay only slightly improve the average hit ratio under different cache sizes (i.e., 4, 8, 16, 32 and 64 MB), when compared with LRU, ARC, and VBBMS. This is because Financial could contain more random accesses and less certain associations between the accessed data so that RAA and RAA-Delay are a little better than LRU, ARC, and VBBMS. As a result, we must point out that RAA and RAA-Delay consider the temporal/spatial locality and the association rules together so that they can improve the cache hit ratio when the workload traces tend to certain associations between the accessed data.

Table 3. Improvement of the average hit ratio.

Workload	RAA when Compared with LRU, ARC, and VBBMS	RAA-Delay when Compared with LRU, ARC, and VBBMS
MSR-usr_0	42.25%/28.77%/35.27%	28.76%/17.09%/22.75%
Systor'17-LUN_0	60.45%/27.68%/40.94%	55.80%/24.29%/37.04%
Systor'17-LUN_1	39.99%/17.99%/27.85%	34.60%/13.62%/23.01%
Systor'17-LUN_2	29.46%/21.65%/24.20%	27.98%/20.29%/22.81%
Systor'17-LUN_3	43.07%/18.52%/29.26%	33.75%/10.89%/20.88%
Systor'17-LUN_4	28.68%/14.73%/21.39%	24.38%/11.05%/17.41%
Systor'17-LUN_6	33.18%/16.15%/24.82%	30.16%/13.59%/22.01%
Financial	3.58%/2%/4.6%	3.48%/1.91%/4.5%
(Normalized to 4MB LRU)		

5.3. Discussion and Overhead Analysis

When a cache miss occurs, its performance penalty includes the extra time to access DRAM-based structures. However, DRAM has a higher access speed (e.g., about 20 GB/s) than NAND flash memory because the read and write time of a 16 KB page in NAND flash memory [25,26] are about 80–90 us and 340–900 us. Therefore, the performance penalty of a DRAM cache miss could cost about 100 ns, but a DRAM cache hit can avoid the access time of a 16 KB page in NAND flash memory and save about 80–900 us. Even if the cache hit ratio is only 1%, the time saved can be $N * 0.01 * 80 \text{ us} = N * 800 \text{ ns}$, but the performance penalty of 99% cache misses could be $N * 0.99 * 100 \text{ ns} = N * 99 \text{ ns}$, where N represents the access count to the cache and 80 us represents the read time of a 16 KB page in NAND flash memory. In fact, the actual cache hit ratio of the proposed method under different cache sizes (i.e., 4, 8, 16, 32 and 64 MB) for the eight workload traces is from 2.21% to 73.22%. As a result, the benefit of the cache hits is greater than that of the cache misses. That is why the proposed method and the previous typical methods (such as LRU, ARC, and VBBMS) have to improve the cache hit ratio.

We used the Python language and the embedded NVIDIA Jetson TX2 board to implement the proposed RAA and RAA-Delay. Based on the specification in Table 1, NVIDIA Jetson TX2 has enough ability to support the Python environment required about 2 GB main memory and act as a cache accelerator to assist SSDs of large enterprises. According to the results in Table 4, we observed that the average computation time by RAA or RAA-Delay is reasonable and short (e.g., about 0.3 s) when compared to the computation interval. We also observed that RAA-Delay can extend the computation interval by about 2 times to 34 times so that the computation overhead caused by the association analysis can be reduced. As a result, the proposed method based on request association analysis is practical and efficient.

Table 4. Overhead analysis.

Workload	RAA Ave. Computation Interval	RAA-Delay Ave. Computation Interval	Ave. Computation Time of Association Analysis
MSR-usr_0	21.15	721.85	0.31
Systor'17-LUN_0	0.97	2.604	0.31
Systor'17-LUN_1	0.87	3.146	0.32
Systor'17-LUN_2	0.96	1.50	0.32
Systor'17-LUN_3	0.82	1.87	0.32
Systor'17-LUN_4	0.78	1.34	0.32
Systor'17-LUN_6	1.03	1.69	0.32
Financial	0.80	1.03	0.31

(Unit: Sec)

6. Conclusions

In the paper, we propose a cache policy based on request association analysis for reliable NAND-based storage systems. The proposed method contains two components: a request association analyzer and request association cache. The request association analyzer is to find out strong requests by performing request association analysis, and then the request association cache can dynamically separate strong and weak request data according to the selected strong requests. Furthermore, the request association cache can keep the strong request data in the cache as much as possible to improve the cache hit ratio. After the cache hit ratio is increased, we can reduce a lot of pages written to NAND flash memory and also reduce the activities of garbage collection. As the activities of garbage collection contain a large amount of valid pages written and time-consuming erase operations to reclaim free space, the proposed method can keep the strong request data in the cache to improve the performance of NAND flash memory. In addition, to reduce the overhead of the request association analyzer, we also propose delaying the execution of the request association analyzer as much as possible without reducing the cache hit ratio too much. When compared with the typical cache methods (such as LRU, ARC and VBBMS), the proposed method can improve the average cache hit ratio by about 35.08%, 18.44% and 26.04%, respectively. Additionally, the proposed method with execution delay of the request association analyzer can also improve the average cache hit ratio by about 29.86%, 14.09% and 21.3%, respectively, when compared with LRU, ARC and VBBMS. Furthermore, the proposed method with execution delay of the request association analyzer can extend the computation interval by about 2 times to 34 times so that the computation overhead caused by the association analysis can be reduced. We can also show that the average computation time of the association analysis is reasonable and short (e.g., about 0.3 s) when compared to the computation interval so that the proposed method based on request association analysis is practical and efficient.

For future work, we will investigate how to reduce the overhead of the request association analyzer and also improve the hit ratio of the request association cache. Therefore, we should find a better method to extend the computation interval of the association analysis without reducing the cache hit ratio too much.

Author Contributions: Conceptualization, C.-H.S. and C.-H.W.; methodology, C.-H.S. and C.-H.W.; formal analysis, C.-H.S. and C.-H.W.; writing—original draft preparation, C.-H.S. and C.-H.W.; writing—review and editing, C.-H.S. and C.-H.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work is supported in part by a research grant from the Ministry of Science and Technology under Grant MOST 108-2221-E-011-057-MY3.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Johnson, T.; Shasha, D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), San Francisco, CA, USA, 12–15 September 1994; pp. 439–450.

2. Chang, L.P.; Kuo, T.W. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, San Jose, CA, USA, 25–27 September 2002; pp. 187–196
3. Megiddo, N.; Modha, D.S. ARC: A Self-Tuning, Low Overhead Replacement Cache. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, 31 March–2 April 2003; pp. 115–130.
4. Du, C.; Yao, Y.; Zhou, J.; Xu, X. VBBMS: A Novel Buffer Management Strategy for NAND Flash Storage Devices. *IEEE Trans. Consum. Electron.* **2019**, *65*, 134–141. [\[CrossRef\]](#)
5. Chen, R.; Shao, Z.; Li, T. Bridging the I/O performance gap for big data workloads: A new NVDIMM-based approach. In Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, 15–19 October 2016; pp. 1–12. [\[CrossRef\]](#)
6. Wu, C.; Chang, Y.; Yang, M.; Kuo, T. Joint Management of CPU and NVDIMM for Breaking Down the Great Memory Wall. *IEEE Trans. Comput.* **2020**, *69*, 722–733. [\[CrossRef\]](#)
7. Chen, R.; Shao, Z.; Liu, D.; Feng, Z.; Li, T. Towards Efficient NVDIMM-based Heterogeneous Storage Hierarchy Management for Big Data Workloads. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, Columbus, OH, USA, 12–16 October 2019; pp. 849–860
8. Mutlu, O.; Moscibroda, T. Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems. *SIGARCH Comput. Archit. News* **2008**, *36*, 63–74. [\[CrossRef\]](#)
9. Micheloni, R.; Aritome, S.; Crippa, L. Array Architectures for 3-D NAND Flash Memories. *Proc. IEEE* **2017**, *105*, 1634–1649. [\[CrossRef\]](#)
10. Fukami, A.; Ghose, S.; Luo, Y.; Cai, Y.; Mutlu, O. Improving the Reliability of Chip-Off Forensic Analysis of NAND Flash Memory Devices. *Digit. Investig.* **2017**, *20*, S1–S11. [\[CrossRef\]](#)
11. Cai, Y.; Ghose, S.; Haratsch, E.F.; Luo, Y.; Mutlu, O. Error Characterization, Mitigation, and Recovery in Flash-Memory-based Solid-State Drives. *Proc. IEEE* **2017**, *105*, 1666–1704. [\[CrossRef\]](#)
12. Luo, Y.; Ghose, S.; Cai, Y.; Haratsch, E.F.; Mutlu, O. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), Vienna, Austria, 24–28 February 2018; pp. 504–517
13. Luo, Y. Architectural Techniques for Improving NAND Flash Memory Reliability. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2012.
14. Park, S.Y.; Jung, D.; Kang, J.U.; Kim, J.; Lee, J. CFLRU: A replacement algorithm for flash memory. In Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems, Seoul, Korea, 22–25 October 2006
15. Kim, H.; Ahn, S. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST 08)*; USENIX Association: San Jose, CA, USA, 26–29 February 2008
16. Jin, P.; Ou, Y.; Härder, T.; Li, Z. AD-LRU: An efficient buffer replacement algorithm for flash-based databases. *Data Knowl. Eng.* **2012**, *72*, 83–102. [\[CrossRef\]](#)
17. Yuan, Y.; Shen, Y.; Li, W.; Yu, D.; Yan, L.; Wang, Y. PR-LRU: A Novel Buffer Replacement Algorithm Based on the Probability of Reference for Flash Memory. *IEEE Access* **2017**, *5*, 12626–12634. [\[CrossRef\]](#)
18. Kang, D.H.; Han, S.J.; Kim, Y.C.; Eom, Y.I. CLOCK-DNV: A write buffer algorithm for flash storage devices of consumer electronics. *IEEE Trans. Consum. Electron.* **2017**, *63*, 85–91. [\[CrossRef\]](#)
19. Agrawal, R.; Srikant, R. Fast Algorithms for Mining Association Rules. In Proceedings of the 1994 International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile, 12–15 September 1994; Volume 1215, pp. 487–499
20. Han, J.; Pei, J.; Yin, Y. Mining Frequent Patterns without Candidate Generation. In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, 15–18 May 2000; pp. 1–12
21. Li, J.; Xu, X.; Huang, B.; Liao, J. Frequent Pattern-based Mapping at Flash Translation Layer of Solid-State Drives. *IEEE Access* **2019**, *7*, 95233–95239. [\[CrossRef\]](#)
22. Gupta, A.; Kim, Y.; Ugaonkar, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Washington, DC, USA, 7–11 March 2009; pp. 229–240
23. Narayanan, D.; Donnelly, A.; Rowstron, A. Write Off-Loading: Practical Power Management for Enterprise Storage. *ACM Trans. Storage (TOS)* **2008**, *4*, 1–23. [\[CrossRef\]](#)
24. Lee, C.; Kumano, T.; Matsuki, T.; Endo, H.; Fukumoto, N.; Sugawara, M. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In Proceedings of the 10th ACM International Systems and Storage Conference, Haifa, Israel, 22–24 May 2017; pp. 1–11
25. Yamashita, R.; Magia, S.; Higuchi, T.; Yoneya, K.; Yamamura, T.; Mizukoshi, H.; Zaitzu, S.; Yamashita, M.; Toyama, S.; Kamae, N.; et al. 11.1 A 512Gb 3b/cell flash memory on 64-word-line-layer BiCS technology. In Proceedings of the 2017 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 5–9 February 2017; pp. 196–197
26. Nie, S.; Zhang, Y.; Wu, W.; Yang, J. Layer RBER variation aware read performance optimization for 3D flash memories. In Proceedings of the 57th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 20–24 July 2020; pp. 1–6.