國立交通大學

網路工程研究所

碩 士 論 文

一種應用於 NAND 型快閃記憶體之基於抹除碼的平行化技術

An Erasure-code-based Striping Scheme for NAND-Flash Storage System

研 究 生：張譽繽

指導教授：張立平　教授

中 華 民 國　九 十 六　年 十 月

一種應用於 NAND 型快閃記憶體之基於抹除碼的平行化技術
An Erasure-code-based Striping Scheme for NAND-Flash Storage System

研 究 生：張譽繽　　Student：Yu-Bin Chang
指導教授：張立平　　Advisor：Li-Pin Chang

國 立 交 通 大 學
網 路 工 程 研 究 所
碩 士 論 文

A Thesis
Submitted to Institute of Network Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年十月

一種應用於 NAND 型快閃記憶體之基於抹除碼的平行化技術

學生：張譽繽　　指導教授：張立平

國立交通大學 資訊工程 學系（網路工程 研究所）碩士班

摘　　要

在一塊大塊的快閃記憶體中，使用多重小塊的快閃記憶體(bank)做平行化存取，主要是用來提高系統存取時的效能，此種做法已經很普遍了，然而，現實環境中的資料存取，將使得各個 bank 間存取工作量的不平均，如此導致平行化存取產生的效能被限制住，在這篇研究中，我們將比較常存取的資料作編碼成另一份資料，這類資料也就是消除碼(erasure codes)，因為消除碼只要與部份的原始資料一起做解碼後，即可回復原本的資料，而一份要求中含很多的小型工作，研究的目的是希望能將小型工作從工作量重的 bank 中，藉由其餘工作量輕的 bank 中的消除碼取代，如此達到 bank 間工作量的平衡，研究中主要討論分成(1)如何製作與分派消除碼(2)如何放置消除碼於 bank 中(3)消除碼分派與放置完後，一份要求來後該如何的做排班。由實驗結果，我們發現只要提供 10%的額外空間作消除碼，即可使得讀取的動作提高了 50%的效能。

關鍵字：快閃記憶體（Flash memory），儲存系統（storage systems），嵌入式系統（embedded systems），作業系統（operating systems）。

An Erasure-code-based Striping Scheme for NAND-Flash Storage System

student：Yu-Bin Chang Advisors：Li-Pin Chang

Department（Institute of Network Engineering）of Computer Science
National Chiao Tung University

ABSTRACT

To use multiple memory banks in parallel is a nature approach to boost the performance of flash-memory storage systems. However, realistic data-access localities unevenly load each memory bank and thus the benefits of parallelism is severely limited. In this work, we propose to encode popular data with redundancy by means of erasure codes. Load balancing is thus achieved by accessing only lightly loaded banks, because to retrieve a subset of data blocks and code blocks sufficiently reconstructs the requested data. The technical issues pertain to redundancy allocation, redundancy placement, and request scheduling. By experiments, we found that, by offering 10% extra redundant space, the read response time is largely improved by 50%.

Keywords: Flash memory, storage systems, embedded systems, operating systems

誌　　謝

終於抵達寫致謝詞的這一刻，這意味著研究所生涯即將正式的畫上句點，回顧兩年來的經歷，內心充滿幸福與感謝，首先誠摯的感謝指導教授張立平老師，老師悉心的教導使我得以了解嵌入式系統的深奧，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。因為有你的體諒及幫忙，使得本論文能夠更完整而嚴謹。

二年的求學生涯裡，讓我學習了不少新的知識和待人接物的方法，非常感謝，每個老師的敦敦教悔，而且勤而不懈的教導我，讓我能夠一路走來都很順利。

再來，感謝在這段期間，游仕宏、曾士豪、黎光仁、杜俊達同學的幫忙，使得我能順利走過這兩年。實驗室的黃千庭、許辰暉、林松德、鄭家明學弟、許惠茹學妹們當然也不能忘記，你/妳們的幫忙及搞笑我銘記在心。

研究口試期間，感謝羅習五老師、陳雅淑老師不辭辛勞細心審閱，不僅給予我指導，並且提供寶貴的建議，使我的論文內容可以更臻完善， 在此由衷的感謝。

感謝系上諸位老師在各學科領域的熱心指導，讓我增進各項知識範疇，在此一併致上最高謝意。

最後，謹以此文獻給我摯愛的雙親。

目　　錄

表　　目　　錄

圖　目　錄

An Erasure-Code-Based Striping Scheme for NAND-Flash Storage Systems

Yu-Bin Chang and Li-Pin Chang
Department of Computer Science
National Chiao-Tung University, Hsin-Chu, Taiwan 300, ROC
bereave0000@gmail.com,lpchang@cs.nctu.edu.tw

Abstract

To use multiple memory banks in parallel is a nature approach to boost the performance of flash-memory storage systems. However, realistic data-access localities unevenly load each memory bank and thus the benefits of parallelism is severely limited. In this work, we propose to encode popular data with redundancy by means of erasure codes. Load balancing is thus achieved by accessing only lightly loaded banks, because to retrieve a subset of data blocks and code blocks sufficiently reconstructs the requested data. The technical issues pertain to redundancy allocation, redundancy placement, and request scheduling. By experiments, we found that, by offering 10% extra redundant space, the read response time is largely improved by 50%.

Keywords: Flash memory, storage systems, embedded systems, operating systems.

1 Introduction

Large flash-memory storage systems, such as solid-state disks (i.e., SSDs) [9, 11, 12], have been strongly demanded in the recent years. However, the throughput of one single NAND-flash chip fails to match neither the system-bus bandwidth nor the speed of the processor. Parallelism is one nature solution to close the performance gap. By using multiple flash-memory banks (a bank refers to a chip in this paper) in parallel, ideally the throughput can be significantly boosted. In past work, two typical parallel architectures, interleaving and striping, are considered [5, 10, 12]. As shown in Figure 1(a), interleaving is to simply tie together the command/statue lines of banks. However, even to service a small write, all the banks are involved.

Different from interleaving, striping involves a bank only when necessary. It overlaps the busy intervals of the involved banks, as shown in Figure 1(b). The attention of the one single controller is switched among banks for command setup and data transfer. However, small requests may poorly utilize the banks, because the entire system has only one request queue. To deal with this problem, the striping system in Figure 1(c) attaches a request queue to each bank. A request is divided and dispatched to the queues, and the entire request is fulfilled upon all the involved banks complete their parts. However, its benefits would be largely limited by traffic to hot data or garbage-collection activities. For

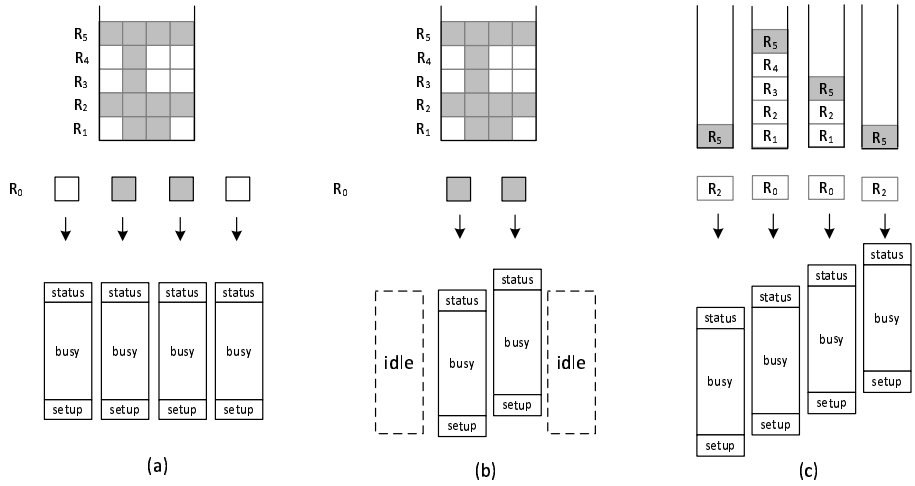


Figure 1: Different parallel architectures of flash-memory storage systems. (a) Interleaving, (b) striping with one single queue, and (c) striping with multiple queues.

example, as shown in Figure 1, the fulfillment of request R_5 is not much benefited by different parallel architectures.

The major problem of the striping system of Figure 1(c) is that heavily loaded banks severely delay the completion of large requests. In this paper, an erasure-code-bases striping policy is proposed to deal with the problem. Our basic idea is to apply the ability of erasure resilience to load balancing over banks. Erasure code is a coding technique that transforms n message blocks into $n + m$ blocks. "Erasure" usually refers to hardware malfunction or transmission loss. Upon erasure, the original message blocks can be reconstructed by retrieving only a subset S of the $n+m$ coded blocks. Optimal erasure codes have $|S|=n$. Optimality is necessarily achieved by expensive computation, such as Reed-Solomon code [13] and information-dispersal algorithm (IDA) [16]. Sub-optimal codes like Tornado Code, low-density parity check (LDPC) [15], and small parity check [14] sacrifice optimality to performance. Erasure codes have been widely adopted in storage systems for fault tolerance [17, 18, 19].

The use of erasure codes would greatly helps to improve the response of large requests. Figure 2 shows a scenario in which a request intends to fetch data blocks a, b, c, and d over a 4-bank striping system. The request is not responded until bank 1 completes all the outstanding requests, as shown in Figure 2(a). Consider that a code block storing a⊕b is in bank 3. Instead to wait for bank 1, as shown in Figure 2(b), to retrieve a, c, d, and a⊕b sufficiently reconstruct the requested data, since (a⊕b)⊕a=b. The response is greatly improved. The rationale behind the approach is that requests can be "erased" from the currently busy banks as soon as the information to be retrieved by the erased requests can be reconstructed by the data of the completed requests. There are some technical issues need investigation: First, it is infeasible to add redundancy to every piece of data. Policies of how redundancy is allocated, how the redundancy is encoded, and where to place the redundancy are needed. Upon the arrival of requests, a scheduling policy is also needed to fulfill the requests

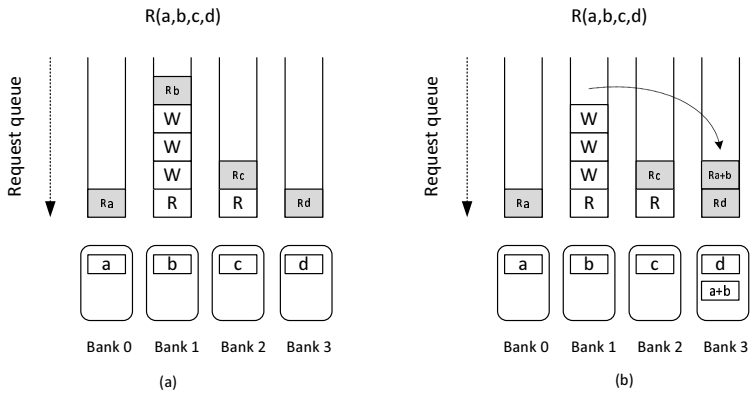


Figure 2: The handling of a read of data blocks a, b, c, and d. (a) The read request can not be fulfilled until data block b is fetched from bank 1. (b) By reading code block a⊕b from bank 3, the read is fulfilled earlier.

as soon as possible without introducing extra traffic from/to banks.

The rest of this paper is organized as follows: Section 2 introduce our system model and some motivating observations. Section 3 introduces our erasure-code-based striping scheme. Section 4 includes experimental results, and this paper is concluded in Section 5.

2 System Model and Problem Formulation

2.1 A Striping System

This section presents a striping architecture, based on which the proposed string policy is developed.

The striping architecture comprises a number NAND-flash chips, each of which refers to a bank. Signals of command and status of every bank are separately maintained so that the banks are capable of independent operations. All the memory banks are managed by one controller unit. A bank operation is of three phases. In the setup phase, the controller set a command and then the bank enters the busy phase to carry out the command. In the busy phase the controller is free. On completion, the controller comes back to examine the status of the bank. The attention of the controller is switched among banks, and parallelism is achieved by overlapping the busy phase of banks.[1]

The physical geometry of a NAND-flash chip is as follows: The entire NAND flash is partitioned into blocks, and each block is of a number of pages. Reads and writes are page-oriented, while erasure is conducted in terms of blocks. The typical block and page sizes are 128 KB and 2 KB, respectively. Let all the chips/banks in the system are homogeneous. FTL (Flash-Translation Layer) [11] is adopted to emulate the striping system as an ordinary block device. Any block-device file systems such as FAT, NTFS, or ext3 can be mounted on the emulated block device.

[1]The architecture is sufficient to demonstrate the purpose of this work. To relieve the controller of data-transfer overheads, a multi-channel DMA controller can be added [10, 12].

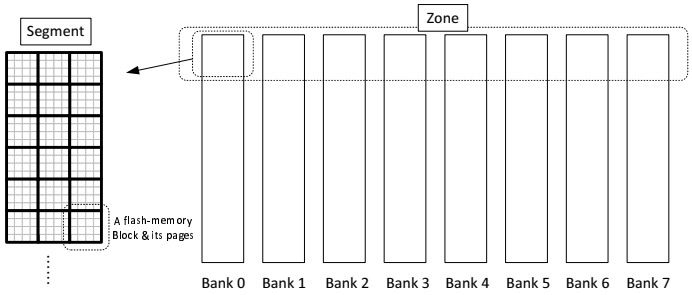


Figure 3: The organization of a striping system: Relationship among bank, zone, segment, block, and page.

To manage data on banks, logical organization is adopted, as shown in Figure 3. Let a bank be partition into equal-size *segments*. A segment is of multiple flash-memory blocks. Let a *zone* refer to the collection of all the segments having the same physical offsets in banks. A *stripe block* is of multiple logical blocks of the emulated block device (i.e., sectors). All the stripe blocks are sequentially interleaved over banks. Reads and writes to the striping system are performed in terms of stripe blocks. A stripe block is no smaller than a flash-memory page, because a partial write to a flash-memory pages is prohibited [3, 4]. In each segment, a fraction of flash-memory blocks are reserved for garbage collection and bad-block retirement. They are referred to as spare blocks. Address translation and garbage collection [1] are conducted internally in each segment.

The striping system has two operation modes, the *on-line phase* and the *off-line phase*. The striping system accepts requests in the on-line phase. Whenever the system is idle or very lightly loaded, possibly in the midnights, it enters the off-line phase for self-reorganization. In the off-line phase, activities of redundancy encoding and placement are conducted.

2.2 Localities of Reads and Writes

This section presents some motivating observations on realistic workloads of flash-memory storage systems. Because our striping system mainly aims at mass-storage devices, such as SSDs, that replace hard drives in mobile computers, the disk I/O workload of a typical mobile computer is analyzed. An arbitrary one-day fragment extracted from the disk traces collected in work [6] is considered. Let a preliminary system geometry be as follows: There are 8 memory banks, the stripe-block size is 8 KB, and the zone size is 256 MB.

Figure 4(a) shows that some particular banks receive much more writes than others. That is because of the *temporal localities* of writes. Writes to some particular data frequently arrive. Among the writes, the majority is small writes, as reported in [6]. Because small writes do not span many banks, the banks in which hot data reside would be popular. The popular banks would delay the completion of large requests, as previously shown in Figure 2. The phenomenon would be exaggerated if garbage collection is involved. On the other hand, Figure 4(a) shows that reads evenly utilize every bank. It is because the I/O cache of the hot system largely weakens the temporal localities of reads.

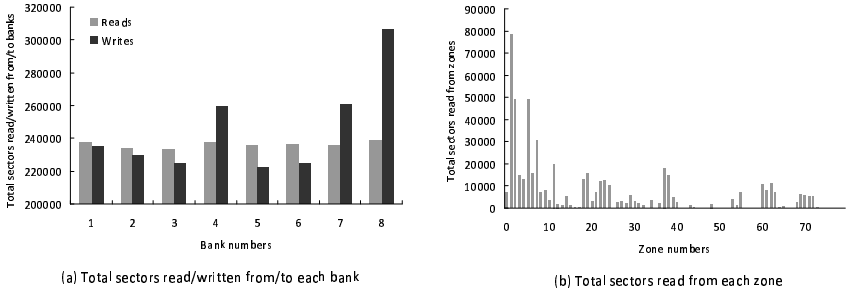


Figure 4: Analyzing the characteristics of reads and writes in an one-day fragment of the one-month disk traces.

Instead, we found strong *spatial localities* among reads. Figure 4(b) shows the amount of data read from each zone. It shows that only a small number of zones receive many reads. In other words, data adjacent to the requested data are possibly to be accessed in the near future. Most of the reads are found sequential and bulk. A bulk read would spans many banks.

The temporal localities of writes and the spatial localities of reads arise a potential performance issue of our striping architecture. The completion of bulk reads is largely delayed by small writes. In the following sections, we shall discuss how erasure codes help to deal with this problem.

3 An Erasure-Code-Based Striping Scheme

3.1 Redundancy Allocation and Placement

Redundancy allocation and redundancy placement refer to what data should be added to redundancy and where the redundancy is placed, respectively. This section aims at policies to deal with the issues.

For the ease of presentation, some terminologies are defined in order. Let Z_i and B_i stand for the i-th zone and the j-th bank, respectively. Let $S_{i,j}$ refers to the segment of zone Z_i on bank B_j. A large request is striped as multiple *sub-requests* of stripe blocks. Let a stripe block storing original data be referred to as a *data block*. A stripe block storing data encoded by data blocks is referred to as a *code block*[2]. We propose to mix **no** code blocks with data blocks in the same segment. Let zones of data blocks be referred to as *data zones*. A small number of extra zones, referred to as *code zones* are added to the striping system for the storage of code blocks. Because the extra space costs of code zones should not be large, how the code blocks are allocated is a technical question.

For writes, a bank would have a deep request queue if any of its segments recently receives many writes. The bank can not response to newly arriving requests unless all the outstanding requests complete. Not all the banks suffer from the congestion because hot data are small. For reads, to service reads involves multiple banks because most of them are bulk and sequential. However, no matter which zone a large read goes, a bank congested by small writes would delay its completion.

[2]To avoid ambiguity, flash-memory blocks are explicitly termed.

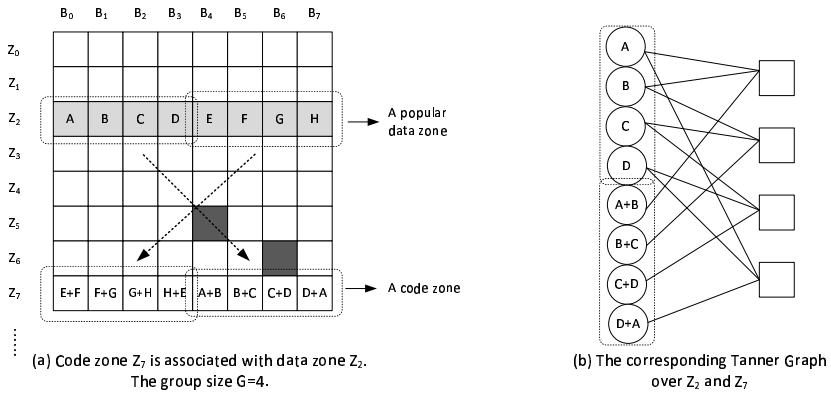


Figure 5: Associating a popular data zone with a code zone. Note that only a small number of popular data zones need to be associated with code zones.

Suppose that we have n data zones and m code zones. During the on-line phase, the number of reads each zone receives is accumulated. Once the system enters the off-line phase, we propose to associate the first m most popular zones with code zones. The relation of code zones to data zones is one-to-one. As shown in Figure 5(a), suppose that data zone Z_2 is found the most popular when the system enters the off-line phase. Code zone Z_7 is then allocated to data zone Z_2. Data in data zone Z_2 are dispersed over banks for the flexibility of load balancing, and the dispersed information are stored in code zone Z_7. An XOR-based coding scheme is proposed, as follows: Let B be the number of banks, and $G \in \mathbf{Z}^+$ an integer. For i=0...B, segment $S_{2,i}$ is XOR'ed to segment $S_{2,(i+G)\%B}$ and $S_{2,(i+1+G)\%B}$. As the example in Figure 5(a) shows, segments in Z_7 are coded as follows:

$$\begin{array}{lcllcl} S_{7,0} & = & S_{2,4} \oplus S_{2,5} & S_{7,4} & = & S_{2,0} \oplus S_{2,1} \\ S_{7,1} & = & S_{2,5} \oplus S_{2,6} & S_{7,5} & = & S_{2,1} \oplus S_{2,2} \\ S_{7,2} & = & S_{2,6} \oplus S_{2,7} & S_{7,6} & = & S_{2,2} \oplus S_{2,3} \\ S_{7,3} & = & S_{2,7} \oplus S_{2,4} & S_{7,7} & = & S_{2,3} \oplus S_{2,0} \end{array}$$

The coding scheme is to regularly disperse a piece of data over two other banks. Thus the information can be found in three banks. The parameter G refers to the *group size*, which will be discussed in the later sections. If segments $\{S_{2,0}, S_{2,1}, S_{2,2}, S_{2,3}\}$ are involved by a read request and bank B_2 happens to be busy, there are many alternatives to fulfill the request. For example, $\{S_{2,0}, S_{2,1}, S_{7,6}, S_{2,3}\}$, $\{S_{2,0}, S_{2,1}, S_{7,5}, S_{2,3}\}$, or even $\{S_{7,4}, S_{7,5}, S_{7,6}, S_{2,3}\}$. The design of the coding scheme will further be explained in Appendix.

3.2 Request Scheduling

This section presents a scheduling algorithm that makes use of code blocks so as to improve the response of large reads.

Request scheduling needs to consider not only load balancing but also the correctness of the schedule. For example, in Figure 2, neither $\{a, b, a \oplus b\}$ nor $\{a, b, a \oplus b, c\}$ are correct schedules. The former schedule fails to fulfill the request, and the latter one introduces extra traffic. As to load balancing,

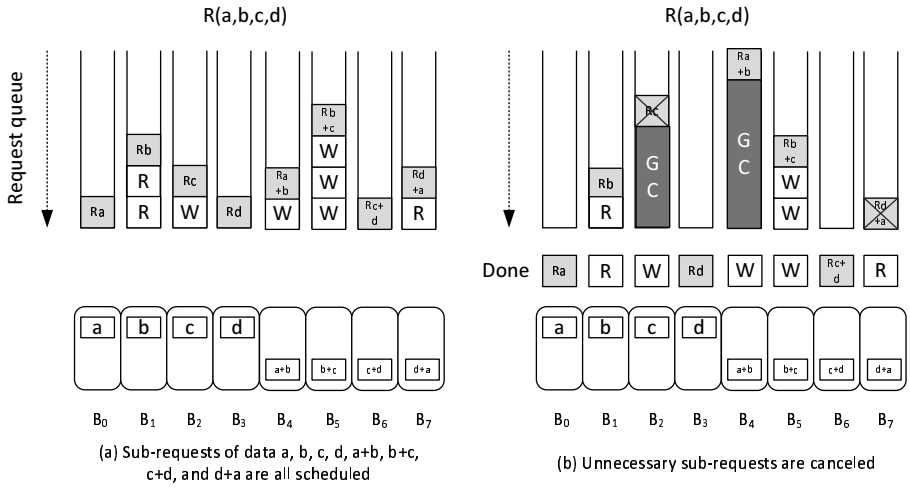


Figure 6: (a) Sub-requests of all the data blocks and code blocks are scheduled. (b) Unneeded sub-requests are removed. "GC" refers to garbage-collection activities.

in realistic workload, localities would change from time to time. Occasionally arriving writes to hot data may transiently congest a bank. Furthermore, it is hard to schedule garbage collection in advance because it is triggered on demand. As a result, dramatic changes to the loads of banks happen from time to time. To schedule sub-requests on lightly loaded banks upon the arrival of a request may not be a good choice.

We propose a lazy scheduling algorithm to deal with the problems. Upon the arrival of a read, we are not concerned with which banks should be used to service the read. Instead, sub-requests of *all* the data blocks and related code blocks are scheduled. On the completion of a sub-request, it is examined to see whether any other sub-request can be removed from queues. A sub-request can be removed if 1) the entire request is fulfilled, or 2) the sub-request is redundant. Actually the two cases are similar. Figure 6(a) shows an example of the arrival of a large read $R(a, b, c, d)$. To service the request, sub-requests of data blocks a, b, c, and d, code blocks a⊕b, b⊕c, c⊕d, and d⊕a are all scheduled. Figure 6(b) shows that, after a short period of time, sub-requests to a, d, and c⊕ d complete. Because d=(c⊕d)⊕c, sub-requests of c and d⊕a are removed as they provide no new information. Sub-requests b, b⊕c, and a⊕b remain. The read is fulfilled if any of them completes. After the fulfilment, all other remaining sub-requests can be removed. Without the lazy scheduling algorithm, on the arrival of $R(a, b, c, d)$, sub-requests of a, d, a⊕b, and c⊕d may be scheduled. Because garbage collection is triggered at bank B_4, the schedule becomes a bad decision.

The decode procedure and the removal of sub-requests are closely related to each other. A graph-based decode procedure is adopted. Before sub-requests of all the data blocks and code blocks are scheduled, a Tanner graph [20] is constructed. Figure 5(b) shows the graph over blocks in zones Z_2 and Z_7. The graph is bipartite. Data blocks and code blocks reside in the left-hand side as "message". The right-hand side nodes are "constraints", and a constraint

connects to a code block and all the data blocks that are XOR'ed to the code block. As Figure 5(b) shows, a constraint connects to A, B, and A⊕B. Whenever a sub-request of a data block or code block completes, the corresponding left-hand node and all the edges connected to the node are removed from the graph. After the removal, if there is any constraint that is connected to only one edge, then the constraint is "resolved" and the only message it connects to can be decoded without being retrieved. In this case, the constraint and all its edges are removed. The sub-request of the removed message can be removed from the queues of the striping system. For example, as shown in Figure 5(b), if B and A⊕B are retrieved and their edges are removed, then the topmost constraint connects to message A only. Message A can be decoded since B⊕(A⊕B)=A. As there is no need to retrieve A, the sub-request of A is removed from queues. The decode procedure terminates when all constraints are resolved.

3.3 A Read-ahead Policy

The proposed coding scheme provides large requests with better flexibility in load balancing. This section presents a read-ahead policy. Conditional read ahead would help to convert a sequence of small reads into large requests.

For example, consider data blocks $\{x, y, z\}$ and their code blocks $\{x \oplus y, y \oplus z, z \oplus x\}$. Suppose that a read of $\{x, y\}$ arrives. There are three different ways to handle the read: To read $\{x, y\}$, $\{x, x \oplus y\}$, or $\{y, x \oplus y\}$. Later a read of $\{z\}$ arrives. There is only one choice, to read $\{z\}$. If $\{x, y\}$ are kept in buffer, then to read $\{z\}$, $\{y \oplus z\}$, or $\{z \oplus x\}$ fulfills the request. There are either 3*1=3 or 3*3=9 ways to service the two requests. However, if a read $\{x, y, z\}$ arrives, then there are 16 ways to service the read (e.g., $\{x, y, z\}$, $\{x, y, y \oplus z\}$, $\{x, z, x \oplus y\}$, $\{x, z, y \oplus z\}$, $\{x, x \oplus z, y \oplus z\}$, etc). As the example shows, fragmented sequential reads largely limit the flexibility of request scheduling.

By analyzing the gathered disk traces, we found two potential problems. As shown in Figure 7(a), some sequential access patterns are severely fragmented as many small reads. The other case is that, as shown in Figure 7(b), even though the reads are large enough, the reads are not aligned on zone/group boundaries. Since both the two cases are sequential access, a read-ahead mechanism can be adopted to convert the fragmented reads into bulk and aligned reads. The intention is to improve the flexibility of load balancing.

A read-ahead policy is proposed. The basic idea behind the policy is to capture spatial localities. If many data blocks have been sequentially requested, then it is highly possible that the adjacent data blocks will be accessed in the near future. It can be realized by using a counter and a threshold value. The counter accumulates if the newly arrival read continues the last requested data block. Otherwise the counter is reset. If it's value is greater than the threshold, then the following requests are serviced in terms of groups. The threshold value is currently 128 sectors, which is the largest read size found in the gathered disk traces. For example, if the group size is 4, and a data block is requested from segment $S_{3,0}$, then four adjacent data blocks are read from segments $S_{3,0}$, $S_{3,1}$, $S_{3,2}$, and $S_{3,3}$. The extra data are kept in a read-ahead buffer to service subsequent reads. If the stripe-block size is 8 KB, then the read-ahead buffer is only 8*4=32 KB.

The workload of the host system may have multiple accesses patterns mixed together because the host system is multiprogramming. To individually extract

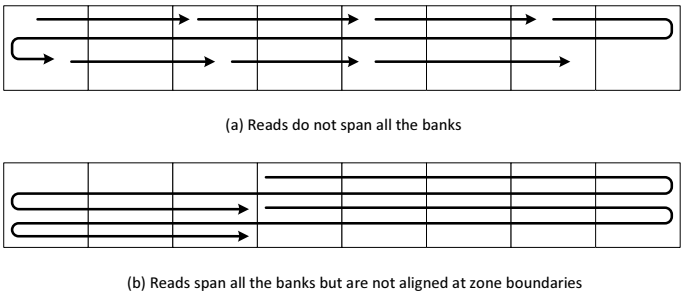


Figure 7: Two cases of a sequential access pattern: (a) The pattern is fragmented as many small reads, and (b) reads are not aligned on zone boundaries. A box stands for one segment.

sequential access patterns from workloads, a *thread table* is adopted. In our current design, the thread table is of twenty entries. The value should be no less than the number of concurrent threads that issue requests to the striping system. Each entry keeps a counter, the address of the last requested data block, and a read-ahead buffer. Upon the arrival of a read, all entries of the table is checked to see if the read continues the last data block of any entry. If so, the counter is increased and read ahead is performed whenever necessary. Otherwise the oldest entry is replaced. However, the replacement policy is vulnerable, because random reads may scrub the entire thread table. To fix this, a *sliding window* is added. The sliding window keeps the addresses of twenty recently received reads. On the arrival of a read, it is checked against the window. If the read is found sequential, then it proceeds to the thread table for read ahead and table-entry replacement. Otherwise, the window slides and the read enters the window.

4 Experimental Results

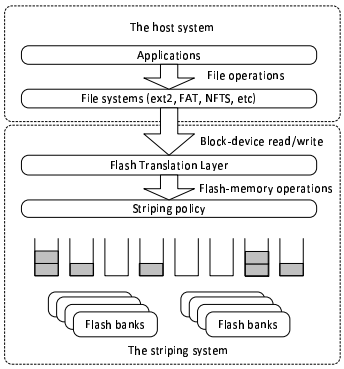
4.1 Experimental Setup and Performance Metrics

The usefulness of the proposed striping scheme is verified by a series of trace-driven simulation. A simulator is built for performance verification. Figure 8(a) includes the default striping-system configuration. The timing characteristics of NAND flash are extracted from the data sheets of real NAND flash [3]. Figure 8(b) shows that the simulated striping system comprises an FTL implementation and the proposed striping scheme. The following assumptions are taken: A request is fulfilled as soon as the requested data can be reconstructed. The XOR operations are much faster than I/O operations, and the computational overheads are negligible. The throughput of one single NAND-flash bank does not saturate the data path to the host system. Overheads of the activities conducted in the off-line phase are not accounted.

The experimental workload is the disk I/O traces collected from the daily use of a real-life UMPC (Ultra-Mobile PC) ASUS R2H. The UMPC is equipped with a Celeron-M ULV processor, 768 MB of RAM, and a 18 GB disk. The operating system is Windows XP, and the file system is NTFS. Applications ran on the UMPC are ordinary to many people, such as web browsers, email

Default striping-system configuration	
Block-device capacity	18.75 GB
Flash-memory capacity	20 GB
Stripe-block size	4 pages
Zone size	256 MB
Group size	4 segments
Banks	8
Redundancy	10%
Spare blocks	2 per segment
Flash-memory physical geometry	
Page size	2 KB
Block size	128 KB
Flash-memory timing characteristics	
Page read (setup)	1 us
Page read (busy)	51 us
Page read (data transfer)	25 us
Page write (setup)	1 us
Page write (data transfer)	25 us
Page write (busy)	200 us
Block erase (setup)	1 us

(a) Experimental system specification



(b) Striping system architecture

Figure 8: The experimental system configuration.

clients, movie players, FTP clients, office suites, and games. The time duration of trace collection is one month. The gathered traces are replayed on the striping system.

The striping system to which the proposed striping scheme is applied is referred to as a *dynamic striping system*. A dynamic striping system that never enters the off-line phase is referred to as a *static striping system*. In other words, the static striping system is not benefited by our striping scheme. Under the same system organization, the performance of a dynamic striping system and a static striping system are compared against each other. The default system configuration is shown in Figure 8(a). The default configuration is changed in terms of the number of code zones, the stripe-block size, and the zone size for evaluation. Because the proposed striping scheme mainly aims at improving response, two metrics are adopted. The read response ratio, *RR ratio*, is defined by:

$$\text{RR ratio} = \frac{\text{Average read response time in the dynamic striping system}}{\text{Average read response time in the static striping system}}$$

. The write response ratio, *WR ratio*, is defined for writes accordingly. For both the ratios, the smaller the better.

4.2 Numerical Results

4.2.1 Number of Code Zones

The first question on the proposed striping scheme would be how the proposed striping scheme improve response and how much redundancy is needed. It concerns because redundancy is a synonym of extra hardware costs. In the default system configuration, the total flash-memory capacity is 20 GB and the zone size is 256 MB. Therefore the system has 80 data zones. Different numbers of code zones are added to the default system configuration for evaluation. The results are shown in Figure 9.

It is shown that the read response is largely benefited by the adding of code zones, as expected. The improvement quickly saturates at around 8 code zones. In this case, the extra space cost is 8/80=10%. The RR ratio is 69% in this

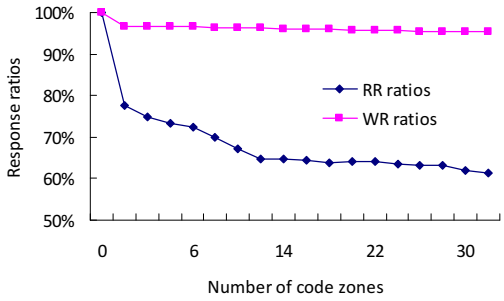


Figure 9: Responsiveness ratios when different numbers of code zones. The smaller the ratios are, the better.

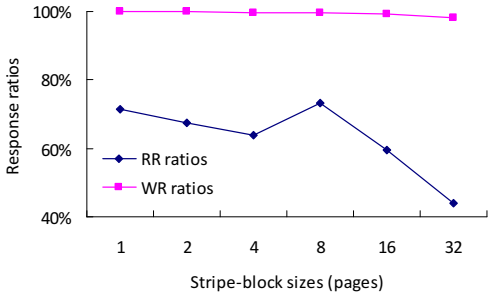


Figure 10: Response ratios with different stripe-block sizes.

case. In other words, in average reads are speeded up by about 31%. It is quite impressive. As to the WR ratios, the proposed striping scheme do not much affect them. Minor improvement of write response is gained because reads are not scheduled to the banks that already have been congested by writes. However, the interpretation of the WR ratios should be that the proposed striping scheme introduces no negative performance impact to the handling of writes.

4.2.2 Stripe-Block Sizes

The choose of the size of the stripe-block size is a tradeoff between performance and RAM-space requirements. The smaller the stripe-block size is, the larger RAM space is required for address translation in FTL. That is because a larger RAM-resident table is needed for fine-grained address translation. However, the benefit of using small stripe blocks is better parallelism. For example, a small write that is of 8 512-byte sectors can be parallelized over 4 banks if the stripe block is 1 page large (i.e., 2 KB). The same write involve only one bank if the stripe block size is 4 page large. Different stripe-block sizes are applied to the default system configuration for evaluation.

The results are shown in Figure 10. In general, the larger the strip-block size is, the larger the improvement of read response is. There are some reasons: First, with large strip blocks, data must be accessed in terms of large stripe blocks even if only a small piece of data is requested. Reads and writes to small data are enlarged. Because small requests involve only a small number of banks or even only one bank, the difference among banks' loads becomes

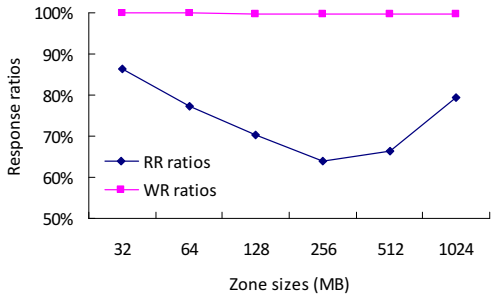


Figure 11: Response ratios with different zone sizes.

large. Second, because writes to hot data are small, many extra data might be written. The extra data writes would consequently introduce more garbage-collection activities. The uneven bank utilization is further exaggerated. Third, large stripe blocks would cluster requests of moderate sizes as if the requests were small. Response of writes are not much affected, as explained earlier.

4.2.3 Zone Sizes

The choosing of the zone size would affect the usefulness of the redundancy-allocation policy. The larger the zone size is, the smaller number of zones we have. In other words, the resolution of redundancy allocation becomes low if large zones are used. As a result, the accuracy to capture the localities of reads is then largely affected. On the other hand, to use small zones would increase the number of spare blocks needed, because every segment in zones are allocated to a fixed number of spare blocks (2 per segment in the default configuration). Different zone sizes are applied to the default system configuration for evaluation.

The results of evaluating our striping system with different zone sizes are shown in Figure 11. Non-intuitively, as the zone size is enlarged, the read response is much improved. It is because the number of spare blocks added to each segment is fixed. The larger the zone size is, the higher a segment's space utilization is. High space utilization would introduce heavy garbage-collection activities, as reported in [2]. Intensive garbage collection would heavily congest the traffic to a bank, and thus the proposed striping scheme would help to utilize other lightly loaded banks. As the zone size is larger than 256 MB, the read response ratios increases. It is because very large zones would reduce the accuracy to capture the read localities, as mentioned in the previous paragraph. Again, the write response is not much affected in this part of experiments.

4.3 Discussions

The experimental results shows that, by adding a small amount of redundancy, i.e., 10% extra space as code zones, the proposed striping scheme can effectively improve the response of reads. The organization of the striping system concerns not only the effectiveness of the proposed striping scheme but also resource requirement to manage flash memory (i.e., RAM-space requirements). Generally, for large-scale flash-memory storage systems such as high-capacity SSDs, large

mapping units are adopted so as to reduce the size of RAM-resident address translation table. The results show that our striping scheme largely helps when large stripe blocks are used. Note that a stripe block is no smaller than a mapping unit. The benefit of using large mapping units (i.e., stripe blocks) is not absolute. There are tradeoffs between performance and resource requirements when choosing the mapping-unit size, as described in [6]. We have shown that, our striping scheme is very beneficial with reasonably large mapping units, such as 4 pages, are used.

To manage large flash memory, in practice the entire flash-memory space is partitioned into fixed-sized zones and to cache only a limited number of translation tables in RAM [11]. It is of two purposes: 1) To reduce the RAM-space requirements, and 2) to avoid scanning the entire flash memory to construct the translation table. However, if the zone size is too large, not only it takes lengthy time to scan flash-memory blocks to construct the translation table but also the cache of translation tables might suffer from threshing. It consists with our observations on the proposed striping system with different zone sizes. The striping scheme helps a lot with the zone size is large, but the improvements of read response is gradually diminished when the zone size becomes fairly large. A reasonably large zone size, 256 MB, is suggested by our experiment results.

5 Conclusion

In this paper, a novel striping scheme for flash-memory storage systems is considered. We found that parallelism does not much benefit the fulfillment of large requests because of the confliction between the localities of reads and writes. An erasure-code-based striping scheme is introduced. The basic idea is to achieve load balancing by the ability of erasure resilience. A redundancy-allocation policy, a request-scheduling algorithm, and a read-ahead policy are presented to realize the idea. Experiments show that the system repones is dramatically improved with the cost of a small amount of extra space for the storage of redundancy.

In the further work, we would continue investigating the performance benefits of using different coding schemes. As optimal erasure resilience might not be a primary concern, to develop simple yet effective coding scheme for load balancing over banks is an interesting issue.

References

[1] A. Kawaguchi, S. Nishioka, and H. Motoda,"A flash-memory based File System," Proceedings of the USENIX Technical Conference, 1995.

[2] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, "Storage Alternatives for Mobile Computers," Proceedings of the USENIX Operating System Design and Implementation, 1994.

[3] Samsung Electronics Company, "K9NBG08U5M 4Gb * 8 Bit NAND Flash Memory Data Sheet".

[4] Samsung Electronics Company, "K9GAG08U0M 2G * 8 Bit NAND Flash Memory Data Sheet (Preliminary)".

[5] L. P. Chang, and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," Proceedings of The 8th IEEE Real-Time and Embedded Technology and Applications Symposium, 2002.

[6] L. P. Chang and T. W. Kuo, "An efficient management scheme for large-scale flash-memory storage systems," Proceedings of the ACM Symposium on Applied Computing, 2004; "Efficient Management for Large-Scale Flash-Memory Stroage Systems with Resource Conservation", ACM Transactions on Storage, Vol. 1, Issue 4, 2005.

[7] Intel Corporation, "Understanding the Flash Translation Layer(FTL) Specification".

[8] M-Systems, "Flash-memory Translation Layer for NAND flash (NFTL)"

[9] Samsung Electronics Company, "NAND Flash-based Solid State Disk Data Sheet," http://www.samsung.com/ Products/ Semiconductor/ FlashSSD/ download/ Standard_type.pdf.

[10] J. U. Kang, J. S. Kim, C. Park, H. J. Park, and J. W. Lee, "A Multi-Channel Architecture for High-Performance NAND flash-based Storage System," Journal of System Architecture, Vol 52, Issue 9, 2007.

[11] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, and Y. Cho, "A Space-Efficient Flash Translation Layer for Compactflash Systems," IEEE Transactions on Concumer Electronics, Vol. 48, No. 2, 2002.

[12] C. Park, P. Talawar, D. Won, M. J. Jung, J. B. Im, S. S. Kim, and Y. J. Choi, "A High Peroformance Controller for NAND Flash-based Solid State Disk (NSSD)," in Proceedings of the 21st IEEE Non-Volatile Semiconductor Memory Workshop (NVSMW), 2006.

[13] V. K. Bhargava and S. B. Wicker, "Reed-Solomon Codes and Their Applications," John Wiley & Sons, Inc., 1999.

[14] J. S. Plank, A. L. Buchsbaum, R. L. Collins, and M. G. Thomason, "Small Parity-Check Erasure Codes - Exploration and Observations," in Proceedings of the International Conference on Dependable Systems and Networks, ISBN 978-0780353916, 2005.

[15] M. Luby, M. Mitzenmacher, A. Shohrollahi, D. Spielman, and V. Stemann, "Practical Loss-Resilient Codes," in Proceedings of the 29th ACM Symposium on Theory of Computing, 1997.

[16] M. O. Rabin, "Efficient Dispersal of Information for Security, Load Banalcing, and Fault Tolerance," Journal of the ACM, Vol. 36, Issue 2, 1989.

[17] A. Bestavros, "IDA-based Redundant Arrays of Inpensive Disks," in Proceedings of the 1st International Conference on Parallel and Distributed Information Systens, 1991.

[18] M. Blaum, J. Brady, J. Bruck, J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," IEEE Transactions on Computers, vol. 44, no. 2, 1995.

[19] W. A. Burkhard, F. Cristian, G. A. Alvarez, "Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering," in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA), 1997.

[20] R. M. Tanner, "A Recursive Approach to Low Complexity Codes," IEEE Transactions on Information Theory, Vol. 27, Issue 5, 1981.

Appendix: Discussions on Coding Schemes

This section provides discussions on different coding schemes and the rationales of choosing the coding scheme shown in Section 3.1.

The price of erasure resilience is not always expensive. For example, the parity check of standard RAID-5 tolerates no more than one erasure. Mirroring is another policy that tolerates erasure. In our problem, the optimality of erasure resilience do not largely concern because erasure does not mean data loss.

Let us first start with some definitions: Let a coding scheme be denoted by a graph, in which a vertex refers to a data block d or a code block c. Code blocks are coded by data blocks only. In other words, only systematic codes are considered [14]. Let G_n denote a graph which has n data blocks (vertices $d_0, d_2, ..., d_{n-1}$) and n code blocks (vertices $c_0, c_2, ..., c_{n-1}$). An edge connecting a data block and a code block indicates that the code block is XOR'ed by the data block. A code block or a data block may have many edges connected to. Let $|G_n|$ be the total number of different subsets of $\{d_0, d_1, ..., d_n\} \cup \{c_0, c_1, ..., c_n\}$, and each of the subsets has exactly n elements and the subset sufficiently decodes $\{d_0, d_1, ..., d_n\}$. It reflects the scheduling flexibility.

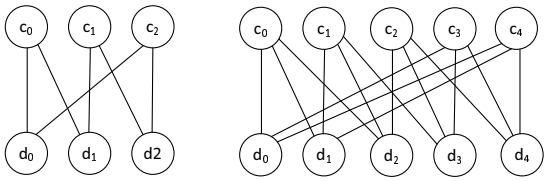
One simple question is that what graph G_n would be so that $|G_n|$ is the largest. To find out, a systematic procedure is adopted. Let a block be denoted by a vector $(a_0, a_1, ..., a_{n-1})$ over field Z^2, in which $a_i = 1$ if it is XOR'ed to data block d_i. For example, data block d_0 is $(1, 0, ..., 0)$, and $(1, 1, 1, 0, 0, ..., 0)$ refers to a code block $d_0 \oplus d_1 \oplus d_2$. To check whether a subset sufficiently reconstruct the original data, the linear independence of the vectors of the blocks in the subset is checked. For example, consider $\{d_0, d_1, d_2, c_0{=}d_0 \oplus d_1, c_1{=}d_1 \oplus d_2, c_2{=}d_2 \oplus d_0\}$. Two subsets

$$\{d_0, c_0, c_1\} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \text{ and } \{c_0, c_1, c_2\} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

. The determinant of the second matrix is zero and thus some data blocks can not be decoded. Let a regular graph G_n^m be defined by

$$\{\forall i < n, \forall j < m | e(c_i, d_{(i+j)\%n})\}$$

Figure 12(a) shows regular graph G_3^2, and Figure 12(b) is regular graph G_5^3. By the test described in the last paragraph, $|G_n^m|$ can be systematically computed. The results are shown in Table 1. The mirror scheme (e.g., m=1) and conventional party-check scheme (e.g., m=n, that of RAID-5) are shown to be the worst and the second worst in all cases. One interesting finding is that, by using the systematic test procedure, for $n \leq 5$, a regular graph is one among



(a) A regular graph G_3^2 (b) A regular graph G_5^3

Figure 12: Examples of regular graphs on G_3^2 and G_5^3.

	m=1	m=2	m=3	m=4	m=5	m=6	m=7	m=8
n=3	8	16	10					
n=4	16	45	56	17				
n=5	32	121	162	176	26			
n=6	64	320	492	472	512	37		
n=7	128	841	1,570	1,737	1,570	1,408	50	
n=8	256	2,205	4,600	4,785	4,600	5,469	3,712	65

Table 1: Enumeration of $|G_n^m|$.

the best choice. For $n \geq 6$, the best choice is not necessarily a regular graph. However, currently we have no explanations on this.

Regarding n, because in a realistic striping system the number of banks is usually a power of 2 (e.g., 4 banks, 8 banks, or 16 banks), we choose $n = 4$ because it divides the total number of banks. As to m, by the results of Table 1, $m = 3$ is suggested because $|G_4^3|$ is the largest. However we decide to take G_4^2 instead of G_4^3. It is the graph-based decoding procedure mentioned in Section 3.2 has some problem with regular graph G_4^3. Consider the Tanner Graph of G_4^3 shown in Figure 13(a). As A⊕B⊕C , C⊕D⊕A, and D⊕A⊕B are retrieved, then the three corresponding left-hand nodes and the connected edges are removed. The left-hand node of A is not removed because it is connected to two edges. However, A can be decoded by (A⊕B⊕C) ⊕ (C⊕D⊕A) ⊕ (D⊕A⊕B). There are many such exceptions in G_4^3. On the other hand, G_4^2 suffers from no the problem, and thus we choose it.

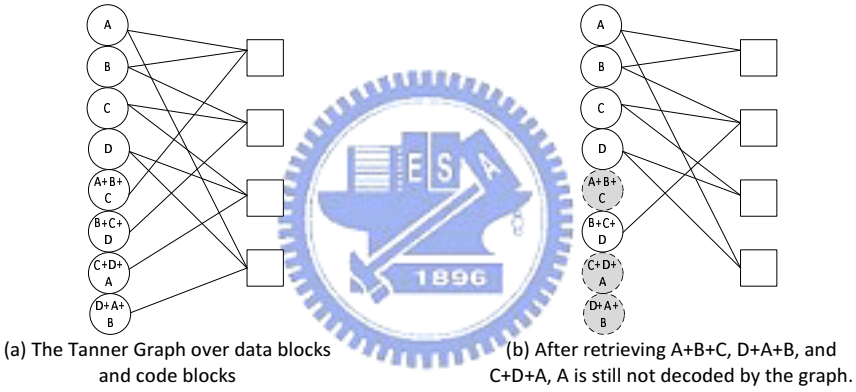


Figure 13: A problem of using Tanner Graph to decode G_4^3.