Block-Based Allocation Algorithms for FLASH Memory in Embedded Systems

Pangfeng Liu¹, Chung-Hao Chuang¹, and Jan-Jan Wu²

¹ Department of Computer Science and Information Engineering
National Taiwan University, Taipei, Taiwan
`pangfeng@csie.ntu.edu.tw`

² Institute of Information Science, Academia Sinica, Taipei, Taiwan

Abstract. A flash memory has write-once and bulk-erase properties so that an intelligent allocation algorithm is essential to providing applications efficient storage service. This paper first demonstrates that the online version of FLASH allocation problem is difficult, since we can find an adversary that makes every online algorithm to use as many number of blocks as a naive and inefficient algorithm. As a result we propose an offline allocation algorithm called *Best Match* (BestM) for allocating blocks in FLASH file systems. The experimental results indicate that BestM delivers better performance than a previously proposed First Re-arrival First Serve (FRFS) method.

1 Introduction

The recent rapid developments of embedded systems have changed many aspects of our daily life. More and more embedded systems are deployed in household appliances, office machinery, transportation vehicles, and industrial controllers. These tiny devices, with the help from increasing computing power of modern microprocessors, are able to perform and control complex operations. With this advancing embedded system technology more and more "smart" devices are able to provide inexpensive and reliable controlling capability.

There are two special properties in the flash file system management – *write-once* and *bulk-erasing*. The term write-once means that if there is a data in a storage space, it cannot be overwritten in place. The new data must be placed into another available place and the original data is declared out-of-date. If a data is updated multiple number of times the correct location of the data will change with time. This characteristic makes the management of flash file system very different from disk file systems.

A *bulk-erasing* operation is performed when there are a large number of unmarked and marked storage spaces mixed together in the flash file system. Before a "erase" operation the latest data within the regions that will be erased needs to be copied to other space with no data. These copy operations and reusing the space occupied by the old version of data are managed by the *garbage collection*.

There have been various techniques proposed to improve the performance of garbage collection for flash memory [6,7,4]. Kawaguchi, et al. proposed a

cost-benefit policy [6], which uses a value-driven heuristic function as a block-recycling policy. Chiang, et al. [4] refined the work by considering the locality in the run-time access patterns. Kim, et al. [7] proposed to periodically move live data among blocks so that blocks have more even life-cycles.

Although researchers have proposed excellent garbage-collection policies, there is little work done in providing a deterministic performance guarantee for flash-memory storage systems. It has been shown that garbage collection could impose almost 40 seconds of blocking time on time-critical tasks without proper management [8]. As a result, Chang, et al. [3] proposed a deterministic garbage collection mechanism to provide a predictable garbage collection mechanism. If the allocation method uses less blocks, the number of flash memory blocks requested to update reduces when the system executes garbage collection. Therefore, a good allocation algorithm improves garbage collection time. This motivates us to develop great allocation algorithms to reduce the resource consumption for flash memory systems.

Chou, et al. [5] proposed several allocation algorithms, including a First Re-arrival First Serve (FRFS) method that provides excellence performance. FRFS sorts the page access sequence by their re-arrival time and assigns each of them an ordinal order accordingly. The algorithm then allocates blocks for page access request according to this ordinal number. In this paper we compare our algorithm with FRFS by examining their performance.

2 Flash Memory Allocation Model

A flash memory system consists of some blocks and each block has a fixed number of cells. We assume that the number of cells in each block is denoted by B . Every cell has the same amount of capacity for data, and every block has the same number of cells. We denote the cells in every block by $C[0]$, $C[1]$, $C[2]$, $\dots$, $C[B - 1]$.

Every flash memory cell is in one of the three states – *free*, *valid* and *invalid*. A cell is free means that there is no data in it. A cell is valid means that there is data valid stored in it. A cell is invalid indicates that the data in it is no longer valid while the valid data is actually stored elsewhere. From the status of its cells we define that a block is either *active* or *inactive*. A block is active means that there is at least one valid cell in it, otherwise it is inactive. In other words an inactive block has only free or invalid cells. Note that only inactive blocks can be erased and reused.

Initially all cells are free and all blocks are inactive. When a data is placed into a free cell, the cell becomes valid and the block which contains the cell becomes active. If an inactive block becomes active, we assign an index to the block. Unlike a disk file system, a valid cell cannot be written in place. If we want to relocate a valid cell, we need to put the data into another free cell so that the original cell becomes invalid. Then the invalid cells can be transformed into free cells by an *erase* operation, which erases all cells in a block simultaneously. After the erase operation all cells of an inactive blocks return to the free state

and can be reused. On the other hand, an active block, which has at least a valid cell in it, cannot be erased.

2.1 Page Access Sequence

We assume that a file is partitioned into pages of the same size, and every page is denote by a letter (e.g., a , b , c). Each page has the same amount of data as a cell in the flash memory system, so that any page can fit into any cell. We access a file by pages, therefore file access can be modeled as a sequence of page access. This sequence is defined as *page access sequence*.

A page can appear multiple times in a page access sequence, and we need to make a distinction among these appearances. For the purpose of assigning pages to cells, we actually mean assigning a particular appearance of a page to a cell. To avoid further confusion in notation, we will use the notation p_i to denote a i -th appearance of a page p .

2.2 Flash Memory Allocation

For a page access sequence, we only need to allocate free cells for page write operation, since reading operations do not change the status of any cell. We need to allocate free cells for page writes and set the status of those cells they previously resided to invalid. Consequently, given a page access sequence, we can focus on page writes and neglect pages reads.

After we retain only the writing operations in a page access sequence, we must assign a free cell for each page write. We use a *page allocation function* for this purpose, that is, an allocation function F maps a page appearance p_i to a block $F(p_i)$ and puts page appearance p_i into the first free cell C in block $F(p_i)$, then changes the state of C from free to valid. If the block $F(p_i)$ is inactive, we set it to "active". In addition, if p_i is not the first appearance of page p in this sequence, we set the status of the cell it previously resided, which is in block $F(p_{i-1})$, to be invalid. If all cells of the block become invalid or free, we can reuse the block and set it to be "inactive".

3 Algorithms

3.1 The Online Problem

We first consider a naive algorithm for the online page allocation problem. Let us assume that there are N different pages in the input sequence. The algorithm simply place all the requests for the same pages into the same block. As a result each block contains only a single most up-to-date content of a page, plus all the previous contents that all have been marked invalid. This simple minded algorithm uses N blocks, which is much more than the obvious lower bound $\frac{N}{B}$, where B is the number of cells in a block.

We design an online adversary that makes every online algorithm to use N blocks, where N is the number of different pages in the input sequence. The

adversary first asks the algorithm to allocate a cell for the first page appearance a_0 . Then the adversary asks the algorithm to allocate a cell for the second page appearance b_0 . If the algorithm puts b_0 into the same block as a_0 is in, the adversary asks the algorithm to allocate a cell for the same page again – the page appearance will be b_1 . That is, the adversary keeps asking the algorithm to allocate a cell for the same page b until the algorithm allocates a block *different* from the block page appearance a_0 is in. The adversary keeps doing this for page c , d and so on. Eventually every page has to be in its “own” block, therefore, every online algorithm needs to use up to N blocks in the worst case.

Theorem 1. *There exists an adversary that will find an input consisting of N different pages, for any algorithm so that the algorithm must use N blocks.*

First-Come-First-Serve Algorithm. For the online problem Chou and Liu proposed a simple First Come First Serve (FCFS) algorithm [5]. The idea of FCFS is to assign blocks to page appearances according to their arrival time, so that we can use new blocks as late as possible. FCFS is simple enough to be used in the online model. However, the experimental results from [5] indicate that it does not provide superior performance.

3.2 The Offline Problem

We first review a previously proposed First-Re-arrival-First-Served (FRFS) algorithm [5]. Chou and Liu showed that FRFS produces good schedules for inputs taken from the actual disk tracing [5].

First-Rearrival-First-Serve. The idea of FRFS is to assign blocks to page appearance according to when the page will *re-arrive*, so that those page appearances that will re-arrive earlier will be placed together, and the block they reside can be reused as early as possible.

Although FRFS reuses blocks as soon as possible, it may use many new blocks before reusing a block. For example, let the page access sequence be $1, 2, \dots, N, 1, 2, 1, 3, \dots, 1, N$, and each block has two cells. FRFS allocates the first N page appearances to N different blocks according to their re-arrival time. As a result FRFS uses N different blocks before reusing the first block. Given any page access sequence of N different pages, the worst possible case is to use N blocks. For the given sequence above, FRFS actually produces the *worst* possible schedule, thus we propose a new method called *Best Match* (BestM) so that this worst case behavior is avoided.

Best Match. The idea of the Best Match algorithm is to assign page appearances to cells according to their *difference*. A difference between two page appearances is defined as the sum of the difference of their arrival time and the difference of their re-arrival time. The reason that we use difference to allocate cells is that it is likely that all cells in the same block will be set to valid (and

invalid) at about at the same time, so the flash memory allocation could use the minimal number of blocks.

Best match allocation algorithm uses a data structure called *block list* to store the flash memory blocks. Every block contains B cells and a block index. Initially the block list does not contain any block. When a page appearance arrives, the BestM algorithm computes a block index i for it. We then search the block list for block i . If block i is not in the block list yet, we insert a new block into the block list and set its index to i . After BestM decides the block index for an incoming page appearance, BestM places this page appearance into the first free cell of the assigned block. If a block becomes inactive and we want to recycle it, we just delete it from the block list.

BestM algorithm has two stages. In the first stage, BestM computes the re-arrival time for every page appearance. After knowing the re-arrival time of every page appearance, we decide the block indices for those page appearances that will *not* appear again. Since these pages will not be relocated, we reserve the first $\lceil N/B \rceil$ blocks for them. Specifically, we sort these “never-again” page appearances by their arriving time, and those B page appearances having the earliest arrival time are assigned to block index 0, and those next earliest B page appearances are assigned to block index 1, and so on. The total number of blocks used in this stage is $\lceil N/B \rceil$, so the first block indices that be assigned in the second stages is $\lceil N/B \rceil$.

In the second stage, we compute the *length* for those page appearances that the corresponding pages *will* appear again. The length of p_i is defined as $A(p_{i+1}) - A(p_i)$, where $A(p_i)$ is the arrival time of page appearance p_i . Then we sort the page access sequence according to their lengths. We first select the page appearance that has the longest length, which is denoted by p_j . When there are multiple page appearances that have the same longest length, we randomly pick one. We then select $B - 1$ page appearances that have the minimum sum of the difference from page appearance p_j . That is, we selected $B - 1$ page appearances that have the smallest sum of differences from p_j among all other page appearances, and place them into the same block as p_j . We repeat this process until all page appearances are assigned to blocks.

After each page appearance is assigned to a block, we know when every block is used and when it can be recycled. thus we can compute the necessary number of blocks for the sequence. BestM places the page appearances into the block according to the block indices. When a page appearance arrives, we put it into the first free cell of the assigned block. We insert a new block into the block list and increase the number of active blocks by one, if it is the first page appearance that is put into this block. When a page re-arrives, we also need to set the status of the cell it previously resided to be invalid. If all cells of a block are invalid or free, the block is inactive and can be reused, therefore we can delete the block in the block list and decrease the number of active blocks by one. We maintain the number of active blocks and at the end we know the maximum number of blocks required by the algorithm BestM.

4 Experimental Results

4.1 Implementation Issues

For the longest page appearance p_j that has not yet been selected, we select other $B - 1$ page appearances that have the minimum difference from p_j and assign them to the same block. Obviously, BestM can be implemented in $O(M^2)$ time, where M is the number of page appearances. When the length of page access sequence is very longer, we will spend a great amount of time to pack page appearances together. Consequently the time complexity of BestM will be much higher than FRFS, which runs in $O(M \log M)$ time. Fortunately, we use an *interval tree* data structure plus a *bounded search* technique to reduce the time complexity of matching page appearances.

We consider a page appearances as a “time windows”. The time window of page appearance p_i is from $A(p_i)$ to $A(p_{i+1})$, where $A(p_i)$ is the time when the page appearance p_i arrives. For the longest page appearance p_j , other page appearances may intersect with it, be contained in it, or be disjoint from it. BestM will find the most suitable page appearances from these three categories.

Intersected Intervals. We use an interval tree to find the most suitable page appearances from the first category, that is, those that intersect with p_j . First, we build the interval tree using a standard technique [9]. Since there are at most N distinct pages, and the page appearances from the same page do not intersect, the time to query the interval tree in order to find all intervals that intersect with p_j is bounded to $O(N + \log M)$ [9]. The number of page appearances we will find is at most $2N$, since each endpoint of p_j intersects with at most N page appearances, and p_j is the longest interval. From these $2N$ intervals, we select the $B - 1$ appearances that have the smallest difference from p_j . Let S_0 denote the set of these intervals. The set S_0 can be computed in time $O(N \log B)$ – we simply maintain a heap of the $B - 1$ elements, and the number of insertion/deletion is bounded by $O(N)$. The total number of selections is $O(\frac{M}{B})$ so the total time is $O(\frac{M}{B}(N + \log M + N \log B))$.

Contained Intervals. We now consider the intervals that are *contained* in p_j . An interval q_k is contained in another interval p_j if and only if the arrival time of q_k is no earlier than the arrival time of p_j , and the re-arrival time of q_k is no later than the re-arrival time of p_j .

The length of the longest interval p_j can be up to M time steps in the extreme case. If we examine each individual page appearance to determine those that are contained in p_j , the time complexity is $\Omega(M)$, which is unacceptable. However, for a page appearance that is contained in p_j , the longer its length, the smaller its difference from p_j , so if we examine these page appearances according to their lengths, we can reduce the time to find the most suitable appearances that should be put together with p_j .

By using the same interval tree described earlier, we can find all page appearances that are contained in every page appearance efficiently. First we sort

the page appearances according to their lengths, then the appearances query the interval tree in non-decreasing length order. For every page appearance q_k , we find the intervals that intersect it by querying the interval tree. If both end points of q_k intersect with the same page appearance p_i , then q_k is contained in p_i . In addition, since a long page appearance queries the interval tree before a short appearance, each appearance will have a list of appearances that it contains in non-decreasing length order. The time complexity for this preprocessing is bounded by $O(N + \log M)$ per page appearance, and is $O(M(N + \log M))$ for all appearances.

We now need to find the $B - 1$ longest intervals that are contained in p_j . For each page appearance, it can be contained by at most N intervals, so the total number of entries in the “contained list” for all appearances is bounded by $O(MN)$. A page appearance can easily determine the longest remaining $B - 1$ appearances by scanning through its list. Hence, the total time to find best $B - 1$ intervals that every page appearance contains is $O(MN)$. The total time including the preprocessing is therefore $O(M(N + \log M))$. We use S_1 to denote the set of the best $B - 1$ intervals that are contained in p_j .

Disjoint Intervals. Now we describe the selection process of finding the best $B - 1$ page appearances that do not intersect with, or are not included by p_j . For each page q , we consider the last appearance q_b that appears before p_j but do not have intersection with p_j . It is obvious that any page appearance $q_c, c < b$ has a larger difference than q_b has. Similarly, let the first appearance q_a that appears after p_j but does not have intersection with p_j . Any page appearance $q_c, c > a$ will have a larger difference than q_a has.

We can think of these appearances as $2N$ queues, where each page has two queues of appearances that are before or after the intervals of p_j . The elements in these $2N$ queues are sorted in increasing “difference” order so that the first element has the minimum difference in the queue.

Now we need to find the $B - 1$ appearances the have the minimum difference from these $2N$ queues. We can accomplish this in two steps. First we construct a heap of size $B - 1$ with the first element from these $2N$ queues. This heap is constructed by repeated inserting the first element of each queue, and those that are not among the $B - 1$ smallest elements are discarded. Second we start removing elements from the heap. Once we remove the minimum element from the heap, we insert the next element from the same queue into the heap. We repeatedly remove $B - 1$ elements from this heap, and these elements are the $B - 1$ appearances that has the minimum difference from p_j . Let S_2 denote the set of these intervals found. The total time complexity is bounded by $O((N + B) \log B)$ per p_j , and $O(\frac{M}{B}(N + B) \log B)$ overall.

Time Complexity. We now calculate the time complexity. The time to find $B - 1$ intersected page appearances with minimum difference is $O(\frac{M}{B}(N + \log M + N \log B))$. The time to find best $B - 1$ contained intervals is $O(MN)$. The time to find the best $B - 1$ disjoint page appearances is $O(\frac{M}{B}(N + B) \log B)$. The time to choose the best $B - 1$ intervals from S_0, S_1 , and S_2 is $O(B)$ per p_j , and

$O(M)$ overall. The total time combined is therefore $O(MN + \frac{M}{B} \log M)$. When B is a constant the overall time complexity is $O(M(N + \log M))$.

4.2 Experimental Settings

We have three different trace files collected from three computer systems. The first file is collected from a NTFS file system. The applications that access this file system include text editor, web browser and P2P software. It is collected with Microsoft trace-log and analyzed by Microsoft tracedmp. The second trace file is downloaded from BYU Trace Distribution Center [1]. It is from a database system Postgres 7.1.2, Redhat Linux with one client running 20 iterations. The third file is downloaded from flash-memory research group in CSIE, NTU. [2]. It is a FAT32 file system over a portable device. The applications have emails sending/receiving, movie playing and downloading, web surfing and so on.

4.3 Effect of the Length of Page Access Sequence

Figure 1 shows the relation between the length of page access sequence and the average number of used blocks for the first, second and third trace files respectively. FCFS performs worst because when there are a large number of pages that will not re-appear, FRFS and BestM allocate them in the same block area, but FCFS may put them into different blocks so that FCFS cannot reuse those blocks.

In Figure 1 although FRFS reuses block faster than BestM does, it may assign those page appearances that arrive at very different time to the same blocks. Consequently, FRFS may allocate a lot of blocks before it can reuse any. On the other hand, BestM places page appearances whose arrival and re-arrival time are close to each other into the same blocks, BestM combines the advantage of reusing blocks as early as possible and using new blocks as late as possible. Consequently BestM performs much better than FRFS, except for this kind of trace file that there are very few pages that usually re-arrive and so many pages that will not re-appear. FCFS performs badly for any kind of trace file.

4.4 The Effects of Cell Number

Figure 2 shows the ratio of the average numbers of blocks used by FCFS and BestM using three different file system traces under different B values. The ratio between the number of blocks used by FCFS and BestM rapidly increases because FCFS may not be able to decrease the number of used blocks but BestM is. However, when B reaches 256 for the first trace file this ratio actually drops because FCFS luckily puts some page appearances that do not re-arrive into the same block.

Figure 2 also shows the ratio of the average numbers of blocks used by FRFS and BestM using three file traces. When the number of cells in a block (B) increases the ratio increases for all three trace files. When B increases, FRFS is more likely and mistakenly to put page appearances with very different arrival time into the same blocks, and consequently use more blocks.

Fig. 1. The number of blocks used by FCFS, FRFS and BestM when given different sized prefix of the NTFS file system trace, BYU Trace Distribution Center trace log, and FAT32 portable device trace

Fig. 2. Effects of the number of cells in each block on the ratio of the number of used blocks used by FCFS and BestM, and used by FRFS to BestM

5 Conclusion

This paper proposes an allocation problem in the context of flash memory systems. We use an online adversary argument to show that when the page access sequence is given one page appearance at a time, for every algorithm we can

always find a bad input so that the performance is as bad as a simple naive algorithm.

We propose a BestM allocation algorithm that puts those page appearances whose arrival time and re-arrival time are close to each other into the same block. To overcome the aggressive behavior of a previously proposed FRFS, BestM matches together those page appearances whose arrival time and re-arrival time are close to each other. The idea is that by doing so we may obtain a balance between having to use new blocks and being able to reuse blocks.

We evaluate the performance BestM by experiments. We compare the allocation results with a previous First Re-arrival First Serve algorithm (FRFS), and show that BestM outperforms FRFS when given real trace data. When the length of page access sequence or the number of cells in every block grows, this advantage becomes more obvious. Although the $O(M(N + \log M))$ time complexity of BestM is slightly higher than FRFS, the number of required blocks is reduced by a factor of 5 from FRFS to BestM.

References

1. Byu trace distribution center. <http://tds.cs.byu.edu/tds/index.jsp>
2. Chang, L.-P., Kuo, T.-W.: An adaptive stripping architecture for flash memory storage systems of embedded systems. In: IEEE Eighth Real-Time and Embedded Technology and Applications Symposium, pp. 601–606 (2002)
3. Chang, L.-P., Kuo, T.-W., Lo, S.-W.: Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Transaction on Embedded Computing Systems* 3(4), 837–863 (2004)
4. Chiang, M.-L., Lee, C.-H., Chang, R.-C.: Manage flash memory in personal communicate devices. In: Proceedings of IEEE International Symposium on Consumer Electronics, pp. 177–182 (1997)
5. Chou, L.-F., Liu, P.: Efficient allocation algorithms for flash file systems. In: 11th International Conference on Parallel and Distributed Systems, pp. 634–641 (2005)
6. Kawaguchi, A., Nishioka, S., Motoda, H.: A flash memory based file system. In: Proceedings of the USENIX Technical Conference, pp. 155–164 (1995)
7. Kim, H.-J., Lee, S.-G.: Memory management for flash storage system. In: Proceedings of the Computer Software and Applications Conference, pp. 284–293 (1999)
8. Malik, V.: Jffs2 is broken. In: Mailing List of Memory Technology Device (MTD) Subsystem for Linux (June 28, 2001)
9. Rivest, R.L., Cormen, T.H., Leiserson, C.E., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)