Efficient Allocation Algorithms for FLASH File Systems

Li-Fu Chou Pangfeng Liu
Department of Computer and Information Engineering
National Taiwan University
pangfeng@csie.ntu.edu.tw

Abstract

Embedded systems have been developing rapidly in recent years, and flash memory technology has become an essential building block because of its shock-resistance, low power consumption, and non-volatile nature. Since flash memory is a write-once and bulk-erase medium, an intelligent allocation algorithm is essential to providing applications efficient storage service. In this paper, we propose three allocation algorithms – a First Come First Serve (FCFS) method, a First Re-arrival First Serve (FRFS) method, and an Online First Re-arrival First Serve (OFRFS) method. Both FCFS and OFRFS are on-line allocation mechanisms which make allocation decision as the requests arrive. The FRFS method, which serves as an off-line mechanism, is developed as the standard of performance comparison. The capability of the proposed mechanisms is demonstrated by a series of experiments and simulations. The experimental results indicate that FRFS provide superior performance when the data access pattern is analyzed in advance, and the on-line OFRFS method provides good performance by run-time estimation of access patterns.

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.

Flash memory system has become a very important part of embedded systems because of its shock-resistance, low power consumption, and non-volatile nature. With recent technology breakthroughs in both capacity and reliability, more and more embedded system applications tend to use flash memory as the storage systems. As a result, the efficient use of flash memory system, including a good allocation algorithm to fully utilize flash memory, is the motivation for this research.

There are two important issues in the efficiency of the flash memory storage system implementation – *write-once* and *bulk-erasing*. Since the existing data within a flash memory cell cannot be overwritten directly, a special “erase” operation must be performed before the same cell can be reused. The new version of data will be written to some other available “live” space, and the old version of data is then invalidated and considered ”dead”. As data being repeatedly updated, the locations of the data change from time to time. This *out-place-update* scheme is adopted by flash memory systems.

A *bulk-erase* is initiated when the flash memory storage systems have a large number of live and dead cells mixed together. A bulk-erase could involve a large number of live data copying since the live data within the regions that will be erased must be copied to somewhere else before the erasing. This *garbage collection* recycles the space occupied by dead data.

There have been various techniques proposed to improve the performance of garbage collection for flash memory [5, 6, 2]. Kawaguchi, et al. proposed a cost-benefit policy [5], which uses a value-driven heuristic function as a block-recycling policy. Chiang, et al. [2] refined the work by considering the locality in the run-time access patterns. Kwoun, et al. [6] 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 [7]. As a result, Chang, et al. [1] proposed a deterministic garbage collection mechanism to provide a predictable garbage collection mechanism.

The focus of this research is to emphasize the importance of *allocation strategy*, so that the system requires less times for garbage collection. In time-critical systems we must consider *both* efficient garbage collection and allocation mechanism. A good allocation algorithm greatly improves garbage collection time by reducing the number of flash memory blocks required to realize a data update pattern. This motivates us to develop intelligent data allocation algorithms to reduce the resource consumption for flash memory storage systems.

The rest of this paper is organized as follows: Section 2 introduces the flash memory operation and allocation model. Section 3 presents three allocation algorithms. Section 4 summarizes the experimental results and Section 5 concludes with possible future research directions.

2. Flash Memory Allocation Model

This section describes the allocation model for flash memory file systems. There are two major architectures in flash memory design: NOR flash and NAND flash [10]. NOR flash is a kind of EEPROM and NAND flash is designed for data storage. This paper focuses on NAND flash as a storage for embedded system. NAND also has a better price/capacity ratio compared to NOR flash. We will describe the properties of flash memory systems and the allocation problem we would like to resolve.

2.1. Flash Memory Systems

A flash memory system is a collection of n cells. Each cell in the flash memory has a unique ID, therefore the cells are denoted by $c_0, c_1, c_2, \dots c_{n-1}$. B cells are grouped into a *block* and we assume that there are F blocks, denoted by $b_0, b_1, b_2, \dots, b_{F-1}$. As a result the cells in block b_k are $c_{B*k}, c_{B*k+1}, c_{B*k+2}, c_{B*k+3}, \cdots, c_{B*(k+1)-1}$.

Each flash memory cell can be in one of the following three states – *free*, *used*, and *invalid*. A free cell has no data in it, a used cell has data written into in it, and an invalid cell has a data that is no longer valid. On the other hand, a block can be in one of the following two states – *active*, *inactive*. An active blocks has valid data written in some of its cells, and an inactive block only has free or invalid cells.

Initially all cells are free. When a data is written into a free cell, the cell becomes used. Unlike a sector in a hard disk, a written cell cannot be rewritten. If we would like to rewrite a data stored a cell, we need to write it to another free cell, and mark the original cell *invalid*. Invalid cells can be put back into the free state only through an "erase" operation. However, a block can be erased only when *every* cell of the block is free or invalid. That means only *inactive* blocks can be erased. After the erase operation all cells become free and new data can be written into them. On the other hand, if any cell of a block is in the used state the block is active, and we cannot erase it.

2.2. Page Access Pattern

A file is divided into several pages denoted by $p_0, p_1, p_2, \dots, p_{m-1}$. Each file page has the same size and can fit into a memory cell. The pages of the file will be written into allocated flash memory cells. We assume that the file operation will be performed in pages, and we will concentrate on those pages that are modified. As a result, file modification operation can be modeled as a sequence of page (and flash memory cell) modification. This sequence is called the *page access pattern*.

2.3. Flash Memory Access

After we define the file access pattern and flash memory model, we formally define the access of flash memory. Access to the flash memory can be divided into two categories – reading and writing. Since only the write operation changes the state of cells and the allocation status of flash memory, we just need to focus on the write operations. That is, we simply ignore the read operations in the page access pattern.

We divide the process of writing to the flash memory into three stages. In the first stage, we transform the file modification process into a page access pattern. In the second stage, we use a function *f* to map each page in the page access pattern to a free cell. This function is called *page allocation function*. We then apply function f on the page access pattern obtained from the first stage and allocate a cell for each page in the page access pattern. At the third stage, the page of the page access pattern p_k will be allocated a memory cell $c_{f(p_k)}$, decided in the second stage, and the state of cell $c_{f(p_k)}$ is set to "used". If the same page appeared in the page access pattern before, we set the state of the cell it was previously allocated "invalid".

3. Algorithms

3.1. First Come First Serve Allocation

The First Come First Serve Allocation algorithm places pages according to their *arrival time*, with the first coming page occupying the first available flash memory cell. This strategy is very intuitive for the following reasons. First, FCFS is very easy to implement and we can simply place the pages without any complex computation. Secondly, it seems reasonable that the first incoming page will become invalid first, so that if we apply FCFS the blocks could be reused in the earliest possible time. As a result FCFS may require less memory blocks for the same page access pattern.

We use a *block list* to store the flash memory blocks. We allocate block from the block list for the incoming pages. Initially the block list contains all the blocks of the flash memory, and the blocks are ordered in increasing identification number order.

FCFS places each page into memory cell according to its arrival time. That is, the first page is placed into the first position of the first blocks in the block list; the second page is placed into the second position of the first blocks in the block list, and so on. In other words, we start placing the pages into the second block in the block list only when the cells of the first block are all used. As a result we place the pages into blocks sequentially one block at a time.

The flash memory recycles blocks when necessary. If a block becomes inactive, it can be erased and become ready to use again. If FCFS finds that a block becomes inactive, it erases the block and moves it to the end of the block list so it can be reused.

During the FCFS allocation procedure, we keep track of the total number of the active blocks. At the end, the maximum of these active blocks number is the number of blocks required by FCFS for this access pattern. Figure 1 gives the FCFS pseudo code.

```
for each page in the page access pattern {
   Place the page into the first available
   cell from the block list.

   If the page appeared before, mark the
   cell it previously resided invalid.

   If the block the page previously resided
   now becomes inactive, erase it and move
   it to the end of the block list.
}
```

Figure 1. The pseudo code for First Come First Serve Allocation (FCFS) algorithm.

3.2. First Re-arrival First Serve Allocation

Despite the fact that FCFS is intuitive and easy to implement, it does not perform well in practice. Instead of placing pages according to their arrival time, we propose another strategy called *First Re-arrival First Serve* (FRFS) that places pages according to their *re-arrival* time. A page *re-arrives* when the same page appears again in the page access pattern. And the re-arrival time is the time when the page re-arrives. if a page does not re-arrive in the access pattern, its re-arrival time is set to infinity

The intuition that we use the re-arrival time to allocate cells is that we want to reuse blocks as soon as possible, so that the allocation could uses the minimal number of blocks. By assigning those pages that will be marked invalid first, the first used block will be reused at the earliest possible time.

FRFS algorithm has three stages. In the first stage FRFS computes the re-arrival time of each page by scanning through the entire access pattern. Note that FRFS needs to know the page access pattern in advance in order to compute the re-arrival time. The re-arrival time of those pages that do not appear again is set to infinity. Figure 1 gives the computation of the re-arrival time.

In the second stage FRFS allocates a cell for each page. We sort the page sequence by their re-arrival time and assign each of them a ordinal order according to its re-arrival time. The page having the earliest re-arrival time is assigned 0, the page with the second earliest re-arrival time is assigned 1, and so on. Table 1 illustrates an access pattern, the re-arrival time of each page, and the ordinal numbers they are assigned according to their re-arrival time.

T	1	2	3	4	5	6	7	8	9	10	11	12	13
P	a	b	c	b	a	a	d	b	d	a	d	d	a
R	5	4	i	8	6	10	9	i	11	13	12	i	i
O	1	0	9	3	2	5	4	10	6	8	7	11	12

Table 1. FRFS timestamps each page and computes its re-arrival time. Note that the re-arrival time *i* means infinity.

In the third stage FRFS places the pages into the blocks according to the ordinal number they are assigned from the second stage. If the page has ordinal number k, it will be placed into the $k\%B$-th cell of the k/B-th block in the block list, where B is the number of pages in a block. Note that when a page re-arrives, we need to set the status of the cell it previously resided to be invalid. Similar to FCFS, if FRFS finds that a block becomes inactive, it erases the block and moves it to the end of the block list so it can be reused. We also keep track of the number of active blocks, and at the end the maximum of these active block numbers is the number of blocks required by FRFS. Figure 2 gives the pseudo code for FRFS.

3.3. Online First Re-arrival First Serve Allocation

Despite the fact that FRFS performs much better than FCFS, as we shall see in the experimental results, FRFS may not be practical since it needs to know the entire access pattern in advance. However, in practice it is impossible to obtain this knowledge beforehand, and most of the time we are required to to make an allocation decision as

```
Compute the re-arrival time for each page
Compute an order of re-arrival time.

for each page in the page access pattern {
   Let k be the ordial number
   Place the page into the k % B-th cell
   of the k / B-th block in the block list.

   If the same page appeared before, mark the
   cell it previously resided invalid.

   If the block the page previously resided
   becomes inactive, erase it and move it to
   the end of the block list.
}
```

Figure 2. The pseudo code of First Re-arrival First Serve Allocation (FRFS) algorithm.

soon as a page request arrives. As a result, we have to modify FRFS so that it will be able to adapt to the on-line scenario.

We observe that most of the page access patterns contain certain amount of “regularity”. By regularity we mean that the interval between the same page appears and re-appears is more or less the same. In the next section we will formally define this interval regularity, and describe evidences that this regularity does exist in file access trace records collected in real systems. Based on this observation, we modify our FRFS algorithm to explore the interval regularity, and build an on-line FRFS (OFRFS) that can correctly estimate the time a page will re-appear. Namely we use the interval obtained from the history and the time it lastly appeared to estimate when the same page will reappear, and allocate the page for it accordingly.

The online FRFS uses two essential data structures. The first data structure is a block list as in FRFS, and the second one is a *prediction table* that contains prediction information for all the pages that have appeared. For actual implementation the prediction table will be placed in random access memory of the embedded devices, so that we can access the prediction information fast. Each page in the prediction table contains two important data – the estimated arrival interval for this page (denoted by α) and the last time it appeared (denoted by β).

Initially, the prediction table is empty. When a new page appears, we set its estimated arrival interval α to a default value v, which is the mean value of the intervals of all pages we have observed in the past. Then we set the last arrival time β of this page to the current time, and insert this entry into the prediction table. If an incoming page is already present in the prediction table, we update the estimated interval as a linear combination of the previous α, and the length of interval between the current time and the previously arrival time β. Formally, we compute the new estimated interval length as the following Equation 1, where α^* is the new estimated interval length, t is the current time, and r is the constant between 0 and 1 [9], and by definition we set the previously arrival time β to the current time.

$$\alpha^* = r\alpha + (1-r)(t-\beta) \qquad (1)$$

We now complete the online FRFS implementation after we know how to estimate the re-arrival time for each page. When a page appeared we first obtain its estimated re-arrival interval from the prediction table. Then we add the new estimated interval and the current time together as the predicted re-arrival time. When FRFS allocates a cell for the incoming page, it needs to “skip” certain number of cells, for those pages that will have re-arrival time smaller than the current page. The reason is that by allocating the order the pages reappear, blocks of cells can be reused as soon as possible. Therefore, we need to count the total number of pages which will have smaller estimated re-arrival time.

We now describe the process of counting the number of pages that will have smaller estimated re-arrival time than the current page, so that we may skip the right number of cells during allocation. When the current page arrives, its re-arrival time is the sum of its estimated interval α and the current time. Now for the other pages, by adding its estimated interval α to its last arrival time β, we obtain the time it will arrive and be allocated into a cell we reserved for it. However, the time this page is expected to *leave* the reserved cell is the sum of *twice* of their estimated interval α and the last arrival time β – the re-arrival time of this page after being allocated into the reserved cell. For each of the other pages p we compare the sum of twice of its estimated interval and the last arrival time with the sum of the estimated interval of the current page and the current time. If the page p has a smaller re-arrival time, we reserve a cell for it.

During the estimation process, we do not know the status of every pages that will eventually appear, and we only know the predicted re-arrival time of those pages that have appeared in the past. As a result we simply count the number of pages that we are aware of that have a smaller predicted re-arrival time than the current page. Let us denote this number of other pages that will reappear earlier than the current page as S. After knowing S, we look for the $(S+1)$-th free cell in the block list and put the page into it.

3.4. Hybrid Online Allocation

Despite that our online FRFS allocation algorithm allocates pages in an on-line manner with the aid of prediction table, we do not have sufficient information to deduce the re-arrival time for those pages that have very little information in the prediction table. When a new page arrives, we just set its estimated arrival interval α to a default value v,

```
Initialize the prediction table
for each page in the page access pattern {
   If the page did not appear before, set
   the estimated interval to a default
   value and insert it into prediction
   table, otherwise, re-calculate the
   estimated interval.

   Use the sum of current time and
   estimated interval as the predicted
   re-arrival time.

   Use the sum of the previous arrival
   time and twice of the estimated interval
   as predicted re-arrival time for other
   pages.

   Count the number of pages which have
   smaller re-arrival time than the current
   page in prediction table. Let the number
   be S. Place the page into the (S+1)-th free
   cell in the block list.
}
```

Figure 3. The pseudo code of On-line First Re-arrival First Serve Allocation (OFRFS) algorithm.

which is the mean value of the intervals of all pages we have observed in the past. This could seriously mislead the prediction since even when the page access pattern does have certain regularity, the interval of one page could be very different from the other. Setting the estimated interval of all pages to the same initial constant seems questionable. To overcome this problem we modify our allocation algorithm as follows: If the number of the times a page has appeared is small, we do not use the estimated interval from the prediction table to predict its re-arrival time. Instead we allocate these page with the naive FCFS method. We will refer to this algorithm as the *Hybrid Online Allocation Algorithm*. This algorithm solves the problem that the estimated interval is not correct because of insufficient data in the prediction table.

4. Experimental Results

4.1. Regularity of Page Access Pattern

In order to predict the re-arrival time of a particular page, we must have certain confidence in the "regularity" of the page access pattern. In this section we formally define the regularity, and demonstrate that it does exist in the trace files we collected from real systems. In order to quantify this interval regularity, for each page we calculate the mean value and standard deviation of the intervals the same page reappears. We then for each page calculate the ratio of the standard deviation to mean value of these intervals. This ratio is denoted by R, and a small value of R indicates that the page reappears in a predictable pattern. We use this ratio R as the criterion to judge the regularity of page access pattern.

Since each page has its own ratio R of standard deviation to mean interval length, the distribution of R from all pages is a good indicator of how well we will be able to predict the re-arrival time. We construct this R histogram as follows. We first find the maximum ratio R among all pages and denote it as R_M. The R domain is then divided to produce 100 equally sized sub-domains, denoted by $r_0, r_1, \ldots, r_{99}$. The range of r_k is between $R_M * k/100$ and $R_M * (k+1)/100$. We then construct the histogram by counting the number of pages that will fall into each sub-domain. Figure 4 gives the histogram on the left.

To compute the probability of the event that a page has a specified range of R, we put a weight on the each page. The weight of a page is the frequency that this page appears in the trace file. Now each page has a weight, so we can compute its weighted contribution towards the overall distribution. We now compute the sum of weight for all pages from a sub-domain of R, and divided it by the total weight from all pages. This quantity is indeed the probability of the event that a page has a specified range of R.

Now we calculate the accumulated probability of these ratio values. We are interested in the calculation that given a fixed probability p, what is the minimum ratio value R^* that the event of having the random variable R smaller or equal to R^* is at least p? Formally we have $Pr(R \leq R^*) \geq p$. For the sub-domain r_k we sum up the probability of sub-domains $r_0, r_1, \ldots, r_k$, and plot the accumulated probability in the right hand side of Figure 4.

We have three trace files collected in three different real systems. The first trace file is collected from an FAT32 file system. Applications are a web browser and an email client. The second trace is a disk trace downloaded from BYU Trace Distribution Center [4]. This disk trace file is from a TPC-C database benchmark with 20 warehouses, with one client running one iteration. The database system is Postgres 7.1.2, and the trace is collected with DTB v1.1 from Redhat Linux kernel 2.4.13. The third trace is collected from an FAT32 file system. The applications include web browser, text editor, P2P software, and email client. It is collected with Microsoft tracelog v5.0.

From Figure 4, as we set p to 90%, R^* equals 31, 3, 49 respectively in each of the three trace records. We conclude that $trace_2$ has higher regularity than $trace_1$, which in turn has higher regularity than $trace_3$.

4.2. Experimental Guidelines and Results

We implemented four algorithms – First Come First Serve (FCFS), First Re-arrival First Serve (FRFS), Online

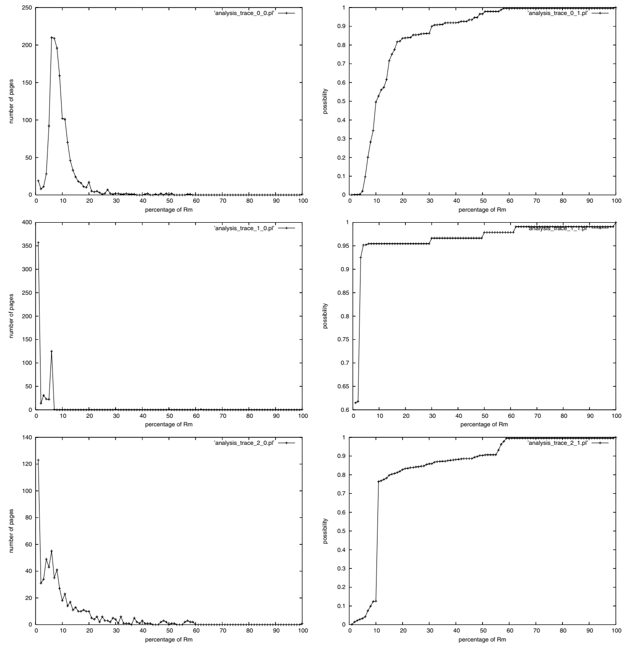

Figure 4. The histogram and accumulated probability of the three trace records.

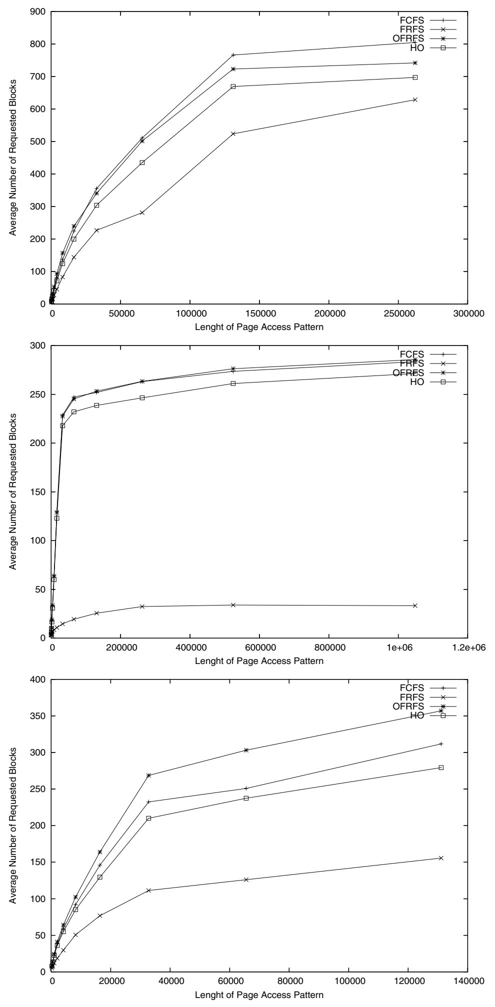

Figure 5. The average number of the requested blocks from the four allocation algorithms on the three trace files.

First Re-arrival First Serve (OFRFS) and Hybrid Online (HO). For each trace file, we compare their performance under different access patterns length N. We conducted experiments for different values of N, which is set to $2^n * 256$ for $n = 1, 2, 3, \ldots$ till the maximum value depending on the length of trace files. For each of these page access patterns we run the three allocation algorithms for 100 times and compare the average number of requested blocks.

Figure 5 plots the relation between the length of the page access pattern and the average number of requested blocks for each trace file.

Observations From Figure 5 we have the following observations. First for each trace file, the average number of requested blocks of FCFS is larger than that of FRFS. Secondly, the average number of requested blocks of OFRFS is less than that of FCFS in the first trace file, almost the same as that of FCFS in the second trace file, and actually larger than that of FCFS in the third trace file. Thirdly, the average number of requested blocks of HO is only larger than FRFS, and is smaller than that of OFRFS in all trace files.

Explanations There are two reasons that FRFC outperforms FCFS. First FRFS permutes the pages of the page access pattern in an order obtained from their re-arrival time. Consequently we put those pages that will become invalid earlier into the first blocks. When the last page in the first block arrives again, the block can immediately be erased and re-used. This accelerates the rate of producing re-usable blocks and decreases the number of requested blocks. Secondly, every page access pattern has some pages that exist forever, which we call *infinite pages*, and these infinite pages do not re-appear. Consequently the infinite pages occupy the cells they reside forever. In FRFS allocation algorithm, these infinite pages are of infinite re-arrival time, so they are all allocated to the last possible blocks. In our implementation we allocate these infinite pages separately in a different block area, so that they will not be distributed among those blocks that could be reused, causing those blocks not being able to be reused. As a result FRFS avoids the case that infinite pages interfere the recycle process and this also decreases the final number of requested blocks.

We observe that OFRFS performs quite differently in different trace files. The reason seems to be that the regularity of the trace files, i.e. the regularity of $trace_2$ is higher than that of $trace_1$, which is higher than that of $trace_3$. As a result the average number of requested blocks of OFRFS is larger than that of FCFS in the third trace file, because the third trace file has the least regularity among all three and OFRFS cannot precisely predict the re-arrival time of each page.

From the experiments we know that FCFS performance improves when the trace file has high regularity, i.e., the average number of requested blocks by FCFS in the second trace file is smaller than that in the first trace file. Similarly the performance of OFRFS also improves from the first to the second trace file. Although OFRFS is sensitive to regularity, it seems that FCFS is even more sensitive, so its performance degrades much more than OFRFS does.

The performance of HO is only second to FRFS, and the average number of requested blocks of HO is smaller than that of OFRFS and FCFS in all trace files. HO eliminates the situation that we allocated page according to wrongly predicted re-arrival time, due to insufficient data in the prediction table. If we do not trust the estimated interval, we resort to allocating the page in FCFS scheme. It is not until we have sufficient data to make reasonably correct interval estimation before we allocate the pages according to OFRFS scheme. This ensures that the performance of HO is at least as good as that of FCFS. The experimental results have verified this.

5. Conclusion

This paper discusses efficient allocation algorithms for flash file systems. The goal is to allocate a flash memory cell for each incoming page from a file access pattern so that the number of requested blocks for the file access pattern is reduced. We also implemented these algorithms and conduct experiments to compare their performances.

From the experimental result, and consisting with our intuition, the First Re-arrival First Serve (FRFS) method outperforms the First Come First Serve (FCFS) method. However FRFS may not be practical since it needs to know the entire access pattern in advance in order to make the allocation decisions. In practice this global knowledge is impossible to obtain, and most of the time we are required to allocate pages in an on-line manner. Nevertheless, FRFS illustrates a very important idea that if we can allocate pages according to their re-arrival time, we can reuse block as soon as possible and the number of requested blocks will decrease.

In our empirical study we demonstrated that file access patterns usually do have regularity. With this observation, we can predict the re-arrival time of each page from the history. This motivates us to allocate the page into the cell according the predicted re-arrival time, instead of the real re-arrival time. The Online First Re-arrival First Serve (OFRFS) method also exhibited good performance as we observed in Chapter 4. OFRFS performs exceptionally well when the length of file access pattern increases, so that it can predict next arrival time of each page precisely, and allocate pages accordingly. This reduces the number of requested blocks for the same access pattern, when compared with FCFS.

The current scope of this paper does not consider the life cycle of flash memory. That is, most flash memories only guarantee a limited number of erase and re-write cycles, and typical values are guaranteed up to 10,000 times [8]. Our allocation algorithm focuses on the number of requested blocks and do not take the number of erase and re-write operations into consideration. It is possible that if an allocation algorithm does not evenly distribute the erase and re-write operations to all cells, some cells may be worn out much earlier than the others are. As a result, our future work includes a more balanced cell allocation strategy by which the number of operations is balanced on all cells. This broader and more complex cost measurement model will certainly be more realistic, and hence more practical.

References

[1] L.-P. Chang and T.-W. Kuo. A real-time garbage collection mechanism for flash-memory storage systems in embedded systems. In *Preceedings of the 8th International Conference on Real-Time Computing Systems and Applications*, 2002.

[2] M. L. Chiang, C. H. Paul, and R. C. Chang. Manage flash memory in personal communicate devices. In *Proceedings of IEEE International Symposium on Consumer Electronics*, 1997.

[3] S. E. Company. K9f2808u0b 16mb*8 nand flash memory data sheet.

[4] K. Flanagan. Byu trace distribution center. http://tds.cs.byu.edu/tds/index.jsp.

[5] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash memory based file system. In *Proceedings of the USENIX Technical Conference*, 1995.

[6] H.-J. Kim and S.-G. Lee. Memory management for flash storage system. In *Proceedings of the Computer Software and Applications Conference*, 1999.

[7] V. Malik. Jffs2 is broken. In *Mailing List of Memory Technology Device (MTD) Subsystem for Linux*, June 28th 2001.

[8] O. Pfeiffer and A. Ayre. Using flash memory in embedded applications. http://www.esacademy.com/faq/docs/flash/index.htm.

[9] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts Sixth Edition*. John Wiley & Sons, Inc., 2003.

[10] Wikipedia. Flash memory. http://en.wikipedia.org/wiki/Flash_memory.