I/O Buffer Cache Mechanism Based on the Frequency of File Usage

Tatsuya Katakami Toshihiro Tabata and Hideo Taniguchi
Graduate School of Natural Science and Technology, Okayama University
katakami@swlab.cs.okayama-u.ac.jp, {tabata, tani}@cs.okayama-u.ac.jp

Abstract

Most operating systems manage buffer caches for buffering I/O blocks, because I/O processing is slower than CPU processing. Application programs request I/O processing from files. In order to improve the performance of I/O processing, a buffer cache should be managed with regard to both blocks and files. This paper proposes an I/O buffer cache mechanism based on the frequency of file usage. This mechanism calculates the importance of each file. Then, blocks of important files are stored in a protected space. The blocks stored in the protected space are given priority for caching. We evaluated the proposed mechanism by kernel make processing. The results show that the proposed mechanism improves the processing time by 18 s (5.7%) as compared to the LRU algorithm.

1. Introduction

Buffer cache management is an important function of the Operating System (OS). In many existing OSs, the buffer cache is managed by a block unit, and the Least Recently Used (LRU) algorithm is employed for replacing blocks.

Block replacement schemes are roughly classified into three categories: (1) a block is replaced on the basis of the reference order or reference frequency; block replacement algorithms such as the LRU, FIFO, LFU, FBR [1], LRU-k [2], IRG [3], and Aging algorithms are employed. (2) a block is replaced on the basis of a block reference pattern offered beforehand by a user; algorithms such as ACFC [4] and UBM [5] are employed. (3) a block is replaced on the basis of the regularity of the reference; algorithms such as 2Q [6], SEQ [7], and EELRU [8] are employed. These schemes are employed on the basis of the access to a block; further, they are independent of the information in the file.

Application Programs (APs) request I/O as a unit of file. In addition, the usage of files depends on the AP. The reference frequency and the tendency of I/O are different for every file. Furthermore, several APs are executed on the OS. For every AP, the reference frequency of a file and the tendencies of I/O for the same file are different. The I/O processing is significantly slow in comparison to CPU processing. Therefore, a buffer cache mechanism that reduces the gap between I/O and CPU processing is necessary for executing I/O processing efficiently.

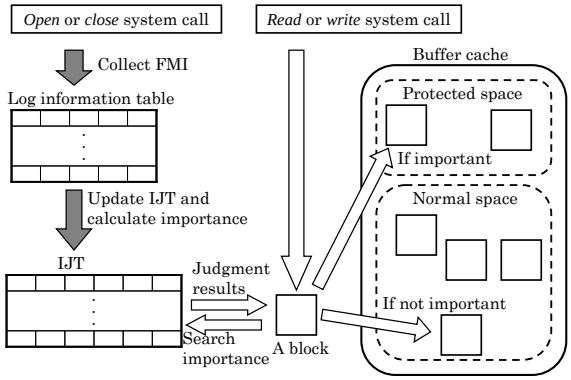

Fig. 1 Design of the proposed I/O buffer cache mechanism

This paper proposes a mechanism for managing a buffer cache on the basis of the frequency of file usage. The proposed system collects file system state information, frequency of usage of the file in the past, and the size of the file. The information collected is called as File Management Information (FMI). According to FMI, the importance of files is determined. A block constituting a file of high importance is stored in a protected space. When a block is replaced, the buffer cache mechanism performs block replacement on the basis of the importance of the files. The I/O performance of the AP is improved by storing blocks in the protected space.

2. Buffer Cache Mechanism Based on the Frequency of File Usage

2.1 Design

Figure 1 shows the design of the buffer cache mechanism based on the frequency of file usage. The proposed mechanism divides the buffer cache into a protected space and normal space. When system calls

such as *open* and *close* are requested, the cache mechanism collects FMI and stores it in a log information table. The importance of every file is calculated on the basis of the data in the log information table, and the Importance Judgment Table (IJT) is updated. During the process of reading a block into the cache, the buffer cache is managed on the basis of importance of the files. The buffer cache is managed according to the following steps:

(1) During the process of reading a block into the cache, the replacement block is selected from among those stored in the normal space, by using the LRU algorithm. If the normal space does not have any block, the least important file stored in the protected space is selected, and all blocks that constitute the file are transferred to the normal space. Further, the replacement block is selected from among those stored in the normal space, by using the LRU algorithm.

(2) When block reading is completed, the cache mechanism judges whether or not the file of the read block is important. If the file is important, the block is stored in the protected space; otherwise, the block is stored in the normal space.

2.2 Concept of Importance

Importance is a parameter or value that is used to decide whether or not a file in a read block must be stored in the protected space or normal space. This parameter is calculated on the basis of information requested by the *open* or *close* system calls in the past. Furthermore, this parameter predicts file access in the future. The concept of Importance is employed for two purposes.

(1) A file to be reused many times is provided high importance.

(2) A large file is provided low importance in order to prevent the buffer cache from being occupied by a single file.

2.3 Problems

A calculation policy by which the AP can express the possibility of file reuse is necessary. In addition, the buffer cache mechanism must collect suitable FMI in order to improve the cache hit rate according to the importance of files. A method for the updation of the IJT must be developed in order to suitably reflect the FMI. The proposed cache mechanism faces three challenges.

(1) Calculation policy for file importance
(2) FMI
(3) Method for the updation of the IJT

The solutions to these problems are provided in section 2.4.

2.4 Solutions

2.4.1 Calculation Policy for File Importance

Information for Calculation of File Importance

To realize the purpose of calculating file importance, as mentioned in section 2.2, the proposed cache mechanism employs the following parameters for the calculation of file importance:

(1) Reference counter
(2) Open frequency
(3) File size

The reference counter indicates the number of processes accessing the file. It shows the current state of a file reference. Hence, if the reference counter of the file is high, the probability of immediate file reuse is high.

The Open frequency indicates the number of times the *open* system call is requested. There exists an important correlation between the reading frequency of a file and the open frequency of the same file; this is because in the existing file system, a file is opened before it is read. Further, it is considered that the probability of reopening a file, which was opened frequently in the past, in the future is considerably high.

The file size indicates the size of an opened or closed file. When a large file is read, the buffer cache may be occupied by a single file. This reduces the cache hit rate. Further, the size of a file that is opened frequently is comparatively small [9]. For the abovementioned reasons, the cache mechanism calculates the file size for the calculation of file importance.

Calculation of File Importance

Among the three parameters mentioned above, the reference counter indicates the state of the opened file. It is assumed that a high reference counter is more read than a high open frequency. The file size is used for the calculation of file importance in order to manage the buffer cache. These parameters do not have a direct relationship with the possibility of file reuse in the future. Therefore, the following formula is used for the calculation of file importance by considering the possibility of file reuse in the future.

(Reference counter > Open frequency > File size)

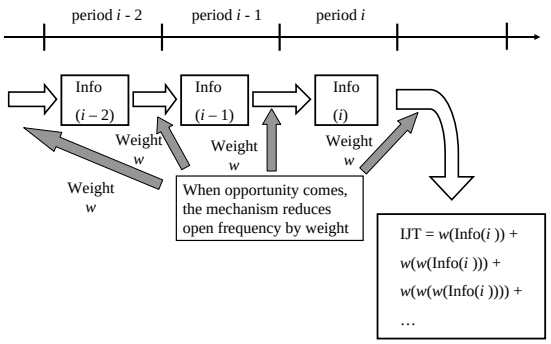

Fig. 2 Method of Updation by using the decrease function

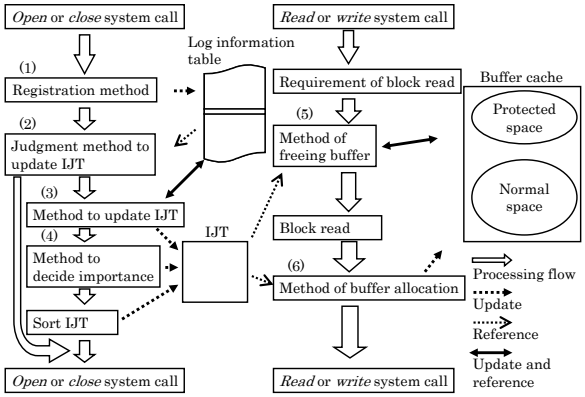

Fig. 3 Process flow

2.4.2 FMI

File management information requires the reference counter, open frequency and file size, which are necessary for the calculation of file importance. Furthermore, FMI requires an inode number to specify a file. In addition, when two files have the same importance, the files are prioritized according to time. According to the proposed mechanism, the most recent open time is required because the open frequency is used for the calculation of file importance.

The FMI possesses the information required for the calculation of file importance, inode number, and the most recent open time.

2.4.3 Method of Updation of the IJT

Opportunity to Update the IJT

The open or close states of a file changes continuously. Information is not stored in the log information table, but the IJT is updated. The overhead related to the I/O processing is high. Further, the IJT cannot reflect the FMI adequately unless it is updated with the frequent opening of closing of a file. Therefore, an updation of the IJT should link the frequency at which the file is opened or closed. The IJT must be updated when a file is opened or closed at a constant frequency after the most recent updation of the IJT.

The log information table overflows before the updation of the IJT if the information table has a fixed size. Therefore, when the FMI is stored in the log information table, the IJT is updated.

Method of Updation of the IJT

The open frequency represents old information and new information equally. To reflect recent open processing in importance more greatly, old information has lower influence. Figure 2 shows the method of updation by using the decrease function. When the IJT is updated, the cache mechanism calls the decrease function and decreases the influence of old information.

2.5 Prospective Effect

(1) Improving the I/O performance of APs
By using buffer cache management mechanisms that improve the I/O behavior and performance of the AP.

(2) Restraining a current process form using large files
According to the LRU algorithm, when a large file is read, the buffer cache is filled up by blocks that constitute the file, and the blocks used for the current process are freed from the buffer cache. According to the proposed mechanism, filling up the buffer cache by using a file is restrained by its size. This prevents a decrement in the performance of the current process.

(3) Restraining the execution of backup processing in a current process
In backup processing, files are opened only once. For backup processing, the I/O processing load is extremely heavy. According to the LRU algorithm, when backup processing is executed in a current process, the buffer cache is filled up by blocks that constitute the backup file, and blocks used by the current process are freed from the buffer cache. According to the proposed mechanism, the blocks used for the current process are protected because the importance of a file opened only once is low. Further, this restrains the decrement in performance of the current process.

Table.1 Registration or updation of information in the registration method

Information	Registration or update opportunity
Reference counter	When file is opened or closed
Open frequency	When file is opened
File size	When file is opened or closed
inode number	When new file is made
Last open time	When file is opened

3. Implementation

3.1 Process Flow

Figure 3 shows the process flow of the proposed mechanism. The *open*, *close*, *read*, and *write* system calls are requested in the proposed cache mechanism. The *open* and *close* system calls correspond to opportunities for collecting FMI. The *read* and *write* system calls correspond to opportunities for managing the buffer cache.

When the *open* or *close* system call is requested, (1) the mechanism collects the FMI and stores it in the log information table by using a registration method. Subsequently, (2) a judgment method is employed to update the IJT; if the IJT is not to be updated, the mechanism returns to the *open* or *close* system call. However, if the IJT is to be updated, (3) the IJT is updated by employing the abovementioned method of updation of the IJT. After updating the IJT, (4) the cache mechanism calculates the file importance and sorts the IJT with respect to the order of file importance order and then returns to the *open* or *close* system call.

When the *read* or *write* system call is requested along with the demand for reading a block, (5) the cache mechanism selects a block stored in the normal space, by using the LRU algorithm. Then, the block is freed by freeing the buffer. If there exists no block in the normal space, blocks that constitute the least important file are transferred to the normal space. Subsequently, a block is selected in the normal space by using the LRU algorithm, and the block is freed by freeing the buffer. When a block is read, buffer allocation is decided according to the buffer allocation method. Subsequently, if the importance of a file that is constituted by read blocks is high, the block is stored in the protected space. Otherwise, the block is stored in the normal space. Finally, the cache mechanism returns to the *read* or *write* system calls.

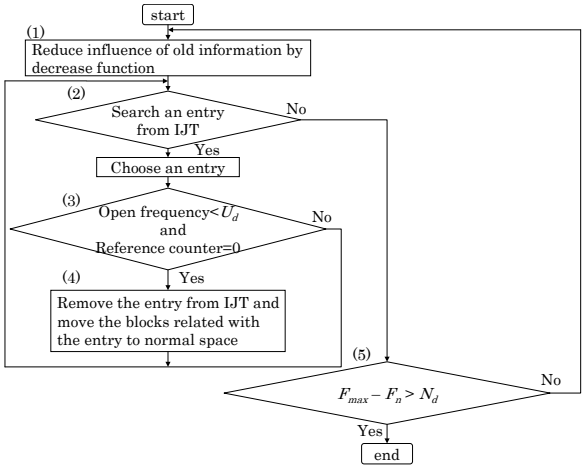

Fig. 4 Method of Updation of the IJT

Reference counter	Open frequency	File size	Inode number	Last open time	Importance
			⋮		

Fig. 5 Importance of the Judgment Table

3.2 Design of Each Process

3.2.1 Registration Method

In the registration method, the FMI is collected and registered in the log information table. When the FMI of file concerned is already registered in the log information table, the mechanism updates the FMI. Table 1 shows the FMI and updation of information in there registration method.

The reference counter is registered or updated when a file is opened or closed because the reference counter indicates whether or not a file is opened. The open frequency is registered or updated when a file is opened. Because the file size indicates the size of the current file, it is registered or updated when a file is opened or closed. The inode number does not change after a file is created. Hence, the inode number is registered when a new file is created, and the inode number is not updated. The most recent open time is registered or updated when a file is opened.

3.2.2 Judgment Method for the Updation of the IJT

In the judgment method employed for updating the IJT, the cache mechanism judges whether or not to update the IJT on the basis of the opportunities available for updation, as described in section 2.4.3.

If the cache mechanism detects an opportunity to update the IJT, it shifts to the method of updation of

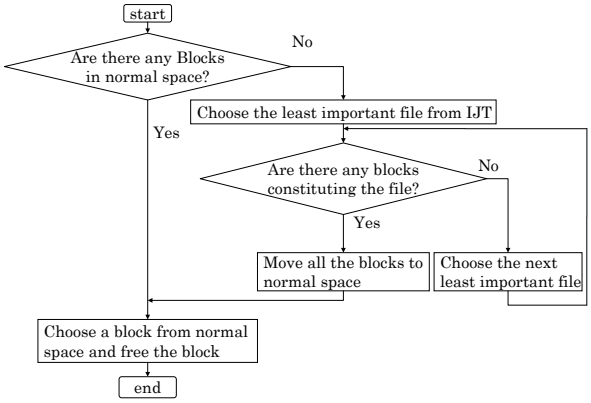

Fig. 6 Flowchart of the method of freeing the buffer

the IJT; otherwise, it returns to the *open* or *close* system calls.

3.2.3 Method of Updation of the IJT

For updating the IJT, the cache mechanism updates the IJT on the basis of the method described in section 2.4.3. Figure 4 shows the flowchart of the method of updation of the IJT, and Figure 5 shows the IJT.

In Figure 4, F_n denotes the number of stored entries in the IJT, and F_{max} denotes the size of the IJT. For updating the IJT, (1) the influence of old information is reduced by using the decrease function, (2) an entry is searched, and (3) it is judged whether or not the open frequency of an entry is less than the threshold U_d and whether or not the reference counter is 0. If the third condition is not satisfied, the next entry is searched. If the third condition is satisfied, (4) the entry is removed from the IJT and blocks transferred from the protected space to the normal space are removed. This prevents reduction in the cache hit rate by protecting the blocks related to entries deleted by the IJT. When all entries have been searched, the cache mechanism judges whether or not the number of entries not used in the IJT exceeds the threshold N_d. If the number of entries not used in the IJT exceeds the N_d, the cache mechanism terminates the updation of the IJT. Otherwise, the mechanism executes the processes (1)-(4) repeatedly until the number of entries not used in the IJT exceeds the N_d.

In Figure 5, the entry in the IJT requires FMI and the file importance in order to manage the buffer cache. Therefore, we define that an entry in the IJT possesses the FMI and file importance.

3.2.4 Method to Decide Importance

The cache mechanism employed for deciding file importance calculates the importance for all the entries in the IJT on the basis of the calculation policy for file importance, as described in section 2.4.1.

3.2.5 Method of Freeing the Buffer

Figure 6 shows a flowchart of the method of freeing the buffer. According to this mechanism, a block is freed from the buffer cache in order to read a new block. Further, if there exists a block in the normal space, it is freed from the normal space. Otherwise, the least important file is selected from the IJT. If there exist some blocks that constitute a file in the protected space, all the blocks are transferred to the normal space, and one of the blocks is freed. If there exists no block constituting the file, the subsequent least important file is selected from the IJT, and the abovementioned processes are repeated.

3.2.6 Method of Buffer Allocation

In the buffer allocation method, the cache mechanism decides whether or not the read block must be allocated in the protected space or in the normal space on the basis of the file importance. If the file that constitutes the read block is important, the block is allocated to the protected space. Otherwise, the block is allocated to the normal space. The cache mechanism requires a criterion to judge whether or not the file that constitutes the read block is important.

The file importance calculated by using the FMI changes significantly during AP processing because the FMI changes significantly. Therefore, if we define the least number to file importance, the cache mechanism cannot sufficiently adapt to changes in file importance. Therefore, in the proposed mechanism, we create files F_v according to the order of importance in the IJT.

4. Evaluation

4.1 Environment

We implement the proposed mechanism to FreeBSD 4.3-RELEASE. We measured the time required to execute “make” of kernel and compared it with that required for the LRU algorithm. The components used for the evaluation are mentioned below.

(1) CPU: Pentium4 (1.95 GHz)
(2) Memory: 512 MB
(3) Buffer cache size: 3.0 MB

To highlight the effect of the proposed mechanism, when a block was not found in the buffer cache, we invalidated a function to search for the block in the VMIO cache.

4.2 Formula and Threshold

Equation (1) shows a formula for calculating the importance based on the calculation policy for file importance described in section 2.4.1.

In equation (1), I denotes the importance; N_{open}, the open frequency; F_{num}, the reference counter; k, a fixed number; F_{size}, the file size; and B_{size}, the size of a single block.

$$I = \frac{N_{open} + F_{num} \times k}{\left[\frac{F_{size}}{B_{size}}\right] + 1} \qquad \text{Equation (1)}$$

In equation (1), considering a difference in the influence of open frequency and reference counter, we use a fixed number k. In our evaluation, we define k = 10.

$$w(N_{open}) = Dec \times N_{open} \qquad \text{Equation (2)}$$

Equation (2) shows a decrease function w that we described in section 2.4.3. In equation (2), N_{open} denotes the open frequency, and Dec denotes the weight.

The threshold values are mentioned below.

(1) R_c: Opportunity of recalculating the importance
(2) F_v: Maximum number of important files
(3) F_{max}: Maximum number of files in IJT
(4) N_d: When the number of the files in IJT is F_{max}, the mechanism deletes N_d files from IJT
(5) U_d: In the method of updation of the IJT, the FMI whose open frequency is less than N_d are deleted

4.3 Results

4.3.1 Effects of the opportunity of recalculating the importance

Figure 7 shows the effect of recalculating the file importance with respect to the processing time. For this measurement, we define F_v = 128, F_{max} = 512, N_d = 1, and U_d = 0. From Figure 7, we assume the following points:

(1) For any weight, R_c, if the value of R_c is too small, the cache mechanism can collect little FMI in a single opportunity, and the precision of importance deteriorates. If R_c is too big, the mechanism cannot adapt well to the file opening or closing because the file importance is not frequently updated.
(2) The weight corresponds to the shortest processing time. If weight is too large, the IJT contains mainly old information, and it is difficult to judge whether or not new information is important. If the weight is too small, the IJT contains only new information, and the mechanism cannot predict the opening and closing of file in the long term.

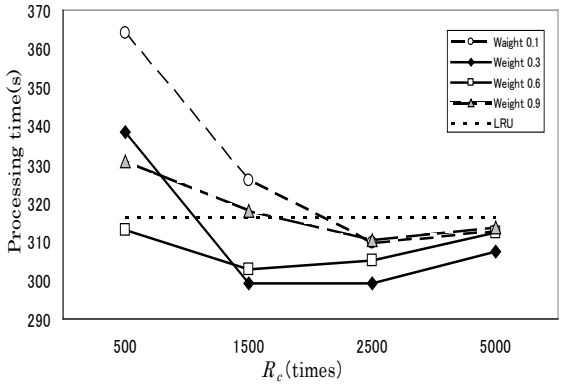

Fig. 7 Effect of the opportunity of recalculating the importance with respect to the processing time

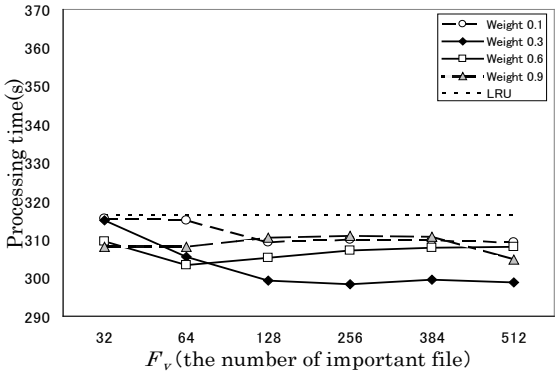

Fig. 8 Effects of the maximum number of important files with respect to the processing time

4.3.2 Effects of the maximum number of important files

Figure 8 shows the effects of the maximum number of important files with respect to the processing time. In this measurement, we define F_{max} = 512, N_d = 1 and U_d = 0. Further, R_c = 2500 because the processing time is the shortest when the weight is 0.3 in the measurement shown in Figure 7. From Figure 8, we assume the following points:

(1) If F_v is greater than 128, the processing time changes slightly. All the blocks stored in the buffer cache are protected, if F_v is greater than 128.
(2) If the weight is 0.3 and F_v is 256, the cache mechanism shortens the processing time by 18 s (5.7%). From this result, the proposed mechanism

improves the processing performance as compared to the LRU algorithm.

5. Conclusion

This paper proposed a buffer cache mechanism that is based on the frequency of file usage. In the proposed mechanism, the buffer cache is divided into a protected space and normal space. The proposed mechanism collects File Management Information from *open* and *close* system calls, calculates the importance from the File Management Information, and updates the IJT. The importance is calculated from the reference counter, open frequency, and file size. During the process of reading a block, the buffer cache is managed on the basis of importance.

We implemented the proposed mechanism to FreeBSD 4.3-RELEASE and evaluated the processing of the kernel. The results indicate that the proposed mechanism improves the processing time by 18 s (5.7%) as compared to the LRU algorithm.

In future studies, we plan to evaluate several APs by using the proposed mechanism.

References

[1] J.T. Robinson and M.V. Devarakonda, "Data Cache Management Using Frequency-Based Replacement," Proc. the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp.134-142, 1990.

[2] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "The LRU-k Page Replacement Algorithm for Database Disk Buffering," Proc. the 1993 ACM SIGMOD Conference, pp.297-306, 1993.

[3] V. Phalke and B. Gopinath, "An Inter-Reference Gap Model for Temporal Locality in Program Behavior," Proc. the USENIX Summer 1994 Technical Conference, pp.291-300, 1995.

[4] P. Cao, E.W. Falten and K. Li, "Application-Controlled File Caching Policies," Proc. the USENIX Summer 1994 Technical Conference, pp.171-182, 1994.

[5] J.M. Kim, J. Choi, J. Kim and S.H. Noh, "A Low-Overhead High-Performance Unified Buffer Management Scheme that Exploits Sequential and Looping References," Proc. 4th Symposium on Operating System Design and Implementation (OSDI 2000), pp.119-134, 2000.

[6] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," Proc. the 20th International Conference on Very Large Databases, pp.297-306, 1993.

[7] G. Glass and P. Cao, "Adaptive Page Replacement Based on Memory Reference Behavior," Proc. the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp.115-126, 1997.

[8] Y. Smaragdakis, S. Kaplan, and P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," Proc. the 1999 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp.122-133, 1999.

[9] J.K. Ousterhout, H.D. Costa, D. Harrison, J.A. Kunze, M. Kupfer and J.G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," Proc. the 10th Symposium on Operating System Principles, pp.15-24, 1985.