國 立 交 通 大 學

電機與控制工程學系

碩士論文

利用混合式快取機制提升行動檔案系統之效能

Improving Mobile File System Performance with Hybrid Caching Mechanism

研 究 生：甄元彬

Student: Un-Pan Ian

指導教授：黃育綸　博士

Advisor: Dr. Yu-Lun Huang

中華民國九十八年六月

Summer, 2009

利用混合式快取機制提升行動檔案系統之效能

Improving Mobile File System Performance with Hybrid Caching Mechanism

研 究 生：甄元彬	Student: Un-Pan Ian
指導教授：黃育綸　博士	Advisor: Dr. Yu-Lun Huang

國 立 交 通 大 學

電機與控制工程學系

碩士論文

A Thesis

Submitted to Department of Electrical and Control Engineering

College of Electrical Engineering

National Chiao Tung University

in partial Fulfill of the Requirements

for the Degree of

Master

in

Department of Electrical and Control Engineering

Summer, 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年六月

利用混合式快取機制提升行動檔案系統之效能

學生：甄元彬　　　　　　　　　　　　指導教授：黃育綸　博士

國 立 交 通 大 學電機與控制工程學系（研究所）碩士班

摘　　要

檔案快取機制與策略是決定分散式網路檔案系統傳輸與存取效能的重要因素，尤其是應用於頻寬受限的行動網路檔案系統中。 常見的檔案快取機制有全檔快取(Whole-File Cache，簡稱 WFC)及區塊檔案快取(Block-File Cache，簡稱BFC)。 其中，WFC 是一種廣泛應用於分散式網路檔案系統中的檔案快取機制。 透過 WFC，使用者可以在本機端保留檔案副本，以節省再次存取相同檔案時所需耗費的網路頻寬與傳輸時間。 BFC 的運作機制則是將檔案切割為許多區塊，並以區塊為單位，進行檔案存取。 在更新檔案內容時，BFC 僅需更新必要的檔案區塊，而不需重傳整個檔案內容。 在檔案內容變動不多的情況下，BFC 可以有效降低檔案更新所需的頻寬需求與傳輸成本。 在這篇論文中，我們設計一個混合式的快取機制（Hybrid Caching Mechanism，簡稱 HCM），結合 WFC 與 BFC 兩種快取機制， 使其能依據行動檔案系統中的檔案類型、大小等特性，選擇適用的檔案快取機制，以提升整體行動檔案系統的傳輸與存取效能。 本研究所提出之 HCM 能有效避免重覆傳送相同的檔案區塊，並能提供類似串流式檔案存取功能。 在本研究中，我們並透過一連串的實驗，分析並說明引入 HCM 後檔案系統的效能改善。

Improving Mobile File System Performance with Hybrid Caching Mechanism

Student: Un-Pan Ian Advisor: Dr. Yu-Lun Huang

Department of Electrical and Control Engineering

National Chiao Tung University

Abstract

File caching mechanisms are essential to the performance of network file systems, especially for mobile file systems. The file cache is a specific, reserved area of virtual storage in the range of system memory address. It operates on files or file blocks. Whole-file cache (WFC) is one of the popular file caching mechanisms used in distributed network file systems. With WFC, the file client keeps a local copy of the file obtained from the remote file server. As long as the requested file remains resident in the cache, subsequent file I/O requests are resolved directly from the cache without accessing the remote file server. Hence, the communication costs are reduced. Another file caching mechanism, block-file cache (BFC), intends to store a file in the local cache in blocks. Although more header overheads are introduced to store the block information, only dirty blocks are transmitted during updates if BFC is applied. Compared to WFC, BFC dramatically reduces communication costs if the size of dirty blocks is much smaller than the whole file. In this paper, we design a hybrid caching mechanism (HCM), combining WFC and BFC, to select a proper caching mechanism for a file according to the file type, file size, etc. After conducting a series of experiments, we show that the proposed mechanism offers a better performance than the traditional mobile file systems.

誌謝

首先我要感謝我最敬愛的指導教授黃育綸博士，本篇論文能夠順利完成實有賴黃育綸教授的耐心指導。十分感激黃教授在選擇研究題目上給了我高度的自由，讓我去做自己喜歡的研究領域，這對我來說真的很重要。除了在學術研究上，黃教授十分關心我的日常生活狀況與情緒變化，而且更是我們的心靈導師，時常關注我們的生活故事，與我們分享她的人生經驗和智慧。能夠當您的學生，絕對是我的榮幸。

接著我要感謝我最親愛的父母，多得您們的養育栽培，還有一直以來對我的默默支持，給我最大限度的自由去追尋我認為最重要最值得的東西，這些都是我人生最重要的資產，感謝您們給我人生最大的幸福。

另一方面，我要感謝RTES實驗室的所有成員,特別是常常義不容辭陪我周遊和探索這美麗的寶島的摯友學弟黃啟彥，在研究上給了我非常大支援的許正道同學、黃詠文學長和蔡欣宜學姐，與我分享快樂分擔悲傷的難兄難弟許育綸同學、張宗堯同學、林宗勳同學和洪精佑同學，還有所有關心我的學弟妹們，多虧有您們，我的留學生涯從不孤單。

此外，我還要感謝我的系足球隊，與我一同在球場上揮灑熱血和汗水的戰友們黃暉鈞、洪堃能、張立穎、董睿晰、蘇成鏗、徐國凱等，特別是邵啟意，六年來的南北征戰收獲了非常豐碩的獎項和回憶，好高興在異國也可以遇到相同興趣的伙伴，好享受大家努力再努力向同一目標邁進的過程，好榮幸與大家一次又一次的獲得獎項贏取尊重，感謝您們。這裡是我求學生涯的重要依靠，交大電控系足球隊。

時光飛逝，轉眼大家都將要步入新的階段。很感激在我的生命裡有您們的故事，很幸榮在您們的生命裡也有我的身影。感謝上天，讓我們在這一刻相遇。但願海闊天

空，我們都會幸福。

　　最後，僅以這篇論文獻給我人生中重要伙伴、在天堂上小白。您在我準備論文最後衝刺的時候仙遊，對我來說嚴然是個重大的打擊。但我相信您一定是希望我堅強到最後，並且會一直守護在我的身邊的。我堅強了，而且一定會一直堅強下去的，您在天堂上也要快樂的奔馳喔!

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In recent years, mobile devices are far intimate with our daily life. Different from the desktop device that typically run network on LAN or ADSL, mobile devices usually access network through WLAN, 3G or even GPRS, which connections are often unstable and speed are usually slow, especially when we are on a moving transportation or vehicle. Many mobile file systems are developed recently to overcome different network application environment, most of them support disconnected operation and file consistency, which support users to access file much more smoothly under mobile network. To accomplish those operations, a local cache is the decisive factor. CODA [1] utilizes the cache policy of whole-file cache (WFC) [1] [2] which fetches a file to local cache entirely and beforehand to achieve the disconnected operation. Users can still read files smoothly even the network is unstable, and users do not need to fetch the file again when they execute the same file again. On the other hand, Youtube support users to watch the video file while downloading it. This is block-file cache (BFC) [3] [4] policy, which usually apply to video streaming. We can start watch the video as soon as possible, unnecessary to wait until the file finish fetching. Commonly most file systems apply one cache mechanism as cache policy. We can see that these two cache mechanisms WFC and BFC cache both have their own advantages, but still inefficient in some situations. So we suppose that if we utilize this two cache mechanisms suitably, we can get better performance and enhance the average efficiency.

1.2 Contributions

In this paper, we will analyze the pros, cons and the most applicable case of employment of the cache mechanisms WFC and BFC in detail. Then we design a hybrid caching mechanism [5] [6] that combining the WFC mechanism and BFC mechanism into a file system, which supporting auto selection of the cache mechanism in the most suitable way by templates. Moreover, in BFC mechanism, we propose some methods to enhance the BFC mechanism performance such as file consistency, block search time, and block management. We expect that file system with hybrid caching mechanism can conserve bandwidth significantly and maintain acceptable performance.

1.3 Synopsis

The paper is organized as follows. In chapter 2 introduces the caching mechanism of WFC and BFC in detail, respectively. In addition we compare the pros and cons of WFC and BFC. Chapter 3 introduces the distributed file systems that are related to our research. Chapter 4 discusses our hybrid caching mechanism design. Chapter 5 show the analysis and some experiments of the HCM performance. Finally chapter 6 concludes the paper.

Chapter 2

File Caching Mechanism

Whole-file-cache (WFC) and Block-file-cache (BFC) are network file system cache mechanisms, with the aim of enhancing the file accessing efficiency, conserving valuable network services bandwidth (services fees and access time). In the following paragraphs, we introduce the characteristics of WFC and BFC, in addition to the advantages and the disadvantages, respectively.

2.1 Whole-file Cache

WFC is a network cache mechanism which fetches files entirely from server to local cache. WFC is designed for reducing bandwidth consumption, supporting disconnected operation, and additionally enhancing transmission efficiency. As shown in Fig. 2.1, WFC treats a file as a single file block plus a header with meta data inside. WFC mechanism performs as a system stores the entire file that users can access through the internet in local cache. Then when users once again access the identical contents can easily find it in the local cache directly, unnecessary to access through internet. This is the key point of saving bandwidth and also execution time; Meanwhile, WFC also allows users to execute the files smoothly even though under the unstable even disconnected network status. WFC has a great performance on saving the re-fetch bandwidth.

However, we do not utilize a file thoroughly in every read, possibly only some part of the

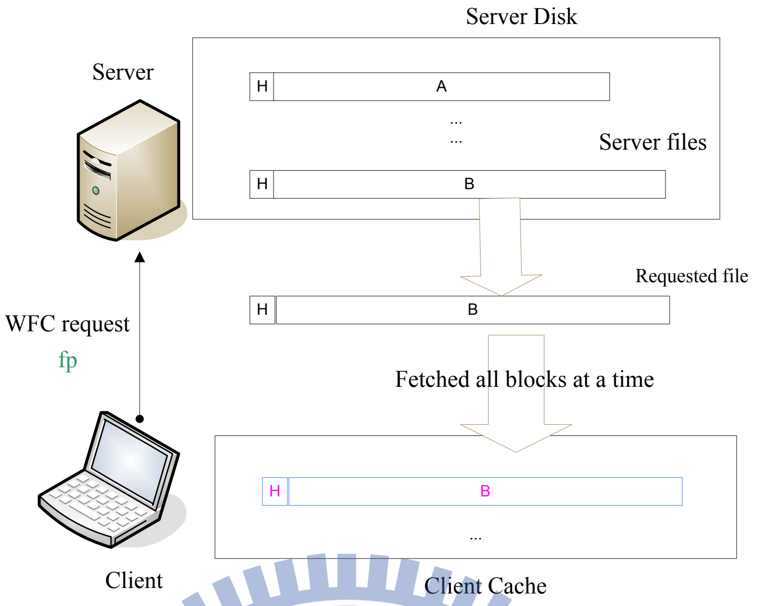

Figure 2.1: Whole-file cache

file; moreover, whole-file cache also means whole-file update, this means that we need to upload a file to server entirely even though we have only modified a few portions of the file. To lower the foregoing redundant bandwidth consuming, we propose to add one more cache mechanism to the cache policy of the system, that is, block-file cache (BFC), trying to gather the advantages of both WFC and BFC and also to complement the mutual shortcoming to get better performance.

2.2 Block-file Cache

BFC is also designed for reducing network bandwidth consumption, utilizing the cross-file similarities. The mechanism of BFC is shown in Fig. 2.2. BFC divided a file into data blocks, where each block size can be fixed or dynamic. Besides, there are a metadata header at the beginning of a file and an index header of each data blocks. During the read flow, BFC is similar to the WFC but only fetches several blocks in each request instead of directly downloading the entire file. Client sends file path (fp), numbers of blocks to fetch (n), data start point (sp) to server in every request. The primary purposes of BFC both are efficient data access: 1. how much we need/change, how much we download/upload. 2. Faster response. For example in 1, when a user reads a pdf document, user may not read the whole of the dozens of pages of documents thoroughly, possibly user may only read several pages and then close the file and never access it again. In this case, downloading the entire file but leaving most of the parts unused is inefficient.

Under this situation, for example, an external prediction process can help to have great performance with our mechanism for predicting the user's reading manner, in order to determine whether system to fetch the next part of the file or not. Continuing the previous example, initially the system fetches the first 4 pages of the document. When the user is reading the 3rd page, we can predict that user is willing to read the file continuously, and then the system fetches the next 4 pages. BFC saves the bandwidth by reduce fetching unused parts.

One key point to the prediction function is that the application is necessary to recognize the existence of BFC, or says application-aware. Otherwise application may not benefit by BFC.

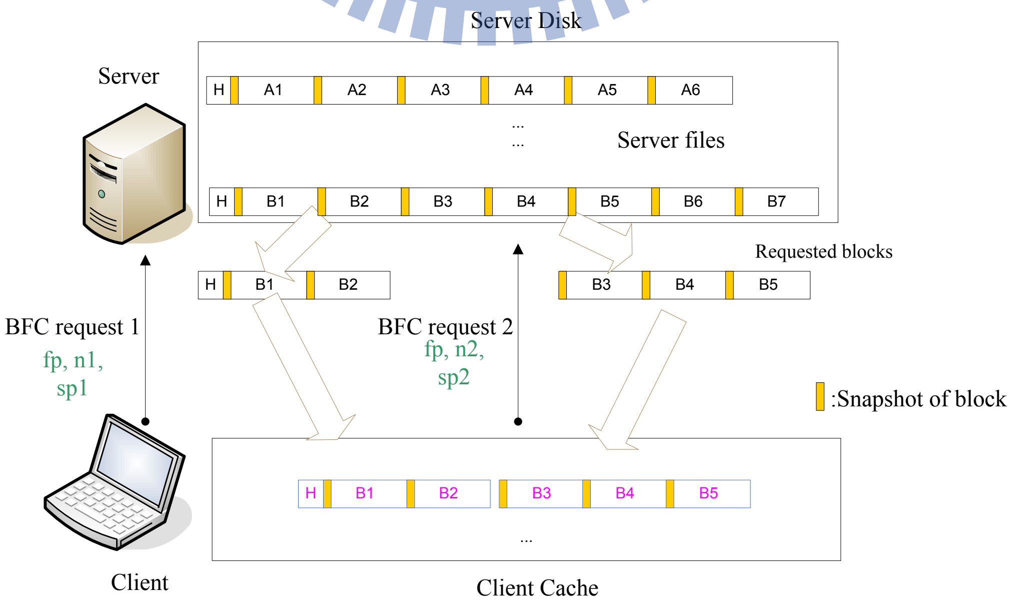

Figure 2.2: Block-file cache

On the other hand, since a file is divided into blocks, we can also update the file to/from server by blocks. That is, we can compare the difference between the original file and the modified file, to find out which blocks have been modified. Then only those dirty blocks need to be uploaded/ downloaded. As a result, the network bandwidth can be reduced substantially.

2.3 Pros and Cons

It seems that BFC is quite efficient than WFC. However, the design cost and hardware support of BFC are respectively much higher and much more complicated than WFC.As described in Table. 2.1

	Advantages	Disadvantages
WFC	Simple	Longer response time
	Directly perceived	Heavier upload workload
BFC	Faster response time	Overhead from snapshot and round trip time
	Less bandwidth	Complicated

Table 2.1: WFC vs. BFC

In BFC, for the partitioning file and distinguishing file blocks, we need to apply a hash function [7] to the system. Compared to the WFC mechanism which fetching the file entirely, BFC must introduce extra overheads on managing data blocks, identifying and preparing each block. Therefore, if the size of the target file is too small, or we partition the file into too many blocks, the rate of those extra overheads would be very high and lower the efficiency seriously.

As a result, there is no significant index can figure out that which cache mechanism will be better, it's totally based on the certain using situation. Actually, WFC and BFC are developed on different aspect, both of them have their own pros and cons. Only one of them cannot achieve the best efficiency necessarily. Instead the use of two mechanisms appropriately is expected to reach two complement each other. Particularly under different situation, they will have significantly different performance.

We suppose that file size, file format, wireless network stability and bandwidth, in addition the accessing frequency of the file, etc. would be the critical factors to the performance of network file system.

Nowadays, there are a lot of file systems implement WFC mechanism; relatively less implement BFC. In this paper, we try to implement these two cache mechanism in a single file system in order to achieve better performance than that of implement only one.

Chapter 3

Distributed File Systems

3.1 Network File System

Network File System(NFS) [8] is the first widely deployed transparent file system which allows hosts to mount partitions on a remote system and use them as though they are local file systems. This allows the system administrator to store resources in a central location on the network, providing authorized users continuous access to them. The NFS protocol was intended to be as stateless as possible. That is, client does not aware what the present server status is, and all the operation will be blocked if there is no any response from server. Besides, NFS uses a minimal caching scheme, no any cache mechanism is applied in NFS. It does not fetch files in local cache so there would be in a high latency easily during the low bandwidth connection status because of a large number of operation messages transferring over the network. Although in the latter extensions, NFS has outstanding improvements on network security, the mobility support of NFS is still not enough.

The authentication of NFS server is through IP and subnet mask. For mobile users, they usually cannot get stable IP, which makes users more difficult to use NFS in mobile network.

3.2 Andrew File System

Andrew File System (AFS) [2] is large location-transparent distributed network file system with very outstanding performance, particular on security and scalability. AFS can support about 5,000 workstations in a time. AFS clients apply the cache mechanisml of whole-file cache to keeps replicas in local cache, which can increase the operation speed once the same file is accessed. AFS adopts the scheme of write-on-close, which clients write the modified file back to server only when the file is closed. Besides, AFS server transmits a callback message to clients to inform that some new updates of file are appeared. AFS has wonderful performance on network file system and with widely acceptation, where the NFS version 4 is heavily influenced by AFS.

3.3 Coda

Coda [1] is a large–scale distributed network file system composed of Unix workstations, and which supports low bandwidth, server replication, disconnected operation and upload conflicts to overcome the network failures. Server replication is a mechanism to store copies of a file at multiple servers in case any designate servers malfunction. Disconnected operation is a mode of execution with the cache mechanism of whole-file cache in which each client utilizes the local disk as a file cache to store files and directories entirely. Whenever the server is out of reach, clients can still operate the file system and execute the fetched files smoothly. If there is any update action requested during the disconnected status, the requested files will be propagated to the server as soon as the connection recovers. Disconnected operation is particularly useful for mobile network and portable devices. Moreover, Coda has security model for authentication, encryption and access control.

The design of Coda optimizes for availability and performance, and tries to provide the highest degree of consistency attainable in the light of these objectives.

3.4 AshFS

AshFS [9], a Coda-like network file system designed for mobile network which supports automatic synchronization and disconnected operation, is based on FUSE(a file system built on user space), and develop using SSHD protocol for transferring messages, sftp and rsync for files to ease installation and deployment. The cache mechanism of AshFS is whole-file cache, which fetching the entire file and store it in local cache. The disconnected operation is similar to Coda, but upload file only when the connection strength is strong. Moreover, it advocates no server-side changes, sever do not need to install any new component when mounting this system. AshFS has great performance on preserving bandwidth. In addition, the network variation does not affect the system's performance severely, the system can aware any change in the network status and adjust the system setting sooner.

3.5 Low-bandwidth Network File System

Low-bandwidth Network File System(LBFS) [3], a network file system designed for low-bandwidth networks. LBFS exploits similarities between files or versions of the same file to save bandwidth. If data is found in client's cache or server's database, LBFS avoids sending it over the network, only the different part of a file will be sent. The cache mechanism of LBFS is block-file cache-like. LBFS file server divided the files into blocks based on content, using SHA-1 hash function [7] on small regions of file to determine blocks boundaries. On the other hand, LBFS also indexes file blocks by hash value and a <file, offset, count> triple in a database. Before transferring a file, server and client will check every block of the file in both sides, then only the different (or modified) blocks will be transmitted, to avoid transferring the redundant parts and save bandwidth. Under common operations such as editing documents and compiling software, LBFS can consume less bandwidth than traditional filesystem. The concept in cache strategy of our system is close to LBFS, but we try to hybridize WFC and BFC in a single filesystem, and we avoid using the term ``offset'' on indexing as we advocate that any block insertion or deletion would cause a considerable workload to system as every latter block's offset need to be modified. In my system, we will use another approach on indexing the blocks to avoid the term `offset'.

3.6 Mobile File and Adaption System

Mobile File and Adaption System(MFAS) [4] is a mobile file system supports fine-grained file caching and consistency, and also supports network connectivity awareness for application to adapt the caching and consistency behavior over the variable network conditions. Moreover, MFAS proposes a policy of application-directed consistency, which allows applications to decide their own consistency model based on their requirement and the available networking resources. MFAS brings out 5 templates to indicate the consistency clearly to application under different file types. In our system, we have similar policy, but we focus more on BFC and the conditions of block segmentation. On the other hand, MFAS supposes that the policies of updates-only and partial-file caching is the most important part of the system, which concepts are similar to our block-file cache and block-file update but in different implementations, where we apply hash function for indexing the blocks.

3.7 Nested FAmiLy of Line Segments

. Nested FAmiLy of Line Segments (nested FALLS) [10] presented an efficient approach for remote partial file access based on a compact pattern description language, which supports client request data from server partially. FALLS sends each request by a compact description for all the required parts of a file to server, to save the handshake traffic between client and server. On the other hand, FALLS proposed an element called file format reader, which assists applications to read the file in partial smoothly without application-awareness. We follow this concept in our system, suppose that there exist a reader between our file system and application, decoding the file blocks to the original file format so they are readable to applications. Moreover, we suggest that application-aware can increase the efficiency of our file system.

Chapter 4

Hybrid Caching Mechanism

This system is designed for reducing bandwidth consumed on file transfering by hybridizing the cache schemes, WFC and BFC, and picking up suitable cache mechanism exactly for different situations. In particular, we suggest using the close-to-open consistency. Only when a user had modified and then closed the file, in addition the network status was strong enough, upload start. On the hybrid solutions of WFC and BFC, we try to analyze our selection in several aspect: file size, file format [11] and application type, etc. In order to save bandwidth, we suppose that adding BFC to original file system can lower down the average communication cost.

For the remainder of this chapter, we first introduce the architecture of HCM, and then we introduce our hybrid caching strategy, after that we discuss the mechanism of WFC and BFC in detail.

4.1 Overview

Fig. 4.1 illustrates the architecture of general file system (a) and a file system with HCM (b). We can see that HCM is designed inside the both client and server file system. In the HCM client side, two local caches are arranged for WFC and BFC, which store the replicas from server in respective mechanism.

HCM takes charge of all the works for hybridizing WFC and BFC that we mentioned

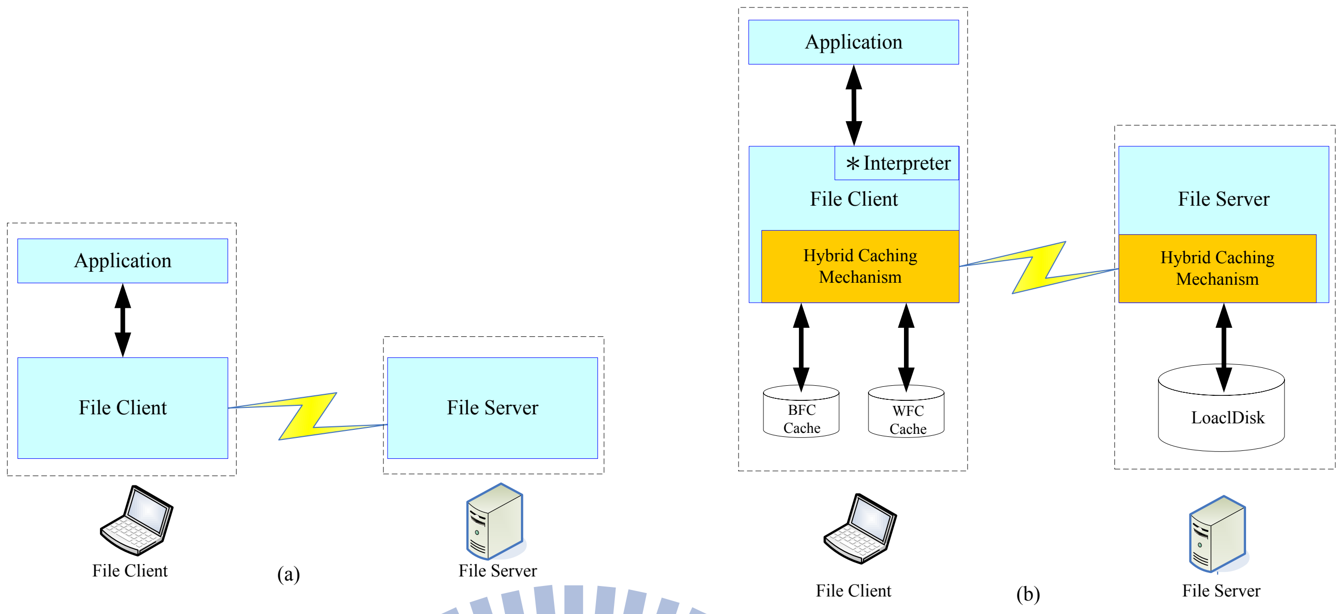

Figure 4.1: Architecture of (a) general file system (b) file system with HCM

before, such as the select strategy of WFC and BFC, BFC preprocess, and handshake protocol of WFC and BFC, etc. Blocks (or file) will be stored in the local caches after every transmission. Once application accesses the same blocks (files), system can provide the blocks (files) directly from the local caches. Except the server support of WFC and BFC transmission, the preprocess initialization of BFC is mainly proceed in the HCM server side, as the computational performance of server are much higher than mobile devices.

When we apply file in BFC, we fetch files block by block. For identifying and searching the target blocks, a snapshot including digest and block identification information must be added into every block. As a result, these extra data (snapshots) beyond the original file data may cause errors during the execution because applications do not familiar with those data. Another example that would also cause error is that in some file format, metadata would be added both at the beginning and the end of the file. If we fetch the blocks sequentially, the block file might be unreadable because of the omitting metadata placing at the end of the file. Consequently, we need an extra handler to coordinate between the application and the file system. In this paper, we focus on the analysis and hybrid cache mechanism design of WFC and BFC. So we assume that there is an interpreter [10] (Fig. 4.1 (b)) that takes charge of reconstructing the blocks into the format that the application can read before application receives the target data from the file system in client side. Moreover, the interpreter also collects the essential metadata for application. On the other hand, if application aware the existence of HCM, then application can execute more effective by skipping the interpreting step. We will discuss more about the file formats later.

4.2 Criteria

In order to enhance the system performance, we need to select WFC and BFC in the most suitable time. Here we suppose the following two factors would be the decisive points to determine which cache mechanism is more suitable under different case.

4.2.1 File format

In HCM, we suppose that large file can gain benefits transmitting by BFC mechanism. However, under the aspect of block execution, not all the file formats can execute with data piece even we obtain the complete header or metadata, and usually the specification documents that describes exactly how data is encoded are also unpublished. For example, Mircosoft Word encodes file into binary code, any of a bit missed in the file content affects a large-scale of the original data content. Even if we can get the encoding algorithm through OpenOffice organization [12], it is still not an good idea to implement the decoding algorithm on mobile device. As a result, it is hard to gain benefit on Word format by using BFC mechanism, and the other encode/decode-like file formats are also unreachable. File formats like Mircosoft Office series have similar property.

Test on different formats

We did some simple tests on common different file formats by deleting some random parts of a file using text editor, try to find out the metadata location and to discuss whether the file formats are suitable to apply BFC or not. We tested audio, video, ASCII text, image file, Mircosoft Office series file and also pdf files. Audio file such as mp3 are store file into frames, it is similar to that divide into blocks, so we can apply our BFC mechanism to audio files. Furthermore, the metadata of mp3 is placed at the beginning. Video files are mostly similar to the audio's, no matter the data contents and metadata storage ways, but some format like .mov which implements special encoding on the file data stream. ASCII text files such as .txt, .c, .h and some calendar format are saved as hexadecimal ASCII code, any of insufficient data of a character just affects itself. As a result, ASCII text file is suitable for BFC mechanism. No file metadata is placed on ASCII text file. Image file such as bmp and jpg builds data file sequentially, it also can put on BFC. Mircosoft series file such as .doc, .ppt, .xls encode the file in their own algorithm, any missing data in data stream would cause a large-scale of precious data section unreadable, or even file inexecutable. Therefore, we do not suggest use BFC mechanism on Mircosoft Office files as the application of read on fetching, but we still can get benefit on the aspect of upload process. Finally pdf file arrange file data based on pages, there is no relation on storing between two independent pages. In addition, PDF has a header and trailer metadata at the front and the end, we need to collect both of them firstly. Furthermore, there are many pairs of control message in a pdf file, missing one of the pairs would cause error, as a result we still can apply BFC mechanism to pdf file but need specially handle.

4.2.2 File size

When we apply BFC mechanism to a file, extra overhead is necessary to be added into the file, and we also need extra workload on handling BFC mechanism, including the BFC initialization, recognition, transmission and execution. As a result, if a small size file apply BFC, not only it cannot obtain better performance, but also increase the device's workload even if the format is fit to use BFC mechanism. So, we suggest that if any file size smaller or equal to a constant, says τ, we use WFC mechanism directly. Otherwise, we will determine it by the next factor.

4.2.3 Templates

After determine the selection of WFC and BFC, as every format (or application) has it's own characteristic, we build some templates to assign different cache setting based on different file format and application. A brief description for the supported templates will be introduced in the following:

BFC Template

For files which size are larger than τ and formats are suitable for BFC's application. Here we define three BFC templates for several different situations.

- **BFCFIX** is appropriate for the file that we seldom or even never modify, but we can get benefit on fetching, such as the video and audio file, .avi, .mp3, etc. BFCFIX divides file into fix blocks.
- **BFCVAR** aimed to the ASCII text-based file, which user make insertion and deletion commonly to the file. File format such as .c, .cpp, .txt, .jpg and the calendar file, etc, can

have greater performance by using the BFC. However, if this kind of files is set to unable to be written, we use WHOLEFILE template.

- **BFCSPEC** particularly focus on the files that metadata locates both at the beginning and the end, for example, pdf files. Moreover, pdf file builds file data page by page, so we divide the file blocks to match the pages. In addition pdf file data also include control messages which exist in pair, we need to keep any of a pair into the same block.

WFC's Template

- **WHOLEFILE** is appropriate to small files which size is smaller than τ, and which formats belong to Mircosoft Office or other unmentioned format, or a text-based file and its file mode(in inode) is set unable to write. WHOLEFILE template treats file without dividing the file and adding any overhead to inside.

Fille type	Cache mechanism	Template	Remark
All	WFC	WHOLEFILE	Size $<\tau$
avi, mp3, flv	BFC	BFCFIX	Block size fix
txt, c, cpp, calendar, jpg	BFC	BFCVAR	Block size variable
pdf	BFC	BFCSPEC	Divided page by page
Mircosoft office series & OTHERS	WFC	WHOLEFILE	

Table 4.1: Templates

Dynamic Regulation

In some special case, the WFC file will exceed the size of τ after user's modification. Under this situation, upload will still apply WFC. Then both server and client apply the divide function to the target file and produce snapshot for each blocks. After that, the cache strategy will be changed to BFC. Besides if a big BFC file size decrease to lower than τ, we take the similar way to handle and change to WFC mechanism.

4.2.4 Strategy

We introduced two criteria to determine template for WFC and BFC mechanism, Fig. 4.2 shows the cache mechanism selection strategy.

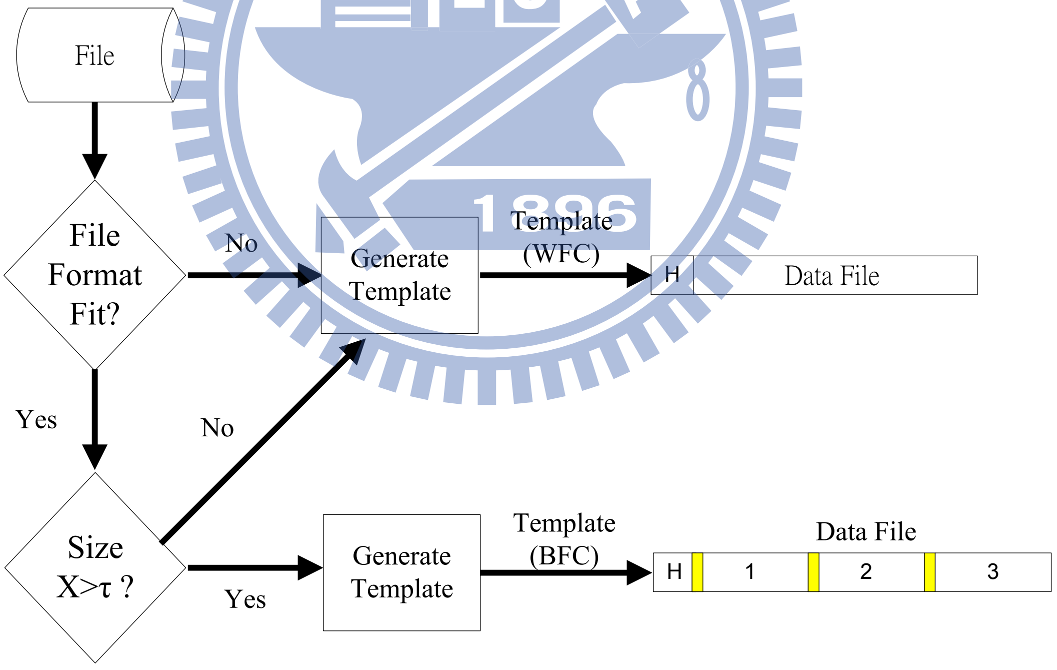

Figure 4.2: Template selection flow

In the following we talk about when and where HCM will be started and who takes it.

The initialization is mainly took charge by HCM server. Firstly, users prepare some files in server, and then server runs the initialization to identify the BFC-suit file, and build snapshot and super snapshot. In some case, for example, user may create a new BFC-suit file with some content, if the file size finally exceeds τ, HCM runs initialization in both client side and server side after client upload the file to server. Afterwards, all the file modification will be handled in the HCM BFC mechanism.

In read operation, HCM client first check the target file existence in local, if it is located in one of the local caches, then it is cache hit and we read it directly and execute according to the ``TEMPLATE" term in super snapshot. Otherwise, we connect to server to ask for file if it is cache miss. HCM client first gets the metadata of target file from HCM server. Next client recognize the cache mechanism according to the file format and file size. Finally HCM client sends relative request based on the mechanism.

In write operation, we suggest taking the write-on-close scheme. When we close a file, we check the dirty bit to recognize the modification was happened or not. If the target file is WFC mechanism, HCM client writes the file entirely to HCM server; else if it is BFC mechanism, HCM client first check the ``CODE" of every blocks to collect the dirty blocks, and then send a BFC-based request to HCM server.

4.3 Initialization

As shown in Fig 2.1, the structure of a file for applying WFC is simple, just include the file header (metadata) and also a single data block without any extra overhead, thus it is unnecessary to take any preprocess on the target files. On the other hand, Fig.4.3 depicts the structure of file to apply BFC does. We can see that except for the header and the data blocks, we also define three more components ``breakpoint", ``snapshot" and ``super snapshot" to approach the BFC function. Therefore, an initialization preprocess is necessary to build up a

BFC-suit file. As a result, after HCM determined the template of a file, if the template is

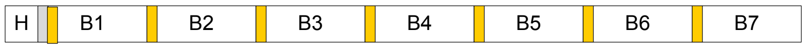

Figure 4.3: BFC-use file structure

belongs to BFC template, HCM then executes the initialization process. As discuss before, files to apply BFC mechanism need to add extra information.

The brief description of BFC Initialization preprocess is shown in Fig. 4.4. There are several parts in the initialization flow: Segmentation, Digest, Identification, Snapshot and Super snapshot.

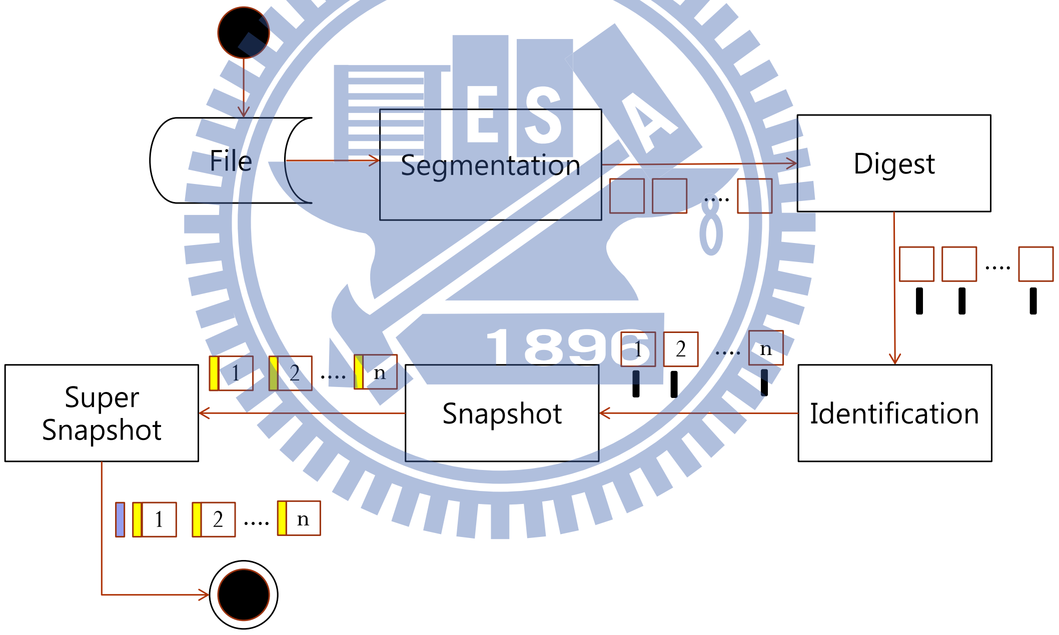

Figure 4.4: BFC Initialization preprocess

4.3.1 Segmentation

The first thing we do is to divide up files into multiple blocks. If the template is ``BFCVAR", then we use we use the Rabin Fingerprint algorithm [13] [14] to get a variable size boundary, with the suggested average block size of 8KB (probability 2^{-13}), and the range of block size from 2KB to 64KB setting. Rabin Fingerprint is a polynomial representation of data modulo a predetermined irreducible polynomial. The reason we choose Rabin Fingerprint is that it provide an effectively computation on a sliding window on block segmentation, and we can easily have random size non-overlapping blocks. Furthermore, we claim that the block boundary should be variable, the boundary is determined based on file contents rather than by certain position, therefore insertions/deletions only affect the surrounding blocks, avoid causing the problem of large file contents shift even only single bytes is change at the start. On the other hand, if the template is ``BFCFIX", we suppose that this kind of file will never be modified, we determine the breakpoint directly by the position, with block size 8KB. Finally if the template is ``BFCSPEC", we determined the breakpoint based on certain file content, take pdf file as an example, we set the breakpoint at the end of one page.

4.3.2 Digest

After dividing a file into blocks, we then give a digest to each block by the SHA-1 hash function [7] [15], in order to simplify the comparison of two blocks. Sha-1 hash function take a input data which size limit to $2^{64} - 1$ bits, and then output a 160 bits hash value. Theoretically, only equal contents can obtain same hash value. We can compare the equality of two blocks easily through the respectively digest. (In 2005, Wang, X. and Yin, Y.L. and Yu, H. announced an attack method on breaking SHA-1 [16]. They indicate that they can find SHA-1 collision in less than 2^{63} operations, where a brute-force search would require2^{80} operations. Here we still use SHA-1 because there only contain at most dozens thousand blocks in a file, and we do not use the hash code into security way, we think that SHA-1 is enough for our design. Additionally, the output of SHA-1 is smaller than upper SHA family [17]).

4.3.3 Identification

Now we divide file into block, and give each block a digest for easily compare, in the next stage we assign identification information to each blocks for searching the blocks more effectively. In LBFS, they use a term ``offset" to find out the target block. Nevertheless, we suppose that this method would be suffer from the issue of block insertion/deletion, any insertion/deletion happen at the start may cause a large scale of ``offset" shift, then we need to modified the shifted blocks' offset record. In order to avoid the shift-offset issue, we assign a term ``BLOCK ID" to instead, then search block by block with ``BLOCK ID". However, this method may avoid the shift-offset issue, but simultaneously produce another issue--longer search time. When we search in a unsequential link-list, the BIG O is O(n) (n:the number of blocks), compare with the ``offset" method, the search efficiency is significant decrease from O(1) to O(n).

To overcome the issue, we propose to assign one more term ``GROUP ID" joining ``BLOCK ID". We arrange all the blocks in to groups, each group has an id and contain 10 blocks at default. The group size is variable, any block insertion/deletion only affect that group itself. At this situation, according to the condition of ``GROUP ID" and ``BLOCK ID", the search time decrease dramatically from O(n) to O(g). (g:average group size, approximately 10 at default). If a certain group experienced a series of insertion and group size increase to more than 30 blocks, then we can execute a regrouping process, allotting blocks to the surrounding groups until group size reduce to 20.

To maintain this method in memory, we store the blocks in a composite data structure combining an array and lists, we call it *aList*. Utilize the array as the group id indexer, and simultaneously have a pointer point to a link-list that store the grouping blocks. The initial process of aList is illustrated in Fig. 4.5. First we open a file from disk then store it in memory with a char * array buffer, finally allot the blocks to aList by the ``GROUP ID" and ``BLOCK ID". Now we can find out the target file effectively with ``GROUP ID" for array index and ``BLOCK ID" for link-list search.

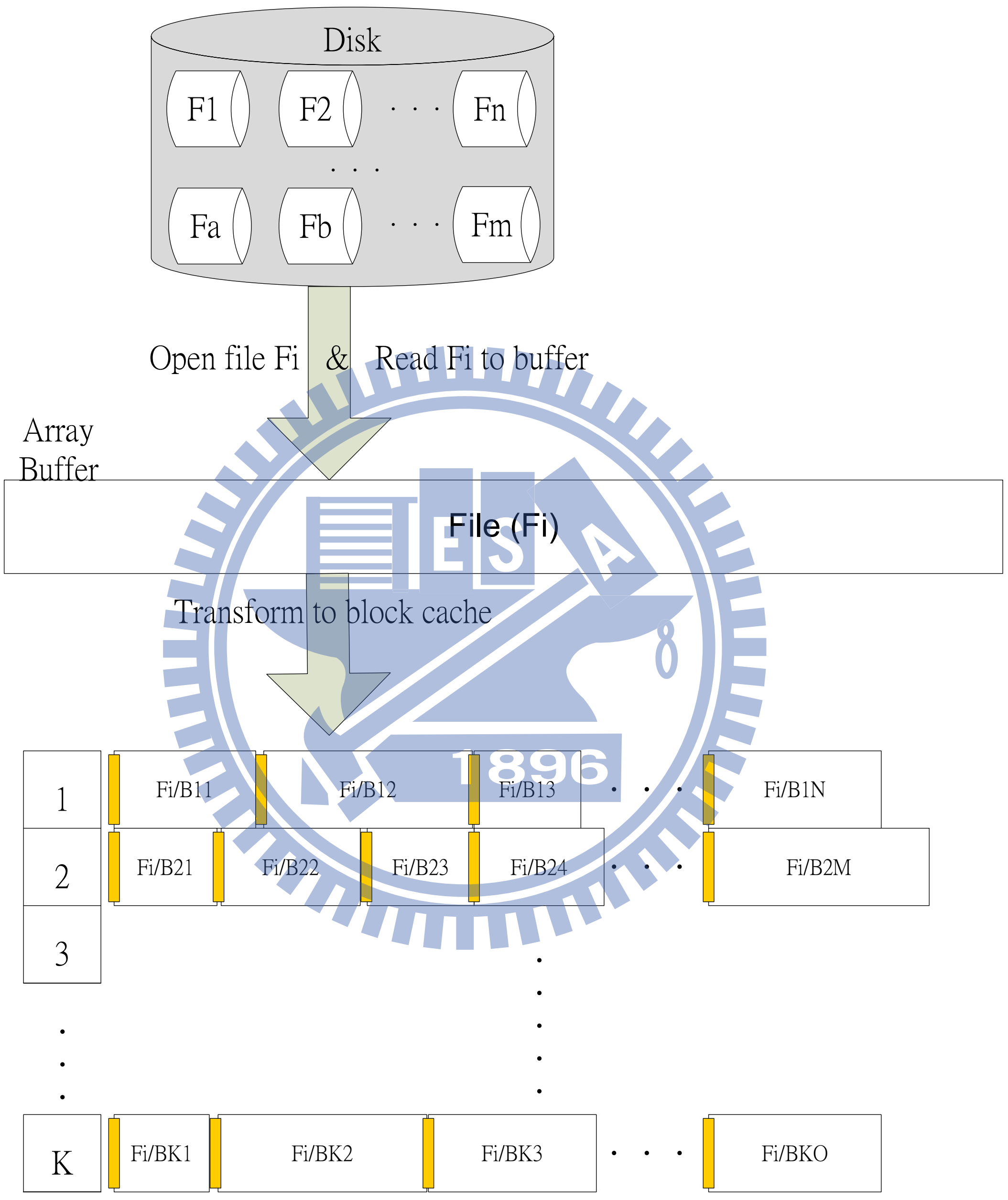

Figure 4.5: Using Array and Link-listed on BFC mechanism

4.3.4 Snapshot

After dividing the file, assign digest and ID to each blocks, we build up a snapshot to store this information, and then save it in front of every block at the breakpoint.

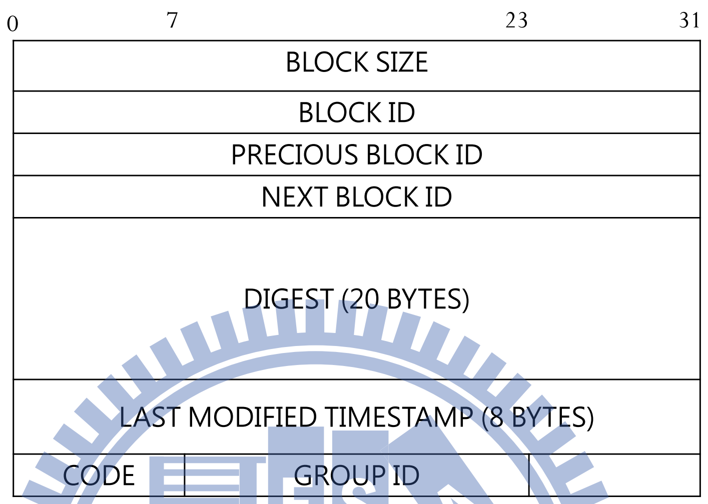

Figure 4.6: Snapshot composition

In our snapshot contents (Fig.4.6), except the SHA-1 hash digest with 20 bytes, ``BLOCK ID'' and ``GROUP ID'', we also define several terms for recording the block state to enhance the efficiency of BFC mechanism, they are described in the following:

- **BLOCK SIZE**: Record the size of this block. Use to allocate memory.
- **PRECIOUS BLOCK ID**: Record the precious block id of current block.
- **NEXT BLOCK ID**: Record the next block id of current block.
- **LAST MODIFIED TIMESTAMP**: Record the last modified time.
- **CODE**: a code stand for ``CLEAN'', ``MODIFIED'', ``CREATED'', ``DELETED'', ``REGROUP'' to express the block state. ``CLEAN'' means no any modification is done on the block since last update; ``MODIFIED'' means the block content is changed and

wait for update to server; ``CREATED" means we create a new block, simultaneously to indicate server to run an insert progress after receive this kind of snapshot; ``REGROUP" means that block has happened to be regrouped. Moreover, as we need a way to indicate to server that we have done a deletion, we also define a term of ``DELETED" stands for an empty block which data is cleared by user. Here we still hold the snapshot in our system even the block content is cleared, then we send the ``DELETED" snapshot to server according to the update progress, after the server receive the snapshot and detects the ``DELETED" code, server searches the certain block and then delete the block and also the snapshot to finish the action of deletion.

When the ``MODIFIED" and ``CREATED" and blocks are updated to server, then we turn them to ``CLEAN"; On the other hand, after the ``DELETED" block(``DELETED" always means empty block) is updated, we will delete the snapshot.

Totally, each snapshot is 48 bytes (with 1 byte empty for option data), which is similar to the LBFS recommended.

4.3.5 Super Snapshot

We have finished the most of the initialization of BFC-use file. Finally we collect some entire information from the initialized file that snapshot does not record but necessary to the BFC mechanism, then we save the information into a structure called super snapshot. Fig. 4.7 shows the super snapshot contents.

- **VERSION**: Record file version with float point, auxiliary term to the consistency issue.
- **QUANTITY of BLCOK**: Record the number of blocks in file. Use to open an aList;
- **QUANTITY of BLCOK** : Record the number of group in file. Use to open an aList;

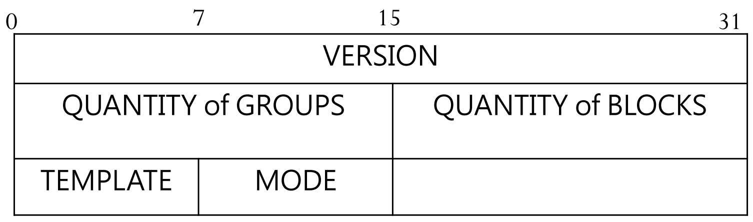

Figure 4.7: Super Snapshot composition

- **TEMPLATE**: record the template type the file use in BFC mechanism(section4.2.3).

- **MODE**: Indicating to other users that this file is being accessed or not.

4.4 Handshake

4.4.1 WFC

File Read

The read process between server and client in WFC is shown in Fig.4.8. The parameter that WFC server need is file path. Once server receive a request from client, firstly server reply the metadata of the target file to client for some checking, then server transmits whole target data file to client if still necessary, finally client store the replica into local cache for the follow-up application. Here we can find a term T_{resp} in the Fig.4.8. It is defined as the total time from the first request start to the end of file transmitting. In other word, how much I need to execute the file after my request. In WFC, applications need to wait for file execution until the file finishes the transmission.

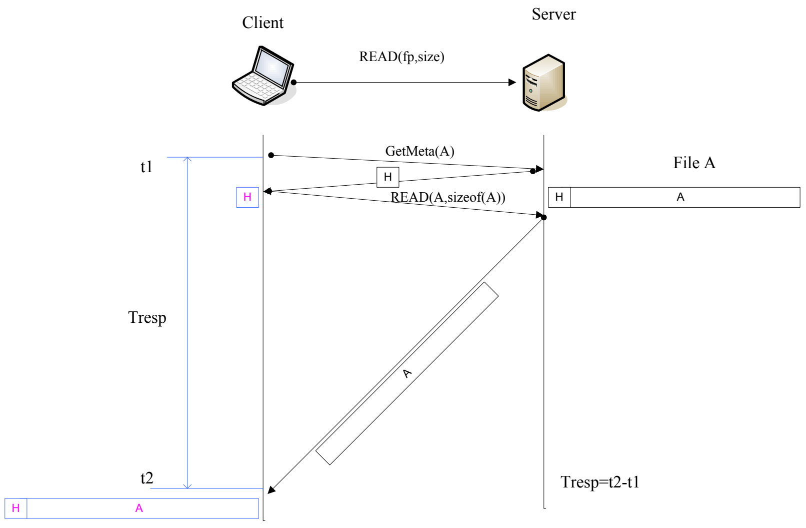

Figure 4.8: WFC-downlaod

File Write

The WFC write process is very similar to the read process, which illustrated in Fig. 4.9. Write process usually follow-up by consistency issue, different file system has different solution on this issue. Here we do not discuss the issue of file consistency, users can have their own setting and mostly will not affect the function HCM does.

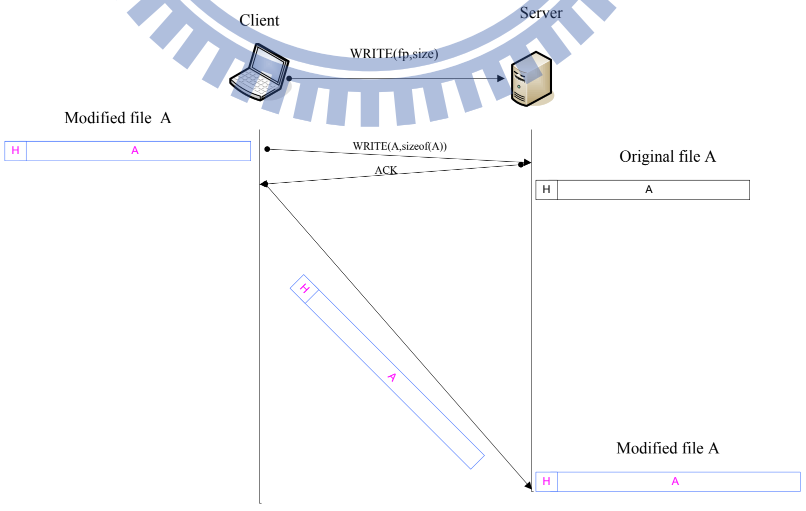

Figure 4.9: WFC-upload

4.4.2 BFC

File Reads

The read flow between server and client in both BFC mechanism is shown in Fig.4.10, respectively. Firstly, a request is sent to server, then server responses the requested the header, the snapshot and also the super snapshot of target file (blocks) to client immediately. BFC mechanism needs not only the file path parameter but also the number of request blocks and the start block id of the target blocks. After receiving the snapshot, client checks the existence of the target blocks through the information in snapshot. Then client require again if it misses the target blocks. Finally server request the target blocks to client. We can see that there is numbers of round trip between server and client if we use BFC mechanism to fetch every block. But of course we can stop fetching at anytime we are enough then consume the unused part's bandwidth. On the other hand, we can see that the T_{resp} of BFC mechanism would be much smaller than the WFC's in 4.4.1 if the file is large enough. T_{resp} is one of the main advantage of BFC, we can get great performance as long as we apply BFC mechanism appropriately.

File Writes

Fig.4.11 depicts the write flow of BFC. In BFC mechanism, client only uploads the modified blocks to server, called block-update, whereas WFC uploads the entire file to server in every modification, even only several characters were changed. This is another main advantage of BFC mechanism. In many case, there are still many blocks in a modified file stay unchanged, utilize those unchanged blocks can save bandwidth efficiently. On the other hand, block-update avoids the potential damage on two or more writing the same file. Block-update copy an original file as a temp file, after the deletion and insertion of the modified blocks on

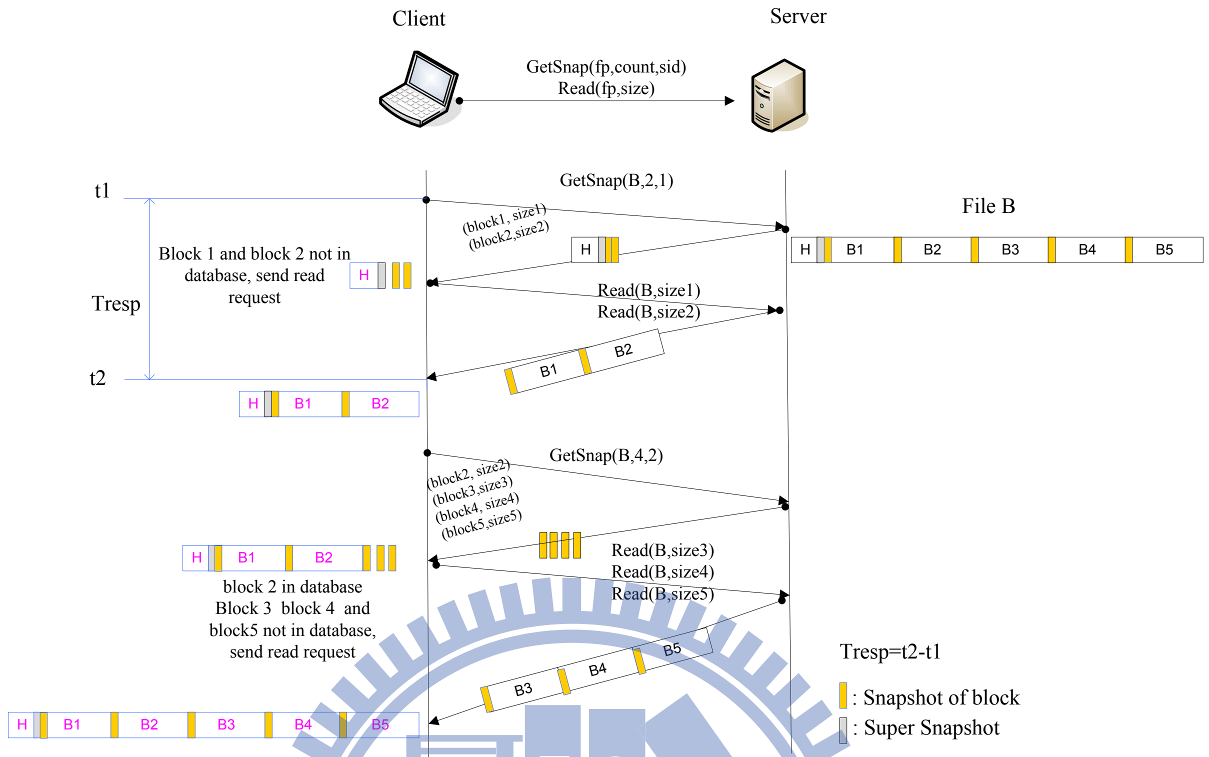

Figure 4.10: BFC-download

server side, system replace the temp file to the original file as a new version.

In BFC mechanism, once a block is modified or created/deleted, HCM assigns a new SHA-1 hash data for new content, and the dirty bit (the term ``CODE") in snapshot will be set until the dirty block be uploaded. In Fig.4.11, block-update firstly check the dirty bit in snapshot to identify the modified blocks. After that client uploads the dirty snapshot to server to check the status or of target block in server side. (Normally dirty bit on means block was modified, but sometimes we may cancel the modification we did, or that block also be modified the same content in server side simultaneously, so we still check the validation.) Then the server creates a temp file to handle the process of block update and file blocks reconstruction. Next server responses message to client to require target blocks. After that client delivers the missing blocks to server, and then sends a commit message to notified server that upload finish. Finally server replace the original file by the temp file as a new version, write flow finishes.

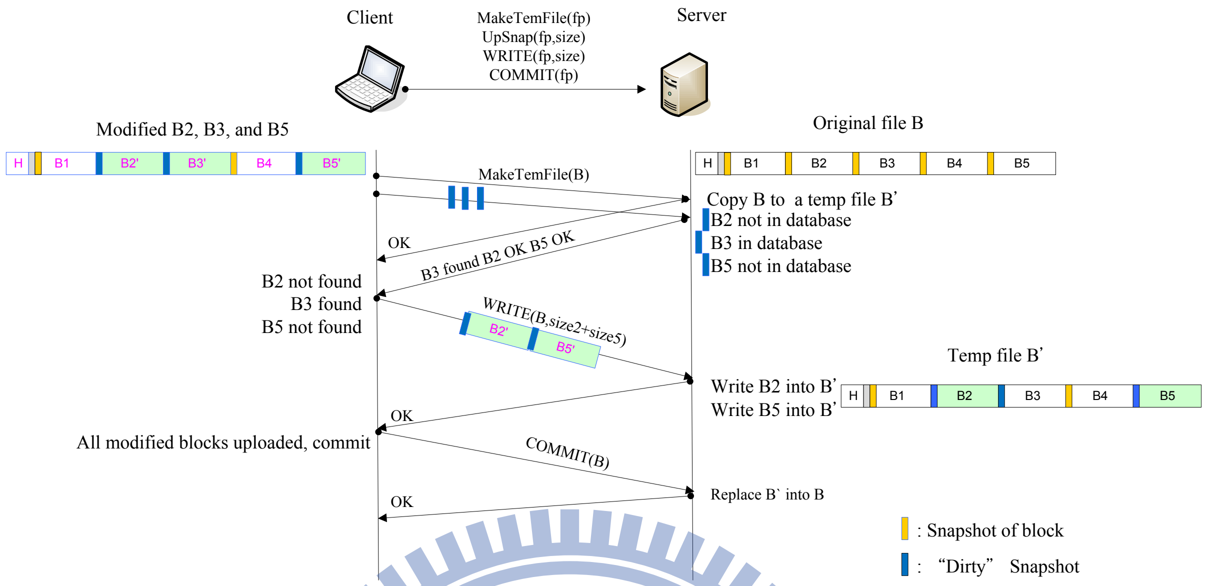

Figure 4.11: BFC-upload

4.5 Consistency

In HCM, we suppose that filesystem designers will have their own consistency model, so we do not discuss the consistency scheme in this paper. Instead, we provide two ways to help handling the consistency issue. First we define a term ``CODE" in snapshot to easily recognize the block's modification status, we have discuss this term before in 4.3.4; Moreover, we define two terms ``MODE" and ``VERSION", we not only can check the validation directly through ``VERSION", but we also can know how much version we have missed. ``MODE" means that what accessing mode is the file. That is, if someone is writing to server, then ``MODE" will be set; and record how many users are writing the target file, then user can determine whether start write his own file to server, or to wait until other users finish writing. ``CODE" and ``MODE" have not completely defined, file system designer can develop new function based on their own requirement.

For example, user A is writing to server, the term ``MODE" was set; at the same time another user B start to write. Under this situation, server can send a conflict notification to user B, if user B determines to continue his writing, server can overwrite the preceding file, or rename each conflict file with version and time.

Another example, users may be notified by server that their upload files are not the freshest through the term ``VERSION". In this case server may reject the write request or rename the target file name like the preceding solution.

Chapter 5

Analysis

In this chapter we show some test result to illustrate the BFC overhead and performance. All the experiments are held in

- **Server side** :Intel P4 3.0GHZ ,1G RAM, 10/100M NIC LAN CARD
- **Client side** :Intel P4 3.0GHZ, 512 RAM, 10/100M NIC LAN CARD

All the local experiment is tested on the server side.

5.1 Storage Cost

The total size of a file using WFC mechanism can be presented as follow:

$$S_{WFC} = S_H + S_D$$

Where S_{WFC} stands for total file size in WFC , S_H is the size of header, S_D is the size of the data block.

On the other hand, the total size of a BFC file can be expressed in:

$$S_{BFC} = S_H + \lceil \frac{S_{TB}}{S_{\overline{B}}} \rceil \times S_{SN} + S_{TB} + S_{SSN}$$

where S_{BFC} stands for total size of file using BFC , S_H is the size of header, S_{TB} is the size of all the blocks, $S_{\overline{B}}$ is the size of average blocks(block size may be variable), S_{SN} is the size of snapshot, S_{SN} is the size of super snapshot, and the $\frac{S_{TB}}{S_{\overline{B}}}$ means the number of total blocks of a file.

We can find that total block size in BFC is equal to the size of data block in WFC, that is $S_{TB} = S_D$. So the extra storage cost of BFC mechanism is all the snapshot size and the super snapshot size, $\lceil \frac{S_{TB}}{S_B} \rceil \times S_{SN} + S_{SSN}$.

The snapshot size is 48 bytes, average block size 8K bytes, so the average overhead of a file to use BFC mechanism in disk is about 0.58% more than WFC mechanism (Moreover super snapshot 12 bytes, but it can be almost too small to ignore). Here the average block size can be changed by the designer. If average block size increase, quantity of snapshot decrease, but relatively every single block uploaded workload increase, and vice versa.

We ran initialization process to build snapshot for showing the overhead size and percentage. Here a 50MB file is applied, with the average block size of 4KB, 8KB and 16KB. The result is shown in Table. 5.1. We can see that larger average block size with lower overhead, but relatively higher transmission cost per block update.

Average block size	Quantity of blocks	Total overhead size	Overhead percentage
4KB	8005	384240 Bytes	0.768%
8KB	4806	230688 Bytes	0.461%
16KB	2693	129264 Bytes	0.259%

Table 5.1: Average block size vs. Overhead

5.1.1 Block Size Distribution

We apply Rabin Fingerprint to determine the variable boundary for BFC-use file. Fig. 5.1 illustrates the block size distribution of a 50MB, with 4806 blocks and average size 8KB setting file. We can see that the block size centralize to around 8KB.

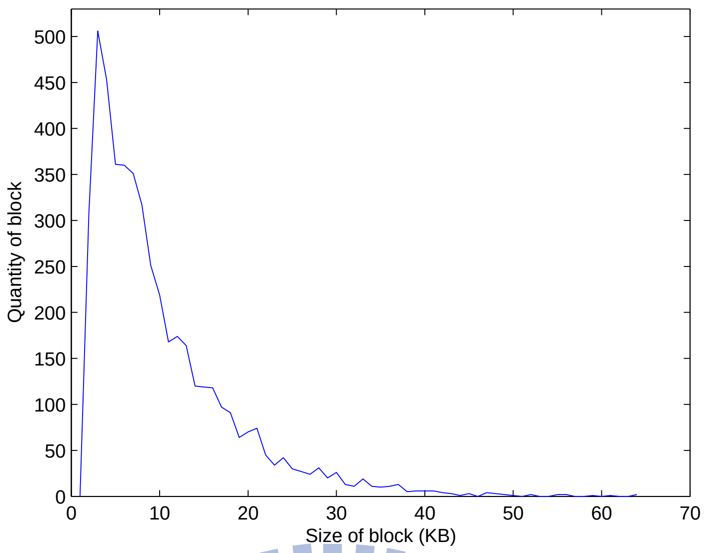

Figure 5.1: Block Size Distribution

5.2 Time Cost

Except the disk maintain overhead, BFC mechanism takes more round trip time on handshake, and extra CPU workload on BFC operation such as snapshot initialization and aList initialization, block location and block comparison, etc than WFC mechanism. We will show

We executed the snapshot initialization and aList initialization with a 50MB, average block size 8KB file, the respectively time cost is shown in Table. 5.2. Snapshot initialization takes only one time per file, and aList need to be maintain in every time file to be opened. We suppose that both this two extra cost is acceptable.

	Time
Snapshot Initialization	19.9 s
aList Initialization	0.09 s

Table 5.2: Time cost of two initialization process

5.3 Block Search

To test aList structure performance, we search 10,000 random blocks in both link-list and aList in 10MB file (995 blocks) and 150MB file (14363 blocks), Table 5.3 shows the result. We can see that the search speed of aList is much faster than the link-list dose, and the larger the file is, the bigger the search time ration. The data also show that we can still find out the block in a very fast speed with aList structure even if the file size is very large.

File Size	Link-list Time	aList Time	Time Ratio
10MB	1.23 s	0.0025 s	≒ 500
150MB	9.45 s	0.0063 s	≒ 1500

Table 5.3: Block search time of aList vs. Link-list

5.4 Bandwidth Utilization (HCM vs. WFC vs. BFC)

5.4.1 Read Cost

We applied the three cache mechanism to three different size range data sets respectively to test the efficiency of read and write. A small file set of the random size range from 50B to 2KB with 100 files, the total size is 55KB; a big file set of 50KB to 50MB range with 30 files, the total size is 439.24MB and finally a file set mixing all the small and big file set with 130 files and 439.29MB size. In this test, we set the τ as 128KB, all block size ad 8KB and all the files in file sets can be BFC-based file.

Fig. 5.2 illustrates the read cost in applying both HCM, simple WFC and simple BFC mechanism to the three file sets. In order to show the result significantly, we magnify the scale of the small file set's. We can see that in small file set, the network traffic of simple BFC is about 10% larger than HCM's and simple WFC's. This is because of the snapshots overhead of BFC mechanism and the size of overhead is relatively large to the file itself. On the other hand, the files in small file set are all smaller than τ, as a result HCM all uses WFC mechanism to transmit the file and without any overhead, that's why the traffic of HCM and WFC is equal.

In the big file set, we can see that both HCM, WFC and BFC are very close, but simple WFC takes the least as no overhead is added to the files.HCM takes BFC mechanism if file is larger than 128KB and takes WFC mechanism if smaller, so the traffic of HCM is between simple WFC and simple BFC. The result of all file set is similar to the big one as the total size of small file set possess relative very little percentage on the total size.

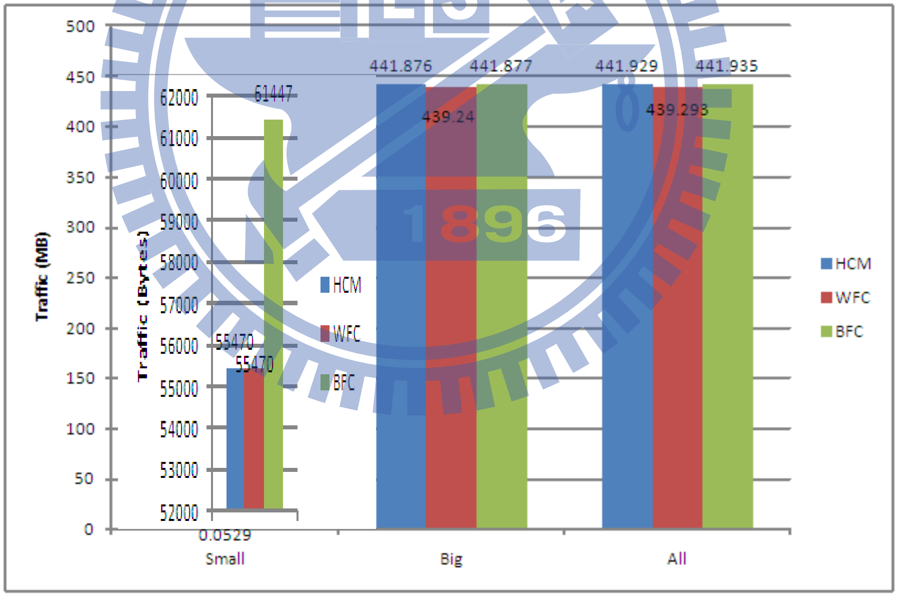

Figure 5.2: HCM Read Cost

5.4.2 Write Cost

Fig 5.3 5.4 5.5 illustrate the write of both three file sets. In the traffic of small file set, it's the same result as read cost did, simple BFC performs about 10% larger than HCM and simple

WFC with the same principle. However, in the result of big file set we can see that simple BFC can be much smaller than simple WFC if the user only writes several blocks. This is because simple WFC mechanism writes the while file to server in each write operation no matter how much the target file was modified but simple BFC mechanism only writes the modified blocks in each write operation. Here HCM takes BFC mechanism if file is large enough, thus the result of HCM and simple BFC is very close, just like result in read cost does. Finally the result of all file set is similar to the big file set does because of the little percentage of small file set possessing in total size.

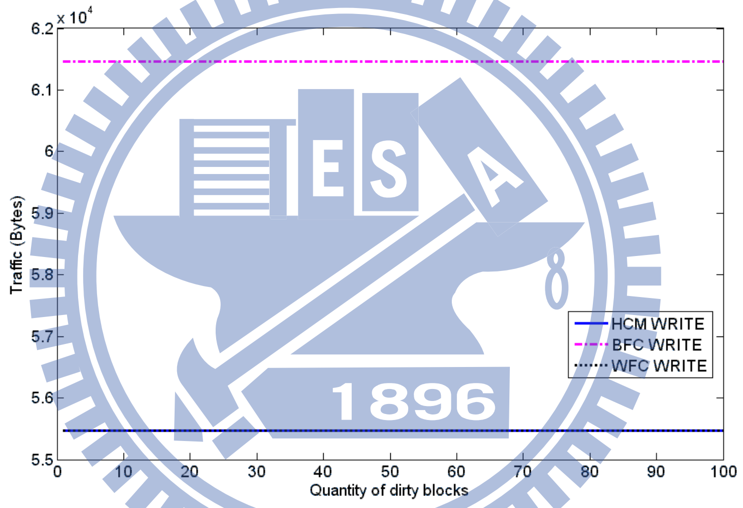

Figure 5.3: HCM Write Cost of Small Data Set

We can conclude that simple BFC can gain significant benefits in right situations and HCM try to cover the remainder situation.

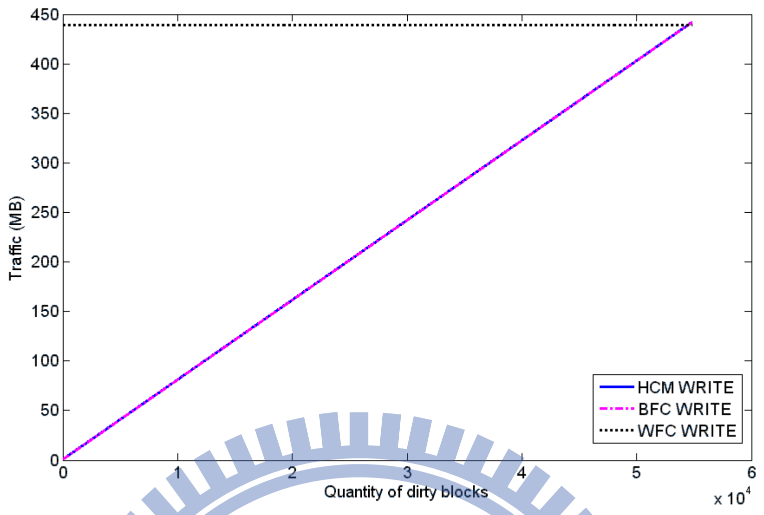

Figure 5.4: HCM Write Cost of Big Data Set

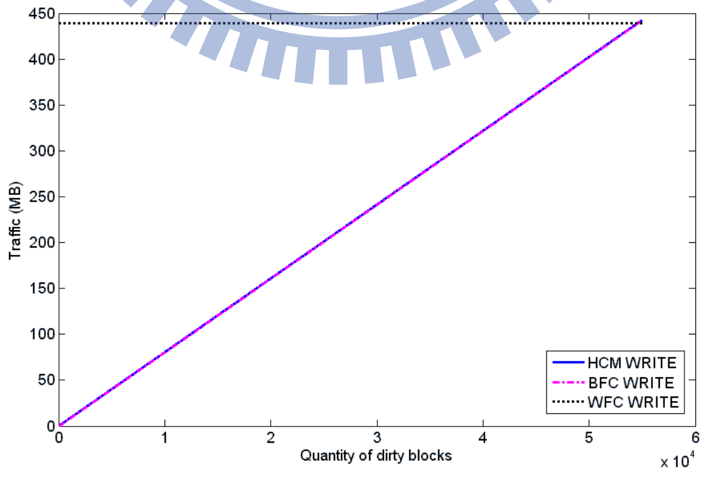

Figure 5.5: HCM Write Cost of All Data Set

Chapter 6

Conclusions

HCM mechanism reduces bandwidth consumption during upload stage and enhances system efficiency by hybridizing two cache models into a file system, gathering the advantages of WFC and BFC, and complement the mutual shortcoming. HCM define templates to determine the cache model selection based on file size, file formats and the file states. In the BFC mechanism of HCM, HCM divide file into blocks based on content, using hash value to distinguish the blocks, assign some id to each group for block searching, and finally collect this information and build in a snapshot located in front of the data block. Although we add overhead to data, file system using HCM can get better performance than traditional mobile file system under some situations, such as editing documents or calendar and then update, or read a pdf document but not reading thoroughly.

Implement HCM takes reasonable initialization cost on storage and time, besides aList solves the offset issue and increase the speed of block search significantly. HCM mechanism uses BFC mechanism in large file and BFC in small file, which gains advantage in both read and write operation. That is HCM increases a bit read cost but reduce write cost significantly, which cover the shortcoming of simple WFC and simple BFC.

References

[1] J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3--25, 1992.

[2] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51--81, 1988.

[3] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*. ACM New York, NY, USA, 2001, pp. 174--187.

[4] D. Dwyer, "Adaptive file system consistency for unreliable mobile computingenvironments," in *IEEE International Computer Performance and Dependability Symposium, 1998. IPDS'98. Proceedings*, 1998, pp. 164--173.

[5] C. Yang, "A Hybrid File-Caching Scheme for Distributed File Access."

[6] K. Froese and R. Bunt, "Cache management for mobile file service," *The Computer Journal*, vol. 42, no. 6, pp. 442--454, 1999.

[7] S. Standard, "FIPS Publication 180-1," *National Institute of Standards and Technology*, 1995.

[8] I. Voras and M. Zagar, "Network distributed file system in user space," 2006, pp. 669--674.

[9] L.-Y. Chang and Y.-L. Huang, "Ashfs: A lightweight mobile file system supporting disconnected operations," Master's thesis, National Chiao Tung University, 2008.

[10] T. Schutt, A. Merzky, A. Hutanu, and F. Schintke, "Remote partial file access using compact pattern descriptions," in *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004*, 2004, pp. 482--489.

[11] "File format resource library," Tech. Rep. [Online]. Available: http://www.wotsit.org/

[12] "Open office organization," Tech. Rep. [Online]. Available: http://www.openoffice.org/index.html

[13] M. Rabin, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[14] X. Li, D. Salyers, and A. Striegel, "Improving packet caching scalability through the concept of an explicit end of data marker," *Proc. of IEEE HotWeb*.

[15] C. De Canniere and C. Rechberger, "Finding SHA-1 characteristics: General results and applications," *Lecture Notes in Computer Science*, vol. 4284, p. 1, 2006.

[16] X. Wang, Y. Yin, and H. Yu, "Finding collisions in the full SHA-1," *Lecture Notes in Computer Science*, vol. 3621, pp. 17--36, 2005.

[17] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott, "Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512," *Lecture notes in computer science*, pp. 75--89, 2002.