國立交通大學

資訊科學與工程研究所

碩 士 論 文

在基於非揮發性隨機存取記憶體系統上提供改善作業系統效能的機制

Operating System Support on NVRAM-Based Systems

研 究 生：蕭智文

指導教授：張瑞川 教授

中華民國九十七年七月

在基於非揮發性隨機存取記憶體系統上提供改善作業系統效能的機制

學生：蕭智文　　　　　　　　　　　　指導教授：張瑞川教授

國立交通大學資訊科學與工程研究所

論 文 摘 要

硬碟效能長久以來一直是電腦系統中一個主要瓶頸，根據 Moore's Law，處理器的效能以每年 60%的速度增進；而硬碟的效能僅以每年 10%的速度增進，因此在處理器和硬碟之間產生了極大的效能差距。非揮發性記憶體(Non-volatile RAM)是最近幾年新興的一項記憶體技術，其具有非揮發性的特性可以有效改善系統效能。由於傳統緩衝快取以及確保檔案系統資料一致性的設計都是建立在揮發性記憶體上，因此我們在非揮發性記憶體性統上提出三種機制，來進一步增加緩衝快取的效能以及減少確保資料一致性的負擔。第一，我們在 VFS 和一般檔案系統之間加入一層 Ram-based 檔案系統來暫時的存放新建立檔案，其目的爲延遲檔案分配給檔案系統的時間以避免檔案零碎的情形以及減少不必要的 IO。第二，我們修改原本以檔案爲單位的寫回策略，而進一步的考慮寫回的區塊在磁碟上的相對應位置，讓較連續且較鄰近的資料一起寫回，以減少寫回的時間；另外，我們還確保不要寫回最近剛被更新過的頁面。最後，我們提供了一個簡單機制可同時確保檔案系統的一致性而不需要額外的磁碟 IO 負擔。實驗結果顯示我們的機制之效能優於 Ext2 大約 76%；優於 Ext3 大約 94%。

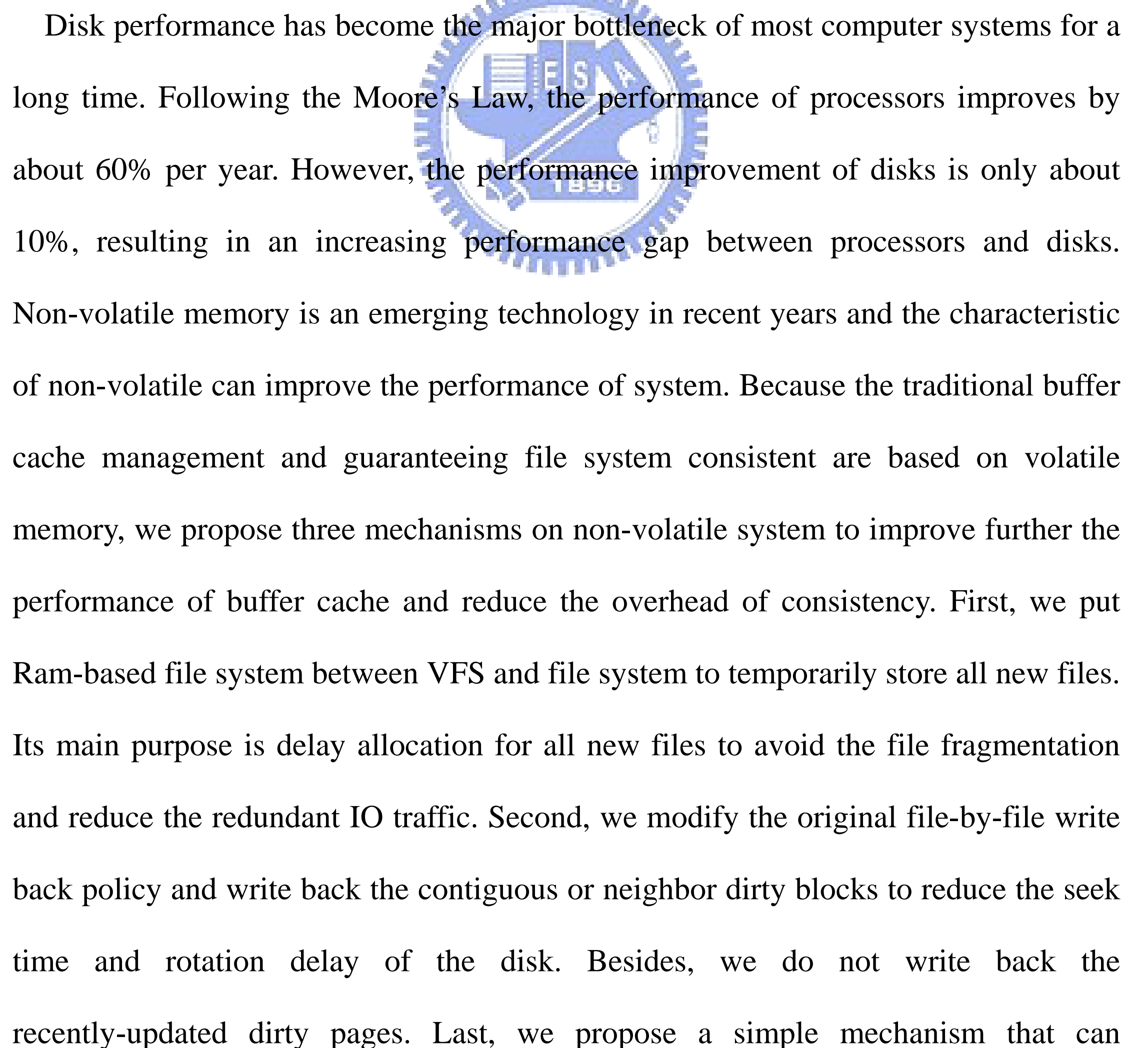
Operating System Support on NVRAM-Based Systems

Student: Chih-Wen Hsiao **Advisor: Prof. Ruei-Chuan Chang**

Computer Science and Engineering College of Computer Science

National Chiao Tung University

Abstract



Disk performance has become the major bottleneck of most computer systems for a long time. Following the Moore's Law, the performance of processors improves by about 60% per year. However, the performance improvement of disks is only about 10%, resulting in an increasing performance gap between processors and disks. Non-volatile memory is an emerging technology in recent years and the characteristic of non-volatile can improve the performance of system. Because the traditional buffer cache management and guaranteeing file system consistent are based on volatile memory, we propose three mechanisms on non-volatile system to improve further the performance of buffer cache and reduce the overhead of consistency. First, we put Ram-based file system between VFS and file system to temporarily store all new files. Its main purpose is delay allocation for all new files to avoid the file fragmentation and reduce the redundant IO traffic. Second, we modify the original file-by-file write back policy and write back the contiguous or neighbor dirty blocks to reduce the seek time and rotation delay of the disk. Besides, we do not write back the recently-updated dirty pages. Last, we propose a simple mechanism that can

guarantee the file system consistent without any overhead of disk IO. The experimental results show that the performance of our mechanisms is superior to Ext2 about 76%, and Ext3 about 94%.

致謝

感謝我的指導老師 張瑞川教授以及 張大緯教授，兩位老師對我的論文所做的指導，感謝在系統實驗室上所有的學長和同學對我的幫助和建議，另外還要感謝張大緯教授兩年來不遲辛苦的和我們遠端視訊做討論，真的很辛苦他。

最後感謝我的家人，在我求學兩年中最我的支持和關心。

TABLE OF CONTENTS

論文摘要……………………………………………………………………...…..…i

Abstract…………………………………………………………………….…ii

致謝……………………………………………………………………...………iv

Table of Contents……………………………………………………....………v

List of Figures…………………………………………………………………vii

List of Tables…………………………………………………….………...ix

Chapter 1 Introduction…………………………….…………………………...1

1.1 Motivation…………………………………………...……………1

1.2 Our Mechanisms………………………………………………….………2

1.3 Structure of the Thesis……………………………………….…………4

Chapter 2 Related Work…………………………………………………….………5

2.1 System Recovery………………………………………………………5

2.2 NVRAM as Storage Device……………………………………………6

2.3 NVRAM as Buffer……………………………………………………...7

2.4 Transaction Support……………………………………………………9

Chapter 3 Design and Implementation…………………………………………… 11

3.1 Background……………………………………………………………11

3.1.1 Temporary-File File System……………………………………..11

3.1.2 Intelligent Write-Back Policy……………………………………15

3.1.3 Transaction Support on File System Operations……………………...18

3.2 Implementation and Integration of the Approaches………………………...22

3.2.1 Implementation of Three Mechanisms………………………………..22

3.2.2 Integration of Three Mechanisms……………………………………..28

Chapter 4 Performance Evaluation……………………………………………..29

4.1 Experimental Environment and Configurations……………………………29

4.2 The Performance Results of TempFFS……………………………………...32

4.3 The Performance Results of Intelligent Write-Back Policy………………...37

4.4 The Performance Results of Transaction Support on Ext2…………………41

4.5 Put it all Together……………………………………………………………45

Chapter 5 Conclusions………………………………………………………...51

References………………………………………………………………………...52

LIST OF FIGURES

Figure 3.1 Architecture of the Temporary-File File System……………………………13

Figure 3.2 Transformation Steps…………………………………………………..14

Figure 3.3 the Zone Information Table……………………………………………17

Figure 3.4 Zone with Different Average Segment Length…………………………...18

Figure 3.5 the Main Data Structure of Journaling…………………………………20

Figure 3.6 the Core Functions of Journaling ……………………………………...20

Figure 3.7 Pseudo Code of Segment/Zone Algorithm………………………………25

Figure 3.8 Implementation of the Transaction API (Pseudo Code) …………………27

Figure 3.9 Integration of the TempFFS, Intelligent Write Back and File Operation Transaction Support ……………………………………………………………28

Figure 4.1 Performance Improvements of TempFFS (Bonnie++)……..……………32

Figure 4.2 Performance Improvements of TempFFS (Untar and Compile Linux Kernel)……………………………………………………………………………33

Figure 4.3 Performance Improvements of TempFFS (Postmark)……………………34

Figure 4.4 Rations of Files Deleted in TempFFS (Postmark)………………………..34

Figure 4.5 Reduction of IO Traffic with TempFFS………………………………….35

Figure 4.6 Average File Distance (Postmark)……………………………………….35

Figure 4.7 Performance Comparison of the Write Back Polices (Bonnie++)……….37

Figure 4.8 Performance Comparison of the Write Back Polices (Untar and Compile

Linux Kernel)……………………………………………………………………..38

Figure 4.9 Performance Comparison of the Different Locative Write Back Polices...39

Figure 4.10 Performance Comparison of the Write Back Polices (Postmark)……….39

Figure 4.11 Accumulate Inter-requests Distance (Postmark)………………………...40

Figure 4.12 Repeated Write Back Blocks (Postmark)………………..…………….…..41

Figure 4.13 Performance of the Transactions Support Mechanism (Bonnie++)…..…41

Figure 4.14 Performance of the Transactions Support Mechanism (Untar and Compile Linux Kernel)……………………………………………………………………..43

Figure 4.15 Performance of the Transactions Support Mechanism (Postmark)…….44

Figure 4.16 Performance Results of Different Combinations (Bonnie++)…………46

Figure 4.17 Performance Results of Different Combinations (Linux Kernel Untarring and Compilation)……………………………………………………………………47

Figure 4.18 Performance Results of Different Combinations (Postmark, 512-10K bytes)………………………………………………………………………….……..48

Figure 4.19 Performance Results of Different Combinations (Postmark, 512-2M bytes)………………………………………………………………………………49

LIST OF TABLES

Table 1.1 MRAM and DRAM Characteristic……………………………………………3

Table 4.1 Evaluation Environment……………………………………………………...30

Table 4.2 Experimental Configurations…………………………………………………31

Table 4.3 Memory Overhead of the Transaction Support Under Different Workloads………………………………………………………………………….45

Chapter 1 Introduction

1.1 Motivation

The speed of processor is double every eighteen months by Moore's Law, but the performance of the computer system grows slowly due to the slow I/O. The disk I/O time is mainly dominated by seek time and rotation delay. Because data in volatile memory (DRAM) is not persistent, system must periodically write back dirty data to the disk for avoiding data loss due to the power failure. Therefore, performance of disk IO is worse.

With the well advances in non-volatile memory (NVRAM) technologies, many kinds of non-volatile RAM such as MRAM [9] (Magnetoresistive RAM), FeRAM (Ferro Electric RAM), PRAM (Phase-change RAM) and OUM (Ovonics Unified Memory) [10] have been proposed. NVRAM is an emerging technique to solve the problem to the slow disk IO. As semiconductor technology makes progress, we can anticipate NVRAM to become a common component of computer systems. Since MRAM among them is comparable with DRAM in terms of capacity, speed and cost, MRAM is considered as the potential replacement of DRAM as the main memory for computer systems. Therefore, we can regard data in memory as persistent and exploit some approaches about NVRAM to improve performance of disk IO.

There are some problems in traditional DRAM-based system if the main memory is NVRAM. Firstly, according to some research [37][38][46], many small files are short-lived. Once the file is created in file system, file system must do some disk IO operations, such as reading metadata of the file. Even many files are deleted soon after created, they also need perform disk IO. Besides, after these files are deleted, it produces some fragmentation in the disk. Secondly, the traditional DRAM-based write-back policy is file-by-file since it can reduce the number of non-up-to-date files

when power outages. But there are two disadvantages: first, the dirty pages per file maybe distributed far in the disk. It must spend many seek time and rotation delay on writing back these dirty pages. Second, if it writes back all dirty pages of a file, system can not make sure whether it does not write back recently-updated pages. According to time locality, recently-updated pages may be re-accessed and re-modified recently. Therefore, if it writes back recently-updated pages, it wastes the disk IO operations. Lastly, the file system consistency has been an important issue recently. Many file systems use the technique to support data consistency such as journaling, but it needs extra journaling IO to write logs into the disk earlier. Therefore, on the basis of three problems, we propose three mechanisms corresponding three problems to improve performance of the file system.

1.2 Our Three Mechanisms

In this thesis, we propose the Buffer Cache management and transaction support on file system operations in NVRAM systems. We have two mechanisms temporary-file file system (TempFFS) and intelligent write back policy (WB) in Buffer Cache management and one mechanism for transaction support on file system operations (trans) for maintaining file system consistency.

Firstly, we add TempFFS between VFS and file system to apply delayed allocation simultaneously on all existing file systems. Different from file system specific implementations that maintain newly-created files on their own, such as XFS[47], and Ext4, TempFFS maintains newly-created files for all the file systems. Upon memory pressure or sync operations, the files are transferred to their original file systems and block allocation of these files takes place. Therefore, an existing file system can enjoy the benefit of delayed allocation without any code modifications.

Secondly, due to the data in NVRAM is persistent; we modify original write back policy which is file-by-file and does not consider recency. We consider the location of dirty pages in the disk and write back contiguous or neighbor dirty pages to reduce the seek time and rotation delay of the disk. Besides, we consider recency that we do not write back the recently-updated dirty pages.

Lastly, our transaction support mechanism can ensure both file system consistency without inducing any extra disk I/O. Since the data in NVRAM is persistent, it needs not write journaling IO before. We only make sure the file operation is atomic. In order to achieve atomic, we duplicate all data and metadata before they are modified in the file operation into undo logs. Once the file operation has finished successfully, the undo logs can be removed immediately. If the crash happens in the progress of the file operation, the undo log can be used to restore to the consistent state. Since our undo logs are placed in NVRAM and deleted later, it needs not any extra disk I/O.

We implement our three mechanisms in Linux 2.6.12. Since large capacity MRAM is not generally available in the market, and the performance characteristics of DRAM and MRAM are comparable and shown in Table 1.1, we use DRAM to emulate MRAM.

Table 1.1 MRAM and DRAM Characteristic

Device Type		**MRAM**	**DRAM**
Characteristic	**Volatility**	No	Yes
	Erase Needed	No	No
Performance	**Access Time**	50ns	~5ns
	Read Time	50ns	50ns
	Write Time	50ns	50ns

Operation	Power Supply	1.8V	1.8-5V

According to our experimental results, the performance improvement of our TempFFS is about 35% compared to Ext2, the performance improvement of our intelligent write-back is about 65% compared to Ext2 and the performance improvement of our transaction support is about 80% compared to Ext3. Lastly, the performance improvement of the combinations of three proposed mechanisms is about 90% compared to Ext3.

1.3 Structure of the Thesis

The remainder of this thesis is organized as follows. Chapter 2 describes the related work about NVRAM. Chapter 3 presents the design and implementation details of the proposed mechanisms. The performance results are shown in Chapter 4. Finally, we give conclusions in Chapter 5.

ES A 1896

Chapter 2 Related Work

In this chapter, we introduce some researches about NVRAM. In Section 2.1, we introduce some researches exploiting NVRAM to recover system when system crashes. In Section 2.2, some researches use NVRAM as storage device to improve the performance of file system. In Section 2.3, some researches use NVRAM as buffer to reduce disk IO, especially write operation. In Section 2.4, we introduce researches about providing file system consistency.

2.1 System Recovery

Ren Ohmura [33] in Keio University exploits the characteristic of NVRAM in system recovery. They propose a scheme to recovery the state of peripheral devices in NVRAM systems so that the system can resume its execution after an unpredictable power failure. They record all messages between CPU and devices in NVRAM and system re-sends messages recorded in memory to recovery devices into previous state when power failure.

Harp [25] records all updates of files in server nodes. Files in individual node can survive after the failure because file operations are logged in memory at several nodes. The Recovery Box [2] stores the state of system in NVRAM and protects the region of storing the system state to not overwrite when system crashes. After the system crashes, it uses the protected system state to recover system.

Rio [8] enables the data in memory to survive operating system crashes and power outages. It uses write protections to protect files in file cache and does not accidentally overwrite the file cache while system is crashing. Therefore, the files in Rio are persistent and safe when system crashes.

2.2 NVRAM as Storage Device

Because the flash is cheap and has the characteristic of non-volatile, the more and more file systems which are designed for flash memory are proposed, such as JFFS2 [44] and Microsoft Flash [24] and so on. However, the flash memory has some limits: firstly, flash must erase the block before writing it. Secondly, the block in flash has finite number of erase-write cycles. Therefore, the flash system usually writes data by using non-in-place update and it makes the number of erase-write cycles in each block are similar by using wear leveling technique.

In order to speed up the writing in flash system, eNVy [45] uses a small amount of battery-backed SRAM as write buffer and uses copy-on-write technique to copy corresponding data in flash into SRAM, then modifies data in SRAM. Lastly, it writes back data into flash memory when the amount of SRAM is full. Hwan Doh [11] also exploits non-volatile memory to enhance the performance of flash file system. They propose a flash-based file system that stores all metadata in NVRAM and stores all file data in flash memory. The advantages of using NRAM as a metadata store are the mount time of flash is reduce to the minimum and access all metadata is speeder than before.

MRAMFS [12] is a prototype in-memory file system to put all data/metadata in NVRAM. However, the amount of NVRAM may be not enough containing of a large number of files. Therefore, they use the compression method to reduce occupied space and use the different compression method to compress metadata and data because metadata often has the fixed format. In metadata they can save about 60% space and save about 40%~60% space for file data.

There are some recent works in Hybrid Disk/NVRAM file system such as

HeRMES file system [31] and Conquest file system [42]. HeRMES considers that the metadata is frequently modified in the file system requests. Therefore, they suggest that use of compression techniques in order to minimize the amount of memory required for metadata and place all metadata in NVRAM to improve the performance of file system requests. Conquest assumes that the system is in the sufficient amount of NVRAM. Therefore, it stores all small files and metadata in NVRAM and disk holds only the data content of remaining large files. The advantages are that it can avoid the overhead of accessing small file and metadata because metadata and small files are placed in NVRAM and it can optimize the arrangements of large files to reduce the fragmentation in disk because there are only large files in disk.

The above works have some disadvantages. Firstly, they almost place all metadata in NVRAM but the occupied space of metadata/data is constantly increasing as users create files at all times. Secondly, although the metadata is frequently accessed in file system, it is not that all metadata are frequently accessed. Therefore, they place all metadata in NVRAM such that there is some non-recently-used metadata occupied the NVRAM space resulting in performance decreases.

2.3 NVRAM as Buffer

In addition to storage device, the general purpose of NVRAM is as the write buffer. eNVy [45] mentioned in Section 2.2 uses a small amount of battery-backed SRAM as write buffer to improve the performance of write operations in flash. Mark Baker [1] proposes that if they provide a NVRAM as write buffer, it can reduce disk access by about 20% on most of file systems, and by about 90% on one frequently-accessed file system.

Theodore R. Haining [19] mentions that the use of non-volatile write caches

provides two benefits: some writes will be avoided because dirty blocks will be overwritten in the cache, and physically contiguous dirty blocks can be grouped into a single I/O operation. They also present some write back strategies, such as least recently used (LRU), shortest access time first (STF) and largest segment per track (LST) to manage non-volatile write buffer and find that write buffer can reduce a large number of write requests to improve the performance of system.

Robert Y. Hou [20] exploits non-volatile memory to improve the performance of RAID5. In each write request, RAID5 needs to execute "read-modify-writes" which means that single-block writes require the old data block and old parity block to be read, modify them to generate the new parity block, and then the new data and new parity can be written to their respective locations. Read-modify-writes can reduce the performance of RAID5 arrays because it needs four disk accesses in each write request. Therefore, they use non-volatile memory as the write buffer of RAID5 to improve the performance of write operations.

Above researches are also about using write buffer to improve write operations, Alex Batsakis [3] mentions read operations may depend upon write operations because buffering dirty pages will occupy the memory for read caching. They address this problem by separately allocating memory between write buffering and read caching and by writing dirty pages to disk opportunistically before the operation system submits them for write-back. They also write back dirty pages which are almost adjacent, but they do not consider whether the dirty pages are not recently-updated.

Due to the capacity of MRAM is increasing continuously, it maybe replace DRAM as the main memory of computing system in the future. We not only use the technique of non-volatile write buffer to delay write, but also use the better write-back policy to

improve the performance of file operations.

2.4 Transaction Supporting

Traditionally, file system consistency has been maintained by using synchronous writes to restrict the proper ordering of metadata updates, but this approach degrades the performance of file system because the proceeding of metadata updates is dominated by the disk speed. Soft updates [30] eliminates the need for synchronous disk I/O. Soft updates is an implementation mechanism that enforces the dependencies of metadata updates and allows the metadata caching for write back.

Log-structured file system [39] proposed by Mendel Rosenblum treats the file system as a segmented log and always writes all modified data blocks and metadata into the end of the log. File system changes are buffered in the cache and then written into the disk sequentially in single disk IO operation. Therefore, it can improve the performance of write operation but it can not write all related metadata in single write operation since if crashes happen in the progress of disk operation, the file system remains an inconsistent state.

Journaling [35][44][47] is nowadays a widely-used technique for file system consistency. It logs metadata and data updates into a stable storage before the updates are performed on the disk. Hence, it produces the extra journaling IO traffic that is critical impact on the system performance.

Kevin M. Greenan [17] introduces two approaches to reliably storing file system structures in NVRAM. Firstly, they strengthen memory consistency by using page-level write protection and error correcting codes. Secondly, it periodically calls online consistency checker to replay all transaction logs for checking file system inconsistency. If it finds the inconsistency in file system, it immediately recovers the

state of file system. However, it needs to periodically replay all transaction logs even if the file system is normal and does not have any failures.

Henry Mashburn [40] proposes recoverable virtual memory (RVM) that is simple user-lever library to handle atomic file operation and data persistence. Firstly, it copies the range of memory which will be updated to the undo log in memory, then updates data, and lastly writes the updated data to the redo log in disk. Therefore, it needs three copy operations for each file operation.

Vista [27] proposed by David Loweel is simple user-library runs on Rio mentioned in Section 4.1. Because Rio protects the files in memory to be persistent, Vista can eliminate the redo log to speed up disk operations and it only uses undo log to make sure the file operation is atomic. However, it must be based on Rio and because it is user-level library, Vista is not user-transparent.

We propose a simple lightweight transaction support on file system operations in NVRAM environment and it only needs to add only about 40 line-codes in kernel and about 300 line-codes in implementation. It also provides the same strength of consistency as the journaling mode of Ext3.

Chapter 3 Design and Implementation

In this chapter, we describe the design and implementation of the proposed mechanisms. In Section 3.1, we first introduce the three mechanisms for improving the performance and ensuring the consistency of file systems on NVRAM based computer systems, namely Temporary-File File System (TempFFS), intelligent write-back policy, and transaction support on file system operations. In Section 3.2, we show the details of implementing and integrating the mechanisms and provide an analysis on the integration of the mechanisms.

3.1 Background

In this section, we describe the proposed NVRAM-based buffer cache management mechanisms, which include Temporary-File File System and intelligent write-back policy. Both mechanisms aim at improving the file system performance based on the non-volatility feature of main memory. Moreover, we also describe a lightweight transaction support mechanism on file system operations, which takes advantage of the non-volatility feature of main memory for ensuring the consistency and data integrity of the file system.

3.1.1 Temporary-File File System (TempFFS)

The first goal of TempFFS is to reduce the fragmentation of the underlying file systems. With numerous and concurrent file creation/deletion/appending activities, a file system is easy to become fragmented, which leads to performance degradation. Moreover, according to the previous studies [37][38][46], many files are short-lived, meaning that they are deleted soon after their creation. Allocating disk space for these files, which involves disk IO operations for reading the file system metadata (e.g. block allocation map), is unnecessary.

To reduce the file system fragmentation and the unnecessary disk IO operations, some advanced file systems such as XFS [47]and ext4 support delayed allocation, which delays the disk block allocation of a newly-created file until the data is needed to be flushed back to the disk due to memory pressure or *sync* operations. However, the delayed allocation feature is not shared among all file systems. Only the file systems that implement the feature can benefit from it.

Instead of integrating the delayed allocation feature into a specific file system, we implement a RAM-based file system named TempFFS in order to apply the feature simultaneously on existing file systems such as ext3 and NTFS. Based on the concept of stackable file systems, TempFFS sits between VFS (virtual files system) and file system implementations and is transparent to the latter, as shown in Figure 3.1. All new files are initially written to TempFFS and associated with their original file systems when they are created. TempFFS uses page cache as the file store, and the files are transferred into their corresponding file systems upon memory pressure or *sync* operations. In this way, existing file systems can benefit from delayed allocation without code modifications. Note that a file can stay for a long time in TempFFS. This raises the risk of data loss if the main memory is volatile. On systems with non-volatile main memory, however, memory data can survive power failures. The implementation of TempFFS was achieved by modify the code of an existing RAM file system (i.e., the RamFS [34]) for ease of implementation.

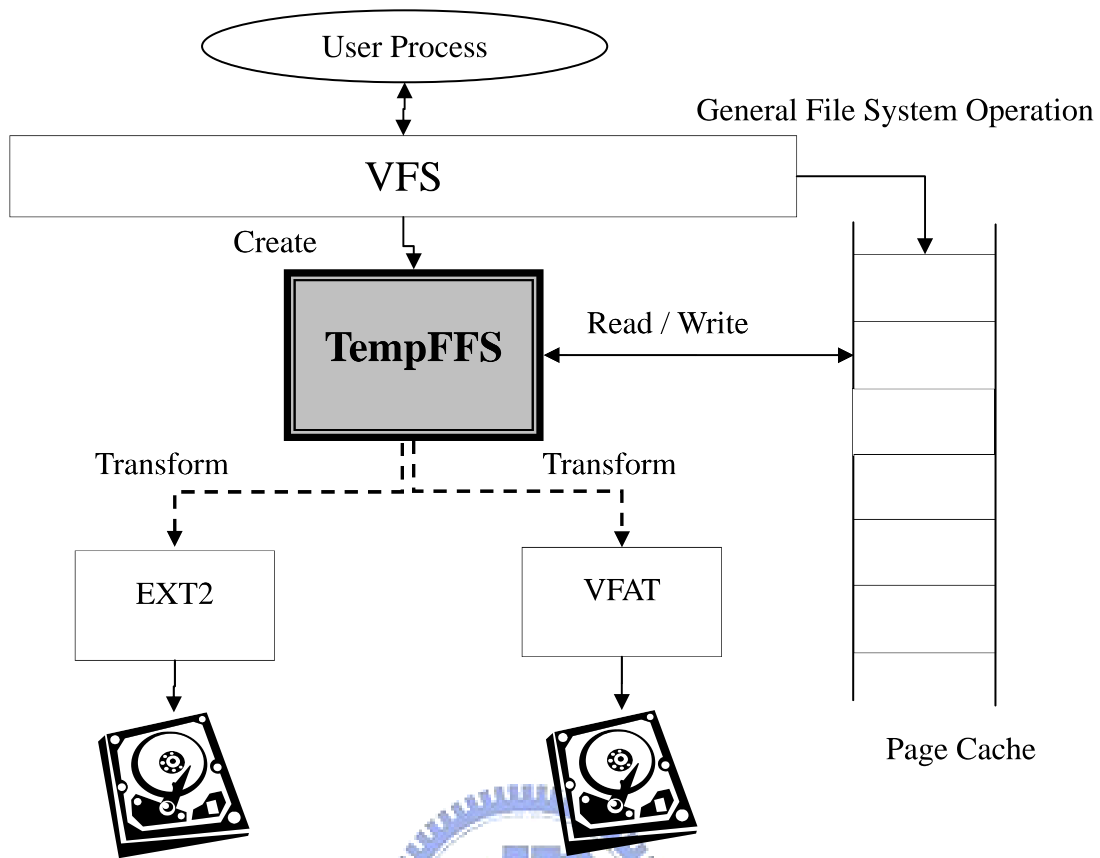


Figure 3.1 Architecture of the Temporary-File File System

TempFFS stores files in kernel memory, which cannot be paged out in traditional UNIX operating systems (including Linux). Upon memory pressure, an OS usually writes back the dirty pages that belong to the buffer cache or user processes to the storage device so as to release more memory space. In this situation, TempFFS checks if its size is larger than a specific threshold. If it is, TempFFS shrinks its size by evicting pages of the least recently used files. All the evicted files are transformed into their original file systems so that the corresponding data can be written back. In addition, we transform files whose sizes are larger than a specific threshold (currently, 1MB) due to the following two reasons. First, according to previous research [37][38][46], most short-lived files are small ones, if it puts short-lived files in TempFFS, it can reduce some IO traffics. Second, creating a huge file may cause the transform of a large number of short-lived small files before they are deleted,

reducing the benefit of delay allocation.

We manage the files in TempFFS in a LRU list. The number of pages that should be evicted from TempFFS, say *N*, is proportional to the number of pages in TempFFS. Specifically, N is calculated according to the following equation:

$$N = NR_WB * NR_TempFFS / NR_Dirty,$$

where *NR_WB* represents the target number of pages that need to be written back, *NR_TempFFS* represents the number of (dirty) pages in TempFFS, and *NR_Dirty* represent the number of dirty pages in the system. As shown in Figure 3.2, transforming a file involves the following three steps.

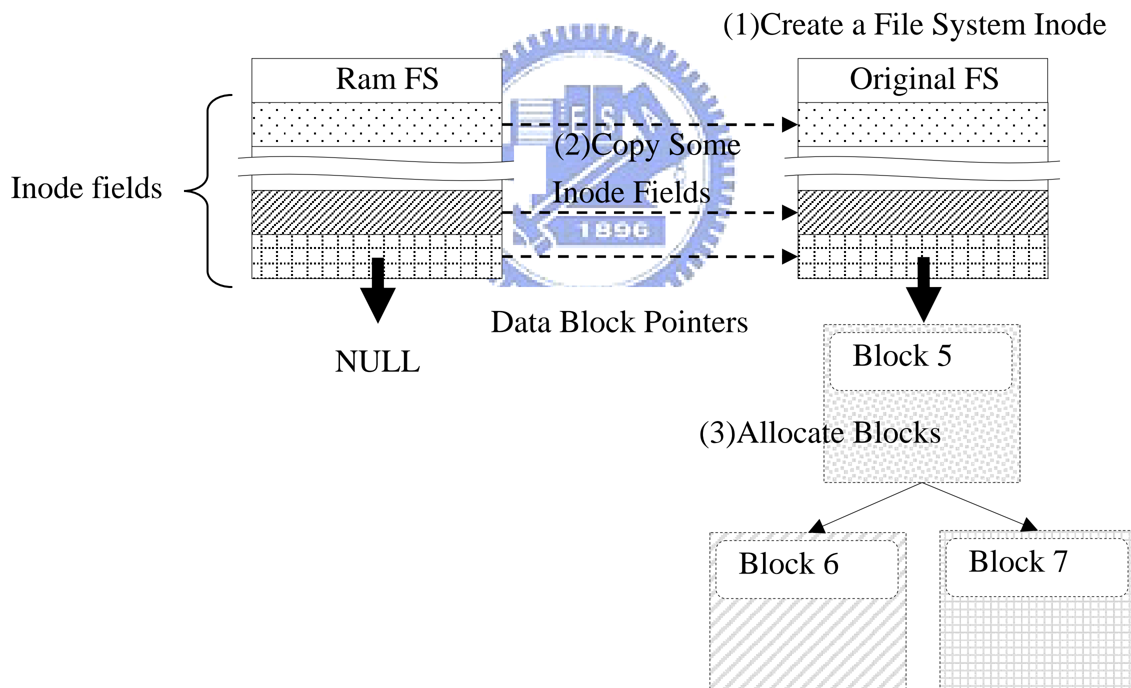


Figure 3.2 Transformation Steps

First, the file create operation of the original file system is invoked to produce the metadata (inode) of the file. Second, several inode fields such as timing information, access rights and file size, are copied to the new inode. Third, a sequence of disk block allocation operations of the original file system are invoked for allocating the

disk space for the file. Because the operations are invoked consecutively, the resulting data blocks tend to be contiguous. After the allocation, the data is associated with the allocated blocks and the metadata in the TempFFS is deleted.

3.1.2 Intelligent Write-Back Policy

Modern operating systems write back dirty pages periodically or when the number of free pages is below a specific threshold (i.e., memory pressure). On systems with non-volatile main memory, dirty pages are already persistent and thus need not to be written back into the disk periodically. Instead, they need to be written back only under memory pressure or sync operations. Currently, Linux utilizes a file-by-file write back policy, which scans the list of dirty inodes and submits the dirty pages of each inode to the IO subsystem. The rationale behind this policy is to reduce the numbers of non-up-to-date files when power outages or system crashes. Assume that 100 files are updated and each file has 10 dirty pages in memory. If the system crashes after 500 dirty pages are written back to disk, it would be better to write all the dirty pages of 50 files than write 5 dirty pages of all the files.

However, this policy may write back recently-updated pages, which has two drawbacks. First, writing back such pages can not help to release the situation of memory pressure since these pages will not be reclaimed by the page replacement policy. One purpose of writing back dirty pages is to reclaim the page so as to maintain a reasonable number of free pages in the system. In Linux, all pages belonging to user processes and page cache are grouped into two lists, the active list and the inactive list. The former includes pages which have been accessed recently while the latter contains pages that have not been accessed for a period of time. The file-by-file policy may write back dirty pages in the active list. However, most LRU-like page replacement policies tend not to reclaim these pages since the pages

are used recently. Second, according to time locality, these pages will be marked dirty soon after their write back. Thus, writing back such pages is of little use. The pages may need to be written back again soon. Some UNIX systems like Solaris do not have such problem. They only write back dirty pages that are not used recently.

The common problem of the write back policies of the existing UNIX operating systems (including Linux) is that they ignore the disk location of the dirty pages when submitting the pages to their IO subsystems. Although an IO subsystem can sort the requests submitted to it, there may still a significant amount of seek and rotation delay among the dirty pages.

In this paper, we propose an intelligent write-back policy, which considers the recency as well as the disk locations of the dirty blocks to reduce the IO traffic, seek time and rotation delay. To reduce the IO traffic, the proposed policy *recency* only writes back dirty pages in the inactive list.

To reduce the seek time, we divide a disk into a number of zones, which is a set of continuous blocks on the disk, and write back dirty pages in a zone-by-zone manner. The dirty page information is recorded in a set of identical data structures called *zone information tables*, each of which correspond to a zone. When a page becomes dirty and inactive, we record the page in the corresponding zone information table according to the disk block number of the page.

Each time the write-back procedure is invoked, the proposed policy selects a zone and writes back dirty pages in that zone. This reduces the seek time because the disk blocks of the written-back dirty pages are close. In order to further reducing the rotation delay, the policy selects a zone with the maximum Average Segment Length (ASL), which is defined in Equation 1. A segment stands for a set of continuous dirty

blocks in a zone, and there is generally no rotation delay between two continuous blocks. Therefore, this policy tries to select a zone which contains more continuous dirty pages to reduce both the seek time and the rotation delay of the IO traffic caused by dirty page write back.

Average Segment Length (ASL) = Number of Dirty Pages in the Zone/Number of Segments in the Zone ____Equation 1

The zone information table, which is shown in Figure 3.3, it records some information such as, dirty pages numbers, segment numbers, segment list which contains of all segment in the zone, page list which includes all dirty pages of the inactive list in the zone, and length (Average Segment Length).

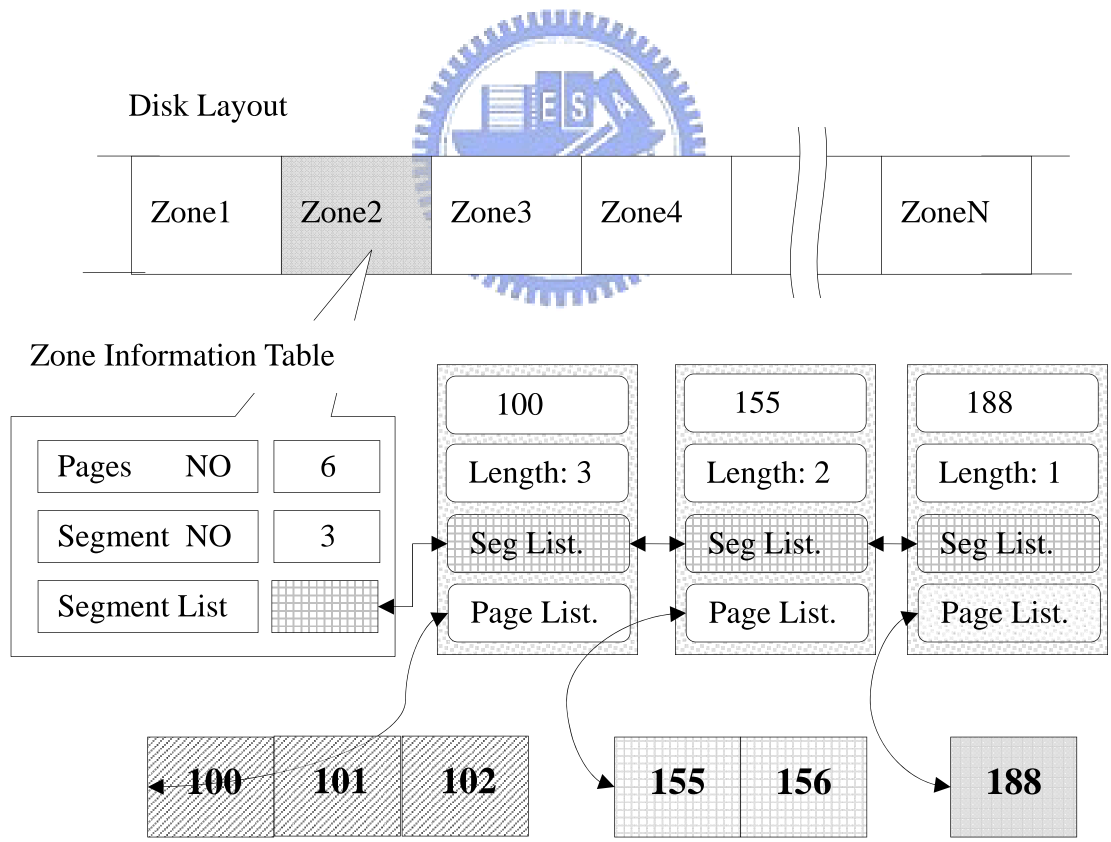


Figure 3.3 the Zone Information Table

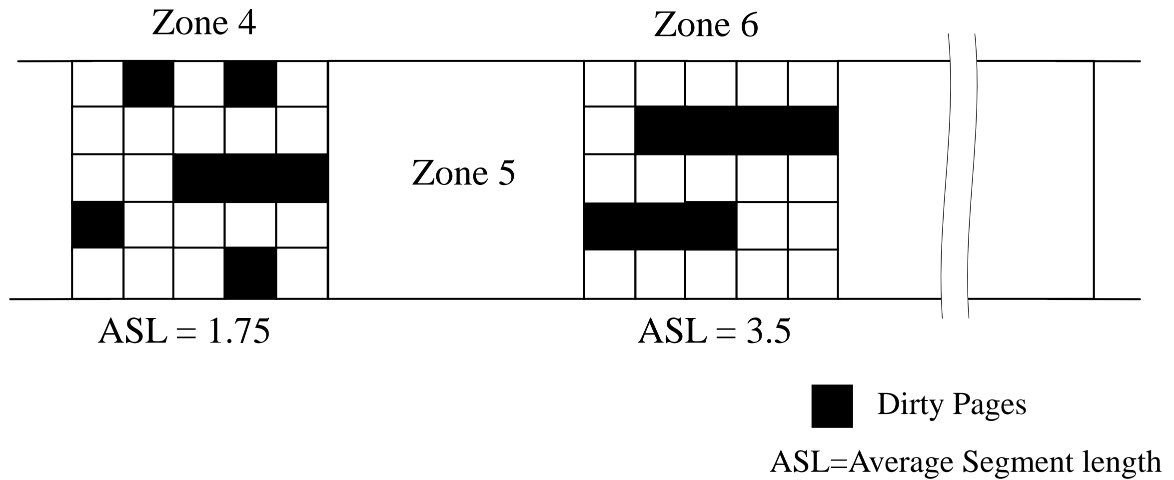


Figure 3.4 Zone with Different Average Segment Length

Figure 3.4 shows an example of the zone selection. The dirty pages of zone 4 and zone 6 are both 7, but the dirty pages of zone 6 are more continuous (i.e., with a larger value of ASL) than zone 4. Therefore, zone 6 is selected to be written back.

As mentioned before, this policy only writes back pages in the inactive list in order to reduce the write back IO traffic. Therefore, only the dirty pages in the inactive list are recorded in the zone information tables. To accomplish this, we need to insert or remove the information about a dirty page when it becomes inactive or active. Specifically, when a dirty page becomes inactive (i.e., moves from the active list to the inactive list), we record it in the corresponding zone information table. When the page becomes active again or clean, the recorded information is removed. This allows us to write back only inactive dirty pages.

3.1.3 Transaction Support on File System Operations

Journaling is a widely-used technique for guaranteeing file system consistently. It logs metadata and data updates that are completed in memory into a stable storage

before the updates are performed on the disk. When the system crashes, the log is replayed to restore the status of the file system. Therefore, journaling ensures file system consistency by using a redo log. The overhead of journaling is that it requires additional disk I/O operations for logging. On non-volatile memory based computing systems, one straightforward approach for eliminating such I/O operations is to place the log in the memory instead of disk. However, the drawback of simply placing logs in memory is that it occupies a large memory space. For example, it typically needs 256MB memory space as journaling space. Additionally, to minimize the IO overhead, a journaling file system usually writes the updates to the log in batches with size about 16 to 64 MB. A significant amount of updates might be lost if the system fails before the updates are written to the log.

To address the problems mentioned above, we design a lightweight transaction mechanism on systems with non-volatile main memory. The mechanism not only ensures file system consistently but also eliminates the need of large memory space and extra disk I/O.

The basic idea of the mechanism is undo log. Because the data in NVRAM does not lose, we need not write journaling logs into the disk before. We only need make sure that each file operation is atomic. To ensure the atomicity of each file operation, we duplicate the data and metadata in the undo log before they are modified. Once a file operation has finished successfully, the duplicated data and metadata can be removed immediately. If the system crashes, the content in the undo log (i.e., the original values of the metadata and data of the uncompleted operations) is used to recover the file system state. The main data structure of our lightweight transaction mechanism is shown in Figure 3.5.

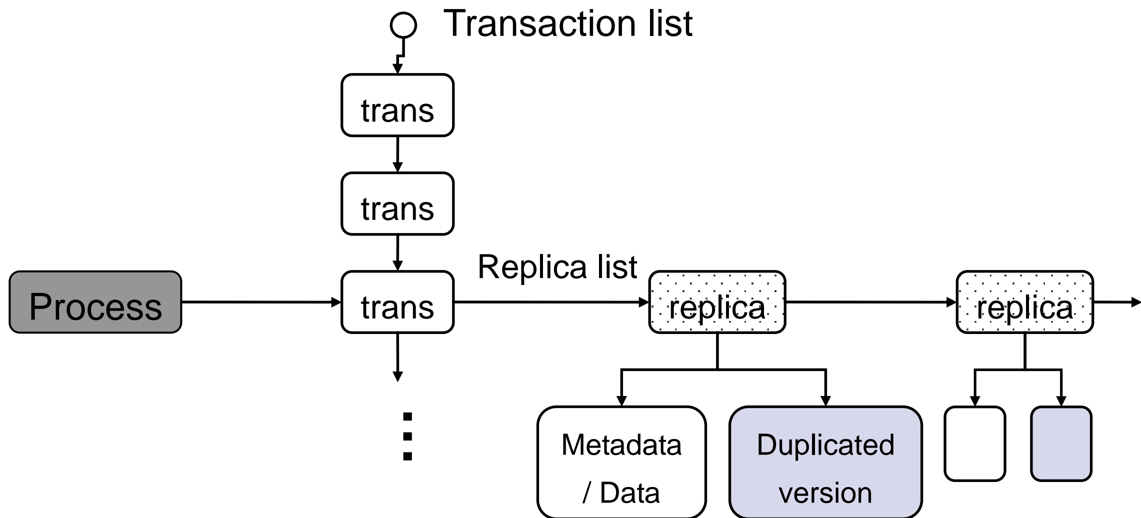


Figure 3.5 Main Data Structure of Transaction Support

In order to make sure the atomicity of the updates involved in a file operation, all metadata and data modified by the file operation are collected into a transaction and recorded in a data structure called *trans*. When metadata or data is going to be modified by a file operation, the original content is copied to a memory area pointed by a data structure called *replica*, which is then attached to *trans*. The *replica* also records the addresses of the metadata/data that are under update. This allows the metadata/data to be recovered by the original content if necessary.

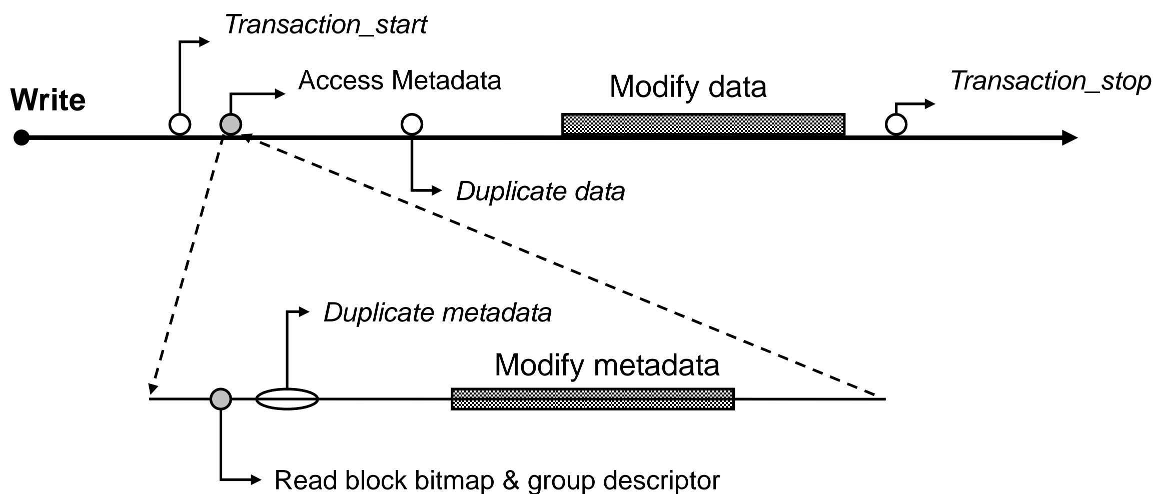


Figure 3.6 Core Functions of Transaction Support

We implemented the transaction support in a file system independent manner and provide a transaction API by which file systems can leverage to achieve file operation atomicity. Figure 3.6 shows an example usage of the transaction API. Before each file operation, the file system firstly invokes the *transaction_start()* function, which initializes the **trans** data structure for the current process and inserts the **trans** data structure into the global transaction list. Then, the file system calls *duplicate()* which duplicates the data and metadata before they are going to be updated. It initializes a **replica** data structure and copies the original values of the metadata/data into a temporally-allocated area, then maps the area to its **replica**. The **replica** data structure will be inserted into the replica list of the corresponding **trans**. Lastly, after the file operation, the file system calls *transaction_stop*() which terminates the transaction. It removes all duplicated data and metadata of corresponding this **trans** without affecting data integrity and file system consistency because all metadata/data in the file operation have already finished upgrading. Therefore, the transaction support mechanism can have the same strength as the journal mode of ext3 because it duplicates data and metadata before they are updated. Moreover, it causes little overhead in file system because it deletes all replicas of data and metadata once file operations finished and does not cause any additional disk I/O.

3.2 Implementation and Integration of the Approaches

In this section, we describe the detailed implementation and integration of the three approaches.

3.2.1 Implementation of Three Mechanisms

As mentioned before, we implement TempFFS by modifying an existing RAM file system called Ram-FS (Resizable simple ram File System), and then insert it between VFS and file system implementations. We intercept the invocation of the VFS file create function (i.e., *vfs_create()*) and direct the invocation to the file create function in RamFS. After the creation, operations on the file will use the file operations in RamFS because the file now is placed in RamFS not in file system, such as Ext2.

Upon memory pressure or the size of file is over the threshold, we transform files in Ram-FS into the file system. The detail steps of transform are shown in Section 3.1.1. After transforming, the file is belong to the file system, we only use the original file operations in file system to access it.

To implement the intelligent write-back policy *relo* (recency and location), we record the information of a page in a zone information table, which is shown in Figure 3.3. When the page becomes dirty and inactive, we record this dirty page into the corresponding zone information table. To achieve this, we invoke a function *add_to_zone()* in two situation. First, when it calls the function that marks the page dirty (i.e., *set_page_dirty()*), we check the active flag (PG_active) of the page. If this dirty page is in the inactive list (i.e., PG_active flag is not set), we invoke a function *add_to_zone()* for *set_page_dirty()* function. Second, when it calls the function that moves the page form the active list into the inactive list (i.e., *add_page_to_inactive_list()*), we check whether the page is dirty or not. If this page is dirty, we invoke a function *add_to_zone()* for *add_page_to_inactive_list()* function.

The pseudo code of the *add_to_zone()* function is shown in Figure 3.7(a). First, we get the block number of the page, and calculate the zone corresponding to this page (i.e., block number of page divides block number per zone). Second, numbers of dirty pages in zone information table increases by one. If the former block and latter block of this page do not record in zone information table, it means that this page stands alone. If this page is recorded in the zone information table, it produces a new segment (contiguous dirty pages). Therefore, segment numbers of zone information table increases by one. Lastly, we calculate the ASL (average segment length) as a basis of selecting the zone to write back.

When the dirty page is clean or active, we also need to remove the information of the page from the zone information table. To achieve this, we invoke a function *remove_from_zone()* in two situation. First, when it calls the function that clears dirty of the page (i.e., *clear_page_dirty_for_io()*), we invoke a function *remove_from_zone()* for each call of the *clear_page_dirty_for_io()* function. Second, when it calls the function that moves the page form the inactive list into the active list (i.e., *add_page_to_active_list()*), we invoke a function *remove_from_zone()* for each call of the *add_page_to_active_list()* function. The pseudo code of the *remove_from_zone()* function is shown in Figure 3.7(b). First, we also get the block number of this page to calculate the corresponding zone. Second, the dirty page numbers decreases by one. If the former block and latter block of this page do not record in zone information table, when it removes this page, it reduces a segment. Therefore, segment numbers of zone information table decreases by one. If the former block and latter block of this page both record in zone information table, when it removes this page, the original segment divide into two segments. Therefore, segment numbers of zone information table increases by one. Lastly, it recalculates the ASL of this zone.

In Linux, when the system writes back the data in memory into the disk, the system wakes up the Pdflush thread to call *background_writeout()*. In *background_writeout()*, we change the original function (*writeback_inodes()*) into *writeback_segment_zone*() which selects a zone to write back. It is shown in Figure 3.7(c). First, we select the zone with largest average segment length. Second, we traverse all page lists recorded in zone information table to write back all pages. Lastly, if the number of written-back pages is greater than or equal to the number of demand for write-back pages, it finishes. If not, it selects the next zone to write back.

```
/* Adding a page to the zone info. table */
add_to_zone( page ){
    get page's block number;
    zone_number = page_block_number / pages_per_zone;
    zone_information_table[zone number].dirty_pages++;
    if (a new segment is created for this page)
        zone.segment++;
    ASL = zone. dirty_pages / zone.segment
}
```

(a)

```
/* removing a page from the zone info. table */
remove_from_zone( page ){
    get page's block number;
    zone_number = page_block_number / pages_per_zone;
    zone_information_table[zone number]. dirty_pages --;
    if (a segment is deleted due to the removal of the page)
        zone.segment--;
    if (a new segment is produced due to the removal of the page)
        zone.segment++;

    ASL = zone. dirty_pages / zone.segment
}
```

(b)

```
/* Segment-zone writeback algorithm*/
writeback_segment_zone(writeback_control wbc){
begin :  select the zone with largest average segment length;
    traverse the all segment's page list of the zone to writeback all pages;
    if (number of pages written back >= wbc.nr_to_write)
        finish;
    else
        writeback_segment_zone( wbc );
    goto begin;
}
```

(c)

Figure 3.7 Pseudo Code of Segment/Zone Algorithm

As mentioned in Section 3.1.3, we provide a transaction API for file systems the require transaction support. Figure 3.8 shows the pseudo code of the major function implementations, *transaction_start(), transaction_stop()* and *duplicate(),* in the API. T*ransaction_start()* firstly creates a transaction data structure *trans,* links this *trans* into current process. Lastly, it inserts this *trans* into the global transaction list. *Transaction_stop()* firstly gets a transaction data structure *trans* from current process, clears the pointer of current process that points to this *tans*. Lastly, it frees all duplicated data (*replica*) of this *trans*, and removes this *trans* form global transaction list. *Duplicate()* firstly also gets a transaction data structure *trans* from current process, creates a *replica* data structure to store the duplicated data, and inserts this *replica* into corresponding *trans*. Lastly, it duplicates the data into this *replica*.

To demonstrate the effectiveness of the API, we augmented the ext2 file system to leverage the API. We inserted the function pair *transaction_start()* and *transaction_stop()* in all ext2 file system operations such as *ext2_create()*, *ext2_link()*, *ext2_mkdir()*, *ext2_unlink()*, *ext2_rmdir(),* etc. Moreover, we inserted the invocation of *duplicate()* in functions that modify metadata and data such as *ext2_new_inode()*, *ext2_free_inode()*, *ext2_new_block()* and *ext2_free_blocks(),* etc. We ensure the invocation of the *duplicate()* function is right before the modification of metadata or data.

```
/* Creating a transaction and inserting it to the transaction list */
transaction_start( ){
        create a transaction data structure;
        link this trans into current process;
        insert this trans into transaction list;
}
```

(a)

```
/* removing a transaction from the transaction list */
transaction_stop( ){
        get a trans from current->journal_info; // journalling filesystem info
        set current->journal_info as NULL;
        free all replicas in this trans;
        free this trans from transaction list;
  }
```

(b)

```
/* duplicate metadata or data*/
duplicate(buffer_head *bh, size_t size){
      get a trans from current process;
      create a replica data structure;
      insert this replica into replica list of trans;
      copy data from buffer_head *bh to this replica;
}
```

(c)

Figure 3.8 Implementation of the Transaction API (Pseudo Code)

3.2.2 Integration of Three Mechanisms

Three mechanisms can operate independently, and also can combine the corresponding both of three mechanisms, even can integrate three mechanisms together. We integrate the proposed three approaches in Linux 2.6.12. The overall structure of three approaches is shown in Figure 3.9.

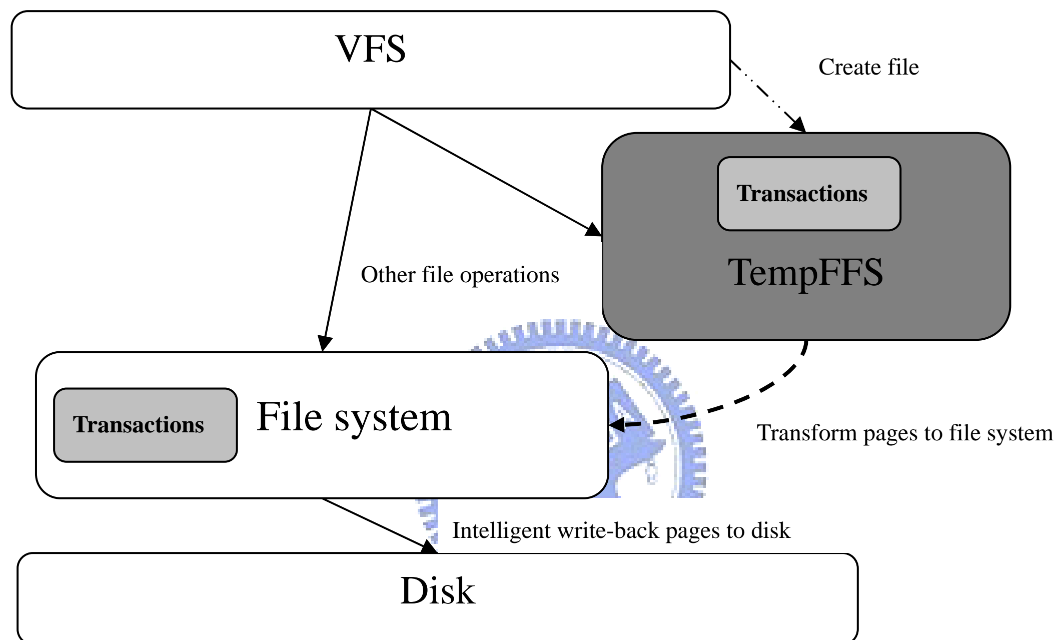


Figure 3.9Integration of the TempFFS, Intelligent Write Back and File Operation Transaction Support

We create all files in TempFFS, the other file operations, such as read, write, and delete use the file operations of Ramfs if this file is placed in TempFFS and use file operations of original file system if this file is transformed into the file system. When it needs flush the dirty pages into the disk, it uses intelligent write back policy (*relo*). Moreover, we inserted the invocation of the transaction API into both ext2 file system and TempFFS to maintain their consistency.

The performance results of different combinations of the three approaches are shown in Chapter 4.

Chapter 4 Performance Evaluation

In this chapter, we evaluate the performance of the three proposed mechanisms. Section 4.1 describes the experimental environment and all the configurations under performance comparison. Section 4.2 presents the performance improvements of TempFFS. In Section 4.3, we compare the performance of various write-back policies mentioned in Section3.1.2. Section 4.4 shows the performance and memory overhead of the lightweight transaction support mechanism on file system operations. Finally, we present the performance results of all combinations of the three proposed mechanisms in Section 4.5.

4.1 Experimental Environment and Configurations

Table 4.1 shows the experimental environment. Since large capacity MRAM is not generally available in the market, and the performance characteristics of DRAM and MRAM are comparable, we use DRAM to emulate MRAM. We evaluate the performance of the proposed mechanisms under two popular benchmarks.

Bonnie++ [5] is a micro-benchmark that measures the performance of single file access. Three kinds of tests in Bonnie++ are performed, *character_write*, *block_write*, and *rewrite*. The *character write* test writes a 2GByte file sequentially in a character-by-character manner. The *block write* test writes a 2GByte file in a (several bytes per block) block-based way. The *rewrite* test reads the existed block of file and modifies it, then writes file by block-based. Postmark [22] is a macro-benchmark that emulates the access pattern of an email server. It creates many files whose size between Max. Size and Min. Size which are defined by users, and then operates the assigned number of transactions which may be create/delete or read/append, lastly deletes all files. In this experiment, we run 200k transactions, with the numbers of

files from 5k to 30k and the file size ranging form 512 bytes to 10 Kbytes. For the other parameters, we use the default settings of Postmark.

Moreover, we also measure the performance under the execution of real application such as untarring and compiling Linux kernel. We untar a package that contains the source code and object files of Linux 2.6.12, and then compiling the kernel.

Table 4.1 Evaluation Environment

Hardware	**CPU**	AMD Athlon 64 3000+
	Memory	1 GB DDR 400
	Disk	Maxtor 80G 7200 RPM
Software	**OS**	Linux 2.6.12
	Workloads	Bonnie++ 1.03a, Untar, Make, Postmark 1.5

Table 4.2 shows all the experimental configurations under performance comparison. In the first two configurations, the original ext2 and ext3 file systems are used. For ext3, we use the Journal mode since it is the only one that provides data integrity. The mext3 configuration is the same as ext3 except that it improves the performance of ext3 by placing the logs in a ramdisk residing on MRAM. The last seven configurations represent various ways of combinations of the three proposed mechanisms.

Table 4.2 Experimental Configurations

Configurations	**Description**
Ext2	Ext2 file system
Ext3	Journal mode of ext3 file system
Mext3	Ext3 with logs on MRAM
Ext2_Trans	Lightweight transaction support on file system operations
TempFFS	Temporary-File file system
WB	Intelligent write-back policy
TempFFS_Ext2_Trans	Temporary-File file system + transaction support
WB_Ext2_Trans	Intelligent write-back + transaction support
TempFFS_WB	Temporary-File file system + intelligent write-back
TempFFS_WB_Ext2_Trans	Temporary-File file system + intelligent write-back + transaction support

4.2 The Performance Results of TempFFS

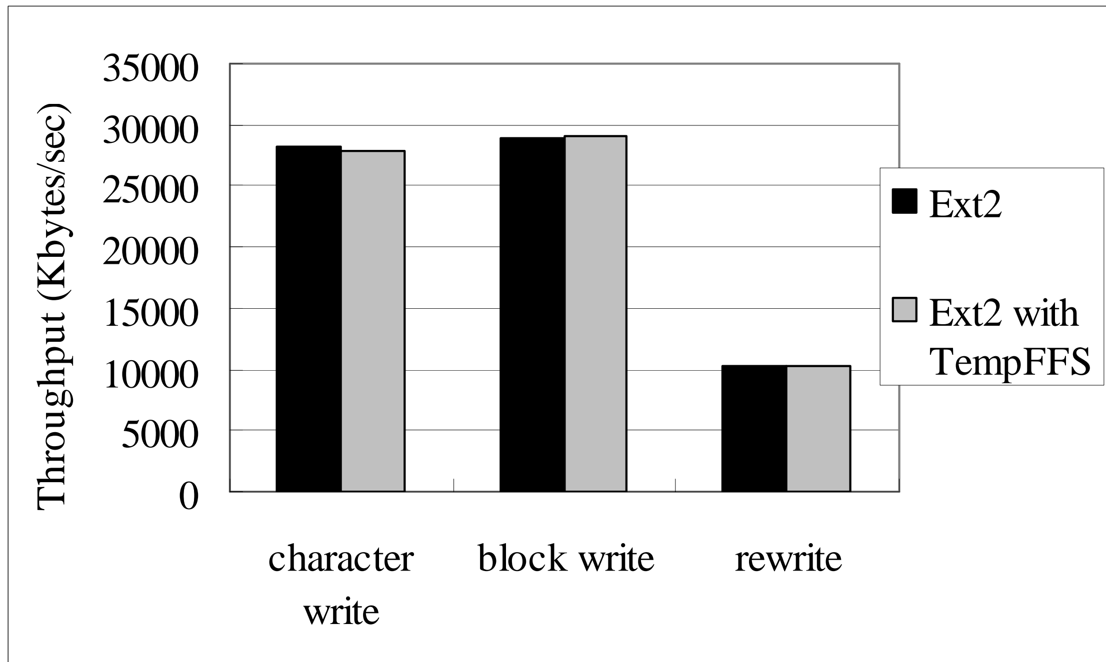


Figure 4.1 Performance Improvements of TempFFS (Bonnie++)

In this section, we present the performance improvements achieved from TempFFS by comparing the performance of the ext2 file system with and without TempFFS under different workloads. In the first experiment, we compare the performance under Bonnie++. The values of the parameters, including the file size and the block size, are the same as those in Section 4.1.

Figure 4.1 shows the results. From the figure, we can see that TempFFS does not result in noticeable performance improvements. This is mainly because the size limitation of a file in TempFFS. As mentioned in Section 3.1.1, a file in TempFFS is transformed to it original file system if its size exceeds the size limitation (currently, 1Mbytes). Thus, the 2GByte file is transformed to ext2 soon after its creation and therefore gets little benefit from TempFFS.

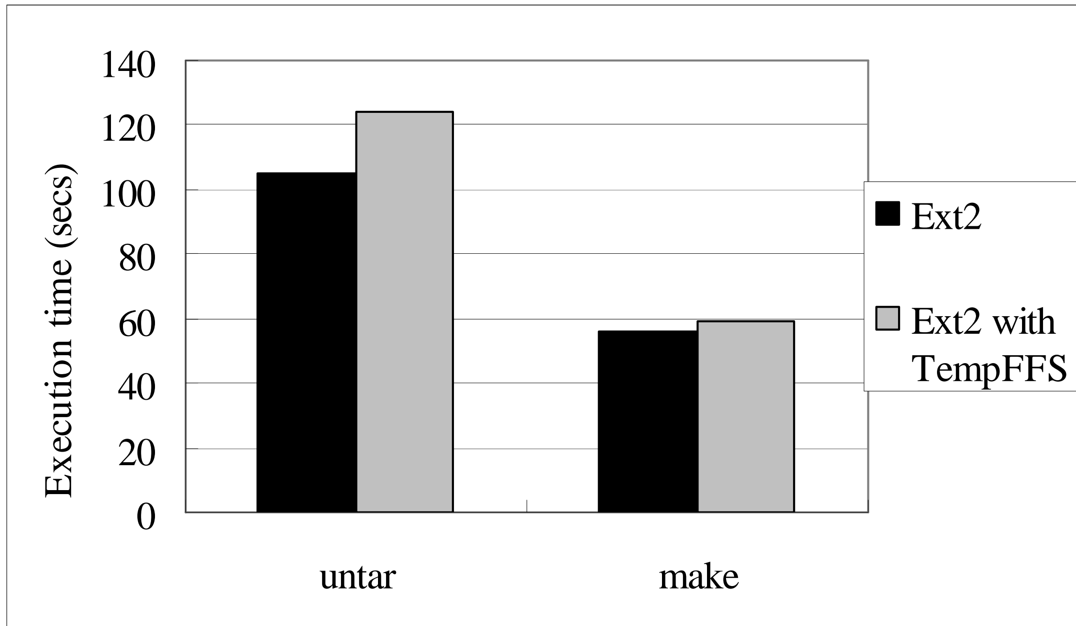


Figure 4.2 Performance Improvements of TempFFS (Untar and Compile Linux Kernel)

In the second experiment, we measure the performance of TempFFS under Linux kernel untarring and compilation. Figure 4.2 shows the results. From the figure, we can see that TempFFS degrades the performance of ext2 by 18% under the untar workload. Although untaring Linux kernel produces a significant number of small files, they are never be deleted. Therefore, the files are just first placed in TempFFS, and then transformed to their original file systems. No IO traffic can be saved. Moreover, the file creation does not result in a large degree of fragmentation, and thus TempFFS can seldom help in this workload. Instead, it degrades the performance due to the file transformation overhead.

For the *make* workload, the presence of TempFFS does not have a noticeable impact on the performance of ext2. This is because the workload is CPU-bound. Therefore, *make* needs not produce I/O operations to create object files because our package has contained of object files.

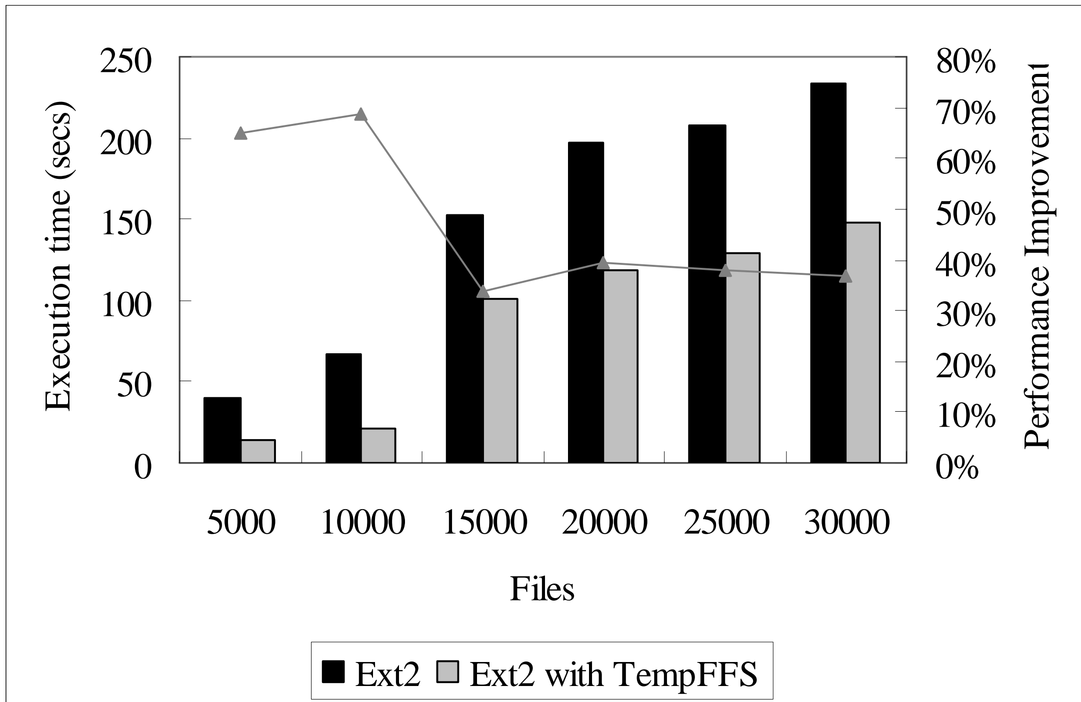


Figure 4.3 Performance Improvements of TempFFS (Postmark)

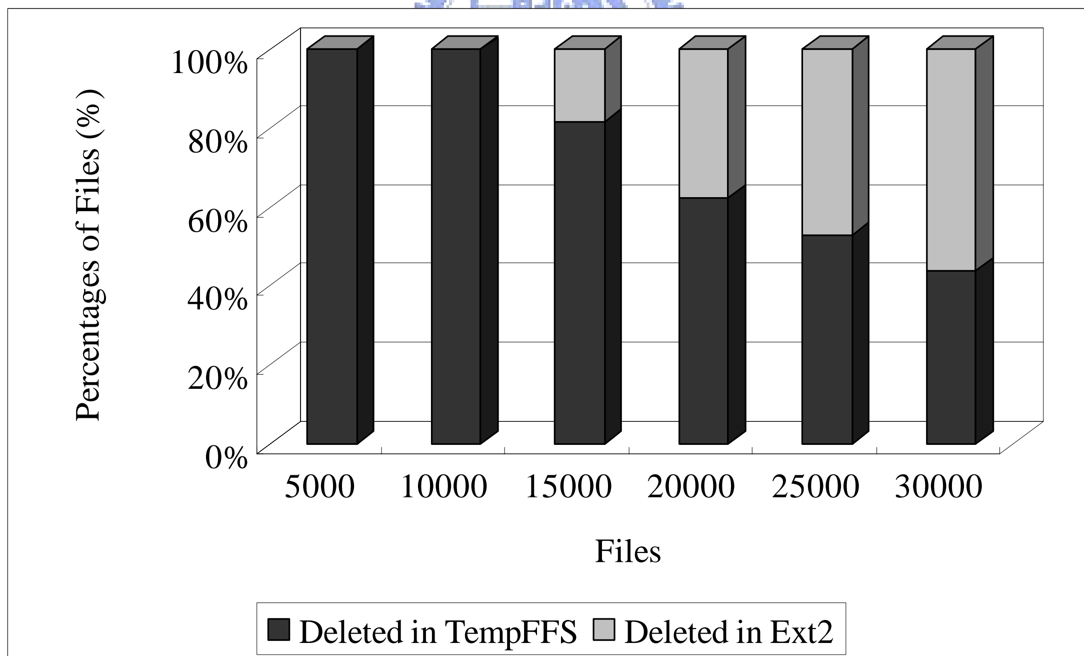


Figure 4.4 Percentages of Files Deleted in TempFFS (Postmark)

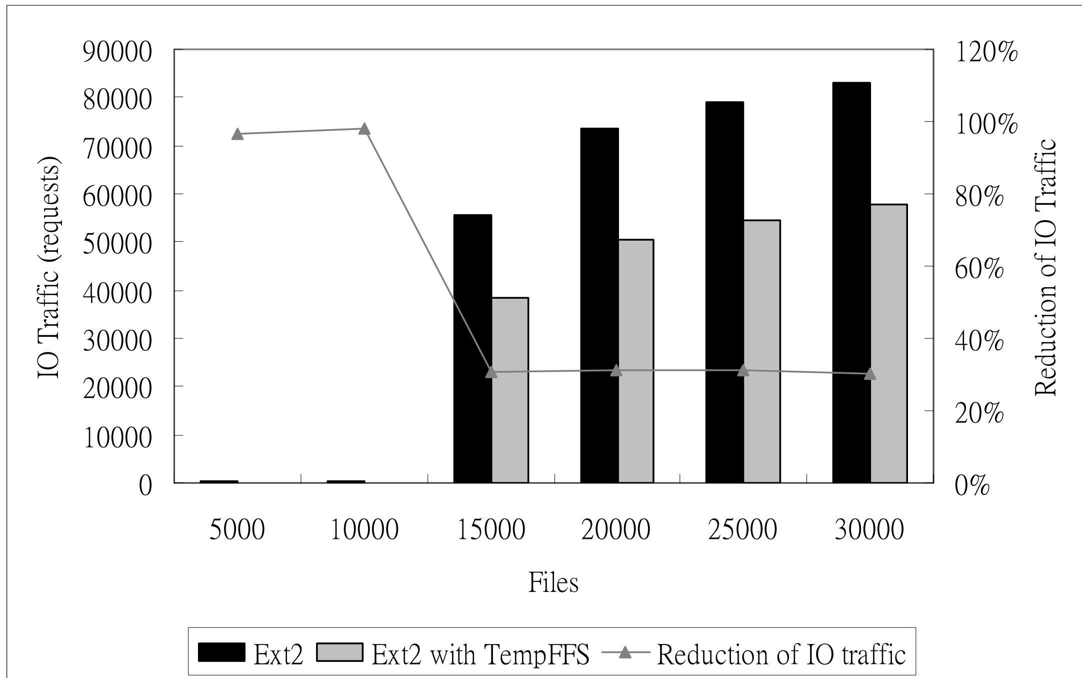


Figure 4.5 Reduction of IO Traffic with TempFFS

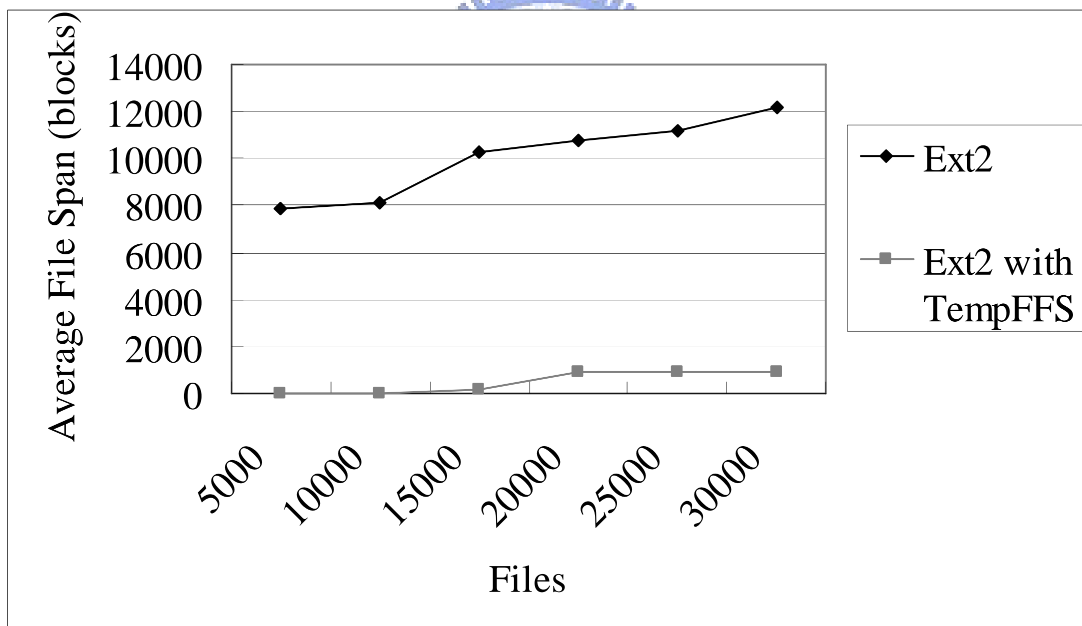


Figure 4.6 Average File Span (Postmark)

In this experiment, we measure the performance improvements of TempFFS under Postmark. In this experiment, 200k transactions were performed and the file size ranges from 512 bytes to 10 Kbytes. We measured the performance under various numbers of files and directories. As shown in Figure 4.3, TempFFS effectively

improves the system performance. Specifically, the performance improvement ranges from 34% to 69%. This is because a number of files have been deleted before they are transformed to the file system, reducing both the I/O traffic and the degree of file fragmentation. We demonstrate this in the following experiments.

As mentioned before, Postmark deletes all the files at the end of its execution. Figure 4.4 shows the percentage of the number of files deleted in TempFFS and ext2, with the presence of TempFFS. As shown in the figure, at least 44% of the files are deleted in TempFFS. In the cases of 5000 and 10000 files, all files are deleted in TempFFS because the capacity of TempFFS is enough to contain all the files. When the number of files increases further, a number of files are transformed to ext2, because of memory pressure, and finally deleted in ext2. For each file deleted in TempFFS, all its file operations are done in memory and involve no disk IO. Figure 4.5 shows the reduction of IO traffic with the presence of TempFFS. In the cases of 5000 and 10000 files, nearly 100% of the IO traffic can be eliminated since almost all file operations are done in TempFFS. For the other cases, about 31% of the IO traffic can be eliminated. Moreover, we show that TempFFS can reduce the degree of file fragmentation, which is evaluated by using the *file span*, the distance between the first block and the last block of a file. During the execution of Postmark, we record the file span of each file upon the deletion of the file. Figure 4.6 shows the average file span of all the files. As shown in the figure, the degree of file fragmentation is largely reduced. Especially, in the cases of 5000 and 10000 files, the average file span is zero because all the files are deleted in TempFFS and do not have corresponding blocks on the disk.

4.3 The Performance Results of Intelligent Write-Back Policy

In this section, we compare the performance of the four write-back policies, *file, recency, location,* and *relo*. The *file* policy is the file based policy used in Linux. It scans the list of dirty files and writes back the dirty pages of each dirty file. The *recency* policy only writes back dirty pages in the inactive list of the Linux page cache. The *location* policy selects the zone with the maximum ASL value and writes back the dirty pages within the selected zone. Finally, the proposed *relo* policy combines the *location* and the *recency* policies and writes back the inactive dirty pages in the selected zone.

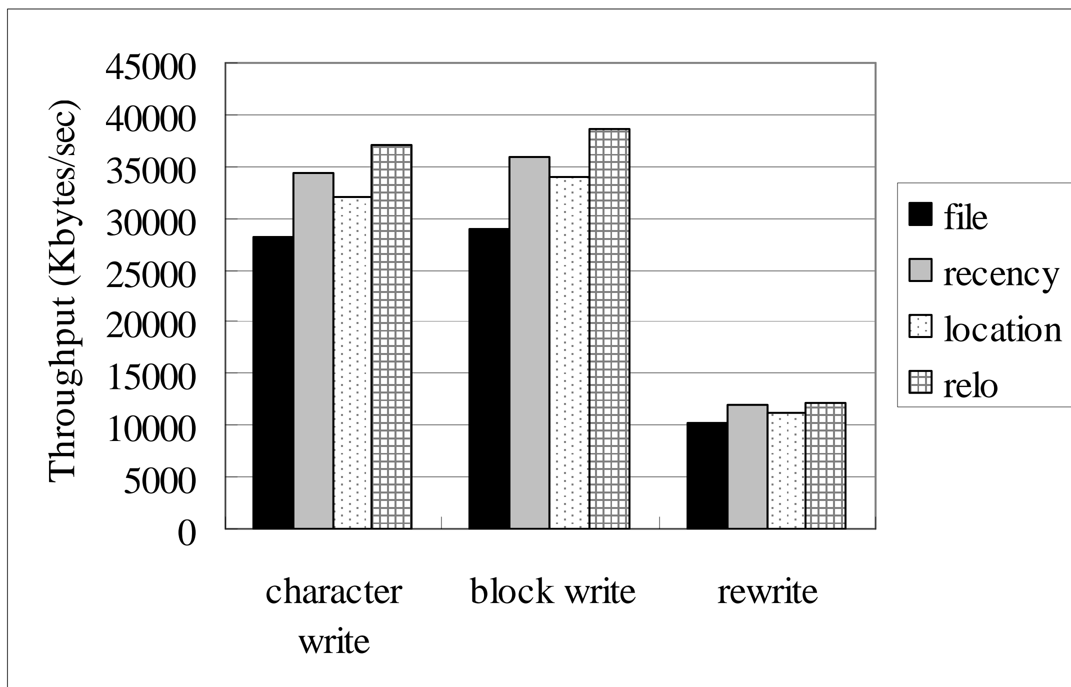


Figure 4.7 Performance Comparison of the Write Back Polices (Bonnie++)

From Figure 4.7 shows the performance comparison of the four policies under the Bonnie++ benchmark. The values of the parameters, including the file size and the block size, are the same as those in Section 4.1. As shown in the figure, the performance results of the latter three policies are all better than those of the first one,

the original write back policy in Linux. As mentioned in Section 3.1.2, this is because the Linux write back policy does not consider the recency and disk location information of the dirty pages, resulting in more redundant page write traffic and longer seek and rotation delay. The proposed *relo* policy considers both the recency and the disk location information and outperforms the original Linux policy by 31% to 34% under the Bonnie++ benchmark.

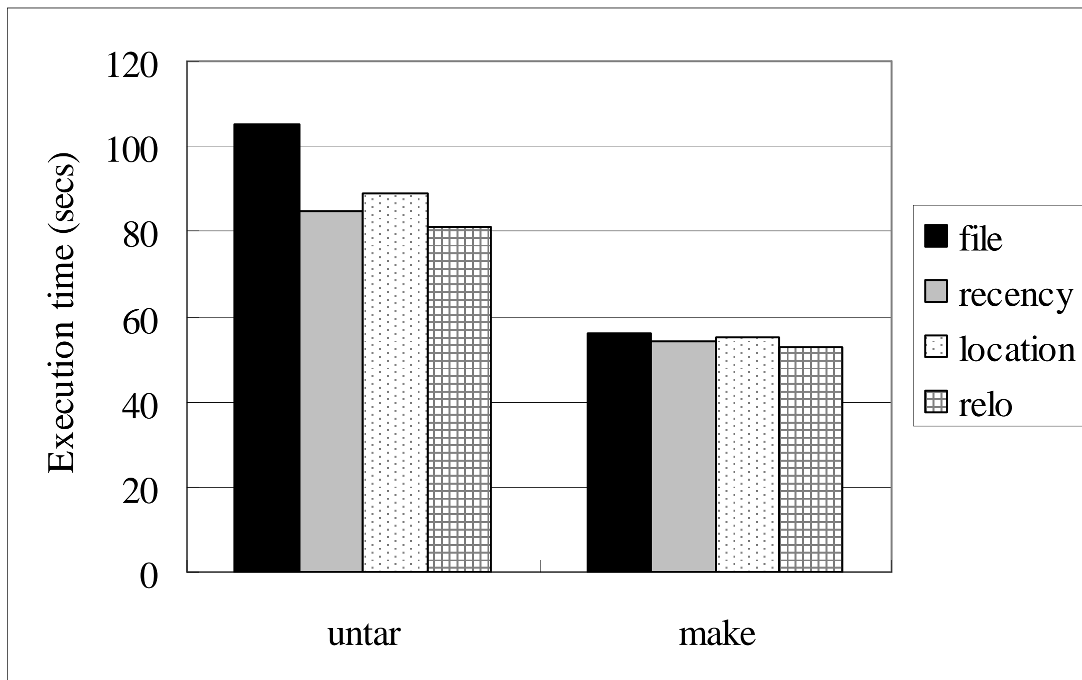


Figure 4.8 Performance Comparison of the Write Back Polices (Untar and Compile Linux Kernel)

Figure 4.8 shows the performance comparison of the four write back policies under Linux kernel untarring and compilation. As shown in figure, all the latter three policies perform better than the original one in Linux, and the proposed *relo* policy achieve the best performance among the four. Specifically, *relo* outperforms the *file* policy by 23% in the untar workload. In the *make* workload, the performance difference is not obvious since the workload is CPU intensive.

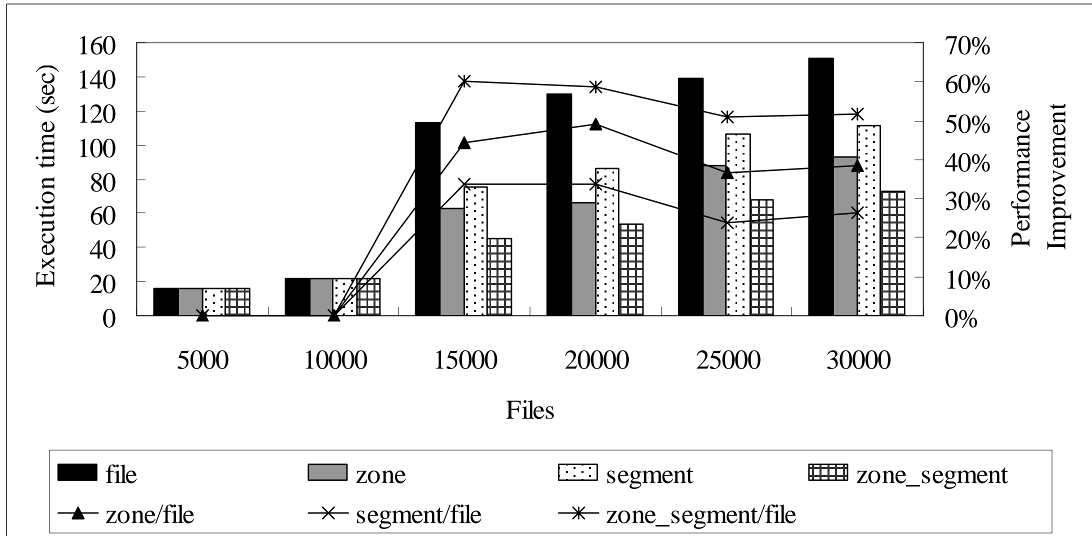


Figure 4.9 Performance Comparison of the Different Locative Write Back Polices

Figure 4.9 shows three different locative write back policies, zone-based which writes back a region of disk and segment-based which writes back a longest contiguous blocks of disk and zone/segment-based which mentioned in Section 3.2. The performance improvement of zone/segment-based has about 58% and is best. The zone-based is better than segment-based because zone-based can save the seek time of disk and segment-based can save the rotation delay.

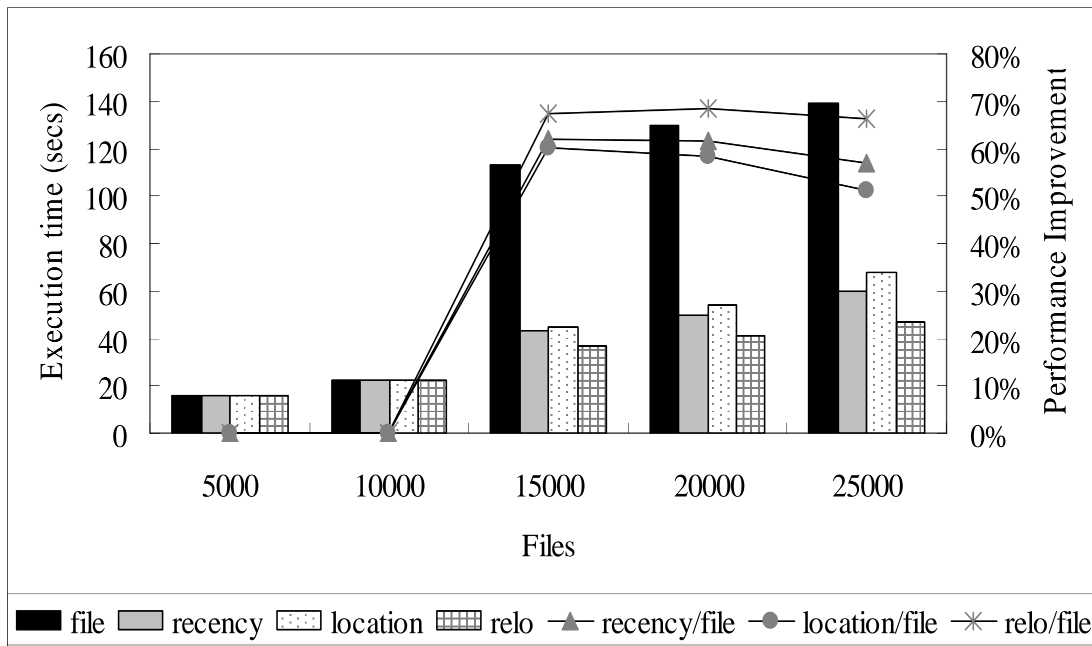


Figure 4.10 Performance Comparison of the Write Back Polices (Postmark)

Figure 4.10 shows the performance comparison of the four write back policies under Postmark. In this experiment, 200k transactions were performed and the file size ranges from 512bytes to 10Kbytes. We measured the performance under various numbers of files. In this figure, the bars denote the execution time of the Postmark and the curves denotes the performance improvements of the latter three policies over the file policy.

As shown in the figure, the performance improvements of the *recency, location, and relo* policies are about 62%, 58% and 67%, respectively, as the file numbers are lager than 15000. In the case of 5000 files and 10000 files, there is no obvious performance difference among the four policies. This is because the working set is smaller than the memory size, and hence the write back procedure is seldom triggered.

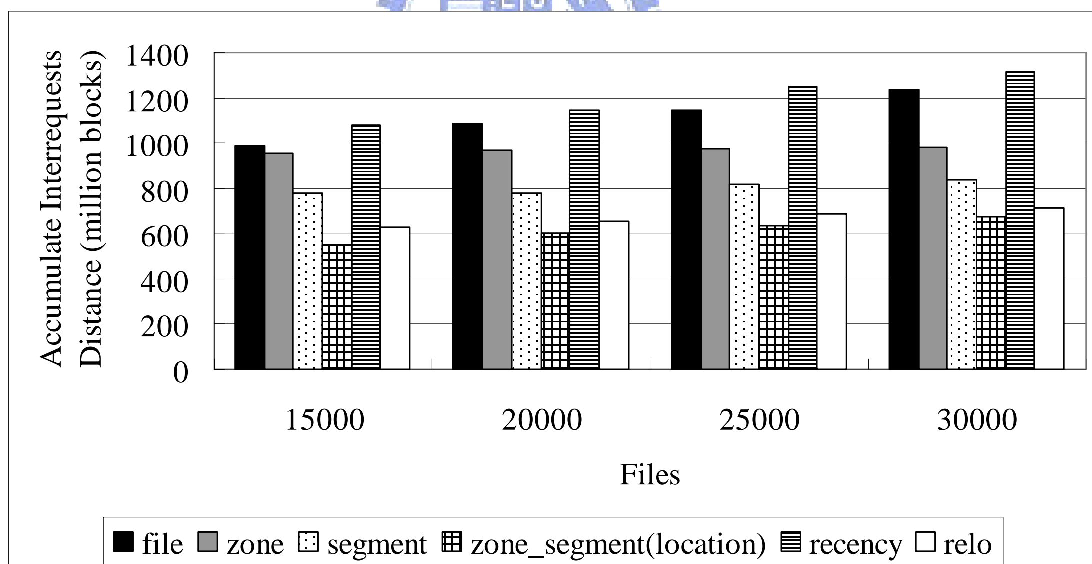


Figure 4.11 Accumulate Inter-requests Distance (Postmark)

Figure 4.11 shows the accumulate inter-requests distance which is the accumulate blocks of all inter-requests. The less of accumulate inter-requests distance means the more contiguous write-back blocks. The zone/segment-based has the least accumulate inter-requests distance because it writes back the contiguous or neighbor blocks.

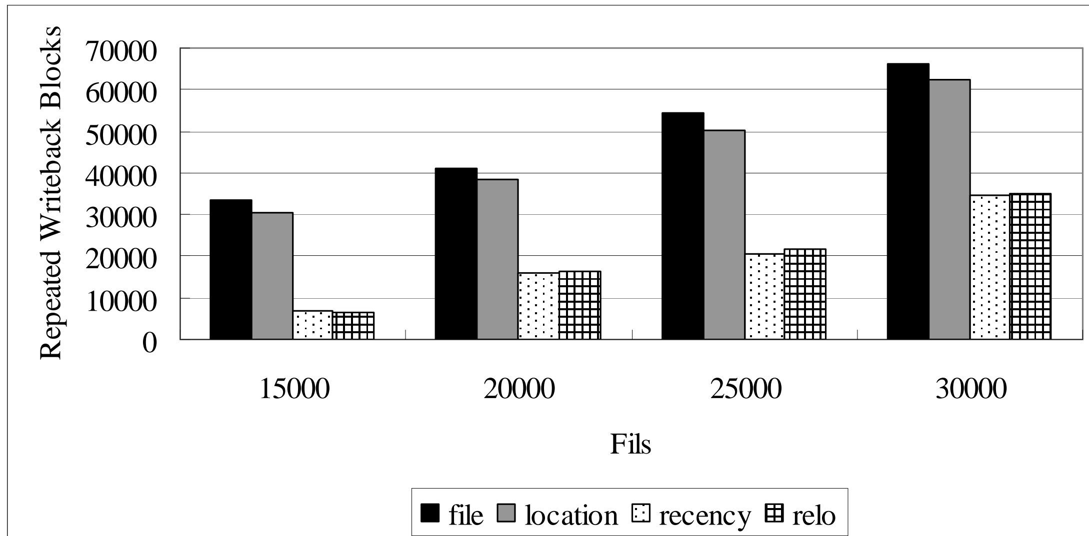


Figure 4.12 Repeated Write Back Blocks (Postmark)

Figure 4.12 shows the total number of repeated write back blocks. In this experiment, the recency and relo has the less repeated write back blocks because they avoid write back the blocks which used recently.

4.4 The Performance Results of Transaction Support on Ext2

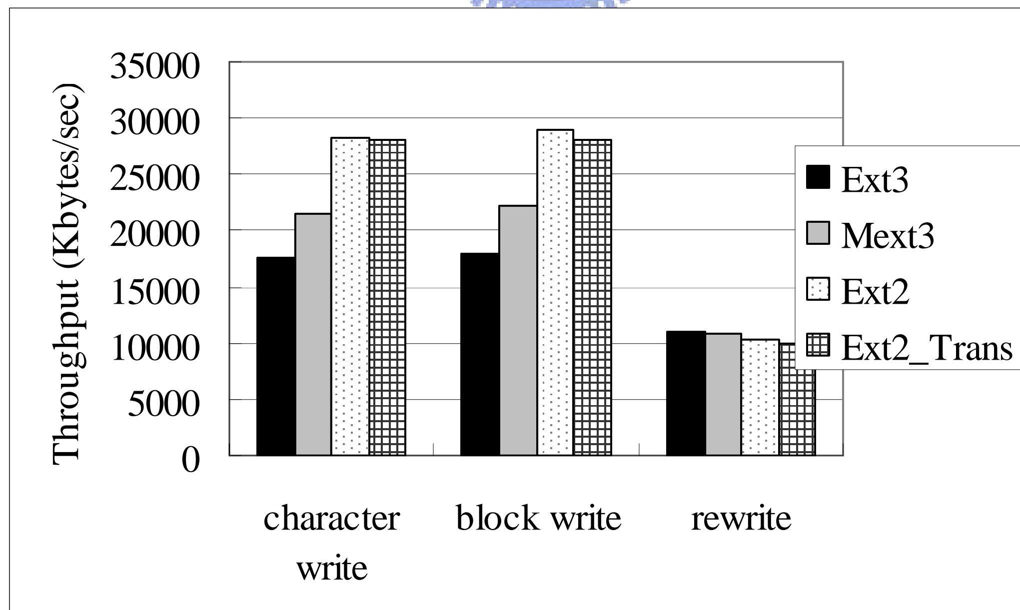


Figure 4.13 Performance of the Transactions Support Mechanism (Bonnie++)

As mentioned in Section 3.1.3, we augmented ext2 to utilize the proposed

transaction API to support atomic file operations. In this Section, we compare the performance of the augmented ext2 with that of ext3. For ext3, we use the journal mode since the augmented ext2 can ensure both file system consistency and data integrity. Moreover, we use two versions of ext3. One places the log in a 256 MB disk partition (ext3), while the other places the log in a 256 MB ramdisk (mext3). Finally, the performance of the original ext2 is also presented for the evaluation of the runtime overhead of the transaction API. Note that ext2 supports neither file system consistency nor data integrity.

Figure 4.13 shows the performance comparison under the Bonnie++ benchmark. As shown in the figure, ext2 with transaction support results in the best performance among the three file systems that ensure file system consistency. This is because it only duplicates data and metadata in memory during the file operations and does not involve any disk I/O. It outperforms the ext3 by 63% and the mext3 by 31%. The performance of ext3 is the worst since the *journal* mode of Ext3 logs both metadata and data updates on the disk, it requires a significant number of extra disk IO. Mext3 eliminates some journaling IO traffic. However, the journaling IO traffic is still required once the journal space is full. Besides, the in-memory journal space of mext3 also occupies the 256MB capacity of the main memory such that the physical memory decreases a lot. The performance results between ext2 and ext2 with transaction support are almost the same because our transaction support does not produce I/O traffics and needs only a small amount of memory space.

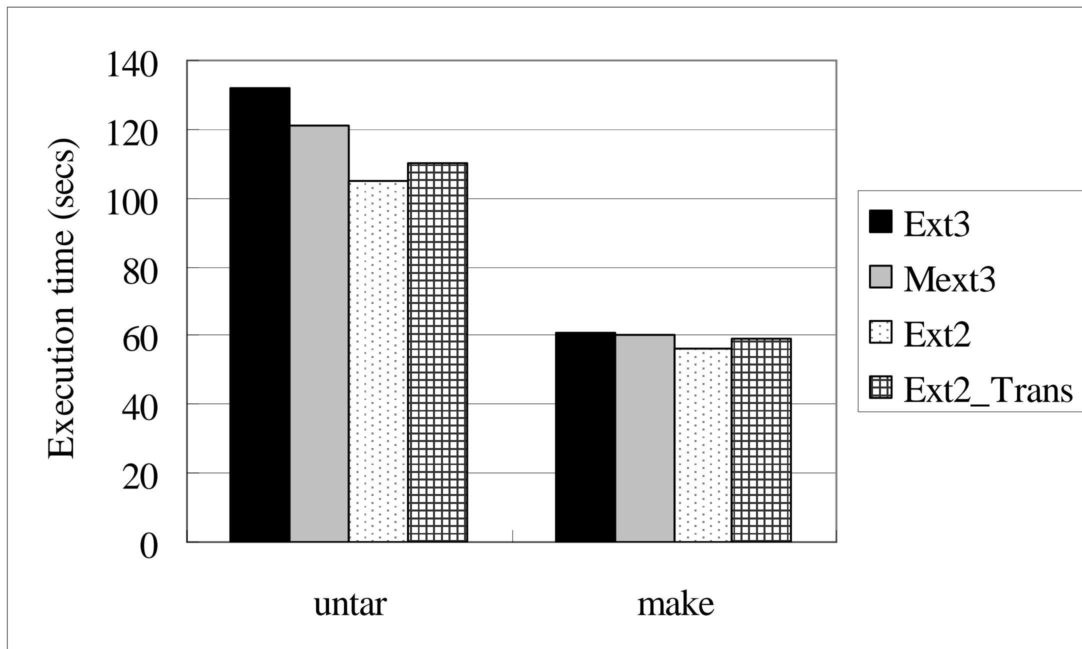


Figure 4.14 Untar and Compile Linux Kernel in transaction support

Figure 4.14 shows the performance comparison under Linux kernel untarring and compilation. Similar to the results in Figure 4.10, ext2 with transaction support always results in the better performance than ext3 and mext3. For the untar case, the performance difference is smaller than that under Bonnie++. This is because untarring the Linux kernel creates many files but does not modify and delete them later, and then it produces journal data less than Bonnie++ or Postmark. Therefore, ext2 with transaction support outperforms the ext3 by 17% and the mext3 by 9%. Besides, the execution time of ext2 transaction is greater than the ext2 a little. For the make case, since *make* application is CPU-bound, the performance results of them are almost the same.

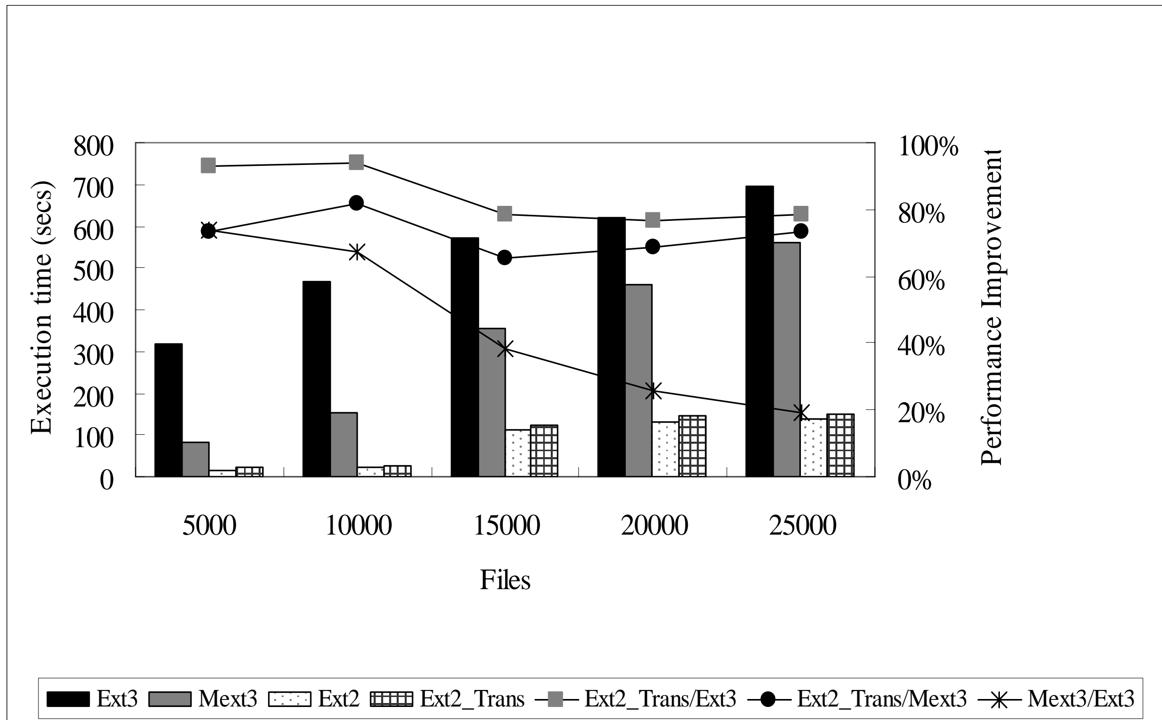


Figure 4.15 Performance of the Transactions Support Mechanism (Postmark)

Figure 4.15 shows the performance comparison under Postmark. As shown in the figure, ext2 with transaction support outperforms ext3 and mext3 by 76~93% and 65~81%, respectively.

Since a large number of transactions (i.e., 20k) were performed during the execution of Postmark, ext3 generates a large volume of journal data and thus a significant number of journaling I/O, largely increasing the total execution time. When file numbers is less, the mext3 has better performance improvement because journal space of mext3 is not often full, and it needs only some writing-back journal data. But the volume of the journal data is increase as the file numbers increase, the journal space of mext3 is often full and it must frequently write back journal data to the disk. Therefore, the performance improvement of mext3 degrades when the file numbers increase.

Instead, ext2 with transaction support just duplicates metadata and data (that are

under update) for not-yet-completed file operations. Once the file operation completes, the duplicated copy is deleted. Thus, it does not generate any journaling IO and its execution time is almost the same as ext2.

Table 4.3 Memory Overhead of Transaction Support with Workloads

Workloads	Maximum Memory Overhead
Untar and Make Linux Kernel	24 KB
Bonnie++	122 KB
Postmark	24 KB

As mentioned before, the transaction support mechanism duplicates metadata and data that are under update. Therefore, it requires some memory space for storing the duplicated copies. In this section, we measure the maximum amount of such extra memory space required during the execution of the workloads used in this paper. We record the maximum amount of total *replicas* in transaction list and update it in runtime.

Table 4.3 shows the results. As shown in the table, the extra memory space required is extremely small, only 24 KB for Linux kernel untarring and compilation, 24 KB for Postmark, and 122 KB for Bonnie++. This is because applications usually do not update a large amount of metadata or data in a single file operation. From the results, we can see that the memory overhead of the transaction support mechanism can nearly be ignored.

4.5 Put it all Together

In the section, we show the performance results of different combinations of the three proposed mechanisms.

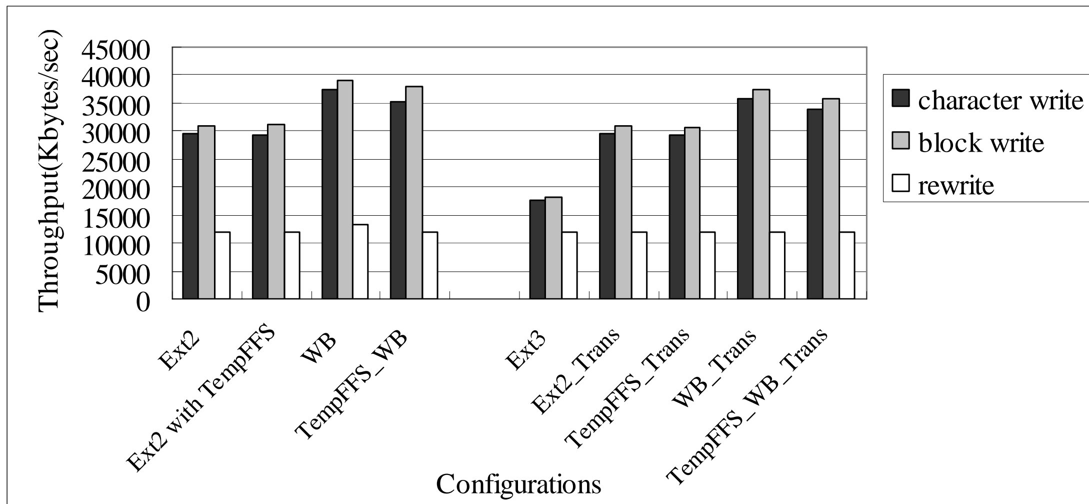


Figure 4.16 Performance Results of Different Combinations (Bonnie++)

Figure 4.16 shows the performance results under Bonnie++. In this figure, the results of each test contain six bars. The left two bars represent throughput under file systems that do not ensure file system consistency, whereas the rest represent throughput under file systems that ensure file system consistency.

Third points in the figure are worth mentioning. First, both the relo write back policy and the lightweight transaction support mechanism lead to performance improvements. Second, incorporating TempFFS into the system causes performance degradation. This is demonstrated by the values of the rightmost two bars of each test. Comparing the values of the left two bars and the values in both Figure 4.1 and 4.4 also lead to a similar result. This is because, as we mentioned in Section 4.2, TempFFS does not enhance performance under Bonnie++. Third, by comparing the values of the first and the fourth bars, we can see that the runtime overhead of transaction support can nearly be ignored.

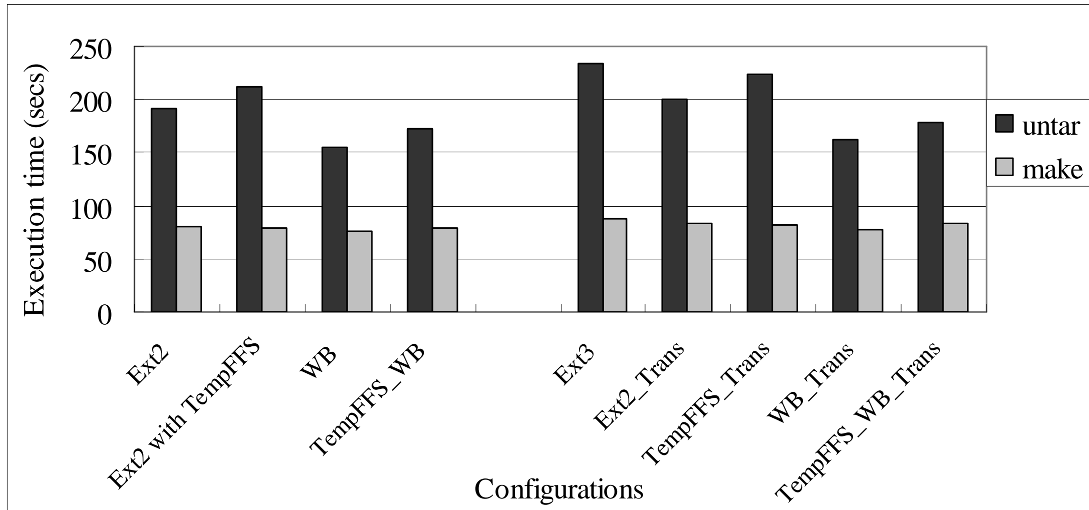


Figure 4.17 Performance Results of Different Combinations (Linux Kernel Untarring and Compilation)

Figure 4.17 shows the execution time of Linux kernel untarring and compilation under various configurations. Similar to the results in Figure 4.12, both relo and the lightweight transaction support mechanism lead to performance improvements, and incorporating TempFFS into the system causes performance degradation. This is because TempFFS does not have performance improvement under the untar workload.

Similar to the previous observations, the performance of the *make* workload is almost the same among all configurations.

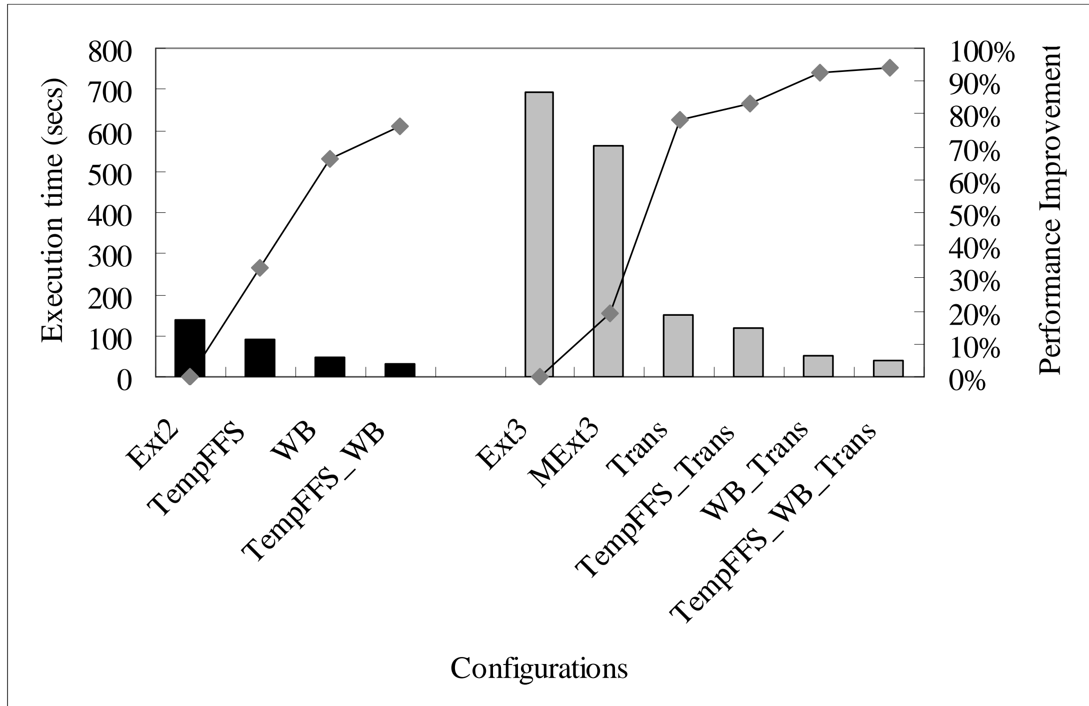


Figure 4.18 Performance Results of Different Combinations (Postmark, 512-10K bytes)

Figure 4.18 shows the performance results under Postmark. In this experiment, the file size ranges from 512 bytes to 10 Kbytes for default setting and the numbers of files are 25000. The results can be divided into two groups. In the first group (shown in the left side of the figure), the file system does not ensure consistency, whereas in the second group (shown in the right side of the figure), the file system does ensure consistency.

According to the results of the first group, the performance of TempFFS_WB achieves the best performance. Specifically, it has a 76% performance improvement, while TempFFS has a 33% performance improvement and the relo write back policy has a 66% performance improvement, when compared to ext2. Different from the previous results, combining TempFFS does have a positive effect on the system performance for the Postmark workload. Therefore, TempFFS_WB results in the best performance.

The second group reveals a similar result. The TempFFS_WB_Trans configuration achieves the best performance and outperforms ext3 by 94%. A large portion of the performance improvement is due to the lightweight transaction support mechanism, which outperforms ext3 by 78% when being used alone. Note that, the small differences in the following pairs (TempFFS, TempFFS_trans), (WB, WB_trans), and (TempFFS_WB, TempFFS_WB_trans) again demonstrate that the performance overhead of our transaction support mechanism can nearly be ignored.

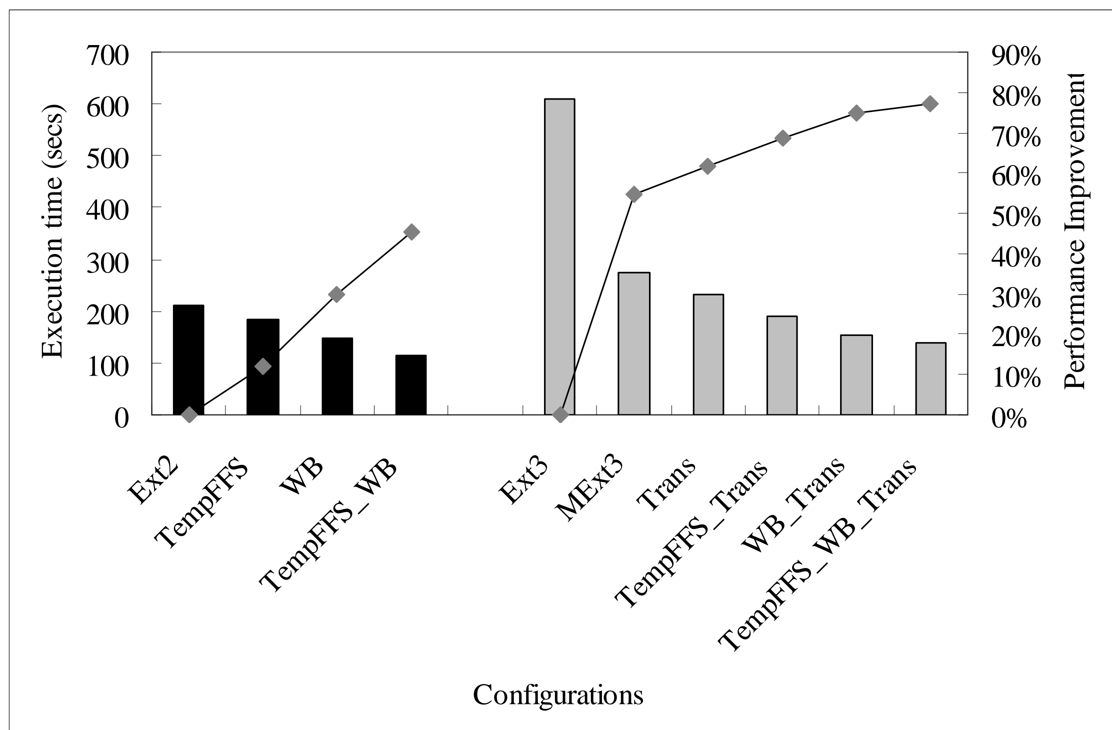


Figure 4.19 Performance Results of Different Combinations (Postmark, 512-2M bytes)

In the last experiment, we measure the performance of different configurations under Postmark that the file size ranges from 512 bytes to 2 Mbytes and the numbers of files are 25000 because we set the threshold of file size in TempFFS is 1Mbytes.

Similar to the results in Figure 4.18, Figure 4.19 shows the results that even if the file size range is increase, TempFFS_WB achieves the best performance in the first

group, and it outperforms ext2 by about 50%. Moreover, TempFFS_WB_tans achieves the best performance in the second group, and it outperforms ext3 by 78%.

Chapter 5 Conclusion

We provide three mechanisms to improve performance of file system on NVRAM system. First, the TempFFS can avoid some unnecessary IO and reduce file fragmentations when short-lived files are deleted in TempFFS. Moreover, TempFFS is put between VFS and file system and it does not modify any codes of file system. Second, we provide an intelligent write-back policy (relo) that considers both locations of dirty blocks in the disk and recency. It not only can reduce the seek time and rotation delay, but also avoids writing back recently-updated dirty pages. Last, we provide a simple transaction support in NVRAM system. It can make sure the file operation is consistent and does not produce any extra disk IO overhead.

In performance results, we not only show the performance improvement per mechanism, but also show the performance improvement of all combinations among three mechanisms.

References

[1] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer, "Non-Volatile Memory for Fast, Reliable File Systems", *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.10-22, October 1992.

[2] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX environment", *Proceedings of the USENIX Summer Conference,* June 1992.

[3] A. Batsakis and R. Burns, "AWOL: An Adaptive Write Optimizations Layer", *Proceedings of the USENIX Conference on File and Storage Technologies*, pp. 67-80, February 2008.

[4] A. Ames, N. Bobb, S. Brandt , A. Hiatt, C. Maltzahn, E. Miller, A. Neeman, and D. Tuteja, "Richer File system Metadata Using Links and Attributes". *Proceedings of the 13th NASA Goddard Conference on Mass Storage and Technologies* (TSST' 05), pp. 49-60, April 2005.

[5] T. Bray, "Bonnie++ benchmark", http://www.coker.com.au/bonnie++/

[6] K. Chen, R. B. Bunt and D. L. Eager, "Write Caching in Distributed File Systems", *Proceedings of the 15th International Conference on Distributed Computing Systems,* pp.457-466, June 1995.

[7] P. M. Chen., "Optimizing Delay in Delayed-Write File Systems", Technical Report CSE–TR–293–96, University Michigan, May 1996.

[8] P. Chen, W. NG, G. Rajamani, C. Aycock and D. Lowell "The Rio File Cache: Surviving Operating System Crashes", *Proceedings of the International*

Conference on Architectural Support for Programming Languages and Operating Systems, pp. 74-83, October 1996.

[9] R. Desikan, S. W. Keckler, D. Burger and R. Austin. "Assessment of MRAM Technology Characteristics and Architecture", Technical Report CS-TR-01-36, University of Texas at Austin, Department of Computer Sciences, April 2001.

[10] B. Dipert. "Exotic Memories, Diverse Approaches", EDN Magazine, pp.56-70, April 2001.

[11] I. H. Doh, J. Choi, D. Lee and S. H. Noh, "Exploiting Non-Volatile RAM to Enhance Flash File System Performance", *Proceedings of the International Conference on Embedded Software*, October 2007.

[12] N. K. Edel, D. Tuteja, E. L. Miller and S. A. Brandt, "MRAMFS: A Compressing File System for Non-volatile RAM", *Proceeding of the IEEE Society's 12th Annual International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, pp. 596-603, Oct. 2004.

[13] N. K. Edel, E. L. Miller, K. S. Brandt and S. A. Brandt, "Measuring the Compressibility of Metadata and Small Files for Disk/Nvram Hybrid Storage Systems", *Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, July 2004.

[14] B. C. Forney, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "Storage-aware Caching: Revisiting Caching for Heterogeneous Storage Systems", *Proceedings of the First Usenix Conference on File and Storage Technologies*, pp.61-74, January 2002.

[15] G. Ganger and M. F. Kaashoek, "Embedded Inodes and Explicit Grouping:

Exploiting Disk Bandwidth for Small Files", *Proceedings of the 1997 USENIX Technical Conference*, pp. 1-18, January 1997.

[16] B. S. Gill and D. S. Modha, "WOW: Wise ordering for writes – combining spatial and temporal locality in Non-Volatile Caches", *Proceedings of the 4th Conference on File and Storage Systems*, pp. 129–142, Dec. 2005.

[17] K. M. Greenan and E. L. Miller, "Reliability Mechanisms for File Systems using Non-Volatile Memory as a Metadata Store", *Proceedings of the International Conference on Embedded Software*, October 2006.

[18] K. M. Greenan and E. L. Miller, "PRIMS: Making NVRAM Suitable for Extremely Reliable Storage", *Proceedings of the 3rd workshop on Hot Topics in System Dependability,* April 2007.

[19] T. Haining and D. Long., "Management Policies for Non-volatile Write Caches", *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference*, pp. 321–328, February 1999.

[20] R. Y. Hou and Y. N. Patt., "Using Non-volatile Storage to Improve the Reliability of RAID5 Disk Arrays", *Proceedings of the 27th International Symposium on Fault- Tolerant Computing*, pp. 206–215, June1997.

[21] S. Jiang, X. Ding, F. Chen,E. Tan, and X. Zhang, "DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality", *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, pp.101-114, December.2005.

[22] J. Katcher, "Postmark: A New File System Benchamrk". Technical Report TR3022 Network Appliance Inc, October 1997.

[23] A. Kawaguchi, S. Nishioka, and H. Motoda., "A Flash Memory Based File System", *Proceedings of the 1995 USENIX Technical Conference*, pp. 155–164, January 1995.

[24] M. Levy, "Memory Products, Chapter Interfacing Microsoft's Flash File System", *Intel Corporation*, pp. 4-318-4-325, 1993.

[25] B. Liskov, S. Ghemawat, R. Gruber, and P. Johnson, "Replication in the Harp File System", *Proceedings of the 1991 Symposium on Operating System Priciples*, October 1991.

[26] S. Lim, H. J. Choi, and K. H. Park, "Journal Remap-Based FTL for Journaling File System with Flash Memory", *Proceedings of the High Performance Computation Conference* , pp. 192-203, September 2007.

[27] D. E. Lowell, P. M. Chen, "Free Transactions with Rio Vista". *ACM Symposium on Operating Systems Principles*, October 1997.

[28] C. Lumb ,J. Schindler ,G. Ganger ,D. Nagle and E. Riedel, "Towards Higher Disk Head Utilization: Extracting Free Bandwidth from Busy Disk Drives", *Proceedings of the Symposium on Operating SystemsDesign and Implementation,* pp. 87-102, October 2000.

[29] M. McKusick, W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, pp. 181-197, August 1984.

[30] M. McKusick, G. Ganger, "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast File System", *USENIX Annual Technical Conference*, pp. 24-24, 1999.

[31] E. Miller, S. Brandt, and D. Long, "HeRMES: High-Performance Reliable

MRAM-Enabled Storage", *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, pp.95-99, May 2001.

[32] J. C. Mogul, "A Better Update Policy". *Proceedings of the USENIX 1994 Technical Conference*, pp.99-111, June 1994.

[33] R. Ohmura, N. Yamasaki, and Y. Anzai, "Device State Recovery in Non-Volatile Main Memory Systems", *Proceedings of the 27th Annual International Computer Software and Applications Conference*, pp.16-21, November 2003.

[34] "Ramfs", http://lwn.net/Articles/156098/

[35] "Reiserfs", http://www.namesys.com

[36] D. Roselli, "Characteristics of File System Workloads", T*echnical Report CSD-98-1029, University of California at Berkeley,* December 1998.

[37] D. Roselli, J. Lorch, and T. Anderson, "A comparison of file system workloads", *Proceedings of the USENIX Annual Technical Conference*, pp. 41–54, Jun. 2000.

[38] C. Ruemmler and J. Wilkes, "UNIX disk access patterns", *Proceedings of the Winter 1993 USENIX Conference*, pp. 405–20, 25–29, January 1993.

[39] M. Rosenblum, and J. Ousterhot, "The Design and Implementation of a Log-Structured File System", *ACM Transaction on Computer Systems,* February 1992

[40] M. Satyanarayanan, H. H. Mashburn, P. Kunar, D. C. Steere, and J. J. Kistler, "Lightweight Recoverable Virtual memory", *ACM Special Interest Group on Operating Systems*, pp. 33-57, February 1993.

[41] W. Vogels, "File System Usage in Windows NT 4.0", *Proceedings of the 17th*

Symposium on Operating Systems Principles, pp.93-109, December 1999.

[42] A. Wang, G. H. Kuenning, P. Reiher, and G. J. Popek, "Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System". *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pp.15-28, Jane 2002.

[43] J. Wang and Y. Hu., "PROFS – Performance-Oriented Data Reorganization for Log-structured File System on Multi-Zone Disks", *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 285–293, August 2001.

[44] D. Woodhouse, "The Journaling Flash File System", *Ottawa Linux Symposium*, July 2001.

[45] M. Wu, and W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System", *Proceedings of the 6th Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 86-97, October 1994.

[46] W. Vogels, "File System Usage in Windows NT 4.0", *Proceedings of the 17th Symposium on Operating Systems Principles*, pp.93-109, December 1999.

[47] "XFS: A High Performance Journaling File System", http://oss.sgi.com/projects/xfs