國 立 交 通 大 學

資訊科學與工程研究所

碩 士 論 文

利用架構在 MRAM 下的緩衝快取增進 I/O 效能

Improving I/O Efficiency with MRAM-based Buffer Cache

研 究 生：李維國

指導教授：張瑞川　教授

中 華 民 國　九 十 五　年 六 月

利用架構在 MRAM 上的緩衝快取增進 I/O 效能

學生：李維國　　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

摘 要

MRAM(Magnetic Random Access Memory)是最近這幾年來新興的一項記憶體技術，其具有非揮發性的特性，並且有著跟傳統 DRAM 相似的存取速度，而且一般研究認為 MRAM 是短期間最可能被量產且取代現有 DRAM 的技術。而由於傳統緩衝快取的設計是建立在非揮發記憶體上，因此若是將非揮發性記憶體取代揮發性主記憶體，傳統的緩衝快取僅能藉由延長將資料寫回的時間以增進系統效能。因此我們設計一個架構在 MRAM 上的緩衝快取，考慮到除了利用記憶體非揮發性的特性延長修改過的資料寫回磁碟的時間，也可以作其他的改進以增進效能。因此在 MRAM-based Buffer Cache 的設計中，我們提出了兩個主要的機制: (1)在 VFS 和一般檔案系統之間加入一層 ramfs 檔案系統。其主要的目的是暫時儲存新建立的檔案，延遲檔案分配給其真正所屬的檔案系統的時間，等到較為確定檔案大小後再進行分配，以避免檔案 fragmentation 的情形並減少不必要的 I/O；(2)修改以往以檔案為單位將資料寫回的演算法，加入了考慮每個要寫回的 block 在磁碟上相對應的位置，讓較連續且較鄰近的資料一起寫回，減少寫回所需的時間。

Improving I/O Efficiency with MRAM-based Buffer Cache

Student: Wei-Kou Lee **Advisor: Dr. Ruei-Chuan Chang**

Computer Science and Engineering College of Computer Science

National Chiao Tung University

Abstract

MRAM(Magnetic Random Access Memory) is an emerging technology in recent years with non-volatile characteristic and access speed comparable to DRAM(Dynamic Random Access Memory). It is generally believed that MRAM may completely replace DRAM and begin mass production in the short-term. Traditional buffer cache design is based on volatile memory, so after MRAM completely replace DRAM as system main memory, we can only take advantage of delaying write to improve system performance.

Therefore we purpose a MRAM-based buffer cache considering that not only exploit non-volatile characteristic to propagate new data to disk, but also exploit non-volatile system main memory to further improve I/O efficiency. In MRAM-based buffer cache architecture, we purpose two main policy:(1) Adding ramfs filesystem between VFS and general filesystem. The main purpose is to store files temporarily ,so we can delay the time to allocate file to its original filesystem until file size is more certain to avoid file fragmentation. (2) Purpose a writeback algorithm, considering file spatial locality, to flush continuous and near blocks together to reduce writeback time.

致謝

首先感謝我最尊敬的指導老師 張瑞川教授。這兩年在老師費心的教導下，學生方能順利完成此篇論文。於授業期間，老師耐心的指導我正確的研究態度與研究方法，讓我受益良多。在撰寫論文期間，感謝大緯學長給予的建議以及不斷和我的討論。也感謝在實驗室一起努力的同學們，學長以及學弟們的支持和勉勵。

最後將此論文獻給我親愛的父母親。感謝您在我求學期間全心全意的支持，讓我得以專心的完成研究。

目錄

圖目錄

第1章 簡介

1.1. 動機

處理器的速度依照著 Moore's Law 每 18 個月就成長一倍，但是系統效能的成長卻仍往往受限於速度較慢的 I/O 之上，尤其是磁碟上。影響磁碟效能的因素除了傳輸速度(transfer rate)以外，最主要在於移動磁碟讀寫頭到對應磁軌所需的時間(seek time)以及旋轉磁盤到資料所在位置的時間(Rotation delay)。因此不同種類的檔案系統根據不同的設計期待能夠最佳化檔案在磁碟上的排列，進而降低磁碟 I/O 所需的時間[8][24]。例如 ext2 檔案系統將磁碟切割許多 block groups，它考慮到相同目錄下的檔案時常是接連著被使用者讀寫，因此在分配檔案磁碟空間時，會盡可能的將同目錄下的檔案放置在鄰近的 block group 上以加速存取時間。PROFS[24]則利用磁軌外圈傳輸速度較快的特性，選擇將較常存取的檔案放在磁軌外圈，讓愈常被存取的檔案能有愈快的傳輸速度。

而除了檔案系統的本身設計之外，減低磁碟以及 CPU 速度上的差異最常見的方法即是緩衝快取(buffer cache)。透過快取，我們可以將常用的資料或是根據資料本身不同的特性，選擇將重要的資料放在記憶體上，因此對這些資料做讀取時即不需要透過磁碟 I/O[7][12]。

而 Buffer Cache 除了可以提昇讀取的效能外，也可以做為寫入資料的緩衝區，暫時將修改過的資料存放在記憶體中，延遲資料寫入磁碟的時間(delayed write)，以致於有機會讓許多寫入到相同資料區塊的寫入要求可以合併成一個，減少寫入要求的數量，或是利用延遲寫回一次聚集大量資料再寫入磁碟，增快 I/O 速度[2]。然而一般來說 Buffer Cache 是存在揮發性主記憶體中，因此對於修改過的資料，為了避免資料因電源中斷而喪失，系統仍需定期的將修改過的資料寫回磁碟。

而隨著理論科學與製程技術的蓬勃發展，許多新一代非揮發性記憶體技術如鐵電式隨機存取記憶體(FeRAM, Ferro electric RAM)、磁阻式隨機存取憶體(MRAM, Magnetoresistive RAM)、相變化記憶體(OUM, Ovonics Unified Memory)等相繼問世[4]。它們同時也都有相當大的機會取代現存以 DRAM 為主的記憶體市場。而其中又以磁阻式隨機存取憶體(MRAM)是目前最廣泛被看好，同時也是短期間最可能被量產且應用在商業產品的技術。因此若在不久之後，系統主記憶體被 MRAM 所取代，我們即可將儲存在 Buffer Cache 中的資料視為 persistent，進而延長資料寫回磁碟的時間，增進效能。

除此之外也有許多新一代建立在非揮發性記憶體上的檔案系統，例如 Conquest[23]

、HeRMES[17]、MRAMFS[5] 等等... 相繼被提出。雖然相關研究均強調將檔案系統較常存取的資料(即 metadata 以及小檔案)放置在非揮發性記憶體中以減少 I/O 存取的次數，或是利用簡化 metadata 結構以及壓縮的技術減少資料所佔記憶體的空間，但我們評估這並非具體可行的方法。因為根據研究[6]指出，metadata 大約佔整個檔案系統容量 1%。以 60GB 的檔案系統為例，其就至少需要 600MB 的記憶體空間來完整存放其 metadata。未來磁碟所容納的使用者檔案個數增加，所需要的記憶體空間將會持續增加。即使利用資料壓縮，化簡 metadata 的結構等方法來減少所需空間，不僅需要額外的 CPU 負擔，也不是徹底解決的方法。

1.2. 我們的作法

在這篇論文中，我們提出一個建立在非揮發性主記憶體上的 Buffer Cache Model，著重於延長修改過的資料儲存在記憶體中的時間，進而做相關的最佳化，並且在不用修改檔案系統下，就可以利用到非揮發主記憶體的好處。

我們所設計的 Buffer Cache Model 將寫入記憶體中的資料視為 persistent，並根據此特性在系統中加入兩個元件來改善現有的 Buffer Cache。第一，不同於傳統的檔案系統 hierarchy，檔案建立時立刻牽涉到所屬檔案系統的運作，我們在虛擬檔案系統(VFS)和傳統檔案系統中加一層 ramfs 檔案系統，我們稱為 Temporary-ramfs，以容納新建立的檔案。因此檔案在建立時並不會在磁碟上有對應的空間，等到一段時間過後，確定檔案的大小，我們才將檔案分配至原有的檔案系統。此舉讓檔案的資料區塊在磁碟上的空間得以更加連續，增快存取的速度。第二，由於我們不需擔心資料存放在記憶體中太久沒被寫回會造成資料喪失的危險，因此我們修改以往依檔案為單位將資料寫回磁碟的演算法。在此演算法中，我們考慮資料在磁碟上對應的位置，盡量讓一次寫回時的資料都是在鄰近的區域或者是連續的資料以減少寫回時所需的 seek time 和 rotation delay，我們稱此演算法 WASL(Write According to Spatial Locality)。

而我們作法跟磁碟排程不同的地方在於磁碟排程是根據上層已經發出的讀寫要求去做排班的動作，從已有讀寫要求中選出一個較佳的處理順序[9]。而我們的演算法則是位於磁碟排程器之上，從記憶體中選出那些具有良好 spatial locality 的資料，再將其送至磁碟排程器，讓磁碟排程能得到更好的結果。

跟新一代檔案系統不同的地方在於我們認為未來主記憶體的運用方式應該仍是依照著現今的模式儲存較常被存取資料在記憶體中，並依照一定的方法作替換，因此即使隨著使用者檔案個數的增加，也不會有新一代檔案系統所遇到記憶體空間不足的問題。

我們將此 Buffer Cache Model 實作在 Linux2.6.12 上。由於無法取得 MRAM，所以

我們採用存取速度跟 MRAM 相近的 DRAM[3]。根據實驗結果顯示，Temporary-ramfs 在處理針對許多小檔案做讀寫的情況下，可以比原有的方法增進大約 10%~20%的系統效能，並改進檔案 fragmentation 的現象。而在針對單一檔案做處裡的 workload 下，由於其功能無法發揮，則有 5%以下的額外負擔。而 WASL 則根據不同的 workload 約可以增進 5% ~ 30%的效能，在檔案個數越多的情況下表現越好，主要是由於其避免了以往以檔案為單位做寫回時可能遇到檔案本身不連續或是檔案彼此之間並不位於鄰近位置的情況，而導致要花較久的磁碟 I/O 時間。

1.3. 論文架構

本論文的架構如下:在第二章中，我們會介紹其他利用非揮發性記憶體的相關研究。第三章介紹 MRAM-based Buffer Cache Model 的設計與實作，其包含了兩個主要的機制 Temporary-ramfs 以及 WSAL 寫回演算法。第四章則是實驗測量的結果，最後在第五章作出結論。

第2章 相關研究

利用非揮發性記憶體特性所做的相關研究基本上可以分成幾方面：

2.1. 系統回復

慶應大學 Ren Ohmura 等人[19]將 NVRAM 非揮發的特性應用在系統回復方面的。他們提出有關於 Device State Recovery 的研究，該研究利用非揮發性記憶體來記錄所有 CPU 與 device 之間溝通的訊息。當電源遭到非預期的中斷時，系統會重送先前記錄在記憶體上的資訊，讓周邊裝置回復到先前的狀態。

2.2. 以非揮發性記憶體為儲存裝置

由於 flash 價格低廉且具非揮發性，因此有許多專門為 flash memory 所設計的檔案系統相繼被提出。例如:Envy[26]、JFFS2 檔案系統[25]、Microsoft Flash 檔案系統[15]…等等。flash 記憶體存在著兩項限制:第一，要覆寫一個 block 之前必須先 erase。第二，flash block 具有寫入次數限制。有鑑於此，針對 FLASH 所設計的檔案系統通常會以 non-in-place update 的方式進行資料寫入，並利用 even wearing 的技術使 flash 上每個 block 被寫入的次數相近[14]。

Envy 為了更進一步加快寫入的時間，Envy[26]利用一小塊 Battery-backed SRAM 做為 write buffer，並使用 copy-on-write 的技術，當有寫入要求時，將 flash 上對應的資料複製到 SRAM 上，在 SRAM 上做修改，等到 SRAM 容量滿了，再將資料寫回 Flash memory。

除了 flash 特性的限制之外，由於目前 flash 的容量仍無法跟一般硬碟相提並論，因此通常僅能適用在較小型的系統，而這類型的檔案系統為了讓 flash 能夠容納更多資料，有的也能夠支援對於 metadata 以及 data 的壓縮，例如 JFFS2 檔案系統。

MRAM 除了讀寫所需的時間均較 flash 來的快速以外， MRAM 也不具有 flash 的限制。舉例來說，覆寫一塊 MRAM 記憶體前不需要先 erase，並且 MRAM 也沒有寫入次數限制的問題，因此 non-in-place update 與 even wearing 技術用在 MRAM 上並無好處。

2.3. 非揮發性記憶體作為快取改善系統效能

除了被拿來做為檔案的儲存裝置以外，NVRAM 最廣泛的用途即是做為 write buffer[10][11] [18] ，其目的主要是利用 NVRAM 非揮發性的特性延長資料寫回的時間。1992 年美國加州大學 Berkeley 分校 Mary Baker 等人的研究中[1]，提出如果利用 NVRAM 做為磁碟的 write buffer，在一般情況下可節省約 10-25%的磁碟寫入的 I/O request，若是檔案存取頻繁的檔案系統甚至可以達到 90%[2]。

另外，有研究[10]利用各種不同的方法，像是 least recently used(LRU)、shortest access time first(STF)、largest segment per track(LST)去管理非揮發性的 write buffer，討論不同方法對整個系統效能的影響，並發現只要少量的 write buffer 就可以減少大量的寫入要求，改善整個系統的效能。

而有研究[11]則利用非揮發性記憶體針對 RAID5 作改善，因為雖然 RAID5 提供了可靠性並且具有相當良好的存取速度。然而，根據 RAID5 的特性，每一次的寫入要求，都需要執行”read-modify-write” 的動作，總共花費四次的磁碟存取，因此 RAID5 在有大量的寫入要求的情況下，需要頻繁的存取磁碟，導致系統效能會有顯著的下降，因此相關研究即利用非揮發性記憶體做為 RAID5 的 write buffer，改善寫入時的效能。

而隨著 MRAM 技術的不斷進步，MRAM 的容量將會持續擴長，甚至將有可能取代現今的 DRAM 成為系統的主記憶體[3]。而當 buffer cache 變為非揮發性之後，我們仍可利用上述 non-volatile write buffer 的技術，延長修改過的檔案存放在 buffer cache 的時間，並利用不同的方法去管理檔案寫回的時間，增進檔案的效能。而跟傳統 non-volatile write buffer 不同之處在於 non-volatile buffer cache 可以容納更多的資料在記憶體中，因此我們認為雖然上述方法依然可增進系統效能，但是若是能夠更加積極的利用 NVRAM 容量變大的好處，例如在我們的設計中將所有新建立的檔案先放置在 Temporary-Ramfs，減少檔案系統 fragmentation 的情形，則可以更進一步的增進系統效能。

2.4. 新一代檔案系統

由於國內外許多科技大廠像 Motorola、IBM…等等已開始積極投入 MRAM 相關技術的研發，使得 MRAM 的容量也不斷的增大。鑒於此，有許多研究設計出新一代的檔案系統，其中最主要的有兩類:

- Pure NVRAM file system
- Hybrid Disk/NVRAM file system

第一類的 Pure NVRAM file system，例如:MRAMFS[5]將所有的資料均放入非揮發

記憶體中。其考慮到記憶體的大小往往不足以容納大量的資料，因此利用壓縮的方法減少資料所佔之空間，而由於 metadata 不同於一般的資料往往有較固定的格式，因此他們對於 metadata 與 data block 採取不同的壓縮方法。在 metadata 方面，壓縮可以節省 60%以上的空間。對一般小檔案而言，亦可節省約 40%~60%的空間。

第二類的 Hybrid Disk/RAM file system，有像是加州大學 Santa Cruz 分校 Storage Systems Research Center 的 HeRMES 檔案系統[17]以及加州大學 Los Angeles 分校的 Conquest 檔案系統[5]。HeRMES 因為考慮到一般檔案系統的存取，最常被存取以及修改的即是檔案的 metadata，提出了三點增進效能的方法。第一，利用壓縮或是簡化 metadata 結構的方法，減少 metadata 所佔的空間，進而將檔案系統中所有的 metadata 都放在記憶體中以增加檔案系統存取的效率。第二，將非揮發記憶體作為 write buffer 延長寫回的延遲時間減少不必要的 IO。第三，利用 MRAM 做為某些特殊資料的儲存裝置，例如儲存每個檔案最前面的幾個 blocks，因此在讀取整個檔案時，可以立刻獲得已在記憶體上的資料並同時對在磁碟上資料做 seek 的動作，加快讀取檔案的反應時間。

Conquest 檔案系統跟 HeRMES 以及 MRAMFS 不同的地方在於 Conquest 建立在假設系統擁有充足的非揮發性記憶體上，因此其將系統中所有的小檔案以及 metadata 都放置在記憶體中而將剩餘的大檔案放在硬碟中，此方法具有兩項最主要的好處。第一，由於將小檔案都移至記憶體中存放，可以避免從磁碟中存取小檔案所需的負擔。第二，由於磁碟中僅具有大檔案，因此可降低磁碟 fragmentation 的情形，最佳化大檔案在磁碟上的排列。

前述的兩類檔案系統有兩項主要的缺點。第一，根據過去研究[6]結果顯示，Metadata 約佔檔案系統的 1 %。隨著使用者擺放的檔案逐日增加，metadata 所佔的空間將持續增加，即使是現今的 DRAM 記憶體容量亦無法將檔案系統中所有的 metadata 放在記憶體中。第二，雖然 metadata 是檔案系統中經常被存取的資料結構，但並非所有的 metadata 均會經常被存取，因此如果將所有的 metadata 均放置在記憶體中，將會花費額外的空間放置非經常存取的 metadata，造成效能的下降。因此，不同於之前的研究，我們並不額外規劃一記憶體空間來放置 metadata 或是某種類型的檔案，而是跟現今一樣利用 Buffer Cache 做為檔案的快取，將經常存取的 metadata/data 置於 buffer cache 中，同時，利用 Temporary Ramfs 容納新建立的檔案，延遲檔案分配至檔案系統的時間，讓檔案的資料區塊更加連續並避免不必要的 I/O。以及在將資料寫回至磁碟時考慮資料彼此之間的 spatial locality，寫回鄰近且連續的資料，加速寫回所需的時間。

第3章 設計與實作

在這一節中，我們將介紹 MRAM-based Buffer Cache 的設計以及實作。MRAM-based Buffer Cache 主要的目的是希望能夠利用非揮發記憶體的特性以增進 I/O 效能。跟現今 Buffer Cache 不同的地方在於其建立於一個主要的假設之下 - 系統的主記憶體即為非揮發性記憶體。

利用非揮發性記憶體的特性，我們可以將尚未寫入至磁碟的資料保留在記憶體內較長的時間，而不需要擔心資料是否會因為電源中斷而流失，因此我們設計出了兩個主要的機制來改進現今 Buffer Cache:

(1)在 VFS 和一般檔案系統之間加入一層 ramfs 檔案系統，我們稱之為 Temporary ramfs。其主要的目的是暫時儲存新建立的檔案，延遲檔案分配給其真正所屬的檔案系統的時間，等到較為確定檔案大小後再進行分配，以避免檔案 fragmentation 的情形並減少不必要的 I/O。

(2) WASL 資料寫回演算法。我們修改以往以檔案為單位將資料寫回的演算法，加入了考慮每個要寫回的 block 在磁碟上相對應的位置，讓較連續且較鄰近的資料一起做寫回的動作，減少寫回所需的時間。

在 3.1 節中，我們會先介紹 Temporary-ramfs，而接著在 3.2 節中我們提出的 WASL 寫回演算法。

3.1. Temporary-ramfs

在這一節中我們將介紹 MRAM-based Buffer Cache 的第一個機制 Temporary-ramfs，3.1.1 中說明設計 Temporary-ramfs 的目標及其所帶來的好處，而在 3.1.2 節中介紹 Temporary-ramfs 的架構，而實作的方法則在 3.1.3 節中描述。

3.1.1. 設計目標

藉由利用 Temporary ramfs 可以達到下列的好處:

1. 減少檔案的 fragmentation

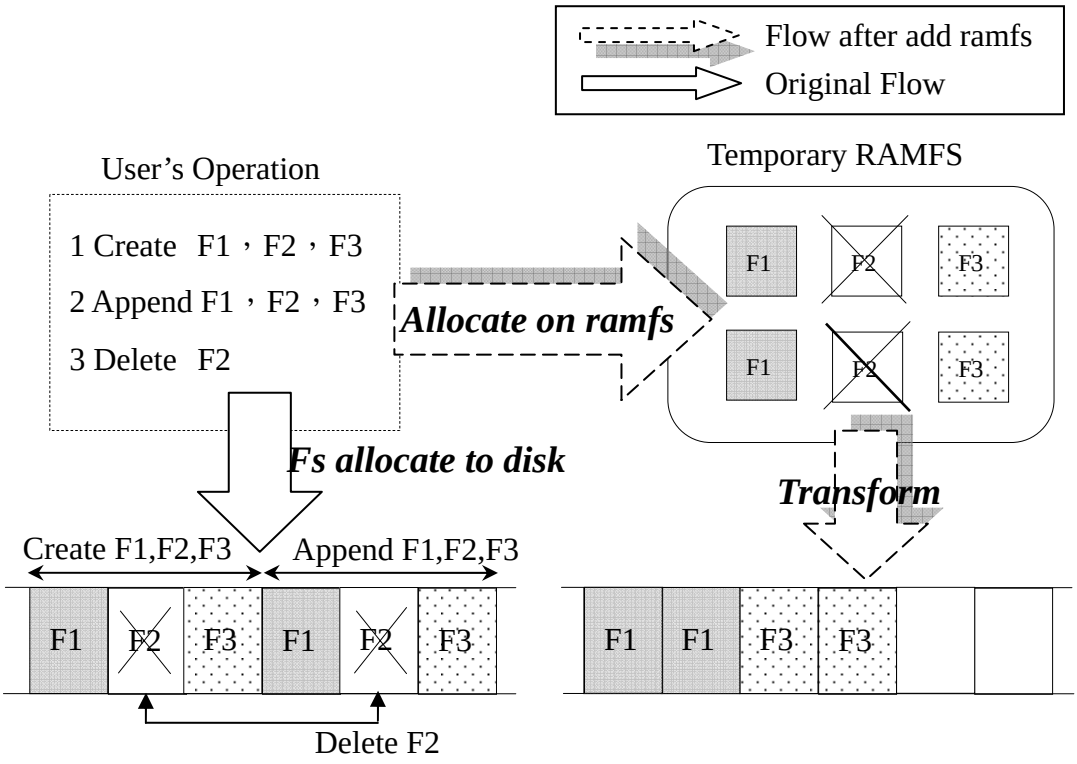
在檔案系統中，fragmentation 的產生主要是由於不斷的新增以及移除檔案，造成磁碟上可用的連續變少，因此無法配置連續空間給新建立的檔案。另外，當檔案的大小隨

著時間而增加，而新增的資料無法配置在原有的資料之後，也會造成 fragmentation。而我們利用 Temporary ramfs 延遲檔案分配給一般檔案系統的時間，一方面可避免檔案系統過於頻繁對磁碟空間做 allocate 以及 de-allocate 以減少磁碟上可用空間的 fragmentation，一方面也可以等到檔案大小較為確定後再分配檔案磁碟空間，讓檔案資料區塊更加連續。

以 Figure 3.1 (a)為例，在原來的架構下，假設使用者在同一個目錄下新增了三個檔案 F1、F2、F3，檔案系統會分配鄰近的磁碟空間給此三個檔案，若之後繼續增加 F1、F2、F3 的大小，則新增的資料區塊會分配在 F3 的資料區塊之後，此時即會造成檔案的 fragmentation。若是利用 Temporary RAMFS，如 Figure 3.1 (b)，會將新建立的 F1、F2、F3 存放在 ramfs 檔案系統中，而之後增加檔案大小以及刪除 F2 的運作都是在 ramfs 中完成，等到我們必須要將檔案寫回至磁碟時，才會把 F1 及 F3 取出來轉換成原本所屬檔案系統的檔案並分配磁碟空間，利用此方法，一方面我們可以不用分配磁碟上的空間給不久就會被刪除的檔案 F2，避免分配空間後檔案 F2 被刪除造成磁碟可用空間的 fragmentation。另一方面，也讓 F1 以及 F3 等到檔案大小較確定以後再一次分配至磁碟，減少檔案不連續的情形。

2. 減少不必要的 I/O

根據研究結果[20][22][27]得知，大部分的檔案在被使用者建立之後不久即會被刪除，但一旦檔案在檔案系統被建立，檔案系統即需要做一些動作。例如:修改對應的 bitmap 或是在其所在的目錄下加入目錄項目(directory entry)…等等的動作。這些動作都有可能需要讀取磁碟上的資料。因此若先將檔案放置在 Temporary ramfs，對於那些被新增過後不久即在 Temporary ramfs 中被刪除的檔案，由於所有的動作都是在記憶體上完成，因此不需要牽涉到磁碟上資料的讀取及寫入而導致磁碟 I/O。



(a) Original Disk Layout (b) Disk Layout with Temporary ramfs

Figure 3.1 Block Allocation

3. 透明性

由於 Temporary ramfs 是放置在 VFS 和一般檔案系統之間，所以對於使用者來說，並不需要知道檔案是在 Temporary ramfs 還是已經被分配至原本應屬的檔案系統。使用者都是透過相同的方法去存取檔案。而對於檔案系統來說，Temporary ramfs 則是架構在其之上，因此檔案系統也不知道其存在，所以不需要做修改就可以利用到 Temporary ramfs 所帶來的好處。

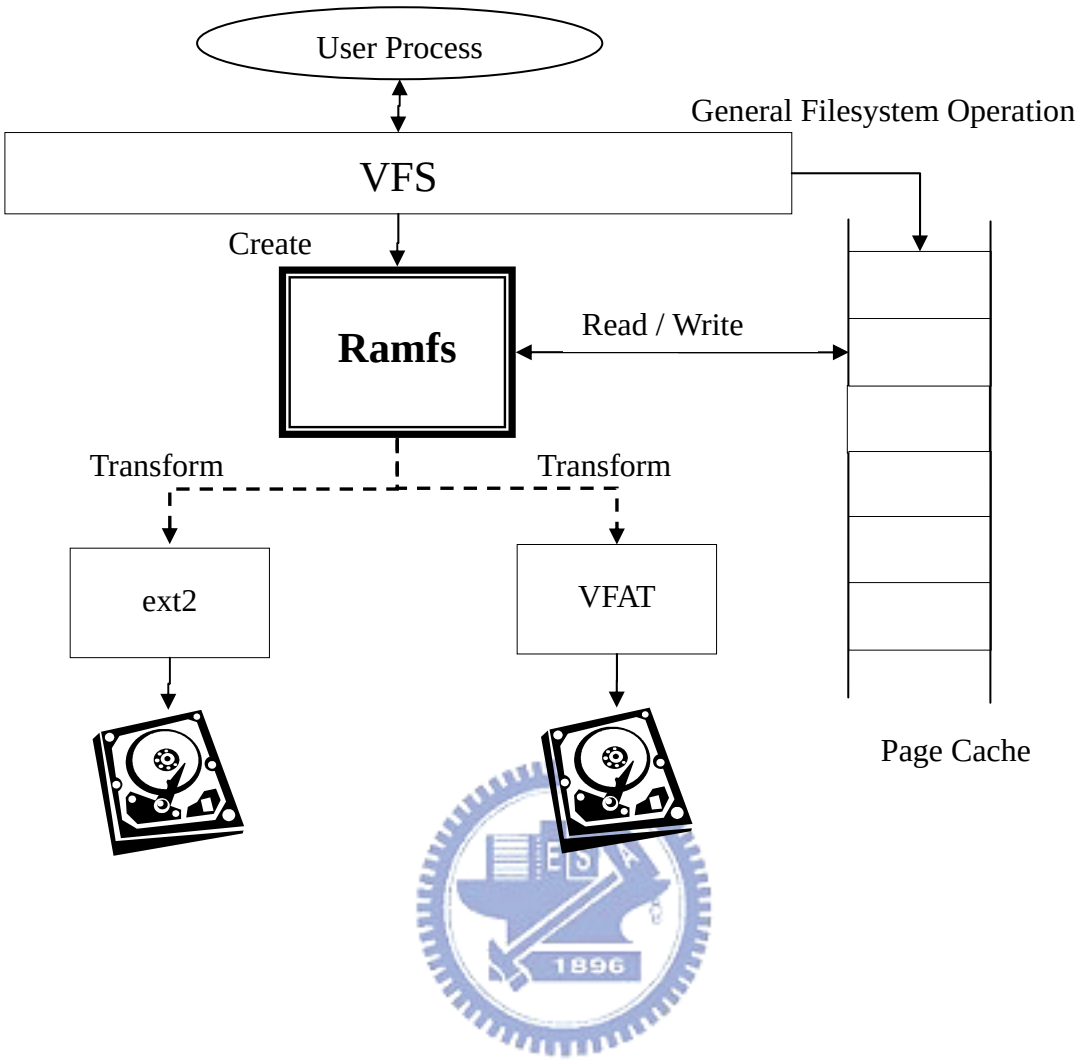


Figure 3.2 Architecture of the Temporary Ramfs

3.1.2. Temporary Ramfs 架構

在原有的檔案系統架構中，當使用者新增一個檔案時，會先透過 VFS 所提供的介面，在 VFS 中做一些處理及檢查，像是建立 VFS 層的 inode 或檢查使用者是否具有足夠的權限建立檔案，之後再呼叫檔案系統新增檔案，產生其 metadata，並在磁碟上分配空間。而我們的方法，如 Figure 3.2 所示，在 VFS 和一般檔案系統中加入了一層 ramfs 檔案系統。所有新建立的檔案都會先被建立在 ramfs 檔案系統上。由於 ramfs 是架構在記憶體上的檔案系統，所以檔案的資料在此時並不會有對應的磁碟空間，而之後對於在 Temporary-ramfs 上的檔案做讀寫都會藉由 Temporary-ramfs 檔案系統提供的 operation 來存取，而對於非 Temporary-ramfs 的檔案仍是依照跟以往相同的存取方式作存取。

而對於使用者來說，由於我們所做的修改是在 VFS 之下，因此在檔案分配至其原屬的檔案系統之前，仍是透過相同的方法去存取，使用者並不會知道檔案是放置在

Temporary Ramfs 中亦或原屬的檔案系統中。

一般在系統記憶體不足時，系統會選擇釋放部分資料所佔之記憶體空間，然而對於修改過的資料，由於資料還未寫回磁碟，因此為了維持檔案的一致性，必須先將資料寫回磁碟，才能夠釋放其所佔空間。然而由於在 Temporary-ramfs 裡的資料在磁碟上並沒有相對應的位置，因此我們無法將在 Temporary-ramfs 內的資料寫回並回收其空間，因此可以預見的是若使用者不斷新增檔案，Temporary-ramfs 裡的資料也將不斷增大，不久系統即充滿著 Temporary-ramfs 的資料而無可回收之記憶體。

因此為了讓系統也能夠釋放 Temporary-ramfs 裡資料所佔的空間，我們必須在系統需要回收記憶體時，將 Temporary-ramfs 裡較久沒被存取的資料轉換回所屬的檔案系統，而在轉換的過程中，檔案系統會分配磁碟空間給該檔案，我們即可將這些資料寫回並回收。

3.1.3. Temporary Ramfs 實作

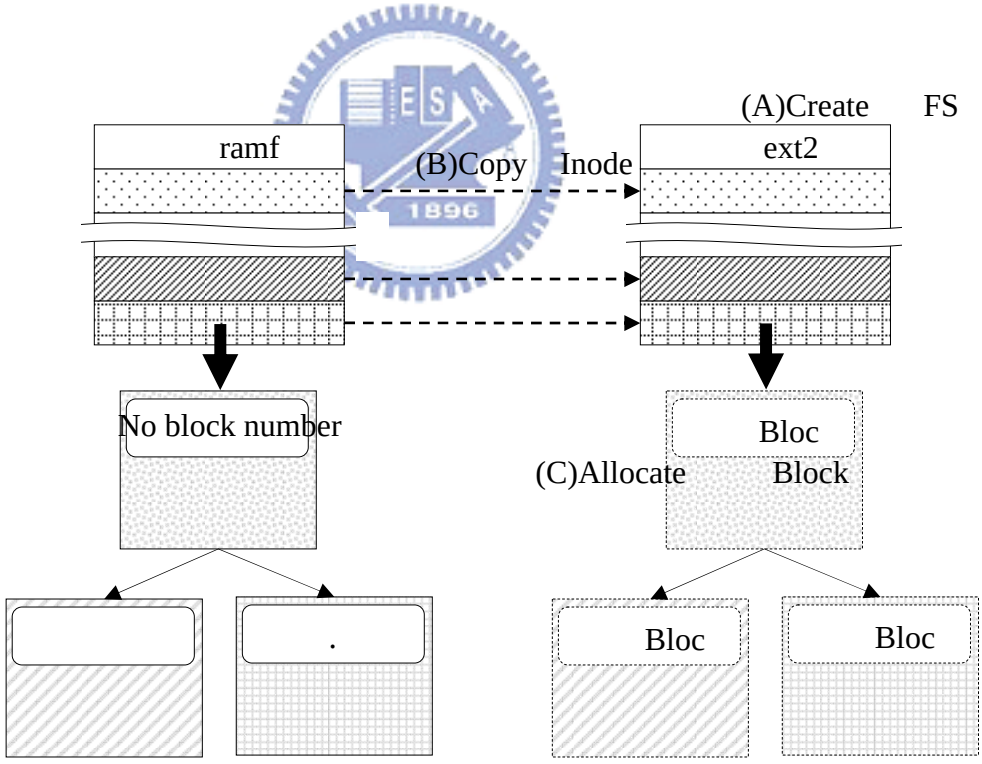


Figure 3.3 Transformation Step

在 Temporary-ramfs 的實作上，我們利用原本 Linux 上的 ramfs(Resizable simple ram filesystem)。ramfs 是以記憶體做為儲存裝置的檔案系統，其將所有檔案資料儲存在 Page Cache 中，而檔案的 metadata 則是利用 VFS 層的資料結構表示，其並不具額外的 fs-dependent 的資料結構。

為了讓所有新建立的檔案均是建立在 ramfs 檔案系統中，我們攔截虛擬檔案系統的 create 函式，並換成呼叫 ramfs 所提供的 create 函式，因此新產生的檔案在系統中會被視為屬於 ramfs 的檔案，而後續所有對此檔案做存取的動作都會轉而透過 ramfs 所提供的各式 file operations 來完成。

而在 Temporary-ramfs 裡所能容納的資料總量是動態調整的。為了充分利用記憶體，我們並沒有限制最多能夠有多少的資料可以存放在 ramfs 中，而只有當記憶體不足時我們才會需要縮減 ramfs 的大小。然而當一個檔案本身的大小超過一定的 threshold 時，我們就會先選擇將其轉換回原本的檔案系統。其主因是因為根據研究[21]顯示，大部分建立不久之後即會被刪除的檔案為較小的檔案。同時，檔案 fragmentation 主要出現在小檔案上。因此將較大的檔案，暫時保留在 Temporary-ramfs 並不會得到好處。而對於越大的檔案，因其資料量越大，在 transform 時也需要更大負擔，因此我們訂定一個 threshold(目前為 1MB)做為轉換檔案的依據。

前面提到當需要回收記憶體時，由於我們並無法直接釋放檔案在 Temporary-ramfs 所佔的空間，因此在記憶體不足時，我們需要做轉換的動作，將儲存在 Temporary-ramfs 裡的檔案轉換至原來的檔案系統以寫回。而當記憶體不足時，系統會喚醒 pdflush 核心執行緒將一些修改過的資料寫回磁碟，因此我們在 pdflush 中檢查 Temporary-ramfs 裡是否含有資料，若有，則我們即會將在 Temporary-ramfs 裡的檔案轉換回原屬的檔案系統，這樣則可將轉換過的檔案寫回磁碟並釋放其記憶體空間。

而在轉換時，我們利用檔案系統 super block 裡維持的一個鏈結串列，此串列將所有屬於此檔案系統的檔案給串連起來，並依照 LRU 做排列。而根據 LRU 我們選出較久沒被存取的檔案先作轉換的動作，因為此種檔案其大小可能已經較為確定也較有可能之後會被使用者所刪除。

而轉換的過程主要包含了三個步驟，如 Figure 3.3:

(A)首先我們會先呼叫檔案系統所提供建立檔案的函式，為此新檔案產生檔案的 metadata，在 linux 中即為檔案的 inode。

(B)由於某些原本在檔案 vfs inode 裡的欄位像是檔案建立時間、存取權限、檔案大小、使用者 ID、指向檔案資料的指標…等等是需要繼續被保留住的，因此我們將這些欄位複製到剛新建立的 inode 中。

(C)而除了 inode 外，由於檔案原本放置在 ramfs 檔案系統裡，因此其資料並沒有在磁碟上有相對應的位置，因此最後還需要呼叫檔案系統分配磁碟空間的函式(i.e. ext2_alloc_block)來分配對應的空間給檔案所有的資料，在完成分配之後就可以將原有在 ramfs 的 metadata 給清除掉。

3.2. WASL 資料寫回演算法

我們設計一個考慮 spatial locality 的寫回演算法。其考慮到資料在磁碟上相對應的位置以提昇資料寫回磁碟的效率。

在 3.2.1 節我們會說明根據 spatial locality 設計演算法的目標，3.2.2 中介紹以 Zone 為基礎所設計的演算法，而以 Segment 為基礎的演算法則會接著在 3.2.4 中描述，最後在 3.2.6 中介紹將前兩個演算法合併的 Hybrid Segment/Zone 演算法。

3.2.1. 設計目標

傳統以非揮發性記憶體為主的系統，為了避免資料因為無預期電源中斷而流失，會定期將 buffer cache 內修改過的資料寫回磁碟。若現今系統主記憶體為非揮發性記憶體，我們可以將寫入記憶體的檔案視為 persistent，因此不需要定期檢查記憶體內是否有超過一定時間仍未被寫回的檔案而將其寫回磁碟。意即，我們不用擔心有哪些過期的資料還留在記憶體中，而可以將修改過的資料累積到一定的程度，當記憶體空間不足而需要回收記憶體時，再選擇合適的資料寫回降低寫回所需的時間。

將資料寫回磁碟最主要所需的時間是在於磁碟的 seek time 以及 rotation delay，因此考慮資料在磁碟上的 spatial locality，我們在寫回時優先選擇鄰近或是連續的資料以增快 I/O 的效能。而一般磁碟排班演算法雖然也會將不同的 I/O 要求根據 block 所在的位置重新排列以減少磁碟 I/O 所需的時間，但其跟我們所設計的方法不一樣的地方在於，磁碟驅動程式是依照目前已經在寫入佇列中的要求去排出一個可以降低 seek time 以及 rotation delay 的順序，並且必須讓讀寫要求在一定的時間內完成，避免飢餓(starvation)的現象。然而我們的方法則是架構在磁碟驅動程式之上，從所有在記憶體上被修改過的資料中依照資料在磁碟上的位置，選擇鄰近或是連續的資料，送至磁碟驅動程式，因此放到佇列中的寫入要求本身就已經具有良好的 locality，因此可以更進一步降低寫入所需的時間。

3.2.2. Zone-based 演算法設計

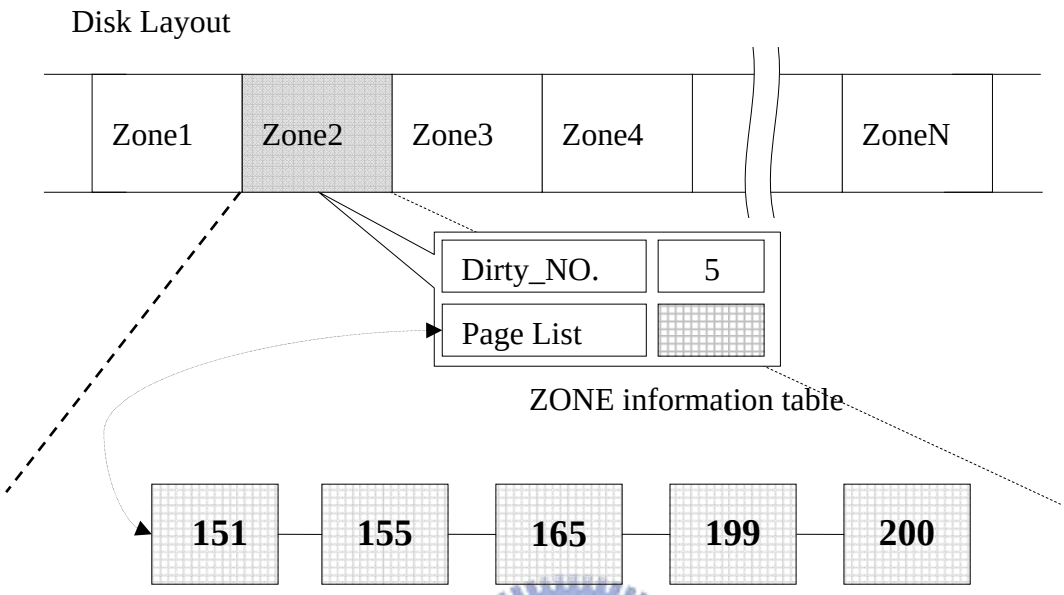


Figure 3.4 Zone Information Table

對於磁碟上 I/O 動作所需的時間，其中一大部分的因素為移動讀寫頭到不同的磁軌所需的時間(seek time) 。而移動的距離越長，所需的時間也越多，因此若我們能夠讓寫回時的資料盡量位於鄰近的磁軌，則可以減少寫回時讀寫頭在磁軌之間移動的距離[8]，進而減少磁碟 I/O 的時間。

為了辨識哪些資料為鄰近的資料，我們將磁碟切割成一段段的 Zones，每一個 Zone 代表磁碟上一段連續的 blocks，如 Figure 3.4 所示。我們將每個 Zone 的狀態記錄在 Zone Information Table 中，其記錄了每個 Zone 目前含有多少個修改過的資料分頁，同時它也把這些資料都串在一鏈結串列中。由於在同一個 Zone 裡的資料分頁在磁碟上是較為接近的，因此此演算法的基本想法就是在需要做大量寫回時，選擇相同 Zone 裡的資料做寫回的動作以減少磁頭移動的距離，進而加速 I/O 動作。

在此演算法中，我們在平時寫入檔案的時候就將相關的狀態記錄至 Zone Information Table。當系統需要將記憶體上的資料寫回時，我們便從所有的 Zones 中挑選出一個具有最多被修改過資料分頁的 Zone，並將此 Zone 裡所有修改過的資料都寫回磁碟，若寫回的資料仍未達到系統所要求的數目，則再選出下一個具有最多資料的 Zone 做寫回，直到數量達到系統的要求為止。

3.2.3. Zone-based 演算法實作

在寫回演算法的實作上，除了要修改系統將分頁寫回時所用的方法以外，還需要在分頁被修改/移除時將資訊記錄起來。在 zone-based 演算法中，當分頁被修改時我們需要將資訊記錄至 zone information table 中。為此，我們攔截系統將分頁標記為修改過的分頁時所呼叫的函式-set_page_dirty，並在此函式中加入更新的資訊(add_to_zone)。如 Figure 3.5(a)所示，此函數首先先根據分頁裡 buffer head 所指向的 block number 以及每個 zone 裡所包含的分頁數計算出此分頁所屬的 zone，將分頁加入此 zone 的 page list 中並且增加 zone 內修改過的分頁的個數。

而除了當分頁被修改時我們需要更新其資訊外，分頁被寫回或是被移除時，我們也需要將此分頁從 zone info. table 中移出(我們實作出一函式來移除分頁 remove_from_zone) 。因此利用相同的方法，我們攔截將分頁標為已寫回的函式 -clear_page_dirty_for_io，以及將分頁刪除的函式- truncate_complete_page 以呼叫此函式，如 Figure 3.5(b)所示，此函數計算出分頁所屬的 zone 後將分頁移出 zone 的 page list，並減少 zone 裡的修改過分頁的個數。

在原來 linux 的設計上，當系統需要將記憶體上的資料寫回磁碟時，會喚醒 pdflush 核心執行緒，並呼叫 write_inodes 函式執行寫回的動作。而寫回的資訊則放置在 writeback_control 資料結構內，此資料結構記錄一些寫回時的資訊，像是需要寫回的分頁數。而我們將原本以檔案為單位作寫回的方法 (即 write_inodes 函式)替換成我們所設計的 zone algorithm(即 writeback_zone 函式)，如 Figure 3.5(c)所示，我們從所有的 zone 中先選出一個具有最多修改過分頁的 zone，並將此 zone 中所有的分頁寫回磁碟。在寫回一個 zone 後我們會利用 writeback_control 資料結構內記錄目前寫回數量的變數 (nr_to_write)檢查寫回的資料量是否滿足系統所要求，若滿足則結束寫回的工作，反之則繼續選擇下一個 zone 寫回。

```
/* 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;
        insert page into the page list of that zone;
        increase the dirty page number of the zone by one;
}
```

(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;
        remove page from the page list of that zone;
        decrease the dirty page number of the zone by one;
}
```

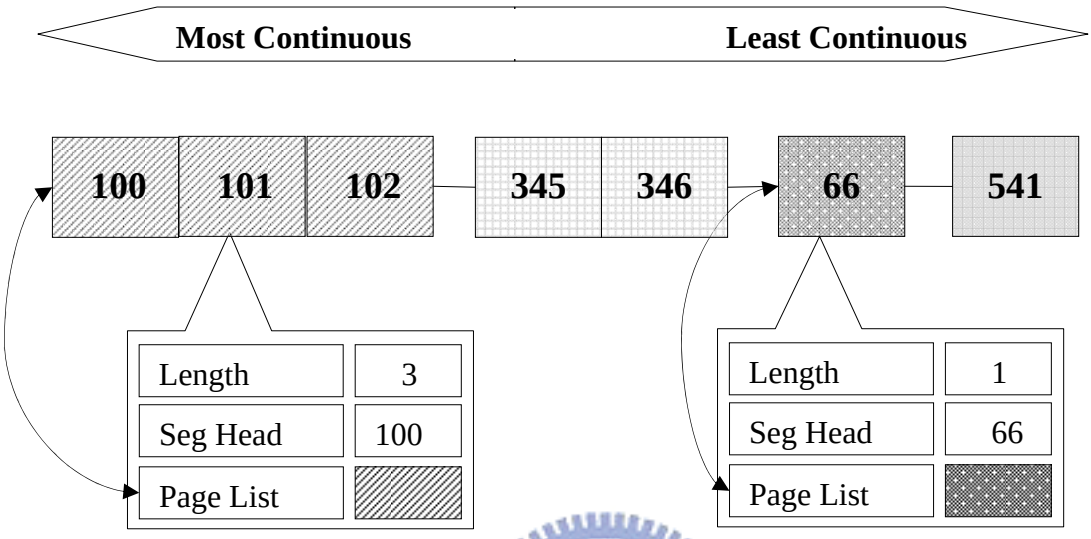
(b)

```
/* Zone-based writeback algorithm*/
writeback_zone( writeback_control wbc ){
        select the zone with most dirty pages;
        writeback all the pages in the page list of the zone;
        if (writeback_pages >= wbc.nr_to_write)
                finish;
        else
                writeback_zone( wbc );          // continue to writeback
}
```

(c)

Figure 3.5 Pseudo Code of Zone-based Algorithm

3.2.4. Segment-based 演算法設計



Segment descriptor

Figure 3.6 Segment descriptors

磁碟 I/O 動作最耗時之處除了磁軌的移動之外，另一即是將磁頭轉動到所需存取的 block 上的動作(該時間稱為 rotation delay)。雖然根據一般的研究[16]顯示，影響存取磁碟所需時間最大的還是 seek time，不過若是存取資料的 locality 到達一定程度時，rotation delay 的影響就會比 seek time 的影響還來的大，因此為了降低 rotation delay，我們傾向在將資料寫回磁碟時，以連續的資料為優先。因此我們在此演算法中，將修改過且連續的分頁視為一個 Segment，在寫回時，以類似 greedy 演算法的方式，從所有的 segments 中選出具有最多修改過的分頁的 segment 做寫回以降低 rotation delay。

在此演算法中，我們將所有被修該過的分頁以 segment 為單位串成一個 global list，對於每一個 Segment 我們都需要利用一個 segment descriptor 紀錄其內容，從 Figure 3.6 中可以看出每個 segment descriptor 含有三個主要的欄位，Length、Segment Head 以及 Page List，Length 表示此 segment 所含有的分頁數、Segment Head 紀錄此 segment 開頭的第一個分頁的 block number，Page List 則用來將該 segment 所包含的所有的 page 都串成一列。

為了在寫回時我們可以很快的選出適當的 segments，我們將所有的 Segment 依照長度做排序。因此當系統需要寫回時，我們從排序好的 Segment 串列中依照長度依序取出較長的 Segment 做寫回的動作，直到數量達到系統的要求為止。

3.2.5. Segment-based 演算法實作

如同 zone-based algorithm 維持其 zone information table 的方式，在 segment-based 演算法的實作上，我們攔截相同的函式，在分頁被修改時將其資訊加入 segment descriptor (add_to_segment)。如 Figure 3.7(a)所示，該函式首先利用分頁 buffer head 欄位取得分頁的 block number，並且利用我們建立的 block bitmap 檢查新增的 block 前後是否已有修改過的 block。而可能有下列三種情況，第一種情況為前後均有修改過的 block 表示我們需要合併兩個 segment，第二種情況為僅有一邊具有修改過的 block，在此情形下則可直接將分頁加入已存在的 segment 並增加其長度，第三種情況則是前後均不具修改過的 block，此情形表示新增的分頁為一新的 segment，因此我們需要建立一個 segment descriptor。

而取得分頁所屬的 segment descriptor 後，即可將此分頁加入該 segment 的 page list，並增加 segment 的長度。反之若是當分頁被寫回或是被刪除後，如 Figure 3.7(b) remove_from_segment 函式所示，也需要先檢查其前後是否已有修改過的 block，而可能的情形共有三種。第一，前後均有修改過的 block 表示需要將 segment 分裂成兩個各別的 segment。第二，若僅有一邊具有修改過的 block 則表示我們僅需減少 segment 的長度，將分頁從 segment descriptor 中的 page list 移出並減少分頁的長度。第三種情況則是當前後均沒有修改過的 block 則代表可以將此 segment 移除掉。

如 Figure 3.7(c)所示，在寫回時(即 writeback_segment 函式)，此演算法會根據依照 segment 長度所排序的串列，取出具有最多修改過的分頁的 segment，並將此 segment 內所有的分頁寫回。而當一個 segment 內所有的分頁均寫回過後，檢查是否已滿足系統要求的寫回數量，若未滿足則繼續取出下一個 segment 寫回，若以達到系統需求，則結束寫回的動作。

```
/* Adding a page to segment descriptor */
add_to_segment( page ){
    get page's block number;
    if (the blk is adjacent to other dirty blocks)
        get the segment descriptor & merge segments if needed;
        S = the resulting segment descriptor;
    else
        S = create a new segment descriptor;
    S.length++;
    add the page to S.page_list; }
```

(a)

```
/* Removing a page from segment descriptor */
remove_page_from_segment( page ){
    get page's block number;
    if (the blk is adjacent to other dirty blocks)
        get the segment descriptor & split segment if needed;
        S = the resulting segment descriptor;
    Else
        get segment descriptor & delete it;
    S.length--;
    remove the page from S.page_list; }
```

(b)

```
/* Segment-based writeback algorithm*/
writeback_segment(writeback_control wbc){
    select the segment with largest number of dirty pages;
    traverse the page list of the segment to writeback all the pages;
    if (writeback_pages >= wbc.nr_to_write)
        finish;
    else
        writeback_segment( wbc ); }
```

(c)

Figure 3.7 Pseudo Code of Segment-based Algorithm

3.2.6. Hybrid Segment/Zone 演算法設計

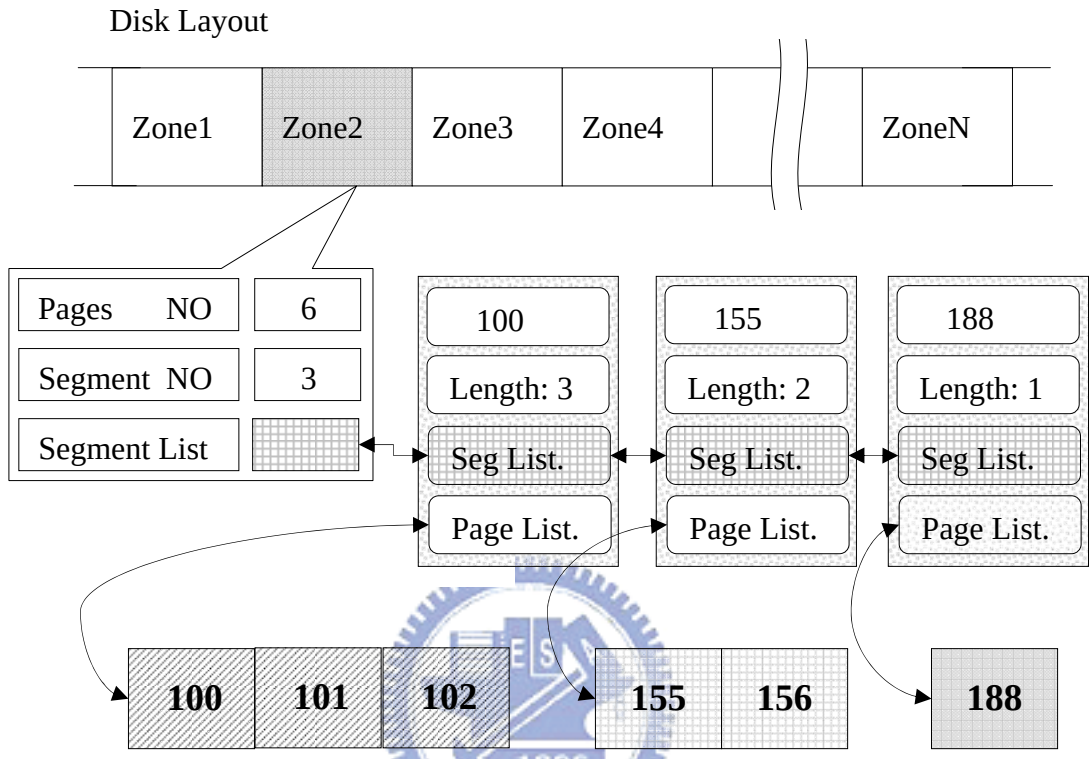


Figure 3.8 Segment/Zone information table

前面兩個演算法都各有其缺點。在 zone-based 的演算法中，當 workload 所產生之寫入要求的 spatial locality 就已經很高時，該演算法所試圖降低的 seek time 就已經不具太大的影響力，此時影響 I/O 效能最主要的因素取決於寫入的資料是否為是連續的資料，因此在此情況下，Zone-based 演算法得不到太多的好處。而在 Segment 的演算法中，雖然我們在每次做寫回時都選擇了最連續的資料，但我們並無法將所有的 Segments 限制在鄰近的區域。因此若是大部分的 Segment 之間彼此距離很遠，可能仍要花相當多的時間在 Segment 之間移動。因此為了改善這兩個演算法的缺點，我們提出了一個 Hybrid Segment/Zone 的演算法。

在 Hybrid Segment/Zone 演算法中，為了讓寫回的資料處在鄰近的位置，我們仍將磁碟切割成一段段的 Zones，如 Figure 3.8，並記錄每個 Zone 裡被修改過的分頁數。除了記錄分頁數之外，我們也利用 Segment 的方法，將一個 Zone 內連續並且修改過的 pages 視為一個 Segment，記錄一個 Zone 裡含有多少個 Segment。當需要寫回資料時，我們以 Zone 為單位選擇較合適的 Zone 寫回。然而和 Zone-based 演算法不一樣的地方在於我們並非以一個 Zone 裡修改過的分頁個數做為選擇的依據，而是依據一個 Zone 裡的平均

segment 長度，即

$$\text{Average Segment Length(ASL)} = \frac{\text{Dirty Pages per Zone}}{\text{Segment Number per Zone}}$$

考慮 Average Segment Length 的原因在於，若是單純以一個 Zone 中修改過的分頁數做為寫回的依據，我們僅能將寫回的資料限制在鄰近區域中，但這些資料可能是相當 random 的分佈在 Zone 中，而 Average Segment Length 越大則代表在寫入相同個數分頁的情形下，所經過的 segment 數較少，代表寫回的資料都是較連續的，因此可以減少 rotation delay。如 Figure 3.9 所示，在 Zone4 以及 Zone6 中被修改過的分頁均為 7 個，但是 Zone6 中的資料卻較為連續。因此若我們選擇 Average Segment Length 較大的 Zone6 則可以更快速的寫回。

利用 Hybrid Segment/Zone 的演算法，除了能夠讓寫回的資料均是在位於鄰近的 block 以減少 seek time，也能夠減少寫回時 segment 的個數減少的 rotation delay。

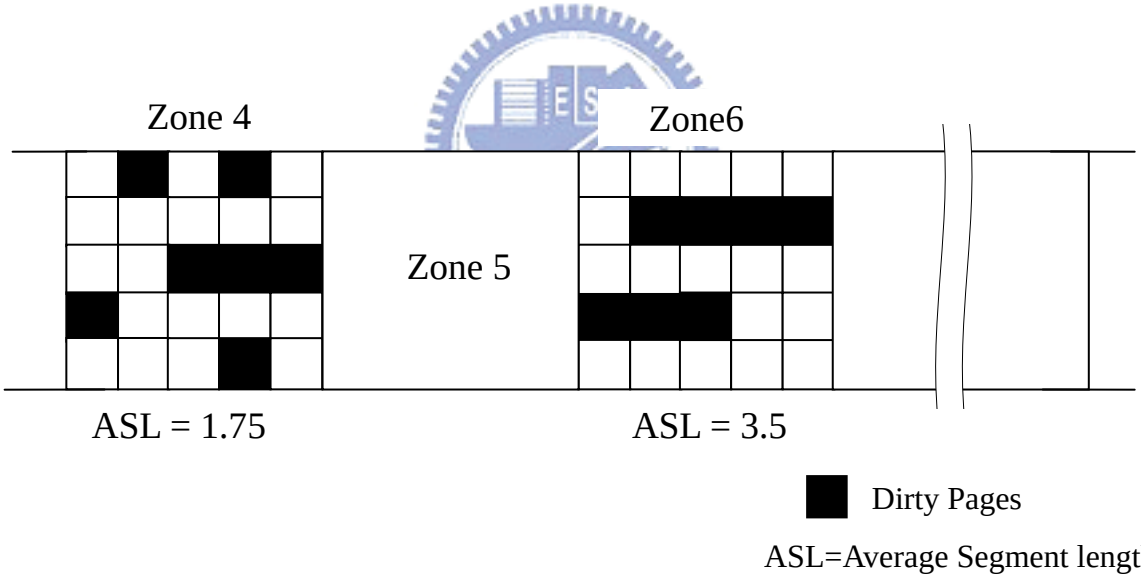


Figure 3.9 Zone with Different Average Segment Length

3.2.7. Hybrid Segment/Zone 演算法實作

Hybrid Segment/Zone 演算法跟之前 zone-based 演算法在實作上是類似的，主要差別在於當分頁被修改或是移除時，前者需要額外記錄一些有關 segment 的資訊。例如:當分頁被標記為修改過的分頁時，除了要更新有關 zone 的資訊以外，還需要檢查新增的分頁是否是一個新的 segment，而如果在 zone 中新增了一個 segment，則需將更新 zone 中

segment 個數的變數。

而將資料寫回至磁碟時則是利用每個 zone 中所紀錄修的改過的分頁數以及屬於此 zone 的 segment 數，計算出每個 zone 平均的 segment length，選出數值最大的 zone 作寫回的動作。若寫回的資料未達到系統所要求的數量，則繼續選出下一個 zone 寫回，直到達到要求為止。

```
/* 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 because of the removement of the page)
                zone.segment--;
        ASL = zone. dirty_pages / zone.segment
}
```

(b)

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

(c)

Figure 3.10 Pseudo Code of Segment/Zone Algorithm

第4章 實驗

本章的內容主要在測量以及分析 MRAM-based Buffer Cache 的效能，4.1 節會先介紹實驗環境，而 4.2 節則測量 Temporary Ramfs 的負擔。

在接下來的實驗中，我們會針對不同的 benchmark 以及設定，測量加入 Temporary Ramfs 後改進的效能，測量在寫回時採用 Segment/Zone 寫回演算法所改進的效能以及測量將兩者一起使用(即 Hybrid Method)的表現。在 4.3 節裡我們利用 Bonnie++ benchmark 測量讀寫一個大檔案時的效能，在 4.4 節中測量存取大量檔案時的效能表現，在 4.5 節則測量在不同的設定下，效能的改進。

Table 4.1 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

4.1. 實驗環境

由於大容量的 MRAM 目前仍無法取得，因此我們以一般的 DRAM 做為代替。DRAM 以及 MRAM 性質的比較如 Table 4.1 所示。從表中可以看出 DRAM 以及 MRAM 除了後者具非揮發性的特性以外，在其他方面的性質都很相似。而實驗環境如 Table 4.2 所列:

Table 4.2 Evaluation Environment

硬體	CPU	AMD Athlon 64 3000+
	Memory	512 MB
	Disk	Maxtor 80G 7200 RPM
軟體	OS	Linux 2.6.12
	Benchmarks	Postmark 1.5 & Bonnie++ 1.03a

4.2. Temporary RamFS 負擔

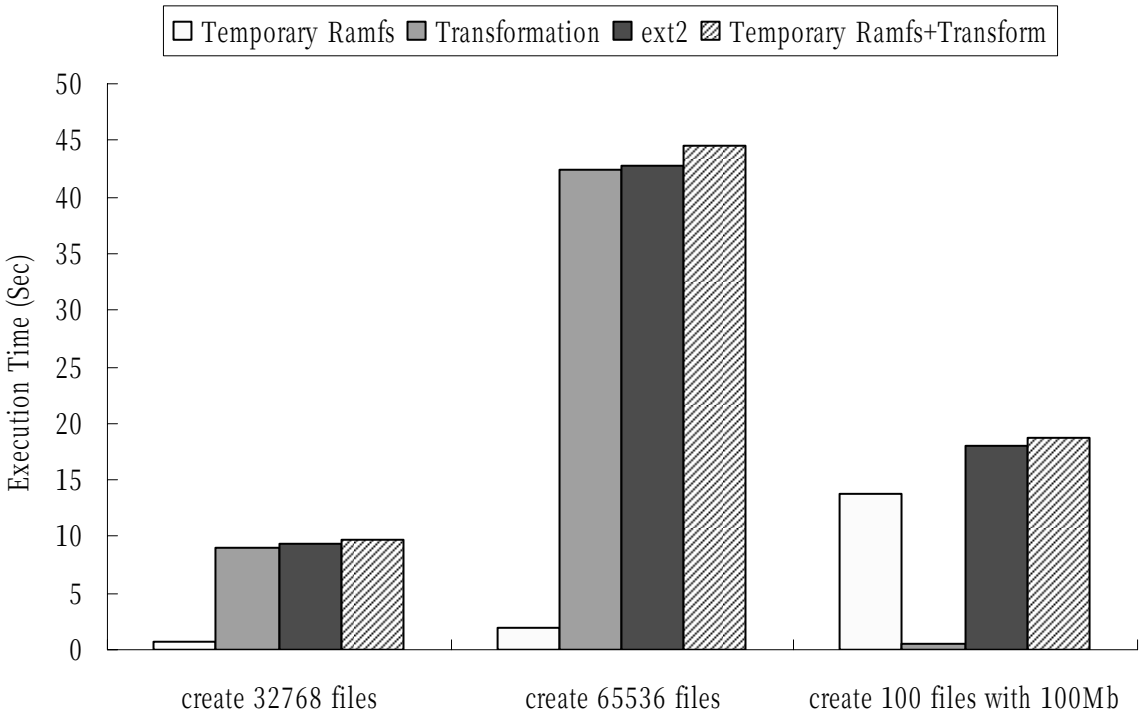


Figure 4.1 Overhead of the Temporary RamFS

在此實驗中，我們測量(1)將檔案建立在 Temporary Ramfs 中所需的時間(Figure 4.1 中 Temporary Ramfs)，(2)將檔案轉換成原本所屬檔案系統所需的時間(Figure 4.1 中 Transformation)，以及(3)將檔案直接建立在 ext2 檔案系統所需的時間(Figure 4.1 中 ext2)。我們分別測量在建立大量不含資料的檔案以及建立數個大檔案這兩種情況所需的時間。

在建立大量不含資料的檔案的測試中，我們利用自行撰寫的程式分別建立 32768 個檔案以及 65536 個檔案。從 Figure 4.1 中可以看出若僅將檔案建立在 Temporary Ramfs 上，則會比原本直接將檔案建立在 disk-based 的檔案系統來的快速。其主要原因是由於在 ramfs 中建立檔案比在 disk-based 檔案系統中建立檔案簡單，並且建立檔案的動作均在記憶體上完成，不需要讀取磁碟上的資料。

而由於轉換的動作僅需呼叫檔案系統建立檔案的函式，以及將原有檔案 metadata 中的欄位複製到新建立的 metadata 中，因此所需的時間則跟直接在檔案系統建立檔案所需的時間相近。若是計算將檔案先建立在 Temporary Ramfs 中以及爾後轉換檔案總共所需的時間，則會比直接將檔案建立在檔案系統多出約在 4%左右。其主要是由於對於每個檔案都多出了建立以及移除檔案在 Temporary-Ramfs 中 metadata 的時間，以及複製 metadata 欄位所需的時間。

在建立數個大檔案的情況下，我們亦利用自行撰寫的程式建立 100 個 100Mbytes 的檔案，結果如 Figure 4.1 右側所示。由此圖我們可以發現，在此情形下將檔案建立在 Temporary Ramfs 和直接將檔案建立在檔案系統的所需的時間較為接近。這是由於當新增一個大檔案時，不論是將檔案建立在 ramfs 中或是檔案系統中，主要的時間均花費在將使用者的資料複製到 Buffer Cache 中。

而轉換 100 個 100Mb 檔案所需的時間相對來說減少了許多，主要是由於在轉換的動作中，因資料已經儲存在 Buffer Cache 中，所以僅需複製檔案指標，無須再做記憶體複製的動作，因此由於轉換的檔案各數較少，所以轉換檔案所需的時間相對建立檔案所需的時間而言是非常少的。若衡量將檔案建立在 Temporary Ramfs 中再轉換檔案總共所需的時間，則跟前一實驗相似，大約比直接建立檔案約多 3%~4%的時間。

4.3. 單一檔案的工作負載

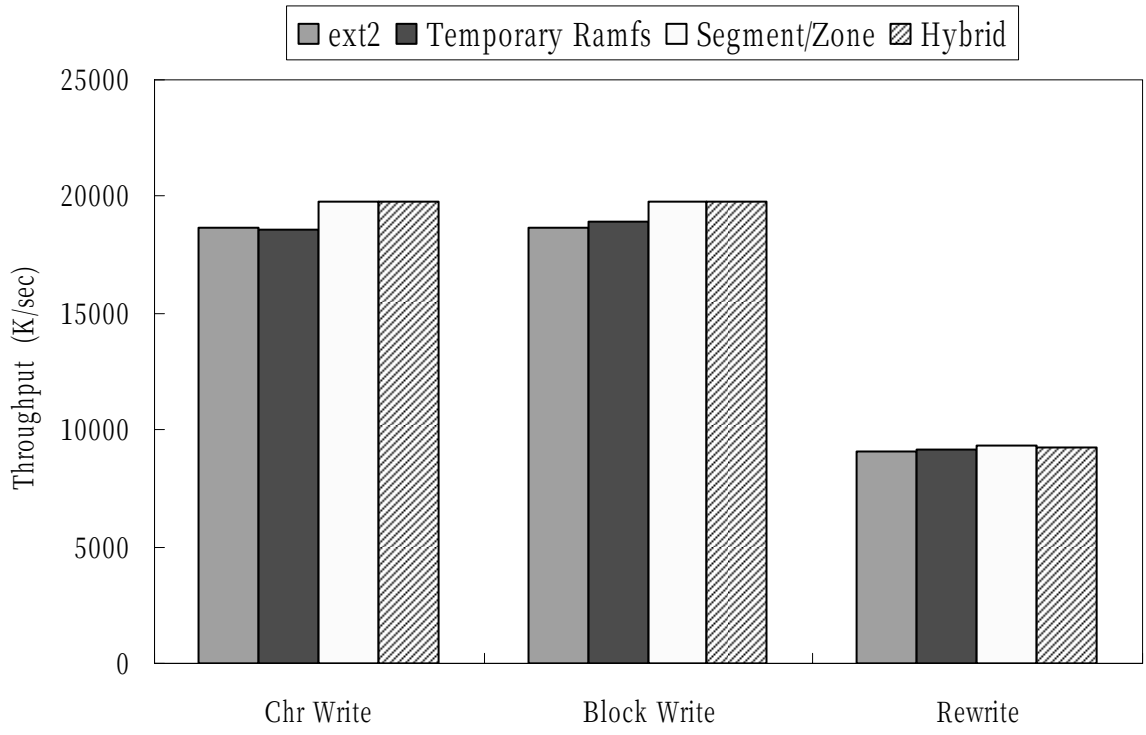


Figure 4.2 Bonnie++ benchmark, with 1-Gbyte File

為了測量針對單一檔案的工作負載，我們利用 Bonnie++ benchmark。Bonnie++為一檔案系統效能的 micro-benchmark，其主要依據使用者所訂定的檔案大小建立一個檔案，並對檔案做連續的讀寫。而在我們的設定中將檔案大小設定為 1Gbytes。我們利用 Bonnie++對此一檔案作以下不同的測試，Chr Write 的動作會以字元為單位寫入此檔案，Block Write 則以(幾 byte 為單位的)block 為單位寫入，而 Rewrite 會以 block 為單位讀取

檔案，修改然後再寫入。

從 Figure 4.2 可以看出在 Temporary Ramfs 的方法中，表現和原本的方法大致上是相同的。其主因是由於針對一個大檔案做連續寫入的 workload 並不易發生我們在 3.1 節描述的 fragmentation 現象，且檔案在轉換前也並不會被刪除，因此無法會發揮 Temporary Ramfs 的好處。

在 Segment/Zone 演算法的實驗，我們將 Zone 的大小設定為 2048e 個分頁中。而在此單一檔案工作負載的情形下，我們的方法比原本的方法增進約 2%至 5%。此效能增進並不顯著，其主要因素在於對於只處理單一檔案的 workload，原本以檔案為單位做寫回的方法不會因為挑選散佈在磁碟上各處的檔案，導致寫回時需要額外的讀寫頭移動。而在 Hybrid 的方法中，效能改進大約和 Segment/Zone 演算法相同，主要是由於在此實驗中由於 Temporary-ramfs 本身並不影響的效能，因此將兩個方法一起使用並無法更進一步增進效能。

4.4. 大量檔案的工作負載

4.4.1. 解壓縮 Linux 核心原始碼

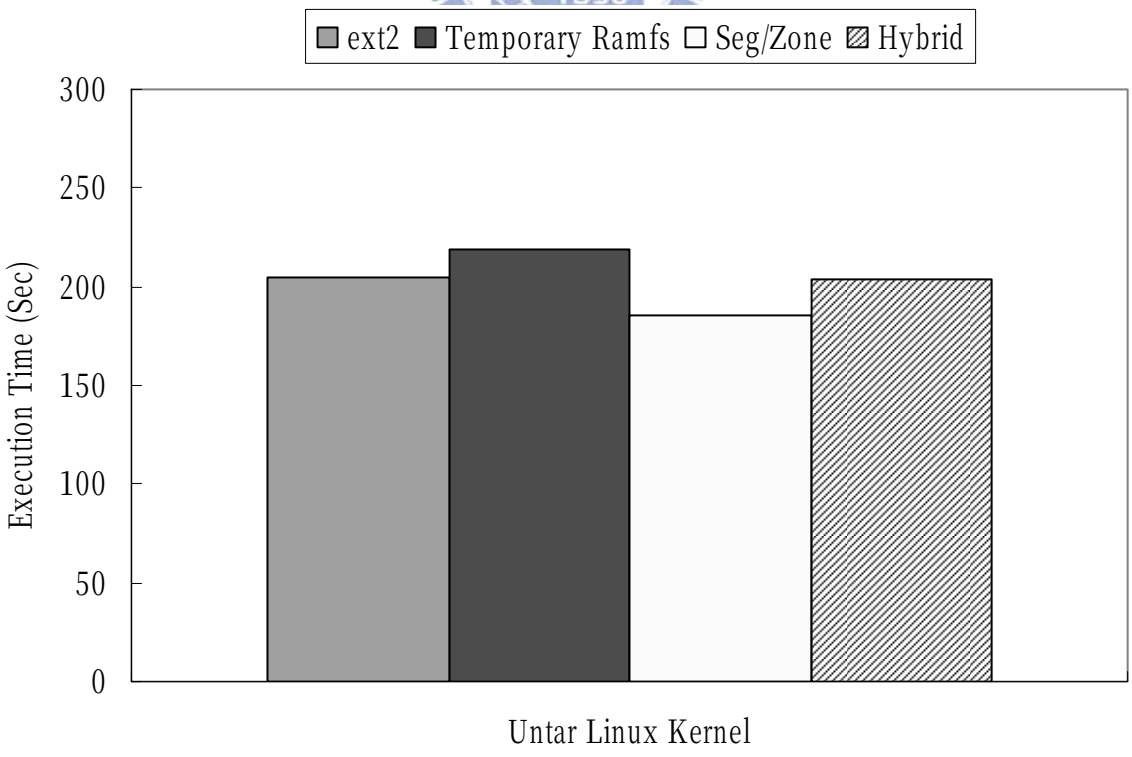


Figure 4.3 Time to Untar Linux Kernel 2.6.12

解壓縮為常見用來測量系統效能的 benchmark 之一，在此實驗中我們解壓縮編譯過的 2.6.12 Linux 核心原始碼(含核心 image、原始碼以及 object file)，壓縮檔大小為 540802444bytes。我們利用 tar 指令做解壓縮，並利用 time 指令測量解壓縮所需的時間。

解壓縮的過程主要是讀取一個壓縮檔並產生許多大小不同的檔案。實驗結果如 Figure 4.3 所示，我們可以看到 Temporary ramfs 的表現比原本的方法多出約 6~7%的額外的負擔，其原因在於雖然 untar linux kernel 會產生許多的檔案，然而並不會在產生檔案後再做修改，增加大小或刪除的動作，因此暫時將檔案放置在 ramfs 並無好處。而額外的負擔則是因為對於所有產生出來的檔案都要做轉換的動作。

Segment/Zone 寫回演算法效能的改進則大約在 10%左右，主要是由於 untar linux kernel 會產生許多檔案，因此利用我們的方法可以避免寫回時讀寫頭在檔案間移動，也因此效能的提昇比針對單一檔案做處裡的 Bonnie++ benchmark 改進的來的多。

在將 Temporary ramfs 以及 Segment/Zone 寫回演算法一起使用的 Hybrid Method 中會發現一個現象，雖然 Segment/Zone 寫回演算法改進的效能約 10%，而 Temporary ramfs 的額外負擔約 6%，但當兩個方法一起使用後，改進的效能並非為兩者相加，而是略低於兩者總合。這是因為 Temporary ramfs 的方法與 Segment/Zone 寫回演算法在設計上有一些彼此衝突的現象。就 Segment/Zone 的演算法而言，其主要是從目前在 Buffer Cache 內所有修改過的資料中選出較鄰近或是較連續的資料做寫回的動作，因此 Buffer Cache 上具有越多的資料就可以選出越好的結果。而 Temporary ramfs 的方法中，則是將資料暫時先放置在 ramfs 避免過早的分配磁碟空間，因此在記憶體中會具有一些修改過的資料，但是其並無磁碟上相對應的空間，因而減少寫回演算法可以選擇的分頁數。如此一來，當兩個機制一起使用後，效能的改進並不直接為兩個機制個別使用時改進的效能總合，而會略有所下降。

4.4.2. Postmark

Postmark[13]主要是設計用來模擬網路郵件伺服器行為的一個 benchmark。執行時，Postmark 會先產生出許多不同大小的檔案，檔案大小是隨機分佈在使用者所訂定的 Max Size 以及 Min Size 之內。之後其會執行指定數目的 transactions，而每一個 transaction 可能是 create/delete 或是 read/append。最後則刪除所有檔案。

4.4.2.1.在小型檔案上的效能

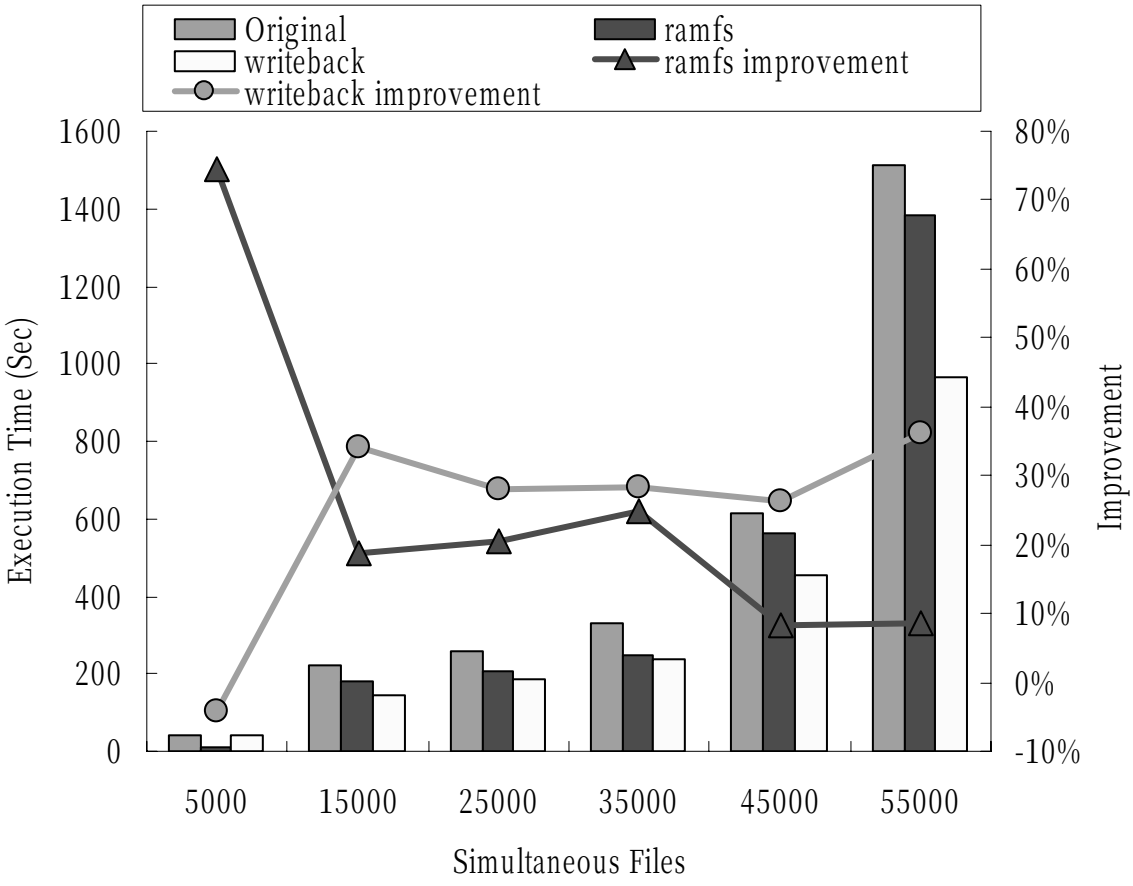


Figure 4.4 Execution Time on Postmark Benchmark with Different Simultaneous Files

在此實驗中，我們將 transaction 數設定成 200000，而初始檔案數則在 5000 至 55000 之間，除此以外都採用 Postmark 原本的設定。我們比較在不同的檔案數下，我們的方法和原本的方法表現的差異。

實驗結果如 Figure4.4 所示，圖中 X 軸代表初始的檔案個數，而 Y 軸則是執行整個測試所花的時間。從實驗結果中我們可以發現在檔案個數為 5000 時，Temporary-Ramfs 表現的比原本的方法好上許多，主要是因為在檔案個數不多的情況下，記憶體上修改過的分頁還未到達需要寫回的 threshold，因此整個實驗都是在記憶體中完成的。而原本的方法雖然產生的檔案也都是置放在記憶體中，但由於新增一個檔案需要牽涉到檔案系統的運作，因此除了行為上較為複雜外也可能需要從磁碟上讀取一些 metadata 資訊而需要額外的 I/O，因此效能表現不如 Temporary-ramfs。而同樣在檔案個數為 5000 時，Segment/Zone 寫回演算法的效能表現則比原本的方法差了約 10%，主要是因為雖然系統沒有將資料寫回磁碟，然而隨著平常對於檔案的修改以及存取，我們仍需記錄一些資

訊，比如說當修改一個分頁時，我們需要做像是將此分頁加入所屬的 Zone，或是當分頁被刪除後我們需要將其從 Zone information table 中移除等等的動作，因此仍會有額外的 CPU 負擔。隨著檔案個數的增加，Temporary RamFS 的改進幅度開始降低。例如：當檔案數為 15000~35000 時，Temporary Ramfs 的效能改進為 20%，而在檔案數超過 45000 之後則降為 10%左右。此時 Temporary ramfs 仍能發揮其功能主要原因有二。第一，許多檔案在我們暫時將其放置在 ramfs 後一段時間就被刪除，因此減少許多不必要的 I/O。第二，Temporary ramfs 所造成的 delayed allocation 讓檔案系統中的檔案得以更加連續。為了檢驗是否有大量的檔案在存放在 Temporary Ramfs 時就被移除，我們在 Postmark 測量的過程中紀錄每一個檔案被刪除時是否屬於 Temporary-ramfs 亦或已被轉換成原有檔案系統的檔案。實驗結果如 Figure 4.5，X 軸表示 Postmark 設定中的初始檔案數，Y 軸表示檔案在被刪除時屬於 Temporary ramfs 以及屬於原有檔案系統的比例。如圖所示，隨著初始檔案個數的增加，在 Temporary-ramfs 中被刪除的檔案所佔的比例則越來越少，主要是因為檔案數越多，系統也需要越頻繁的作寫回以及轉換的動作。因此也可以說明在 Figure 4.4 中隨著檔案個數的增加，Temporary ramfs 效能改進從 20%降至 10%。

我們在檔案被刪除時測量檔案資料中第一個 block 和最後一個 block 在磁碟上所相差的距離，我們稱之為 File Distance，並在執行完實驗後將所有檔案的該距離取算數平均數，結果如 Figure 4.6 所示。由圖可知，在啟用 Temporary ramfs 後，每個檔案的平均距離大約是在 1000 個 block 左右，而原本的方法則在 4000 ~ 12000 個 block 之間，並且隨著 Postmark 設定中的初始檔案數增加而增大。根據觀察，在原本的方法中，檔案資料的平均距離有如此龐大的差距的原因並非表示所有檔案都有嚴重的 fragmentation 現象，而在於某些檔案具有較嚴重的 fragmentation，因此將資料的平均距離拉高。而利用 Temporary ramfs 將檔案放置在 ramfs 一段時間，能夠減少檔案 fragmentation 情況，降低檔案資料的平均距離以增加讀寫檔案所需的時間。

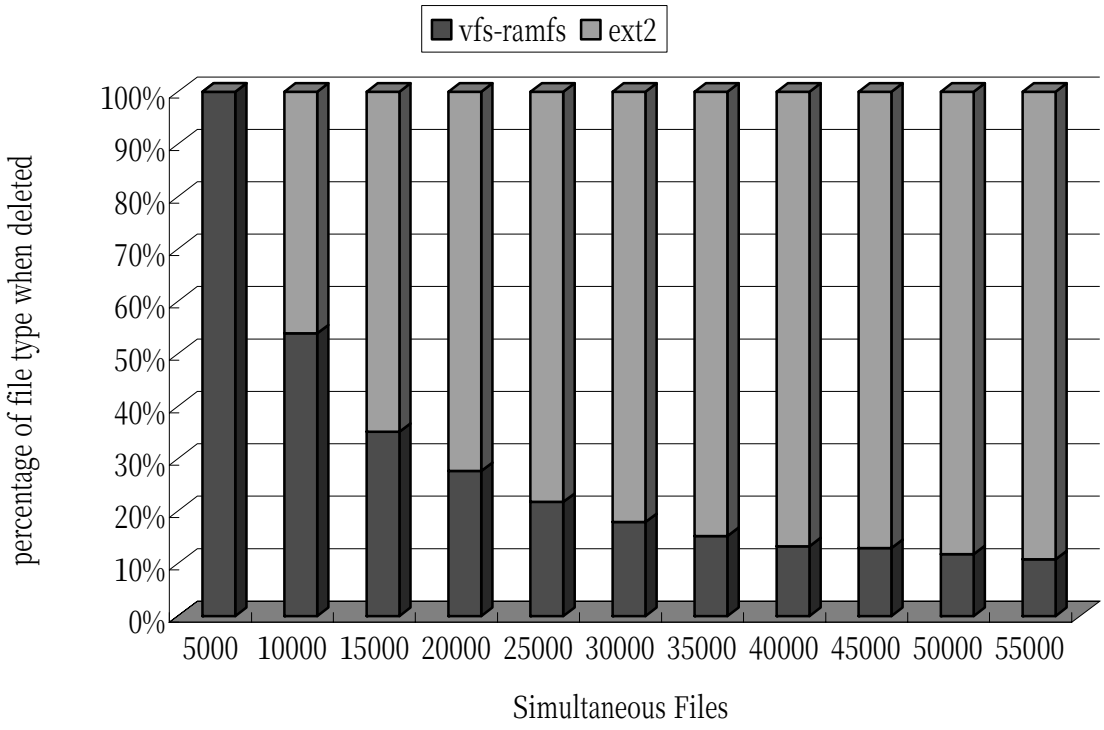


Figure 4.5 Ratios of Files Deleted in Temporary RamFS

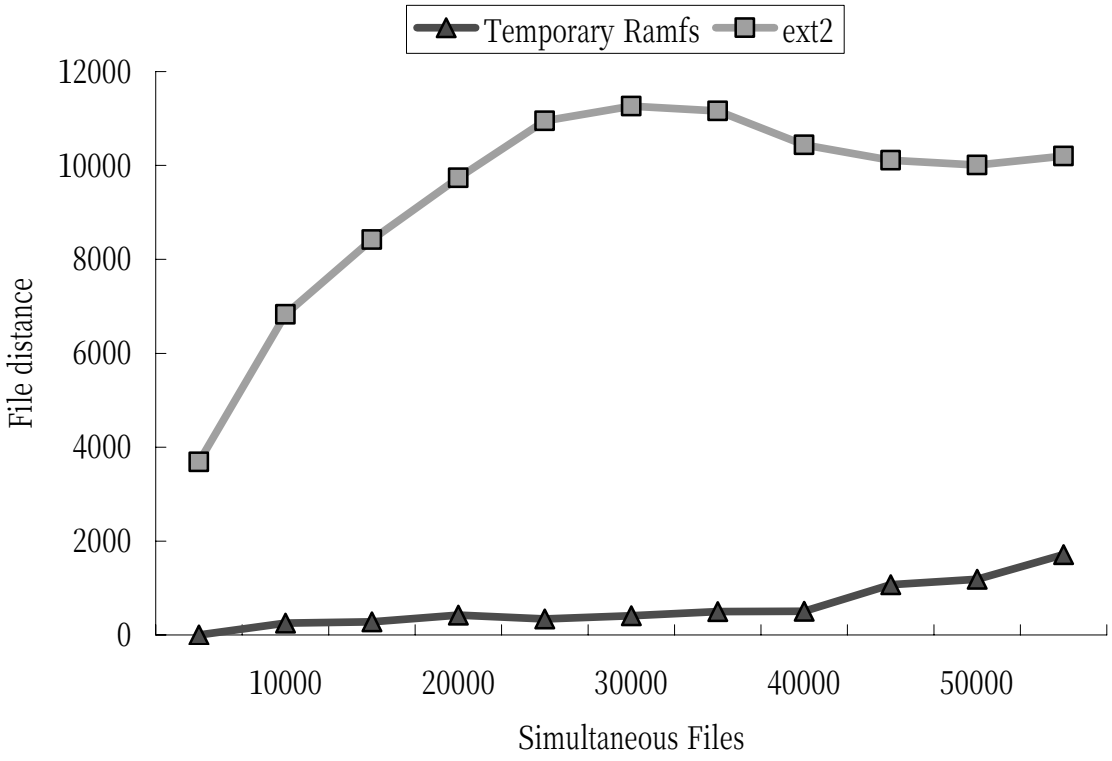


Figure 4.6 Average File Distance

在圖 4.4 中，Segment/Zone 的寫回演算法效能增進約 30%左右。其在此實驗中有較好的表現主要是因為原本 Linux 以檔案為單位做寫回的方法，在寫回許多小檔案時，可能會由於檔案彼此之間是位於較遠的距離或是檔案本身有 fragmentation 的現象，造成寫回時讀寫頭在檔案與檔案之間作較大的移動，或是寫回的資料是不連續導致需要額外的時間，而利用我們的方法，只會選擇較鄰近以及較連續資料，因此可以加速寫回的速度。

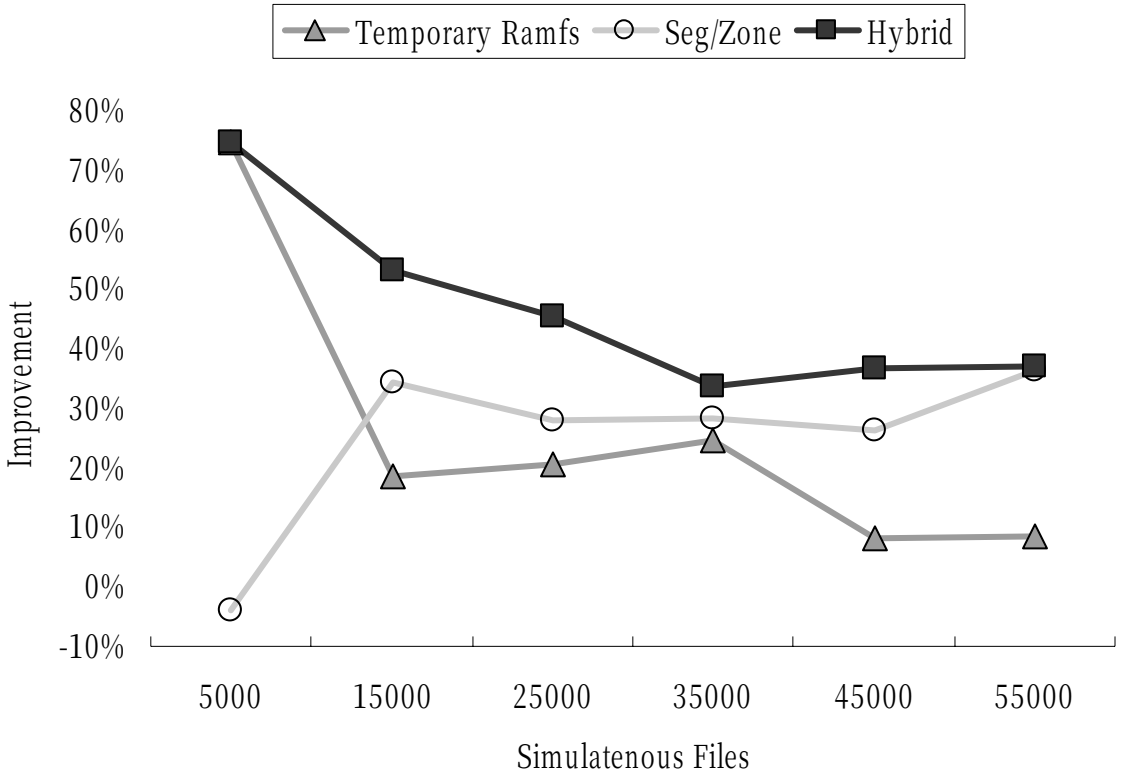


Figure 4.7 Postmark Benchmark: Hybrid Method

如前所述，隨著檔案數的增加，Segment/Zone 的方法的效能改進增加，而 Temporary RamFS 的效能改進則降低。而 Hybrid 方法的效能改進如 Figure 4.7 所示，當檔案數為 5000 時，由於不需作寫回，因此產生的檔案均放置在 Temporary Ramfs 中，因此不會有寫回演算法需要記錄額外資訊的負擔，也可同時利用 Temporary Ramfs 的優點增進效能。

而在檔案數超過 15000 的效能上，實驗結果顯示 Hybrid 的方法改進的效能約在 30%~40%之間，比僅用 Temporary Ramfs 或是 Segment/Zone 寫回演算法均來的多，但並非兩者改進的效能之總合，主要原因如同 4.4.1 節中所提，因為 Temporary Ramfs 以及寫回演算法的表現會互有干擾所致。

4.4.2.2.在大型檔案上的效能

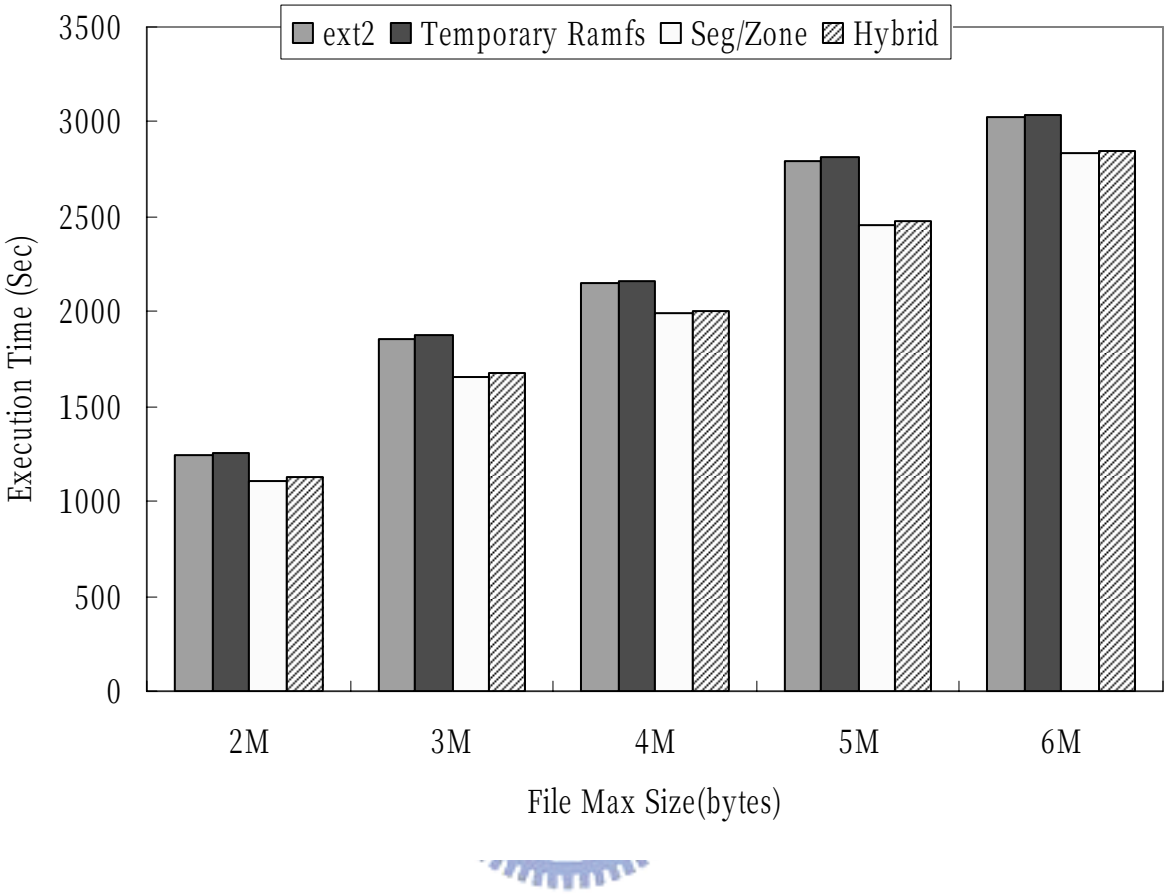


Figure 4.8 Execution Time on Postmark Benchmark with Large-scale Files

在此實驗中，我們調整 postmark 的設定值，測試存取的資料為大檔案時的情形。圖中 X 軸表示檔案最大的大小，檔案大小從 2Mbytes 到 6Mbytes 之間，transaction 數為 200000，檔案數為 500。

實驗結果如 Figure 4.8 所示，在 Temporary Ramfs 的效能上，由於處裡的檔案均超過 1Mbytes，因此檔案在建立後即會因為超過轉換檔案大小的 threshold 而被轉換成原本檔案系統的檔案，因而不會利用到 Temporary Ramfs 的功能，所以所需時間會比直接建立至檔案系統的時間多了轉換的負擔，約在 5%以內。

由於 Postmark 對檔案作隨機的寫入，以檔案為單位做寫回的方法在對於一個檔案內修改過的部分作寫回時仍有可能寫回不連續的資料，因此即使在此實驗中檔案數較少，我們仍可從結果中發現 Segment/Zone 寫回演算法比原本的方法約增進 10%左右的效能。在 Hybrid 方法改進的效能上，由於 Temporary Ramfs 並未發揮其好處，因此結果和 Segment/Zone 寫回演算法相同。

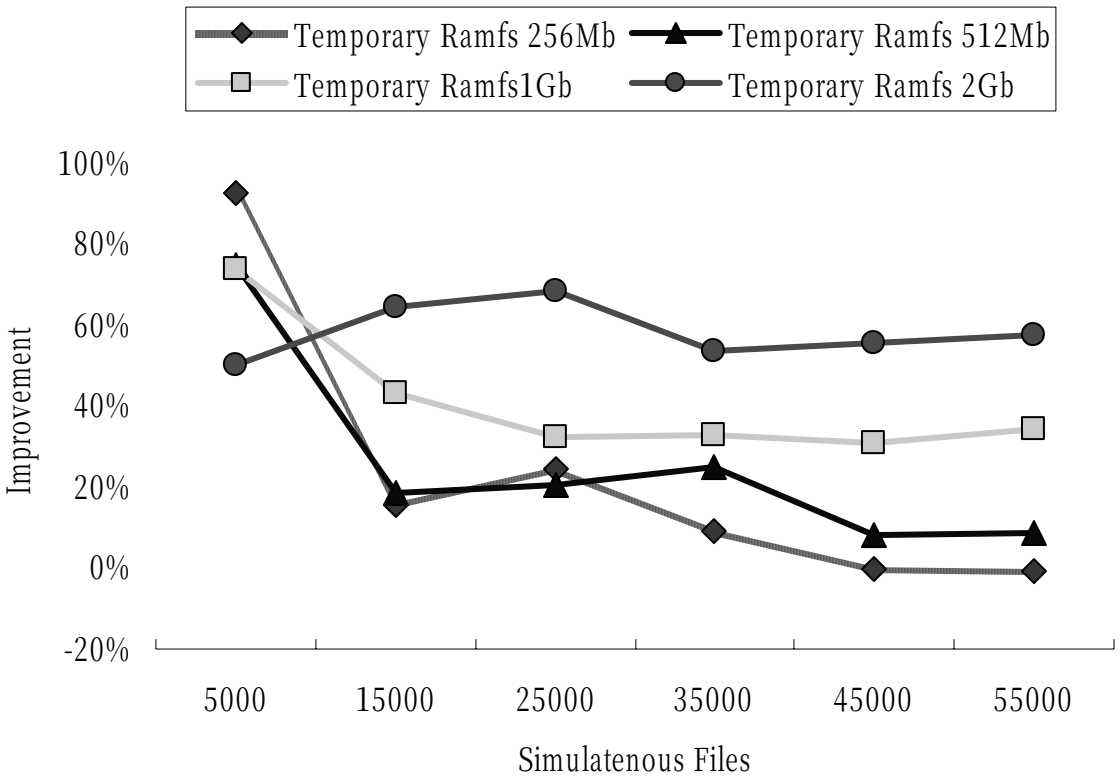


Figure 4.9 Temporary Ramfs Improvement on Different Memory Sizes

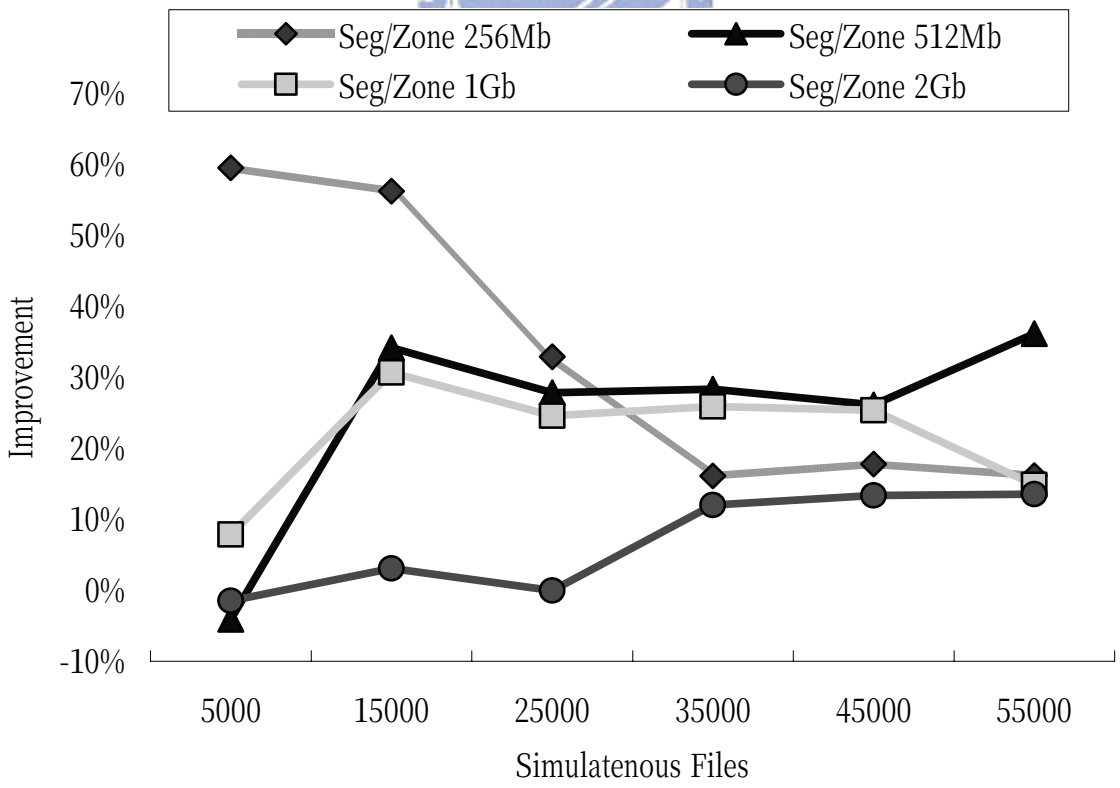


Figure 4.10 Segment/Zone Algorithm Improvement on Different Memory Sizes

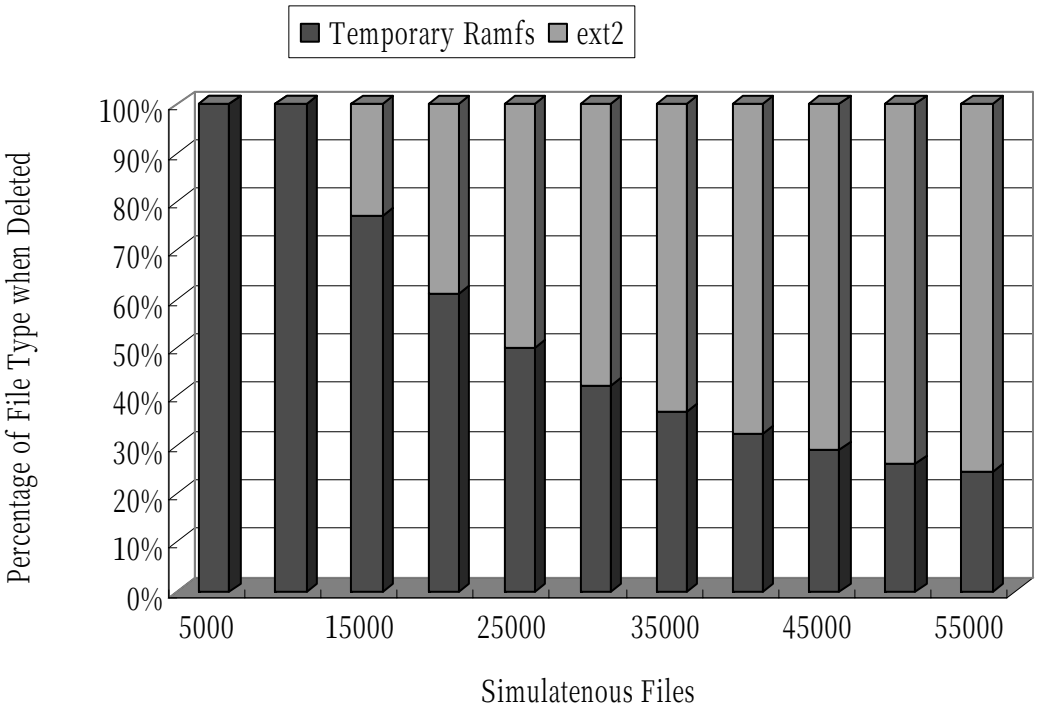


Figure 4.11 Measure the Rate of File Deleted in Temporary Ramfs with 1GByte Memory

4.5. 不同設定下的效能

4.5.1. 記憶體大小

此實驗中，我們測試不同記憶體大小時的 Temporary Ramfs 以及 Segment/Zone 寫回演算法執行 Postmark benchmark 效能之差異。實驗結果如 Figure 4.9 所示，根據實驗結果可以發現，Temporary Ramfs 在記憶體容量變大的情形下，其效能會有所提昇。原因在於對於 Temporary Ramfs 而言，若記憶體越大，則表示能暫時存放在 Temporary Ramfs 內的資料越多，並且可將檔案保留在 Temporary Ramfs 中更久的時間，也因此表示有更高的機會檔案可以利用到 Temporary Ramfs 的好處。Figure 4.11 為我們在記憶體為 1G 時測量檔案被移除時所屬的檔案系統，比較與在記憶體為 512Mbytes 時所做的實驗(如 Figure 4.5)可以看出在相同的初始檔案數下，當記憶體較大時，檔案有較大的比例在 Temporary ramfs 中被刪除，此結果證明在記憶體容量越大的情況下，效能會表現的更好。而對 Segment/Zone 寫回演算法而言，實驗結果如 Figure 4.10，當檔案個數為 5000 時除了記憶體大小為 256Mbytes 時的表現較好其餘均較差的原因主要是因為只有當記憶體大小為 256Mbytes 系統才會在檔案個數為 5000 時做寫回的動作。而記憶體為 2G 時表現則

都維持約在增進 10%左右的效能，主因是當記憶體容量越大，系統上做寫回的動作也越少，所以在相同的工作負荷下，當 I/O 量越少時，Segment/Zone 寫回演算法所能改進的效能也會相對下降。而在 256Mbytes 的情況下，當 I/O 量愈大時，效能改進會下降至約 10%，主要原因在於記憶體越小我們能夠選擇的資料也越少，因此效能增進並不如記憶體為 512Mbytes 和 1G 的情況下。

4.5.2. Zone 的大小

在 Segment/Zone 的寫回演算法中，為了降低寫回的時間，我們考慮到在寫回資料時，選取適合的 Zone，而由於 Zone 的大小也影響到寫回演算法的效能表現。因此在此實驗中，我們以分頁為單位調整 Zone 的大小，測量不同大小時寫回演算法的表現。我們使用 Postmark 預設的設定值並將 transaction 數設定為 20000，初始檔案數設訂為 45000。在此實驗中，最小的 Zone 為 32 個分頁，最大的 Zone 則為 8192 個分頁。

實驗結果如 Figure 4.12 所示，其中 X 軸表示 Zone 的大小，Y 軸則是寫回演算法的效能改進。根據結果顯示，當 Zone 的大小太小時會使改進的效能下降。主要原因在於 Zone 的大小太小會導致每次寫回時均要選取數個 Zone 才能達到系統所要求的數量。而選取過多的 Zone 除了需要額外的 CPU 時間外，也可能造成寫回時讀寫頭在 Zone 之間移動，造成額外的 I/O 負擔。

而當 Zone 的大小太大(例如：超過 2048 個分頁時)效能也會逐漸下降。其主因有二：第一，過大的 Zone 會導致演算法所選取的 Zone 較不具意義，因為過大的 Zone 會使得寫回時並沒有把寫回的資料限制在較小的一段區域。第二，由於在寫回時，我們必須將屬於 Zone 裡的分頁依照 block number 作排序，再依序寫回。而過大的 Zone 會導致我們需要花許多的時間將一個 Zone 中所有的分頁做排序，而實際寫回的數量卻遠小於所排序的分頁數(當 Zone 中分頁的數量比系統所要求寫回的數量來的多時)。因此當 Zone 的大小過大時也會導致效能的下降。

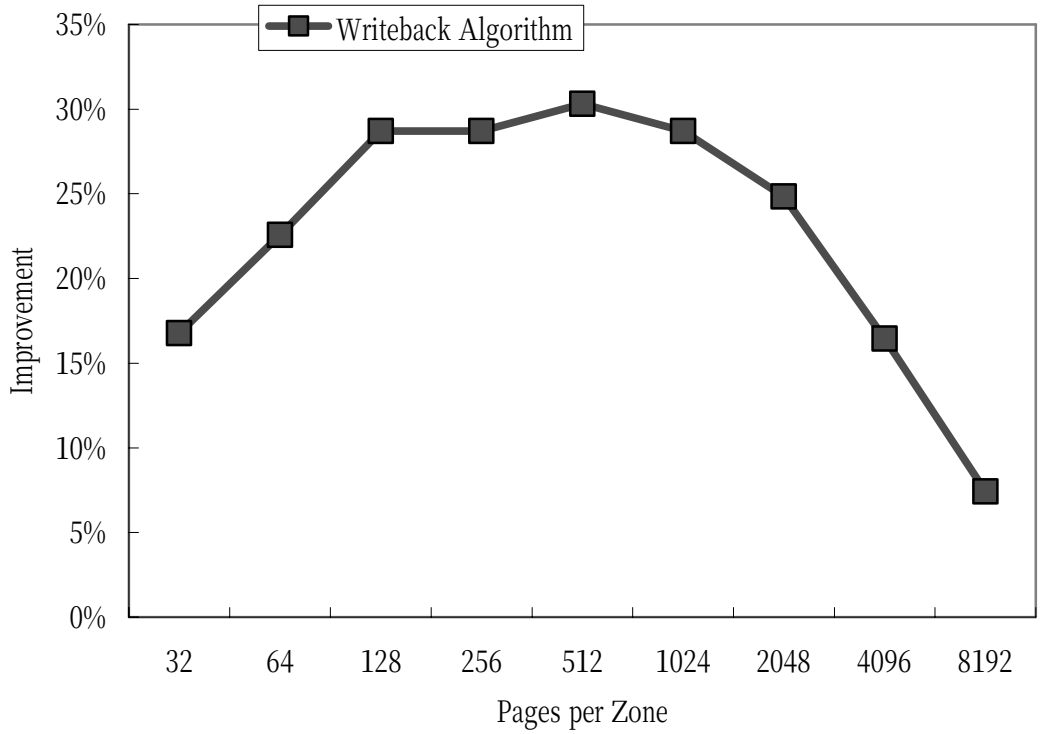


Figure 4.12 Performance of Different Zone Size

第5章 結論

利用 MRAM 非揮發性的特性，我們改良了現今的 Buffer Cache Model，增進 I/O 效能，並對於檔案系統來說是具有透明性的，因此若將來 MRAM 取代 DRAM 成為系統上的主記憶體，也不一定需要有 MRAM-based 檔案系統，就可以利用 MRAM 所帶來的好處。

而在此 MRAM-based Buffer Cache Model 中，我們利用其非揮發的特性，提出兩個改進 I/O 效能的主要機制:

- Temporary ramfs
- WASL writeback algorithm

在第一個機制 Temporary Ramfs 中，我們將寫入記憶體上的檔案視為 persistent，因此對於所有新建立的檔案，都先將其放置在 ramfs，等到確定檔案大小後再分配至磁碟上。根據實驗，在系統工作負載為處理許多小檔案時，成功的減少檔案 fragmentation 的現象並且避免不必要的 I/O，增進系統效能。

而在第二個 WASL 寫回機制中，我們利用非揮性主記憶體的特性，考慮到由於不需擔心是否具有儲存在記憶體中太久而沒被寫回的資料，我們選擇在寫入時選擇較為連續或是鄰近的資料，減少磁碟讀寫頭的移動，加快寫回所需的時間。

參考文獻

[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, Oct. 1992.

[2] P. M. Chen., “Optimizing delay in delayed-write file systems”.Technical Report CSE–TR–293–96, University Michigan, May. 1996.

[3] 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, Apr. 2001.

[4] B. Dipert. “Exotic memories, diverse approaches”. EDN Magazine, pp.56-70, Apr. 2001.

[5] 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.

[6] 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*,Jul. 2004.

[7] 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, Jan. 2002.

[8] 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, Jan. 1997.

[9] 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.

[10] 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, Feb. 1999.

[11] 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, Jun.1997.

[12] 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, Dec.2005.

[13] J. Katcher."PostMark: A New File System Benchamrk". Technical Report TR3022 Network Appliance Inc, Oct. 1997.

[14] A.Kawaguchi, S. Nishioka, and H. Motoda., "A flash memory based file system". *Proceedings of the 1995 USENIX Technical Conference*, pp. 155–164, Jan. 1995.

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

[16] 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, Oct. 2000.

[17] 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.

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

[19] R. Ohmura, N. Yamasaki, 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, Nov. 2003.

[20] D. Roselli, "Characteristics of File System Workloads".Technical Report CSD-98-1029, University of California at Berkeley, Dec. 1998.

[21] 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.

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

[23] An-I. 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, Jan. 2002.

[24] 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, Aug.2001.

[25] D. Woodhouse. “The journaling flash file system”. *Ottawa Linux Symposium*, Jul. 2001.

[26] M. Wu, 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, Oct. 1994.

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