# 國立交通大學

## 資訊科學與工程研究所

## 碩 士 論 文

# 嵌 入 式 儲 存 系 統 效 能 評 估

Benchmarking Embedded storage system

研 究 生：曾士豪

指導教授：張立平　教授

中 華 民 國　九 十 六　年 十 一 月

# 嵌入式儲存系統效能評估
# Benchmarking Embedded storage system


研 究 生：曾士豪　　Student：Shi-Hao Tzeng
指導教授：張立平　　Advisor：Li-Pin Chang

國 立 交 通 大 學
資 訊 科 學 與 工 程 研 究 所


碩 士 論 文

A Thesis
Submitted to Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Computer and Information Science

June 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年十一月

# 嵌入式儲存系統效能評估

學生：曾士豪　　指導教授：張立平

國立交通大學 資訊工程 學系（資訊科學與工程 研究所）碩士班

## 摘　　要

快閃記憶體已經成爲嵌入式計算環境的主流儲存媒體。拜嵌入式處理器效能長足進步之賜，處理大量資料已經不是桌上型電腦或者大型主機的專利了。快閃記憶體本身有著相當獨特的物理特性，因此以往設計給磁碟的檔案系統並不能直接使用在快閃記憶體上。也由於這個原因，對以磁碟爲基礎所設計的效能評比與工具，並不能反映一個針對快閃記憶體設計的檔案系統的良莠。因此，本論文著眼於如何針對以快閃記憶體爲基礎的檔案系統做一有系統的評比測試，目的是爲了呈現以往測試評比工具所不能顯現的效能差距。這樣的結果，除了可建議使用者在某些工作負荷下應該用哪一種檔案系統，更重要的是，避免選擇在該工作負荷下效能不彰的檔案系統。

關鍵字: **Filesystem Benchmark, embedded storage, flash memory, Garbage collection, Wear-leveling, JFFS2, YAFFS, NFTL**

Benchmarking Embedded Storage System

student：Shi-Hao Tzeng Advisors：Li-Pin Chang

Department（Institute of Computer Science and Engineering）of Computer Science
National Chiao Tung University

## ABSTRACT

Flash memory has become a crucial component in building embedded computing systems. With the dramatic improvement of the computing power of embedded processors, embedded computers is now capable of processing massive data. In past work, the benchmark of disk-based file systems has been widely studied. However, because of the very different physical characteristics of flash memory, these existing benchmark suites are not suitable to evaluate flash-memory-based file systems. This work is motivated by the needs for the benchmark tools for embedded file systems. Our objectives are to reveal the how efficient the existing flash-memory file systems deal with the fundamental issues of flash-memory management. The benchmark results provide users valuable information regarding not only to choose a good solution in their embedded computers but also to avoid a poorly performing approach under certain workloads.

**Key word: Filesystem Benchmark, embedded storage, flash memory, Garbage collection, Wear-leveling, JFFS2, YAFFS, NFTL**

# 誌　　謝

終於抵達寫致謝詞的這一刻，這意味著研究所生涯即將正式的畫上句點，回顧兩年來的經歷，內心充滿幸福與感謝，首先誠摯的感謝指導教授張立平老師，老師悉心的教導使我得以了解嵌入式系統的深奧，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。因為有你的體諒及幫忙，使得本論文能夠更完整而嚴謹。

二年多的求學生涯裡，讓我學習了不少新的知識和待人接物的方法，非常感謝每個老師的敦敦教悔，而且勤而不懈的教導我，讓我能夠一路走來都很順利。

再來，感謝在這段期間，游仕宏、張譽璿、黎光仁、杜俊達同學的幫忙，使得我能順利走過這兩年。實驗室的黃千庭、許辰暉、林松德、鄭家明、許惠茹…等學弟妹們，當然也不能忘記與你們相處的日子。還有各位在新竹一起奮鬥的“捧油”們，在這非常感謝小螞蟻、sfancy、老大、小傅、小卷、kayla 以及應數所的球友們和親愛的室友們，你/妳們的幫忙、陪伴及搞笑我深刻地銘記在心，由衷感謝各位的相助。謝謝！

研究口試期間，感謝羅習五老師、陳雅淑老師不辭辛勞細心審閱，不僅給予我指導，並且提供寶貴的建議，使我的論文內容可以更臻完善， 在此由衷的感謝。

感謝系上諸位老師在各學科領域的熱心指導，讓我增進各項知識範疇，在此一併致上最高謝意。

最後感謝我摯愛的雙親和家人，因為他們的支持，才有此篇論文的產生。

# 目　　錄

# 表　　目　　錄



# 圖　目　錄

# Benchmarking Embedded Storage System

Shi-Hao Tzeng and Li-Pin Chang
Department of Computer Science
National Chiao-Tung University, Hsin-Chu, Taiwan 300, ROC
sulapam@gmail.com,lpchang@cs.nctu.edu.tw

**Abstract**

快閃記憶體已經成爲嵌入式計算環境的主流儲存媒體。拜嵌入式處理器效能長足進步之賜，處理大量資料已經不是桌上型電腦或者大型主機的專利了。快閃記憶體本身有著相當獨特的物理特性，因此以往設計給磁碟的檔案系統並不能直接使用在快閃記憶體上。也由於這個原因，對以磁碟爲基礎所設計的效能評比與工具，並不能反映一個針對快閃記憶體設計的檔案系統的良莠。因此，本論文著眼於如何針對以快閃記憶體爲基礎的檔案系統做一有系統的評比測試，目的是爲了呈現以往測試評比工具所不能顯現的效能差距。這樣的結果，除了可建議使用者在某些工作負荷下應該用哪一種檔案系統，更重要的是，避免選擇在該工作負荷下效能不彰的檔案系統。

**Key word: Filesystem Benchmark, embedded storage, flash memory, Garbage collection, Wear-leveling, JFFS2, YAFFS, NFTL**

## 1. Introduction.

在嵌入式儲存系統(Embedded storage system)中，由於快閃記憶體(快閃記憶體)擁有低成本,體積小和低耗電量的好處，因此經常被作爲嵌入式系統的主要儲存體。目前常見的 NAND 快閃記憶體有 NAND 和 NOR 兩種技術，特別是 NAND 快閃記憶體有許多有更利於儲存系統的優點(低價格，高容量)，導致它在嵌入式資料儲存方面上受到重用。隨著 NAND 快閃記憶體 (NAND 快閃記憶體)的可存容量(Storage capacity)快速的發展和增加，不久的將來除了在嵌入式系統上的儲存體應用之外，也有很有可能被廣泛地應用到一般桌面 PC 上。

一般而言在現今桌上型個人電腦和筆記型電腦中，常用的儲存裝置爲磁碟爲主的儲存體(磁碟爲主 storage)，所以傳統的檔案系統的設計考量大多是因應磁碟特性來設計的，並不適合直接使用來管理快閃記憶體。主要的差別在於磁碟儲存體的資料更新動作是採取in-place updates[1]策略。由於快閃記憶體晶片中，是被劃分爲許多相同尺寸的區塊

[1] In-place updates：一種file-system處理檔案placement的策略。一般disk file-system中常見的策略。更新檔案時，file-system會直接將新的檔案data覆蓋到舊有的data儲存的physical位置上。

(Block)，而這些區塊又被切分成許多相同大小的頁(Page)。資料寫入的動作是以一個頁爲單位，被寫過的頁是不能直接再被重新寫入資料的(overwrite)，必須經過抹除(Erase[2])的動作之後，才能再次寫入資料，而抹除的動作則是以一個區塊爲處理單位，所以每次抹除的動作會消除多個頁中的資料。因此爲了避免每次更新都執行抹除的動作，在更新資料時，通常是再找另一個free page來存放更新的資料(valid data)，並將舊資料標示爲無效(invalid)，這一種資料更新策略稱之爲out-place updates。在這兩者不同的策略下，使得快閃記憶體有許多不同於磁碟的議題要考慮。快閃記憶體中還有另外一個特別限制，就是每個區塊的抹除次數是有限的。目前的技術下，一個區塊大概可承受 100K次(約$10^5$)的抹除動作，超過次數的區塊可能就無法正常地執行讀寫動作。

由於 out-place updates 的緣故，爲了管理快閃記憶體的資料儲存情形，就要額外處理和紀錄資料邏輯位址和實體位址的變化狀況，所以快閃記憶體檔案系統會建立位址轉換的機制，來掌握檔案的邏輯位置和快閃儲存體中實體位址對應的關係(logical to physical address translation)。當回收快閃記憶體中使用過的空間時，這些空間需要經過抹除之後才能再次利用，若要抹除的空間中有存放 valid data 則要將它搬移到其他的可用空間上存放，這一連串跟回收空間相關的動作稱爲垃圾回收(Garbage collection)。由於每個區塊被抹除的數量有限，爲了延長快閃記憶體的壽命，通常會希望快閃記憶體上的區塊是很平均地被抹除，這概念稱爲平均磨損(Wear-leveling)。

由於上述的原因，若將磁碟使用的 inplace-update 方式運作在快閃記憶體上是沒有效率的，因此目前磁碟檔案系統不適合直接用在快閃記憶體上，而且在磁碟檔案系統中強調的迴轉延遲(Rotational delay)以及讀寫頭性能最佳化(seek penalty optimization)對快閃記憶體來說，也是沒有太大功效的，另外在磁碟檔案系統對於平均磨損並沒有作額外的考量和支援。

由於現有檔案系統的不合適直接用來管理快閃記憶體，因此便有人設計出新的檔案系統的形式來解決快閃記憶體的檔案管理問題，主要有下列兩種。1.利用 Flash translation layer，例如 FTL 和 NFTL。好處是讓現在既有的磁碟爲主的檔案系統(ext2,FAT 等)可以直接管理快閃記憶體，只需透過一層 Flash translation layer 來處理邏輯位址和實體位址的轉換關係。2. Native flash 檔案系統，例如 YAFFS 和 JFFS2。好處是針對快閃記憶體的特性製作的檔案系統，能更有效地利用快閃記憶體上的空間和更快速地處理邏輯位址和實體位址的轉換。

不論是磁碟爲主的檔案系統或是快閃記憶體的檔案系統都可以在各種情形下良好的運作。而“哪一種檔案系統是我的系統應該使用的？”。這個問題的解答，通常都可以

[2] Erase： Flash 中特有的operation，flash是一種write once的儲存裝置，需依靠erase這個動作，來清除區塊上已寫過的資料，經過erase手續之後的區塊，區塊中的bit會回覆到全部都是 1 的狀態，並可在此區塊再次執行寫入的動作。

透過檔案系統效能評估(File-system Benchmark)評量結果的好壞來獲得。我們認爲好的效能評估方法除了可以根據使用者的工作需求(workload)和系統的資源限制來評量出適合的檔案系統之外，更應該進一步告知使用者應該避免使用的檔案系統有哪些。由於現有的檔案系統評估方式(File-system benchmark)都是用於量測磁碟爲主的儲存系統，大多是針對讀取的回覆時間和磁碟的動作時間來分析效能，並沒有針對快閃記憶體儲存系統特性(位址轉換，垃圾回收和平均磨損)的評估方法。除了沒有適合快閃記憶體評量結果的數據之外，也缺乏告知使用者，"哪一種檔案系統適合我們自己的系統？”以及”我們的系統應該避免使用那一種檔案系統？”等相關資訊。

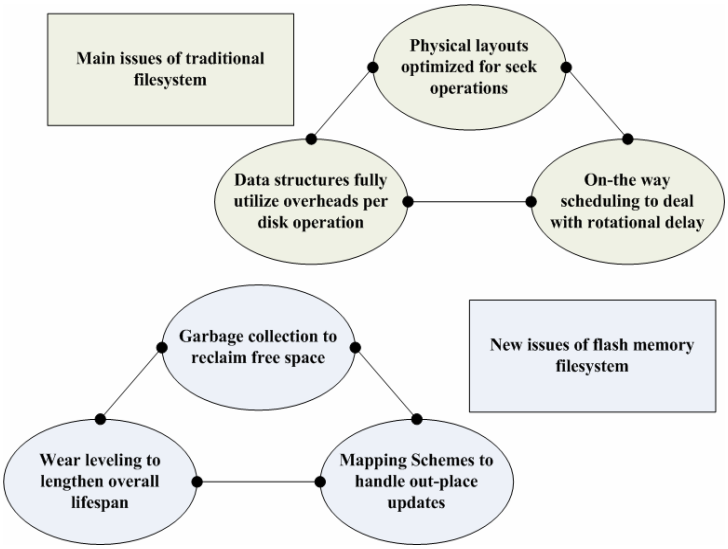
在這份研究中，我們提出一些嵌入式儲存系統特有的議題，包括上述提到的快閃記憶體特性位址轉換,垃圾回收以及平均磨損等，對於以上的議題在檔案系統的效能評估上的影響作深入地探討。我們在之後的章節中會利用兩個主要的實驗方式：黑盒子(Black-Box)和白盒子(White-Box)。在黑盒子的實驗方法中，我們就是利用常用的檔案系統效能評估方法(Postmark)或是模擬日常普遍的檔案分佈和運作模式來測量。相反地，白盒子的測試方式中，除了基本的循序讀寫之外，我們也會透過觀察和了解嵌入式檔案系統的行爲之後，在針對各個快閃記憶體檔案系統的優缺點去設計不同的實驗樣本。觀察分析的數據，除了評量檔案系統中常見的 CPU 效能,RAM 效能以及反應速度的快慢和儲存空間的利用度之外，還會分析快閃記憶體特有的數值，如區塊壽命的磨損快慢程度和垃圾回收的效能好壞等。更進一步，希望能透過實驗的結果，藉此說明在哪些特殊情形限制下應該避免使用哪一種的嵌入式儲存系統。

本論文的架構組織如下：第二章摘要了一些過去針對檔案系統和 I/O 系統的效能評估方式與工具作一個概括性的整理。第三章敘述兩種快閃記憶體 NAND 和 NOR 的特性和快閃記憶體檔案系統主要的議題。第四章主要是敘述每項議題的目的、重要性以及所要評比的重點爲何。第五章爲實驗的環境設定、方法以及實驗的結果和討論。第六章爲本論文的結論。

## 2. Motivation & Related work

由於快閃儲存體被大量的使用於 3C商品上或是工業的應用上，因此如何位自己的作業系統選擇一個合適的嵌入式儲存系統，也是許多人煩惱的課題之一。雖然目前檔案系統的效能評估方法有許多種，但普遍都是針對磁碟儲存系統設計的。雖然也有人使用其中的一些方法來評量快閃記憶體檔案系統的效能，但大多是止於讀寫的反應時間和掛載時間的測試。但磁碟型儲存體和快閃記憶體的物理架構不同，也導致這兩大類型種儲存系統的設計走向的不同(如圖表 1)。除了一些原本的測試項目之外，更應該著重觀察快閃記憶體相關的數值表現，例如：抹除次數,垃圾回收的效率,空間利用度等。或是更進一步的配合嵌入式系統的需求，去研究系統耗電量,記憶體使用量,系統穩定度等和嵌入式應用先關的項目。

因此我們便想進行對於嵌入式儲存系統的效能評估，去設計一些需要觀察的項目和可行的辦法。




圖表 1.Difference between traditional File-System and flash File-System

目前常見的磁碟型檔案系統的評量方法，可以分爲下列 3 大類[19]：

- Macro-benchmarks：利用一些特別的工作量去代表一些現實常見的工作量進行測試，進而表現出各個檔案系統的優點和缺點。
- Micro-benchmarks：只著重於某一部分的檔案操作指令，去呈現出各個檔案系統在一些特別觀點的表現。
- Trace replays：利用程式去重新表現出真實系統所紀錄下來的操作指令，進而得到更貼近現實的工作量。

依照上述分類的方式我們可以將一些經常被使用的檔案系統評估方式都分成這 3 大種。

首先看到屬於 Macro-benchmarks 的量測方式有 Andrew benchmark [14,20] , Postmark [17], AIM7[21],,和 IOzone [18]。Andrew benchmark 主要是模擬使用者日常生活會使用的指令建構出一連串的執行步驟，並測量這些動作在待測的檔案系統運作時的各種數據。 而 PostMark 是模擬使用者日常使用網路的習慣，例如收發 email,觀看網路新聞和其他瀏覽網頁的行爲。PostMark 會在待測的檔案系統中自行創造出一些類似 cookie 的小型檔案以及一些類似增刪網頁暫存檔的動作來達到效果，並紀錄過程中檔案讀寫的次數和總量，以及花費的時間，經常被拿來當作檔案系統的評比工具之一[22,23,24,25]。最後 IOzone 是可以產生和量測許多種的指令動作，可以觀察各種動作在檔案系統中花費的時間，比較著重於觀察 I/O

的效能表現[25]。AIM7 會產生許多平行執行的程序，每一個程序都執行一些動作和隨機的呼叫內建的工作內容，包含一下系統指令(磁碟讀/寫)動作，pipe I/O，以及一些消磨 cpu 時間的迴圈工作)。之後計算出每分鐘完成的工作個數，在[25]中被使用爲評比工具之一。

其他的 Macro-Benchmark 的測量工具還有 Netbench[29]和 Dbench [30]。NetBench 是一種量測檔案伺服器的評比工具，主要利用多台 PCs 之間的網路連接，來模擬使用者向伺服器請求檔案的 I/O 的要求。Dbench 是一個作用於 LINUX 上的 open-source 程式，和 NetBench 相同是評測檔案伺服器的工具，Dbench 所產生的工作量就像是 NetBench 作用於不需要發出網路請求的 Samba server 上一樣。這兩個評比檔案伺服的工具在[35]中被使用到。

Micro-benchmarks 中較具代表性的量測方式就是 Bonnie and Bonnie++[16]，主要是測試 I/O 輸出的產能，會製造一些 page cache 無法完全吸收的大檔案來量測檔案系統在 I/O 表現上的極限，盡可能快速地去讀寫這些大檔案。其他的 Micro-benchmark 方法還有 Sprite LFS[31]，Sprite LFS 的測試方式有兩種 Sprite LFS Large File Benchmark 和 Sprite LFS Small File Benchmark。Sprite LFS Large File Benchmark 被使用於[32,33,34]作爲測試效能的工具，主要的步驟示對於容量爲 100MB 的大檔案作循序和隨機的讀寫運算(讀取和寫入的資料量大小都是 100MB)。Sprite LFS Small File Benchmark 在[32,33,34]被使用到，主要的實驗步驟是先建立 10000 個以上 1KB 的檔案，再利用 flush 寫回 page cache 中暫存的資料後，再進行讀取全部檔案的動作，最後再刪除所有檔案。

Trace replays 這一類型的測量方法，主要都是使用者自行撰寫模擬程式去分析和重現，他們自己收集到的紀錄檔(trace log)，而這些程式也都會依照使用者自己的需求和各自紀錄的紀錄檔不同而有不同的部份。如[13]中作者利用自行撰寫的程式分析收集的 trace log。

Iometer[26]不屬於上述 3 種類型的量測工具，可以透過使用者設定而模擬任意的磁碟和網路 I/O 動作，所以 Iometer 既可以當作工作量產生器(workload generator)來使用，也可以當成一種量測的工具。在[25]中被作者自訂成自己的量測工具 pmeter。

但上面提到的檔案系統測量方法大多數主要還是用於測量磁碟爲主的檔案系統，而非針對快閃記憶體去量身打造的。在嵌入式儲存系統上，評測工具非常少見。[8,9]中作者提出量測評量嵌入式系統中處理器和快閃記憶體裝置的個別的耗電量，利用一個 macro-model 來量測 CRAMF,RamFS,JFFS2 的耗電量，且不需要任何實際量測電力的工具[8,9]。

接下來我們會先介紹快閃記憶體檔案系統和磁碟檔案系統不同的幾個主要

的議題，並探討在之後的量測方法中應該注重的部份爲何，以及爲什麼要量測這些數據。

# 3. Fundamental Management Issues of Flash-Memory Storage Systems

在這個章節，我們在 3.1 首先介紹快閃記憶體的物理架構，然後在 3.2 介紹目前快閃記憶體在儲存系統中所佔的層次爲何，接著 3.3 至 3.5 則介紹快閃記憶體 3 個主要管理議題：位址轉換，垃圾回收和平均磨損。

## 3.1 Flash-Memory Physical Geometry

由於快閃記憶體有非揮發性(non-volatile)，抗震(shock-resistant)，和 省電(power-economic)優點，是被經常用於充當嵌入式系統(embedded system)中的儲存體。目前市場上主要有兩種不同的快閃記憶體: NAND 和 NOR 快閃記憶體。這一章節中主要著重在介紹 NAND 快閃記憶體的概觀和硬體特性。

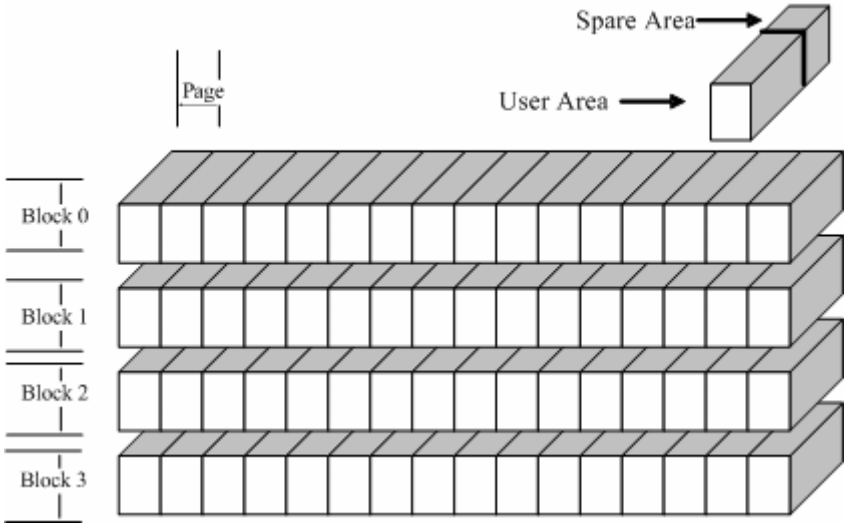
NAND快閃記憶體主要就是設計作爲儲存資料之用，而NOR 快閃記憶體主要用來取代EEPROM的。由於NOR是採取標準的記憶體介面,所以CPU處理器可以直接執行儲存再NOR上的程式(亦可稱爲eXecute-In-Place, XIP)。表格 1.爲比較NAND快閃記憶體和NOR快閃記憶體主要的差異[3]。因爲NAND快閃記憶體更適合作爲主要的儲存系統，所以之後的討論我們會將焦點集中放在NAND快閃記憶體的議題上。

表格 1. NAND 快閃記憶體 V.S. NOR 快閃記憶體

| | NAND | NOR |
|---|---|---|
| Density | High | Low |
| Read/write | Page-oriented | Bitwise |
| eXecute In Place (XIP) | No | Yes |
| Read/write/erase | Moderate/fast/fast | Very fast/slow/very slow |
| Cost per bit | Low | High |

NAND 快閃記憶體由於便宜，體積小，耗電量小，以及其可媲美硬碟容量的儲存空間，因此被廣泛地應用在各式常見的 3C 商品上作爲主要的儲存空間。因此用於管理 NAND 快閃記憶體上空間應用的檔案系統也開始顯得格外重要。但因爲 NAND 快閃記憶體擁有許多不同於一般磁碟的性質，使得目前桌面型電腦上所使用的檔案系統不適合在快閃記憶體上運作。而快閃記憶體共通的特別之

處在於 1.一次寫入(Write-Once)，2.大量抹除(Bulk erase)，3.有限的抹除次數(limit erase cycles)。




圖表 2. The structure of Flash Memory

首先介紹快閃記憶體的基本架構，如圖表 2所顯示。每一個快閃記憶體晶片主要是由許多同等大小的區塊所組成，而每一個區塊又是由許多個相同大小的頁所構成。每個頁又分爲兩個區域分別是使用者區域(user area,通常爲 512bytes)和備用區域(spare area,通常爲 16Bytes)，其中使用者區域用來是擺放使用者儲存的一般資料的地方，而備用空間是留給檔案系統擺放管理系統所需的Metadata[3]所使用的。一般快閃記憶體中執行抹除動作的單位爲區塊，而讀/寫動作則是以頁爲單位。在快閃記憶體中，我們通常將存有最新資料的頁稱爲live page(或 clean page)，反之儲存舊資料的頁則稱爲dead page(或dirty page)。一般的情形下，同一份資料在快閃記憶體中應該會有許多份的dead page和一份最新的live page，而這一份擁有valid data的live page位址會因爲更新的狀況不斷的改變，不一定會固定儲存在快閃記憶體的固定的位置。

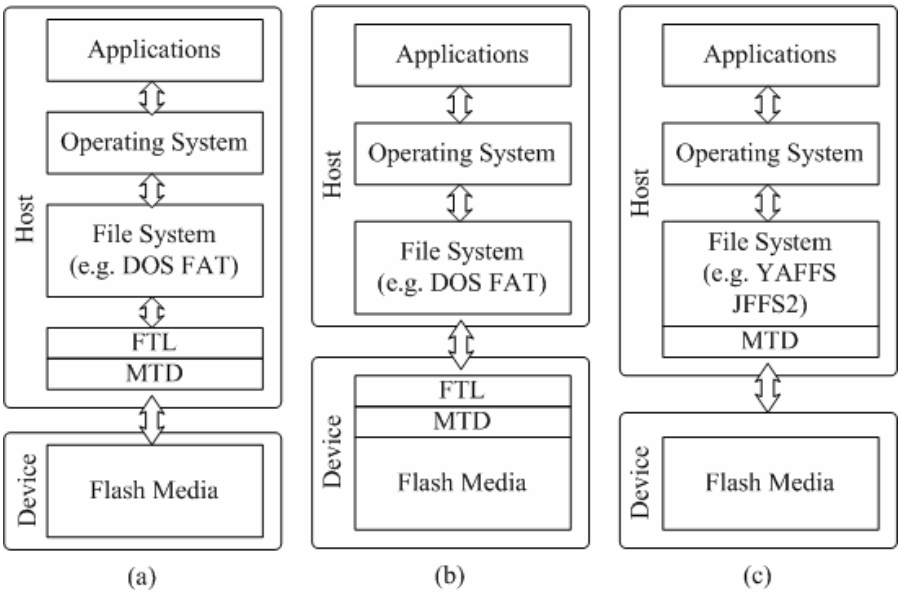
## 3.2 Flash Memory in Storage Hierarchy

一般使用快閃記憶體作爲非揮發性的儲存媒介(non-volatile storage media)的方法有 3 種，如圖表 3 所示。1. Host driver emulate: Smart Media™(SM)[5], MemoryStick™(MS)，由於內部不含控制晶片，必須搭配特定的控制器(driver controller)才能運作，泛用性和系統支援性較低。(如圖 3.a)2. Standalone block device:在NAND快閃儲存裝置的本身內部就包含了控制晶片，例如 CompactFlash™(CF)卡[6], Secure Digital Memory Card (SD)卡,USB thumb drives 和 M-system Disk On Module(DOC) (如圖 3.b)[7]。3. Native Flash file system：

[3] Metadata：檔案系統中用來處理和管理檔案資訊的資料，也能說成是資料的資料(data of data)。

JFFS1, JFFS2/JFFS3, YAFFS1/YAFFS2, MFFS[10,12,15]等，特別爲NAND快閃記憶體量身訂做的檔案系統。其中Native flash file system的各種特性便是這篇研究要探討的重點所在。(如圖 3.c)

Native file system 抓取檔案是實體位址的方式，是利用檔案名稱(file ID) 和偏移量(offset)，再透過檔案系統的資料結構轉換得知實體資料在快閃記憶體上的實體位址(physical address)。而 block device emulation 則由原本所使用之磁碟檔案系統(Disk-based file system)獲知檔案名稱和偏移量，得到儲存資料所在區段數(sector number)後，再把這個區段數經由 NFTL 和 FTL 等的 block device emulation 轉換和對應下得知資料在快閃儲存體上得實體位址。




圖表 3. 三種不同的 NAND Flash Memory 使用方法 (a) Host driver emulate，(b) Standalone block device，(c) Native Flash file system

## 3.3 Address translation

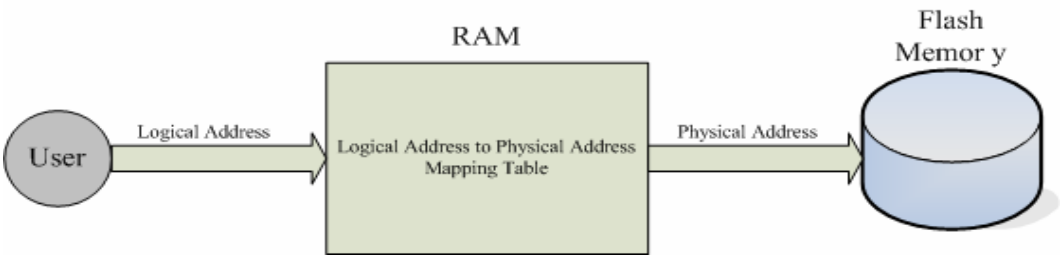
爲什麼在快閃記憶體檔案系統中需要另外再做一層邏輯位址到實體位址的轉換呢？原因有二：1.不想每次寫入之前，都必須做抹除。2.Hot Data 都有區域性(locality)。若重複讀寫在同一塊區塊中，會快速耗損快閃記憶體壽命。兩個原因主要都是受限於 NAND 快閃記憶體的物理行爲。

原因一，是由於在 NAND 快閃記憶體特殊的物理性質(參考 **3.1 Flash-Memory Physical Geometry**)，導致快閃記憶體再更新資料時不能採取一般磁碟的常見處理方式(In-place)。假若要仿造磁碟爲主的檔案系統(disk-based filesystem)處理資料擺放的方式(data placement)，則必須在執行更新資料的寫入

動作之前先將已寫過的 dirty 區塊作抹除動作，而處理抹除的時間比一般讀/寫(unit: page)時間還要長，並且在動作期間 NAND 快閃記憶體 device 無法並行執行其他動作，倘若每次要對 NAND-快閃記憶體作資料寫入的動作時，都需要先強迫浪費一段時間要處理抹除動作，迫使執行跟更新(Update)有關的寫入動作時就需要花費大量時間(erase time+ write time)，因此會導致檔案系統的產能(throughput)降低，而拖慢檔案系統整體的運作速度。

原因二，主要是由於資料的從讀寫的頻率上來看是有Hot data與Cold data之分[4]的，然而在一般的磁碟爲主檔案系統中Hot data都具有相當高的區域性(locality)。經常性更改的資料通常都會群聚在一起，作用在快閃記憶體之上之後，由於這些資料經常性的更動，會不停的將資料更新到快閃記憶體上。因此隨著資料經常性的更新，便必須不斷地對儲存過這些hot data的區塊執行抹除回收再複寫新資料的動作，長久下來會導致這幾塊區塊的壽命快速的被耗損，最後成爲無法使用的bad block。然後不斷重複相同的動作，使得NAND快閃記憶體中的可用空間不斷減少，惡性循環之下也加快耗損整個快閃記憶體的壽命。

由於以上兩種 NAND 快閃記憶體的運作缺點，使得快閃記憶體檔案系統在資料的更新寫入時，爲了避免抹除動作產生的時間和額外的負擔，所以在資料的更新上是採取不同於 in-place updates 的策略。快閃記憶體爲主的檔案系統在處理檔案的更新時，是直接將寫入資料放置在與原本不同的頁面中，來避免每次更動資料就必須抹除的動作。以上這種將資料寫入非原始的資料位址的方法，稱之爲“Out-place”。但這個時候就必須要作位址轉換的處理，否則 read/write 相同檔案時會發生錯誤(由於檔案在快閃記憶體上的實體位址已改變)。因此快閃記憶體檔案系統通常必須在 RAM 中額外建立一層位址轉換的機制來處理在使用者眼中檔案的邏輯位址(Logical address)和檔案實際對應到快閃記憶體上的實體位址(Physical address)之間的映成關係(圖表 4.)，避免檔案 read/write 時會發生錯誤。

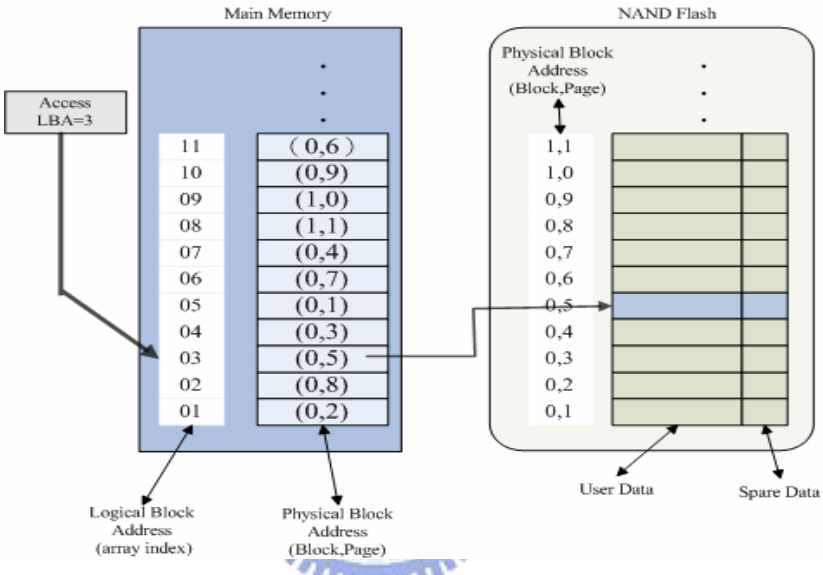



圖表 4. 利用 RAM 中建立的 Mapping table 處理 Logical Address 和 Physical Address 之間的轉換

所以爲了要方便去處理和實作Out-place，快閃記憶體檔案系統是利用上述作

[4] Hot Data/ Clod Data：對儲存於storage中的data，利用讀取的頻率的高低不同所作出的區分。Hot data 是指經常被系統update或是user 更改的data，一種經常不斷的在更動的data，反之稱爲Cold data。

一層位址轉換(address translation)的方式。利用各自定義的資料結構儲存額外的資訊(Metadata)，進而利用這些Metadata在主記憶體中建造處理位址轉換的對應表。例如：Flash Translation Layer (FTL)便是利用一個位址轉換表(address translation)來進行logical block address(LBA) 和 physical block address(PBA)的轉換(如 圖表 5 **FTL address translation** )。先是在主記憶體的位址轉換表中讀取LBA(3)的數值發現是對應到PBA(0,5)，即第 0 個區塊中的第 5 個頁面，因此可知在LBA(3)的這個頁面中的資料，是在實體快閃記憶體上第 0 個區塊中的第 5 個page上作讀取或寫入。然而各個快閃記憶體 檔案系統都有必須要有各自處理位址轉換的資料結構和位址轉換表的機制，會是快閃記憶體檔案系統的主要核心課題之一。



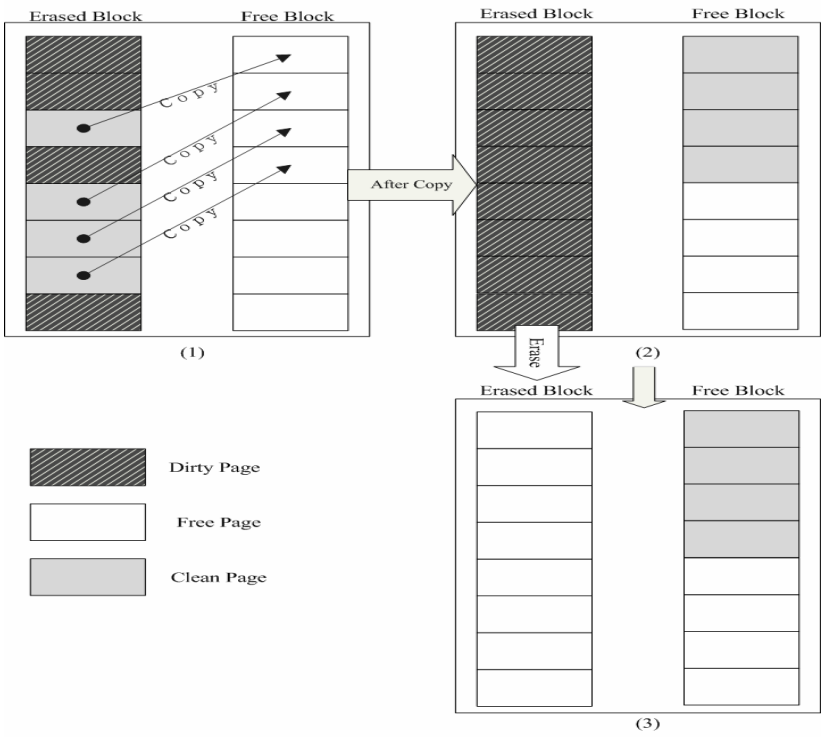

圖表 5. **FTL address translation**

然而，位址轉換雖然解決了實作Out-place的主要問題，但也引發了不少值得深入探討的議題。首先由圖表 5中FTL可看出每一個主記憶體中的頁面也是對應到快閃記憶體中的一個頁面，表示即使當使用者只是對檔案作了一點點的更改，依然是要快閃記憶體重新寫入一個頁面，但可能是整個頁面資料更變小部份後再寫入，或是只寫入那更動的資料(其他剩餘空間則填空浪費)，所以在這邊可以知道位址轉換的對應階層(mapping level)(NFTL和YAFFS爲page-Level，JFFS2 爲 node-Level較特別)會影響到快閃記憶體 檔案系統處理微小更動產生的read/write和回覆時間(response time)。

## 3.4 Garbage Collection (GC)

由於快閃記憶體 write-once 的特性和快閃記憶體 檔案系統中採用 Out-place 的方式，而 invalid (dirty) 區塊必須藉由抹除動作來還原成可用的空間，其中由於抹除的單位和 read/write 單位不同，會導致需要抹除的區塊中會有 valid(clean) page 的存在，必須將這些頁面中有用的頁面保存搬移到其他的區塊上之後，再

進行抹除區塊的動作，而這一連串步驟和觀念，稱之爲垃圾回收(Garbage collection。)




圖表 6. The Example of Garbage Collection

垃圾回收在一般快閃記憶體檔案系統上的處理動作，大致上的步驟如圖表 6的範例表示的。首先要被抹除的Erased block上的含有有效資料(valid data)的頁面(page)一個個都拷貝到其他可用頁面(free pages)上(圖表 6中的(1))。再將被拷貝過的頁面通通標記成無效頁面(dirty page)(圖表 6的(2))，最後對這個全是無效頁面(dirty page)的區塊進行抹除，完成回收這個區塊空間 (圖表 6的(3))。從圖 6. 的範例可看出，這個垃圾回收動作實際所回收的可用空間是 4 個頁面，所以回收的效益是 50%(抹除一個包含 8 個頁面的區塊，但實際只多得到 4 個可用頁面的空間)。

其中整個垃圾回收動作中最主要的額外負擔(overhead)，就是上述範例中的拷貝的動作，可看出垃圾回收的動作花費的時間長短和需要被拷貝的頁面數目或有效資料(valid data)量有關，當上述步驟裡要被抹除的區塊中需要被搬移的頁面數越少，表示垃圾回收中額外花費在執行拷貝的時間相對的越少，那麼整體垃圾回收的動作越快。因此選擇一個”較適當的區塊”來執行垃圾回收(期望拷貝次數低，回收空間大)，可以大大降低花費在垃圾回收的時間和提高垃圾回收的效益。

由於檔案系統中的資料種類一般區分爲Hot data和 Cold data[4]。其中hot data由於更新資料的頻率相當高，因此假如垃圾回收所要抹除的區塊中包含hot data

較多，則會發生區塊抹除過後，被拷貝的頁面馬上又變成dirty page的情形，也會導致剛剛所作的拷貝動作是無意義的和多餘的，而且也可能馬上又造成回收區塊的可用空間又耗盡，馬上又需要再作一次垃圾回收。如此惡性循環下，會導致系統花費許多時間不停地在作垃圾回收，而不是去處理使用者存取請求(r/w request)。最後造成系統不斷地由於垃圾回收去抹除區塊，除了系統效能低落之外，也會使得NAND快閃記憶體壽命快速地消耗殆盡。

因此如何做到垃圾回收執行時，做好hot/cold data的區分，盡量避免垃圾回收去拷貝到hot data之page，來這種減少無意義的拷貝發生，也是垃圾回收演算法中一個相當重要的考量和議題[11]。由於現今的快閃記憶體檔案系統中的垃圾回收動作，大部分是採取貪婪策略(Greedy policy)[5]的作法，雖然回收空間的效益是最高的，但卻沒有考量到挑選出的區塊中所要搬移的頁面是否被hot data所盤據。雖然在短期內從效益看起來是最佳的作法，但長期下來就極有可能會發生我們上述的最壞情形。當不斷地挑選到Hot data爲多數的區塊來抹除，並持續地惡性循環下，系統只能不斷地執行垃圾回收動作，無暇處理使用者的命令。之後的章節也會考量這些情況的發生，並分析各個快閃記憶體檔案系統的應對能力如何。

## 3.5 Wear-Leveling

由於快閃記憶體中每一個區塊都有抹除的次數限制。若某個區塊的抹除的次數超過限制，則此區塊則會標示成 bad block，表示這個區塊已損壞且不能再使用了。過多的 bad block 會導致快閃記憶體檔案系統運作困難，甚至也可能無法再正常運行，表示那塊快閃記憶體無法再使用。因此如何去平衡各個區塊的抹除次數，達到充分的利用到整個快閃記憶體空間的方法和觀念，稱之爲平均抹除(Wear-leveling)。

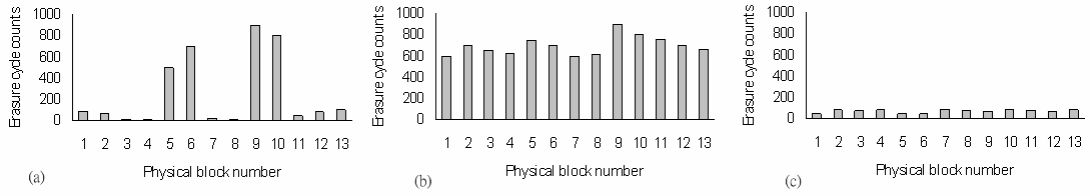
由於快閃記憶體的區塊都有有限的 erase cycles 限制。一般而言，一個 NAND 快閃記憶體的區塊的 erase cycle 約在 100K 次左右[2]，表示當區塊被抹除超過 100K 次之後便可能會不能再被抹除，導致區塊無法再被垃圾回收回收使用，形成 bad block。而當一個快閃記憶體中 bad block 越多，會導致檔案系統中可用的區塊越少，因此可供分擔因 Hot Data 流竄造成的負擔的區塊越少，更加速了剩餘區塊被抹除的速度，最後當 bad block 的數量到達一定的程度之後，會使得檔案系統無法再正常運作下去，即使系統中顯示有不少可用空間。在快閃記憶體 檔案系統中經常是造成 dirty page 的主要原因。然而，一般而言快閃記憶體檔案系統的垃圾回收策略都是採取貪婪(Greedy)，表示 dirty page 越多的區塊通常較受到這垃圾回收策略的喜愛(可回收的空間大，且拷貝搬移的次數少，花費的時間

[5] Greedy policy: 挑選dirty程度最高的區塊，換言之就是所需拷貝動作最少的區塊

短，垃圾回收所得的效益高)，反之擁有許多 Cold data 的區塊通常不會受到這類垃圾回收策略的青睞(可回收的空間小，需拷貝搬移的次數多，花費時間長，垃圾回收所得效益低)。乍看之下，使用貪婪(greedy)法則的垃圾回收似乎能得到最大效益(花費時間短，回收空間大)。

但若將檔案系統中常見的 hot/cold data 分佈的情形考慮進去的話[3]，一般情形之下，是由 Cold data 佔據整個檔案系統的大部分的空間，hot data 只佔整體 data 量的少部份。由於 Cold data 極少更動，導致垃圾回收在挑選犧牲者區塊時，會經常去挑選到由於 Hot data 變更所製造出來的區塊(因爲包含 dirty page 的成份通常較高，回收空間的效益較好)，最後可能導致 Hot Data 只在一小部份的區塊中流竄，使得這些區塊的抹除次數不斷增加，相對的其他被 Cold data 佔據的區塊則鮮少被抹除(垃圾回收不喜愛挑選這類的區快來回收，需要拷貝搬移的頁面數量太多)，造成快閃記憶體中整體區塊的各區塊的抹除次數極不平均。最後不斷地惡化下去，迫使這些區塊不斷執行垃圾回收動作，導致他們的壽命(erase cycle)快速地被消耗殆盡，最終成爲不能使用的 Bad Blocks。由於這些 bad block 已無法再利用，所以原本在這些 bad block 中流竄的 Hot data，只好移動到剩餘可供 read/write/erase 的區塊上，開始另一次類似上述動作的循環。如此下去也就加快 flash 中其他區塊的使用壽命，直到整塊 NAND 快閃記憶體不能再被使用爲止[4]。

圖表 7 中所表示的就是使用平均磨損演算法後常見的 3 種結果圖表 7(a), 圖表 7 (b)和圖表 7 (c)分別是代表沒有使用任何的平均磨損演算法,使用額外負擔(overhead)[6]高的平均磨損演算法,以及使用良好的平均磨損演算法的 3 種現象。當然由圖表 7 (b)上可以得知不好的平均磨損演算法，雖然能使達到區塊平均磨損，但額外付出的代價卻相當的高。例如圖表 7 (b)中表示的，使用演算法後整體區塊的抹除次數全都一起提高，額外的抹除動作多出許多，反而消耗更多的抹除次數(erase cycle count)。所以在額外負擔很大的影響之下，利用來延長快閃記憶體壽命的平均磨損演算法，反而加速耗損快閃記憶體的使用壽命，最後的結果會比沒有使用平均磨損演算法的圖表 7 (a)更糟糕。由此可以平均磨損最大的目標是”利用盡可能少的額外負擔(overhead)，去達到flash中所有區塊的磨損平衡。”




圖表 7. 各種 Wear-leveling Algorithm 產生的結果(a)未使用 wear leveling 或者 wear leveling 無

[6] Overhead:在垃圾回收中，主要的額外負擔是爲了搬移有用資料所花費頁面拷貝的次數。而在平均抹除中，主要的額外負擔是用來平均各區快抹除次數所執行額外的讀/寫/抹除產生的，其中最看重的是抹除次數。

效，(b) wear leveling 帶進過多的成本，(c)期望之 wear leveling 效果。

# 4. Flash-Memory-Based File-System Benchmarks

在之後的部份，我們會討論我們所觀察到的 3 個主要的快閃記憶體檔案系統(JFFS2,YAFFS 和 FAT32+NFTL)運作概念，並逐一解說他在各個主要議題的實作方式，以及獨特的地方。我們會先針對幾個評量的主題作個說明和解釋實驗的精神和目的，除了主要的概念之外，也會分析各檔案系統的設計優缺點和之後觀察評量的要點項目(如：時間，抹除次數，記憶體使用量等)，之後會再利用這些項目去設計實驗的著重部份。

## 4.1 Flash-Space Utilization

這邊要先深入探討的是空間使用率(space utilization)，這一部份討論的重點在於 Flash 的空間是否有被檔案系統充分的利用。換言之，也就是看各種快閃記憶體檔案系統利用空間的效益如何，消費者(使用者)關係的問題也不外乎爲”同樣的 NAND 快閃記憶體可用空間之下，誰能儲存的使用者資料量最多？”，還有”儲存相同的檔案資料，哪個檔案系統最節省使用空間?” 以及”爲什麼明明還有可用空間，但卻無法再寫入任何的資料？” 上述 3 個問題，主要都和快閃憶體檔案系統設計用來儲存在 NAND 快閃記憶體上的資料結構有絕大部分關係，另外快閃記憶體檔案系統在處理位址轉換時的對映階層關系也對空間使用率有很大的影響。

### 4.1.1 Performance issues

本章節討論會對 flash-space utilization 效能上造成差距的因素。基於這些因素，我們後續章節會提出對應的效能評比測試方法。

#### 4.1.1.1 User data & Metadata of File system

首先我們要先了解到有哪些東西是快閃記憶體檔案系統必須要儲存在 flash 上。第一種不用考慮一定是使用者資料，使用者資料也就是使用者主要想儲存於 NAND 快閃記憶體上的資料，也是使用儲存裝置的主因。另外就是快閃記憶體型檔案系統儲存用來紀錄和管理整個檔案系統的運作時所需的 metadata。這些儲存於 NAND 快閃記憶體上的 metadata 除了用來紀錄使用者眼中檔案的邏輯位址(logical address)和實際儲存在 NAND 快閃記憶體中的實體位址(physical address)兩者之間的對映關係之外，還能用來協助快閃記憶體裝置在開機(boot)階段和檔

案系統掛載(mount)與卸載(umount)的過程中位址轉換對應表(address translation mapping table)的建立以及主要位址對應(address mapping)的方式。因此除了需要紀錄的資料不少之外，再加上快閃記憶體 write-once 的限制和更新方式使用 out-place 的策略，讓快閃記憶體形成一種特別的特例：***"每一份使用者資料都需要夾帶著一份 metadata 來紀錄新舊。"***因爲無法使用磁碟爲主的檔案系統，對於同一份檔案只使用一份 metadata 來紀錄，所以每變更一次檔案便要再多一份新的 metadata 來辨別新舊，使得快閃記憶體檔案系統的整體使用的 metadata 比一般傳統檔案系統來的多。雖然已經提供了備用空間(Spare area)供檔案系統來儲存 metadata，但還是有快閃記憶體檔案系統會使用到使用者空間(User space)來存放 metadata(ex: JFFS2)。

在檔案系統建立初期，或是裝置區塊(device block)被以這些檔案系統格式掛載時，需要讀取整塊快閃記憶體中 metadata 的資料，以便能利用這些資訊來建立檔案系統的目錄結構和檔案之間整體的樹狀架構，並還能進一步的在主記憶體中建立檔案和 NAND 快閃記憶體上實體位址之間的映成關係。因此檔案系統中所使用的 metadata 設計上的優劣，和檔案系統建構步驟的快慢，還有佔用的系統資源，例如記憶體和 NAND 快閃記憶體空間，以及檔案系統運作時處理程序的反應快慢有密不可分的關係。基於每個快閃記憶體型檔案系統設計的原理不同，所擁有的 metadata 量大小也就不同。一般而言，在檔案系統最理想的情形，是只利用少許的 metadata，不佔用太多使用者空間，還能快速有效的管理系統和正確處理檔案命令。在之後的實驗中，我們也會去探討這三種檔案系統使用的 metadata 和使用者資料之間比重關係，以及各種變數相同的條件之下各檔案系統儲存資料量和剩餘空間的比率關係。

### 4.1.1.2 Fragment

什麼是 Fragment? Fragment 是儲存空間中的零碎無用的部份。在 NAND 快閃記憶體上形成的原因有幾種情形，可能是剩餘空間太小而不能使用導致被捨棄的空間，或者是爲了達到資料的末端的 word alignment 效果強行填入的少量且無用的資料，亦或是由於寫入的資料量小於一個頁面大小所浪費的空間。然而由於 fragment 既不是儲存使用者資料，也不是儲存檔案系統的 metadata，所以一個檔案系統運行時產生的 fragment 所佔的空間越多，則表示檔案系統對於儲存空間的利用度越差，長期下來也是一種增加快閃記憶體磨損程度的原因之一，所以之後我們的實驗過程中，也會紀錄分析這 3 種檔案系統中 fragment 產生的多寡。

### 4.1.1.3 Mapping size effect

了解過 3 種檔案系統的實作之後，我們接下來要研究的是這三種檔案系統再

快閃記憶體上得資料結構和讀/寫的使用的對應大小(Mapping size)差異下，是否會有其他更深層的影響。首先在快閃記憶體上影響 Fragment 的大小程度主要原因就是檔案系統使用 Mapping size。由於這三種檔案系統(JFFS2,YAFFS,NFTL)所使用的對應大小不一，因此他們所產生的 fragment 大小也各不相同。例如 YAFFS 和 NFTL 是使用 Page-level 的寫入，因此若是要寫入快閃記憶體的檔案資料量相當小(小於一個頁面的話)，那麼這兩個檔案系統所採取的策略便是將剩餘非資料的空間用”0”填入，利用”0”補滿一個頁的大小之後再進行寫入，所以在快閃記憶體中這個頁中除了開頭的部份是使用者資料之外，其餘部份都是檔案系統填入的垃圾資料，是沒用且無意義的。

相對的，在 JFFS2 中，一份使用者資料的儲存可能是分成多個 Raw node 存放在 NAND 快閃記憶體上的，而在每一個頁面中擺放的資料可能是包含多個 node 的資料。導致雖然 JFFS2 每次寫入的單位也是頁，但實際寫入的資料大小可以細分到 node-level。所以在處理許多小檔案的時候，JFFS2 可以將這些不滿一個頁的小檔案形成的 node 一個一個的集結起來，再一起寫入到一個頁面中，因此對於空間的使用上相對的會比 YAFFS 和 NFTL 來的節省。但 JFFS2 並沒有充分利用到快閃記憶體中的備用空間(Spare space)，而是將管理系統。JFFS2 也是有 fragment 的問題，JFFS2 產生的 fragment 就大多是在每一個頁末端剩餘空間不足以再寫入下一個 node 的時候，也是採取利用”0”來填滿空缺所產生的 fragment，但這樣的浪費比起另外兩種檔案系統卻顯得微不足道。簡單的說，對 YAFFS 和 NFTL 而言，當使用的快閃記憶體晶片的 page size 越大，就可能會產生越大的 fragment。

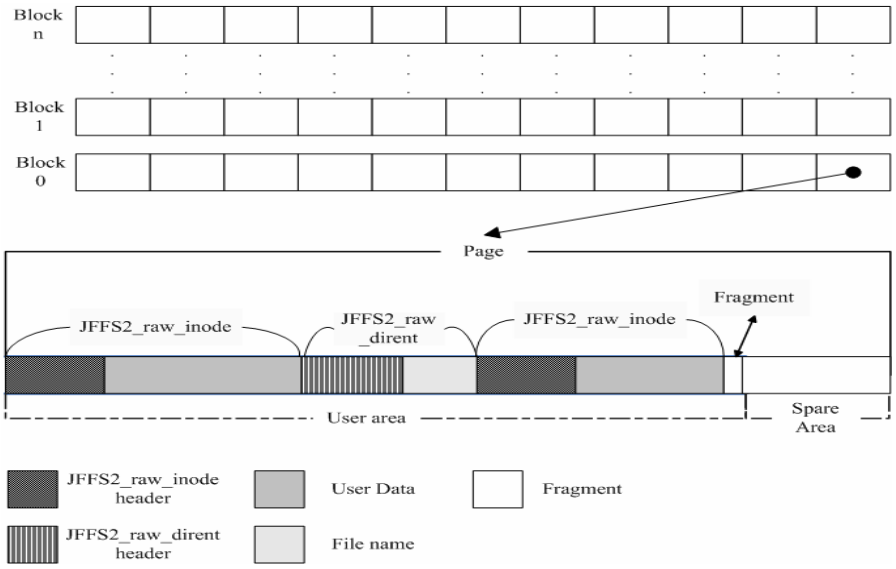
但是，是否檔案系統使用對應大小(mapping-size)越小所得到的效果就越好呢？這答案當然是否定的，檔案系統所使用的對應大小越小(mapping-size)，檔案被切分得越細碎，管理上也就越困難，需要紀錄的 metadata 也就越多。以 JFFS2 爲例，當一個檔案的各部位被細分成 node，又放置到不同的頁或區塊時，因爲 out-place 的關係每一個 node 都需要帶一份 metadata，再加上對應大小(mapping-size)的微小(node-level)所需要額外紀錄用於管理檔案系統的 metadata 也就多一點，結算下來所用的 metadata 就會比其他檔案系統更多一些。而 JFFS2 爲了實現 node-level，使得每一個形成的 fragment size 較小，但也捨棄了使用備用空間(spare area)，而將 metadata 存放在使用者空間，卻也減少了使用者可用的快閃記憶體空間。因此在對應大小(mapping-size)的大小之上的取捨是各有優缺點，沒有絕對的好壞之分。總結，對應大小(mapping-size)越大的使用的 metadata 量越少，相反對應大小(mapping-size)小的使用的 metadata 量較多。

### 4.1.2 File-System Implementations

了解過檔案系統在儲存體上會存放的各種形式之後，接下來我們會一一介紹 JFFS2,YAFFS 以及 NFTL 三種檔案系統在 NAND 快閃記憶體上各會使用哪些資料結構來管理使用者資料和檔案系統本身的 metadata。

### 4.1.2.1 JFFS2

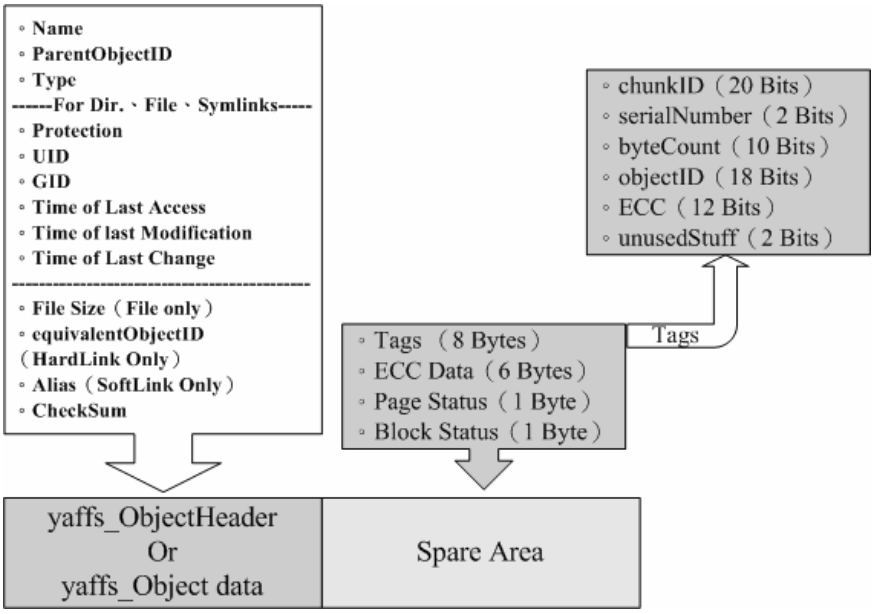
JFFS2 儲存在快閃記憶體上的內容物只有兩種基本的 nodes 分別爲 JFFS2_raw_inode 和 JFFS2_raw_dirent 。其中 JFFS2_raw_inode 主要是用於存放使用者 資料，而 JFFS2_raw_dirent 則是存放 file name 和檔案的目錄關係的 metadata。不論是 JFFS2_raw_inode 或是 JFFS2_raw_dirent，每一個 node 的資料量不會大於 NAND 快閃記憶體的頁面大小 (JFFS2_raw_inode 最大的尺寸是 512bytes 等同於一個頁面的大小)。由於使用這兩種基本的資料結構來管理使用者資料，而每一個 node 大小隨著後面伴隨的資料量不同或是 file name 長度不同所產生出來的 node 尺寸也就不相同，因此存放在一個區塊中的 node 數量也就不固定。因爲儲存的 node 數量不固定，無法利用統一的格式去管理這些 node 資訊，所以 JFFS2 放棄使用快閃記憶體中的備用空間(spare area)，將所有的 metadata 和使用者 資料全都集中存放在 使用者空間中。JFFS2 各種資料結構的關係範例，就如圖表 8.所表示的。圖中只表示出區塊中最後一個頁面中的內容(除了區塊中第一個頁面外，並非所有 page 都會剛好對齊某一個 node 的起始位置，只是舉例說明)，由圖可知在 JFFS2 當中，儲存在快閃記憶體上的內容物不外乎爲上述的 JFFS2_raw_inode 和 JFFS2_raw_dirent 兩種 node 結構，以及跟隨在 node 之後的使用者資料和檔案名稱(file name)。在圖中可看到當 JFFS2 在區塊最後的頁面中，剩餘空間無法再填入任何的 node 時，會標示成無用的 fragment 空間。




圖表 8. The data structure of JFFS2 stored in flash.

### 4.1.2.2 YAFFS

在YAFFS中存放資料的基本單位爲chunk(對應到記憶體中的一個頁面大小)，每一個yaffs_Object可能會由許多chunk構成，個數依照檔案的資料量大小來決定。而每一個yaffs_Object都會有一個yaffs_ObjectHeader，yaffs_ObjectHeader是YAFFS主要的On-Flash 資料結構。yaffs_ObjectHeader存在於快閃記憶體上，當快閃記憶體掛載上YAFFS時，會爲每個yaffs_Object找尋該物件的yaffs_ObjectHeader，讀出yaffs_Object的相關資訊。yaffs_ObjectHeader所儲存的資訊如圖表 9所顯示的。快閃記憶體中並非每個頁(chunk)都是存放yaffs_ObjectHeader，還有其他存放使用者資料的chunk。主要是依靠備用空間(spare area)中存放的資訊chunkID來辨別這個chunk是否爲yaffs_ObjectHeader。若chunkID=0 則此chunk爲yaffs_ObjectHeader，反之則爲存放資料的資料chunk。所以除了chunk0 之外的使用者空間(User area)都是用來擺放使用者資料。其餘關於YAFFS 資料結構詳細的資訊請參考[10]。

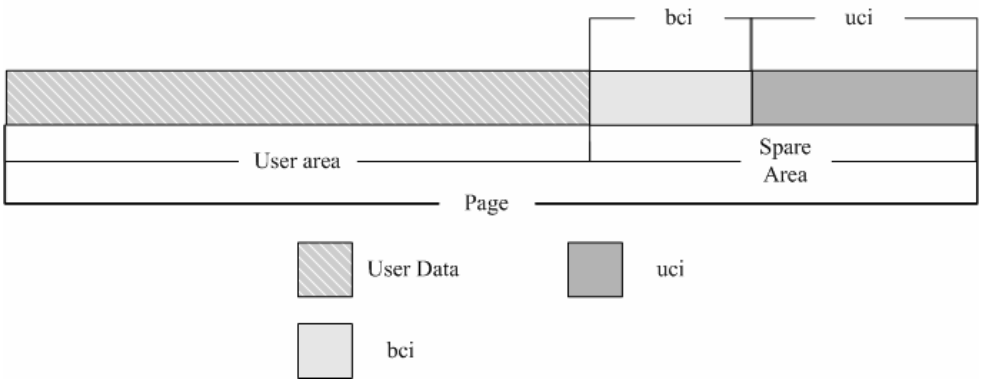



圖表 9. The data structure of YAFFS stored in flash.

### 4.1.2.3 NFTL

NFTL中使用者區域的部份大都是用來儲存使用者 資料，但根據使用的檔案系統(ext2/FAT)不同，所會佔用的使用者區域的空間也就不相同。例如NFTL上的檔案系統若是採用FAT，則當然必須對應建造一個FAT table來管理檔案系統的運作。NFTL在Flash上的資料結構主要則是存在備用空間(**Spare Area)**的**oobinfo** (如圖表 10. 所示)。OOB則是由**bci** (block control information)及**uci**(uni control information)組成，bci記錄每一個頁面的使用狀態status，uci則記錄每一個區塊相

關的資訊。其餘詳細bci和uci紀錄的metadata內容請參考[7]。




圖表 10. The data structure of NFTL stored in flash.

在之後的實驗的部份 Chapter 5，我們會就分爲兩種測試方式黑盒子和白盒子來設計實驗。黑盒子的設計是採用一般使用者在檔案系統中常見的檔案分佈。黑盒子測試主要是用來測試這三種快閃記憶體檔案系統在一般使用者常見的檔案分佈下，在看間利用度上表現的情形會是如何？白盒子測試分別是 Large file case(檔案大小約介於 512Kbytes-2Mbytes 之間 ，類似照片等圖檔或是較小的 mp3 檔案,WMA 檔案等等) 和 Very large file case(檔案大小介於 4Mbytes-16Mbytes 之間 ，類似是完整的 mp3 檔或是影片檔)，以及 Small File case(檔案大小大多小於一個頁的大小,約 512bytes,模擬當細小的檔案如網際網路暫存的 cookie,系統檔案和 linux source tree 等，系統中眾多細小檔案時的情形)。

***NOR Flash Memory*** 在 NOR 快閃記憶體上，由於 NOR 可以作 byte-level 的 update 和寫入，因此在 fragment 上的問題便減少了許多，不過由於 NOR 的價錢昂貴，所以使用空間也就到了錙銖必較的地步。即使產生 fragment 的原因較少，但是 NOR 快閃記憶體的大小普遍都不大，其可用空間也就格外珍貴。在這 3 種檔案系統中只有 JFFS2 能支援到 NOR 快閃記憶體，而 JFFS2 運行時便會有 word alignment 的情形發生，可能會造成片段的 fragment。但我們研究的重點著重在 NAND 快閃記憶體，所以在 NOR 這部份便不深入探討了。

## 4.2 RAM-Space Requirements

這一個章節我們要探討的便是在這眾多的檔案系統中，我們要探討的是三種檔案系統在一般的工作量(workload)下和一些特殊的情況下，爲了維持系統運作所要佔用的記憶體空間的多寡。

由於在使用快閃記憶體爲主要儲存體的嵌入式系統中，大部分的嵌入式系統只用於某些特定用途，通常系統整體的大小不大，加上爲了節省成本和降低電力消耗量(power consumption)，導致嵌入式系統使用的資源相當有限(低頻率的

CPU，少許的記憶體)。一般的情形下，嵌入式系統所擁有的記憶體，不管是 ROM 記憶體或是 RAM 記憶體都局限在 64MB 以下，所以在資源有限的情形下，一個設計良好的嵌入式儲存系統運作時是不應該佔據太多 RAM。優良的快閃記憶體檔案系統應該盡可能節省維持運作所佔用的 RAM 量，並將絕大部分資源都給使用者進程(process)去運用。

### 4.2.1 Performance issues

本章節針對爲一個快閃記憶體檔案系統所需要的各種 RAM-resident 資料結構作一討論。這些需求往往與快閃記憶體的物理特性有關，因此與以往之磁碟檔案系統之需求大不相同。基於這些影響 RAM 空間使用量的因素，我們後續章節會接著討論如何對這些因素做效能評比。

#### 4.2.1.1 Address Translation

由之前章節 3 所提及的觀念位址轉換可知，快閃記憶體檔案系統爲了實現位址轉換都會利用一些儲存在記憶體中的位址對應表。相對於各檔案系統所使用的對應階層不同，所需的位址對應表大小和佔用的記憶體空間也就大不相同，會使得影響記憶體使用量的因素不僅僅祇有檔案系統中儲存檔案的個數，例如：JFFS2 中類似 node-level 的對應關係，使得 JFFS2 的記憶體使用量是和檔案系統中儲存的 raw node 個數有絕對的關系，而且影響的效果相當顯著。所以在之後的三種檔案系統實作位址轉換的段落中，我們會一一的介紹 JFFS2,YAFFS 和 NFTL 中實踐位址對應表(address translation mapping table)的觀念，並在之後的實驗中，再借由這些原理來設計實驗範例，來探討這 3 種檔案系統在特定情形和工作量(workload)下，所使用的記憶體大小和利弊關係。

#### 4.2.1.2 File-system management structure

在這三種記憶體檔案系統中，使用記憶體的地方，除了上一段提到的實現位址轉換功能所使用的資料結構之外，快閃記憶體檔案系統儲存在記憶體的資料結構還有哪些呢？扣除一些 read/write 時讀入的使用者的檔案資料之外，其他還有管理檔案系統所必須的 metadata 資訊。這些 metadata 資訊會紀錄著每個檔案系統的運作時的即時資訊，例如：JFFS2 中有使用三種以上不同的串列來區別檔案系統中每個區塊的使用狀態，依照區塊內含無效資料的多寡區分成 clean_list、dirty_list、very_dirty_list 和 erasable_list 等狀態不同的區塊，方便之後垃圾回收挑選目標區塊時，能快速區分出較有回收效益的區塊。而這些 metadata 通常是在檔案系統掛載完畢後，就一定會存在於記憶體的資料結構，不管檔案系統中是否存有任何的檔案，也就是檔案系統天生擁有的最低記憶體消耗的資料結構，不過這些 metadata 佔有的記憶體空間差異不大，而且不管在那一種情況中都是消耗固定的記憶體，不會隨著使用者使用的情況的改變而產生大變化，因此我們對

這部份的探討便不太深入的探究，因爲我們對會依照情況改變而有大程度改變的項目較有興趣。

#### 4.2.1.2 Directory hierarchy structure

在記憶體中的資料結構，還有微詞目錄階層架構的功能，例如:YAFFS 使用 yaffs_object 的資料結構來建構出整個檔案系統的目錄架構。JFFS2 在記憶體中的資料結構除了位址轉換的功能之外，也提供了類似目錄階層架構的關係，可以使得檔案系統迅速的找到所屬目錄和檔案在檔案系統中相對映的親子(parent-child)關係。這也是檔案系統中必須的功能之一，但在 NFTL 只提供了邏輯位址和實體位址的對應關係，其檔案系統中的目錄結構關係，必須藉由底下搭配的 FAT 和 EXT2 等檔案系統去管理和控制，所以在之後實驗記憶體使用量上的比較，我們也就必須考量到 NFTL 底下搭配的檔案系統所使用的記憶體量。

### 4.2.2 File-System Implementations

接下來我們會依序介紹 3 個快閃記憶體檔案系統，是如何實作位址轉換的機制，以便實現 out-place 策略來管理快閃記憶體的空間。

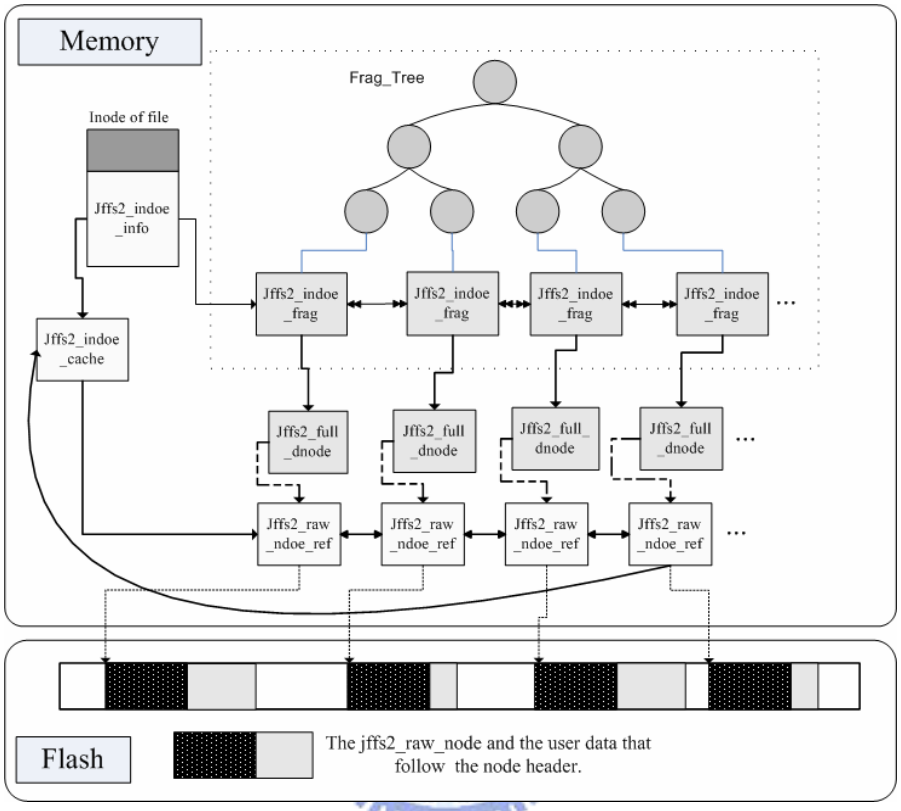
#### 4.2.2.1 JFFS2

在 JFFS2 中使用的資料結構相當的多，簡單的分類介紹一下。首先是一般常備在 filesystem 的資料結構，主要是每一個檔案和目錄都會有一個 inode 去表示，而 JFFS2 會在這些 indoe 的附加資訊中加入 jffs2_indoe_info 的資料結構，其中會連接著一個 jffs2_full_dirent 的 list 和一個 jffs2_inode_cache，而這個 jffs2_inode_cache 中又會指向一個 jffs2_raw_node_ref 構成的 list。這些 jffs2_raw_node_ref 主要是用來指向儲存在 flash 上 jffs2_raw_inode 和 jffs2_raw_dirent 的實體位址。在檔案系統剛建立的時候，系統會掃描整個快閃記憶體來建構這些資料結構，再利用這些資料結構來組成系統初期的目錄階層關係。

JFFS2 爲了實作位址轉換的機制，所使用記憶體映成(memory mapping)的關係，大致上可以分作兩種。一種是當 inode 爲開啓的檔案的時候，另一種情形 indoe 是一般目錄。這兩種情況下的對應關係圖，分別爲圖表 11 和圖表 12 所表示。

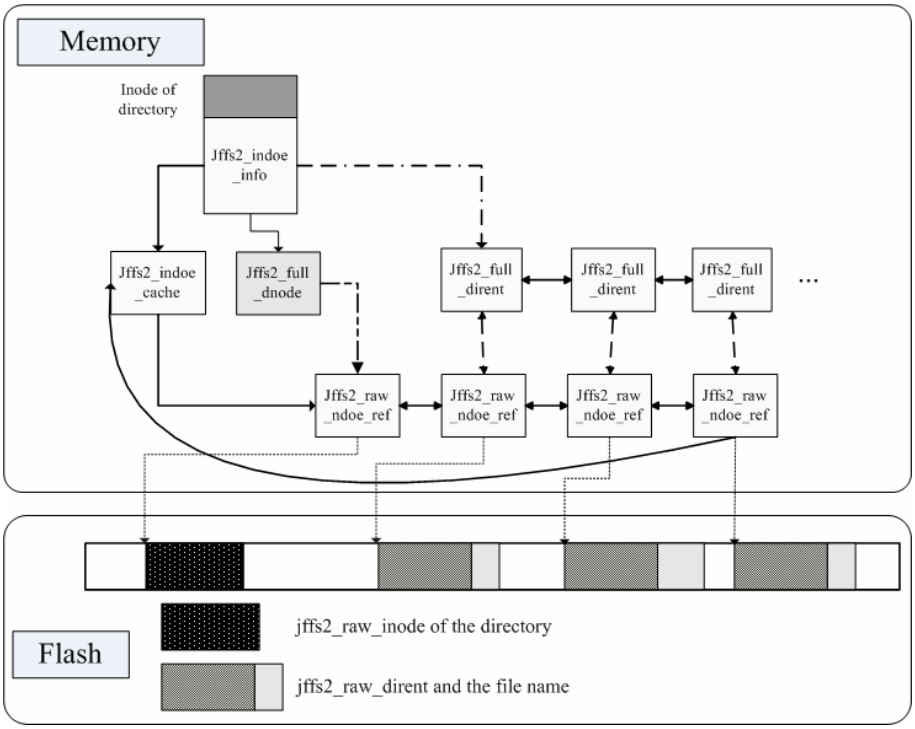
首先當 inode 用來表示 file 時，在 jffs2_indoe_info 中會有一個指向由 jffs2_inode_frag 所構成的 list，當開啓檔案時會利用這個串列(list)建立 frag tree，而這個 frag tree 是依照所對應的 raw node 的版本(version)大小所建立，所以參照這個 frag tree 可以建構出擁有最新版本(version)的檔案內容。而每一個 jffs2_inode_frag 會對應到一個 jffs2_raw_node_ref，並紀錄所對應到的

jffs2_raw_node_ref 中的實體位址的偏移量(physical address offset)爲何以及被多少個 frag nodes 參照到，沒被任何 frag node 參照到的 jffs2_full_dnode 便會被刪除。因此每當檔案更新片段的資料，除了新增一個 jffs2_raw_node_ref 之外，還要新建對應的 jffs2_node_frag 和 jffs2_full_dnode 這兩個資料結構。(圖表 11)




圖表 11. The JFFS2 structure when open a file.

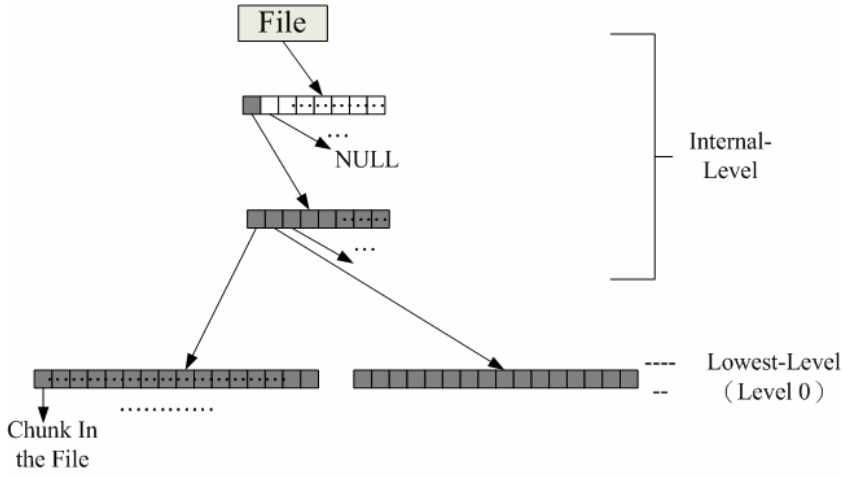
當 inode 爲目錄時，jffs2_indoe_info 中 jffs2_full_dirent 的 list 便是表示著這個目錄底下所擁有的子目錄或檔案，每一個 jffs2_full_dirent 又會指向一個 jffs2_raw_node_ref，這些 jffs2_raw_node_ref 就指向 flash 中儲存 jffs2_raw_dirent 的實體位址。jffs2_indoe_info 會另外對應到一個 jffs2_full_dnode，再尋線(透過 jffs2_raw_node_ref)可找到對應的 jffs2_raw_inode 的實體位址。這些 jffs2_raw_node_ref 會構成 jffs2_indoe_info 上指向 jffs2_inode_cache 中的 jffs2_raw_node_ref 的串列。 (圖表 12)




圖表 12. The JFFS2 structure of opening directory.

### 4.2.2.2 YAFFS

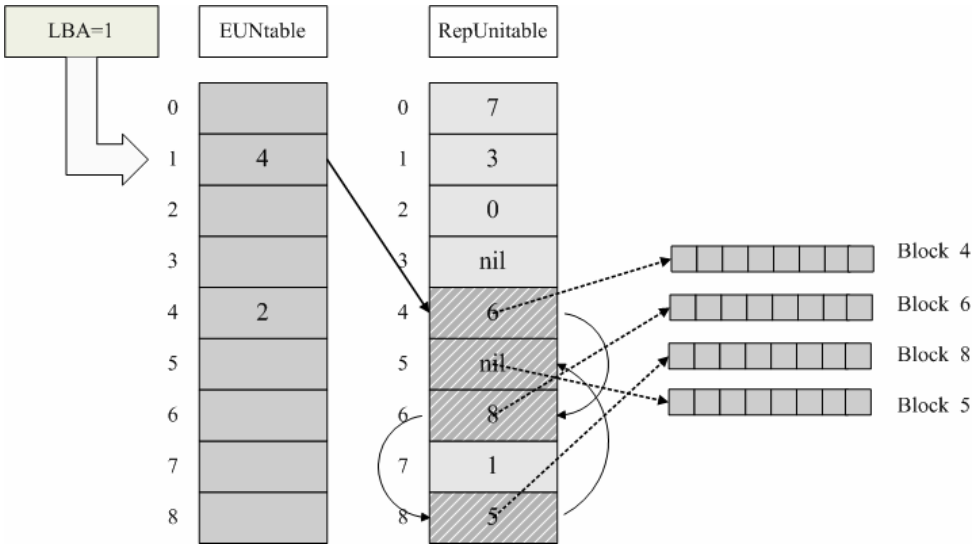
YAFFS 儲存在主記憶體上的資料結構主要有 3 種：yaffs_Object, yaffs_Tnode 和 yaffs_Device。Yaffs_Device 是類似 Superblock 的資料結構，用來紀錄整個儲存裝置的上下對應關係(EX:紀錄對應到的 MTD device 和讀寫所需的運算函數)。Yaffs_nodes 是 YAFFS 中主要用來處理邏輯位址(logical address)和實體位址(physical address)之間關係的資料結構(如圖表 13 所示)，每一個檔案可以視爲許多的 yaffs_tnode，而底層 LEVEL 0 的 yaffs_tnode 會指向快閃記憶體中真正儲存資料的 yaffs_object 的實體位址。




圖表 13. YAFFS 的 address translation (Tndoe-tree)

### 4.2.2.3 NFTL

NFTL在RAM上的資料結構只有NFTLrecond，而NFTLrecond的大小是固定的佔用 364bytes的memory，其中主要是利用兩張表格(EUNtable和RepUnitable)來模擬Virtual Unit Chain[7]進而達到位址轉換的關係，其中**EUNtable**是一個logical to physical的表格，紀錄每一個block所屬的串鏈(chain)的起始實體位址，而**ReplUnitTable**是一個physical to physical的表格，紀錄每一個在串鏈中的區塊，以及之後的區塊的實體位址，以維持Virtual Unit Chain的結構。NFTL管理所有的區塊的對應關係和運作如圖表 14.所示，圖表 14 有兩條virtual unit chain 分別爲 4-6-8-5 和 2-0-7-1-3。




圖表 14. NFTL address translation

## 4.3 Garbage Collection

垃圾回收主要的工作就是回收已經被寫過的快閃記憶體空間，並將這已被寫過的快閃記憶體空間利用抹除的動作回復到原始可執行寫入動作的狀態。關於垃圾回收的部份，我們主要關心的是各個檔案系統執行垃圾回收的動作所耗廢掉的時間，以及所消耗掉的抹除次數和額外的寫入資料。我們還是利用 Postmark 來作爲黑盒子測試的主要方式，再搭配[1]中 Linux 作業系統中常見的檔案分佈的關係，去調設 Postmark 的參數，將檔案大小設定在最常出現大小範圍內來進行測試。而白盒子的方法分爲兩種：分別爲範例一:循序地大檔案寫入(Sequential Write Big file)和範例二:交錯地寫入大檔案和小檔案(Interleaving Write Big/Small file)。

[7] Virtual Unit Chain：由一個以上的**Virtual Unit**所組成，virtual unit則是在RAM中用來對應flash中的erase unit，實際上virtual unit並不儲存資料，只儲存所對應的erase unit的physical address，即ReplUnitTable的entry。而Virtual unit chain則是一個logical address指著virtual unit chain的第一個virtual unit的位置。

分別測試循序讀寫和交錯寫入 hot/cold 資料的影響。

### 4.3.1 Performance Issues

#### 4.3.1.1 Hot data and Cold data

.由於使用者的資料有 hot data 和 cold data 的分別，如果在資料儲存的管理上沒有將兩種不同類型的資料作個區隔，使得冷熱資料交雜的存放在快閃記憶體中的話，會造成垃圾回收的效率低落。主要是由於在垃圾回收時，冷熱資料的交替存放會造成挑選到的目標區塊都擁有 cold data 的 live page，而較難找到全都是 dirty page 的區塊來作爲垃圾回收的對象，所以使得每一次的垃圾回收都需要作資料搬移的動作，對垃圾回收的效能便大大的降低了。

#### 4.3.1.2 Spare-block reservation

在快閃記憶體檔案系統中，都會預留空間作爲 spare block 方便垃圾回收時使用。因爲 spare block 的數量有限，所以檔案系統垃圾回收在 spare block 的數量剩下很少量的時候，會啓動垃圾回收已清出 free block 作爲 spare block。可是此時，由於資料擺放的配置的策略不同，將會對 spare block 使用有所影響，如 JFFS2,YAFFS 都是有還有剩餘空間(free block)就可以拿來作爲 spare block 存放使用，而 NFTL 會根據 LBA 被寫到的位址，再去要求 spare block 作爲對應的區塊。在 JFFS2 和 YAFFS 中，由於位址轉換的單位不是區塊，因此比較不會快速消耗可用空間和 spare block，但若不特殊處理容易發生 hot/cold mixture 的問題。但 NFTL 中是依據 LBA 存放資料的緣故，兩個不同檔案的資料不會交錯存放(LBA 不同)，所以 hot/cold mixture 的情形沒那麼嚴重，但是可能在隨機小檔案的寫入的情形下，可能對應了很多 LBA's 的區塊，使得 spare block 會被快速消耗。垃圾回收的效能也會由於使用者資料量增加或剩餘可用的區塊數量的減少而下降[28]，主要是可以作爲 spare block 的空間減少，使得垃圾回收的難度和負擔增加。

#### 4.3.1.3 Garbage-collection policy

由於垃圾回收的主要overhead在於回收區塊中所包含的live page數目和區塊抹除的次數，因此垃圾回收所使用的回收策略(Garbage collection policy)的選擇就格外的重要。在這一方面的研究， Kawaguchi and Nishioka [27]提出的方法除了採取最基本的Greedy policy[5]之外，還有考量過資料冷熱影響的cost-benfit policy。Kawaguchi and Nishioka在觀察垃圾回收的過程中發現，如果可以避免回收到存有經常性更新的hot data的live page的區塊，就可以大大地提昇垃圾回收的效益。主要是爲了防止回收時搬移的live page的hot data，由於經常性更新的關係馬上變

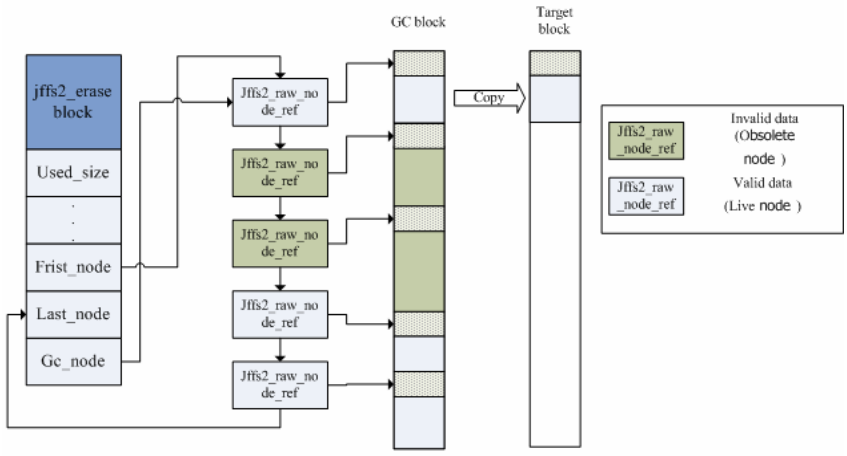
成dead page，使得在垃圾回收階段的搬移動作沒有太大意義，而且需要再要求空間存放更新後的資料，可能又觸發下一次的垃圾回收產生。因此Kawaguchi and Nishioka提出了Cost-benfit policy的回收策略，是對每個區塊中都加入一個數值，再以這個數值作爲執行垃圾回收時挑選區塊的好壞依據，好的話表示選擇回收的區塊花費額外的負擔低且存有hot data的live page少。

### 4.3.2 File-System Implementations

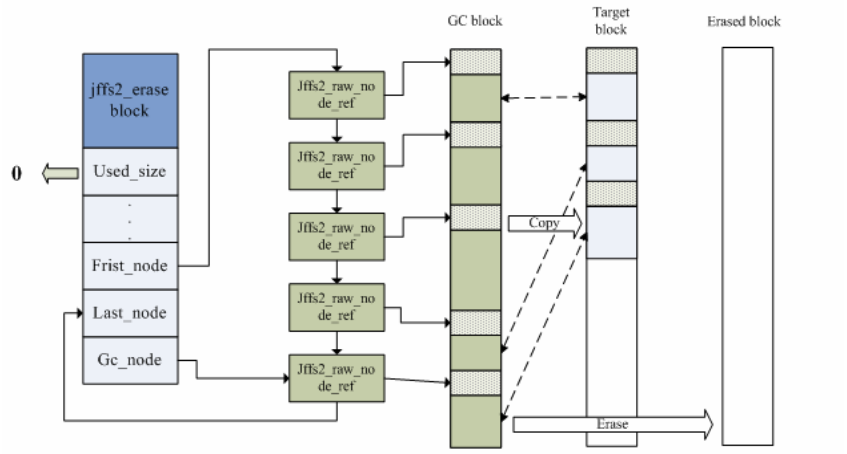
接下來我們會依序介紹三個快閃記憶體檔案系統，是如何實作垃圾回收的功能，簡單的介紹 3 種檔案系統，回收已寫過區塊的空間以便再使用的過程和步驟。

#### 4.3.2.1 JFFS2

首先在 JFFS2 中會使用名爲 jffs2_eraseblock 的資料結構來管理每一個區塊，在資料結構中會將儲存於區塊上所有的 jffs2_raw_node_ref 聯成一個串列，利用指標欄位 Frist_node 指向串列的開頭，而每個 jffs2_raw_node_ref 會指向下一個儲存於實體位址中的 node，串列的最後指回 jffs2_eraseblock 結構中的 Last_nod。當垃圾回收發生時，JFFS2 發生時會先選定一個區塊作爲 GC_block，再將這個區塊中的 raw node 一個接著一個的檢查回收 。先利用 Gc_node 的指標欄位指向目前選擇回收的 raw node，先檢查此 node 是否爲廢棄(Obsolete)。若不是，則搬移有效資料(valid data)到目前選擇寫入的區塊(Target block)上。反之表示 node 上資料爲無效不需要搬移，繼續跳至下一個 node 作檢查搬移的動作。(參考圖表 15(a))重複這些回收有用 raw node 的動作，直到檢查完所有此區塊上的 raw_node 或 jffs2_eraseblock 結構中的 used_size 欄位爲零，表示整個區塊中已經沒有有效資料，再進行抹除的動作回收整個區塊空間。(參考圖表 15(b))




(a) Copy valid data (live node) to free block and mark the node obsolete repeatedly.



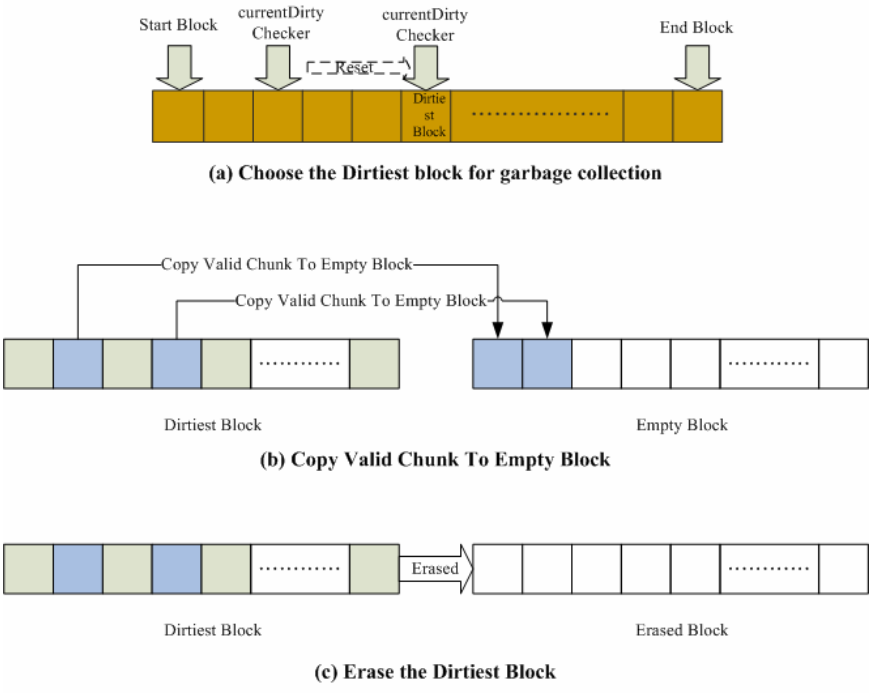

(b) After copying all live node (Used_size=0) ,then erase the GC block

圖表 15. Garbage collection policy of JFFS2

由於 JFFS2 的垃圾回收是一個一個 node 回收執行的，因此若是區塊中 live raw node 數目很多的話會導致垃圾回收速度受到影響而變得緩慢。爲了加快垃圾回收的效率，JFFS2 中有 gc_thread 的機制(掛載時便會產生)，會利用系統空閒時間偷偷地進行垃圾回收的動作，或多或少能提昇一點垃圾回收的效率。而其他垃圾回收執行的時機大都是在寫入時要求可用空間時，因此當寫入動作頻繁時，系統無多餘的空閒時間，會導致 gc_thread 失去功用，垃圾回收都需要檔案系統自行處理。

### 4.3.2.2 YAFFS

YAFFS 選擇垃圾回收的區塊的策略相當簡單。如圖表 16 所示，YAFFS 再處理垃圾回收時，一開始先在一段選定的區間中找尋一個含有無效資料最多的區塊(Dirtiest Block)(圖表 16 (a))，將此區塊上有用的資料頁面拷貝到另外的可用區塊上(圖表 16 (b))，最後再對 Dirtiest block 執行抹除動作回收空間(圖表 16 (c))。之後再從上次選擇爲垃圾回收的目標區塊開始，作向後尋找 Dirtiest block 的動作。



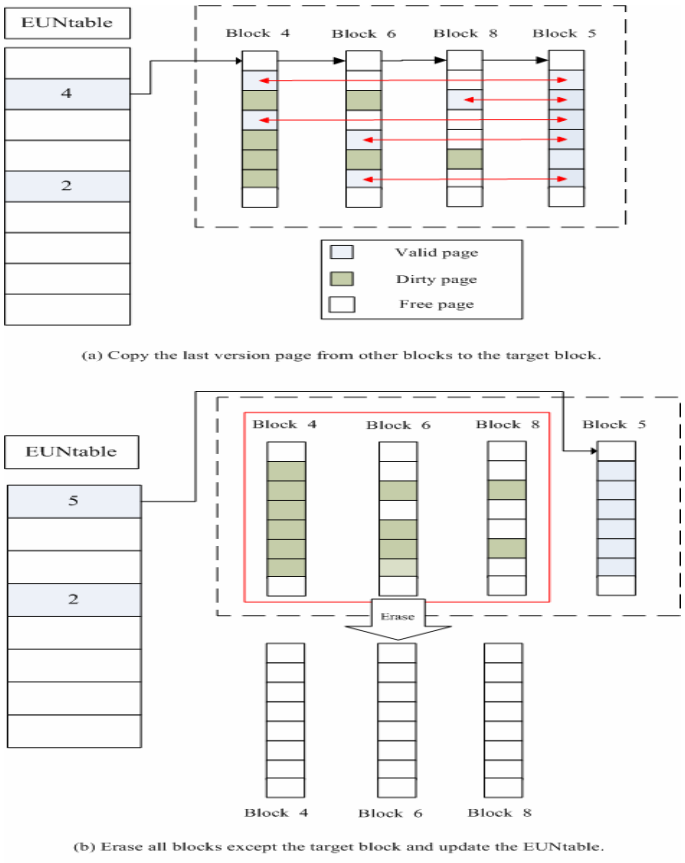

圖表 16. Garbage collection policy of YAFFS

由於 YAFFS 選擇垃圾回收的目標區塊的策略非常簡單，而爲了避免花費太多時間再尋找 Dirtiest block，在有用區塊充足的情況下，YAFFS 會限定每次的搜尋範圍在 200 個區塊中，所花費的時間也會有所限制。除此之外，才會對全部的區塊作檢查，挑選出可回收最多空間的區塊。

### 4.3.2.3 NFTL

在NFTL中，處理垃圾回收的方式是先找出串連在RepUnitable中最長的Virtual unit chain (可參考章節 4.2.2.3.NFTL)，之後逐一的將最新的資料版本搬移到目標區塊(target block[8])上，之後便執行抹除和回收串列上的其他區塊，再將EUNtable對應到的區塊編號改成target block。(如圖表 17 所示)

[8] Target block：在NFTL中選定用來存放垃圾回收時搬移資料的區塊，通常是選擇作垃圾回收的virtual unit串列中最後一個區塊，或是另尋一個可用區塊(free block)來充當target block。




圖表 17. Garbage collection policy of NFTL

由於 NFTL 的資料排放方式(Data Placement)是利用這種串列串連的方式，而不管串列多長，同一個串列中串連的資料都是相同的，都是表示儲存在同一個 LBA 相同位址的資料。因此再進行垃圾回收時，可以將整個串列的有效資料都回收存放到一個區塊上，故選擇最長的串列可達到最大的空間回收效益。

## 4.4 Wear-Leveling

在平均抹除的這個部份，我們主要關心的還是檔案系統所造成的區塊抹除次數的多寡，和各個區塊抹除的次數是否平均，能不能達到平均抹除的效果。

### 4.4.1 Performance Issues

由於在處理 Wear-leveling 的過程之中，有一個長程的目標，也就是盡可能的讓每一塊區塊的抹除次數相近，而且額外的抹除次數不會太多，使得整體的抹除次數只有小幅度的增加，也就是在花費最小成本的狀態下達到平均抹除的效果。所以好的平均抹除演算法會找尋出擁有較小抹除次數的區塊來進行抹除，而通常擁有較少抹除次數的區塊大多是儲存不常更動的 cold data 和幾乎不更動的靜態資料。但選擇這些區塊來作爲目標區塊執行抹除，獲得的空間回報率是偏低的且額外的拷貝動作也較多。這樣和章節 4.3 中提到的垃圾回收的本質是有所牴

觸的，因爲垃圾回收的策略是找尋無效資料最多的區塊，已減少搬移的資料量，增加回收空間的效率。所以如何再這兩者互相矛盾的方法中，找到一個恰當的平衡點，對每一個快閃記憶體檔案系統來說都是的重要議題。

### 4.4.2 File-System Implementations

由於在 JFFS2,YAFFS 和 NFTL 中，只有 JFFS2 有使用平均抹除的演算法。而 YAFFS 和 NFTL 都沒有使用特別的策略去處理平均磨損。因此在之後設計實驗的考量中，主要是針對 JFFS2 的特點來設計。

#### 4.4.2.1 JFFS2

JFFS2 的平均抹除演算法是 99/1 法則，在每 100 次的抹除運算中，會有一次是挑選區塊內全是 clean data 的區塊來作垃圾回收，剩餘的 99 次會機率性在部份擁有 dirty data 的區塊內挑選，擁有較多的 dirty data 的區塊被挑選爲目標區塊的機率便越高。

#### 4.4.2.2 YAFFS,NFTL

由於 YAFFS 和 NFTL 沒有採取任何平均抹除的演算法，只是單純的選擇擁有最多無效資料的區塊來作垃圾回收動作。但在業界也稱垃圾回收爲 Dynamic Wear-leveling。所謂的 Dynamic Wear-leveling，是由於垃圾回收不會去選擇儲存靜態資料的區塊作爲抹除的目標區塊，因此被挑選來作垃圾回收的區塊，大多是儲存過一些經常變動的檔案資料，而當這些資料不斷地更新時，垃圾回收不停的啓動下，每一次挑選的目標區塊都不會是那些內部全儲存靜態資料的區塊，而是擁有最多無效資料的區塊。因此 Dynamic Wear-leveling 只會侷限在那些沒有儲存靜態資料的區塊中，而儲存著靜態資料的區塊都不會被抹除和更動。

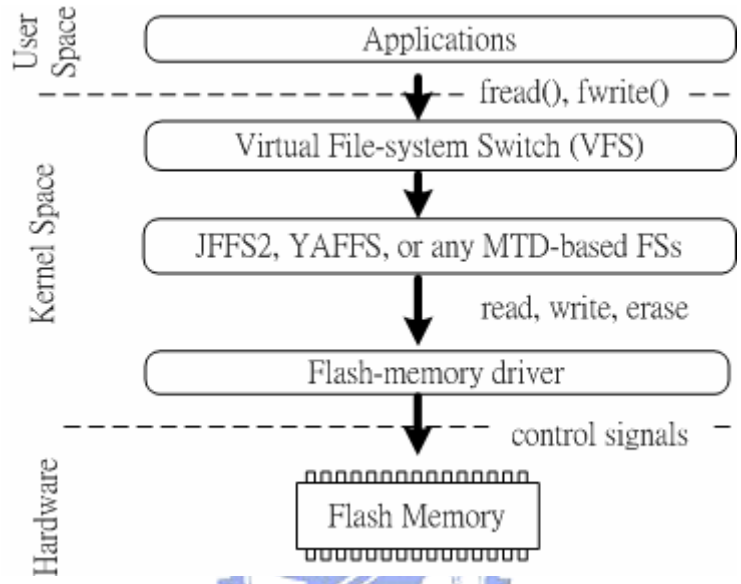
## 5. Experiment Design and Experiment Result

首先說明一下我們的實驗環境，我們使用Windows中的軟體VMware，創造一台virtual machine(VM)，實驗的VM使用的linux kernel 的版本是 2.6.17。然後利用nandsim模組在主記憶體中模擬出一塊 128MB，Block size爲 16Kbytes和Page size 爲 512bytes 的 NAND 快閃記憶體。之後在這模擬的 NAND 上，掛載 JFFS2,YAFFS 和 NFTL+FAT32。 其中 JFFS2 的模組沒有開啓 summary 和 compression的功能，而NFTL上運行的檔案系統是FAT32。詳細的實驗環境如表格 2所表示，而實驗平台的系統架構則如所圖表 18 表示的。

表格 2.實驗環境設定

| Experiment configuration (VMware+Linux-2.6.17) |
|---|

| | |
|---|---|
| NAND Flash Memory simulator (NANDsim) | 128MBytes |
| Page size | 512Bytes |
| Block size | 16KBytes |
| Page read delays | 25 us |
| Page write delays | 200 us |
| Block erase delays | 1.5 ms |




圖表 18. 系統架構圖

## 5.1 Flash-Space utilization

### 5.1.1 Experimental Setup and Performance Metrics

在實驗的步驟上，我們先行製造出符合我們實驗範例的樣本檔案分佈，之後在將這些樣本檔案複製到掛載有 JFFS2、YAFFS、NFTL 等等案系統的 block device 上，在將其卸載之後在利用 MTD-Until Tool 中的 nanddump 程式得到實際的 NAND 快閃記憶體的映像檔(Image)，利用我們自行設計的程式去判讀分析此映像檔。由於我們所寫入的檔案式一些擁有重複字串的二元碼(binary code)，因此很容易可以在 NAND 快閃記憶體映像檔中被判讀到，簡單的去區分出 metadata 和使用者 資料之間的差別，進而得到 NAND 快閃記憶體上得使用者資料、可用空間、Metadata、Fragment 等等佔用空間大小的資訊。

### 5.1.2 Test plans：

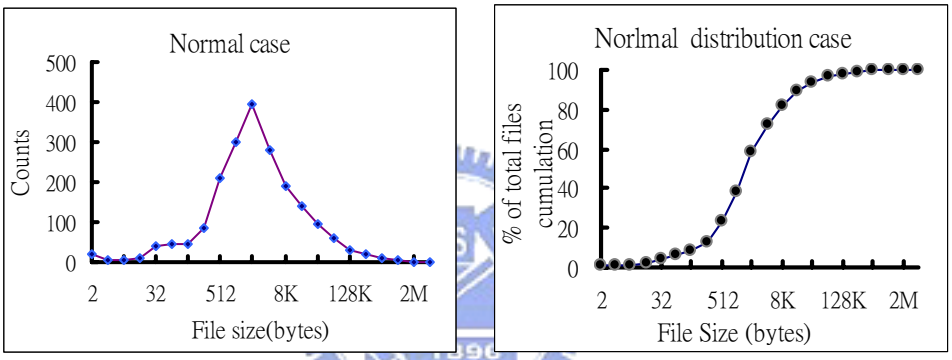
首先我們利用幾個不同的實驗範例，分別爲常態分佈(normal distribution), 大檔案(all large files),非常大檔案( all very large file) ,小檔案 (all small files)等 4

個實驗範例。其中利用 normal case 作爲這裡的黑盒子測試，而另外 3 種特殊或病態的例子，是針對特定的檔案系統去量身訂做的例子。可以讓我們瞭解到這幾種檔案系統在這些特別的例子中的行爲模式，並且可以觀察出他們的原始設計中個別在哪些情形下表現的優劣。

### 5.1.2.1 Black-Box test

## Normal Case (normal distribution)

Normal case爲一般的檔案系統中檔案大小分佈的情形[1]，詳細分佈的情況如圖表 19所表示。在這裡使用Normal case 作爲我們的黑盒子測試，是以Unix中常見的檔案分佈爲基礎，依照檔案大小所佔的檔案個數比例創造出約 2000 個檔案來當作實驗的樣本，來分析快閃記憶體爲主的檔案系統在一般常見的磁碟爲儲存體中檔案分佈下的表現，並且用來作爲之後白盒子測試主要的對照。



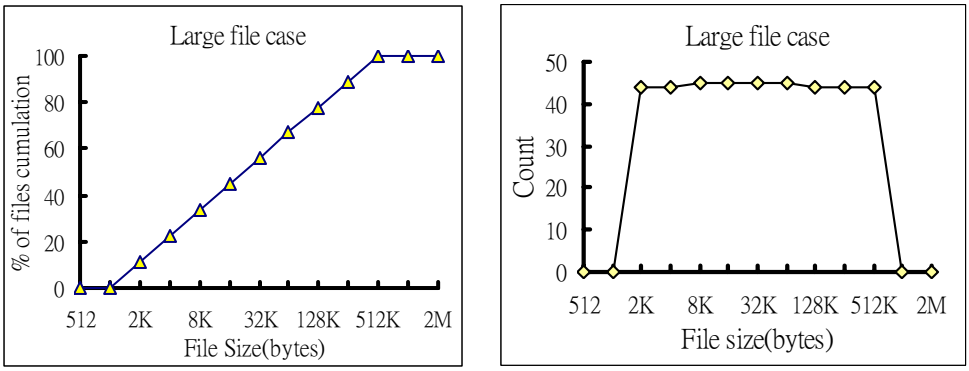

圖表 19.Normal case (By count) and Normal case.(use file cumulation)

### 5.1.2.2 White-Box test

## Case.1 Large file (2KB-2MB)

Large size case 是特別設計來呈現這 3 個檔案系統在檔案尺寸都是大於頁面大小的情形下的表現爲何。主要是我們基於觀察檔案系統設計的原理，我們猜想在超大檔案的表現中，YAFFS 的表現應該是相較於其他兩個檔案系統來的好的。但若是在這個無小檔案的實驗範例中(每一個檔案皆大於頁面大小)，但又不到一般的 mp3 格式的大小(約 4mb 左右)。由於好奇結果又是會如何，因此也設計了這個例子。

我們所設計的實驗範例就如圖表 20所示，做出 400 多個檔案，每個檔案大小都介於(2Kbytes-512Kbytes)之間。

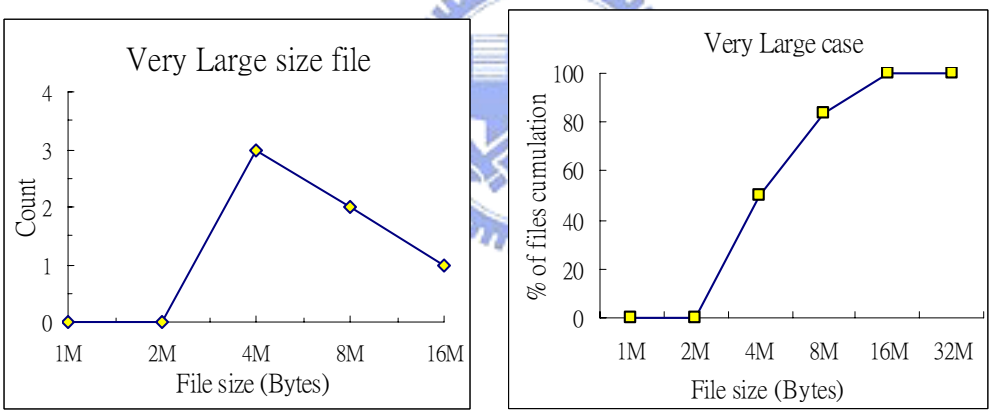



圖表 20. Large file case (by count) and Large file case.(use file cumulation)

## Case.2 Very Large size files

Very large file case 主要是設計來反應 YAFFS 和 NFTL 的優勢，由於這兩個檔案系統的資料結構在大檔案和檔案數目較少的情況下會相當的有利。由於這兩個 mapping size 較大的關係，所需要的 metadata 量較少。

我們所設計的實驗範例都爲 4MB以上的file，如圖表 21所示，檔案分佈分別爲 4MBytes(3)，8Mbytes(2)，16Mbytes(1)。之後再將這些檔案寫到測試的快閃記憶體檔案系統中，卸載之後分析印象檔所得。



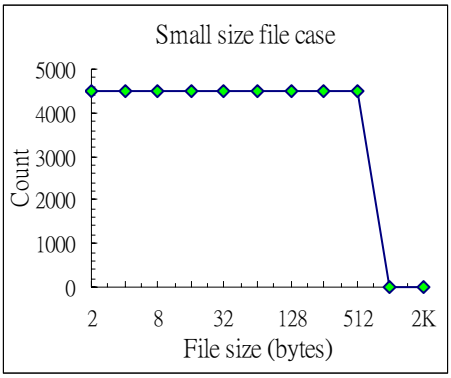

圖表 21.Very Large file case(by count) and Very Large file case(use file cumulation)

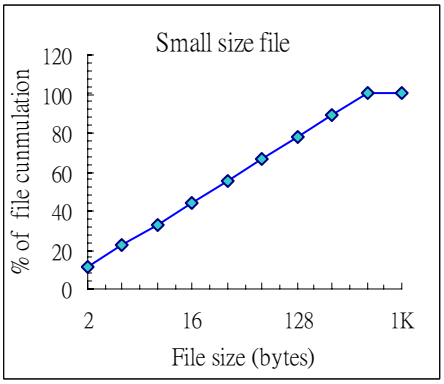
## Case.3 Small size files

Small size files case 是爲了突顯 JFFS2 對於細小的 File 的處理會較節省空間的優勢。在檔案數目多且檔案大小非常小(小於 page size)的情形之下，JFFS2 所使用的近乎 node-level 的檔案對應方式，能將許多個小檔案寫入到一個頁面中，相對於 YAFFS 和 NFTL 要將每一個檔案就佔據一個頁面的處理方式會更節省使用到的快閃記憶體空間和減少寫入的次數。

我們製造的實驗範例的等檔案分佈，就如同圖表 22所表示的那樣，我們建立 40000 多個細小的檔案(檔案大小介於 2bytes和 512bytes之間 ，使用的快閃記憶體page size是 512bytes)，利用這個小檔案爲多數的分佈來量測，當系統中處理

的檔案大多數爲細小的檔案時，各個快閃記憶體檔案系統在快閃記憶體空間使用的情形會是如何。




圖表 22.Small size file case(by count) and Small size files case(use file cumulation)

### 5.1.3 Numerical Result

#### 5.1.3.1 Black-Box test

#### Normal Case (normal distribution)

實驗結果如下列表格 3和圖表 23所示，可以觀察出JFFS2,YAFFS和NFTL之間各項數值都相差不遠，除了在fragment的部份。這是由於實驗環境下的頁面大小爲 512bytes，然而YAFFS中的用來儲存資料的部份是以chunk的大小 (等同於一個page size的大小)爲單位，因此在檔案的資料量小於 512bytes的情況下，資料量無法填滿一個chunk，因此剩餘的空間變成了無法使用的fragment space。由於這個normal case中有 23.4%的file小於等於 512bytes，所以迫使YAFFS和NFTL產生不少的fragment。相對於YAFFS和NFTL而言，JFFS2 的儲存是利用許多個jffs2_raw_inode和jffs2_raw_dirent去儲存，每一個raw node size會根據其後跟隨的資料量影響而變動，但最終不會大於一個頁面的大小。因此對於細小的檔案而言在JFFS2 的管理機制比較節省空間，不會產生過大的fragment。

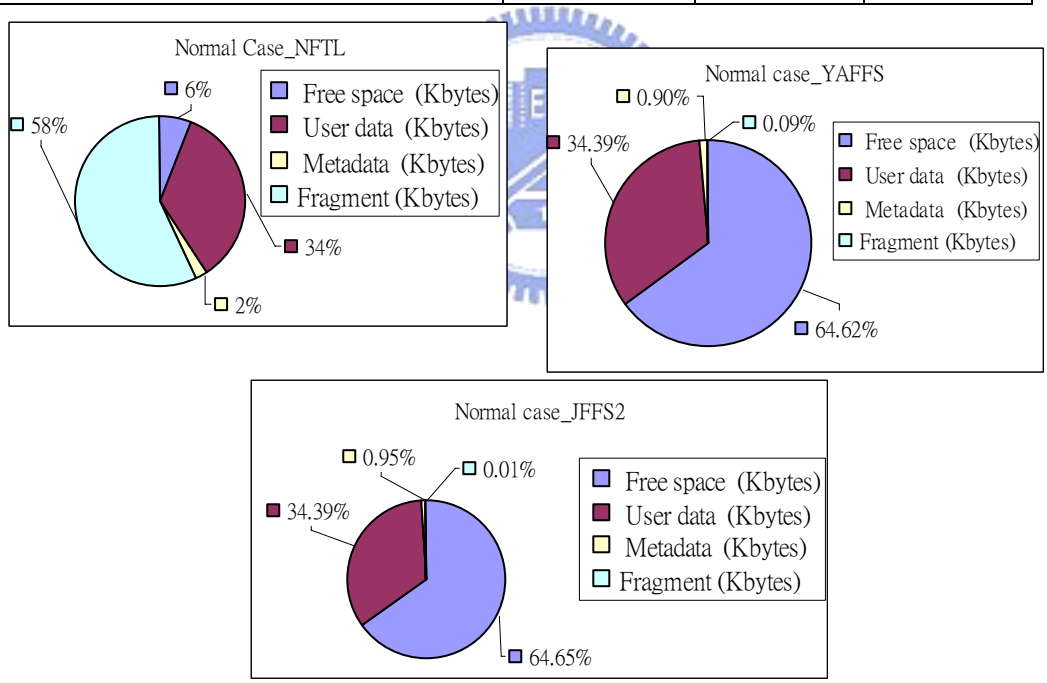
JFFS2 中 fragment 產生的原因除了在 node align 所填入的 00x00 之外，還有當區塊中最後一個 page 沒有足夠的空間去填入下一個 node 時，會採取捨去剩餘的空間(設成 waste size)，找下一個區塊來填入。所以在這個實驗範例中，相對之下 JFFS2 產生 fragment 機會和浪費的 size 都較 YAFFS 小，所以浪費的空間也較小。不過也由於 JFFS2 的 node 可以很細小(不見得是以 page 爲單位)，比較容易產生同樣檔案散亂在不同區塊中的各處的結果，所以要用來管理紀錄 file 的 metadata 也相對的複雜，所以其所需的 metadata 的量會多一些。

在 NFTL 中由於底下掛載的系統爲 FAT32 所以會有 fat table 的使用，共有約

2000 個檔案，所以 Fat table 至少要更新 2000 次以上，所以儲存 FAT TABLE 的區塊會被不斷地串接起來。(後面會不斷看到這個策略所造成的影響，不但儲存 table 耗費固定的空間並且也是產生垃圾回收的主要原因。) 除了 FAT TABLE 的影響之外，再加上 NFTL 和 YAFFS 相同是以 page 爲單位寫入資料，所以對於那 23.4%的小檔案也會有 page size 未寫滿的 fragment 產生，因此耗費較多的可用空間。

表格 3 The experiment result of Normal case

| Normal case (file size 2Bytes~4MB) | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Total space (Kbytes) | 131072 | | |
| Free space (Kbytes) | 84738.5 | 84700.5 | 8279.3 |
| Used space (Kbytes) | 46333.5 | 46371.52 | 122792.7 |
| User data (Kbytes) | 45075.09 | 45075.09 | 45075.09 |
| Metadata (Kbytes) | 1241.8 | 1183.94 | 2669.95 |
| Fragment (Kbytes) | 16.6 | 112.5 | 75047.66 |
| % of used space is user data | 97.3% | 97.2% | 36.71% |




圖表 23. Result of Normal case

### 5.1.3.2 White-Box test

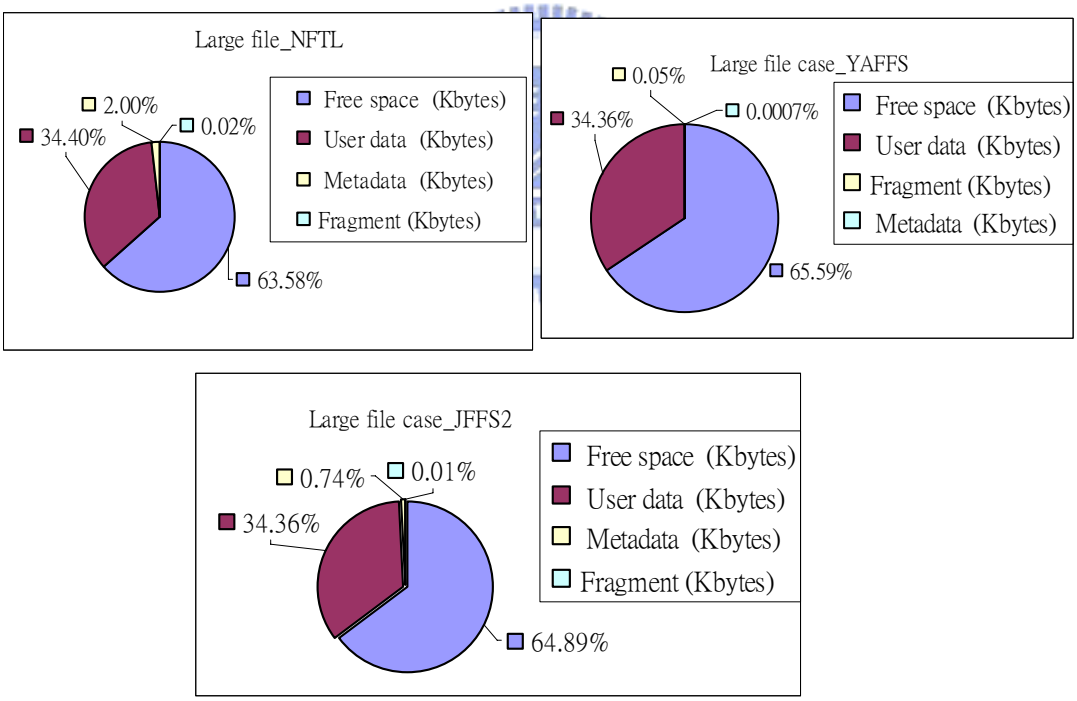
### Case.1 Large File (All Large file>page size)

實驗結果如表格 4和圖表 24所示，可以看得出來在大檔案爲多數時，YAFFS 由於使用的metadata object較少而有優勢，主要是每一個檔案只需要一個object

header(Chunk 0)，其他的都是儲存使用者 資料。而JFFS2 由於每一個raw node的大小被限制不能超出一個page，所以每一個page中都會有raw node header的overhead，所以在空間使用上略多於YAFFS2。在NFTL中由於底下掛載的系統爲FAT的影響，在必然的fat table耗費下，依舊使用了較多的實體空間來儲存相同資料。

表格 4. The experiment result of Large file case.

| Large file case size 2KB~2MB | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Total space (Kbytes) | 131072 | | |
| Free space (Kbytes) | 85055 | 85967.8 | 83337.3 |
| Used space (Kbytes) | 46017 | 45104.2 | 47734.7 |
| User data (Kbytes) | 45035 | 45032 | 45088 |
| Metadata (Kbytes) | 968.87 | 71.25 | 2625.11 |
| Fragment (Kbytes) | 16.13 | 0.94 | 21.6 |
| % of used space is user data | 97.87% | 99.84% | 94.46% |




圖表 24. Result of Large file case

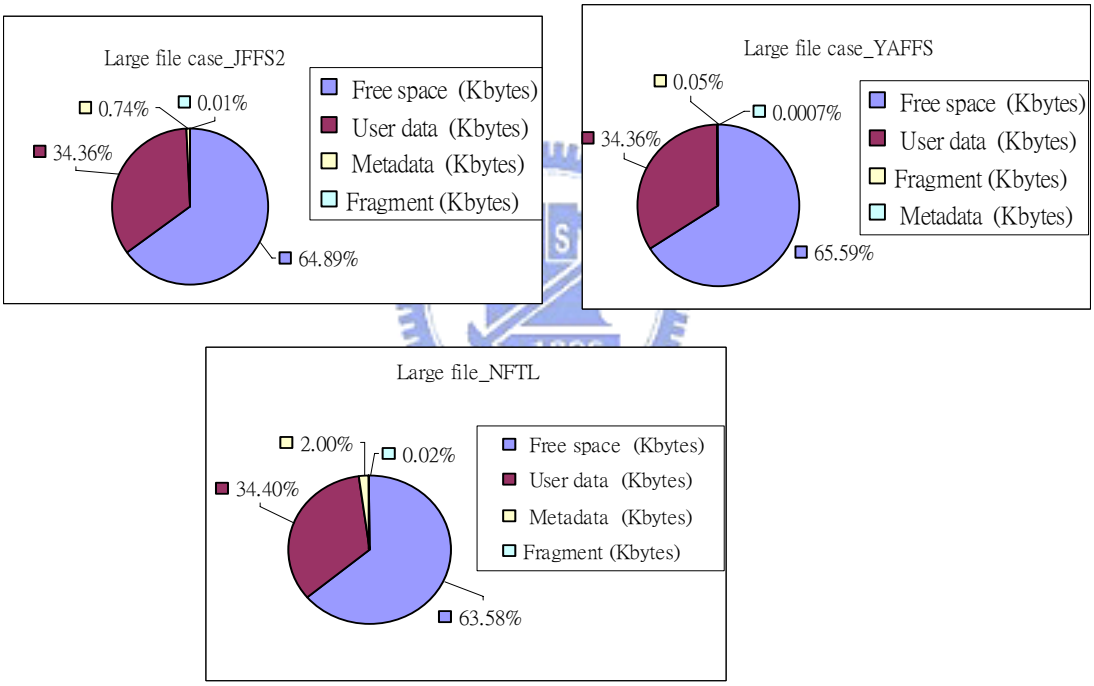
## Case.2 Very Large size file

實驗結果如表格 5和圖表 25所示，可以看出YAFFS在檔案size大且檔案數量少的情形之下，metadata所佔用的空間非常少，空間利用度非常的好，近乎 100%都可以給予使用者使用。而JFFS2 依然還是被node size給侷限著，一樣是上一個實驗範例的原因。即使是連續儲存的大檔案依然要切割成page大小node來存放，

並在每一個page都要加入一個jffs2_raw_inode header的負擔，所以比YAFFS更耗費一些空間再儲存metadata。NFTL由於FAT TABLE的關係，還是保持著相同的metadata 消耗量。

表格 5 The experiment result of Very Large file case

| Very large case (file size ) 2KB~2MB | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Total space (Kbytes) | 131072 | | |
| Free space (Kbytes) | 85033 | 86012.39 | 83385.31 |
| Used space (Kbytes) | 46039 | 45059.61 | 47686.7 |
| User data (Kbytes) | 45056 | 45056 | 45056 |
| Metadata (Kbytes) | 967 | 3.563 | 2614.152 |
| Fragment (Kbytes) | 16.01 | 0.047 | 16.55 |
| % of used space is user data | 97.87% | 99.99% | 94.48% |




圖表 25. Result of Very large file case

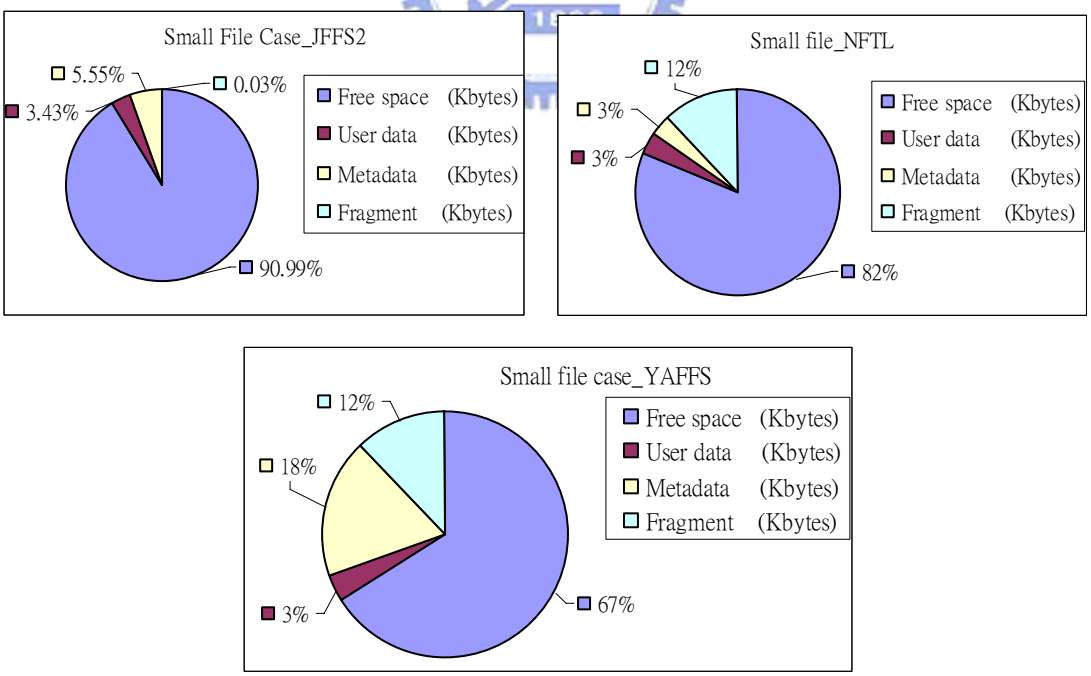
## Case.3 Small size files

實驗結果如表格 6和圖表 26所表示，由於所有的檔案都小於 512bytes，所有的檔案大小都小於一個page size。因此對於YAFFS和NFTL的影響會較爲劇烈，相反地這種小檔案的寫入，對於變相的node-level的JFFS2 是比較有利的(因爲一個page中可以塞入多個node來表示多個檔案)。我們可以從實驗結果看出YAFFS和NFTL這兩個以page-level寫入的檔案系統，所產生的fragment量相當的大，都佔了總體容量的 12%，但實際儲存的使用者 資料量只有 3%，所以產生

的fragment就大約是寫入資料量的 4 倍左右了。在metadata方面，YAFFS由於對每一個檔案都要建立一個object header(Chunk 0)的關係，等於每一個不到一個page SIZE的檔案，便需要使用兩個page去儲存。加上fragment的影響，所以使用的空間至少是存入的使用者 資料兩倍以上，加上檔案數量不少(40000 多個檔案)，所以YAFFS至少要耗費 20000Kbytes來儲存metadata，因此YAFFS在small size file 大量的寫入時，空間使用率上非常差。但JFFS2 雖然表現較好，但還是由於檔案數量太多，檔案的資料量太小(2～32Bytes)，而迫使node header的大小大於使用者資料量(jffs2_raw_indoe 和jffs2_raw_dirent分別需要 48Bytes和 52bytes)，所以導致JFFS2 空間利用度上的表現也是偏低的 38%。

表格 6 The experiment result of Small size file

| Small case (file size ) 2B~512Bytes | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Total space (Kbytes) | 131072 | | |
| Free space (Kbytes) | 119261 | 86458.42 | 106302.3 |
| Used space (Kbytes) | 11811 | 44613.58 | 24769.703 |
| User data (Kbytes) | 4491.211 | 4491.211 | 4491.211 |
| Metadata (Kbytes) | 7278.79 | 24047.17 | 4332.15 |
| Fragment (Kbytes) | 41 | 16075.2 | 15946.34 |
| % of used space is user data | 38.03% | 10.07% | 18.13% |




圖表 26. Result of Small file case

綜合以上的實驗結果，在 space utilizeation 實驗的表現中可以得知，在於儲存大容量檔案的表現上 YAFFS 表現的最爲出色，但若是系統中多數的檔案大小

小於一個 page size 的話，使用 JFFS2 會比較節省空間。

## 5.2 RAM-space Requirements

### 5.2.1 Experimental Setup and Performance Metrics

實驗的步驟大致上都是執行測試的程式後，再透過我們加入的在各個檔案系統各自的記憶體分配量計數(memory allocation count)來紀錄 log，其他像是 JFFS2 和 FAT 有使用 slab 分配器，一些需要使用的資料結構會在系統中先要求記憶體再分割成等大小的資料結構等待被使用。計算這些在系統中實際使用到的 slab 記憶體，可以由/proc/slabinfo 得知各 slab 的大小和分配的個數，之後根據這些資訊，再實驗的進行過程中，我們會每隔一段時間便將資訊寫入 log 中，之後再根據 log 檔，觀察實驗進行的過程中每單位時間記憶體使用量的變化。

### 5.2.2 Test plans

關於記憶體使用量(RAM utilization)實驗的部份，我們依舊主要劃分爲黑盒子測試和白盒子測試兩大類。其中這裡使用的黑盒子測試是採取 Postmark，而白盒子則有兩種實驗範例。分別爲範例一爲經常性更新的大容量檔案(Large file with frequency small update) 和範例二非常多的小容量檔案（Many Small Files should be written or read）。下面一一說明這兩個白盒子測試實驗的目的，以及我們準備實驗的方法。

#### 5.2.2.1 Black-Box test

**PostMark**

在這裡我們選用的 Black-Box 測試是採用 Postmark 作測試，Postmark 是相當常見用來測試檔案系統的效能評估工具，主要是模擬使用者使用網路的行爲。由於大多的嵌入式系統，如 3G 手機, PDA, Pocket PC 等大多有網頁瀏覽的功能，所以選擇使用 Postmark 作爲我們共同的黑盒子。在掛載檔案系統後，執行原始設定的 Postmark，之後再卸載檔案系統。

#### 5.2.2.2 White-Box test

**Case.1 Large file with frequency small update**

這個實驗範例只要是針對 JFFS2 所設計出來的極端行爲。再分析過 JFFS2 原始碼和觀察執行的行爲之後，我們發現在 JFFS2 中，在增加或變更檔案中一段資料時，JFFS2 便會多寫入一個 raw node 去取代舊有的資料。此時 JFFS2 至少會

在主記憶體中爲這個新的 raw node 多建構兩個資料結構, jffs2_raw_node_ref 和 jffs2_full_dnode ，而此時檔案爲讀入狀態，爲了建構 frag tree—用於判別最新的檔案版本，又會多建構一個 jffs2_inode_frag 的資料結構 (可參考圖表 11)。因此我們推測 JFFS2 在一個大檔案經過不斷的更新/複寫資料，會使原本大的 raw node 分裂成許多細小的 raw node。之後讀取這個碎裂成許多細小 raw node 的大檔案時，會消耗相當龐大的記憶體空間。爲了測試這個情形下消耗的記憶體空間是否非常巨大，便設計了這個實驗。首先我們在 nandsim 上掛載測試的檔案系統 (JFFS2,YAFFS,NFTL+FAT32)，之後先寫入一個大小 16MB 的檔案，再利用自定的程式不斷對它進行 100,000 次的隨機微小更新(random small update)，來達到使大檔案碎裂的目的，在步驟完成之後在卸載測試的檔案系統，然後觀察實驗期間檔案系統使用記憶體的數量。

### Case.2 Many Small Files should be written or read

也是針對 JFFS2 所設計，理由也等同範例一中提到的，由於 JFFS2 每一個 raw node 都必須在主記憶體中建構至少兩個資料結構, jffs2_raw_node_ref 和 jffs2_full_dnode。因此隨著檔案的數量增多，消耗的記憶體空間也會隨之上升。其中爲極端的例子，也就是檔案的數量多且都爲小檔案時，便可能會發生爲了建構每一個小檔案的位址轉換的關係，使得檔案系統所消耗的記憶體比檔案大小本身還要大的情形(每一個 raw node 在記憶體中最少需要 jffs2_raw_node_ref + jffs2_full_dnode=56bytes 的空間)。我們實驗的方法是在乾淨的 nandsim 上掛載待測的檔案系統之後，再利用之前 5.1.3.2 White-box Test 中 Case c Small size files 相同的樣本寫入到測試的檔案系統上，完成之後再卸載檔案系統，並觀察和紀錄寫入到卸載這期間記憶體的變化。

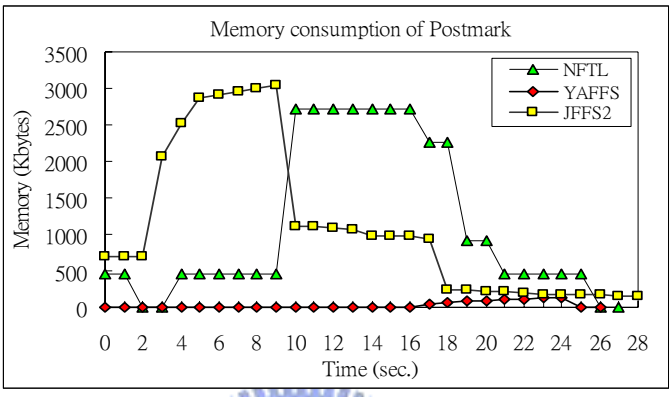
## 5.2.3 Numerical Result

### 5.2.3.1 Black-Box test

### PostMark

實驗環境是在nandsim之上，利用nandsim模擬 128MB的NAND 快閃記憶體，並在掛載有JFFS2,YAFFS和NFTL的nandsim上執行Postmark。Postmark的參數是採取原本的預設值，實驗結果如圖表 27 Memory consumption of Postmark所表示。由於postmark是模擬使用者瀏覽網頁的行爲，一開始會開始產生許多細小檔案以備之後來作read/write。所以可以看到在不斷產生檔案的同時，JFFS2 由於需要不斷建造jffs2_raw_node_ref, jffs2_full_dnode和jffs2_node_frag等 3 種資料結構，所以不斷的消耗記憶體空間，直到程序進行到刪除後才開始慢慢的釋放出memory空間。而YAFFS開始也是由於檔案建立的關係慢慢的增加RAM使用量，直到process結束後，才慢慢平穩不再增加，而YAFFS由於釋放記憶體的機制較不

同於另外兩個檔案系統，會利用到一定的RAM使用量或是進行垃圾回收和卸載(umount) 檔案系統之後，才釋放記憶體空間，所以會有上升後再平穩的曲線。而NFTL雖然只利用了兩個固定大小的表格(只與NAND快閃記憶體大小和區塊數量有關)，但掛載FAT32 之後會在主記憶體中替每一個檔案都建造FAT的entry，每一個entry大小約 4Kbytes，因此RAM使用的程度和JFFS2 類似，都會依照檔案系統中管理的檔案的個數來決定使用量的多寡，因此也可以看到NFTL+FAT32的曲線表現也是呈現出一個起伏的線條，差別在於程式執行時耗費的時間長短不一。




圖表 27 Memory consumption of Postmark

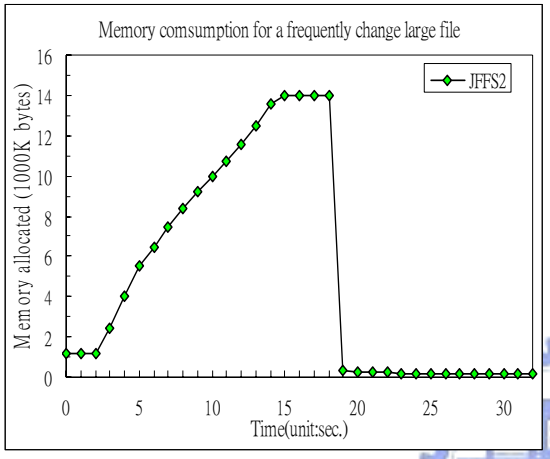
### 5.2.3.2 White-Box test

### Case.1 Large file with frequency small update

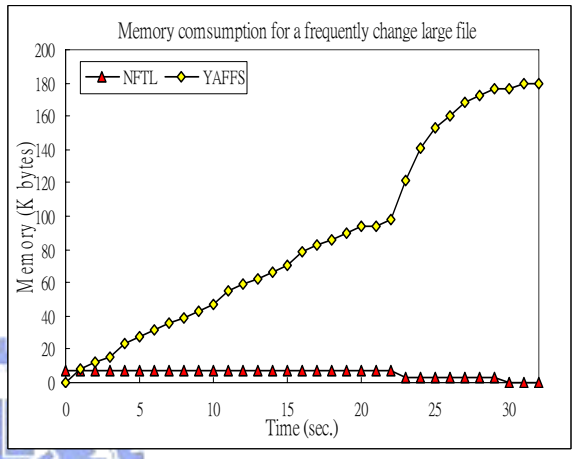
首先第一個 White-Box Way 是針對 JFFS2，因爲由 Chapter 4.2.1 JFFS 中可以得知，每一次檔案的更新都必須要在主記憶體中新增 jffs2_raw_node_ref, jffs2_node_frag 和 jffs2_full_dnode 這三種資料結構。因此我們認爲在 JFFS2 中，當一個大型檔案頻繁地作細小的更新時的情況下，會產生出許許多多的 raw node。所以在這個實驗範例中，我們利用一個 16MB 的檔案，不斷地隨機的在檔案中更新定量且細小資料。目的在製造原有儲存檔案的 raw node 破碎和分裂，利用上述的方式模擬出這樣的實驗範例，來測試 JFFS2 是否會在此種情形下佔據大量的記憶體。

實驗結果如圖表 28和圖表 29所示，可以看出隨著細小更新(Small update)的次數增加，JFFS2 所消耗的RAM空間也就越大，表示開啓檔案時資料分散儲存的碎裂程度越嚴重(raw node數量變多，相對應存放在RAM資料結構也多)，要建立的frag tree也就越龐大。可以看出到最後開啓這一個 16MB的檔案時，最大需要耗費約 14MB的RAM來建立相關的位址轉換的資料結構。這是相當驚人的，尤其在嵌入式系統的環境之下也不見得有那麼大的RAM空間可以使用，因此突顯出JFFS2 在使用記憶體上得需求度是相當大的。在YAFFS方面，因爲細小更新的

關係，每隔一段時間就必須重新先寫入object header (chunk 0)的緣故，會去一直allocate yaffs_object的資料結構來進行寫入。隨著時間的增加，使用的記憶體也慢慢的增加，不過整體使用的RAM空間也不到 200Kbytes，遠小於JFFS2 佔用的記憶體。NFTL由於是使用固定的大小的兩張表格: ENUtable和RepUnitable在管理，而底下掛載的檔案系統FAT32 中影響RAM使用量的是檔案個數。但在這個實驗中只有一個大檔案，因此只需要在記憶體中分配一個fat_cache的entry，所以RAM使用量不高，也不會隨著程式執行和更新資料的次數變動，維持在一個固定的值。




圖表 28. Memory consumption (JFFS2)




圖表 29. Memory consumption (YAFFS/NFTL)

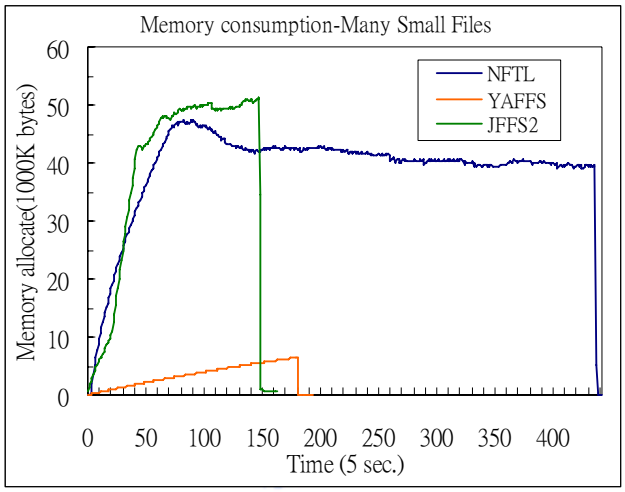
## Case.2 Many Small Files should be written or read

在這個白盒子測試中，主要是要測試 JFFS2 和 YAFF 在檔案數目龐大的情形之下，開啓和讀進所有檔案所需的記憶體空間。首先實驗範例等同於 Chapter 5.1.2.3 White-Box Tests 的 Case.3 Small size files 的樣本，我們將許多細小的小檔案分別寫入到這三個檔案系統中，並觀察在耗費的時間中，三個檔案系統記憶體消耗程度的過程。

在圖表 30的實驗結果中，首先JFFS2 如同我們想像地耗費大量的記憶體。由圖表 30 明顯的看出JFFS2 會隨著時間的增加，不斷地消耗著記憶體空間，主要原因還是那 40000 個小檔案至少也會各產生 40000 個jffs2_indoe_info, jffs2_indoe_cache, jffs2_raw_node_ref, jffs2_node_frag和jffs2_full_dnode這 5 種資料結構，而JFFS2 的各種資料結構是利用slab allocator預先分配的，加上一些其他儲存在superblock上的資訊，至少也需要 20MB左右的記憶體來處理這個程序。實驗結果也正如我們所預料的約消耗了 22MB空間。不過雖然耗費的主記憶體空間很大，但是由於多個檔案產生的jffs2_raw_inode可以被page cache緩衝(buffer)後，再寫入快閃記憶體上同一個頁面中的影響，讓整體的寫入次數減少，所以整個程序執行使用的時間只有YAFFS的 1/3。

YAFFS 隨著檔案的增加，使用的記憶體使用量呈現線性成長，每加入一個檔案都只要新增加一個 yaffs_object 和 2 個 yaffs_tnode (chunk 0 和資料)，雖然也是增加兩個資料結構，但不像 JFFS2 是使用 slab allocator 事先分配好記憶體等待被使用，而是採取需要再 allocate 的方式。時間上由於對應程度(mapping level)是以頁面爲單位，並非像 JFFS2 寫入爲 node-level 的關係(每一個檔案觸發一次寫入)，實際讀寫的速度較 JFFS2 慢上一些。

NFTL+FAT32 由於寫入的檔案個數相當龐大，對每一個寫入的檔案都需要建立一個 fat_cache 的 entry，所以隨著讀寫檔案的個數增加，花費的記憶體也隨著增加，但每次更新檔案都需要直接寫入到快閃記憶體中，導致整體花費的時間是 JFFS2 的 3 倍以上。(圖表 30.)



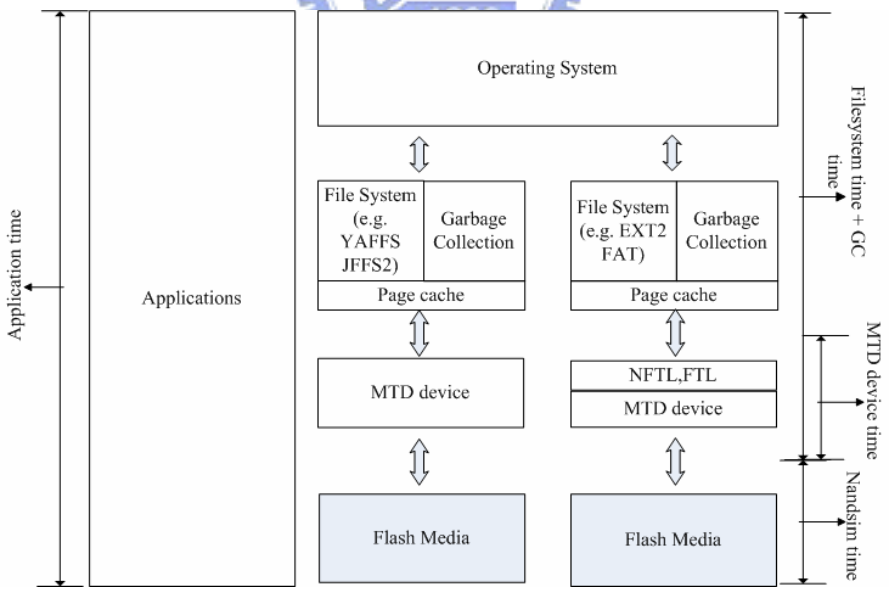

圖表 30. Memory consumption - Many Small Files should be written or read

綜合以上的實驗結果，在 Memory utilization 實驗的表現中可以得知，雖然 JFFS2 在處理檔案的執行動作上比較迅速，但由於使用來作位址轉換的資料結構太多，導致會消耗掉大量的記憶體空間，消耗的記憶體空間的大小主要是和檔案系統中儲存的 raw node 數量有關和儲存資料大小無關，例如：利用不同數量的 raw node 數去儲存同樣的資料內容，消耗的記憶體也就隨之不同。YAFFS 對於記憶體的使用量上相當的節制，在 3 個檔案系統中表現示最亮眼的。YAFFS 使用記憶體消耗的程度和檔案系統中的檔案個數成正相關，即檔案數量越多消耗的記憶體越多，不過使用的記憶體量還是遠遠的少於 JFFS2。另外檔案大小也會些微的增加 YAFFS 記憶體的使用量，但影響不大，只要檔案大小固定，那麼消耗記憶體容量爲定值。而 NFTL+FAT 的組合，雖然 NFTL 本身只處理邏輯位址和實體位址的轉換，只建構了兩個主要的表格，但由於 FAT32T 的組合，雖然 NFTL 本身只處理邏輯位址和實體位址的轉換，只建構了兩個主要的表格，但由於在 FAT32 的檔案系統下，建構一個檔案就需要消耗一個 FAT_ENTRY (4Kbytes)，所以記憶體消耗的程度和檔案系統中的檔案個數有關，不受檔案大小影響。

## 5.3 Garbage Collection

### 5.3.1 Experimental Setup and Performance Metrics

實驗紀錄的時間是利用 Rdtsc.h 來紀錄兩個區間所花費的 cpu cycle 再換算成實際時間，而讀/寫/抹除次數都是再 nandsim 模組中安插入我們的計數參數來計算的。簡要的實驗模組大致如下圖表 31 所示，我們量測的時間爲主要的 Application time、filesystem time、GC time 以及 MTD device time(上述 4 個時間計算是沒有使用 nandsim delay，所以純粹是計算出模擬動作時所花的 cpu cycle，其中 MTD device time 是 File system+GC time 的一部分。而最終實驗數據中的 other time=Application time-(filesystem time+GC time))，最後還有實際花費在 nandsim 上的 NANDsim device time。而量測的數據除了上述的時間外，還有計數實驗過程花費的基本的讀/寫/抹除的次數。主要的步驟我們會利用安插在檔案系統中的參數累計花費在檔案系統本身，垃圾回收部份以及耗損在 MTD Device 層的時間和實際在 nandsim 上執行動作的時間，計算出在每一個階段花費的時間。爲了快速進行實驗，在後期部份我們取消 nandsim delay 時間的機制，利用參數累計來得到耗費的時間，其中美中不足的地方，是無法分別判別出花費的讀/寫/抹除次數哪些是由檔案系統本身引起的哪些是垃圾回收策略所觸發的，但對於實驗的速度提昇有非常大的幫助。




圖表 31. Garbage collection 的實驗模組

### 5.3.2 Test plans：

首先在垃圾回收這一部份要量測的數據，大致上爲垃圾回收花費的時間、消

耗的 erase cycle，讀/寫的次數。下面我們會介紹我們評比垃圾回收的實驗範例，黑盒子測試，還是採用 PostMark。而白盒子測試方面，有兩個實驗範例，範例一是循序寫入，範例二是交錯寫入大檔案和小檔案。

#### 5.3.2.1 Black-Box test

**PostMark**

在黑盒子測試上，我們依舊是採用 Postmark 作爲黑盒子測試的範例，但是再配合[1]得知，Linux 系統中最常見的檔案大小範圍 128-256K，因此對 Postmark 的執行參數中的 file size 作調整後，再連續執行 Postmark 10 次。共計寫入資料量爲 1073.5MB，大約是模擬的 nandsim 大小的 9 倍左右，藉此觸發一定數量的垃圾回收的動作，方便我們紀錄和觀察。

#### 5.3.2.2 White-Box test

**Case.1 Sequential Write Big file**

第一個白盒子測試的例子，我們採用常見的循序寫入，藉此觀察各個檔案系統的垃圾回收情形。我們一次寫入的檔案的資料總合容量大小爲 104MB，之後刪除寫入的檔案後再進行同樣檔案的寫入，總共寫入 10 次，總寫入資料量約爲 1040MB，遠大於模擬快閃記憶體 nandsim 的 128MB，所以垃圾回收的動作必定要能清出空間以供寫入，主要作爲 Case.2 Interleaving Write Big/Small file 的對照參考。

**Case.2 Interleaving Write Big/Small file**

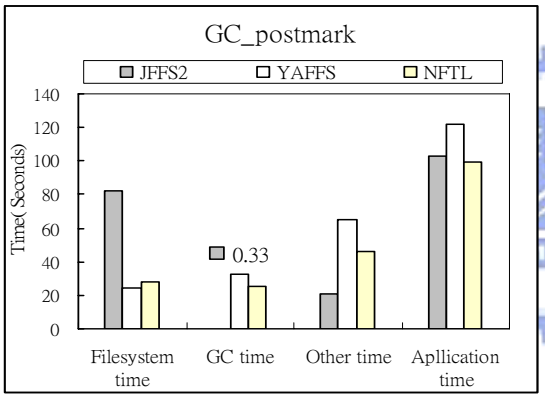
而這個白盒子測試，我們主要是想產生出讓檔案系統難以處理垃圾回收的情況，也就是每個區塊中都混雜著相似比例的有效資料和無效資料，讓但系統在處理垃圾回收時，一定要會有搬動有效資料的動作，而且每個區塊無效資料的比例差異不大，讓檔案系統再挑選垃圾回收的目標區塊時也需要耗費不少比較時間。實驗設計上，我們首先交錯的寫入一個 100MB 的大檔案和一個 6K 左右的小檔案(每一次就寫入 10K 的大檔案資料和 6K 個小檔案)，之後再循序的更新大檔案(每次更新 10K 的資料)和不斷更新 6K 的小檔案。由於我們使用的區塊大小爲 16K，所以希望每次寫入都能形成區塊中有大檔案和小檔案的資料，而由於小檔案會不斷的更新，所以上一個寫入區塊中的小檔案資料很快地就形成的無效的資料，借此產生有效資料和無效資料混雜的例子。

### 5.3.3 Numerial Result

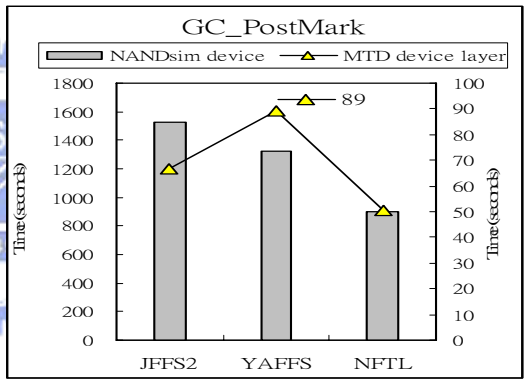
#### 5.3.3.1 Black-Box test

## PostMark

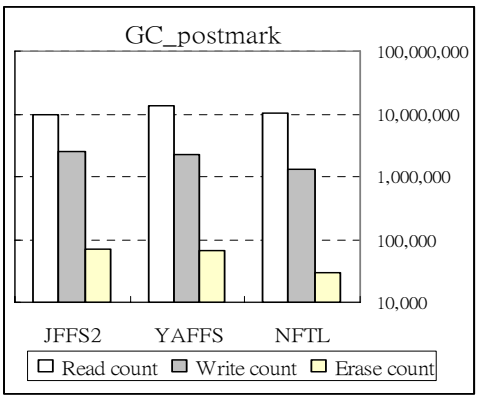
在 PostMark 的實驗中，我們主要是利用 Postmark 並將程式產生的檔案大小設定在 UNIX 中的最常出現的檔案大小區段(128-32Kbytes，參考[1])，由於 Postmark 一旦設定後產生的執行動作和檔案大小便會固定。NFLT 因爲傳遞的階層較多的緣故，實際花費在 cpu 上的時間也較多。但 NFTL 實際在 nandsim device 上花費的時間(圖表 33..)和讀/寫/抹除次數(圖表 34)都相當的少，這是由於重複的讀寫動作反應在 NFTL+ FAT32 上會被 page cache 吸收掉，導致 NFTL+FAT32 的讀/寫/抹除次數減少，花費在 nandsim 時間也就減少。JFFS2 部份，因爲 JFFS2 平均抹除的影響，使得 JFFS2 的額外負擔(讀/寫/抹除)的次數都較多，所以花費的時間也較多，而 JFFS2 花費的 filesystem time 較多，主要是由於其檔案系統在記憶體中建構的資料結構複雜，所以處理 postmark 增刪檔案的動作也耗費較多時間，但 JFFS2 在挑選垃圾回收的目標區塊時只是機率性的從多個串列中某一個串列挑出串列前端的區塊，因此實際花費的時間不多(圖表 32.所表示 JFFS2 花費的 GC time 只有 0.33 秒)。




圖表 32 PostMark time performance (Garbage collection)




圖表 33 .NANDsim device and MTD device layer's time performance in PostMark (Garbage collection)

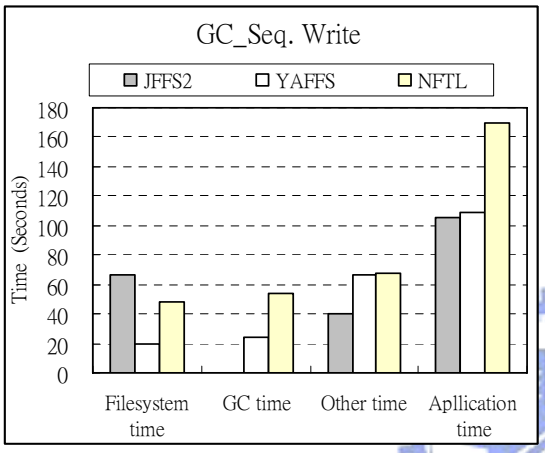



圖表 34. PostMark r/w/e count (Garbage collection)

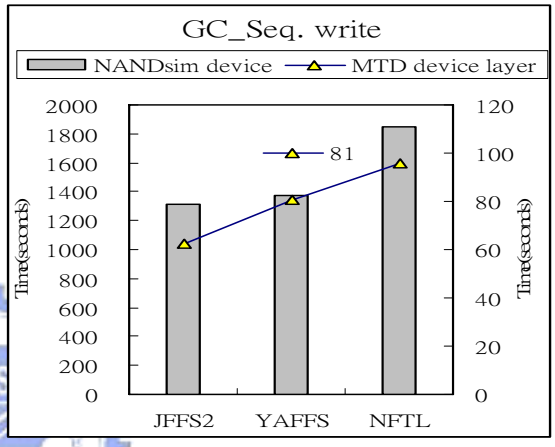
### 5.3.3.2 White-Box test

## Case.1 Sequential Write Big file

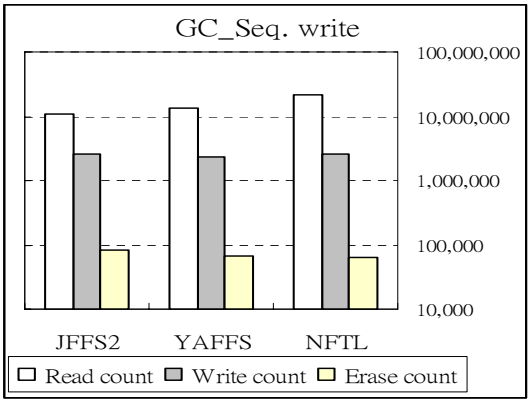
在循序寫入的測試實驗中，由於是循序更新檔案資料的關係，而YAFFS和NFTL的垃圾回收策略是dynamic garbage collection，會挑選變動過的資料區塊來作垃圾回收的動作，所以很輕易的挑選出目標區塊來作垃圾回收，而NFTL中由於記憶體中的對映關係是Block-level的，所以每一次變更資料時，都要讀取整個virtual unit chain的區塊的資料一次，方便找到目前的最新版本，所以花費的讀取次數較多(圖表 37)，導致整體的時間增加(圖表 35-圖表 37)。JFFS2 中由於另外要作平均磨損的關係，所以耗費的erase cycle較多(圖表 37)。




圖表 35. Sequential write time performance (Garbage collection)




圖表 36. NANDsim device and MTD device layer's time performance in Seq. write (Garbage collection)

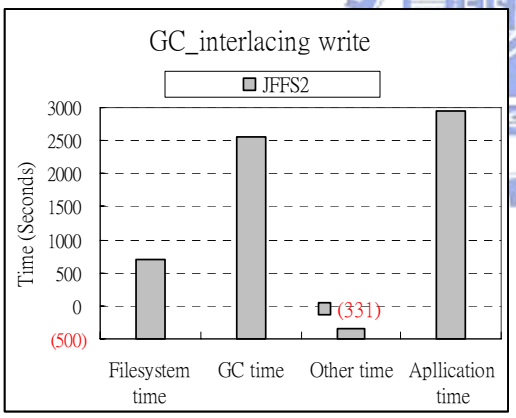



圖表 37. Sequential write r/w/e count (Garbage collection)

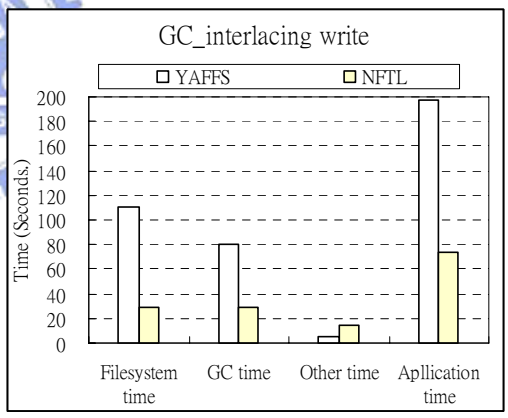
## Case.2 Interlacing Write Big/Small file

在交錯寫入這裡這一部份，JFFS2 花費的時間遠大於另外兩個檔案系統(由圖表 38,39 可知)，但實際跑到MTD device Layer的時間卻沒有那麼多(圖

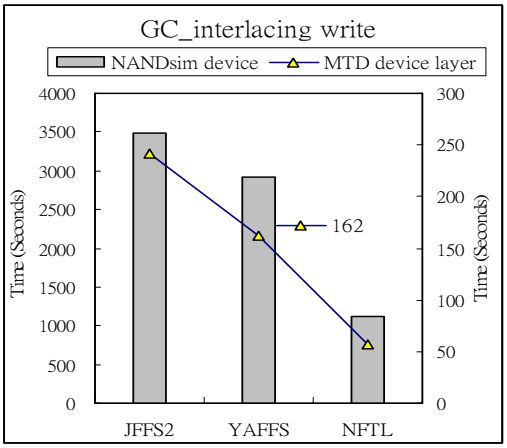
表 40)。主要是由於每次的寫入都迫使JFFS2 要去更動記憶體中的階層架構，所以JFFS2 花費的時間大多在處理記憶體的分配使用和記憶體中檔案目錄的資料結構變更。而在垃圾回收花費大量時間的部份，只要是由於JFFS2 要讀取到快閃記憶體上的資料，獲得實際node type的狀態，假設此要回收的node不在記憶體中，則可以直接讀取拷貝到新的區塊上。若要回收的node有存在記憶體中則要建立檔案的frag tree，來判別檔案的新就在作拷貝搬移的動作。而且從章節 5.2.3.3 的實驗結果可知，JFFS2 再開啓大檔案時消耗的記憶體非常龐大，所以再消耗掉大量的記憶體之下，可以得知系統在執行其他讀/寫動作會相當的緩慢。再從實驗的角度和內容來看，JFFS2 花費的讀/寫/抹除次數各是另外兩個檔案系統 2 倍以上(圖表 41)，因此花費的時間也較YAFFS多。而在JFFS2 中的other time 出現負數，這是因爲實驗的過程中jffs2_gc_thread啓動和每一次計算GC time的實驗誤差累計而成。NFTL則是由於寫入的檔案只有兩個，所以在NFTL中會被LBA侷限在一定得範圍內(參考圖表 17)，使得經常被更動的熱資料會被限定在同樣的LBA中，因此垃圾回收執行起來非常的輕鬆，總是可以挑選到全是dirty page的區塊，加上page cache吸收掉部份的讀寫動作，所以再nandsim device上實際執行的讀/寫/抹除動作少，整體耗費的時間也就少。



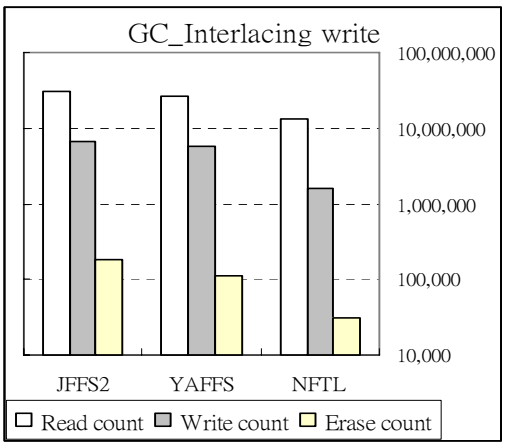

圖表 38. JFFS2 , Interlacing write time performance (Garbage collection)




圖表 39. YAFFS and NFTL , Interlacing write time performance (Garbage collection)




圖表 40. NANDsim device and MTD device layer's time performance in Interlacing write (Garbage collection)




圖表 41. Interlacing write r/w/e count (Garbage collection)

## 5.4 Wear-leveling

### 5.4.1 Experimental Setup and Performance Metrics

在 Wear-leveling 的實驗中，我們依舊是利用 proc 檔案系統來取得我們在各個檔案系統中的設立的計數參數，在這我們主要的是得到各個檔案系統的區塊抹除次數以便計算個別的抹除次數的平均值，標準差，以及最大值和最小值。

### 5.4.2 Test plans：

在平均磨損的實驗中，主要觀測的數據爲每個區塊的磨損次數(Erase cycle)的分佈情形和磨損的程度是否有達到平均，還有額外的磨損次數是否會很多，所以在平均磨損的實驗觀念中我們會著重觀察磨損次數的變化和分佈情形。所以在之後的實驗我們依然是使用 Postmark 作爲我們黑盒子測試工具，在白盒子的部份則是量測範例一：循序寫入(Sequential Write)，和範例二：模擬快閃記憶體約 90%的空間是存放靜態資料，只有剩餘約 10%的空間可供使用的情形 (Static/Dynamic file ratio 90/10)。

#### 5.4.2.1 Black-Box test

**PostMark**

在實驗中，我們將 Postmark 的執行參數中的 file size 作修改，配合[1]將檔案大小設定在 Linux 系統中最常見的檔案大小範圍 128-256K，再之後連續執行 Postmark 10 次，共計寫入資料量爲 1073.5MB，大約是模擬的 nandsim 大小的 9 倍左右，藉此觸發一定數量的抹除的動作，方便我們紀錄和觀察。

### 5.4.2.2. White-Box test

### Case.1 Sequential Write

採用和章節5.3.2 中White Box tests的Case.1 Sequential Write Big file的相同設定，主要是作爲 Case.2 Static/Dynamic file ratio 90/10 的對照範例。由於循序寫入的動作，會迫使每個檔案都被依序更新到，因此沒有靜態資料的部份，所以我們的預想猜測在這個例子中，3 種檔案系統的平均磨損的方法都應該可以發揮效用。

### Case.2 Static/Dynamic file ratio 90/10

由[1]的研究所知，在一般的使用者電腦上通常有 90%寫入動作會落在 10%的儲存空間上，而且經常更動的資料量只佔總體資料量的 10%左右。因此我們便想創造出類似的使用狀況，來分析 JFFS2、YAFFS 和 NFTL+FAT32 這 3 種快閃記憶體檔案系統使用區塊的情形。首先我們會先寫入 100MB 的靜態資料，之後再剩餘的空間中，寫入一些經常會變更的小檔案(1K-32K)，再對這些小檔案不斷地作更新，直到寫入的資料量達到 1000MB 以上。由於有約 90%資料是不會更動的靜態資料，所以再不斷的更新之下，會導致抹除的區塊只集中再剩下的 10%的區域中。

### 5.4.3 Numberical result

### 5.4.3.1.Black-Box

### PostMark

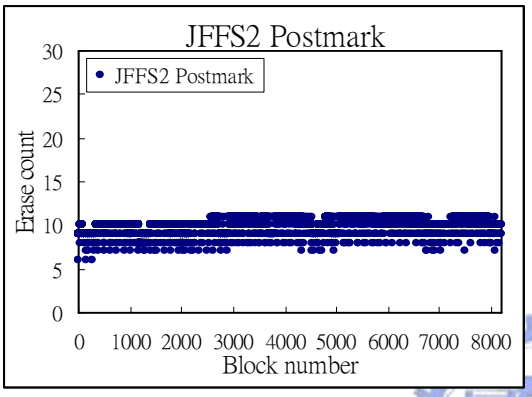
在實驗中，我們將 Postmark 的執行參數中的 file size 作修改，配合[1]將檔案大小設定在 Linux 系統中最常見的檔案大小範圍 128-256K，再之後連續執行 Postmark 10 次，共計寫入資料量爲 1073.5MB，大約是模擬的 nandsim 大小的 9 倍左右，藉此觸發一定數量的抹除的動作，方便我們紀錄和觀察。

實驗的數據如下表格 7. Wear-leveling Experiment result (PostMark)所示，各區塊抹除次數的分佈如圖表 42-圖表 44所示。在數據方面看來YAFFS的表現較優異，不但平均值較低，且大小分佈也很平均。這主要是由於Postmark是利用循序創造和寫入檔案，再進行刪除的動作來模擬網頁的讀取動作。當這樣的動作不斷重複執行之下，行爲非常相似循序寫入大量資料的動作，所以依照YAFFS的垃圾回收策略下，只需要一個個循序的挑選block來作抹除即可。而JFFS2 的平均值與標準差都略大於YAFFS，應該是由於JFFS2 夾帶的node header造成的影響，造成垃圾回收時的額外的負擔，導致JFFS2 的額外抹除的次數較高。在NFTL中，由於有NFTL的影響，加上postmark的設定和動作全都相同，使得page cache得以

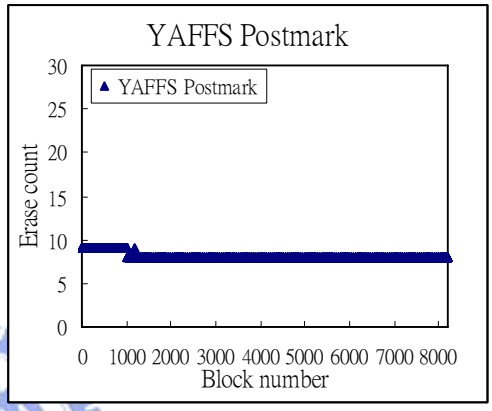
吸收掉大部分的讀寫動作，導致直接寫入到NFTL的次數便少，所以觸發的抹除次數便下降。

表格 7. Wear-leveling Experiment result (PostMark)

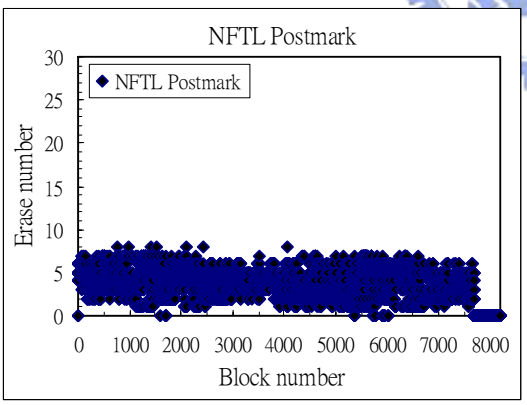
| PostMark | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Average | 10.14 | 8.13 | 3.48 |
| Stand deviation | 0.85 | 0.33 | 1.78 |
| MAX | 11 | 9 | 8 |
| MIN | 6 | 8 | 0 |




圖表 42. Erase Distribution after PostMark (JFFS2)




圖表 43. Erase Distribution after PostMark (YAFFS)




圖表 44. Erase Distribution after PostMark (NFTL)

### 5.4.3.2 White-Box test

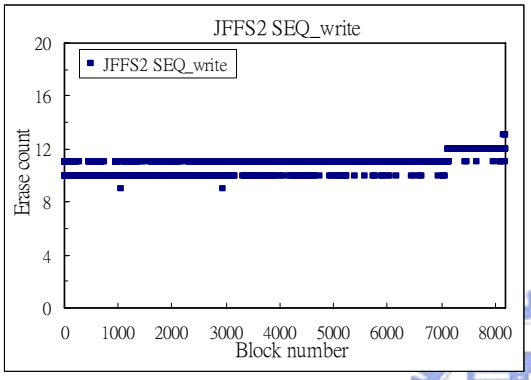
## Case 1. Sequential write.

循序寫入這邊主要是測量在大量檔案循序寫入的情形，我們利用多個大於1MB的檔案不斷的進行copy和delete的動作。這邊表現的成果和postmark大致相同，主要也是由於postmark的行爲，也類似於大量資料的循序寫入，所以JFFS2

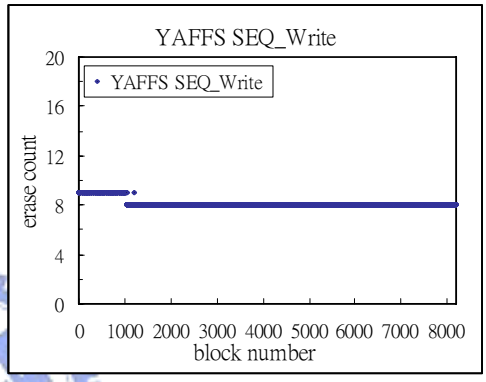
和YAFFS的表現也和postmark的實驗裡的反應相去不遠(圖表 45-圖表 47)。而NFTL則是由於檔案容量大無法被page cache完整的吸收掉，加上檔案全都會被依序的更改，所以dynamic Wear-Leveling得以發揮作用，平穩地依序抹除使用過的區塊(圖表 47)。

表格 8. Wear-leveling Experiment result (Sequential write)

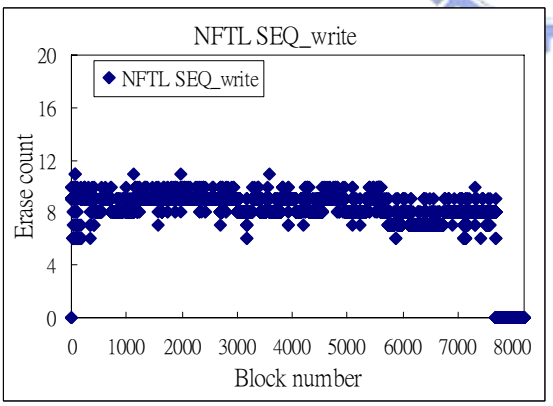
| SEQ. write | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Average | 10.90 | 8.126 | 7.563 |
| Stand deviation | 0.59 | 0.332 | 2.65 |
| MAX | 13 | 9 | 12 |
| MIN | 9 | 8 | 0 |




圖表 45. Erase Distribution after Sequential write (JFFS2)




圖表 46. Erase Distribution after Sequential write (YAFFS)




圖表 47. Erase Distribution after Sequential write (NFTL)

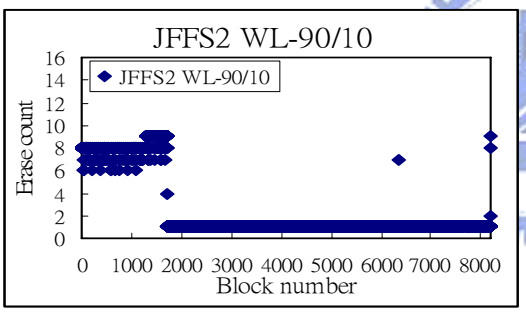
## Case2. Static/Dynamic file ratio 90/10

static和dynamic這邊所要測試的構想，是由於[19]的研究可得知，其實大部份的使用者電腦中，大部分的儲存空間(90%)都是存放較少更動的cold data,而大部分的寫入多落在剩餘的空間(10%)中。我們在 128MB的快閃記憶體中放入100MB的資料且不去更動它，之後再剩餘的空間中不斷地更新資料和寫入資料，

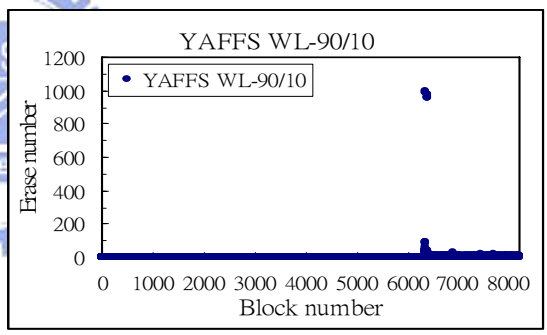
模擬出類似上述行爲的範例。由下列圖表 48-圖表 50的實驗結果可以看出各個快閃記憶體的表現大多是相同的結果，都擁有一長大段的區是幾乎沒有被抹除或是只抹除了 1-2 次。其中JFFS2 抹除數的平均值是最平均的，只要是由於 99/1 的策略所造成，讓MAX和MIN的比率大概在 9 比 1 的情形。而YAFFS中會發現有幾個區塊的抹除次數特別的多，這是由於在我們實驗的test file中混雜著有大於且等同於block size倍數的檔案，也有小於Block size的檔案，這讓YAFFS再挑選犧牲者區塊的範圍內，會較容易挑選到這等同於block size倍數的檔案所接觸過的區塊(整個區塊爲dirty)，成爲垃圾回收的目標，因此造就了幾個抹除次數特別高的區塊。NFTL中由於並沒有特別WL策略，也無視區塊中是否有剩餘的valid檔案，只是純粹的找尋最長的區塊串列來執行垃圾回收，所以抹除的次數多且效率差。

表格 9. Wear-leveling Experiment result (Static/Dynamic 90/10)

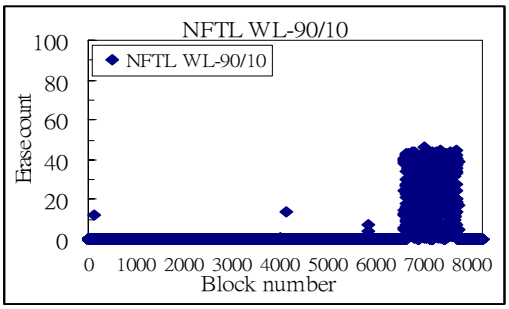
| Static/Dynamic space 90/10 | JFFS2 | YAFFS | NFTL |
|---|---|---|---|
| Average | 2.51 | 1.652 | 2.972 |
| Stand deviation | 2.93 | 30.48 | 9.14 |
| MAX | 9 | 992 | 46 |
| MIN | 1 | 0 | 0 |




圖表 48.Erase Distribution on Static/Dynamic 90/10 space (JFFS2)




圖表 49.Erase Distribution on Static/Dynamic 90/10 space (YAFFS)




圖表 50.Erase Distribution on Static/Dynamic 90/10 space (NFTL)

綜合以上的實驗結果，在平均磨損實驗的表現中可以得知，除了 JFFS2 使用 99/1 的法則選擇垃圾回收的目標區塊之外，另外兩個檔案系統只做 Dynamic Wear

- leveling。若是儲存的資料皆爲動態，變動性很大的檔案時，三者平均磨損效果沒有什麼太大的差別。但當系統中靜態資料爲大多數時，Dynamic Wear-leveling 不會去更動到儲存靜態檔案的區塊空間，導致選擇來作抹除的目標區塊會集中在那些不斷變更儲存資料的區塊上，所以 Wear-leveling 的效果不彰。JFFS2 由於 99/1 的法則，效果也是有限，只能維持抹除次數最大和最小的比值約爲 9：1 狀態，比其他兩檔案系統略好一些。所以整體來說 3 種檔案系統 wear-leveling 的效果都不近理想。

## 6. Conclusion

以快閃記憶體爲基礎的儲存系統目前已經是嵌入式儲存系統的主流。本論文的目的是透過設計新的效能評比方式，來測試突顯各個快閃記憶體檔案系統設計上的優勢與劣勢。測試評比對象爲目前可公開取得之 JFFS2, YAFFS, 以及 NFTL。評比對象包含快閃記憶體原生檔案系統以及區塊裝置模擬系統。而評比的項目，主要以快閃記憶體管理議題，以及嵌入式系統的特徵爲主。。包括了快閃記憶體空間的使用率，RAM 空間的使用率，垃圾回收的效能，以及平均磨損的效能。除了一般常用典型的檔案系統評比程序，一些設計來暴露設計缺陷的測試程序也被採用。透過新的評比程序，我們發現了既有快閃記憶體檔案系統一些以往並沒有被廣爲周知的設計缺陷。本論文的數據結果，是除了顯現一般典型工作量下的效能之外，也希望能告訴讀者在哪種工作量下盡可能的避免使用哪些檔案系統，以免陷入成效不彰且效能低落的情形。

## Reference


**[1] G. Irlam, “Unix File Size Survey,” 1993. http://www.gordoni.com/ufs93.html**

**[2] A. Inoue and D. Wong, "NAND Flash Applications Design Guide" , April, 2003.**

**[3] Li-Pin Chang and Tei-Wei Kuo, "Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation," ACM Transactions on Storage, Volume 1, Issue 4, 2005.**

**[4] Intel Corporation, “Understanding the Flash Translation Layer (FTL) Specification".**

**[5] SSFDC Forum, "*SmartMedia*™ Specification", 1999.**

**[6] Compact Flash Association, “*Compact Flash*™ 1.4 Specification," 1998.**

**[7] Linux MTD project and M-System, “NAND Flash Memory Translation Layer (NFTL).”**

**[8] Nitesh Goyal and Rabi N Mahapatra "Energy Characterization of CRAMFS for Embedded Systems", Proceedings of International Workshop on Software Support for Portable Storage (IWSSPS), March 2005.**

**[9] Siddharth Choudhuri, Rabi Mahapatra, "Energy Characterization of**

**Filesystems for Diskless Embedded Systems", Design Automation Conference (DAC), June 2004.**
**[10] Aleph One Company, "Yet Another Flash Filing System (YAFFS) "'**
**[11] L. P. Chang and T. W. Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," 8th IEEE RTAS, September 2002, pp. 187-196**
**[12] Han-joon Kim and S. Lee, "A New Flash Memory Management for Flash Storage System.", In Proceedings of the 23rd Annual International Computer Software and Applications Conference, pages 284–289, 1999.**
**[13] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J., A Trace- Driven Analysis of the UNIX 4.2 BSD File System, Proceedings of the Tenth Symposium on Operating Systems Principles, Orcas Island WA, December 1985, pp. 15-24.**
**[14] Atsuo Kawaguchi et al., "A Flash-Memory Based File System", in USENIX Technical Conference, 1995**
**[15] Linux MTD Project, "Journaling Flash File System (JFFS), Journaling Flash File System 2 (JFFS2), and Journaling Flash File System 2 (JFFS3)."**
**[16] Bray, T, "The Bonnie home page.", http://www.textuality.com/bonnie, 1996.**
**[17] Katcher, J., "PostMark: A New Filesystem Benchmark. Tech. Rep. TR3022, Network Appliance.", www.netapp.com/techlibrary/3022.html ,1997**
**[18] William D. Norcott., Don Capps, "Iozone Filesystem Benchmark", http://www.iozone.org/docs/IOzone_msword_98.pdf**
**[19] N. Joukov, A. Traeger, CP Wright, Zadok, "Benchmarking File System Benchmarks", ETechnical Report FSL-05-04b, CS Department, Stony Brook University, 2005.**
**[20] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West., "Scale and performance in a distributed file system.", ACM Transactions on Computer Systems, 6(1):51–81, February 1988.**
**[21] AIM Technology, "AIM Multiuser Benchmark - Suite VII Version 1.1. ", http://sourceforge.net/projects/aimbench , 2001**
**[22] Nathan Edel, Deepa Tuteja, Ethan L. Miller, and Scott A. Brandt, "MRAMFS: A Compressing File System for Non-Volatile RAM" ,IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2004)**
**[23] R. Bryant, R. Forester, and J. Hawkes. "Filesystem Performance and Scalability in Linux 2.4.17." , In FREENIX '02, Monterey, CA, June 2002.**

**[24] Sun, “File System Performance: The Solaris OS, UFS, Linux ext3, and ReiserFS”, White paper August 2004.**
**http://sun.com/software/whitepapers/solaris10/fs_performance.pdf**
**[25] R. Bryant, R. Forester, and J. Hawkes. "Filesystem Performance and Scalability in Linux 2.4.17." , In FREENIX '02, Monterey, CA, June 2002.**
**[26] Iometer Project. “Iometer” ,OPEN SOURCE DEVELOPMENT LAB. 2004. www.iometer.org/.**
**[27] A. Kawaguchi, S. Nishioka, and H. Motoda,“A Flash Memory based File System,” Proceedings of the USENIX Technical Conference, 1995.**
**[28] F. Douglis, R. Caceres, F. Kaashoek, K. Li, B. Marsh, and J.A. Tauber, “Storage Alternatives for Mobile Computers,” Proceedings of the USENIX Operating System Design and Implementation, 1994.**
**[29] Veritest. “NetBench.” , www.veritest.com/benchmarks/netbench/ , 2002**
**[30] A. Tridgell, “dbench-3.03 README.” http://samba.org/ftp/tridge/dbench/README, 1999.**
**[31] Rosenblum M, Ousterhout J. “The Design and Implementation of a Log-Structured File System.”, Proceedings of the 13th ACM Symposium on Operating Systems Principles, October 1991.**
**[32] Mazieres, D. et al. , "Separating Key Management from File System Security",17th ACM Symp. on Operating Systems Principles. (1999).**
**[33] M. Kaminsky et al, “Decentralized User Authentication in a Global File System”, 19th ACM Symposium on Operating Systems Principles, 2003.**
**[34] An-I Wang, Peter L. Reiher, Gerald J. Popek, Geoffrey H. Kuenning, “Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System.”, USENIX Annual Technical Conference, General Track 2002: 15-28**
**[35] Schmuck, F. and Haskin, R., ”GPFS: A Shared-Disk File System for Large Computing Clusters”, Proceedings of the Conference on File and Storage Technologies (FAST’02), 2002**