國立交通大學

資訊科學與工程研究所

碩 士 論 文

應用於固態硬碟上的高效能寫入緩衝管理演算法

An Efficient Buffer Management Algorithm for Solid-State Disk

研 究 生：蘇宥全

指導教授：張立平　教授

中 華 民 國　九 十 八　年 七 月

應用於固態硬碟上的高效能寫入緩衝管理演算法
An Efficient Buffer Management Algorithm for Solid-State Disk

研 究 生：蘇宥全　　Student：You-Chiuan Su

指導教授：張立平　　Advisor：Li-Ping 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 Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

應用於固態硬碟上的高效能寫入緩衝管理演算法

學生：蘇宥全　　指導教授：張立平

國立交通大學資訊科學與工程研究所碩士班

摘　　要

SSD 使用 NAND Flash Memory 為基本的儲存元件具有讀取速度快、抗振性佳等優點，在部份的應用上逐漸取代傳統硬碟成為系統的主要儲存裝置。但因為 Flash Memory 的物理特性為在寫入資料前必需先進行抹除的動作，由於抹除單位比寫入單位大很多的緣故，所以需要對 random write 的寫入動作特別處理，否則會導致寫入效能不佳成為系統的效能瓶頸。而且因為檔案系統內的資料破碎以及 Multi-Thread 的應用，寫到儲存裝置的 write pattern 會更加 random，如何改善因為 random write 造成的 SSD 寫入效能下降問題愈顯重要。我們提出一個應用在 SSD 上的新 Buffer 管理演算法 HitStat，除了同時考慮時間區域性和空間區域性外，並能夠動態的調整得到較佳的 Padding 設定來大幅改善寫到 NFTL 的 write pattern。最後實驗結果顯示，HitStat 與既有 SSD 上的緩衝管理方法 FAB、BPLRU 相比，寫入效能平均改善了 30%以上。

關鍵字：NAND 快閃記憶體（NAND Flash memory），
固態硬碟（Solid-State Disk），寫入緩衝(Write Buffer)。

An Efficient Buffer Management Algorithm for Solid-State Disk

student：You-Chiuan Su Advisors：Dr. Li-Ping Chang

Department（Institute）of Computer Science
National Chiao Tung University

ABSTRACT

The base storage component of SSD is NAND Flash Memory, it has some advantages such as the high speed of read operation and shock resistance. On some application areas, SSD gradually replaces traditional hard drive disk becoming the major storage device of entire system. But the physical property of Flash Memory is Erase-before-Write, because of the erase-unit is much larger than the write-unit, we need take care of the random write pattern on SSD, otherwise the speed of write operation will degradation and become the bottleneck of entire system. Furthermore, due to disk fragmentation and applications of Multi-Thread, the write pattern will become more random, how to improve the problem of write throughput degradation on SSD caused by random write becoming more and more important. We proposed a new buffer management algorithm applied to SSD called HitStat, not only consider temporal locality and spatial locality simultaneously, but also adjust the Padding Threshold to better settings dynamically, much improve the write pattern written to NFTL. Finally, it shows about 30% enhancement of write throughput compared to FAB and BPLRU, which are other buffer management algorithms on SSD.

Keyword: NAND flash memory, SSD（Solid-State Disk）, Write Buffer

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，每個老師的敦敦教誨，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，黃士庭、郭郡杰、楊明毅、廖秀芬同學的幫忙，使得我能順利走過這兩年。實驗室的黃偉杰、郭晉廷、黃義勛學弟和黃莉君學妹們當然也不能忘記，你/妳們的幫忙及搞笑我銘記在心。

研究口試期間，感謝謝仁偉老師和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，謹以此文獻給我摯愛的雙親。

目　　錄

表　　目　　錄

圖　　目　　錄

一、Introduction

由於 NAND Flash Memory 的硬體特性優勢：耐震以及體積小，使得大部份的嵌入式系統大都採用 NAND Flash Memory 做爲其儲存系統。因此 NAND Flash Memory 的應用相當的廣泛，如：行動電話、多媒體播放器（PMP）、數位相框等。隨著技術的進步，NAND Flash Memory 的容量也跟著變大，因此衍生出使用 NAND Flash Memory 爲主要儲存媒體的固態硬碟-Solid-State Disk（SSD），用以取代傳統硬碟，如 Notebook、Netbook 已逐漸使用抗振性佳的 SSD 爲主要的儲存裝置。

NAND Flash Memory 的物理特性是在寫入資料頁（Page）前必須先抹除（erase）整個區塊（Block），Block 的大小比 Page 大很多，即一個區塊包含好幾個資料頁，所以在抹除整個區塊前需要先搬移其中的有效資料頁（valid Page）到別的區塊，造成抹除的區塊的成本非常高，尤其是待抹除區塊包含愈多的有效資料頁時。爲了提供上層簡單的 read、write 操作，會有一層 NFTL（NAND Flash Translation Layer）來管理 NAND Flash Memory，通常 NFTL 會把 NAND Flash Memory 的區塊分成兩種：大部份爲 Data Blocks，區塊內的資料是按照位址順序儲存、小部份爲 Log Blocks，用來儲存新的寫入資料。跟傳統硬碟相比，SSD 有非常好的隨機讀取效能（random read），因爲 NAND Flash Memory 沒有傳統硬碟需要移動機械手臂到待讀取位置所造成的搜尋時間（seek delay）；而在順序讀取（sequential read）和順序寫入（sequential write），SSD 比起傳統硬碟有相當或者更好的效能；但是 SSD 在隨機寫入（random write）下的效能非常差，雖然 NAND Flash Memory 的寫入沒有 seek delay，但是它有另外一個問題。

因爲在一般電腦用途的 workload 中有許多只有偶爾寫入的 random write 以及分成好幾段 requests 的 sequential write，由於這類資料會被寫到 Log Block 且不容易被 invalidate 的特性，引發 NFTL 在回收空間時需要做大量 valid Page 搬移的動作，使 SSD 的寫入效能大幅下降。而這個問題也會愈來愈嚴重，主要有兩種原因：（1）作爲一般電腦上的儲存裝置，它的寫入 pattern 不同於一般的使用，如：數位相機對儲存系統的寫入大都是 sequential write。而作業系統對儲存裝置的寫入大多是交雜的 sequential、random write，隨著一般電腦效能愈來愈強大，使用者同時間使用多種軟體更會加劇這種現象，增加 random write 的程度。（2）隨著 NAND Flash Memory 容量愈來愈大以及 Multi-Channel 的應用，SSD 上的 Logical Block size 也會非常的大，意謂說要回收區塊時，需要搬移的 valid data 資料量也會上升，增加回收區塊的成本。因此如何改善 SSD 在一般用途下因爲寫入所造成的效能瓶頸問題愈來愈重要。

目前有兩種方式試圖改善這個問題，Gupta 等人提出 DFTL[1]，使用 full Page-level mapping 的 NFTL 演算法可以增加 random write 的寫入效能。而我們則是在 SSD 內使用 Buffer 來改善 random write 的寫入效能瓶頸，Buffer 可以直接套用到現有的 SSD 產品以及既有的 NFTL 上，因爲重新設計 mapping 方式複雜的 NFTL 演算法可能衍生出一些不可預期的問題。

現有的 Flash 儲存系統上之 Buffer 管理方法有 FAB[2]、BPLRU[3]，它們都是以 Block 爲單位把 Buffer 中的資料分群（Group）管理，其中 FAB 注重空間區域性（spatial locality）：儘量把含有較少資料的群留在 Buffer 中，對於含資料量較多的群優先成爲 victim，目的是儘量增加寫到 NFTL 資料 sequential write 的比例，但由於忽略了時間區域性的重要性，如果某個群所含的資料量很少，可能會一直留在 Buffer 沒有機會被寫出而佔住許多空間；BPLRU 則注重時間區域性（temporal locality）：用 LRU 策略儘量把最近寫入的資料留在 Buffer 中以吸收熱資料。但因爲用 LRU 的方

式來管理群，可能使 hot 和 cold 資料同時在群中，因爲 hot 資料一直被 hit 的緣故，造成 cold 的資料沒有機會 flush 出去而積累在 Buffer 中，而且沒有特別留住資料較少的群，這些資料是造成 random write 的主要原因。

我們提出一個在 Flash 儲存系統上的新 write Buffer 管理演算法稱爲 Hit Statistic（HitStat）來改善 random write 的寫入效能，藉由 Buffer 把 write requests 整理成 NFTL 較容易處理的 write pattern，主要是儘量減少 random write 比例、增加 sequential write 的比例。爲了達到這個目的，我們同時考慮時間區域性及空間區域性，一方面要留住最近被寫入的群來收集 sequential data，另一方面也要把含有較少資料的群留在 Buffer 中。要充份運用 Buffer 空間避免只著重一種特性會排擠到另一個特性的資料留在 Buffer 中，HitStat 使用能夠依不同 workload 動態調整的 Cost-Benefit Analysis 來決定 Buffer replacement policy，而 Buffer 的 flush Groups 總數在 512KB Block size、16MB Buffer、相同的 workload 下可以達到 FAB 的 54.4%，BPLRU 的 70.2%。

除了 Buffer 本身可以運用的容量，NFTL 有更大的 Log Blocks 區域，因此 HitStat 也考量到跟 NFTL 配合來運用 Buffer 和 Log Block 的空間以發揮各自的特長提昇 SSD 整體效能。對 NFTL 來說 Padding[4]過的資料可以直接取代原本舊的 Data Block；而不做 Padding 的資料則會直接寫到 Log Block 中，但之後有回收區塊的代價。對於什麼時機應該做 Padding，[3, 4]中沒有說明，而這卻是一個相當關鍵的問題，因爲過度的 Padding 不能利用到 Log Block 的空間，反而會導致效能下降。此篇 Paper 中我們定義一個 NFTL 的 Padding Model，HitStat 可以根據該 Model 導出的方程式，動態決定是否對寫到 NFTL 的資料做 Padding 的動作，而實驗結果 Padding Model 可以收斂接近到最佳的 off-line Padding 設定，讓 SSD 整體效能達到最高。

接下來的章節中，第二章 Backgrond 我們會介紹 NAND Flash Memory 以及基本的 NAND Flash Translation Layer 架構，並在 Related Work 說明既有在 Flash Storage 上的 Buffer/Cache 演算法及它們的問題；第三章先說明應用在 Flash Storage 上的 write Buffer 管理方法設計時需要考量的原則，然後介紹我們提出的方法 HitStat 的 replacement policy 以及 Padding Model；第四章主要說明 HitStat 的 implementation 以及 replacement policy 細節；第五章實驗會把我們的 Buffer 管理演算法 HitStat 跟另外兩種既有方法 FAB、BPLRU 比較。首先比較 Buffer 的基本特性，接來來我們把 Buffer 架在兩種基本的 NFTL 架構 BAST、FAST 上比較效能，然後再驗證我們提出的 Padding Model 可以動態收斂到最佳的 off-line Padding 設定；最後第六章 Conclusion 爲此篇論文做總結。

二、Background

2.1 Flash Geometry

NAND Flash Memory 的組成如圖 1，主要是由多個區塊所組成，每一個區塊又是由多個 Pages 所組成。NAND Flash Memory 的一次寫入單位（write unit）則爲一個 Page，每一個 Page 會有一個 Spare Area，可以用來儲存 User Area 中資料的 mapping 資訊或者 Error Correcting Code。而刪除單位（erase unit）則爲一個區塊，即每次執行 erase 時都是以區塊爲單位。NAND Flash Memory 在更新資料時與一般的 block device 不同，如：disk，在更新某部份的資料後可直接寫回其原本的實體位置，但是 NAND Flash Memory 無法將更改完後的資料直接寫入至該資料原本所在的實體位置，必須再額外找空閒的 Page 寫入，此動作即稱爲 outplace-update。而原本的 data 則視爲失效（invalid），但也因爲如此 NAND Flash Memory 中常常會

存有很多 invalid data，因此 NAND Flash Memory 每隔一段時間即要清除那些 invalid data，以空出空間提供其他 data 儲存，此動作稱之爲 Garbage Collection。

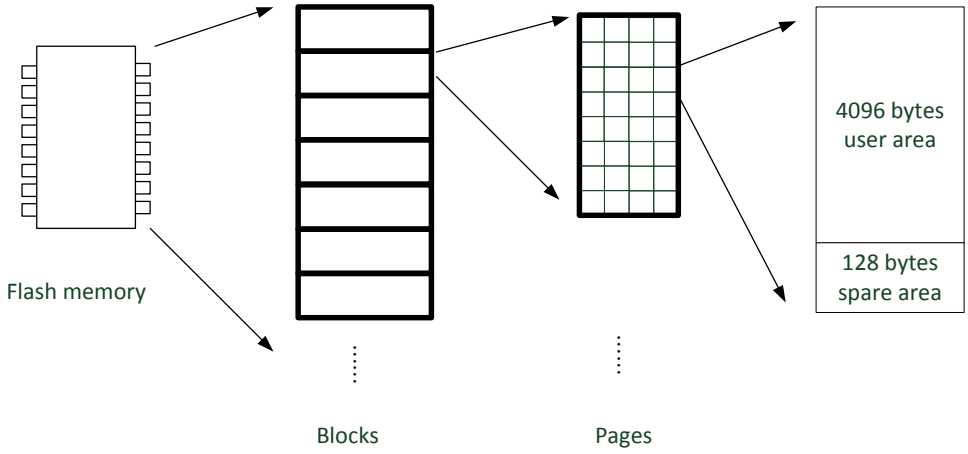


圖 1：Flash Memory Geometry

2.2 NAND Flash Translation Layer

由於 SSD 與 Hard Drive Disk 的硬體架構的不同，因此以往的 File Systems，如：Ext2、Ext3、NTFS、FAT 等皆無法直接在 NAND Flash Memory 上使用，但可能在 SSD 上使用的作業系統種類相當多，會使用到的 File System 則會依所使用的作業系統不同而不同，爲了能讓 SSD 能適用於現有的 for block device 的 File System，因而有了一個 Translation Layer：NFTL（NAND Flash Translation Layer），把自己模擬成一般硬碟以提供上層 File System 的讀寫要求，而 NFTL 主要的功能包含(1)update policy、(2)Address Translation、(3)Garbage Collection、(4)Wear Leveling、(5)Error Code Correction 以管理整塊 NAND Flash Memory，因此 NFTL 的演算法對 SSD 的整體效能影響很大。

討論兩種極端 mapping 方式，Page-level 和 Block-level mapping 的 NFTL。Page-level mapping NFTL 雖然可以得到很好的效能，但在 RAM 中需儲存每個 Page 從邏輯位置到實體位置的對應資訊，假設儲存一個 Page 的位置對應資訊需要 4Byte 空間、一個 Page 的大小爲 4KB，一塊 16GB 的 NAND Flash Memory 就需要用掉 16MB 的 RAM 空間來儲存整個 mapping 資訊，如此大的 RAM 顯然不合乎 SSD 成本考量，而且突然斷電後如何重建 mapping 資訊是一個問題。另一種極端的 NFTL mapping 方式，Block-level mapping NFTL 只把 Block 的 mapping 資訊存在 RAM 中，雖然需要的 RAM 容量非常小，但即使每次只寫入少量的資料都需要抹除整個 Block 來寫入新增料，造成大量的 overhead 所以效能非常差。

爲了解決上述兩種極端 NFTL 的缺點，各種 Hybrid NFTL[5-8]架構被提出來，主要是 mapping 方式混合了 Page-level 和 Block-level mapping。它們把 Blocks 分成兩種：Data Blocks、Log Blocks，Data Blocks 內的資料都按邏輯位置擺放依 Block-level 方式 mapping，全部 Data Blocks 所佔的空間即爲上層 File System 看到的硬碟總容量；而 Log Blocks 則依 Page-level 方式 mapping，用來儲存新寫入的資料。當 Log Block 沒有空間可以寫入時，NFTL 需要做 Garbage Collection 的動作把部份在 Log Block 內的資料寫回其應該擺放的 Data Block 中，在 Hybrid NFTL 中我們稱之爲 Merge，主要 Merge 方式有 Switch Merge、Partial Merge 以及 Full Merge 三種。

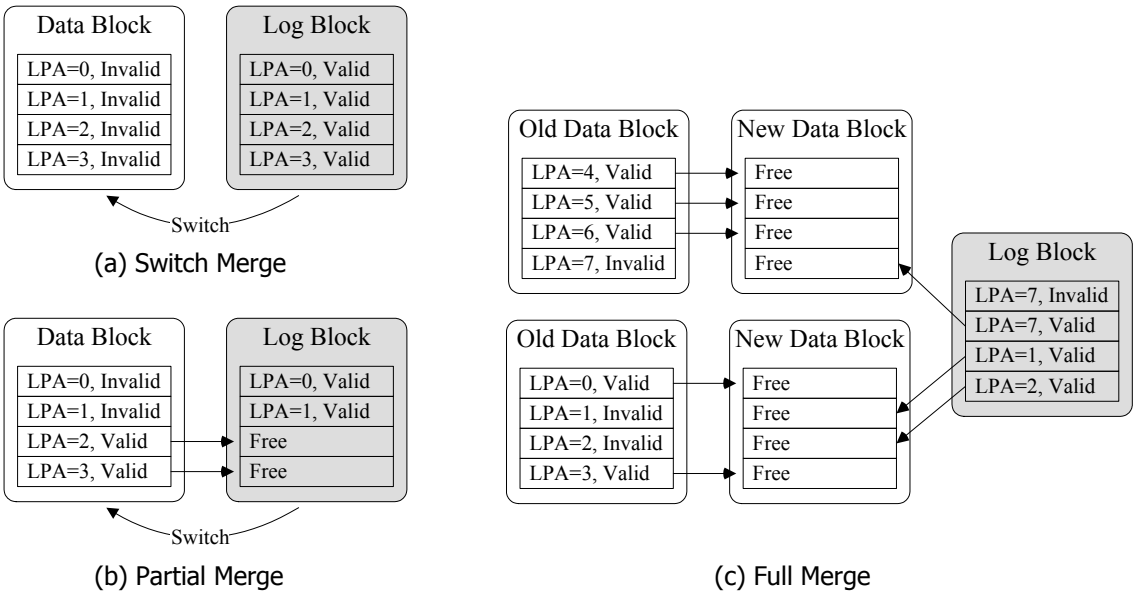


圖 2：Merge operations of Hybrid NFTL

LPA 爲 Logical Page Address = Logical Address / Sectors per Page

LBA 爲 Logical Block Address = Logical Address / Sectors per Block

藉由 LBA 和 LPA 我們可以查詢 NFTL 的 mapping Table 得知資料存放在 NAND Flash Memory 中的實際位置

第一種爲 Switch Merge 如圖 2(a)，如果 Log Block 內的資料是依照邏輯位置按順序寫入的，在回時空間時我們只需要更改 Block-level 的 mapping 資訊，把對應到舊 Data Block 的實體位置改成該 Log Block 的實體位置，用 Log Block 來取代舊的 Data Block 後，再對舊的 Data Block 進行抹除的動作。Switch Merge 在三種 Merge 方式中代價最低，只需抹除一個 Block 的代價，而且這種更新方法符合 Block-level mapping 的規則。

第二種爲 Partial Merge 如圖 2(b)，如果 Log Block 內的資料是依照邏輯位置按順序寫入但還未被寫滿，則我們必需要先從別的地方讀取資料把 Log Block 依序補滿後，再做如圖 2(a)Switch Merge 的動作，因此 Partial Merge 比 Switch Merge 額外多出搬移數個 Pages 的代價，如果原本 Log Block 內含的資料量愈少，需要搬移的資料愈多，會造成 Partial Merge 的代價愈高。

最後一種爲 Full Merge 如圖 2(c)，如果 Log Block 內的資料不是按邏輯位置順序寫入的，則只能做 Full Merge，需要先把原本在 Data Block 和 Log Block 內的 valid 資料搬移到新的 Block 後，再更改 Data Block 的 mapping 資訊指到新的 Data Block，再對 Log Block 和舊的 Data Block 進行抹除的動作。如果原本 Log Block 內的 valid 資料分別屬於 N 個不同的 Block，則 Full Merge 的代價一共需要 N+1 個 Blocks 抹除和 N*每個 Block 內 Page 數量的 Pages 搬移動作，N 愈大回收的代價愈高。某些 NFTL 架構下由於 N 可能很大造成 Full Merge 的代價過高，有時候在寫入資料時會有系統 delay 的情況。

2.3 NFTL Characteristics

雖然 Hybrid NFTL 改善了 Page-level 和 Block-level mapping NFTL 的缺點，但對於一般用途的 write pattern，也有它們不足的地方導致效能下降，主要原因是 Full Merge 的代價太高。而最近提出的 SuperBlock[6]、LAST[8]也試圖把冷、熱資料分開以降低 Merge 的代價，但 SuperBlock 需要明確調校參數而 LAST 也要外部的 locality

檢測機制使 Hybrid NFTL 的設計趨向複雜。爲了說明 Hybrid NFTL 的問題，接下來我們介紹兩種基本的 Hybrid NFTL 架構 BAST[5]、FAST[7]以及造成其效能瓶頸的原因。BAST 和 FAST 各儲存兩張 Table，一張是 Logical-to-Physical Table（L2P Table）主要存 Block-level mapping 資訊（圖 3(a)），而另一張則是儲存 Log Blocks 的 mapping 資訊。

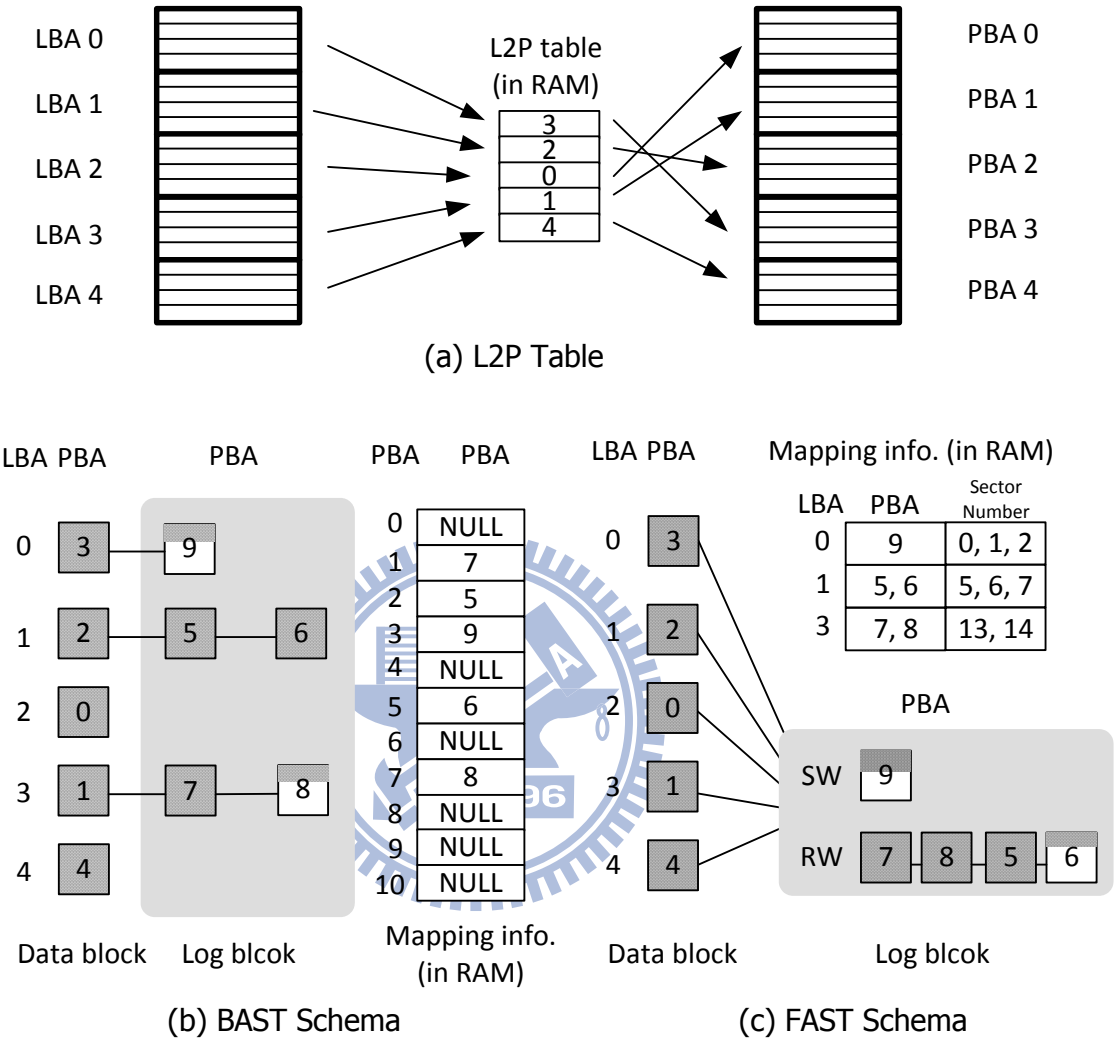


圖 3：BAST and FAST Schema，LBA：Logical Block Address、PBA：Physical Block Address

參考圖 3(b)，BAST 採用的是 Block chain 的方式，當一個 Log Block 被寫完之後則再向系統要求一個 Log Block，並會串接在原本的 Log Block 後面，若該 LBA 相當的 hot，則該 LBA 會串上相當多的 Log Block，而做 GC 時會選擇一條 Block chain 來回收。由於 BAST 會把屬於不同 LBA 的資料寫到不同的 Block chain 中所以不會混在一起，因此優點是可以把交錯的 multiple sequential write streams 分開寫到不同的 Log Block 中，只需代價低的 Switch Merge 或 Partial Merge；而且每一條 Block chain 中的資料都屬於同一個 LBA，這樣在做 Full Merge 的時候只需要搬移 1 個 Block 的資料量。但缺點爲，(1)對於應用在電腦儲存系統的 SSD 來說，write pattern 大多是相當 random 的，而且只有少部份是 hot data，大部份爲 cold random data 會打中大量不同的 LBA，當 Log Block 數量不夠時，Block chains 之間會競爭有限的 Log Block，造成很多 Log Block 只有寫入少量 data 就被回收，這個現象我們稱爲 Log Block

Thrashing，其它跟 BAST 架構相近的 NFTL 如 optimistic FTL[9]，也有 Log Block Thrashing 的問題。(2)Log Block 內資料的 mapping 資訊被儲存在各個 Page 的 Spare Area 中，如果要讀取某個 Page 時可能需要讀完整個 Block chain 的資料才能找到該 Page 最新的 update 位置，造成讀取效能下降。

參考圖 3(c)，FAST 的 Log Block 有一個 SW Log Block（sequential write Log Block），其它都爲 RW Log Block（random write Log Block），FAST 會試圖把 sequential write 依位置順序寫到 SW Log Block，當 SW Log Block 寫滿或者要寫另一筆新的 sequential 資料時會對 SW Log Block 做 Switch Merge 或 Partial Merge 來取代舊的 Data Block。FAST 主要的優點是可以完整的利用 Log Block 空間，沒有像 BAST 有 Log Block Thrashing 的問題，但有其它的缺點，(1) 隨著 P2P 以及 Multi Thread 軟體的發展會產生交錯的 multiple sequential write streams，FAST 會把大部份的這類資料也交雜寫到同一個 RW Log Block 中，因爲無法把這些 sequential write 分開寫到不同的 Log Block 做 Switch Merge，增加大量額外的回收成本。(2)FAST 做 GC 時會回收最早寫入的 RW Log Block，如果其中含有大量的 valid data 屬於不同的 LBA 時，需要把每個不同 LBA 的 valid 資料搬移到新的 Block 後再對原本的 Data Block 和 Log Block 進行抹除，造成有時候做單次 Full Merge 動作的成本相當高，導致系統 delay 的問題。(3)由於 Log Block 是用完全的 Page-level mapping，儲存 mapping 資訊需要用到的 RAM 空間比 BAST 高很多，因爲使用的 RAM 空間受到限制實際上能夠運用的 Log Blocks 數量有限。

2.4 Related Work

傳統硬碟跟 SSD 一樣在 random write pattern 有效能下降的問題，傳統硬碟是因爲需要移動機械式手臂到待讀取位置產生的搜尋時間（seek delay），而 SSD 主要是要回收因 outplace-update 而被 invalid data 佔據的空間。SSD 可以使用別種的 NFTL mapping 方式如 Page-level mapping[1]、SuperBlock[6]、multi mapping granularities[10]來改善，但 NFTL 演算法會變的相當複雜，可能會有一些不知如何調校的參數或者產生其它如 reliable、開機時間、需要過多 CPU 及 RAM 資源的問題。另一種改善方式是在 SSD 上使用 Write Buffer 來改善 write pattern，並配合基本的 NFTL 架構，也是此篇論文主要的研究部份。

Jiang 等人提出 Dual Locality（DULO）[11]，不同於大部份舊方法只著重在時間區域性單純使用 LRU 順序 replace，DULO 同時考慮時間區域性及空間區域性來決定 replacement policy，雖然 DULO 主要是爲傳統硬碟設計的方法，但主要的目的跟我們的方法相近，會同時考量資料留在 Buffer 的時間以及 sequential 程度。而 HitStat 主要是針對 SSD 的 Buffer 管理演算法，因應 NAND Flash Memory 的物理特性來設計，在實作方面跟 DULO 有很大的差異。

目前既有對於 NAND Flash Memory 的 Buffer/Cache[2, 3, 12]演算法，CFLRU[12]是 OS 上的 Buffer/Cache 管理機制，因爲 Flash Storage 的讀取比寫入快很多，CFLRU 會把 LRU 分成兩個區段，從較冷的區段優先選擇 clean Page 成爲 victim 以減少寫出資料到 Flash Storage。DULO 以及 CLC[9]也同樣把 LRU 分區段從較冷的區段中挑選 victim，但如何決定各個區段的長度成爲一個需要設定的參數，在不同 workload、不同的 Buffer/Cache size 條件下需要實際調校優化這個參數才能得到較好的效能。FAB[2]、BPLRU[3]是 Flash-Aware，即直接在 Flash Storage 上使用 embedded RAM Buffer 的管理方法，爲本篇論文的主要比較對象，兩者都會把屬於同一個 LBA 的資料分在同一群（Group）的方式來管理，當要 flush 時會選某個群爲 victim，一次寫

出該群都屬於同一個 LBA 的資料。

FAB 的 replacement policy 注重空間區域性，優先選擇含最多資料的群爲 victim，它希望從 Buffer flush 出的資料是大筆 sequential 的資料，而儘量把含較少資料的群留在 Buffer 中。但該方法的缺點是忽略了時間區域性的重要性，如果某個群所含的資料量很少，可能會一直留在 buffer 而沒有機會被寫出，這類資料會佔住 Buffer 的可用空間；另外對於 sequential write 來說，大多的群只收集到小部份的 sequential data 就會成爲含資料最多的群而先被寫出，最會導致留在 Buffer 中的都是含資料量少的群。

BPLRU 的 replacement policy 注重時間區域性，依 LRU 順序選擇最久沒有被寫入資料的群爲 victim，它目的主要除了吸收熱資料外，對於 sequential write 也可以收集滿整個群的資料後再寫到 NFTL，而且使用 Padding 的機制來增加 Switch Merge 的機會。但缺點是因爲用 LRU 的方式來管理群，如果 hot 和 cold 資料同時在群中，因爲 hot 資料一直被 hit 的緣故，造成 cold 的資料沒有機會 flush 出去而積累在 Buffer 中；另一個缺點是沒有特別留住資料較少的群，這些資料是造成 random write 的主要原因，而且沒有說明什麼時候做 Padding，如果對這些含較少資料的群也使用 Padding 代價反而會更高。

就以上的討論來說，既有的 Buffer/Cache 演算法如 FAB、BPLRU 只單獨考量時間區域性或空間區域性，由於過份注重單一特性，造成某些資料會霸佔住 Buffer 空間而排擠到另一種特性的資料留在 Buffer 中，造成 Buffer 無法吸收這種特性的 workload 來幫助 NFTL，導致整體 SSD 效能改善有限。其它一些方法如 DULO、CLC 把 LRU 分成熱區段和冷區段，再從較冷的區段選擇 victim，但卻有如何調整區段大小的參數問題，固定的設定也無法適用到不同的 Buffer size 以及不同的 workload。爲了改善既有方法的問題，我們提出新的 Buffer 管理演算法 HitStat 能夠同時考量時間區域性或空間區域性，並且避免掉非直觀、可能會對實驗結果造成很大影響的參數設定。

三、HitStat

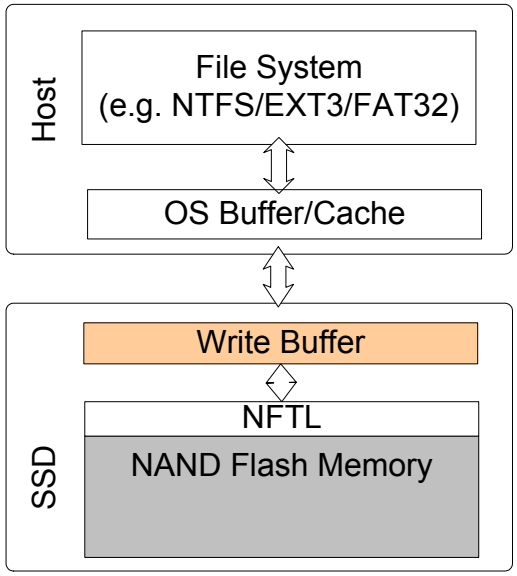


圖 4：System Overview

爲了改善 Flash Storage 的效能瓶頸，我們提出一個 Buffer 管理演算法 HitStat，主要應用在 Flash Storage 的儲存裝置上如 SSD。HitStat 只對寫入的要求做分配和管理 Buffer memory 的動作，因爲主要是寫入資料時會造成 SSD 的效能瓶頸，而對 read 來說大部份 NFTL 架構都能夠快速讀取資料，而且作業系統中 Cache 有容量更大的 DRAM 來處理 read。圖 4 爲本篇論文考量的系統架構，Host 端底層由 File System 和 OS 管理的 Buffer/Cache 來處理對儲存裝置的讀寫操作；而 SSD 裝置主要有 NFTL 和主要的儲存元件 NAND Flash Memory，我們的方法是在 NFTL 之上使用 write Buffer，把複雜的 write requests 重新整理成 NFTL 善於處理的 write pattern 來改善 SSD 的效能瓶頸。

HitStat 演算法的概念主要包括兩部份，使用 Cost-Benefit Analysis 同時考慮時間區域性和空間區域性的 Buffer replacement policy；能應用於各種 Hybrid NFTL 的 Padding Model，我們根據 Padding Model 得到的結果動態決定是否對寫出的 Group 做 Padding 的動作。

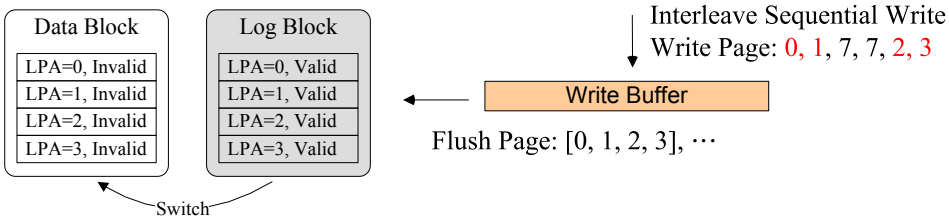
3.1 Problem Definition

我們的 Buffer 主要是建構在 NFTL 之上，因此 Buffer 方法需要能針對 NFTL 的弱點來改善，就章節 2.3 中對 BAST、FAST 的敘述，在一般電腦儲存系統上大量的 random write 以及交錯的 sequential write pattern 是造成 Hybrid NFTL 效能瓶頸的主要原因，我們需要發揮 Buffer 的優點來處理這些 NFTL 不善處理的 write pattern。以下幾點是應用在 SSD 上的 Buffer 管理演算法在設計時需要考量的部份，(1)時間區域性方面，必需能夠留住較年輕的資料才能吸收 hot data。(2)空間區域性方面，要讓寫出的資料儘量 sequential 必需把含資料較少的 Block 儘量留住 Buffer 中來減少 random 程度，而且要儘早寫出 sequential 資料避免它們佔住 Buffer 空間。(3)對交錯的 sequential write pattern，爲了要循序寫到 Log Block 才能做代價低的 Switch Merge，因此能 inplace-update 的 Buffer 必需能把分散成好幾個 write requests 的 sequential write 整理成以 Block 爲單位再 flush 到 NFTL，避免 sequential 資料被寫到 Log Block 佔住空間。(4)從 Buffer 寫出的資料可以做 Padding，把資料補滿整個 Block 後寫到 NFTL 直接做 Switch Merge，適當的 Padding 設定能夠大幅改善效能。(5)在不同的作業系統不同用途的電腦上 write pattern 也會相差很多，因此好的 Buffer 管理方法必需可以隨著不同的 write pattern 來改變 replacement policy。(6)演算法在搜尋、加入、挑選 victim 時都不能過慢，最好是 O（1），不隨著 buffer size 增大而變慢。

3.2 Buffer Management Concept

對於 SSD 的寫入效能不佳的問題，主要是因爲回收 invalid 資料的代價過高，在章節 2.2 中有介紹關於 Hybrid NFTL 回收空間的三種 Merge 方式，這邊我們說明對於交錯的 sequential write 和 random write 來說，write Buffer 應該如何幫助 NFTL 才能降低 Merge 的代價來改善整體效能。

首先寫到 NFTL 的資料能夠儘量提昇做 Switch Merge 的機會，因爲 Switch Merge 代價最低只需抹除一個 Block，所以 Buffer 要能夠收集循序的寫入資料。如圖 5(a)中原本的循序寫入資料 0, 1, 2, 3 被分成前後兩筆 request，在沒有 Buffer 的情況下，資料 0, 1, 2, 3 會被分開寫到 Log Block 佔住空間，之後還需要做回收的動作。如果有 Buffer 的話我們能夠把資料 0, 1, 2, 3 收集在一起再同時寫到 NFTL，因爲 Log Block 內是按順序被寫滿的，所以可以做 Switch Merge 只需抹除一個 Block 的代價。



(a) Increase Switch Merge

Random Write
Write Page: 0, 4, 8, 2, 3, 5, 9

Write Buffer

Flush Page:
[0, 2, 3], [4, 5], [8, 9]

Buffer		
Write Page	2 Log Blocks	Merge Operation
0, 2, 3	[0, 2, 3]	
4, 5	[4, 5], [0, 2, 3]	
8, 9	[8, 9], [4, 5]	Full Merge [0, 2, 3]

(b) Reduce Full Merge Cost

No Buffer		
Write Page	2 Log Blocks	Merge Operation
0	[0]	
4	[4], [0]	
8	[8], [4]	Full Merge [0]
2	[2], [8]	Full Merge [4]
3	[2, 3], [8]	
5	[5], [2, 3]	Full Merge [8]
9	[9], [5]	Full Merge [2, 3]

Buffer with Padding		
Write Page	2 Log Blocks	Merge Operation
0, 2, 3		Partial Merge [0 ~ 3]
4, 5	[4, 5]	
8, 9	[8, 9], [4, 5]	

(c) Take Advantage of Partial Merge

圖 5：How Buffer can help NFTL

第二點要減少寫到 NFTL 資料的 random 程度來降低 Full Merge 的代價。因爲 random 的寫入資料在 BAST 架構上會發生 Log Block Thrashing 的問題；而在 FAST 上也會因爲要回收的 Log Block 內的 valid 資料屬於多個不同 LBA，造成 Full Merge 的代價過高，因此不管在哪種 Hybrid NFTL 架構，我們都不希望寫入的資料過於凌亂。參考圖 5(b)原本的 random 寫入資料爲 0, 4, 8, 2, 3, 5, 9，在沒有 Buffer 的情況下寫到 NFTL 一共需要 4 次 Full Merge 的動作，有 write Buffer 的話我們可以延緩 random 資料寫到 NFTL，待集合成一個較大的 request 後再一起寫到 NFTL 來改善 write pattern 過於凌亂的問題，這樣只需要做一次 Full Merge 的動作。

第三點要最大化從 Padding 得到的效能改善，Padding 類似 Partial Merge，需要先把資料補滿整個 Block 後再寫到 NFTL。參考圖 5(c)，如果我們有做 Padding 的話，可以把原本代價高的 Full Merge 轉變成 Partial Merge 來降低 overhead；但從另一點來說 Padding 需要做額外的資料搬移動作，所以過度的 Padding 反而會增加 overhead，因此 Buffer 管理方法要適當的調整 Padding 條件才能得到較大的效能改善。另外如果 Buffer 管理方法能滿足第二點讓寫出的 request 所含的資料量較大，做 Padding 的代價也會比較少，能夠從 Padding 得到的效能改善幅度愈大。

3.3 Buffer replacement policy

因爲 NAND Flash Memory 的抹除單位爲 Block，HitStat 會把 Buffer 中屬於相同 LBA 的資料整合在一起以 Group 的方式來管理，而我們的目標是最大化 Group Hit Ratio，如方程式(1)，其中 $P(Group_i)$代表 Group i 會被 hit 的機率。最大化 Group Hit Ratio 可以讓 Buffer 具有章節 3.2 中敘述的優良特性，以下就三點來說明爲何要最大化 Group Hit Ratio。

第一點是收集 sequential wrtie 增加 Switch Merge 的機會，由於循序寫入會短時間重覆寫入同一個 LBA 的資料，所以爲了要增加 Group Hit Ratio，Buffer 勢必要能夠收集這類資料，否則 Group Miss 的次數會大幅上昇；第二點是降低寫到 NFTL 資料的 random 程度來降低 Full Merge 的代價，而儘量增加 Group Hit 的機會可以把數個含資料量少的 random write 集合成一個含資料較多的 Group 後再寫到 NFTL，另一方面也減少把含資料量少的 Group 寫到 NFTL；最後一點是參考 Partial Merge 的優點，適當的使用 Padding 來改善效能，而 Padding 的代價是需要搬移那些不在 Group 內的資料把 Group 補滿，因此如果我們能達到第二點的要求，可以增加 flush Group 平均含的資料量，做 Padding 的代價就會較低，可以藉由 Padding 來得到更大的效能改善。

$$Max \sum_{Group_i \text{ in Buffer}} P(Group_i) \tag{1}$$

但是要最大化方程式(1)的問題是，我們必需同時考慮 Buffer 中每個 Group 的兩個值：**Weight of Group**（空間區域性），即該 Group 含的 Sector 總數，若 Weight 值愈大代表此 Group 佔住很多 Buffer 空間，排擠到其它的 Group 留在 Buffer 中，因爲所有在 Buffer 中的 Groups 它們的 Weight 總和必需小於等於 Buffer 的總容量，所以 Weight 值當作 Cost，該值愈大會愈早成爲 victim。另一個爲 **Hit Probability of Group**（時間區域性），即方程式(1)中的 $P(Group_i)$，我們希望留在 Buffer 內的每個 Group 其 $P(Group_i)$值愈大愈好，如此方程式(1)總和的值也會愈大，所以 $P(Group_i)$ 當作 Benefit，該值愈大會愈晚成爲 victim。

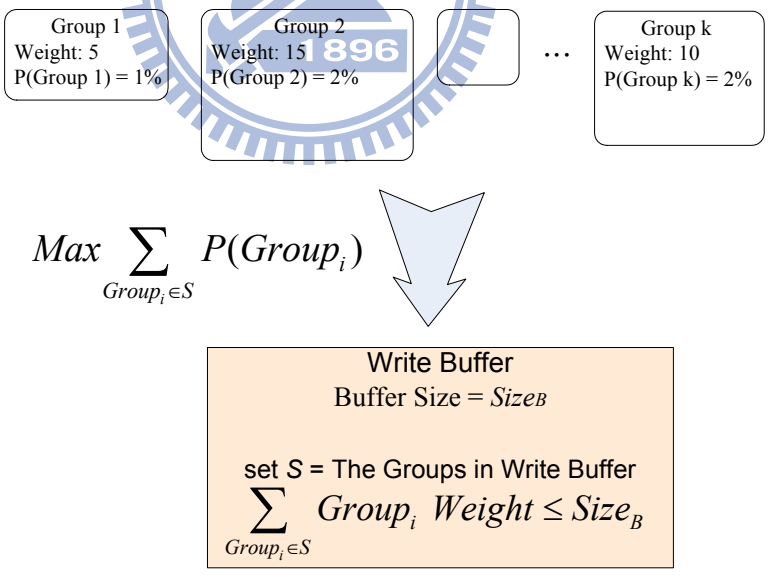


圖 6：The Knapsack Problem

參考圖 6 假設有 k 個 Group，我們可以把每個 Group 看成一件物品，Weight of Group 看成是物品的重量而 Hit Probability of Group 則是物品的價值，我們希望背包內裝的物品價值總和愈高愈好，相當於方程式(1)，但因爲 Buffer 的容量有限，最多能裝的物品總重量不能大於 $Size_B$。

$$\frac{\text{benefit}}{\text{cost}} \text{ of } Group_i = \frac{P(Group_i)}{Group_i \ Weight} \quad (2)$$

事實上經過轉化後的問題就是相當著名的背包問題（Knapsack Problem），而問題的難度則爲 NP Hard[13]。針對這個問題我們使用的方法是用 Cost-Benefit Function 來計算，參考方程式(2)，儘量把價值/重量比大的留在 Buffer 中，因此 HitStat 的 Buffer replacement 策略是優先選擇得到較小比值的 Group 成爲 victim，這是背包問題中一個計算比較迅速的 heuristic 解決方法。

3.4 Padding

對於 Hybrid NFTL 來說，我們可以使用補齊（Padding）的方式來降低 Buffer flush 的成本。因爲從 Buffer flush 到 NFTL 的資料都會被寫到 Log Block，當 Log Block 用完時需要跟 Data Block 做 Merge 的動作來回收，如果寫到 Log Block 的資料是 sequential 的寫滿整個 Block，那我們可以直接做 Switch Merge 的動作。因此如果被 Buffer 挑選的 victim Group 當中的資料沒有滿，HitStat 可以從 NFTL 讀取那些缺少的 Page 把資料補滿整個 Group，之後就可以從 Buffer 循序寫出整個 Group 的資料，雖然 Padding 需要做額外做一些 Page 的讀取和寫入動作，但是它可以把 Full Merge 的動作變成只需要 Switch Merge，能降低 Merge 的成本。

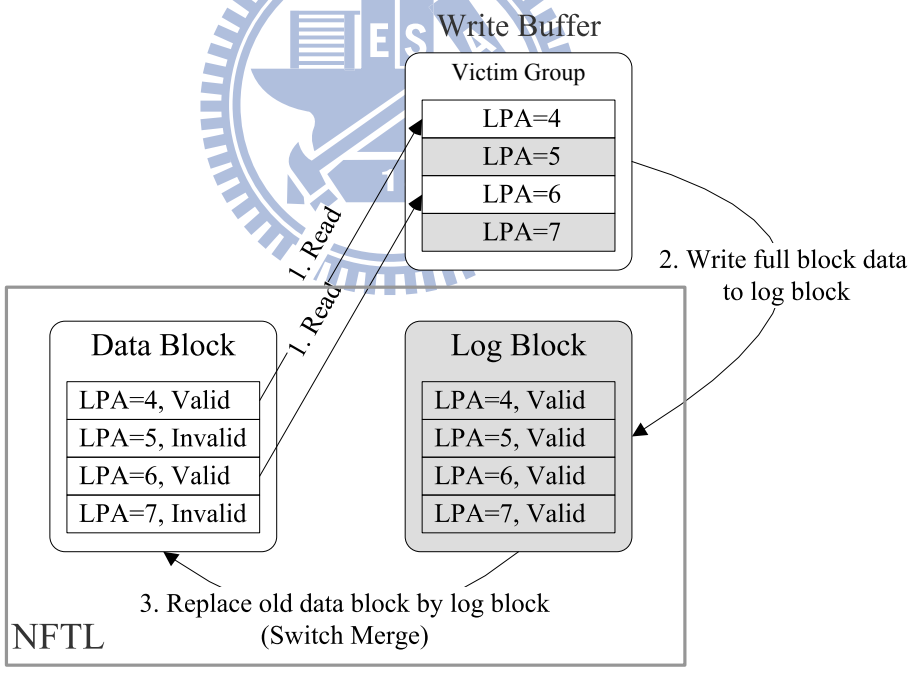


圖 7：Illustration of Padding

圖 7 是一個做 Padding 的範例，被選中的 victim Group 中含有兩個 Page 的資料（5 and 7），HitStat 從 NFTL 讀取 Page 4 and 6 把資料補齊整個 Group，接下來循序寫出 Page 4 ~ 7 到 Log Block，因爲 Log Block 中完全是循序排列的資料，NFTL 會直接做 Switch Merge 用 Log Block 取代舊的 Data Block。

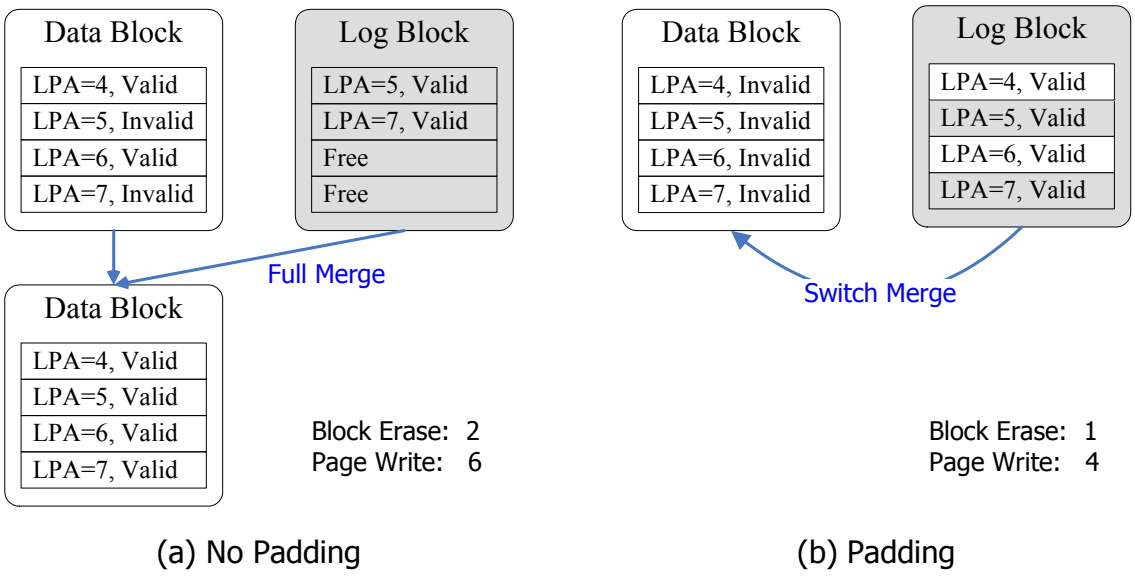


圖 8：No Padding vs. Padding

依照圖 7 的情況假設 victim Group 含有兩個 Page 的資料，我們分別比較沒有 Padding 和有 Padding 的情況下把資料寫到 NFTL。參考圖 8(a)若不做 Padding 該 Group 兩個 Page 的資料直接被寫到 Log Block 中，之後還需要 Full Merge 的動做才能回收該 Log Block，全部的操作包含 2 次 Block Erase 和 6 次 Page Write；而圖 8(b)因爲有對 victim Group 的資料做 Padding 的動作，所以 Log Block 內的資料是按位置循序被寫入，只需要 Switch Merge 就能取代舊的 Data Block，一共只需 1 次 Block Erase 和 4 次 Page Write，比圖 8(a)沒有 Padding 之後還要 Full Merge 的代價低很多。

3.5 Write-Back Policy

比起 Buffer 容量，NFTL 有更大空間的 Log Block 可以運用，這些空間可以看成延伸的 Buffer 儲存空間，但因爲 NAND Flash Memory 的特性只能 outplace-update 資料；而 RAM Buffer 則是可以做 inplace-update 但容量較小。好的 Buffer 管理方法除了要能吸收 hot data 以及增加寫出資料的連續性之外，還需要能夠與 NFTL 配合，運用 RAM Buffer 跟 Log Block 互補的特性發揮各自善長的部份。

參考圖 9 爲我們從 Buffer 選出 victim Group 之後的流程圖，如果 victim Group 內的資料是滿的話，可以直接寫到 NFTL 做 Switch Merge；如果資料不是滿的，則我們必需決定是否對 victim Group 做 Padding 的動作。做 Padding 類似 Partial Merge，先把 Group 內資料補滿後再寫到 NFTL 做 Switch Merge 的動作，直接將資料儲存在 Data Block；不做 Padding 的話該 Group 的資料就會被寫到 Log Block 中，佔住一部份 Log Block 空間，待稍後做 Full Merge 時才會寫回 Data Block 回收這些空間。因此好的 Buffer 管理方法可以藉由適當的 Padding 來跟 NFTL 配合，充份利用 Log Block 的空間讓 Merge 的成本降低來提高 SSD 的效能。

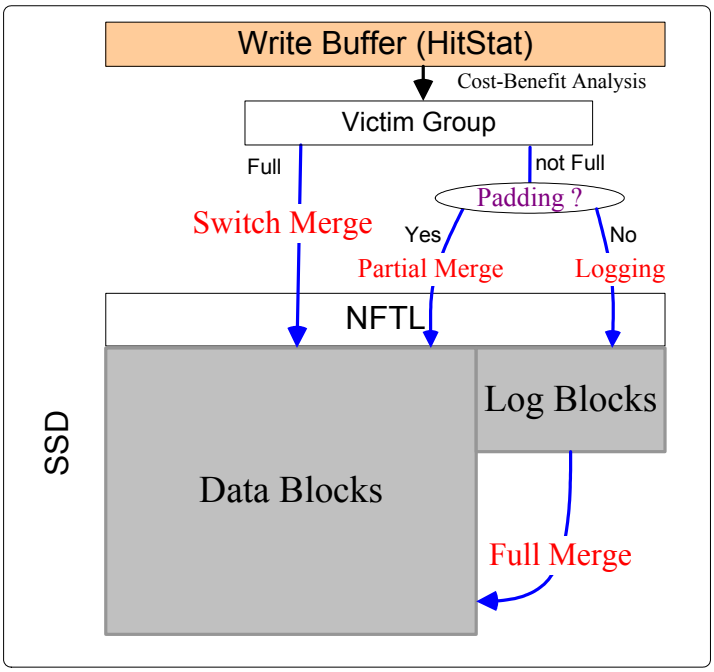


圖 9：Interact with NFTL by Padding

而什麼樣的 victim Group 適合做 Padding 呢？由於從 Buffer 寫出的大部份是 cold data，顯然不能參考資料的冷熱程度。因此可以用來決定是否做 Padding 的主要因素是 victim Group 內所含的資料量多寡，對含資料較多的 victim Group 做 Padding 需要額外搬移資料的代價較少；參考圖 10 我們以 Log Block 的角度來看，範例中的 write stream 前後有兩次關於 Block A 的資料（A_i、A_j）寫入，圖 10(a)在沒有 Padding 的情況下在 A_j 的資料到來之前 A_i 就會因爲 Log Block 空間不足而被 Merge，Block A 先後需要參與到兩次 Full Merge，因此資料 A_i 被寫到 Log Block 是 overhead，因爲它佔用 Log Block 空間而且沒有跟 Block A 的其它資料一起 Merge 回 Data Block；圖 10(b)爲對某些資料做 Padding 的動作，其中資料 B_i、C_i 因爲有 Padding 所以直接做 Switch Merge 而沒有佔住 Log Block 空間，可以看到在資料 A_i Merge 之前 A_j 已經寫到 Log Block 中，Block A 只需要參與到一次 Full Merge 的動作。所以對含資料較多的 victim Group 做 Padding 除了代價較少外，也可以避免大量資料被寫到 Log Block 佔住其空間，較適合做 Padding；而含較少資料的 Group 則剛好相反，把含較少資料的 Group 寫到 Log Block 不會佔住許多空間，Padding 反而因爲要讀寫許多額外的 Pages 導致 overhead 很高，不適合 Padding。

從上面的分析來看，適當的 Padding 可以避免大量資料佔住 Log Block 的空間，延長需要做 Full Merge 的時間間隔，增加屬於同一個 Block 的資料一起 Merge 的機會，所以大幅降低了 Merge 的成本。但是過多的 Padding 會產生很多額外讀寫 Page 的 overhead，反之過少的 Padding 也會導致 NFTL 的 Full Merge 成本無法降低，因此如何找到最佳的 Padding Threshold 才能達到最好的效能顯然是一個問題。由於 write pattern 會改變、各個 NFTL 使用的架構及 Log Block 數目也不一樣，只用固定的 Padding 設定並不是最佳的方式。

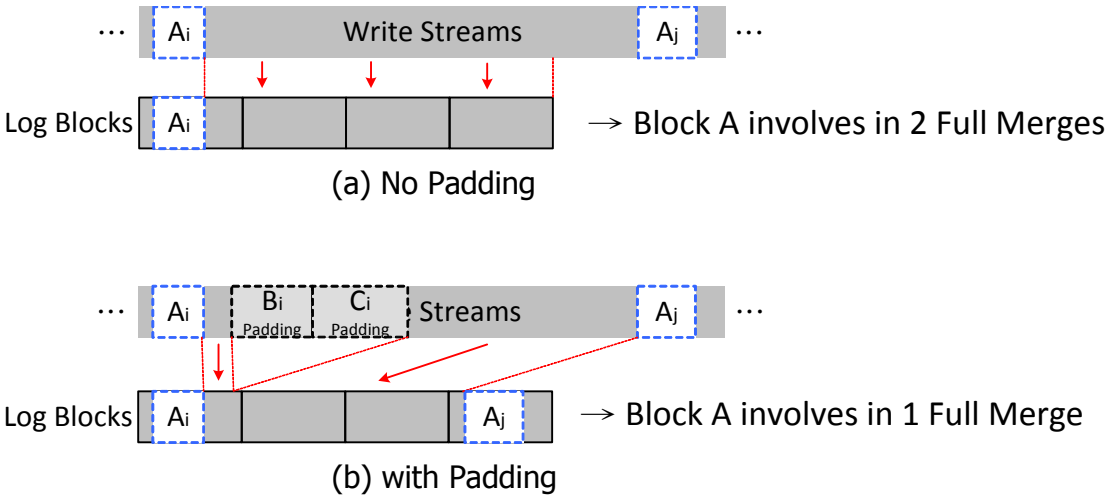


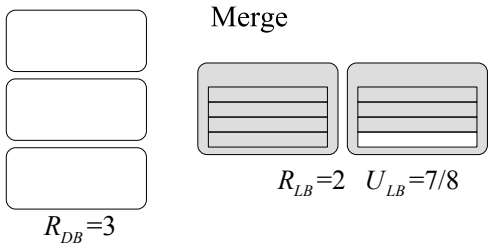
圖 10：Example of (a)Padding　(b)No Padding

爲了找到最佳的 Padding 設定，我們定義一個 Padding Model，需要關於 NAND Flash Memory 的參數 C_e（erase cost）、C_w（write cost）、N_{BlkPg}（Pages per Block）。沒有把 read cost 放入 Model 中，是因爲不管是 SLC[14, 15]或者 MLC[16]的 NAND Flash Memory，讀取代價相對於寫入和抹除的代價來說可以忽略不計，而且有些 NFTL mapping 方式在讀取一個 Page 時可能要額外讀取數個 Page 的 Spare Area，帶進許多無關緊要的參數只會增加 Model 的複雜度。Model 中我們需要有平均的統計數據 R_{DB}、R_{LB}、U_{LB}，藉由 R_{LB} 和 U_{LB} 我們可以算出 Log Block 在回收前平均會被寫入的總資料量，該值除以 B_W 爲 Expected number of write requests（即 Buffer 一共需要 flush 幾次才可以寫滿這些資料量）。有了這些資訊後我們可以分別計算出等式(1)：Padding and Switch Merge Cost 和等式(2)：Logging and Full Merge Cost，讓等式(1)=(2)我們推導出 B_W 的等式(3)。在 BAST 的架構下由於每次回收的 Block chain 內的資料都屬於同一個 LBA，所以 R_{DB}=1，代入等式(3)得到 BAST 適用的 B_W 等式(4)；而在 FAST 架構下則會回收最早被寫滿的那個 Log Block，即 R_{LB}=1、U_{LB}=1，得到 FAST 適用的 B_W 等式(5)。

B_w 爲 Expected balanced burst of write pages，該值代表的意義爲當 victim Group 含的 Page 資料量剛好等於 B_w 時，Model 預期做 Padding and Switch Merge 跟 Logging and Full Merge 的代價爲相等的。因此當 victim Group 含的 Page 資料量大於 B_w 時，我們的 Model 預期做 Padding 對 NFTL 的代價會最低；相對的如果小於 B_w，Model 預期把資料寫到 Log Block 的代價最小所以不做 Padding。在定義 Padding Model 後，我們的 Buffer 管理演算法 HitStat 可以藉由算出的 B_w 值動態決定 Padding 條件來跟不同的 NFTL 配合，儘量減少 NFTL 的 overhead 以提昇 SSD 效能。

R_{DB} - average number of Data Blocks to merge
R_{LB} - average number of Log Blocks to merge
U_{LB} - average space utilization of Log Blocks
B_w - expected balanced burst of write pages
N_{BlkPg} - number of pages per block
C_e - cost of erase a block
C_w - cost of write a page

Example:



Pages written to logblock = $R_{LB} \times N_{BlkPg} \times U_{LB}$

Expect number of write requests = $\frac{R_{LB} \times N_{BlkPg} \times U_{LB}}{B_W}$

(1) Padding & Switch Merge Cost = $\frac{R_{LB} \times N_{BlkPg} \times U_{LB}}{B_w} \cdot \left[C_e + N_{BlkPg} \times C_w \right]$

(2) Logging & Full Merge Cost = $\left[R_{LB} + R_{DB} \right] \cdot C_e + \left[R_{LB} \times N_{BlkPg} \times U_{LB} + R_{DB} \times N_{BlkPg} \right] \cdot C_w$

Let (1) = (2) $\Rightarrow B_w = \frac{R_{LB} \times N_{BlkPg} \times U_{LB} \cdot \left[C_e + N_{BlkPg} \times C_w \right]}{\left[R_{LB} + R_{DB} \right] \cdot C_e + \left[R_{LB} \times U_{LB} + R_{DB} \right] \cdot N_{BlkPg} \times C_w}$ (3)

on BAST $R_{DB} = 1$ $\Rightarrow B_w = \frac{R_{LB} \times N_{BlkPg} \times U_{LB} \cdot \left[C_e + N_{BlkPg} \times C_w \right]}{\left[R_{LB} + 1 \right] \cdot C_e + \left[R_{LB} \times U_{LB} + 1 \right] \cdot N_{BlkPg} \times C_w}$ (4)

on FAST $R_{LB} = 1$ $U_{LB} = 1$ $\Rightarrow B_w = \frac{N_{BlkPg}}{R_{DB} + 1}$ (5)

使用 Padding 可以把 Full Merge 轉變爲 Switch Merge，但是因爲做 Switch Merge 可能會提早回收 Log Block 造成 Log Block Utilization 下降。爲了減少這個誤差造成的影響，我們 U_{LB}、R_{LB} 的統計方式除了正常做 Full Merge 的操作外，對於因爲 Switch Merge 導致 Log Block 被回收的情況，我們統計方式爲該 victim Group 在 Padding 前的資料是先寫到 Log Block 後再做 Full Merge 來計算。

四、Implementation

4.1 An approximation of *P(Group$_i$)*

在章節 3.3 我們的 Buffer replacement policy 依照 Cost-Benefit Analysis 選出值最小的 Group 成爲 victim，但是由於方程式(2)中 *P(Group$_i$)*的值代表未來會 hit 的機率，在 on-line 的 Buffer 管理中不能直接得到該值，而且在不同用途的電腦上會有不同的 pattern，因此必需使用能隨實際 workload 而變動的方法來預測 *P(Group$_i$)*，在實際環境中我們會使用純量 Age Rank 來代替 *P(Group$_i$)*。

4.1.1 Age Transformation Function

我們會定義一個 Age Transformation Function，把 Group 的 Age（即距該 Group 上一次被寫入的 request 間隔）值代入該 Function 後得到的回傳值即爲 Age Rank。Age Rank 值的範圍可以從 1 到 Levels，而 Levels 是我們設定的正整數可以從 1 到無限大，最近被寫入的 Group 我們會給較大的 Age Rank 值，這是因爲時間區域性的關係該 Group 再次被寫入的機會較大。

當 Levels 設定較大時，則我們的方法較接近 BPLRU，會保留時間區域性的資料，這是因爲各個 Group 得到的 Age Rank 可以有較大的差異性，最近被寫入的 Groups 會得到較大的 Age Rank 值愈有機會留在 Buffer 中；如果 Levels 設定較小，則我們的方法較接近 FAB，因爲 Age Rank 的差異性較小所以會優先寫出 Weight 較大的 Group，把含少量資料的 Group 留在 Buffer 中，當 Levels=1 時我們的方法會跟 FAB 一樣。

我們在後面的實驗中會依照每個 period 的 Group Miss 次數來動態調整 Levels，

目的是最小化 Group Miss 的次數。因此我們的方法介於 FAB 和 BPLRU 之間，既會把最近被寫入的 Group 留在 Buffer 中也會留住含資料量少的 Group，而在不同的 write pattern 下我們的方法可以根據統計數據來動態調整，讓 Buffer 內保留的資料可以偏向時間區域性或者空間區域性。

4.1.2 Hit Log: Dynamically Define Age Transformation Function

要隨 write pattern 的不同來定義 Age Transformation Function，我們會使用 Hit Log（circular queue）來記錄最近 Buffer 中被 Hit 的 Group 當時的 Age 資訊，然後再根據 Hit Log 內的記錄來定義 Age Transformation Function。如何定義參考圖 11 中的範例，圖 11(a)是 32 筆由小到大的數據，爲經過排序後的 Hit Log 資料，這邊我們設定 Levels=9，然後從 32 筆數據中按等份取出 8 份資料，接下來我們可以依照這 8 份資料來定義 Age Transformation Function，如圖 11(b)這 8 份資料可以把 Age 區分成 9 個區段來給定 Age Rank，Age 最大的區段得到的 Age Rank 值爲 1，接下來逐漸遞增，Age 最小的區段得到的 Age Rank 值即爲 Levels。

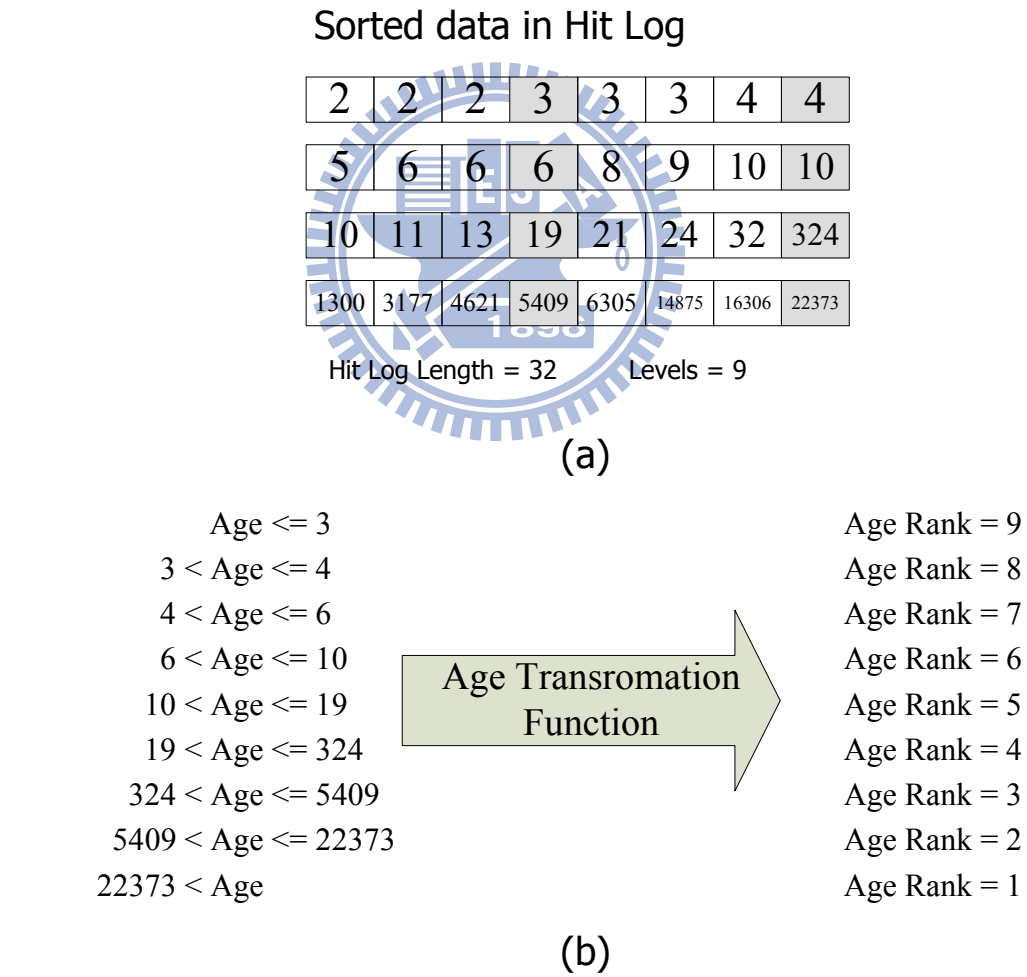


圖 11：Illustration of (a) Hit Log，(b) Age Transformation Function
Age：距該 Group 上次被寫入的 request 間隔

圖 11 中有兩個參數 Hit Log Length 和 Levels 需要設定，Hit Log Length 相當於要抓 locality 時取的 window size，設定太小會容納不下 locality，太大會導致抓到的

locality 不精準，而且較大的 array size 需要較久的排序時間，在後面的實驗中我們設定 Hit Log Length=64 可以當作參考。而 Levels 的設定在章節 4.1.1 中有說明，關係到 Age Transformation Function 的定義，由於最多只能從 Hit Log 中取出 Hit Log Length 筆資料來劃分各個 Age 區段，所以它的值不能大於 Hit Log Length+1。

4.2 Essential Data Structure

爲了能夠快速搜尋 Group、加入 Group、挑選 victim，必需設計有別於 FAB 及 BPLRU 的資料結構，使用單一個 LRU chain list 的資料結構並不能滿足 HitStat 的需求。因爲 HitStat 的 replacement policy 使用 Cost-Benefit Analysis 需要同時考慮每個 Group 的 Age 以及 Weight，如果要計算出每個 Group 的 Cost-Benefit 值並找到值最小的成爲 victim Group 需要很大的計算量，而且隨著 Buffer size 變大 Buffer 中的 Group 數量會增加，計算時間也會變長。

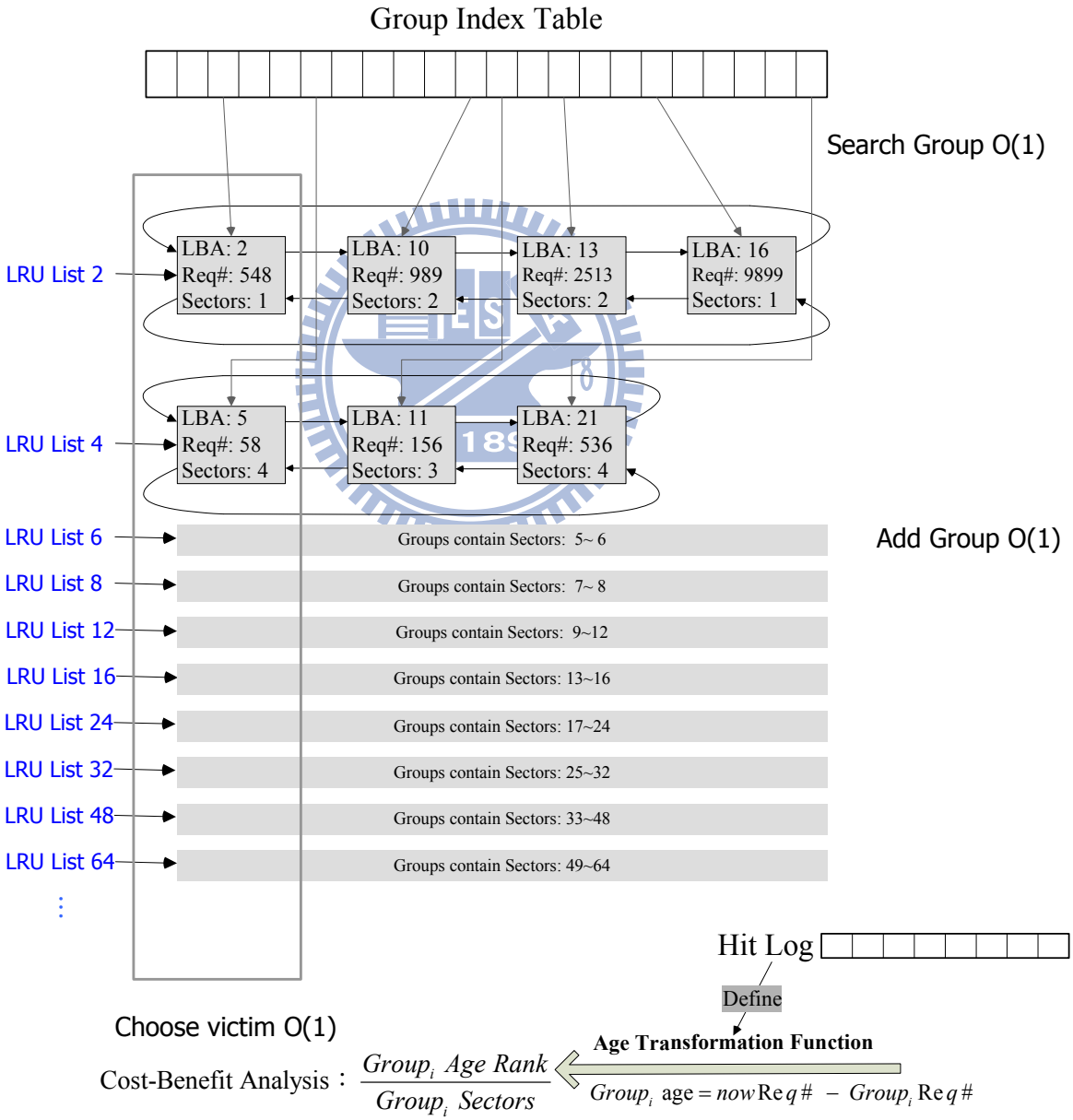


圖 12：Data Structure of HitStat

圖 12 是我們爲 HitStat 設計的資料結構，有一張 Group Index Table 記錄 Buffer 中每一個 Group 位置的指標，搜尋時可以查詢 Group Index Table 直接找到該 Group；Hit Log 內的記錄資訊會用來定義 Age Transformation Function；另外我們會管理數個 LRU Lists，對於每個在 Buffer 中的 Blocks Groups，我們依照 Weight（Group 包含的 Sector 數量）把它們加入相對應的 LRU List，圖中有 LRU List 2、LRU List 4、LRU List 6 …，表示說含 Sector 數量介於 1~2 之間的 Group 都在 LRU List 2 中、含 Sector 數量介於 3~4 之間的 Group 都在 LRU List 4 中…依此類推，由於我們的 Buffer 方法會留住許多含少量資料的 Group，因此我們會讓較小 Sector 數的 LRU Lists 之間間距較小、較大 Sector 數的 LRU Lists 之間間距較大，如在 Block size=512KB 時，我們設定 LRU List 有 2, 4, 6, 8, 12, 16, 24, 32, 48, 64, 96, 128, 192, 256, 384, 512, 768, 1024 這幾種，一共 18 個 LRU List 就可以分配 weight 範圍從 1-1024 的 Groups。

每一個 Group 內都會記錄三筆資訊，LBA 記錄 Group 內的資料是屬於哪一個 Logical Block Address；Req#記錄最後一筆寫入這個 Group 的 request number；Sectors 記錄該 Group 總共包含幾個 sector 資料，可以直接做爲 Weight 值。在計算某個 Group 的 Cost-Benefit 時，會先把現在的 Req Number 減掉該 Group 的 Req#得到該 Group 的 Age，再把該 Age 代入 Age Transformation Function 後可以得到 Age Rank 值，最後用 Age Rank 除以 Group 的 Sectors 即可算出 Cost-Benefit 的值。

4.3 overhead Analysis

對我們設計的資料結構分析其搜尋、加入、挑選 victim Group 的 Time Complexity。由於有 Group Index Table 記錄位置指標，搜尋 Group 時直接查詢該 Table 就可以直接找到 Group，搜尋 Group 的時間爲 O（1）；要加入或因新的寫入資料搬移 Group 時，依照 Group 中的 Sector 值找到應該加入的 LRU List，因爲是 LRU 所以只要把 Group 加入該 list 的尾端即可，加入 Group 的時間爲 O（1）；而挑選 victim Group 我們只需針對每個在 LRU List 最前端（List 中年紀最大）的 Groups 計算它們 Cost-Benefit 的值，從這些候選的 Group 中決定哪一個成爲 victim，因此挑選 victim Group 的時間也只需 O（1）。

另外我們跟 BPLRU 比較 RAM space overhead，HitStat 需要額外的空間爲數個 LRU List pointers、Hit Log 陣列以及每個 Group 的 Req#。假設有 17 個額外的 LRU List pointers、Hit Log 陣列大小爲 64、一共有 2000 個 Groups，每一筆以 4Byte 來計算，HitStat 所耗費的 RAM space 比起 BPLRU 只多出 8KB 左右的空間。

4.4 Other Optimizations

4.4.1 Detail Replacement Policy

除了用 Cost-Benefit Analysis 來挑選 victim 外，另外有兩種 Group 應該有較高優先權被挑選爲 victim。(1)如果某個 Group 內的資料已經滿了的話，這 Block 內的資料極有可能是 sequential write，因爲 sequential 資料未來很少有機會再被寫入，而且非常佔 Buffer 空間所以應立即被寫出。(2)我們會設定一個 Age Threshold，如果某個 Group 的 Age 超過該值，則優先把該 Group 選爲 victim 而不透過 Cost-Benefit Analysis，如此我們可以藉由 Age Threshold 決定留在 Buffer 中最久的 Group 其 Age 上限。因此 HitStat 選 victim Block 的優先順序爲：1.被寫滿的 Group（sequential write）；2.當 Age of Group > Age Threshold（ex: 150000），3.依 Cost-Benefit Analysis 選出值最小的 Group。

4.4.2 The HitStat Adjustment

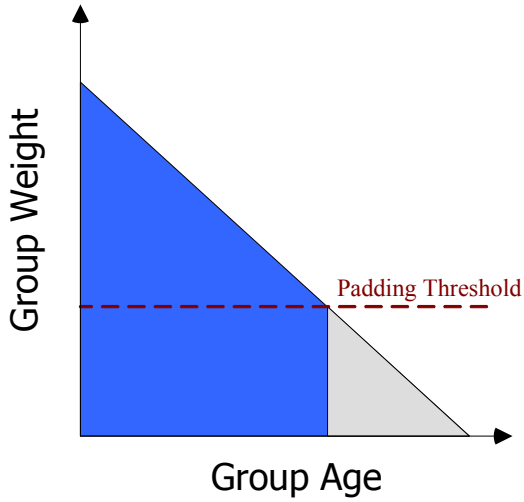


圖 13：The Group characteristics of HitStat and the Padding Threshold

爲了能跟 NFTL 中的 Log Block 做更好的配合，HitStat 在選擇 victim 的策略仍然可以改進。參考圖 13，HitStat 的 replacement policy 依照 Cost-Benefit Analysis 除了留住年輕的 Group 外也會留住未達到 Padding 條件且 Age 較大的 Group，這些 Group 屬於冷資料被更新的機會很低，即使將來能被 Hit 而使其 Weight 增長一些，但大部份也不會達到 Padding Threshold，造成這些 Group 佔住部份 Buffer 空間很久的時間，但最後還是要寫到 Log Block，這樣不如把這些資料早點從 Buffer flush 到 NFTL 由容量更大的 Log Block 來處理。因此我們稍微更改了 HitStat 的 replacement policy，在計算 Cost-Benefit 挑選 victim 時，會把 Weight 值小於 Padding Threshold 的 Groups 讓其 Weight 值等於 Padding Threshold 再代入 Cost-Benefit Analysis 計算，相當於讓 Weight 在 Padding Threshold 以下的 Groups 整體爲一個 LRU，調整過後的方法我們稱爲 HitStat(adj)，HitStat(adj)會把圖 13 中淺灰色區域 Weight 較小且稍冷的 Group 提早寫到 NFTL，讓 Buffer 有更多空間來留住較年輕的 Group。但當發生 Log Block Thrashing 時並不適合使用 HitStat(adj)，因爲 flush 出 Weight 小的 Group 數量會變多，反而使 Thrashing 現象更加嚴重。在後面的實驗中，我們會在 FAST 上使用 HitStat(adj)；而 BAST 則因爲有 Log Block Thrashing 問題，所以還是使用原本的 HitStat。

五、Performance Evaluation

5.1 Experiment Settings

在我們的實驗中，所使用的 Buffer size 和 NAND Flash Memory 規格如表 1，其中 NAND Flash Memory 的 Block size、Page size 和抹除及讀寫時間我們參考[16]，所有的實驗如果沒有特別說明皆在這個設定下完成。另外我們會在不同大小的 Buffer size 情況下測試各種 Buffer 管理演算法，以觀察隨著 Buffer size 變化對 Buffer 特性的影響以及整體效能的趨勢。在實驗測試的過程中，我們會在 Windows XP 上使用 diskmon[17]蒐集爲期一個月中 Operating System 對 Hard disk 所下 write 指令，將這些指令對各種 Buffer 和 NFTL 整合的架構進行測試。

表 1：Environment Setup

Buffer size (MB)	2, 4, 6, 8, 10, 12, 14, 16, 20, 24, 28, **32**, 40, 48, 56, 64, 80, 96, 112, 128
Sector size	512 Bytes
Page size	4 KB
Block size	512 KB
Pages per Block	128
Total Physical Blocks	40,960
Total Logical size of SSD	20 GB
Log Blocks of BAST	2048 (1 GB)
Log Blocks of FAST	128 (64 MB)
Time of Page Read	60 us
Time of Page Write	800 us
Time of Block Erase	1500 us

我們的 Buffer 主要比較對象爲 FAB[2]以及 BPLRU[3]。FAB 的 Buffer replacement policy 爲每次挑選含最多資料的 Group 當作 victim，沒有 Padding 的機制；而 BPLRU 的 Buffer replacement policy 則是依 LRU order 來管理 Group，關於 Padding 的設定我們參考相關論文[4]中的 fragmented write 設定，假定當 Group 的資料滿一半時 BPLRU 就會使用 Padding 機制。表 2 爲關於三種 Buffer 管理演算法的 replacement policy 和 Padding 設定之比較。

表 2：Comparisons of Buffer Management Algorithms

Name	Buffer Replacement Policy	Padding	Default Settings	Reference
FAB	1. Group contains the most sectors. 2. Group of Least Recently Used.	No		[2]
BPLRU	1. Full Group. 2. Group of Least Recently Used.	>= 0.5 refer to [4]		[3, 4]
HitStat	1. Full Group. 2. Group Age > Threshold=150000. 3. Cost-Benefit Analysis.	Padding Model	HitLog_Len=64 Rank_Initial=32 Padd_Initial= 1.00 on BAST 0.33 on FAST	

而實驗中我們使用基本的 NFTL 架構 BAST 和 FAST，把 Buffer 管理演算法架設在這兩種 NFTL 上比較它們的整體效能。有關 BAST 和 FAST 的基本設定參考表 3。

表 3：The Settings of NFTL

Name	Data : Log Blocks	GC Policy	Reference
BAST	1 : N	1. Log Block Length>4 2. RR lookahead 50, select longest chain	[5]
FAST	M : 1-N	The oldest written Log Block	[7]

我們對從一般的使用環境中截取的 Workload 資料做分析，分析結果如表 4，對同一個 LBA 寫入次數愈多的數量愈少，表示大部份的資料都僅寫入數次，即 cold data 較多，只有少數的資料寫入次數較高，即 hot data 較少，可知 hot data 的數量遠少於 cold data 的數量。另外我們定義，如果跟前後 10 筆 write requests 寫入的資料位置有連續的情況，則將該筆 request 歸類爲 sequential write，否則爲 random write；另外若該筆 request 的起始位置之前有被寫過，我們歸類爲 update request。分析結果顯示 sequential write 佔所有的 request 比例 30%，其它 70%爲 random write，可知 workload 中大部份爲 random write，並摻雜少量的 sequential write。

表 4：disk trace Analysis

Total Writes(次數)：1,509,409		
	sequential	random
write requests	82,648	102,174
update requests	359,161	965,426
	29.3%	70.7%
write size (MB)	4,545	3,022
update size (MB)	11,686	7,176
Update 次數	LBA 個數	
0		26,646,903
1~100		15,286,890
101~1000		8,791
1001~10000		419
10001~30000		21
>=30001		16
	Total：41,943,040 個 LBAs	

5.2 Characteristics of NFTL & Write Buffer Policies

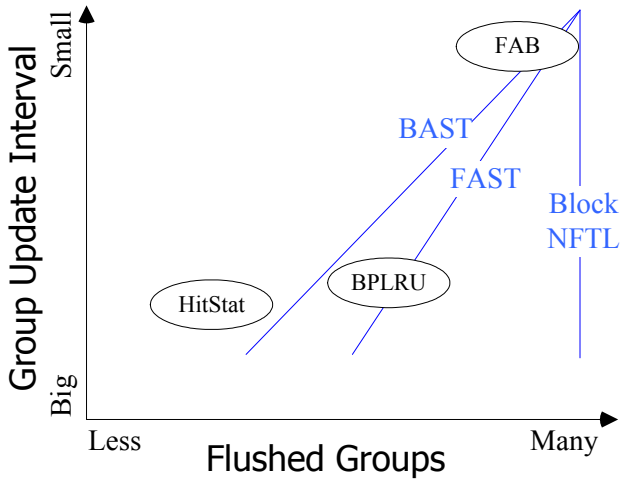


圖 14：Characteristics(flushed Groups、Update Interval) of Buffer on various NFTLs

圖 14 我們針對 Buffer 的 flushed Groups 總數、Group Update Interval 這兩種特性以二維的方式對三種 NFTL（Block-level NFTL、BAST、FAST）來做分析。橢圓代表各 Buffer 方法的二維特性約略位置分佈，X 軸爲 flushed Groups 參考圖 15(a)，愈小愈好因爲代表寫到 NFTL 的資料愈不 random；Y 軸爲 Group Update Interval，參考圖 15(b)，代表的意義爲先後 flush 出兩個屬於同一個 LBA 的 Group 間隔，Y 愈大愈好這是因爲如果 Group Update Interval 間隔較短，可以較好的利用 Log Block 來吸收屬於同一個 Block 的資料。

因此我們可以估計 Erase Count 約略等於 X – α Y，α Y 代表 Log Block 能夠吸收的 Erase Count 數量，其中 α 爲非負值，在不同的 NFTL 架構有不同的 α 值，由於 Block-level NFTL 沒有 Log Block 所以 α =0，而 BAST 因爲有 Log Block Thrashing 問題，對於是否能夠充份利用 Log Block，Group Update Interval 的影響較大，所以比 FAST 的 α 值還大。EC = X – α Y 經過的點代表在該 NFTL 架構下會得到接近的 Erase Count，而在線的左邊距離愈遠代表 Erase Count 愈少，因此就這兩個特性的分佈來說，在圖的愈左上方愈好，表示從 Buffer 寫到 NFTL 的資料較不 random，而且會重覆寫到相同的 LBA。但事實上 Buffer 要兼顧兩個特性是互相衝突的，因爲如果要使 flushed Groups 總數較少，Buffer 需要儘量吸收熱資料，所以寫到 NFTL 都是較冷的資料，造成 Group Update Interval 變大。

而我們的方法是最大化 Group Hit Ratio 可以使 flushed Groups 最少，這樣的特性可以適用在大部份 NFTL 架構上，參考圖 14 因爲 HitStat 在三條線的左邊距離最遠，所以我們預期 HitStat 在三種 NFTL 架構上的 Erase Count 都最少，而 FAB 的 Group Update Interval 較短適合用在 BAST 架構上、BPLRU 則較適合用在 FAST 架構上。

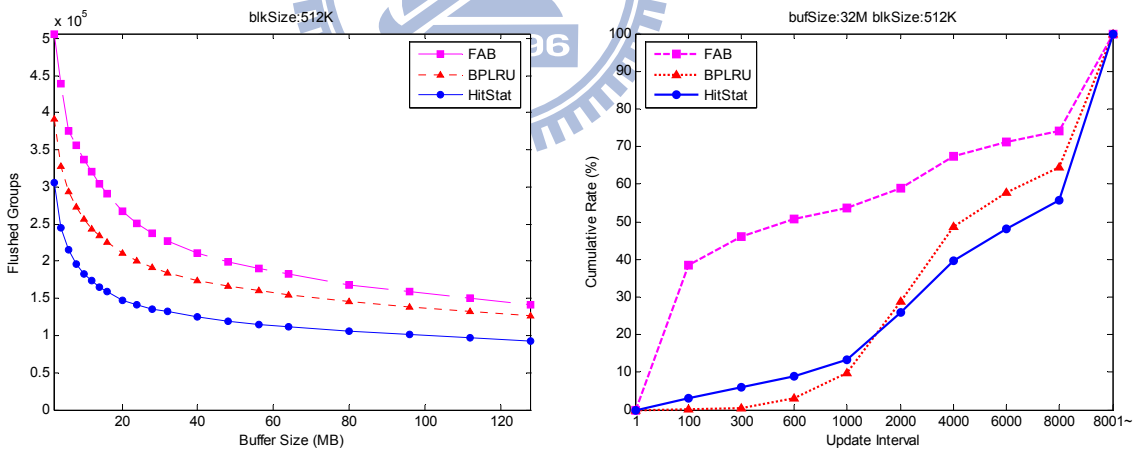
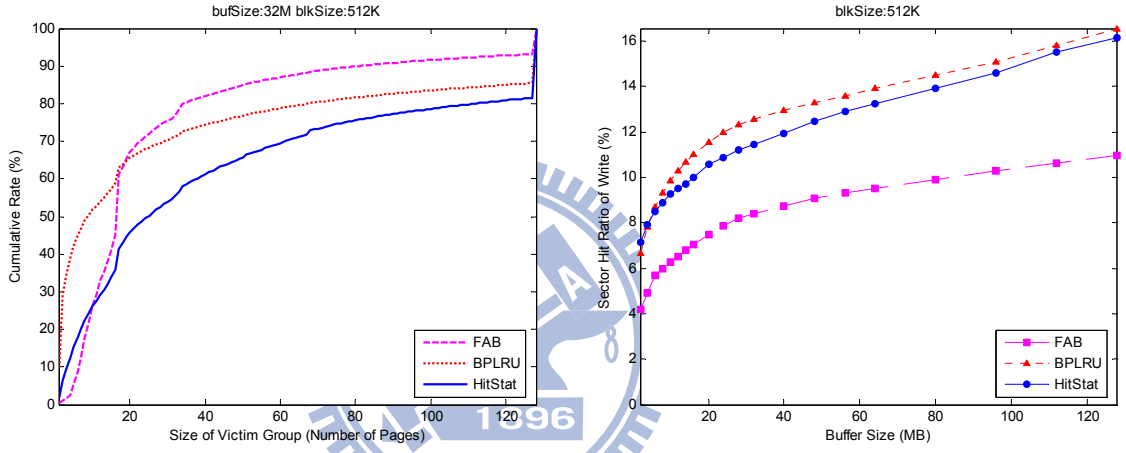


圖 15：Comparison the characteristics of Buffer
(a) Number of flushed Groups, (b) Update Interval

圖 15(a)比較三種 Buffer 方法 flush 出的 Group 總數。以沒有 Log Block 的 Block-level NFTL 來說，flushed Groups 總數就等於 Erase Count，因此它是比較 write Buffer 管理方法的基本數據。圖中 FAB 的 flushed Groups 數爲最多，因爲 FAB 只寫出含最多資料的 Group，沒有考慮留住熱資料，這些沒有被 Buffer 吸收的熱資料會造成多次的 Group flush，造成 flushed Groups 數拉高。而 HitStat 原本設計的考量即爲最大化 Group Hit Ratio，因爲 Group Miss 的次數最少，所以有最少的 flushed Groups 總數。

另外一項需要考量的數據爲是否可以利用到 NFTL 的 Log Block，良好運用 Log Block 可以吸收 Buffer 寫出屬於同一個 LBA 的資料來減少 Merge 的代價。但是由於 Log Blocks 的容量有限，如果寫到 Log Block 的 Group 資料很久之後才會再從 Buffer flush，（參考圖 10(a)）那之前寫入屬於同一個 LBA 的資料很可能早已做 Full Merge 而不能讓 Log Block 吸收 Merge 的代價，反而造成 overhead。因此我們在圖 15(b) 比較 Update Interval，即 Buffer flush 出兩次屬於同一個 LBA 的 Group 之間的間隔，如果 flush 的資料其 Update Interval 愈小，表示相同的 flush 次數內有較多次屬於相同 LBA 的 Group 資料被寫到 Log Block 中，Log Block 愈有機會能夠吸收 Merge 的代價。圖 15(b)以 CDF（Cumulative Distribution Function）方式來呈現，如 Update Interval 爲 2000 以下的次數 FAB 佔 59.0%、BPLRU 佔 28.7%、HitStat 佔 25.7%，FAB flush 的資料很多是 Update Interval 低的，這是因爲 FAB 著重的是把含資料少的 Group 留在 Buffer 中，而最近被 Access 的 hot data 或 sequential data 較容易成爲 victim，這些資料變成需要交給 Log Block 來吸收。



(a) (b)

圖 16：Comparison the characteristics of Buffer
(a) Size of flushed Groups, (b) Sector Hit Ratio

圖 16(a)以 CDF 方式來比較 victim Group 所含的資料量。參考章節 3.4，Padding 能把代價高的 Full Merge 變成 Switch Merge，而 victim Group 含的資料量愈多，代表做 Padding 需要額外搬移的資料量愈少，Padding 的幫助也愈大。圖中寫出資料含 40 個 Page 以上的 Group，HitStat 佔 38.7%、BPLRU 佔 25.5%、FAB 佔 17.8%，HitStat 寫出的 Groups 大部份含的資料量較多，這跟圖 15(a)中有效減少 flushed Groups 總數有關，因爲寫出愈少的 Group 代表平均每個 Group 內含的資料量愈多，配合 Padding Model 來做 Padding 可以最大幅的減少 Merge 所需的代價。另外 FAB 含資料量爲 16 ~ 17Page 的 flushed Group 其 Cumulative Rate 一舉從 45.2%上升到 61.3%，這是因爲資料量稍多的 Groups 會優先被選爲 victim，造成 Buffer 被許多含資料量少的 Group 佔住空間，從這邊可以看出 FAB 留在 Buffer 中的 Groups 所含的資料量大部份都在 17 個 Page 以下。

圖 16(b)中我們比較 Sector Hit Ratio，如果 Buffer 的 Sector Hit Ratio 愈高，Buffer 必需要 flush 到 NFTL 的總資料量也會愈少。圖中 FAB 的 Sector Hit Ratio 最低，這是因爲 FAB 的 replacement policy 沒有考慮到時間區域性的緣故；而 BPLRU 以 LRU order 選擇 victim 能使 Sector Hit Ratio 最高；HitStat 同時考慮時間區域性和空間區域性只比 BPLRU 稍微低一些。Sector Hit Ratio 愈高表示 Buffer 能夠吸收大部份的 hot data，

由於 flush 的都是較冷的資料，造成 Group Update Interval 變高(圖 15(b))，如 HitStat 和 BPLRU；而 FAB 的 Sector Hit Ratio 很低，相對的其 Group Update Interval 較低，表示從 Buffer Flush 的還有很多熱資料，需要由 Log Block 來處理。

5.3 Performance Evaluation

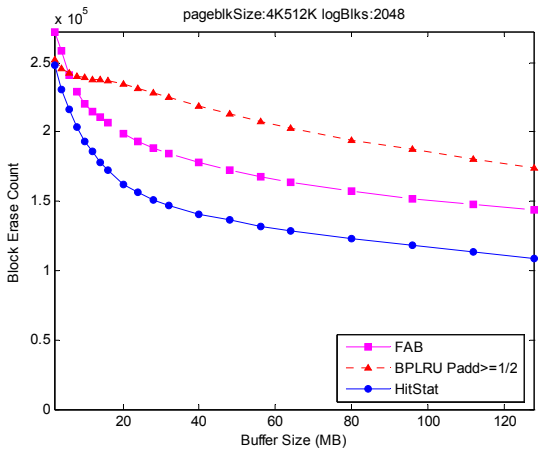
這一節我們把 Buffer 管理演算法與基本的 NFTL 架構配合，分別在 BAST 以及 FAST 上面比較效能。對於 NAND Flash Memory 來說，由於抹除以及寫入的代價很高，是影響寫入效能的主要因素，所以這邊我們會比較 Block Erase Count、Page Write Count。然後我們把寫入的總資料量除以 erase、write 花費的總時間來比較寫入的 Throughput，另外 Log Block 的使用情況也是我們比較的對象。

5.3.1 Using Buffer on Block-level mapping NFTL

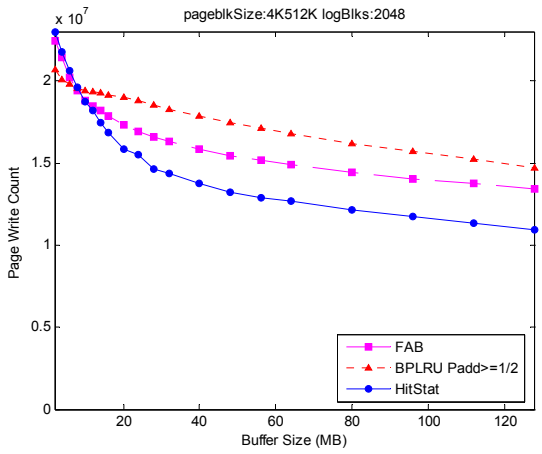
Block-level NFTL 由於沒有 Log Block，Buffer 每次 flush 出 Group 都需要取代舊的 Data Block 資料，所以得到的 Erase Count 會等於 flushed Groups 總數，而 Page Write Count 也跟 Erase Count 成等比例的關係，因此寫入效能跟 flushed Groups 總數成反比（參考圖 15(a)），HitStat 效能最高，BPLRU 次之，FAB 由於沒有 Log Block 的幫忙所以效能最低。

5.3.2 Using Buffer on BAST

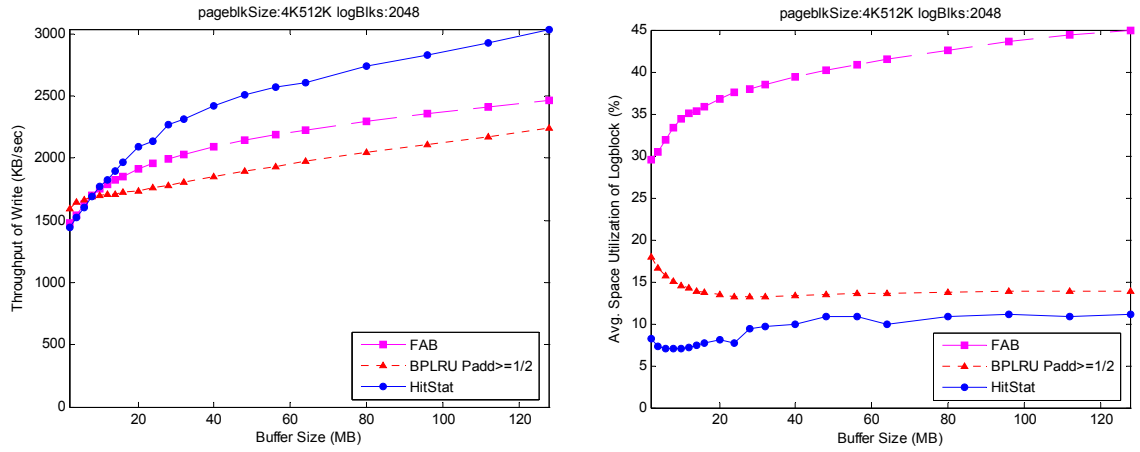
圖 17(a, b)為 Block Erase Count 以及 Page Write Count 的比較，兩者沒有成比例的關係，這是由於 BAST 在做 Full Merge 時，被回收的 Log Block 中可能有還未被寫入資料的 Page。在 BAST 上的 Buffer 要儘量留住大部份資料較少的 Group，才可以降低因為 Log Block Thrashing 所造成的額外 overhead，就這點來說 HitStat 和 FAB 有這個特性；而 BPLRU 由於缺乏這個特性，雖然 Buffer size 變大可以減少 flushed Groups 的總數，但同時也讓 Group Update Interval 變高，反而造成 Log Block Thrashing 問題更加嚴重，引發大量的 Full Merge，所以即使 BPLRU 的 Buffer size 變大，Block Erase Count、Page Write Count 下降的幅度卻很小。



(a)



(b)



(c) (d)

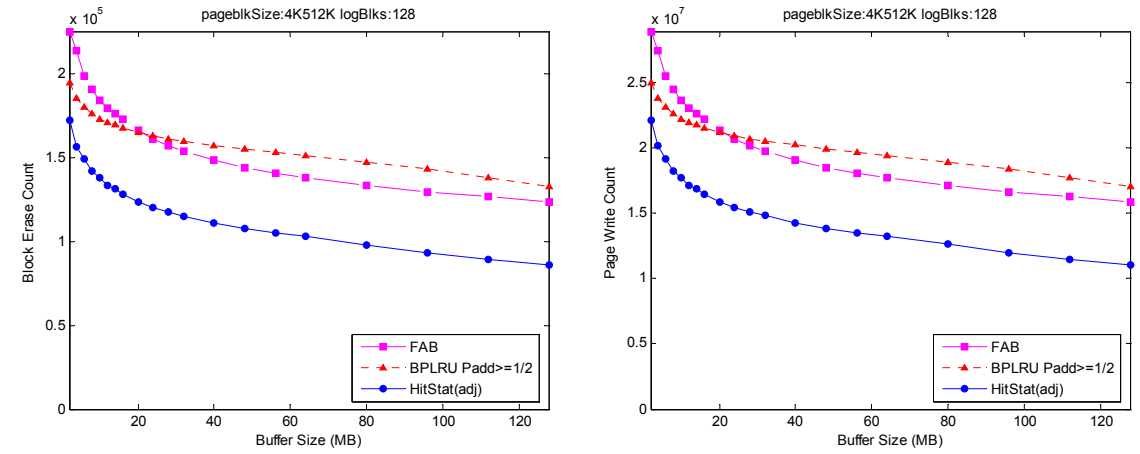
圖 17：Comparison the throughput on BAST
(a) Block Erase Count, (b) Page Write Count, (c) Throughput of Write, (d) Space Utilization of Log Block

圖 17(c)爲整體的 Throughput，可以看出影響 Throughput 的主要因素爲 Page Write Count。雖然 Block Erase 操作的時間爲 1500us 是 Page Write 800us 近兩倍的時間，但由於 1 個 Block 含有 128 個 Page，所以 Page Write Count 的基數比 Block Erase Count 大很多，對 Throughput 影響較大。

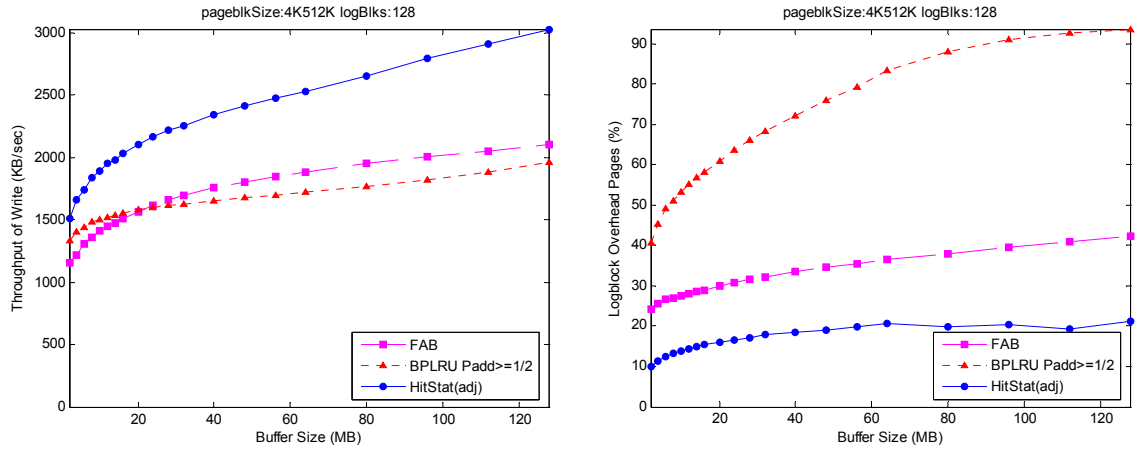
圖 17(d)爲 Log Block 的平均 Space Utilization，即 Log Block 回收前有被寫入資料的 Page 所佔比例。這邊可以看出 Log Block 的 Space Utilization 偏低都在 45%以下，這是因爲如果寫到 NFTL 的資料其 Group Update Interval 較高，在 BAST 架構下會有 Log Block Thrashing 的問題，屬於不同 LBA 的資料會競爭有限的 Log Block，造成大部份 Log Block 內只寫入少量資料就被回收的情況。參考圖 15(b)因爲 HitStat、BPLRU 的 Group Update Interval 比 FAB 高很多，造成 Log Block Thrashing 問題較嚴重，所以 HitStat、BPLRU 的 Log Block Utilization 比 FAB 低很多。

5.3.3 Using Buffer on FAST

這一節中我們以 HitStat(adj)來跟 FAB、BPLRU 比較，這是因爲 HitStat(adj)會把稍冷且沒達到 Padding 條件的 Group 提前 flush 交給 Log Block 吸收（參考章節 4.4.2），這可以更有效利用 FAST 架構下的 Log Block，而不用擔心跟 BAST 一樣有 Log Block Thrashing 的問題。



(a) (b)



(c) (d)

圖 18：Comparison the throughput on FAST

(a) Block Erase Count, (b) Page Write Count, (c) Throughput of Write, (d) Overhead Pages to Log Block

Logblock Overhead Pages：寫到 Log Block 的資料沒有跟其它筆資料一起 Merge，造成 overhead 增加

由於 FAST 回收的 Log Block 其 Utilization 都爲 1，所以圖 18(a, b)的 Block Erase Count 以及 Page Write Count 爲成比例的關係，而圖 18(c)的 Throughput of Write 則跟兩者成反比，因此要有最好的寫入效能要儘量減少 Erase Count 次數。而要達到這個目的，Buffer 除了要儘量減少 flush 出的 Group 總數外，也要能妥善利用 Log Block 的空間。

HitStat(adj)的策略是讓能 inplace-update 的 Buffer 儘量收集 sequential 的資料後做 Switch Merge，而那些稍微冷的資料就交給 Log Block 去吸收。針對 Log Block 的資訊我們參考圖 18(d)的 Overhead Pages to Log Block，比例愈低表示 Log Block 吸收到愈多的 Merge 代價，圖中 HitStat(adj)的 Overhead Pages to Log Block 比例最小，這是因爲 HitStat 配合 Padding Model 把含較多資料的 Group 做 Padding 後可以直接 Switch Merge，避免大量資料進入 Log Block，讓其它資料可以在 Log Block 待較久的時間增加一起 Merge 的機會，參考圖 18(a, b)，這種策略使 HitStat(adj)的 Erase Count 最低。而對於 FAB 以及 BPLRU 來說，FAB 因爲 flushed Groups 數量太多（圖 15(a)），BPLRU 則是寫入太多單純造成 overhead 的資料到 Log Block（圖 18(d)），所以這兩個方法的 Erase Count 比起 HitStat(adj)高很多。

可以看到 FAB、BPLRU 兩者的效能在 Buffer size 約 20MB 左右交叉，這是因爲一方面 flushed Groups 的差距被拉近（參考圖 15(a)）；另一方面 BPLRU 的 Group Update Interval 會隨 Buffer size 增加而變大，但 FAB 的 Group Update Interval 變化不會太大。爲了說明這兩個因素的影響，隨著 Buffer size 變大，BPLRU 是往圖 14 的左下移動，而 FAB 是往圖 14 的左邊移動，所以 BPLRU 的效能被 FAB 超越。

5.4 Buffer size and Number of Log Blocks

這一節我們把重點放在不同比例的 Buffer 大小、Log Block 數量做比較。我們設置了不同的 Buffer 大小（1、2、4、8、16、32、64、128MB）與不同的 Log Block 數量（64、128、256、512、1024、2048）分別在 BAST（圖 19）與 BAST（圖 20）上比較它們的效能。在 BAST 上我們把 Page Write Count、Block Erase Count 分別乘以寫入和抹除需要花費的時間後以長條圖的方式呈現；而在 FAST 由於 Page Write Count 跟 Block Erase Count 爲成比例的關係，所以我們只比較 erase 次數，並分成因 Switch Merge 引發的 erase 與因 Full Merge 所引發的 erase 兩種。

參考圖 19 在 BAST 上，當 Buffer size 小於 8MB 的時候，增加 Log Block 數量能降低 overhead，但當 Buffer size 足夠大時，overhead 降低的幅度非常有限，這是因爲隨著 Buffer size 增加，Group Update Interval 會變大，造成 BAST 架構下的 Log Block Thrashing 問題會更嚴重，所以增加數量有限的 Log Block 並無法改善這個問題，除非 Log Block 的數量能大過大部份資料的 Update Interval，因此隨 Buffer size 變大會愈來愈難靠增加 Log Block 數量來改善效能。

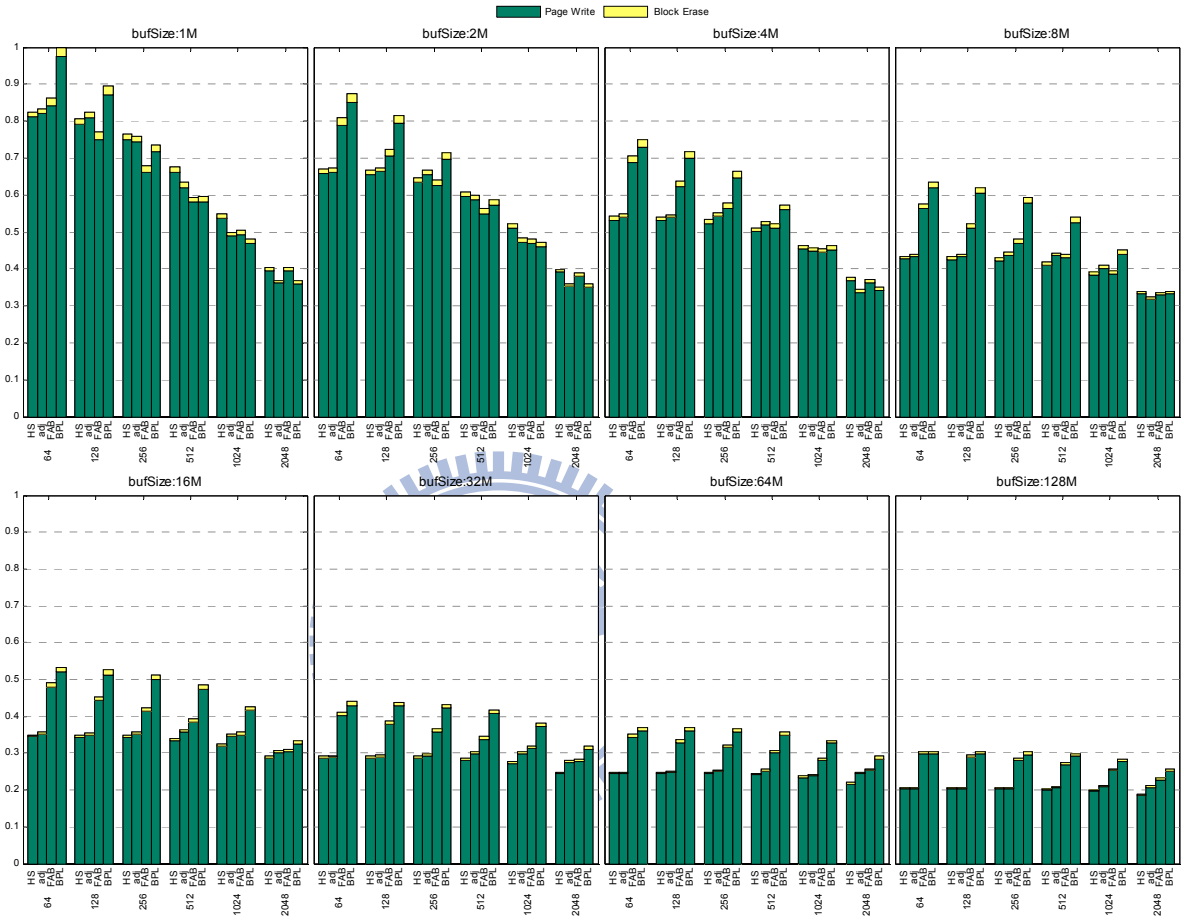


圖 19：Various Buffer size and Log Blocks on BAST
HS is HitStat、adj is HitStat(adj)、FAB、BPL is BPLRU
X 軸：Number of Log Blocks
Y 軸：Normalized Elapsed Time

參考圖 20 在 FAST 上，由於沒有 Log Block Thrashing 的問題，不論增加 Buffer 容量或 Log Block 的數量都能有效降低 overhead。而 FAST 的問題是做 Full Merge 的代價很高，做一次 Full Merge 可能會引發數個區塊需要抹除，因此由 Full Merge 引起的 Erase Count 愈少愈好。圖中 FAB、BPLRU 的 Switch Merge 數量並不會隨 Log Block 數量而改變，這是由於它們使用固定的 Padding 設定；而我們的 Buffer 演算法由於是按照 Padding Model 動態決定 Padding 的條件，Switch Merge 次數會隨 Log Block 數量的多寡而變化，在 Log Block 較少的時候 Padding Model 會降低 Padding 條件的門檻，增加 Switch Merge 的次數來減少 Full Merge 的代價，能夠控制住因爲 Full Merge 所引發的系統 delay 問題。在 Buffer 爲 32MB、Log Block 數量爲 128 的條件下，FAB 的 Full Merge erase 比例最高佔 89.9%，BPLRU 次之佔 76.1%，而我們的 Buffer 方法 HitStat、HitStat(adj)只佔極少部份，分別爲 14.8%、16.1%。

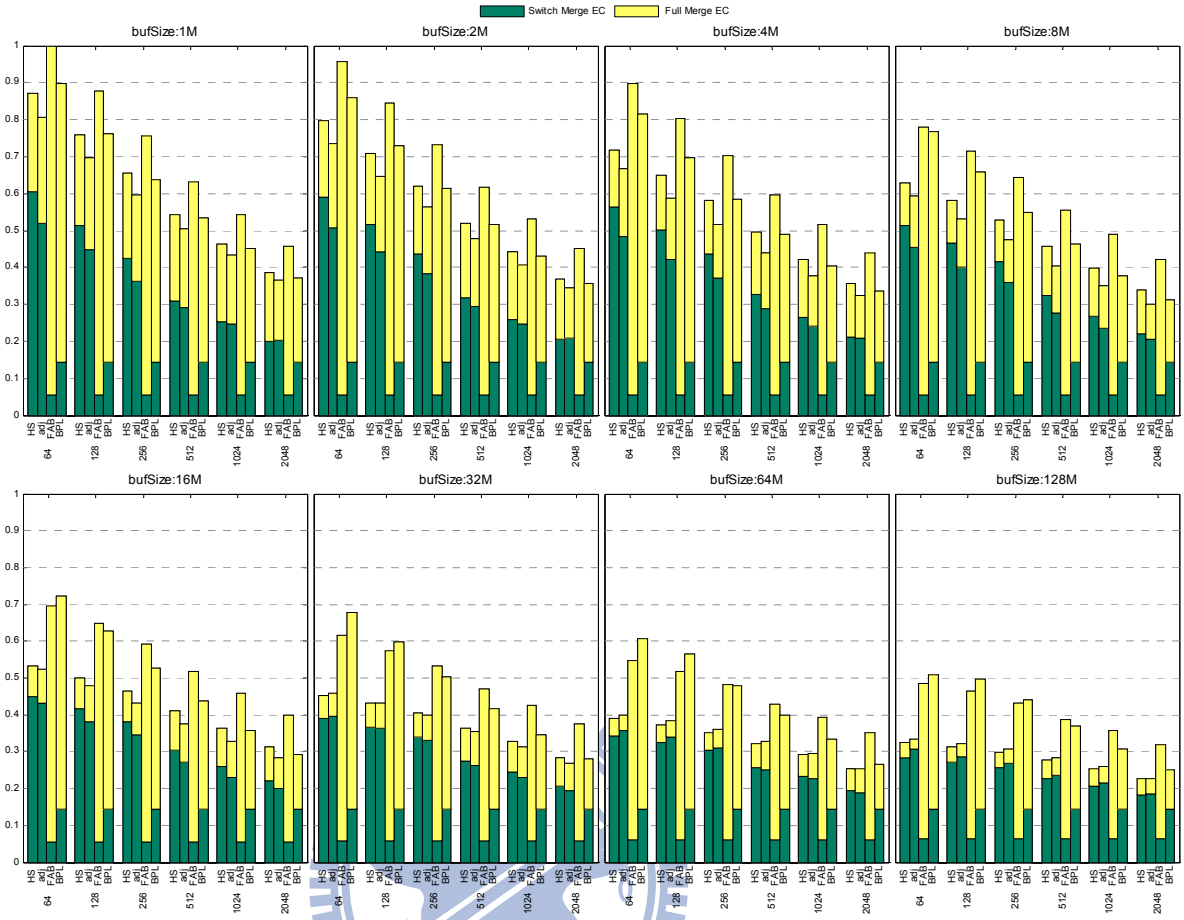


圖 20：Various Buffer size and Log Blocks on FAST
HS is HitStat、adj is HitStat(adj)、FAB、BPL is BPLRU
X 軸：Number of Log Blocks
Y 軸：Normalized Elapsed Time

在章節 4.4.2 中我們提出 HitStat(adj)，HitStat(adj)會把沒有到達 Padding 條件的 victim Group 提早 flush 寫到 Log Block，會比 HitStat 依賴 Log Block 的空間。在 BAST 上因爲 HitStat(adj)會使 Log Block Thrashing 的問題加重，所以只有當 workload 不會造成 Log Block Thrashing 或者 Buffer size/ Log Block size 比值較小（參考圖 19）的情況下適合使用 HitStat(adj)；而在 FAST 上由於沒有 Log Block Thrashing 的問題，所以適合使用 HitStat(adj)，參考圖 20 在 Buffer size 小於等於 16MB 的情況下 HitStat(adj)的 overhead 都低於 HitStat，而在 Buffer size 大於 16MB 時兩者差距不大。

5.5 The Padding Model

針對章節 3.5 中我們提出在 Hybrid NFTL 架構上的 Padding Model，HitStat 藉由該 Model 可以算出 B_W 來動態決定 Padding 條件。這一節我們會分別驗證在 BAST 與 FAST 上，HitStat 動態 Padding 的方式可以收斂接近到用 off-line 方式找到的最佳 Padding 設定，這樣可以妥善的利用 Log Block 空間，避免因爲過度或過少 Padding 導致額外的 overhead。

表 5, 6 中，每一列的資料代表每一個不同 Padding 設定所得到的數據，而第一欄 Fixed Padding 的值爲 Padding 的設定，如 0.25 表示若 victim Group 所含的 Page 資料個數大於等於 Pages per Block 的四分之一，則會先對該 Group 的資料做 Padding

後再寫到 NFTL 中；1.00 表示不做 Padding、0.00 表示 Always Padding。而 Block Erase 跟 Page Write 的值分別代表實驗所得到的 Block Erase Count 以及 Page Write Count。Elapsed Time 則是計算抹除及寫入動作所花的總時間，Elapsed Time 愈少表示寫入效能愈好。

表 5：Information of HitStat with BAST

Fixed Padding	Block Erase	Page Write	Elapsed Time(s)	R_{LB}	U_{LB}	B_W/N_{BlkPg}
1.00	179756	15684893	12817.548	1.10	29.863%	0.24
0.50	168620	14741808	12046.376	1.06	23.500%	0.20
0.33	162769	14500509	11844.561	1.04	20.001%	0.17
0.25	157567	14414711	11768.119	1.03	17.633%	0.15
0.20	154201	14387487	11741.291	1.02	16.487%	0.14
0.16	150274	14342751	11699.612	1.02	15.834%	0.14
0.13	145671	14631209	11923.474	1.02	14.366%	0.13
0.11	144621	14801632	12058.237	1.02	13.529%	0.12
0.10	142821	14888096	12124.708	1.02	13.111%	0.12
0.09	141930	14939008	12164.101	1.02	12.912%	0.11
0.08	141027	14982949	12197.900	1.02	12.838%	0.11
0.07	138339	15089628	12279.211	1.02	12.710%	0.11
0.06	136441	15125308	12304.908	1.02	12.970%	0.12
0.05	134724	15188352	12352.768	1.02	13.554%	0.12
0.04	132458	15272566	12416.740	1.03	14.390%	0.13
0.03	129059	15609567	12681.242	1.05	18.861%	0.16
0.02	128616	15928989	12936.115	1.07	20.308%	0.18
0.01	130778	16590525	13468.587	1.10	22.835%	0.20
0.00	131960	16890880	13710.644	N/A	N/A	N/A
dynamic by Model	147293	14340979	11693.723	1.02	16.190%	0.14

表 5 爲在 BAST 上的實驗數據，其中 Fixed Padding 設定爲 0.16 時可以得到最少的 Elapsed Time，是實驗中找到的最佳 off-line 設定。而根據章節 3.5 中 Padding Model 的方程式(4)，BAST 架構下需要統計 R_{LB}、U_{LB} 的值來計算 B_W，如表中各個實驗得到的數據把 R_{LB}、U_{LB} 代入方程式中，我們發現得到的 B_W/N_{BlkPg} 值會動態收斂到 0.12~0.14 的範圍，接近最佳的 off-line 設定 0.16。而造成些微的誤差原因可能爲 Padding 所造成的干擾，因爲 Padding 過的資料引發 Switch Merge 可能會提早回收 Log Block，我們的統計方式已儘量減少這個干擾所造成的影響。表中最後一列的實驗數據爲根據 Padding Model 來動態調整 Padding 條件，我們的 Padding 初始設定爲 1.00，最後動態調整到 0.14，Elapsed Time 甚至比最佳的 off-line 設定還少。

分析 U_{LB} 在 Fixed Padding 設定接近 1.00 會變大，這是因爲不論 victim Group 所含的資料大小都會寫到 Log Block 中；而設定接近 0.01 時也會變大，主要是我們改良過的統計方法把那些會引起 Switch Merge 並造成提早回收 Log Block 的 victim Group 當作其資料是先寫到 Log Block 後再做 Full Merge 來計算，因爲這種 case 比例增加的原因所以拉高 U_{LB}。

表 6：Information of HitStat with FAST

Fixed Padding	Block Erase	Page Write	Elapsed Time(s)	R_{DB}	B_W/N_{BlkPg}
1.00	143623	18400029	14935.458	3.98	0.20
0.50	133771	17138992	13911.850	5.75	0.15
0.33	129124	16544157	13429.012	7.26	0.12
0.25	124708	15978999	12970.261	8.58	0.10
0.20	121906	15620255	12679.063	9.28	0.10
0.16	119258	15281375	12403.987	9.76	0.09
0.13	117099	15004969	12179.624	11.47	0.08
0.11	116638	14946016	12131.770	12.51	0.07
0.10	115636	14817696	12027.611	12.95	0.07
0.09	115125	14752384	11974.595	13.05	0.07
0.08	114744	14703525	11934.936	13.06	0.07
0.07	114603	14685532	11920.330	13.14	0.07
0.06	114444	14665148	11903.784	12.42	0.07
0.05	114887	14721920	11949.867	11.80	0.08
0.04	115676	14822902	12031.836	9.86	0.09
0.03	120380	15424991	12520.563	6.77	0.13
0.02	123787	15858717	12872.654	N/A	N/A
0.01	129595	16590525	13466.813	N/A	N/A
0.00	131960	16890880	13710.644	N/A	N/A
dynamic by Model	115027	14739805	11964.385	12.36	0.07

表 6 爲在 FAST 上的實驗數據，off-line 找到的最佳 Fixed Padding 設定爲 0.06。根據章節 3.5 中 Padding Model 的方程式(5)，FAST 架構下需要統計 R_{DB} 的值來計算 B_W。表中算出的 B_W/N_{BlkPg} 值會收斂到 0.07，非常接近最佳的 off-line 設定 0.06。表中最後一列數據爲根據 Padding Model 來動態調整 Padding 條件，Padding 初始設定爲 0.33，最後會動態調整到 0.07，效能接近最佳的 off-line 設定。

分析 R_{DB} 在 Fixed Padding 設定接近 1.00 時會變小，這是因爲缺乏 Padding，許多從 Buffer flush 屬於同一個 LBA 的大資料會寫進 Log Block 中，這些資料雖然會稀釋使 R_{DB} 的值變小，但是造成要經常 Full Merge 來回收 Log Block，是 Elapsed Time 大量增加的原因；而設定接近 0.01 時也會變小，這是因爲只有含小量資料的 victim Group 會寫進 Log Block 中，所以寫到 Log Block 的資料量很少，使資料可以在 Log Block 中待很久的時間，更有機會被後來的資料 invalidate，所以 R_{DB} 值降低。

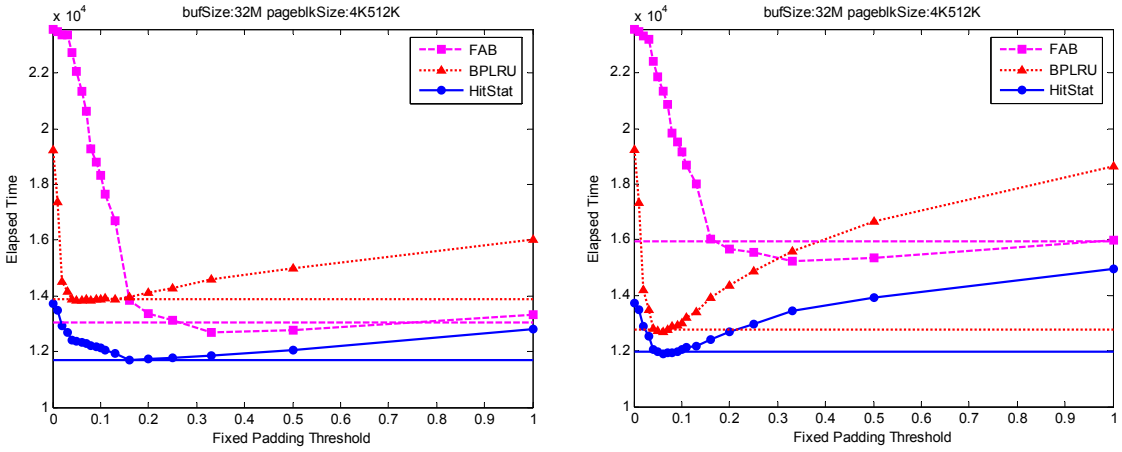


圖 21：Padding settings - fixed Threshold and by Padding Model
(a) on BAST, (b) on FAST
跟 X 軸的平行線代表根據 Padding Model 做 Padding 所得到的數據

圖 21 我們以二維圖的方式來呈現，X 軸爲不同的 Padding 設定，Y 軸爲 Elapsed Time 愈低代表效能愈好。圖中可以看到 FAB 的最佳 Fixed Padding 設定跟不做 Padding 相比兩者的效能差距非常有限，這是因爲 FAB 大部份的 victim Group 所含的資料量很少，參考圖 16(a)在 32MB Buffer 情況下大部份的 flush Group 只含 17 個 Page 左右的資料量，所以即使使用 Padding Model，FAB 的整體效能也不會改善很多。

而對 HitStat 以及 BPLRU 來說，使用 Padding 能夠讓效能改善的幅度較大，但是過度的 Padding 設定反而會拖累效能，因爲對含資料量小的 victim Group 做 Padding 代表需要做大量額外的 Page 讀寫，這是造成圖中的 Padding 設定如果接近 0 時 Elapsed Time 會迅速升高的原因，其中 FAB、BPLRU 由於這類 victim Group 比例較多所以這種情況更明顯。因此如何達到最佳的 Padding 設定是一個重要的問題，HitStat 使用 Padding Model 在 BAST（表 5）和 FAST（表 6）中都能夠動態的趨近到最佳 Padding 設定，而在圖 21 可以看到 BPLRU 也可以使用 Padding Model 來做 dynamic Padding，跟各種 Fixed Padding 設定所能達到的最低 Elapsed Time 幾乎一致，表示 Padding Model 也適用於其它同樣以 Group 方式來管理的 Buffer 演算法。因爲 Padding Model 能夠在不同的 Buffer size 和 Log Block 數量環境下動態決定 Padding 條件來達到最佳的效能，這也是在圖 19, 20 中大部份情況下，HitStat 能夠得到最低的 Elapsed Time 而大幅勝出 FAB、BPLRU 的原因之一。

六、Conclusion

隨著 Multi Thread 以及多方面的軟體應用，Host 端跑的應用程式愈來愈多樣，造成寫到硬碟的 workload 也愈來愈複雜。SSD 把資料儲存在 NAND Flash Memory 中，而由於採用 outplace-update 所以在寫入資料時可能會因爲閒置空間不夠而觸發 GC，在寫入愈 random 的資料時回收 invalid 資料代價愈高，導致愈慢的寫入效能，這是 SSD 在一般應用上的效能瓶頸。

用 write Buffer 或較複雜的 NFTL mapping 方式都可以試圖改善寫入的效能瓶頸，但重新設計複雜的 NFTL 演算法極有可能產生出其它方面的問題。而使用 write Buffer 也需要注意因爲突然斷電造成資料遺失的問題，在一般桌上型或者伺服器電腦的應用上可能會導致整個 File System 損壞。而目前有兩個方法可以解決這個問

題，第一個方法爲使用電容延緩 SSD 的關閉時間，這段時間需要足夠把 Buffer 內的資料寫到 NAND Flash Memory 的固定位置上；而另一個方法爲使用 Non-Volatile RAM（NVRAM）來避免 write Buffer 中的資料遺失，在最近發表的 4-state MLC（Multi-Level Cell）PRAM[18]以及[9]中的介紹，我們可以期待使用 NVRAM write Buffer 的 SSD 產品問市。

既有應用於 Flash Storage 上的 Buffer 管理演算法，FAB 的 replacement policy 著重空間區域性，只適用於 Log Block 主要能吸收時間區域性的 NFTL 如 BAST；而 BPLRU 的 replacement policy 著重時間區域性，只適用於 Log Block 主要能吸收空間區域性的 NFTL 如 FAST。即使有 Log Block 的幫助，但它們由於過份注重單一特性，會造成某些資料霸佔住 Buffer 空間而排擠到另一種特性的資料留在 Buffer 中，使 Buffer 能發揮的效能有限。

此篇論文提出的 Buffer 管理演算法 Hit Statistic（HitStat），使用隨 workload 動態調整的 Cost-Benefit Analysis，同時考量時間區域性和空間區域性來最大化 Group Hit Ratio，能夠把許多小的 random write request 轉變成大的 sequential write 再寫到 NFTL，改善 SSD 主要因爲 random write 導致的效能下降問題，也解決了既有方法的問題以及侷限性，能適用於各種不同的 NFTL。實驗結果在 32MB 的 Buffer size、512KB 的 Block size 下，HitStat 的 flushed Groups 數量爲 FAB 的 58.1%、BPLRU 的 71.3%。適當的 Padding 可以把部份資料做 Switch Merge 而不佔住 Log Block 的空間，能夠有效減少 Full Merge 的代價來改善 SSD 的整體效能，但過度的 Padding 由於沒有充分利用到 Log Block 的空間反而使效能更差。所以除了 Buffer replacement policy 外，我們也提出 Padding Model 可以在不同 Buffer size、不同 Log Block 數量環境下動態決定做 Padding 的條件，能夠趨近於用 off-line 方式得到的最佳設定使效能達到最高，Padding Model 也能夠應用到其它的 Buffer 管理演算法。實驗結果在 32MB 的 Buffer size、4K512KB 的 Page Block size、NFTL 爲 BAST 使用 2048 個 Log Block 的設定下，使用 Padding Model 動態決定 Padding 條件的 HitStat 其寫入效能爲 FAB 的 113.9%、BPLRU 的 128.0%；而在 NFTL 爲 FAST 使用 128 個 Log Block 的設定下，HitStat(adj)的寫入效能爲 FAB 的 133.2%、BPLRU 的 138.6%，除了比既有的 Buffer 管理演算法更大幅度改善 SSD 的寫入效能外，也能夠適用於各種的 NFTL 架構。

參 考 文 獻

1. Aayush, G., K. Youngjae, and U. Bhuvan, *DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings*, in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. 2009, ACM: Washington, DC, USA.
2. Heeseung, J., et al., *FAB: flash-aware buffer management policy for portable media players.* Consumer Electronics, IEEE Transactions on, 2006. **52**(2): p. 485-493.
3. Hyojun, K. and A. Seongjun, *BPLRU: a buffer management scheme for improving random writes in flash storage*, in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. 2008, USENIX Association: San Jose, California.
4. Hyojun, K., et al., *A Page Padding Method for Fragmented Flash Storage* Lecture Notes in Computer Science, 2007. **4705**: p. 164-177.
5. Jesung, K., et al., *A space-efficient flash translation layer for CompactFlash systems.* Consumer Electronics, IEEE Transactions on, 2002. **48**(2): p. 366-375.
6. Jeong-Uk, K., et al., *A superblock-based flash translation layer for NAND flash memory*, in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*. 2006, ACM: Seoul, Korea.
7. Sang-Won, L., et al., *A log buffer-based flash translation layer using fully-associative sector translation.* ACM Trans. Embed. Comput. Syst., 2007. **6**(3): p. 18.
8. Sungjin, L., et al., *LAST: locality-aware sector translation for NAND flash memory-based storage systems.* SIGOPS Oper. Syst. Rev., 2008. **42**(6): p. 36-42.
9. Sooyong, K., et al., *Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices.* IEEE Trans. Comput., 2009. **58**(6): p. 744-758.
10. Yong-Goo, L., et al., *μ-FTL: a memory-efficient flash translation layer supporting multiple mapping granularities*, in *Proceedings of the 8th ACM international conference on Embedded software*. 2008, ACM: Atlanta, GA, USA.
11. Song, J., et al., *DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality*, in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*. 2005, USENIX Association: San Francisco, CA.
12. Seon-yeong, P., et al., *CFLRU: a replacement algorithm for flash memory*, in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. 2006, ACM: Seoul, Korea.
13. Michael, R.G. and S.J. David, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979: W. H. Freeman & Co. 338.
14. *K9F1208U0C 64M x 8 Bits NAND Flash Memory Data Sheet.* Samsung Electronics Company, 2006.
15. *K9NBG08U5M 1G x 8 Bit / 2G x 8 Bit / 4G x 8 Bit NAND Flash Memory Data Sheet.* Samsung Electronics Company, 2005.
16. *K9GAG08U0M 1G x 8 Bit / 2G x 8 Bit NAND Flash Memory Data Sheet (Preliminary).* Samsung Electronics Company, 2006.
17. Russinovich, M., *DiskMon for Windows v2.01.* 2006.
18. Bedeschi, F., et al. *A Multi-Level-Cell Bipolar-Selected Phase-Change Memory*. in

Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International. 2008.