國立交通大學

資訊科學與工程研究所

碩 士 論 文

團樹: 一種 NAND 快閃記憶體上的 B-Tree 原生實作方式

Blob Trees : A Native B-Tree Implementation over NAND flash

研 究 生：許蕙茹

指導教授：張立平　教授

中 華 民 國　九 十 七 年 十二 月

團樹：一種 NAND 快閃記憶體上的 B-Tree 原生實作方式
Blob Trees : A Native B-Tree Implementation over NAND flash

研 究 生：許蕙茹　　Student：Hui-Ju Hsu

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 Science

December 2008

Hsinchu, Taiwan, Republic of China

中華民國九十七年十二月

摘要

快閃記憶體廣泛應用在各項嵌入式系統，並且容量擴增快速，需要更有效的資料管理方式。隨著快閃記憶體容量增大，若使用邏輯位址的管理方式必須耗費更多掃描時間及儲存 translation table 的 main memory，因此提出使用實體位址來避免這些問題，而 B-tree 是目前管理大型資料最常用的索引結構之一，因此本篇論文提出 blob 的概念將 B-tree 索引結構結合快閃記憶體並且採用實體位址的實作方法。Blob 的設計為利用 B-tree 索引結構的 locality 存取模式來聚集 blob 的更新部份，達到減少快閃記憶體的 page-read、page-write 及 erasure 次數的目的。Blob 包含的 node 為 B-tree 之 subtree，也就是 B-tree 索引結構會由數個 blob 所構成。實作 blob 方法包含 blob 的 split/merge 及修正後的 garbage collection。效能評估方面以 B-tree on NFTL 與 blob 的實作方式比較，有 micro-benchmark、macro-benchmark 兩項主要的實驗部分，並以 page-read、page-write 及 block erase 次數分析之。

關鍵字:NAND 快閃記憶體、B-tree 索引結構

誌　　謝

即將結束研究所生涯，開心之餘也回想起這段時間以來的種種過程，有挫折也有困惑，但是最後終能度過，我想這要感謝許多人對我的照顧及幫忙。

首先要感謝的就是我的指導教授張立平老師，猶記剛進入實驗室時，對於研究的領域很陌生，在老師細心指導與帶領下慢慢的學習，讓我能夠逐漸熟悉並且有這篇論文的產生。還有感謝口試委員謝仁偉、陳雅淑教授，口試期間給我的意見有很大的幫助。

研究所生活中最密切的就是實驗室的各位，感謝千庭、辰暉、松德、家明同學們以及士庭、明毅、秀芬、郡杰…等等學弟妹們的幫忙，還有你們總是會讓實驗室充滿歡笑，我會懷念實驗室的快樂時光。另外感謝我其他朋友們，在我不順利時陪我度過，有你們的鼓勵跟支持對我真的很重要。

最後要感謝的就是我的家人:爸爸、媽媽及妹妹。感謝你們一直以來的支持與包容。

目　　錄

表　　目　　錄

圖　目　錄

一、Introduction

快閃記憶體具有抗震、省電及非揮發性的特質，因此被廣泛且大量的使用於各種嵌入式系統的裝置。但是由於快閃記憶體的物理特性不可直接覆寫(overwrite)，必須將更新的資料寫至尚未被寫入資料的部份，這樣的方式稱為 outplace-update。由於上述的物體特性，以至於在快閃記憶體上的儲存方式不同於其他儲存體(如:傳統磁盤)。隨著快閃記憶體的容量越來越大，運用的範圍也越來越廣，當快閃記憶體用於儲存大量資料時，原本所使用的儲存方式就較沒有效率。B-tree 是一般組織大量資料最常使用的索引結構，並且以 block device 的方式，傳送資料都是以固定大小長度為單位。快閃記憶體目前亦常用於大量資料的儲存，並且 NAND 快閃記憶體為以 page 為取向做讀取，此種方式類似於 B-tree 的 block device，因此將快閃記憶體以 B-tree 索引結構來組織大量資料是一個可行的方法。

快閃記憶體結合 B-tree 的方式有兩個問題；一個為 B-tree operation 通常是很小的資料更新，另一個為快閃記憶體 write-once 的特性導致必須使用 outplace-update 的更新方式。

針對第一個問題，管理 B-tree 索引結構的 operation 會引起很多索引上的指標更新，即使這些需要更新的資料遠小於 B-tree 一次存取的最小單位，但是仍然必須更新整個 block。這樣的作法在磁碟上成本不高，因為更新 byte 與更新一整個 block 成本接近相等，但是在快閃記憶體上重複寫入未更新過的資料，會浪費快閃記憶體的抹除次數(erasure cycle)，因為快閃記憶體為有限次數的抹除，超過限制的次數會造成快閃記憶體的資料不穩定，而快閃記憶體的寫入也會導致抹除的成本增加及整個系統的存取速度變慢。

針對第二個問題，由於快閃記憶體只能一次寫入(write-once)，當資料更新時無法直接覆寫，必須採用 outplace-update 的方式，若是採用實體位址的方式，因為無法直接在原來的指標上做更新，因此實作上會較困難。採用實體指標還可能發生 pointer-update propagation 的問題。當一份資料的儲存位址改變後，必須更新新的位址到 reference 這份資料的 block，並且以 outplace-update 的方式更新，因此若 reference 到這份資料的 block 空間不足時，必須寫到新的 block，也就是又改變了一份資料的儲存位址，此時需再往上追蹤至 reference 到第二份資料的 block，再以 out-place update 的方式更新新的位址，這樣的情況可能會發生數次，就會引起一連串的快閃記憶體位址更新，這就是 pointer-update propagation 的問題。這個問題，在以往的研究中，通常是以邏輯指標來解決[4,5,7,17]。由於邏輯位址的更新只需要更改

translation table，修改邏輯位址對應的快閃記憶體位址即可。然而使用邏輯位址的缺點是儲存 translation table 造成 RAM 的使用率增加，以及開機時必須讀取完整個記憶體才能建出 translation table 導致開機時間變長，並且隨著目前快閃記憶體的容量越來越大，建出 translation table 所耗費的讀取成本及時間也越多。

本篇論文提出 blob 的概念來管理快閃記憶體上大量資料，其主要概念有兩個；第一個為利用 B-tree 的 locality 特性，對寫入做最佳化。B-tree 的存取通常具有 locality 的特性，需要修改的 node 通常是鄰近並且相連的關係，因此將這些鄰近 node 的更新部份聚集後，就可以用較少量的 pages 完成寫入更新的部份。Blob 就是這些可共同使用 page 的 B-tree nodes 的集合，以這樣的寫入最佳化來減少 page-write。第二個為嘗試解決 pointer-update propagation 的問題。採用實體位址無法避免 pointer-update propagation 的問題，但是 blob 可以減少 pointer-update propagation 的發生機率及影響程度。由於 blob 預留空間來儲存 blob 的更新內容，可將更新後的參考指標以 log 的方式存於 blob 中，而不需將 blob 全部重複寫入來更新指標，因此能吸收指標更新。在多層的參考指標關係下，每一層的 blob 均能吸收多次指標更新，因此並不容易發生 pointer-update propagation。

實驗部份將以程式模擬並實作本文提出之 blob 整體系統，及與 B-tree on NFTL 的模擬比較兩者效能。Blob trees 之實驗為自 B-tree operation 至快閃記憶體的完整系統模擬，以及 blob trees 的相關 operation。由於 blob 主要目的為減少快閃記憶體的 read/write operations，因此實驗主要需量測目標為 page-read 及 page-write 次數，而 block erase 次數對於快閃記憶體的使用是很重要的依據，因此也列入量測目標。將與 blob trees 比較的是 B-tree on NFTL 的方法，這是以 B-tree node 對應 page 且架構在 NFTL 之上的基本 B-tree 實作方式，並且去除 NFTL 實作上產生的 page-read/page-write 的 overhead 部分，比較結果會更確實及公平。我們的實驗結果顯示在 micro-benchmark 及 macro-benchmark 下，blob 除了發揮本身可聚集更新內容及降低 pointer-update propagation 發生機率的功能外，更加善用 locality 的特質來大幅減少 page-write 需求，並且在 page-read 和 block erase 次數統計上亦少於 B-tree on NFTL。

本論文的結構如下：第二章為簡述相關文獻及描述問題，第三章為說明 blob 的主要架構及方法的論述，第四章將會使用模擬的方式來實驗，實驗部份包含 micro-benchmark、macro-benchmark 及 cache size 的效能評估，並與 B-tree on NFTL 來比較。

二、Motivation

由於快閃記憶體廣泛的使用，此章節先針對快閃記憶體的特性做大略的介紹，並描述這篇論文所提出將 B tree 索引結構用於 NAND flash 所產生的問題。

2.1 Flash memory characteristics

市面上的快閃記憶體分為 NOR flash 和 NAND flash 兩種。NOR flash 速度快並且支援 XIP(eXeute In Place)，但是成本高；而 NAND flash 成本低，因此廣泛使用且容量擴增快速，本篇論文則是以使用 NAND flash 為研究基礎。NAND flash 具有以下物理特性：單次寫入(write-once)、大塊抹除(bulk-erasing)及有限的抹除次數(erase)。如 Figure 1.所示，快閃記憶體會分成多個區塊(block)，而一個區塊會再區分成多個 page。NAND 快閃記憶體只能提供一次讀取/寫入單位為一個 page，而一次抹除(erase)單位為一個區塊(block)的管理方式，並且已被寫過的 page 必須經過 erase 後才可再次寫入，也就是每次最少必須讀取或寫入一個 page，而執行 erase 動作時必須將一個 block 內的所有 page 全部都清除資料。然而快閃記憶體的 block erase 次數限制約為 1000000 次，若快閃記憶體的 block erase 次數超過限制，快閃記憶體會出現不穩定的狀態而可能導致資料遺失[16]。

快閃記憶體主要的三個動作：page read、page write 和 block erase。由 Table 1.可看出 block erase 最耗費時間，page write 次之，而 page read 最少。由於快閃記憶體僅有有限的 erase 次數，除了降低快閃記憶體 page-write 及 block erase 次數，還必須考慮到每個 block 的 erase 的程度，因此要儘可能平均每個 block 的磨損狀態來延長快閃記憶體的整體使用時間，這就是 wear-leveling 的主要目的[15]。

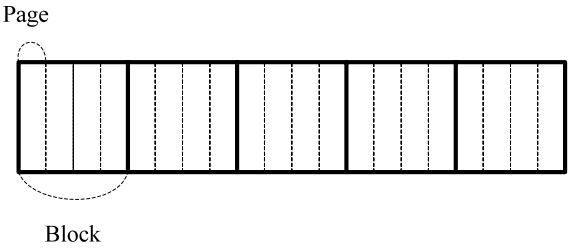


Figure 1. The structure of flash memory

	SLC NAND	MLC NAND
Page size	2KB	4KB
Block size	128KB	512KB
Read latency	77.8μs	165.6μs
Write latency	252.8μs	905.8μs
Erase latency	1500μs	1500μs

Table 1. The characteristics of SLC[2] and MLC[3] NAND flash memory

根據快閃記憶體的物理特性，當有資料更新時，由於快閃記憶體只可write-once，因此不能直接將舊資料覆寫(overwrite)，必須將新的資料寫到另一個尚未被寫入資料的 page，這樣的更新方式稱為 outplace-update。由 Figure 2.可看出，即使更新的部份小於一個 page 的大小，但是仍需重寫一整個 page，並將其餘未更新的資料一併重複寫入新的 page，浪費寫入的空間及執行時間。

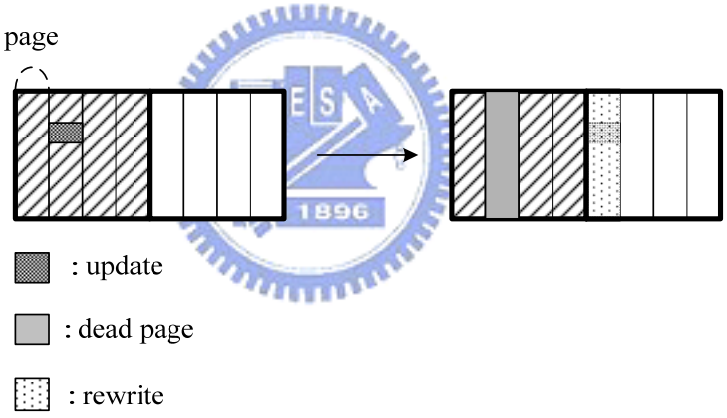


Figure 2. Outplace-update

由於快閃記憶體 outplace-update 的方式，經過多次更新後，dead page 會散佈在各個 block，當系統可用空間不足時就必須要回收 dead page 來釋放空間。然而快閃記憶體 bulk-erase 的特性，一次必須 erase 一整個 block，所以必須將準備 erase 的 block 內的有效 page 重複寫至另一個 block，再將此 block 做 erase 的動作，這就是 garbage collection[12]。

2.2 Related work

由於快閃記憶體的物理特性無法當作一般磁碟使用，因此若想將磁碟的存取方式套用在快閃記憶體上，就必須經過一層軟體來驅動並管理快閃記憶體，Flash Translation Layer(FTL)就是負責處理應用層與快閃記憶體之間的連

結與轉換[11]。目前已有多種檔案系統，如:YAFFS[19]及 JFFS[14]，也有多種實作方法的研究[20]。NAND Flash Translation Layer(NFTL)[6]的概念類似於 FTL[8]，但是只用於 NAND 快閃記憶體。NFTL 主要架構由 Virtual Unit Chain 組成，每個 chain 對應一個 erase unit，chain 實際上則由許多快閃記憶體的 block 連結而成，當 block 第 i 個 page 需更新而 chain 最後一個 block 第 i 個 page 已被寫入時，則必須在 chain 的最後再加入一個 free block，並在此 block 第 i 個 page 寫入更新資料。執行 garbage collection 的動作時，則會選擇長度最長的 chain 來作為 erase 的對象，這是因為長度越長的 cahin 能回收越多的 block 數。Chain 的連結必須由兩個分別為 logical to physical(EUNtable)及 physical to physical(ReplUnitTable)的 table 來達成，logical to physical 的 table 用來將邏輯位址轉換為實體位址，而 physical to physical 的 table 則做為連結 chain 的 block，entry 內容為此 chain 的下一個 block，因此可能有多個 physical to physical 的 table，並且 table 個數取決於 chain 的長度。然而當 chain 的長度增加，快閃記憶體的讀寫動作也會增加讀取 chain 中 block 的 page 數的 overhead，因此在 NFTL 的實作中系統的 page-read 次數會變的非常多。

目前已有數篇論文提出不同的方法來改善 B-tree 索引結構實作在 NAND 快閃記憶體上的效能，並以降低 page-write 次數為首要目標，像是 BFTL、μ-tree 及 flashDB。以 C. H. Wu、L. P. Chang 及 T. W. Kuo 所提出的 BFTL 為例[4,17]，BFTL 的架構由 reservation buffer 及 node translation table 組成。屬於不同 node 的 index unit 可寫在同一個 page，再由 node translation table 以 list 方式記錄所有儲存此 node 的 index unit 之 secctor number。充分使用 page 寫入的空間，因此可以減少 page-write 次數。然而當 node translation table 之 list 長度太長時，必須做 compact 的動作，也就是將此 node 的 index unit 重新寫入新的 sector，達到縮短 list 長度並降低 B-tree 的 search time，但是 compact 的 operation 亦會造成 page-write 次數的增加，因此 compact list 的 threshold 設定存在一個 trade-off 在 page-write 次數及 search time 之間。

JFFS3的設計使用B+-tree來管理檔案，但是由於缺乏FTL及快閃記憶體outplace-update的因素，當leaf node更新時就必須將leaf node到root node一整個path上的所有node都進行更新的動作，以這種更新方式的tree則稱為wandering tree[1,14]。D. Kang、D. Jung、J. U. Kang及J. S. Kim提出了μ-tree[5]。μ-tree是一種用於NAND快閃記憶體類似B+-Tree的ordered index structure。在NAND快閃記憶體上實作B+-tree最自然的方式就是以一個page儲存一個node的key值及內容，但是以wondering tree的更新方式會造成過多的page-write。μ-tree的概念是將每個page的layout依照tree的高度分配固定比

例的page空間，並且只有具有相連關係的node才可存放在同一個page。當leaf node需要更新時，可將leaf node到root node的path上所有node都更新並寫在同一個page，減少page-write次數。由於一個page必須要能容納從leaf node至root node之path的node資訊，因此tree的高度及node的大小均會受限於page的空間大小，並且除了最近被更新的leaf node之path的node會儲存在同一個page，其餘node幾乎都是各自分散在不同的page，因此μ-tree的page-read次數與B+-tree差異不大。

FlashDB 是由 S. Nath 及 A. Kansal 所提出[7]，為用於 NAND 快閃記憶體的一種動態自我調整的 B+-tree 資料庫系統。FlashDB 具有兩個 mode；一個為 Log mode，另一個為 Disk mode。顧名思義，在 Log mode 的狀態下，b+-tree node 的資訊必須轉換為 log entry 的集合，並且更新時只紀錄更新部份的那些 log entry；而 Disk mode 則是一次儲存完整的一個 node 資訊，更新時必須整個 node 全部重寫。然而兩個 mode 之間轉換的動作必須付出 cost，若從 Log mode 轉換到 Disk mode，必須將 node 的所有 log entry 讀出並且改以 Disk mode 的 node 方式儲存；相反地若從 Disk mode 轉換到 Log mode，node 資訊也必須改以 log entry 的方式儲存。系統執行過程中會考量目前 mode 的處理效能加上轉換 mode 的 cost，再判斷是否要轉換 mode。若轉換 mode 過於頻繁，這部份的 cost 會更大，可能導致系統的效能不如預期。

2.3 Problem definition

以 B-tree 的架構實作在 NAND 快閃記憶體上來管理大量資料會產生以下兩個問題；第一個問題為 page-oriented writes 的特性造成過多的 page write。B-tree 索引結構為了達到平衡，一個資料的新增或刪除都有可能引發多次的 B-tree operation，由於快閃記憶體只能以 outplace-update 方式做更新，B-tree 索引結構為 block-device，最小的存取單位為一個 B-tree node，相同的特性在快閃記憶體上則是以一個 page 為最小的存取的單位，因此若以最自然的方式實作 B-tree，也就是將一個 node 對應一個 page，則每一次的更新都必須至少要寫入一個 page。然而 B-tree 更新寫入的資料通常遠小於一個 page 的大小，更新時必須重複寫入許多尚未被修改的資料，如此會造成浪費快閃記憶體空間及導致龐大的寫入需求。

第二個問題為邏輯位址造成的 overheads。由於 B-tree 索引結構是由 B-tree node 連結所組成，因此指標在 B-tree 的實作上是必須的。若以邏輯位址來實作，系統一開啟時就必須要讀出快閃記憶體的所有資料來建立 logical to physical 的 translation table，才能建立 B-tree 結構，耗費了太多快閃記憶

體讀取的時間，以及儲存 translation table 所需的記憶體空間。隨著快閃記憶體容量越來越大，初始化掃描快閃記憶體的時間及所需的 RAM 空間也會越大，因此使用邏輯位址在大容量的快閃記憶體上，勢必會造成更大的 overheead 而大幅降低整體效能。另一種方法為使用實體位址，但是由於 outplace-update 的方式，導致每次的資料更新可能會引起一連串指標的更新，因此 pointer-update propagation 會是採用實體位址的最大問題。

本篇論文以高階系統為基準來設計將 B-tree 實作在 NAND 快閃記憶體上的方法，也就是系統具有相對大的快閃記憶體及 RAM 空間。設計實作方法的目標有以下幾點: （一）快速啟動：這個部份將以採取實體位址的方式來解決，即可去除邏輯位址在 initialization 及 RAM-space 上的 overheads，並且必須降低 pointer-update propagation 的發生。（二）有效地處理細小的存取動作：依據 B-tree 存取的 locality ，將對鄰近 B-tree 節點的更新動作打包在相同的頁中，藉以減少快閃記憶體讀寫的動作次數； （三）高效能的快取設計：因為 B-tree 應用的場合是外部儲存媒體，因此搭配有效的快取機制與策略，將能大幅提昇資料存取的速度。（四）有效的快閃記憶體空間管理，如 garbage collection 策略。

三、Design implementation of blob

3.1 Blob Trees: Overview

本篇論文提出 blob 的概念並利用 B-tree index structure 具有 locality 特性及實體位址的實作方式來提升整體效能。Blob 主要想法為將具有 locality 的相連 B-tree nodes 收集在同一個 blob，即可將同一個 blob 內的更新部份壓縮儲存在少量的 page 中，以減少快閃記憶體的 page-write 次數。雖然 Blob 採用實體位址的方式，但 blob 更新時可附加更新內容於對應的 block 剩餘空間，因此不需每次更新位址，可大幅降低發生 pointer-update propagation 的次數。

Blob、B-tree 與快閃記憶體的關係如 Figure 3(a).所示，B-tree 結構會改變成由 blob 所組成，而 blob 則為 B-tree 之 subtree，並且以 locality 作為 blob 切割策略的依據。換言之，B-tree structure 以參考 locality 將整個 B-tree 分裂為多個 subtree，而每一個 subtree 即為一個 blob，並且 blob 與 blob 之間仍具有互相連結的關係，將 blob 連結組成後就是完整的 B-tree structure。Blob 與快閃記憶體的關係為一個 blob 對應一個快閃記憶體的 block，此 block 僅可儲存對應 blob 之內容及更新部份，不可儲存其他 blob 的內容，也就是每

一個快閃記憶體上有效的 block，都配置給單一個 blob 儲存使用。

Blob 的內容會轉換為數個 index units 來表示，並且儲存於 page 中，如 Figure 3(b).所示，page 內容均由 index units 所組成，由 index units 可建立出 blob 的完整 subtree 結構。從快閃記憶體來看，每一個有效的 block 內的內容可建立相對應 blob 的 B-tree node 資訊及結構，將所有 blob 建立完成並連結後，即是 B-tree 的原始結構。

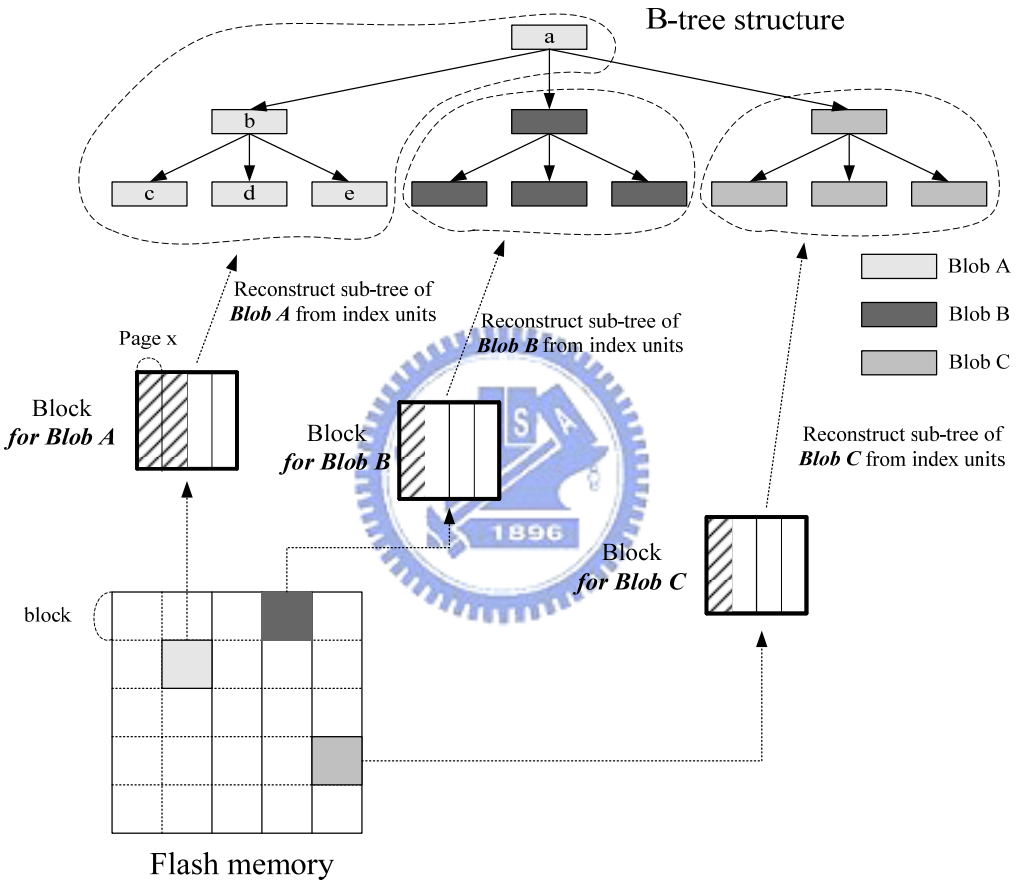


Figure 3(a). B-tree of blobs over NAND flash.

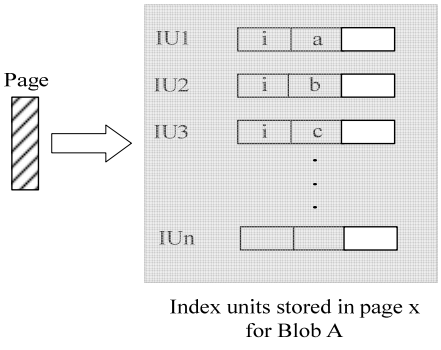


Figure 3(b). Index unit

實作 blob 概念的完整結構，如 Figure 4.所示，包含 application、B-tree logical structure、blob manager、blob cache 及快閃記憶體，其中 blob manager 由 blob split、blob merge、GC merge 和 blob status table 組成。

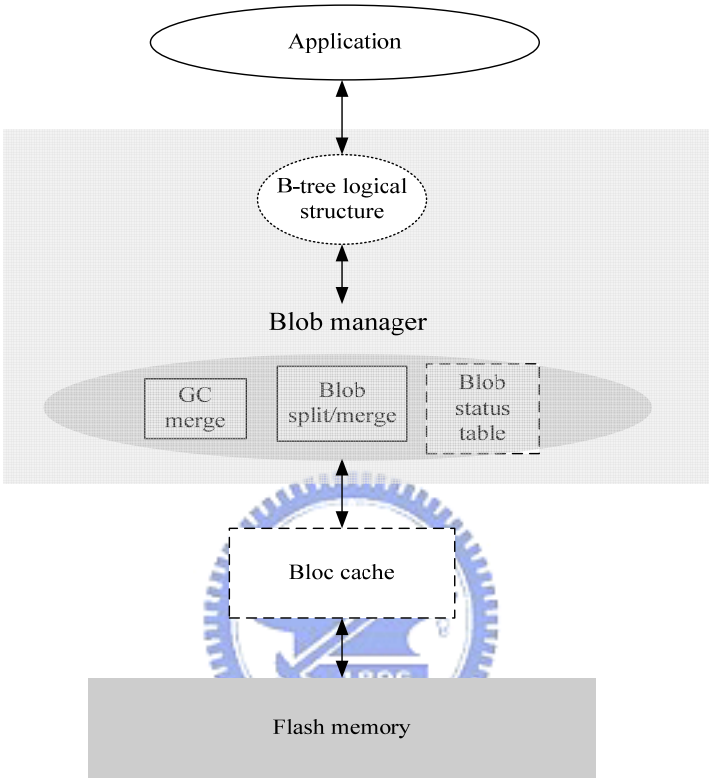


Figure 4. System architecture

B-tree logical structure 為目前處理的 B-tree operation 的結構部份，blob manager 為主要設計重點。由於 blob 的結構為 B-tree 之 subtree，因此 blob 大小會隨著 B-tree operation 而改變，然而 blob 的容量必須受限於 block size，並且 locality 亦會隨 B-tree operation 產生更動，必須要有可重新考量 locality 來重組 blob 結構的策略，因此提出 blob split/merge 的方式來達到上述目的，並在 3.4 中有詳細說明。當快閃記憶體空間不足時需要執行 garbage collection 來釋放空間，但是一般 garbage collection 的方式不適用於 blob trees，因此專為 blob trees 設計了 GC merge 來處理 garbage collection。為了能更快速的處理 blob manager 的相關 operation，整理架構加上了 blob status table 來輔助。Blob cache 負責儲存 blob 的內容及更新部份，並且 blob cache 內所儲存的內容形態與快閃記憶體內的 block 均相同。當 cache 空間不足時，則需要一些寫回策略，以及一些能夠將 cache 空間釋放出來的額外技巧。這邊會在 3.6 提到。

3.2 Blob concept

由於 B-tree index 更新的部份通常為很小的資料，遠小於快閃記憶體的 page 大小，但是快閃記憶體的最小寫入單位為 page，若是以 B-tree 以往的實作方式，一個 B-tree node 對應儲存一個 page，如 Figure 5(a).所示，當 B-tree node 有更新部份時，必須將更新後的 B-tree node 內容全部重新寫入一個 page，其中包含 B-tree node 中沒有修改舊有的部份，因此會有大量的資料是不需更新卻又必須重新寫入快閃記憶體，造成 page-write 次數增加也浪費了快閃記憶體的 erasure cycle。以 Figure 5(b).為例，一次的 B-tree operation 可能引發三個 B-tree node 的小量更新，但是 page-oriented 的實作方式，導致必須更新三個 page。

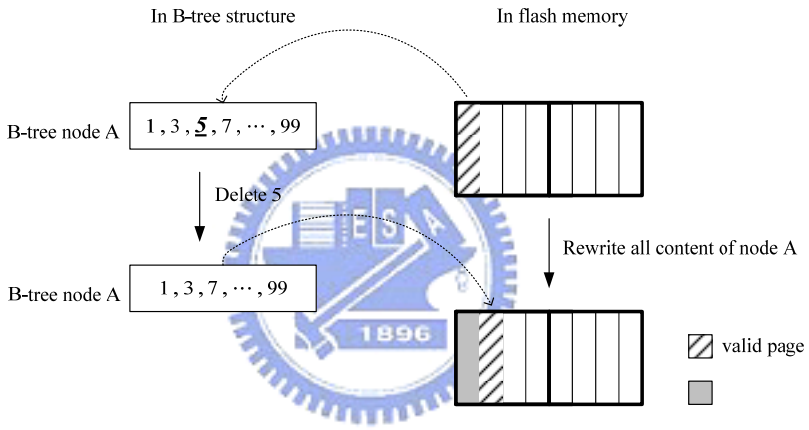


Figure 5(a). Page-oriented writes.

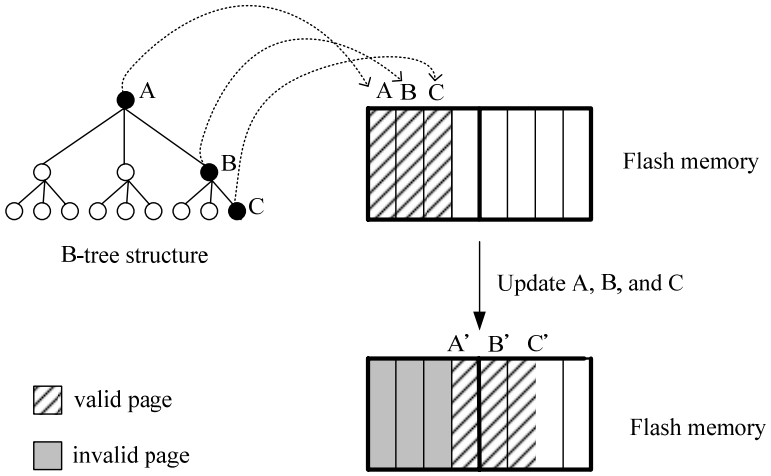


Figure 5(b). B-tree updates.

為了減少重覆寫入的 overhead 及過多的 page-write，參考 LFS[9,10]的方

法將更新的改變部分以 log 的方式聚集之後再一次寫入。利用這樣的概念就可以將更新的部份集中在一個 page 來減少 page-write 次數。以 Figure 6(a). 為例，若是經歷三次 update，則必須重寫三個 page，但若是能將 update 部分以 log 的方式聚集在同一個 page 如 Figure 6(b).，就能減少 update 必須重新寫入的 page 數，使用的 page 數減少亦能降低 block erase 的次數。B-tree 索引結構的存取通常具有 locality 的特性，因此 blob 的目的就是要善用 B-tree 存取的 locality 特性來更進一步減少 page-write 需求。由於每次的 B-tree operation 可能有多個 B-tree node 有更新的部份，若這些需要更新的 node 都屬於同一個 blob，則可以將這些 node 的 update 部分一起寫入在同一個 page，大幅減少 page-write 次數。

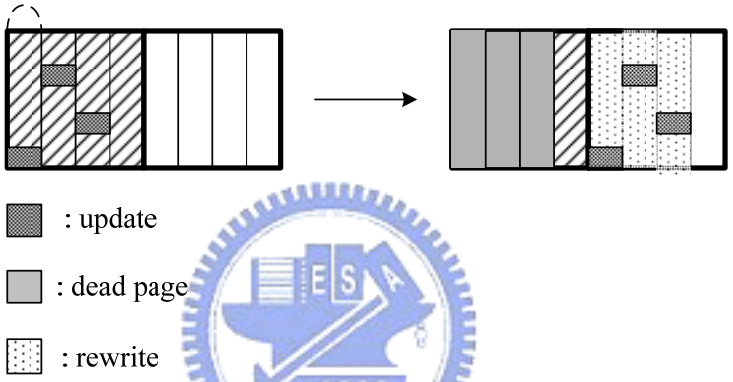


Figure 6(a). Update three items and write three pages.

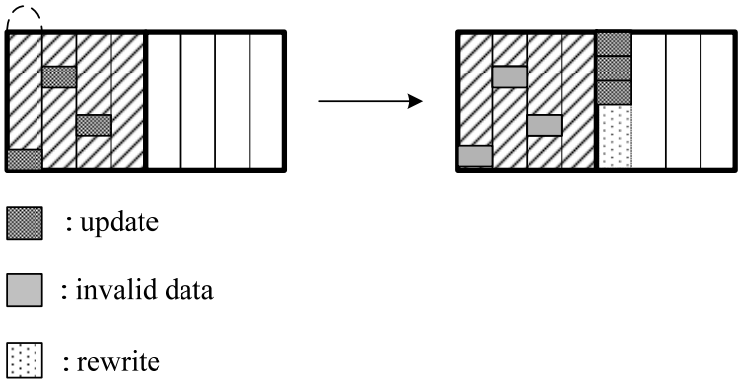


Figure 6(b). Update three items and write only one page because of gathering updated items.

Blob 的內容均由 index unit 所組成，index unit 則是由 B-tree structure 轉換而來，包含新增/刪除 key 值、新增/刪除 node 及 blob 對外的 address，其中新增 node 和 blob 對外的 address 會額外記錄 parent node 來輔助 B-tree structure 的重建動作。B-tree operation 執行過程中，依據 B-tree index structure

的更新部分轉換成 index units，這些 operation 所對應的 index units 會儲存於 cache 中所屬 blob 之 page 空間。如 Figure 7.所示，B-tree 處理更新過程中，必須使用 index unit 以 log 的方式繼續附加在 blob 原先的內容之後，因此當 blob 寫回快閃記憶體時，僅需將更新部份所對應的 index unit 寫回，再與 block 內原有資料結合即可建立完整的 B-tree structure。

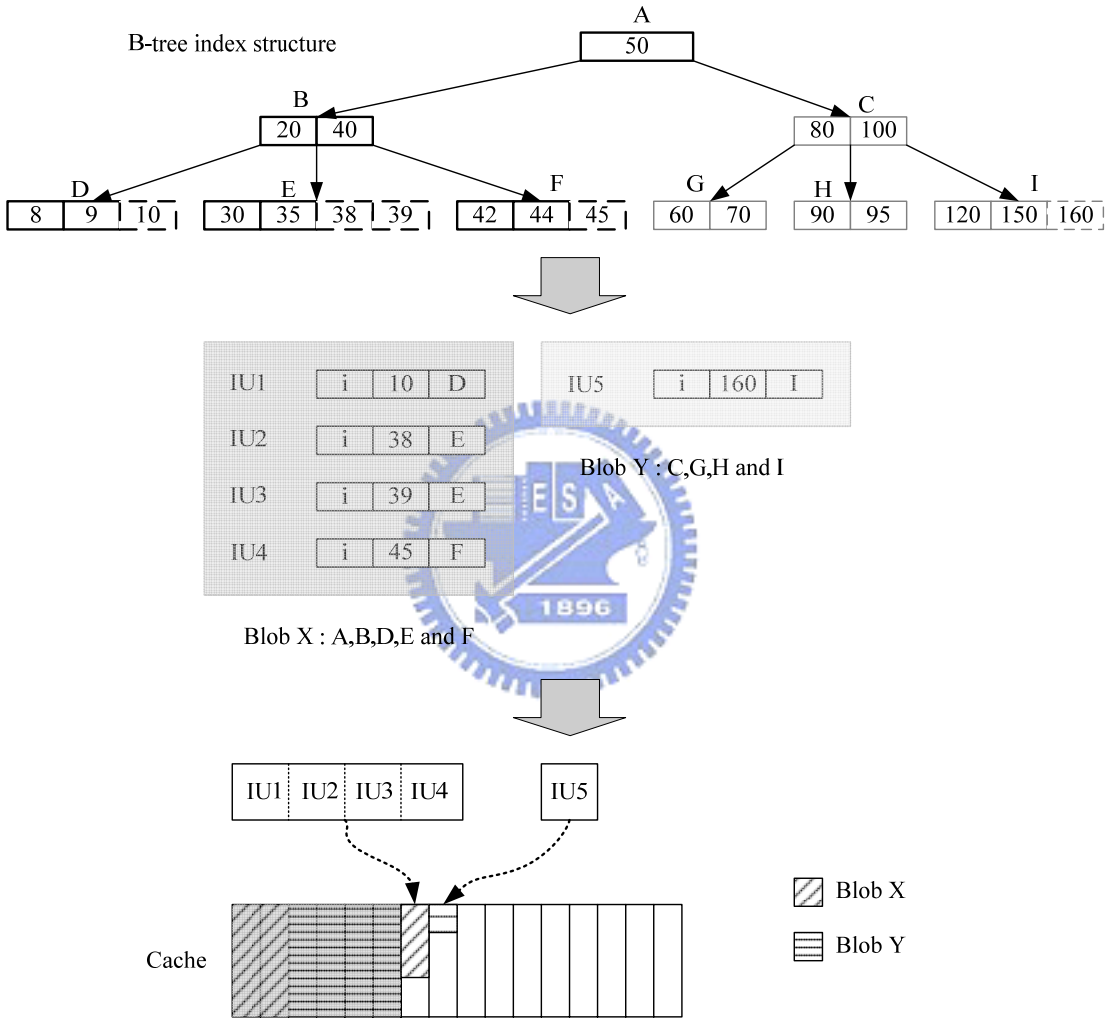


Figure 7. The process of update.

由於 B-tree structure 是以經過轉換後的 index units 來作為儲存於快閃記憶體中的主要內容，因此也必須具有能將 index units 還原成原來的 subtree 型態的轉換方式。當進行 B-tree operation 時，必須先讀取此 subtree 所對應的 blob 完整內容，並執行 B-tree structure 重建的動作。若系統具有 cache 機制並且已存在欲存取的 blob，則可直接從 cache 中讀取，cache 的相關說明可參考 3.6；若無 cache 機制則從快閃記憶體讀取之。重建 B-tree structure

的第一個步驟為建立此 blob 的 tree structure，依照每個新增 node 及 address 的 index unit 可建立出 parent 與 child 之間的 node 關係及相連 blob 的關係。第二個步驟為建立每個 node 的 key 值並排序。第三個步驟為藉由每個 node 的 key 值重排至適當的 child node 的位置，最後再將對外相連的 blob 插入正確的位置。若有多個相連 blob 位於同一階層則必須藉由相連 blob 內的最大 key 值來輔助排序，最大 key 值可以由讀取 blob 後得知，但為了減少讀取快閃記憶體及處理時間，blob status table 會紀錄每個 blob 目前包含 key 值中的最大值，因此重建 B-tree structure 時只需查詢 blob status table 即可而不需要再讀取整個 blob 內容。經過以上步驟即可將 blob 內的 index units 轉換建立為 subtree 的型態並可處理此 subtree 內的所有 B-tree operation。

3.3 Physical Pointers and Pointer-Update Propagation

目前較常使用的定址方式為 logical address，但是必須額外使用一個 translation table，將邏輯位址轉換為實體位址，轉換過程如 Figure 8(a).所示。每次存取記憶體時，都必須先至 translation table 找到此邏輯位址所對應的實體位址，再根據此實體位址存取快閃記憶體的內容。然而當快閃記憶體空間越來越大時，translation table 的大小是對應於實體位址的大小，因此 translation table 必須更大，而維護 translation table 所耗費的記憶體也就越多，並且在開機掃描整個快閃記憶體建立 translation table 時需要更長的時間，因此提出改用實體位址的方式來實作，其存取方式則如 Figure 8(b)。

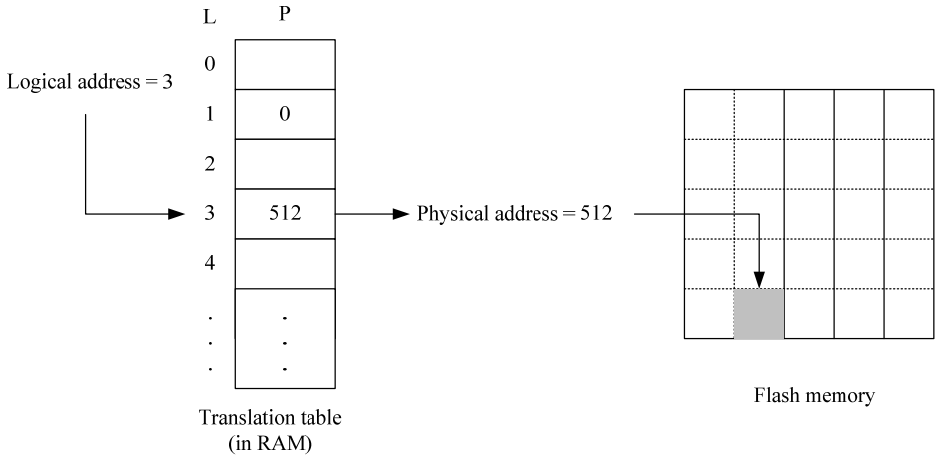
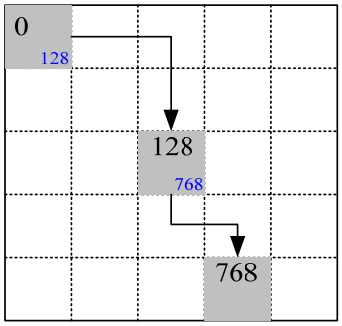


Figure 8(a). Logical address and translation table.



Flash memory

Figure 8(b). Physical address.

實體位址的好處是不需要建立及維護 translation table，可以減少 RAM 的使用空間及加速開機時間。當有位址更新時，參考到此位址的指標必須更新為新的位址，若是使用邏輯指標只需要更新 translation table。然而快閃記憶體的更新方式為 outplace-update，從 Figure 9(a).可看出一個位址的更新時，參考到此位址的 block 必須更新為新的位址，但是此 block 可能空間不足必須寫到另一個新的 block，此時又產生了一個位址的更新，又必須再次更新參考到這個 block 的指標，若是繼續更新並且 block 空間均不足，就會引發一連串的位址更新，這就是 pointer-update propagation 的問題。

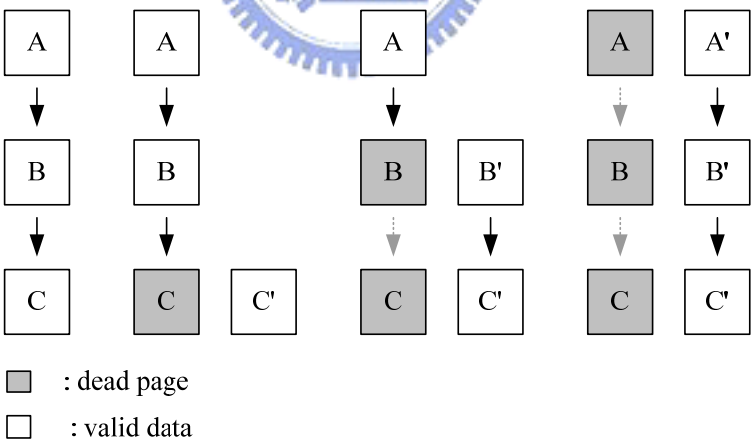


Figure 9(a).Pointer-update propagation

在 blob trees 的實作中，如 Figure 9(b).每個 blob 均對應一個 block，並且保留此 block 的剩餘空間做為儲存 blob 的更新內容，更新部份可直接寫入到此 block 的 free page，因此 blob 可允許不需以 outplace-update 方式更新數次，也就是不需更改此 blob 的實體位址，而參考此位址的指標也不需更新。若是以一個 node 對應一個 page 的方式，node 的更新必須以 outplace-update 方式來執行，也就是一個 node 的更新就會引起一個位址的更新，並且必須

一連串往上更新參考到的部份。以一個 block 具有 N 個 page 且參考位址的階層有 k 層為例，blob trees 需要經過 N 次更新之後才會引發一次的位址更新，因此要發生 k 層的 pointer-update propagation 需要經過 N^k 次的更新，而一個 node 對應一個 page 的方式只要每一次的更新都會引發 k 層的 pointer-update，blob trees 的方式將發生機率由 1/N 降到 1/N^k，大幅減少 pointer-update propagation 的次數。

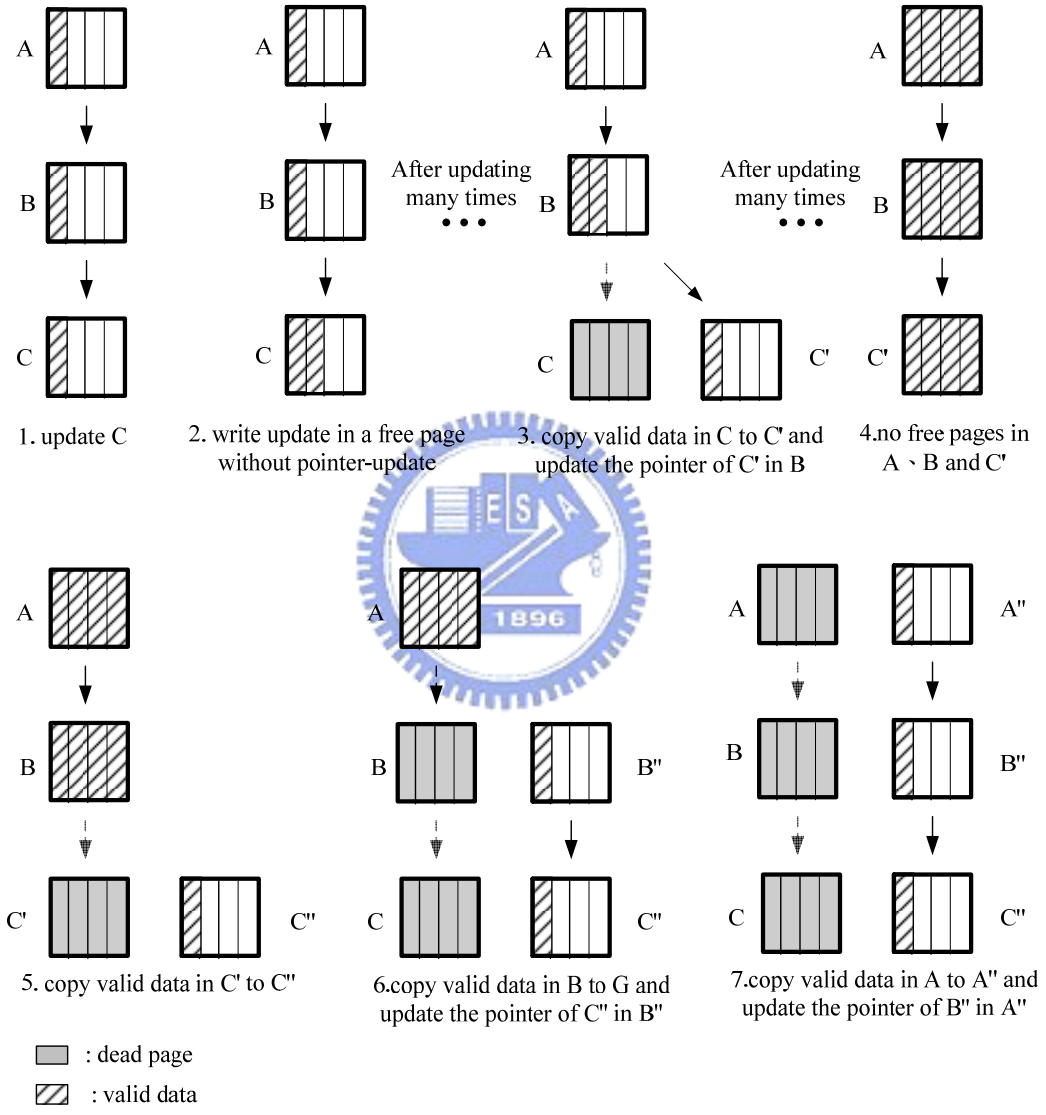


Figure 9(b). Pointer-update propagation in blob trees.

3.4 Blob split/merge policy

執行多次 B-tree operation 之後，blob 的大小可能會隨之改變，為了使 blob 能充分利用 locality 的特性並且提升聚集更新的效果，blob 需要重新組合，並以 blob split/merge 來改變 blob 結構。為了能更快速的處理 blob

split/merge，blob status table 可用來輔助選取 blob。Blob status table 會紀錄 blob 的相關資訊，像是 parent blob、node 數、access weight、最大 key 值及最後存取時間。這些資訊可以幫助系統更快速處理目前需要做 blob split/merge 的 blob 及將 blob 內容重建為 B-tree structure，若無 blob status table，這些資訊則必須藉由讀取多個 blob 的資料來判別，因此使用 blob status table 可以有效減少 search time 及 page-read 次數。

由於快閃記憶體的 block 大小限制，blob 的大小也必須受限，因此當 blob 超過設定的最大值時，blob 必須進行 split 動作來縮小 blob size。Blob split policy 以 node 的 hot/cold 屬性為依據，將原來的 blob 分裂成兩個屬性接近的 blob，因此 split 後會將 hot node 儘可能集合在同一個 blob。由於 node 的屬性可能會隨著資料存取的變動而改變，hot node 可能不再經常地被存取，而其他 node 亦可能漸漸時常被存取成為 hot node，導致原本在相同 blob 的 node 可能屬性已有差異，因此 split 對於 blob 的好處不只可以縮小 size，更可以藉由 split 重新調整 blob 的屬性一致，將目前的 hot node 集合在同一個 blob，可以更有效的聚集 update 部分，減少 page-write 次數。

Blob split operation 的目的是要分成兩個屬性接近的 blob，也就是分裂成一個為屬於 hot node 較多的 blob 與另一個 hot node 較少的 blob。依據 hot node list 判斷欲 split 的 blob 內哪些屬於 hot node，若為 hot node，則此 node 的 weight 值為 1，而每個 node 的 total weight 值是由 child node 之 weight 值與此 node 之 weight 的總和。如 Figure 10.，依照此法則由 leaf node 逐一往上計算至 blob root，從中選擇一條 link 具有最大的 weight 差值，此 link 作為分裂 blob 的界線，分成兩個 blob。根據上述的方式即可分裂出 hot node 數較多且相連的 blob。

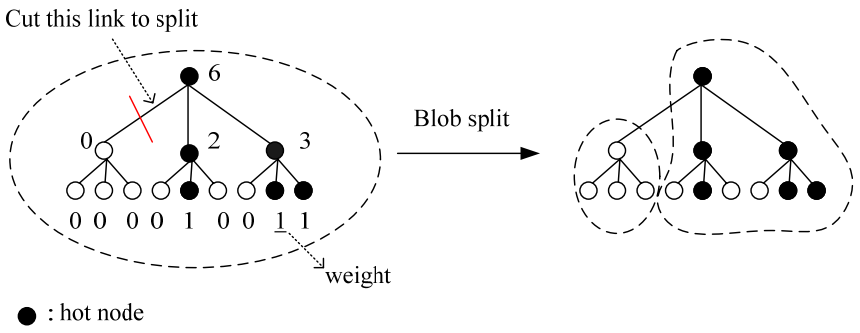


Figure 10. Blob split

Blob 的大小會隨著 B-tree insert operation 增大，也會隨著 B-tree delete operation 縮小，而資料存取的 locality 亦可能隨時間而改變。若能將屬性相

近且相連的 blob merge 為一個 blob，更可以發揮 blob 聚集 update 資料的優點。然而必須考慮到 merge 後新 blob 的大小，若 blob 太過於龐大接近 blob split operation 的門檻，則放棄 merge。

Blob merge 的目的是要將 write 頻繁且 blob size 較小的兩個 blob merge 為一個新的 blob，可以更有效的充分利用 locality 的特性，聚集 update 部分來減少 page-write 次數，並且減少 cache 佔有的空間以提高 cache utilization。實作上的方法是採取以 blob 的 access weight ratio 為是否執行 blob merge 的判斷準則，access weight ratio 是由 access weight/total access weight(目前存於 cache 中所有 blob 的 access weight 總和)得來的，而 access weight 為 blob 的存取次數，但是每隔一段時間會將所有 blob 的 access weight 全部減半，更能符合 locality 的特性。當 blob 的 access weight ratio 超過設定的門檻後，才會進一步繼續尋找是否可執行 blob merge 的對象。

如 Figure 11.所示，執行 blob merge 時，先清除 cache 中欲執行 merge 的兩個 blob 所儲存的資料，以及將此兩個 blob 所對應的兩個 block 標示為可回收的 block，而 merge 後的新 blob 將內容繼續存於 cache 中。這個部份會有記憶體空間成本的問題，由於每個 block 均已配置給一個 blob 使用，執行 blob merge 的兩個 blob 就會造成兩個 block 必須經過 erase 後才可再使用，可能造成 block erase 次數的增加，因此 blob merge 必須考慮到快閃記憶體的空間成本以及 erasure cycle，blob merge 太過頻繁對於系統的效能可能不升反降。

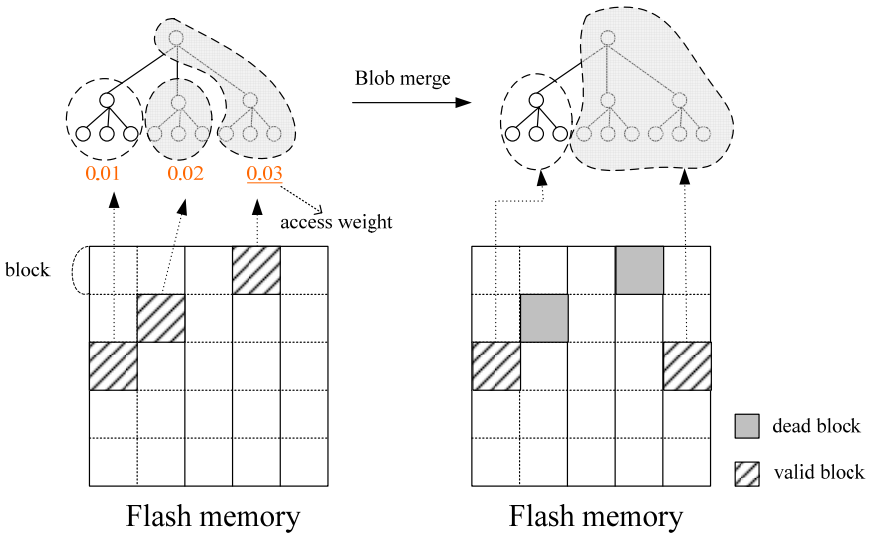


Figure 11.Blob merge

3.5 Garbage collection

當快閃記憶體空間不足時，必須藉由 garbage collection 來釋放空間。以 Figure 12(a).所示，一般的方式是選取一些 block，將這些 block 中 valid 的 page 重新寫到新的 block，再回收這些舊的 block 並且執行 erase 動作後釋放空間。但是在 blob 的設計下，每個 block 都已分配給一個 blob 使用，不同的 blob 內容不可存於同一個 block，所以上述一般的 garbage collection 方法並不適用於 blob 的設計下，因此必須要藉由 blob merge 的方式來達到 garbage collection 的目的。由於 garbage collection 與上述利用 locality 來減少 page-write 次數的目的不同，因此 blob merge 的 policy 也不同，另稱之為 GC merge。

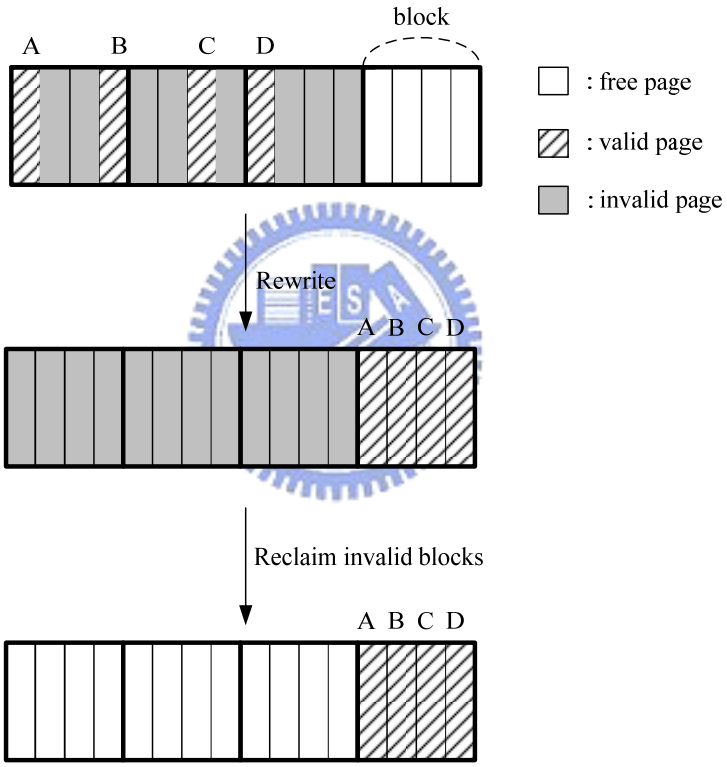


Figure 12(a). Garbage collection

GC merge 的目的是要將兩個 blob merge 後，釋放一個 blob 所分配的快閃記憶體空間。當發生 GC merge 時表示目前所有的 block 都已被分配給 blob 使用，為了避免執行 GC merge 太過頻繁，盡量降低 GC merge 後的新 blob 寫滿或是執行 blob split 的機率，所以選取 blob 內 node 數較少的 blob 來做 GC merge，其過程如 Figure 12(b).。因為 node 數較少可以避免 GC merge 後的新 blob 在不久之後寫滿，而再次需要快閃記憶體空間來寫入，或是 node 個數增加而必須執行 blob split。特別需要注意的是 GC merge 與 blob merge 的 policy 是不同的，這是因為兩者的目的亦不相同。

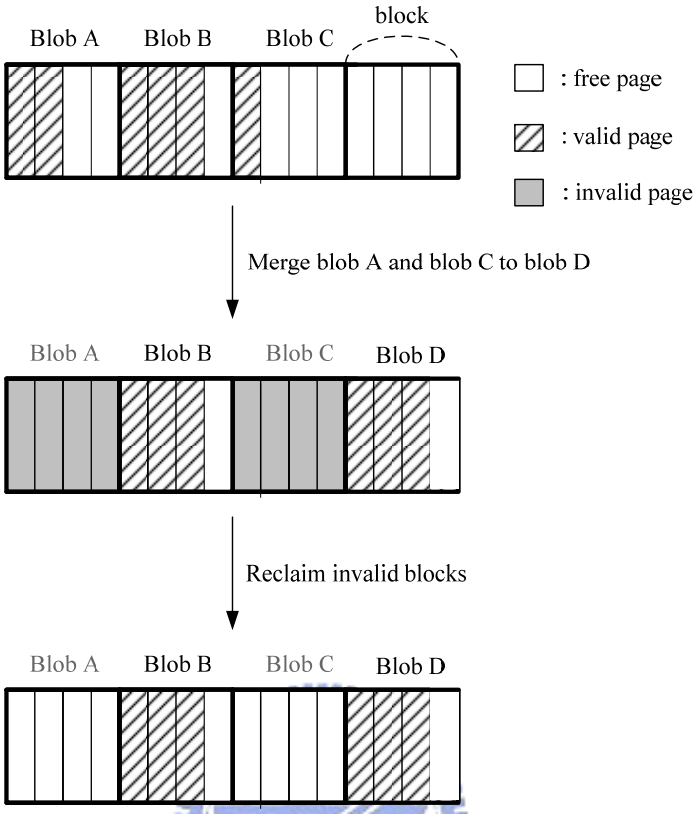


Figure 12(b). GC merge.

3.6 Blob cache

Cache 除了儲存從快閃記憶體讀取的 blob，也作為暫存更新的 index unit，其存取關係如 Figure 13.所示。為了讓快閃記憶體與 blob cache 存取更快速，不需要經過轉換。Cache 中所儲存的內容與儲存於快閃記憶體上的內容相同，即均為 index units。因此 blob 資料在 cache 與快閃記憶體之間存取時不需再經過轉換，可從快閃記憶體所讀取的 block 內容直接複製存到 blob cache，而要寫回的 blob 內容亦可直接完整寫入快閃記憶體。只有在執行 B-tree operation 時需先將 blob 內容轉換為 B-tree 的邏輯結構。Blob 的內容經過多次更新之後可能有越來越多已被更新的過期資料仍佔據 block 的空間，為了提高 cache utilization 避免 cache 浪費空間儲存這些已被更新的舊資料，因此從快閃記憶體讀取 blob 之後並不直接完整的將整個 blob 存放至 cache 中，而是先將 blob 的內容做 compact 的動作之後，再將 compact 後的 blob 內容存放至 cache，如此可以確保儲存在 cache 中的 blob 內容都是更新後的新資料，cache 不會浪費空間儲存已被更新的舊資料。Compact 的動作就是將 B-tree 的結構以最精簡的 index unit 儲存，以節省儲存的空間。

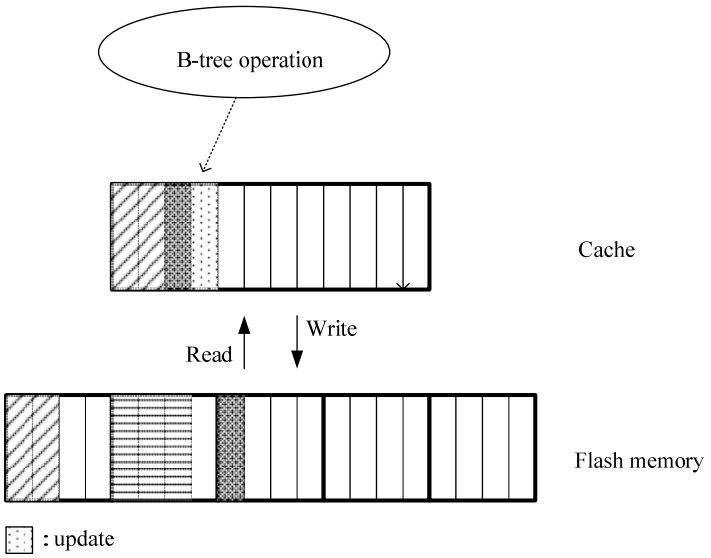


Figure 13. The relation between flash memory ,cache and B-tree layer.

當 cache 滿時會先將 cache 的內容做 compact，即為 cache compact，這個動作是要移除 cache 中無效的過期資料，並且將相同 blob 的資料壓縮，以最少量的 page 來儲存。如 Figure 14.所示，在 B-tree operation 的過程中，也可能製造一些 redundant 的 index units，如刪除 key 值或 node 的 index unit，若新增相同 key 值或同一個 node 的 index unit 仍存於 cache 中，即可將兩者一併刪除，消除 cache 中 obsolete items 的部份也可以提升 cache utilization。Cache 中有些 page 可能尚未存滿，但是更新的內容不屬於同一個 blob，因此必須重寫至 cache 其他 free page，為了將每個 page 存滿並且儲存有效的 index unit，屬於同一個 blob 的內容會往前填補尚未存滿的 page，藉此釋放出 free page。

若執行完 cache compact 之後 cache 仍空間不足，就必須要做 replace 來增加 cache 的可用空間。cache replace 的機制基本上是採用 LRU 的方式，但是為了減少 replace 的 overhead 做了一些修正。Cache replacement 的第一步會先看是否有 clean blob，也就是讀取至 cache 後尚未被修改過的 blob，因為選擇 clean blob 作為 replace 的對象不需寫回快閃記憶體，可以降低 cache replace 產生的 overhead，即減少 page-write 次數及處理時間。然而包含 root node 的 blob(稱為 root blob)每次都必須存取卻很少更新，即使為 clean blob 被取代之後，下一次的 B-tree operation 也立即存取到 root blob，若選擇 root blob 為取代的 blob，會造成每次 B-tree operation 都必須讀取 root blob，增加過多的 page-read 次數，直到 root blob 已被更新才能不被取代，因此選擇的 clean blob 不可包含 B-tree structure 的 root node。若找不到 clean blob，再進行 cache replacement 的第二步即由 LRU 來挑選出要 replace 的 blob。以 LRU

選出 replace 的 blob 必須做寫回快閃記憶體的動作。寫回之前會先計算更新部份所需的 page 數以及完整 blob 重新轉換所需的 page 數，再選擇較少 page 數的方式寫回快閃記憶體。

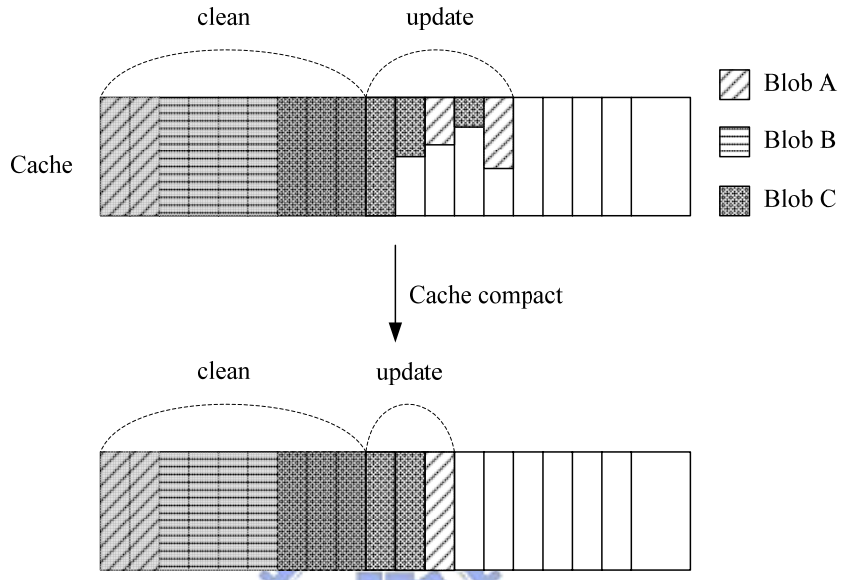


Figure 14. The process of cache compact.

四、Experiment results

此篇論文的實驗部份為實作 B-tree 索引結構，並分別以 blob 概念及 B-tree on NFTL 方式的模擬進行比較。

4.1 Experiment Setup and Performance Metrics

實驗部分為實作 B-tree 索引結構及存取動作，包含 insert key 及 delete key。首先採用本文所提出的 blob 的概念及相關 policy，包含 blob split/merge 及 GC merge；另一個部份為採用 B-tree on NFTL 的架構，並將系統中 scan virtual unit chain 的額外 page-read/page-write 部分去除，以達到公平的比較。環境部分以 64MB 的 NAND flash memory 及分別以 512、640 個 page 的大小作為 cache，其中 page size 為 2KB 且 block size 為 128KB，並且 Blob 的實作方式與 NFTL 均以相同的環境來做模擬。Blob 的實作部份有幾個 threshold 的參數必須事先設定，像是 blob split 的 node 數及 blob merge 的 access weight，以本實驗的設定為 blob split 的 node 數為 60，而 blob merge 的 access weight ratio 為 0.001。

實驗的目的為測量 page-read 次數、page-write 次數及 blob erase 次數。

由表可看出處理時間最長的是 block erase，page-write 次之，而 page-read 最短，因此比較的重點會著重於 page-write 次數及 block erase 次數。此論文提出的 blob 是針對大容量的快閃記憶體的設計，因此 block erase 次數會在實際的大容量快閃記憶體下減少許多，不會對於整體效能造成太大的影響。然而 page-write 次數不會因快閃記憶體容量大小而改變，因此 page-write 次數會是影響整體效能的關鍵。由於 blob 的設計是將快閃記憶體的存取最小單位設定為一個 blob，因此整體而言實驗數據可看出在 page-read 次數部份 blob 會較高，但是在 page-write 次數部份均大幅少於 NFTL 的架構。

4.2 Workload description

Workload 主要有兩部份；第一部分為 micro-benchmark，第二部份為 macro-benchmark。Micro-benchmark 以程式依照 sequential insert、random insert 及 normal distribution access(包含 insert 及 delete)的存取方式產生 B-tree key 值。Sequential insert：分別產生個數為 40000、80000、120000、160000 及 200000 的遞增數列作為 B-tree 存取的 key 值；Random insert：分別產生個數為 40000、80000、120000、160000 及 200000 的隨機數列作為 B-tree 存取的 key 值；Normal distribution access：先循序新增 130000 筆資料，再以 normal distribution 的方式進行 insert 及 delete 的動作，並分別產生個數為 40000、80000、120000、160000 及 200000 的 B-tree key 值，以及分別實驗具有一個 hot spot 的 normal distribution access 和具有兩個 hot spot 的 normal distribution access。單一 hot spot 的 normal distribution 的數值平均值為 60000，標準差為 2000；兩個 hot spot 的 normal distribution 的數值平均值各為 40000 及 100000，標準差為 2000。Macro-benchmark 以 firefox 的 SQLite 實際 B-tree 存取的 trace 作為實驗模擬的 key 值，分別實驗個數為 40000、80000、120000、160000、200000、240000 及 280000 的 B-tree key 值。

4.3 Micro-benchmark

4.3.1 Sequential insert

此實驗 page-read 及 block erase 次數均為 0，因此只討論 page-write 的部份。由 Figure 15.可看出 blob 在 page-write 次數遠低於 B-tree on NFTL，約為 B-tree on NFTL 的 10%~15%之間。因為循序存取的寫入部份幾乎都位在同一個 blob，所以 blob 吸收 update 部分的效果很好，可將每個 page 近乎寫滿。

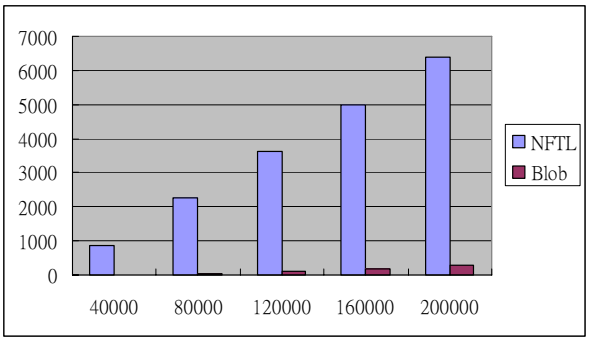


Figure 15. Number of page-write

4.3.2 Random insert

在 random 的存取下，很明顯採用 NFTL 的實作方式效能很差，雖然有 cache，但是存取的 node 變動太快，cache 的 replace 動作頻繁，造成 page-read 次數及 page-write 次數都提升很多。以 blob 的實作方式，由於 random 的新增 key 值會較為平均，因此每個 node 內的 key 數平均而言也會比較多，也就是相同 key 數的情況下形成的 node 數較少，並且整體系統的 blob 個數也較少。然而 blob 的個數越少，就越能有效的聚集 update 部份，減少 page-write 次數。這就是 blob 在 random 存取下仍能優於 NFTL 的原因，從 Figure 16(a).、Figure 16(b).及 Figure 16(c).可看出 blob 優於 B-tree on NFTL 的實驗結果。

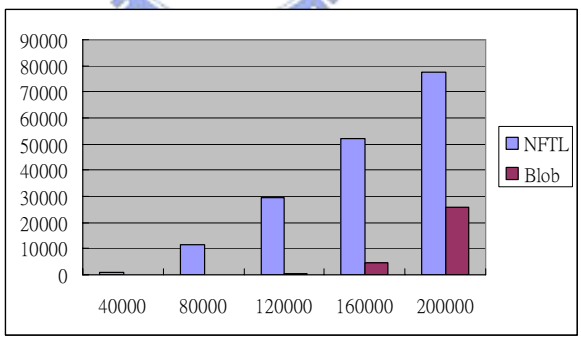


Figure 16(a). Number of page-read

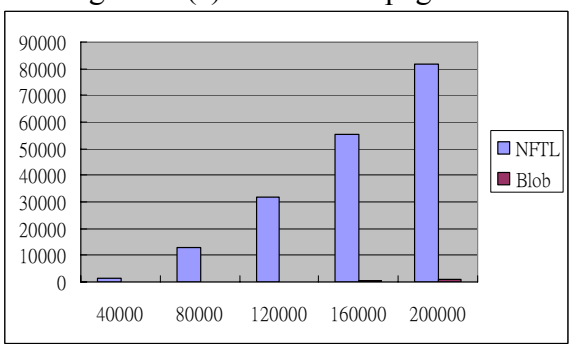


Figure 16(b). Number of page-write

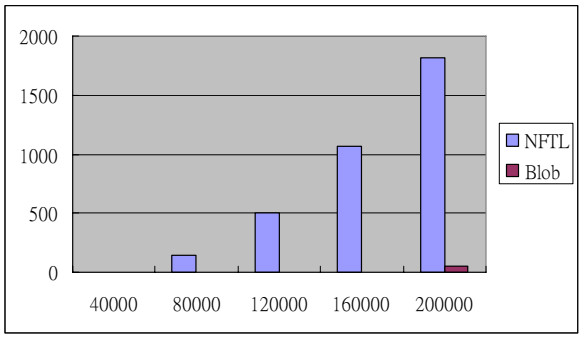


Figure 16(c). Number of block erase

4.3.3 Normal distribution access

從 Figure 17(a).、Figure 17(b).及 Figure 17(c).中可看出 blob 在 page-read 次數、page-write 次數及 block erase 次數都較 B-tree on NFTL 低，尤其是 page-write 次數及 block erase 次數更是大幅少於 B-tree on NFTL，page-write 次數僅為 B-tree on NFTL 的 4%~10%之間，block erase 次數最多也不超過 B-tree on NFTL 的 16%。在資料量增加之後，blob 的各項次數仍是以穩定的幅度增加，表示 cache 的處理上有穩定的 utilization，更可證明上個章節所提到 cache 的 policy 能有效穩定 utilization。

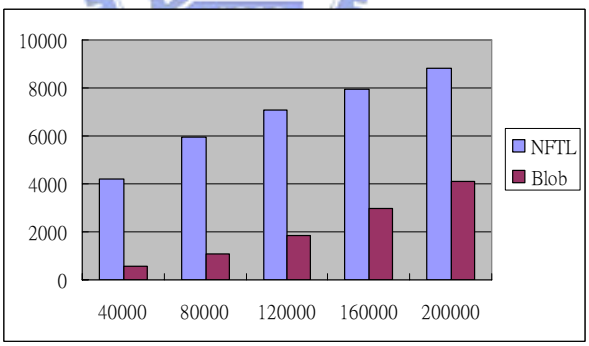


Figure 17(a). Number of page-read (1 hotspot).

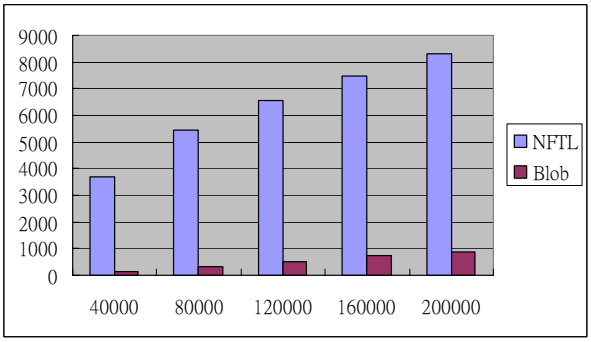


Figure 17(b). Number of page-write (1 hotspot).

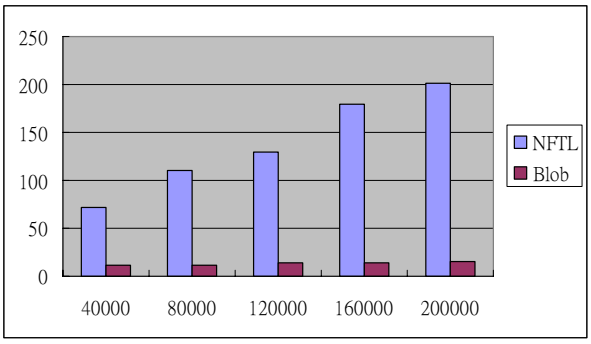


Figure 17(c). Number of block erase (1 hotspot).

此次實驗分別做了一個 hotspot 及兩個 hotspot 的 normal distribution，實驗結果看來兩個 hotspot 的 normal distribution 在 page-read 次數及 page-write 次數略多於一個 hotspot 的 normal distribution，因為兩個 hotspot 的 normal distribution 的資料相對於一個 hotspot 的 normal distribution 較為分散，資料會落在兩段 key 值的範圍，並且會隨機在此兩段 key 值的範圍輪流存取，因此會有兩群關聯度低的 blob 在 cache，使得 cache 執行 replace 的次數增加，導致 page-read 次數及 page-write 次數增加。

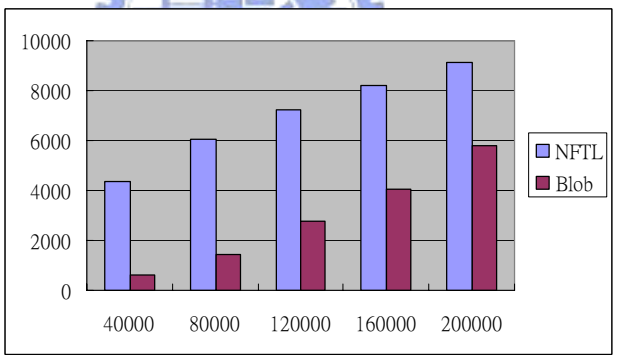


Figure 18(a). Number of page-read (2 hotspots).

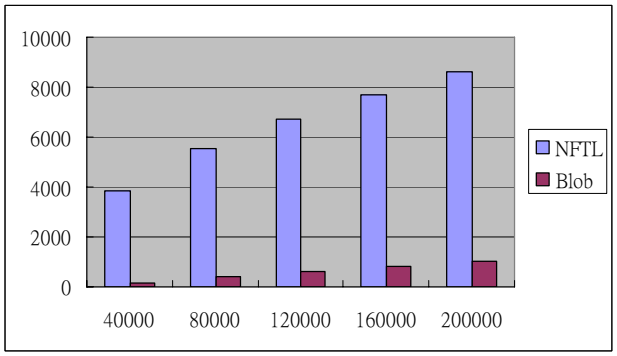


Figure 18(b). Number of page-write (2 hotspots)

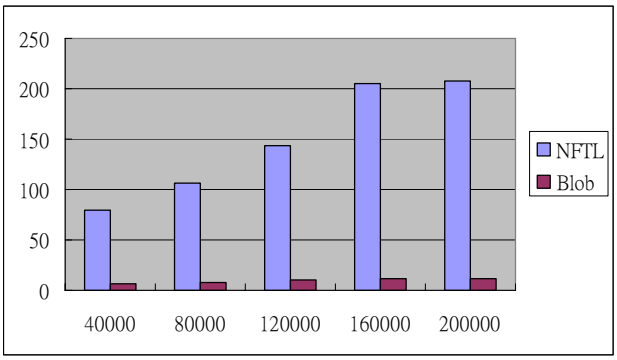


Figure 18(c). Number of block erase (2 hotspots).

4.4 Cache size

Cache 在提升整體效能上是很重要的角色，上述章節討論到 cache utilization 會影響 blob 的 page-read 次數及 page-write 次數，若是能充分利用 cache 空間並且提升 cache utilization，就能讓 cache 發揮最大的功效。因此這部份的實驗將探討不同的 cache size 分別對 blob 和 B-tree on NFTL 的效能影響，以及增加 cache size 之後的改良幅度。實驗 workload 為先循序新增 250000 筆 key 值，再以 normal distribution 的存取方式進行 50000 次的 request，而 normal distribution 的平均值為 150000、標準差為 2000，cache size 分別為 256、512、768、及 1024 個 page size。

由於 blob 是存取快閃記憶體的最小單位，不同於 B-tree on NFTL 是以一個 node 為最小的存取單位，每當發生 cache miss 時，讀取至 cache 的便是一整個 blob，而 B-tree on NFTL 可能僅少數幾個 node，因此 blob 在 cache 的使用上可能會較耗費空間，導致 cache replace 的動作可能也較頻繁。若使用過小的 cache size，以 blob 的設計可能無法達到預期的效能，從 Figure 19(a). 即可看出，cache size 為 256 個 page size 時，blob 的 page-read 次數暴增許多，顯示在這個 workload 之下這樣的 cache size 對 blob 而言是過小的。在 page-write 也有相同情況，從 Figure 19(b).發現 cache size 為 256 個 page size 時，blob 的 page-write 次數部分增加幅度較大，但是相較於 B-tree on NFTL 僅為二成左右的 page-write 次數，效能仍優於 B-tree on NFTL 許多。如 Figure 19(c).可知在 block erase 次數方面，blob 的數據顯示較不易受到 cache size 大小的影響，次數仍少並且增加幅度穩定。

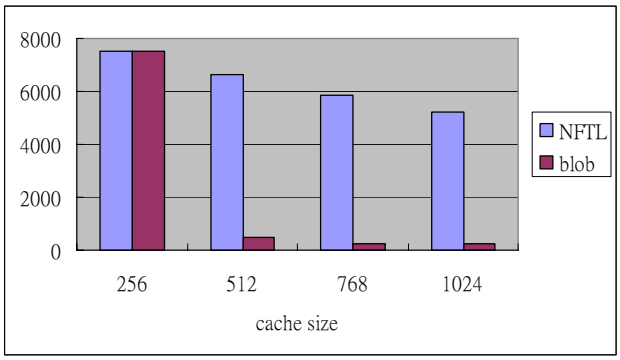


Figure 19(a). Number of page-read

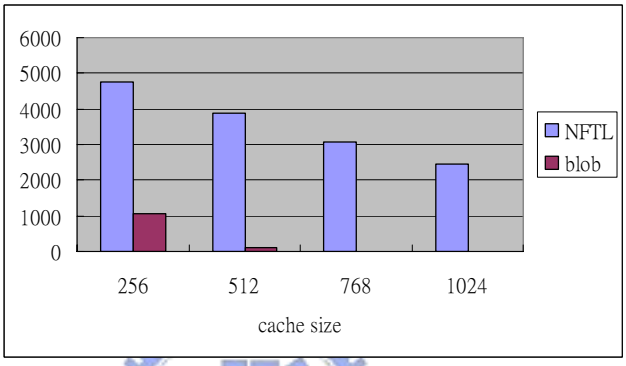


Figure 19(b). Number of page-write

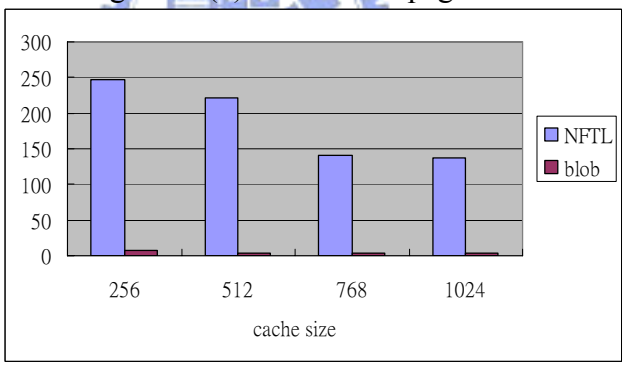


Figure 19(c). Number of block erase

4.5 Macro-benchmark

這個部份所使用的 workload 是網際網路瀏覽器 Mozilla Firefox 3.0.1 所產生的。Mozilla Firefox 使用的資料庫管理系統是 SQLite。SQLite 是採用 B+-tree 索引結構的關聯式資料庫，並且目前用於多種嵌入式系統上，像是 Nokia's Symbian OS 和 Google's Android。Mozilla Firefox 使用 SQLite 管理歷史紀錄、cookies 及密碼。Trace file 由經過網路使用一段時間 Mozilla Firefox 的 B+-tree operation 所記錄而成。

Mozilla Firefox 的使用亦具有 locality，但是可能在過程中會改變 locality

的區域，因此結合了 normal distribution 與 random access 的部份特性，從 micro-benchmark 來看，blob 在 normal distribution 及 random access 都能優於 B-tree on NFTL。因此從數據中看，blob 在 page-read 次數、page-write 次數及 block erase 次數都較 NFTL 少，並且隨著 key 值個數增加呈現緩慢增加的趨勢，可看出 blob 實作上的穩定度。從 Figure 20(a).、Figure 20(b).及 Figure 20(c).可看出，blob 在 page-read 次數、page-write 次數及 block erase 次數都大幅少於 B-tree on NFTL。

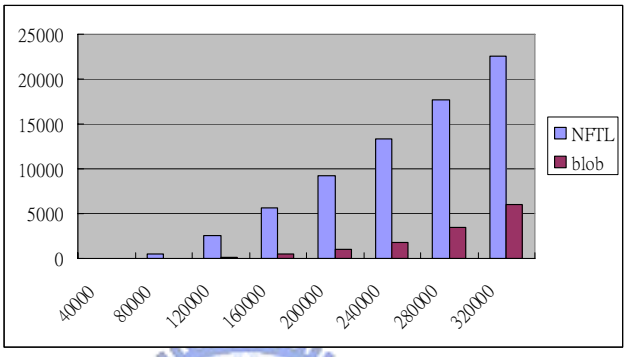


Figure 20(a). Number of page-read

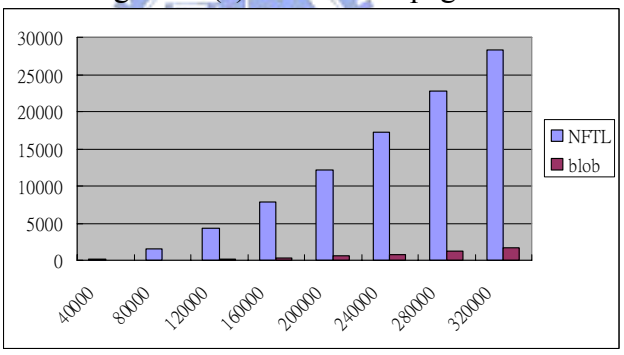


Figure 20(b). Number of page-write

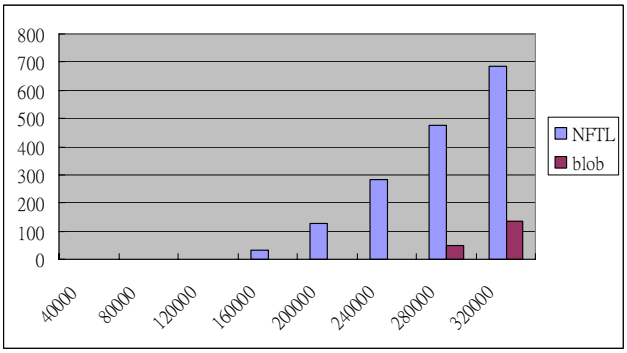


Figure 20(c). Number of block erase

五、Conclusion

由於快閃記憶體的容量擴增快速，需要一個更有效管理快閃記憶體的方

式。B-tree 是目前管理大量資料最常使用的索引結構，因此採用 B-tree 來實作管理大容量的快閃記憶體。邏輯指標的實作方式，造成系統開啟時掃描時間及儲存 translation table 空間上的浪費，因此本篇論文採用實體位址的實作方式。若以最自然的 B-tree 實作方式，即一個 node 對應儲存於一個 page，在 B-tree operation 中常會造成一連串的索引更新，而索引更新通常為很小部分的資料，並且 B-tree 的索引存取通常具有 locality 的關係，因此本篇論文提出 blob 的的實作方式有效管理快閃記憶體的檔案系統。

利用 B-tree 的存取特性將相連並且具有 locality 關聯的 B-tree node 聚集在同一個 blob，則可以將同一個 blob 內所有 node 的更新部份都存在同一個 page，因此大幅減少 page-write 次數。通常 B-tree node 的 key 值不會在全滿的狀態下，而 blob 的方式也可以充分利用 page 空間，將同一個 blob 的 node 資訊共同儲存於數個 page 中，可以減少快閃記憶體的使用空間。透過 blob split 及 merge 的動作除了可控制 blob 適當的大小之外，更重要的是重新將相同屬性的 node 聚集在相同的 blob，更加發揮 locality 的優勢來提升效能。

從實驗結果可看出不論是在 microbenchmark 還是 macrobenchmark 的部份，blob 的 page-read、page-write 及 block erase 次數均少於 B-tree on NFTL，因此證明 blob 可以作為管理大容量快閃記憶體的有效方法。

References

[1] A.B.Bityutskiy. JFFS3 design issues. http://www.linux-mtd.infradead.org.

[2] Samsung Elec. 2Gx8 Bit NAND Flash Memory (K9WAG08U1A). 2006.

[3] Samsung Elec. 2Gx8 Bit NAND Flash Memory(K9GAG08U0M-P).2006.

[4] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo, “An Efficient B-tree Layer Implementation for Flash-Memory Storage Systems,” ACM Transactions on Embedded Computing Systems.

[5] Dongwon Kang ,Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim,”µ-tree **:** An Ordered Index Structure for NAND Flash Memory”.

[6] M-systems, Flash-memory Translation Layer for NAND Flash(NFTL)

[7] Suman Nath and Aman Kansal, “FlashDB:Dynamic Self-tuning Database for NAND Flash”.

[8] Intel Corporation, “Understanding the Flash Translation Layer(FTL) Specification”.

[9] M. Rosenblum, and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” ACM Transactions on Computer Systems 10(1)

(1992)pp.26-52.

[10] M. Seltzer, K. Bostic, M. K. McKusick, and C.Staelin, “An Implementation of a Log-Structured File System for UNIX”, Proc. ’93 Winter USENIX, 1993.

[11] A. Kawaguchi, S. Nishioka, and H. Motoda, “A Flash-Memory Based File System,” USENIX Technical Conference on Unix and Advanced Computing Systems, 1995.

[12] L. P. Chang, T. W. Kuo,“A Real-time Garbage Collection Mechanism for Flash Memory Storage System in Embedded Systems,” The 8th International Conference on Real-Time Computing Systems and Applications (RTCSA 2002), 2002.

[13] SANG-WON LEE ,DONG-JOO PARK,TAE-SUN CHUNG, DONG-HO LEE,SANGWON PARK and HA-JOO SONG, ” A log buffer-based flash translation layer using fully-associative sector translation”, ACM Transactions on Embedded Computing Systems, Vol. 6, No. 3, Article 18, Publication date: July 2007.

[14] D. Woodhouse, Red Hat Inc., JFFS: The Journalling Flash File System.

[15] H. Kim, S. Lee,” A New Flash Memory Management for Flash Storage System, “. In: Proceedings of the Computer Software and Applications Conference (1999) 284.

[16] ERAN GAL and SIVAN TOLEDO, “Algorithms And Data Structures Of Flash Memories”, ACM Computing Surveys (2005).

[17] C. H. Wu, L. P. Chang, T. Kuo, An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems, ACM Transactions on Embedded Computing Systems 6 (2007).

[18] Hyun-Seob Lee1, Sangwon Park2, Ha-Joo Song3 and Dong-Ho Lee4, “An Efficient Buffer Management Scheme for Implementing a B-Tree on NAND Flash Memory”

[19] Aleph One Company,”Yet Another Flash Filing System”.

[20] S. Lim and K. Park, “An Efficient NAND Flash System for Flash Memory Storage,” IEEE transactions on Computers 55 (2006).