國立交通大學

網路工程研究所

碩 士 論 文

Skip-list 為基礎的 nor 快閃記憶體之檔案系統

A Skip-list-Based NOR-FLASH File System

研 究 生：游仕宏

指導教授：張立平　教授

中 華 民 國 九 十 六 年 十一 月

Skip-list 為基礎的 nor 快閃記憶體之檔案系統
A Skip-list-Based NOR-FLASH File System

研 究 生：游仕宏　　Student：Shih-Hong Yo

指導教授：張立平　　Advisor：Li-Pin Chang

國 立 交 通 大 學
網 路 工 程 研 究
碩 士 論 文

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

Computer and Information Science

November 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年十一月

..

Skip-list 為基礎的 nor 快閃記憶體之檔案系統

學生：游仕宏　　指導教授：張立平

國立交通大學 資訊工程 學系（網路工程 研究所）碩士班

摘　　要

在快閃記憶體大量用於嵌入式系統的同時，嵌入式系統上的應用程式強大的資料管理需求和快閃記憶體的大容量皆反應需要有一個好的資料管理方案。目前在快閃記憶體上的檔案系統或索引結構大都使用邏輯位址的方式，不僅掃描時間長且系統資訊需要佔用掉大量的記憶體。這篇論文試著使用實體參照的方式實作一個 skip list 為基礎的索引結構，克服實體參照在 nor 快閃記憶體上獨特的議題，如指標更新傳遞、垃圾回收死結等，提供一個有效率的資料管理方案。而文末將 demo 如何在 skip list 為索引結構的資料管理前提下，簡單的在上面疊架一個檔案系統。

關鍵字：快閃記憶體（Flash memory），儲存系統（storage systems），嵌入式系統（embedded systems），作業系統（operating systems），檔案系統(file system)。

A Skip-list-Based NOR-FLASH File System

student：Shih-Hong Yo　　Advisors：Li-Pin Chang

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

ABSTRACT

Flash memory has become a crucial component in building embedded computing systems. To manage a massive amount of data, index structures or file systems are needed. The physical characteristics of flash memory, write once and bulk erase in particular, impose many difficulties on realizing structured data. In past work, the notion of logical address space is introduce to decouple changes to data and to the physical addresses of them. To map between logical addresses and physical addresses, RAM-resident translation tables are needed, it may require lengthy initialization scan procedure. In this work, a skip-list implementation based on physical pointers is proposed. It is to investigate the feasibility of organizing data with physical pointers over flash memory. The benefits are that no translation tables and initialization scan are needed. It is also demonstrated that a file system can be easily realized based on the proposed skip-list implementation.

Keywords: flash memory, storage systems, embedded systems, operating systems, file system.

誌　　謝

帶著高興夾雜著些微感傷的心情，我即將大步邁出交大北大門。這二年四個月的時間，多謝教授的指導、朋友同學的相伴。回顧這段時間，有歡笑，有淚水－曾在工三 705 流過淚、在電資 701 爆過肝、在電資 705 組過團、在電資頂樓夜烤、夜唱笑傲江湖、夜尋大山背火金姑.....。因為有大家的參與，讓我的研究生生活在書本與程式之外，還伴著驚喜快樂，謝謝大家。

這一路走來，有了這篇論文的誕生，最需要達謝的是指導教授　張立平教授。在您的嚴謹但開放式的指導下，讓我可以自由的想像問題的解決之道；在我失去問題重心時，給予我指引、導正思路。因為您的研究和教導熱忱，帶領我進入且熟悉這個研究領域，才能完成這篇完整且嚴謹的論文。

感謝交大各位授課的老師們，提供如此多元的課程，增廣了我在資訊領域的範疇。也感謝遠道而來指導口試的教授　羅習五教授、陳雅淑教授，辛勞且細心的審閱，並且提供寶貴的建議，使得論文更臻完善。

兩年多來，感謝同學世豪、譽縉、光仁、俊達的激勵和幫助。因為大家的相互合作，使得我可以順利的走過這些日子。感謝千庭、辰暉、松德、家明學弟和惠茹學妹，因為你們的加入，使得實驗室溫暖熱鬧了許多。

感謝在我身旁的所有朋友，書顗、睿玨、秀花、建智、瑞銘、...，在這個求學的階段，還是不能少了你們，因為你們是我最大的精神糧食。

最後，謹以此文獻給我摯愛的雙親。

謝謝

目　　錄

中文提要 …… i
英文提要 …… ii
誌謝 …… iii
目錄 …… iv
表目錄 …… vi
圖目錄 …… vii
一、 Introduction …… 1
二、 Motivation …… 2
2.1 Flash-memory characteristics …… 3
2.2 Logical pointers …… 4
2.3 Physical pointers …… 5
2.4 Skip-list over Nor Flash …… 7
三、 Background and Related Work …… 8
四、 A Skip-List-Based File System over NOR flash … 9
4.1 Fundamental Issue …… 9
4.1.1 Pointer-Update Propagation …… 10
4.1.2 Garbage - Collection Deadlock …… 11
4.1.3 Fast initialization …… 12
4.2 A Skip-List Implementation …… 13
4.2.1 Physical Pointers and Block Layout …… 13
4.2.1.1 Object layout in blocks …… 13
4.2.1.2 Pointer-update mechanism …… 14
4.2.2 Garbage Collection …… 16
4.2.3 Split and Merge action of Partition …… 18
4.2.4 Pointer Scrubbing …… 21
五、 A Skip-List-Based File System …… 23
5.1 Metadata Objects and Data Objects …… 24
5.2 Key-coding scheme …… 24
5.3 File Operations and Directory Operations …… 26
5.4 File System mount …… 29
六、 Evaluation …… 30
6.1 Experimental Setup and Performance Metrics …… 30
6.2 系統指標讀取負擔分析 …… 34
6.3 機率和階層的分析 …… 36
6.4 一個區塊中預留指標個數和物件個數的比例評估 …… 39
6.5 區塊穩定度策略對系統 hot/code 資料的分佈影響 …… 40
七、 Conclusion …… 41

八、 Future Work …… 42
參考文獻 …… 43

表　目　錄

表一	各種不同索引結構以不同鍵值的排列順序插入並完全刪除65536個鍵值之後所發生之指標更新次數 ……	8
表二	在指標個數和物件個數的比例為2的條件下，不同指標最大高度使得物件大小不同而造成的空間分配情形 ……	31
表三	在十三組指標和物件比例下各空間的分配情形……	33

圖　目　錄

圖一	out-place update ……	4
圖二	logical address with mapping table ……	5
圖三	傳統上使用邏輯指標處理 out-place 更新。必需掃描整個快閃記憶體，同時建立轉換表和管理各資料間的控制訊息於記憶體中 ……	5
圖四	實體指標與邏輯指標的比較 ……	6
圖五	out-place 更新帶來的指標失誤 ……	7
圖六	out-place 更新會造成有問題的指標，而指標更新的同時，會有指標更新向源頭傳遞的現象 ……	10
圖七	垃圾回收的死結問題 ……	12
圖八	JFFS2 使用 RAM 的概念圖 ……	13
圖九	spare pointers can block the pointer update propagation	14
圖十	區塊分成 context area 和 pointer area ……	14
圖十一	快閃記憶體上指標更新的動作 ……	15
圖十二	區塊裡物件的放置和指標的使用狀況 ……	15
圖十三	部份分割示意圖……	16
圖十四	部份分割將指標更新傳遞侷限在最大高度指標的高度 ……	16
圖十五	區塊實際上的空間分配和邏輯示意圖 ……	17
圖十六	垃圾回收動作 ……	18
圖十七	區塊利用度依照資料的 hot\cold 來分配，hot 資料駐足的區塊利用度高，cold 資料駐足的區塊利用度低 ……	19
圖十八	分裂動作。將一個區塊裡的資料搬移到不同的二個區塊而降低空間利用……	19
圖十九	合併動作。將二個區塊合併成一個區塊而提高空間利用度 ……	19
圖二十	區塊穩定度和區塊利用度、分裂次數的示意圖和行為策略 ……	20
圖二十一	使用 lru 輔助合併動作時區塊的挑選 ……	21
圖二十二	pointer scrubbing ……	23
圖二十三	目錄結構隱藏於 skip list 的 key-coding 中 ……	25
圖二十四	目錄結構與檔案資料分別由 metadata list 和 data list 來維護	26
圖二十五	定址一個物件的動作 ……	27
圖二十六	建立或刪除一個物件的動作 ……	27
圖二十七	列舉動作 ……	28
圖二十八	對一個檔案寫入資料 ……	29
圖二九	sticky list ……	30
圖三十	super block list ……	30
圖三十一	skip list 上 20k 循序寫入和 20K 讀取，在各機率和各階層中一般指標讀取和 thread 指標讀取的趨勢圖 ……	34

圖三十二	skip list 上 20k 循序寫入、10K 讀取和 10k 更新，在各機率和各階層中一般指標讀取和 thread 指標讀取的趨勢圖	35
圖三十三	skip list 上 20k 循序寫入和 20k 更新，在各機率和各階層中一般指標讀取和 thread 指標讀取的趨勢圖	35
圖三十四	thread 讀取佔系統讀取次數的分析	36
圖三十五	skip list 上 20k 循序寫入和 20k 讀取後，各組機率階層配對的次數分佈圖	37
圖三十六	skip list 上 20k 循序寫入和 20k 讀取後，各組機率階層配對的時間分佈圖	37
圖三十七	skip list 上 20k 循序寫入、10k 讀取和 10k 更新後，各組機率階層配對的次數分佈圖	38
圖三十八	skip list 上 20k 循序寫入、10k 讀取和 10k 更新後，各組機率階層配對的時間分佈圖	38
圖三十九	skip list 上 20k 循序寫入、20k 更新後，各組機率階層配對的次數分佈圖。	39
圖四十	skip list 上 20k 循序寫入、20k 更新後，各組機率階層配對的時間分佈圖。	39
圖四十一	區塊抹次數與 ptr/ctt 比例的關係圖	40
圖四十二	區塊穩定度與區塊利用度呈一定正向關係	41
圖四十三	將物件中的串列頭端移進指標區域，可使用動態增長的方式更有效的利用空間	43

A Skip-List-Based NOR-Flash File System:

Exploiting the Use of Physical Pointers

Shih-Hong YO and Li-Pin Chang
Department of Computer Science
National Chiao-Tung University, Hsin-Chu, Taiwan 300, ROC
Mcinnis1010@gmail.com, lpchang@cs.nctu.edu.tw

Abstraction

在快閃記憶體大量用於嵌入式系統的同時，嵌入式系統上的應用程式強大的資料管理需求和快閃記憶體的大容量皆反應需要有一個好的資料管理方案。目前在快閃記憶體上的檔案系統或索引結構大都使用邏輯位址的方式，不僅掃描時間長且系統資訊需要佔用掉大量的記憶體。這篇論文試著使用實體參照的方式實作一個skip list為基礎的索引結構，克服實體參照在nor快閃記憶體上獨特的議題，如指標更新傳遞、垃圾回收死結等，提供一個有效率的資料管理方案。而文末將demo如何在skip list為索引結構的資料管理前提下，簡單的在上面疊架一個檔案系統。

1 Introduction

快閃記憶體因為其省電且抗震性佳等優點，目前已大量的使用在行動/可攜性的裝置上。以往的行動/可攜性裝置功能較單一，對儲存體的使用僅有儲存資料的需求；現今的行動/可攜性裝置多具有較強功能，其上的系統不乏有完整人機界面的作業系統，對儲存體的需求不在侷限於資料的儲存，更希望可以提供資料庫等索引結構或是檔案系統。因此隨著容量日趨增大的快閃記憶體，需要開始思考在快閃記憶體獨特的特性下，如何簡單實作管理資料的索引結構，並提供有效率的檔案管理。

雖然其獨特的物理特性，受到行動市場的青睞，然而也留下了待克服的問題－outplace update。快閃記憶體為單次寫入(write-once)且(bulk erase)的裝置，在效能和 wear-leveling 的考量下，update 的行為是以 out-place 的方式呈現。

傳統磁盤或磁帶的海量儲存體，可以很容易的在其上實作以實體指標(physical pointer)為基礎的結構性資料和指標參照；然而快閃記憶體

在行動/可攜性的裝置上漸漸取代傳統儲存體的同時，以實體指標為基礎而成就的結構性資料和指標參照，因為快閃記憶體 out-place 更新的物理特性，並無法如法炮製的直接移植到快閃記憶體上。以往在快閃記憶體上捨棄了實體指標的應用，改取採用邏輯指標的策略，雖然如期解決了 out-place 更新的問題，但卻需要負出額外的代價－1.掃瞄快閃記憶體裝置用以建立邏輯位址與硬體位置的對應表，需要耗費大量時間；2.儲存對應表需消耗大量的記憶體。然而這些代價剛好與快閃記憶體運用在行動/可攜性的裝置相抵觸。對於行動/可攜性裝置上寶貴的記憶體而言，大量消耗記憶體用以儲存對應表，會降低系統整體性的效能；而掃描時佔用的 cpu 資源，影響了行動/可攜性裝置的時效性並考驗著使用者的耐心程度。

本篇論文試著採用實體指標實作一個適合在 NOR 快閃記憶體上的索引結構－skip list[1]，避免邏輯指標在行動/可攜性裝置上佔用寶貴記憶體和掃描等待時間。然而實體指標在快閃記憶體 out-place 更新下有著幾個待解決的議題；1. 指標更新傳遞，2. 垃圾回收死結和 3. 快速啟動。更新指標傳遞會使得指標更新往源頭傳遞，而造成大量混亂的指標更新，文中使用預留指標(spare pointer)共用的概念，可有效的降低指標傳遞數量；因為垃圾回收時需要搬移有用的資料，使得指標更新問題在垃圾回收動作時耗掉部份的乾淨區塊(free block)，最後因為系統沒乾淨的區塊而讓垃圾回收動作無法完成，造成垃圾回收死結，文中採用部分分割的方式，減少垃圾回收時資料搬移的更新傳遞並帶進分裂(split)、蝙合(merge)和垃圾回收動作之間的策略，最後使用 pointer scrub 的方法同時解決指標傳遞和垃圾回收死結。而在快速啟動議題方面，採用 sticky list 的概念，提供快速啟動的同時並不會快速損耗快閃記憶體。

文末使用模擬的方式，對使用實體參照在 NOR 快閃記憶體上的 skip list 索引結構做一個完整的測試，1. 分析系統指標讀取負擔(system overhead)，在正常搜尋動作外還有那些額外負擔，這些負擔對系統會造成什麼影響 2. 探討機率和階層在讀取指標、讀取物件和寫入指標間造成的效能差異 3. 評估文中採用空間管理方法預留指標個數和物件個數間的比例　4. 最後測試分裂(split)、蝙合(merge)和垃圾回收採用策略的成效。

2　Motivation

著眼於快閃記憶體在講求有時效且有效利用資源的行動/可攜性裝置上的應用，這篇論文試著使用實體指標於 NOR 快閃記憶體，探討並解

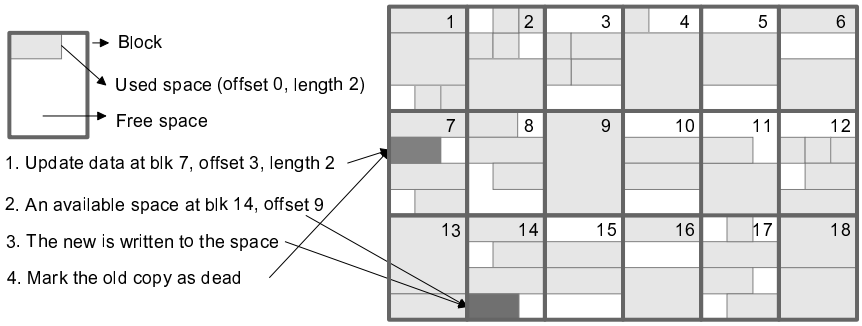
決所帶來的問題，期望提供一個組織結構性資料的索引結構，進而運用此索引結構完成一個檔案系統，為行動/可攜性裝置提供一個良好的儲存體管理機制。

2.1 Flash-memory characteristics

快閃記憶體在提供行動/可攜性裝置喜愛的省電和抗震性佳等特性外，也帶來一些嚴重待克服的問題。市面上的快閃記憶體分為NAND快閃記憶體和 NOR 快閃記憶體二類，共通的物理特性為單次寫入(write-once)、大塊抹除(bulk-erasing)和有限的抹除(erase)次數。一個 NOR flash 被區分為許多的區塊。在快閃記憶體上，連續兩次寫入到同一個實體位置上，都必須間隔一次區塊抹除。而區塊的大小相當大，一般來說大約為 128KB。一次區塊抹除會將該區塊內所有資料全部擦除。一個典型的 NOR 快閃記憶體區塊可容忍 10k 次的抹除動作。若超過該極限，則一個區塊將會無法可靠地存取資料。NOR 快閃記憶體讀取速度非常的快，但寫入的速度慢，抹除速度更為緩慢。參考實驗模擬對像廠商的快閃記憶體硬體規格[17]和市面上主流廠商 samsung 的快閃記憶體硬體規格[16]，目前普遍的 nor 快閃記憶體讀取一個位元組的時間為 80ns、寫入一個位元組的時間為 9us 而抹除一個區塊的時間是 0.7sec。

因為一次寫入(write-once)和大塊抹除(block-erasing)等特性，當一份資料需要更新，該資料並不能直接被覆寫(overwrite)。一種方法是將該資料所在的區塊中有用的資料先複製到 RAM，在 RAM 中做資料修改，抹除該區塊，再將更新過的資料寫回原本區塊。如此一來，每次寫入之前都會需要一次區塊抹除。此外，如果某些資料常常被更新，則該所屬的區塊將快速地被磨損。再者，做抹除動作亦需先將該區塊上有用的資料(valid data)保存在 RAM 中，而發生系統忽然斷電遺失資料的情況。因此在效能、平均抹損次數以及資料一致性的考量下，快閃記憶體上皆是採用 out-place 更新的方式。

圖一為在 NOR 快閃記憶體上 out-place 方式的示意圖,當第 7 個區塊上偏移位置為 3 且長度為 2 的資料需要更新，必需找一個同等大小的空間（區塊 14，遍移位置 9）為更新位置；然後將資料搬移過去並且將原本位置標記為 dead 而完成一個 out-place 更新的動作。



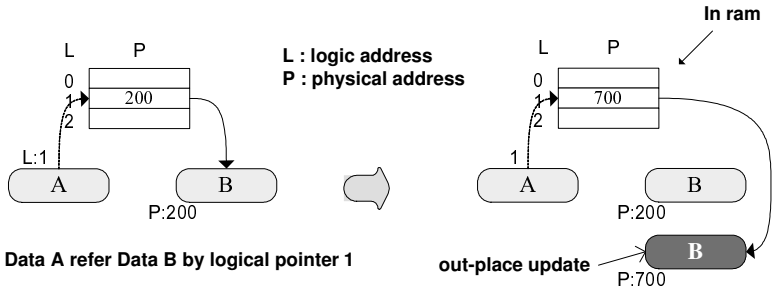
圖一　out-place update

NOR 快閃記憶體最小的讀寫單位可至位元層次(bitwise)。而因為採用標準記憶體界面，所以可以就地執行(eXecute-In-Place, XIP)。一般來說，NOR 快閃記憶體主要用來儲存二進位執行碼，而 NAND 則用在外部儲存系統。NOR 快閃記憶體的密度低且價格昂貴，所以位元成本偏高。為了節省成本，小型的嵌入式平台只會使用一顆 NOR 快閃記憶體，而將它分割為儲存執行碼的部份以及儲存資料的部份。本論文是以 NOR 快閃記憶體的管理為研究基礎。

2.2　Logical pointers

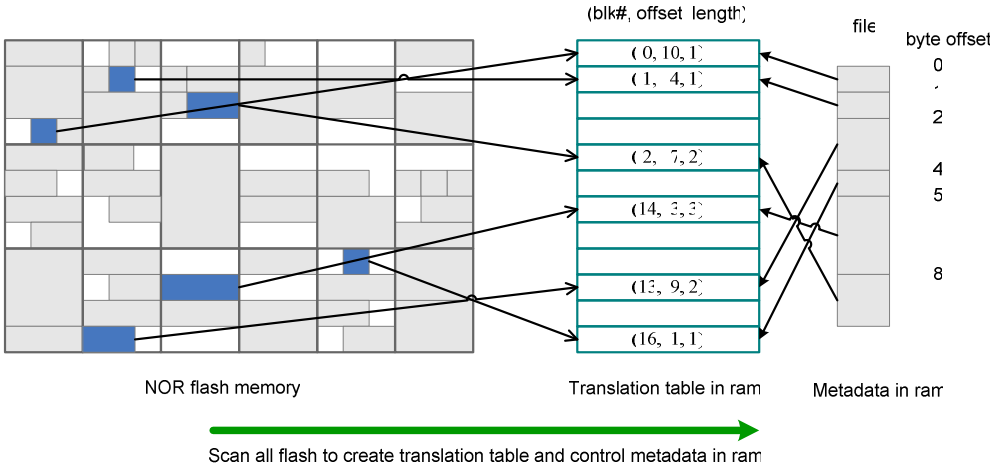
目前在快閃記憶體上的大部份資料管理方式，大多是使用邏輯指標的設計方式。邏輯指標是將每一份資料都賦予一份邏輯位址，並以邏輯指標來參照資料。如此一來，每一份資料便對應了其邏輯位址以及實體位址（快閃記憶體上的物理位址）。邏輯位置可以是檔案系統中的檔案編號或是檔案位移。當一份資料被更新時，心須先找到對應到這個邏輯位址的資料是在快閃記憶體上的哪個實體位址，接著執行 out-place 更新動作，而後把對應這個邏輯位址的快閃記憶體實體位址更新。

如圖二所示，說明資料 A 如何使用邏輯指標參照到資料 B。首先在記憶體中需在有一份邏輯位址對應到實體位址的轉換表，對應某資料的邏輯位址到其實體位址。假設賦予資料 A 和資料 B 的邏輯位址分別為 0 和 1，則資料 A 上記錄參照到資料 B 的邏輯位址即為 1，透過此邏輯位址 1 為索引值查尋對應表，即可得到資料 B 在快閃記憶體上實體位址為 200。假設系統對資料 B 做更新動作，則新的資料因為快閃記憶體的 out-place 特性而需要寫於另一個位址 700，此時僅需更改資料 B 邏輯位址參照到的轉換表欄位的值為 700，資料 A 即可再透過邏輯指標 1 參照到資料 B。



圖二 logical address with mapping table.

然而邏輯指標方式會帶來兩個主要的缺點：1.掃描整個儲存體建立轉換表會耗費過長的掃描時間；2.此轉換表和其它的系統控制訊息將消耗大量的主記憶體。如圖三所示，邏輯指標必需掃描整個快閃記憶體裡的資料，得到各資料的實體位址和該資料對應的邏輯位址，才能建立各資料的邏輯位址到實體位址轉換表。因此，掃描動作將隨記憶體愈大而需要愈長的時間。除了轉換表將佔用大量 RAM 空間外，因為各資料間相依關係的控制訊息無法使用邏輯指標參照呈現，所以在掃描的過程中，必需建立系統控制訊息，而這些控制訊息也額外的佔用部份 RAM。掃描時間過長，會使得系統停頓一段很長的時間無法工作。而轉換表和控制訊息佔用大量的記憶體，會使得系統上系統程式或使用者程式可用的 RAM 變少，降低整體系統效能。



圖三 傳統上使用邏輯指標處理 out-place 更新。必需掃描整個快閃記憶體，同時建立轉換表和管理各資料間的控制訊息於記憶體中。

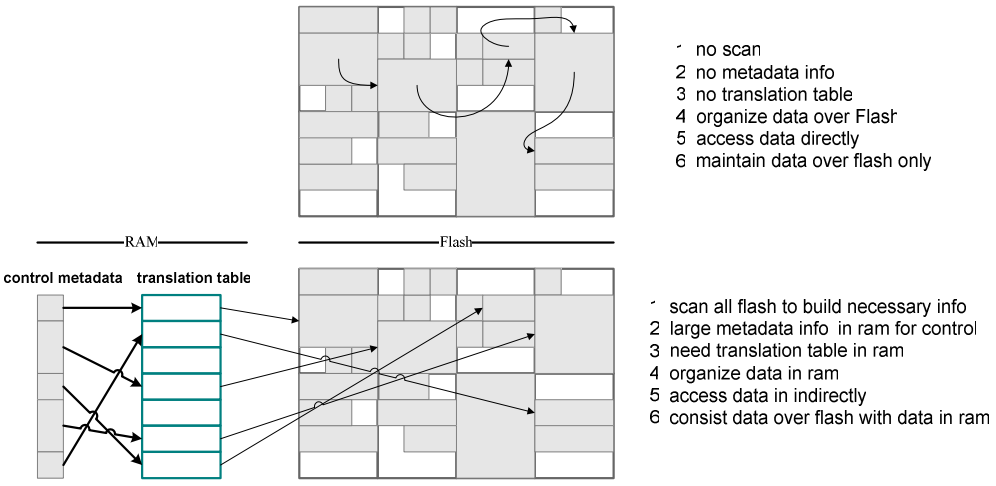
2.3 Physical pointers

邏輯指標主要是一種對付 out-place 更新方式的作法。這種作法下，不論資料目前被放置的實體位置為何，該資料的邏輯位置仍然不變。

不過，卻也付出了 1.掃描時間過長和 2.儲存轉換表或控制資訊於記憶體中的缺點。

基於嵌入式平台需要快速啟動，卻又沒有足夠的 RAM 的條件下，我們轉而研究是否能夠在 NOR 快閃記憶體上使用實體指標。使用實體指標的好處恰巧可與使用邏輯指標的缺點互補。因為資料彼此之間是以實體位址互相參照，因此並不會有需要掃描或者建構位址轉換表的需求。若能成功地在快閃記憶體上實施實體指標，則可達到節省 RAM 以及降低掃描時間的目的。且實體指標帶來直接存取資料的好處，不需透過轉換表來轉換位址。在資料的控管方面，實體指標僅需管理快閃記憶體上的資料，而不用像邏輯指標同時管理 RAM 和快閃記憶體上的資料，讓資料間的關係維持一致性。

圖四顯示實體指標相較於邏輯指標帶來的好處。實體指標不需要掃描動作，而邏輯指標需要掃描快閃記憶體才能收集並建立轉換表和系統控制訊息。在 RAM 的使用方面，因為實體指標以實體位址互相參照來組織資料，不需像邏輯指標必需將資料間的關係組織在 RAM 中，消耗一部份的 RAM;且邏輯指標需要的轉換表，也佔掉一部份的 RAM 空間。實體指標方式直接透過指標參照即可存取資料，不同於邏輯指標需要轉換到實體位址後才可存取在快閃記憶體上的資料。且邏輯指標在更動快閃記憶體的資料後，也需要對 RAM 中的控制訊息或轉換表更新，維持快閃記憶體和 RAM 中訊息的一致性，然而實體指標僅需更新快閃記憶體上的資料，沒有訊息一致性的問題。



圖四 實體指標與邏輯指標的比較

雖然使用實體指標在一般的 RAM 或者磁碟上是一種理所當然的選

擇，不過由於 out-place update 的限制，在快閃記憶體上實作實體指標將會有相當大的挑戰！如同文後所探討的，將會有 1.指標更新傳遞(pointer update propagation)－簡單的更新動作會造成一連串的失去控制的指標更新動作；2.垃圾回收死結(garbage-collection deadlock)－指標更新在垃圾回收其間將消耗乾淨區塊，造成系統死結；3.快速啟動(fast initialization) －如何在不快速損毀特定區塊的條件下，加啟動速度等三個議題。

2.4 Skip-list over Nor Flash

在一般組織資料的結構資料，如檔案系統和索引結構，皆是將預設格式套用在資料上，並且使用指標參照來組織資料。如 ext2 檔案系統的 inode 是用指向資料區塊的指標和間接指標；而高等樹會將各節點(node)以指標連接起來。而在各個結構性資料中，因為需要維護各資料間的相依關係，在維護的操作中往往需要更新指標。因此，一般結構性資料的更新動作，大部份會由指標參照所引起。

Out-place 更新的物理特性，將更新的資料寫入其他的實體位址，若用實體指標的設計方式，會使得指標參照出現問題。如圖五所示，因為 out-place 更新是將新資料寫入其他位址，而原本參照到資料 B 的指標將指向無用的舊資料。所以在快閃記憶體上使用實體指標設計，必需額外處理這些因為 out-place 更新而有問題的指標。

圖五 out-place 更新帶來的指標失誤

因此，不論是檔案系統或者索引結構，在快閃記憶體上實作的問題本質正是指標及資料的更新。如果考慮要在快閃記憶體上實作一種資料結構，在資料結構撰擇上，由於資料的更新本身無法避免，因此會希望該資料結構所需要的指標更新動作不能太頻繁。

以 RBtree 與 hash 這兩種典型的索引結構為例。表一為 skip list、RB tree 和 hash 在不同鍵值的排列順序插入並完全刪除 65536 個鍵值之後由指標所引起的更新次數。因為 RB tree 需要維持平衡的旋轉動作，使其在循序插入時產生了大量的更新(指標的更新)；也因需拿 leaf node 來填補被刪除節點的空缺，在隨機順序的刪除動作亦會造成大量的更新

(指標的更新)。雖然雜湊帶來較少指標的更新負擔，但因為雜湊為一個空間換取時間的資料結構，對於錙銖必較的 nor 快閃記憶體來說，是一項不利的因素；再則雜湊無法提供範圍查尋亦是索引結構的選擇中，望之怯步的原因。

	Sequential order	Random order
Red Black Tree	360274	310968
Skip List (p=0.5)	193306	192360
Skip List (p=0.25)	168108	165408
Hash (linear-list buckets)	131072	82776

表一 各種不同索引結構以不同鍵值的排列順序插入並完全刪除 65536 個鍵值之後所發生之指標更新次數

在各種索引結構中，我們發現 skip list 是一種相當好的選擇。Skip list 為一個單一鏈結串列，節點高度是在有限制的高度下隨機成長，而維持各節點高度的關係，僅是處理串列上的指標問題，不僅更新次數少，並可帶來 amortized O(logn) 的效率。Skip-list 將問題聚焦在指標的處理上，單就串列上的指標做更新的動作，不像一般高等樹除了平衡高度時需要大量的更新(指標的更新)外，還有更改記錄平衡高度訊息的額外更新。

綜上所述，本論文嘗試以使用實體指標的方法，來在 NOR flash 上有效率地建構 skip lists。成功實作 skip lists over NOR 快閃記憶體後，我們便可以在上面方便地堆疊資料庫或者是檔案系統。

3 Background and Related Work

目前在快閃記憶體的空間管理和使用方面，大致可分為系統整合，原生檔案系統和索引結構三個方向來探討。在傳統的儲存媒體中，即有很多有效的管理儲存媒體的研究，如何將這些方法快速的移植到新的儲存媒體快閃記憶體上為系統整合這層面的首要之務，在這方面的研究稱為區塊裝置模擬方法(block-device emulation)。將現有的檔案系統或應用程式架構在所模擬的區塊裝置上，不需要有任何的改變，即可直接使用快閃記憶體。可以透過中央處理器(cpu)和主記憶體(main-memory)來完成[2][3][19][20]，也可以由儲存裝置上獨立的微控制器以及嵌入式記憶體(embedded memory)來達成[18]。主要概念即是在系統和儲存媒體之間置入一個邏輯區塊到實體區塊的轉換層於記憶體中，巧妙的掩蓋了下層儲存媒體快閃記憶的特性，提供可在其上快速疊架目前現有的檔案儲存系統的優勢。

原生檔案系統即是在檔案系統的開發過程中，便周詳的考慮快閃記憶體的物理特性。有別於區塊裝置模擬方法，原生檔案系統可以提供檔案系統層級的操作，因為可獲得檔案系統的資料資訊，使得在檔案的管理方面，可以有效的結合快閃記憶體的物理特性而制定適合快閃記憶體的各項策略，如垃圾回收機制和平均區塊抹除方法。目前知名的原生檔案系統有 JFFS1/2/3[4][5][6]、YAFFS1/2[7]，它們在管理儲存媒體和使用記憶體方面的共同特性除了皆使用邏輯位址而需要一份邏輯位址到實體位址的輯換表外，在系統可以正常執行前，亦需要將整個檔案系統的控制資訊或是階層架構收集於記憶體中，以供檔案系統的管理使用。

在管理外部儲存媒體的索引結構方面的研究，有架構在 FTL 之上的 B-tree[8]和 R-tree [9]，還有使用在感測裝置(sensor device)上的 MicroHash[10]。在 FTL 上的 B-tree 和 R-tree 考慮小量寫入對快閃記憶體造成的不良影響而提出保留緩衝器的概念，收集一定量對索引結構的修改於保留緩衝器後，在將其以分區(sector)的方式透過 FTL 寫入快閃記憶體中，因為對某一部份索引結構的修改散佈在不同的分區中(sector)，尚須一個結點轉換表(Node Translation Table)來管理這些對索引結構的修改而散落在不同分區(sector)的資訊。MicroHash 為感測裝置(sensor device)所設計的外部儲存媒體的管理索引結構，使用實體指標實作方式且可提供快速的搜尋和良好的平均區塊抹除策略，然而存於其上的資料皆為時序性資料(temporal data)，記錄時間和有限數字為其資料特性，不同於一般外部索引結構所管理的資料。

由於快閃記憶體容量不斷提升，近來開使探討如何快速初使化一個高容量的快閃記憶體儲存系統。在以往使用邏輯位址的方式下，有 Szeded 大學項獻到開放原始碼的摘要計術 JFFS-summary[11]，其作法是檔案系統被卸載(unmount)前，將記憶體中的邏輯實體位址的轉換表與空間管理資料結構等資料，壓縮儲存在快閃記憶體較小的集中區域，供下次啟動時快速讀取。類似的尚有 Yim 等人在論文”A Fast Start-Up Technique for Flash Memory Based Computing Systems”[12]提出的方法。而指導教授也在"Efficient Initialization and Crash Recovery for Log-based File Systems over Flash Memory"[13]一文中提出藉由遞增的去寫入一啟動時所需的資料，達到快速開機的同時，亦可提供系統在無預警的斷電後，還有部份資料可供系統啟動。

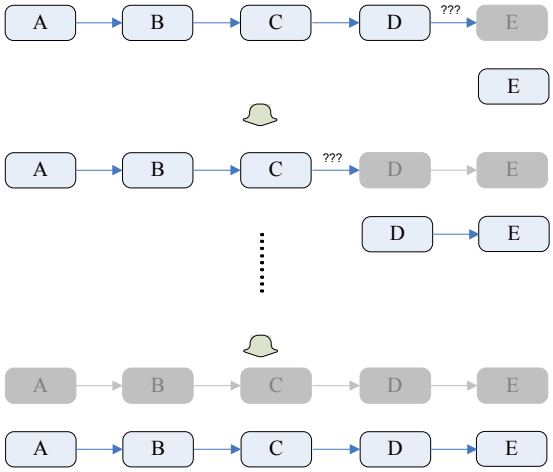
4 A Skip-List-Based File System over NOR flash

4.1 Fundamental Issue

在快閃記體上實作實體指標與在快閃記憶體上使用邏輯指標有即大的不同。在捨棄需要耗費大量 RAM 且需要掃瞄 RAM 的邏輯指標而採用實體指標，最直接面對的問題即是指標更新傳遞(pointer update propagation)。指標更新傳遞不僅在索引結構中造成大量且失去控制的指標傳遞，也使得垃圾回收動作不同於以往，以下為此篇論文在快閃記憶體中使用硬體指標所要解決的議題。

4.1.1 Pointer-Update Propagation

索引結構的基礎就是指標的參照，而快閃記憶體為一次性寫入的裝置，這跟以往以磁碟或磁帶為海量儲存體可以直接寫入(overwrite)更新指標參照有著極大的差別；在快閃記憶體上實現指標參照需採用out-place的更新方式。然而out-place更新實體指標參照會產生一個代表性的問題－wandering list，如圖六所示：假設一個list中，所有node皆是使用實體指標參照，考慮node E被更新，因為out-place更新的特性，node E需被更新在其他空餘空間上，因而改變其原本的位址。而參照到node E的node D，因為其指標參照的位置為舊node E的位置，不是被更新後的位置，所以node D必需被更新，並將參照到node E的指標更新到新的位置。但此時node C參照到node D舊的位址，所以node C也必需重新寫入。如此更新或是重新寫入的動作，沿著list的前後指標參照往list的頭端傳遞，而僅是一個簡單的node被更改所造成，這就是wandering list效應。



圖六　out-place 更新會造成有問題的指標，而指標更新的同時，會有指標更新向源頭傳遞的現象。

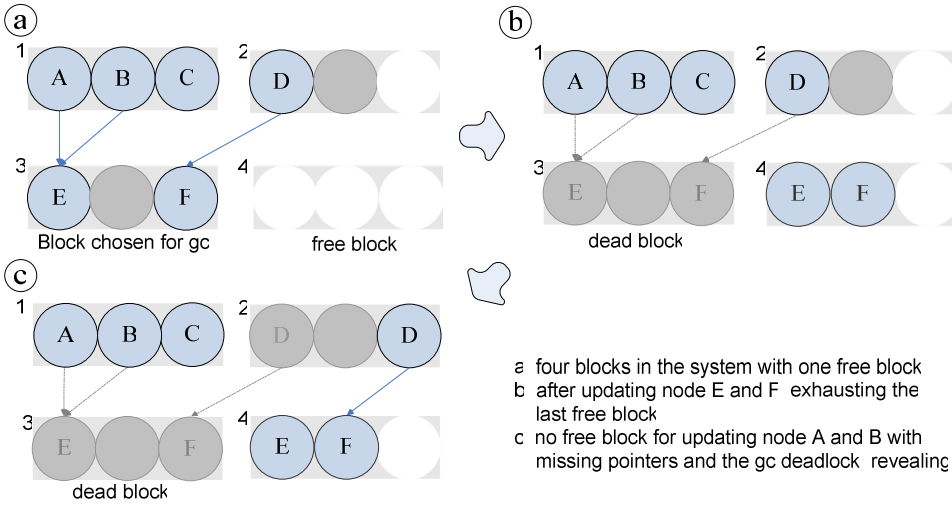
Wandering list 效應使得簡單的 out-place 更新一個資料物件時，卻造成一連串無法預估的大量重覆更新/重新寫入的動作，這在文中想以

實體指標參照來完成索引結構來說，是首要解決的基本議題。

4.1.2 Garbage-Collection Deadlock

垃圾回收在快閃記憶體上是一個很古典的問題。隨著不斷out-place更新的動作，快閃記憶體上可供寫入的空間逐漸的變少，然而佔據快閃記憶體空間的不見得都是有用的資料(valid data)，有一部份是out-place更新所遺留的無用資料(dead data)。因此，必須以抹除(erase)的方式，回收那些被無用資料(dead data)佔據的空間，但在快閃記憶體上抹除(erase)的動作是以一個區塊(block)為單位，因此混雜著有用資料(valid data)和無用資料(dead data)的區塊在被抹除之前，必需將有用的資料搬到其它空餘空間，而後才能被抹除，如此才完成一個垃圾回收的動作。

因為實體指標參照的關係，使得垃圾回收如此古典的問題不同於已往以邏輯指標參照來的直覺簡單。垃圾回收需要將有用的資料搬移到空餘空間，在實體指標參照為前提下，必定使得前後有指標參照關係的node必需跟著被更新，重新寫入在其它地方，讓區塊中的有用資料和無用資料顯得更混雜；因為資料搬移動作會使得指標更新向外擴散，使得垃圾回收動作在不斷更新的過程中必需消耗額外的乾淨區塊。如圖七所示，左上角的圖a顯示系統中的四個區塊，其中一個為乾淨區塊(free block)，一個為被垃圾回收機制選擇為抹除的區塊；右上角的圖b顯示垃圾回收機制在區塊抹除前需將有用的資料節點(node)E和節點F搬移到乾淨區塊(區塊4)；此時節點A、B和D的指標皆會出現參照到舊址的問題，而必需將這些指標更新。左下角圖c顯示節點D可以更新在相同區塊的剩餘空間。然而，若要更新資料A和資料B的指標，將發現系統沒有多餘的空間供重新寫入而無法完成垃圾回收動作，造成系統的死結。

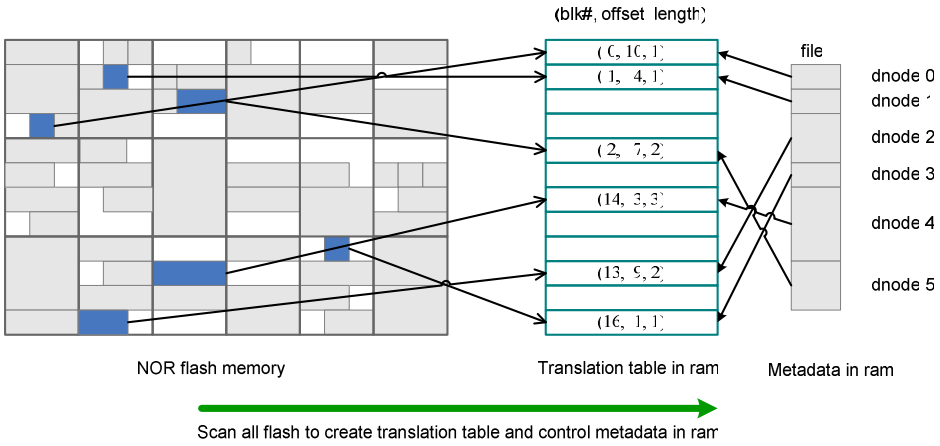


圖七　垃圾回收的死結問題

垃圾回收的死結問題，在過往的研究中之所以沒有被探討，是因為過往的作法是採取邏輯指標參照。在邏輯指標的策略下，節點 A、B 和 D 不需要被更新。而實體指標策略引起指標更新傳問題，會使系統消耗更多的乾淨區塊，而造成系統死結。此篇論文著眼於實體指標參照為行動/可攜性裝置所帶來的好處，因此垃圾回收的死結問題為另一個待克服的議題。

4.1.3　Fast initialization

行動/可攜性裝置的各元件啟動速度愈來愈被使用者所挑剔。以往在快閃記憶體使用的邏輯指標策略下，因為要建構出邏輯位址到實體位址的轉換表和系統控制訊息，往往需要耗費大量的掃瞄時間。圖八顯示 jffs2在NOR快閃記憶體上使用轉換表的大致情況。針對jffs2的一個檔案而言，除了建立一個轉換表外，在元件開始執行前，一個檔案中每部份的資料參照(dnode)皆需存在於記憶體中，而這些metadata為元件啟動時必要收集的資訊。而為了收集這些必要資訊，使用邏輯指標參照的方式需在元件啟動時掃描快閃記憶體上的各資料，使得啟動速度隨著快閃記憶體愈大、資料量愈多而愈來愈慢，而收集的必要資訊也大大的佔用了記憶體空間。然而若是將常被更新的特定資料或表格固定在快閃記憶體的某一區塊，雖可以加快啟動速度，但這些持定資料或表格因為常被更新，勢必會使得其所駐足的區塊率先因為多次抹除而損毀。



圖八 JFFS2 使用 RAM 的概念圖

因此必需在提供快速啟動和不造成特定區塊被快速worn-out的前提下，迅速收集啟動時所需的所有資料，為此篇論文需顧慮的第三個議題。

4.2 A Skip-List Implementation

skip list為文中選擇的索引結構。skip list將所有的問題聚焦在list本身的問題上，也就是指標的管理，不需要雜湊的大量空間，也不用額外的balance動作，然而針對快閃記憶體的out-place更新特性，實作在快閃記憶體上的skip list也需要有所改變。以下為此篇論文在指標空間管理，區塊空間管理和指標更新傳遞所帶來問題的討論和解法。

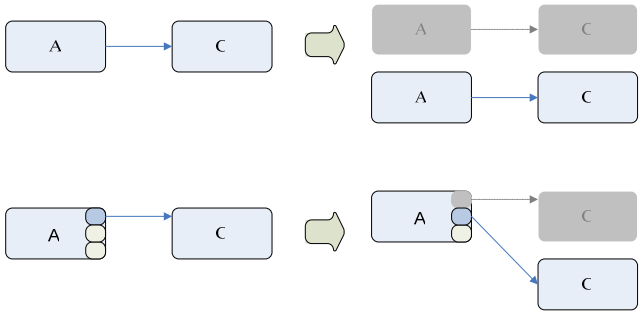
4.2.1 Physical Pointers and Block Layout

Nor 快閃記憶體的 out-place 更新特性，使得實體指標更新傳遞(pointer update propagation)會產生 wandering list 效應，實體指標更新情況失去控制，簡單的一個指標更新卻造成大量的重新寫入。Out-place 更新為快閃記憶體無法更改的特性，因此必需接受指標更新會傳遞的事實，所以此篇論文將針對指標更新時的空間管理和防堵指標更新傳遞兩方面著手。

4.2.1.1 Object layout in blocks

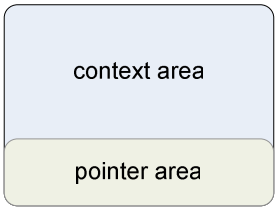
因為 nor 快閃記憶體為 bitwise 的 write once 設計，若在每一個物件(object)中預留指標空間(spare pointer)給指標更新時使用，便可以適度的降低指標傳遞的狀況。如圖九所示，上圖為沒有 spare 指標的情況，在 node C 被更新或搬移後，參照到 node C 的 node A 也必需被更新

而重新寫入；而下圖為 node A 中預留了 spare 指標，所以在 node C 被更新或搬移後，僅需將指標更新在預留的指標空間上，而不需將 node A 整個物件重新寫入於其它地方，如此便可將指標更新傳遞阻擋在 node A。



圖九　spare pointers can block the pointer update propagation

如何才能有效的分配且有效利用預留指標空間(spare pointer)？物件有 hot 和 cold 之分，hot 的物件必定使參照到它的物件之預留指標空間較快使用完，而參照到 cold 物件的物件之預留指標空間將較少被使用，且不能被其它物件所使用，形成有些物件使用預留指標空間不足，但有些物件存有將不被自己或被它人使用的指標預留空間，進而造成預留指標空間使用不平均且使用效能低落的情況，所以此篇論文提出預留指標空間共用的觀念。如圖十，將一個區塊內所有物件的本文區和所有物件的預留指標區分離。因此，當一個物件需要更新其指標參照時，可以向一個區塊中的指標區域(pointer area)索取，而對於參照到 cold 物件的物件來說，不會霸佔該物件將不使用的預留指標空間，藉此達到預留指標空間共用的目的。

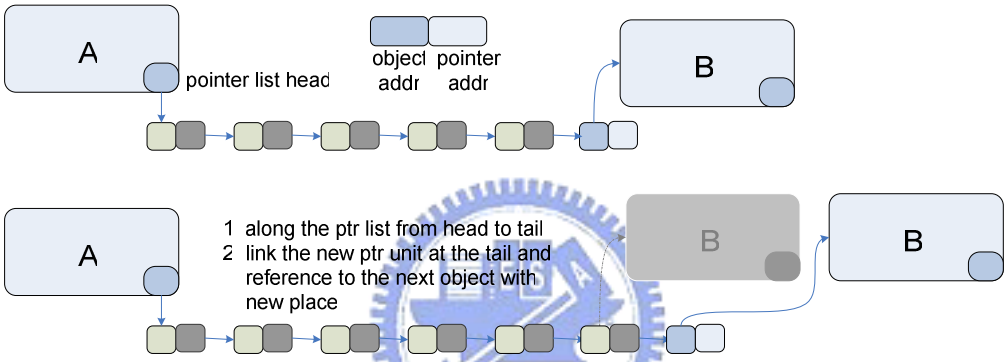


圖十　區塊分成 context area 和 pointer area

4.2.1.2　Pointer-update mechanism

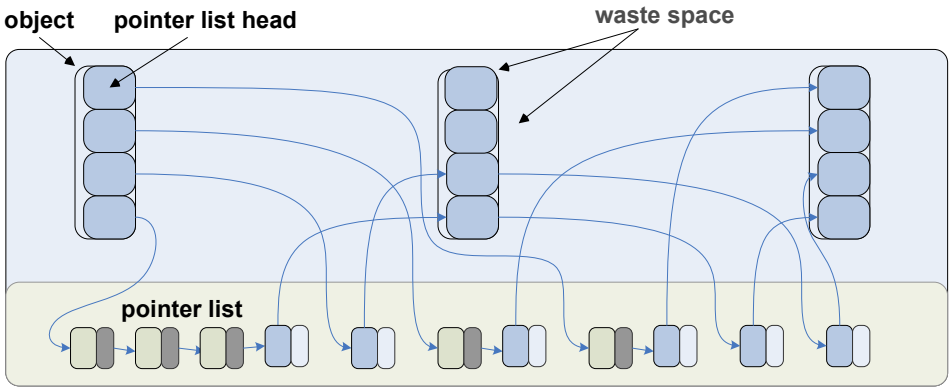
在如此區塊空間的分配下，配合 out-place 更新產生的指標更新傳遞問題，指標的更新動作將與以往的在傳統儲存體上的直接寫入(over write)更新不同。nor 快閃記憶體擁有 bitwise write-once 特性，因此在物件本文區僅存有指標串列(pointer list)的頭端參照，而一個指標單位設計也將不同於已往。依圖十一所示，對於物件內的一個實體指標

參照，僅儲存指標串列的頭端參照；而一個指標單元是由兩個部份所組成，第一部份是物件的指標，用來參照下一個物件真正的位址，第二部份是指標單元的指標，用來連接新的指標單元所用。而一個物件指標更新的行為，修正為依照物件內的指標串列頭端參照找到所代表的指標串列，而後在該串列尾端加入一個新的指標單元，參照到更新或搬移後的下一個物件。指標串列的尾端才是真正有用的指標單元，指標串列中間部份的指標單元僅為連接所用，而指標串列的長度可視為此區塊在被垃圾回收前該物件指標被更新的情形。



圖十一 快閃記憶體上指標更新的動作。

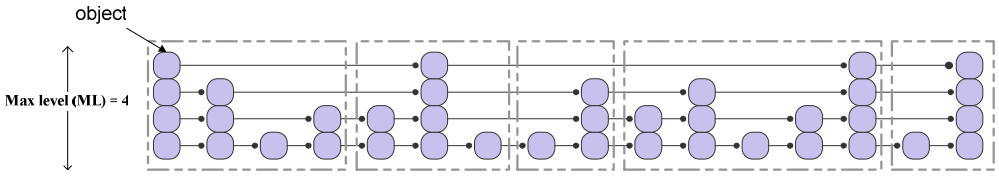
透過如此的設計，可以依照需求來索取適用的指標單元，將更有效率的使用預留指標空間。圖十二顯示 skip list 資料結構中前後節點空間在如此分配下連接的情況。因為每一個節點的高度不同，所以每一個 node 的記錄的指標串列頭端亦不同，使用的指標串列個數也有所差別。圖中顯示在儲存體上用 skip list 需要付出指標串列頭端空間，每個物件為固定大小，而指標高度為隨機決定的，在給定最高高度的條件下，未達到最高高度的物件會浪費些許指標串列頭端，這在未來的工作(future work)中會有所提及。



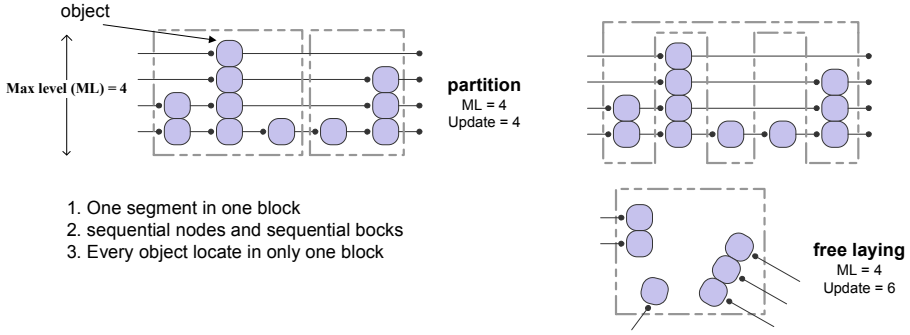
圖十二 區塊裡物件的放置和指標的使用狀況

4.2.2 Garbage Collection

垃圾回收機制是要將區塊中被無用資料(dead data)佔據的空間回收，然而因為 out-place 更新的物理特性，在抹除該區塊前必需將有用的資料先行搬到其它地方，造成參照到這些有用資料的其它物件必需跟著更新而重新寫入，問題成因始於實體指標參照所引起的指標更新傳遞。因此為了防堵指標更新傳遞造成的大量物件的更新/重新寫入，對 skip list 的資料結構在 nor 快閃記憶體上的擺放，將採用部分分割(partition)的方式來加以限制垃圾回收時資料搬移造成的指標更新的範圍。部份分割即是將 skip list 上的所有節點，依資料的 hot/cold 成度將所有節點分成不同區段(segment)，每一個區段由連續的節點組成，且每一個區段放進一個區塊中。圖十三為部分分割的示意圖。一個區塊為一個分割單位大小，該區塊儲存著 skip list 上的一個段(segment)，而該段裡為連續任意個數的節點，每一個段裡的節點個數是依該段裡的資料是趨向 hot 或 cold 來分配。圖十四說明了在相同 skip list 的串列下，部分分割在應用實體指標做垃圾回收時所帶來的優勢；左圖為採用部分分割的方式，若是要對後面那個區塊採取垃圾回收策略，只需付出 skip list 指標最高高度次數的重新寫入；圖中最高指標高度為 4，所以最多需要四個指標更新而重新寫入。而右圖顯示，若將 skip list 的節點隨意擺放，在對該區塊做垃圾回收策略時將會使得指標更新超過 skip list 的最高高度次數；圖中顯示需更新的指標個數為 6 個，較最高指標高度 4 個還要多。

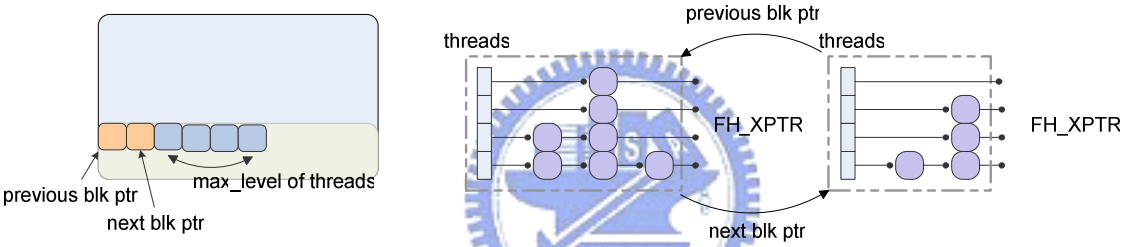


圖十三　部份分割示意圖



圖十四　部份分割將指標更新傳遞侷限在最大高度指標的高度；若是隨意擺放，將會造成大於最高指標高度的指標更新傳遞問題

部分分割策略(partition)有效的限制了垃圾回收時搬移有用資料(valid data)所造成的指標更新傳遞，若再將區塊加以規劃，可以使指標更新傳遞所帶的效應降至更低。圖十五呈現此篇論文採用的區塊規劃，右圖為真實在 nor 快閃記憶體上的空間配置，而左圖為邏輯的示意圖。由左圖的示意圖可知 threads 為一個 block 的起始點，其高度為 skip list 的最高高度，主要的功用在間接承接上一個區塊中的搜尋高度，依照目前高度往後搜尋，並可以有效的阻擋前後連接區塊中物件指標的參照；圖十三中橫跨各個區塊間的指標參照，將以特定數值 FH_XPTR 填充，代表此指標參照為接續下一個區塊中同等高度 threads 的指標參照；而前後區塊的連接關係僅有 next 區塊指標參照和 parent 區塊指標參照完成。



圖十五　區塊實際上的空間分配和邏輯示意圖

由以上的區塊規劃可知，當一個區塊被選為垃圾回收的對像而做資料的搬移後，在圖十三中原本skip list中參照到該區塊內物件的前一個區塊裡的物件指標，將不需被更新而重新寫入。指標傳遞的問題將只會發生在垃圾回收後新區塊與前後區塊間的連結，而連接前後區塊的指標參照的預留指標空間使用，如同物件間指標參照的使用方式，也是向指標區域索取空的指標單元，串成一個因為更新而增長的指標串列。圖十六顯示垃圾回收機制完成前後的情況。垃圾回收時,依循skip list的link可以很快的搬移有用的資料，而回收被無用資料佔據的空間；圖中也顯示出，垃圾回收所造成的指標更新傳遞僅關係到前後區塊的parent blk ptr和next blk ptr這兩個指標(圖十五)。

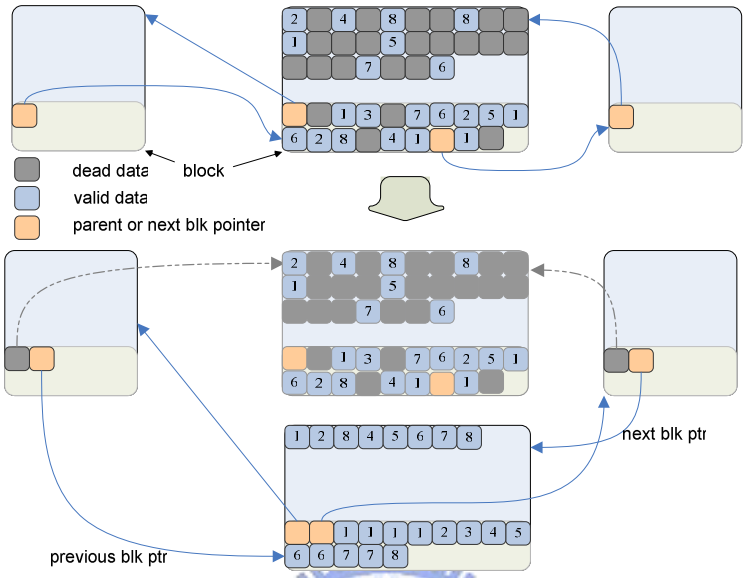


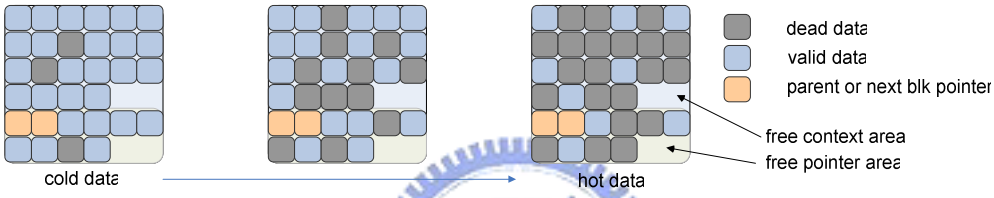
圖 十六　垃圾回收動作

4.2.3　Split and Merge action of Partition

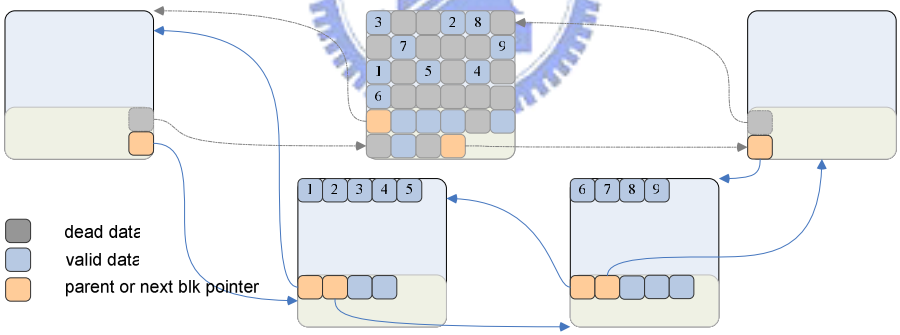
採用部份分割的方式可以降低垃圾回收時的指標傳遞，卻侷限了一個物件的擺放。在 skip list 中每個節點有其連續性，部份分割將 skip list 分成連續節點組成的區段，而每一個區段存在一個區塊中，造成一節點僅能放入單一區塊或區段的現象。若想在一個區段的中間加入一個新的物件，而此時該區塊呈現滿載的狀態，即沒有空餘的空間可供插入一個新物件，而不能滿足此一新增物件的動作。因此，採用部份分割策略的同時，需有額外的分裂(split)和合併(merge)動作來空出一區段的空間或是合併兩個區段騰出一個乾淨的區塊。分裂動作是將一個區段分成二個區段，分別放入二個不同的區塊當中，如此便可以使呈現滿載的區段，因為分成二個區段而騰出空間。合併動作在防止系統中沒有乾淨的區塊供系統使用，但快閃記憶體上仍有空間，只是有些區塊被較小的區段給佔據；因此合併這些區塊即可獲得乾淨區塊給系統使用。但若是在區塊沒有空間供插入時才做分裂動作，或在系統沒有乾淨區塊時才做合併動作，不僅太過於被動且不能依資料的特性來分配區塊上的資料，使得快閃記憶體有更好的空間利用。因此，文中將依資料的 hot/cold 特性，採用分裂或合併動作來分配一個區塊中的資料，即是依資料的特性，來決定該區塊是否該被分裂或合併。

資料依照被更新的頻率可分為常被更新的 hot data 和不常被更新的 cold data 二類。透過降低某區塊的空間利用度方式，讓被 hot data 駐足的區塊可以有較多的空間提供 out-place 更新；而對 cold data 駐足

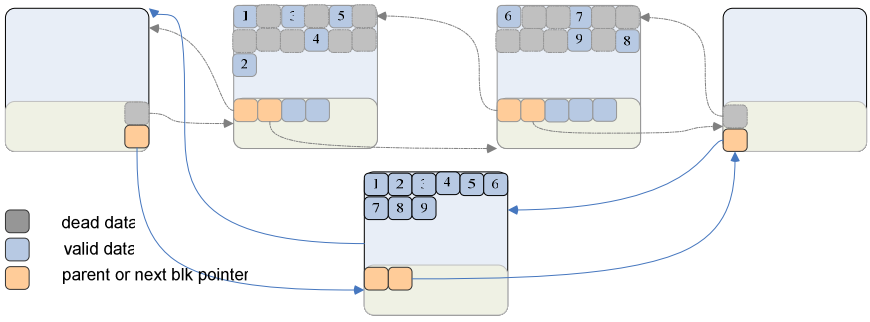
的區塊則提升其空間利用度。如此可平均各區段(segment)的垃圾回收次數和減少總合的垃圾回數次數，進而縮小垃圾回收資料搬移的指標傳遞，如圖十七。分裂(split)和合併(merge)動作即是依據如此的資料特性來降低或提高各別區段(segment)的利用度，平均各別區塊的垃圾回收次數。圖十八和圖十九分別說明了分裂動作和合併動作如何降低和提升一個區塊的利用度。圖十八呈現 split 將一個區塊裡有用的資料搬移到另兩個乾淨的區塊，接著在更新前後區塊指標的連結，降低一個區塊的空間利用度；而圖十九則是將兩個少量有用資料的區塊合併一個區塊，如此可提高一個區塊的空間利用度。



圖十七 區塊利用度依照資料的 hot\cold 來分配，hot 資料駐足的區塊利用度高，cold 資料駐足的區塊利用度低

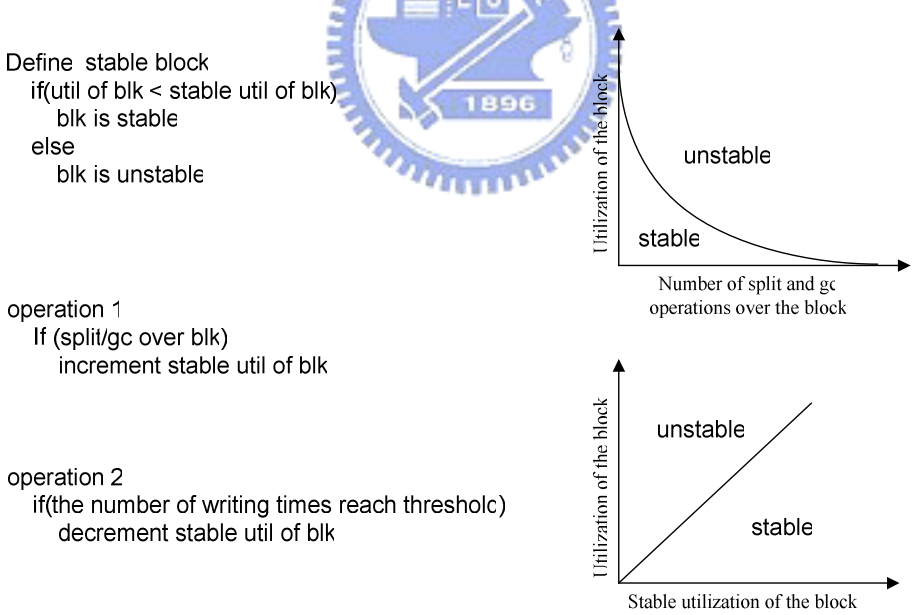


圖十八　分裂動作。將一個區塊裡的資料搬移到不同的二個區塊而降低空間利用度。



圖十九　合併動作。將二個區塊合併成一個區塊而提高空間利用度

如何辨別一個區塊中裡的資料是較 hot 而需給予較低的空間利用度或屬於較 cold 的資料而要給予較高的空間利用度？在文中使用區塊利用度(Block stability)概念來加以判別。一個區塊的區塊利用度 (block stability) 代表一個區塊在目前當下最適當理想的空間利用度。當一個區塊目前的空間利用度大於其當下的區塊利用度(block stability)，則稱此區塊為穩定區塊(stable block)，反之則稱為不穩定區塊(un-stable block)。系統啟動時即將每一個區塊的區塊利用度設定為 1，隨著系統資料的存取將動態的調整區塊的區塊利用度——當分裂或垃圾回收一個區塊時即將其區塊利用度降低，而在一段老化寫入次數後，將各個區塊的區塊利用度調高。其中老化寫入次數，即是系統定義的寫入次數常數，當寫入次數達到此一常數時，便將個區塊的利用度調高，用來使那些在這段寫入次數中不常被分裂或垃圾回收的區塊，呈現較高的區塊穩定度。圖二十上圖顯示分裂(split)或垃圾回收動作對區塊利用度動態更動的情況，區塊利用度將隨著分裂或垃圾回收次數成指數方式降低，而圖二十下圖說明區塊利用度(block stability)和區塊空間利用的關係。

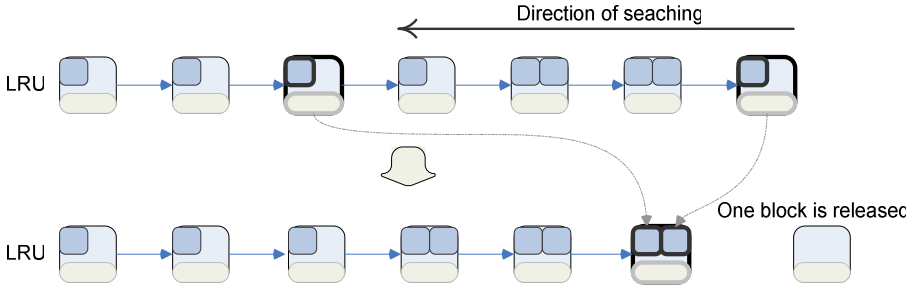


圖二十　區塊穩定度和區塊利用度、分裂次數的示意圖和行為策略。

分裂(split)為在部分分割的策略下將區塊裡的有用資料分配在搬移目標的兩個區塊中，可以減少單一區塊裡有用資料(valid data)的個數，降低一個區塊裡的空間利用度；垃圾回收動作可以挪出一個區塊中無用的資料(dead data)空間，不會更動一個區塊裡的空間利用度。因此區塊在空間不足時採取的動作分為在穩定狀態和不穩定狀態兩種。一個區塊呈現不穩定狀態，代表該區的區塊利用度小於目前區塊的空間利用

度，表示目前的區塊利用度不符合區塊中資料 hot 程度的需求，因此對此區塊採取分裂(split)的動作，以降低該區塊的空間利用度；相反的，一個區塊呈現為穩定狀態，代表區塊的穩定狀態利用度大於區塊的空間利用度，目前區塊空間不足是因為長期的少量更新所致，僅需要挪出該區塊被無用資料佔據(dead data)的空間，對其作垃圾回收動作即可。

很明顯的，合併動作與分裂動作相對，目的是為了收集被 cold 資料駐足的區塊空間，將多個區塊合併成一個區塊，因為駐足在上的資料不常被更新，所以可以合併區塊來提高區塊的空間利用度，進而釋出乾淨區塊為分裂或垃圾回收時所用。挑選合併區塊的方式著眼於那些被 cold 資料駐足的區塊，也就是不常被分裂的區塊，所以實作上採用 LRU 的方式；假設 LRU 頭端存放最近常被分裂的區塊，而尾端即為最近最不常被分裂的區塊；按照資料的 hot/cold 特性，較 hot 的資料所駐足的區塊因為較不穩定而易被分裂，會被放於 LRU 的頭端，反之存有 cold 資料的區塊因為穩定而不常被分裂，所以放於 LRU 的尾端；因此，挑選那一個區塊為被合併的對像即可從 LRU 的尾端往前搜尋，找出可合併且合併後亦處於穩定狀態的區塊，如圖二十一所示。使用 LRU 除了在挑選目前較 cold 的資料簡單外，亦可防止剛被分裂的區塊馬上被合併，使得前一次分裂動作為一個無意義的舉動。而在合併後檢查是否處於穩定狀態，是為了要防止合併後若是區塊處於不穩定狀態而很有可能馬上被分裂，會造成此一合併動作的浪費。



圖二十一　使用 lru 輔助合併動作時區塊的挑選

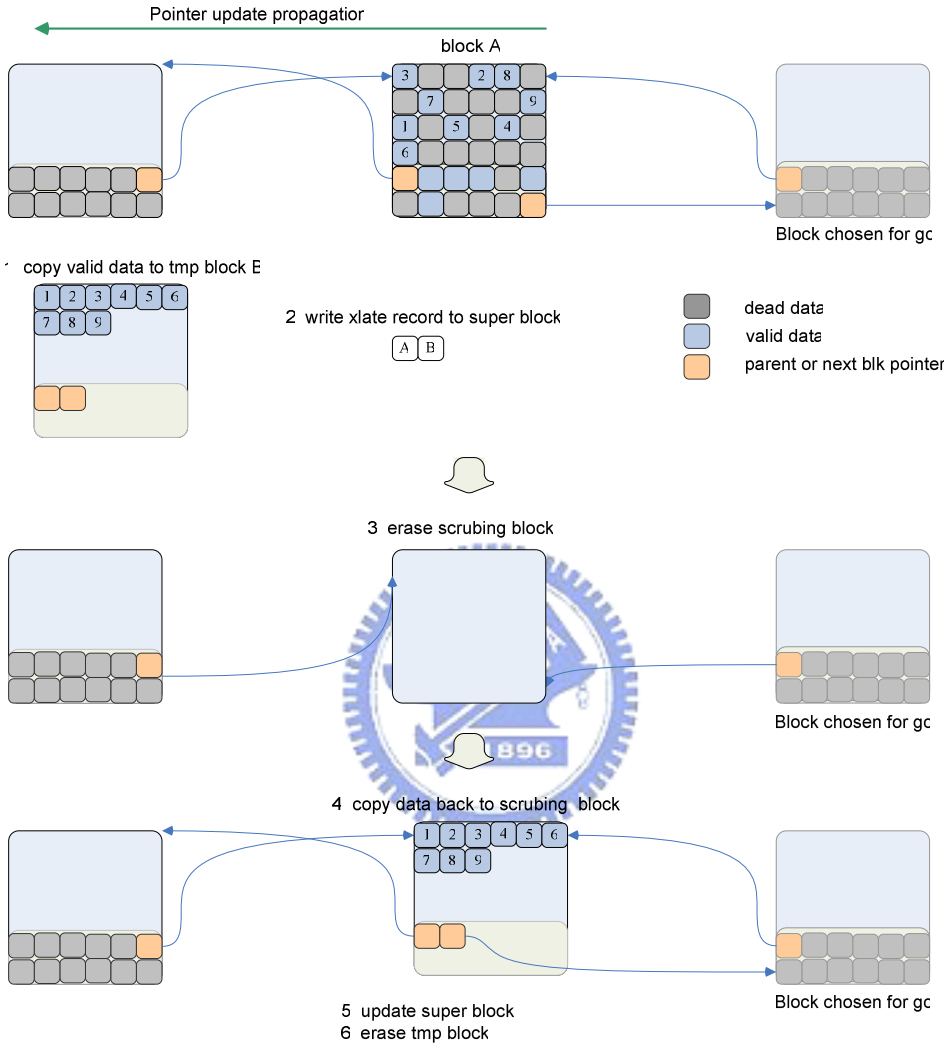
實體指標參照在快閃記體上的應用，讓垃圾回收策略和以往在邏輯指標參照下的情況大不相同。文中使用部分分割的方式大量的降低垃圾回收時搬移有用資料(valid data)造成的指標更新傳遞，加以設計過後的區塊配置將指標傳遞侷限在連接前後區塊的兩個指標；透過分析區塊穩定度的行為而適當的分裂或合併區塊來動態調整區塊的空間利用度，進而平均每一個區塊或段的垃圾回收次數。

4.2.4　Pointer Scrubbing

透過部分分割的方式，已經將指標傳遞的問題大量的降為連結前後二個區塊的兩個指標參照（圖十五的next blk ptr和previous bk ptr ），雖然這兩個指標參照與該區塊中物件的指標參照皆共用該區塊裡的指標區域(pointer area)，然而也有可能一連串區塊中的指標區域皆用完的最壞情況。形成一連串因為空間不足而造成的指標更新，大量的分裂、垃圾回收或合併動作；或甚至找不到可供合併的區塊，不能釋放可用的乾淨區塊(free block)讓垃圾回收使用，使系統形成死結(garbage-collection dead lock)導致系統當機。

針對如此的狀況文中採取的策略為pointer scrubbing。因為out-place更新的關係，物件重新寫入在別的地方才會造成指標更新傳遞(pointer update propagation)；若將一個因為指標空間不足而會傳遞指標更新的區塊直接抹除(erase)並將原本的資料寫入該區塊中，如此參照到該區塊裡物件的其它區塊物件，它們的指標將不必被更新，可以有效的阻擋指標參照的傳遞，進而解決指標更新傳遞所造成的一連串分裂/合併的動作，也可克服最糟糕狀況的垃圾回收死結(garbage-collection dead lock)。

直接抹除再寫入的動作，為一個危險的行為。在抹除但尚未寫入時，若系統因為外在因素而關機，將造成重要資料的遺失。所以pointer scrubbing提供良好的保護機制，以下為此篇論文所提的方法。如圖二十二，首先會將被選為pointer scrub的區塊A的有用資料存入一個特定為pointer scrubbing動作保留的區塊B中，而後在超級區塊(super block)裡記錄區塊A和B的轉換關係，接著便可抹除(erase)被選為作pointer scrub的區塊，再來便可把原本的資料由區塊B搬回區塊A，更新超級區塊(super block)，最後抹除為pointer scrub預留的區塊B，以供往後使用。因此，在抹除原本區塊A的同時，若發生系統無法預估的狀況而斷電，因為超級區塊中存有對應表的記錄，所以可依循對應表的記錄由區塊B中回復區塊A。有關啟動時收集足夠資料的超級區塊(super block)將於後面5.4 File System mount章節解說。



圖二十二 pointer scrubbing

明顯的，指標傳遞問題僅有在區塊搬移的狀況才會發生，而連接前後區塊的previous和next指標與一個區塊中的其它物件共用指標區域(pointer area)。垃圾回收發生的機率很小，加上有效的管理指標下，指標傳遞機乎很少才會發生，而pointer scrub僅有在最糟糕的情況下才需使用到。對平均抹除區塊次數來說，為pointer scrubbing預留的乾淨區塊(spare block)並不會造成抹除多次而提早損毀的問題，且可以有效的解決一連串分裂/合併動作和垃圾回收死結的情況。

5 A Skip-List-Based File System

上文中呈現一個在nor快閃記憶體上有效的索引結構，處理了有關實體指標在out-place更新下所帶來的議題，因此設計一套檔案系統將不需

直接面對實體指標和out-place更新帶來的不便，而是直接堆疊在此索引結構上即可。接下來即要介紹如何在nor快閃記憶體的skip list上疊架一個檔案系統。

在索引結構上設計檔案系統第一個遇到的問題即是命名空間(namespace)的設計，用以提供一般的檔案系統存取使用，文中將命名空間隱含在索引結構的key-coding中。在這方面的研究有檔案系統reiserfs[15]和jffs3，兩者皆使用B^+tree的索引結構。還有建構在資料庫上提供ACID(atomicity, consistency, isolation, and durability)檔案系統的Amino[15]。

而在元件啟動時收集資料的問題方面，元件啟動速度在行動/可攜性裝置中愈來愈被重視，後文中將對快速啟動元件和元件的啟動資訊的儲存可能造成區塊快速損毀問題，使用sticky list的策略來解決。

5.1 Metadata Objects and Data Objects

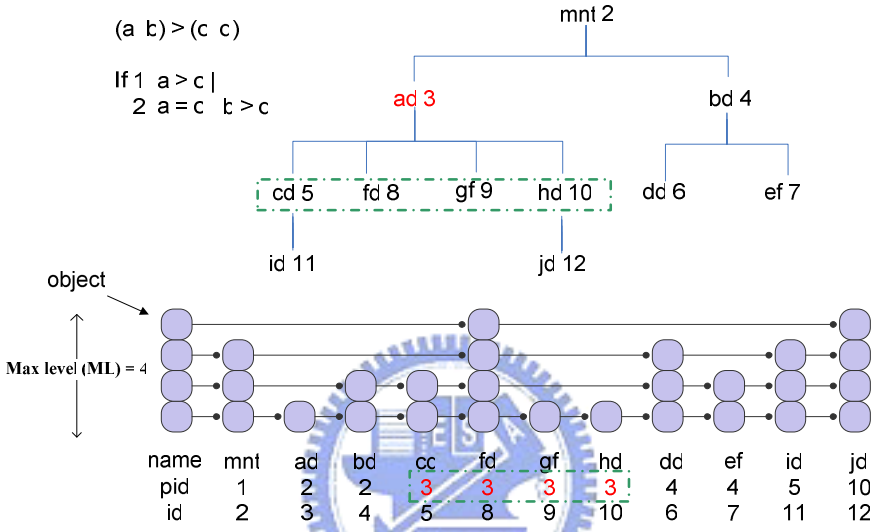
針對一個當案系統中需存有維持目錄架構的資訊和真實的檔案資料，文中將採用兩種基本的物件－metadata物件和data物件。metadata物件用來架構整個檔案系統的命名空間(namespace)，提供目錄和檔案的階層關係，為使用者平常所熟知的檔案系統提供基本操作；而每一個data物件皆屬於檔案的一部分，一個檔案的全部資料即由多個屬於該檔案的全部data物件依前後關係排列組成。其中metadata物件尚可細分為與目錄有關的目錄物件(directory object)和檔案系統最底層的檔案物件(file object)。

5.2 Key-coding scheme

在定義檔案系統的命名空間中有那些物件後，便開始思索著在選定的索引結構skip list中，如何提供階層目錄關係和定址一個物件，以下為此篇論文的key-coding方案(key-coding scheme)。

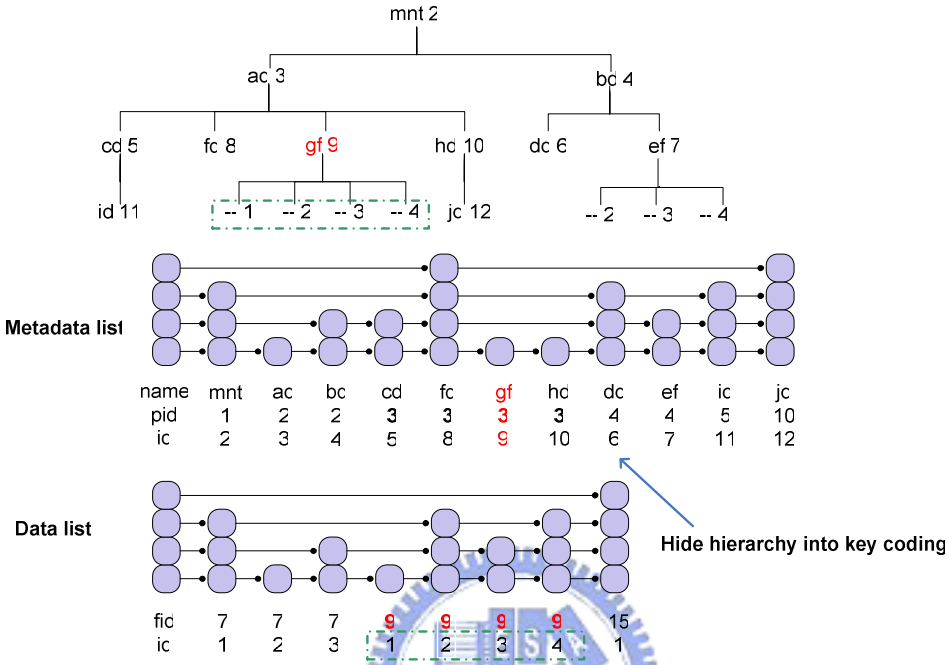
對於檔案系統中每個物件，皆會給予一個獨一無二的識別號碼(id)。在metadata物件的key-coding方案方面，一個名稱為name的物件，便是用該物件的parent物件識別碼(pid)和該物件的name為其key-coding的組成元件，形成(pid, name)的索引鍵值，而將階層架構隱含在其中，如圖二十三。健值比較大小如圖二十三左上圖的公式；首先比較pid的大小，pid較大的鍵值代表鍵值較大；若pid相同時，則比較name的大小，name較大的鍵值其鍵值較大。如此，skip list即可依照鍵值由

小到大的順序排列。比較圖二十三目錄樹和skip list的排列可以發現，ad目錄下的cd、fd、hd目錄和gf檔案，存在以ad識別碼為其pid的子串列中。因此，隱含目錄階層架構於上面所述的key-coding方案，便是由這些子串列構成。



圖二十三 目錄結構隱藏於 skip list 的 key-coding 中

在儲存真實檔案資料的data物件方面，key-coding的長相大致與metadata object相同，不同的是引領data物件子串列的pid必為所屬檔案的識別碼，文中以fid加以區別，而name部分則由代表資料順序的片塊識別碼(chunk id)取代，形成(fid, cid)為索引鍵值的組合。除了在key-coding方案組成不同外，metadata物件在當案存取方面屬於較hot的資料，而data物件則屬於較code的資料，且metadata物件的資料量較data物件的資料量小的多，因此系統中將存在二種skip list，一個存有管理目錄階層架構的metadata物件，另一個存有帶有真實資料的data物件；圖二十四為加入data object的檔案系統完整架構；一樣的，屬於該檔案的所有data object皆存在檔案識別碼(fid)所引領的子串列中。

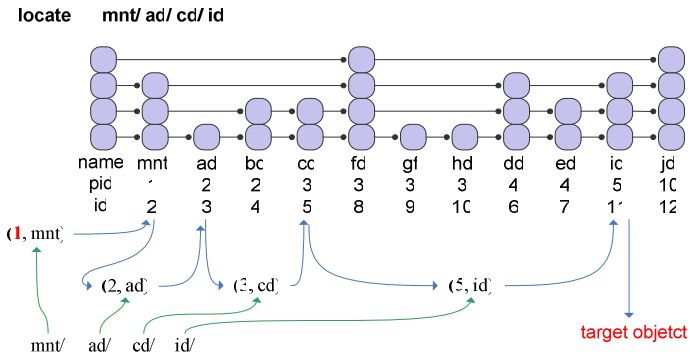


圖二十四　目錄結構與檔案資料分別由metadata list和data list來維護。

5.3　File Operations and Directory Operations

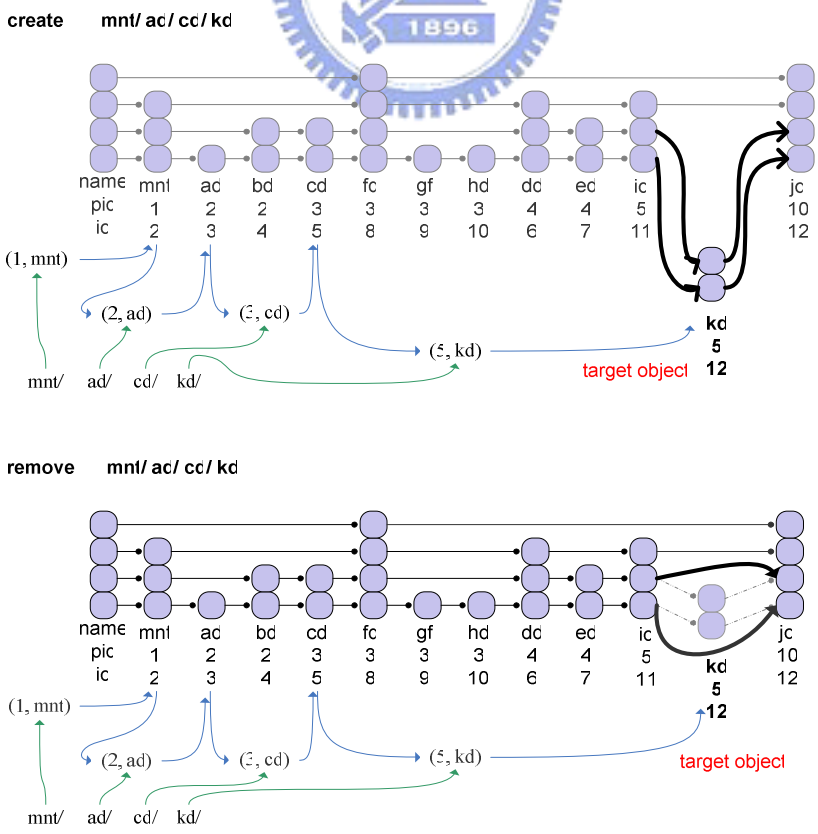
有了完整的檔案物件和物件的組成資料結構後，接下來為此篇論文在如此架構下的檔案系統操作。由文中上述可知，metadata物件和data物件存在不同的skip list中，因此目錄階層的管理操作對像和檔案的讀寫操作對像也分別動作於這兩個不同的skip list上。

目錄架構管理基本操作方面，不外乎定址(locate)目錄/檔案、建立目錄/檔案、刪除目錄/檔案和列舉(enumerate)，對應管理目錄的metadata物件來說，即是在對應的skip list上作定址、建立、刪除和列印子串列的metadata物件動作。首先來介紹最基本的動作－定址(locate)。作業系統上對一個檔案的定址，即是在對檔案系統的結構性資料作一連串的搜尋(search)動作。圖二十五為作業系統定址一個絕對路徑位於mnt/ad/cd/id的目錄操作情形，作業系由該檔案系統的根目錄(id = 1)開始，逐一搜尋絕對路徑的每一個部份。首先搜尋mnt目錄，其pid為1，形成(1, mnt)的索引鍵值；在讀取mnt的物件後可知其id為2，而搜尋ad目錄便可由鍵值(2, ad)達成，重覆如此的搜尋動作，最後便可搜尋並讀取到目錄id。



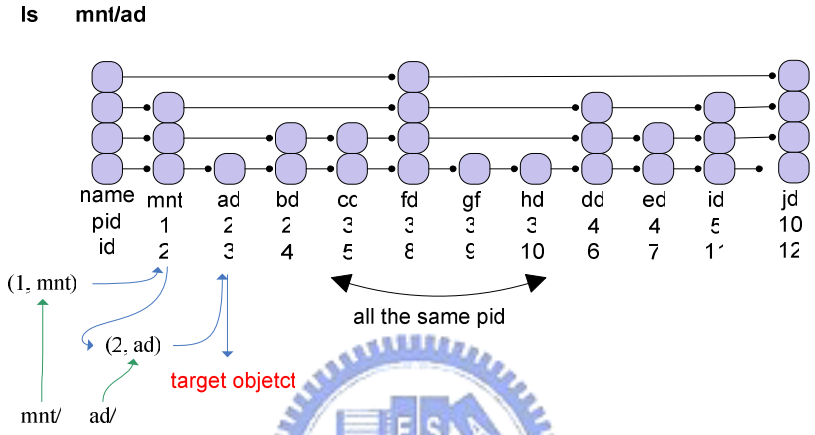
圖二十五　定址一個物件的動作。

對應建立目錄/檔案動作和刪除目錄/檔案動作的metadata物件建立/刪除動作方面，在定址動作完成後，僅是索引結構skip list上的建立物件和刪除物件動作。如圖二十六上圖，在索引到物件kd的位置後，僅是一個skip list的插入動作，即可完成一個建立目錄/檔案的動作；下圖則說明刪除目錄/檔案是在定址kd物件後，更動指標刪除metadata物件即可。



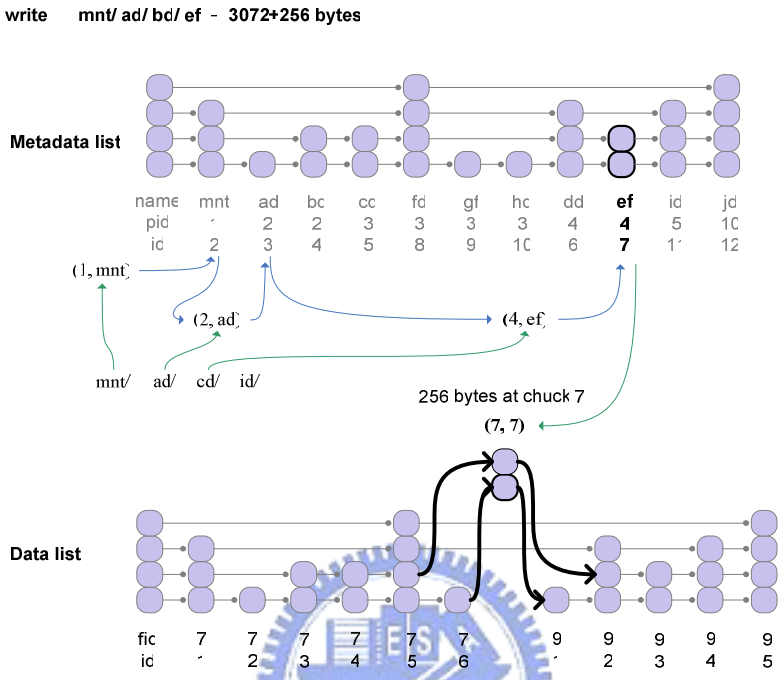
圖二十六　建立或刪除一個物件的動作。

而一個目錄下的列舉動即是定址該目錄在skip list上的metadata物件後，列印出其所引領的子串列即可。圖二十七顯示，在一個目錄下的所有子目錄或是檔案，皆有同一個pid所引領，因此僅需循著此一子串列即可完成列舉的動作。



圖二十七　列舉動作

在讀案資料的讀取、寫入和刪除方面，即是分別對應data物件在skip list上定址、建立和刪除的動作，與metadata物件操作大同小異。以下以一個檔案的寫入略作說明。如圖二十八，對絕對路徑為mnt/ad/bd/ef的檔案末端寫入256位元組，該檔案原本大小為3072位元組，假設一個資料片塊(data chunk)的大小為512位二組，可知新寫入檔案ef末端的256位元組位於第7個資料片塊。因此在作業系統定址檔案ef且取得ef檔案的識別碼為fid後，即可組成待插入data物件的新鍵值(7, 7)，以此鍵值將該資料片塊(data chunk)插入data物件的skip list(data list)中。



圖二十八　對一個檔案寫入資料。

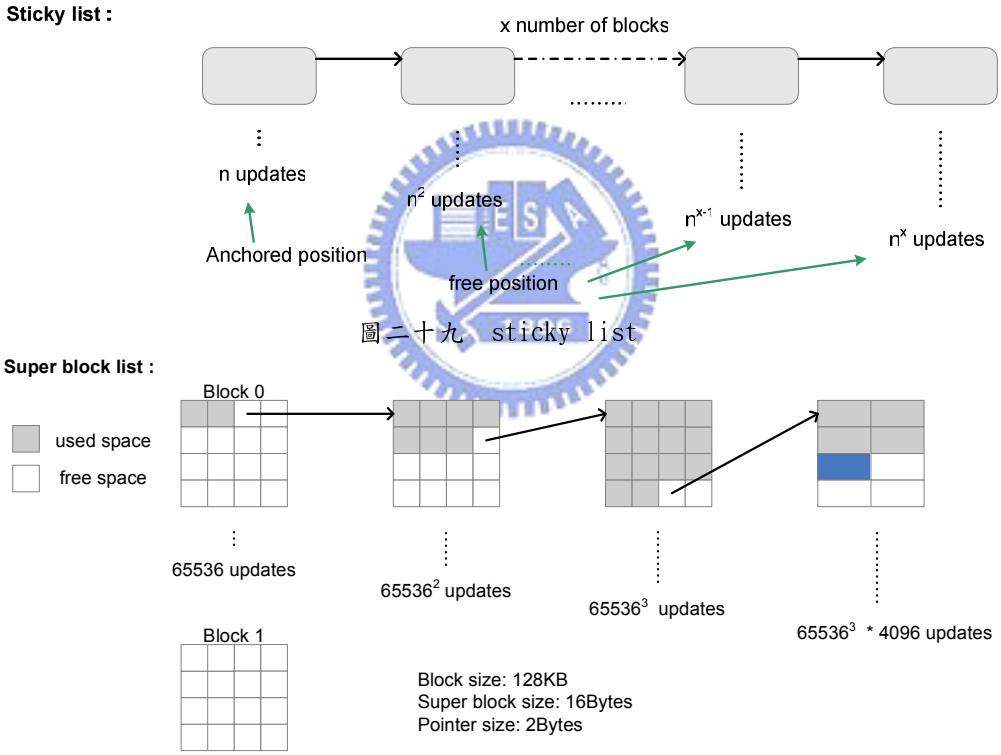
5.4　File System mount

隨著行動/可攜性裝置提供的功能愈來愈多元，每一個元件都需花部份時間初始化，造成總啟動時間過長而考驗使用者的耐心。由圖三可知使用邏輯位址的方式帶進邏輯到實體位址的轉換表且必需收集相當量的檔案系統初始資訊，導致系統開機必需掃描整個快閃記憶體而增加系統開機的時間。可是若將轉換表或這些系統資訊儲存在特定區塊以便快速存取，雖然解決開機速度的問題，但這轉換表或是系統資訊皆為常被更新的hot資料，會造成這些特定區塊快速的磨損。明顯的，待解決的目標即是提供快速開機的同時並不會快速磨損特定區塊。

文中使用一種稱之為sticky list的技術，概念上跟預留指標空間相去不遠。如圖二十九，每一個區塊可提供n次的更新，因此在sticky list的根區塊(root block)在被抹除前，長度為x的sticky list需經過n^x次的更新；一個足夠長的sticky list，將可有效減緩在固定位置的根區塊抹除次數。而文中使用檔案系統存有開機時需要資料的超級區塊，便是用如同sticky list方式的超級區塊串列(super block list)。如圖三十中128M的NOR快閃記憶體為例，將超級區塊的根區塊固定在區塊0的位址，在一個區塊大小為128K，指標大小為2bytes，系統開機資訊所需的超級區塊為32bytes，一個區塊將可提供65536次指標更新與4096次超級區塊

的更新，而圖中四個區塊使得在抹除超級串列的根區塊前，將可提供 65536^3 * 4096次的超級區塊更新。

因此在系統開機時，僅需依循超級區塊串列即可找到開機時所需的資訊，提供快速收尋開機資訊的同時，不會造成特定區塊(根區塊，區塊0)的快速抹除而損毀。在文中的檔案系統中，也將預留一個區塊(區塊1)，以確保超級區塊串列的根區塊在達到更新上限而需被抹除時，有固定的額外區塊可供替換，而不會造成系統沒有乾淨區塊(free block)但需更新超級串列根區塊的系統死結狀況。



圖二十九 sticky list

圖三十 super block list

6 Evaluation

此篇論文在實作進度方面，僅完成索引結構的實作，而檔案系統的設計方面，僅有在 linux-kernel 中以 ram-resident 的 skip list 驗證其可行性，但尚未疊架在快閃記憶體的索引結構之上，因此在測試結果方面，僅就索引結構來做探討。

6.1 Experimental Setup and Performance Metrics

實驗的方式大致是在不同的空間配置下插入一連串的物件後，對插入的物件做一連串常態分佈的讀取或者更新的動作後，統計個別操作在該組實驗中次數和時間進而分析其行為。

目前實驗是模擬一個 128M 的 nor 快閃記憶體，設定在此快閃記憶體上的區塊大小為 128k，所以共有 1024 個區塊；讀取一個位元組的時間為 80ns，寫入一個位元組的時間為 9000ns 而抹除一個區塊的時間為 $7*10^8$ ns 供實驗計算執行時間的標準。表二為實驗中更動的參數，顯示在不同的最大指標高度(max level)下，物件大小隨著指標高度愈高而愈大，在物件個數和指標個數比例為 2 的前提下，各空間的分配情況。物件大小(object size)會隨著最大指標高度愈大而增大，一個指標串列頭端需要 2bytes;在快閃記憶體上的空間管理是使用 bitmap 的方式，因此 context area 和 pointer area 皆有一個對應的 bitmap(ctt bitmap 和 ptr bitmap)來管理個區域空間物件的使用。因為一個物件會佔用 2 位元的 ctt bitmap 空間而一個指標單元僅佔用 1 位元的 ptr bitmap 空間，由兩著空間大小約為 1:1 可知物件個數和指標個數比例約為 2。而實驗中將使用到的常態分佈設定為 mean=5000、sdev=500 且 range 為 1 到 20000 的鍵值分佈，而老化寫入次數(number of age write)為 10000 次。

variable / Max Level	2	3	4	5	6	7	8	9	10
object size	72	74	76	78	80	82	84	86	88
ctt bitmap	406	396	386	378	370	362	354	346	338
context area	116928	117216	117344	117936	118400	118736	118944	119024	118976
ptr bitmap	416	407	404	386	372	362	356	354	356
pointer area	13312	13024	12928	12352	11904	11584	11392	11328	11392
remain	10	29	10	20	26	28	26	20	10

表二　在指標個數和物件個數的比例為 2 的條件下，不同指標最大高度使得物件大小不同而造成的空間分配情形。

以下為四個實驗設計的方法和參數說明：

1.系統指標讀取負擔(system pointer reading overhead)分析:

在做實驗各部份的統計時發現，指標讀取的次數異常的高，想探究 skip list 操作過程中，除了 skip list 本身和區塊串列必定造成一定的指標讀取外，是否設計的方法中還有其它會造成額外指標讀取的因素。

固定參數：常態分佈(mean=5000，sdev=500，range=1~20000)，

ptr/ctt=2，老化寫入次數(number of age write)=10000。

變化參數:分別對機率為 0.1、0.3 和 0.5 三種機率在指標最大階層由 2~10 間的 27 組配對組合做實驗。

工作量(workload)：插入 20000 筆循序的鍵值的物件，並作常態分佈的 20000 次讀取、10000 更新後 10000 次讀取和單純 20000 次更新的三組操作。

評估準則(metrics)：比較 thread 的讀取次數和 skip list 讀取次數在這 27 組實驗的分佈。

在此將 thread 的讀取次數拿出來比較，是因為在實驗中發現 thread 在系統指標讀取次數方面佔有很大的部份，透過此統計可以分析出 thread 對整體系統有何影響。

2.機率和階層的分析：

高階層會使高階層指標的機率小，帶來大幅度的跳躍，然而階層太高，造成的極低機率，在系統中可能永遠都用不到如此高的階層而沒有增進效能，相反的可能增加系統負擔。但若是提供低機率但階層高度不夠，會使得系統跳躍密度不佳，讓搜尋近於線性搜尋。所以文中針對五種不同的機率(0.5~0.1)和 10 種不同的階層高度做測試，觀查讀取指標、讀取物件、寫入指標、寫入物件和抹除區塊的次數和時間的關係。

固定參數：常態分佈(mean=5000，sdev=500，range=1~20000)，ptr/ctt=2，老化寫入次數(number of age write)=10000。

工作量：插入 20000 筆循序排列鍵值的物件，對五種不同的機率指標和階層的組合({0.5, 10}, {0.4, 10}, {0.3, 7}, {0.2, 6}, {0.1, 4})分別做常態分佈的 20000 次讀取、10000 更新後 10000 次讀取和單純 20000 次更新。

評估準則：統計所有指標讀取、物件讀取、指標寫入、物件寫入和區塊抹除的次數與時間，觀查在不同的機率和階層下，那一組配對有較好的效能，並權衡機率和指標高度在效能上所佔的比重。

3.一個區塊中預留指標個數和物件個數的比例評估：

一個區塊滿載而需要分裂(split)或垃圾回收的原因有物件區域使用完或預留指標區塊沒有空間，因此試著找出在如何預留指標個數和物件個數的比例下，可以讓分裂次數和垃圾回收次數的總和（抹除次數）最低。

固定參數：常態分佈(mean=5000，sdev=500，range=1~20000)，(probability, level)=(0.2, 6)，老化寫入次數(number of age write)=10000。

工作量：插入 20000 筆循序排列鍵值的物件，觀查在常態分佈更新下,13 種預留指標個數和物件個數的比例(ptr/ctt = 2.5, 2.4, 2.3, 2.2, 2.1, 2, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3)對區塊抹除次數的影響。表三顯示在此十三組比例下的空間分配情形。

variable / ptt over ctt	2.5	2.4	2.3	2.2	2.1	2	1.9	1.8	1.7	1.6	1.5	1.4	1.3
ctt bitmap	360	362	364	366	368	370	370	372	374	376	378	380	382
context area	115200	115840	116480	117120	117760	118400	118400	119040	119680	120320	120960	121600	122240
ptr bitmap	470	450	431	411	392	372	372	353	333	314	294	275	256
pointer area	15040	14400	13792	13152	12544	11904	11904	11296	10656	10048	9408	8800	8192
remain	2	20	5	23	8	26	26	11	29	14	32	17	2

表三　在十三組指標和物件比例下各空間的分配情形

評估準則：在不同的機率和階層組合下，量測各別的區塊抹除次數。

4.區塊穩定度(block stability)策略對系統 hot/code 資料的分佈影響：

上文中帶入區塊穩定度的概念，此測試在老化次數 (number of age write)10000 次下，區塊穩定度在仲裁分裂動作在 hot/cold 資料區分上的表現。

固定參數：常態分佈(mean=5000，sdev=500，range=1~20000)，(probability, level)=(0.2, 6)，老化寫入次數(number of age write)=10000。

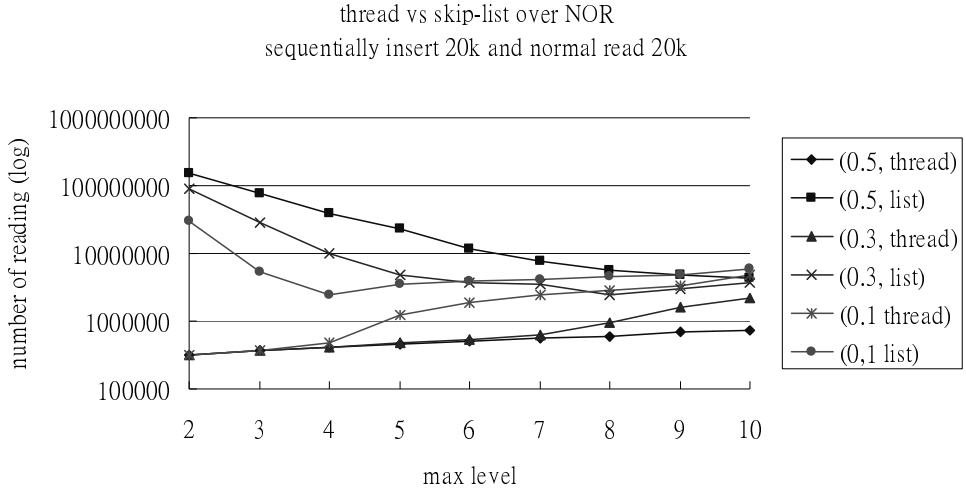
工作量:插入 20000 筆循序排列鍵值的物件，10000 筆常態分佈(mean = 5000, stdev = 500)的資料更新。

評估準則：在如此的工作量下的區塊空間利用度和區塊穩定度的關係。

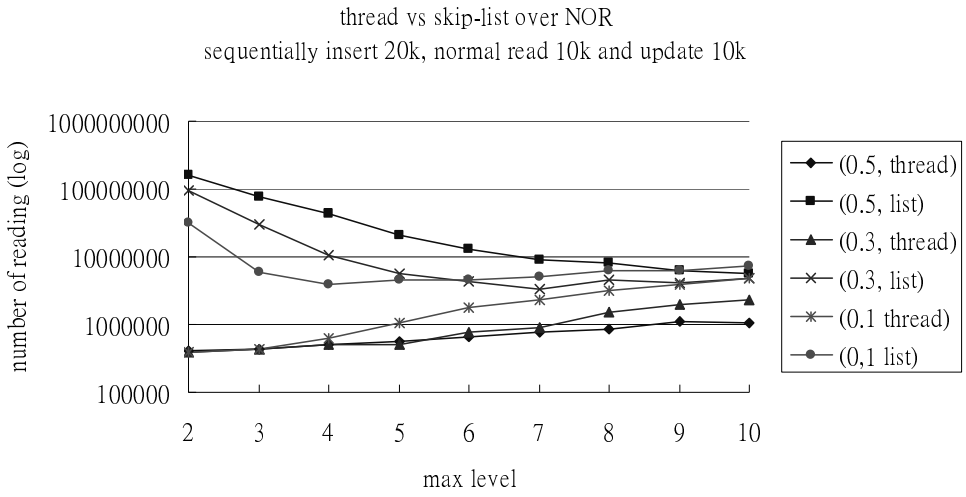
6.2 系統指標讀取負擔(overhead)分析

圖三十一、三十二、三十三分別表示在 20000 次讀取、10000 次更新後 10000 次讀取和 20000 次更新的 thread 讀取次數和一般 skip list 指標讀取次數的分佈圖。由圖三十一中所示，在縱軸為讀取次數，橫軸為階層高度下，指標階層愈高而產生的跳躍距離愈遠，所以使得指標讀取次數下降，如機率為 0.5 的 list 指標讀取次數。然而若是機率低(0.3，0.1)，因為高階層指標發生的機率太低而跟本使用不到，使得 skip list 由高階層向下搜尋時，多次的無效指標讀取，因此使得 list 讀取次數向上提升，而機率 0.1 較機率 0.3 更早發生讀取次數向上提升的趨勢。在 thread 讀取次數部份，在三種機率下階是隨著階層高度愈大而向上增長，而每條 thread 讀取曲線皆有一段大幅向上增長的情況，是因為此階層的 thread 較少被使用到，而可能產生一連串 thread 的指標讀取直到 skip list 的尾端。而後每增加一層即增加一定個數的無效 thread 讀取次數，而使線條趨於固定斜率。由圖可知，愈高指標階層會使得 thread 讀取次數愈接近一般的 list 讀取次數。

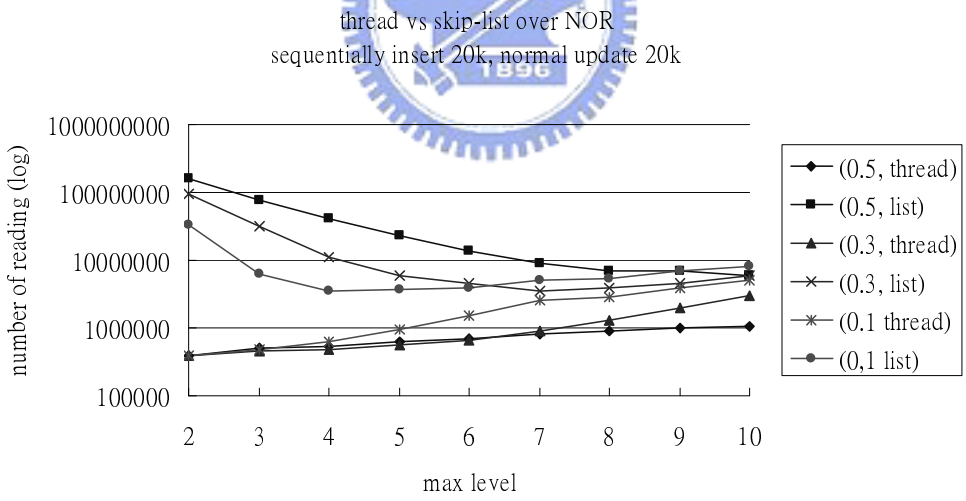
圖三十二與圖三十三的成長趨勢如同圖三十一，各圖中的各曲線隨著更新次數的增加而呈現向上微調的關係。因為更新次數愈多，使得指標串列愈長，而使得讀取指標次數些微增長。



圖三十一　skip list 上 20k 循序寫入和 20K 讀取，在各機率和各階層中一般指標讀取和 thread 指標讀取的趨勢圖。

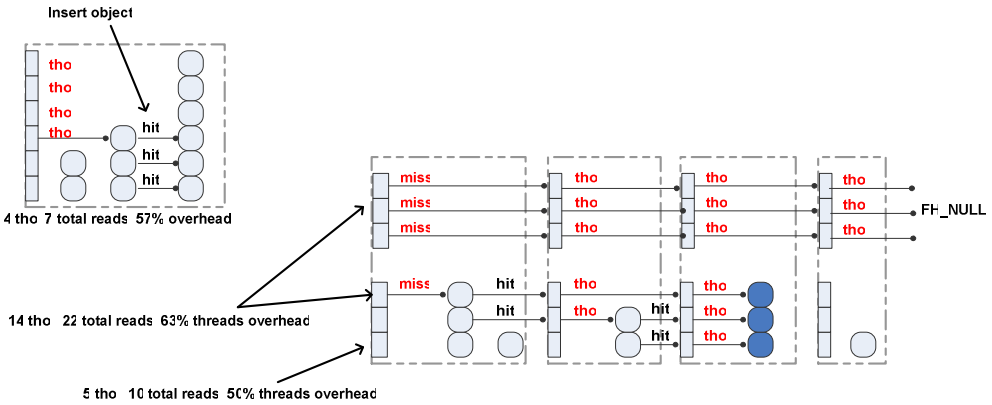


圖三十二　skip list 上 20k 循序寫入、10K 讀取和 10k 更新，在各機率和各階層中一般指標讀取和 thread 指標讀取的趨勢圖。



圖三十三　skip list 上 20k 循序寫入和 20k 更新，在各機率和各階層中一般指標讀取和 thread 指標讀取的趨勢圖。

為什麼 thread 會造成如此大的負擔？如圖三十四顯示，左上圖說明單就一個區塊裡的搜尋就有 4 個 threads 負擔，在全部 7 個讀取指標動作下佔了 57%的比例；而右圖顯示，threads 在無用且高的指標階層所造成的影响將更顯著，因為 threads 有阻擋且延續指標的作用，所以當一個指向空指標的高階層，會使得一個 threads 的搜尋變得冗長，且需要到最後一個區塊才會停止。因此 threads 所造成的效能負擔是未來需要努力的一個目標。

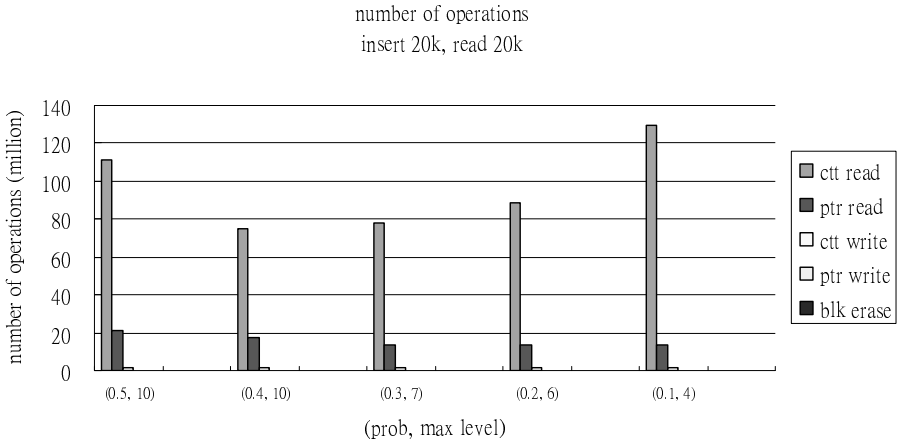


圖三十四 thread 讀取佔系統讀取次數的分析

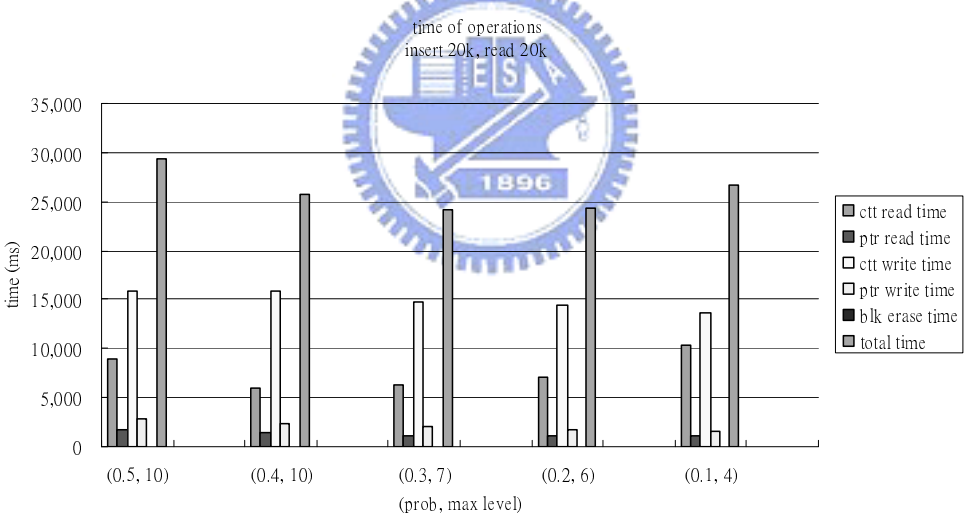
6.3 機率和階層的分析

圖三十五和圖三十六分別表示系統 20000 次插入後，20000 次讀取下指標和物件的讀取、寫入和區塊抹除情形，圖的縱軸為各讀取和寫入動作的次數或時間，橫軸為各機率和階層配對的數組。圖三十五顯示，(0.4, 10)配對有較好的物件讀取次數，是因為各配對在提供大致相同的最高機率下，第二高階層將影響到搜尋的次數，所以機率 0.4 在階層 9 的條件下，還可以提供不錯的跳躍；然而階層 0.1 在階層 3 時則跳躍力大減，所以才會有高機率且高階層有較少的物件讀取次數的情況。而(0.5, 9)配對因為在高機率(0.5)的前提下，階層 9 無法提供好的跳躍，所以讀取次數仍偏高。而在讀取指標方面，因為 thread 的影響，使得較高階層有較多的指標讀取；而在此實驗中，物件的寫入皆為兩萬次，但在寫入指標方面較少階層會有較少的指標寫入次數。

在時間關係方面，如圖三十六，指標階層高低會影響到物件的大小。讀取物件的時間明顯是由(0.4, 10)配對佔優勢，雖然讀取物件較大，但讀取次數少太多；然而在寫入物件時間方面，因為在快閃記憶體中寫入一個位元組需 9000ns，較讀取一個位元組需 80ns 來說所耗的時間大太多，所以雖然(0.4, 10)和(0.3, 7)物件大小僅差了六個位元組，但在寫入物件時間方面還是多出許多。圖中呈現寫入物件和寫入指標所花的時間隨著階層高度愈低呈現向下的趨勢。在寫入物件和指標為另一個影響因素下，數組(0.3, 7)有較好的讀取效能。



圖三十五　skip list 上 20k 循序寫入和 20k 讀取後，各組機率階層配對的次數分佈圖。

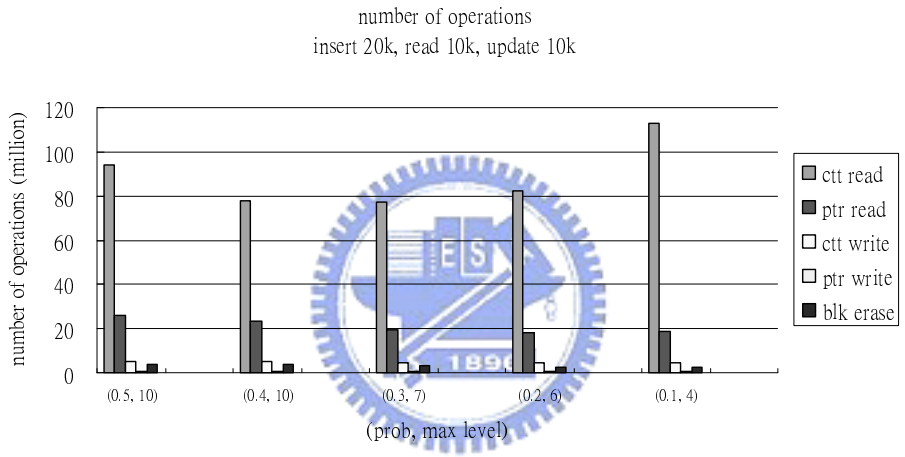


圖三十六　skip list 上 20k 循序寫入和 20k 讀取後，各組機率階層配對的時間分佈圖。

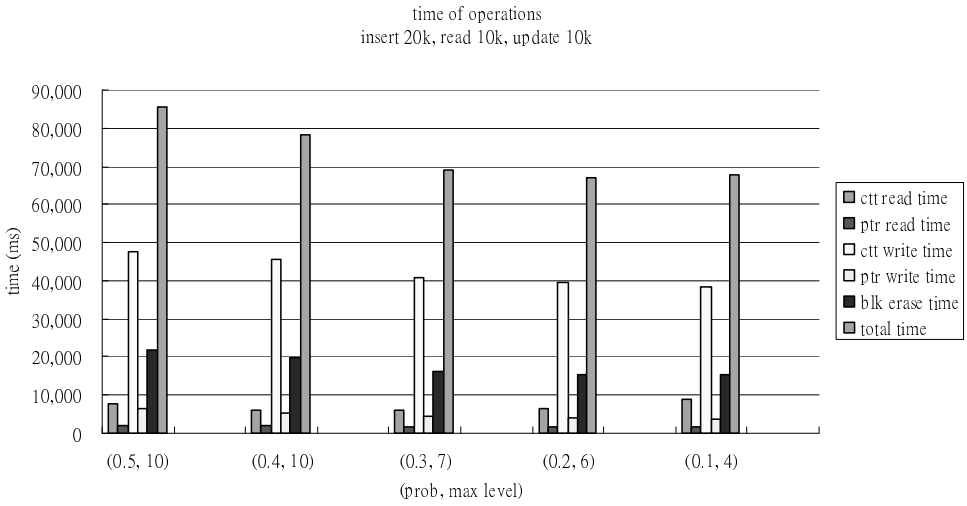
圖三十七和圖三十八為 10000 次更新後 10000 次讀取的分佈圖，和上述一樣，縱軸為操作的次數和操作時間，橫軸為機率和階層的配對。因為 10000 次更新產生的分裂和垃圾回收動作，增加了各數組的物件讀取次數和寫入次數。與圖三十五相較，圖三十七有較高的物件的讀取和寫入次數。因為較高機率和較高指標階層的配對使得指標空間使用的較快，且指標階層高會使得物件大小較大，所以物件空間亦較少，相較於低機率和低階層的配對，高機率和高指標的數組因為空間不足，會有較多次的區塊抹除次數而產生較多次的物件讀取和寫入，因此在這組實驗

中可看出，因為區塊抹除次數的關係(0.3, 7)數組有較(0.4, 10)數組少量的物件讀取次數。而(0.3, 7)數組相對於(0.2, 6)數組為較高機率和較高階層的組合，所以物件讀取較少。

在時間分析上，由圖三十八可看出，區塊抹除次數在低機率和低指標階層較少，所以高機率高階層在區塊抹除方面會消耗較多的時間。因此在高機率和高階層的配對中，讀取次數不在有那麼大的優勢，而區塊抹除造成的額外寫入，使得此組實驗最好的效能配對為(0.2, 6)。



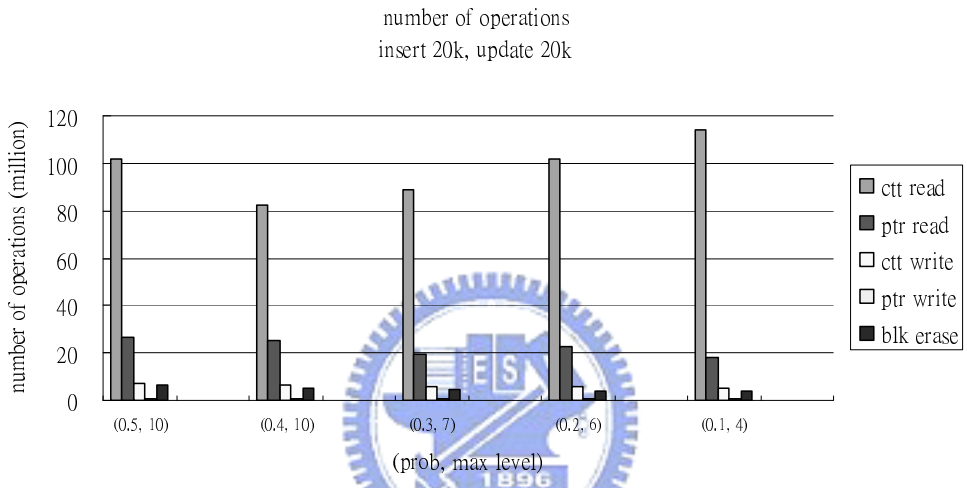
圖三十七　skip list 上 20k 循序寫入、10k 讀取和 10k 更新後，各組機率階層配對的次數分佈圖。



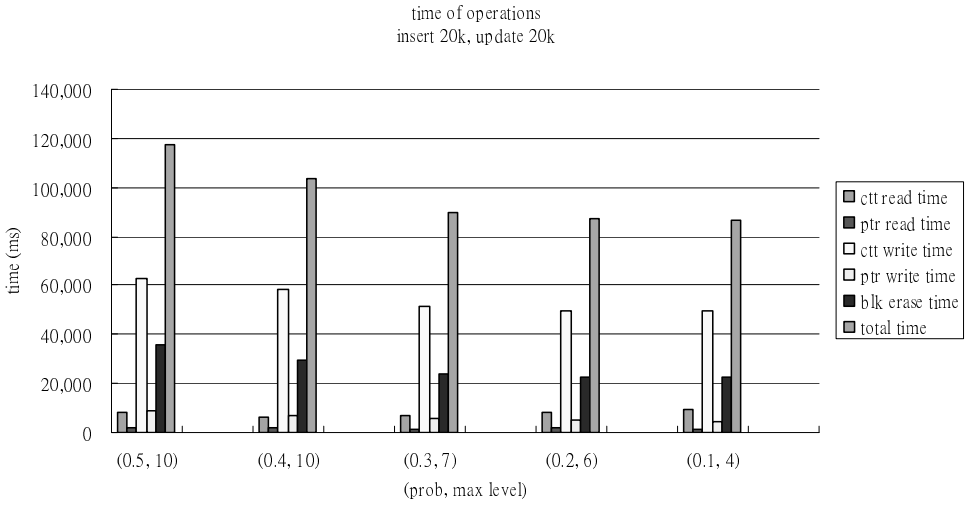
圖三十八　skip list 上 20k 循序寫入、10k 讀取和 10k 更新後，各組機率階層配對的時間分佈圖。

圖三十九和圖四十說明在大量更新的狀況下，將帶來更多的區塊抹

除次數，各操作次數皆呈現大量的增長。一樣的，因為高機率在高指標階層下會有較多的區塊抹除次數，讀取物件次數方面的優勢大大的下降，而在寫入物件次數和寫入指標次數方面皆較低機率低指標階層的數組多，因此在此實驗中還是數組(0.2, 6)有較好的整體執行時間，如圖四十；由圖可知總執行時間數組(0.1, 4)與(0.2, 6)僅有些微差距，可推算在更大量的更新下，數組(0.1, 4)將會有較好的執行總時間。



圖三十九 skip list 上 20k 循序寫入、20k 更新後，各組機率階層配對的次數分佈圖。

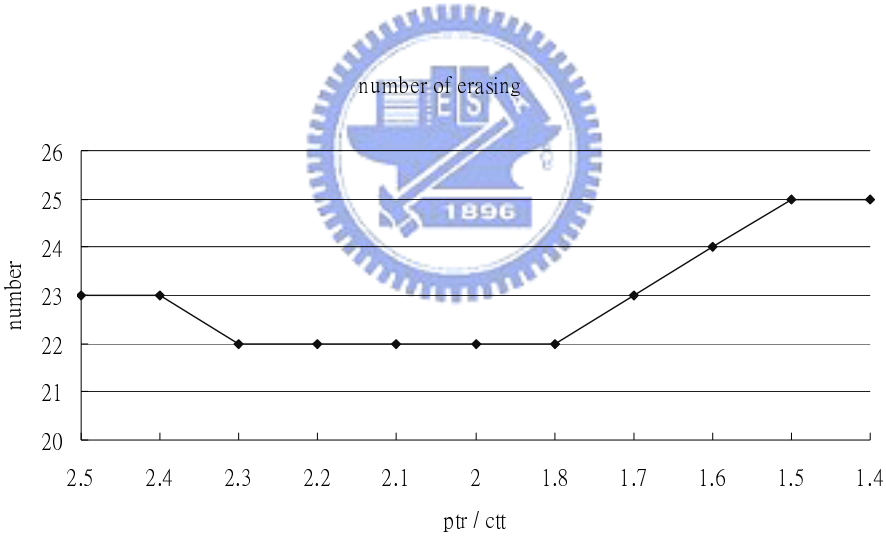


圖四十 skip list 上 20k 循序寫入、20k 更新後，各組機率階層配對的時間分佈圖。

6.4 一個區塊中預留指標個數和物件個數的比例評估

由上述實驗可知，區塊抹除在整體的時間效能佔了一定的影響因素，而造成區塊抹除主要由物件空間不足或指標空間不足兩者其一所觸發，因此實驗針對(0.2, 6)數組比較區塊抹除次數與指標單元個數和物件個數的比例間存在何種關係。

圖四十一為區塊抹次數與物件個數和指標個數的比例關係，縱軸為區塊抹除次數，橫軸為物件個數和指標個數的比例。其中比例的計算方式為指標單元個數除於物件個數。圖中顯示在中間有個谷底狀態，為較少的區塊抹除次數；在圖中左方的指標與物件比例較大，說明物件空間較少，因為物件空間不足而觸發較多次的區塊抹除；而圖中右方指標與物件比例小，所以指標空間較少，由指標空間所解發的區塊抹除次數亦會隨著指標區域愈小而增多。因此物件和指標空間多與不足皆不是好事，實驗中較少的區塊抹除次數落於比例 2.3~1.8 之間



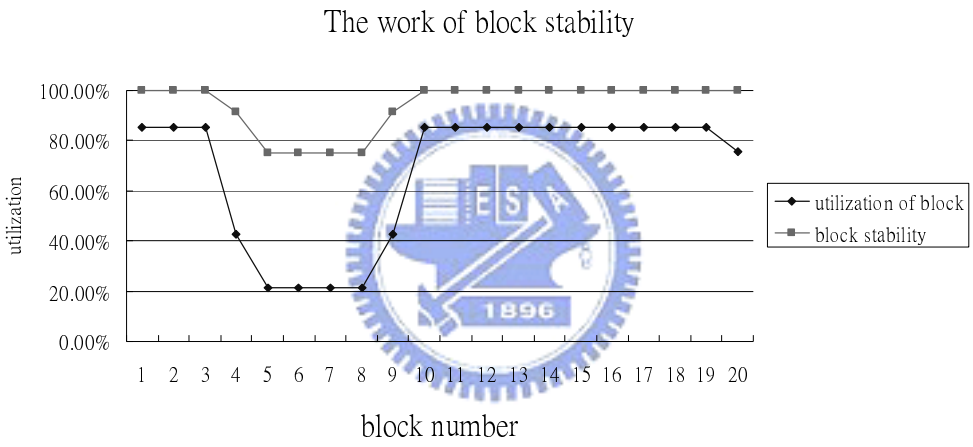
圖四十一　區塊抹次數與 ptr/ctt 比例的關係圖。

6.5　區塊穩定度策略對系統 hot/code 資料的分佈影響

在”4.2.3　Split and Merge action of Partition”章節中，介紹各區塊的區塊穩定度(block stability)會隨著駐足在其上資料的 hot/cold 特性而不同，愈 hot 的資料會使得該區塊的區塊穩定度愈低，希望該區塊的利用度愈低，而有較多的更新空間；而愈 cold 的資料會使得該區塊的區塊穩定度愈高，而呈現區塊有較高的空間利用度。

圖四十二為數組(0.2, 6)在指標單元個數和物件個數比例為 2 的前下，連續插入 20000 個物件後，對其常態分佈(mean=5000, sdev=500, range=1~20000)更新後的分佈後的結果，縱軸為區塊的利用度，橫軸為

區塊編號。由圖中可知，在二十個區塊中，mean 為 5000 的物件在多次常能分佈更新後座落於 6 或 7 的區塊中，因此區塊的區塊穩定度曲線以區塊 6 和 7 為最底，代表愈 hot 資料所駐足的區塊應該要有較少的空間利用度；而曲線顯兩側因為更新次數較少，所以曲線有在兩側較高的現象。圖中區塊的空間利用度曲線，隨著區塊穩定度區線而反應出因資料的 hot/cold 特性所分配的區塊空間利用度。兩曲線之間會有一個最小間距，是因為在連續插入 20000 個物件時，每個區塊僅放 8.5 分滿，所以大約有 15%的差距。而在區塊空間利用度區線的末端有向下的現象，是因為該區塊裡的物件較少，且因為不常被更新所以其區塊穩定度將不會向下調整。



圖四十二　區塊穩定度與區塊利用度呈一定正向關係

7　Conclusion

以往在快閃記憶體上使用邏輯定址的方法，造成儲存體掃描耗費大量時間，且收集的檔案系統資料也佔掉不少記憶體空間。本篇論文使用實體指標來管理資料，然而實體指標在快閃記憶體的 out-place 特性下尚有些問題待克服。首先是更新指標傳遞(update pointer propagation)的問題，簡單的一個更新動作而造成大量的失去控制的一連串更新；垃圾回收死結問題，是因為垃圾回收動作需將區塊中有用的資料搬移到其它區塊，產生的指標更新傳遞消耗系統中更多的乾淨區塊，使得系統乾淨區塊消耗殆盡而造成垃圾回收時沒有乾淨區塊可用的死結狀況；元件啟動時需要收集必要資訊，在不加速持定區塊的損毀條件下，希望可以提供快速啟動的解決方案。

文中使用指標共用的概念將區塊分為物件區域和指標區域，有效的利用區塊空間外，亦可降低指標更新的傳遞。運用 partition 的技巧可使得垃圾回收時因為搬移有用資料而造的更新傳遞降低至最高階層高

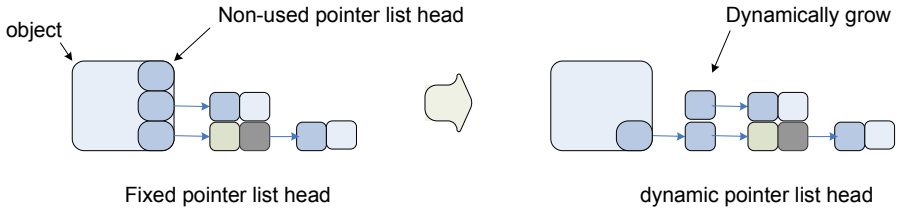
度，使用 thread 的設計可使指標更新關係由各物件中抽離，使得指標更新傳遞在區塊搬移(區塊分裂、合併和垃圾回收)時才會發生。雖然指標傳遞在區塊間發生，但在最壞情況亦會有一連串區塊更新的動作，文中採用 pointer scrubbing 的方式阻擋指標傳遞，亦可間接解決垃圾回收死結。資料有 hot/cold 之分，而 hot 資料較多的資料更新會使得駐足的區塊有較多的垃圾回數次數；為了平均和降低各區塊的垃圾回收次數，文中帶入區塊穩定度(stable stability)的概念，可有效的使區塊利用度依其所駐足資料的 hot/cold 特性而調整，達到平均和降低垃圾回收次數的效果。

在選用 skip list 為本篇論文的索引結構並解決其上實體指標在快閃記憶體的議題後，呈現在其上疊架一個檔案系統是如此的簡單。因為 metadata 物件和 data 物件的大小差異和 hot/cold 特性，系統中會存在二個 skip list。Metadata list 為管理目錄架構所用，而 data list 為真實的檔案資料所用。

文末實驗分別呈現 thread 讀取佔系統指標讀取一定成度的負擔、指標機率和階層配對的關係、物件區域和指標區域的比例對區塊抹除次數的影響和區塊穩定度可以有效的使得區塊依照資料的 hot/cold 程度調配區塊的空間利用度。

8 Future Work

目前系統上對於每個物件中皆需要預留指標串列頭端，在高階層指標的狀態下會浪費一定空間，應該利用指標空間區域的優勢，將階層高度隱涵其中。如圖四十三所示，應該將指標的串列頭端隱藏於指標區域中，如此可更有效的使用空間。而在測試中發現為了避免指標傳遞到多個區塊而引入的 thread 概念，會對系統帶來不小指標讀取次數，因此需要再詳細的評估 threads 帶來的好處和不用 threads 是不是真的會造成大量的指標傳遞。在實驗的過程中發現，較 hot 的資料所駐足的區塊會使用較多的指標空間，而較 cold 的資料使用指標空間較少，因此各區塊的指標區域或許可依系統當時的資料屬性而動態調整其所駐足區塊的物件和指標空間比例而達到更好的空間使用。



圖四十三　將物件中的串列頭端移進指標區域，
可使用動態增長的方式更有效的利用空間。

在完成 nor 快閃記憶體上的 skip list 索引結構且在其上疊架一個檔案系統後，需與目前在 nor 快閃記憶體上的 jffs2 檔案系統比較，探討實體指標解決邏輯指標的開機掃描時間和記憶體大量消耗的問題外，在存取的效能上是否可提供更好或相當的效能。再者可比較 skip list 和 jffs2 在維護檔案各 data node 間的 RB-tree 索引結構，而呈現更客觀的比較與分析。

References

[1]: William Pugh, "Skip lists: a probabilistic alternative to balanced trees," Communications of the ACM archive, Vol 33, Issue 6, 1990

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

[3] M-Systems, Flash-memory Translation Layer for NAND Fash (NFTL).

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

[5] D. Woodhouse, "JFFS: The Journaling Flash File System,"

[6] Artem B. Bityutskiy, "JFFS3 design issues," http://www.linux-mtd.infradead.org/tech/JFFS3design.pdf.

[7] C. Manning and Wookey, "YAFFS Specification," Aleph One Limited, http://www.aleph1.co.uk/node/37, Dec, 2001.

[8] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo, "An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems," Accepted and to appear in ACM Transactions on Embedded Computing Systems.

[9] Chin-Hsien Wu, Li-Pin Chang, and Tei-Wei Kuo, "An Efficient R-Tree Implementation for Flash-Memory Storage Systems," the ACM GIS conference, 2003

[10] Zeinalipour-Yazti, D., Lin, S., Kalogeraki, V., Gunopulos, D., and Najjar, W. 2005. Mi-crohash: An efficient index structure for flash-based sensor devices. In 4th USENIX Conference on File and Storage Technologies (FAST). 31 - 44.

[11] University of Szeged , "JFFS2 improvement project ," http://www.inf.u-szeged.hu/jffs2/mount.php

[12] Keun Soo Yim, Jihong Kim, and Kern Koh, "A Fast Start-Up Technique for Flash Memory Based Computing Systems," the ACM Symposium on Applied Computing (ACM SAC), 2005.

[13] Chin-Hsien Wu, and Tei-Wei Kuo, Li-Pin Chang, "Efficient Initialization and Crash Recovery for Log-based File Systems over Flash Memory," the 21st

ACM Symposium on Applied Computing (ACM SAC), 2006.
[14] Hans Reiser, "ReiserFS file system," http://www.namesys.com/
[15] C. P. Wright and R. Spillane and G. Sivathanu and E. Zadok, "Extending ACID Semantics to the File System," ACM Transactions on Storage, To appear.
[16] Samsung Electronics Company, "K5A3x40YT(B)C 32M Bit (4Mx8/2Mx16) Dual Bank NOR Flash Memory datasheet
[17] MXIC Electronics Company, "MX-29LV160CBTC-90G 16M-BIT [2Mx8/1Mx16] CMOS SINGLE VOLTAGE 3V ONLY FLASH MEMORY" datasheet
[18] Compact Flash Association, "Compact FlashTM 1.4 Specification," 1998.
[19] SSFDC Forum,"SmartMediaTM Specification," 1999.
[20] Li-Pin Chang, Tei-Wei Kuo, "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems," The 8th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2002) September 24 ?27, 2002. San Jose, California.