國立交通大學

資訊科學與工程研究所

碩 士 論 文

基於固態硬碟的韌體演算法與硬體架構快速標準制訂工具

A Rapid Algorithm/Architecture Prototyping Tool for Solid-State Drive

研 究 生：郭晉廷

指導教授：張立平　教授

中 華 民 國 九 十 九 年 六 月

基於固態硬碟的韌體演算法與硬體架構快速標準制訂工具

A Rapid Algorithm/Architecture Prototyping Tool for Solid-State Drive

研 究 生：郭晉廷　　Student：Ching-Ting Kuo

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

June 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年六月

基於固態硬碟的韌體演算法與硬體架構快速標準制訂工具

學生：郭晉廷　　指導教授：張立平

國立交通大學資訊科學與工程研究所

摘要

固態硬碟所採用的快閃記憶體具有獨特的物理特性，不論在韌（軟）體或硬體都需要採用特殊的設計策略。目前不論就硬體架構與資料管理演算法方面，都已經有許多良好的設計方案可供選擇。然而，固態硬碟的整體效能表現，取決於軟體，硬體，以及工作環境三大要素的交互影響。如何在已知的產品定位下尋找最佳的軟硬體組合，以達到最佳的效能成本比，則需要反覆的調整與測試，且為一曠日費時的工作。為了克服此一設計上的課題，本研究計畫開發了一個固態硬碟的快速雛型工具，該工具提供非常豐富的軟硬體設計選擇，並以相當簡易一致化的程式介面呈現。該工具可幫助系統工程師快速地測試不同的軟硬體組合的效能，而可大幅減少系統工程師在調校與測試方面的時間。

關鍵字：固態硬碟，效能模擬，雛型工具

A Rapid Algorithm/Architecture Prototyping Tool for Solid-State Drive

Student：Ching-Ting Kuo **Advisor：Dr. Li-Pin Chang**

Department of Computer and Information Science
National Chiao Tung University

Abstract

Designing high performance solid-state disks is a very challenging task because of the diversity of host applications, complexity of SSD hardware architectures, and firmware algorithms. SSD performance is mainly subject to not only hardware designs but also software algorithms. One practical problem that industry faces is how to combine hardware/software design options for the best performance under a specific niche market. This study introduces a fast hardware-software prototyping tool for SSD design. It features a set of highly simplified programming interfaces and a rich collection of hw/sw design options. This tools aims at reducing the cost of debugging and help to find out the best design without lengthy trial-and-error cycles.

Keywords：Solid-state disks, performance simulation, prototyping tools

誌謝

謝謝交大，謝謝土地公，謝謝你總在奇蹟的一刻回應無神論的我的祈禱。

謝謝張立平老師，謝謝您！聽說過很多嚴格老師帶學生的方式，相比之下我深深感受到您的體貼入微，在您眼中我是個難搞的人吧！時常誤會老師的意思，有時候則是我沒有把想法清楚表達出來，老師應該沒遇過這麼有溝通障礙的學生吧？幸好我是個還算牛的人，總會刻苦耐勞把老師的要求做好，有機會我真的很想跟老師當好朋友，但老師恐怕很害怕讓我讀博班吧？不過我也不想讀啦！我還是沒辦法喜歡上做研究，務實一點對我的簡單頭腦比較沒有負擔，再次謝謝您。

謝謝我的家人，謝謝爸爸，不管我做什麼都給我支持，謝謝媽媽，我可以從眼神中感覺您對我的呵護，謝謝您們對我生活上的一切支援，不管是機車、房子還是電腦，也謝謝妹妹，雖然妳好像沒什麼實質的幫助，但妳的存在就是對我的鼓勵。很抱歉接下來又要在新竹工作，但我保證有時間就會回家去的，謝謝！

謝謝我親愛的同學們，給話很少很悶燒兼戰友再兼未來同事的偉杰，謝謝你陪我經歷過的所有，不管是 wrk、游泳、魔獸、衝浪還是喝酒都讓我印象深刻，給講話總是理直氣壯讓我很想跟你吵架的義勛，謝謝你常常跟我討論研究上的問題，你也畫出我人生另一塊視野，給有貴婦氣質但卻做很多傻事的莉君，我們原先真的可以是很要好的朋友，可能我這個人比較固執吧！總之，祝福妳。最後再次謝謝你們這些朋友，我無法想像人生中沒有你們的存在，以後務必要常相聚。

謝謝 ESSLAB 的前輩，謝謝畢業前後的你們對我的關心，謝謝士庭、明毅、宥全、Show、JJ、小狐狸和學姊，謝謝你們不吝於對我的經驗分享與指導，除了打球、打電動、烤肉，我也會永遠記得每一次的謝師宴。謝謝 ESSLAB 的後輩，謝謝阿誠、小節、玟蕙和 Uma，謝謝你們讓我了解指導別人前要先教育自己的準則，你們經歷很多，而那些也讓我感同身受，花蓮行的美好時光我會一直放在心底，也謝謝碩 0 的阿平、A 導和逸康，謝謝你們，希望我們在未來一起努力。

謝謝很多很多新舊朋友，謝謝成大 GY 團的不離不棄，謝謝創惟夥伴們給的學習機會，謝謝東大同學們的相伴，謝謝其他實驗室的球友，謝謝彰師大朋友們的關心，謝謝 RTOS 修課同學們的支持，謝謝國科會計畫的教授、同學和負責人，謝謝口試委員的指導，謝謝神秘朋友給我的心靈經驗，謝謝很多很多人，你們都是影響這篇論文產生的蝴蝶效應中重要的一環，謝謝！

最後的最後，謝謝我自己，只想對自己說一聲：「辛苦了，幹得好！」

目錄

圖目錄

表目錄

1. INTRODUCTION

近年來由於筆記型電腦、手機、PDA 等可攜式裝置的蓬勃發展，固態硬碟（Solid State Drive，簡稱 SSD）的技術和應用也一再被探討，再者傳統硬碟（Hard Disk）具有低耐震、體積大、壽命短且發出噪音等為人詬病的缺點，因此 SSD 在可攜式裝置上取代傳統硬碟已成為必然的趨勢，同時它所擁有的快速隨機讀寫能力更可以作為高效能需求的伺服器應用，雖然市場需求日益加溫，但 SSD 仍有價格高的困境必須克服，因此如何找到 SSD 成本和效能的平衡點是目前最重要的議題。

如何發揮 SSD 效能的解答在於如何設計它，SSD 是由 NAND 快閃記憶體（Flash Memory）所組成，設計上可以分為硬體和韌體兩部分，硬體包含快閃記憶體晶片（Flash chip）、訊號線（Control line）和資料匯流排（Bus）等，設計上需要考慮各種元件的平行架構與其之間的排列關係，而韌體的必要性是由於 NAND Flash 不同於傳統儲存裝置的各種特性，需要採用特殊的演算法去設計，而不同的演算法在不同的用途中也會表現不一樣的效能，因此充滿變化。綜合而言，SSD 的設計可以分為硬體架構（Hardware Architecture）和韌體演算法（Firmware Algorithm）兩大議題，而兩大議題又彼此有關連，例如不同的硬體架構就需要相對應的韌體演算法支援。

SSD 韌體設計的作用是穩定各類型存取的效能，由於 Flash 具有的物理特性，需要採用特殊的機制去處理，但為了實現這些機制通常會對系統產生額外的負擔，像是使用較多的隨機存取記憶體（RAM）或者花費額外的時間搬移資料，而且這些機制在面對不同的存取行為（Workload）時會表現不同的效能，這些行為包含存取的空間與時間性，各種影響因素像是隨機或循序、密集度、次數和頻率等，因此 SSD 韌體演算法設計的目標就是在不減少系統負擔的前提下，平衡各種用途不同存取行為的效能。

SSD 硬體設計的需求是提升循序存取的效能，由於不同的 Flash chip 可以獨立存取資料，通常會將要求存取的資料（Request）分散到不同的 Chip 處理，因此增加 Flash chip 的數量除了增加容量外更可以增加 SSD 平行處理的效能，但每顆 Chip 是否可以平均分配到 request 就變成影響平行處理效能的關鍵，所以 SSD 硬體設計一般增加的是循序存取的效能，因為循序資料或較大筆的資料可以使 Flash chip 的利用度較高，平行處理的效能就比較明顯，相反的隨機存取的效能則會被限制。

現今產業界在 SSD 的開發過程最常遇到問題有兩點，第一點是“不知道什麼是最好的硬體和韌體組合”，一般的大型系統如資料庫上較多循序、低頻率存

取的資料，且要求回應時間較短，所以必須考量高平行度高效能的硬體架構，小型系統像筆記型電腦等的存取行為則比較複雜，很多隨機、高頻率存取的小檔案，必須透過韌體吸收掉這種存取造成的效能負擔，因此 SSD 在大型系統和小型系統的效能需求就會不一樣，如果把適合大型系統的硬體架構用在小型系統上就會因為平行利用度不足而使得效能發揮不如預期，而用在小型系統的韌體演算法通常會針對高頻率存取的資料做特別處理，若這種演算法用在大型系統上則會因為並沒有這類型資料而產生額外的負擔，於是怎樣的硬體和韌體設計最好？以及怎樣的搭配最適合怎樣的應用？開發人員往往無法選擇。而 SSD 開發的第二個問題就是“過長的產品開發與測試週期”，從規劃硬體架構、撰寫韌體演算法到實際產品測試，往往需要很長一段時間才能得到結果，很多時候發現有錯誤或結果不如預期時，這些步驟就必須重新來一次，因此如何縮短開發週期卻又不降低品質也是開發者想知道的。

本篇針對 NAND Flash 的 SSD 提出一個韌體演算法和硬體架構的標準制定工具（Rapid Algorithm/Architecture Prototyping Tool），簡稱為 RAPT，使用比硬體測試更加快速的軟體技術來設計 SSD，因此不僅有效解決開發週期過長的問題，其定義的標準模型更可以提供各種韌體演算法的概念，幫助開發者找到適合的設計。

RAPT 建立一個仿真的軟體環境，模擬 Flash-based SSD 的韌體和硬體運作。首先，其建立完整的 SSD 架構，包含豐富的硬體元件與指令支援，並結合 System C [1]函式庫增加模擬的效能與準確度，使用者可以很輕易的勾勒出希望的 SSD 架構。其次，RAPT 提出了一套 SSD 韌體演算法的標準模型，這套模型也符合 NAND Flash 的特徵，適用於所有 NAND Flash 的儲存裝置，它清楚點出設計概念與方法，因此可以利用它快速且容易的實現任何一種韌體演算法。最重要的是，RAPT 所建立的硬體和韌體環境都具有 SSD 設計的共通性，因此修改十分方便，換句話說若要由一種架構改成另一種架構或者由演算法 A 改成演算法 B 都不需要對程式做大幅度的更動。

RAPT 的開發目標是成為一套實務且便捷的工具，站在使用者的角度，不需要具備豐富的硬體或電路知識，也不需要熟知各類型的韌體演算法，就可以透過它快速的完成模擬 SSD 的工作。RAPT 首先必須實現各種硬體架構，像傳輸通道（Channel）、通道群（Gang）、通道葉（Interleave）等，為了真實符合現今的 Flash 設計，也提供 Copy-Back 和 Multi-Plane 等優化指令，這些原本複雜的硬體架構或平行關係除了實現之外還要簡化成為一套僅有數個參數的 API，且簡化後要能不失其功能性。其次，RAPT 要建立一組有效的韌體描述語言，韌體演算法的設計包含緩衝區（Buffer）、位址轉換（Address Mapping）、回收（Garbage Collection）等眾多議題，任何一點的變動都可以視為不一樣的演算法對 SSD 產

生不同的結果，而且不同議題的設計也是彼此相關的，很可能因為其中一部分的更動而導致演算法全盤的重新設計，而這些動作可以抽象為 Index、Group 和 Queue 三種模型標準，首先必須將各種設計議題視為平等，才有辦法以一種概念討論所有的設計變化，目的在於即使差異很大的韌體演算法，也可以用相同的設計架構去完成它，現存已知的 SSD 韌體演算法都可以用這三項標準去制定，這樣做的好處是不論設計或修改演算法都可以更加清楚且快速的完成。

一個模擬環境的實作需要考量很多面向，首先是模擬的準確度，隨著參數的調整模擬誤差可能越來越大，因此與真實結果的比較是必須的，但通常實際產品內部的設計方法難以取得，所以要達到準確度的目標通常必須實作真實的硬體。其次是模擬的多樣性，一個工具提供的環境能夠模擬越多類型的設計，用途就越廣泛、價值也越高，但也表示其功能與支援必須越多，在開發與維護上也會花費越多時間。最後是穩定性與效能，需要多少系統資源才能完成某項模擬？模擬的速度又可以多快？這些全都是 RAPT 必須考慮的，因此與真實 SSD 做準確度的比較、提供豐富的硬體或韌體支援還有優化執行效能都是必要的工作。

本篇餘下的章節如下：第二章介紹 SSD 設計的文獻以及目前已發表的軟體模擬工具，第三章說明 RAPT 的韌體開發概念，第四章討論實作以及給予 SSD 的設計支援，第五章探討各種韌體演算法與硬體架構的組合，並實現 RAPT 的驗證，第六章為本篇的結論與未來展望。

2. PROBLEM FORMULATION

2.1 Background

一顆 Flash chip 是由很多區塊（Block）組合而成，每一個區塊包含很多頁（Page）。如圖 1 所示，page 中包含使用者部分（User area）和備用部分（Spare area），User area 就是一般存取資料的區域，而 Spare area 是韌體的保留區通常用來記錄位址轉換或位元錯誤校正等資訊，其中 User area 又可細分多個區段（Sector），sector 是使用者可以存取的最小單位，一個 sector 為 512 byte，而 page 和 block 的大小則隨著設計不同有所變化，普遍來說一個 page 有 4~8 個 sector，而一個 block 有 32~128 個 page。

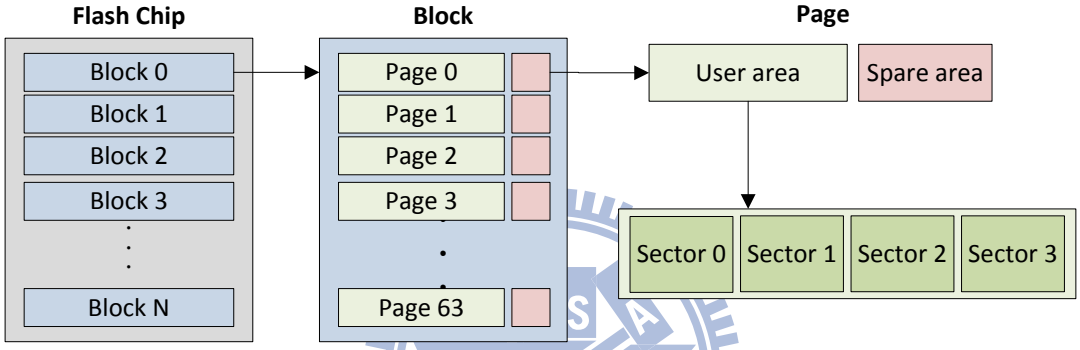


圖 1、快閃記憶體的物理結構

Flash 內部的讀取（Read）和寫入（Write）是以 page 作為單位，不同於傳統硬碟，Flash 採用電子式的運作方式，其無法在寫過的 page 上重複寫入資料，除非經過抹除（Erase）的動作將寫過的 page 清掉才能再次寫入，erase 是以 block 為單位，且經過一定次數的 erase 後，那個 block 就不能再被使用，這種被限制的 erase 次數，我們稱為 Flash chip 的磨損容忍度（Endurance）。

表 1、SLC 與 MLC 比較

	SLC	MLC
Page / Block size	2 KB / 128 KB	4 KB / 512 KB
Read latency	25 μs	60 μs
Write latency	200 μs	800 μs
Erase latency	1500 μs	
Endurance	100 K	5 K

Flash 依製程可以分為 Single Level Cell（SLC）和 Multi Level Cell（MLC）兩種，SLC 表示單位晶元中可以存取一個位元的資料，而 MLC 則是單位晶元中允許存取多位元的資料，因此相對而言 MLC 的單位價格較便宜，但是 MLC 的

存取速度較慢，Endurance 也較低。表 1 為 Samsung 的 Flash chip [2]規格，由表可見 SLC 性質優於 MLC，但兩者共通的是讀取速度遠快於寫入速度，而寫入又比 erase 更快。

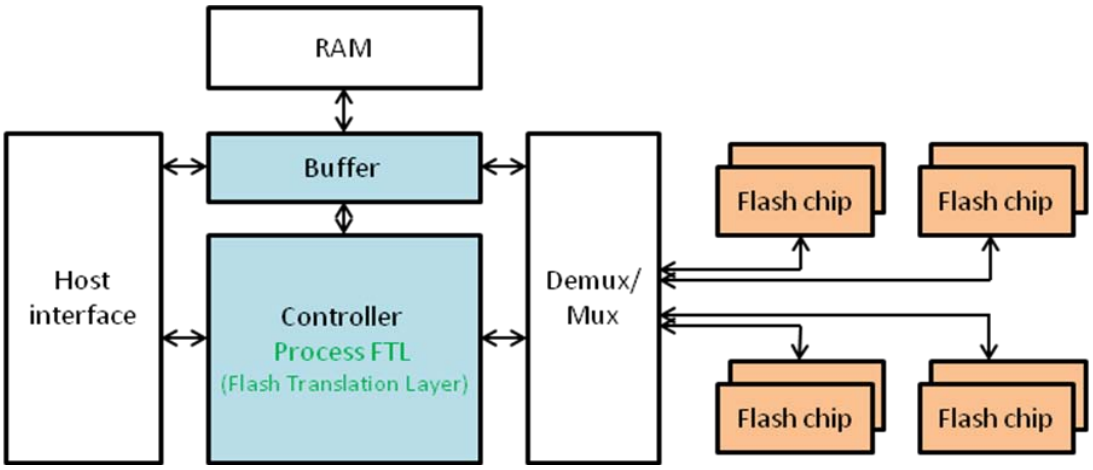


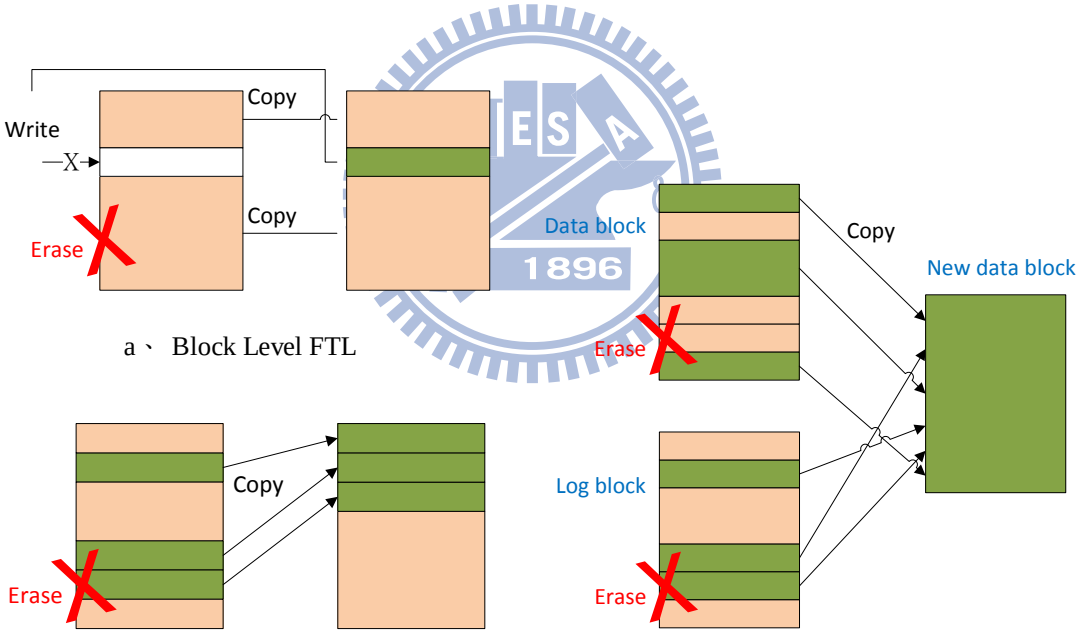
圖 2、Solid State Drive 硬體架構

SSD 是由多顆 Flash chip 所組成，是 Flash 極致的一種表現，圖 2 為一個典型的 SSD 硬體架構，其傳輸介面和傳統硬碟相同，採用 SATA [3]、PCI [4]等規格，資料進出前會先交由處理器（Controller）處理，所謂韌體指的就是處理器負責的動作，為了應付 Flash 的物理特性，韌體必須採用各種機制加以處理，Flash Translation Layer（FTL）就是為了實現這些機制所加入的轉換層，現今由於效能的考量，許多 SSD 又會在傳輸介面和處理器之間增加一塊 RAM 作為緩衝區（Buffer），其同樣由處理器管理，因此這邊把 Buffer 的管理機制和 FTL 合稱為 SSD 的韌體演算法（Firmware Algorithm），而 Flash chip 之間排列方式的討論則為 SSD 的硬體架構（Hardware Architecture）。

2.2 SSD Firmware Algorithm

SSD 的韌體設計包含 Flash Translation Layer（FTL）、緩衝區的替換機制（Buffer Replacement）、錯誤校正（Error Correction Code，ECC）和壞區塊偵測（Bad Block Detect）等，其中 FTL 和 Buffer 是韌體設計中影響 SSD 效能的關鍵，因此在本節稍作討論。

有關 FTL 的設計可以分為幾個議題，首先，由 Host 來的存取要求（Request）和實際被放入 Flash 中的位址通常不會一樣，因此需要將被寫入 page 本來的位址記錄下來，這種紀錄位址的行為稱為 Address Mapping。其次，由於 Flash 讀寫是以 page 為單位，但 erase 是以 block 為單位，因此每次進行 erase 之前必須把在 block 中還有效 page 搬移到別的 block，這種分類後回收的機制稱為 Garbage Collection（GC）。同時，為了減緩 block 超出 erase 次數限制而壞掉發生的情況，設計時會想辦法平均每一個 block 的 erase 次數，以延長 Flash 的使用壽命，這種機制稱為 Wear Leveling（WL）。



a、 Block Level FTL

b、Page Level FTL

c、Log-based FTL

圖 3、各類型 Flash Translation Layer 的 GC 流程

最陽春的 FTL 是採用 in-place 的寫入方式，即資料寫在指定的位址，由於大部分的 NAND Flash 無法容許隨機的寫入，也就是說每一次寫入的 page 必須接續上一次這個 block 寫到的位址，因此每次在做這種 in-place 的寫入之前都必須將原本 block 的資料循序複製到新的 block，同時將要寫的 page 寫到新的 block，最後對原 block 做 erase 的動作並以新的 block 取代原本的，如圖 3a 所示，這種方法每一次寫入需要付出較高的時間成本，但 RAM 裡面只需要紀錄 block 的位

址，又稱為 Block Level FTL。於是為了改善效能，產生另一種 out-place 的寫入方式稱為 Page Level FTL，它將不同位址的資料依序寫入到空的 block 中，當 block 用完時必須做 GC 將有效資料複製出來，如圖 3b，這種方法每次 GC 後可以得到一個空的 block，經過寫滿一個 block 的時間才需要再次 GC，且每次 GC 所花費的時間成本較低，但缺點是必須紀錄每一個 page 的位址，RAM 的需求十分龐大。

現今多採用 Log-based 的 FTL，其結合 Block Level 和 Page Level 兩種方法，將 block 分為 data block 和 log block 兩種，使用者只會看到 data block 的部分，所以寫入 request 的位址都是 data block 的部分，但資料實際上會以 out-place 的方式存到 log block 裡，每次寫完一個 log block 就再配置一個，至於未配置的 log block 則稱為 spare block，而當 spare block 用完時就必須做 GC，如圖 3c 所示，它的 GC 要做一個合併（Merge）的動作，就是將 log block 裡面的有效 page 及其本來所屬的 data block 以 in-place 的方式搬移到新的 data block 中，然後 erase 本來的 data block，一直重複 merge 的動作等到所有 log block 中的有效 page 都被搬走後再將 log block 做 erase，而 erase 後的 block 都放到 spare block 中等待再次被配置。這種 Log-based 的 FTL 不僅能有效節制 RAM 的消耗，且效能表現佳，甚至 Page Level 像 Super Block FTL [5]也曾引用到這種 log block 的概念，因此目前廣泛被使用在 SSD 上面，其包括各式各樣的類型像是 BAST [6]、FAST [7]、N-K [8]等，本篇的討論也多以 Log-based FTL 為主。

採用 Buffer 的主要目的是可以馬上吸收掉 request，避免回應時間過長，因此 Buffer 通常使用存取速度較快的 DRAM 實現，Buffer 又分為 Read 和 Write buffer，這裡討論的 Buffer 都作為 Write 用途，Buffer 寫滿後要做寫回（Write Back）的動作，於是該挑哪一筆資料寫回這個問題就衍伸許多不同的機制，像 First In First Out（FIFO）、Least Recently Used（LRU）等就是基本的方法，大部分的 Buffer Replacement 機制都是為了保留頻繁存取的資料，以減少寫回的次數。在 SSD 的設計上，Buffer 的使用還可以增加系統的平行度，由於隨機存取的行為容易造成不同 Flash chip 的資料分配不平均，因此可以透過 Buffer 的挑選機制把 request 平均分散到不同 Chip 上，這也將在之後做討論。

2.3 SSD Hardware Architecture

硬體設計的技術主要著重在資料的平行處理，可以大幅增加循序資料的處理速度，分為外部設計（Inter-Chip）和 Flash chip 內建（Intra-Chip）兩種，Inter-chip 的設計指的就是 Flash chip、資料線（Data line）和訊號線（Chip enable line）之間的組合，可以分為同步運作通道（Synchronized-Channel，簡化為 Syn-Channel）、獨立運作通道（Independent-Channel，簡化為 Ind-Channel）和共用通道（Interleave 或 Shared-Bus），而目前常見的 Intra-chip 設計則包含 Multi-Plane 和 Copy-Back。由於 Intra-chip 的限制較多且平行度較低，因此 Inter-chip 的設計就變成硬體設計中提升 SSD 效能的關鍵。

在 Flash 寫入的過程，資料會先由 Host 進入 SSD 的 Buffer，然後再由 Buffer 寫入 Flash chip 內部的暫存器（Register），這段時間稱為傳輸時間（Bus Delay），其多寡取決於用來搬移資料的處理器或 DMA（Direct Memory Access）的速度，但瓶頸也有可能在於 register 的存取速度，接著資料會再由 register 寫入到 Flash chip 中，這段時間稱為內部存取時間（Chip Busy），不同規格的 Flash chip 或不同行為的 Chip busy 時間也會不一樣（參考表 1），這兩段時間是 SSD 資料平行處理的主要考量。

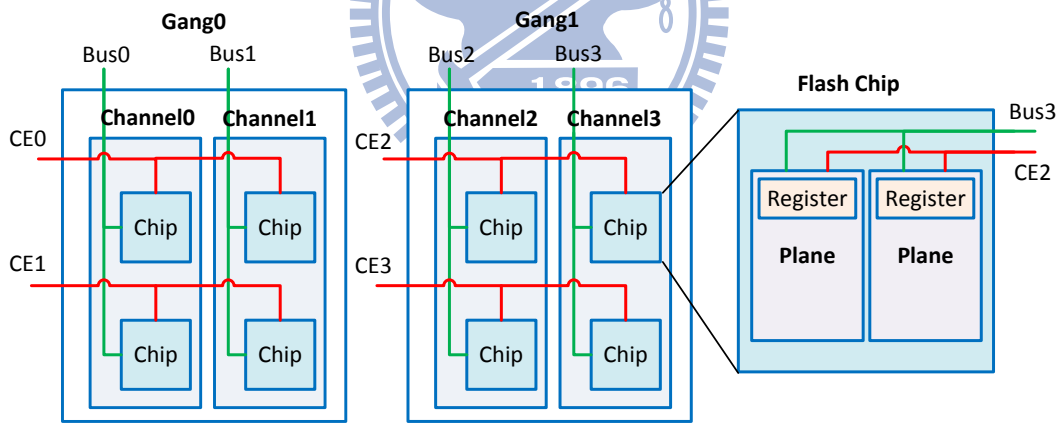
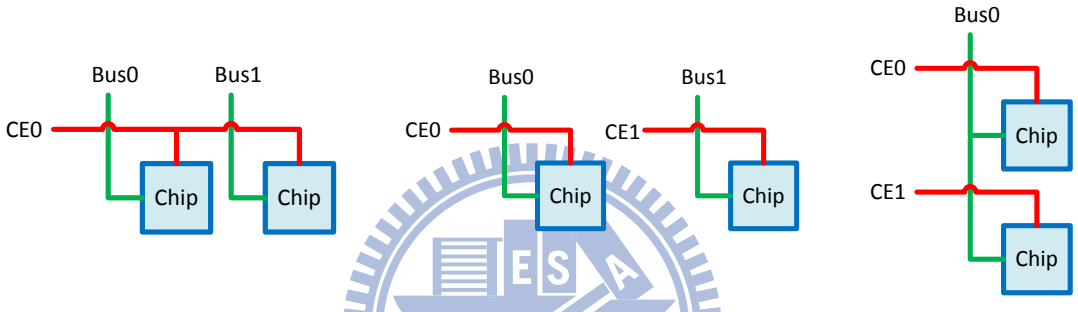


圖 4、SSD 硬體元件

Inter-chip 的硬體架構可以根據 Flash chip、Data line 和 Chip enable line（CE）三種硬體元件的組合來分類，Data line 即為 Bus，是資料傳輸的通道，CE 則是訊號線，是處理器用來啟動 Flash chip 動作的管道。如圖 4 與相同 Data line 連接的 Chip 稱為一個通道（Channel），每一個 Channel 必須由一個處理器或 DMA 負責資料傳輸的工作，而連接相同 CE 的 Channel 又稱為通道群（Gang），連接相同 CE 的 Chip 則必須同時運作，每一個 Channel 中相同對應位址的 Chip 又合稱為晶片排（Bank），Gang 和 Bank 都必須保持資料的平行度，其中 Gang 中的每一個 Channel 要存取相同位址的資料，因此 Gang 可以想像成較大的 Channel 有

著數倍大小的 block 和 page。

經由上述硬體元件的組合，Inter-chip 的架構可以簡單歸類為三種：Syn-Channel、Ind-Channel 和 Interleave，如圖 5a 所示，Syn-Channel 就是 1-Gang 的表現，每一個 Channel 可以同時存取資料，但由於 CE 是共用的，因此就算其中一個 Channel 沒有工作但也必須存取和另一個 Channel 相同的位址，所以為了增加 Channel 的利用度較適合處理循序的資料。相反的，圖 5b 的 Ind-Channel 架構下 Gang 與 Gang 的 CE 是不同的，因此可以獨立的運作並存取不相同的位址，這種設計可以適應較複雜的 Workload，但要如何平行不同 Channel 的工作也值得思考。圖 5c 為 Interleave 的架構，其 CE 分開，但 Bus 共用，因此必須輪流傳輸資料，Bus delay 的時間無法重疊，因此平行度低於 Syn-Channel 或 Ind-Channel。



a、Synchronized-Channel　　b、Independent-Channel　　c、Interleave

圖 5、SSD Inter-Chip 硬體架構

Intra-chip 的硬體支援包括 Copy-Back 和 Multi-Plane。Copy-Back 是 Flash chip 提供的特殊指令，使同一顆 Chip 的資料搬移可以不經過 Bus，而是直接透過 register 的讀寫來完成，因此相較於一般的搬移動作免去兩次的 Bus delay，但缺點是不經過 Bus 也就不會經過處理器，因此無法對資料做額外的處理例如錯誤校正（ECC）等，所以使用上會有一定的風險。一般的 Flash chip 可以分為多個區階（Plane），如圖 4 每一個 Plane 都有一個 register，Plane 的資料讀寫都可以經過 register 來完成，因此同一顆 Chip 的不同 Plane 可以同時進行 Chip busy 的工作，其平行度類似 Interleave，但限制是 Plane 必須同時運作而且要存取相同的位址，這種 Multi-Plane 的行為是大部分 Flash chip 所支援的特殊指令，其支援的行為包含 read、write 和 erase。

2.4 Design Objectives

RAPT 提供一個軟體模擬的環境，允許事先評估可行的 SSD 設計，使用者可以自由規畫所需要的韌體演算法和硬體架構，然後透過 RAPT 的建構成為一個虛擬的 SSD。如圖 6 所示，在 RAPT 中使用者可以接觸到的部分包括兩個輸入檔案 Design 和 Trace 以及一個輸出檔案 Result，Design 是一個.cpp 檔，RAPT 的函式庫提供各種 API，使用者引用後就可以利用那些 API 完成韌體部分包含 Buffer、FTL 等設計以及硬體部分包含架構、規格的定義，這些設計或定義就寫在 Design 檔案內，Trace 則是使用者要測試的存取行為（Workload），以文字檔表示，這兩個輸入檔在經過 RAPT Interface 整合後首先會根據 Design 檔的設定產生一個虛擬 SSD，內容包含對應的硬體和韌體模組，再透過虛擬 SSD 執行 Trace 中的存取行為，並且將最後的結果輸出到 Result 檔案，預設輸出的內容包含效能（Throughput）、磨損情況（Wear state）等統計資料。

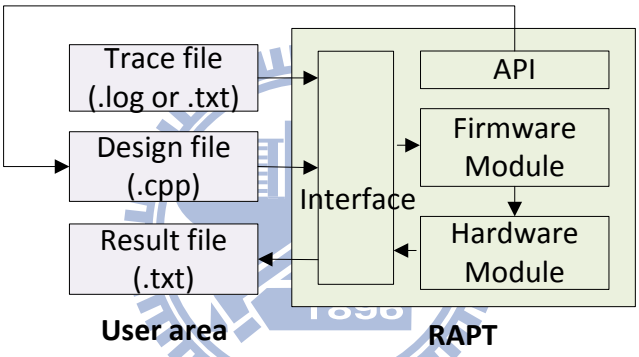


圖 6、RAPT 操作示意圖

為了確保 RAPT 的實用性，我們規範了功能性（Function）、可靠性（Reliability）和移植性（Portability）三項指標，並且個別進行測試，功能性要求 RAPT 所提供的 API 必須能夠快速實現目前知名的 FTL 像是 BAST、FAST、N-K 等，並且可以正確模擬已知的 SSD 規格以及架構，可靠性部分將以真實 SSD 進行驗證，我們定義在各項硬體參數值正確的情況下，模擬誤差不能高於 10%，最後關於系統移植性，RAPT 是由 C++搭配 System C 函式庫實作完成，因此允許在 Windows 或 Unix 平台上編譯執行，是具有高度可攜性的系統。

2.5 Related Work

以軟體模擬硬體的概念很早就被提出來，各類型的模擬器（Simulator）也行之有年，近幾年 SSD 的熱度持續飆高，因此各種模擬與測試方法也一再被提出來，本節針對已發表的 NAND Flash-based SSD simulator 作討論，它們有各自的做法或用途，但目的都是為了正確模擬 SSD 的行為。

微軟公司（Microsoft）的 Nitim Agrawal*等人首先提出基於 Disksim [9]的 simulator，並以此為基礎發表 SSD Design Tradeoffs [10]，其歸類 SSD 效能的瓶頸，並整理出各種 SSD 的硬體架構，將可平行化的硬體元件分為 Gang、Die、Plane 三個層級，同時測試各種組合與設定下的效能，對於 SSD 架構的認識有很大的幫助。隨後 Ji-Yong Shin 等人又以相同的 simulator 發表關於 High-Performance SSD [11]的研究，主旨以 Page Level FTL 為基礎探討在高平行度的架構下各種添加的 FTL 設計方法，其討論包括資料擺放位址（data allocation）、冷熱資料分離（hot/cold separation）和資料傳輸平衡（load balance）等。兩篇對於 SSD 的硬體和韌體上分別提供前瞻性的研究，但其所採用的 simulator 架構完全與 Disksim 綁死，因此韌體與硬體的架構難以修改，Disksim 本來用於模擬傳統硬碟，其原始碼的混雜也無法提供給使用者充足的設計概念，且其所作的 SSD 模擬研究以硬體架構為主，FTL 方面僅討論 Page level，研究範圍太過狹隘。

Jongmin Lee 等人提出 SSD simulator 的研究，並命名為 CPS-SIM [12]，其主要探討 SSD 各硬體元件之間的溝通行為與時間，它將會影響 SSD 效能的時間列為 tR、tPROG、tE、tCMD 和 tBUS 五項，並且透過模擬找出平行架構像 Syn-Channel 和 Interleave 下效能的瓶頸。本篇提到的 SSD simulator 跳脫 Disksim 成為獨立開發的模擬工具，但同樣沒有足夠的韌體支援，且只有對單一 FTL 做過討論，因此僅適合作為 SSD 硬體架構的研究用途，並不是完整的模擬工具。

前幾篇研究的議題都關注在以 simulator 所模擬的研究成果，但其實只要有便利的工具，使用者可以更自由並更快速的產生高價值的研究成果。Youngjae Kim 等人提出 FlashSim [13]的研究，並建立一份詳盡的 simulator 架構，它將處理器、FTL、SSD、Channel、Bus、RAM 等韌體和硬體全部以元件的方式表示，元件和元件之間可以觸發不同的行為，基於物件導向的設計可以讓使用者對於 SSD 的模擬更熟悉。FlashSim 提供豐富的硬體模擬資源，但其依然缺乏給予使用者的韌體支援，雖然其預設有四種 FTL 給使用者選擇，但並沒有提出修改或設計的概念，因此使用者難以對 SSD 的韌體設計做出選擇，同時其開發彈性過少，例如使用者無法存取底層讀寫統計資料等的統計資訊，也沒有提供自定變數或函式的窗口等，且其定義的各種元件的分類是否適用於所有組合也無法判定。

綜觀以上 SSD 的 simulator，大部分單純為學術研究用途而開發，因此並未提供給使用者足夠便捷且完整的開發環境，大部分的 simulator 都強調硬體資源的強度，但複雜的硬體設定對精確度幫助有限，反倒增加韌體設計的負擔，增加使用的困難度，所以模擬工具提供韌體支援是必要的。其次由於設計藍圖難以取得，因此所開發的 simulator 往往沒有討論其模擬準確度，FlashSim 雖然有與真實 SSD 做過比較，但僅限於以 workload 複雜度測試效能的走向，並沒有關於模擬誤差的討論。另外，眾多關於 SSD 的研究都著重在單方面硬體或韌體設計的討論，而忽略不同韌體與不同硬體之間的搭配關係。因此本篇提出 RAPT 工具，首先改善傳統 SSD 模擬韌體支援不足的問題，提出一套便利有效的韌體模型，並建立 Block level、Page level、BAST、FAST 和 N-K 等 FTL 為範本，使用者可以透過韌體模型輕易的修改 SSD 韌體，同時 RAPT 也將各種支援整合成 API 讓所有設計可以更方便進行。其次，本篇透過與實際 SSD 的比對，並歸納各種誤差可能性，以增加使用的有效性。最後利用 RAPT 探討各種韌體與硬體架構的組合，使得 SSD 的研究可以更加具有宏觀性。

3. DESIGN ISSUE

RAPT 的訴求是讓 SSD 的開發能夠更加迅速，因此本篇提出全新的韌體和硬體設計概念，幫助使用者簡化大部分的模擬工作。本章將分別在 3.1 和 3.2 中討論韌體演算法和硬體架構的設計概念，前者將探討如何抽像 FTL 的各種行為，後者說明 RAPT 以何種方法快速建構各種 SSD 硬體架構。

3.1 Abstract Firmware Layer

現存各種 SSD 的模擬器，多著重在硬體模擬的支援，使得 SSD 硬體架構漸漸變成制式化的設計，所有允許的支援都大同小異，卻沒有研究是針對韌體設計來討論，因此 RAPT 特別針對 SSD 韌體的設計提出一套模型。

為了減少使用者開發 SSD 韌體的負擔，並且建立一個簡易修改的韌體架構，簡化設計的行為是必要的，簡化的方式首先要找到各種 FTL 的共通行為，其必要的行為像是 Address Mapping、Garbage Collection 等就變成值得討論的重點，其次，找到這些行為後必須以怎樣的方式呈現才有辦法減少模擬的負擔？從規劃設計的角度而言，最能夠減輕負擔又不喪失彈性的方法就是紀錄資訊，若所要用的資訊能夠以有效率的方式紀錄起來，透過適合的窗口供使用者取用，就可以大大減少設計的麻煩，於是本篇提出紀錄各種 FTL 共通行為中的資料關係（Data Relation）的方法。

要紀錄的資料類型很多種，本篇定義元素（Element）、群組（Group）和關聯性（Relation）三者來解釋各種共通的 FTL 行為。Element 表示單一數值的資料，像是索引（Index）、位址（Address）或計數值（Counter）等。而 Group 指的是單一類型的資料集合，成員可以是 Element、Group 或 Relation，其用來表示較大筆的資料。Relation 則用來表示資料與資料之間的關聯性，可以是 Element 和 Element 之間的關聯或 Group 和 Group 之間的關聯等，舉例像索引與資料之間的對應關係或順序性（Priority）等。本篇就以 Element、Group 與 Relation 制定了三種韌體標準模型，分別是 Index、Association 和 Prioritization，依序在本節接下來的小節討論。

3.1.1 Index

由於 Flash-based 的 SSD 具有無法在相同位址重複寫入的特性（Write-Once），必須將要寫入的資料暫時寫到其他 block 或 page，所以 Address Mapping 的行為在任何 FTL 都是必要的，因此本篇定義 Index 為 RAPT 的第一種韌體標準模型，用來表示鍵（Key）與值（Value）的關係，如式 1 所示：

Definition

$$R = \{(k_i, v_j) \mid k_i \in Key, v_j \in Value\} \qquad (式\ 1)$$

其中 Key 是唯一，用來作為索引，Value 則表示對應到的資料，簡而言之，Index 是一對一的關係，如圖 7 所示，以 Key 為索引，每一個 Key 會對應到一個 Value，並且允許重複。

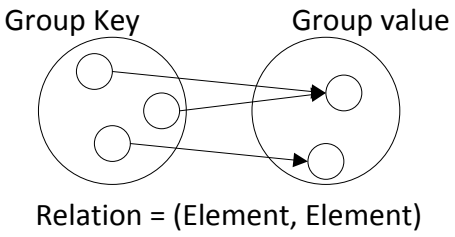


圖 7、Index 標準模型關係圖

Index 的應用範圍很廣泛，可以用來紀錄資料的範圍或屬性，但最主要用途還是在 Address Mapping，圖 8 為 Index 的應用範例，其中 LBA 表示邏輯頁位址（Logical Block Address），PPA 表示實體頁位址（Physical Page Address），依此類推。在 BL 中由於 page 寫入每一個 block 的相對位址是固定的，因此只需要紀錄 LBA 對應到 PBA 的關係，同理 PL 中則需要紀錄 LPA 對應到 PPA 的關係，而結合兩者的 Log-based FTL 不僅要紀錄 data block 中 LBA 對應 PBA 的關係也需要紀錄 log block 中 LPA 對應 PPA 的關係，以 Log-based 的圖為例，在 block 位址對應上，LBA 5 和 LBA 0 就是 Index 中的 Key，而 PBA 0 和 PBA 1 則是 Value，而 page 位址的 Key 和 Value 對應關係要記錄的就是 21、0、23、3 和 1 邏輯位址分別對應到 8、9、10、11 和 12 實體位址，因此這類的位址對應關係都可以用 Index 來表示，其在各種 FTL 的使用都是必要的。

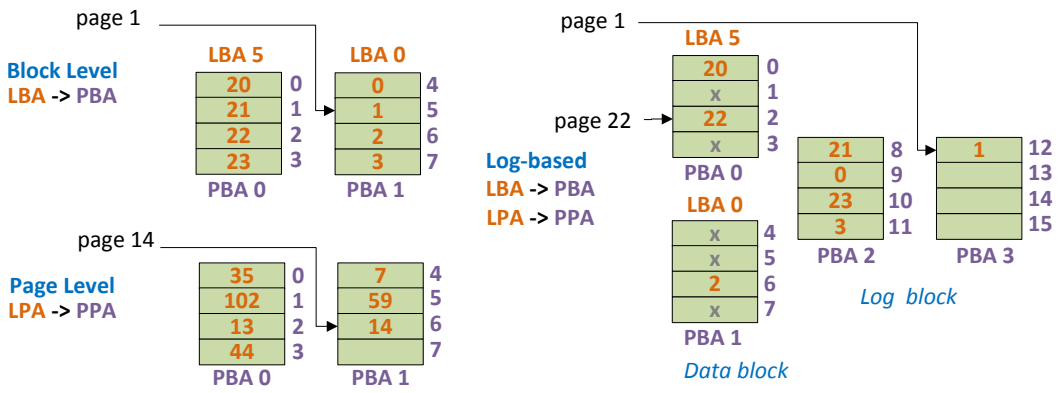


圖 8、Index 應用範例

3.1.2 Association

BL 和 PL 架構簡單，主要考慮 Address Mapping 的關係，因此實作起來比較容易，但 Log-based FTL 包含 block 和 page 的對應搭配，其 Garbage Collection 時候的變化與種類都很繁複，因此除了以 Index 表示，也必須用其他韌體模型刻劃，於是 RAPT 定義 Association 韌體標準模型，用來表示資料集合（Data Set）之間的關係，如式 2：

Definition

$$R = \{(a_i, b_j) \mid a_i \subseteq A, b_j \subseteq B\} \qquad \text{（式 2）}$$

其中 A 和 B 是兩個集合，兩者的成員以 a 和 b 表示，a 和 b 在各自的集合中都必須是唯一，其可以是位址、計算值或者其他結構的索引，因此也可以用來表示 Group 或 Relation，歸納而言，Association 表示的是多對多資料的關聯性，如圖 9 所示，它將兩個 Group 關聯在一起，Group 中可以僅有一個 Element，因此也可以表示一對多或多對一的關係。

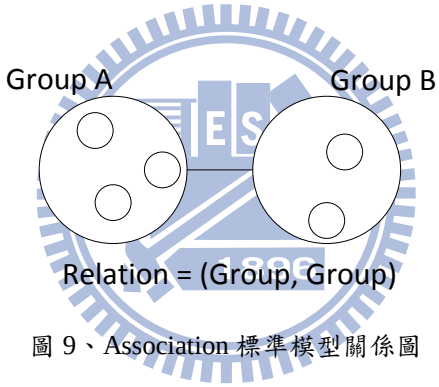


圖 9、Association 標準模型關係圖

建立 Association 的目的在於可以更方便的處理 FTL 中 block 與 page 的關係，以 Log-based FTL 為例，request 在寫入時會進入 log block，由於 GC 時要做 data block 與 log block 的 merge 動作，所以要記錄 log block 中的 page 是屬於哪個 data block，這種關聯性在 Log-based FTL 中是必要的，如圖 10，範例中的 Association 模型可以用來表示兩種關聯性，第一種是 log block 內包含的 LPA 資訊，是一對多的關係，像是 PBA 2 與 LBA 21、0、23 和 3 具有的關聯性，兩者可以分別表示 Group A 和 B，第二種是 log block 與其所關聯到的 data block，因為相同 data block 的資料可能存在不同的 log block 內，所以兩者的 PBA 就構成多對多的關係，以圖例而言，PBA 0、1 和 PBA 2、3 就構成 Association 的關聯性，當空間不夠用時，就可以直接找到要 GC 的 Association，將 Group A 和 Group B 中的 PBA 全部 erase。

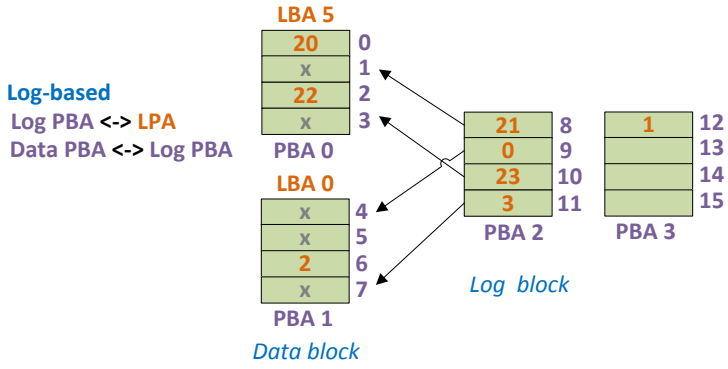


圖 10、Association 應用範例

3.1.3 Prioritization

存在 FTL 的另一項必要行為就是空間配置（Allocation）與回收（Recycle），舉例而言，在 SSD 中每次要寫入資料而舊的 block 沒有空間時，就必須配置一個新的 block 做寫入，但要根據什麼因素來作配置並沒有統一的方法，最久未被使用到的還是磨損次數（即 erase 次數，表示 block 的 Endurance）最少的 block 等，這些配置方法的共通點是都具有優先序關係，像是最久或最少，於是 RAPT 提出 Prioritization 標準模型，用以表示有順序性的資料，如式 3：

Definition

$$(P,\psi) = orderset$$
$$\psi = \{(p_i, p_{i+1}) \mid (p_i, p_{i+1}) \in P \times P, p_i < p_{i+1}\} \quad \text{（式 3）}$$

其中 P 為一個偏序集合（Partially Ordered Set），ψ 用來表示其成員具有由小至大的順序關係，如圖 11 所示，P 為一個包含很多 Key 的 Group，Key 不能重複，但會依據其優先權關係被排序，透過 Prioritization 標準模型，可以表示所有 FTL 中包含的順序性資料。

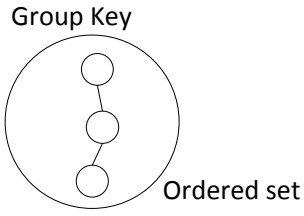


圖 11、Prioritization 標準模型關係圖

圖 12 為 SSD 中 block 配置的示意圖，做 GC 時首先要選擇 erase 哪一個 block，被選中的稱為犧牲者（Victim），這些 victim 的挑選就與優先權有關係，可以挑選最 cold 的資料或回收成本最低的資料，同理在做 Allocation 時也必須從 spare block 中挑選空白的 block（Free block）以供寫入，這些挑選都存在著先後關係，

因此這些 victim 或 free block 都可以透過 Prioritization 模型建立為順序性的資料。圖中以三種優先權為例，分別是時間、GC 負擔和磨損次數，而所需要排序的 Key 則是 PBA。

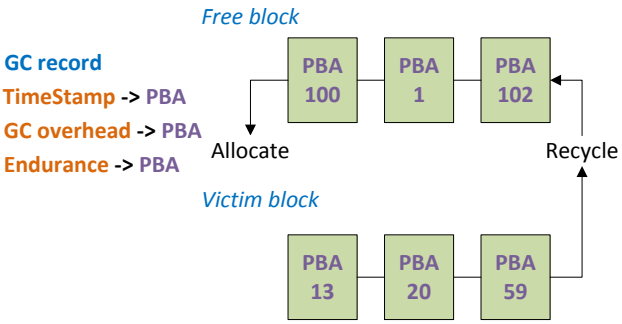


圖 12、Prioritization 應用範例

綜合而言，RAPT 提出三種韌體模型，足以抽象 SSD 韌體中 FTL 的各項主要行為，其中 Index 模型用來處理 Address Mapping，Association 模型用以處理 Garbage Collection，而 Prioritization 可以用來表示 Allocation 和 Recycle，透過這些模型可以很快速的將 FTL 制定出來，由於這些模型也代表各種 FTL 的共通行為，所以制定的架構可以適用於多種 FTL，使得不同 FTL 的實現可以透過模型的部分修改就達成，這樣可以更加速 SSD 的韌體開發，此部分可以參考 4.1.1 的實現範例。

3.2 Timing Simulation Framework

硬體架構是 SSD 的設計重點之一，在模擬上通常會制定各種相關的硬體像是 Chip、Bus 等，依據這些硬體的組合便可以模擬不同 SSD 的架構，多數的工具提供十分細節的硬體模擬，但是精細的時間單位與硬體溝通就需要較長的模擬時間，而越複雜的硬體架構也會加深韌體實作的難度，因此 RAPT 簡化部分硬體溝通的機制，並且建立對應的設計介面以避免硬體設計對韌體開發造成負擔。

RAPT 以時間元件取代傳統工具以硬體元件模擬的方法，舉例而言，在 SSD 運作上最花費時間的動作是資料傳輸，於是 RAPT 將模擬分為 Bus 上的傳輸和 Chip 內部的資料搬移時間，分別以 Bus delay 和 Chip busy 兩個時間元件（Timing Component）表示，每個時間元件有各自對應的硬體，像是 Bus delay 可能是屬於 Bus 0 或 Bus 1，透過這些時間元件的組合就可以表示各種不同行為的時間，像是讀取的動作相當於經過一次 Chip busy 和一次 Bus delay，寫入的動作則相反，先會經過 Bus delay 才是 Chip busy。

而要進行平行時間的模擬則必須透過一套時序系統（Timing engine）來完

成，為了提升模擬效率和準確度，RAPT 基於 System C 的時序概念做設計，SystemC 是一套硬體描述語言，以 C/C++為基礎，具備模組化的概念，包含很多硬體溝通的支援像是通道口（Port）與事件（Event）等，其多執行緒的運作可以模擬各種平行的時間，RAPT 採用其事件傳遞的溝通機制與時間計算函式，並且利用 System C 在模組建立虛擬的時間軸。

模擬時 RAPT 允許虛擬時間軸同時存在多個屬於不同硬體的時間元件，於是 RAPT 利用所定義的時間元件與 SystemC 時序系統，就可以模擬不同架構的平行關係，像 2-Channel 的讀取是重疊的 Chip busy 加上 Bus delay，而 2-Interleave 則是重疊的 Chip busy 加上兩次 Bus delay，因此，透過這種概念就可以很輕易的呈現各種 SSD 的硬體架構（更多實現方式可以參考 4.2）。

4. IMPLEMENT

RAPT 可以分為內部介面和外部介面討論，如圖 13 所示，內部介面分為韌體模組（Firmware Module）、硬體模組（Hardware Module）和軟體模組（Software Module），韌體模組負責處理 SSD 韌體的設計，包含 FTL、Buffer 等，依據使用者設計不同而有所變化，硬體模組負責計算不同硬體元件的運行時間，主要維持平行架構的運行，軟體模組負責接收 Trace 中的 request，同時也統計各項測試數據。外部介面則包含使用者介面（User Interface）和三個使用者檔案，使用者介面建立三個模組對應的 API，負責與內部介面的溝通，使用者可以利用那些 API 撰寫 Design 檔，並選用適合的 Trace，最後 RAPT 會將測試結果輸出至 Result。

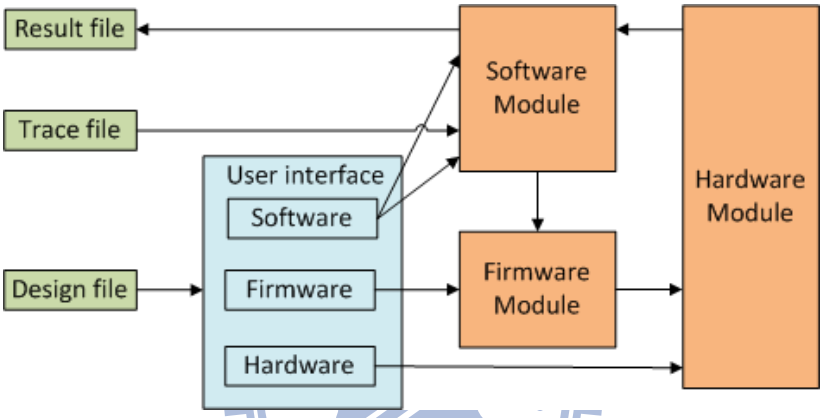


圖 13、RAPT 系統架構圖

目前 SSD 模擬器的實作方式可歸類為兩種，一種是建立在已存在的模擬工具中，與原來模擬工具的原始碼混雜，因此不僅很多硬體參數無法動態做調整，韌體上要修改也很不容易，另一種則是獨立為一個完整的程式或模組，通常這樣的做法必須提供一個設計窗口（Design Window），提供使用者包含參數設定、演算法修改的介面，RAPT 的 Design file 就是一個 Design window，但是一般工具卻缺乏 Interface 幫助使用者存取模擬的資訊，舉例而言若要得到某個 block 的 erase 次數就要看懂大部分的程式碼才有辦法，為了避免這項困擾 RAPT 分別在三個模組中建立 API。

本章分為三節，依序討論韌體模組的實現方式、硬體模組的實作與各種架構的實現、軟體模組與使用者介面的支援。

4.1 Firmware Module

韌體模組就是 SSD 的韌體演算法內容，根據不同的需求而有所不同，使用者可以利用所提供的 API 進行設計，API 包含三種韌體模型的對應關係、Flash

指令讀、寫、抹除等以及資訊查閱的功能，實作時，使用者可以透過 Index、Association 和 Prioritization 三種韌體模型的 API 快速的將不同的 FTL 以類似的架構制定出來，4.1.1 小節便以 Log-based FTL 為例，討論如何使用 Index、Association 和 Prioritization 建立一套架構可以適用於 BAST、FAST 和 N-K 三個 FTL，4.1.2 則討論目前的韌體實作方式仍需要做的改善。

4.1.1 Modeling FTL

Index、Association 和 Prioritization 包含不同類型資料對應的關係，RAPT 採用改良式的 Avl-Tree 實作，其不僅可以儲存順序性的資料，還具有節省空間、快速插入與搜尋等優點，因此可以很容易解決所有模型的需求。

BAST、FAST 和 N-K 都是 Log-based 的 FTL，都有區分 data block 和 log block，寫入時資料會進入 log block 中，但三者寫入的位址不一樣，BAST 是一個 data block 對應一個 log block，FAST 是全部的 data block 對應全部的 log block，而 N-K 則是 N 個 data block 對應 K 個 log block，因此 N 個 data block 與 K 個 log block 表示一個 Group，如圖 14 所示，request 會根據邏輯位址所在的 data block 寫到對應的 log block 中，這樣的對應方式根據資料的 locality 不同會產生不同的效能，這部分可以參考 5.3.2 的實驗。

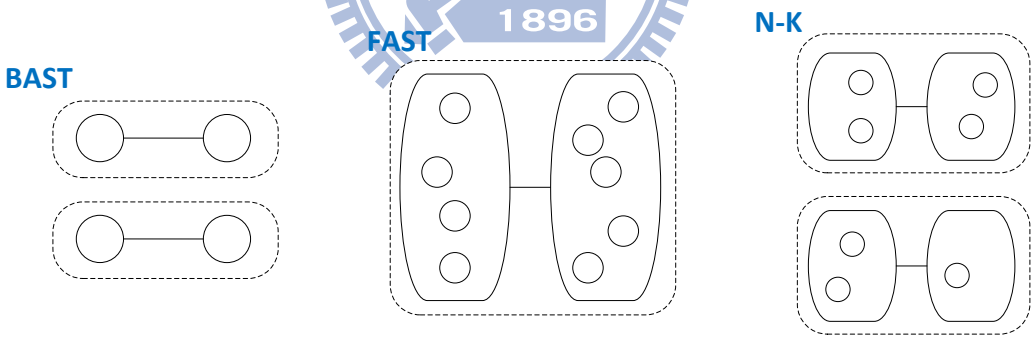


圖 14、BAST、FAST 和 N-K 的寫入對應關係

首先，為實現這三者 FTL 必須先紀錄其 Address Mapping 的關係，RAPT 使用 Index 來記錄邏輯位址對應 log block 中 page 的實體位址，同時也記錄 data block 的邏輯位址對應到的實體位址，前者在 request 寫入 log block 的時候作記錄，後者當 GC 時以新的 data block 替換舊的時候需要紀錄，因此，在 Index 的使用上三個 FTL 是相同的。實作上每一個樹節點可以表示一筆對應資料，每次對應需要花費 log N 的時間蒐尋資料，比雜湊（Hash）演算法略慢，但較省空間。

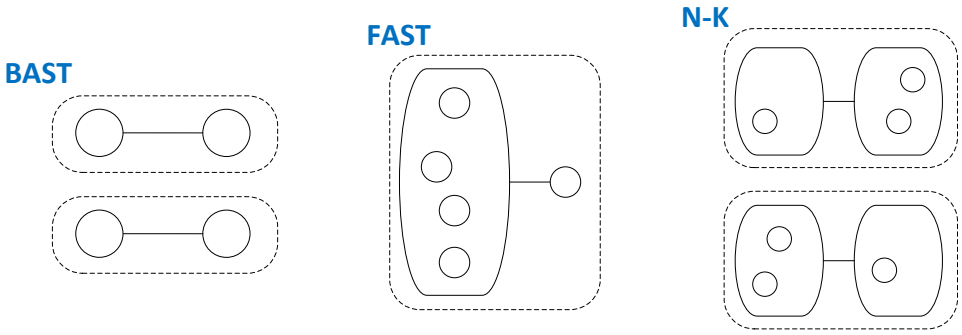


圖 15、BAST、FAST 和 N-K 的 GC 範圍

而 Association 用來表示三個 FTL 不同的 GC 範圍，如圖 15 所示，BAST 每次會 GC 所挑中的 log block 與其對應的 data block，會付出一次 merge 包含兩次 erase，而 FAST 每次 GC 一個 log block 以及與其有關連的 data block，也就是說假設一個 block 包含 64 個 page，則其最多會做 64 次 merge 包含 65 個 erase，因此 FAST 的其中一個缺點就是 GC 時的 response time 過高，至於 N-K 的 GC 範圍與其寫入位址一樣，但只有關聯到的 data block 需要做 merge，因此最少是一次 merge 兩次 erase，而最多是 N 次 merge 加上 N+K 次的 erase。實作時每個 Association 會有一個索引值，用來找到對應的樹節點，每一個樹節點可以關聯到兩個鍊結串列（Linked List），假設為 Group B 和 Group A，分別記錄每次要 GC 的 log block 以及所對應到的 data block，則 GC 時就可以由 Group A 依序挑選 data block 作 merge，等到 data block 都 merge 完再挑選 Group B 中的 log block 作 erase。

Prioritization 用來表示 Allocate 和 Recycle，在 Allocate 時三者都是依據 FIFO 的原則，因此 Priortization 紀錄的 Key 是 block 的實體位址，而優先序關係則為 block 成為 Free block 的時間，因此每次 Allocate 時都會挑選時間最早的。在 Recycle 時三者的優先權都是最後被寫入的時間，表示挑選最久未被更新的位址做回收，而紀錄的 Key 則不相同，BAST 和 FAST 記錄單一 log block 的實體位址，而 N-K 是以一個範圍的 log block 位址為考量，因為挑選中的位址也表示要做 GC 的位址，所以 Prioritization 的用途此處除了 Allocation 也和 Garbage Collection 有關。實作時每一個 Prioritization 就表示一個優先權佇列（Priority Queue），每一筆優先權與 Key 的對應關係就是一個樹節點。

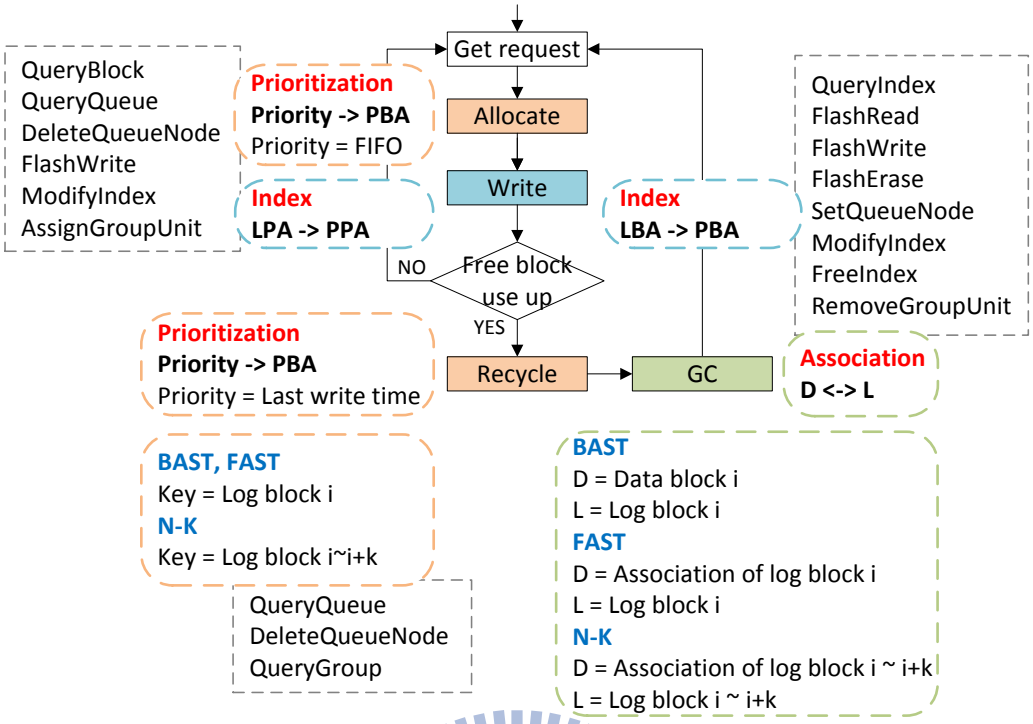


圖 16、BAST、FAST 和 N-K 的寫入共同架構

圖 16 為 BAST、FAST 和 N-K 的共同架構，以寫入為例，一開始由 Host 端接收到 request，然後透過 *QueryBlock* 找到要寫入 block 的 page 位址，若寫入的空間不足則以 *QueryQueue* 和 *DeleteQueueNode* 配置一個 Free block，再利用 *FlashWrite* 做寫入，並使用 *ModifyIndex* 和 *AssignGroupUnit* 建立 page 以及 data block 與 log block 的對應關係，完成寫入後依據剩餘的空間判斷是否需要做 GC，若不需要則繼續接收 request，若需要就透過 *QueryQueue* 找到 victim，如圖 BAST 和 FAST 是以一個 log block 的 PBA 為 Key，而 N-K 以一個範圍的 PBA 為 Key，這些實體位址都需要透過 *QueryIndex* 的查詢，而所挑中的 victim 也表示 GC 範圍，就是 data block 對應到 log block 的關係，BAST 是一對一，FAST 是一個 log block 與所關聯到的 data block，而 N-K 是最多 K 個 log block 與關聯到的 data block，這些都以 *QueryGroup* 得知，merge 時使用 *FlashRead* 和 *FlashWrite* 做資料搬移，最後再依序將 data block 和 log block 做 *FlashErase*，同時再以 *SetQueueNode* 記錄 Free block 的順序，並用 *ModifyIndex* 重設 data block 的實體位址，並且用 *FreeIndex* 和 *RemoveGroupNode* 清除不必要 page 和 block 的關係。

透過 RAPT 所制定的韌體 API，可以適應各種不同 FTL 的變化，並且任何程式碼的修改都可以透過這些 API 快速完成，舉例而言，由 FAST 修改為 N-K 只需要更動不到 10 行的程式碼即可完成，因此 RAPT 韌體模組的實作方式提供一種創新的 SSD 模擬方式。

4.1.2 Deficiencies of the Current Firmware Module

RAPT 的韌體模組可以用來設計任何 FTL，但在使用上仍然有很多無法簡化的動作，以圖 17 為例，KAST [14]是近年提出的 FTL，與 N-K 一樣是多個 data block 與多個 log block 組成一個 Group，不同點在於 N-K 的 Group 有連續性，例如 LBA 0~3 是 Group 0 而 LBA 4~7 是 Group 1，依此類推，而 KAST 每一個 Group 的 data block 的 LBA 是不連續的，這樣的好處是關聯性的利用度較高、存取的彈性較大。但在 RAPT 的實作上 N-K 可以直接以數學方法得到 Group 的索引，再利用過索引查詢 data block 對應到 log block 的 Association 關係，但 KAST 沒有固定的運算法則，因此實作上就必須先以 Index 記錄每個 data block 對應到的 Group，才能找到對應的 Association，實現過程也就多一道手續。

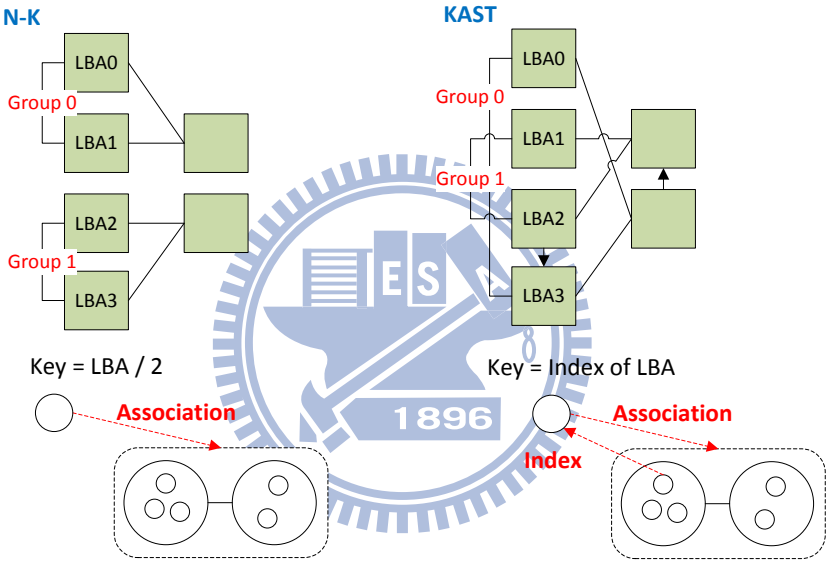


圖 17、N-K 與 KAST 的 Association 模型使用示意圖

雖然任何 FTL 都脫離不了 RAPT 所提出的資料對應關係，但很多關係是複雜的，以 RAPT 所提出的 API 仍無法有效簡化它們，除了上述的 KAST，還有採用樹結構（Tree）實作的 FTL [15]等都必須透過多重對應的關係才能建構，因此，不僅已存在的 API 還有細分的空間，也有更多的設計議題是值得討論的。

4.2 Hardware Module

一個完整的 SSD 模擬工具最重要的就是硬體架構的支援，RAPT 的硬體模組基於 SystemC 時序系統完成，結合 3.2 所定義的時間和硬體元件便可以很容易的模擬出各種不同的 SSD 架構，並且也將各種架構的設定以簡易 API 表示，使用者可以透過調整 API 的參數很容易達成模擬不同 SSD 架構的目的。4.2.1 說明硬體模組的實作方式，4.2.2 和 4.2.3 分別呈現 Inter-chip 和 Intra-chip 架構在實作上的細節，最後一小節則分析目前硬體模組的缺失。

4.2.1 Modeling Architecture

圖 18 是硬體模組的架構，當使用者使用 RAPT 的韌體 API 如 *FlashRead*、*FlashWrite* 等指令後，request 便會切割成對應的時間元件進入到指令佇列（Operation Queue），等到韌體模組的動作結束後便會進入到硬體模組中運行各個時間元件，每一個時間元件都有歸屬的硬體，硬體模組會將所有 Operation queue 中對應到準備完成（Ready）硬體的時間元件都放到時序系統中，同時更改該硬體狀態為忙碌（Busy），當某個時間元件運行結束後其對應的硬體就會被釋放回 ready 的狀態，此時其他要用到此硬體的時間元件才可以存取它，透過這樣的實作方式就可以模擬時間的平行關係與硬體之間的相依關係。

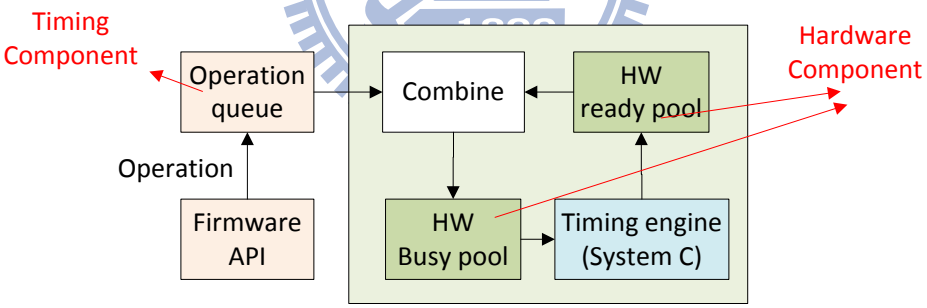


圖 18、RAPT 硬體模組架構圖

以圖 19 為例，假設要模擬 2-Gang 4-Channel 2-Interleave 架構的 SSD，可以直接採用 RAPT 所提供的 API 來完成，*SetupArchitecture* 就是用來設定 SSD 的 Inter-chip 架構，所有的模擬必須先經過這些硬體參數的設定後才能進行，上述的架構經過設定後便會在模組中建立對應的硬體，包含兩條兩倍寬的 Bus 和四組兩個同時運作的 Chip。而假設要制定 Flash chip 的規格，像是 2-Plane、MLC 或 SLC 等參數，也可以很快速的透過 *SetupFlashChip* 來完成。此外，由於時序系統的運行必須花費比較長的模擬時間，因此 RAPT 也提供 *DisableHardwareModule* 的指令讓不需要模擬時間只要計算次數等統計資料的使用者可以關閉 RAPT 的硬體模組，藉此加速模擬速度。

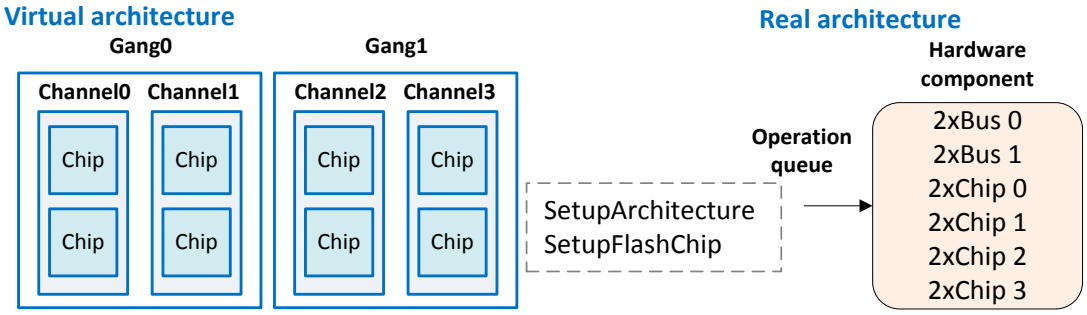


圖 19、硬體模組模擬示意圖

RAPT 的硬體模組不僅提供豐富的資源，也建立簡易的 API 讓使用者在模擬設定上非常方便，同時透過 SystemC 時序系統的輔助，RAPT 在平行時間的模擬效能也表現不俗，接下來將就各種 RAPT 所提供的架構支援做探討。

4.2.2 Inter-chip Architecture

Inter-chip 分為 Syn-Channel 和 Ind-Channel 兩種，兩者都是 Multi-Channel 的架構，以 RAPT 所定義的時間元件很容易描述其平行關係，如圖 20 所示，模擬時相當於同時存取不同 Chip 與不同 Bus，因此在 Bus delay 與 Chip busy 的時間都可以平行，不同點在於 Syn-Channel 中每個 Channel 必須同時運作且存取相同的位址，而 Ind-Channel 的每一個 Channel 可以獨立存取，所以 Syn-Channel 又可以稱為 1-Gang Multi-Channel，而 Ind-Channel 則稱為 Multi-Gang。Syn-Channel 的缺點在於較容易造成空間利用度不足，因為小或隨機 request 的寫入無法同時利用到所有 Channel，同時運作的方式就使得沒有資料要寫入的 Channel 的位址空間浪費掉，相對而言，Ind-Channel 架構對不同 Channel 存取的利用度較高，但由於每個 Channel 要記錄的位址資訊不同，因此消耗的 RAM 較大，且為了使各個 Channel 的存取時間較平衡，其韌體演算法的實作考慮也會比較複雜。

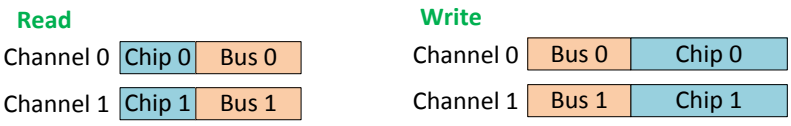


圖 20、Channel 運作示意圖

Interleave 又稱為 Shared-Bus，因為其存取的不同 Chip 的 Bus 共用，因此 Bus delay 無法重疊，運作示意如圖 21，寫入與讀取都會有 Chip 等待的時間，因此 Flash chip 的選擇與 Bus 的傳輸速度對 Interleave 的影響較大，像是 MLC 會比 SLC 佔便宜，因為其 Chip busy 的時間較長，相對而言平行賺到的時間較多，而 Bus 傳輸越快，圖中的等待時間就越短，其平行度就越高，也越接近 Multi-Channel。

Iterleave 的韌體實作方法與 Ind-Channel 類似，因為其不同 Chip 的運作也可以視為獨立，因此如何使各個 Chip 的存取平衡也值得討論。

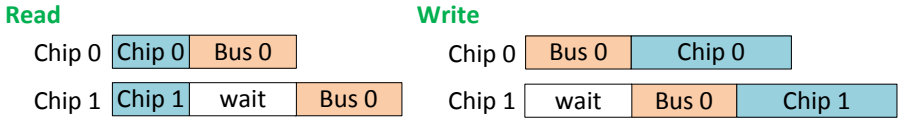
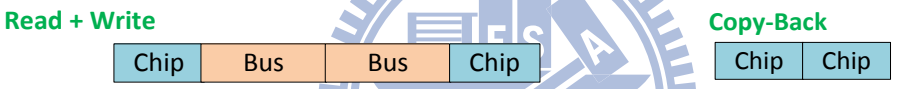


圖 21、Interleave 運作示意圖

4.2.3 Intra-chip Operation

Intra-chip 是透過 Chip 內部 register 加速 Chip 運行的指令，通常有 Multi-Plane 和 Copy-Back 兩種。圖 22a 是一般的資料搬移方法，等同於一次讀取加上一次寫入，而圖 22b 是 Chip 內建的 Copy-Back 指令，其透過內部的 register 協助搬移資料，不需要經過 Bus，因此一次 Copy-Back 就等於先讀後寫的兩個 Chip busy 時間元件。Copy-Back 有效增加 SSD 效能，但其缺點是不經過 Bus 的資料無法經由 Contoller 加上 ECC 等除錯機制，因此大大的增加資料遺失或錯亂的風險。



a、讀取後寫入　　b、Copy-Back 指令

圖 22、資料搬移運作示意圖

Plane 是 Chip 內的單位，每一個 Plane 有自己的 register，因此資料的讀寫和抹除在相同 Chip 的不同 Plane 可以同時運行，但是因為一個 Chip 僅會有一條 Bus，所以不同 Plane 的 Bus delay 無法重疊，如圖 23，Multi-Plane 的平行度類似 Interleave，不同點在於其 Plane 的運作時間要同時，所以已經準備好的 Plane 必須等待另一個 Plane 的資料傳入 register 才一起運作，RAPT 在實作上為每一個 Plane 建立虛擬暫存器（Virtual Register），使用者可以單獨存取某一個 Plane，但此時未被存取的 vitual register 會進入等待，直到所有 Plane 的 virtual register 都被存取後才會執行這個 Chip 不同 Plane 的提交動作（Commit），經 commit 後的 Chip 才算完整執行過 Multi-Plane 指令，這樣的設計可以讓使用者在使用上更有彈性。Multi-Plane 的缺點是共用 CE，所以不同 Plane 同時存取的位址也必須一樣，這個限制使得 Multi-Plane 無法成為提升效能的關鍵。

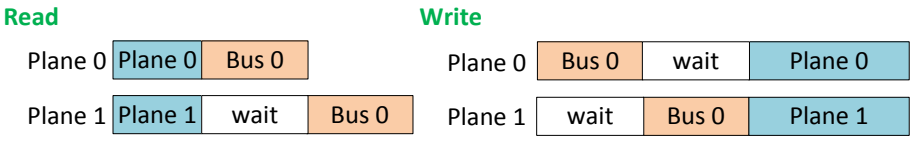
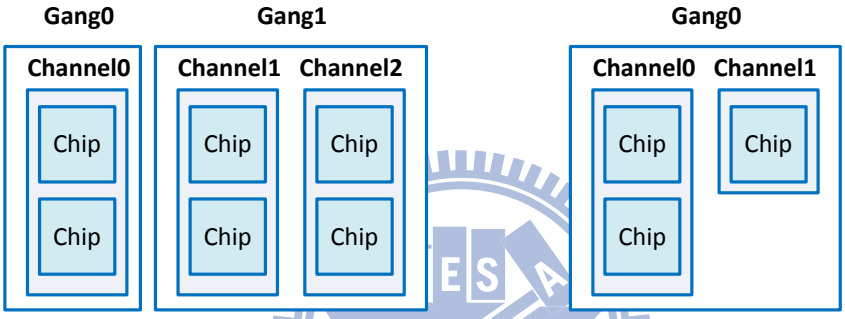


圖 23、Plane 運作示意圖

4.2.4 Deficiencies of the current Hardware Module

本篇所定的各種硬體架構與指令足以模擬現存大部分的 SSD，但是仍然無法滿足推陳出新的 SSD 設計，例如目前 RAPT 的硬體模組是以單一種類 Chip 的設計為主要考量，但有些研究提出與硬碟混合的架構[16]，或者以 NAND Flash SLC 作為快取（Cache）而 MLC 作為儲存空間的概念[17][18]，還有為了適應不同作業系統高頻率存取的 Hot-data 而經過特殊處理的硬體架構等，像是圖 24 所舉例的一個 Gang 中包含不同的 Channel 數量或一個 Channel 中包含不同 Chip 數量，這類的設計目前並不常見且韌體的演算法會相對複雜，所以本篇仍然以對稱性的硬體架構為主題。



a、不同 Channel 數量的 Gang　　b、不同 Chip 數量的 Channel

圖 24、非對稱性 SSD 架構

4.3 Software Module and User Interface

為了增加使用者輸入與輸出的方便性，RAPT 規劃軟體模組特別處理這部分，如圖 25，其包含行為描述檔處理器（Trace Player）和模擬結果產生器（Result Generator）兩個小模組，而 Trace player 中又分為 Request Loader 和 Request Queue 兩個部分，RAPT 使用讀寫系統行為描述檔（Trace）的方式做模擬，使用者可以採用任何格式的 Trace 或不同的 workload，並自行修改讀檔方式，然後透過 *BufferRequest* 這個 API 將 request 放到 Request Queue 中，再執行 *ProcessRequest* 平行處理那些 request，藉此模擬 Buffer 的功能。同時也可以在 Result generator 中自定要輸出的統計資訊，或者採用 RAPT 提供的預設輸出 API，執行 *DefaultOutput* 後 RAPT 會將回應時間（Response Time）、效能（包含 Thoughput 和 IOPS）以及每一個 Chip 的讀寫與抹除次數等資訊輸出到一個預設的輸出檔。

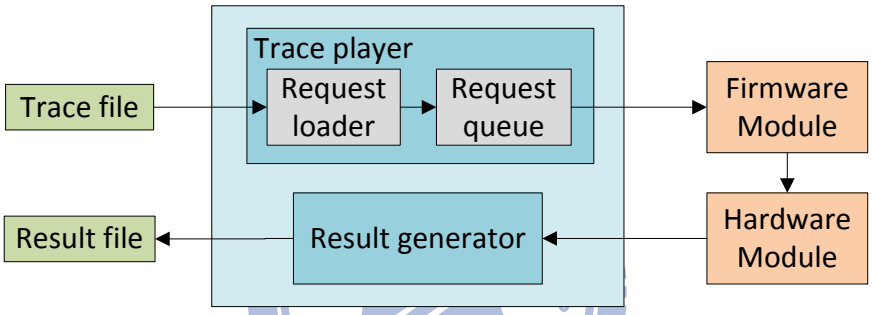


圖 25、RAPT 軟體模組架構圖

開放式的模擬工具都會提供一個窗口給予使用者設定模擬對象的行為，但是大部分的工具卻沒有提供和底層溝通的介面，表示使用者要修改或取得模擬的內容就必須看懂全部的程式碼，為了解決這個問題 RAPT 建立一套完整的介面，採用物件導向的 C++撰寫而成，如圖 26 所示，分為開發函式（Design API）和開發平台（Design Area）兩個部分，補述如下。

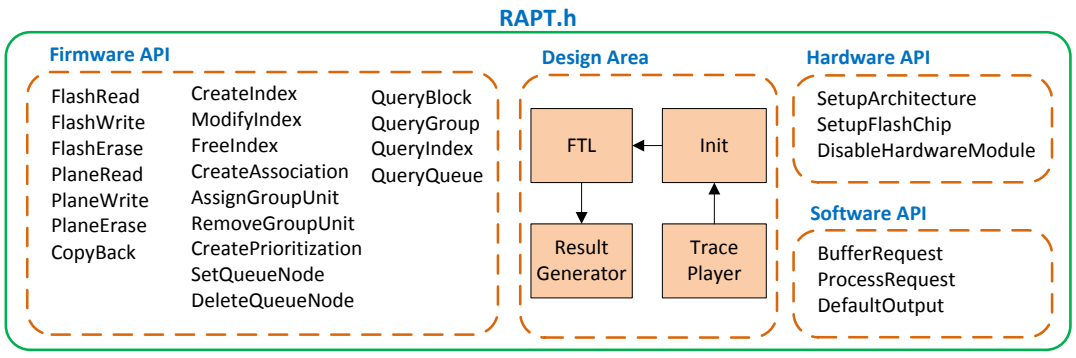


圖 26、RAPT 使用者開發介面

開發函式即為韌體、硬體和軟體模組的 API，韌體 API 可以分為指令（Command）、抽象模組（Abstract Model）和資料查詢三個部分，指令像是 FlashRead、FlashWrite、FlashErase、Multi-Plane 和 Copy-Back 等，而抽象模組則是 3.1 所述的 Index、Association 和 Prioritization，資料查詢的 API 提供使用者獲取由 RAPT 所記錄的一切資訊。硬體 API 主要用來設定 SSD 的硬體架構，包含架構參數和 Flash chip 的規格等。軟體模組則包含暫存和處理 request 的 API 以及輸出預設統計資料的 API，透過這些介面使用者可以在開發與設計上可以更加清楚且方便。

開發平台就是使用者的設計檔案 Design file，內容包含使用者自定的韌體演算法和硬體架構，其分為四個設計主要設計區塊，分別是 Trace Player、Init、FTL 和 Result Generator，Trace Player 和 Result Generator 提供軟體 API 供給使用者規劃輸入 Trace 和模擬輸出資訊，其中 Buffer 的實作也包含在 Trace Player 中，而 Init 提供韌體和硬體 API，主要是作為使用者初始化硬體架構和韌體演算法的區域，而 FTL 中提供給使用者韌體 API，是主要設計 SSD 韌體演算法的地方。

這些分散的模組資源被統一規劃到使用者介面中，所有開發函式和開發平台皆建立為 RAPT.h 函式庫檔中，因此使用者可以很方便的引用，達到 RAPT 快速與容易開發的目的。

5. EXPERIMENT

本章首先在 5.1 提出各項會影響模擬的指標參數，並呈現 RAPT 與真實硬體的比較，接著再以 RAPT 平台開發各種不同的 SSD 韌體演算法與硬體架構，5.2 將說明各種實驗環境的設定，5.3 到 5.6 再分為四個部分依序討論各種 SSD 研究，第一部分討論各種 FTL 的設計，第二部分討論各種硬體架構的變化和瓶頸，第三部分再探討硬體架構實作需要關注的韌體設計議題，最後一部分結合各種韌體演算法與硬體架構的設計組合作探討。

5.1 Simulation Validation

由於實際硬體的時間不確定因素太多，且必須在對韌體與硬體完全了解的情況下才有辦法做模擬，因此目前仍然沒有關於 SSD 模擬正確性的討論，本節首先針對各種可能的模擬影響參數做討論，再呈現 RAPT 與實際 SSD 的比較結果。

5.1.1 Timing Spec Analysis

圖 27 為 SSD 的硬體溝通示意圖，其溝通分為指令和資料的傳輸，其中資料的傳輸是影響 SSD 效能的主要因素，因此在模擬上 RAPT 主要針對這點做考量，至於其他未考慮的時間參數，就為模擬的誤差值，底下就 RAPT 模擬可能的誤差做分析。

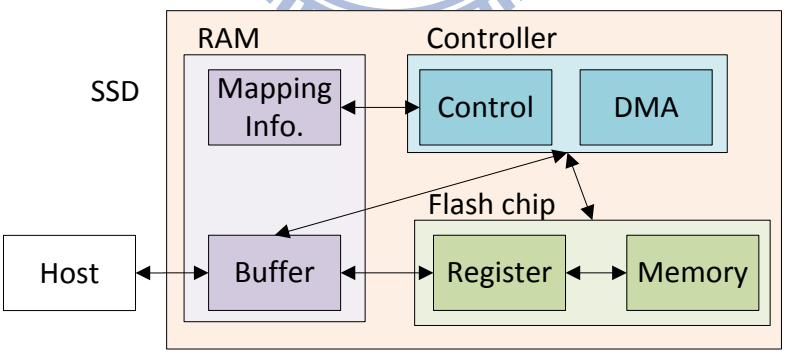


圖 27、SSD 硬體溝通示意圖

SSD 的內部溝通行為包含 Controller 指令、 RAM 的存取、DMA 的資料搬移延遲、Buffer 和 Flash 暫存器的資料傳輸以及暫存器與 Flash memory 的資料傳輸等，分列如表 2，打勾者表示 RAPT 有加入的模擬參數，其餘未加入模擬的參數總影響力僅佔不到每一次讀寫時間的 1%，以 DRAM 為例，其多用於 mapping 資訊的存取，因此每次讀寫的存取不超過 2 Byte，所以至多也只需要花費 120 ns 的時間，對整體的模擬不會有很大的影響。其中 Flash bus delay 的時間瓶頸可能

為 Host 介面、DMA 傳輸速率、Bus 頻寬或 Flash 暫存器，Host 介面的影響像是 SATA 或 USB 等，而 DMA 速率則依處理器不同有所不同，Bus 的寬度一般為 Byte 或 Word，前幾者的傳輸速度都較快，因此本篇的模擬還是以 Flash 暫存器的延遲作為瓶頸考量。

大部分的模擬工具都著重在細節的時間參數，但通常那些參數對結果影響不大，且為了考慮過小的時間單位，容易造成模擬的速度下降，因此 RAPT 目前僅針對時間影響大於 1 us 的參數做討論，其餘時間參數對 RAPT 的總影響僅在正負 1%。

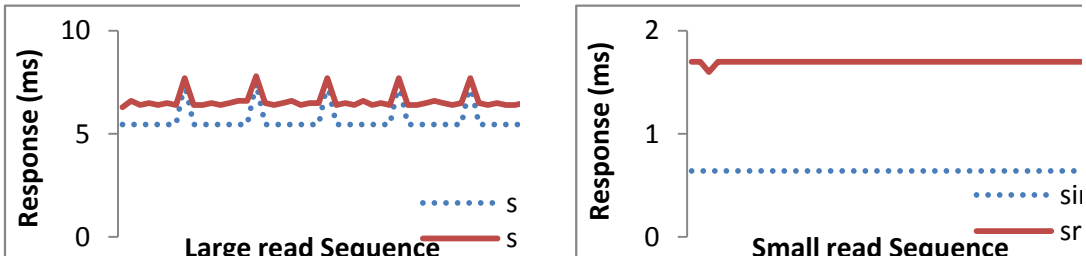
表 2、時間影響參數對照表

	Simulated behavior	Time delay
	DRAM access delay per cycle (t_{RC})	60 ns
	SRAM access delay per cycle (t_{RC})	10 ns
	Controller CMD latch (t_{CS}+t_{CH})	25 ns
	Controller address latch ((t_{ALS}+t_{ALH})x5)	85 ns
V	Flash bus delay = R/W cycle (t_{RC}, t_{WC})	25 ns
V	Flash read busy (t_R)	60 us
V	Flash write busy (t_{PROG})	800 us
V	Flash erase busy (t_{BERS})	1500 us
	Flash multi-plane write busy (t_{DBSY})	500 ns
	Flash spare area read busy (t_{AR}+t_{REA})	30 ns

5.1.2 Verification

為了驗證本篇所提出 RAPT 的正確性與可用性，本小節根據台灣 SSD 開發廠商提供的韌體演算法和硬體架構與其真實產品做比較。首先針對單一 Chip 的韌體演算法比較，採用的美光科技（Micron）的 NAND Flash，規格為 MLC，FTL 則採用加入 pending 機制的 Block Level，為了觀察驗證的細節，使用 HD tune [26] 工具記錄每一筆 request 的回傳資訊，藉此針對細部的回應時間做比較，比較結果就讀取和寫入兩種行為討論，又分為 64KB 大資料和 4KB 以下小資料的存取。

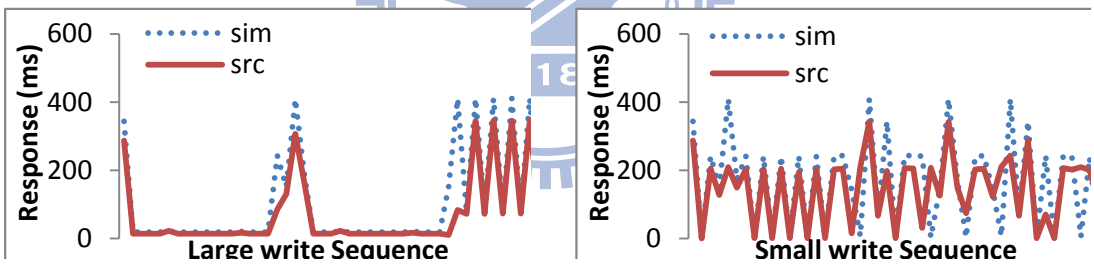
圖 28 為讀取行為的驗證結果，其中大檔案存取的不規律跳動曲線是讀取行為跨越 block 時會去重新載入 RAM 中的 mapping 資訊的原因，此項已加入模擬考量，圖中虛線部分表示模擬行為，實線部分為實際 SSD 行為，可見讀取的趨勢兩者大致相同，但模擬時間約略少實際時間 1 ms，這段誤差來源主要是 Flash bus delay 的傳輸，如同 5.1.1 所述，其影響因素很多，因此要確認讀寫的瓶頸往往需要煩雜的交叉比對驗證，暫時不在本篇的討論範圍內。



a、大檔案存取（64KB）　　b、小檔案存取（4KB）

圖 28、韌體演算法讀取驗證

圖 29 為寫入行為的驗證，可見在小檔案中的回應時間較不規則，因為小檔案存取對 BL 造成較大的負擔，回應時間較低的部分是 BL 做 pending 優化的結果，由於頻繁的 block 回收動作與寫入交錯，因此使得小檔案相較於大檔案在每一個 request 的驗證較不容易，但其 response time 曲線的變化與實際 SSD 走向雷同，其中因為模擬過程沒有根據資料搬移成本考量是要做 pending 或者做 merge，與實作方式略有不同所以造成部分曲線不雷同。綜合而言，讀取行為的模擬時間小於實際 response time，而單一 page 寫入行為的模擬時間卻大於實際時間約 3 ms，這也受到 bus delay 傳輸速度的影響。



a、大檔案存取（64KB）　　b、小檔案存取（4KB）

圖 29、韌體演算法寫入驗證

除了針對單一 Chip 的 FTL 驗證，本節也驗證在不同硬體架構下 RAPT 的表現是否符合預期，其採用的 FTL 以 KAST 為基礎做變化，Over provisioning 將近 15%，使用 MLC 的 Flash chip，並且採用本篇所提及的平行演算法。表 3 就兩種不同的架構測試，使用 IOmeter 純隨機的 workload 寫入，Time 表示測試的時間長短，分為 3 分鐘和 30 分鐘的測試，結果以 IOPS 呈現，其中前 3 分鐘大部分的資料被 spare block 吸收掉，因此效能會表現得較好，如表所示，在僅有 2-Plane 的架構下模擬與實際差異不大，誤差約 5%，但當架構變複雜以後誤差就上升到 15%，因為高平行架構的模擬需要考慮韌體與硬體實作的細節，而少部分細部資訊是本篇無法從廠商取得的。

表 3、硬體架構寫入驗證

	Time	Source IOPS	Simulation IOPS
1-Channel 1-Interleave 2-Plane	30 min	250	262
4-Channel 4-Interleave 2-Plane	3 min	1039	894
4-Channel 4-Interleave 2-Plane	30 min	474	554

本節的測試包含很多影響因素，像是未知的韌體錯誤、ECC 造成的重新讀寫時間、RAM 的存取以及未知的硬體溝通時間等，所以數據的誤差仍然存在，但也可以由驗證中得知，所有模擬結果與實際差異的趨勢相同且符合預期，所以 RAPT 對於 SSD 的測試與設計是有效的，未來將與本計畫開發的真實 SSD 做比較，可以得到更完整且正確的驗證資訊。

5.2 Environment

本篇除了討論 SSD 設計的概念並開發一套快速模擬工具，也針對 SSD 韌體演算法和硬體架構作一連串的探討，有關 SSD 的討論已持續好幾年，近年的研究不管是 FTL 還是硬體架構的同質性都越來越高，因此本篇一方面歸納整理以往的方法，一方面根據這些方法結論表現最好的韌體與硬體組合，也希望透過這些研究給予 SSD 的設計更多啟發，同時從這一章也可以驗證 RAPT 的功能性與準確度。

實驗部分採用之各項參數如表 4，Flash chip 規格取自 Samsung NAND K9XXG08UXA 和 K9XXG08UXM，分別代表 SLC 與 MLC，傳輸速度方面假設 DMA 與 Bus 的延遲很短，因此瓶頸為 Flash chip 內部 register 的讀寫（換算後約 40 MB/s），另外，所有 Log-based 的 FTL 在沒有特別說明的情況下一律以 workload 大小的 5 %作為閒置空間也就是 spare size。未計的時間參數的部分包含 RAM 以及 page spare area 的存取，還有處理器下指令（Command）的延遲，由於這些時間十分微小，平均每一個 request 影響不到 1 μs，所以忽略與否並不會干涉結果。

表 4、實驗參數設定

	SLC	MLC
Page / Block size	2 KB / 128 KB	4 KB / 512 KB
Read latency	25 μs	60 μs
Write latency	200 μs	800 μs
Erase latency	1500 μs	
Data transfer delay	25 ns/byte	
Spare block area	5 %	

SSD 一般用於高效能要求的系統，本篇假設實驗對象的 SSD 為個人用儲存系統和伺服器儲存系統，個人用的儲存系統像是筆記型電腦或 PDA 等，因為個人使用習慣大多不一致，因此可能的存取行為較混雜，包含較多的高頻率存取資料（Hot Data）和眾多的隨機（Random）與循序（Sequential）資料，本篇以使用者在 Windows XP 和 Ubuntu 9.04 作業系統上的行為表示，兩者分別採用 Diskmon [19]和 Blktrace [20]擷取約一個月的存取行為。而伺服器系統就是資料庫系統，其種類繁多，但共同特徵是存取行為遠比個人用系統極端，本篇簡單分為隨機與循序兩種，前者是採用 IOmeter [21]產生的小 request 隨機存取行為，後者是採用 Diskmon 擷取使用者在 Windows 上對多媒體影音的存取行為。

因為讀取的動作較單純，不會影響 GC 等機制，所以本篇僅討論寫入的行為，上述的 workload 也都是單純的寫入，表 5 是其詳細分析資料，其中 Windows

和 Ubuntu 的表現十分相近，都具有固定量的隨機、循序資料和 hot data，不相同的是 Windows 的 request 對齊（Align）的比例較高，也就是說每次是由 page 中第一個 sector 開始寫入的 request 較多，這樣可以使空間利用度較高，FTL 在處理時也會比較容易。此外由於 Ubuntu 與 Windows 的檔案系統（File System）不一樣，因此 Ubuntu 的 request 都是 4 KB 的倍數，且最大 request 可以到達 512 KB，而 Windows 只有 64 KB。至於 IOmeter 和 Multimedia 都是低頻率存取的資料（Cold Data），但前者包含的都是 4 KB 大小的隨機資料，而後者大部分都是 64 KB 大小的循序資料。

表 5、Workload 分析結果

	Windows	**Ubuntu**	**IOmeter**	**Multimedia**
Disk size	40 GB	30 GB	15 GB	20 GB
Data transferred	81.19 GB	70.96 GB	18.05 GB	20.16 GB
Request count	7,428,600	3,718,295	4,731,296	352,353
Avg. request size	11.46 KB	20.01 KB	4 KB	60 KB
Sequential ratio	39.45 %	28.56 %	0.61 %	93.99 %
Rewrite ratio	72.77 %	57.32 %	2.25 %	9.24 %
Align 2K	22.49 %	0 %	24.69 %	98.78 %
Align 4K	5.45 %	0 %	12.35 %	96.91 %

假設系統在飽和狀態（Warm-Up State），SSD 中的邏輯位址填滿資料，但備用區域是空的，而所採用 workload 的資料量都可以填滿備用區域，因此所有的測試將是有效的，同時假設所有的 workload 的 request 為連續不斷，因此 Buffer 不會有閒置。接下來數據的效能都以 IOPS（IO per Second）呈現，每個 workload 的 IO 大小如表 5 的 Avg. request size。

5.3 Exploration of FTL Design

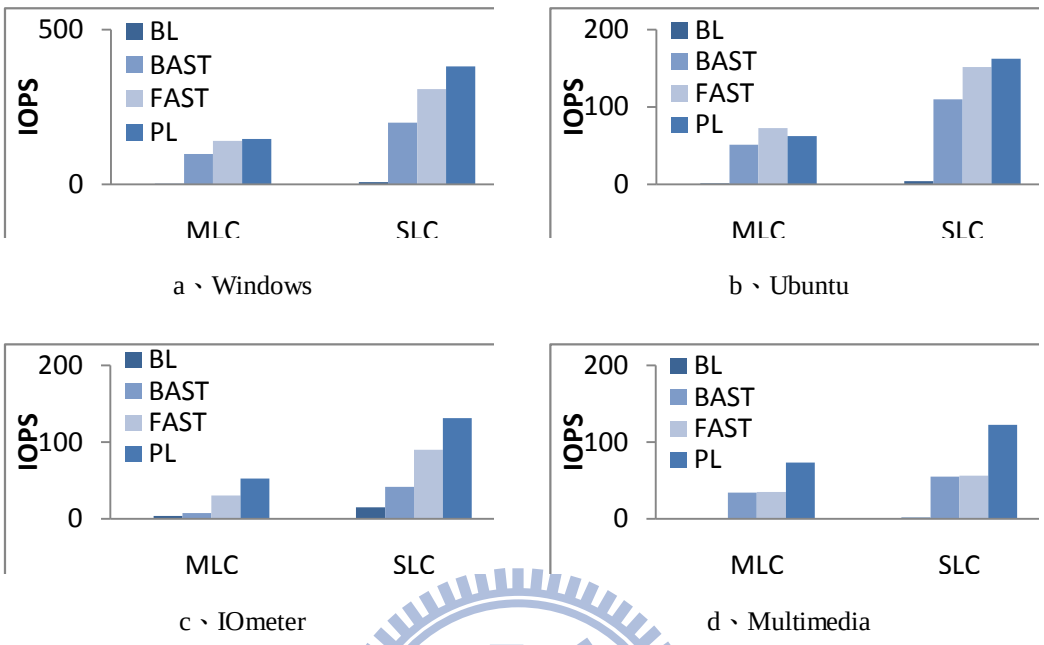
在 SSD 韌體中最重要的就是 FTL 的設計，其 Address Mapping 方法影響 RAM 的需求量，也就決定了成本多寡，Garbage Collection 影響效能，而 Wear Leveling 則影響 SSD 的壽命，應此本節針對 FTL 各種設計的效能表現做討論，其中多數 Wear Leveling 的實作對效能幫助不大，且其方法與種類過於複雜，並不是本篇討論的重點，因此暫不將其列入討論。本節分為四個小節，第一小節是關於 Address Mapping 的議題，根據位址對應的大小分別討論 Block Level、Page Level 和 Log-based 三種 FTL，第二小節討論各種 FTL 的循序優化機制，第三小節再針對 Log-based FTL 的 data block 與 log block 對應特性做討論，最後一小節討論備用空間大小對 FTL 的影響，同時結論各 FTL 的效能。

5.3.1 Address Mapping Granularities

關於 FTL 的討論首先會想到位址轉換，由 Host 端進來的 request 的位址稱為邏輯位址（Logical Address），由於 Flash 的 Erase-Before-Write 特性，通常會將 request 寫入到不同的位址，也就是實體位址（Physical Address），而為了知道哪筆邏輯位址的資料存在哪個實體位址，必須要建立一個對應表（Mapping Table），此表暫存在 RAM 中，若 mapping 的位址單位越小，要存的資訊就越多，需要的 RAM 就越大，SSD 的實作成本也會越高，稱為微對應（Fined-Grained Mapping），相反的 mapping 單位越大就越省 RAM，但效能也會比較差，稱為廣對應（Coarse-Grained Mapping），前者以 Page Level 為代表，後者以 Block Level 為代表，此處簡稱 PL 和 BL，而混合這兩者的 FTL 則稱為 Log-based FTL，這裡討論的 Log-based FTL 以 BAST 和 FAST 為例。

圖 30 是各 FTL 的效能表現，其中 BL 因為每一次寫入就必須搬移整個 block 的資料，所以表現最差。而 BAST 除了在 Multimedia 與 FAST 差不多，其他的 workload 都遜於 FAST，那是因為其他的 workload 較多隨機存取，因此 BAST 容易發生此處配置 log block 後另一處又必須再配置，而導致 spare block 永遠不夠用的情況，這種情形稱為 Block Threshing。此處 PL 的作法是直接將資料寫入空的 page 中，而 GC 時則用貪婪法（Greedy Method）挑選有效資料最少的 block 回收，同時也加入簡單的冷熱資料分離機制，但 PL 在 Ubuntu 的行為下還是輸給 FAST，這是參雜許多 hot 和 cold 資料的關係，cold page 在 GC 時會被搬到新的 block 和其他 hot page 混合在一起，造成每次 GC 挑中回收的 block 都還是要搬移許多 cold page，而這種情形會受到 block 大小的影響，MLC 的 block 較大，所以混合的情況也會較嚴重，因此表示本篇所實作的冷熱資料分離（Hot-Cold Separation）方法仍有待改進，但可以從隨機資料的寫入看出 PL 的潛力。其次 PL 的 Mapping table 十分佔用 RAM，因此近幾年也提出很多方法討論如何在有

限的 RAM 實現 PL 的效能，像是第二章提到的 Super Block 以及 DFTL [22]等都很具代表性。



a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

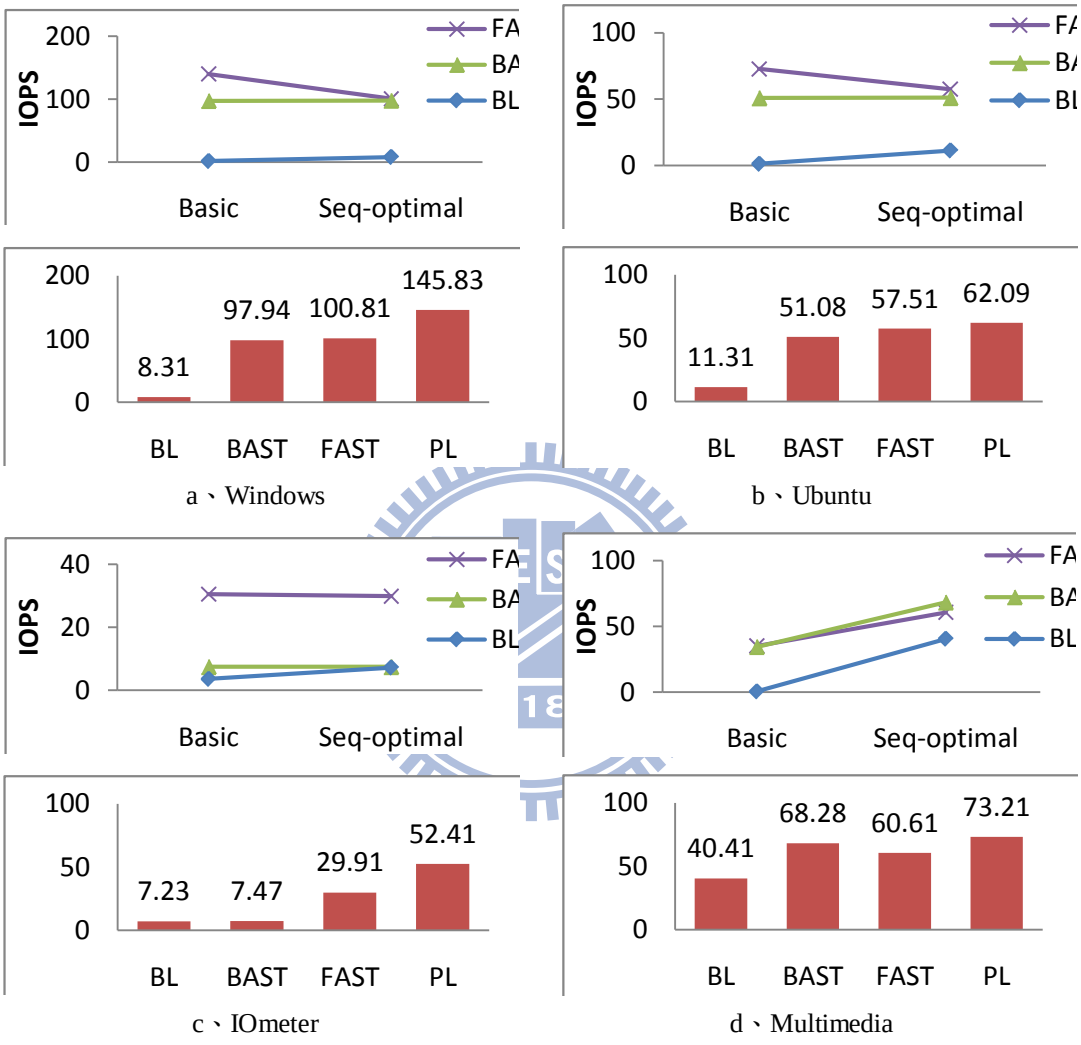
圖 30、不同 mapping 單位的 FTL 效能

綜合而言，Fined-grained mapping 的 FTL 可以減少空間的浪費和額外的有效資料搬移動作，因此在各種存取行為下表現較好，但其大量 RAM 的消耗使得目前大部分 SSD 都採用 Log-based 的 FTL，至於 Coarse-grained mapping 的 FTL 若加上特殊的機制則可以有效利用於隨機資料的存取，將在下一小節討論。

5.3.2 Sequiantiql Optimal

Block Level 和 Log-based 的 FTL 各自有針對循序存取的優化方法。BL 的方法稱為延遲抹除（Pending），BL 每一次寫入都要將 block 中其餘 page 搬移到新 block 的動作對效能破壞很大，有時候明明是可以接續寫的資料，卻還是要依循搬移、寫入、erase 的流程，因此 pending 就是紀錄上一次寫入的 block 位址和寫到的 page，若下一次要寫的 page 在同一個 block 且在上次寫入的 page 之後，則搬移資料到這次要寫的 page 再寫入，而不需要花費多餘的搬移或 erase 動作。Log-based 的 FTL 在 GC 時必須做 merge，而 merge 又分為直接替換（Switch Merge）、部分搬移（Partial Merge）和全數搬移（Full Merge）三種，其中 switch merge 是指要 GC 的 log block 內的每一個 page 及其相對位址都與 data block 內的相同，因此可以直接以其替換原本的 data block，而 partial merge 就是只有前段某些 page 與 data block 相同，所以替換前還是搬移剩下的 page，而 full merge 就是全數的 page 都要搬移到新的 data block，所以 Log-based FTL 優化循序存取的

方法就是盡量製造 switch 或 partial merge。BAST 的做法是每次 GC 前先判斷要 GC 的 log block 是否可以做 switch merge，若可以就直接 switch，而 FAST 則是另外建立一個 sequential block，若寫入為一個 block 的開頭 page，則放進此 block 中，接著每一次寫入都判斷有否辦法做 pending，直到無法 pending 時才 switch 或 partial merge 此 block。



a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

圖 31、加入 Sequential 優化機制後的 FTL 比較

（上圖為與為優化前的比較、下圖為優化後各 FTL 的比較）

圖 31 是加入循序優化機制後各 FTL 的比較，由於 MLC 與 SLC 趨勢大致相同，這裡僅列出 MLC 的結果，圖中可見 FAST 在前三個包含較多隨機存取的 workload 上表現的都比未優化前糟糕，圖 28a 的 FAST 甚至表現比 PL 還要糟，因為以 block 開頭作為循序資料的依據十分容易誤判，但在 Multimedia 存取行為下三者的表現都很好，而在圖 31c 中 BAST 處理循序資料的能力較優異，因為 FAST 單一的 sequential block 無法處理混合的循序資料，而 BAST 是以每一個 data block 區分，所以即使循序資料被斷開也可以在之後接續起來。另外，BL 在

IOmeter 下優於 BAST，一方面是因為 BAST 在純隨機的 workload 下會發生嚴重的 block threshing 問題，一方面是因為大部分的 request 沒有 align，雖然是小 request 卻會寫入兩個以上的 page，而 BL 的 pending 機制可以將這些 page 視為是同一個 request。

FAST 的 sequential block 機制其實並不完善，過多的誤判容易造成隨機存取的效能低落，若要改善其誤判的機率，則可以加入額外的判斷機制例如 Request Size Filter 等，另外為了避免循序與隨機或循序與循序交錯資料帶來的影響，也可以透過增加 sequential block 的個數來改善。。

5.3.3 Group Association Exploration

由於 Log-based FTL 的效能與資源使用比較平衡，因此目前仍然是最廣為討論的 FTL，其主要差異點在於 data block 和 log block 的對應個數（Association），於是才有 N-K 的討論出現，當 N=1 且 K=1 時，就類似於 BAST，而當 N=所有 data block 且 K=所有 log block 時則類似 FAST，但兩者還是有實作上的差異。

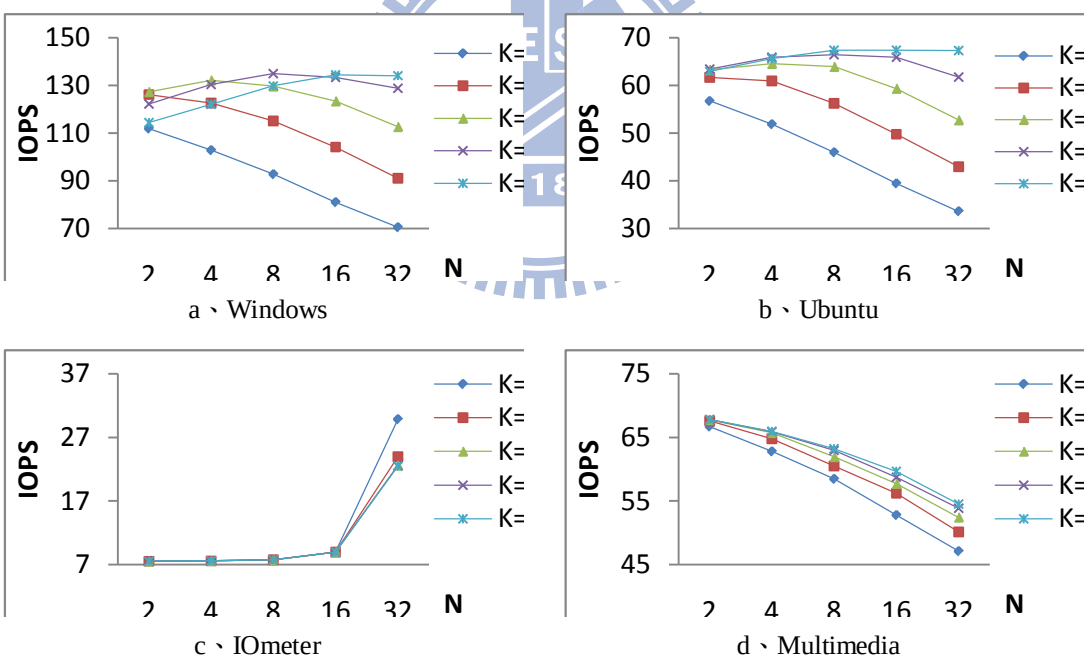


圖 32、Log-based FTL 中 N 與 K 的對應關係

圖 32 表示在限制 K 個 log block 的前提下 N 值變化的效能，由於 MLC 單位數值的結果變化較明顯，故以其為參考，其中圖 32a 和圖 32b 的趨勢十分相近，在 K 值夠大的前提下 N 值與 IOPS 的變化成正比，因為 Windows 和 Ubuntu 的資料更新頻率與密度都很高，所以較大的 K 值才可以吸收 hot data，而加大 N 值則可以避免 block thrashing 發生的機率，此處的 spare size 為 workload 存取範圍的

5%，表示 N 大於 20 時就一定不會發生 block thrashing，此點在圖 32c 更加明顯，因為 IOmeter 是純隨機的資料，因此 N 小於 20 的時候效能都很差，反之 Multimedia 是循序的資料，所以 N 值越小越不容易受到參雜的隨機或循序資料影響，越有機會做 switch merge。歸納而言，Log-based FTL 中 N 值應與資料的空間區域性（Spatial Locality）成反比，K 值則與資料的時間區域性（Time Locality）成正比，兩者巨觀來說就是單位範圍內資料存取的頻率，以本篇的 workload 舉例，Windows 的最佳 N-K 為 8-16，Ubuntu 的最佳 N-K 為 16-32，IOmeter 為 20-1，而 Multimedia 為 1-1。

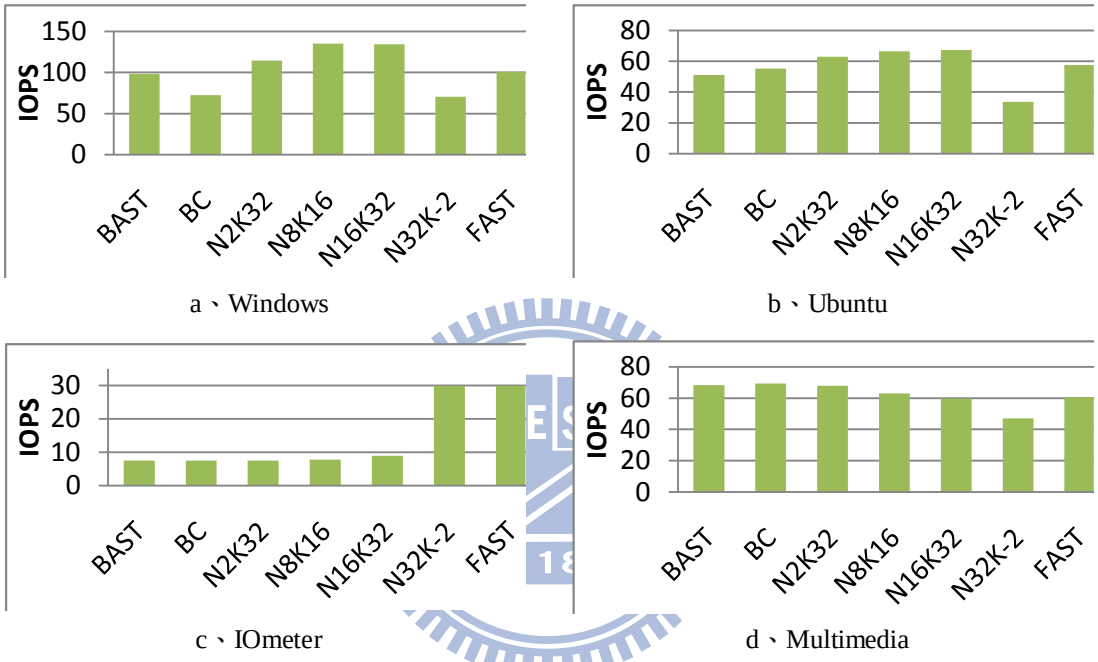
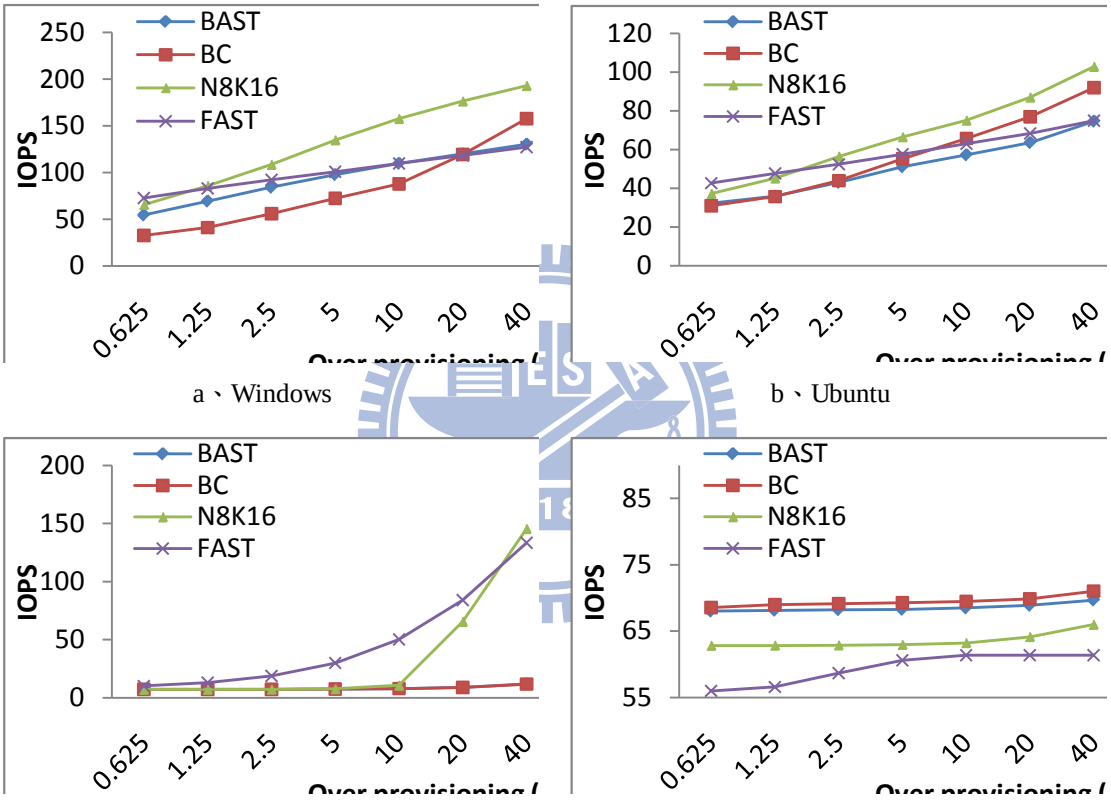


圖 33、Log-based FTL 中 N 與 K 的對應關係

此處取上述各 workload 的 N 與 K 最佳值再跟 BAST、BC 和 FAST 做比較，其中 BC 為 Block Chain，是 BAST 的變型，其 N=1，但 K 值沒有限制，在圖 33a 和圖 33b 的部分 N 與 K 的值越大越好，因為其 hot 與隨機的資料都很多，但 FAST 表現反而不如大部分的 N-K 設定，這是因為前述其循序優化誤判機率太高的關係，不過也因為循序優化的關係使 FAST 在 Multimedia 中表現不會差太多。另外在 Windows 中 BC 的表現較差，但在類似的存取行為 Ubuntu 中 BC 表現卻又比 BAST 好，此情形也可以由圖 33 發現，這是因為 Windows 中 hot data 存取的位址範圍較廣，也就是說 N 值要大到某個範圍才能足夠蒐集到 K 個 block 的 hot data。截至目前的 N-K 討論都以 MLC 為主，若是 SLC 則 K 值的設定要增加 4 倍，因為 SLC 的 block 比 MLC 小 4 倍，所以 K 值也要調整才足夠容納相同大小的 hot data，但是總空間不變所以 block 數量在 SLC 上也會相對變多，因此 N 值可以不需要更動。

5.3.4 Overprovisioning Ratio vs. FTL Performance

除了 N 與 K 對應的特性之外，Log-based FTL 與 BL 和 PL 不一樣的地方在於它必須有額外的 spare size 做為 log 的使用，因此相同大小的空間，使用者真正能使用部分會隨著 spare size 提升而減少，但也因為 log 的空間更大，所以得到的效能也較好，但如何取得使用者與 spare 區域的平衡，這類討論我們稱為備用區覆蓋率（Over Provisioning）的問題，簡稱 Op，Op 等於 spare 空間除以使用者可用空間，以百分比表示。圖 34 是各種 Log-based FTL 在不同 Op 下的效能表現，資料以 MLC 為參考，其中 N8K16 是取先前測試中表現較好的 N-K 值的 FTL。



a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

圖 34、不同 Over provisioning 下 Log-based FTL 的效能

圖 34 顯示，隨機存取的 workload 在 Op 小的時候，以 FAST 的表現最好，因為其他 FTL 發生 block thrashing 的情況比較嚴重，而圖 31a 和圖 31b 中 NK 8-16 在 Op 低時還是與 FAST 差不多，因為這兩個 workload 多數的 hot data 會將設定的 K=16 個 log block 填滿而做 GC 的關係。但在 IOmeter 中 NK 8-16 除非 Op 提升到 20%以上不然無法吸收掉龐大的隨機存取，而 BAST 和 BC 在 IOmeter 中的表現一模一樣（標線重疊），即使 Op 到達 40%效能還是很低落。Multimedia 中除了 FAST 以外的 FTL 的 IOPS 都會隨著 Op 成長而增加，這是因為 FAST 的循序存取效能是靠一個 sequential block 維持，因此和 Op 的變化比較沒有關係。整

體而言，各 FTL 各有不同的表現，但 FAST 比較不會受到 workload 的影響，可以隨著 Over provisioning 的變化下穩定成長。

結論各種 FTL 的比較關係，整理如表 6，除去 RAM 消耗度過大的問題，PL 還是表現最好的 FTL，相對而言 BL 效能過差所以不作為 SSD 考量，多用於循序存取行為多的隨身碟或記憶卡，而 Log-based FTL 則可以 BAST 和 FAST 為代表，由前面的測試可知，BAST 較適合循序存取的資料，FAST 則具有優異的隨機存取效能，而隨機存取又是一般 SSD 無法避免的，所以 FTL 的設計以 FAST 為考量再加上循序存取的優化會比較好，不過 FAST 也有每次 GC 的 Association 過高導致回應時間過長的缺點，因此 FTL 在 SSD 上的使用必須根據不同的用途而定，而更多關於 Log-based FTL 的變型與討論在持續，像是改良 N-K 為不連續 data block 的 KAST，還有依照 request 大小分類後再以 BAST 或 FAST 處理的混合體 LAST [23]都是著名的 FTL，這些 FTL 都是為了解決前述的缺點，讓 SSD 可以適用於各種不同的存取行為。

表 6、FTL 關係對照表

（由優至劣為 Excellent > Good > Bad > Worst）

	Sequential Access	Random Access	RAM Consumption	Another Drawback
Block Level	Good	Worst	Small	
BAST	Excellent	Bad	Medium	Block thrashing
FAST no seq	Bad	Excellent	Medium	Long response
FAST	Good	Good	Medium	Long response
Page Level	Excellent	Excellent	Large	Hot-cold mixed

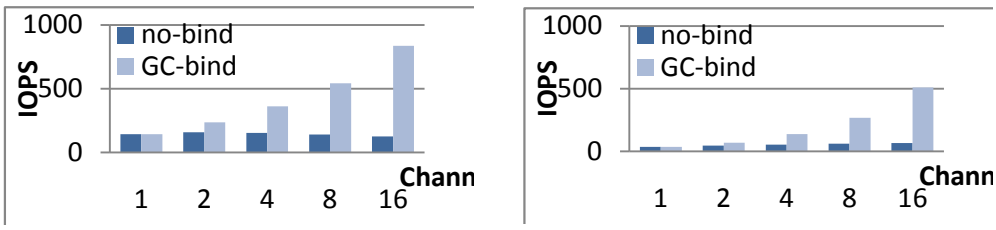
5.4 Exploration of Architecture Design

硬體架構的設計可以分為 Inter-chip 和 Intra-chip 兩種優化方法，5.4.1 討論 Multi-Channel 實作的演算法，5.4.2 和 5.4.3 針對 Inter-chip 的架構討論，其又分為 Syn-Channel、Ind-Channel 和 Interleave，而 5.4.4 和 5.4.5 討論的是 Intra-chip 的指令，包含 Copy-Back 和 Multi-Plane，而 5.4.6 和 5.4.7 則是結論各種硬體架構的比較。接下來所採用的 FTL 皆以 5.2 中表現較穩定的 FAST 為例，spare size 同樣是 5%，預設的 Buffer 大小為 Channel 個數乘以實際 page 大小，意即假設採用 MLC 的 Chip 搭配兩個 Channel，則 Buffer 大小為 2 乘以 4 KB 等於 8 KB，是 Channel 同時傳輸資料的最小容許設定，並採用 FIFO 機制，其餘更多關於 Buffer 的作用與分析將在 5.5.1 中討論。

5.4.1 Parallel Algorithm in Multi-Channel

Inter-chip 架構的設計指的是 SSD 中 Chip 和 Chip 之間的排列組合，在這類討論中，以 Syn-Channel 的討論最簡單，其做法相當於放大一個 Flash chip 的 page 和 block，但讀寫這樣大的一個邏輯 page 的傳輸時間卻與讀寫單一 page 無異，因為每一個 Channel 都有一條 Bus 的支援，這樣對於循序存取的效能非常有幫助，但若是隨機或不平衡（Unbalance）存取的資料，則很容易發生利用度過低的問題，所謂不平衡的現象表示資料集中存取某一個 Chip 或某一個 Channel，若只讀寫某一個 Channel 的資料，在 Syn-Channel 的架構下其他 Channel 就算沒有資料也必須同時讀寫相同的位址，所以造成空間的浪費，效能也會因此受影響。

為了解決 Syn-Channel 的問題，就有另一種將 Chip enable line 和 Bus 都分開的架構，使得不同的 Chip 可以獨立運作，稱為 Ind-Channel，這種架構的平行度與 Syn-Channel 一樣高，且可以存取不同位址的資料，因此可以解決 Syn-Channel 的問題，面對隨機的存取可以有效的提升每個 Channel 的利用度。



a、Windows　　b、Multimedia

圖 35、平行演算法效能表現

Ind-Channel 透過各元件之間的平行作用，往往可以達到數倍的效能，但要令所有 Chip 的工作量相同，才可以有最佳的平行處理能力，這樣的設計方法有

很多種，而且不同的 FTL 適合的方法也不一樣，我們稱這種方法為硬體的平行演算法（Hardware Parallel Algorithm），舉例來說，在 FTL 中最耗費時間的動作是 GC，所以其中一個重點就是在做 GC 時，其他元件也必須同時做 GC 以避免閒置，但不同的元件要做怎樣的 GC 也是值得討論的，這裡針對 Log-based 的 FTL 設計，其特點是具有 data block 和 log block 的概念，在 GC 時必須 merge 這兩者，因此其中一種平行處理的演算法就是 merge 不同硬體元件中的相同 data block，如此一來可以避免在做 GC 時某些硬體元件的閒置，二來透過這種被綁住的 GC，也可以減少下次 GC 的負擔，圖 35 中顯示採用此種演算法的效能改善，採用 Chip 為 MLC，workload 為 Windows，圖中可 s 見若 GC 沒有同步則 Channel 越多反而會因為頻繁卻不同時的 GC 造成效能低落。

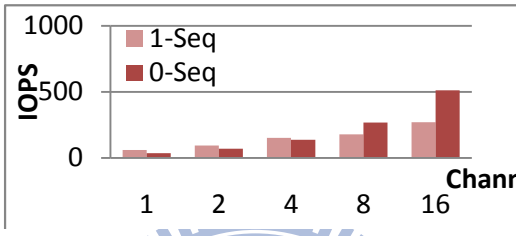


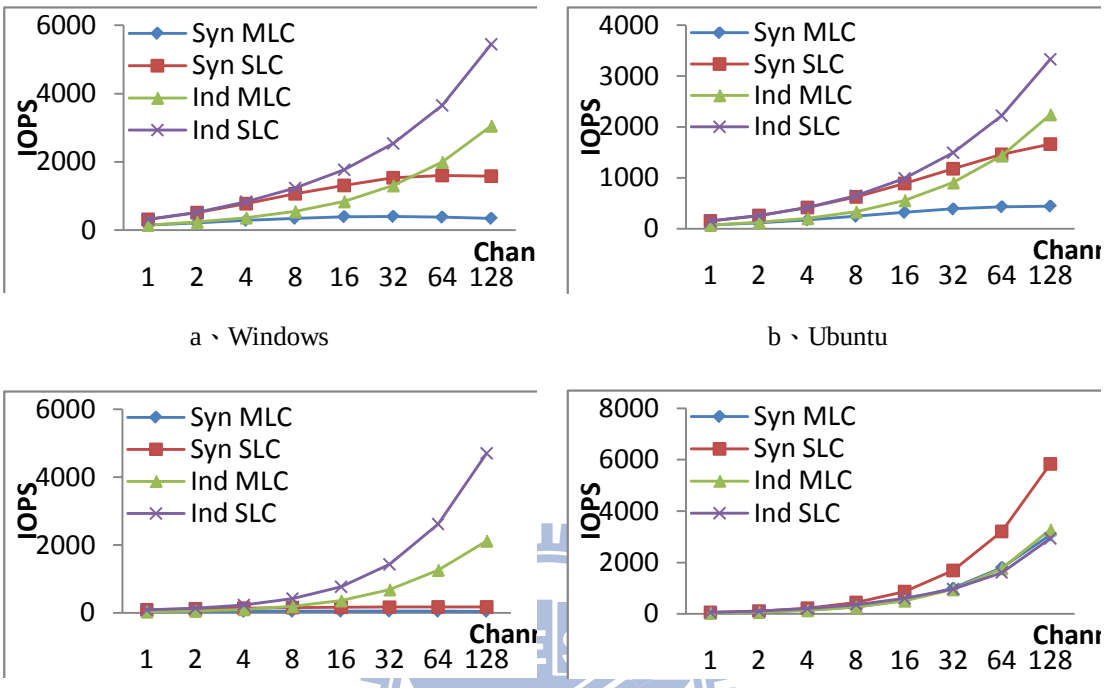
圖 36、FAST 循序優化在平行演算法中的表現

上述平行演算法對於 FAST 循序優化的實作也是一樣的，當其 sequential block 要 GC 時，其他 Chip 也必須 GC 相同對應的 data block，但是那個 block 不一定也是 sequential block，可能是 log block，所以會發生某些 Chip 要做 full merge 但某些只要做 partial merge 的現象，容易破壞平行演算法的時間利用度，圖 36 以 MLC 在 Multimedia 上的表現為例，當 Channel 數量越多上述的缺點越明顯，在 8 個 Channel 後就比沒有加循序優化還要糟，若在非循序存取的 workload 則誤判造成的空間利用不完全更會隨 Channel 增加而加倍，因此在多 Chip 架構的討論本篇以未做循序優化的 FAST 為主。

5.4.2 Synchronized-Channel vs. Independent-Channel

本小節比較 Syn-Channel 和 Ind-Channel 的效能表現，如圖 37，橫座標是 Channel 的個數，其中 Syn 指的是 Syn-Channel 架構，而 Ind 則為單純 Ind-Channel 架構。在 a、b 和 c 三個 workload 中都是 Ind-Channel 表現比較優異，是因為隨機的存取會讓 Syn-Channel 產生嚴重空間利用度不足的現象，其中又以純隨機的 IOmeter 最明顯，也因此其隨著 Channel 數上升效能提升十分緩慢，在 16 個 Channel 左右就發生瓶頸，而在 Multimedia 中 Syn-Channel 表現較好有兩個原因，第一個是循序的資料不容易造成空間利用度不足的情況，第二個原因是 Ind-Channel 採用的 data block 同步 GC 方法，其 log block 多數時候沒有辦法一起回收，因此雖然減少 Association 的成本，但 log block 的 erase 還是無法平行，這

種不平衡隨著 Channel 數量增加後就愈來愈大，其中又以 SLC 特別嚴重，因為 SLC 的 block 比較小，但 erase 的時間和 MLC 是一樣的，所以相對而言 erase 的成本比較高，這算是 FAST 演算法在 Independent-Channel 架構上的特例。

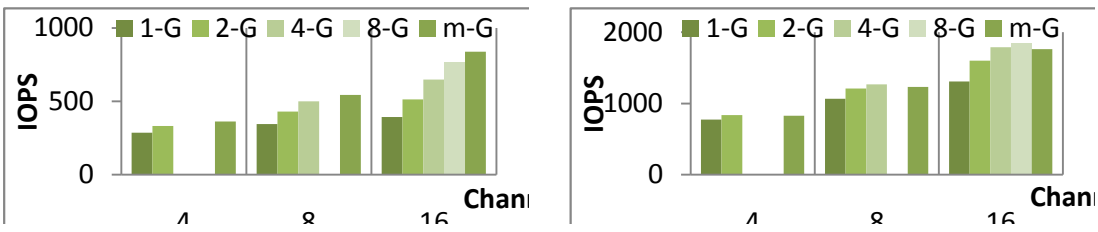


a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

圖 37、Synchronized 和 Independent-Channel 的比較

除了上述單純的架構，Syn-Channel 和 Ind-Channel 也可以結合為混合的架構，圖 38 是針對此項作討論，1-G 表示 1-Gang，即一個 Gang 中有全部的 Channel，而 2-Gang 的每一個 Gang 中包含總 Channel 數除以 2 個 Channel，依此類推，圖中列 Windows 為參考，可見在 16 個 Channel 以前，MLC 的表現還是以純 Ind-Channel 架構的效能最好，但 SLC 以兩者結合的效能較佳，原因即前述 log block 的回收不平衡，由實驗得知，整體而言 Ind-Channel 還是 SSD 的最佳架構，雖然在循序存取的的部分其效能較差，但那是因為平行演算法不夠完善的緣故，本篇不在此處多做贅述。



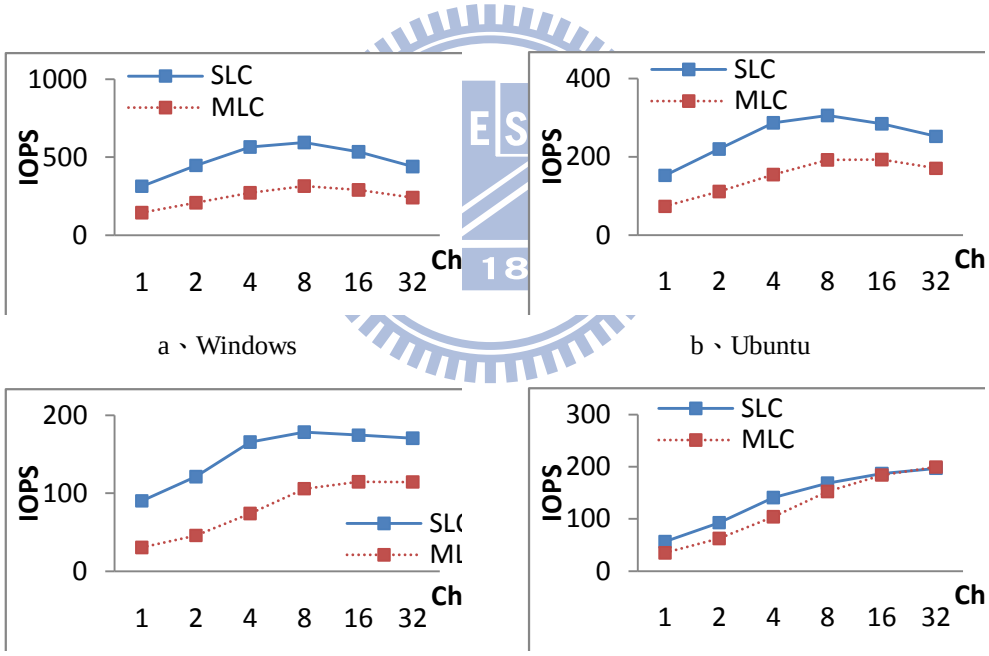
a、MLC　　b、SLC

圖 38、混合式 Multi-Channel 在 Windows 上的表現

5.4.3 Interleave Architecture

Interleave 也就是 Shared-bus 的架構，其共用 Bus，但每一個 Chip 會有各自的 CE，因此也需要採取平行演算法去解決 GC 不平衡的問題，此處採取與 Ind-Channel 相同的平行演算法，整體而言，Interleave 平行度沒有 Channel 高，隨著 Chip 增加而 spare block 減少，其效能的增進也會逐漸降低，平行演算法發生的不平衡也會越來越嚴重。

圖 39 是 Interleave 在單一 Channel 上的效能表現，研究顯示大約在 4~8 個 Chip 的時候其效能就會遇到瓶頸，在 Multimedia 中因為循序的大筆資料存取使得各個 Chip 的利用較平均，所以效能還有提升的空間，而 MLC 效能下降前容忍的 Chip 比 SLC 多則是因為 Interleave 的 Chip busy 是可以同步的，而 MLC 的 Chip busy 時間較長（約比 SLC 慢 3~4 倍），所以其效能瓶頸發生前允許的 Chip 數量較多。



a、Windows　　b、Ubuntu

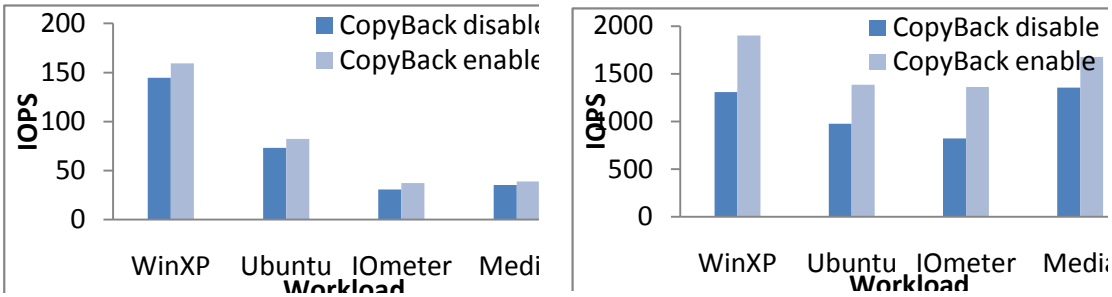
c、IOmeter　　d、Multimedia

圖 39、Interleave 的效能表現

5.4.4 Copy-Back Operation

Intra-chip 是 Flash chip 所提供的優化，主要藉由韌體指令控制硬體的行為，常見的像是 Copy-Back 和 Multi-Plane 指令，由於過去鮮少研究討論其平行度與優劣，因此本篇特別獨立兩個小節簡要的呈現兩者效能。首先，搬移 page 的行為在 Flash 中是必要的，以 Log-based FTL 為例，在做 block 的 merge 時，勢必

要將要合併的資料搬移新的 data block，因此產生 Copy-Back 指令使得搬移 page 的行為可以直接透過 Chip 的 register 完成而省下 Bus 傳輸的時間。



a、1-Chip　　b、16-Gang 16-Channel 4-Interleave

圖 40、Copy-Back 的效能表現

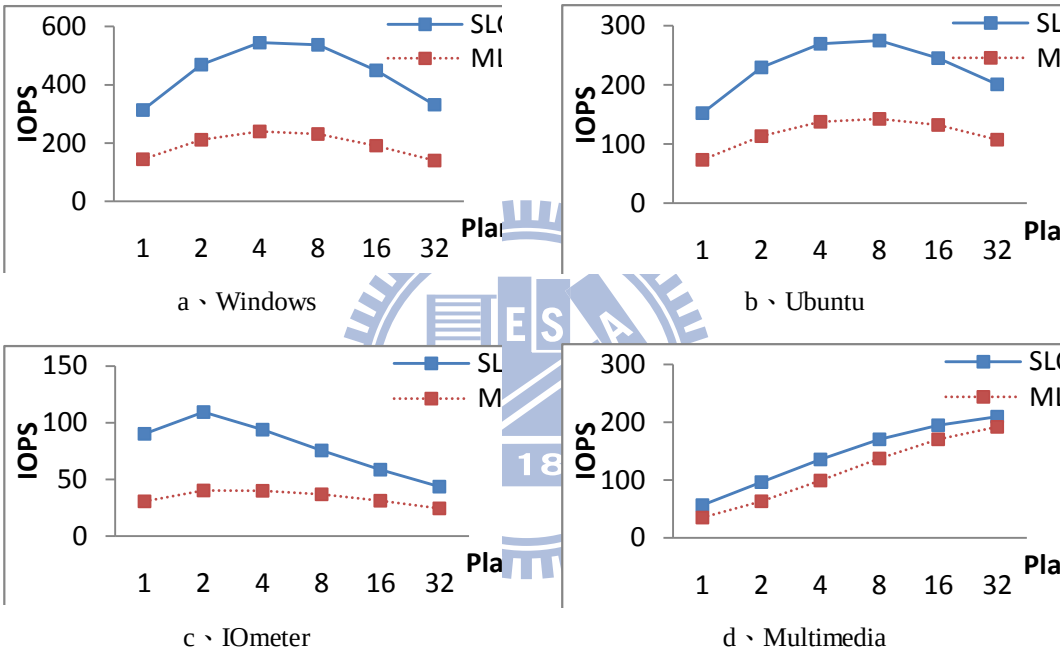
圖 40a 是 MLC 單一 Chip 加上 Copy-Back 指令後的效能，其中表現最好的是 IOmeter，加入後效能增加了 21%，其它 workload 則提升大約 10%，這是因為 Copy-Back 主要用於 Garbage Collection，而在隨機且 cold data 的存取下每次 GC 的 Association 較高，需要搬移的資料量較大，因此 Copy-Back 對它的幫助比較大，圖 40b 則是使用前面幾節表現較好的架構加上 Copy-Back 的測試，可見 Chip 數量增加後效能的提升遠比 1-Chip 明顯，因為 GC 被歸類為額外的負擔，而多 Chip 架構可以分散這種負擔，因此像 Copy-Back 這種減少 GC 負擔的優點也被放大了。Copy-Back 雖然可以很有效的提升 SSD 的效能，而且越高平行度的架構可以獲得的幫助越大，但是由於不將資料讀到 RAM 中，Controller 無法對資料做諸如 ECC 等額外的處理，就包含很大的安全性風險，所以大部分的 SSD 並不採用 Copy-Back 的指令，本篇僅在此提出其價值，接下來的實驗不將其納入討論。

5.4.5 Multi-Plane Operation

相較於 Copy-Back，Intra-chip 的另一個指令 Multi-Plane 卻很常被使用，因為其被內建在 Flash chip 中，使用上很方便，是最小的 SSD 平行單位，Multi-Plane 的平行度與 Interleave 類似，兩者之間的關係就像 Syn-Channel 與 Ind-Channel 的關係，前者共用 CE，必須存取相同的位址，後者獨立運作，可以採用平行演算法增進效能。

圖 41 是 Multi-Plane 在單一 Chip 的情況下提升 Plane 個數的效能表現，針對其採用演算法也有很多種，而通常會將同一個 Chip 不同 Plane 的 block 或 page 位址綁住，以利 Multi-Plane 的運作，此處採用完全綁死的方式，也就類似於 Syn-Channel，一個 Chip 中可以視為有較大的 block 和 page，但由於 Multi-Plane

的平行度與 Interleave 差不多，所以 Plane 越多表現也會越差，如圖在 Plane 個數到達 8 以後就無法再有所提升。Multi-Plane 在各個 workload 的表現也與 Syn-Channel 的架構大同小異，一樣在 Multimedia 中效能提升最多，而 IOmeter 中最差，這是其容易對同時運作的 Plane 造成空間利用度不足或者不平衡讀寫的原因，與其不同的是，因為其存取位址綁死，所以相較於獨立存取的 Interleave 在 IOmeter 中表現較差，因此可以把 Interleave 和 Multi-Plane 想像成平行度較低的 Ind-Channel 和 Syn-Channel。現在大部分的 Flash chip 都只支援 2-Plane 的指令，因此本篇接下來的討論也以 2-Plane 為主。另外，Mult-Plane 的指令除了讀寫和抹除，也支持 Copy-Back，但本篇暫不討論。



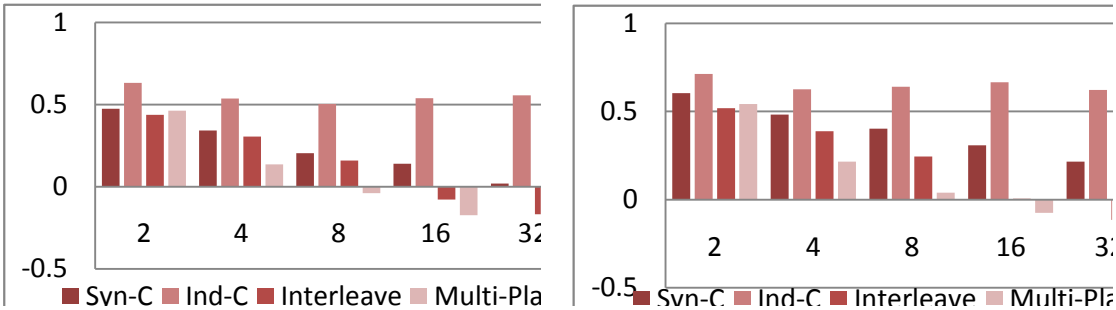
a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

圖 41、Multi-Plane 的效能表現

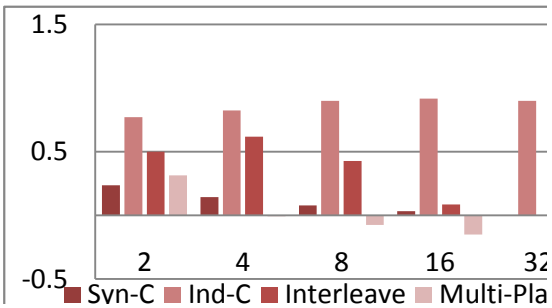
5.4.6 Overall Hardware Design

截至目前，Channel、Interleave 和 Plane 是硬體架構的設計中討論的重點，但當 Chip 或 Plane 數量變多以後這些組合的效能就會因為 spare block 減少或演算法的不平行而有限制，如圖 42 所示，以 MLC 為例，橫座標表示對應的硬體元件像 Channel、Chip 或 Plane，縱座標表示相較於硬體元件減半所提升的 IOPS 比率，例如橫坐標為 2 時表示比起只有 1 個單位時效能提升的百分比。

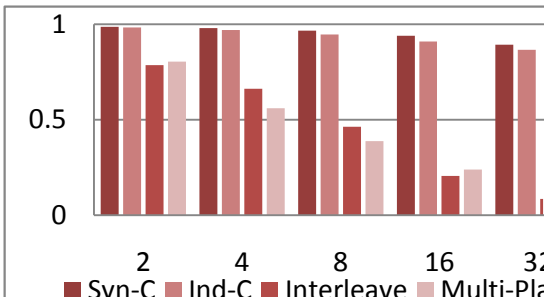


a、Windows

b、Ubuntu



c、IOmeter



d、Multimedia

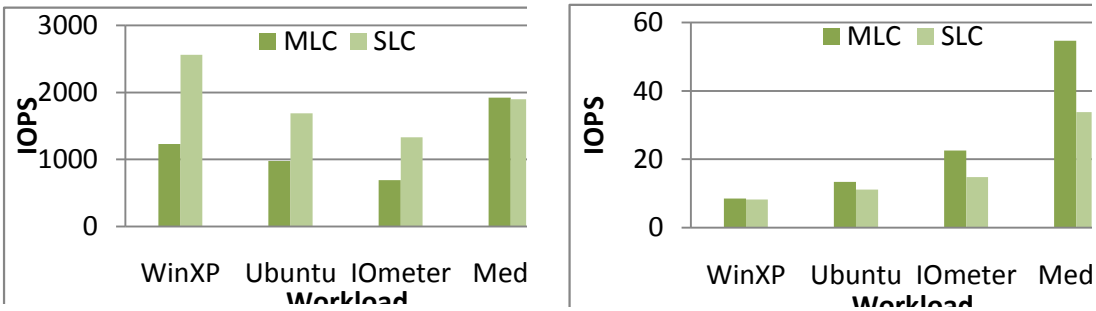
圖 42、各種硬體架構的效能提升率

圖中 Channel 數量的增加也表示對應的 Buffer 大小增加，因此硬體元件數量上升對於 Syn-Channel 或 Ind-Channel 的效能相較於 Interleave 或 Multi-Plane 影響不大，其中 Syn-Channel 在 hot data 較多的 workload 中會有下降後上升的趨勢，是因為 hot data 通常是小筆資料，小筆資料會對 Syn-Channel 造成空間利用度不足的情況，而 Buffer 大約提升到 16 個 page 的大小後就可以吸收掉大多數的 hot data，所以 Syn-Channel 在 16 個以上的 Channel 效能提升較明顯，同理 Ind-Channel 就不會有這個問題，整體而言，除了 Multimedia 增加 Channel 時可以倍數的提升外，其他 workload 每次加倍所得的效能提升都在 50%左右。至於 Interleave 和 Multi-Plane 在 MLC 的表現大約到 8 個單位之後的效能提升就很有限，特別是 Multi-Plane 越多單位表現越差，其兩者在不同 workload 情況下如前面所述與 Ind-Channel 和 Syn-Channel 類似。

5.4.7 Performance Summary

最後基於效能提升、排線寬度（Bandwidth）限制等原因，本篇採取 16 個包含 1 個 Channel 的 Gang 與 4 個 Interleave 總共 64 個 Chip，加上 2-Plane 指令，並使用 Ind-Channel 的平行演算法作為本篇認為的最佳 SSD 架構，圖 43 是此架構在各種 workload 的效能表現，圖 43b 中可見 MLC 在的效能提升幅度比較大，最高在 Multimedia 中可以比未做任何平行的架構升 44 倍，且圖 43a 中在 Multimedia 的也勝於 SLC，原因除了前述平行演算法所造成的 log block 回收不

平衡，也因為 MLC 的 Chip busy 時間較長，所以 Interleave 和 Multi-Plane 在 MLC 上比較占便宜，再基於成本考量，MLC 單位價格較 SLC 便宜，因此在不考慮 Endurance 和 ECC 等的前提下，本篇認為使用 MLC 作為 SSD 的 Flash chip 可以獲得較高的效益。



a、效能表現　　b、相較於未平行架構效能上升倍數

圖 43、最佳 SSD 架構的效能

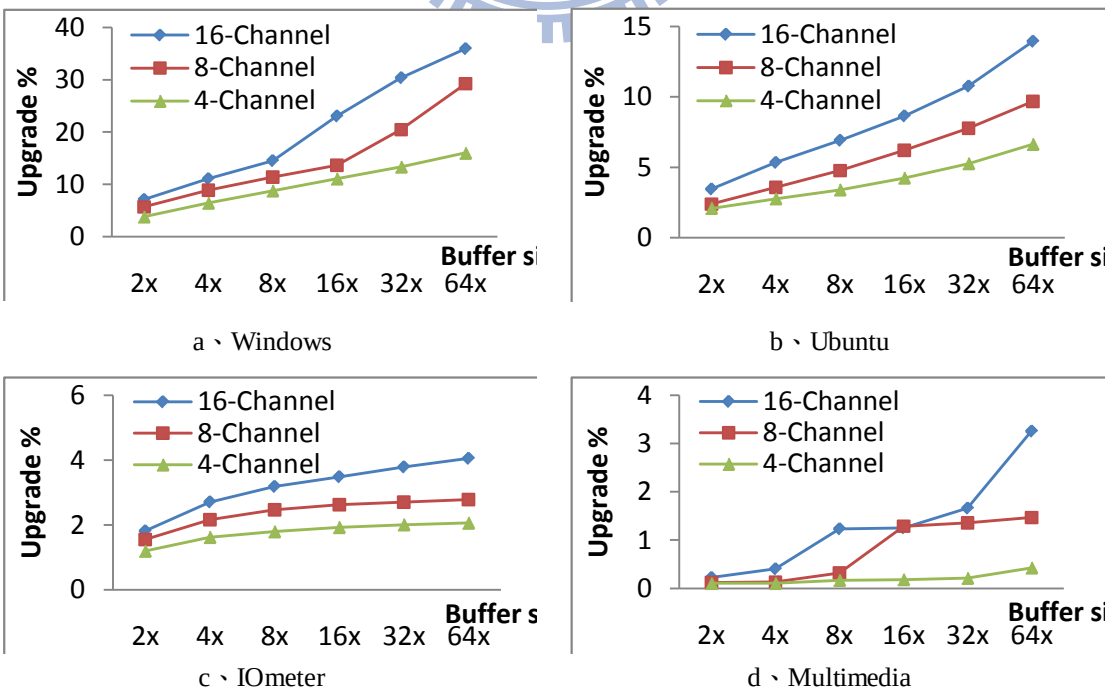
關於各種硬體架構的設計，目的是增加系統平行度，提升 SSD 的效能，由前面的實驗可知，較佔優勢的是可以獨立存取的架構，像是 Ind-Channel 或 Interleave，其設計較具有彈性，在隨機與循序存取下都表現很好，尤其以 Ind-Channel 為最佳的 SSD 硬體架構設計。

5.5 Hardware Design Option

在分別探討過 SSD 韌體演算法與硬體架構之後，本節將結合兩者做討論，分為兩小節，第一小節討論在使用高平行度的 SSD 硬體設計時須要考量的韌體機制，包含 Buffer 大小對平行架構的幫助以及資料擺放在各個硬體元件的位置優先權討論（Data Placement），第二小節探討不同 FTL 包括 BL、PL、BAST 和 FAST 在不同硬體架構像 Syn-Channel 或 Ind-Channel 的表現與瓶頸，最後一小節探討各種韌體與硬體的挑選原則，並根據本節所研究的成果做結論。

5.5.1 Buffer Effect

SSD 為了要使資料同時寫入或讀出不同的 Chip，並且達到 Multi-Channel 的目的，Buffer 的使用是必要的，其功用主要有兩點，第一是作為緩衝區，作用是使資料可以快速寫入以避免回應時間過長，第二是作為保留區，用來提升系統效能，包含的優點此處列舉兩項，首先，Buffer 可以吸收 hot data，並且根據替換演算法（Replacement Policy）的優劣產生不一樣的效能，最簡單的方法是 First In First Out（FIFO）或 Least Recently Used（LRU），較著名的方法則像是 LIRS [24]、ARC [25]等，由於本篇主題不與 Buffer 有關，因此僅以 FIFO 方式實作，另外一點好處是 Buffer 可以減少平行架構 SSD 的資料不平衡現象，例如隨機存取的 request 可以透過 Buffer 一起被寫入不同的 Channel 以提高每一個 Channel 的利用度，由於這些好處使得 Buffer 在 SSD 上已成為必然的趨勢。



a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

圖 44、Buffer 大小對 Ind-Channel 架構的影響

圖 44 是不同大小的 Buffer 對多 Chip 架構的 SSD 的影響，橫軸為採用的 Buffer 大小，表示與原始大小相比加大的倍數，縱軸為與原始 Buffer 大小鄉比所得到的效能提升百分比，一樣以 MLC 為例，採用 Ind-Channel 架構為實驗對象，其中原始 Buffer 大小設定不同架構下的最小值，也就是與 Channel 數量一樣的 page 大小，舉例而言 16-Gang 原始 Buffer 為 16 x 4 KB = 64 KB，而 8-Gang 和 4-Gang 分別為 32 KB 和 16 KB，橫座標和縱座標表示 Buffer 提升倍數後與原始大小比較所提升的效能百分比。圖中可以見 Buffer 越大效能表現都會越好，但在 Multimedia 中提升效果較低，因為它包含循序和 cold 的資料，但當 Buffer 提升到某些大小就會出現效能突然上升的現象，這是因為 Buffer 的大小剛好容納循序與循序資料間摻雜的隨機資料的關係。而表現最好的是圖 44a 和圖 44b，因為除了隨機資料，兩者還包含很多 hot data，又以 Windows 特別明顯。而 IOmeter 的 workload 正好可以看到 Buffer 對資料平衡的幫助，可知 Buffer 大約提升到 4~16 倍後對硬體平衡的幫助就趨緩，但是越高平行架構的 SSD 也會越需要 Buffer 的幫助，因為資料不平衡的現象也會越明顯，如圖所示 16-Gang 的效能在 Buffer 幫助下提升幅度最大，8-Gang 次之，而 4-Gang 最差。

5.5.2 Data Placement

在平行系統下，資料分散（Striping）的概念非常重要，而如何分散資料又可以分為很多種方法，大部分都與資料擺放位置（Data Placement）有關，這類討論在過去的研究也很常見，像 RHP-SSD [8]就研究過此問題。

本篇根據 Channel、Interleave 和 Plane 三種硬體元件的資料擺放優先權做討論，如圖 45 所示，把 Data placement 的方法分為七種，其中 P0 表示完全未做 Striping 的方法，也就是資料直接先由第一個 Channel 的第一個 Chip 的第一個 Plane 開始順序性的擺放，擺完後才放到第二個 Plane，其次是 Chip 和 Channel，這樣的方法就如同單純加大 SSD 的空間，並沒有任何平行度可言。

圖 46 可以清楚看出各種 Data placement 的效能，採用的硬體架構為上小節結論的 16-Channel 4-Interleave 2-Plane 最佳 SSD 架構，以 Ind-Channel 下 MLC 為例，就平行度而言，Channel 高於 Interleave，而 Interleave 又略高於 Plane，但折線圖卻顯示以 Plane 為優先的 P2 和 P5 有較好的效能，這是因為在多 Chip 的架構下，最重要的是韌體處理的平行度，而韌體處理中又以 GC 最花費時間，而 Ind-Channel 和 Interleave 的存取位址是獨立的，使用平行演算法可以使其 GC 同時運行，而此處 Plane 的位址故定，除了 GC 的問題還容易有空間利用度的問題，其優先存取重要性就因此提升，也就是說若沒有優先將資料寫入 Plane，空間利用度所造成的效能損失會遠高於寫入未平行的損失。反觀 Channel 和 Interleave

的優先權則一向是 Channel 較高。各 workload 的表現趨勢大致相同，比較特別在於 Data placement 對 IOmeter 的影響較小，因為它是純隨機的小檔案存取，因此固定的資料擺放對其沒有意義，相反的 Multimedia 下差異則最大。

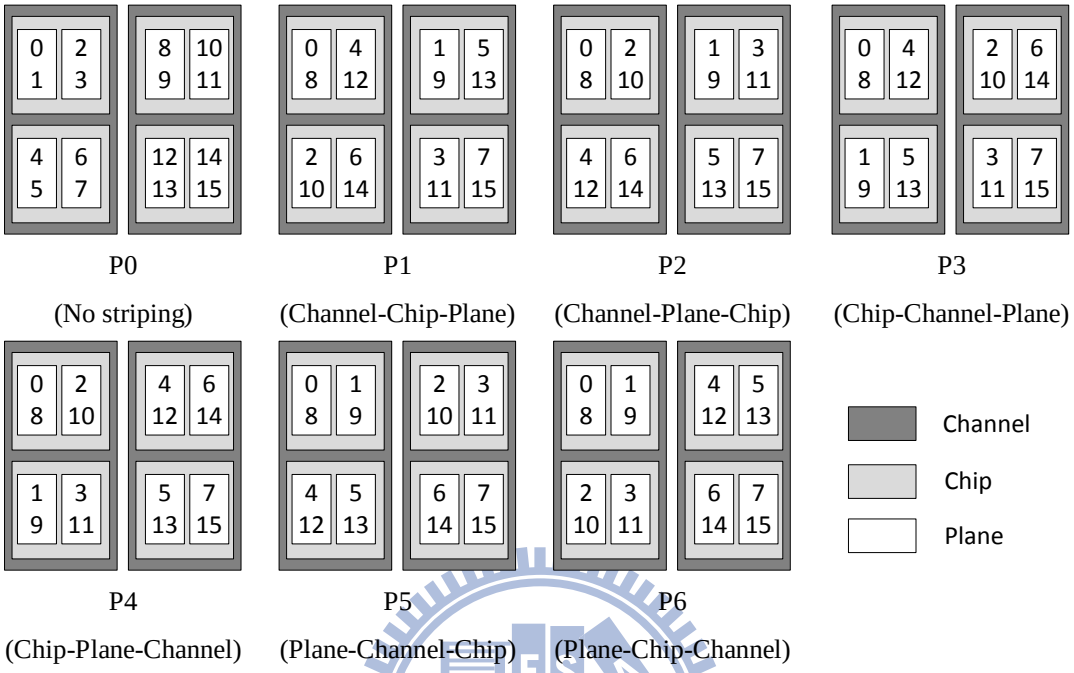


圖 45、Data placement 優先權關係
（高優先-次優先-低優先）

結論而言，本篇認為若採用的 Multi-Plane 方式是位址完全綁死的，則資料擺放位置應以 Plane 為優先，以避免空間利用度不足而使效能下降，其次以較高平行度的 Channel 為優先，最後則是 Interleave 的 Chip。

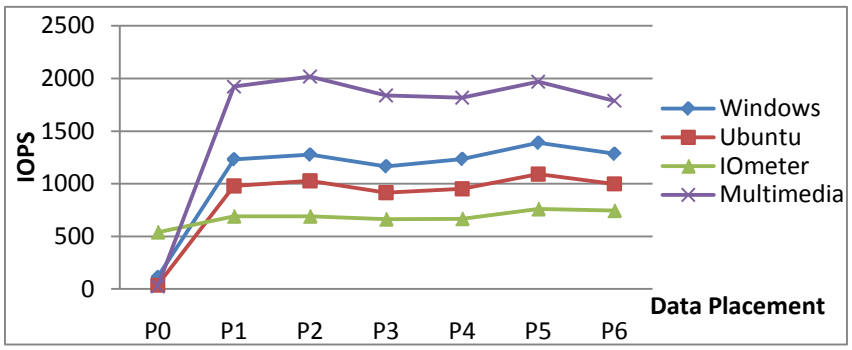


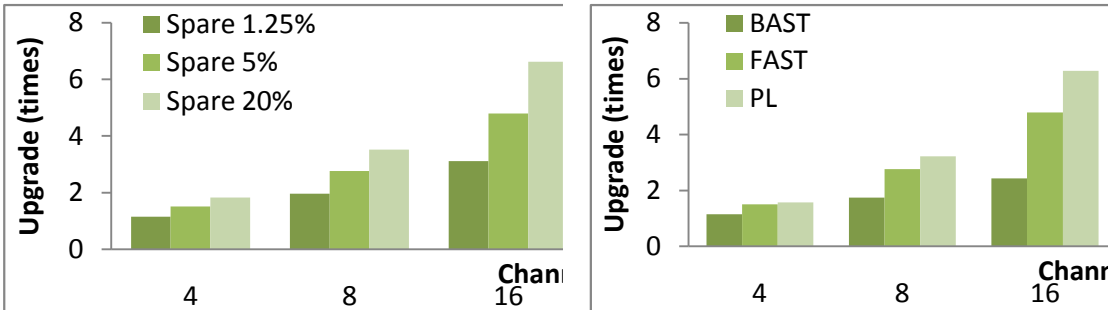
圖 46、Data placement 對效能的影響

5.6 Exploration of Firmware/Hardware Combination

本節在 5.6.1 發現表現越好的韌體設計可以在高平行度 SSD 架構下可以發揮越好的效能，5.6.2 討論不同 FTL 和不同硬體架構的關係，並在 5.6.3 結論兩者的搭配原則，由於不同 FTL 的效能會受到 Over provisioning 大小的影響，因此本小節將 Over provisioning 定為 20%來討論，目的在於希望各個 FTL 的初始效能較一致，另外與 5.3 討論不同的是，此處的研究都有加上基本的 Buffer，且除了 FAST 因為其 sequential block 會拖慢平行架構的效能不採用，其餘的 FTL 都有加上循序優化的機制。

5.6.1 Firmware Design Effect

本小節探討韌體演算法的優劣對於高平行度硬體架構的影響，如圖 47 所示，以 Windows 存取行為下的 Ind-Channel 架構為例，橫座標表示 Channel 的個數，就是架構的平行度，縱座標表示與單一 Chip 架構比較得到的效能提升倍數，分為 Spare 大小和 FTL 兩者討論，我們發現若採用的 Spare 越大或越適合 workload 的 FTL，在平行 SSD 架構上可以得到的效能提升越多，這是因為在 Ind-Channel 架構的平行度主要來寫入平行和 Garbage Collection 平行，當 Spare 增大或 FTL 變好表示 GC 的效能也會上升，所以 Channel 數增多時，那些效能的提升也隨著倍增，這與 5.4.4 越高平行度架構採用 Copy-Back 指令效能提升越多的原因一樣，因此除了表示越高平行度的 SSD 架構設計越需要好的韌體演算法支援，也可以得到越好的韌體設計可以使高平行 SSD 架構的效能提升越多的結論。



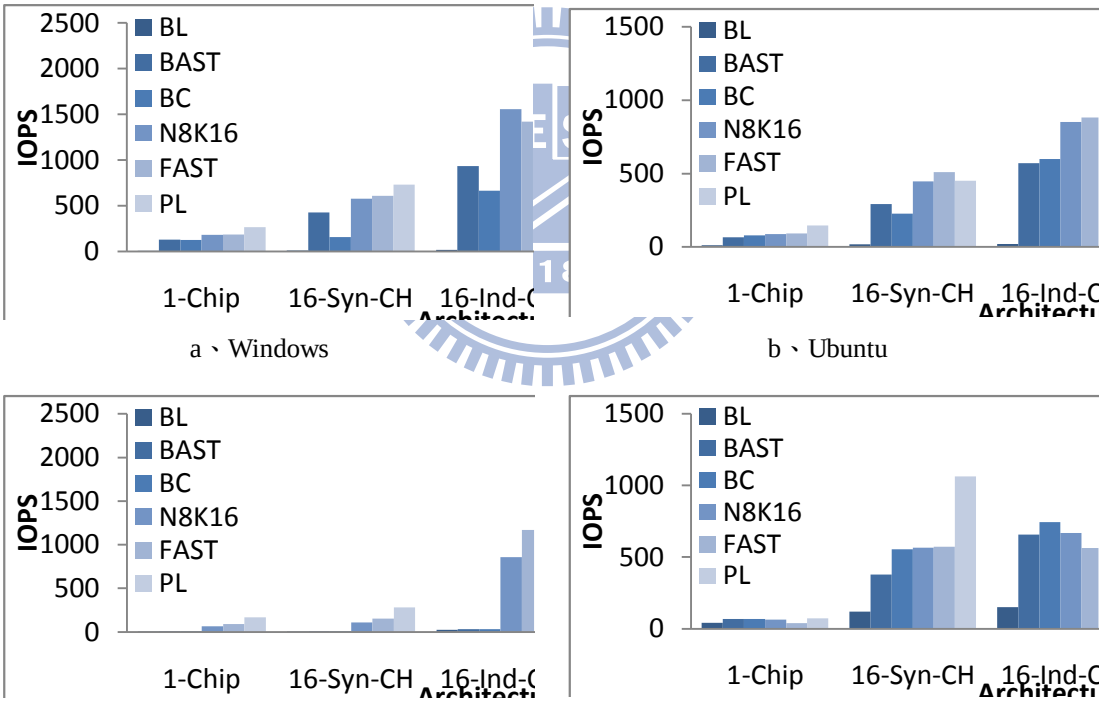
a、Spare szie 的影響　　b、FTL 的影響

圖 47、韌體機制優劣對硬體架構的影響

5.6.2 Different FW/HW Combinations

影響 SSD 效能與設計的關鍵在於韌體演算法和硬體架構，而兩者又分別以 FTL 和 Inter-chip 架構為討論重點，於是本小節針對不同 FTL 與不同 Inter-chip 架構的搭配做討論。

FTL 的實作主要是為了適應不同的存取行為，而不同硬體架構的對同一種韌體演算法的影響是一樣的，以不同 FTL 在單一 Chip、16-Syn-Channel 和 16-Ind-Channel 的表現為例，如圖 48，同樣以 MLC 為例，圖中可見各種 FTL 在同一種架構中的優劣情況都差不多，主宰 FTL 表現的主要因素為 workload，例如在隨機存取下 N816 和 FAST 都表現較好，而 BAST 和 BC 等就因為 block thrashing 的緣故表現較差，但 Multimedia 則相反，其中 BC 因為不限制 log block 個數，所以更容易在隨機存取中將 spare 消耗完，但也在循序存取中包含較大的寫入彈性，因此 BC 在 Multimedia 上優於 BAST 而 Windows 上遜於 BAST。FTL 中表現最好的還是 PL，但其在 Ubuntu 中的 Syn-Channel 架構下還是因為前述 Hot-cold separation 的問題輸給 FAST，因為 Syn-Channel 相當於把 block 加大，使得冷熱資料更難分乾淨。另外，各種架構對同一種 FTL 的影響也都一樣，都以 Ind-Channel 對 FTL 的效能提升最有幫助，所以整體而言 FTL 和硬體架構的設計是可以不互相影響的。



a、Windows　　b、Ubuntu

c、IOmeter　　d、Multimedia

圖 48、各種 FTL 在 Multi-Channel 的表現

由上圖中發現仍然有某些 FTL 在某些架構中無法發揮應有的效能，因此本段針對這些特例做討論，這些特例都發生在 Multimedia 的 workload 中，首先，Ind-Channel 應該是表現最好的架構，但可以發現唯獨 FAST 在 Inde-Channel 的效能略遜於 Syn-Channel，如圖 48d 所示， N8K16 在 Windows 中單一 Chip 的表現不顯眼，但當採用 16-Ind-Channel 時就贏過 FAST，這是因為 5.4.2 所提過的 FAST

平行演算法 log block 的 GC 不平衡造成的特例。另外，在 Syn-Channel 中 BAST、BC 和 N8K16 表現沒有如預期般超過 FAST，這是因為 Syn-Channel 中 Channel 數增加相對於 block 也變大，造成循序資料的存取被切斷而無法做 switch merge 的情況增多的緣故，如圖 49 所示，隨著 Channel 數量增加這種情形也愈明顯，當 BAST 等 FTL 缺乏了循序優化的支援，就容易凸顯 block thrashing 的缺點，因此在這種情況下採用 FAST 還比較有利。這些偶然會發生的特例雖然很少且影響不劇烈，但都需要藉由模擬提早發現，並且採用特別的機制去應對。

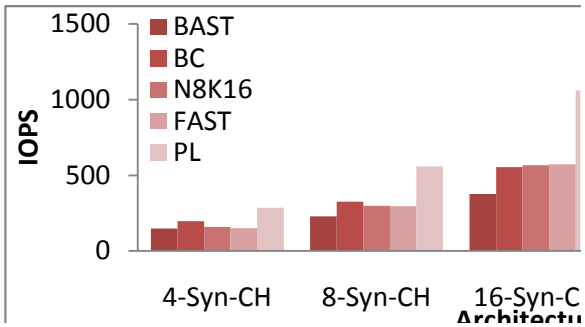
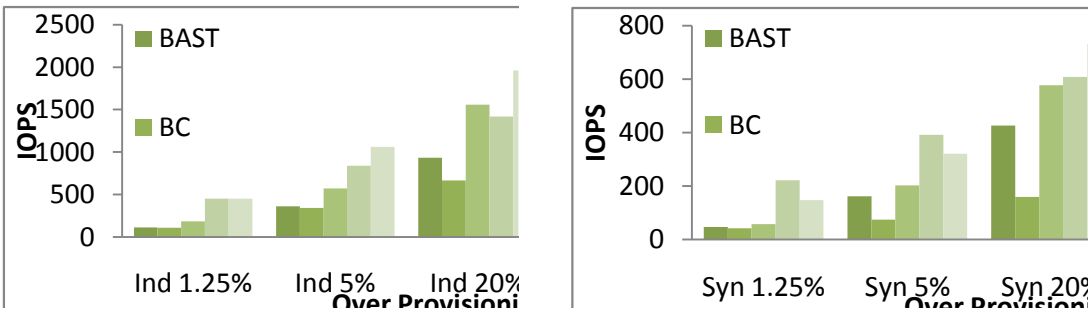


圖 49、Channel 數量對 Synchronized 架構下 FTL 的影響

除了 20%以外此處也加入其他比例的 Over provisioning 做參考，如圖 50，以 Windows 的 workload 為例，當 Op 小的時候除了 FAST 之外其他 FTL 包括 PL 在內表現都很差，BAST、BC 和 N8K16 是因為 block thrashing 的緣故，而 PL 則是因為 Op 變小讓 hot data 來不及累積就要被搬移，使得冷熱資料混合的情形更加嚴重。



a、Synchronized-Channel　　b、Independent-Channel

圖 50、Over Provisioning 對高平行架構下 FTL 的影響

圖 51 是各個 FTL 在 4-Interleave 架構下面對不同 workload 的效能表現，本篇發現其呈現的效能關係與 Ind-Channel 架構完全相同，因為其同樣是存取位址分開運行且採用相同的平行演算法。

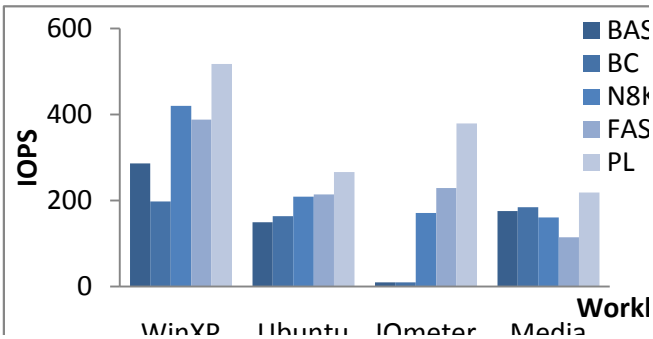
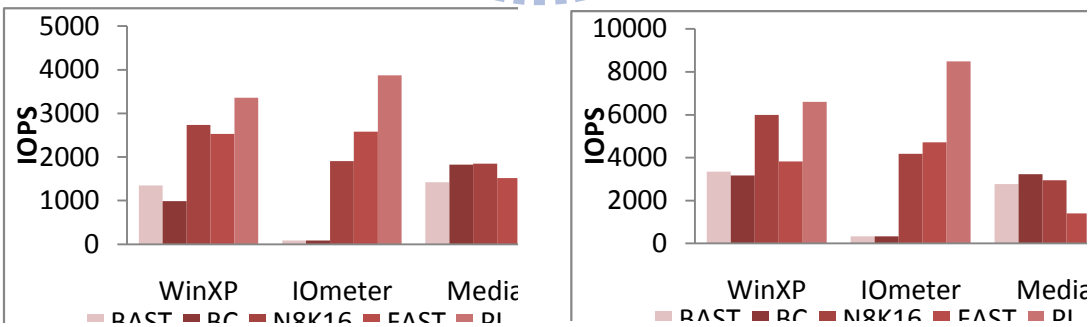


圖 51、各種 FTL 在 4-Interleave 中的表現

5.6.3 FW/HW Combination Principle

最後本篇在本小節討論 SSD 各種韌體演算法與硬體架構的搭配原則，首先結合前述表現較好的架構做測試，採用 16-Ind-Channel 4-Interleave 的架構與各種 FTL 做綜合比較，如圖 52 所示，以 Windows、IOmeter 和 Multimedia 的存取行為呈現，圖 52a 為 MLC，而圖 52b 為 SLC，其中比較需要注意的是 BAST 在 MLC 的 Multimedia 下輸給 FAST，BAST 之所以會在循序存取贏 FAST 是因為其 GC 可以有做 switch merge 的機會，但隨著 Chip 數量越多或 block 越大，循序資料被截斷的情況也會越嚴重，因此越不容易有機會做 switch merge，而又因為 BAST 的單一 log block 使得 GC 的觸發可能很頻繁，因此就略輸於 FAST。但在 SLC 中反倒是 FAST 的表現較不突出，一方面是因為 block 較小使得剛剛的情形不容易發生，一方面還是因為前述 FAST 平行演算法的弊病。



a、MLC　　b、SLC

圖 52、各種 FTL 在 16-Independent Channel 4-Interleave 中的表現

綜合而言，韌體演算法的設計主要 SSD 的用途有關，也就是會受到不同存取行為影響，而硬體架構的主要目的是透過增加平行度來提升 SSD 效能，本篇發現不管在 Syn-Channel、Ind-Channel 還是 Interleave 架構下各種 FTL 的比較都是類似的，且不同的硬體架構對於各種 FTL 的影響是差不多的，表示 FTL 和硬體架構的選擇是可以分開來考量的，但某些組合下會有如同前述 FAST 等的特例

產生，因此在這種情況下就必須額外搭配適當的韌體機制。

綜觀韌體演算法和硬體架構，由 5.3 到 5.6 的討論可以得到幾點結論，第一點，韌體演算法的設計取決於 SSD 的用途，不同的用途會有不同的資料存取行為，其中 Page level 是表現最好的 FTL，但其所消耗的 RAM 較大，而 Log-based FTL 可以經由 data block 的 N 值和 log block 的 K 值分別處理不同空間區域性和時間區域性的 workload。第二點，硬體架構的設計決定 SSD 的效能，其中可獨立存取的架構表現較優異，其中 Ind-Channel 具有高彈性和高平行度，是最佳的架構設計。第三點，SSD 的硬體架構設計需要考量到的韌體機制包含 Buffer 和 Data placement，而越高平行度的架構在越好的韌體環境例如 Spare 和 FTL 下可以發揮出越好的效能。最後，FTL 與各種硬體架構的關係是互相獨立的，舉例而言，若 SSD 的用途是要處理隨機資料則採用 PL 或 FAST 會比較好，相反的處理循序資料則應該採用 BAST 或 BC，硬體架構只是用來優化各種 FTL 的效能，並不容易影響 FTL 的表現，但也需要考慮到少數的例外，所以簡而言之，有最好的硬體架構，需要相對應的機制搭配，但沒有最好的韌體演算法，只有最適合某種用途或某個硬體架構的演算法。

6. CONCLUSION AND FUTURE WORK

SSD 不論在設計或測試上都包含很多困難，過長的產品開發週期也使得這些困難更加巨大化，因此軟體模擬的輔助是必要的。現今的 SSD 模擬軟體都注重硬體元件的分類與參數的制定，但沒有搭配足夠的韌體支援，往往使得本來應該增加便利度的工具難以上手。

為了改善這個問題，本篇建立一個韌體演算法和硬體架構的標準制定工具（RAPT）。由韌體支援著手，提出 Index、Association 和 Prioritization 三個韌體設計模型，使用者可以透過三者快速抽象出 FTL 的架構並制定 Address Mapping 和 Garbage Collection 等機制，建立好的 FTL 架構同樣可以透過三個模型輕易的修改。在硬體支援方面，RAPT 將 SSD 硬體元件簡分為 Controller、Bus、Chip enable line 和 Flash chip，透過彼此的溝通可以模擬 Inter-chip 如 Synchronized 和 Independent-Channel、Interleave 架構以及 Intra-chip 如 Multi-Plane 和 Copy-Back 指令。RAPT 也規劃一套富彈性的軟體介面允許同步的 Buffer 機制與自行設計的輸入輸出檔，以物件導向的方式撰寫，然後以 API 的形式呈現，並且經過驗證這些 API 可以有效的模擬真實 SSD，讓 RAPT 成為一個真正便捷的 SSD 開發測試平台。

本篇以 RAPT 為基礎對各種 SSD 韌硬體組合做探討，並且呈現有力的實驗數據作為參考，在 SSD 的設計上得出四點結論，首先，FTL 的設計根據不同用途的不同存取行為而定，而硬體架構的設計主要用來提升 SSD 效能，獨立位址存取的架構是最好的設計，此外硬體架構的設計也需要考量 Buffer 和 Data placement 等機制，而韌體和硬體的搭配上，兩者的選擇可以是互相獨立的，可以根據不同需求和資源限制分別選擇適合的韌體演算法和硬體架構，但也需要注意兩者之間的溝通機制，舉例而言怎樣實現平行演算法就很重要。因此若綜合考量以上數點，要在怎樣的環境開發適合什麼用途的 SSD 就不是難事。

雖然 RAPT 提供的支援可以模擬大部分已知的 SSD 演算法或架構，但許多 API 的使用上像是 Association 和 Multi-Plane 等仍然還有改善的空間，同時因為模擬效率的問題，RAPT 省略許多硬體溝通的細節考量諸如位元傳輸（Byte Transfer）以及指令傳輸時間（Command Time）等，這些都還有再做修正的空間。RAPT 也預期將目前採用的 C++語言以更容易代表 SSD 設計的少量描述性語言呈現，希望讓 SSD 的設計之路更加順暢。

REFERENCE

[1] SystemC library, http://www.systemc.org/home/.

[2] Samsung, K9XXG08UXA and K9XXG08UXM NAND Flash Specification.

[3] Fast Just Got Faster: SATA 6Gb/s, SATA-IO, 2009.

[4] PCI Express 3.0, PCI-SIG, http://www.pcisig.com/.

[5] Kang, J., Jo, H., Kim, J., and Lee, J. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM &Amp; IEEE international Conference on Embedded Software* (Seoul, Korea, October 22 - 25, 2006). EMSOFT '06. ACM, New York, NY, 161-170. DOI= http://doi.acm.org/10.1145/1176887.1176911.

[6] Kim, J., Kim, J, M., Noh, S, H., Min, S, L., and Cho, Y. 2002. A Space-Efficient Flash Translation Layer for Compact Flash Systems, *IEEE Transactions on Consumer Electronics*. (May. 2002), DATE '02, IEEE. 366-375.

[7] Lee, S., Park, D., Chung, T., Lee, D., Park, S., and Song, H. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* 6, 3 (Jul. 2007), 18. DOI= http://doi.acm.org/10.1145/1275986.1275990.

[8] Park, C., Cheon, W., Kang, J., Roh, K., Cho, W., and Kim, J. 2008. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Trans. Embed. Comput. Syst.* 7, 4 (Jul. 2008), 1-23. DOI= http://doi.acm.org/10.1145/1376804.1376806.

[9] DiskSim, http://www.pdl.cmu.edu/DiskSim/.

[10] Agrawal, N., Prabhakaran, V., Wobber, T., Davis, J. D., Manasse, M., and Panigrahy, R. 2008. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Boston, Massachusetts, June 22 - 27, 2008). USENIX Association, Berkeley, CA, 57-70.

[11] Shin, J., Xia, Z., Xu, N., Gao, R., Cai, X., Maeng, S., and Hsu, F. 2009. FTL design exploration in reconfigurable high-performance SSD for server applications. In *Proceedings of the 23rd international Conference on Supercomputing* (Yorktown Heights, NY, USA, June 08 - 12, 2009). ICS '09. ACM, New York, NY, 338-349. DOI= http://doi.acm.org/10.1145/1542275.1542324.

[12] Lee, J., Byun, E., Park, H., Choi, J., Lee, D., and Noh, S. H. 2009. CPS-SIM: configurable and accurate clock precision solid state drive simulator. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (Honolulu, Hawaii). SAC '09. ACM, New York, NY, 318-325. DOI= http://doi.acm.org/10.1145/1529282.1529351.

[13] Kim, Y., Tauras, B., Gupta, A., and Urgaonkar, B. 2009. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. In *Proceedings of the 2009 First international Conference on Advances in System Simulation* (September 20 - 25, 2009). SIMUL. IEEE Computer Society, Washington, DC, 125-131. DOI= http://dx.doi.org/10.1109/SIMUL.2009.17.

[14] Cho, H., Shin, D., Eom, Y. I. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition*. (April 20 - 24, 2009), DATE '09, IEEE. 507-512.

[15] Chang, L. and Kuo, T. 2005. Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage* 1, 4 (Nov. 2005), 381-418. DOI= http://doi.acm.org/10.1145/1111609.1111610.

[16] Kim, Y., Kim J. 2008. DAC: A Device-Aware Cache Management Algorithm for Heterogeneous Mobile Storage Systems, *IEICE Transactions on Information and Systems* (December. 2008), DATE'08, IEICE 2818-2833.

[17] Chang, L. 2008. Hybrid solid-state disks: combining heterogeneous NAND flash in large SSDs. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference* (Seoul, Korea, January 21 - 24, 2008). Asia and South Pacific Design Automation Conference. IEEE Computer Society Press, Los Alamitos, CA, 428-433.

[18] Park, J., Bahn, H., and Koh, K. 2009. Buffer Cache Management for Combined MLC and SLC Flash Memories Using both Volatile and Nonvolatile RAMs. In *Proceedings of the 2009 15th IEEE international Conference on Embedded and Real-Time Computing Systems and Applications* (August 24 - 26, 2009). RTCSA. IEEE Computer Society, Washington, DC, 228-235. DOI= http://dx.doi.org/10.1109/RTCSA.2009.32.

[19] DiskMon, http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

[20] Blktrace, http://manpages.ubuntu.com/manpages/intrepid/en/man8/blktrace.8.html.

[21] IOmeter Project, http://www.iometer.org/.

[22] Gupta, A., Kim, Y., and Urgaonkar, B. 2009. 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* (Washington, DC, USA, March 07 - 11, 2009). ASPLOS '09. ACM, New York, NY, 229-240. DOI= http://doi.acm.org/10.1145/1508244.1508271.

[23] Lee, S., Shin, D., Kim, Y., and Kim, J. 2008. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *SIGOPS Oper. Syst.*

Rev. 42, 6 (Oct. 2008), 36-42. DOI= http://doi.acm.org/10.1145/1453775.1453783.

[24] Jiang, S. and Zhang, X. 2002. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *SIGMETRICS Perform. Eval. Rev.* 30, 1 (Jun. 2002), 31-42. DOI= http://doi.acm.org/10.1145/511399.511340.

[25] Megiddo, N. and Modha, D. S. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (San Francisco, CA, March 31 - 31, 2003). Conference On File And Storage Technologies. USENIX Association, Berkeley, CA, 115-130.

[26] HD tune, http://www.hdtune.com/