國立交通大學

資訊科學與工程研究所

碩 士 論 文

高階固態硬碟架構下的通道管理方法

On Channel Management

in Advanced Solid-State Disk Architecture

研 究 生：黃義勛

指導教授：張立平　教授

中 華 民 國　九 十 九 年 七 月

高 階 固 態 硬 碟 架 構 下 的 通 道 管 理 方 法

On Channel Management in Advanced Solid-State Disk Architecture

研 究 生：黃義勛　　Student：Yi-Hsun Huang

指導教授：張立平　　Advisor：Li-Ping Chang

國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年七月

高 階 固 態 硬 碟 架 構 下 的 通 道 管 理 方 法

學生：黃義勛　　指導教授：張立平

國立交通大學資訊科學與工程研究所碩士班

摘要

隨著快閃記憶體的快速發展，類似固態硬碟(SSD)這樣子輕巧、耐震且儲存密度高的裝置越來越普遍。而這類的裝置內部會使用像 RAID 一樣的平行架構，其目的是藉由並排存取快閃記憶體的通道來提升單位時間的存取速度。傳統上這些通道的管理方式是以同步運作的方式來處理，然而因為快閃記憶體具有與一般儲存裝置截然不同的特性，在面對小資料隨機寫入的情況下無法發揮多通道應該有的效能，因此這些通道需要一套有效的管理方法來提升存取的平行度，使小資料隨機寫入的效能可以提高。本篇論文提出三項多通道的管理機制，解決多通道在過去所面臨的多項管理瓶頸。實驗結果發現，與傳統的管理方法比較，我們的方法在 8 個通道及 2.5% overprovision 的情況下，小資料隨機寫入的效能為同步運作通道的 3.72 倍。

關鍵字: 固態硬碟, 多通道, 快閃記憶體

On Channel Management in Advanced Solid-State Disk Architecture

Student：Yi-Hsun Huang　　Advisor：Li-Ping Chang

Department of Computer and Information Science
National Chiao Tung University

Abstract

With the rapid development of flash memory, the devices, similar to solid-state disks, are compliant to small form-factor but offer very high storage density. Advanced SSDs employ parallel architectures, performing like small RAIDs. The purpose is to enhance the throughput of access by side by side these channels. Traditionally, the management of these channels is the way to deal with synchronization. However, because the flash memory storage devices in general have different characteristics, synchronized channels cannot exploit the behavior of multi-channels against to the small and random write requests. Smart management of channels helps to improve parallelism of read and write. This paper proposed 3 mechanisms of channel management to improve the performance of multi-channel against to the small random write. Experimental results show that our channel management scheme, in small random write, can be 3.72 times of IOPs improvement over synchronized channels under 8 channels and 2.5% overprovision.

Keyword: SSD, Multi-Channel, Flash Memory

致謝

需要感謝的人事物非常多，這短短的篇幅是寫不完的。記得小時候的國文課有讀到：要感謝的太多了，那就謝天吧。

剛進實驗室的時候，還不太能夠適應實驗室這個新環境，而且沒有勇氣面對錯誤與挫折，曾經一度想要休學。對於兩年前的我來說，現在我長大了，我有了更多的責任感與信心去完成手上該做的工作。這一切都要感謝一路上指導我、帶領我的張立平教授。

記得在跟老師 meeting 的時候，常常會說「不知道要怎麼講耶」，真的會覺得自己的表達能力很差，但是老師還是很有耐心的慢慢和我討論出我想表達的東西。也許這一頁簡單的幾句話無法清楚的表達我對張立平老師的感謝與感觸，而且在私底下偶爾也會有不滿與抱怨，但是兩年下來，真的很感謝張立平老師讓我受益良多。

除了老師，也要感謝實驗室同學與學長姐、學弟妹的幫忙與照顧，像是電鍋(郭晉廷)都要去花園街夜市幫忙買牛排，Master(黃偉杰)分一半的床借我睡了一個月，BouBou(黃莉君)要幫忙上台報告和教外國人功課以及學妹 Uma(王薇涵)辛苦的交接我的研究…等等。

另外，也要感謝小噴火恐龍梁文珊的體諒與包容。感謝大學同學張琮偉跟我一起住了一年，劉彥聖寢室借我住了一個暑假。感謝朋友阿潘(潘志翔)、小博士(張維廷)和偉哥(陳世偉)在我回高雄的時候常常陪我去網咖。感謝很厲害的爸爸、拼命賺錢的媽媽、面惡心善的爺爺，以及對我很好的叔叔嬸嬸姑姑姑丈們。最後要感謝高雄縣家扶中心與施振榮志工獎學金的幫忙。

黃義勛(yishion)
ESSLAB
2010.07.26

Contents

圖目錄

表目錄

1 INTRODUCTION

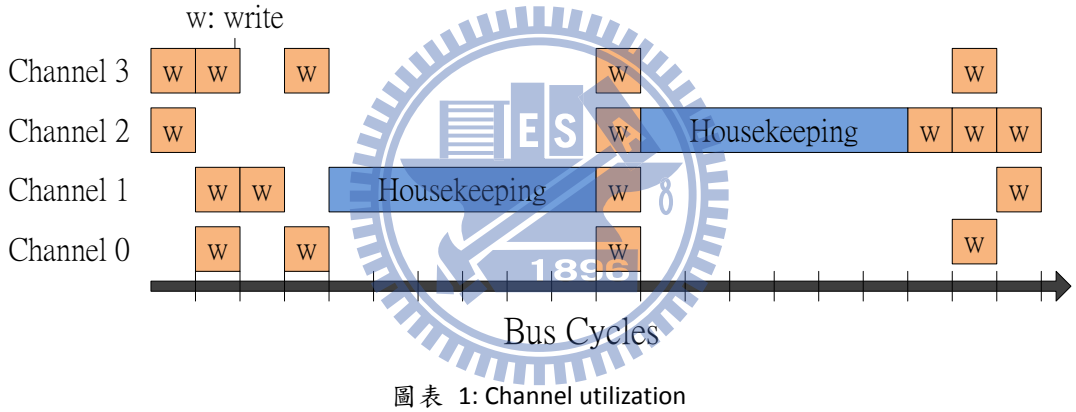
固態硬碟(Solid State Disk, SSD)是新一代的大量資料儲存裝置，其內部使用像是 NAND 快閃記憶體(flash memory)的固態記憶體來儲存資料。由於固態記憶體具有體積小、低耗電以及耐震等多項優點，在加上隨著容量越來越大、價格越來越便宜，以 NAND 快閃記憶體為基礎的 SSD 正快速發展，逐漸取代傳統的硬碟，成為新一代大量儲存裝置的新選擇。

SSD 內部連接快閃記憶體並能夠獨立的操作快閃記憶體的連接線路稱之為獨立通道(channel)，這些線路包含匯流排及控制線。資料由匯流排進出快閃記憶體模組的速度太小，遠不及 SSD 對 Host 端的介面傳輸速度，因此 SSD 傳輸資料的瓶頸在於內部資料進出快閃記憶體模組的匯流排頻寬(Bandwidth)。由於 NAND 快閃記憶體的體積小，相對於硬碟來說並不會因為使用多通道並聯的平行架構而增加體積和散熱成本，因此 SSD 內部會有許多通道平行連接到控制晶片藉以提升效能。然而增加平行的通道雖然可以增加存取的頻寬，但是如果這些通道的常常閒置而使得利用率不高的話就無法表現出應有的效能。SSD 內部的平行架構就像是微型的 RAID-0[8]，資料擺放的方式會做 Striping，平均分攤在每一個傳輸通道的快閃記憶體模組上，如此可以使每一個通道被使用的機率都一樣，增加存取的平行度而使傳輸通道的利用率上升。

而這種平行概念最簡單的實作方式就是讓這些通道同步運作，每一個通道都在相同的位址同步執行相同的操作，在同一時間對全部的通道做存取，除了可以提升讀取與寫入資料的效率，連快閃記憶體內部資料重整的效率也提高了。這種處理方式對於較大的 request 來說確實可以增加存取資料的頻寬，有 N 個平行通道，效能就會提升將近 N 倍；但是對於小的 request 來說，原本若只需要一個通道便可以滿足存取的需求，但是卻要使所有的通道都啟動並做存取，那些原本不需要使用的通道所做的存取動作是沒有義意的，不但沒有提升存取的速度，反而會導致快閃記憶體模組內部空間因為額外無意義的寫入造成空間利用率不佳，使得內部資料重整的頻率增加，拖累整體的效能。事實上在多工(multitasking)系統環境下(windows、linux)，對於儲存裝置的存取會有很多小寫入動作，即使這些小寫入動作是一個接一個連續，但是由於多工的關係，許多程序的小寫入動作會混雜交錯在一起，儲存裝置收到的存取模式就變成小資料的隨機寫入動作。因此對於複雜多工系統所需要的 SSD 來說不只是大範圍的連續資料存取效能要好，小資料的隨機寫入也要有一定水準的需求。因此強迫每次存取都同步運作的方式並不是一個很好的方法。

如果這些通道沒有強迫同步存取，不同通道可以同時對不同的位置做不同的

操作。在處理大的 request 時，所有的通道也是會同時啟動運作；而處理小的 request 也不會造成無意義的額外存取。但是在處理小的 request 時，只有啟動部分的通道來完成寫入，使得其他未啟動的通道閒置而沒有運作，造成通道利用率不佳。不僅如此，對於 SSD 來說不只是 request 資料寫入或讀取的頻寬會影響效能，因為不同步運作的關係，當其中一個通道的快閃記憶體模組因為空間不足而執行內部資料重整的工作時，要寫入這個通道的 request 要等內部資料重整完成才能寫入，這個時候其他沒有做內部資料重整的通道就會閒置，也使得通道利用率不佳。相較於同步運作通道的方法，獨立的通道反而會因為內部資料重整的時候不同步而使得 SSD 無法表現出多通道應有的效能。實際上，SSD 有一大部分的時間在運作快閃記憶體模組的內部資料重整，而小資料的隨機寫入也可能會使內部資料重整的時間增加，在比較糟糕的情況一次內部資料重整所佔用的時間可能與執行數百甚至數千次寫入的時間相同。如圖表 1 為每個通道在不同時間所執行的操作，通道在很多時候都是閒置的。使用不同步運作的通道就面臨空有硬體資源而無法有效利用的這項棘手的問題。



圖表 1: Channel utilization

通道同步運作會使得通道做出無意義的利用，通道不強迫同步運作會使得通道閒置沒有利用。要提高效能就必須要提高通道的有效利用率。然而當其他通道在忙碌的時候，閒置的通道應該要怎麼利用才能提升有效的利用？第一個困難在寫入資料時的平行度，當寫入 request 所需要使用的通道正在忙碌於寫入上一個 request 時，因為受限於 striping 資料擺放的規則，不能將 request 寫入其他閒置的通道。另一個困難是內部資料重整的平行度，當一個通道正在執行內部資料重整的時候，其他的通道也會處於閒置的狀態。

加入緩衝區暫存記憶體可以提升平行度。若是要先跳過目前這個因為通道忙碌而等待的 request 而先執行下一個 request，就需要一個緩衝區來暫時存放等待的 request。當通道忙碌的時間很長，像是執行內部資料重整，那就需要較大的緩衝區來存放一連串等待的資料，等到緩衝區滿了還是會造成通道閒置。因此加入緩衝區雖然可以輔助增加 request 寫入平行度但並不能決執行內部資料重整時

的平行度，尤其是在處理小資料的隨機寫入動作，會使得內部資料重整的時間比較長。而且加入大量的緩衝區會大量增加硬體成本與電能消耗。

使用動態的方式排列資料可以提升平行度。若是打破 striping 的資料排列方式，將資料以動態的方式排列，把寫入的資料存放到其他空閒的通道，這樣子可以提高寫入時的平行度，提高通道的利用率。但是相對的會破壞讀取的平行度，我們相當重視連續資料讀取的效能，因此本研究不考慮破壞 striping 資料的擺放方式。

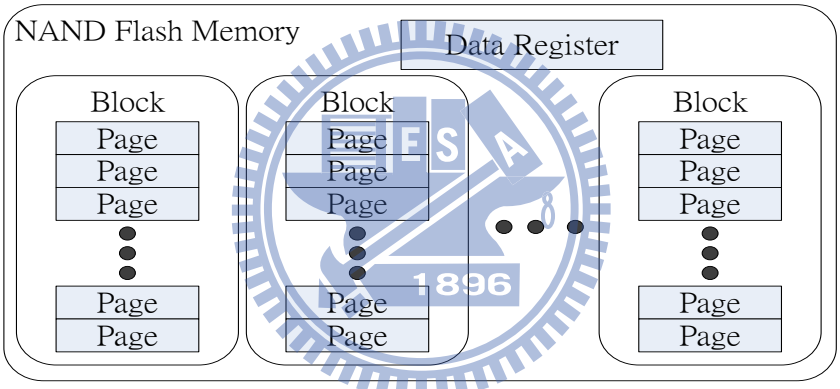
本篇論文提出一套管理獨立通道的方法，整合了通道同步運作與獨立運作互補的優點，除了解決同步運作所產生的空間利用率不佳的問題，也改善不同步運作下通道利用率不佳的問題。有幾項特點如下：第一，**高效能的連續資料讀取**：不破壞資料 striping 擺放，所以不犧牲大範圍連續資料讀取的效能(如電影、音樂及圖片等多媒體的檔案操作)。第二，**高平行度的寫入**：這個管理演算法建立在獨立的通道上，利用小小的緩衝暫存區來增加寫入的平行度，使不同通道可以盡可能的同時寫入有意義的資料而不造成空間浪費。第三，**高平行度的內部資料重整**：類似同步運作的通道，使所有通道同時進行內部資料重整，避免通道閒置進而提升通道的利用率也提升 SSD 整體效能。最後，**穩定的回應時間**：使每一次內部資料重整所花費的時間維持穩定在可容忍的時間內，讓內部資料重整所花的時間受管制。

本論文其他部分的組成如下：下一章為 Background，將會說明 SSD 的軟硬體架構、特性以及其他相關的研究。第三章將會介紹本論文所提出的通道管理方法。第四章為模擬的實驗結果。最後一個章節為結論。

2 BACKGROUND

2.1 NAND Flash Memory

NAND flash memory 是一種非揮發性記憶體，因為其體積小、便宜、低耗電、耐震等多種特性，已被廣泛的作為大量儲存資料使用，例如各種記憶卡、隨身碟以及固態硬碟等。如圖表 2，快閃記憶體模組內基本上分成多個 Block，Block 為快閃記憶體操作 Erase 指令的最小單位，而每一個 Block 又含有多個 Page，Page 為 flash memory 操作 Program(write)或是 Read 指令的最小單位。這些 Page 並不支援 in-place update，寫過的 Page 在還沒被 Erase 之前都不能再寫入資料。另外，快閃記憶體模組內部會有一個暫存記憶體，寫入資料時必須先把資料傳進這個暫存記憶體，然後才執行 program 的操作，將資料寫入快閃記憶體；而讀取資料時，必須先執行 read 的操作讀取到暫存記憶體中，然後才由暫存記憶體往外傳輸。



圖表 2: NAND Flash Memory

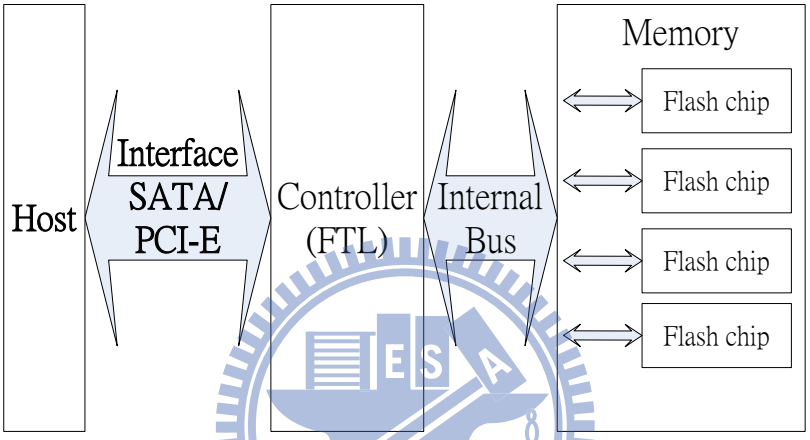
以 SAMSUNG 系列模組[1][2]為例，如表格 1 為兩種不同架構的快閃記憶體模組，其中暫存記憶體大小與 Page 相同，進出快閃記憶體模組的匯流排速度為每一個位元組 25ns，也就是約為 40MB/S。

SAMSUNG	K9XXG08UXA	K9XXG08UXM
Page Read to Register	25µs	60µs
Page Program from Register	200µs	800µs
Block Erase	1500µs	1500µs
Serial Access to Register (per byte)	25 ns	25 ns
Block Size	128 KB	512 KB
Page Size	2 KB	4 KB
Register size	2 KB	4 KB

表格 1: SAMSUNG NAND flash chip

2.2 SSD Architecture

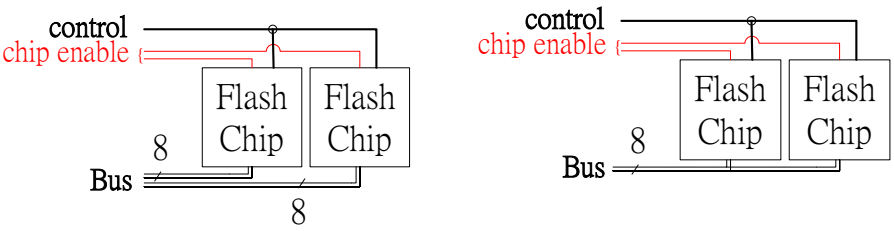
SSD 內部包含有快閃記憶體模組、管控整個 SSD 的控制晶片以及與 Host 連接的介面。Flash memory 模組會以陣列的方式擺放，多個模組會串連在一起共用一個傳輸通道，而會有多個傳輸通道並連在一起連接至 SSD 的控制晶片。控制晶片內部運行著 Flash Translation Layer(FTL)的演算法，讓 SSD 隱藏 Erase 的特性而模仿硬碟，其中會有一些 RAM 做為 buffer 或是存放演算法所需要的資料結構，例如 mapping table。對 Host 連接的介面常見的有 USB、Fiber Channel、PATA/SATA、PCI-E 等等。



圖表 3: SSD Architecture

傳輸介面速度約有每秒數百甚至數千 MB(例如 SATA3 約 700MB/S[3], PCI-E 3.0 16X 約 16GB/S[4])。而快閃記憶體模組的速度大約為每秒數十 MB，因此並聯的多通道(Multi-channel)可以增加控制晶片對快閃記憶體模組單位時間的存取資料量，也就是增加 SSD 內部存取快閃記憶體的最高極速。

在快閃記憶體模組執行內部操作(Read/Write/Erase)時，連接快閃記憶體模組的通道是閒置的，因此一個通道可以連接許多快閃記憶體模組，而這一群快閃記憶體模組會共用相同的控制線。若這些這一群快閃記憶體模組的資料匯流排是平行獨立的可以同時傳輸，稱為 Ganging(如圖表 4(a))；若是這一群快閃記憶體模組共用同一個的資料匯流排，輪流以 pipeline 的方式運作，這種技術稱為 Interleaving(如圖表 4(b))。但是這兩種硬體的平型架構的操作方式基本上是固定的， 因此本篇論文所討論的通道管理方法是針對控制線及匯流排完全獨立的不同通道之間的管理。

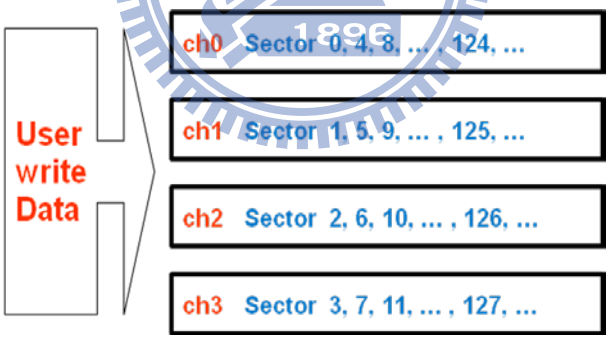


(a) Ganging (b) Interleaving

圖表 4: Hardware parallelism

2.3 Data Placement

資料的擺放方式也必須配合 SSD 多通道的硬體架構，以增加傳輸通道的使用率。存放的資料在多通道的架構下，會以類似 RAID-0[8]做 striping 的方式排列，將資料平均的分散到 SSD 中每一個通道，使得這些傳輸通道的存取機率相等，以便盡量讓每一個通道都能保持平行的運作，進而達到增加效能的目的。如圖表 1，資料依照 sector 的編號做 striping，sector 號碼除以 4 於數為 0,1,2,3 的資料分別放在不同的通道內。這種資料擺放的方式可以讓連續讀取的效能達到硬體的極限。因為破壞這種 striping 會使讀取的效能受到寫入資料的影響，為了不破壞讀取的平行度，在本篇論文中的討論將以不破壞這種 striping 的資料擺放方式為前提。



圖表 5: Data Striping

2.4 Flash Translation Layer

FTL(Flash Translation Layer)是將邏輯位址(logical address)操作命令轉換成物理位址(physical address)操作命令的轉換層。快閃記憶體具有寫入資料前要做 Erase 的特性，無法像傳統硬碟一樣可以做 in-place 更新。一般用於傳統硬碟的系統並無法直接對快閃記憶體做存取，因此在 SSD 內部會有 FTL，由控制晶片裡面的微型處理器運行 FTL 的演算法，這些對邏輯位址 in-place 更新的指令轉換成對物理位址 out-of-place 更新的指令。除此之外，FTL 演算法也會有邏輯位址對應

到物理位址的對照表(mapping table)，對應的方式會影響到這個表的大小。也因為需要 out-of-place 更新的原因，實際上物理位址空間會比邏輯位址可用空間來的大，多出來的部分稱為 Overprovisioning。

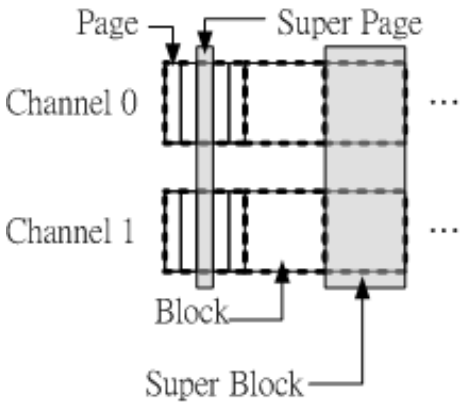
在經過一連串的 out-of-place 更新指令之後，舊的資料會被當作無效的資料，而這些無效的資料所佔據的空間會分散在不同的 block 內。因此 FTL 的演算法也需要處理這些無效的資料，稱為 Garbage Collection(GC)。GC 是適當的將一些 block 內部有用的資料複製搬移到別的地方，使得 block 內部都剩下無效的資料，才可以執行 Erase 指令才回收整個 Block 的空間。

另外，因為快閃記憶體的最小寫入單位為一個 page，在寫入資料小於一個 page 的時候，就必須做 read-modify-write 的動作。read-modify-write 就是要先將舊的資料從快閃記憶體中讀取出來到控制晶片中，然後再將要修改的地方做修改，最後再把這個 page 以 out-of-place 更新的方式寫回快閃記憶體中。Page 越大 read-modify-write 的成本就越高。

2.5 Problem Formulation

傳統的 SSD 多通道設計大部分都是針對多媒體應用或是備份儲存裝置等大量連續資料傳輸來設計，並沒有針對小資料的隨機寫入來提升效能，但是隨著 SSD 的應用越來越廣泛，小資料的隨機寫入就越來越重要了。就一般的 PC 來說，雖然作業系統內部會有設計緩衝區來存放對硬碟讀寫的資料，但是系統上會有許多重要的資料會跟緊跟著一個 flush 的指令來確保資料可以立即平安的寫入硬碟中，像是重要的系統檔或是檔案系統的 meta data 及 table 之類的；而在使用者的應用程式操作下，會有像 web 瀏覽器之類的程式會對大量資料有 index 類的資料，這些 index 資料也會緊跟著 flush 的指令。因此小資料的隨機寫入是不容忽視的。

在同步運作通道的方式下，如圖表 6，不同通道中相對應位置的 block 會組成 super block、相對應位置的 page 會組成 super page，操作快閃記憶體的單位由 block、page 轉變為 super block、super page[20]。小資料的隨機寫入動作會讓 super page 產生 read-modify-write 的負擔，通道越多 super page 越大，read-modify-write 的負擔就會越大。不僅如此，因為很多空間用來重寫那些不需要更新的資料，使得空間利用率不佳，造成 GC 觸發的頻率增加，也讓系統增加了許多的負擔。因此為了提升小資料隨機寫入的效能就不能強迫所有通道同步運作。



圖表 6: Super page/block

在多通道多晶片的架構下，將資料以 striping 的方式擺放，連續的資料分散在不同快閃記憶體模組和不同通道中可以讓連續讀取速度達到硬體的極速，在這篇論文中我們相當重視連續讀取的效能，因而不考慮破壞 striping 的資料擺放方式。

然而在通道獨立運作下會使得部分通道出現閒置的現象。在寫入資料的時候，request 要寫入的通道還在執行上一個 request 的寫入動作，因為 striping 資料擺放的關係，request 就會卡住並等待，此時其他通道就會閒置而沒有利用。另外在其中一個通道正在處理 GC 所需要資料搬移時，其他通道也會閒置，相較於同步運作通道的方式，單位時間 GC 所回收的空間比較少。因此使用獨立通道必須要解決通道閒置的問題，提升通道在寫入 request 及 GC 時的利用率。

另外，在執行 GC 回收空間的時候，需要暫用數百個甚至數千個快閃記憶體寫入操作的時間，有可能會因為時間太久而使儲存裝置不能及時的完成 request 並回應給 host 端，造成 host 工作延遲或是誤判裝置損毀等問題。因此必須設法限制 GC 的時間來穩定回應時間。

綜合以上幾點，這篇論文所要達到的目標如下：首先是要減少小資料隨機寫入所造成的負擔，降低 read-modify-write 負擔及 GC 頻率。第 2 個目標是維持多通道高速的讀取效能。第 3 個目標是盡量讓 GC 活動平行運作。第 4 個目標是盡量讓寫入資料平行運作。最後是控制穩定的回應時間。

2.6 Related Work

在過去的文獻中，大都將注意力放在共用匯流排的平行度，也就是多個快閃記憶體模組共用相同匯流排，讓不同快閃記憶體模組輪流使用匯流排傳輸資料，並讓快閃記憶體模組的 Erase、Program 和 Read 的時間重疊而達到平行的效果。

L. P. Chang 等人 [13] 提出了動態 striping 的方式來提升平行度。針對共用相同匯流排的架構，讓資料寫入空閒的快閃記憶體模組，而不強迫資料須放置在固定的快閃記憶體模組內，如此可以省下資料因為要放置在固定的快閃記憶體模組內而等待的時間。將這樣的設計帶入多通道的運作也可提升寫入的平行度，但是這樣的設計會使得連續讀取無法平行運作而降低讀取效能。

Y. B. Chang 等人 [15] 提出可以用 striping 的方式存放經常讀取資料的複本來提升讀取的平行度。針對共用相同匯流排的架構，將經常需要讀取的資料編碼後放置在與原本資料放置位置不同的快閃記憶體模組內，使讀取的時候有更多選擇可以讀出原始資料，讓讀取可以減少等待時間而提升讀取效能。這篇論文主要是在提升讀取效能，並沒有提升寫入或是 GC 的效能。

J. Seol 等人 [19] 提出 MCA，使用緩衝區來提升寫入平行度。針對共用相同匯流排的架構，動態 striping 寫入資料時，read-modify-write 需要讀取的 page 存放在其他正在忙碌的快閃記憶體模組內，因此必須等待而導致閒置。加入緩衝區可以暫存一些 request，當發現會因為 read-modify-write 需要讀取其他正在忙碌的快閃記憶體模組時，就先讓其他的 request 優先寫入，避免閒置。將這樣的設計帶入多通道的運作中，首先是動態 striping 需要解決讀取無法平行的問題，其次是雖然可以提升寫入平行度，但是卻無法提升 GC 的平行度，而且在他的實驗中需要使用數 MB 的 RAM。

C. Park 等人 [14] 針對大量連續資料，提出多通道且自動化 interleaving 的硬體架構，可以大量提升存取的效能。但是這篇論文並沒有解決小資料隨機寫入的問題。

J. U. Kang 等人 [16] 提出 DUMBO，其中有四個獨立通道，包含三種平行機制，分別是 striping、interleaving 和 pipelining，striping 及 interleaving 是將資料分散在不同快閃記憶體模組藉以提升平行度，而 pipelining 是提升同一個通道內的快閃記憶體模組的平行度。然而 DUMBO 並沒有去解決部分通道會閒置的問題。

Y. J. Seong 等人 [20] 提出 Hydra，將多通道並聯並同步運作以加大存取的頻寬，並讓多組獨立的快閃記憶體模組串連並共用匯流排來增加匯流排的使用率而提高效能。與一般共用匯流排的 interleaving 技術不同的地方在於他是獨立的，不同群組的快閃記憶體模組可以執行不同的操作。然而 Hydra 並沒有解決小資料隨機寫入效能不佳的問題。

N. Agrawal 等人 [17] 與 J. Shin 等人 [18] 討論了許多 SSD 設計上的議題。

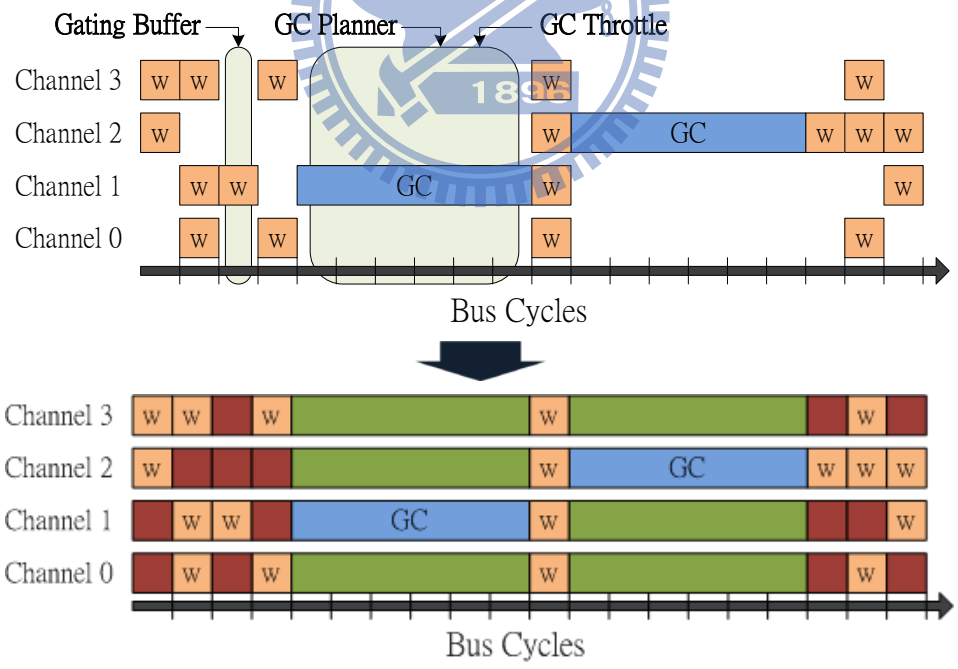
他針對 page-mapped FTL 模擬在各種設定下的表現，包括 mapping 的單位、overprovisioning、interleaving、ganging 以及一些門檻(threshold)的設定，最後歸納出 FTL 設計上的各種考量因素的優缺點。這兩篇研究並沒有提出新的方法，只是針對現有的議題進行比較測試。

本篇論文提出多通道的管理方法，除了提升小資料隨機寫入的效能，也要兼顧讀取、寫入以及 GC 的平行度。

3 CHANNEL MANAGEMENT

3.1 Overview

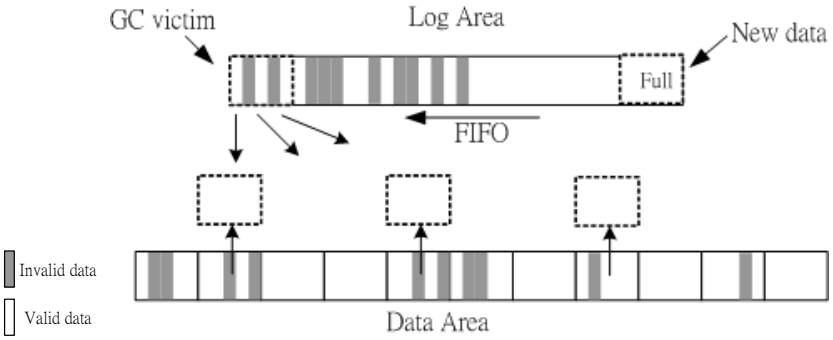
這篇論文為了要處理同步運作通道中，小資料隨機寫入所造成的負擔及衍生出來的相關效能不佳的問題，在第 2.5 小節中有歸納出五個目標。首先是要減少小資料隨機寫入的負擔，在此我們使用獨立的通道管理，降低 read-modify-write 負擔及 GC 頻率。第 2 個目標是保持高速的讀取效能，在此我們使用靜態的 striping 資料擺放方式，讓資料平均得分散在不同通道上，以達到最高速的讀取速度。第 3 個目標是讓 GC 活動平行運作，在此我們使用 Garbage Collection Planner 的機制來提升 GC 活動的平行度，在 3.2 節中會有詳細的說明。第 4 個目標是讓寫入資料平行運作，在此我們使用 Gating Buffer 的機制來提升寫入的平行度，在 3.3 節中會有詳細的說明。最後是穩定回應時間，在此我們使用 Garbage Collection Throttle 的機制來限制 GC 的處理時間，防止回應時間超過限制的問題發生，在 3.4.節中會有詳細的說明。如圖表 7，我們想要在獨立運作的通道下加入一些機制提升通道的有效利用率，讓每一個通道的每一個 cycle 都可以充分利用，進而提升整體的效能。



圖表 7: Mechanism Overview

因為 FTL mapping 方式並不是本篇論文的重點，我們參考 log-based block-mapped FTL[9][10][11][12]，本論文的討論將採用簡單的 log buffer[10]方式管理快閃記憶體的 mapping。如圖表 8 所示，快閃記憶體內部分成 log 區域及 data

區域，data 區域為邏輯位址的大小，由 data block 所組成，log 區域為 overprovision 的大小，由 log block 所組成，資料 out-of-place 更新到 log 區域，空間用完時以 FIFO 的方式挑選 log block 來做 GC。GC 需要對 data block 執行合併動作才能將資料合併回 data 區域。合併的過程是將屬於該 data block 的有效資料全部複製到新的 block 中，並取代舊的 data block。反覆執行這樣的合併動作，直到被 GC 的 log block 中已經沒有有效資料就可將此 log block 回收。

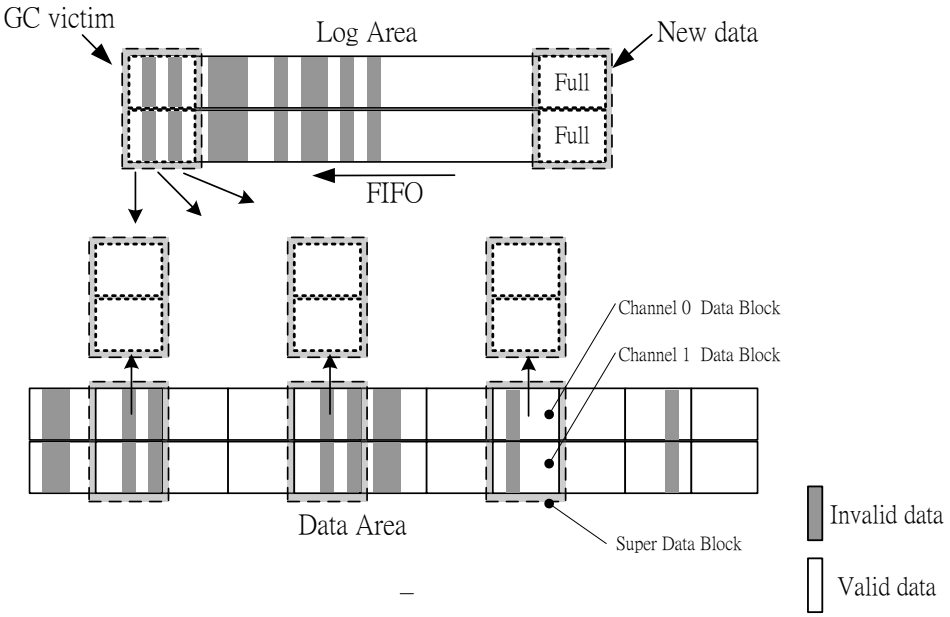


圖表 8: GC of log-based block-mapped FTL

3.2 Garbage Collection Planner

3.2.1 A Basic Approach: Synchronized Channels

在說明 GC planner 之前，先來瀏覽一下 FTL 如何在同步運作的通道中運行。不同通道中相同物理位址的 page/block 會合併成為 super page/block，可以簡單的將單通道的 FTL 演算法套用到多通道上面，如圖表 9 為兩個通道同步運作的 FTL 示意圖，大的虛線矩形為 super block，一個 super data block 由兩個通道各一個 data block 組成，super log block 同理。寫入資料以 out-of-place 更新寫到 super log page 中，GC 時會將有效的 super log page 與 super data page 複製合併到新的 super data block 中。



圖表 9: FTL under synchronized channels

同步運作的通道將不同通道中的 page/block 綁死視為同一個 super page/super block。優點是平行度非常的高，所有的通道都同時執行動做。缺點是寫入的最小單位隨著通道數量提升，使小的 request 無法隨著通道數量的提升而提升寫入的效能。

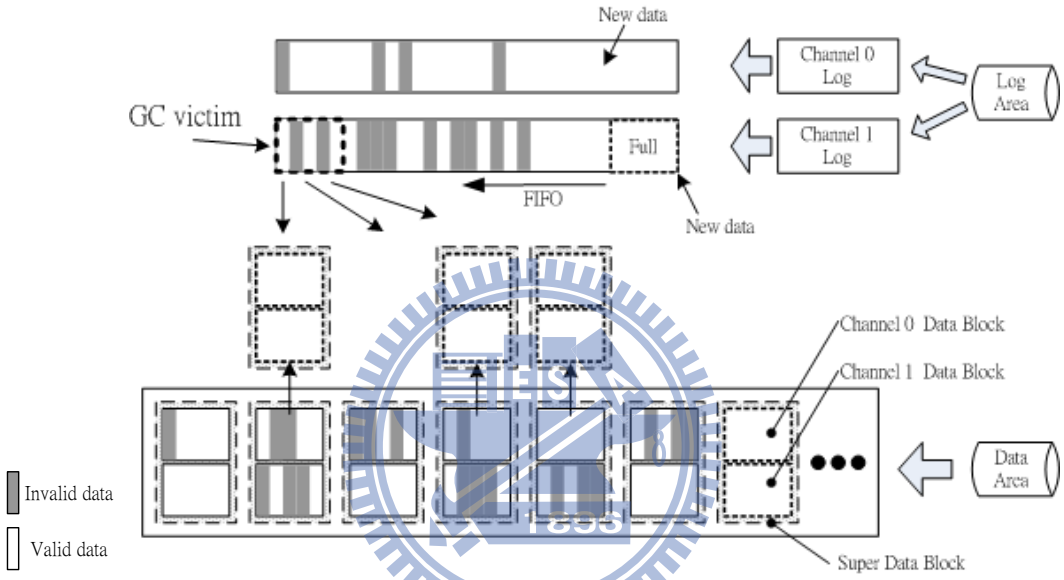
3.2.2 An Extension: Super Data Block over Independent Channels

觀察同步通道的運作方式，在 GC 時平行度相當的高，是因為每一個通道都同時對同一個 super data block 中屬於不同通道的 data block 執行合併動作，單位時間內合併的數量多，回收的空間也多。Super data block 其實是由不同通道間邏輯位址相鄰的 data block 所組成的，其內部的 super data page 也是由不同通道間邏輯位址相鄰的 data page 所組成的。因此合併 super data block 並不是只有同步運作通道才可以執行，在獨立通道中也可以合併 super data block。

獨立通道可以解決小資料隨機寫入的問題，使得空間的有效利用率提升。但是當其中一個通道觸發 GC 時，其他的通道並不會同時觸發 GC 而會閒置。為了提升獨立通道 GC 的平行度，我們將合併 super data block 的觀念帶進獨立通道中。也就是說，將同步運作通道中的 super data block 概念留下，只是把 logging 機制換成獨立通道。而造成觸發 GC 不平行的問題就以合併 super data block 來處理。

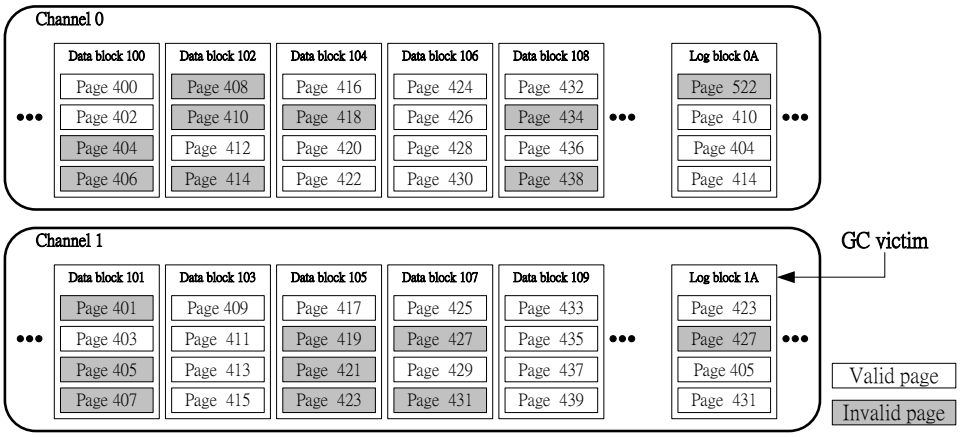
讓其他原本空閒的通道來合併與目標 data block 邏輯位址相鄰的 data block，藉以提升 GC 的平行度。

如圖表 10 所示，通道 1 的 log 區域已經寫滿因而觸發 GC，block 中有效資料分別屬於三個 data block，而這三個 data block 又分別屬於三個 super data block，這三個 super data block 都需要執行合併動作，如此通道 0 及通道 1 皆須要合併 3 個 data block，兩個通道在這段時間都會利用通道將資料從 log 區域搬移到 data 區域，因此提升了 GC 時通道的使用率，降低了通道 1 未來 GC 的負擔。唯一不平行的地方在於通道在 GC 的最後要對回收的 log block 執行 Erase，此時通道 0 並無法 Erase 而閒置，但是這個 Erase 的時間佔整個 GC 的時間比例極低，影響甚小。



圖表 10: Independent channels with super data block

如圖表 11 為兩個獨立通道在靜態 striping 的資料擺放下某一時間點的記憶體狀態示意圖，圖上的數字皆為邏輯位址，log block 0A 及 1A 分別為通道 0 及通道 1 最早被使用的 log block。通道 1 觸發 GC 的時候，必須要回收 log block 1A，因此要將 page 423、page 405 及 page 431 合併回 data 區域，也就是要對 data block 101、data block 105 及 data block 107 執行合併動作。data block 101 所屬的 super data block 還包含 data block 100，data block 105 所屬的 super data block 還包含 data block 104，data block 107 所屬的 super data block 還包含 data block 106。因此通道 0 也要執行合併 data block 100、data block 104 及 data block 6 的動作。如此便可以提升通道的利用率，增加 GC 時候單位時間的回收空間。



圖表 11: Example: Merge super data block

通道 1 觸發 GC，是由 FTL 演算法挑選出最適合回收的對象(在本論文中是以 FIFO log buffer 挑選出含有最久沒被更新 page 的 block 為對象)，因此如圖表 11 合併通道 1 的 data block 101、data block 105 及 data block 107 相當合理。但是通道 0 中所合併的對象僅僅只是因為與通道 1 所合併的 data block 相鄰而共同成為 super data block，雖然相鄰的邏輯位址具有冷熱資料的空間區域性，但是當通道數量增加的時候，super data block 也會增大，空間區域性的特性會被稀釋掉，可能使其他通道所合併的 data block 太早合併。如圖表 11，合併通道 0 中的 data block 104，其中只含有 page 418，而這個 page 418 可能是剛剛才被更新寫入 log 區域的資料，之後還有很大的可能會被更新，如此便被合併回 data 區域，若短期內再被更新，則 data block 104 在此時合併便是徒勞無功。不僅如此，合併 data block 106 根本就是沒有意義的合併，這個 data block 根本就沒有需要合併的資料，只是在浪費通道的利用。由此可以知道 GC 的時候合併 super data block 是有其缺陷的。

在獨立通道中合併 super data block 的方法可以讓全部的通道都跟著合併 data block，使 GC 的平行度提升。但是因為邏輯位址相鄰 data block 中的狀態並不一樣，可能會出現太早合併的情形，使得合併 super data block 的效益並不夠高。另外在 GC 最後要 Erase log block 的部分也因為其他通道沒有完整無效資料的 block 而無法同步 Erase。

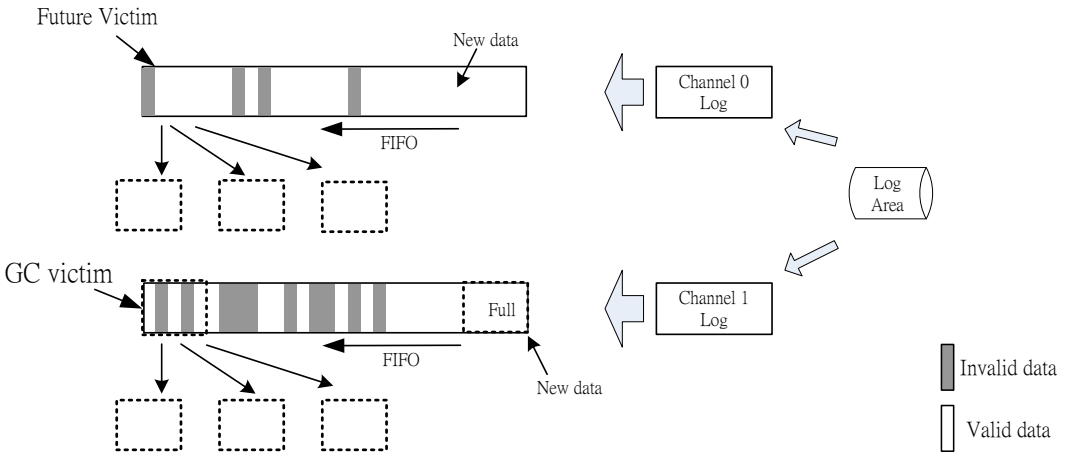
3.2.3 Cycle Stealing with Multiple Channels

在獨立通道中合併 super data block 會有太早合併的情況，要改善這個缺點就不能強迫合併 super data block，必須要打破 super data block 的限制。在這一小節會將合併 super data block 的方法改進成為 Cycle Stealing 的方法來管理多通道在 GC 時的平行運作方式。

從上述 3.2.2 合併 super data block 的方法中，我們了解到除了觸發 GC 的通道必須要完整的執行 GC 活動才能達到回收空間的目的，但是其他通道並不需要執行完整的 GC 活動，只需要利用閒置的時間跟著做合併動作就可以帶走 log 區域的資料並降低未來 GC 的負擔。利用閒置的時間跟著做合併動作其實並不需要合併邏輯位址相鄰的 data block，而是只要合併 data block 就可以有同步 GC 的效果。

既然合併邏輯位址相鄰的 data block 所組成的 super data block 會使得其他通道發生太早合併的現象，在這邊我們就不要強迫其他閒置通道合併這些 data block，只讓其他通道主動偷取這段 GC 空閒的時間跟著做合併動作，每一個通道獨立選擇 data block 來合併。如圖表 12，當通道 1 的 log 區域寫滿而觸發 GC 的時候，通道 1 以 FIFO 的方式挑選要回收 log block，其中需要合併三個 data block。此時通道 0 會有合併三個 data block 的時間是空閒的，在此我們讓通道 0 偷取這段空閒的時間，主動挑選三個 data block 來執行合併動作。

而要如何挑選要跟著合併的 data block，也是很重要的。為了避免太早的合併，必須要挑選未來最需要被合併的 data block。而這要從未來最有可能被回收的 log block 中去尋找需要被合併到 data 區域的 page，然後合併這個 page 所屬的 data block，如此便可以合併在最近的未來最有可能被合併的 data block，以避免太早的合併。在圖表 12 中，通道 0 未來需要回收的對象為 log 區域中最老的 log block，其中所包含的都是最老的 log page（最久沒被更新的 page），因此通道 0 就根據這些最老的 log page 依序挑選出三個不同的 data block 來執行合併動作，如此就不會有太早合併的問題。

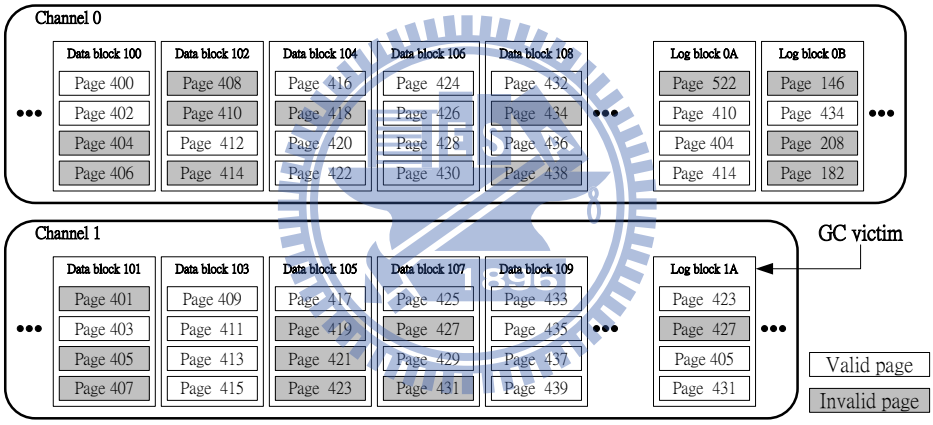


圖表 12: Independent channels with cycle stealing

GC 活動除了一連串的合併動作之外，還要外加一個 Erase log block 的操作才能達到回收空間的目的，所以對於其他通道來說，也會有一個 Erase 操作的時間

可以利用。我們也讓其他通道在一次 GC 活動中可以執行一次 Erase 無有效資料的 log block，讓 GC 活動期間可以更完整的利用通道。

如圖表 13 為兩個獨立通道在靜態 striping 的資料擺放下某一時間點的記憶體狀態示意圖，圖上的數字皆為邏輯位址，log block 0A 及 1A 分別為通道 0 及通道 1 最早被使用的 log block，log block 0B 為通道 0 第二早被使用的 log block。通道 1 觸發 GC 的時候，必須要回收 log block 1A，因此要將 page 423、page 405 及 page 431 合併回 data 區域，也就是要對 data block 101、data block 105 及 data block 107 執行合併動作。通道 1 所觸發的 GC 需要合併三個 data block，此時通道 0 就有合併三個 data block 的時間。從 log block 0A 挑選出兩個合併對象，分別為 page 410 所屬的 data block 100，page 404 所屬的 data block 102，其中 page 410 與 page 414 屬於同一個 data block。接著再從 log block 0B 挑選出 page 434 所屬的 data block 108。最後再利用通道 1 Erase “log block 1A” 的時間，通道 0 也 Erase “log block 0A” 。



圖表 13: Example: Cycle stealing

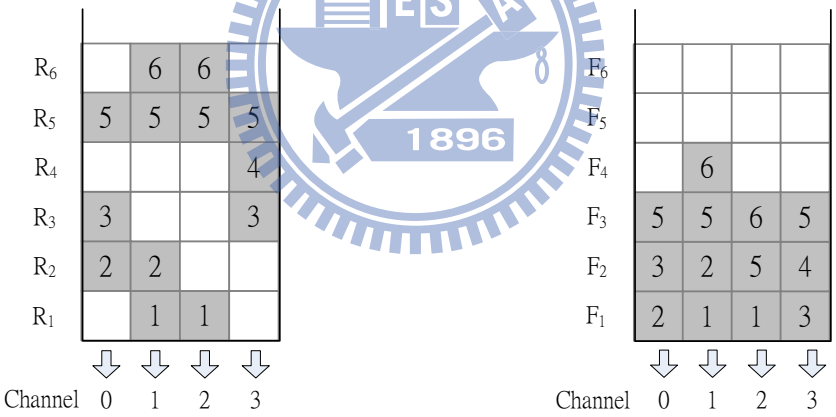
Cycle stealing 的方法使得 GC 的時候可以更平行的運作，整理來說 FTL mapping 的邏輯跟運作的方法都像是同步運作的通道一樣平行，只是把 log 區域改成獨立運作，因此具有高度的平行度。但是當不同通道的負載差很多的時候，別的通道會沒有資料可以平行合併，也是會產生閒置的問題。

3.3 Gating Buffer

為了提升使用者寫入資料的平行度，可以在 SSD 內部加入一些 RAM 來當做緩衝暫存區。在這邊緩衝暫存區的功能並不是要用來吸收 hot data，而是用來收集 request，然後再讓每一個通道同時寫入資料，這些不同通道所寫的資料可能分別來自於不同的 request。寫入的 request 可以先存放在這個緩衝區裡面，資料要寫到快閃記憶體的時候再以 FIFO 或是結合其他緩衝區管理的方式為每一個通道都選出一個 page 來做平行寫入的動作，如此每一個通道都可以寫入資料而不

會閒置。不過當然有可能在寫資料進入快閃記憶體的時候，緩衝區裡面並沒有其中一個通道所需要的資料，這個時候這個沒資料的通道也是會閒置的。

如圖表 14 為四個通道下有沒有加入 Gating Buffer 的比較，格子內的數字為 request 的編號。(a)是沒有 Gating Buffer 的寫入狀況，R 為 request 的順序，灰色格子為 request 所需占用的通道，白色格子則為寫入資料時閒置的通道。當 request 1 寫入資料到通道 1 及 2 的時候，此時已知 request 2 需要使用通道 0 及 1，但是因為通道 1 正在忙於處理 request 1，使得 request 2 無法寫入，通道 0 及 3 就會閒置；當 request 2 寫入資料到通道 0 及 1 的時候，此時已知 request 3 需要使用通道 0 及 3，但是因為通道 0 正在忙於處理 request 2，使得 request 3 無法寫入，通道 2 及 3 就會閒置；由此類推，如此造成寫入時通道利用率不佳。(b)是加入 Gating Buffer 後的寫入狀況，F 為由緩衝區寫入快閃記憶體(flush)的順序，緩衝區可以收集一些 request，要寫入資料時可以平行的將不同 request 的資料寫入快閃記憶體的每一個通道中。假設緩衝區可以存放 8 個 page 的資料，當第 5 個 request 要寫入緩衝區的時候，緩衝區裡面存放有 request 1、2、3 及 4，此時會將 F_1 那一列的資料平行寫入快閃記憶體中，緩衝區便多出 4 個 page 的空間可以寫入 request 5，然後再把 F_2 那一列的資料平行寫入快閃記憶體中。



(a) Without Buffer (b) Gating Buffer

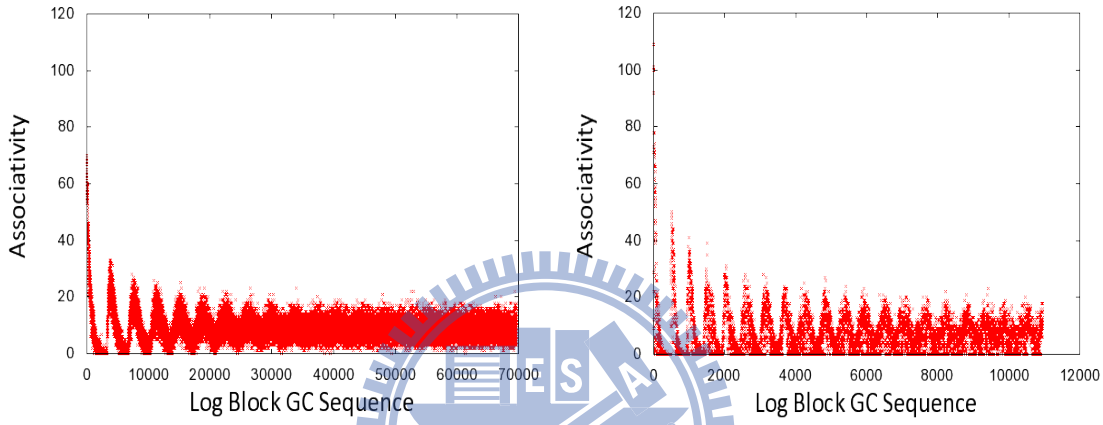
圖表 14: Usage of Gating Buffer

Gating Buffer 的優點可以藉由收集 request 來提升寫入資料的平行度。但是這樣做就需要增加硬體成本，以及需要考慮在斷電時的資料安全性問題。

3.4 Garbage Collection Throttle

Overprovisioning (log space)的大小會影響 GC 的負擔。如果 Overprovisioning 的空間小，則在快閃記憶體中的無效資料就少，GC 的對象就容易含有更多有效的資料，也就是要搬移更多的資料，因而使負擔增加。

在邏輯位址可用空間與 Overprovisioning 空間固定的情況下，當通道的數量增加時，Overprovisioning 空間平分到每一個通道中的量就會減少，使得每一個通道中的 log buffer 的深度變淺了，造成 GC 時的 associativity（associativity 是指 GC 時關聯到的 data block 數量，也就是需要合併的 data block 數量）變大，使得 GC 的負擔變大。圖表 15 是在小資料的隨機寫入下記錄每次 GC 時所執行的合併 data block 數量，使用的設定為 4k page size, 512k block size, 10% overprovision。圖表 15 (a)(b)分別為 1 個通道與 8 個通道的結果。可以看到通道在 8 個通道時一開始的峰值比較大，也需要比較多次的週期才能讓這些峰值漸趨平緩。這些 GC 合併次數的峰值，可能會超過系統等待時間，導致一些不可預期的後果。



(a) 1 channel　　(b) 8 channel

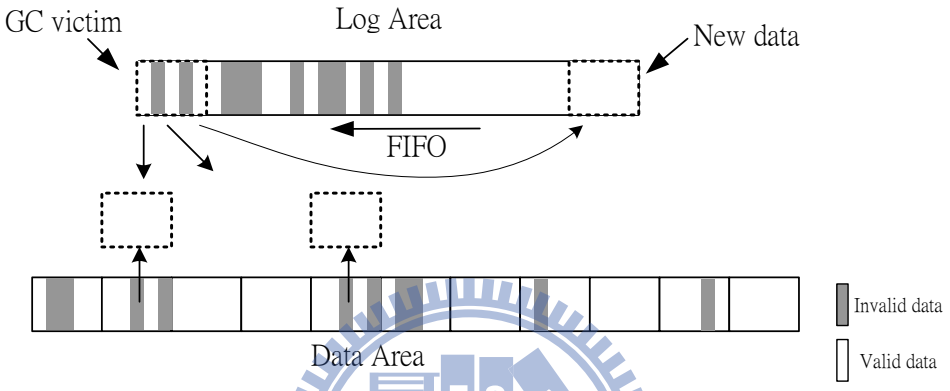
圖表 15: Large variation of GC cost

在圖表 15 中可以看到一開始 GC 的 associativity 非常高，這是因為資料是先寫入 log 區域，在 log 區域寫滿時才觸發第一次 GC，此時幾乎所有的 page 都是有效的資料（沒有更早的 GC 可以把資料合併到 data 區域），又因為是小資料隨機寫入的關係，log block 中的每一個 request 也幾乎都屬於不同的 data block。而隨著一次又一次的 GC，因為前面的 GC 會帶走後面的資料，所以慢慢的 GC 所需要的合併次數會越來越少。

在圖表 15 中也可以發現的峰值會成週期性的出現，這是因為在第一次 GC 一直到所有的 data block 都合併過的這段時間，原本在 log buffer 中的資料會大量被合併到 data 區域中，但是新寫入的資料有可能是屬於已經合併過的 data block 而不會被帶走，這些含有新資料的 log block 就會讓 GC 需要合併很多 data block，成為另一個 GC 合併次數的峰值。而這個峰值會比之前的小，因為這個 log block 中的部分資料已經被先前的 GC 合併到 data 區域中。以此類推，圖中的峰值會漸趨平緩。

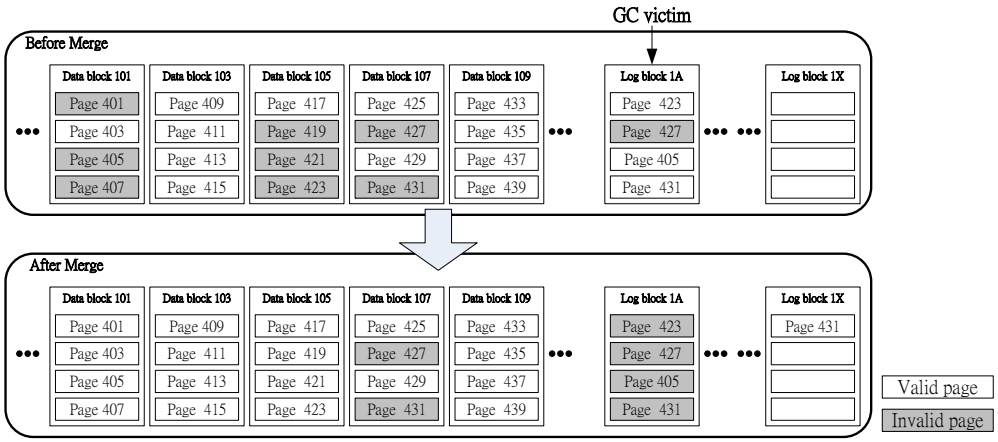
由於合併 data block 的時間是固定的，所以可以限制合併的次數來達到管制

最長的 GC 時間。GC Throttle 是用來控管每次 GC 所執行合併動作的次數，在 GC 的時候，去限制回收 log block 所需要將資料合併到 data block 的次數，而剩下沒有合併的資料就搬移到新的 log block 裡面，如此便可以使裝置的回應時間受到控管，避免 Time out 的問題。如圖表 16，GC 需要合併的資料是屬於三個 data block，所以原本是需要合併三個 data block。在加上 GC Throttle 限制為兩次合併之後，第三個需要被合併的有效 page 就不做合併動作，直接搬移到後面新的 log block 中。而這邊需要付出的成本就是搬移的動作，會加速 log buffer 空間的消耗，使 GC 的頻率增加。



圖表 16: Garbage collection throttle

如圖表 17 為 GC 前後的記憶體狀態圖。在合併前，很清楚的看到若要回收 log block 1A，需要合併 page 423、page 405 及 page 431，也就是需要合併 data block 105、data block 101 及 data block 107。若是系統限制只能做 2 次合併的操作，那在合併完 data block 105 及 data block 107 之後，剩下的 page 431 就被搬到最新的 log block 裡面去。



圖表 17: Example: Garbage collection throttle

GC Throttle 可以用來限制 GC 時所需要合併的 data block 數量而達到限制 worst case response 的效果。但是它的缺點是必須要剩下的資料搬移到新的空間，使得回收的可用空間會減少，並會增加觸發 GC 的機會。

3.5 Applicability

本論文提出的管理方法適用於多通道固態硬碟中的通道管理。這些管理方法不只可以用在 log buffer[10]這種處理方式，也可以很容易的應用於其他 FTL 演算法。以下將描述每一個方法的主要精隨。

GC Planner 是用來提升內部資料重整(Garbage Collection)的平行度。當有一個通道在處理內部資料重整的時候，其他閒置的通道可以利用這些空閒的 cycles 來處理一部分未來會遇到的內部資料重整，如此可以提升通道在執行內部資料處理的平行度。

Gating Buffer 是用來提升寫入資料的平行度。可以使用一個小小的緩衝暫存區來收集小的 request，然後在平行的寫入通道中，讓每一個通道可以同時都盡量有資料可以寫入，以提高通道在寫入資料時的平行度。

GC Throttle 是藉由限制內部資料處理的時間來穩定裝置的回應時間。一次完整的 GC 往往需要合併很多個 block，為了達到回應時間的限制與穩定性，可以將部分的資料當作是新資料遷移到新資料將寫入的位置去，限制需要合併的數量。

4 EXPERIMENTAL RESULT

這一章將會呈現實驗的結果。4.1 說明實驗的環境及使用的 workload，4.2 將針對我們所提出的三種方法進行效能測試，4.3 是將三種方法合併在一起測試不同設定的影響。

4.1 Experimental Setting

4.1.1 Environments

我們在 PC 上使用 C-language 實做了一個 SSD 模擬器(simulator)。模擬器會讀取 trace 檔案，執行模擬的過程，最後輸出模擬的結果。模擬器的各項操作時間是依照表格 1 的 SAMSUNG NAND 快閃記憶體模組資料[1][2]來設定，包含執行的忙碌時間及資料的傳輸時間。設定如表格 2 所示，Write 一個 page 的時間 MLC 為 906ms、SLC 為 306ms，Read 一個 page 的時間 MLC 為 131ms、SLC 為 166ms，Erase 一個 block 的時間皆為 1500ms。因為讀取的動作並不會改變快閃記憶體內部的狀態，這個模擬器會只針對寫入的 request 做動作，會忽略所有的讀取 request。另外也沒有考慮 SSD 中各種硬體的延遲及微處理器運算時間，畢竟這些時間所佔的比例極低。

Type	SLC	MLC
Page Size	2K	4K
Block Size	128K	512K
Erase Block	1500 μs	1500 μs
Write Page	306 μs	906 μs
Read Page	131 μs	166 μs

表格 2: Simulated Timing

模擬器所設定的快閃記憶體總量是邏輯位址的大小加上 overprovision 的大小，邏輯位址的大小是根據 trace 檔案所存取的範圍來決定，而 overprovision 的大小是根據邏輯位址的大小來設定一定比例的 overprovision。整個快閃記憶體的總容量會平分在每一個通道內，每一個通道內的邏輯位址與 overprovision 的大小皆相等，所以通道數量越多，每一個通道內的容量就越少。

本章的實驗將藉由調整不同的通道數量、overprovisioning 大小以及不同型態的快閃記憶體來觀察與比較效能的差異。實驗將以比較 IOPS 為主，就是每秒鐘

執行 Input / Output 的次數。

4.1.2 Workload

使用的 workload 總共有四種，分別為 Random、Sequential、WinXP 及 Ubuntu 四種 trace。Random、Sequential 及 WinXP 是在 windows 環境下使用 DiskMon 來收集對硬碟的存取動作。Ubuntu 是在 ubuntu 環境下使用 Blktrace 來收集對硬碟的存取動作。表格 3 為四種 workload 的特性。

Trace	IOmeter	Sequential	WinXP	Ubuntu
Logical space	16 GB	20 GB	40 GB	30 GB
Average request size	4 KB	60 KB	11.46 KB	20.01 KB

表格 3: Workload

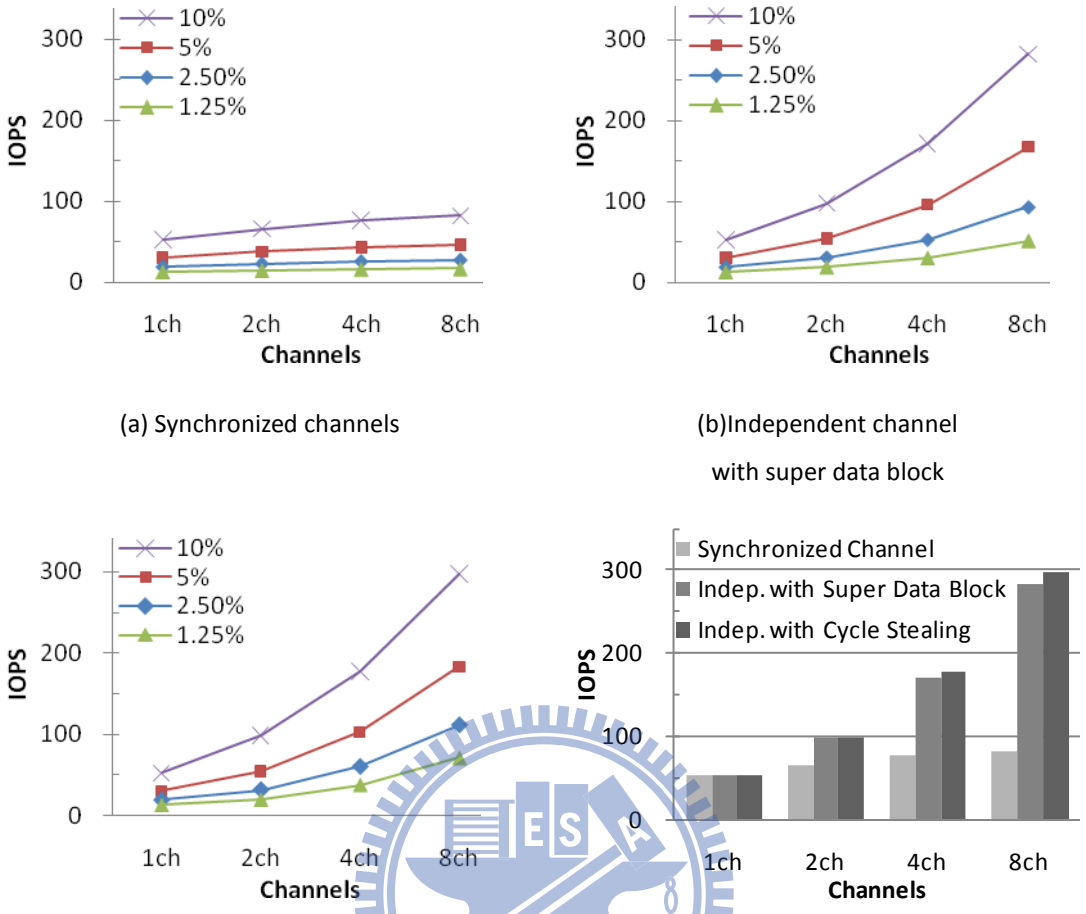
Random trace 是在 windows 的環境下使用 IOmeter [5]對 16G 的硬碟分割進行 100%隨機寫入的動作，每一個 request 皆為 4KB。**Sequential trace** 是在 windows 的環境下對 20G 的硬碟空間進行複製 MP3 及電影等大檔案，製造出大量連續寫入的情況。**WinXP trace** 是在 windows 的環境下對 40G 的硬碟空間正常使用 windows 來上網瀏覽、編輯文件，下載檔案等等的操作所收集到的 trace，其內容混合了連續存取和隨機存取。**Ubuntu trace** 是在 ubuntu 的環境下對 30G 的硬碟空間正常使用 ubuntu 來上網瀏覽、編輯文件，下載檔案等等的操作所收集到的 trace，其內容混合了連續存取和隨機存取。其中因為 WinXP trace 所含的小 request 比 Ubuntu trace 多，所以 WinXP trace 的平均 request 比較小。

4.2 Mechanisms Evaluation

在這一節中我們將使用 Random trace 在 MLC（page size 4KB，block size 512KB）的快閃記憶體下來量測我們所提出的三種多通道的管理機制，因為 Random trace 為小資料的隨機寫入動作，在這樣惡劣的寫入模式下，可以更容易觀察出多通道管理方法的必要性與優點。

4.2.1 Garbage Collection Planner

這一小節將對 3.2 節中的三種 GC 平行方法作效能測試，分別為原本的同步運作通道、獨立通道加上 super data block 以及獨立通道加上 cycle stealing。圖表 18 為三種方法使用 Random trace 在 MLC 的快閃記憶體下，不同的通道及不同 overprovision 的比較，其中橫軸為通道的數量，縱軸為 IOPS，圖表 18(a)(b)(c)上的四條線分別為 1.25%、2.5%、5%、10%的 overprovision。



(a) Synchronized channels

(b)Independent channel with super data block

(c) Independent channel with cycle stealing

(d) Different channel management

圖表 18: Effect of channels with different channels

圖表 18 (a)為同步運作通道在不同比例的 overprovision 下增加通道數量所測量出來的結果。可以發現隨著通道數量增加，效能不會有明顯的增加，曲線會漸趨平緩成為開口向下的曲線。由於同步通道對快閃記憶體的最小存取單位為一個 super page，super page 的大小為 page 的大小乘上通道的數量。而 Random trace 每一次寫入的資料只有 4KB，小於 super page 的大小，需要做 read-modify-write，並消耗掉一個 super log page 的空間。因此空間浪費的缺點與多通道平行 GC 回收較多空間的優點幾乎抵消了。也就是說在一個通道的時候，每一次寫入消耗一個單位的空間，每次 GC 回收一個單位的空間；而在 8 通道的時候每一次寫入消耗 8 單位的空間，每次 GC 也回收 8 單位的空間。但是增加通道還是會讓效能有些微提升，是因為每一個 request 並不只寫入一個 super page。Request 的起始位址並不會對齊在 page 的邊界上，只會對齊 sector 的邊界（每一個 sector 為 512 bytes），所以 request 會有機會橫跨兩個 super page。在 1 個通道的時候，一個 request 寫入兩個 page 的機率是 7 / 8 = 87.5%；在 8 個通道的時候，一個 request 寫入兩個 page 的機率是 7 / 64 = 10.9%。由此可見增加通道會降低空間的浪費，

因而效能有些微的提升。

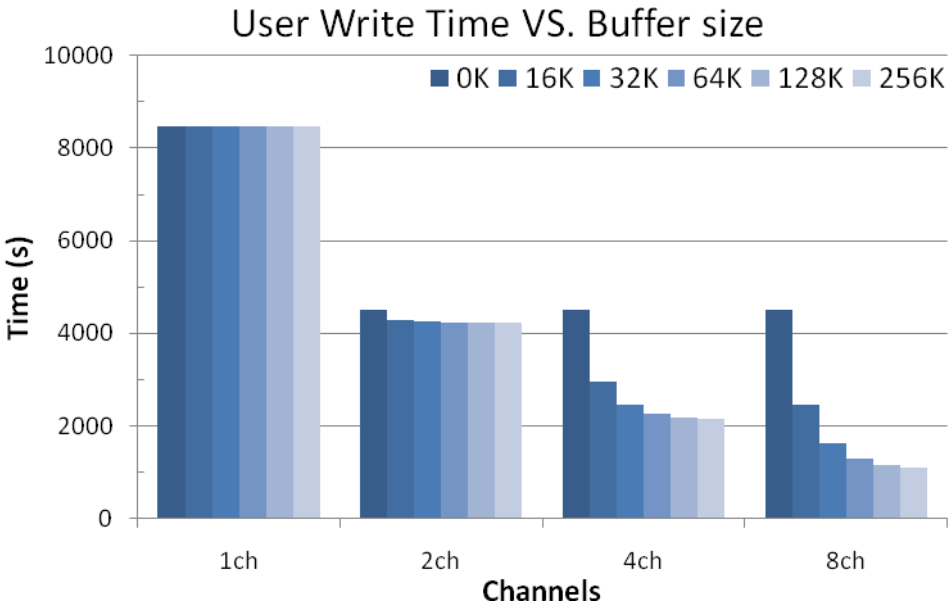
圖表 18 (b)為獨立通道搭配 super data block 在不同比例的 overprovision 下增加通道數量所測量出來的結果。可以發現隨著通道數量增加，效能會有明顯的增加，上升的趨勢成為開口向上的曲線。這是因為獨立通道不會將不同通道的 page 綁死成為 super log page，所以每一個 request 寫入所消耗的空間並不會因為通道數量增加而增加。因此通道數量增加時，可以充分展現出通道平行 GC 回收更多空間的特性，如此才是增加通道所該獲得的效能。

圖表 18 (c)為獨立通道搭配 Cycle Stealing 在不同比例的 overprovision 下增加通道數量所測量出來的結果。這個結果與獨立通道搭配 super data block 的結果類似，但是 Cycle Stealing 在平行合併的時候可以回收更有效益的空間，所以效能提升的趨勢會比較多一些。

圖表 18 (d)為三種 GC 平行方法在 10% overprovision 下增加通道數量所測量出來的結果比較圖。可以看到獨立通道搭配 Cycle Stealing 可以更有效的利用多通道。隨著通道數量的提升，同步運作的通道在小資料寫入的時候會浪費更多的空間而使 GC 頻率提升，抵銷掉多通道平行 GC 的優點。合併 super data block 的方法有可能出現無意義的合併或是過早的合併，使得效能不如 Cycle Stealing，而且隨著通道數量增加也會讓出現無意義的合併或過早的合併機率增加。

4.2.2 Gating Buffer

這一小節我們將對 Gating Buffer 做測試，藉由調整緩衝區的大小來觀察其效果。圖表 19 為獨立通道搭配 Cycle Stealing 並使用 Random trace 在 MLC 的快閃記憶體下，不同緩衝區大小及不同通道數量的比較。緩衝區的管理將以 page 為單位搭配簡單的 FIFO 來運作。在這邊的縱軸我們使用寫入的次數（write cycles）來做比較，因為加入 Gating Buffer 主要的目的是要改善寫入的平行度，對於 GC 並不會有影響。同理，overprovisioning 也不會影響寫入的平行度。



圖表 19: Effect of Gating Buffer size

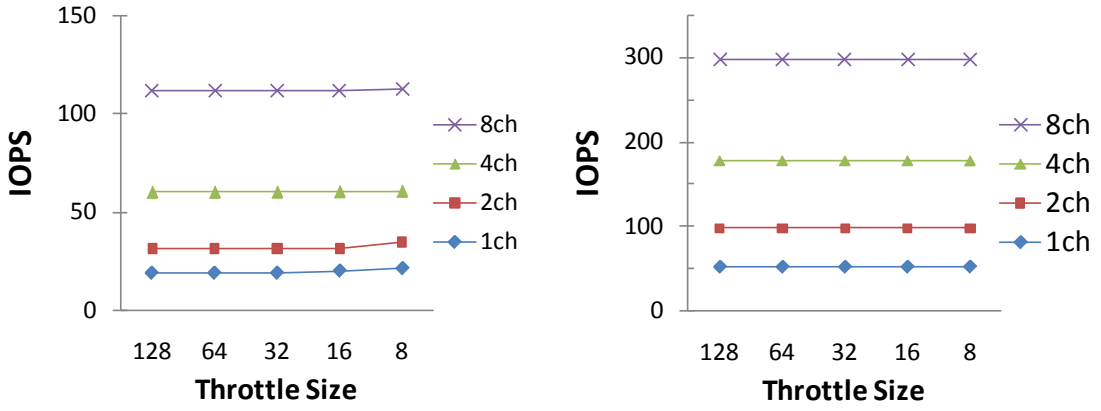
在圖表 19 中，只有一個通道的時候，加大緩衝區空間並不會減少寫入的次數，因為沒有其他通道可以平行寫入。有四個通道的時候，只需要 16KB 的緩衝區空間就可以讓效能有大幅度的提升。因為 16KB 為四個 page 的大小，剛好可以讓四個通道各保留一個 page 的資料，而得以平行寫入。但是因為 request 並不會每次都要求寫入不同的通道，緩衝區裡面有時候會有一些 page 是屬於同一個通道的資料，在寫入的時候就無法完全平行寫入，所以繼續增加緩衝區空間仍然可以增加平行寫入的機會，但是增加的效益就會慢慢減少。在八個通道下，緩衝區空間超過 8 個 page（32KB）之後，效益也會漸趨平緩。在兩個通道的時候我們可以看到效果增加的不明顯，其原因是因為隨機寫入 4KB 的資料，有 7 / 8 = 87.5%的機率會寫入兩個 page，所以在兩個通道的時候本身就是平行度很高的結果，能改善的部分只有 12.5%只寫一個 page，而不平行的部分。

4.2.3 Garbage Collection Throttle

這一小節我們將對 GC Throttle 做測試，藉由調整限制的數量來觀察其效果。圖表 20 為獨立通道搭配 Cycle Stealing 並使用 Random trace 在 MLC 的快閃記憶體下，不同 Throttle 限制的數量及不同通道數量的比較。縱軸為 IOPS，橫軸為限制合併 data block 的數量，128 表示完全不限制（最多就是合併 128 次）；8 為限制合併次數最多為 8，超過的部分就搬到新的 log block 中。

圖表 20 (a)為 2.5% overprovision 的結果，圖表 20 (b) 為 10% overprovision 的結果。可以看到在 2.5% overprovision 下，一個通道及兩個通道在限制合併次數為 8 的時候負擔會比較大。在 overprovision 少的情況下或是通道數量少的情況

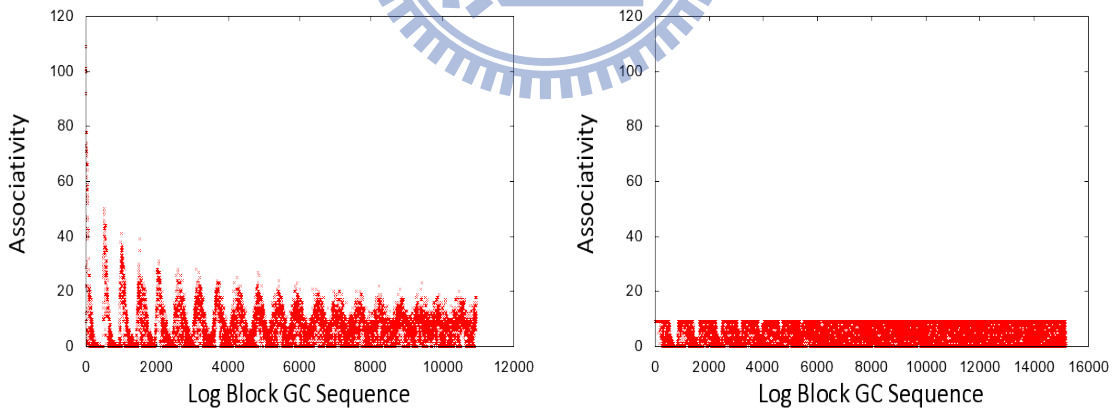
下，因為 GC 的負擔比較沉重，所以搬移到新的 log block 的資料會比較多，所以負擔會比較大，但是隨著通道數量的增加或是 overprovision 的增加，負擔會逐漸減輕。



(a) 2.5% overprovision　　(b) 10% overprovision

圖表 20: Overhead of GC Throttle

圖表 21 是在小資料的隨機寫入下記錄每次 GC 時所執行的合併 data block 數量，使用的設定為 4k page size, 512k block size, 10% overprovision。圖表 21 (a)為原始的狀況，圖表 21 (b)為加入 Throttle 限制合併次數為 8 的狀況。可以看到 GC 的時間都限制在合併 8 個 data block 以下。



(a) Original　　(b) GC Throttle: 8

圖表 21: Effect of GC Throttle

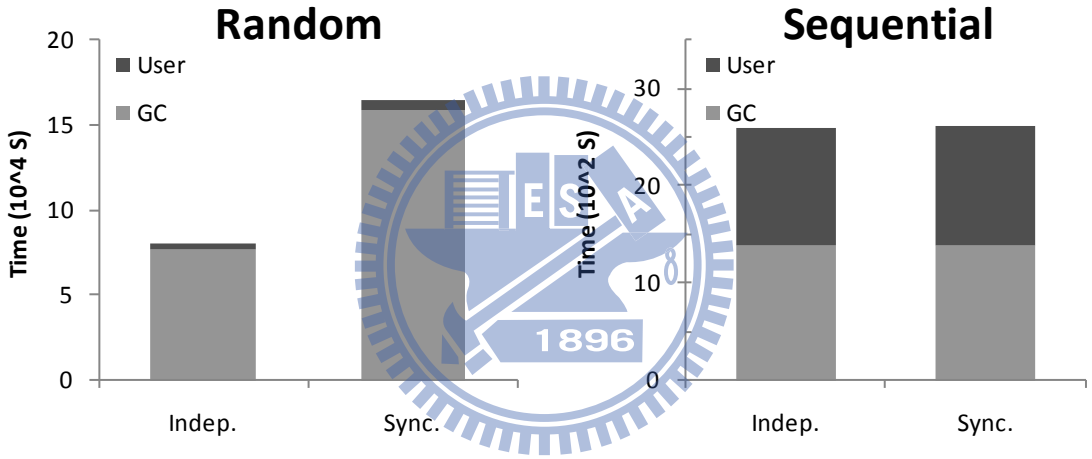
4.3 Overall Performance

在這一節中我們針對三種通道管理方法的最佳組合，調整各種參數來觀察整體的表現。以下將針對獨立通道搭配 Cycle Stealing 以及同步運作通道兩個方法進行實驗。為了更公平的比較，我們同時給予兩種運作機制 32KB 的緩衝區空間，

以及 Throttle 皆限制為 8。緩衝區的管理將以 page 為單位搭配簡單的 FIFO 來運作。4.3.1 將觀察不同 workload 的影響，4.3.2 將觀察不同通道數量的影響，4.3.3 將觀察不同 overprovision 比例的影響，4.3.4 將觀察不同種類的快閃記憶體的影響。

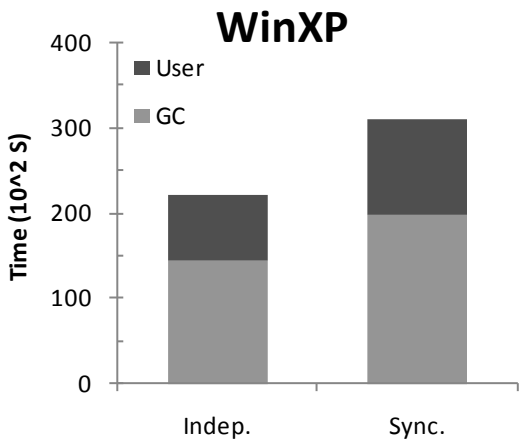
4.3.1 Effect of Workload

這一個小節我們將觀察不同 workload 所造成的影響。使用的硬體設定為四個通道、2.5% overprovision、MLC 的快閃記憶體模組、32KB 的緩衝區空間及 Throttle 設定為 8。圖表 22 為四種不同 trace 針對獨立通道搭配 Cycle Stealing 以及同步運作通道的實驗結果，縱軸為跑完整個 trace 所花費的時間。為了方便觀察其平行度，長條圖將分為使用者 request 寫入所花費的時間（包含 read-modify-write 的 read）以及內部處理 GC 所花費的時間兩部分。

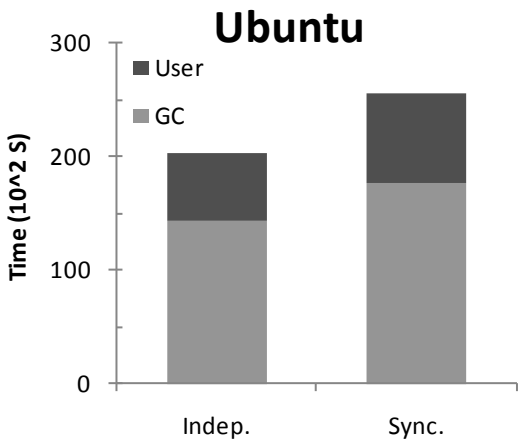


(a) Random trace

(b) Sequential trace



(c) WinXP trace

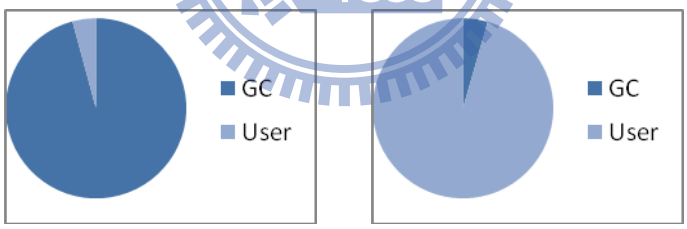


(d) Ubuntu trace

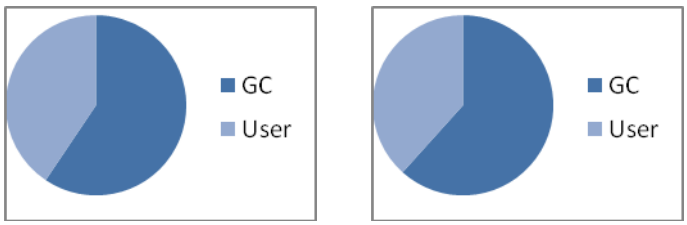
圖表 22: Effect of workload

在圖表 22 中，我們可以看到不管在哪一種 trace，獨立通道搭配我們提出的三種管理方式都比同步運作通道效能還要好。使用者 request 寫入所花費的時間由 Gating Buffer 來改善，內部處理 GC 所花費的時間由獨立通道搭配 Cycle Stealing 來改善。Gating Buffer 在獨立通道表現比較好，因為獨立通道可以同時平行寫入不同的 request。同步運作的通道因為必須要寫入 super page 而無法平行寫入屬於不同 super page 的 request。獨立通道搭配 Cycle Stealing 可以大幅改善 GC 效能在 4.2.1 中有詳細的說明，主因為小資料隨機寫入造成浪費空間使 GC 觸發頻率增加。在圖表 22 (b)中，因為大部分的 request 都是連續的寫入，兩者的效能幾乎都一樣，但是其中還是會參雜一些小資料的隨機寫入，所以獨立通道搭配 Cycle Stealing 還是會比較好一點點。

圖表 23 中獨立通道搭配 Cycle Stealing 的效能會比同步運作通道來的好、時間花得比較少。其時間減少可以分為使用者 request 寫入所花費的時間減少與內部處理 GC 所花費的時間減少兩部分，圖表 23 將這兩部分畫成圓餅圖，由此來觀察這兩部份的貢獻程度。在圖表 23 (a) Random trace 中，因為小資料隨機寫入的原因，所以主要的貢獻為獨立通道改善浪費空間的問題。而在圖表 23 (c) WinXP 及(d) Ubuntu 這兩個正常使用的 trace 就可以發現 Gating Buffer 增加寫入平行度有超過三分之一的貢獻。甚至在圖表 23 (b)複製大檔案時，因為 GC 的效益很高，Gating Buffer 就成為了主要的貢獻。



(a) Random trace　　(b) Sequential trace



(c) WinXP trace　　(d) Ubuntu trace

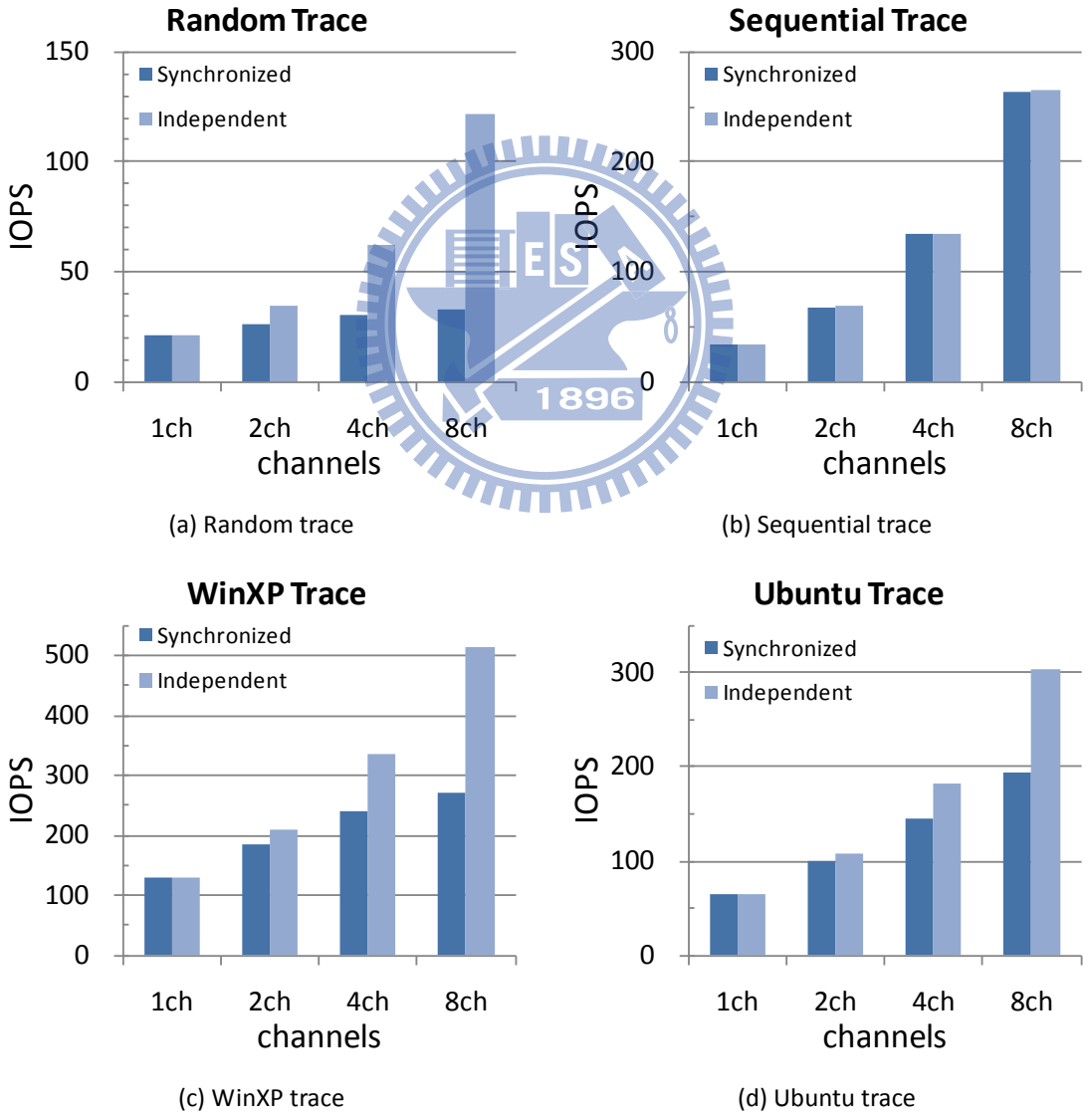
圖表 23: Contribution of parallelism

4.3.2 Workload VS Channels

這一個小節我們將觀察提升通道數量對不同 workload 所造成的影響。使用的硬體設定為 2.5% overprovision、MLC 的快閃記憶體模組、32KB 的緩衝區空間

及 Throttle 設定為 8。圖表 24 為不同通道數量在獨立通道搭配 Cycle Stealing 以及同步運作通道的實驗結果，縱軸為 IOPS，橫軸為通道的數量。

在圖表 24 中，我們可以發現獨立通道搭配 Cycle Stealing 的方法會隨著通道增加而效能比同步運作通道更好。如同在 4.2.1 中所說，小資料的隨機寫入會讓同步運作的通道因為浪費空間而增加 GC 的頻率，隨著通道增加浪費空間的情況會越來越嚴重，使得增加通道並無法大幅提升效能。而獨立通道不僅不會浪費空間，還可以利用 Gating Buffer 收集不同通道的 request，以便於同時平行寫入。同步運作的通道只有在大量連續資料寫入的情形下才會表現出多通道的優勢，而我們所提出的通道管理方法，不管在哪一種 workload 下，都可以充分表現出多通道平行的優勢。



(a) Random trace

(b) Sequential trace

(c) WinXP trace

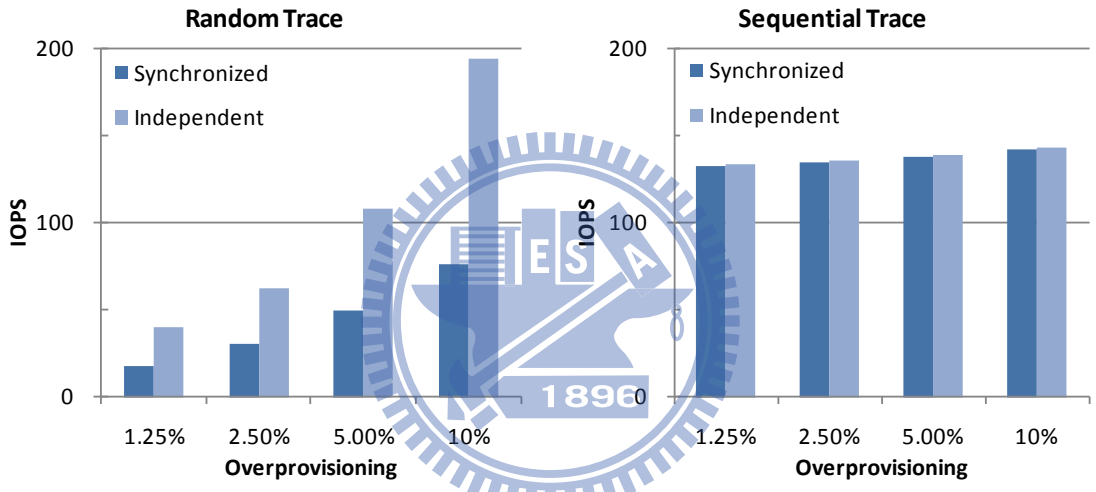
(d) Ubuntu trace

圖表 24: Effect of channels

4.3.3 Effect of Overprovisioning

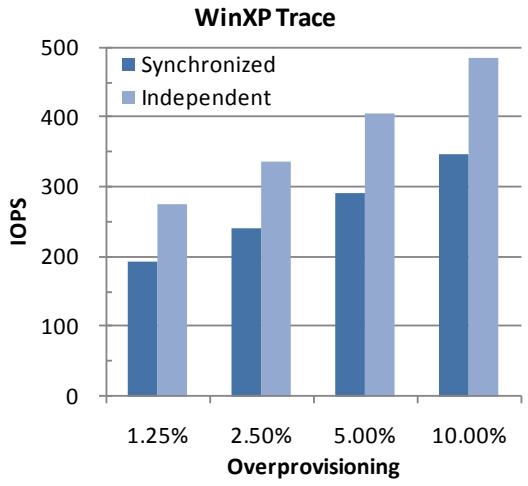
這一個小節我們將觀察提升 overprovision 的比例所造成的影響。使用的硬體設定為四個通道、MLC 的快閃記憶體模組、32KB 的緩衝區空間及 Throttle 設定為 8。圖表 25 為不同通道數量在獨立通道搭配 Cycle Stealing 以及同步運作通道的實驗結果，縱軸為 IOPS，橫軸為 overprovision 的比例。

在圖表 25 中，我們可以發現隨著 overprovision 的比例增加，效能會越來越好。而且隨著 overprovision 的比例增加，四種 workload 效能增加的比例為 (a)>(c)>(d)>(b)。這是因為越多的 log 區域可以存放並收集越多屬於同一個 data block 的資料，然後在一起回收減少 GC 的負擔。所以小資料隨機寫入的比例越多，增加 overprovision 比例就可以獲得越多的效益。

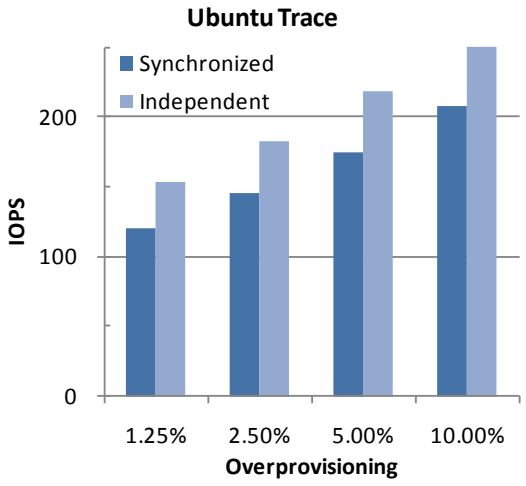


(a) Random trace

(b) Sequential trace



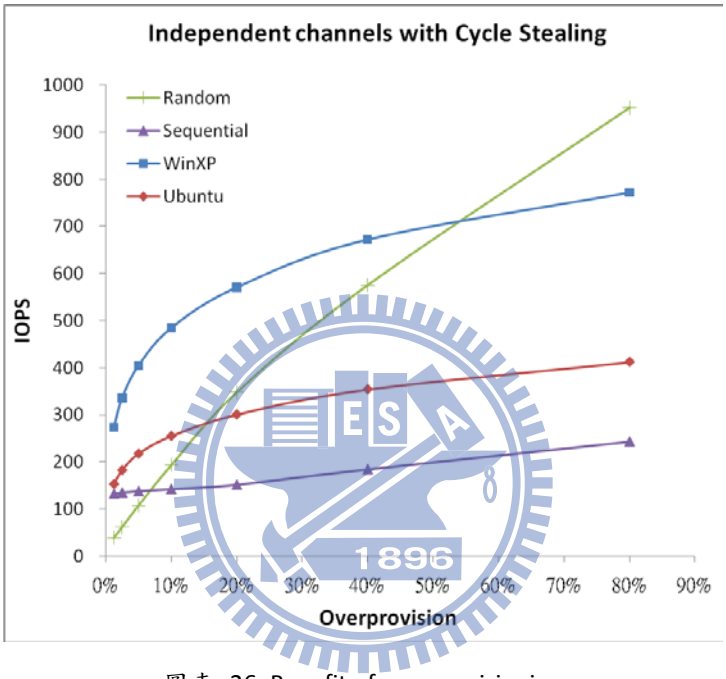
(c) WinXP trace



(d) Ubuntu trace

圖表 25: Effect of overprovisioning

圖表 26 為四種 workload 在獨立通道搭配 Cycle Stealing 下增加 overprovision 比例所測量的結果。使用的硬體設定為四個通道、MLC 的快閃記憶體模組、32KB 的緩衝區空間及 Throttle 設定為 8。可以看到 Random trace 可以隨著 overprovision 的空間增加而大幅度上升效能，而 Sequential trace 上升的效果最為平緩。在一般使用的 workload 中，一開始效能會有大幅度的增加，但是到 overprovision 比例超過大約 10%之後，效能的提升就會漸趨平緩，而 Ubuntu trace 會比 WinXP trace 更早平緩。這就是因為小資料隨機寫入的比例越高，就需要越多的 log 區域，而當 log 區域的空間超過所需之後就會漸趨平緩。

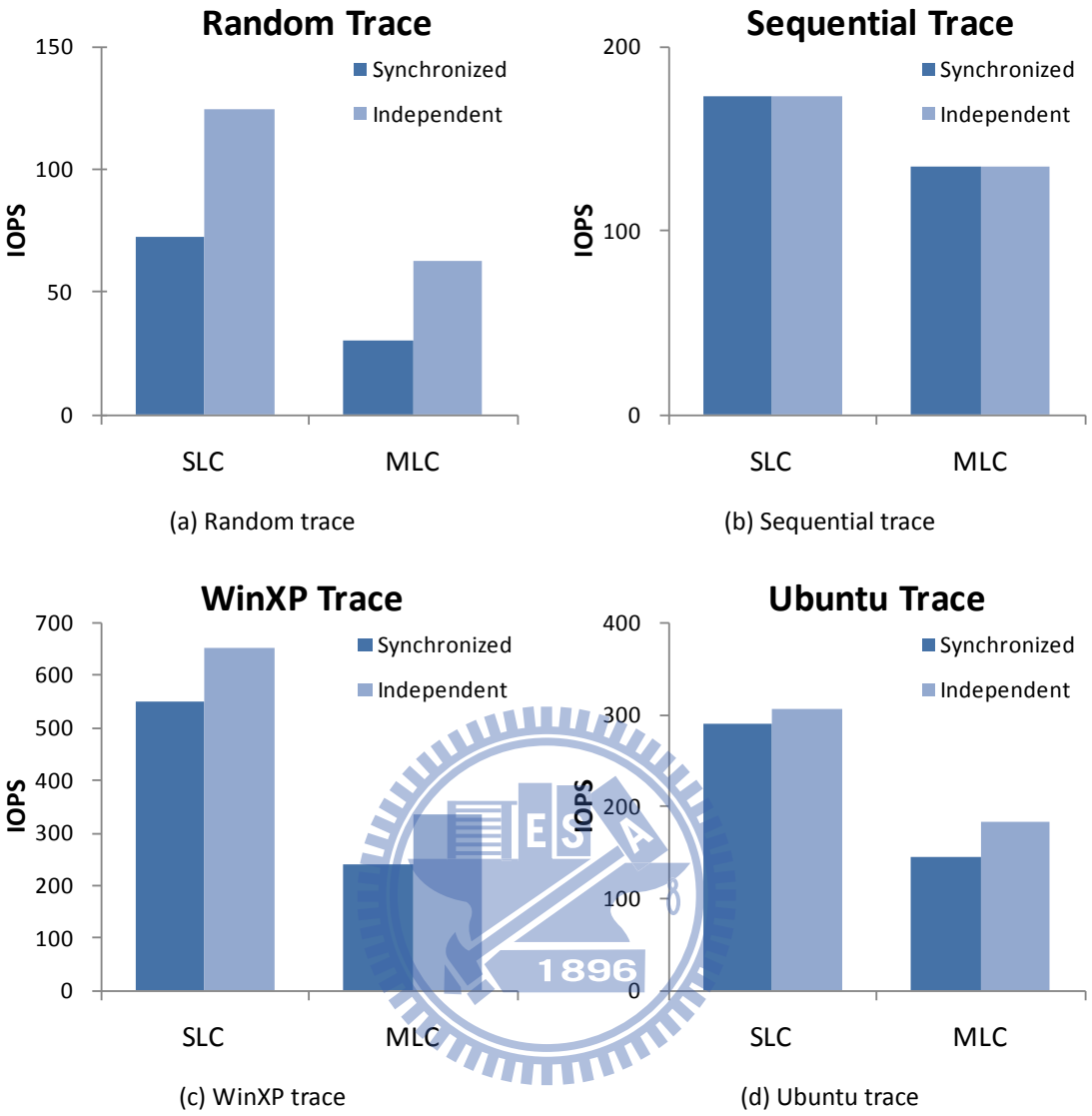


圖表 26: Benefit of overprovisioning

4.3.4 Effect of Geometry

這一個小節我們將觀察不同種類的快閃記憶體所造成的影響。使用的硬體設定為四個通道、2.5% overprovision、MLC 的快閃記憶體模組、32KB 的緩衝區空間及 Throttle 設定為 8。圖表 27 為不同種類的快閃記憶體在獨立通道搭配 Cycle Stealing 以及同步運作通道的實驗結果，縱軸為 IOPS，橫軸為 MLC（page size 4KB，block size 512KB）及 SLC（page size 2KB，block size 128KB）。

在圖表 27 中，我們可以發現 SLC 的效能都會比 MLC 來的好，這主要的原因是因為 MLC 寫入 4KB 需要 906 ms 而 SLC 寫入 4KB 只需要 306 X 2 = 612 ms 的原因。另外也可以發現在 MLC 中獨立通道可以贏比較多，這是主要是因為 MLC 的 page 比較大，同步運作的通道會有比較大的 super page，所以在面對小資料隨機寫入的時候會浪費更多的空間。



(a) Random trace

(b) Sequential trace

(c) WinXP trace

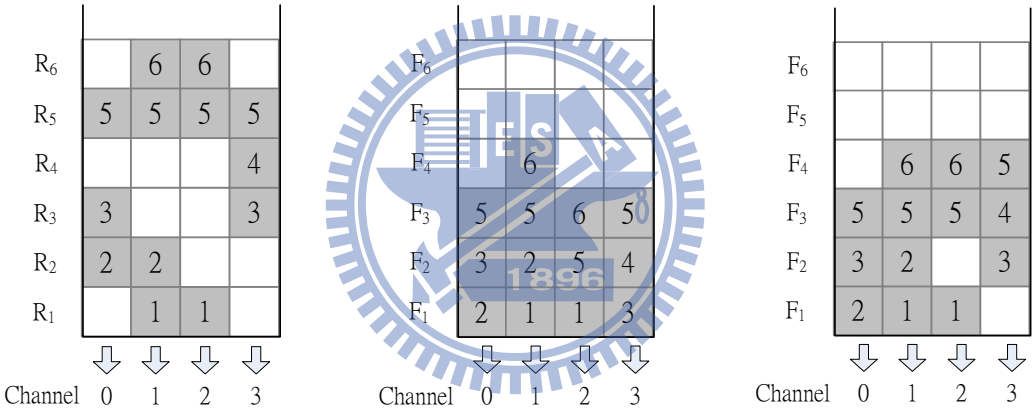
(d) Ubuntu trace

圖表 27: Effect of Geometry

5 CASE STUDY: Integrity of Buffer

在比較高階的 file system 中，對於儲存裝置會利用 journal 或 log 的方式做保護，而重要的資料也會緊跟著 flush cache 的命令。但是有些存取儲存裝置的系統並不會有這一類的保護機制，像是 embedded file system。因此資料若存於儲存裝置的 buffer cache 中，並寫以 out-of-order 的方式寫入，勢必會有一些安全性上的疑慮，更有可能導致使用此儲存裝置的系統損壞。這一章要針對 write buffer 對寫入資料安全性的影響做討論。

在 3.3 Gating Buffer 中的方法，有可能會讓比較後面的 request 的部分資料優先寫入快閃記憶體中，而這樣不同 request 交錯寫入的過程若遇到斷電，有可能會出現 out-of-order 寫入而造成安全性上的疑慮。在此考慮讓資料以 in-order 的方式寫入，也就是說不同 request 只能同時寫入，不能提早寫入。

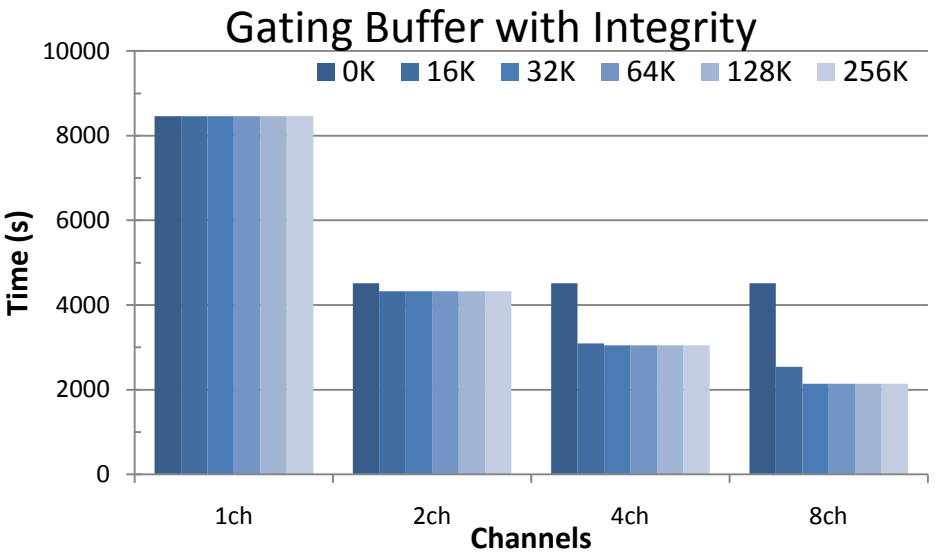


(a) Without Buffer (b) Gating Buffer (c) Integrity of Buffer

圖表 28: Strategy of buffer

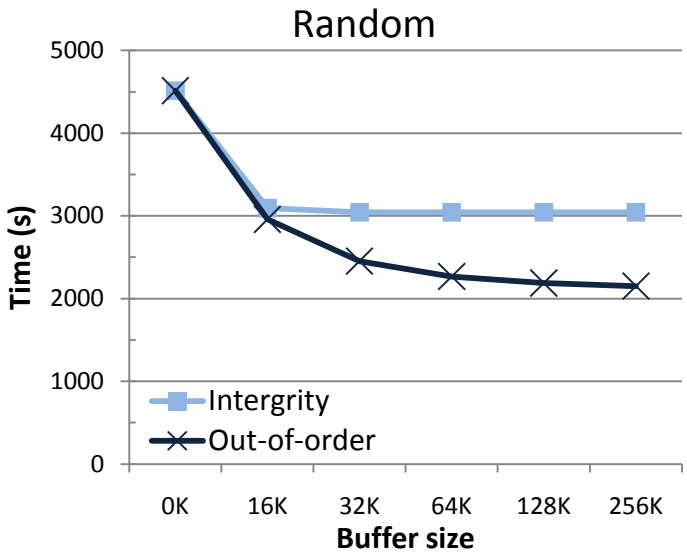
圖表 28 為四個通道下不同緩衝區管理方法的比較圖，格子內的數字為 request 的編號，圖表 28 (a)為沒有緩衝區的情況下寫入的狀況，圖表 28 (b)為原本的 Gating Buffer，而圖表 28 (c)為 integrity 的方法。可以清楚的看到圖表 28 (c)中因為 request 2 與 request 1 有重複使用到通道 1，造成無法平行寫入，而 request 3 就無法遞補到閒置的通道 3 之中。因此通道數量若為 N，則在寫入 N 個 page 之間，若有重複使用到同一個通道的情況就會讓平行度降低。

以下為獨立通道搭配 Cycle Stealing 並使用 Random trace 在 MLC 的快閃記憶體下，不同緩衝區大小及不同通道數量的比較。緩衝區的管理將以 page 為單位搭配簡單的 FIFO 來運作，並考慮 in-order 的寫入。縱軸使用寫入的次數（write cycles）來做比較。



圖表 29: Gating Buffer with Integrity

圖表 29 為 Integrity 的實驗結果，可以看到在四個通道下，緩衝區在 16KB 的時候有大幅度的改善，這是因為緩衝區的空間剛好足夠平均非配給每一個通道一個 page size 的空間，而達到收集 request 的效果。而緩衝區大小大於 32KB 之後就沒有改善的效果，這是因為緩衝區裡面的其他 request 受到 in-order 的限制，並無法超前寫入，所以加大緩衝區空間雖然可以留住更多 request 但是並不能超前填補空閒的通道，因此留住更多 request 並無增加平行度的效果。同理，兩個通道與八個通道也在超過 16KB 與 64KB 之後就沒有改善的效果。



圖表 30: Compare with Integrity

圖表 30 中的兩條線為四個通道下有無 Integrity 的比較，可以清楚的看到在 16KB 的時候兩種方法都可以表現出收集 request 的效果，而加大緩衝區空間之後，Integrity 的方法無法有效利用增加的緩衝區來收集後面的 request。這主要是因為 Integrity 的方法中，相鄰的 request 在通道的使用上會有重疊的情況而無法同時寫入，此時剩餘空閒的通道也無法由後面的 request 來遞補。

Integrity 的方法讓 request 以 in-order 的方式寫入，優點是增加安全性的考量，這是一種設計上的選擇。缺點是會犧牲掉一些 Gating Buffer 的效果。

6 CONCLUSION

本篇論文針對 SSD 的硬體架構提出三種多通道的管理機制，分別為 GC Planner、Gating Buffer 及 GC Throttle。主要在解決小資料隨機寫入在多通道下的各種問題，包括同步通道的空間利用率不佳、獨立通道的通道利用率不佳以及 GC 等待時間過長的問題。在使用獨立通道的硬體架構下，Gating Buffer 可以提升寫入的平行度，GC Planner 採用 Cycle Stealing 可以提升 GC 的平行度，而 GC Throttle 可以在很微小的負擔下穩定的限制 GC 的回應時間。本論文所提出的管理機制具有以下幾項優點：第一是可以保持大量連續資料存取的高效能。第二是增加空間的利用率。第三是提升多通道運作的有效平行度。第四是維持穩定的回應時間。第五是適用於各種 FTL 的演算法。我們的方法在 8 個通道及 2.5% overprovision 的情況下，小資料隨機寫入的效能為同步運作通道的 3.72 倍。

我們在實驗中有以下發現：首先，Cycle Stealing 可以提升獨立通道執行 GC 的平行度，並可以避免太早 merge 的情況。第二，只需要很小緩衝區空間來收集 request，就可以讓寫入的平行度大幅提升。第三，GC Throttle 所付出的負擔相當的小，隨著通道數量或是 overprovision 增加，負擔會更小。第四，在不同 workload 中，因為 GC 與使用者寫入所花的時間比例不同，Gating Buffer 與 GC Planner 會有不同的重要性。最後，在正常的使用下，overprovision 超過 10%以後，所獲得的效益就會慢慢減少。

本論文所未提及的 well-leveling 議題及靜態 striping 所面臨的負載平衡（load balance）問題將會是接著需要考慮的因素。另外，將提出的通道管理方法應用在其他各種不同 FTL 上的效果，也將會是我們接著需要實驗與觀察的部分。

REFERENCE

[1] 2Gx8 bit NAND flash memory (K9XXG08UXA), Samsung Electronics, 2006.

[2] 2Gx8 bit NAND flash memory (K9XXG08UXM), Samsung Electronics, 2006.

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

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

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

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

[7] DiskMon for Windows v2.01, http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

[8] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. 1994. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.* 26, 2 (Jun. 1994), 145-185. DOI= http://doi.acm.org/10.1145/176979.176981

[9] Kim, J., Kim, J. M., Noh, S.H., Min, S. L., Cho, Y. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Trans. Consu. Electrns.* 48, 2 (May. 2002), 366-375. DOI= 10.1109/TCE.2002.1010143

[10] 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

[11] 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

[12] 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.

[13] Chang, L. and Kuo, T. 2002. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems. In *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (Rtas'02)* (September 25 - 27, 2002). RTAS. IEEE Computer Society, Washington, DC, 187.

[14] Park, C., Talawar, P., Won, D., Jung, M., Im, J., Kim, S., Choi, Y. 2006. A High Performance Controller for NAND Flash-based Solid State Disk (NSSD). In *Proceedings of the 21st Non-Volatile Semiconductor Memory Workshop.* (Feb. 12 - 16, 2006), *NVSMW IEEE.* 17-20. DOI= 10.1109/.2006.1629477

[15] Chang, Y. and Chang, L. 2008. A self-balancing striping scheme for NAND-flash

storage systems. In *Proceedings of the 2008 ACM Symposium on Applied Computing* (Fortaleza, Ceara, Brazil, March 16 - 20, 2008). SAC '08. ACM, New York, NY, 1715-1719. DOI= http://doi.acm.org/10.1145/1363686.1364094

[16] Kang, J., Kim, J., Park, C., Park, H., and Lee, J. 2007. A multi-channel architecture for high-performance NAND flash-based storage system. *J. Syst. Archit.* 53, 9 (Sep. 2007), 644-658. DOI= http://dx.doi.org/10.1016/j.sysarc.2007.01.010

[17] 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.

[18] 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

[19] Seol, J., Shim, H., Kim, J., and Maeng, S. 2009. A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks. In *Proceedings of the 2009 international Conference on Compilers, Architecture, and Synthesis For Embedded Systems* (Grenoble, France, October 11 - 16, 2009). CASES '09. ACM, New York, NY, 137-146. DOI= http://doi.acm.org/10.1145/1629395.1629416

[20] Seong, Y. J., Nam, E. H., Yoon, J. H., Kim, H., Choi, J., Lee, S., Bae, Y. H., Lee, J., Cho, Y., and Min, S. L. 2010. Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture. *IEEE Trans. Comput.* 59, 7 (Jul. 2010), 905-921. DOI= http://dx.doi.org/10.1109/TC.2010.63