# 國立交通大學

## 資訊科學與工程研究所

## 碩 士 論 文

# 固態硬碟結構偵測方法

# Geometry Detection for Solid-State Drives

研 究 生：郭郡杰

指導教授：張立平　教授

中 華 民 國　九 十 八　年 七 月

# 固態硬碟結構偵測方法
# Geometry Detection for Solid-State Drives


研 究 生：郭郡杰　　Student：Chun-Chieh 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

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

# 固態硬碟結構偵測方法

學生 : 郭郡杰　　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

最近，以 NAND flash memory 為基礎的 Solid State Drive (SSD)正快速的發展，並且已經實際運用於筆記型電腦、數位攝影機、UMPC 等嵌入式產品中，更有多家業者提出以 SSD 為主要儲存裝置的伺服器解決方案。然而過去的二十年中，許多研究都致力於 Hard Disk Drive (HDD)，因此整個儲存系統架構都針對 HDD 的物理特性進行最佳化，如作業系統、檔案系統、應用程式甚至資料庫的設計等。但 SSD 並不單純只是存取速度較快的 HDD；由於 NAND flash memory 與磁碟截然不同的物理特性，傳統對於 HDD 的最佳化方式並不能直接套用在 SSD 中，甚至可能會帶來反效果。因此在這篇論文中我們想藉由了解 SSD 內部的硬體架構以及其韌體管理演算法，來對整個 SSD 的效能表現進行完整的分析。並且對於明顯影響效能的 SSD geometry 設計出一系列的測試方法，讓使用者能從產品外部藉由存取測式取得各種影響效能表現的參數，如基本存取單位大小或最佳循序寫入單位大小等。在得到這些明顯影響效能表現的參數後，便可直接運用於各種對 SSD 的最佳化中，如檔案系統的檔案位置分佈或記憶體空間分配大小等，讓 SSD 優異的存取效能達到更完整的發揮。

**關鍵字 : 快閃記憶體, 固態硬碟, 結構**

Geometry Detection for Solid-State Drives

Student : Chun-Chieh Kuo Advisor : Dr. Li-Pin Chang

Department (Institute) of Computer Science
National Chiao Tung University

Abstract

The Solid State Drive (SSD) based on NAND flash memory is fast developed recently, and being used in embedded products like notebook, digital camera, and UMPC etc., and many manufactures also choose SSD as a suitable storage for database server. However, in past 20 years, many researches were focusing on the geometry optimization of HDD, for example improving the design of OS, files system, application, and database. SSD is not just a faster HDD, so the geometry optimization for HDD can not be totally used is SSD. In the paper, we want to know the inside hardware design and firmware management for SSD. Therefore we design a series of detections for SSD geometry, and let the users get the important elements affecting SSD performance. By using these elements, we can optimize SSD design, and take fully usage of SSD outstanding data accessing ability.

**Keyword : NAND Flash Memory, SSD (Solid-State Drive), Geometry**

# 誌謝

終於抵達寫誌謝詞的這一刻，這意味著研究所生涯即將正式的畫上句點，回顧兩年來的經歷，首先誠摯的感謝指導教授張立平老師，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺，使得本論文能夠更完整而嚴謹。研究所的生涯裡，讓我學習了不少新的知識和待人接物的方法，讓我能夠一路走來都很順利。並感謝在這段期間，楊明毅、黃士庭、蘇宥全、廖秀芬同學的幫忙，使得我能順利走過這兩年。實驗室的黃莉君、黃偉杰、郭晉廷與黃義勛這些學弟妹們當然也不能忘記，你們的幫忙及搞笑我銘記在心。

最後，謹以此文獻給我摯愛的雙親。

# 目錄

# 圖目錄

# 表目錄



# 1. INTRODUCTION

NAND flash memory 為一種非揮發性記憶體，其主要的設計目的是為了儲存使用者資料。而 NAND flash memory 本身具有隨機存取速度快、價格便宜、體積小、耗電量低與耐震等各種特性，因此目前已經被廣泛的應用在各種嵌入式裝置中。隨著製程技術的進步，一個 NAND flash memory chip 的容量越來越大，以 NAND flash memory 為基礎的 Solid State Drive (SSD)正在快速的發展，並且已經實際運用於如筆記型電腦、數位攝影機、UMPC 等嵌入式產品中，更有多家業者設計出以 SSD 為主要儲存裝置的伺服器解決方案，SSD 已經逐漸成為取代傳統 Hard Disk Drive (HDD)的一項新選擇。

由於 SSD 通常使用了與 HDD 一樣的傳輸介面，如 SATA、PATA 等，因此大部分的消費者在不了解 SSD 特性的情況下，都將 SSD 視為速度較快的新一代 HDD。而在過去的二十年中，許多研究致力於 HDD，因此在整個儲存系統的發展都針對 HDD 進行最佳化，如檔案系統會將 metadata 或常被存取的資料放在磁盤中間的位置盡量減少讀寫頭的移動距離。或是將小檔案夾帶在其目錄檔案中，以便在讀取目錄檔案的同時一起對小檔案進行存取，來減少過多的機械動作延遲。甚至是許多的 I/O 排班與空間分配演算法，都是針對減少 HDD 的 seek time 與 rotation time 所設計的。

但是這些最佳化的行為並不適用於與 HDD 特性截然不同的 SSD 中，甚至可能會帶來反效果。如在常見的檔案系統中的磁碟排程，都會考慮到 HDD 具有磁盤旋轉的特性，因此會依據其資料在磁盤中的擺放位置進行最佳化，重新排列 write request 的處理順序，期望能配合磁盤或讀寫頭的移動方向來減少存取資料時所需要的機械延遲。但由於 SSD 本身並沒有任何的機械動作，甚至具有重複進行寫入前要先對該區塊進行抹除動作的特性，因此這樣的最佳化對於 SSD 來說不見得是有意義的，甚至可能導致原本可以集中處理的隨機 write requests 被以非常分散的順序被處理，成為對 SSD 不利的寫入樣式，造成 SSD 內部觸發不必要的管理活動，甚至帶來負面效果。因此，對於 SSD 的內部 geometry 以及效能表現進行完整的分析與解剖是非常重要的。

而在這篇論文中，我們首先將分析 SSD 內部硬體架構以及韌體管理演算法的行為，並針對重要的 SSD geometry 進行討論與分析，以了解整個 SSD 的實際運作方式。首先會討論 SSD 內部進行抹除運算的基本單位，這也

是 SSD 對其內部空間進行管理的重要基本單位，並且直接決定了 SSD 的最佳循序寫入樣式與效能，以及其在應付隨機寫入樣式時的處理能力。接著探討 SSD 進行存取資料的基本單位，基本存取單位的大小會直接的影響 SSD 在進行資料寫入時的 write amplification，而過大的基本存取單位容易造成小量寫入資料時效能非常低落，因此了解基本存取單位的大小將可利用於非常多的儲存系統最佳化中。另外 SSD 通常會對其內部記憶體空間進行分段管理，而分段管理的單位大小也會影響其內部管理資料的存取行為，間接的影響對於隨機存取時的效能表現。最後我們也對 SSD 內部的 read cache 以及 write buffer 進行討論，而 cache buffer 的大小也直接決定 SSD 所能應付的存取樣式的 locality 大小與效能。而我們也實際測試這些 SSD geometry 在面臨不同的存取樣式時對於效能表現的影響，直接觀察並分析出 SSD 特有的優點與缺點。

最後，我們更進一步的設計出一系列由產品外部進行測試 SSD geometry 參數的方法，在取得這些明顯影響效能表現的 geometry 參數後，便可直接提供給 application 的設計者，利用在 real-time 或是 database 的設計中，以更適合的存取方式進行存取，避免觸發不必要的管理動作造成延遲。也可以將 geometry 資訊提供給 file system，並且利用在分配檔案空間或磁碟排班的演算法中，以得到更快的存取速度。或將 geometry 資訊運用在作業系統內部的 disk cache 中，針對 data replacement 的機制進行最佳化，讓 SSD 能夠完全發揮其優異的存取效能。

本論文中接下來的部分將以下列架構呈現：第二章描述動機與 SSD 的相關背景知識，第三章及第四章則深入討論 SSD geometry 及其對效能表現的影響，第五章將描述取得 SSD geometry 參數的測試方法並將實際對產品的測試結果一併呈現，而最後一章將會對整篇論文進行總結。

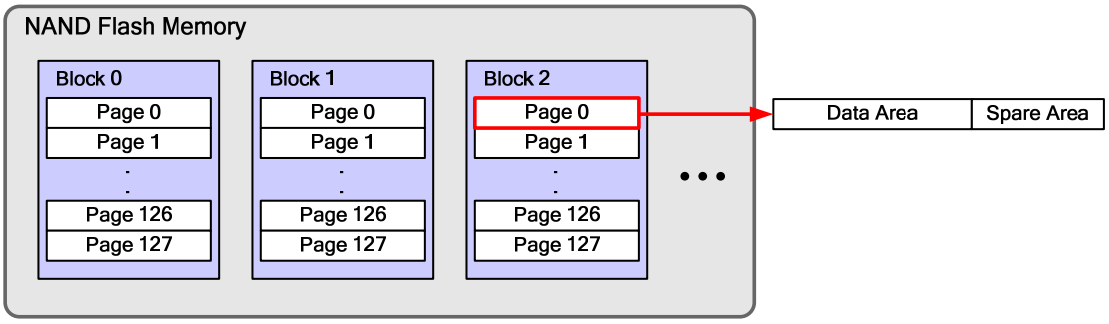
# 2. BACKGROUND

## 2.1 NAND Flash Memory

NAND flash memory 為一種非揮發性記憶體，其主要的設計是為了儲存使用者資料。由於 NAND flash memory 便宜、體積小、耗電量低與耐震等各種特性，目前已經被廣泛的應用在各種嵌入式裝置中。而隨著製程技術的進步，NAND flash memory chip 的容量也越來越大，因此目前已經被大部分的 SSDs 做為主要的儲存元件。

NAND flash memory 又可以被分類為二種類型，分別為 Single-Level Cell (SLC)與 Multi-Level Cell (MLC)，SLC 的一個記憶體單元只能儲存一位元的資料，並擁有寫入速度快與產品使用壽命長的優勢。而 MLC 的一個記憶體單元可以同時儲存二位元以上的資料，因此有便宜與容量大的優勢，但是卻有寫入速度較慢與產品使用壽命較短的缺點。而目前高階的 SSDs 為了達到更佳的寫入效能通常使用 SLC NAND flash memory，而中、低階的 SSD 產品則以成本以及容量做為考量，大部分都採用 MLC NAND flash memory。

下圖所示，不論是 SLC 或 MLC，一個 NAND flash memory chip 是由許多相同大小的 physical blocks 所組成，每一個 physical block 又由固定數量的 physical pages 所組成，其中 physical page 的儲存空間可進一步分為用來儲存使用者資料的 data area，與用來儲存管理系統所需要的 metadata 的 spare area。而 NAND flash memory 的寫入與讀取運算的基本單位為 physical page，但 erase 運算卻是以較大的 physical block 為基本單位，因此 NAND flash memory 本身並不支援 in-place update，若一個 physical page 曾經被寫入資料，則下一次對此 page 寫入資料前必須先對整個 physical block 進行 erase operation，才能對此 physical page 寫入新的資料，一般將此特性稱為 erase-before-write 或 write-once。



圖表 1 : NAND Flash Memory 架構

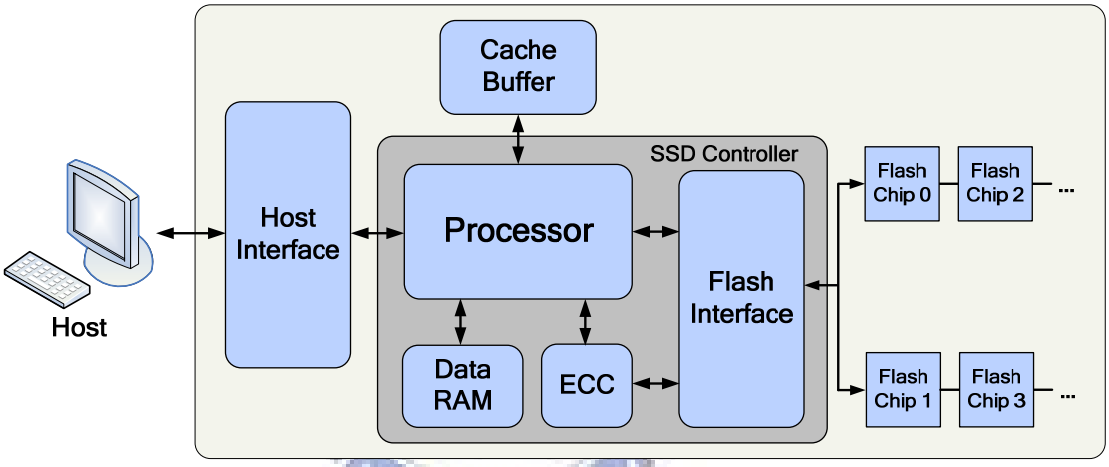
在最近常見的新型的 NAND flash memory chip 中，為了達到更高的容量與更快速的存取效能，通常採用了較大的 physical page size 與 physical block size，如 2 Kbytes 以上的 physical page 與 256 Kbytes 的 physical block。但是增加了一項新的寫入限制，physical block 內的空間必須由第一個 physical page 開始循序的使用，無法隨機的對 physical block 內的空間進行寫入。另外，NAND flash memory 製造商也提出許多能增加存取效能的新技術，其中 multi-plane program 技術將 chip 內的空間分為許多相同大小的 plane，並提供對於不同 plane 之間的 physical pages 或 physical block 可以同時的進行寫入運算或 erase 運算的能力。而在接下來的討論中，將以 SAMSUNG K9XXG08UXM 系列做為範例介紹，其詳細之規格參數如下表所示。

| SAMSUNG K9XXG08UXM | | |
|---|---|---|
| Structure | Physical Page Size (Data Area) | 2 Kbytes |
| | Physical Block Size | 128 Kbytes |
| | Spare Area Size | 64 bytes |
| Cycle Time | Page Read Time | 20 µs |
| | Page Program Time | 200 µs |
| | Block Erase Time | 1.5 ms |
| | Data Transfer Time (per byte) | 25 ns |

表格 1 : SAMSUNG NAND Flash Chip

## 2.2 SSD Architecture

在正式探討各種影響 SSD 效能的 geometries 前，我們將先描述基本的 SSD 系統架構以及其主要元件的功能。下圖以簡單的區塊圖表示整個 SSD 的系統架構，首先不論是由 SSD 輸入或輸出的資料都會藉由 Host Interface 來與 Host 端溝通，而 Host Interface 的類型可能為 PATA、SATA、SCSI 或 USB 等各種不同傳輸介面。SSD 內部會採用一顆 SSD Controller 來控制 SSD 內部元件的運作，處理所有與 Host Interface 之間傳輸的資料。其中 controller 內部採用一顆微處理器來運作韌體:Flash Translation Layer 作為整個 SSD 的控制中樞，並且使用一小塊隨機存取記憶體做為 Data RAM 來存放要被執行的程式碼，以及韌體執行時所需要的各種資料結構。




圖表 2 : SSD 內部元件

而 SSD controller 內部也包含 error correcting code (ECC)元件來更正 NAND flash memory 內部所發生的資料儲存錯誤。如同磁碟裝置一般，大部分的 SSD 也會採用一塊或一部分的隨機存取記憶體做為 Cache Buffer，並藉由將讀取或寫入的資料暫時儲存於其中，來改善讀取或寫入時的效能。在磁碟裝置中，Cache Buffer 主要目的為隱藏磁碟旋轉所需的 seek time，而在 SSD 中，Cache Buffer 可以提升其隨機寫入的能力，並隱藏 block erase time 以及 page program time 所需要的負擔。而在 SSD controller 與 NAND flash memory chip 的連結方式中，更可進一步區分為類似並聯的 multi-channel 以及類似串聯的 share-channel 二種型式，不同的連結方式也決定了 flash chip 的控制方式以及所能提供的存取頻寬。

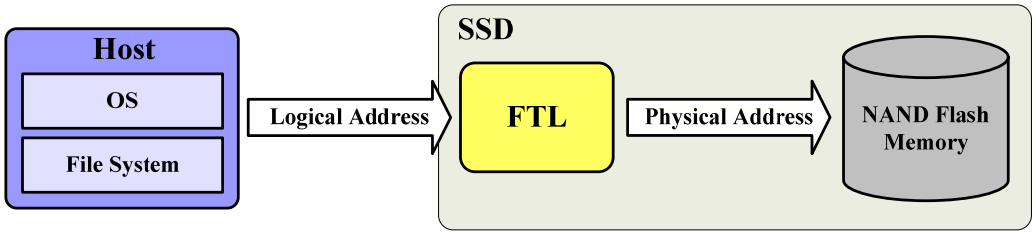
其中 SSD controller 所能提供與 NAND flash memory chip 連結的 channel 數量幾乎決定了整個 SSD 在進行寫入與讀取資料時所能達到的速度極

限。而 flash memory chips 與 controller 之間的連結方式，也與韌體共同決定了進行 read、write 與 erase 運算時的基本單位。另外，Cache Buffer 以及 Data RAM 的尺寸也間接的影響寫入與讀取資料時的效能表現與韌體的運作能力。因此，上述這些 SSD geometry 在由硬體架構所決定後，對整個 SSD 儲存裝置的運作以及效能表現有著莫大的影響。

## 2.3 Flash Translation Layer

由於 NAND flash memory 與磁碟截然不同的物理特性：
erase-before-write，且其 erase 運算必須一次 erase 整個 block 而 write 運算卻以較小的 page 為基本單位。因此無法直接採用與磁碟一樣的 in-place update，即被更新的資料不能於原來的記憶體位置上直接寫入，所以一般常用於磁碟上的區塊裝置檔案系統無法直接使用在 NAND flash memory 上，如常見的 NTFS 檔案系統與 FAT32 檔案系統等。因此以 NAND flash memory 為基礎的儲存裝置會採用 out-of-place update，並透過 flash translation layer (FTL)來將 NAND flash memory 儲存裝置模擬為與硬碟相容的區塊裝置，讓使用者可以在上面運行常見的磁碟檔案系統。其中，FTL 最主要的二項功能分別為 address translation 與 garbage collection。

由於 out-place update 的方式下，隨著檔案不斷的被更新，資料在 NAND flash memory 中的儲存位置會不斷的改變，所以 FTL 採用固定的 logical address 來定址，並採用 address translation 機制進行 logical address 與 physical address 之間的轉換，如下圖所示，FTL 會將 host 端所傳來的 logical address 進行 address translation 以得到真正對應於 NAND flash memory 中的 physical address，接下來才能對 SSD 底層的 NAND flash memory 實際進行讀取或寫入資料的動作。而根據 address translation 的管理單位，可分為 page-level mapping 與 block-level mapping 二種，而以 page-level mapping 的方式中，FTL 以最小的存取單位進行 address translation，雖然擁有非常彈性的空間管理方式，但是卻需要非常大量的 RAM 來存放 address translation 時所用的 mapping table。由於成本與耗電量考量，page-level mapping 較少被應用於實際產品中。

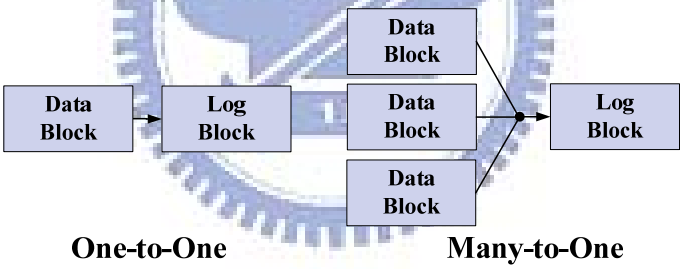



圖表 3 : FTL

在 block-level mapping 的機制中，FTL 以 erase 運算的基本單位做為其

address translation 的基本單位。因此 logical address 會被進一步切割為 logical block number 以及 logical page number，並在 mapping table 中以 block 為基本單位進行對應；一個 logical block 可以對應到 flash memory 中的一個 physical block，我們稱此 physical block 為 data block。但與 page-level mapping 彈性的管理方式不同，block-level mapping 中一個 data block 內部的 pages 必須依照 logical address 循序的被儲存。而由於 block-level mapping 的管理單位較大，可以明顯降低 mapping table 的大小，並解決 RAM 空間需求量過大的問題，因此目前被廣泛使用在各種 NAND flash memory 儲存系統的 FTL 中。在接下來對 SSD 的測試中也將以 block-level mapping 的產品為主。

另外 block-level mapping 的 FTL 根據其 data block 與 log block 的對應關係，大致上可分為 one-to-one mapping 與 many-to-one mapping 二種方式。如下圖所示，在 one-to-one mapping 的方式中，一個 log block 只能儲存一個 data block 的資料，也就是說每一個被更新的 data block 都會有自己專屬的 log block 用來暫存更新資料。而 many-to-one mapping 的方式中，則是全部的 data block 對應到同一個 log block，因此對任何 data block 寫入的資料都將先被暫存於共用的 log block 中。

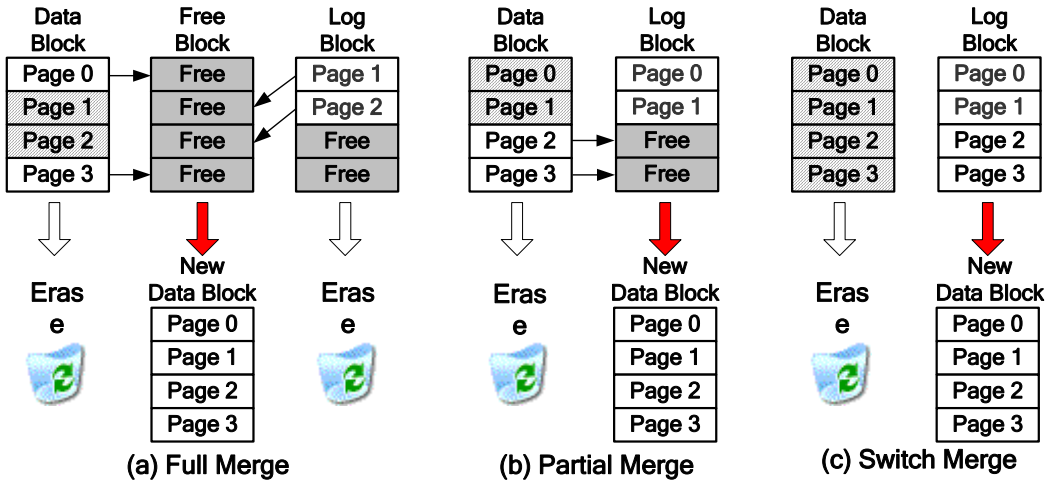



圖表 4 Mapping Scheme

由於以 out-of-place update 的方式寫入資料，許多已經無效的資料會散布於各個 data block 與 log block 內，因此便需要 garbage collection 來回收這些被佔用的可用空間。而 garbage collection 會挑選適當的 block 來進行 erase，以回收此 block 內部被無效資料佔據的空間，而當要被 erase 的 block 內部還有有效資料時，便需要 merge operation 來將這些有效資料在 block 被進行 erase 之前先搬移到別的 block 中，以免在 erase 之後有效資料遺失。

而在 block-level mapping 的 FTL 中，通常 FTL 會選擇一個 log block 與其對應的 data block 進行 merge operation，並將其內部分散的有效資料還原為以邏輯位置循序儲存並皆為有效資料的新 data block。根據進行 merge operation 之情況，一般將其回收方式分為 full merge、partial merge 與

switch merge 三類，其中 full merge 如下圖(a)所示，FTL 會將 log block 與 data block 內部的有效資料以邏輯順序複製到一個 free block 中，接著便對 data block 與 log block 進行 erase，在所有的有效資料都被複製之後 free block 即成為新的 data block，而 full merge 之成本為讀取並寫入整個 logical block 內所包含的 pages，並進行二次的 erase 運算，才能回收到一個 block 的空間。



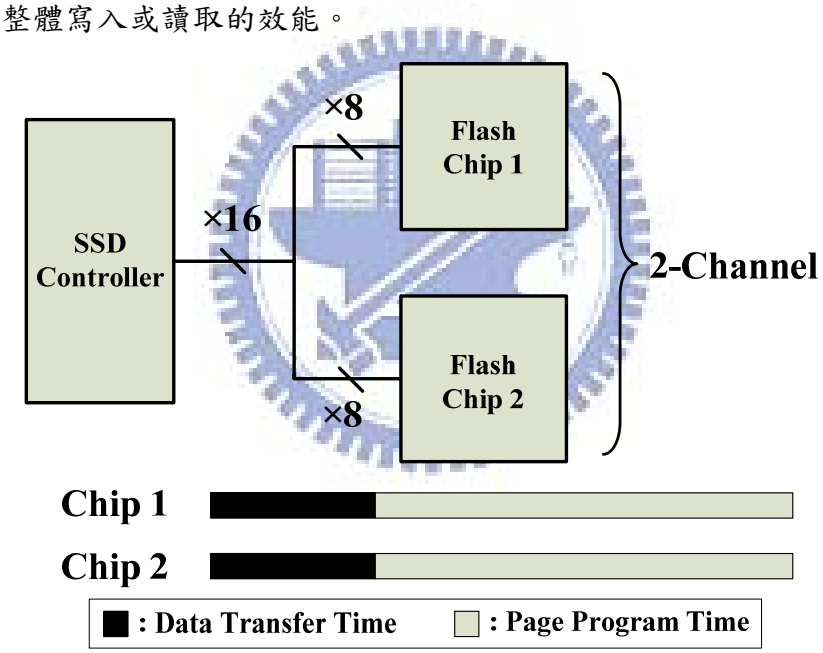

圖表 5 : Merge Operation

而 partial merge 則如圖(b)所示，若 data block 內的資料是由第一個 page 開始被更新時，此時只需要簡單的先將 data block 內剩餘的 pages 循序的複製到 log block 內，log block 便成為新的 data block，接著對 data block 進行 erase 運算。而 partial merge 之成本為複製 data block 內存有有效資料的 pages，並只要進行一次 erase 運算就可以回收一個 block 的空間。如圖(c)所示，switch merge 為最理想的 merge operation，當 data block 內的資料完全以循序方式更新時，此時 data block 內部完全為無效資料因此可以直接進行 erase 運算，而 log block 也可以直接作為新的 data block，因此不用進行任何資料複製即可回收到一個 block 的空間。

對於 SSD 等 NAND flash memory 儲存裝置而言，其造成讀取或寫入效能明顯降低的主要原因，即為 merge operation 時進行有效資料複製的讀取與寫入成本，以及耗時的 erase 運算。

## 2.4 Parallel Data Access

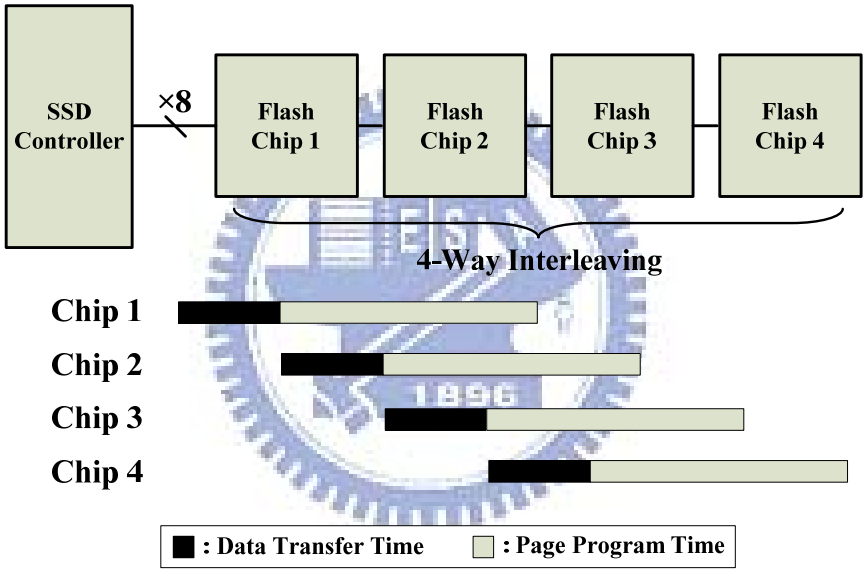
隨著 SSD 容量不斷的成長，現今的 SSD controller 都連結非常多顆的 NAND flash memory chips，而依照其與 controller 的連結方式可分為 multi-channel 與 shared-channel 二種。在 multi-channel 架構中每一個 flash chip 都有各自用來傳輸資料、指令、記憶體位置的 channel，藉由同時讓多顆 flash memory chip 平行運作而達到更好的效能。如下圖所示，在 2-channel 的架構中 SSD controller 分別以二條 channel 連結至二顆 NAND flash memory chips。由於這二顆 flash memory chips 都各自擁有用來傳輸資料與指令的 channel，因此可以完全獨立地運作。下圖中的時脈圖所描述，SSD 在 2-channel 架構下進行 write operation 時，SSD controller 幾乎可以同時控制二顆 flash memory chips 進行任何動作，如同時下達寫入指令與傳輸資料，並進行寫入運算。因此採用 multi-channel 的架構，可以提高整體寫入或讀取的效能。




圖表 6 : Multi-channel

由於受到 NAND flash memory 本身進行讀取或寫入動作所需要的忙碌時間影響，單一 NAND flash memory chip 並無法充分的利用 channel 所能提供的頻寬。如 K9XXG08UXM 系列中，一個 flash memory chip 於寫入時最高僅能使用 channel 約 20%的頻寬，這意味著此時 channel 有 80%的時間是閒置的。因此在 shared-channel 的連結方式中，為了達到更高的寫入與讀取效能，通常會藉由盡量重疊多顆 flash memory chips 進行寫入或是讀取運算的時間，並且讓 flash memory chips 輪流的使用單一 channel 進行資料傳輸，來增加 channel 的使用率，而我們稱此運作方式為 interleaving。

如下圖所示，在 shared-channel 的架構中，四顆 NAND flash memory chips 都連結至同一條 channel，並共用此 channel 來與 SSD controller 進行資料傳輸和接收指令。當對此四顆 flash memory chips 進行 interleaving 寫入時之運作情形如下時脈圖所描述，首先 SSD controller 透過 channel 對 chip 1 下達寫入指令並傳輸資料，待一個 physical page 的資料傳輸完成後 chip 1 便開始進行寫入運算。而在 chip 1 開始進行寫入運算的同時，channel 已經成為閒置狀態，因此 SSD controller 便立即對 chip 2 下達寫入指令與傳輸資料，並接著進行寫入運算。而以此方式驅動四個 flash memory chips 輪流地使用 channel，除了增加 channel 頻寬的利用度之外，同時也可以重疊 flash memory chips 之間傳輸資料與進行運算的時間，以在 shared-channel 的架構下達到更高的寫入效能。

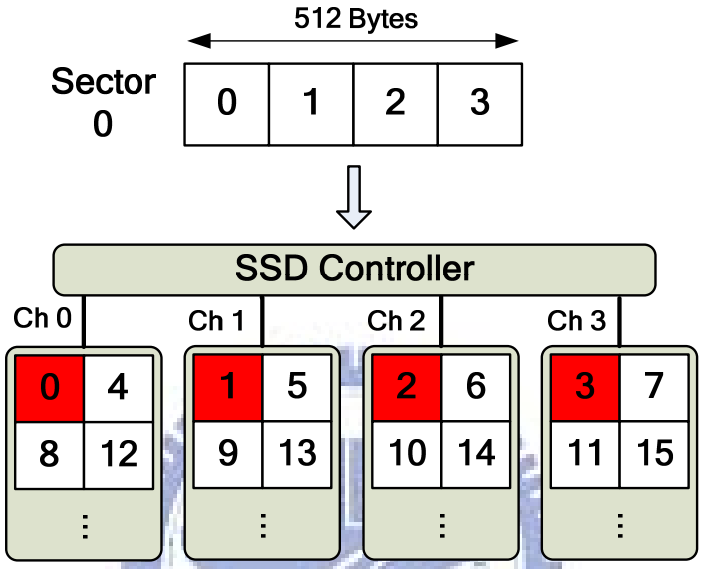



圖表 7 : Interleaving

而在 SSD 儲存裝置內部有限的 channel 數量中，採用 interleaving 運作的方式進行 read 運算和 write 運算，除了可以增加 channel 頻寬的使用率，提高寫入或讀取時的效能。採用 shared-channel 的方式連結 flash memory chips，更能在不用改變 SSD controller 的設計的情況下，只需要對硬體線路進行微小的修改，就能簡單地增加 SSD controller 所能連結的 flash memory chip 數量，以加大 SSD 儲存裝置的容量。

而許多 SSD controller 在 multi-channel 的架構中，會以類似磁碟陣列中 RAID-0 的概念[3] [7] [8] [9] ，將一個 sector 的資料平均地分散到 SSD 中所有的 channels，讓每一個 read request 或 write request 都橫跨全部的 channel，強迫動用全部的 channel 來存取資料，以增加 SSD controller 對

於內部 multi-channel 架構同時運作的平行度。並藉由將資料傳輸時間平均地分攤給全部的 channels 來降低寫入或是讀取運算的成本，進而達到增加 SSD 儲存裝置效能的目的。而平均分散一個 sector 的資料，同時也使不同 channel 之間的 flash memory chips 達到更均勻的負載平衡，減輕某些 flash memory chips 被過度磨損的問題。另外，由於此資料分佈方式，對於處理任何的 read request 或 write request 時全部的 channels 皆會同時運作，因此可以簡化 SSD controller 對於 multi-channel 進行控制的複雜度。

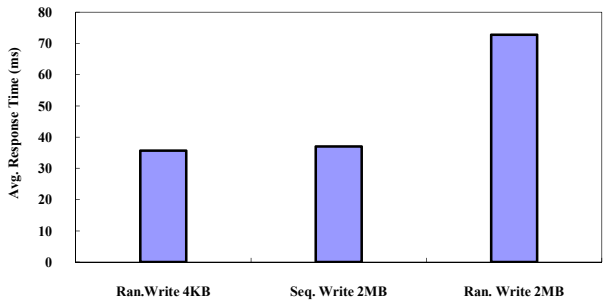



圖表 8 : Sector Stripping

## 2.5 Motivation

目前常見的 SSD 產品都採用與 HDD 相同的傳輸介面，但 SSD 並不單純只是存取速度較快的 HDD，他們具有截然不同的物理特性，如 NAND flash memory 無任何的機械動作等，因此消費者普遍認為 SSD 具有非常優秀的隨機存取能力。但是在實際使用 SSD 中，有時候對 SSD 寫入小量資料的文件檔案時，其所需要的完成時間卻遠比寫入檔案尺寸大好幾倍的多媒體檔案更長，而這樣的現象對於不需要機械手臂動作與磁盤旋轉時間的 SSD 是非常令人訝異的。

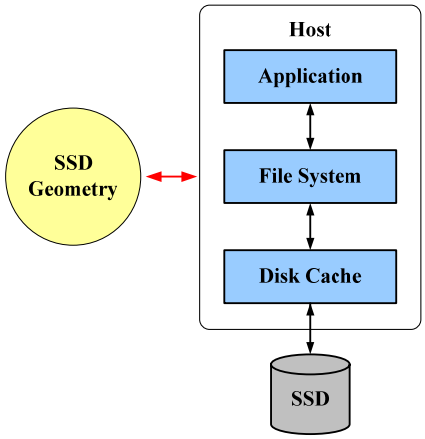
如下圖的簡單實驗所示，實際對 SSD 進行隨機寫入 4 KB 資料所需要的完成時間幾乎跟循序寫入 2 MB 資料是一樣的；事實上這就是由於 one-to-one mapping 的 block-level FTL 中，若不斷的對於不同的 data block 進行隨機寫入，很容易造成 log block 被快速消耗，進一步成為 log block 不足而被 data block 大量競爭的情形，因此造成 SSD 內部不斷的觸發 garbage collection 來回收被使用的 log block，而導致效能異常低落。更讓我們了解，儘管 SSD 本身具有非常優秀的存取能力，但在不了解 SSD 內部的 geometry 與運作行為時，以不適當的方式進行存取還是會造成效能非常低落。




圖表 9 : write response time of SSD

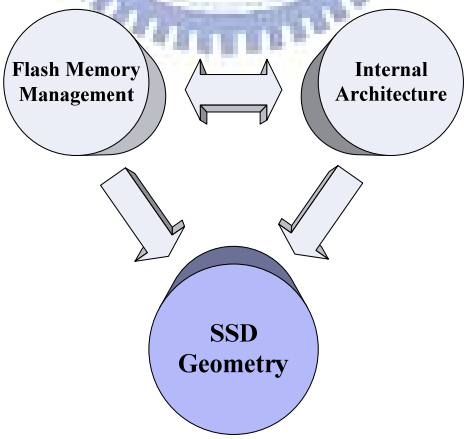
若我們取得 SSD 內部的 geometry，就能利用在整個儲存系統架構的設計或最佳化中。如下圖所示，將 geometry 資訊提供給 application 的設計者，就能在 real-time 或是 database 的應用中，以更適合的存取方式進行存取，並盡量避免 garbage collection 等非預期的寫入延遲。也可以將 geometry 資訊提供給 file system，並且利用在分配檔案空間的演算法中，將檔案對齊 SSD 基本存取或抹除單位，以得到更快的存取速度或是減少 garbage collection 時的成本。甚至可以將 geometry 資訊運用在作業系統內部的 disk cache 中，針對 data replacement 的機制進行最佳化，來降低 garbage

collection 被觸發的頻率。




圖表 10 Storage System

因此在這篇論文中我們想藉由了解 SSD 內部的硬體架構以及其韌體管理演算法，來對整個 SSD 的效能表現進行完整的分析與解剖。如下圖，韌體管理演算法與 SSD 內部硬體架構會互相影響彼此的設計，綜合以上二者的影響則直接決定了整個 SSD 儲存裝置的 geometry。而除了研究這些 SSD geometry 對於各種不同的存取樣式的效能影響外，我們更希望能針對這些 SSD geometry 設計出一系列的測試方法，讓使用者能在產品外部取得各種 SSD geometry 的參數，如基本存取單位大小或最佳循序寫入單位大小等。若我們能得到這些明顯影響效能表現的參數，便可直接運用於各種對 SSD 的最佳化中，讓 SSD 優異的存取效能達到更完整的發揮。




圖表 11 : SSD Geometry

## 2.6 Related Work

到目前為止，已經有非常多的人使用既有的 benchmark 工具對 SSD 進行效能測試，如 HD Tune、HD Tach、h2benchw、ATTO Disk Benchmark 與 IOMeter 等。但是這些為 HDD 所設計的測試工具，主要只採用循序寫入與隨機寫入等少數幾種典型的存取型為，針對儲存裝置的平均存取頻寬以及平均 I/O 處理能力進行測試，而這些存取樣式與測試標的並不能真正呈現出 SSD 在效能上的瓶頸，也無法讓消費者了解不同的 SSD 產品之間於實際使用時的差異。而目前市面上並沒有一款專門針對 SSD 所量身訂做的 benchmark 工具。

在少數的論文中，已經有一些研究者嘗試著對 SSD 或其他 NAND flash memory 儲存裝置進行更進一步的效能測試，希望能了解更多 SSD 等儲存裝置與 HDD 不同的效能表現。如 P.C. Huang 等人針對 NAND flash memory 儲存裝置的特性及行為進行分析，並提出了對此類產品進行效能評比時的測試方式與應該注意的測試標的，但是由於測試樣式僅僅只有循序以及隨機存取，因此並無法突顯 SSD 與 HDD 在效能表現上的差異 [1]。而在 Luc 等人所提出的各種存取樣式的測試中，對 NAND flash memory 儲存裝置進行了非常詳盡的測試，其中某些測試結果也突顯了 SSD 特有的效能表現，如存取單位以及資料空間區域性的影響，但是其討論僅限於測試樣式的分析，並沒有對其測試結果的原因進行更深入的探討 [2]。

而另外在一些討論 SSD 內部硬體架構的論文中 [3] [4]，大部分都將重心放在 multi-channel 與 shared-channel 等硬體架構對於存取頻寬的影響，非常少對完整的 SSD geometry，甚至 SSD 實際的運作情形進行分析，如結合 garbage collection、address translation 機制等，更沒有一篇論文探討內部硬體架構對於各種存取樣式的效能影響。而此篇論文同時針對 SSD 內部硬體架構以及 FTL 之管理方法進行徹底的分析，並詳細的說明各項 SSD geometry 在不同的存取樣式下，對效能表現所造成的影響。

最後在對儲存裝置的 geometry 進行截取的論文中，diskbench 與 traxtents 中皆以 HDD 為測試對象[5] [6]，並在使用者層面藉由不同樣式的外部存取測試，來取得各項 HDD geometry 之詳細參數；如 traxtents 中可偵測出 HDD track 的 boundary，若在檔案系統分配檔案位置時能盡量放置在同一個 track 時，在存取時就可以減少讀寫頭的移動，降低 seek time。但是由於 SSD 與 HDD 的物理特性全然不同，因此所提出的測試項目與測試方

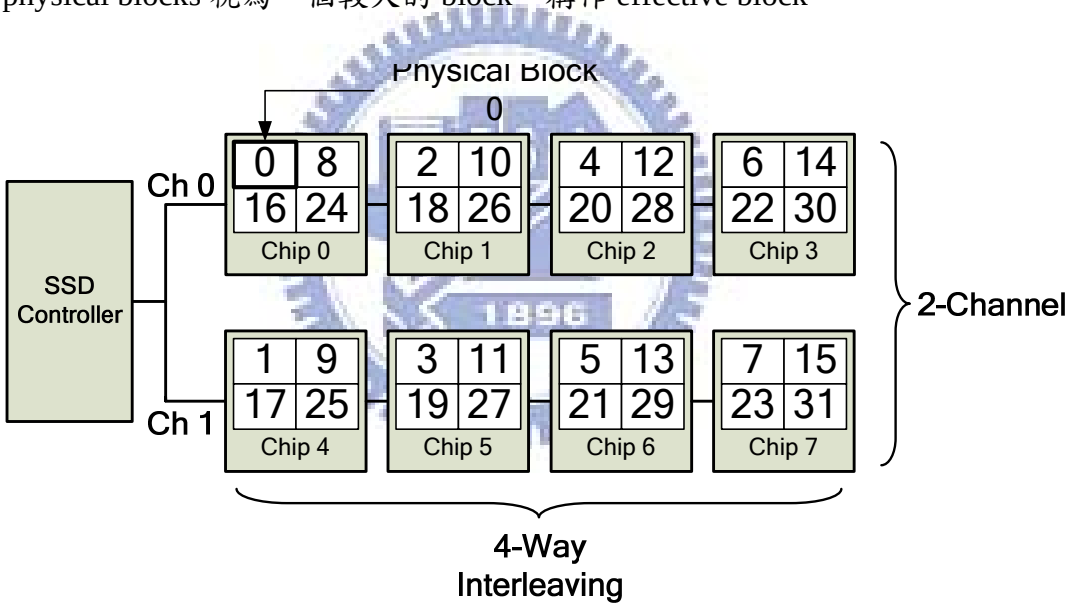
法並無法直接套用於 SSD geometry 的測試，其中 diskbench 對於 write cache 的測試方式更已不適用於許多大容量且寫回速度較快的 write cache 中。

# 3. SSD Geometry

## 3.1 Effective Block

### 3.1.1 Definition of Effective Block

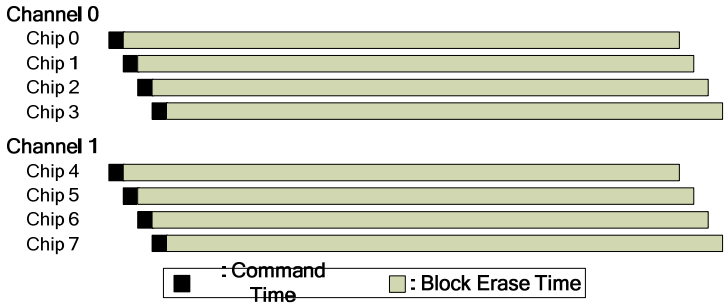
Effective block 為 SSD controller 在進行 erase 運算時之最基本單位。由於 NAND flash memory 對 physical block 進行 erase 運算是非常耗時的，因此設計者通常會利用 multi-channel、interleaving 或 multi-plane erase 等平行化運作的技術，盡可能的在一次的 merge operation 中，同時對多個 physical blocks 平行地進行 erase 運算，以期望在一次的 block erase time 中回收到更多的有效空間。我們將這些被群組在一起並同時進行 erase 運算的 physical blocks 視為一個較大的 block，稱作 effective block。




圖表 12 : Effective Block

對於大部分的 SSD controllers 來說，effective block size 以及其所包含的 physical blocks 的分布是取決於 SSD 內部所採用的硬體架構設計。如上圖所示，在採用 2-channel 與 4-way interleaving 架構的 SSD 中，若一個 effective block 由 8 個分散於不同的 flash chip 中的 physical blocks 所組成，如 effective block 0 由 physical block 0 至 physical block 7 組成。上圖中表示對 effective block 0 進行 erase 運算時的實際運作情況，可以看出在 2-channel 同時運作並且採用 interleaving block erase 技術時，由於 flash

memory chip 進行 erase 運算所需要的是非常長的，因此 effective block 0 中所包含的全部 physical blocks 幾乎可以完全平行的進行 erase 運算。所以大約只需要一個 block erase time，便可以同時對 8 個 physical blocks 進行 erase 運算，以在最短的時間內回收到最多的空間。

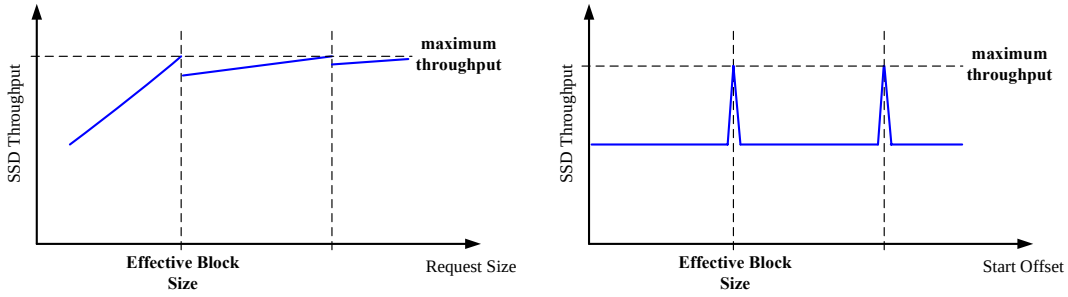



圖表 13 : Erase Timing Diagram

由於 effective block size 是取決於 SSD 內部的硬體架構，因此通常 effective block size 之最大值為 physical block size 乘上 the degree of multi-channel 再乘上 the degree of shared-channel。

## 3.1.2 Impact of Effective Block

在 block level mapping FTL 中，由於同一個 data block 內的資料被更新時，都會先記錄在同一個 log block 內。因此在循序更新資料時，若 write request 的起始位置正好對齊 effective block 的起始邊界，並循序地更新 data block 內的資料。那麼在進行 garbage collection 時，就可以採用較快速的 partial merge 或是最理想的 switch merge。避免因為 full merge 時所產生的巨大 overhead 而造成寫入效能低落，讓 SSD 在 sequential write 情況下可以達到更佳的效能表現。如下圖 A 所示，在 write request 對齊 effective block 的起時邊界時，當更新的資料量正巧為 effective block size 的倍數，由於可以直接進行 switch merge 所以不必進行任何的有效資料搬移，達到最佳的寫入效能。而 write request size 不為 effective block size 倍數時，則因為要進行額外的有效資料搬移而寫入效能較差。另外如圖 B 所示，當寫入資料量恰巧為 effective block size 時，若 request 之起始位置沒對齊 effective block 的邊界則無法進行 switch 而速度較慢。

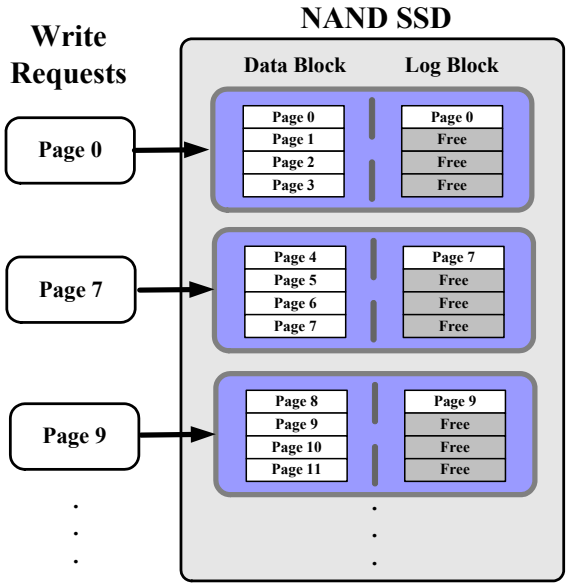



Align effective block boundary　　Align effective block size

圖表 14 : Performance of sequential write

在 one-to-one mapping scheme 的 FTL 中，一個 log block 只能分配給一個 data block 使用，也就是說當一個 data block 內的資料被更新時，FTL 會去檢查此 data block 現在是否有自己的 log block。若此 data block 已經有被分配 log block，則被更新的資料將直接寫入此 log block 中。否則 data block 還沒被分配任何的 log block，FTL 便會分配一個新的 log block 提供此 data block 作為資料暫存空間，此時如果 SSD 內已經沒有可用的 log blocks，flash translation layer 便會進行 garbage collection 來回收已經被使用的 effective blocks。

因此當 SSD 在進行隨機寫入時，若隨機寫入的資料範圍侷限在一個 effective block size 內，那麼僅需要一個 log block 就可以吸收這些寫入資料，並在 log block 的空間完全被使用完時才進行回收，此時能充分利用一個 effective block 的儲存空間。而對於 flash translation layer 最糟糕的寫入樣式，如下圖的 write requests 分佈圖所示，write requests 隨機的對不同的 data blocks 內的資料進行更新時，flash translation layer 必須不斷的為被更新的 data block 分配 log block，而在全部的 log blocks 被消耗殆盡之後，每一個對不同的 data block 進行更新的 write request 都會由於沒有可用的 log block 而不斷的觸發 garbage collection，導致許多 log blocks 在還有很多閒置空間時就被迫回收，造成大範圍隨機寫入的效能非常低落。




圖表 15 : Random write to SSD

大部分的 FTL 設計中，當 log block 內的空間使用完時，flash translation layer 便會觸發 garbage collection，以清理出可用的 effective blocks。因此 effective block size 成為影響 flash translation layer 進行 garbage collection 頻率的重要因素，當一個 data block 被更新的資料量達到 effective block size 時，意味著此時 log block 已經無可用空間，便會觸發一次 garbage collection，將此 log block 與對應的 data block 進行 merge operation。所以當 log block size 越大時，就能盡量延後 garbage collection 的發生。

實際上較大的 effective block size，雖然可以增加 log block 對於所能吸收的資料量，也可以增加 log block 所能吸收的 random write 的空間區域性，但是由於 SSD 中可用的記憶體空間是有限的，因此若採用的 effective block size 越大，則總共的 log blocks 數量則較少。而在某些情況下，當可用的 log block 數量太少時，會造成 data block 之間對於 log block 分配的競爭，導致許多 log block 內的空間並未被完全利用完，就因為 log block 數量不足而被迫提早回收，造成 garbage collection 發生頻率反而提高，導致整體寫入效能降低。

因此在檔案系統擺放大型檔案時，若能對齊 effective block 的大小或起始邊界就能增加被更新時使用 switch merge 的機率，最佳化循序寫入的速度，避免有效資料搬移的成本，否則就必須使用 full merge 或 partial merge 增加 garbage collection 的成本。而在隨機寫入方面，若能在檔案分佈時就盡量的將容易被更新的資料或是 metadata 盡量的集中在同一個 effective block 中時，就可以盡量增加 log block 的利用度或是降低 merge operation 的複雜度，最佳化隨機存取效能，否則在資料分佈過於分散的

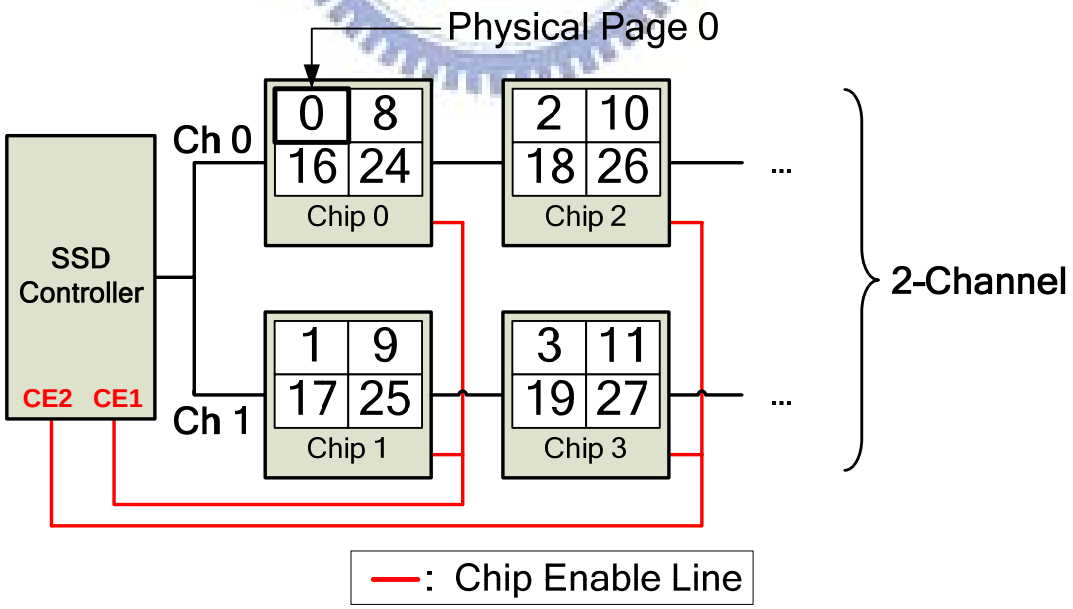
情況下，在 one-to-one mapping 中可能會造成 log block 被大量競爭而導致 garbage collection 觸發頻繁，或在 many-to-one mapping 中提高 merge operation 時的複雜度，都會造成效能大幅低落。

## 3.2 Effective Page

### 3.2.1 Definition of Effective Page

Effective page 為 SSD controller 在進行 write 運算與 read 運算時的最基本單位，effective page 是由一個或多個 physical page 所構成。而對於 SSD 來說，effective page size 除了取決於所採用的 NAND flash memory chip 的 physical page size 規格之外，更與 SSD 的內部硬體架構與 controller 的設計有直接的關係。

下圖中描述了一個典型的 SSD 架構下 effective page 的分佈情形，此時一個 sector 的資料被平均分散到每一個 channel 中，所以任何的 read request 或 write request 都會動用全部的 channels。因此每一條 chip enable line 皆平均連接至每一個 channel 中的一個 chip，如：CE 1 連結至 channel 0 的 chip 0 與 channel 1 的 chip 1、CE 2 連結至 chip 2 與 chip 3，使得 chip 0 與 chip 1 或 chip 2 與 chip 3 會同步地進行所有的運算。而每一個 effective page 都會由一組同步運作的 chips 中各一個 physical page 所組成，例如 chip 0 中的 physical page 0 與 chip 1 中的 physical page 1 共同組成 effective page 0、physical page 2 與 physical page 3 共同組成 effective page 1，而此時 effective page size 為二倍的 physical page size。




圖表 16 : Effective Page

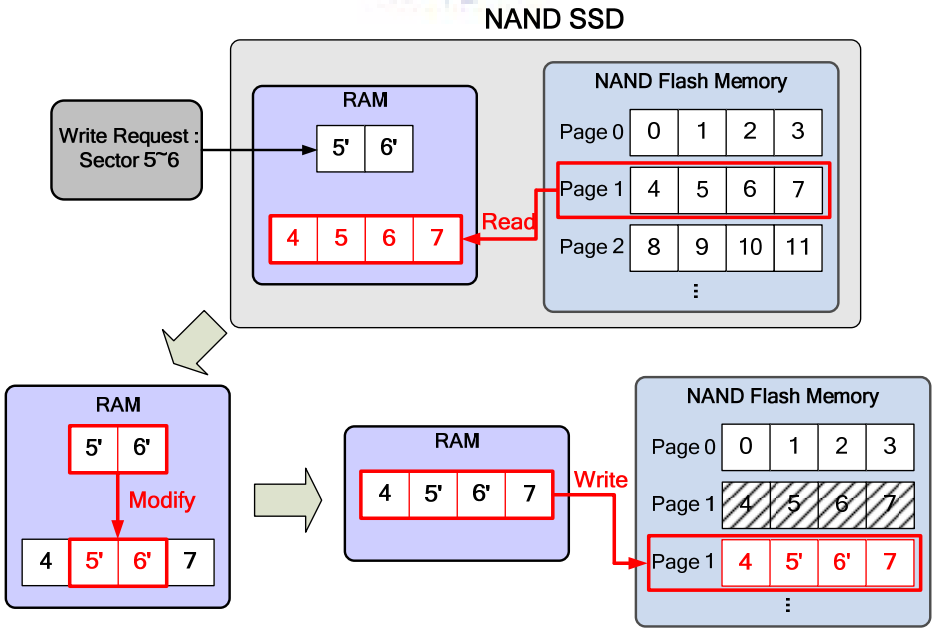
另外，若 SSD controller 有使用 flash memory chip 所提供的 multi-plane

program 技術，那 effective page size 亦會隨著 multi-plane program 可以同時寫入的 physical page 數量等倍率的放大，例如採用了 two-plane program 技術後，單一 chip 在進行一次 write 運算時之最小單位將變成二個 physical page，因此新的 effective page size 成為原來的二倍。

## 3.2.2 Read-Modify-Write Operation

對於一般的磁碟檔案系統而言，其進行 write 運算與 read 運算的基本單位為一個 disk sector，也就是 512 Bytes。但是在 SSD 中，進行 write 運算的基本單位為一個 effective page，其中一個 effective page 是由一個或多個 physical page 所組成的。而現今常見的 physical page size 都為 2 Kbytes 以上，因此一般來說一個 effective page 中都包含許多個 sectors。

由於 SSD 之 write 運算基本單位大於檔案系統之 write 運算基本單位，導致當進行 write 運算時，若要被更新的資料量小於 effective page size 時，SSD controller 必須先從 flash memory 中讀取整個 effective page 至資料暫存區，接著修改 effective page 中有被更新的部分，最後再將整個 effective page 重新寫回 flash memory 之有效空間，我們稱此過程為 read-modify-write。因此 SSD 在進行 write 運算時，當被更新的資料量小於 effective page size，SSD controller 必須對此 effective page 進行 read-modify-write 運算才能完成資料的寫入，因此會造成額外 overhead，導致 write request size 較小時，反而寫入速度較慢的異常現象。如下圖…



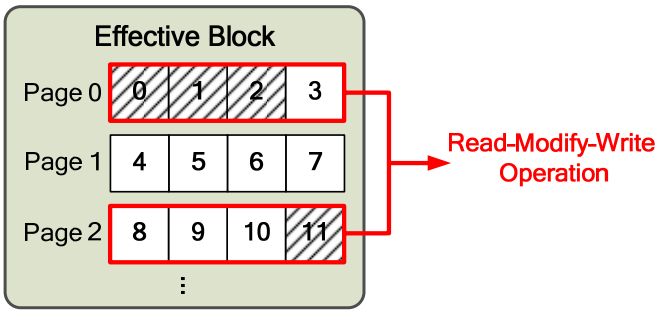

圖表 17 : Read-Modify-Write Operation

而過大的 effective page size 會造成處理 small write request 時 read-modify-write 的成本非常高，導致效能低落，但是對於 flash translation layer 來說，由於管理單位的 granularity 較大，因此所需付出的管理成本是較低的。反言之，採用較小的 effective page size 雖然在處理 small write request 時，享有較高的寫入速度，但是也由於管理單位的 granularity 較小，因此必須付出較高的管理成本。

### 3.2.3 Impact of Effective Page

在 effective page 對 SSD 效能的影響中，主要會對於小量寫入資料造成效能低落，與降低 garbage collection 時進行較快速的 switch merge 以及 partial merge 的機率。而其原因都是由於 effective page size 大於 HDD 所採用的 sector size，導致 write request 沒有對齊 effective page 的大小或起始位置。首先在小量寫入資料時，由於 write request 並不會特易對齊 effective page，因此大部分的 write requests 都必須伴隨著 read-modify-write operation 而造成效能低落。而 SSD 在處理許多連續寫入的 write requests，也會因為沒有對齊 effective page，而在 SSD 內部成為不連續的更新，導致無法使用 switch merge 與 partial merge，增加 garbage collection 時的負擔。

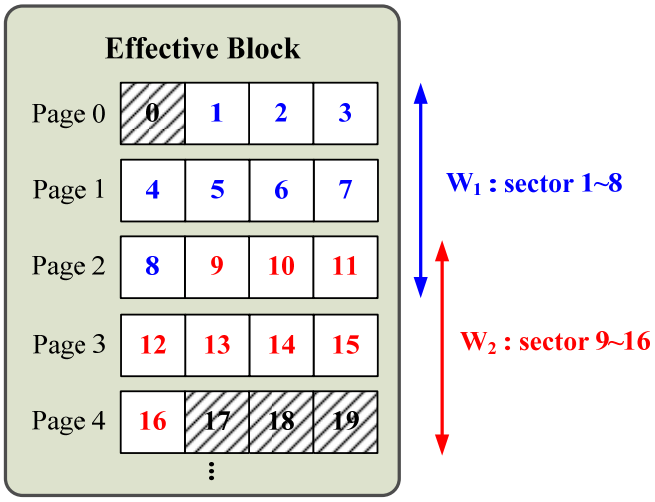
當 write request 的 size 沒有對齊 effective page size 的倍數或起始位置沒對齊 effective page 的 boundary 時，會造成 write request 所對應到的最後一個或第一個 effective page 所要寫入的資料量小於 effective page size，所以對此 effective page 必須執行 read-modify-write 運算，而造成效能低落。如下圖所示，若一個 effective page 中包含了四個 sectors，若 write request 要由 sector number：3 更新至 sector number：10，實際上 request 總共更新的資料量只有二個 effective page，但是由於此 write request 的起始位置並未對齊 effective page 中的第一個 sector，因此必須寫入 P0、P1 及 P2 三個 effective page。其中 P0 與 P2 這二個 effective page 所要被寫入的資料量都小於 effective page size，因此都須要進行 read-modify-write 運算，導致此時寫入效能非常低落。




圖表 18 : Impact of small write

而在block-level的FTL scheme中，處理循序的寫入資料時可以使用switch merge或partial merge來降低merge operation的成本；藉由降低erase運算與有效資料搬移的次數，進而達到較高的寫入效能。但事實上由於write request size或起始位置沒有對齊effective page，造成許多循序更新的資料到達底層NAND flash memory時，會因為write request的要寫入的最後一個effective page與下一個write request要寫入的第一個effective page重疊，導致資料無法循序被更新。

如下圖所示有二個連續的write requests分別為$W_1$與$W_2$，其中$W_1$寫入由sector 1至sector 8，$W_2$寫入從sector 9至sector 16。寫入資料量為循序更新sector 1至sector 16，總共16個sectors。而$W_1$實際於NAND flash memory中橫跨了effective page 0至effective page 2，$W_2$則橫跨了effective page 2至effective page 4，因此SSD controller在進行$W_1$時要寫入的最後一個effective page與$W_2$要寫入的第一個effective page皆為effective page 2，因此NAND flash memory中effective page被更新的順序成為Page 0 → Page 1 → Page 2 → Page 2 → Page 3 → Page 4，此一不循序的更新，使得FTL在進行回收時只能採用成本最高的full merge。




圖表 19 : Impact of sequential write

雖然 SSD controller 在進行 read 運算時也是以 effective page 作為基本單位，但是當 read request size 沒有對齊 effective page 時，SSD controller 只需要從 read request 所對應到的 effective pages 中，將 read request 所要讀取的資料由 NAND flash memory 中輸出即可，並不如進行 write 運算時因為須要進行 read-modify-write 運算而造成額外的 overhead。另外，也由於 read operation 的速度較 write operation 快許多，因此不論 read request 有沒有對齊 effective page size 的倍數或起始邊界，所需要的 response time 不會造成明顯的增加。

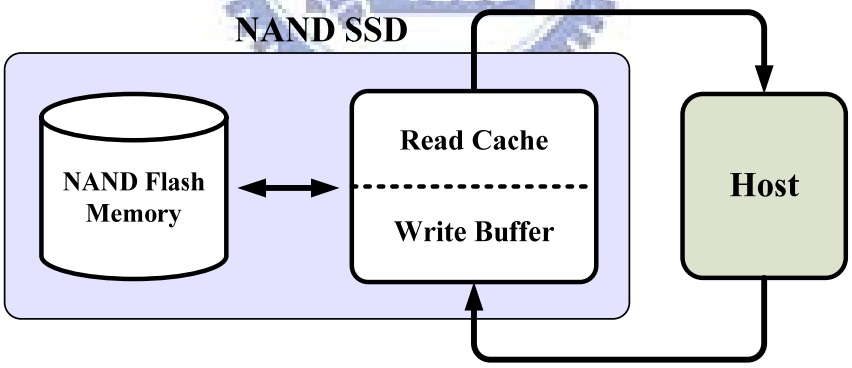
由於 SSD 為了達到更高的存取頻寬，會採用 multi-channel 數量非常多的架構，所以大部分的 SSD 都有著非常大的 effective page size，更導致了大部份的 write requests 無法對齊 effective page，使得寫入效能大幅降低。因此若檔案系統分配檔案位置時，就能對齊 effective page 的邊界，或是在發出 write request 時，能對齊 effective page 的尺寸，就可以盡量的避免 read-modify-write operation，達到更快的寫入效能，甚至是提升 FTL 進行 switch merge 的機率。否則，就必須要承受 read-modify-write 的負擔造成寫入速度大幅降低，並導致循序更新時無法使用 switch merge，更進一步的造成 SSD 效能低落。

## 3.3 Read Cache/Write Buffer

### 3.3.1 Definition of Read Cache/Write Buffer

隨著電腦運算能力快速的進步，使用者對於儲存系統的效能要求也越來越高。因此，越來越多高階的儲存裝置在產品中加入一塊固態揮發性記憶體，一般稱為 cache 或是 buffer。cache 技術主要藉由加速資料讀寫與傳輸，來增進儲存裝置的效能，而消費者常見的硬碟以及 SSD 中即以隨機存取記憶體做為 cache。

而 cache 技術對於儲存裝置而言，主要提供二項重要功能。第一，cache 可做為 computer 端與記憶體元件之間的資料緩衝區。若資料直接的傳遞於速度不同的 computer 與記憶體元件之間，經常會因為速度上的差異造成較慢的一方需要等待，導致整體處理效能低落。因此採用一塊 cache 做為資料緩衝區，可協調二者之間的速度。第二，cache 可以做為資料暫存區域，儲存要被寫入至記憶體元件或是從記憶體元件中被讀取出來的資料。因此採用讀取與寫入速度較快的隨機存取記憶體作為 cache 時，便可以縮短 read request 與 write request 的 response time，進一步提升整個電腦系統的效能表現。如下圖所示，任何在 NAND flash memory 與 Host 之間傳遞的資料都會先經過 cache buffer。




圖表 20 : Cache Buffer

Read Cache 目前已經非常普遍的存在於大部分的 disk 與 SSD 中。由於 RAM 與 NAND flash memory 的讀寫效能在先天體質上的差異，從 RAM 中讀取資料比起直接從 NAND flash memory 中讀取資料是非常快速的。因此 SSD 儲存裝置會將最近被讀取過的資料或是 pre-fetch 的資料暫存於 read cache 中，當儲存裝置接收到 read request 時，便會先檢查所要讀取的資料是否在 read cache 中。當資料已經儲存於 read cache 內時，便可以從 read cache 中直接取得，以達到快速的 response time。而 read cache 中

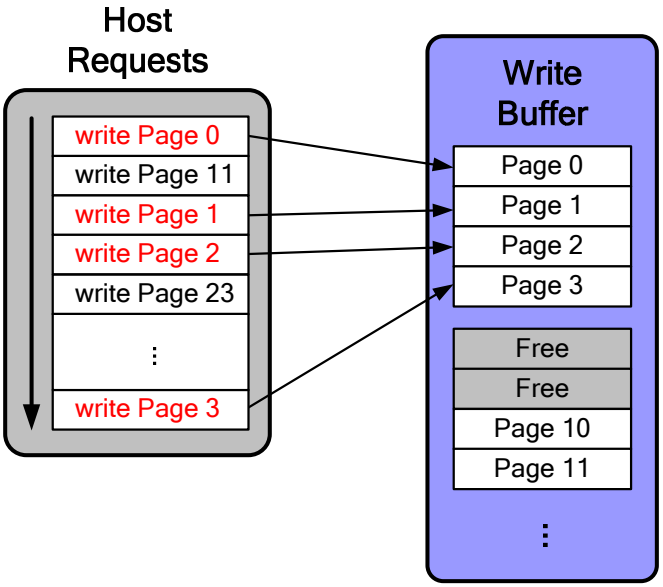
沒有所要的資料時，儲存裝置才需要從底層 NAND flash memory 中讀取資料。

與 read cache 的概念相似，write buffer 也是藉由先將要被寫入至 NAND flash memory 中的資料暫存於 write buffer 內，來改善儲存裝置的寫入效能，唯一的差別在於 write request 實際上改變了底層 NAND flash memory 的資料，而 read request 並未對底層資料造成任何改變。當 SSD 儲存裝置沒有採用 write buffer 時，對於所有的 write requests，computer 端都必須花費漫長的時間等待 SSD 完成寫入資料至 NAND flash memory 中，導致儲存裝置較慢的寫入速度拖累整個電腦系統的效能表現。

當採用 write buffer 時，SSD 儲存裝置對於所接收到的 write requests 會直接將其資料寫入速度較快的 write buffer 中，並立即的回報此 write request 已完成。然而事實上要被寫入的資料尚未被寫入底層 NAND flash memory 中，而在 write buffer 內的資料會在稍後由 SSD controller 決定適當的時機點進行寫回。藉由立即的回報寫入完成，write buffer 大幅的縮短 SSD 處理 write request 所花費的回應時間，computer 端也不必等待 NAND flash memory 漫長的寫入時間而能繼續的進行其他運算，進而提升整個電腦系統的效能表現。

對於處理記憶體位置連續的 write requests 而言，採用 write buffer 能在實際寫入 NAND flash memory 之前先在 buffer 內聚集這些較小的 write requests，待結合成為一段較大的循序資料後，再進行寫回 NAND flash memory 中。如此除了可以充分發揮 interleaving 等平行化寫入的技術，達到更快的寫入速度。更可以改善由於 write request size 小於 effective page size 時，需要進行 read-modify-write 運算所造成的明顯效能降低。同時也解決 write requests 起始位置沒有對齊 effective page size 時，造成循序的資料寫入時無法進行 switch merge 與 partial merge 的現象。因此 write buffer 可以改善這些由 effective page size 所造成的負面影響，進一步地增進 SSD 的寫入效能。

下圖中描述，當 host 端下達了寫入 Page 0 → Page 11 →Page 1 → Page 2 →…→ Page 3，此一系列非循序的寫入命令時，這些要被寫入的 pages 會在 write buffer 內被重新整理為一段循序資料，如 Page 0 → Page 1 → Page 2 → Page 3，因此資料能以較有效率的循序寫入方式被寫回 NAND flash memory 中，也使得 flash translation layer 有較高的機率可以進行快速的 switch merge 或 partial merge 以提升 SSD 的寫入效能。




圖表 21 : reordering request

而使用 write buffer 也能減少 SSD 實際對於底層 NAND flash memory 的寫入次數，以提升 SSD 處理 write request 的能力。由於 SSD 在處理 write request 時，會先將資料暫存於 write buffer 中，因此對於重複且頻繁被更新的資料，在資料還未被寫回 NAND flash memory 前，SSD 只需要更改 write buffer 內此段資料為最新的內容，並在 write buffer 決定此段資料要被寫回時，才對 NAND flash memory 進行一次的寫入動作即可。而減少 NAND flash memory 實際寫入的資料量除了可以提升 SSD 處理 write request 的效能外，更由於 SSD 採用 out-of-place 更新的特性，還可以延緩 garbage collection 被觸發的時間，進而提升儲存裝置的整體效能，同時也能降低快閃記憶體區塊的進行抹除的次數，並延長 NAND flash memory 甚至是整個 SSD 儲存裝置之壽命。

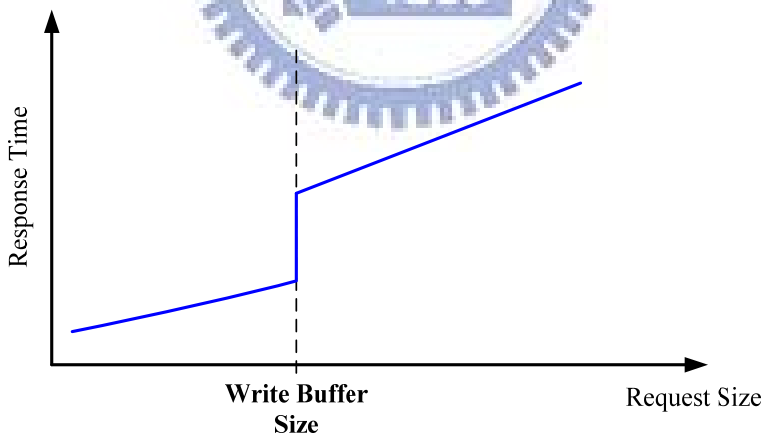
## 3.3.2 Impact of Read Cache/Write Buffer

因為許多應用程式都有循序讀取資料的行為，因此很有可能接下來要被讀取的資料其邏輯記憶體位址是連續上一個 read request 的。若要被讀取的資料已經預先被讀取至 read cache 時，儲存裝置就可以直接由 read cache 中讀取資料。由於 SSD 採用了 multi-channel 與 interleaving 等平行化運作方式，一次讀取較多連續的資料時可以達到較高的效能，因此搭配 pre-fetch 的機制可以改善許多尺寸較小且位址連續的 read request 效能。但對於隨機讀取的情況時，read pre-fetch 機制除了無法對讀取效能帶來任何好處外，一次讀取較多的資料也增加了底層 NAND flash memory

的忙碌時間與使用 channel 傳輸資料的使用時間，還有可能為了儲存過多 pre-fetch 的資料而不斷的將暫存在 read cache 中還有利用價值的資料替換掉，反而讓 read cache 對讀取效能的幫助進一步的下降。

而在啟用 write buffer 時，所有要被寫入 SSD 中的資料都會先暫存於 write buffer 中，並由 write buffer 所採用的管理機制決定寫回底層 NAND flash memory 中的時機。而在實際寫入資料時，當 write buffer 內還有閒置的空間可使用時，SSD controller 只需要先將資料寫入 write buffer 中，即可以立即回報 computer 端 write request 已經完成，因此可以得到非常短的回應時間，讓寫入效能明顯提升。但是當 write buffer 內暫時無閒置空間時，SSD controller 就必須先將 write buffer 內的資料立即的寫回 NAND flash memory 中，才能清理出足夠的閒置空間來儲存新的寫入資料。由於將資料寫入 NAND flash memory 中是非常耗時的，所以此時完成 write request 的回應時間就會明顯增加，寫入效能也大幅降低。因此非常明顯的 write buffer size 直接影響到 write buffer 所能暫存的資料量，而採用容量越大的 write buffer 則越不容易發生 write buffer 內無閒置空間的情況。

下圖描述對 SSD 寫入不同大小資料量時的回應時間，當寫入 SSD 的資料量大於 write buffer size 時，由於需要等待 write buffer 將內部的資料寫回 NAND flash memory，因此 response time 會有明顯的提升。




圖表 22 : Impact of Write Buffer Size

因此若作業系統能了解 write buffer 尺寸時，就可以運用在 disk cache 層面，在 data replacement 時 flush 適當的資料量，達到最佳的寫入效能。或是運用在 application 層面，讓設計者了解儲存裝置的 write buffer 與 read cache 能應付多大 locality 的資料存取行為，避免對於 cache buffer 進行無法負荷的存取，來最佳化整個應用程式的效能。

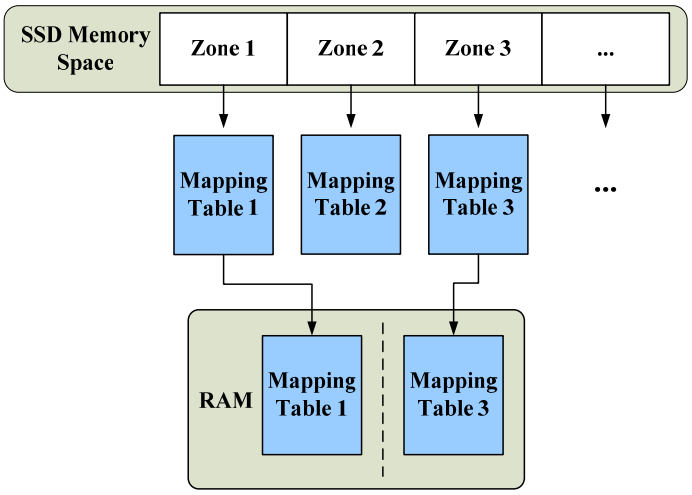
## 3.4 Zones

### 3.4.1 Definition of Zones

隨著 SSD 儲存裝置的容量快速地成長，造成 flash translation layer 運作時所需要參考的一些資料結構大小也越來越大，如：記錄 free blocks 或是 log blocks 的 lists，進行 address translation 所使用的各種 mapping tables 或是進行 wear leveling 所需要參考的 erase count table 等。而為了儲存裝置效能上的考量，其中許多經常被查詢或是變更的資料結構，是需要常駐於 SSD 內部的隨機存取記憶體中，以便隨時存取。因此對於大容量的 SSD，需要配合容量非常大的隨機存取記憶體，才能完整的儲存這些資料結構。而在 SSD 中採用過大的隨機存取記憶體除了造成製造成本上的負擔，也需要額外消耗大量的電力來維持隨機存取記憶體的運作，而抵消 NAND flash memory 本質上所具有的低電源消耗的優點。

因此許多 SSD controllers 會將內部的 NAND 快閃記憶體儲存空間切割為許多相同大小的區域，我們稱之為 zones，而每一個 zone 皆擁有與其他 zones 獨立的資料結構。如：每一個 zone 都有自己的 mapping tables，erase count table 等。因此採取分段管理後，原本一份龐大的資料結構，如：整個 NAND 快閃記憶體儲存空間的 mapping table，將會以 zone 為單位被分割為許多尺寸較小的 mapping tables。而採用將儲存空間進行分段管理的方式，對於 SSD 內部有限的隨機存取記憶體空間而言，可以在需要存取 zone 的資料結構時才將此部分的資料結構載入記憶體內，而同一時間點只儲存一個或是多個 zones 的管理資訊，可以進行更彈性化的空間管理與運用，並有效的解決各種資料結構大量佔用隨機存取記憶體空間的問題。

另外，與節省所佔用的隨機存取記憶體空間相同的原理，採用分段管理之概念亦有利於縮短 SSD 儲存裝置之初始化時間。也就是說，SSD 在開機時並不需要重建或載入整個 NAND 快閃記憶體空間的 mapping table，僅需於系統存取資料時，在將被存取到的 zone 的 mapping table 載入，便可立即開始運作。




圖表 23 Zones

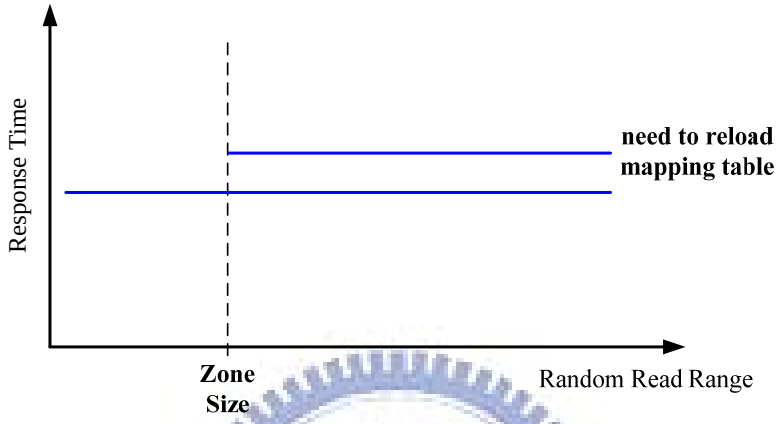
## 3.4.2 Impact of Zones

對於 SSD controller 採用分段管理之機制，雖然能有效降低各種資料結構所佔用的隨機存取記憶體空間，但也由於每一個 zone 都有自己的 mapping table，因此當所要存取之 mapping table 不在隨機存取記憶體中時，flash translation layer 必須先從 NAND flash memory 中讀取所需要的 zone 的 mapping table，甚至是需要重新掃描此 zone 之 spare area 中所記錄的邏輯記憶體位置對應資訊，以建立所需之 mapping table。而這些重新載入或是重建 mapping table 的行為都會影響寫入或是讀取時的回應時間，並降低整體效能。

因此當 SSD 內部所儲存的資料被以隨機的方式進行讀取或寫入時，很容易會因為所要存取的資料分散於不同的 zone 中，而造成所要使用的 mapping table 經常不存在於隨機存取記憶體中的現象，導致 flash translation layer 頻繁地花費額外的時間來進行重新載入 mapping table 的動作。而大量替換隨機存取記憶體內的 mapping table，除了要花費許多時間至 NAND flash memory 中讀取 mapping table 的資料，根據不同的 SSD controller 設計，flash translation layer 還有可能需要先將隨機存取記憶體內現有的 mapping table 寫回 NAND flash memory 中，才能清理出多餘的空間來存放要被載入的 mapping table，而這些額外寫入或讀取 mapping tables 的負擔在隨機的寫入或讀取資料時都會更加的明顯。

下圖描述 zone size 對於隨機讀取固定大小資料時的影響，當隨機讀取的資料都分布於在同一個 zone 裡時，由於所需要的 mapping table 已經存在於隨機存取記憶體中，所以不必進行任何替換 mapping table 的動作，因

此完成 read request 所需要的回應時間非常穩定而且較低。而隨機讀取的範圍大於 zone size 時，一旦相鄰的 read request 分別橫跨於不同 zone，就會造成隨機存取記憶體中的 mapping table 被重新替換，此時便對讀取效能造成額外的負擔。因此完成 read request 所需要的回應時間會呈現二種分佈，其中較高的部分為需要額外載入 mapping table 動作時的回應時間，較低的部分則為單純進行讀取資料時的回應時間。



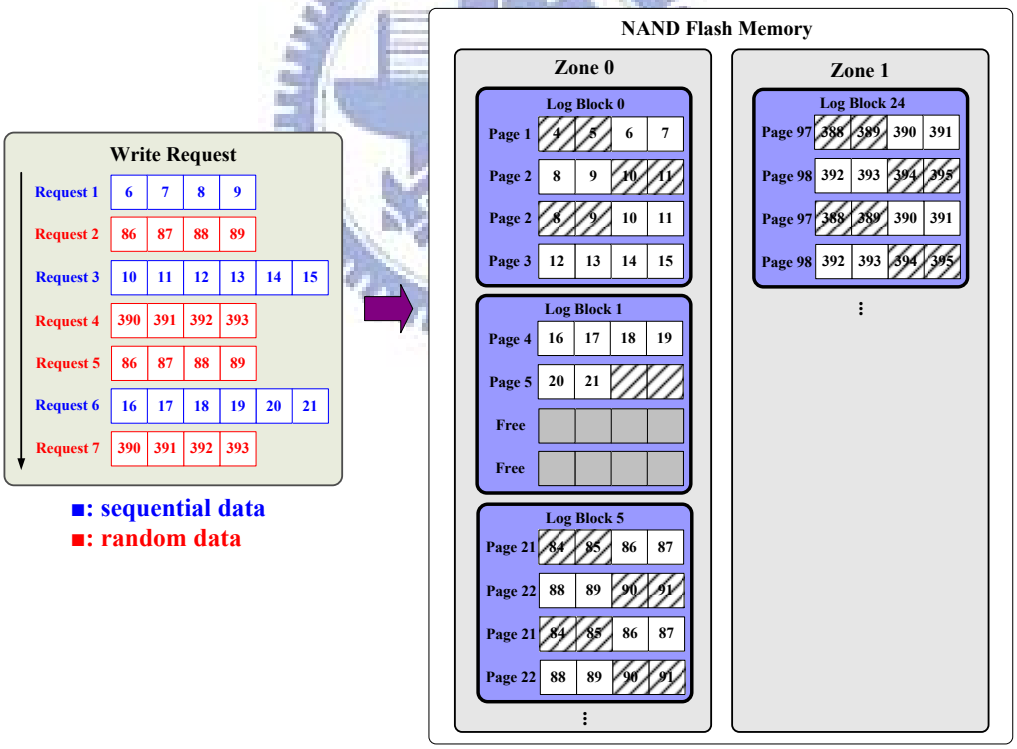

圖表 24 : Impact of Zone Size

因此在檔案系統分配容易被存取的資料或 metadata 時，能盡量的集中在同一個 zone 裡，就能盡量避免重新載入 mapping table 的動作，以增加隨機存取得的效能。否則在存取行為不斷分散在不同的 zone 時，就可能會造成 mapping table thrashing 的行為，導致 FTL 花了額外的時間不斷的進行重新載入 mapping table 的動作。

## 3.6 Example of Optimization

在介紹完所要測試的 SSD geometry 後，我們舉出一個簡單的範例，說明如何在檔案系統分配檔案空間時利用所測得的 effective block size、effective page size 以及 zone size 進行效能最佳化。

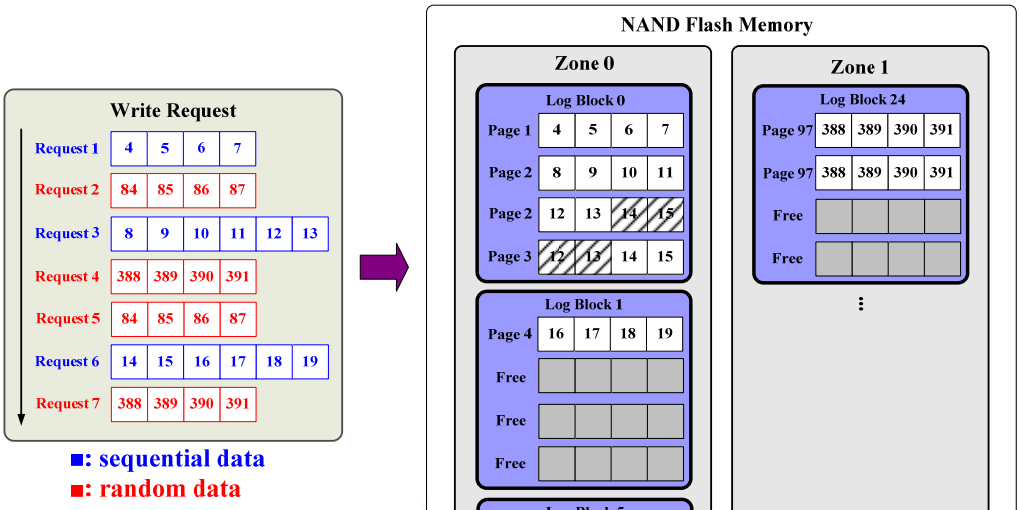
如下圖所示，在未經過任何最佳化時的 write request 中，可明顯觀察出 Request : 1、3、6 為循序更新，由 sector 6 更新至 sector 21 共 4 個 effective page 的資料(1 effective page = 4 sectors)；而 Request : 2、4、5、7 為寫入被隨機存取的 hot data。而 NAND flash memory 中呈現 write request 寫入 SSD 後的資料分佈情形，其中含有斜線區塊的 effective page 需要進行 read-modify-write operation，我們可以發現由於 request 都沒有對齊 effective page 的起始位置，因此大部分的 effective page 都必須要進行 read-modify-write operation 而造成效能低落，且必須寫入 14 個 effective page，遠大於實際更新的 8 個 effective page size 的資料。




圖表 25 : Optimization - Initial

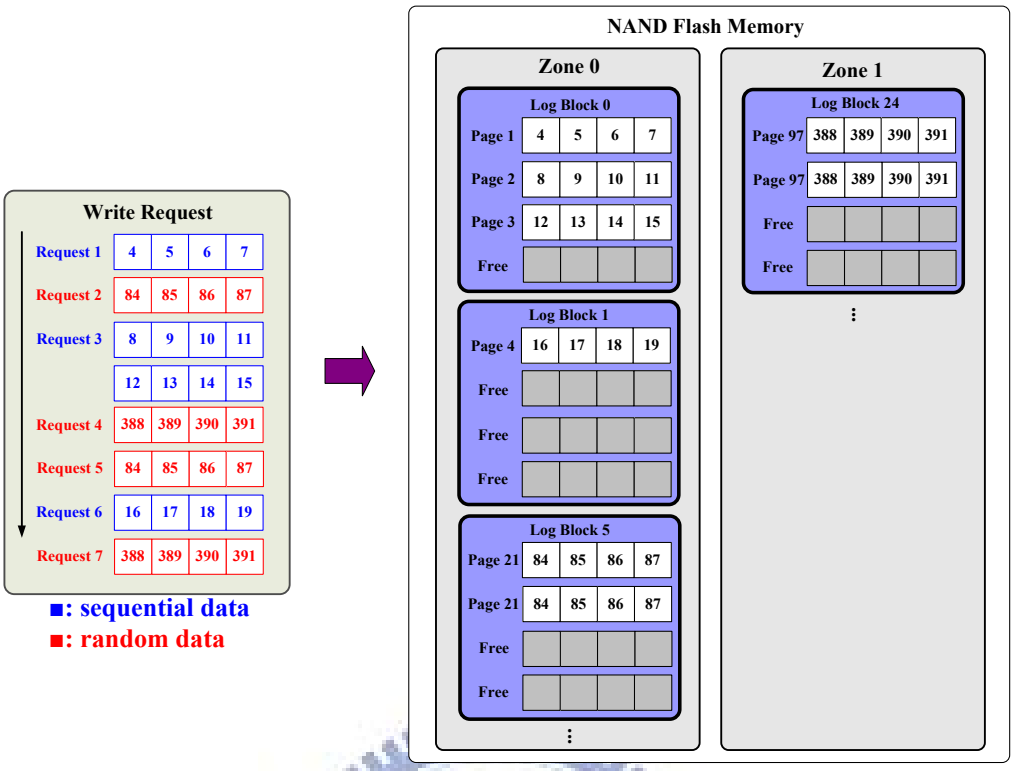
若我們能在檔案系統分配檔案空間時，就直接的將檔案對齊 effective page 的邊界，如下圖所示，大部分的 write request 就可以對齊 effective page，幾乎去除了大部分的 read-modify-write 的負擔，並大幅減少了 NAND flash

memory 中實際寫入的資料量，所需寫入的 effective page 數量由 14 個降低為 9 個。但是由 request size 沒對齊 effective page size 倍數而造成循序資料在 SSD 內成為不循序的問題還是無法解決。



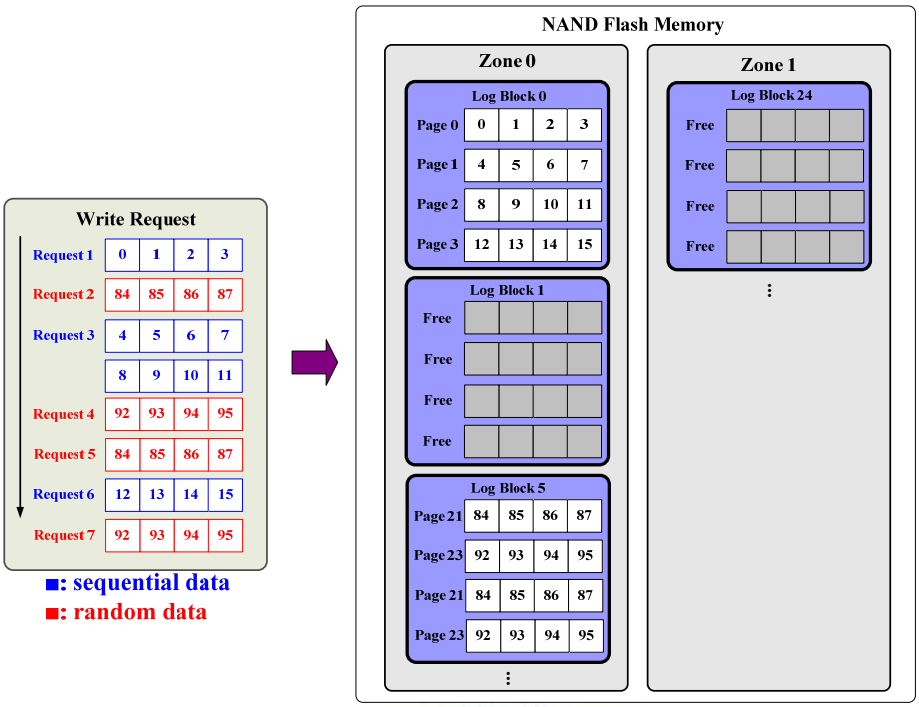

圖表 26 : Optimization – Step 1

接下來，若檔案系統能利用 effective page size 做為基本的存取單位，所有的 write request size 就會對齊 effective page size 的倍數，除了能完全去除 read-modify-write 的現象，更能解決循序資料無法被循序更新的問題。如下圖所示，在寫入循序資料時只需循序的寫入 effective page : 1~4。




圖表 27 : Optimization – Step 2

最後，若能在分配檔案位置時就將大型檔案對齊 effective block 的邊界，就能大幅增加 sequential write 時進行 switch merge 的機率。如下圖所示當 request 由 effective page 0 的起始 sector 開始進行更新時，就能使用 switch merge 以降低 garbage collection 的成本。另外，在容易被隨機存取的 hot data 部分，若能將這些檔案分配在同一個 zone 裡，就可以減少 reload mapping table 的動作；甚至是將這些檔案分配在同一個 logical block 內，更可以增加 log block 的利用度，減少 garbage collection 的負擔進而提升隨機存取的效能。




圖表 28 : Optimization – Step 3

在經過了完全的最佳化後，我們可以明顯看出我們除去了所有的 read-modify-write operation，並且將所需要寫入的 effective page 數量由未最佳化前的 14 個 effective page 降低至 8 個 page。同時讓循序更新的部分可使用 switch merge，也最佳化了對於 hot data 的隨機存取能力。大幅的提升了 SSD 的效能表現。

# 4. EXPERIMENTAL RESULTS

## 4.1 Experimental and Performance Metrics

在我們呈現SSD geometry的測試結果前，我們將簡單的介紹我們對於各個geometry的測試方法以及原理，而我們所有的測試程序是在採用Intel Pentium 4 CPU (3.4 GHz)的個人電腦中，在Windows XP作業系統中進行。並利用Windows API：ReadFile()、WriteFile()直接對於底層儲存裝置進行read/write運算，以避免檔案系統的干擾。也利用Windows API所提供的DeviceIoControl()配合IOCTL_ATA_PASS_THROUGH功能直接對儲存裝置下達ATA command，對SSD進行如DISABLE READ CACHE、DISABLE WRITE CACHE、FLUSH WRITE CACHE等特殊控制。

而在校能指標方面，主要以read/write request的response time進行分析，來觀察SSD內部所花費的運算成本，以及FTL所觸發的管理活動。而在時間量測則以read time stamp counter (RDTSC)的方式，直接計算所花費CPU clock數來達到更精準的response time量測。而由於觸發garbage collection時的response time的分布是非常廣的，因此在判斷是否觸發garbage collection時也常以throughput來進行更簡易的觀察。而在一些SSD geometry的測試中，為了能更精準的測試出read/write request真正的response time變化，通常會關閉write buffer或read cache來直接觀察出FTL對於底層NAND flash memory所進行的操作。

下表為我們所測試的所有 SSD，由於 MLC SSD 在寫入效能表現較不穩定，因此此章節中主要的測試結果將以 SLC SSD 的產品進行呈現。

| Brand | Model | Type | Size |
|---|---|---|---|
| Transcend | TS16GSSD25S-S | SLC | 16 GB |
| Transcend | TS32GSSD25S-M | MLC | 32 GB |
| SAMSUNG | MCBQE32G5MPP-0VA | SLC | 32 GB |
| Mtron | MSP-SATA7525-032 | SLC | 32 GB |
| Intel | SSDSA2MH080G1GC | MLC | 80 GB |
| OCZ | OCZSSD2-1C64G | MLC | 64 GB |
| OCZ | OCZSSD2-1VTX60G | MLC | 60 GB |

表格 2 : Testees

## 4.1 Effective Block Size

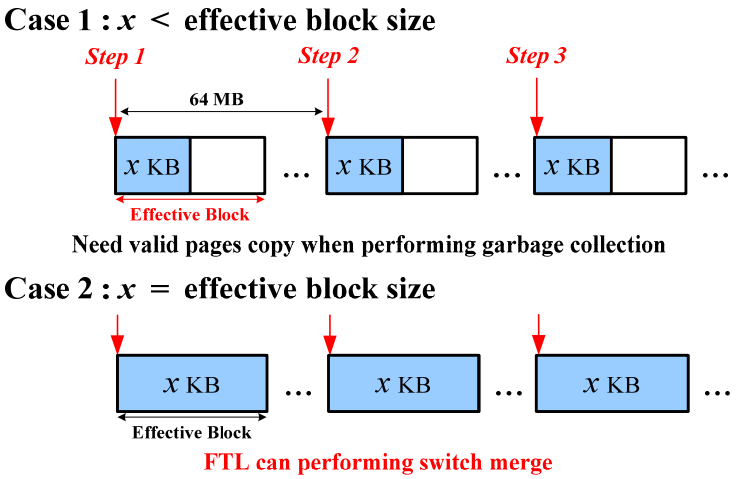
### 4.1.1 Effective Block Size Detector

在 block-level mapping FTL 中，當整個 data block 內的資料完全被循序更新時就可以進行成本較低的 switch merge，而不需要如 partial merge 與 full merge 的任何有效資料搬移的動作。因此在 effective block size 的測試方法中，我們便是利用調整循序更新的資料量讓 SSD 內部可進行 switch merge，來測試出我們的 effective block size。下表呈現 ***Effective Block Size Detector*** 的測試程序：

| Step | Action | Description |
|---|---|---|
| 0 | Variables | *star_address* ＝ 0<br>*x* ＝ 64 (KB) |
| 1 | Initialize | Fill the SSD by sequential writing all the LBAs of the device |
| 2 | Exercise | Write the *x* KB data from *start_address*<br>then *start_address*＝*start_address*＋(64 MB) |
| 3 | Steady-State | Jump to step 2 until throughput is steady |
| 4 | Calculate | If throughput reach the MAX，return *x*<br>else jump to step 1 and *x*＝*x* × 2 |

表格 3 : Procedure of Effective Block Size Detector

隨著寫入資料量：*x* 不斷的增加，如下圖所示，主要可以分為下列二種情況；寫入資料量小於 effective block size 與寫入資料量恰巧為 effective block size。如 case 1 所示，當寫入資料量小於 effective block size 時， FTL 在進行空間回收時必須使用 partial merge，此時必須進行額外的有效資料搬移，因此 throughput 較差。但在 case 2 中，由於寫入資料量恰巧為 effective block size 時，FTL 可直接採 switch merge，可以避免任何的有效資料搬移，而達到循序寫入的最高 throughput。由於 request 之間採取一段足夠的距離(如 64MB)以確保每次的 write request 都對應到不同的 logical block，因此在 one-to-one mapping 的方式中會造成 log block 被大量消耗而不斷觸發 garbage collection，而在 many-to-one mapping 的方式中也會造成 merge operation 之複雜度提高，導致 SSD 處在 garbage collection 的 worst case 下，使得有效資料搬移的成本可以明顯的觀察出來。

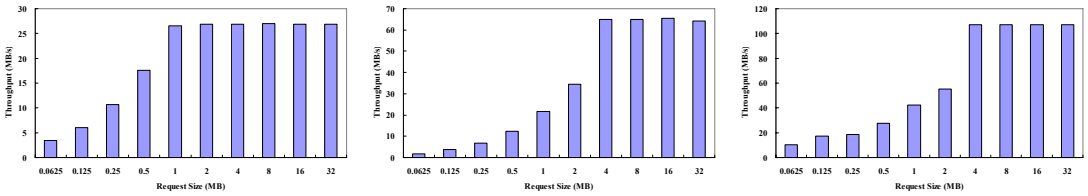



圖表 29 : Effective Block Size Detector

## 4.1.2 Result of Effective Block Size

下圖中為對於 effective block size 的實際測試結果，非常明顯的可以觀察出，Transcend SLC、SAMSUNG 與 MTRON 在 throughput 的成長都會有很明顯的分界點，如 1 MB、4 MB、4 MB 之後，由於可以使用 switch merge，循序寫入都可以達到穩定且最高的速度。因此他們的 effective block size 分別為 1 MB、4 MB 與 4 MB。

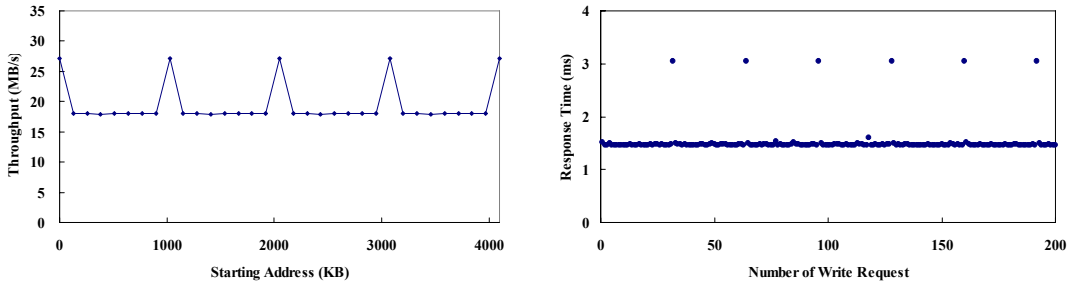
。




Tran-s, block = 1MB　　SAMSUNG, block = 4MB　　MTRON, block = 4MB

圖表 30 : Result of Effective Block Size

另外在寫入資料量恰巧為 effective block size 時，如果更新的起始位置沒有對齊 effective block 的起始邊界時，FTL 也會因為不能使用 switch merge 而在成效能低落。如下圖(A)所示，在 Transcend SLC 中儘管更新的資料量都為 1 MB，在起始位置沒有對齊 1 MB 倍數時寫入效能會大幅滑落。而我們也發現，很多時候 garbage collection 被觸發的間隔內，其更新的資料量十分接近 effective block size；如下圖(B)所示，在對同一個邏輯位置不斷進行更新時，當更新資料量達到 effective block size 時，SSD 會非常穩定的觸發一次 garbage collection。

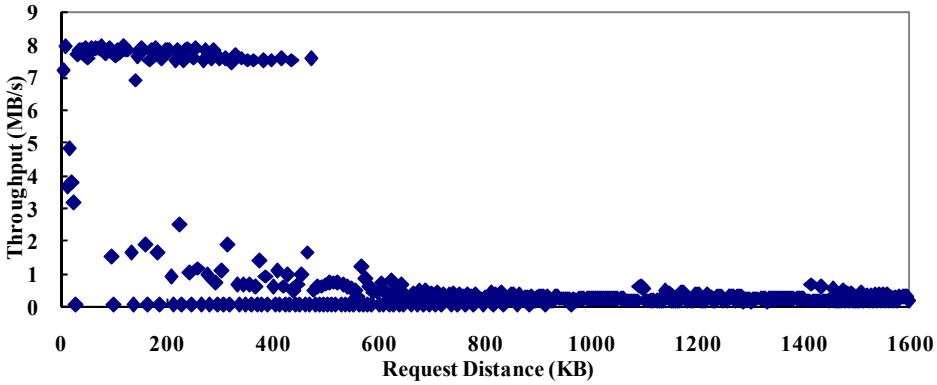



Align the block boundary　　Garbage Collection Frequency

圖表 31 : Impact of Effective Block

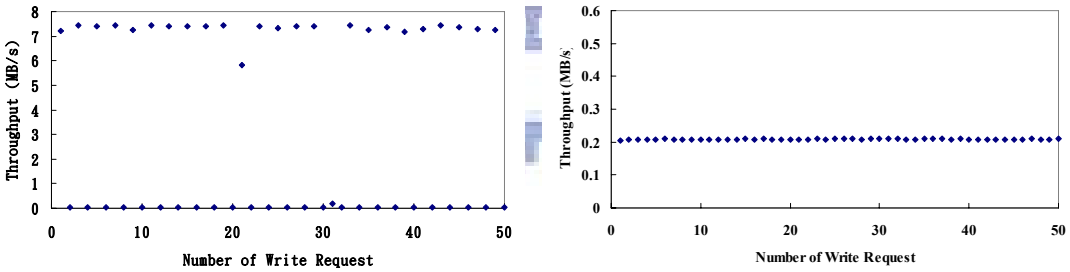
在實際產品的 random write 表現中，如下圖所示，我們也發現 random write 的速度並不是全然都是非常慢的，在 write request 之間的距離小於一個 effective block size 時，其所需的 response time 幾乎與循序寫入相同資料

量時差不多。但在隨機寫入的範圍大於 effective block size 時，效能則非常低落。




圖表 32 : Performance of Random Write

而在進一步實驗中可以發現，當 write request 之間的間隔距離越長時，進行 GC 的頻率越來越高，如下圖(A)當距離為 1/2 個 effective block size 時，每二次寫入會有一次造成回收。但是當跳躍距離達一個 effective block size 以上時，如下圖(B)，則會因為 log block 被大量的競爭，而造成 garbage collection 不斷的被觸發。




request distance : 1/2 effective block size　　request distance : 1 effective block size

圖表 33 : Impact of effective block to random write

## 4.2 Effective Page Size

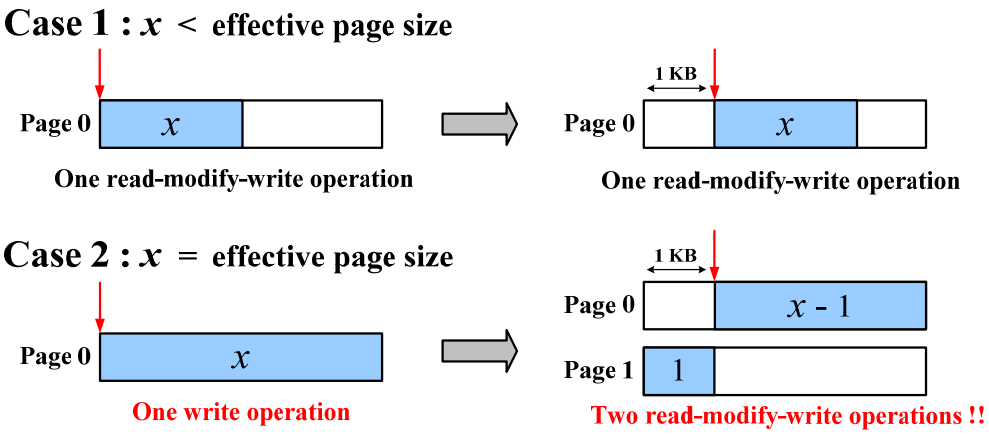
### 4.2.1 Effective Page Size Detector

在 effective page size 的測試中，我們主要藉由 write request 沒有對齊 effective page size 時需要進行 read-modify-write operation，導致所需要的 response time 較恰好更新一整個 effective page 時高，來觀察 SSD 內部的 effective page size。下表呈現 ***Effective Page Size Detector*** 的測試程序：

| Step | Action | Description |
|---|---|---|
| 0 | Variables | *start_address* ＝ 0 ;<br>*x* ＝ 1 (KB) ;<br>*THr-m-w* : threshold value of read-modify-write operation |
| 1 | Initialize | Fill the SSD by sequential writing all the LBAs of the device，<br>**then disable write-buffer** |
| 2 | Exercise | Write *x* KB data from *start_address*，then write *x* KB data from (*start_address*＋1), compute the difference between the response time of write requests：*dif* |
| 3 | Calculate | If *dif* > *THr-m-w*，return *x*<br>else jump to step 2 and (*x*＋＋) |

表格 4 : Procedure of Effective Page Size Detector

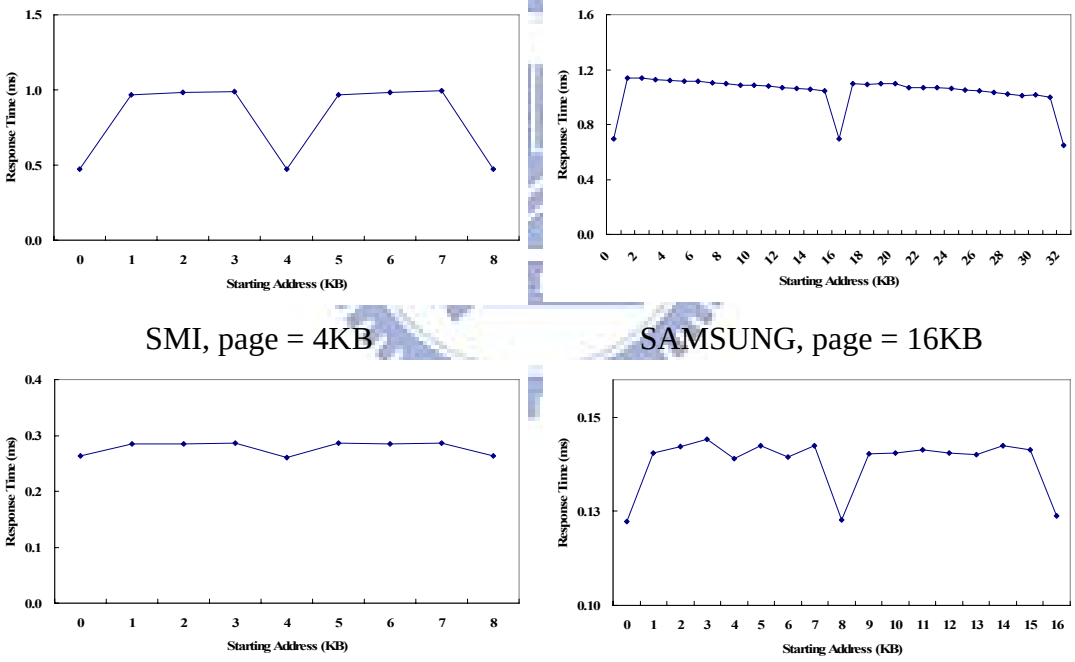
隨著寫入資料量：*x* 不斷的增加，如下圖所示，主要可以分為下列二種情況;寫入資料量小於 effective page size 與寫入資料量恰巧為 effective page size。如 case 1 所示，當寫入資料量小於 effective page size 時，不論是由起始位置 0 KB 或 1 KB 開始寫起，所需要的成本都是一次的 read-modify-write operation，因此二者所花費的 response time 是差不多的。但在 case 2 的情況下，由於所寫入的資料量恰巧為 effective page size，因此從起始位置 0 KB 開始寫起時只需要一次的 write operation，速度是非常快的；但是從起始位置 1 KB 開始寫起時卻需要二次的 read-modify-write operation，因此所需要的 response time 是非常久的，藉由比較二者之間 response time 的差異，我們就可以觀察出 effective page size。而此項測試必須在關閉 write buffer 的狀態下才能準確的測試，因為若啟用 write buffer 則無法直接測得資料寫入 NAND flash memory 的回應時間，因此觀察不出 read-modify-write 的成本。




圖表 34 : Effective Page Size Detector

## 4.2.2 Result of Effective Page Size

由下圖為(A)與(B)中，對於 Transcend SLC 與 SAMSUNG 的 effective page size 的測試結果，可以觀察出在寫入資料量分別為 4 KB 與 16 KB 時，從起始位置 0 KB 與 1 KB 寫入時，response time 的差異是非常大的，因此他們的 effective page size 分別為 4 KB 與 16 KB。另外我們也對於 read 進行測試，如下圖(C)所示，由於 read 不會受到 read-modify-write 的影響，因此在讀取時不論 request 是否對齊 effective page 的起始位置並不會造成明顯的差異；但是在一些無法關閉 write buffer 的產品中，可以簡用 read 的方式進行 effective page size 的測試，如下圖(D)中 MTRON 在 read request 為 8 KB 時，若正好對齊 effective page 的起始位置時，其 response time 還是會有些微的落差，這正是因為所需要進行的 read request 由於沒有對齊 page 的起始位置，因此所需要的 read operation 從一次變成二次所增加的微小成本。

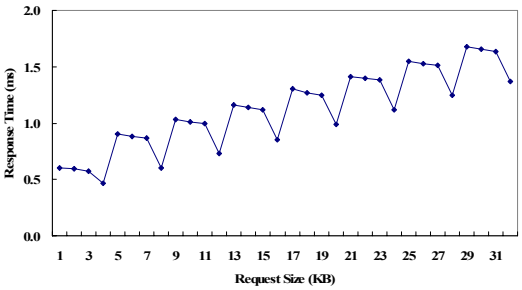



SMI, page = 4KB　　SAMSUNG, page = 16KB

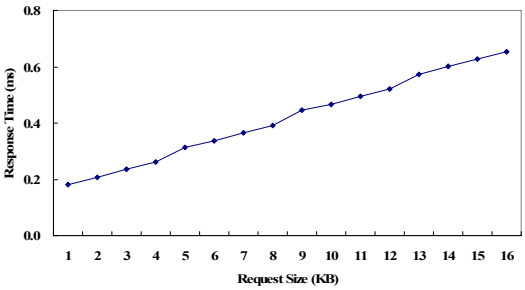
SMI READ　　MTRON READ, page = 8KB

圖表 35 : Result of Effective Page Size

下圖為 effective page size 對於 read/write request size 的影響，可以明顯看出在 write 時，當 write request size 沒有對齊 effective page size 的倍數時，由於必須進行 read-modify-write operation，因此會有 response time 較高的情況，response time 會呈現鋸齒狀的成長。但在 read 時，不論 read request size 是否對齊 effective page size 的倍數，由於不會受到 read-modify-write 的影響，所以 response time 是已接近線性的方式成長。




SMI, impact of write request size




SMI, impact of read request size

圖表 36 : Impact of Effective Page Size

## 4.3 Read Cache/Write Buffer Size

### 4.3.1 Read Cache/Write Buffer Size Detector

在 read cache size 與 write buffer size 的測試中，我們利用當要存取的資料量大於 cache buffer size 時，勢必需要等待 SSD 直接對於底層 NAND flash memory 直接進行存取資料的時間，因此 response time 較長，來測試出 cache buffer 所能暫存的資料量。而由於 read cache 與 write buffer 的測試原理相同，因此我們僅呈現出較有難度的 write buffer 測試方法。下表呈現 ***Write Buffer Size Detector*** 的測試程序：

| Step | Action | Description |
|---|---|---|
| 0 | Variables | $x$ ＝ 1 (KB) ; |
| 1 | Initialize | Perform random 4 KB write to consume the log blocks |
| 2 | Exercise | **Flush write-buffer and wait 5 sec**，then write the $x$ KB data |
| 3 | Calculate | If the response time of write request is increase obviously then return $x$, else jump to step 1 and ($x$＋＋) |

表格 5 : Procedure of Write Buffer Size Detector

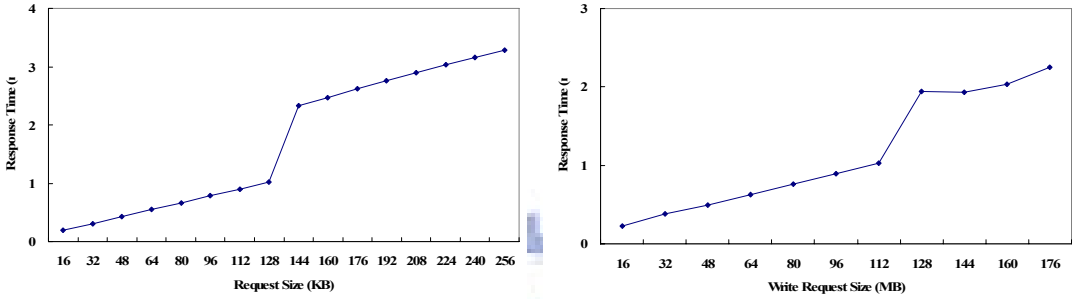
在 write buffer size 的測試方法中，所採用的主要概念為當寫入資料量大於 write buffer size 時，host 端必須等 write request 的部分資料先從 write buffer 中被寫回 NAND flash memory，才能將全部的 write request 寫進 write buffer 內；由於 NAND flash memory 的寫入速度與 RAM 相比是非常慢的，因此等待資料被寫回 NAND flash memory 的時間會明顯的反應在 response time 上。為了確保準確的測試出 write buffer size，我們在每次寫入資料前，都會利用 ATA command 下達一次 FLUSH BUFFER 指令，讓 write buffer 內部沒有任何的資料。但是在實際測試中卻發現，許多 SSD 產品在接收 FLUSH BUFFER 指令後，並不會一次就將所有內部資料進行寫回，因此我們必須額外等待一小段時間，以確保 write buffer 內完全沒有任何資料。

而在實際產品的測試中，我們發現對於許多尺寸較大且寫回速度較快的 write buffer，非常難將 write buffer 立即的填滿；write buffer 的空間被寫滿之前已有部分資料被寫回 NAND flash memory 中，因此很難準確的測

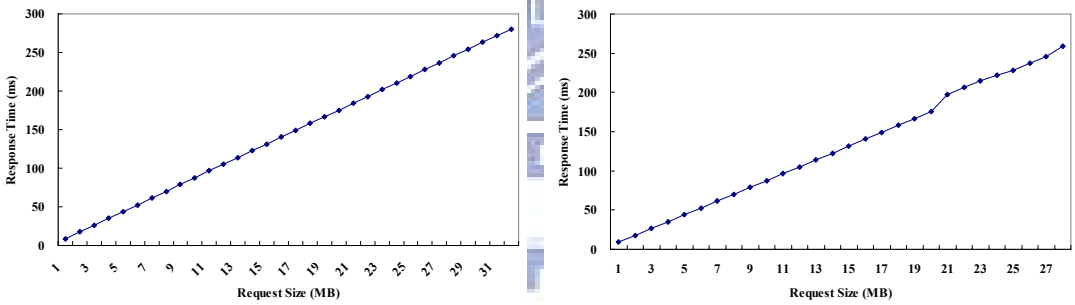
試出 write buffer size。所以我們採用了 random write 進行初始化，甚至直接利用 effective block size 的測試結果直接進行 worst case 的寫入，讓 SSD 內部的資料分佈成為容易發生 garbage collection 且 merge operation 複雜度較高的情況，盡量減緩 write buffer 內部資料寫回的速度，達到更精確的測量。

## 4.3.2 Result of Read Cache/Write Buffer Size

如下圖所示，對於 SAMSUNG 與 INTEL 的測試中當寫入資料量分別大於 128 KB 與 112 KB 時 response time 就會有明顯的增加，因此可以明顯的穿查出他們的 write buffer size 分別為 128 KB 與 112 KB。但在下圖(C)中可以發現，在沒有以 random write 對 MTRON SSD 進行初始化時，寫入資料量不會對 response time 造成任何影響，但是在進行 random write 的初始化之後，下圖(D)中可以看出在 20 MB 之後 response time 會有明顯的增加。



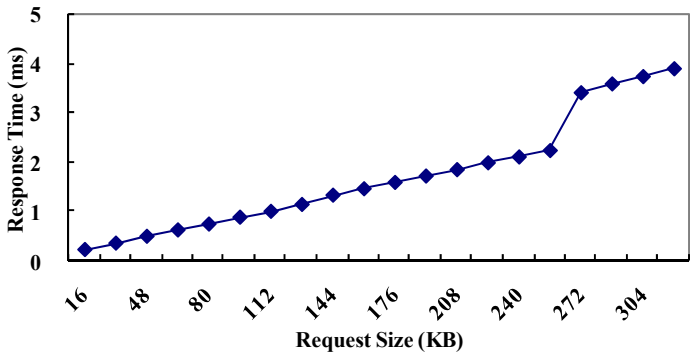

Samsung, write buffer = 128 KB　　INTEL, write buffer = 112KB




MTRON, no random write　　MTRON, write buffer = 20MB

圖表 37 : Result of Write Buffer Size Detector

而對於 read cache size 的測試中，由於並沒有資料寫回的問題，因此只需要簡單的以類似 write buffer size 的測試方式，慢慢的增加要讀取資料量的大小，並藉由反覆的讀取來測試是否所有的 read request 資料都可直接從 read cache 中取得。如下圖所示當要讀取的資料量小於 256 KB 時，read cache 能暫存所有要被讀取的資料，因此當資料再次被讀取時就可以直接由 read cache 中取得，達到較快的 response time。但在 read request size 大於 256 KB 時，由於 read cache 無法暫存所有要讀取的資料量，因此必須承受 SSD 直接從 NAND flash memory 中讀取資料的成本，造成 response time 有明顯增加。因此可觀察出 read cache size 為 256 KB。




SAMSUNG, read cache = 256KB

圖表 38 : Result of Read Cache Size Detector

## 4.4 Zone Size

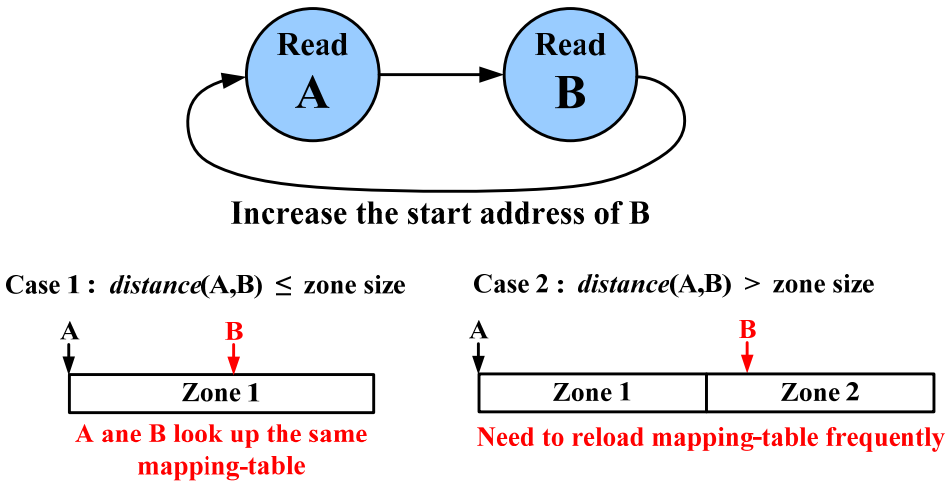
### 4.4.1 Zone Size Detector

在 zone size 的測試中，我們利用隨機讀取範圍大於 zone size 時容易造成 mapping table thrashing，而導致 response time 較高，來觀察 zone 的分界點。下表呈現 ***Zone Size Detector*** 的測試程序：

| Step | Action | Description |
|---|---|---|
| 0 | Variables | *start_address_A* ＝ 0 ;<br>*start_address_B* ＝ 0 ; |
| 1 | Initialize | fill the SSD by sequential writing all the LBAs of the device |
| 2 | Exercise | Read 512 bytes data from *start_address_A and start_address_B* separately and output the response time of the request from *start_address_B*, then (*start_address_B* ＝ *start_address_B*＋(1 MB)), jump to step 2 |

表格 6 : Procedure of Zone Size Detector

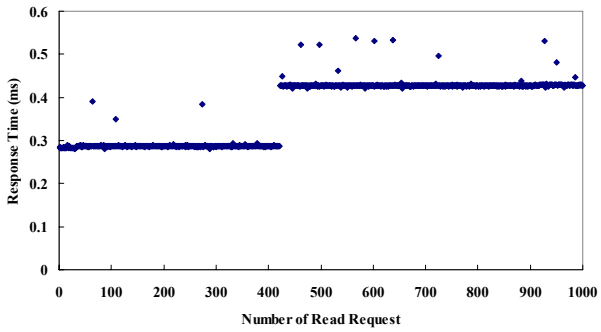
如下圖所示，在 zone size 的測試中我們不斷重複的讀取 A 點跟 B 點，並且慢慢遞增 B 點的起始位置來測試 request 之間的距離對於 response time 的影響。而測試的過程主要可分為二種情形；在 case 1 中當 A 點與 B 點的距離小於 zone size，此時由於 A 點與 B 點都落在同一個 zone 裡，因此不會產生任何重新載入 mapping table 的動作。但在 case 2 中當 A 點與 B 點的距離大於 zone size 時，由於 A 點與 B 點分別落在不同的 zone 中，因此會造成重新載入 mapping table 的動作，而造成 response time 較高。而來回讀取 A 點與 B 點的測試方式正是希望能不斷觸及不同的 zone，而造成 mapping table thrashing 的情況，讓載入 mapping table 的成本更加明顯。




圖表 39 : Zone Size Detector

## 4.4.2 Result of Zone Size

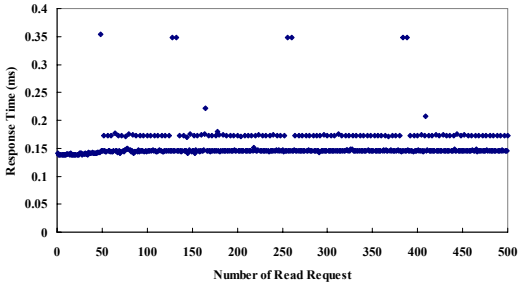
實際產品的測試結果如下圖所示，在 Transcend SLC 的產品中，可以明顯看出 zone size 為 476 MB。當 A 點與 B 點的距離小於 476 MB 時，由 B 點讀取 512 Bytes 所花費的 response time 明顯較低，但在 A 點與 B 點距離大於 476 MB 時由於必須進行載入 mapping table 的動作，因此從 B 點讀取 512 Bytes 所需要的 response time 會明顯且穩定的增加。



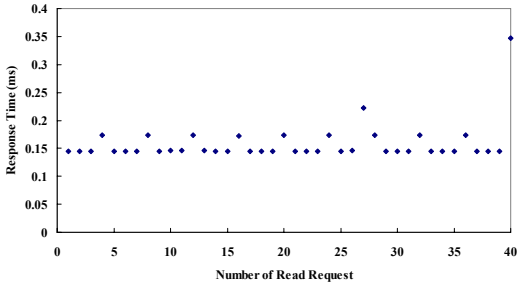

**Tran-s, zone size= 476 MB**

圖表 40 : Result of Zone Size Detector

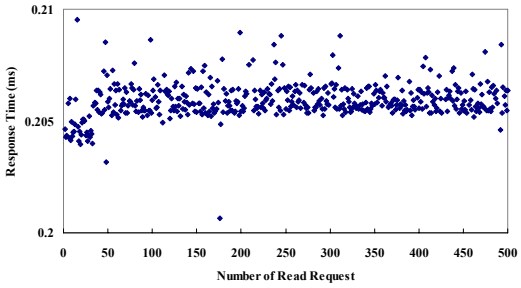
而在下圖(A)SAMSUNG 的測試結果中可以發現，從 B 點讀取 512 Byte 的 response time 大致上會直接呈現二種分佈，這與 Transcend SLC 中 A 點與 B 點的距離大於 zone size 時每次讀取 B 點都必須載入 mapping table 的現象不同。而在我們進一步的將 response time 分佈圖放大來看時，如下圖(B)所示，較高的 response time 分佈為每 4 MB 固定有一次較慢的讀取，而 4 MB 正是 SAMSUNG 的 effective block size，因此我們簡單的推斷 SAMSUNG 是以 effective block 為 zone 的基本分割單位，並且在一個 logical block 被存取時，才將所需要的管理資訊載入 RAM 中。另外對於 MTRON 的測試中，如下圖(C)中可以發現 MTRON 的 zone size 為 40 MB，在 A 點與 B 點距離超過 40 MB 左右時 B 點的 response time 會稍微的增加。最後在下圖(D)我們也呈現 SAMSUNG 在 random read 時由於受到載入 mapping table 動作的影響，response time 會呈現二種分佈。

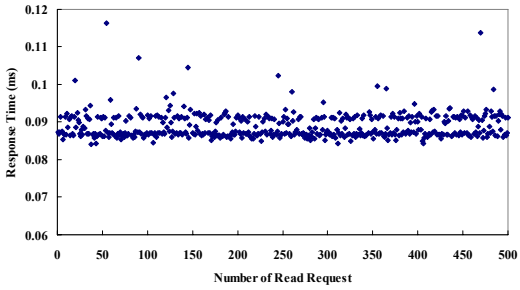



SAMSUNG, zone size detector




SAMSUNG, zone size = 4 MB




MTRON, zone size = 40 MB SAMSUNG Random read

圖表 41 : Result of Zone Size Detector

# 5. Conclusion

在這篇論文中我們提出一套從使用者層面測試出 SSD 內部 geometry 參數的方法，其中包括 effective block size、effective page size、zone size、write buffer size 與 read cache size，並分別呈現這些 geometry 在實際產品的使用中對效能表現的影響，接著說明要如何利用這些 geometry 參數對於 SSD 的儲存系統進行最佳化。而我們也相信若能將這些針對 SSD geometry 最佳化的方式運用在檔案系統、磁碟快取甚至是應用程式的設計中，除了能直接增加寫入與讀取的速度，更能有效降低 garbage collection 的負擔，讓 SSD 的存取能力得到更完整的發揮，明顯的提升整體的存取效能表現。

在未來，我們希望能進一步修改這些 geometry 的測試方法，突破如 effective page size 的測試中必須關閉 write buffer/read cache 等測試上的限制，並確保測試方法能推廣到更多或更新的 SSD 中。另外也希望能對更多的 SSD geometry 進行深入的測試，如 read cache/write buffer 的基本管理單位大小等，以達到更完整的最佳化。

# Reference


**[1]Po-Chun Huang, Yuan-Hao Chang, Tei-Wei Kuo, Jen-Wei Hsieh,and Miller Lin,” The Behavior Analysis of Flash Memory Storage Systems “,11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), 2008**

**[2]Luc Bouganim, Björn Þór Jónsson,and Philippe Bonnet,” uFLIP: Understanding Flash IO Patterns”, 4th Biennial Conference on Innovative Data Systems Research (CIDR),2009**

**[3] Nitin Agrawal, Vijayan Prabhakaran,and Ted Wobber,” Design Tradeoffs for SSD Performance”, USENIX TechnicalConference,June 2008**

**[4] Cagdas Dirik, and Bruce Jacob,” The Performance of PC Solid-State Disks (SSDs) as a Function of Bandwidth, Concurrency, Device Architecture, and System Organization”, ISCA’ 09, 2009**

**[5] Zoran Dimitrijeví´, Raju Rangaswami, David Watson, and Anurag Acharya2, ” Diskbench: User-level Disk Feature Extraction Tool”, springer, 2004**

**[6]Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger, “Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics”], Conference on File and Storage Technologies (FAST) January 28-30, 2002**

**[7] Dongjun Shin, and Samsung Electronics “About SSD” , www.usenix.org/event/lsf08/tech/shin_SSD.pdf**

**[8]David A. Patterson, John L. Hennessy ,"Computer Organization & Design The Hareware/Software Interface"**

**[9]Silberschatz,Galvin,Gagne,"Operation System Concepts"**

**[10]Samsung Electronics. OneNAND Features and Performance, 11 2005**

**[11]J.-H. Lee, G.-H. Park, and S.-D. Kim. A new NANDtype**

**flash memory package with smart buffer system for spatial and temporal localities. JOURNAL OF SYSTEMS ARCHITECTURE, 51:111–123, 2004.**

**[12]Software Concerns of Implementing a Resident Flash Disk. Intel Corporation.**

**[13]Ajwani, D., Malinger, I., Meyer, U., Toledo, S. Characterizing the performance of flash memory storage devices and its impact on algorithm design. Proc. Workshop on Experimental Algorithms (WEA), Provincetown, MA, USA, 2008**

**[14]A. Birrell, M. Isard, C. Thacker, and T. Wobber. A Design for High-Performance Flash Disks. Operating Systems Review, 41(2):88–93, 2007.**

**[15]H. Kim and S. Ahn. A Buffer Management Scheme for Improving Random Writes in Flash Storage. In Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST ' 08), pages 239–252, 2008.**

**[16]Samsung Corporation. K9XXG08XXM Flash Memory Specification. http://www.samsung.com/global/system/ business/semiconductor/product/2007/6/11/ NANDFlash/SLC_LargeBlock/8Gbit/K9F8G08U0M/ ds_k9f8g08x0m_rev10.pdf, 2007**

**[17]Shah, A. Samsung, Microsoft in talks to speed up SSDs on Vista. ComputerWorld (computerworld.com), August 18, 2008**

**[18]Trayger, A., Zadok, E., Joukov, N., Wright, C. P. A nine year study of file system and storage benchmarking. ACM Transactions on Storage, 4(2), 2008**

**[19]Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. USENIX Workshop on File Systems (Ann Arbor, MI), pages 79–91, 21–22 May 1992.**

**[20]Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. The DiskSim simulation environment version 1.0 reference manual, Technical report CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.**

**[21]Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file**

**systems with adaptive methods. ACM Symposium on Operating System Principles (Saint-Malo, France, 5–8 October 1997). Published as Operating Systems Review, 31(5):238–252. ACM, 1997.**

**[22]Jeffrey Katcher. PostMark: a new file system benchmark. Technical report TR3022. Network Appliance, October 1997.**

**[23]David M. Jacobson and John Wilkes, "Disk scheduling algorithms based on rotational position," HPL Technical Report, February 199**

**[24]Birrell, A., Isard, M., Thacker, C., and Wobber, T. 2007. A Design for High-Performance Flash Disks. ACM SIGOPS Operating Systems Review, vol. 41, no. 2, 88-93.**

**[25]Bisson, T., and Brandt, S. A. 2007. Reducing Hybrid Disk Write Latency with Flash-Backed I/O Requests. In Proceedings of the 15th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems. MASCOTS'07.**

**[26]Dumitru, D. 2007. Understanding Flash SSD Performance. http://managedflash.com/news/papers/easycoflashperformance-art.pdf (August 2007).**

**[27]Kim, Y., Lee, S., Zhang, K., and Kim, J. 2007. I/O Performance Optimization Techniques for Hybrid Hard Disk-Based Mobile Consumer Devices. IEEE Transactions on Consumer Electronics, vol. 53, no. 4 (November 2007), 1469-1476.**

**[28]Manning, C. 2004. YAFFS: Yet Another Flash File System. http://aleph1.co.uk/yaffs**

**[29]NAND Flash-based Solid State Disk Module Type Product Data Sheet. Samsung Electronics Co., Ltd., http:// www.bigboytech.com/new/v1.5/ssd/docs/ ssd_module_type_spec_rev121.pdf, January 2007.**