國立臺北科技大學

資訊工程系碩士班

碩士學位論文

以資料冷熱分群之策略增進開放通道固態硬碟空間回收效率

Exploring Hot/Cold Data Separation for Garbage Collection Efficiency Enhancement on Open-Channel SSD

研究生：吳承岳

指導教授：陳碩漢 博士

中華民國一百一十一年九月

摘 要

論文名稱：以資料冷熱分群之策略增進開放通道固態硬碟空間回收效率

頁數：28

校所別：臺北科技大學 資訊工程系碩士班

畢業時間：一百一十學年度第二學期

學位：碩士

研究生：吳承岳

指導教授：陳碩漢 博士

關鍵字：LightNVM、Open-Channel SSD、SSD、Garbage Collection、垃圾回收效率、Linux、Kernel Module

一直以來，SSD (Solid State Drive) 都有三個限制，Erase Before Write 跟 Limited Program/Erase Cycles 跟 Asymmertic Program/Erase Unit，而過去都是讓 SSD 之中的 FTL (Flash Translation Layer) 獨自管理。但是這樣 SSD 並沒有 Host 的資訊，也就是並沒有跟 Host 溝通，造成 Host 與 SSD 雙方的資訊落差，進而導致效率下降。而 Open Channel SSD 就是為了讓 Host 與 FTL 溝通而改良而成的一種新型 SSD。

而本論文針對 Open-Channel SSD 提出一個設計，將資料劃分為冷門到熱門四個不同的等級，並將資料依照等級分開擺放來增進 Garbage Collection 的效率，。讓 Open-Channel SSD 不再只能循序寫入，現在可以透過集中熱門資料 (Hot Data) 以及冷門資料 (Cold Data)，來提升整體效率。

ABSTRACT

Title:Exploring Hot/Cold Data Separation for Garbage Collection Efficiency Enhancement on Open-Channel SSD

Pages:28

School: National Taipei University of Technology

Department: Computer Science and Information Engineering

Time: September,2022

Degree: Master

Researcher: Cheng-Yueh Wu

Advisor: Shou-Han Chen, Ph.D.

Keywords: LightNVM, Open-Channel SSD, SSD, Garbage Collection, Performance, Linux, Kernel Module

In the past, the Flash Translation Layer (FTL) in the Solid State Drive (SSD) has been left to manage the SSD internal alone. However, the SSD did not communicate with the Host, resulting in a gap in information between the Host and the SSD, which in turn led to a decrease in efficiency. Open Channel SSD is a new type of SSD that is improved to allow the Host to communicate with the FTL.

This thesis proposes a method by dividing the data into four different levels, from cold to hot, for Open Channel SSD. We separate the data according to the level to improve the efficiency of Garbage Collection. Open-Channel SSD can no longer only write sequentially. They are able

to improve the overall efficiency by centralizing Hot Data and Cold Data now.

誌謝

在此要先感謝我的指導教授陳碩漢老師，由於本身有先離開學校一兩年再回歸學習的環境，雖然大學畢業就是資訊工程系，但是除了為了考試所準備的東西比較熟悉，其他範圍其實已經有點生疏。不過在碩士兩年時光，修了很多作業很多的課程，還有老師的要求不斷的激勵我，讓我在短時間內學習到眾多的專業知識與技能，也讓我有機會鑽研 Linux Kernel，並且包容我的魯莽以及失誤。雖然這段時間有時候也會不知所措，不過最後看到成果，心中還是充滿著喜悅。

同時感謝口試委員陳碩漢老師、梁郁珮老師、劉傳銘老師，對於我的論文提出十分多的建議，讓我的論文能夠更加完整。也感謝吳俊青學長和林稟宸學長，在我開始就讀碩士之後，對於我進修課程以及程式設計上給予許多建議。此外也感謝黃國豪同學，在我在寫作業或是研究時如果有想法卡關之類的狀況，都會給予我協助與建議。此外十分感謝研究室內的同學們、學長們、學弟妹們，在我寫論文的過程中給予我很多幫助，以及兩年內所有的陪伴，在這個環境下成長是一件十分快樂的事情。

最後感謝我的家人們對於我在讀研究所時，給予關心以及關懷，讓我可以無後顧之憂準備論文，真的非常感謝你們的支持與鼓勵，讓我知道我不是一個人在面對這一切，並且順利取得碩士學位。

目錄

中文摘要 i

ABSTRACT ii

誌謝 iv

目錄 v

表目錄 vii

圖目錄 viii

第一章 緒論 1

1.1 研究背景與動機 1

1.2 研究目標 2

1.3 論文組織架構 2

第二章 背景知識 3

2.1 SSD 3

2.1.1 NAND FLASH Memory 3

2.1.2 Asymmertic Program/Erase Unit 4

2.1.3 Erase Before Write 5

2.1.4 Limited P/E Cycles 5

2.2 FTL 6

2.2.1 Wear Leveling 7

2.2.2 Mapping Table 8

2.2.3 Garbage Collection 9
2.3 Open-Channel SSD 9
2.4 LightNVM 10
2.4.1 Line 11
第三章 研究設計與實作 12
3.1 初始化時修改為準備四個 Line 14
3.2 處理檔案系統要求之流程 15
3.2.1 以平均值劃分熱門資料及冷門資料 16
3.2.2 擷取 Request 大小並判斷分群 16
3.2.3 計算平均值 17
3.2.4 將 Request 所屬分群的資訊放入 Ring Buffer 中 18
3.3 實際寫入之過程 18
3.3.1 從 Ring Buffer 中蒐集 Entry 19
3.3.2 切換 Line 之後傳送給 Open Channel SSD 19
第四章 案例實驗 21
4.1 實驗環境與參數 21
4.2 實驗規劃 21
4.3 實驗一：Garbage Collection 次數比較 22
4.4 實驗二：寫入資料量比較 23
第五章 結論與未來展望 25
5.1 結論 25
5.2 未來方向 25
參考文獻 27

表目錄

表 4.1　在同樣資料量情況下，Garbage Collection 次數比較 22

表 4.2　在同樣時間情況下，寫入資料量比較 23

圖目錄

圖 2.1 Asymmertic Program/Erase Unit 4

圖 2.2 Erase Before Write 5

圖 2.3 Limited P/E cycles 6

圖 2.4 Flash Translation Layer 7

圖 2.5 Wear Leveling [1] 8

圖 2.6 Mapping Table [2] 8

圖 2.7 Garbage Collection [3] 9

圖 2.8 Open Channel SSD [4] 10

圖 2.9 單位大小 11

圖 3.1 Garbage Collection 處理未分群資料 12

圖 3.2 Garbage Collection 處理有分群資料 13

圖 3.3 設計方法 14

圖 3.4 準備四個 Line 15

圖 3.5 熱門資料與冷門資料 16

圖 3.6 判斷 Request 分群 17

圖 3.7 計算新平均值 17

圖 3.8 將分群資訊放入 Ring Buffer 的單位資料結構 (Entry) 18

圖 3.9 從數個 Entry 取出 Line 之後平均 19

圖 3.10 切換 Line 之後 Allocation，最後傳給 Open Channel SSD 20

圖 4.1　Garbage Collection 次數比較圖 (單位：次數) 23

圖 4.2　寫入資料量比較圖 (單位：GiB) 24

圖 5.1　Entry 代表的資料可寫入至對應的 Line 26

第一章　緒論

1.1　研究背景與動機

隨著整個市場對於 SSD (Solid State Drive) 的需求日漸增加，現今的技術也越朝著把更多的資料塞進 SSD 裡面。也意味著對於容量的需求急速成長。但是 SSD 的諸多缺點也因此而越來越明顯，寫越多的資料進去，每單位的壽命就越少，資料錯誤的可能性也越高。也就需要越來越多的技術來維護 SSD 的壽命以及資料的正確性。

Open Channel SSD 就是在這樣的需求下誕生了，這項技術嘗試把某些以往給 SSD 控制器管理的技術交給 Host 管理，既可以減輕 SSD 控制器的負擔，還能讓 Host 管理哪些是可能之後還會用到的資料，進而只將這些資料存在記憶體，達到減少記憶體使用量及提升快取效率的目的。

資料通常可能會分為兩種，一種是熱門資料 (Hot Data)，通常很快就會被覆寫，例如暫存檔或是存在硬碟的虛擬記憶體；另外一種是冷門資料 (Cold data)，會存放很久才有可能會更改，例如照片，影片，遊戲檔案等。現行的 Open Channel SSD 沒考慮到資料特性，目前是以循序寫入為主，這樣極有可能將 Hot Data 及 Cold Data 擺在一起，導致之後後要整理資料時需要花費較多的資源來運作。

相較於 SSD Controller，Host 的資源很豐富，有很多記憶體以及強大 CPU，於是我們把劃分資料的工作放進 Open Channel SSD 的核心模組，希望可以透過冷熱分群的方式，將經常更改的熱門資料與不常更改的冷門資料分開，進而改善 Garbage Collection 的效率。

1.2 研究目標

Open Channel SSD 是一種與作業系統互相合作的一種 SSD [5]，而 LightNVM 則是在 Linux 核心模組中專門管理 Open Channel SSD 的模組 [6]，此模組目前是以循序寫入的模式實作，不過並沒有將資料分類，再依據各分類改變寫入位置，而我們認為可能還有改善效率的空間，因此本論文將針對此模組修改以增進效能。

首先我們紀錄過往 Request 的大小，並取平均值，以此為依據將資料分為四個等級，由熱門資料 (Hot Data) 到冷門資料 (Cold Data)，再判斷當下收到的寫入要求位於哪個等級，最後決定將這次的要求寫入到最佳的位置，以達到最佳的效率。

1.3 論文組織架構

本論文共有五個章節，第二章將介紹背景知識及使用工具，第三章介紹如何劃分冷熱資料的演算法之設計，其中介紹了欲修改的 LightNVM 模組運作模式，並且介紹如何將設計加入至 LightNVM 模組中，第四章將以實際效能測試工具測試 Garbage Collection 效率，並且與原本無修改的 LightNVM 比較，第五章會分析實驗結果以及未來研究方向。

第二章　背景知識

2.1　SSD

SSD，全名：Solid State Drive，中文：固態硬碟。相較於傳統硬碟 HDD (Hard Disk Drive)，SSD 改善了噪音、震動、讀寫速度慢，容易因震動外力等因素而毀損等等的問題；SSD 也不需要在資料分散時，藉由重組硬碟磁區來減少讀取時間。而重點是，SSD 作為儲存裝置，還具有這些優點：[7]

- 存取時間固定，不像 HDD 有不穩定的搜尋時間
- 體積較小，適合放在筆記型電腦等行動裝置之中
- 少了機械結構，所以不會因機械故障導致毀損

而這些都是因為 SSD 採用了 NAND FLASH Memory。[8]

2.1.1　NAND FLASH Memory

是電子抹除式可程式化唯讀記憶體 (Electrically-Erasable Programmable Read-Only Memory，簡稱 EEPROM 或 E2PROM) 的其中一種，通常為 SSD、記憶卡、以及隨身碟的主要材料元件，可透過電子方式多次複寫。

這種元件的基本單位為 Cell，而依據每個 Cell 可儲存的 bit，可分為：[9]

- **SLC**：Single-level Cell 一個 Cell 存一個位元
- **MLC**：Multi-level Cell 一個 Cell 存兩個位元

- **TLC**：Triple-level Cell 一個 Cell 存三個位元
- **QLC**：Quad-level Cell 一個 Cell 存四個位元

不過，所有的 NAND FLASH Memory 均具有以下特性：

- Asymmertic Program/Erase unit
- Erase before write
- Limited P/E cycles

以下將逐一介紹這些特性。

2.1.2 Asymmertic Program/Erase Unit

一個 Page 由多個 Flash Cell 所組成，而數以百計的 Page 會組成一個 Block。而 Asymmertic Program/Erase Unit 指的意思是，當要讀取或是寫入時，可以 Page 為單位寫入，但是如果要 Erase 的時候，一次只能以 Block 為單位，無法以 Page 為單位 Write/Read 跟 Erase 的大小不一樣，所以稱為 Asymmertic (非對稱)。(如圖2.1所示)

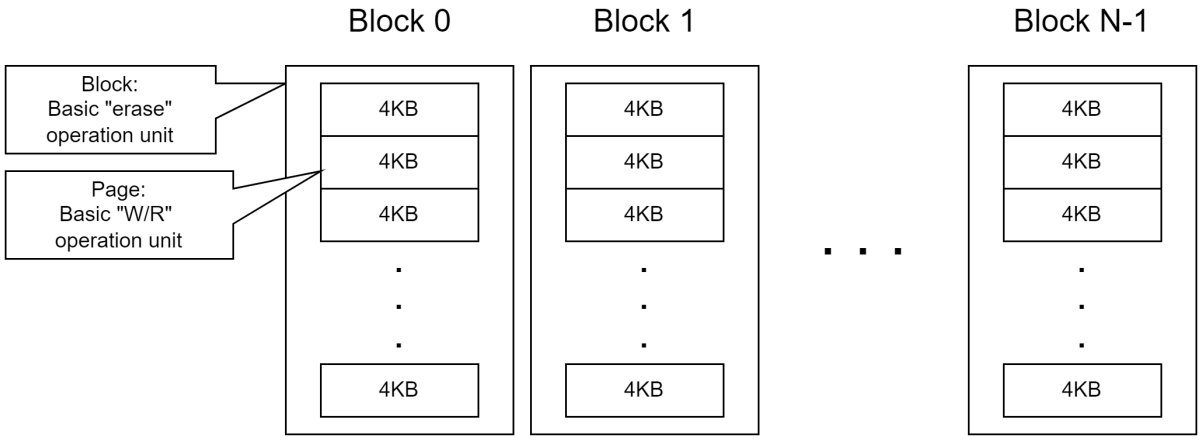

圖 2.1　Asymmertic Program/Erase Unit

2.1.3 Erase Before Write

一旦寫入資料到一個 Page 裡面，如果要更改 Page 裡面的資料，只能先清空 (Erase)，再將資料寫入。而且如2.1.2節所述，清空時只能清空比 Page 還大的 Block 才行。(如圖2.2所示) [10]

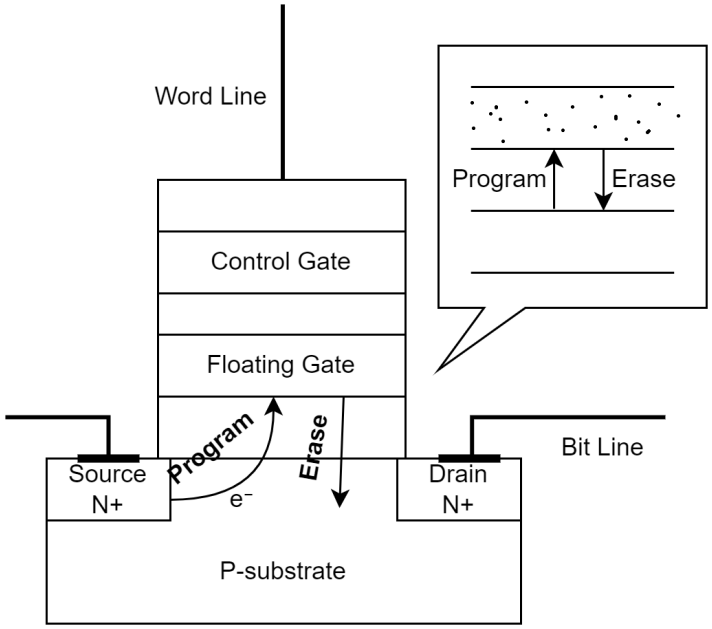

圖 2.2　Erase Before Write

2.1.4 Limited P/E Cycles

Limited Program/Erase Cycle，簡稱 Limited P/E Cycle，又被稱為 Wear Out。每次寫入會為用來儲存電子的 Flash Memory Cell 帶來不可逆的損害，原因是每次電子出入 Cell ，Floating Gate 旁邊的 Oxide layer 就會越來越薄 (如圖2.3所示)，最後薄到無法鎖住電子，整個 Cell 就會毀損 [11]。而隨著 Flash Memory 的製造商想要把越多的位元塞進一個 Cell 裡面，這些 Cell 的壽命就越來越少，例如 MLC，TLC，QLC 等等。

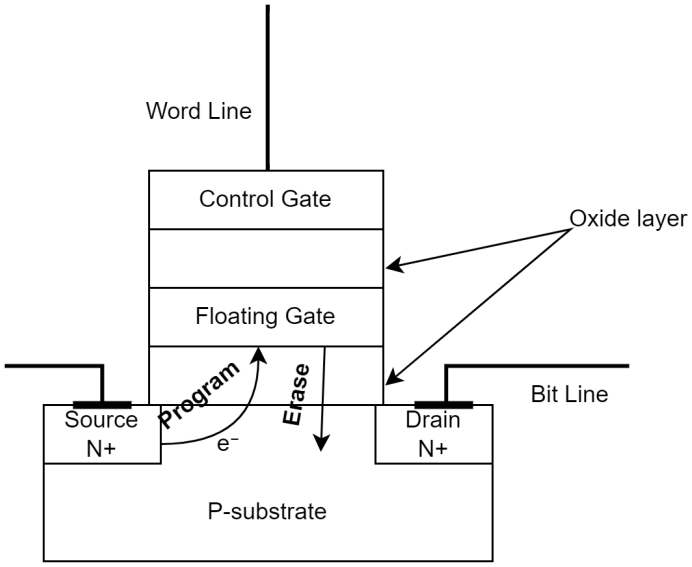

圖 2.3　Limited P/E cycles

2.2　FTL

為了降低2.1.2節至2.1.4節所提到的特性所造成壽命銳減的問題，SSD 的製造商通常都會將 FTL 導入至 SSD 裡面 (如圖2.4所示)。FTL 的全名為 Flash Translation Layer，負責針對 Flash Memory 特性而產生的技術，並模擬成一個普通的硬碟，讓作業系統管理 SSD 與傳統硬碟 (HDD) 的方式一致。而其中的技術通常包含：[12]

- Wear Leveling
- Mapping Table
- Garbage Collection

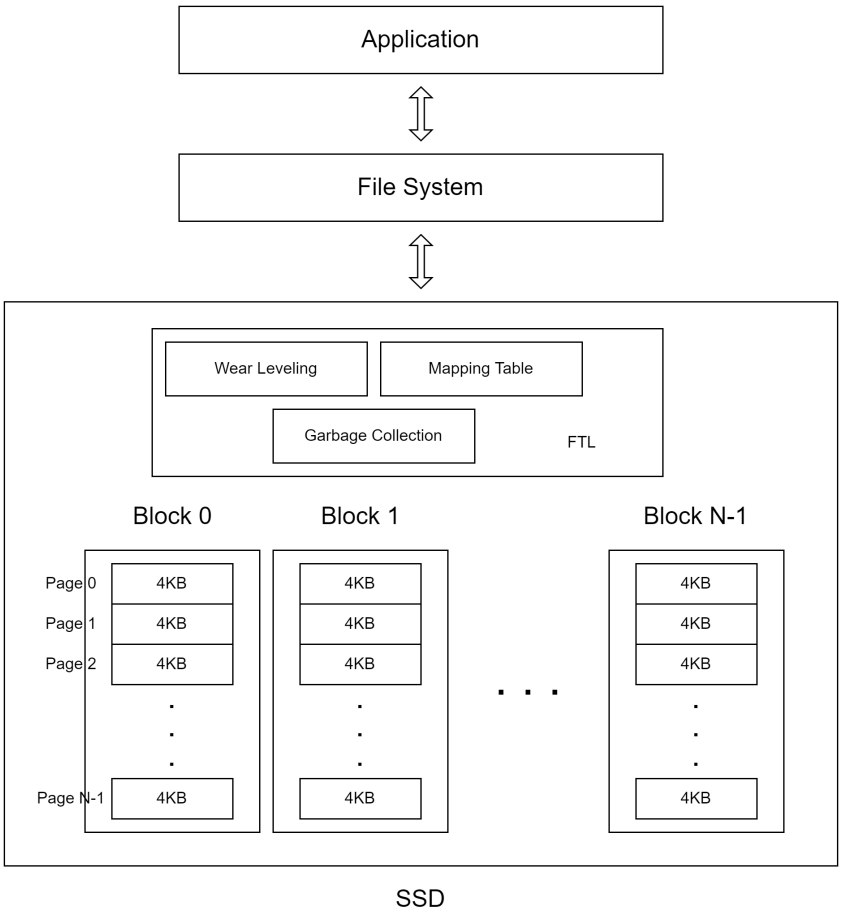

圖 2.4　Flash Translation Layer

以下將逐一介紹這些技術。

2.2.1　Wear Leveling

因為2.1.4節所提到的問題，如果每次都寫入同一個位置會導致 Cell 的壽命大幅衰減，因此為了延長每個 Cell 的壽命，寫入時需要從壽命比較長的 Cell，也就是當下寫入次數較少的 Cell 開始寫入，以平衡整體 SSD 的壽命 (如圖2.5所示)。所以 SSD 會記錄每個 Cell 的存取次數，來決定下次要寫入到哪裡。[13]

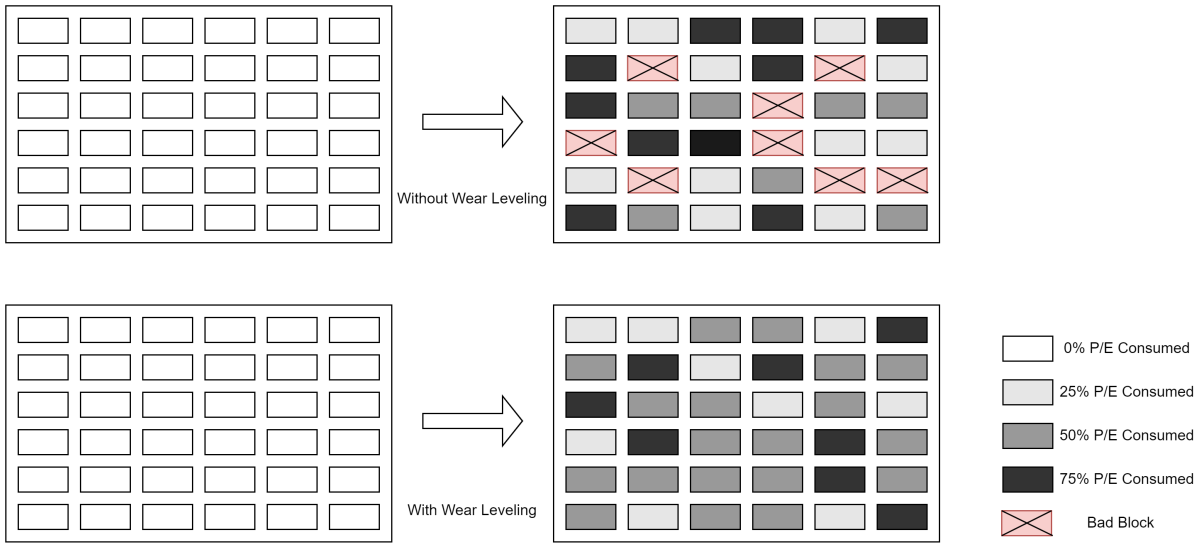

圖 2.5　Wear Leveling [1]

2.2.2　Mapping Table

由於2.1.3節所提到的 Erase Before Write 問題，因此在 SSD 會先將接收到的資料暫存，之後寫到其他區域，這樣可以減少寫入的延遲；同時為了讓 Host 覺得使用跟 SSD 與傳統硬碟的方式一致，所以增加轉換表，紀錄 Host 傳給 SSD 的 Logical Address 所對應的實際 Physical Address(如圖2.6所示)。通常會跟 2.2.1 所提到的 Wear Leveling 結合。

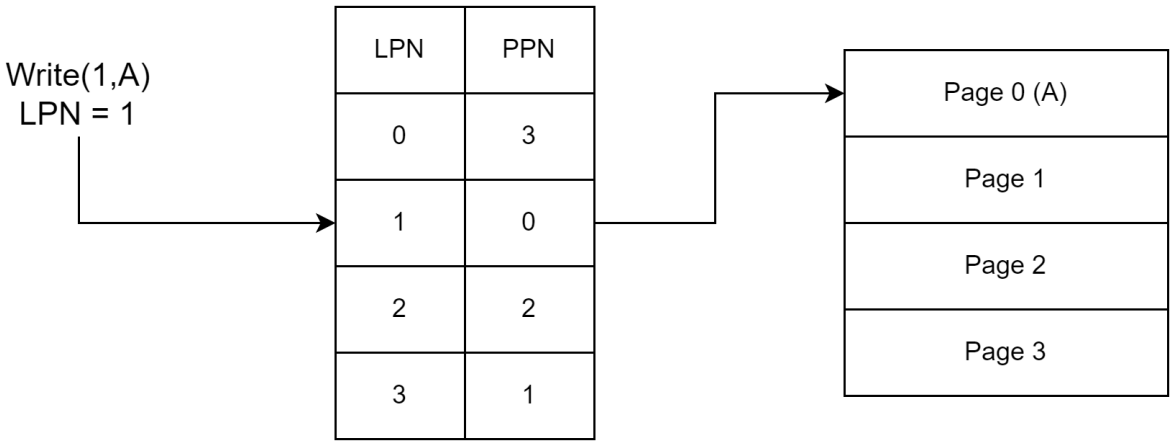

圖 2.6　Mapping Table [2]

2.2.3 Garbage Collection

由於2.2.2節所提到的功能，將資料寫到別的位置時，原本位置的 Block 就會有某些資料被視為無效，但還是有些資料是有效的，這時候就需要讀出來放到其他 Block 裡面，才可以將原本位置的 Block 清空，繼續作為空的 Block 寫入 (如圖2.7所示)。由於清除資料時的延遲較高，FTL 通常都會挑選比較空閒的時候才做這些搬移資料、清空的動作。

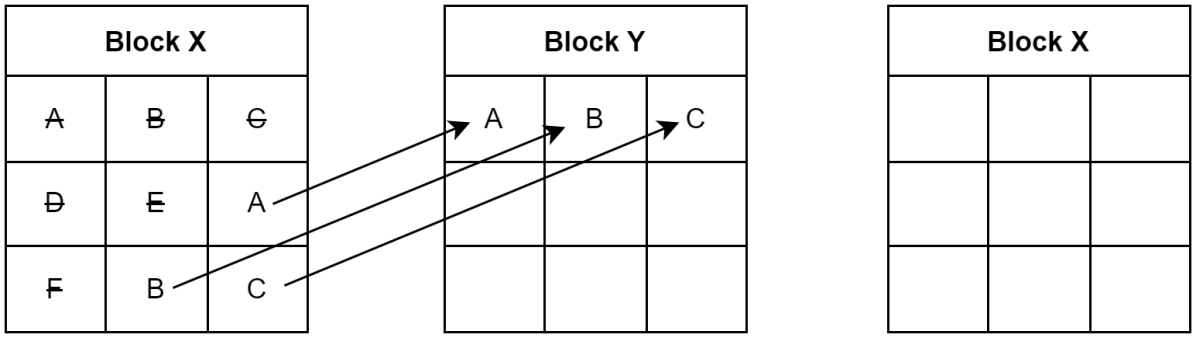

圖 2.7　Garbage Collection [3]

2.3 Open-Channel SSD

雖然 SSD 已經有了上述技術來管理以及延長壽命，不過由於這些功能都位於 SSD 內部的 FTL，Host 不知道這些資訊。而在雙方沒有溝通的狀況下，SSD 其實不知道 Host 想要的資料是什麼，容易造成重複讀取或是效率下降的問題，例如 SSD 可能會 Caching 到 Host 不需要的資料等問題。而 Open-Channel SSD 開啟了一個可以利用 Host 資訊增進存取效能的可能性。

這種 SSD 將下列三種功能交給 Host 管理 (如圖2.8所示)：

- Mapping Table
- Garbage Collection

- 部分的 Wear Leveling

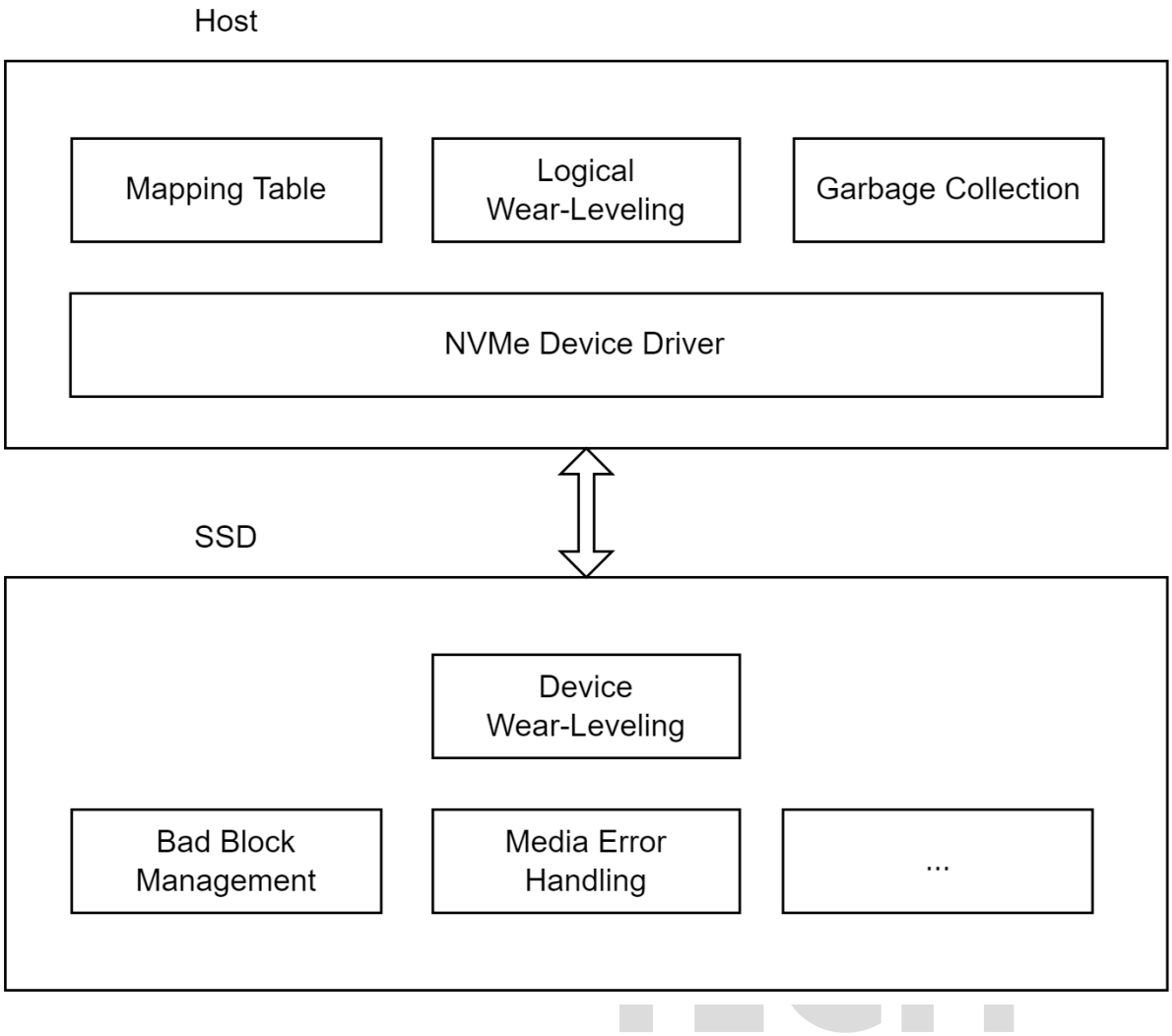

圖 2.8　Open Channel SSD [4]

2.4 LightNVM

LightNVM 是 Linux 之中用來管理 Open Channel SSD 的一個 Kernel Module，來實作上述交給 Host 管理的功能，可讓 Linux 系統正確辨識並能夠用控制一般 SSD 的方式對 Open Channel SSD 提出讀取、寫入、刪除等要求。本論文也是修改此模組來實作我們所需的功能。

2.4.1 Line

一般來說，SSD 內部的大小單位可以分為下列幾種，由大而小依序為:

- Channel
- Lun
- Block
- Page
- Sector

而 Line 會由每個 Lun 之中的一個 Block 所組成，如圖2.9所示。而在 LightNVM 之中，寫入以及 Garbage Collection 均採用此單位作為管理的主要架構，本論文提出的修改部分也與此有關。

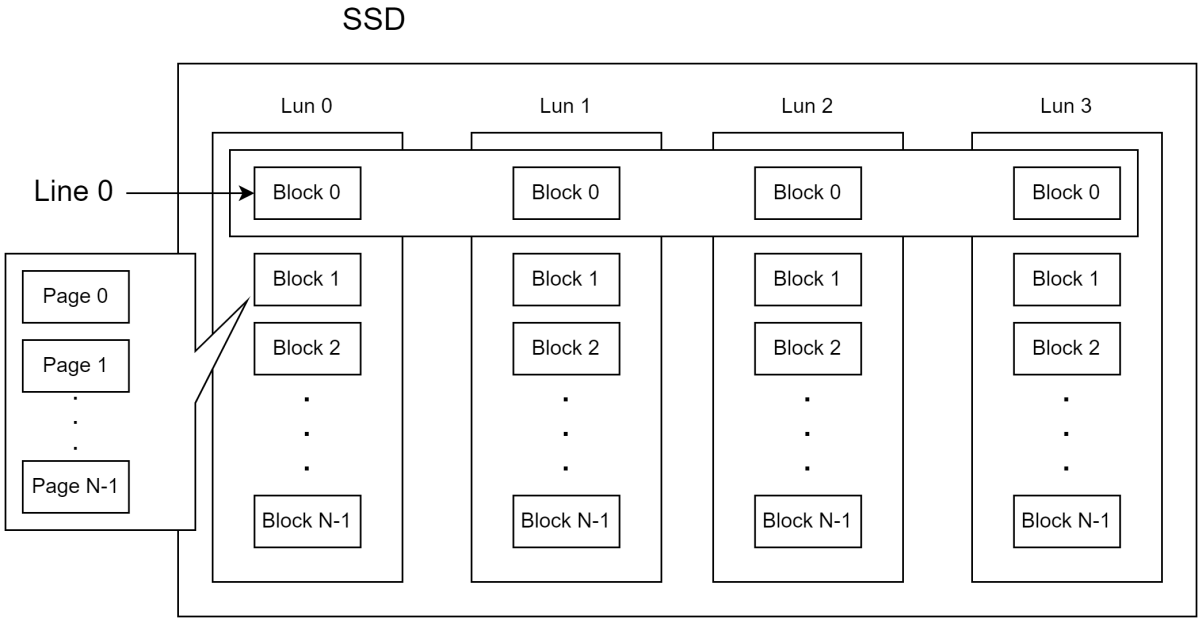

圖 2.9　單位大小

第三章　研究設計與實作

Garbage Collection 在處理沒有依據特性分開擺放的資料時，所需的動作較多、時間較長，所造成的壽命損耗也比較多。如圖3.1所示，灰色為無效資料，打勾的方塊為有效資料，若此時需要 2 個 Block 儲存，則需執行下列動作，才可完成 Garbage Collection：

- 複製 Block 1 與 Block 2 中的有效資料
- 將有效資料寫入到 Block 3
- 清空 Block 1 與 Block 2

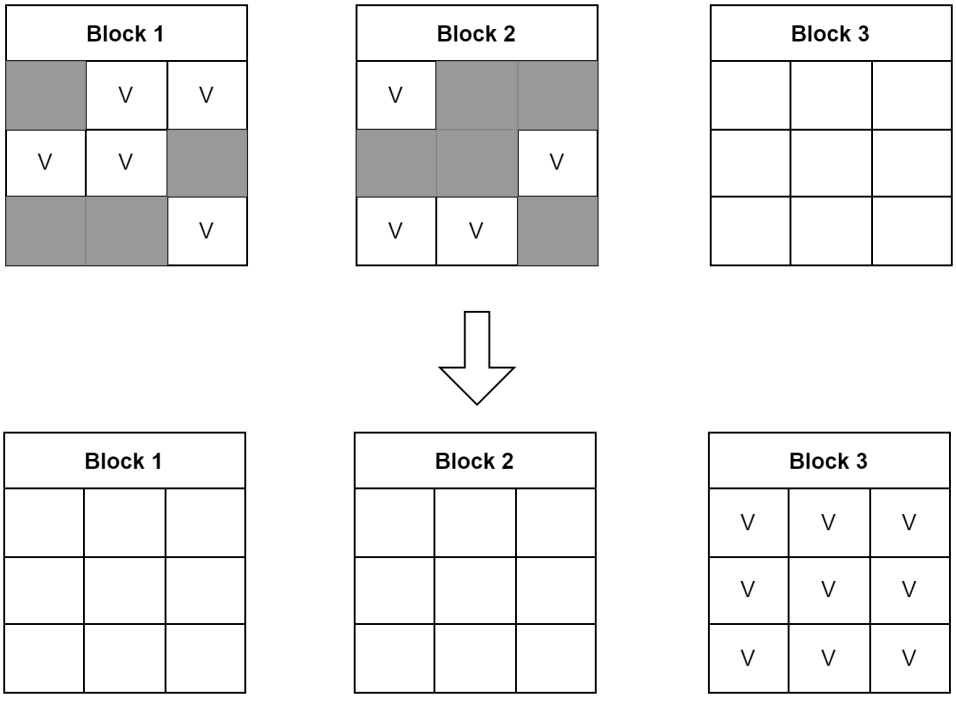

圖 3.1　Garbage Collection 處理未分群資料

而 Garbage Collection 在處理有依據特性分開擺放的資料時，所需的動作較少、時

間較短，所造成的壽命損耗也比較少。如圖3.2所示，灰色為無效資料，打勾的方塊為有效資料，若此時需要 2 個 Block 儲存，只需執行下列動作，即可完成 Garbage Collection：

- 清空 Block 1

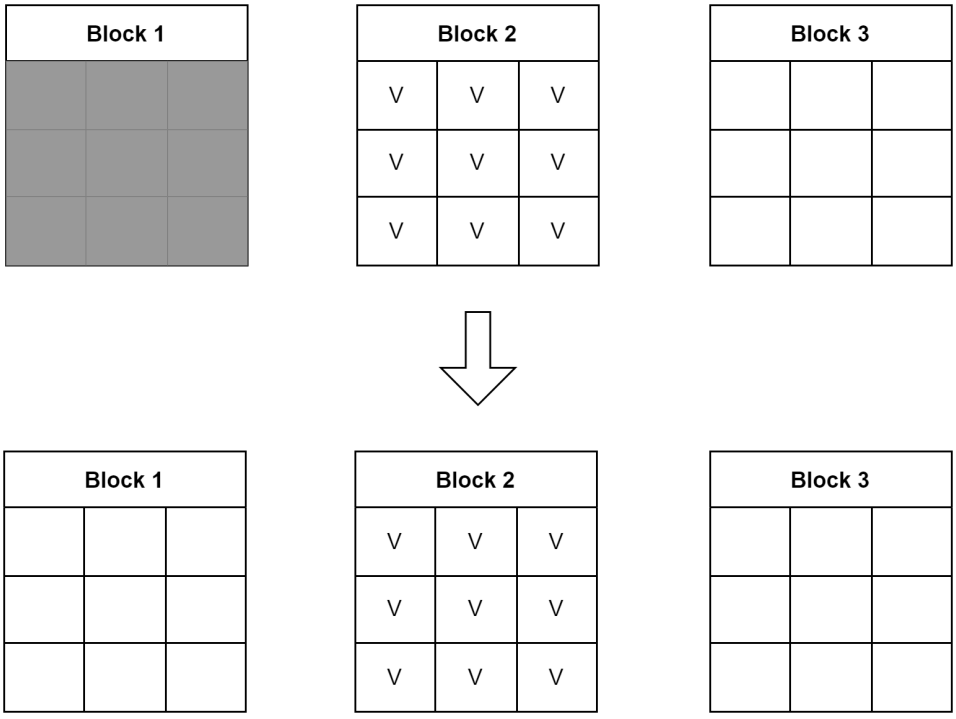

圖 3.2　Garbage Collection 處理有分群資料

因此，本論文為提升 Garbage Collection 的效率，以及降低 Garbage Collection 的次數，而提出此設計 (如圖3.3所示)。將資料依據熱門到冷門的程度分開擺放，而熱門資料有可能較快被視為無效資料，而因為各自集中的關係，Garbage Collection 所需的動作變少，影響範圍也變小，效率也因此提升。

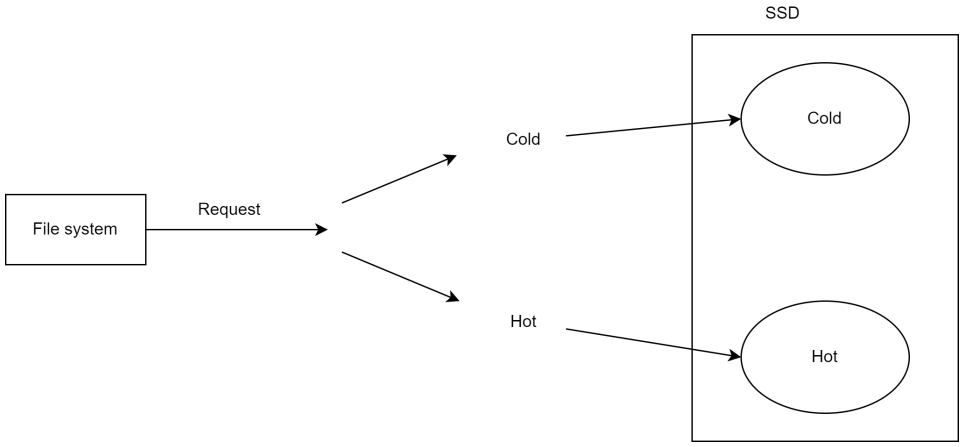

圖 3.3　設計方法

透過將此方法與 LightNVM 既有的寫入流程結合，如此一來，LightNVM 在寫入時，會透過這個方法挑選寫入位置，以達成提升效率的目的。

修改大致可分為三部分，第一部分為初始化，讓原本 LightNVM 從準備一個 Line 變成準備四個 Line，一部分為 LightNVM 處理檔案系統的 Request，將資料存入 Ring Buffer；另一部分為 LightNVM 的 Write Thread 會被喚醒，檢查 Ring Buffer 並將 Entry 中紀錄的資料取出，寫到真正的 SSD 之中，本章節會解釋修改這三部分的哪些環節。

3.1　初始化時修改為準備四個 Line

在初始化時，LightNVM 會先準備好一個 Line，後續有檔案系統傳遞給 LightNVM 寫入要求時，就會從這個 Line 開始寫入，滿了之後會繼續挑選下一個有空間的 Line 來使用，因此我們為了將資料分開寫入，要修改為在初始化時先準備好四個 Line，以便之後的寫入分群使用 (如圖3.4所示)。

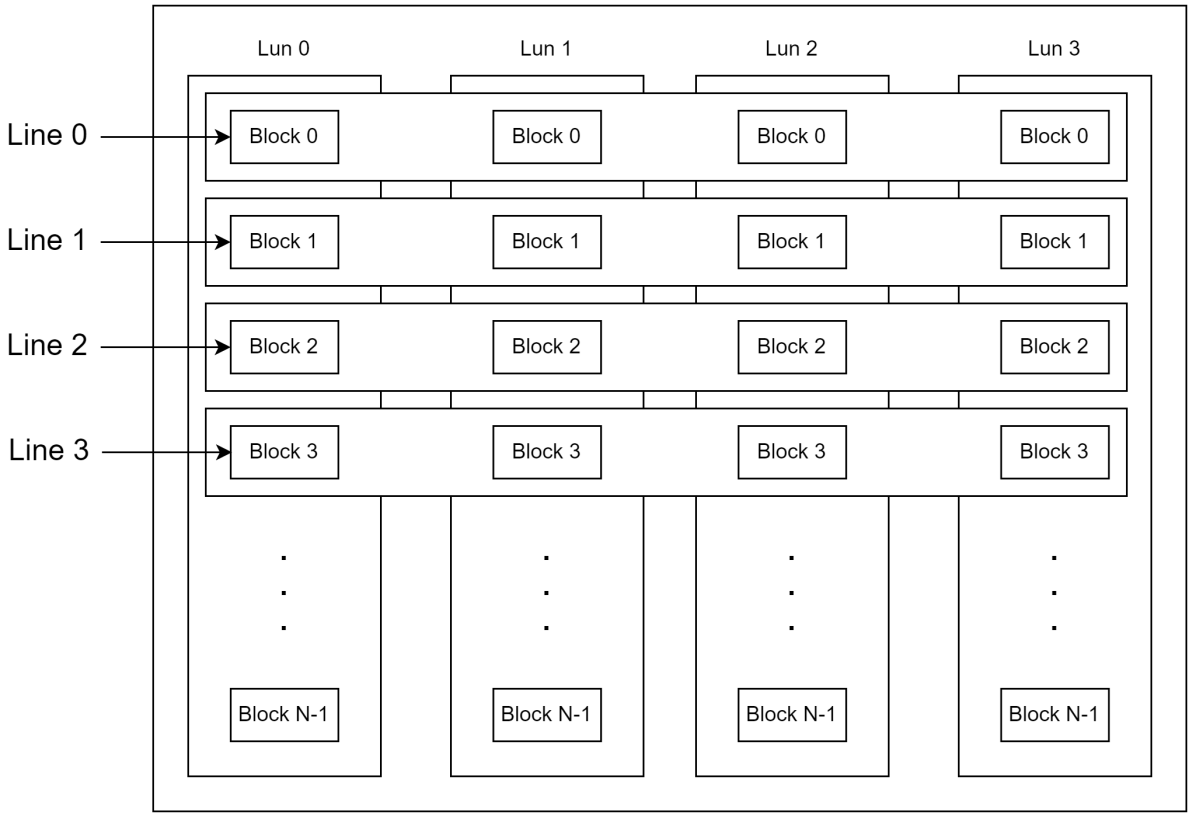

圖 3.4　準備四個 Line

3.2　處理檔案系統要求之流程

檔案系統提出 Request 要求之後，LightNVM 會對 Request 拆解，儲存成 LightNVM 自己容易處理的形式，但是在拆解的同時，會損失一些資訊，所以首先我們要先在拆解 Request 之前把 Request 的大小先從 Request 的資料結構 bio (Block I/O) [14] 擷取出來，用大小來判斷這個 Request 的資料屬於哪一個分群，並將資訊塞入 LightNVM 用來儲存 Request 的地方 - Ring Buffer 之中，最後計算平均值，給下次 Request 傳進時使用。[6]

3.2.1 以平均值劃分熱門資料及冷門資料

首先為了劃分熱門及冷門資料，我們將過去一萬筆 Request 大小的平均值當作基準，劃分四個等級，平均值的 0% - 50% 為最熱門的資料，平均值的 50% - 100 % 為第二熱門的資料，平均值的 100% - 150% 為第二冷門資料，平均值的 150% 以上為最冷門的資料。並且由熱門到冷門所對應到的 Line 為 Line 0 到 Line 3(如圖3.5所示)。

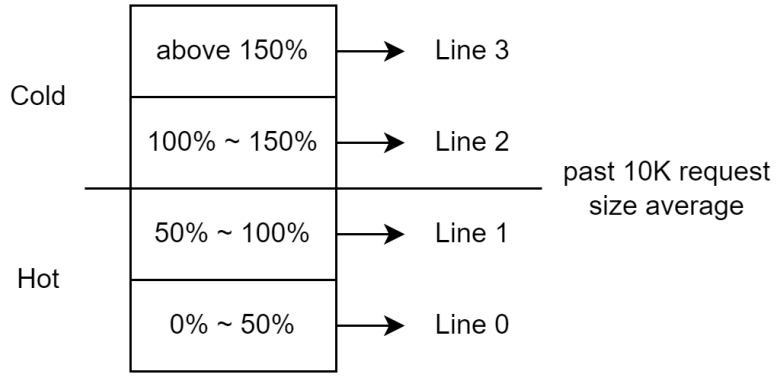

圖 3.5 熱門資料與冷門資料

3.2.2 擷取 Request 大小並判斷分群

在得到 Request 大小時，我們需要從檔案系統傳給 LightNVM 的資料結構 bio 之中，找到 Request 大小的值；之後再以 Request 大小與平均值，判斷當下的 Request 屬於哪一類 (如圖3.6所示)。

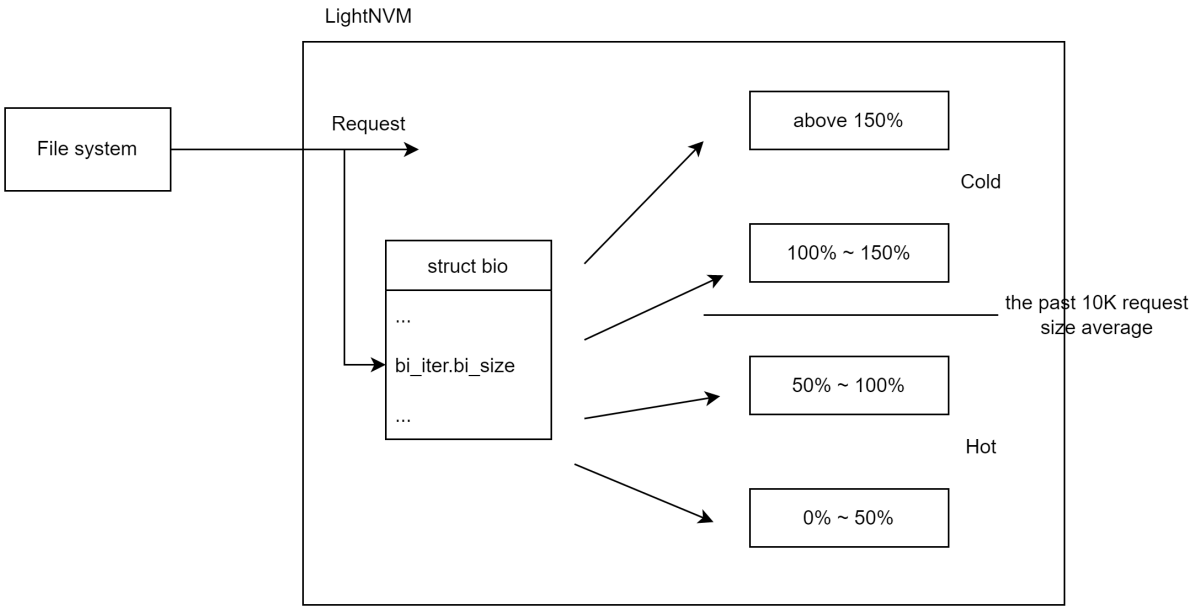

圖 3.6　判斷 Request 分群

3.2.3　計算平均值

在判斷完分群之後，我們需要拿剛剛得到的 Request 大小與平均值計算出新的平均值；由於我們需要前一萬筆 Request 大小的平均值，所以得到大小之後將其乘以 0.0001 倍再與我們加入 LightNVM 的平均值乘以 0.9999 倍之後相加，之後下一次檔案系統傳送 Request 進來時，就可以繼續用來判斷分群 (如圖3.7所示)。

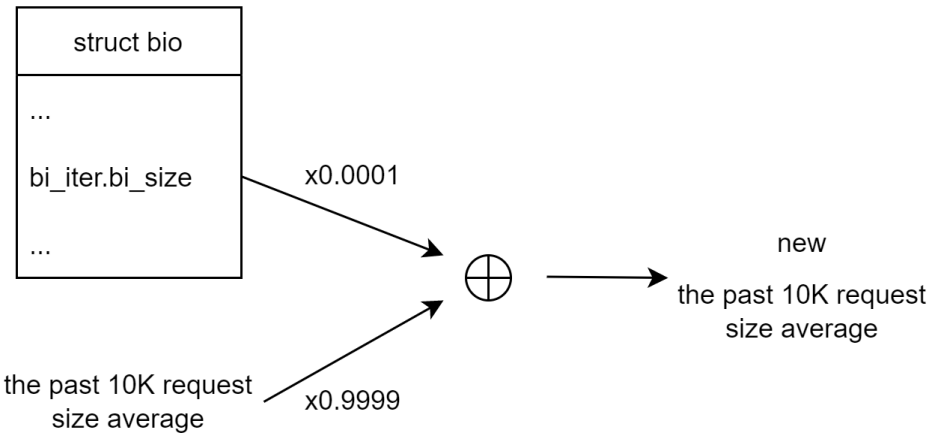

圖 3.7　計算新平均值

3.2.4 將 Request 所屬分群的資訊放入 Ring Buffer 中

LightNVM 在接到檔案系統的 Request 之後，會將 Request 的資料拆解成數個 Page，每個 Page 都會記錄成 Ring Buffer 一個 Entry 之中，之後便會喚醒 LightNVM 背景的 write thread，從 Ring Buffer 存取剛剛放進去的 Entry，來得知寫入的資料位於 Host 記憶體，我們在這個 Entry 之中加入要存入哪個 Line，也就是冷熱分群的資訊，以便之後的 write thread 使用 (如圖3.8所示)。

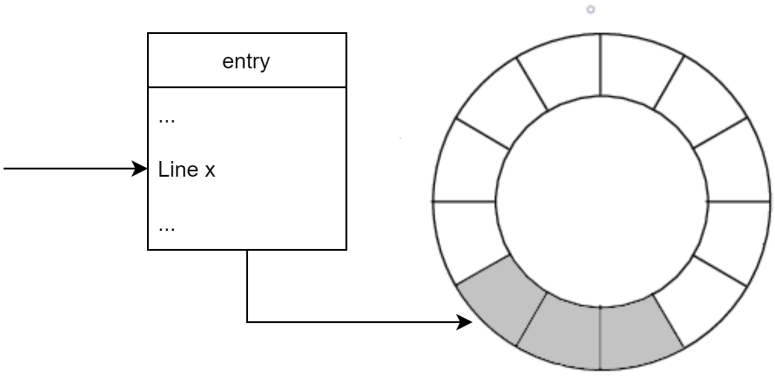

圖 3.8　將分群資訊放入 Ring Buffer 的單位資料結構 (Entry)

3.3 實際寫入之過程

LightNVM 的 Write Thread 被喚醒之後，會從剛剛紀錄到 Ring Buffer 之中的 Entry 得知要寫入的資料在哪裡，而我們也跟著得知剛剛我們放入的資訊，也就是每個 Entry 所屬的冷熱分群，接著我們將所有當次所提取的 Entry 的所屬冷熱分群總和之後取平均，得到的平均值就是我們最後寫入的 Line，最後我們將 Line 切換到我們的目標之後，LightNVM 就會根據我們所提供的 Line 來做 allocation，最終傳給 Open Channel SSD。

3.3.1 從 Ring Buffer 中蒐集 Entry

每次 LightNVM 從 Ring Buffer 之中抽取出的 Entry 數量不一定一樣，假設這次會取四個 Entry 的資料，分別所屬為 Line 3、2、1、2，那最後平均值是 2(如圖3.9所示)。

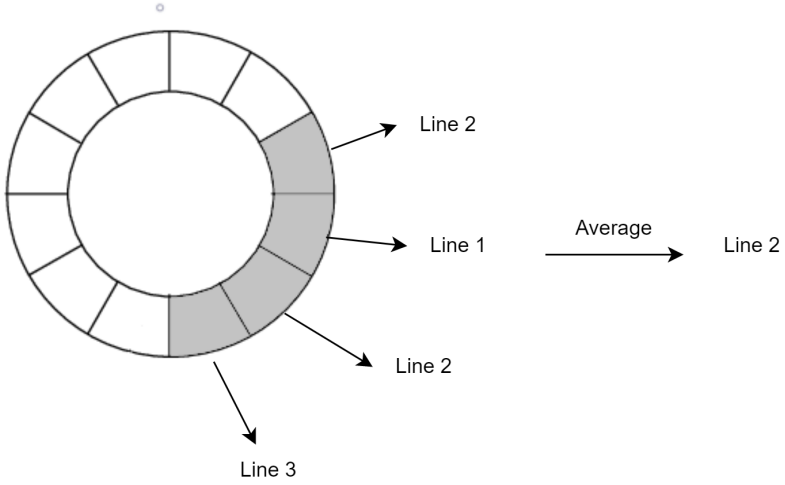

圖 3.9　從數個 Entry 取出 Line 之後平均

3.3.2 切換 Line 之後傳送給 Open Channel SSD

按照剛剛計算的平均值切換 Line 之後，LightNVM 會根據我們指定的 Line 做 Allocation，最後將資料以及位置往下傳給 Open Channel SSD(如圖3.10所示)(圖例延續 3.9)。

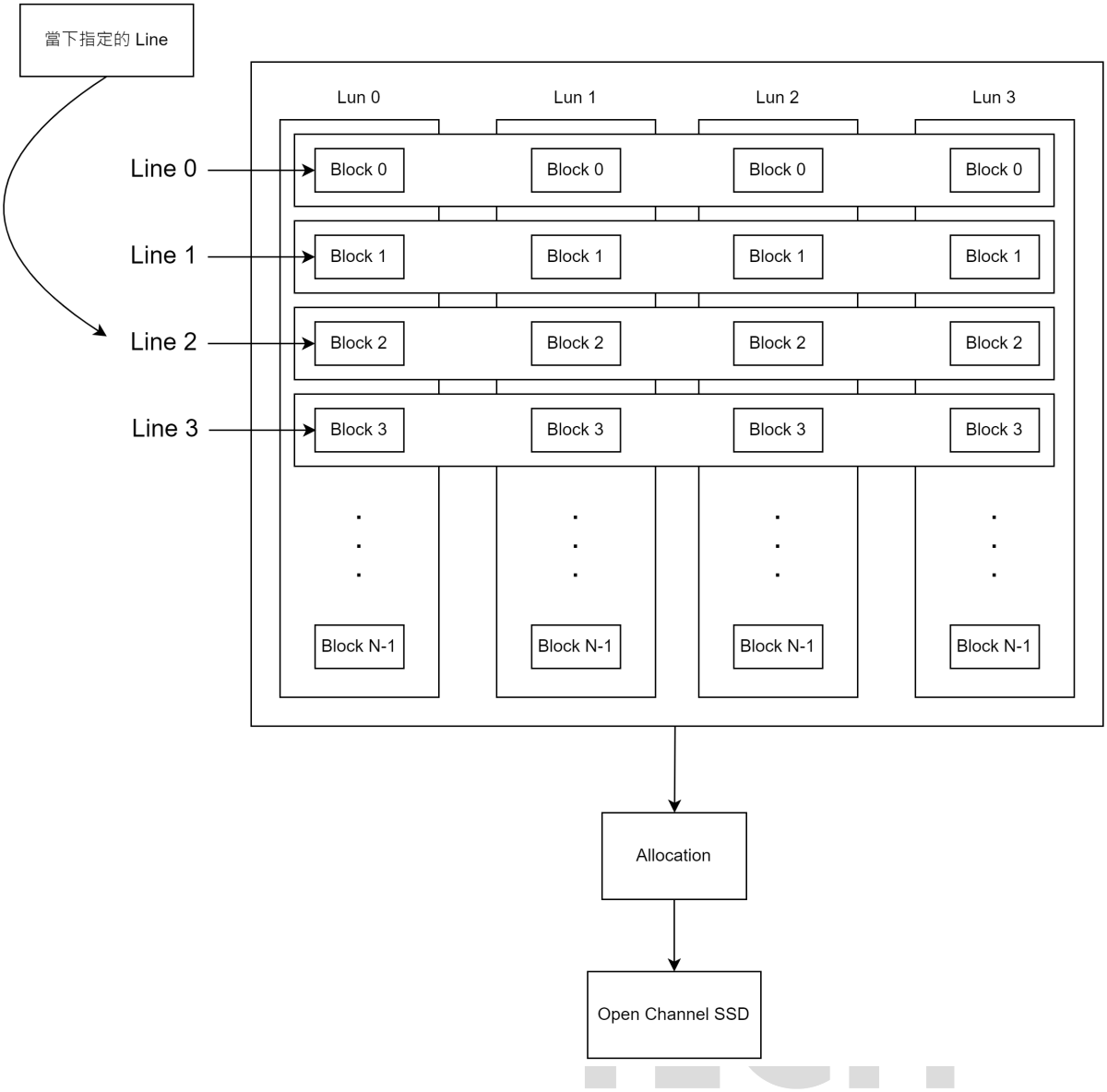

圖 3.10　切換 Line 之後 Allocation，最後傳給 Open Channel SSD

第四章　案例實驗

4.1　實驗環境與參數

此次實驗採用 FEMU 作為 SSD 模擬器，FEMU 是一個由 QEMU 修改而成的模擬器 [15]，支援模擬下列幾種 SSD

- NoSSD
- Black-box SSD
- Open-Channel SSD
- Zoned-Namespace (ZNS) SSD

實驗環境參數如同下列所述：

- Linux Kernel 5.10.83
- 4 核心
- 5G 記憶體
- 模擬 4G 硬碟

4.2　實驗規劃

本論文將以在 Linux 上常用的硬碟效能檢測工具 FIO 來實測效能差異。實驗將分為兩個部分，第一部分是測試在同樣的資料量底下，採用原版的 LightNVM 與我們修改過的 LightNVM 所產生的 Garbage Collection 次數差異，第二部分為在同樣時間下，採

用原版的 LightNVM 與我們修改過的 LightNVM 所寫入的資料量差異。以這兩種實驗來確認採用本論文的設計實作是否對於 Garbage Collection 效率有所幫助。

4.3 實驗一：Garbage Collection 次數比較

表4.1跟圖4.1為在同樣資料量情況下，使用原版以及修改過後的 LightNVM 進行 Garbage Collection 次數之比較，從表中可見隨著資料量增加，兩邊所產生的 Garbage Colleciton 次數也隨之增加，而我們所修改的 LightNVM 版本產生的次數與原版 LightNVM 比較起來，減少的比例由 4G 到 64G 依序為 44%、61%、65%、66%、68%，平均大約減少了 60.8%；稍微要注意的是，由於 LightNVM 本身架構，這邊的次數代表的是有多少 Line 被 Garbage Collection 處理過。

表 4.1　在同樣資料量情況下，Garbage Collection 次數比較

	使用原版 LightNVM 的 GC 次數	使用修改版 LightNVM 的 GC 次數
4G	25	14
8G	159	62
16G	482	170
32G	1144	392
64G	2493	812

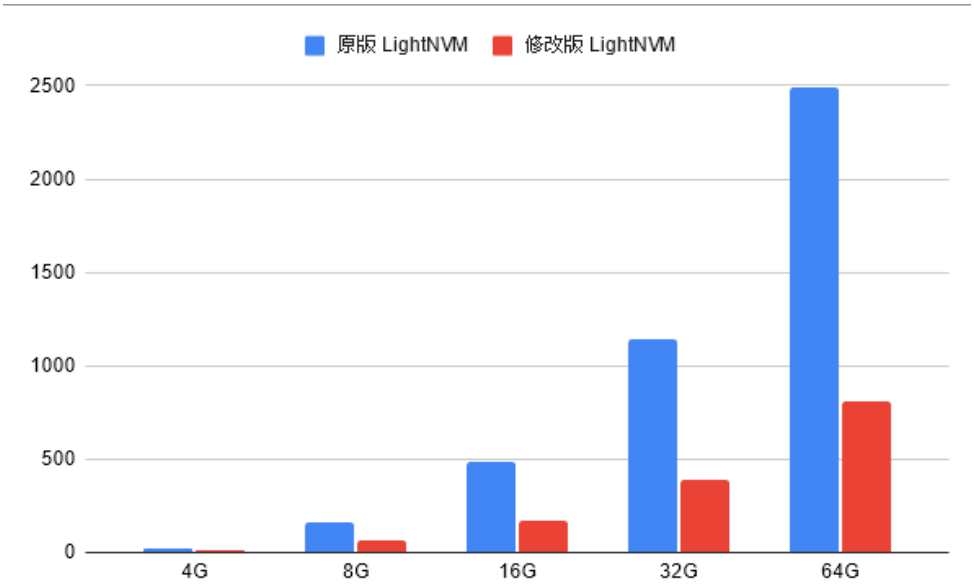

圖 4.1　Garbage Collection 次數比較圖 (單位：次數)

4.4　實驗二：寫入資料量比較

表4.2跟圖4.2為在同樣時間情況下，使用原版以及修改過後的 LightNVM 進行資料寫入量之比較，從表中可見隨著時間增加，兩邊所產生的寫入資料量也隨之增加，而我們所修改的 LightNVM 版本產生的次數與原版 LightNVM 比較起來，增加的比例由 1 分鐘 到 4 分鐘 依序為 10.3%、4.8%、1.6%、5.1%，平均大約增加了 5.45%。

表 4.2　在同樣時間情況下，寫入資料量比較

	使用原版 LightNVM 的 寫入資料量	使用修改版 LightNVM 的寫入資料量
1 分鐘	193GiB	213GiB
2 分鐘	431GiB	452GiB
3 分鐘	669GiB	680GiB
4 分鐘	862GiB	906GiB

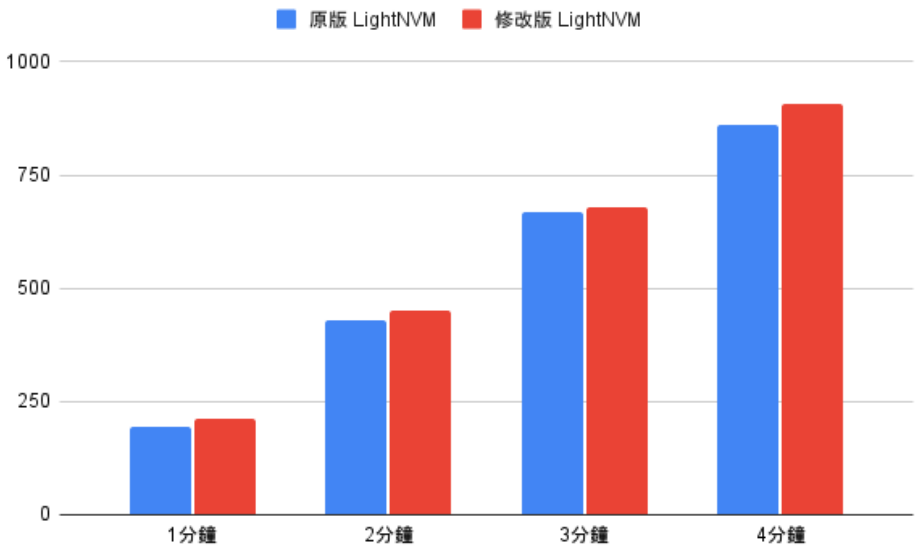

圖 4.2　寫入資料量比較圖 (單位：GiB)

第五章　結論與未來展望

5.1　結論

本論文修改了 Linux kernel 之中 LightNVM 模組，調整寫入模式，讓 LightNVM 從原本不管資料的種類，只循序寫入的方式，改成可將熱門資料以及冷門資料分群的方式寫入 SSD，最後經實驗驗證的確有讓 Garbage Collection 的次數下降，提升 Garbage Collection 的效率。不過在寫入效能方面是幾乎沒有影響，經我們分析，如果 Garbage Collection 設計得宜，應該是要盡可能地不要妨礙讀取或寫入的任務，所以我們即使提升了 Garbage Collection 的效率，對於提升寫入效能的成效有限是合理結果。

5.2　未來方向

本論文修改 LightNVM 之設計仍有待改善之處，以下說明：

- **寫入模式之調整：**於3.3節的寫入過程，如果可以將 entry 的資料實際寫入所指向的 Line；而不是讓 LightNVM 用原本的挑選模式挑選完之後，將 entry 所指向的 Line 總和平均，再切換 Line。這樣應該可以更加提升效率 (如圖5.1所示)。

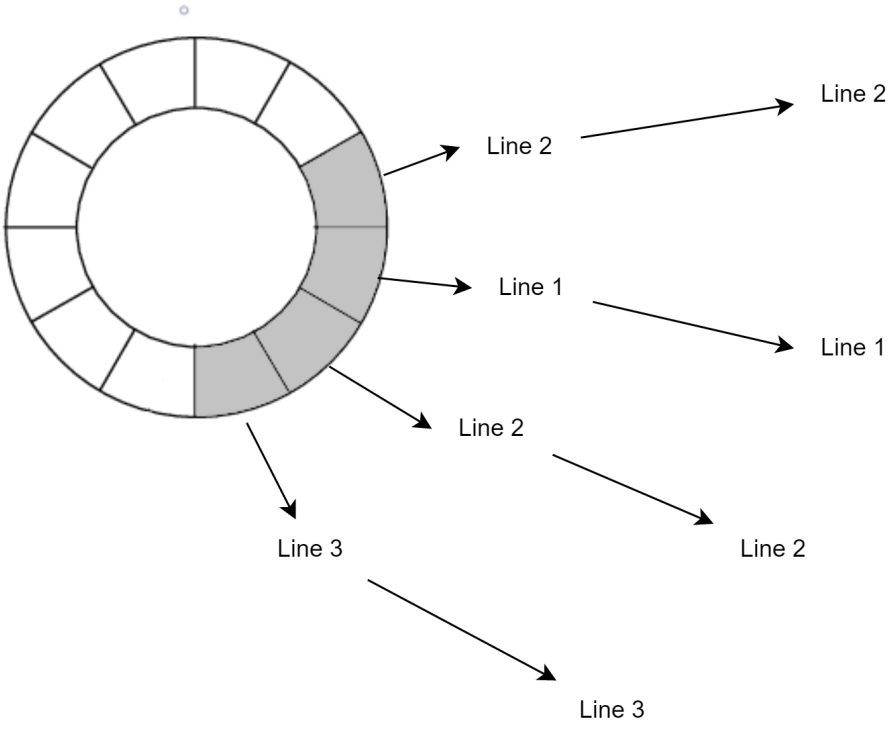

圖 5.1　Entry 代表的資料可寫入至對應的 Line

參考文獻

[1] SMART-Modular. Wear leveling. https://www.smartm.com/ch/technology/wear-leveling/.

[2] yunchih. Ssd-ftl. https://www.csie.ntu.edu.tw/~yunchih/docs/cs/ssd-ftl/.

[3] SSSTC Solid State Storage Technology Corporation. Garbage collection. https://ssstc.com/industrial-ssd-features/garbage-collection-ssd/.

[4] Western Digital Matias Bjørling. Introduction to open-channel/denali solid state drives. In *SDC 2018*, September 2018.

[5] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-channel ssd (what is it good for). In *Conference on Innovative Data Systems Research*, January 2020.

[6] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The linux Open-Channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association.

[7] Ssd 筆記 - 第二篇 ssd 結構與性能評定概述. https://www.owlfox.org/blog/2019-11-25-coding-for-SSD-part-2/.

[8] SSDFANS. *深入淺出 SSD：固態存儲核心技術、原理與實戰*. 機械工業, 2018.

[9] Yoshio Nishi. *Advances in Non-volatile Memory and Storage Technology*. Woodhead Publishing, 2014.

[10] Xin Li, Zhaoyan Shen, Lei Ju, and Zhiping Jia. Srftl: An adaptive superblock-based real-time flash translation layer for nand flash memory. In *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*, pages 332–339, 2014.

[11] Yung-Yueh Chiu, I-Chun Lin, Kai-Chieh Chang, Bo-Jun Yang, Toshiaki Takeshita, Masaru Yano, and Riichiro Shirota. Transconductance distribution in program/erase cycling of nand flash memory devices: a statistical investigation. *IEEE Transactions on Electron Devices*, 66(3):1255–1261, 2019.

[12] Jalil Boukhobza, Pierre Olivier, and Stéphane Rubini. A scalable and highly configurable cache-aware hybrid flash translation layer. *Comput.*, 3:36–57, 2014.

[13] Li-Pin Chang. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, page 1126–1130, New York, NY, USA, 2007. Association for Computing Machinery.

[14] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Trans. Comput. Syst.*, 32(2), jun 2014.

[15] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 83–90, Oakland, CA, February 2018. USENIX Association.