國立中興大學資訊科學與工程學系

碩士學位論文

指導教授：張軒彬 博士 Hsung-Pin Chang

研究生：黃聖穎 Sheng-Ying Huang

中華民國一百一十一年六月

**摘要**

為了改善SSD效能與壽命，SSD使用了一塊RAM-based write buffer來減少flash memory寫入次數。然而，由於write buffer容量有限，在面對隨機且大量的使用者request時，hit ratio仍然不甚理想，因此，本篇論文使用machine learning的方式，來提升write buffer的hit ratio。

在機器學習方面，為了彌補誤判所造成的overhead，在online將很久不用的資料進行demoting，使cold data漸漸地從write buffer被踢除。與此同時，引入host端的資訊，讓write buffer在挑選victim block時，透過host端的資訊，可以更準確地找出剔除的對象。

實驗結果顯示，AI本身就可以對write buffer hit ratio進行改善，套用host資訊之後，更能夠進一步提升write buffer hit ratio，提升整體效能。

**Abstract**

SSD lifetime is short because of its out-of-place-update feature. To improve SSD performance and lifetime, SSD used RAM-based write buffer, reducing flash memory write count.

However, the write buffer hit ratio is not good enough when facing random and large user write requests because write buffer capacity is limited. Therefore, this paper uses machine learning to improve the write buffer hit ratio. In machine learning, we fix misprediction by online demoting, which can kick cold data out of the write buffer, even if that data is currently in a late or mean queue. At the same time, we are also using host information to help us to select victim blocks. With host information, we can choose the victim block more precisely.

The experiment shows that the write buffer hit ratio can be improved even if only with the AI model, but the hit ratio can improve further when the AI model combines with host information. As a result, we can improve the hit ratio and overall performance with the AI model and host information.

目錄

[第一章 緒論 1](#_Toc103971104)

[1.1 簡介 1](#_Toc103971105)

[1.2 研究動機 1](#_Toc103971106)

[1.3 貢獻 1](#_Toc103971107)

[1.4 論文架構 1](#_Toc103971108)

[第二章 背景知識與相關研究 1](#_Toc103971109)

[2.1 SSD 1](#_Toc103971111)

[2.2 Flash Translation Layer 1](#_Toc103971112)

[2.3 Linux Flush 1](#_Toc103971113)

[2.3.1 定量Flush 1](#_Toc103971114)

[2.3.2 定時Flush 2](#_Toc103971115)

[2.3.3 Hint Queue 3](#_Toc103971116)

[2.4 write buffer管理方式 4](#_Toc103971117)

[2.4.1 FAB(Flash-aware-buffer management policy) 4](#_Toc103971118)

[2.4.2 BPLRU(Block Padding Least Recently Used) 5](#_Toc103971119)

[2.5 資料在SSD的擺放方式 6](#_Toc103971120)

[2.5.1 Page Striping 7](#_Toc103971121)

[2.5.2 Block Striping 7](#_Toc103971122)

[2.6 擺放方式相關研究 8](#_Toc103971123)

[2.6.1 BLAS 8](#_Toc103971124)

[第三章 Using LSTM combine with Host information for selecting write buffer victim block 9](#_Toc103971125)

[3.1 系統架構 9](#_Toc103971126)

[3.2 AI運作流程 10](#_Toc103971127)

[3.2.1 Benefit function 11](#_Toc103971128)

[3.2.2 Data collected 11](#_Toc103971129)

[3.2.3 Transform duration vale into duration label 11](#_Toc103971130)

[3.2.4 LSTM 11](#_Toc103971131)

[3.2.5 Demoting 11](#_Toc103971132)

[3.3 AI與hint 資訊做結合 12](#_Toc103971133)

[3.3.1 選擇victim block 13](#_Toc103971134)

[3.3.2 過早踢除 13](#_Toc103971135)

[4. 第四章 實驗結果與討論(以下是學姊的論文…..我還沒有寫到第四章) 13](#_Toc103971136)

[4.1 實驗環境 13](#_Toc103971137)

[4.2 實驗結果 13](#_Toc103971138)

[4.2.1 參數測試 13](#_Toc103971139)

[4.2.2 實驗結果 13](#_Toc103971140)

[5. 第五章 結論及未來工作 13](#_Toc103971141)

[6. 參考文獻 13](#_Toc103971142)

圖目錄

[圖 2‑1定量flush 5](#_Toc17463062)

[圖 2‑2定時flush 6](#_Toc17463063)

[圖 2‑3 SSD架構 7](#_Toc17463064)

[圖 2‑4 FAB架構 10](#_Toc17463065)

[圖 2‑5 BPLRU的架構 [3] 11](#_Toc17463066)

[圖 2‑6 Page Padding 11](#_Toc17463067)

[圖 2‑7 CLC的架構 12](#_Toc17463068)

[圖 2‑8 Page Striping 13](#_Toc17463069)

[圖 2‑9 Block Striping 14](#_Toc17463070)

[圖 2‑10 BLAS State [5] 15](#_Toc17463071)

[圖 2‑11 LPB的架構 [6] 16](#_Toc17463072)

[圖 2‑12 SUPA的架構 [7] 17](#_Toc17463073)

[圖 3‑1 系統架構圖 [8] 18](#_Toc17463074)

[圖 3‑2 主機端的提示資訊 19](#_Toc17463075)

[圖 3‑3 定量「flush」間隔與dirty page關聯 20](#_Toc17463076)

[圖 3‑4 dirty ratio 39% 時啟動預測 24](#_Toc17463077)

[圖 3‑5 時啟動預測 25](#_Toc17463078)

[圖 3‑6 發生replacement時啟動預測 26](#_Toc17463079)

[圖 3‑7 Hint Queue 28](#_Toc17463080)

[圖 3‑8 Situation 1 (具有page striping block的候選) 31](#_Toc17463081)

[圖 3‑9 Situation 1 (只有block striping block的候選) 31](#_Toc17463082)

[圖 3‑10 Situation 2 (具有page striping block的候選) 33](#_Toc17463083)

[圖 3‑11 Situation 2 (只有block striping block的候選) 33](#_Toc17463084)

[圖 3‑12 Situation 3 (具有page striping block的候選) 34](#_Toc17463085)

[圖 3‑13 Situation 3 (只有block striping block的候選) 35](#_Toc17463086)

[圖 3‑14 Write to each channel 36](#_Toc17463087)

[圖 4‑1 Hint Group參數 38](#_Toc17463088)

[圖 4‑2 Write Ratio參數 39](#_Toc17463089)

[圖 4‑3 Write Buffer write hit ratio (moodle) 39](#_Toc17463090)

[圖 4‑4 Write Buffer write hit ratio (UG-fileserver-1) 40](#_Toc17463091)

[圖 4‑5 Write Buffer write hit ratio (UG-fileserver-2) 40](#_Toc17463092)

[圖 4‑6 Write Buffer write hit ratio (User-1) 40](#_Toc17463093)

[圖 4‑7 Write Buffer write hit ratio (User-2) 41](#_Toc17463094)

[圖 4‑8 從Write Buffer剔除的pages個數 (moodle) 41](#_Toc17463095)

[圖 4‑9 從Write Buffer剔除的pages個數 (UG-fileserver-1) 42](#_Toc17463096)

[圖 4‑10 從Write Buffer剔除的pages個數 (UG-fileserver-2) 42](#_Toc17463097)

[圖 4‑11 從Write Buffer剔除的pages個數 (User-1) 42](#_Toc17463098)

[圖 4‑12 從Write Buffer剔除的pages個數(User-2) 43](#_Toc17463099)

[圖 4‑13 Avg. request response time (moodle) 43](#_Toc17463100)

[圖 4‑14 Avg. request response time (UG-fileserver-1) 44](#_Toc17463101)

[圖 4‑15 Avg. request response time (UG-fileserver-2) 44](#_Toc17463102)

[圖 4‑16 Avg. request response time (User-1) 44](#_Toc17463103)

[圖 4‑17 Avg. request response time (User-2) 45](#_Toc17463104)

[圖 4‑18 Avg. read request response time(User1) 圖 4‑19 Avg. read request response time(User2) 45](#_Toc17463105)

[圖 4‑20 Avg. read request response time / GC copy page count (moodle) 46](#_Toc17463106)

[圖 4‑21 Avg. read request response time / GC copy page count (UG-fileserver-1) 46](#_Toc17463107)

[圖 4‑22 Avg. read request response time / GC copy page count (UG-fileserver-2) 46](#_Toc17463108)

[圖 4‑23 Avg. request response time, only use hint group=5 (User-1) 48](#_Toc17463109)

[圖 4‑24 Avg. request response time, only use striping, write ratio=0.4 (User-1) 48](#_Toc17463110)

[圖 4‑25 Avg. request response time, only use write to each channel (User-1) 48](#_Toc17463111)

[圖 4‑26 Avg. request response time, only use hint group=5 (User-2) 49](#_Toc17463112)

[圖 4‑27 Avg. request response time, only use striping, write ratio=0.4 (User-2) 49](#_Toc17463113)

[圖 4‑28 Avg. request response time, only use write to each channel (User-2) 49](#_Toc17463114)

[圖 4‑29 Avg. request response time, only use hint group=5 (UG-flieserver-1) 50](#_Toc17463115)

[圖 4‑30 Avg. request response time, only use striping, write ratio=0.4 (UG-fileserver-1) 50](#_Toc17463116)

[圖 4‑31 Avg. request response time, only use write to each channel (UG-fileserver-1) 50](#_Toc17463117)

[圖 4‑32 Avg. request response time, only use hint group=5 (UG-flieserver-2) 51](#_Toc17463118)

[圖 4‑33 Avg. request response time, only use striping, write ratio=0.4 (UG-fileserver-2) 51](#_Toc17463119)

[圖 4‑34 Avg. request response time, only use write to each channel (UG-fileserver-2) 51](#_Toc17463120)

[圖 4‑35 Avg. request response time, only use hint group=5 (Moodle) 52](#_Toc17463121)

[圖 4‑36 Avg. request response time, only use striping, write ratio=0.4(Moodle) 52](#_Toc17463122)

[圖 4‑37 Avg. request response time, only use write to each channel (Moodle) 52](#_Toc17463123)

表目錄

[表 2‑1快閃記憶體的基本操作 6](#_Toc17201987)

[表 4‑1　Trace特性說明 37](#_Toc17201988)

# 第一章 緒論

## 簡介

NAND flash memory組成的固態硬碟(SSD)，比起傳統的HDD具備速度更快、抗震性佳、體積小等特性。近年來，SSD價格逐漸下降，且容量變大，漸漸取代傳統硬碟，現今手機、平板等許多儲存裝置都使用SSD。

但是SSD仍然存在的些許缺點，像是，NAND flash memory無法in-place-update，這讓SSD在更新資料變得較為麻煩，首先要知道在SSD讀寫最小單位是page，但擦除卻是block，因此若想更新一個page，必須先擦除那個page所在的block，然後才能寫入。此外，NAND flash memory的每一個block有被擦除次數上限，當一個block被擦除次數超過上限，代表這個block無法被存取，為了能盡可能延長SSD的壽命，通常會在SSD上層加入一個RAM-Based的buffer，也就是write buffer，短時間內頻繁被寫入的資料，可以先在write buffer內in-place-update，這樣的做法可以大量減少SSD被寫入的次數，進而提升SSD的壽命。

## 研究動機

目前，page cache與write buffer都是各自獨立運作，因此write buffer在選擇替換victim時，可能會做出不好的選擇，導致過早剔除，為了能夠更精確地挑選替換的victim，我們從Host端獲取一些相關資訊，來幫助我們選擇victim，提升write buffer的hit ratio，減少Flash memory的寫入次數，提升SSD的壽命。

此外，由於我們無法得知未來的資訊，對於使用者隨機的request，在挑選victim時，容易做出錯誤的判斷，為了解決這樣的問題，我們使用machine learning來預測，希望machine learning藉由訓練能夠掌握到我們無法得知的【未來】資訊，進而做出更精準的判斷。

## 貢獻

本篇論文利用AI來選擇合適的victim block做踢除，根據資料的特性，AI model會判斷出資料屬於hot data or cold data，以此來決定踢除資料的順序。

此外，我們同時利用host端的資訊，來輔助管理SSD內部write buffer，讓host端的page cache與device端的write buffer能夠互相合作。在過去的研究中，page cache與write buffer是各自獨立的，因此write buffer只能夠根據當下&以前的資訊來決定踢除的對象。

然而，由於write buffer容量有限，且只依靠SSD內部資訊，對於hit ratio的改善幅度有限，因此，本篇論文除了使用AI model以外，更參考了host端的資訊，在資料被寫入write buffer以前，事先得知可能被寫入的block以及寫入的次數，根據這些資訊，並結合AI model預測的結果，來決定該優先將那些block挑選為victim。

## 論文架構

第二章介紹一些相關的背景知識以及write buffer的相關論文， 第三章描述本論文的架構與實作方法，第四章為實驗數據與分析。最後，第五章為結論、未來工作。

# 第二章 背景知識與相關研究

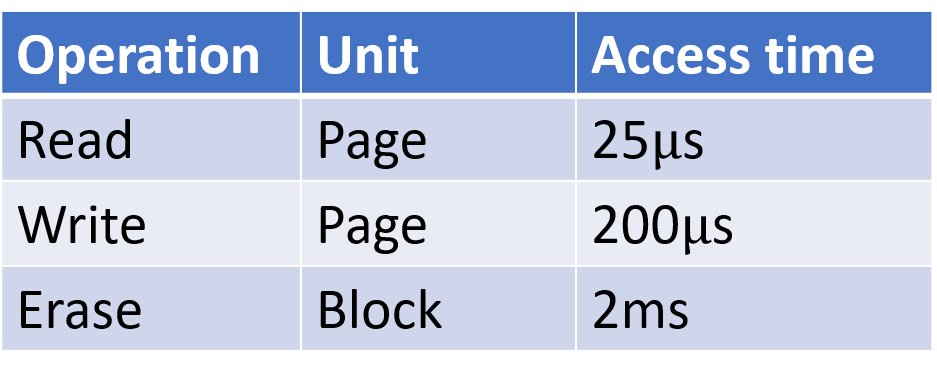
本章先介紹SSD相關背景知識，接著介紹Linux flash方式、flash memory特性快閃記憶體的特性、固態硬碟的管理方法，最後介紹資料在SSD的擺放方式。2.1將介紹SSD， 2.2介紹flash translation layer(FTL)，2.3介紹Linux Flash的方式，2.4介紹write buffer的管理方式，2.5介紹資料在SSD的擺放方式，2.6介紹擺放方式的研究



## SSD

SSD是由NAND Flash memory所組成，因為是非揮發性記憶體，因此能在斷電的情況下仍然保存資料，此外還有一些吸引人的特性，例如，體積小、抗震，但最吸引人的部分在於它的速度，比起HDD快上非常多，因此漸漸被個人電腦、手機、平板當作主要的儲存裝置。

表2-1 快閃記憶體的基本操作單位



如表2-1，SSD基本操作可分成read、write、erase，但由於NAND Flash memory的物理特性，SSD存在幾個缺點：

1. 從表2-1得知，read/write速度不對稱
2. SSD out-of-place-update的特性，當SSD在overwrite時，沒辦法像HDD一樣in-place-update，必須找另一個空的block才能夠寫入，這讓SSD在overwrite時，變得很麻煩，首先，每次overwrite都找新的block寫入，很耗費SSD的空間，因此會需要在空間不足時，藉由erase來釋出更多空間，而erase的最小單位又與read/write不同，是以block為單位。
3. 每個block有erase的次數限制，如果過於頻繁的erase，當某個block達到erase次數上限，這時候整個SSD內的所有block都無法再被存取

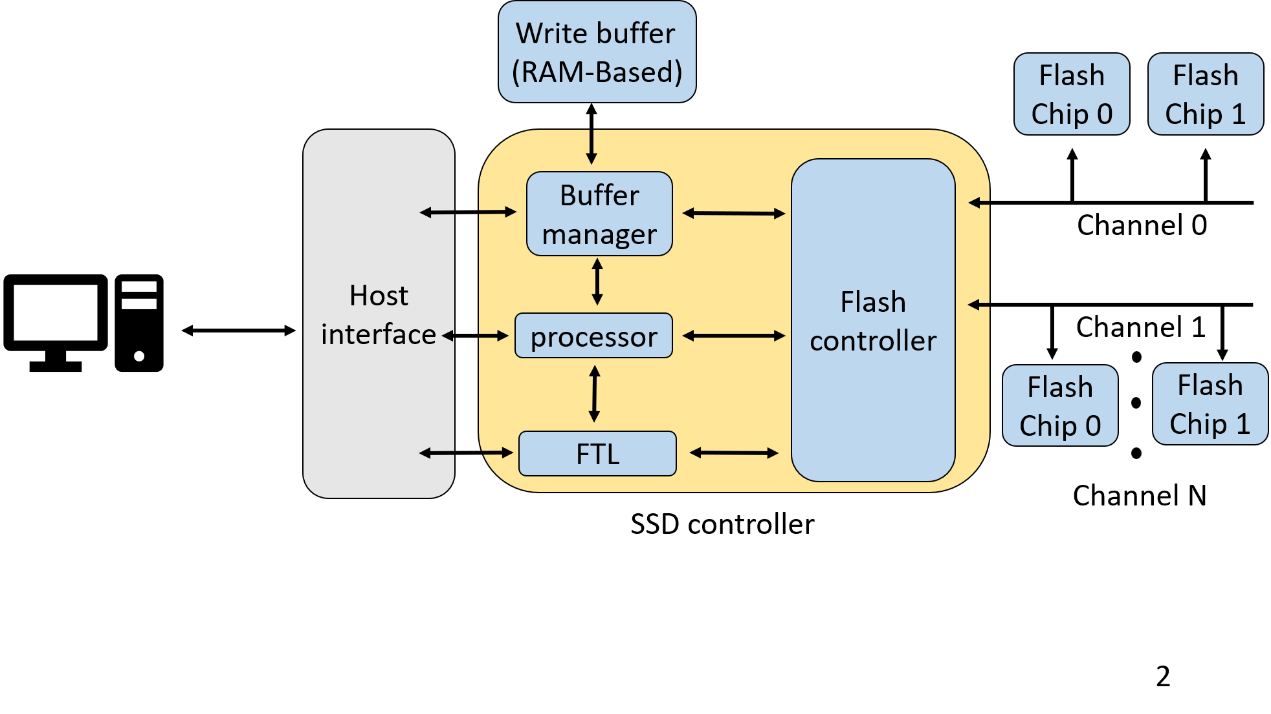


圖2-1 SSD 架構圖

接下來介紹SSD架構圖，如圖2-1可看到SSD內部由process、buffer manager、flash controller所組成，ssd controller負責處理內部元件運作以及與host interface之間的溝通，而SSD controller內部有FTL做為控制中樞，此外，為了減少寫入次數& response time，SSD使用一塊ram-based write buffer放在SSD的上層吸收來自host端的write request，只有在write back，或是write buffer滿的時候，才需要寫入SSD。

## Flash Translation Layer

由於NAND Flash memory物理特性的關係，我們無法對SSD in-place-update，overwrite時，必須以out-of-place-update的方式，因此SSD原本是無法使用於一般的檔案系統，但為了能夠用用到SSD速度快等優點，於是有了Flash Translation Layer(FTL)這個轉換機制，能夠將NADN Flash memory模擬成一般HDD，讓上層的檔案系統(File System)可以存取

因此對於SSD來說，FTL可以說是不可或缺，它主要管理兩項功能，分別是Address mapping以及Garbage collection

1. Address mapping：將邏輯位址(logical address)與物理位址(physical address)進行轉換，將被write buffer踢掉的page mapping到正確的物理位置。
2. Garbage collection：由於out-of-place的特性，SSD很容易發生空間不足的狀況，此時就會以Garbage collection來釋出一些被invalid page所占用的空間，讓SSD可以有更多空間可用

## Linux Flush

由於page cache為揮發性記憶體，遇到斷電或是突發性狀況時，資料無法保存，為了避免資料遺失，Linux kernel可透過Flush將資料寫回HDD或是SSD，而Flush可分為兩種，分別是，定量Flush以及定時Flush。

### 定量Flush

這個章節，主要介紹定量Flush，當Page cache中的Dirty page超過Linux所制定的比例(dirty ratio)---40%時，就會觸發定量Flush，將Page Cache中的dirty page寫回HDD或是SSD，以確保資料不會遺失，為了能快速找出Page Cache中Dirty page的位置，Linux會維護一個Dirty page queue，假設Page Cache內使用的替換策略是LRU，那Dirty page queue的排序方式，也會依照Page Cache中的LRU方式去排列，如圖2-2，當Page Cache中的Dirty ratio達到40%在寫回的時候，會以Dirty page queue的LRU為起點，將Dirty page依序寫回HDD或是SSD，當Page Cache中的Dirty ratio達到40%時，Linux kernel會將Dirty page queue中的1024個Dirty pages寫回硬碟，如果不足1024個pages，則將全部Dirty page寫回。

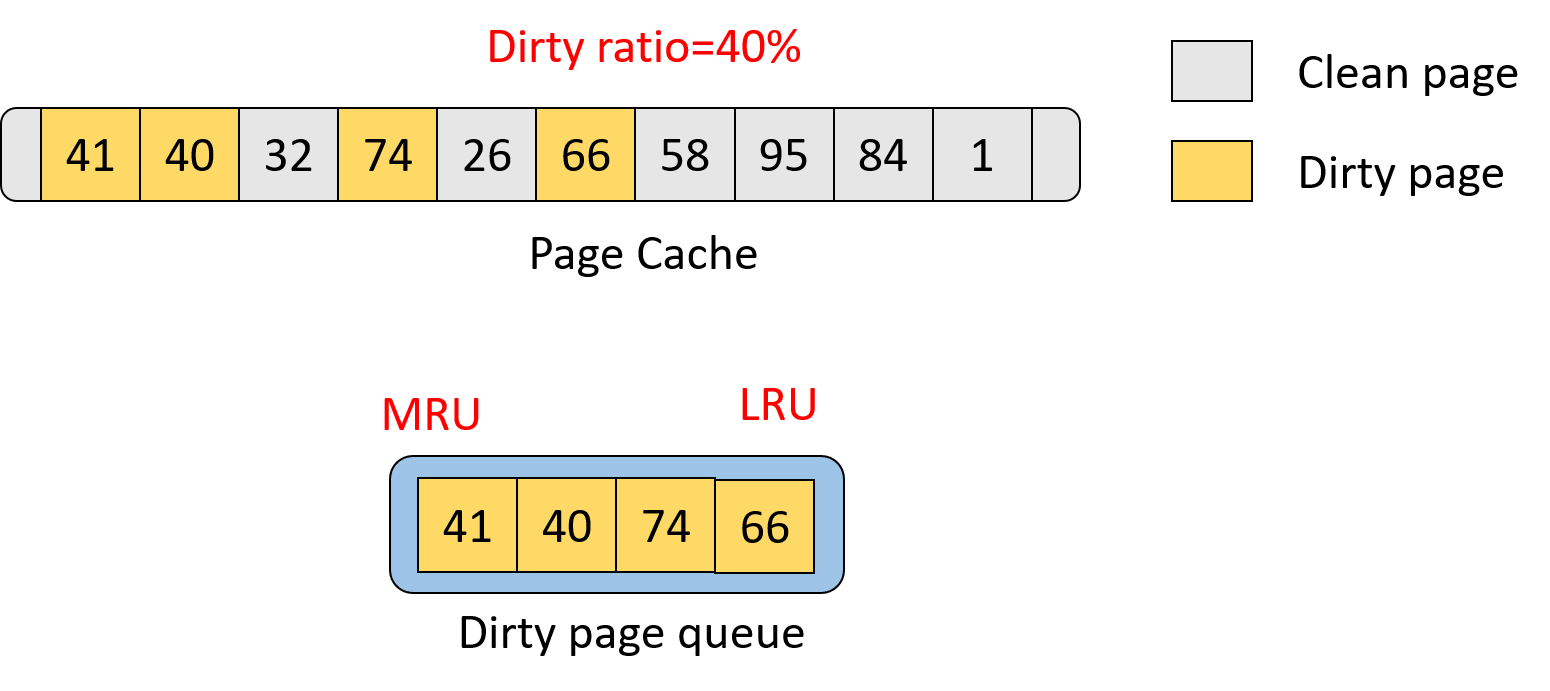


圖2-2 定量Flush

### 2.3.2 定時Flush

接下來介紹定時Flush，在2.3.1中，我們將從LRU端開始將Dirty page寫回，這麼做是讓比較常存取的資料能在Page Cache繼續被hit，但問題是，我們寫回的目的是為了確保資料不會因為意外狀況而遺失，因此，除了LRU端的資料，MRU端的資料也需要被寫回，但若總是從LRU開始做寫回，那MRU端的資料就沒有機會被寫回。

為了解決這樣的問題，我們在這章節使用定時Flush來寫回，與2.3.1不同，這次是以時間為衡量標準，我們挑選在Page Cache待超過30秒的Dirty page寫回硬碟， 每5秒運作一次，利用global time flag的方式，第一次會從LRU端開始搜尋1024個page，然後將global time flag設在第1024個page，下一次觸發定時Flash，就根據global time flag的位置繼續往MRU的方向搜尋1024個page，最後抵達MRU端時再往回找，直到碰到LRU端再轉向，如此循環下去。

以圖2-3來說明，每次定時Flush都會尋找1024個Pages，代表global time flag要不是停在第1024,2048…個Page上，就是位於LRU or MRU端，每次都會針對尋找範圍內的page挑出待超過30秒的Dirty page Flush，在這樣的機制下，就不用每次都尋找整個Page Cache，可減少Overhead。

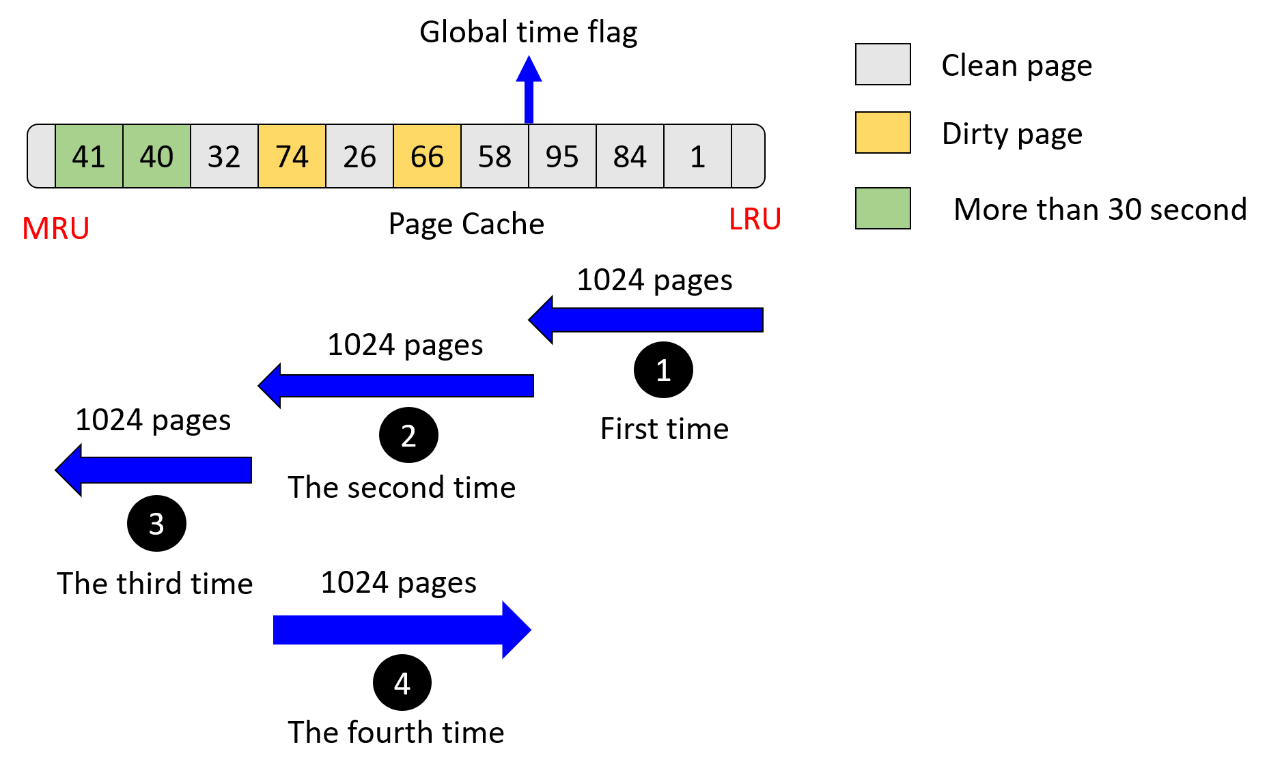


圖2-3定時Flush

### 2.3.3 Hint Queue

我們透過觀察之前的存取狀況，來預測接下來的存取狀況，分別預測request數量、write ratio、Miss ratio、victim page為dirty的ratio，透過這四個預測的資訊，進而可得出預測的dirty ratio以及dirty page count。

首先介紹下面四個公式代表意義：

* 下面公式中，p所代表的是predict，也就是預測某個資訊的意思，rc代表request count，也就是當下page cache內有多少個request，n+1是指第n+1週期，代表權重
* 所以第一個公式，其實就是在透過上次request count()和上次預測的request count()來預測這次的request count，藉由調整權重()來逼近真正的request count
* 第二~四個公式類似，先透過實驗來找出accuracy最高的，接著再算出各自的預測值：write ratio()、miss ratio()、victim page is dirty page ratio()
* 值得一提的是，其實我們並沒有直接去預測dirty page count，但我們知道request count、write ratio、miss ratio、victim page is dirty page ratio，整合起來，我們其實就能夠知道dirty page count的預測值
* 最後，分別提一下剩下三個公式代表的意義
  + ：在第n+1週期，write ratio的預測值
  + ：在第n+1週期，miss ratio的預測值
  + ：在第n+1週期，victim page is dirtyt page ratio的預測值

在得知上述資訊之後，透過預測dirty ratio就可以知道是否要執行定量flush，而透過預測victim page is dirty page ratio也可以知道是否有dirty page要被定時flush踢掉，定時flush不需要預測，因為固定時間就會執行一次。

最後，預測request count可以知道是否要做replacement，預測的結果會被存在Hint queue中，給SSD做參考。

## 2.4 write buffer管理方式

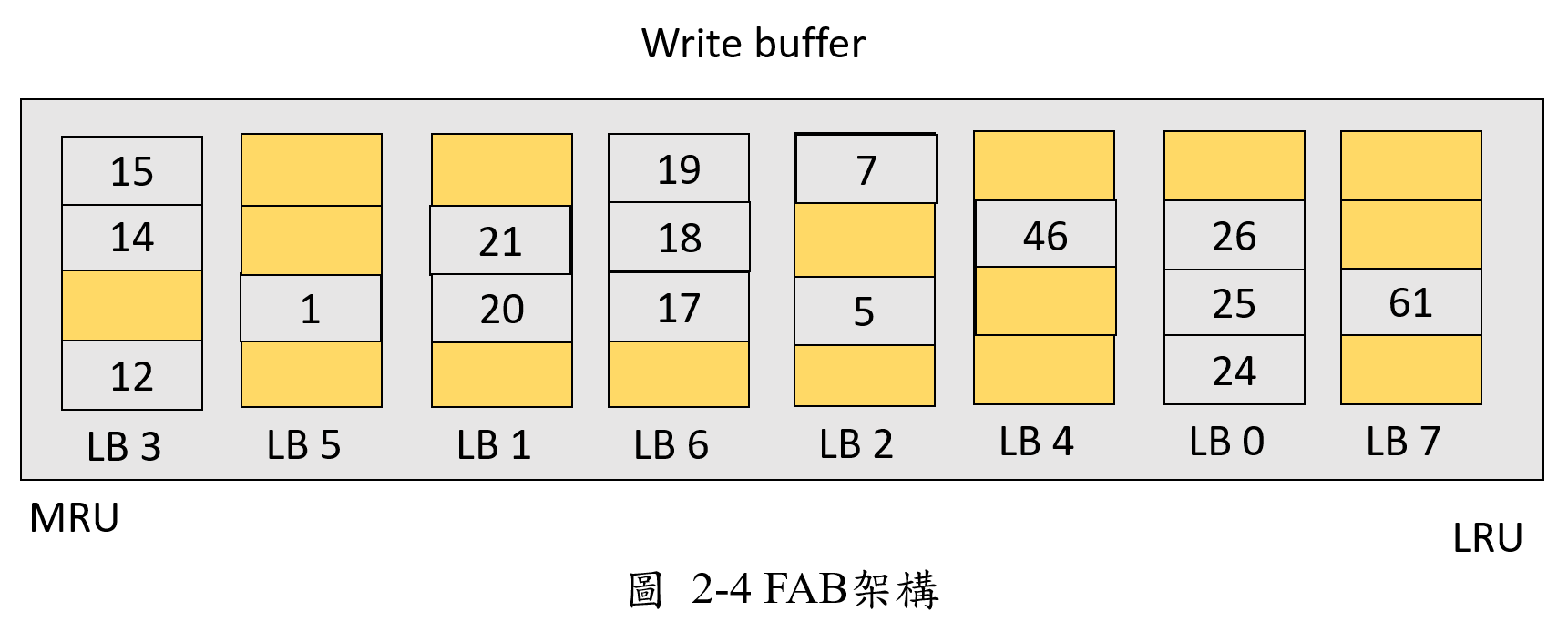
為了延長SSD的壽命，目前常見的方法是將一塊RAM-based write buffer放到SSD的上層，當page cache將資料從host端踢下來時，資料會先在write buffer做寫入，若之後寫入同樣資料，就會直接在write buffer hit。等到write buffer滿了，或是之後做寫回的時候，才會需要寫入NAND flash memory，以這樣的方式，可以大量減少NADN flash memory的寫入，進而達到延長SSD壽命的目的。

然而，write buffer能夠延長SSD壽命，是建立在【hit ratio夠高】這個前提下，因此，該踢掉那些資料、該保留那些資料，成為一個不可忽視的重點，接下來介紹幾個常見的write buffer管理策略。

### 2.4.1 FAB(Flash-aware-buffer management policy)

由於NAND flash memory物理特性，Erase 是以block 為單位，因此，FAB希望挑選連續的logical page踢到NAND flash memory，減少GC copy所花的時間。

此外，若遇到有多個block中的page數量一樣時，則選擇靠近LRU端的block進行踢除。如圖2-4，block 0、block 6、block 3中都有三個page，其中block 0、block 6是連續的，依照FAB，會優先挑選block 0作為踢除的對象，因為block 0最接近LRU端。

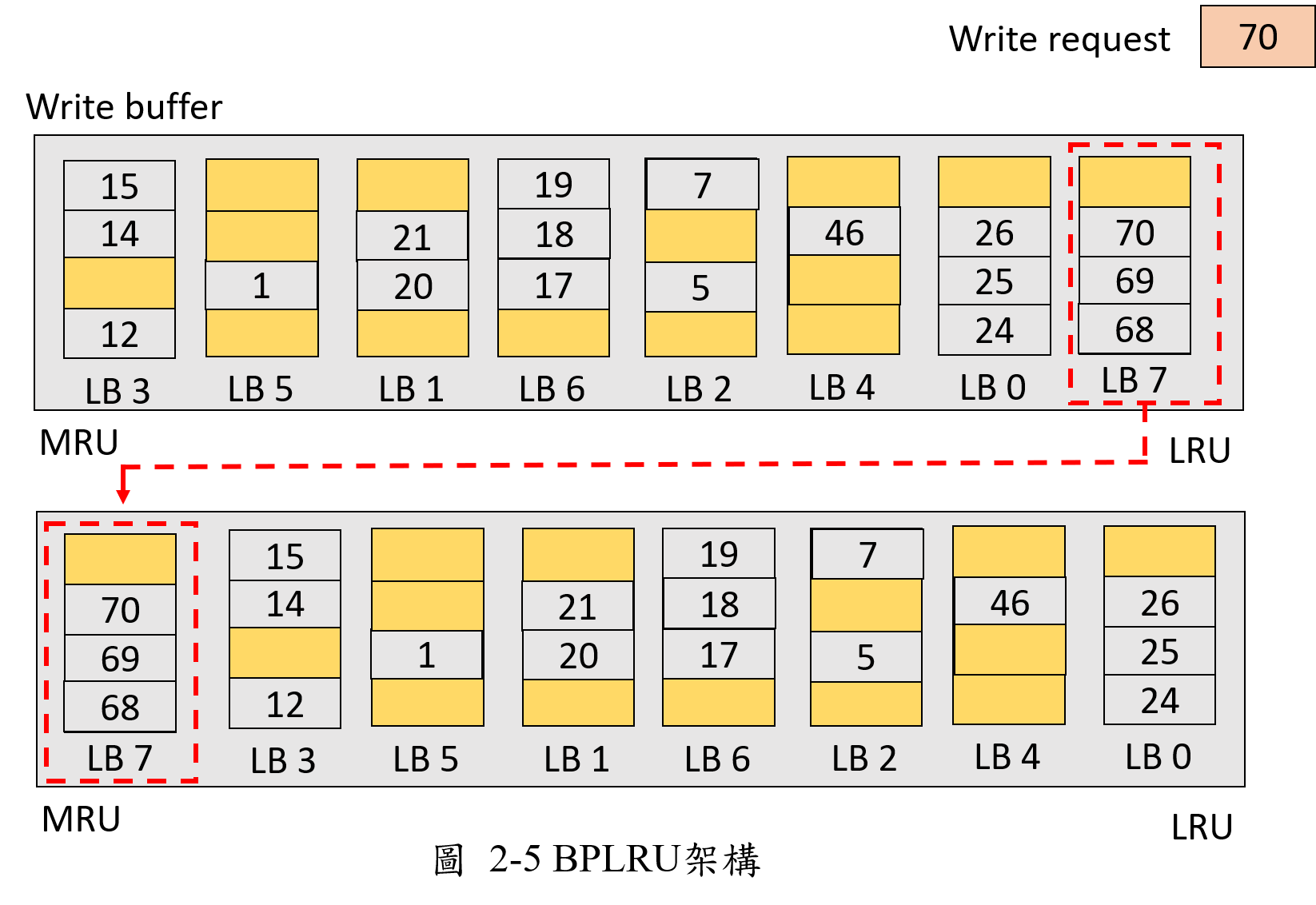


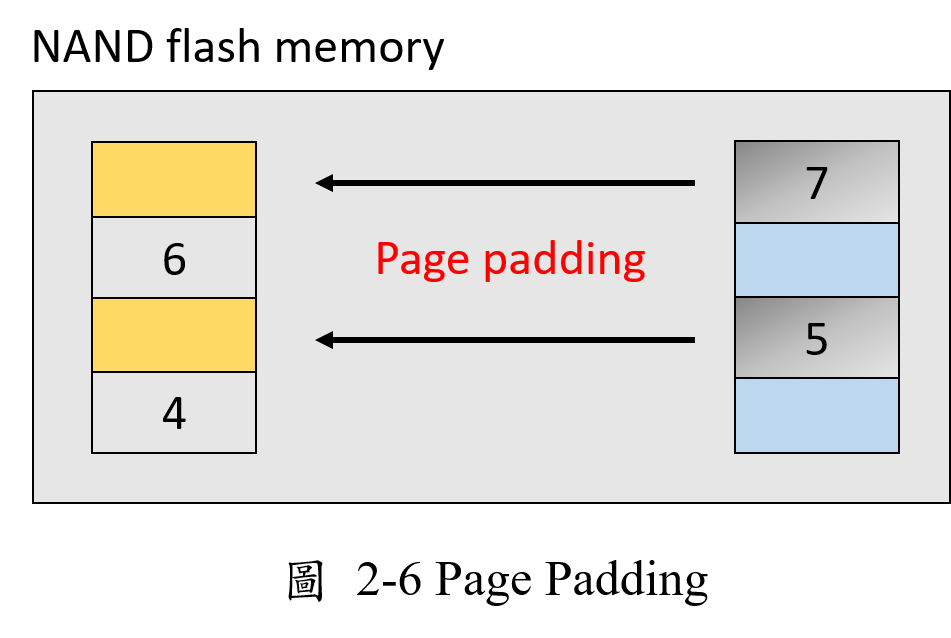
### 2.4.2 BPLRU(Block Padding Least Recently Used)

BPLRU考慮temporary locality、spatial locality，將同個logical block的page放在一起，如果存取到block中某個page時，整個block會一起移動到MRU端，當write buffer空間不足時，會從LRU端將某個block踢掉，此外BPLRU還有一項重要的機制，因為victim block也許不完整，為了減少GC所花的時間，BPLRU使用【Page padding】將victim block變成一個完整的block，減少GC的overhead。

以圖2-5為例，victim block 為block 7，但因為block 7不是一個完整的block，此時會啟動【Page padding】，將block 缺少的部分(page 71)從NADN flash memory讀上來，讓原先的block成為一個invalid block，當空間不足時，就可直接擦除，然後再將block 7踢下去。

最後，BPLRU還有一項機制叫做【LRU compensation】，當MRU端的block被循序寫滿時，將該block搬到LRU端，讓它能優先被踢掉。





## 2.5 資料在SSD的擺放方式

由於SSD為multi-channel架構，每個channel可以平行存取，因此在SSD中可分為兩種擺放方式：Page Striping與Block Striping，圖2-8、2-9為兩種Striping的例子。假設有4個channel，每個block有4個page，先後針對logical page 0-3做read，logical page 12-15做write。

### Page Striping

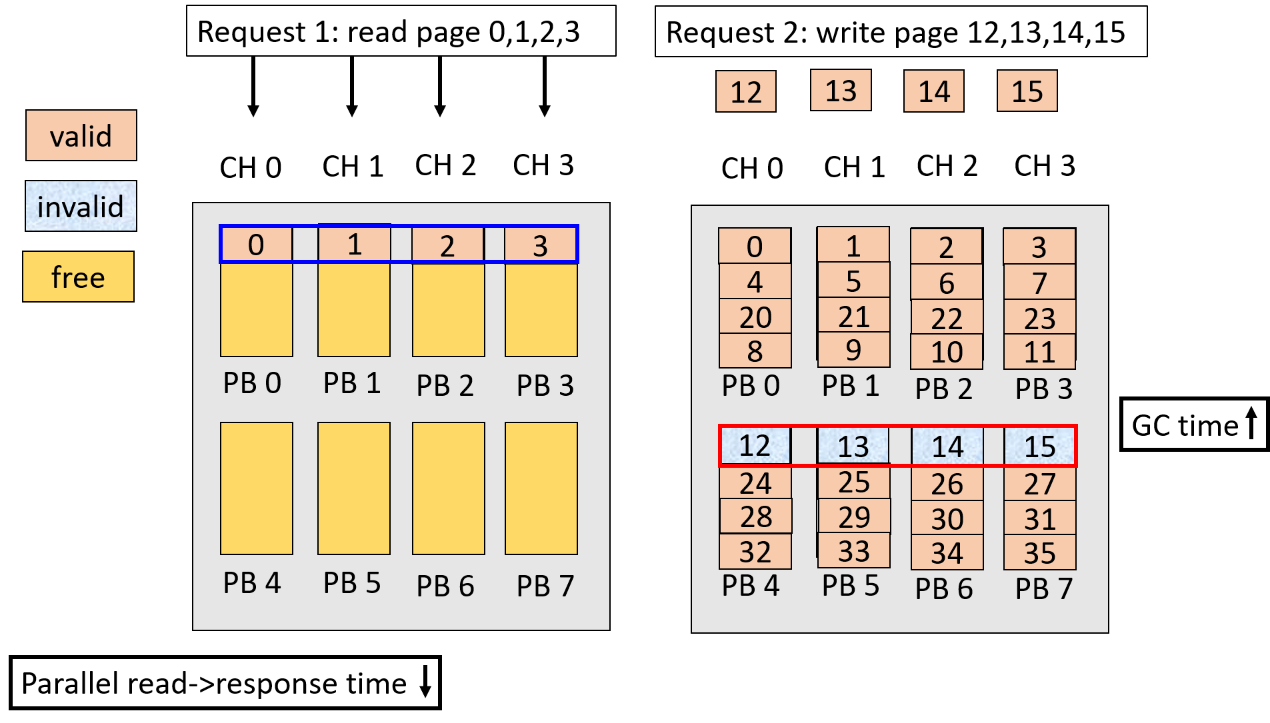


圖 2-8 Page Striping

圖2-8為page striping的例子，logical page分布在各個channel上，可利用multichannel的並行性，對page 0-3做平行讀取，減少response time，對write request來說，雖然也可以平行寫入，但因為page分散在不同的block上，在GC時所花費的時間成本遠大於平行寫入帶來的效益。

以圖2-8右邊為例，physical block 4-7在做GC時，每次都得複製3個page，因為每個block只有一個invalid page。

### Block Striping

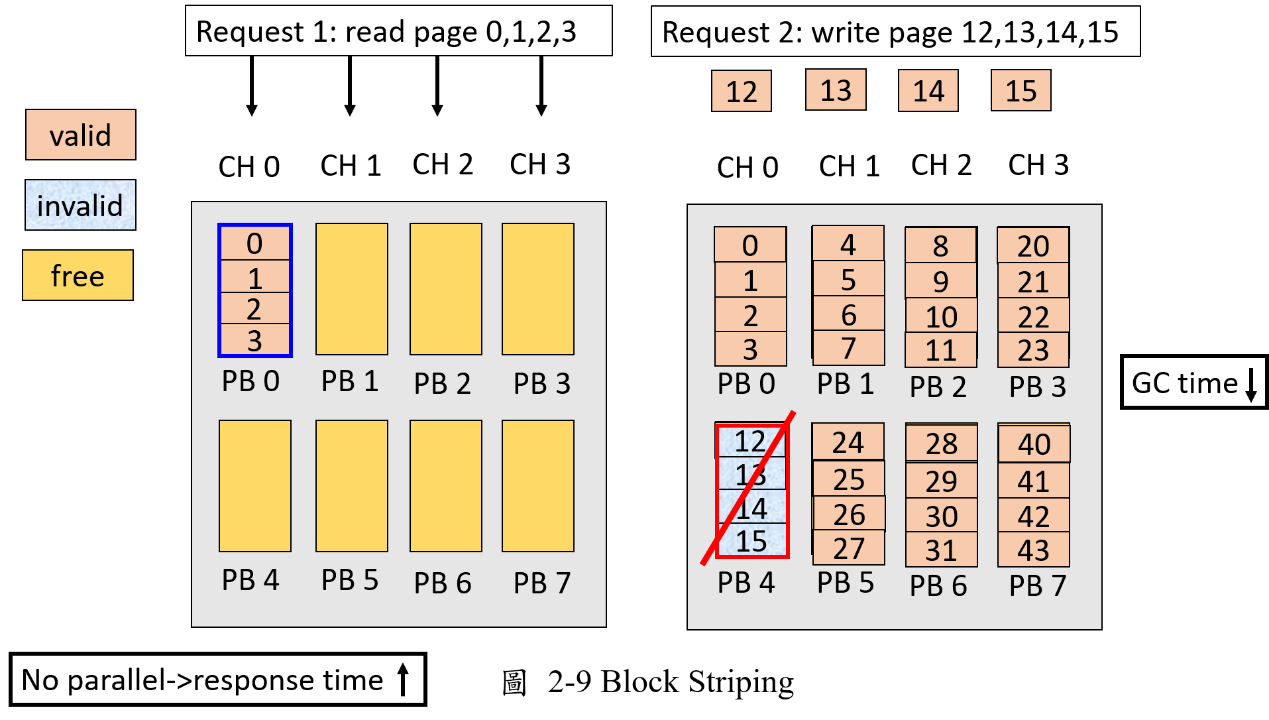


圖2-9為block striping的例子，將同一個logical block的page放在同一個physical block，希望能夠利用到空間局部性。

對於read request，沒辦法像page striping那樣平行化，但是對於write request，能夠增加invalid page的數量，進而降低GC所花費的時間。以圖2-9右邊為例，連續寫入page 12-15會使PB3的所有page都變成invalid，若使時發生GC，可直接Erase PB3不需要額外的GC copy。

## 擺放方式相關研究

### BLAS

SSD採用multichannel架構，SSD中的擺放方式會影響channel平行化的效果。如果完全不考慮workload 的read/write intensive，可能會導致response time上升，或是GC time上升。好比說，對read intensive block以block striping方式擺放，或是對write intensive block以page striping方式擺放，前者會造成response time上升，因為一次只讀一個page。後者則會造成GC time 上升，因為一個block內有較多的valid page。

BLAS提出動態擺放的方式，在最大化平行度的同時，也兼顧GC的成本。對於read-intensive block，使用Page striping，最大化平行度，對於write-intensive block，則採用Block striping，將page 寫入同一個block，以減少GC所花的時間成本。至於如何判定是否為read-intensive，BLAS根據連續讀取次數來判定是否為read-intensive block，若連續讀取次數到達制定的Threshold，就判斷為read-intensive，反之，則為write-intensive。

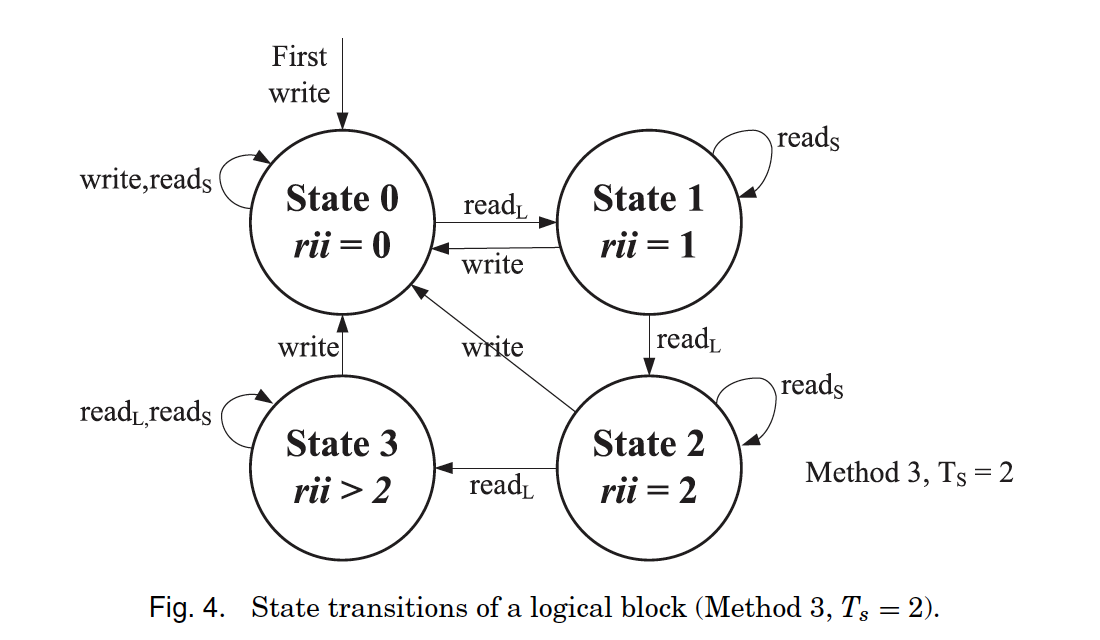


圖 2-10 BLAS State

如圖2-10，假設Threshold(Ts)=2，共有4個State，當對此block連續讀取3次，則判定為read-intensive，反之，則為write-intensive，並根據對應的擺放方式，將資料放入SSD。

# 第三章 Using LSTM combine with Host information for selecting write buffer victim block

## 系統架構

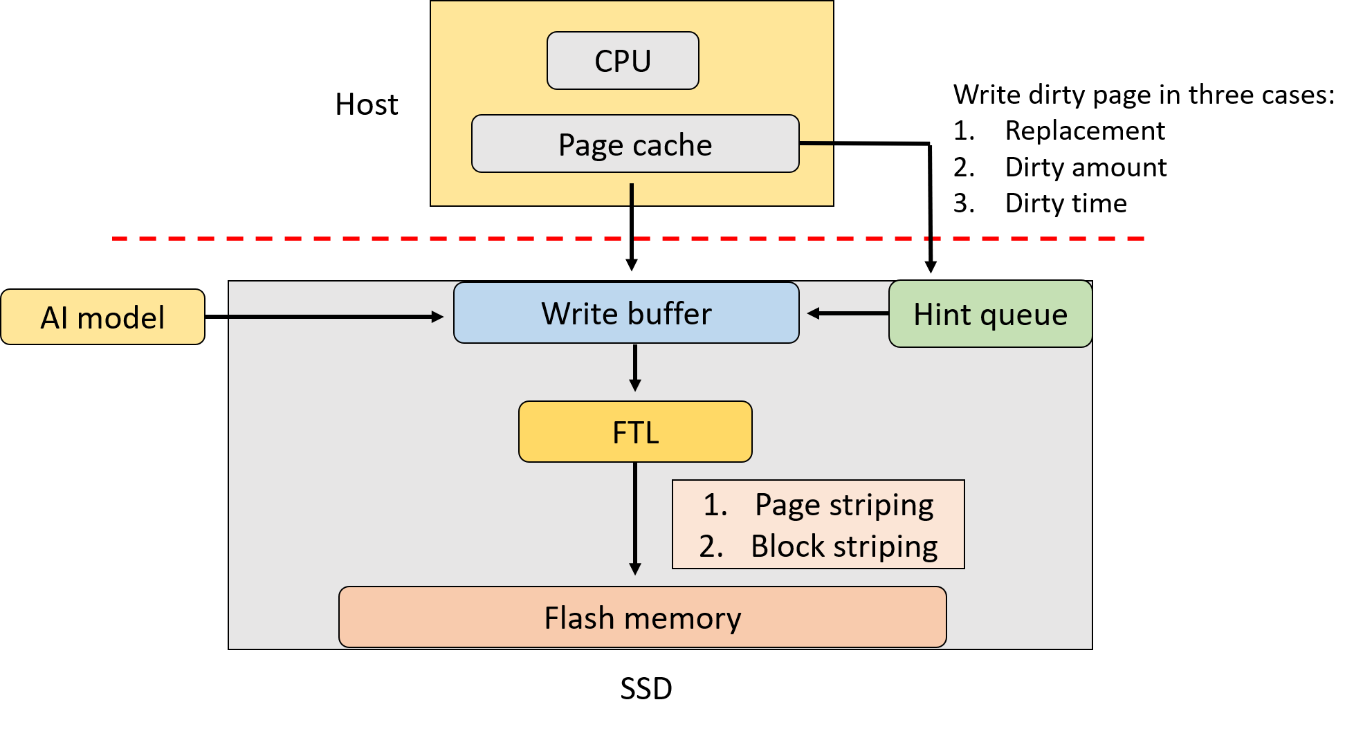


圖 3‑1 系統架構圖 [8]

圖3-1是我們的系統架構，上層Host端分成CPU與Page Cache，下層為SSD，SSD內部有Hint queue與RAM-based Write Buffer，以及FTL和Flash memory。

Page Cache在三種情況會將dirty page寫入write buffer，【Page Cache已滿】【定時flush】【定量flush】，藉由週期性預測，將預測會被寫入的dirty page number寫進Hint queue，然後送到SSD中，透過這樣的方式，讓write buffer能夠獲取host端的資訊，達到host(hint queue)與SSD(AI model)相互合作效果。

另一方面，使用AI model預測的結果和Hint queue的資訊，選擇合適的victim block 踢除。

## AI運作流程

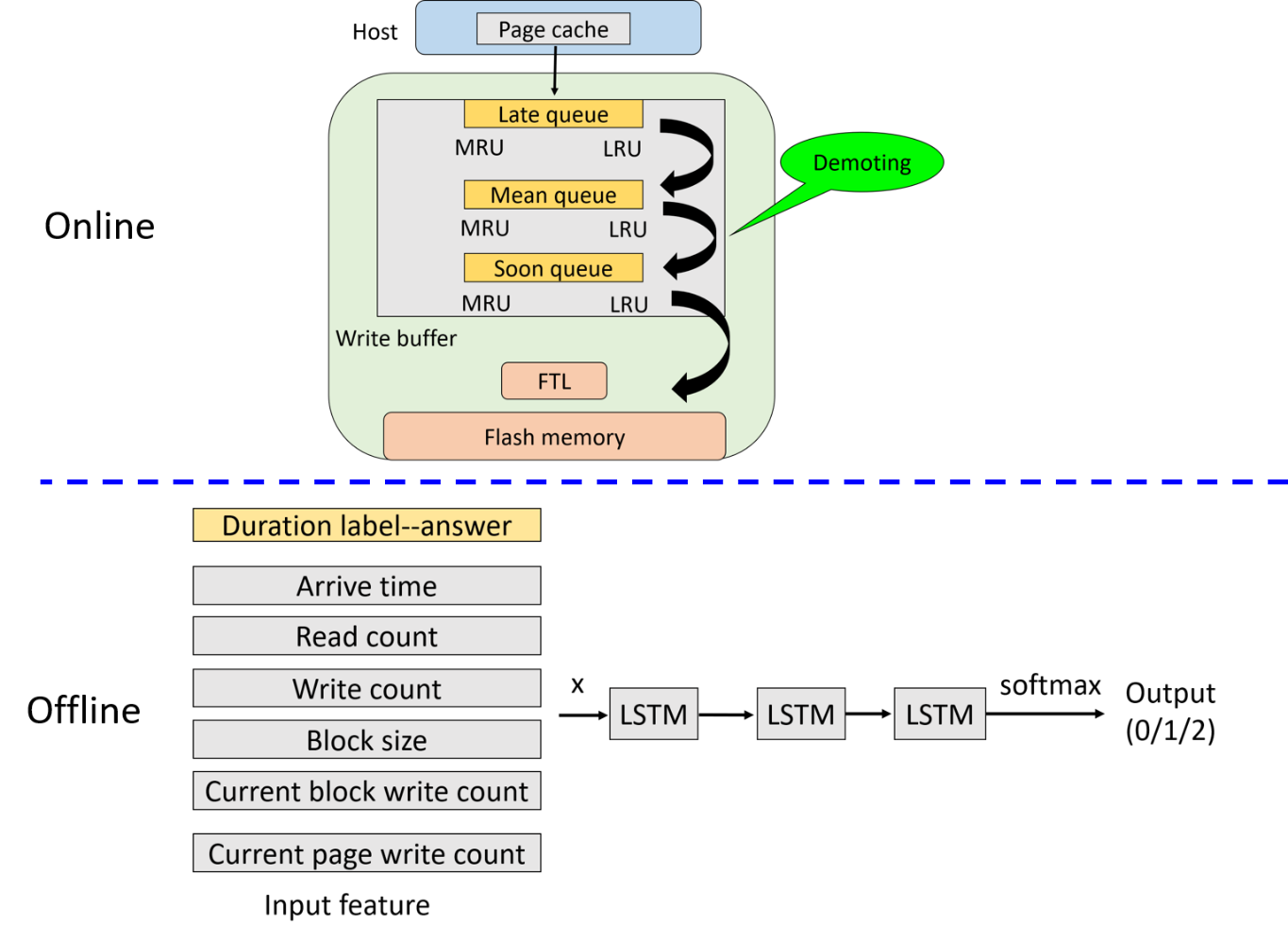


圖 3‑2 AI架構圖

AI 會分成online testing、offline training兩階段，在offline training會先將input feature送進三個LSTM層，input feature就是model會看到的資訊，會被用來做訓練，通過LSTM後，將輸出結果通過softmax activation function的output就是預測的結果，之後loss function會計算出loss，也就是預測值與正確答案的誤差，之後model會根據loss來調整權重，以獲得更高的accuracy。

在這篇論文，我們使用的input feature分別是【arrive time】【read count】【write count】【block size】【current block write count】【current page write count】，以下分別介紹上述每個feature的用意:

* Arrive time: 為了讓model知道不同時間點的request狀況
* Read count: 因為頻繁寫入的資料通常讀取次數會比較少，因此希望藉由read count讓model區分這個block是write intensive或是read intensive block
* Write count: 為了避免model單靠read count無法掌握block的read、write資訊，這裡多給一個相對的特徵讓model對照
* Block size: 由於我們的目的是希望model能夠踢掉large block，因此給予block size讓model知道當下block size是多少
* Current block write count/read count:給予current block read、write count的用意在於希望能夠讓model掌握recency的部分，所以告訴model最近讀寫那些page、那些block，不過這部分似乎model還沒辦法掌握到。

Online testing部分，由於考慮到也許會發生預測錯誤的問題，也就是明明不常被存取，卻被放置於應該要很常被存取的queue，這時候就會藉由將資料做demoting，將那些不常被存取的block從write buffer踢掉。

接著，我們會詳細介紹如何收集必要的Data，以及LSTM、demoting具體是如何運作的。

### Benefit function

Write buffer在踢資料的時候，基本概念就是少用的優先踢掉，以此為原則，我們想出了一個function—benefit function，用來當作判斷是否要將資料留在write buffer的標準。

Benefit function:

Benefit function是以block為單位做計算，因為write buffer在踢資料時，也是以block為單位。以下介紹benefit function中每個名詞的意義：【block write count】指的是每個block被寫入的次數總和，因為block是由多個page組成，且寫入的最小單位也是page，因此block write count其實就是該block每個page write count的總合；【block size】，指的是當下所計算的block中，目前有多少個page。

會這樣設計benefit function的用意在於，一方面我們希望頻繁被寫入的資料能保留在write buffer，與此同時，又希望victim block能夠是large block，因此公式的前半部會去計算單位page被寫入的次數，後半部則是針對block size，讓large block獲得較小的benefit value，如此一來，在選擇victim block時，若存在多個write count相同的block，則large block更容易被選為victim block從write buffer踢掉。

### Data collected

我們除了要知道每個block的benefit value以外，我們還必須知道每個block實際上在write buffer待多久，為了達成這個目的，我們會先建立一個write buffer simulator，模擬request從write buffer進入以及離開的狀況，以此來掌握每個block的資訊。

首先，我們為了掌握每個block的大小以及被寫入的次數，我們會先跑一次write buffer simulator，這在獲得每個block的benefit function之後(也就是未來的資訊)，我們會再跑一次，這次是為了觀測每個block實際待在write buffer多久，這個資訊，我們以duration來稱呼它，duration的單位是request，也就是【該block進入write buffer到它被選為victim block踢掉】這段期間，總共有多少個request進入write buffer。而在write buffer simulator中，我們踢掉資料的依據就是【benefit value】，前面已經提過benefit value的公式，而benefit value的意義，顧名思義，其實就是每個block的價值，因此在選擇時，會挑小價值最小的block踢掉，換句話說，也就是min benefit block，如圖3-3。

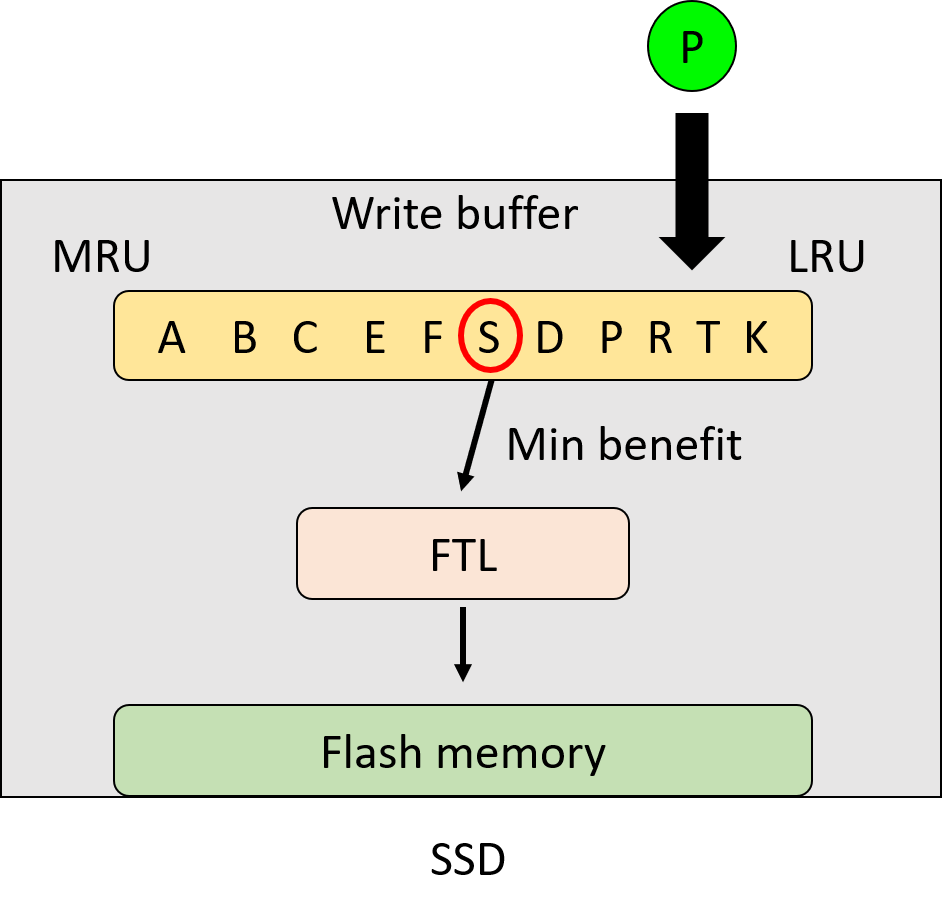


圖 3‑3 generate duration value

以這樣的方式，在write buffer simulator跑完第二輪之後，每個block可獲得：

1. Duration value
2. Benefit value

Benefit value在跑完write buffer simulator之後就沒有用處了，duration value則是會轉換成label的形式，當成正確答案，送進model做訓練。

### Transform duration vale into duration label

因為我們打算讓model預測label，因此，我們需要先將duration value轉換成duration label。轉換方式如圖3-4，我們先設定一個threshold，若duration value<threshold，則判定為soon(label=0)，若duration value介於threshold\*1~threshold\*5之間，則判定為mean(label=1)，若duration value>=threshold\*5，代表這個block很可能是hot block，因此判定為late(label=2)。

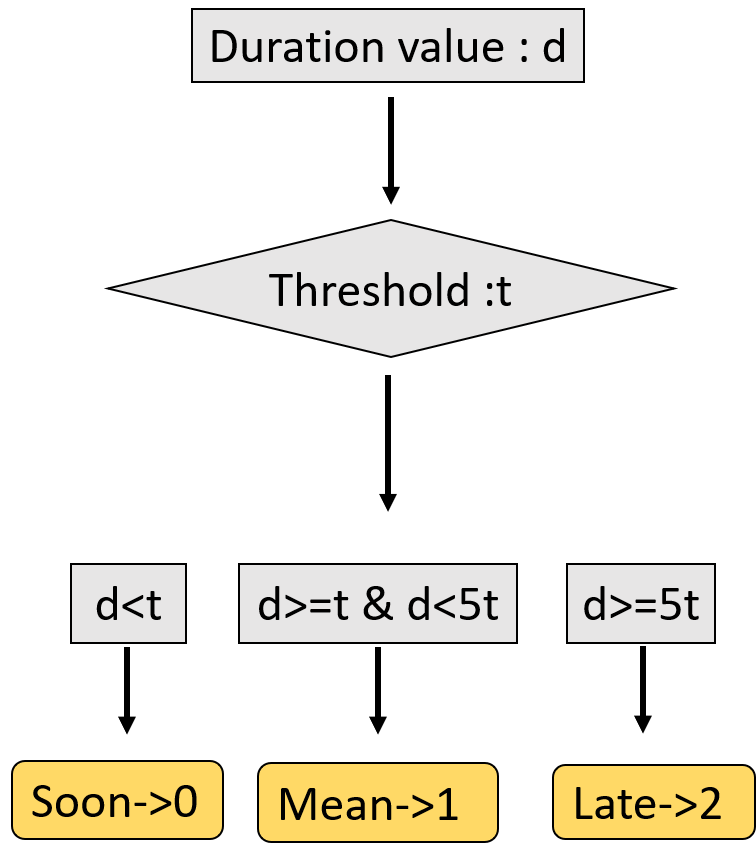


圖 3‑4 generate duration label

將每個block的duration value轉換成label之後，就可以開始進行訓練了，接著，我們會介紹offline training中最開始碰到的unit，【LSTM】。

### LSTM

LSTM本身分成3個gate，分別是【forget gate】【input gate】【output gate】，這三組gate會各自決定【是否要讓當下資訊往前傳遞】。如下圖3-5，g是希望能夠被放到memory cell( C )的資訊，而Zi則是進入activation function的input，由於LSTM主要是以gate形式去運作，因此這裡的activation function是使用sigmoid，主要是利用sigmoid output 為0~1的這個特性，能夠以0來表示關閉gate，1表示將gate開啟。

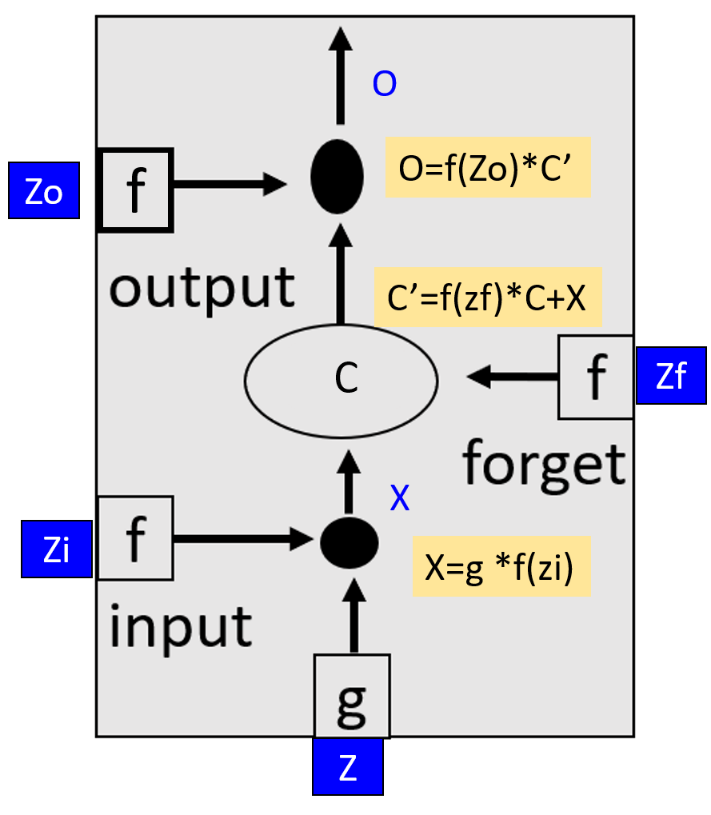


圖 3‑5 LSTM

當input的資訊(g)進入memory以前，會先與f(zi)相乘，由f(zi)決定是否要將資訊往前送，若f(zi)=1，這時候資訊，也就是X，才會進入memory cell( C )，在進入memory cell以前，會先依照第二個gate，也就是forget gate來決定是否要記住【當下memory cell的資訊( C )】，換句話說，也就是上次存入memory的資訊( C )，而這個也是由activation function所決定，也就是f(zf)。

所以當X實際被放入memory cell以前，會加上f(zf)\*【上次memory cell的資訊( C )】，假如model認為上次的資訊無助於學習，甚至有害，那f(zf)的output就會是0，反之，則會output 1，最後會將相加後的結果(C’)覆蓋掉memory cell現有的資訊。

通過memory cell之後，仍然不一定能夠成功output，因為接著會經過output gate，要將目前的資訊(C’)\*f(zo)，最終由f(zo)決定是否要output，若f(zo)判定為可以輸出，那才會真的將資訊( O )輸出。到目前為止，只是說明1個lstm unit在1個時間點的output，然而，lstm同一時間點通常是多個unit，且通常會去看多個時間點，因此，接下來會介紹，進入每個gate的input到底是如何產生的。

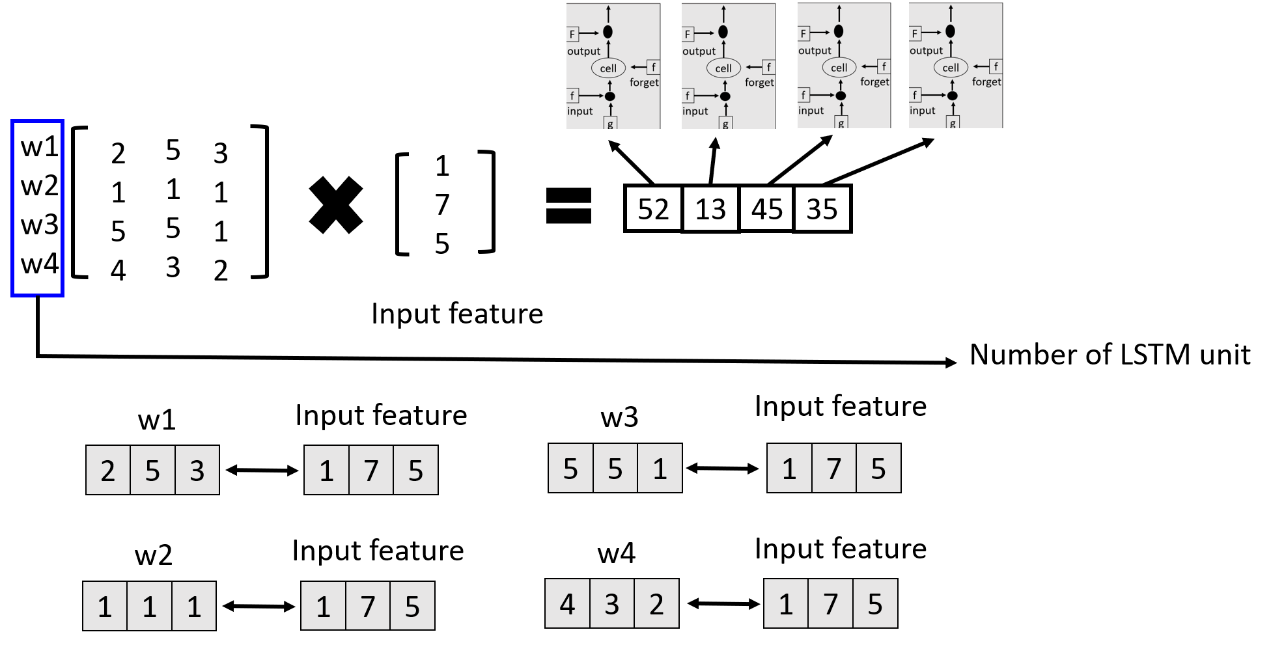


圖 3‑6 LSTM input

當input feature進入LSTM後，會乘上4個不同的權重矩陣，目的是為了操控input gate、output gate、forget gate、input，會多出一個input是因為，神經網路input本身原本就是有對應的權重，因此input也會有對應的權重矩陣。

如圖3-6，假設當前是要產生input gate的input，首先會乘上一個權重矩陣，而這個權重矩陣會包含n組權重，以圖3-6來說，就是4組，這就是input gate權重的個數，而這個個數，也會決定同一時間，會有多少個LSTM單元。因為每一組權重會和input feature做矩陣乘法，而做完矩陣乘法會得到一個純量，圖3-6有四組權重，那就會是4組純量，而這4組純量，會各自去操控一個LSTM unit的input gate，前面圖3-3，Zi就是4組純量中的其中一組，然後Zo, Zf, Z也是以相同方式算出來的。

由於每個時間點都會有各自的input feature，所以每個時間點也都有各自lstm的output，有時候為了特殊目的，會抓取特定時間點的output，但在本篇論文中，所抓取的是最後一個output，比如說model看16個時間點，那就會是拿time stamp 16的output當作output。最終，會將各個LSTM unit output，以圖3-4來說，也就是4個純量，合併為一個向量，根據圖3-2的流程，通過softmax就會得到model所預測的output label。

到目前為止，完成了offline training，接下來，會介紹online testing，在online中是如何解決model誤判的狀況。

### Demoting

首先介紹online部分，write buffer的擺放方式，有別於以往的write buffer只有一個LRU queue，在本篇論文中，write buffer共有三個LRU queue，如圖3-7，分別是soon、mean、late，分成三個queue的用意在於，以此來區分cold data、hot data，late queue放很頻繁被存取的資料，接著是mean queue，最後soon queue放的資料會是很少被用的資料。

因此，在踢資料時，會優先從soon queue的LRU端開始踢，當踢光soon queue的資料或是soon queue是空的的狀況發生，就往上到mean queue LRU端找，如果mean queue也是空的，最後才會去late queue LRU端找victim block踢掉。每次資料被存取，就會被移動到該queue的MRU端，而不常被存取的資料會位於各個queue的LRU端，這時候就產生了一個問題，late queue和mean queue LRU端的資料，是否如預期【至少比soon queue LRU端的資料來的更常被存取?】但因為存在model 誤判的狀況，所以mean queue和late queue LRU端的資料，是有可能長期完全沒有被存取，單純占用write buffer的空間，為了避免這種狀況發生，我們在online使用了【demoting】來解決這樣的情形。

在online，我們會監測每個block待在write buffer的時間，目前先以【pass\_req\_count】來稱呼它，單位與duration一樣，是以request為單位，差別在於，一旦該block被存取，pass\_req\_count就會被歸零，以這樣的方式，就能夠知道在mean queue、late queue中，有哪些block很久沒被存取，卻仍然在write buffer占空間。除了pass\_req\_count以外，我們還需要有一個threshold當作標準，當pass\_req\_count>=threshold時才會針對該block做demoting，每次有request進入write buffer就會檢查一次write buffer，看是否有需要做demoting。而目前這篇論文的threshold是設定為4000(request)。

這樣的做法，能夠確保write buffer內所有block的 pass\_req\_count不會超過4000(request)，因為一旦超過，就會被demoting，如果沒有被存取的話，很快就會從write buffer被踢掉，如此一來，就能確保write buffer內的資料都是真正必要的資料。

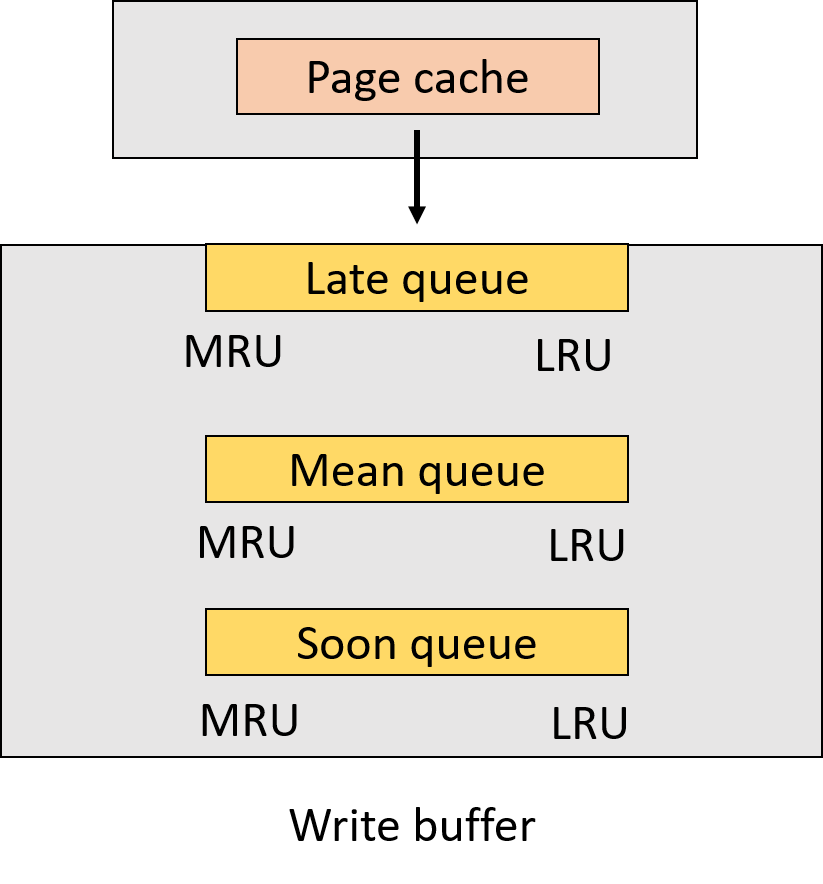


圖 3‑7 our write buffer data placement

## AI與hint 資訊做結合

儘管我們在AI做了online demoting與offline training，但仍然可能因為model無法徹底掌握使用者request的規則，而造成表現不佳。為了改善這樣的問題，我們除了使用AI以外，更將AI與hint 資訊做結合。如此一來我們一方面使用model預測，另一方面使用hint 資訊，有時候可能model會預測錯誤，有時候可能hint 資訊會預測錯誤，但透過model與hint資訊交叉比對，我們就能夠選出最佳的victim block。

但如何知道應該依照hint 或者是model的資訊?下一章節，我們將會詳細介紹。

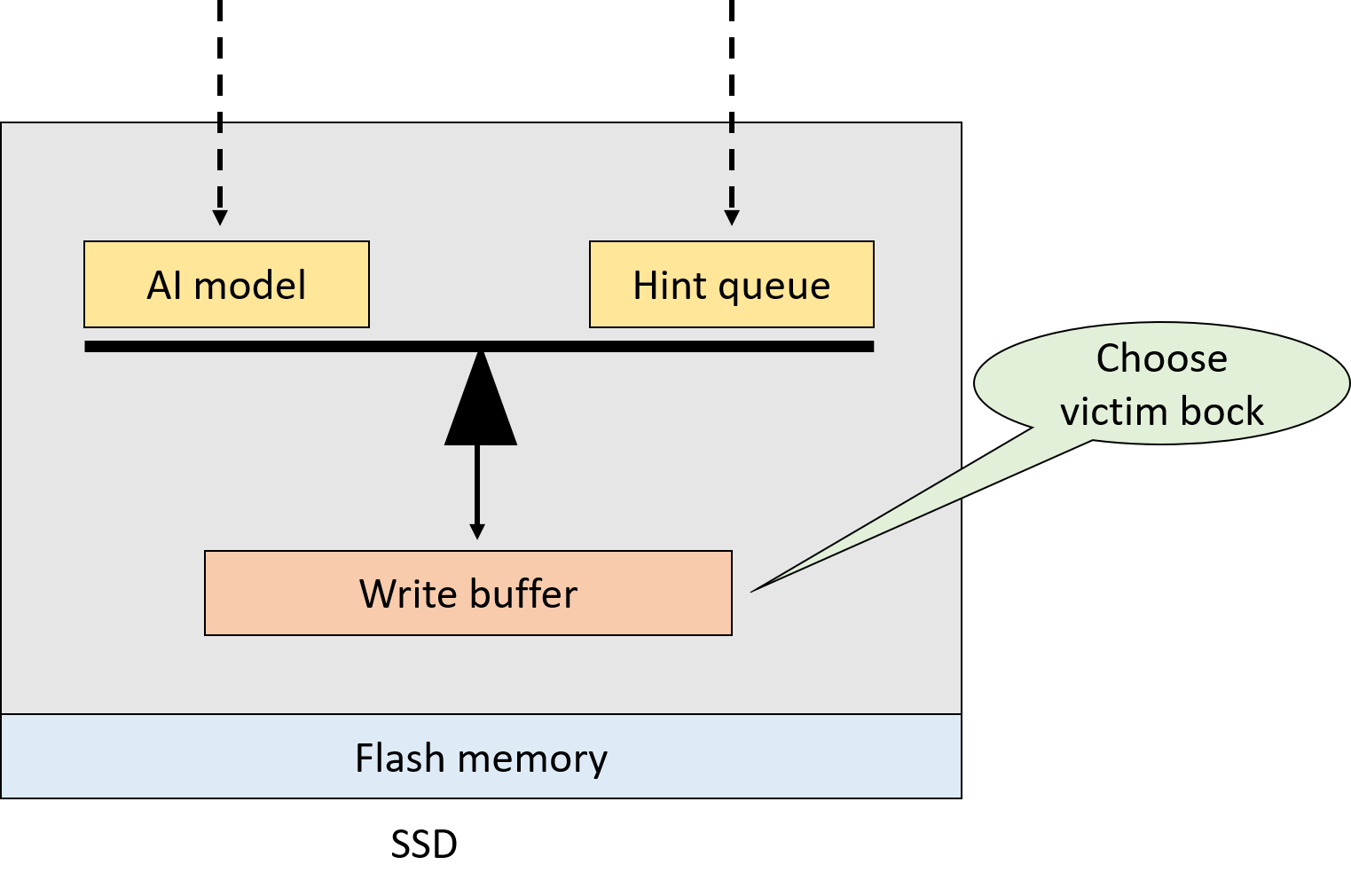


圖 3‑8 AI+Hint架構圖

### 選擇victim block

如圖3-8，由於我們不知道該依照AI的資訊或是Hint queue的資訊，因為兩邊都是預測的資訊，只是預測的內容不太一樣，因此，我們透過實驗來決定AI與hint quque各自的權重。

首先介紹上面的公式: 【profit】，這個公式也是以block為單位，因為目的是為衡量victim block的重要性；【p】是一個0~1之間的機率，在這裡當成權重使用，目的是為了得知到底model預測比較能夠影響效能，還是hint queue的資訊；【priority】是為了將LRU queue數字化，簡單來說，將soon queue(假設soon queue是有資料可以踢)LRU端的資料設為0.01，然後接著往MRU的方向，每次遇到一個新的block就遞增0.01，如下圖3-9；最後是【Hint】，也就是預測會被寫入write buffer的page中，寫進block x的總和，這裡的寫進block x不單單指overwrite，也包含new write，也就是該page當下不在block x中，但之後會寫入。所以如圖3-9，E、B、C、A、D之後各自會被write(包含overwrite)5、3、11、24、21次。

介紹完公式，我們仍然不知道該將哪邊給予更高的權重，因此，我們做了很多次測試，發現當p=0.42的時候，write buffer hit ratio最高，因此我們就將p固定為0.42，並以上述公式來決定victim block：當需要挑選victim block時，挑選profit值最小的block當成victim block。

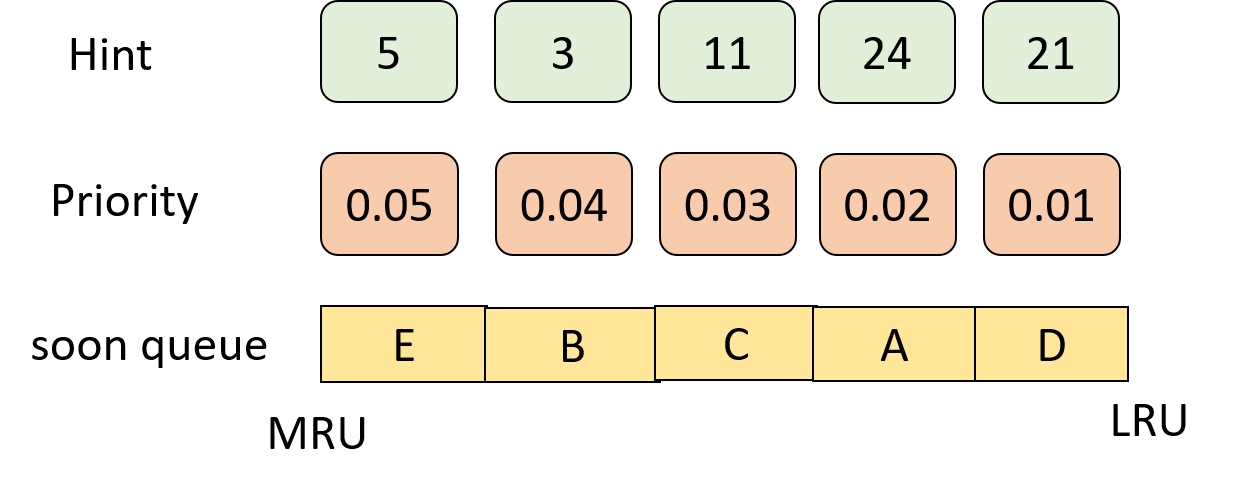


圖 3‑9 AI+Hint資訊

### 過早踢除

雖然我們看似已經能夠選出合適的victim block，但仍然存在一個問題: 【過早踢除】，因為被我們選中的victim block有可能位於MRU端，因此未來會再度被存取。

然而，雖然看起來是這樣，其實這個問題並不存在，因為Hint queue預測的是未來的資訊，而AI model預測的是當下的資訊因此write buffer內看到的狀況，和實際的狀況並不相符。如圖3-10，乍看之下我們選中了block B位於MRU端，高機率會再度被存取，但是為什麼過早踢除仍然能夠有更高的hit ratio?

關鍵在於Hint queue所存放的(預測的)是未來的資訊，代表即將寫入，但目前還沒有寫入的資料，而透過profit這個公式得知，我們會選擇Hint盡可能小的同時盡可能選擇位於LRU端的資料，若兩者衝突，將會以Hint為優先(因為Hint 權重更高一點)，因此若選擇block B代表block B未來被寫入的次數應該很少。未來不會寫入block B，但仍然有request會進入write buffer，那被寫入的就會是除了block B以外的block(C、A、D、E)。所以在未來的時間點，block B會被其他block 擠到LRU端，因此victim block 選擇block B是正確的。

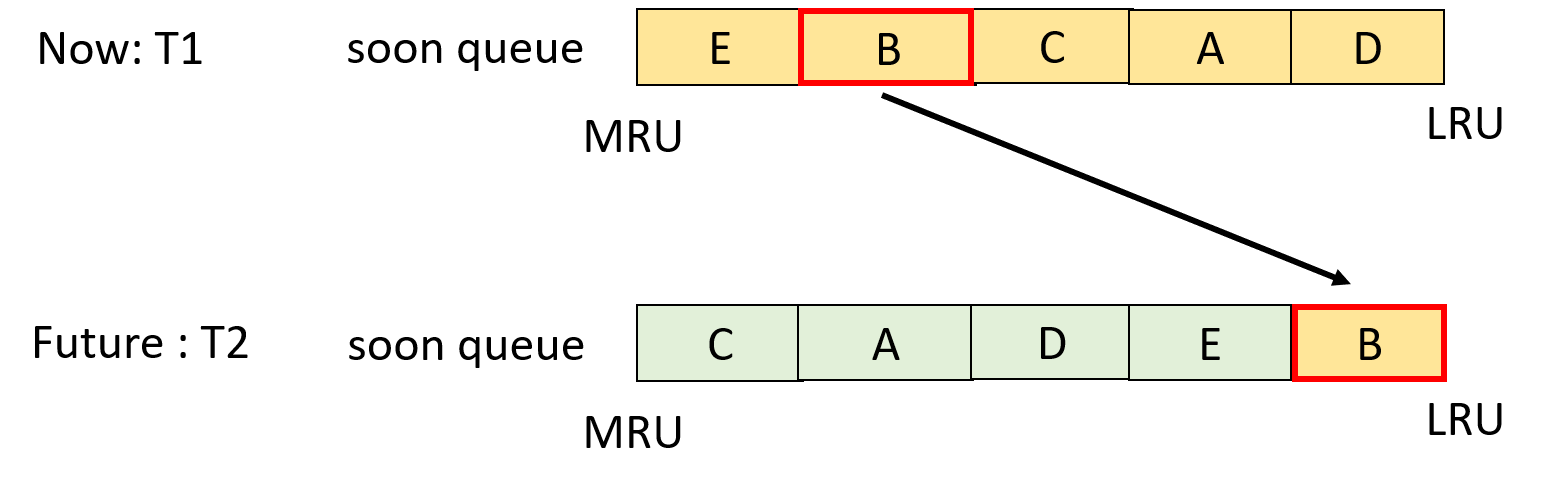


圖 3‑10 誤判的狀況

# 第四章 實驗結果與討論(以下是學姊的論文…..我還沒有寫到第四章)

## 實驗環境

我們在Page Cache中所使用的置換策略為LRU，另外將flush機制[11]加入於Page cache中，促使模擬更趨近於現實狀況，我們與第二章提到的BLAS [5]、LPB [6]做比較。在表4 - 1中是我們實驗中所使用的trace，簡略的描述trace的相關特性。本篇論文使用DiskSim [9]模擬器，用來模擬固態硬碟。

表 4‑1　Trace特性說明

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Trace** | **Number of request** | **page access count** | **write ratio** | **Avg. read/write size** |
| moodle | 844933 | 1552240 | 58.14% | 2.08/1.69 |
| UG-fileserver-1 | 444779 | 1371169 | 24.21% | 4.30/1.63 |
| UG-fileserver-2 | 525414 | 1512067 | 27.56% | 3.14/2.34 |
| User-1 | 1153878 | 1288121 | 27.11% | 1.11/1.12 |
| User-2 | 1091901 | 1232075 | 21.06% | 1.14/1.09 |

這些trace是從Storage Networking Industry Association(SNIA) 獲得的，moodle trace是在運行Linux的Web和DB server上進行一天的紀錄，UG-fileserver trace是在NFS/CIFS fileserver上進行一天的紀錄，User trace是在運行Linux的 User Desktop上進行一天的紀錄。

## 實驗結果

### 參數測試

### 實驗結果

圖4 – 3到圖4 – 7中，我們比較的trace為moodle、UG-flieserver-1、UG-flieserver-2、User-1、User-2，Y軸代表Write buffer中的write hit ratio，X軸代表Write buffer size，可以看到在4個trace中我們的方法都能呈現較好的write hit ratio，表示利用主機端的資訊能夠有效保留價值較高的資料，踢除較無用的資料。

# 第五章 結論及未來工作

本篇論文提出透過傳遞page cache的資訊，達到跨階層合作，藉由參考Page cache傳遞的資訊，決定資料在SSD中的擺放方式，並根據擺放方式的特性，再加上預測未來可能寫入的page 資訊，定義出挑選踢除資料的規則，使Write buffer需要執行置換策略時可以更準確的判斷出價值性較低的block踢除，進而降低寫入快閃記憶體的次數以及更佳利用channel平行處理的效果，降低response time，有效提升整體系統效能。由模擬的實驗結果可以看到，本篇論文之方法可以有效提升Write buffer內的hit ratio，同時，降低從Write buffer踢除的次數、Average response time、GC overhead。

未來工作的部份，Eviction window size該如何根據trace的特性調整為其中一個重要的考量，也可以參考LPB [6]的physical group設計，考慮到資料在SSD中的實際擺放狀況，以降低GC cost。

# 參考文獻

|  |  |
| --- | --- |
| [1] | “Understanding the Linux Kernel,3rd Edition,” [線上]. Available: http://www.johnchukwuma.com/training/UnderstandingTheLinuxKernel3rdEdition.pdf. |
| [2] | H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, “FAB: Flash-aware buffer management policy for portable media players,” *IEEE Trans. Consumer Electron., vol. 52, no. 2,* p. 485–493, May 2006. |
| [3] | H. Kim and S. Ahn, “BPLRU: A buffer management scheme for improving random writes in flash storage,” *Proc. 6th USENIX Conf. File Storage Technol.,* p. 1–14, 2008. |
| [4] | Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha, “Performance Trade-Offs in Using NVRAM Write Buffer for Flash Memory-Based Storage Devices,” *IEEE Trans. Comput., vol 58, no. 6,* pp. 744-758, Jun 2009. |
| [5] | D. W. Chang, H. H. Chen, D. J. Yang, and H. P. Chang, “BLAS: Block-Level Adaptive Striping for Solid-State Drives,” *ACM Transactions on Design Automation of Electronic Systems, Vol. 19, No2,* March 2014. |
| [6] | Sheng-Min Huang and Li-Pin Chang, “Exploiting Page Correlations for Write Buffering in Page-Mapping Multichannel SSDs,” *ACM Trans. Embedded Comput. Syst.,* pp. 15(1): 12:1-12:25, 2016. |
| [7] | D. Kim, K. H. Park, and C.-H. Youn, “SUPA: A Single Unified Read-Write Buffer and Pattern-Change-Aware FTL for the High Performance of Multi-Channel SSD,” *ACM Transactions on Storage, Vol. 13, No. 4, Article 32,* November 2017. |
| [8] | J.S Park, H.K Bahn and K. Koh, “Buffer Cache Management for Combined MLC and SLC Flash Memories Using both Volatile and Nonvolatile RAMs,” *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications,* pp. 228-235, 2009. |
| [9] | “DiskSim,” [線上]. Available: http://www.pdl.cmu.edu/DiskSim/. |
| [10] | S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, “CFLRU: A replacement algorithm for flash memory,” *Proc. Int. Conf. Compil., Arch. Synth. Embed. Syst.,* p. 234–241, 2006. |
| [11] | J. Seol, H. Shim, J. Kim, and S. Maeng, “A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks,” *Proc. Int. Conf. Compil., Arch., Synth. Embed. Syst.,* pp. 137-146, 2009. |
| [12] | L. Shi, C. J. Xue, and X. Zhou, “Cooperating write buffer cache and virtual memory management for flash memory based systems,” *Proc. 17th IEEE Real-Time Embed. Technol. Appl. Symp.,* pp. 147-156, Apr 2011. |
| [13] | Wu, Guanying, Xubin He, and Ben Eckart, “An adaptive write buffer management scheme for flash-based ssds.,” 於 *ACM Transactions on Storage (TOS) 8.1*, 2012. |
| [14] | L. Shi, J. Li, C. J. Xue, C. Yang, and X. Zhou, “EXLRU: A unified write buffer cache management for flash memory,” *Proc. ACM Int. Conf. Embed. Softw.,* pp. 339-348, 2011. |
| [15] | Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, “CCF-LRU: A new buffer replacement algorithm for flash memory,” *IEEE Trans. Consumer Electron., vol. 55, no. 3,* pp. 1351-1359, Aug 2009. |
| [16] | J. Seol, H. Shim, J. Kim, and S. Maeng, “A buffer replacement algorithm exploiting multi-chip parallelism in solid state disks,” *Proc. Int. Conf. Compil., Arch., Synth. Embed. Syst.,* pp. 137-146, 2009. |