國立中興大學資訊科學與工程學系

碩士學位論文

利用長短期記憶體和主機資訊來選擇寫入緩衝的犧牲區塊

Using LSTM and Host information for selecting write buffer victim block

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。

由於使用者所下的request屬於時序性，也就是說，前面的request與後面的request有一定關係，而要預測這樣的資訊，一般神經網路無法做到，因為一般的神經網路沒辦法得知上一個時間點的input，因此這篇論文，我們使用的是RNN，RNN與一般神經網路不同的地方在於，能夠記住上一個時間點的資訊，相較於其他神經網路更適合解決使用者I/O相關問題。

但是RNN並不能做到100%完美預測，因此我們在online也做了優化，為了避免model誤判使不必要的資料持續被放在write buffer，我們針對很久沒被存取的資料做Demoting，讓那些資料能夠漸漸地從write buffer被踢除，盡可能減少model誤判所帶來的影響。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，也就是預測會被踢到SSD的page number，可以更準確地找出踢除的對象。

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 uses a RAM-based write buffer to reduce flush memory write count and improve lifetime and performance. However, due to the limit of write buffer capacity, the hit ratio is not good enough when facing random and large user requests.

This paper uses RNN(recurrent neural network) to improve the write buffer hit ratio. Unlike common neural networks, RNN has internal memory that stores previous state information and is more suitable for solving user I/O-related problems.

But RNN model is not perfect, so we use online demoting for misjudging data, which will move data to the next queue(late->mean, mean->soon) if data has not been used for a long time. Online demoting can flush unnecessary data from the write buffer and lower the influence caused by model misjudging.

At the same time, we use the host information, which records dirty page numbers that flush to the write buffer. With host information, we can select victim block more accurately.

目錄

[第一章 緒論 1](#_Toc105769658)

[1.1 簡介 1](#_Toc105769659)

[1.2 研究動機 1](#_Toc105769660)

[1.3 貢獻 2](#_Toc105769661)

[1.4 論文架構 2](#_Toc105769662)

[第二章 背景知識與相關研究 3](#_Toc105769663)

[2.1 SSD 3](#_Toc105769665)

[2.2 Write buffer管理方式 4](#_Toc105769666)

[2.2.1 FAB(Flash-aware-buffer management policy) 5](#_Toc105769667)

[2.2.2 BPLRU(Block Padding Least Recently Used) 5](#_Toc105769668)

[2.2.3 Physical block-based write buffer 6](#_Toc105769669)

[2.3 Neural network 8](#_Toc105769670)

[2.4 RNN 9](#_Toc105769671)

[2.4.1 LSTM 10](#_Toc105769672)

[第三章 系統架構與實作方法 14](#_Toc105769673)

[3.1 系統架構 14](#_Toc105769674)

[3.2 AI運作流程 15](#_Toc105769675)

[3.2.1 Offline 15](#_Toc105769676)

[1. Benefit function 16](#_Toc105769677)

[2. Data collected 16](#_Toc105769678)

[3. Transform duration value into duration label 17](#_Toc105769679)

[4. Offline training 18](#_Toc105769680)

[3.2.2 Online 19](#_Toc105769681)

[1. Demoting 19](#_Toc105769682)

[3.3 AI與Hint 資訊做結合 20](#_Toc105769683)

[3.3.1 選擇victim block 21](#_Toc105769684)

[3.3.2 過早踢除 22](#_Toc105769685)

[第四章 實驗結果 23](#_Toc105769686)

[4.1 實驗環境 23](#_Toc105769687)

[4.2 實驗結果 23](#_Toc105769688)

[4.2.1 參數測試 23](#_Toc105769689)

[4.2.2 實驗結果 23](#_Toc105769690)

[第五章 結論及未來工作 24](#_Toc105769691)

[參考文獻 25](#_Toc105769692)

圖目錄

[圖 2‑1 SSD架構 4](#_Toc105769693)

[圖 2‑2 FAB 架構 5](#_Toc105769694)

[圖 2‑3 BPLRU架構 6](#_Toc105769695)

[圖 2‑4 Page Padding 6](#_Toc105769696)

[圖 2‑5 Physical Block-Based Write Buffer 7](#_Toc105769697)

[圖 2‑6 Logical Data placement 8](#_Toc105769698)

[圖 2‑7 Neural Network 9](#_Toc105769699)

[圖 2‑8 Recurrent Neural Network 10](#_Toc105769700)

[圖 2‑9 LSTM 11](#_Toc105769701)

[圖 2‑10 LSTＭ input 12](#_Toc105769702)

[圖 2‑11 LSTM Architecture (多個LSTM單元，在單一時間點內的狀況) 13](#_Toc105769703)

[圖 2‑12 LSTM unfold Architecture(一個LSTM單元，多個時間點的狀況) 13](#_Toc105769704)

[圖 3‑1 系統架構圖 14](#_Toc105769705)

[圖 3‑2 AI架構圖 15](#_Toc105769706)

[圖 3‑3 Offline架構 15](#_Toc105769707)

[圖 3‑4 Write buffer simulator 17](#_Toc105769708)

[圖 3‑5 Generate Duration Label 18](#_Toc105769709)

[圖 3‑6 Online架構 19](#_Toc105769710)

[圖 3‑7 Our Write Buffer Data Placement 20](#_Toc105769711)

[圖 3‑8 AI+Hint架構圖 21](#_Toc105769712)

[圖 3‑9 AI+Hint資訊 22](#_Toc105769713)

[圖 3‑10 誤判的狀況 22](#_Toc105769714)

表目錄

[表 2‑1 快閃記憶體的基本操作單位 3](#_Toc103975273)

[表 4‑1　Trace特性說明 24](#_Toc103975274)

# 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，若想原地覆寫一個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，而write buffer要產生作用，勢必需要具有一定的hit ratio，因此選擇victim block就很重要。

## 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write buffer的管理方式，2.3介紹physical block-based write buffer，2.4介紹neural network的運作，2.5介紹LSTM



## 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。

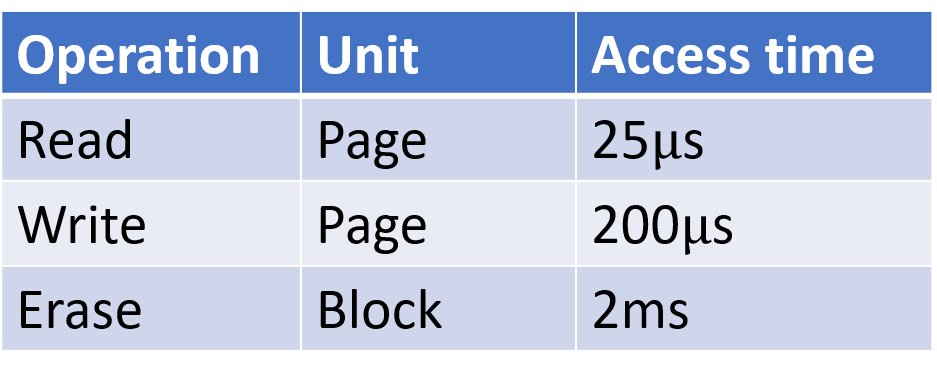


表 2‑1 快閃記憶體的基本操作單位

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的空間與時間，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

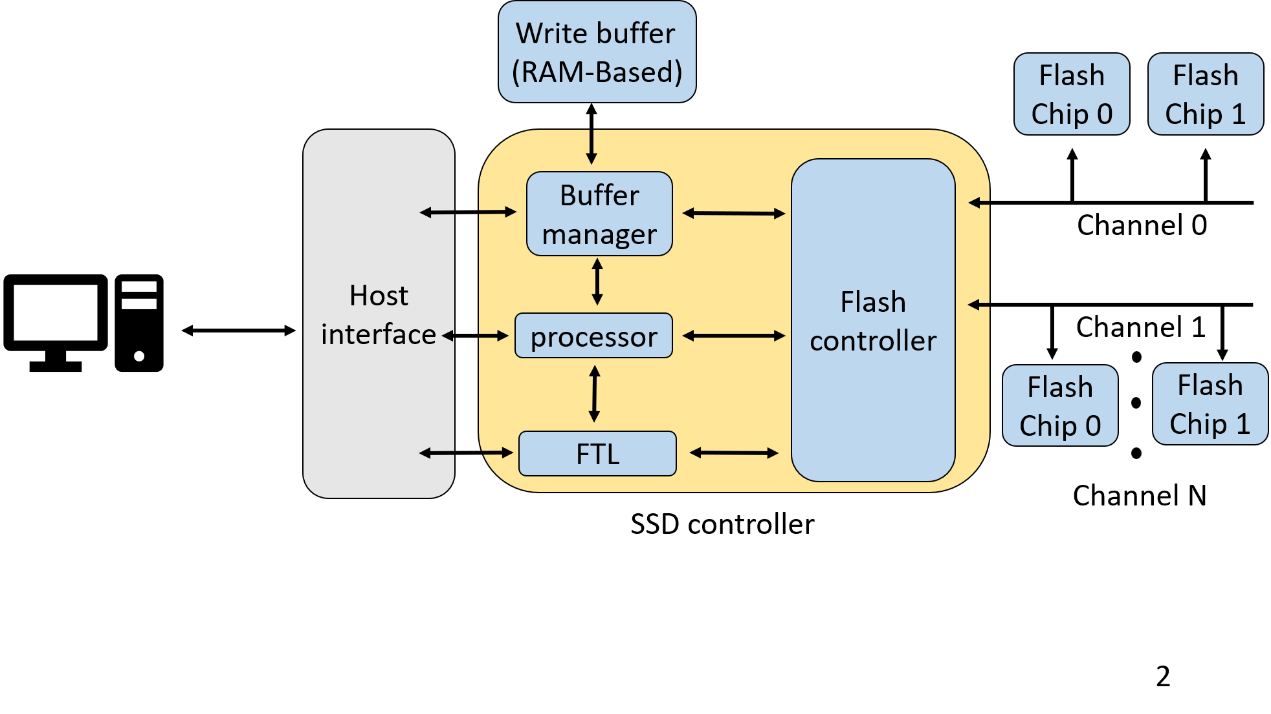


圖 2‑1 SSD架構

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。

## 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。

### 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。

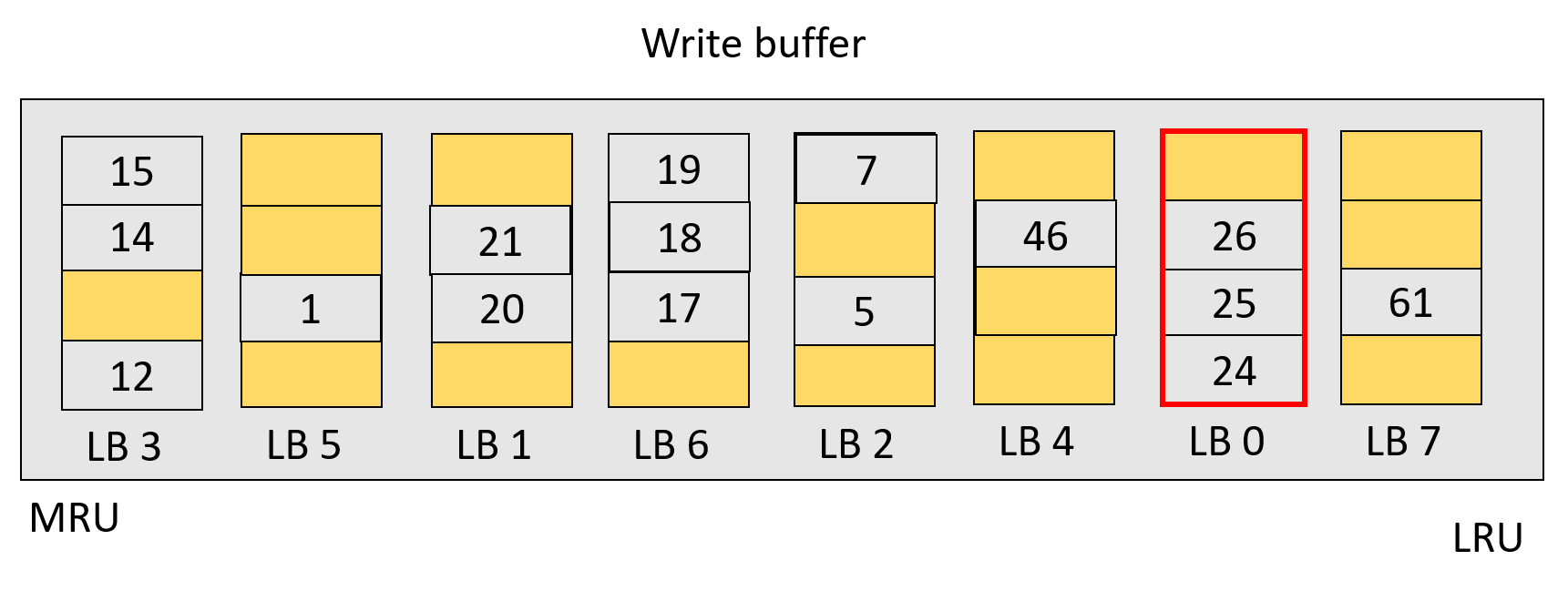


圖 2‑2 FAB 架構

### 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(如圖2-5)，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 0但因為block 0不是一個完整的block，此時會啟動【Page padding】，將block 缺少的部分(page27)從NADN flash memory讀上來，讓原先的block成為一個invalid block，空間不足時，就可直接擦除原先的block，然後再將block 0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。

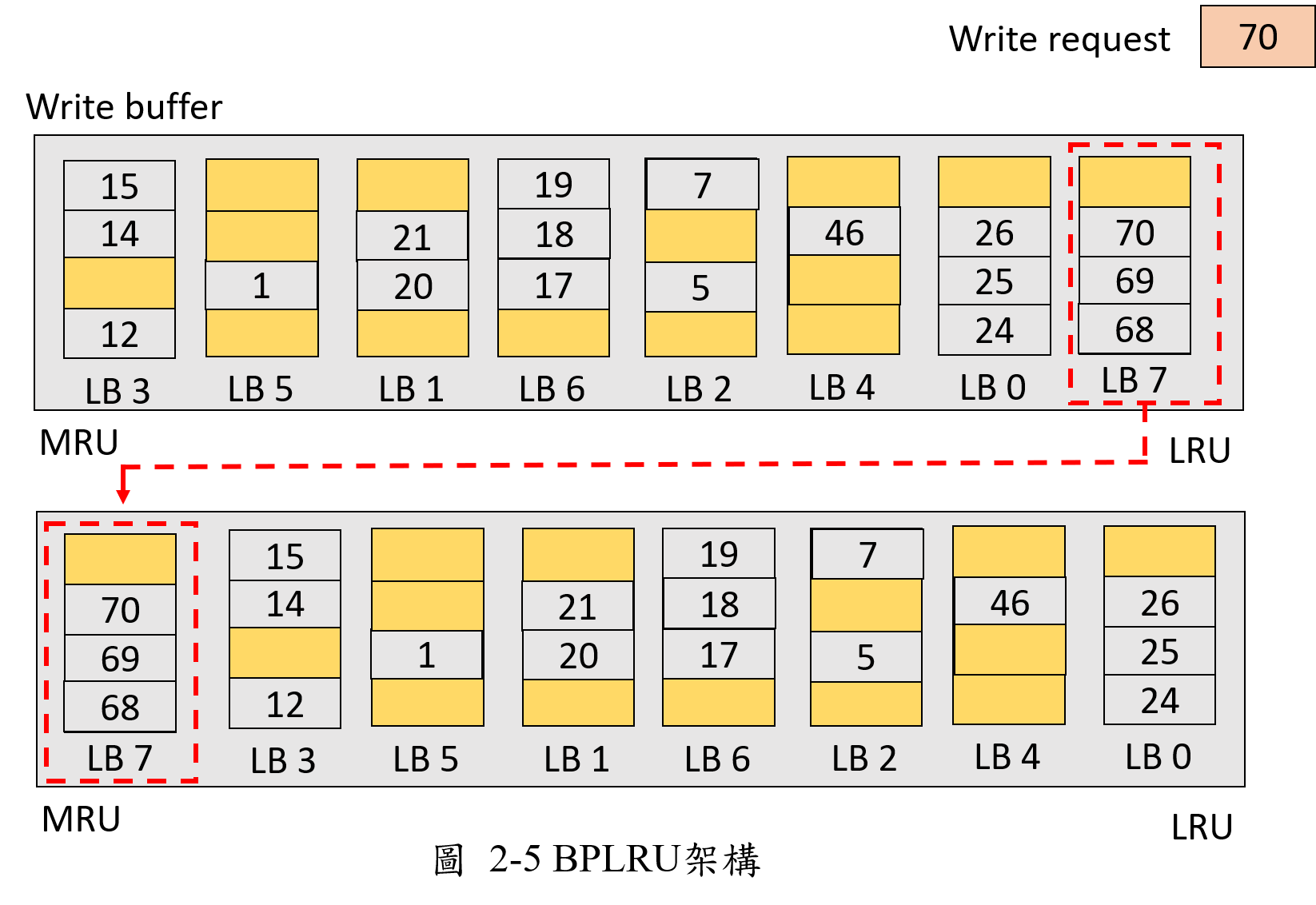


圖 2‑3 BPLRU架構

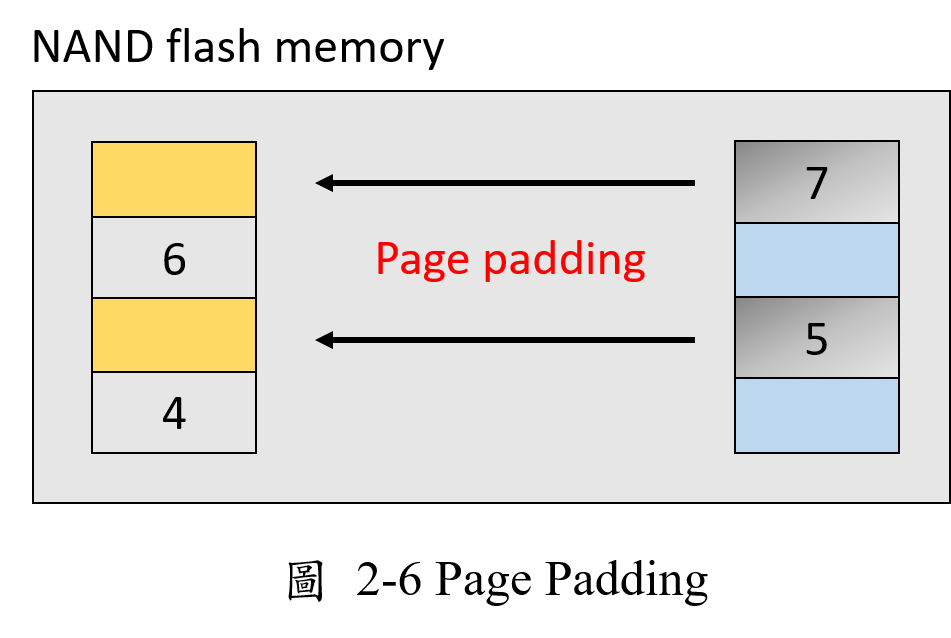


圖 2‑4 Page Padding

### Physical block-based write buffer

Physical block-based write buffer為了能更準確的挑選victim block，write buffer在挑選victim block時，會參考Hint queue資訊。

Page cache會在三種狀況下將資料踢到SSD：

1. Replacement：當Page cache滿了，會將資料踢到SSD，此時會優先選擇LRU端的dirty page。
2. Dirty amount：當dirty page ratio達到所制定的threshold時，會將LRU端的dirty page踢到SSD
3. Dirty time：:為了避免MRU端的dirty page一直沒被寫回，Page cache會定期監測每個dirty page待在Page cache的時間，超過某個threshold，就會將dirty page給踢下去。

根據這三種Flush方式，Physical block-based write buffer會週期性預測即將被寫入到SSD的dirty page，然後將dirty page number存在Hint queue並放入SSD給write buffer當作挑選victim block的參考，如圖2-5，如此一來，就能夠避免挑選【即將被寫入】的block為victim block。

此外，由於write buffer在選擇victim block時，為了減少GC所花的時間，會傾向於將large block給踢下去，但是write buffer通常是以logical block的方式擺放，這時候會產生一個問題，如圖2-6，logical block 4位於LRU端，同時也是large block，理論上踢下去可以節省不少GC的時間成本，但實際上logical block 4實際在Flush memory的擺放位置並不如預期，在Flush memory，Logical block 4被分散寫入到不同的physical block，如此一來，仍然會增加GC所花的時間。因此physical block-based write buffer將資料以physical block的方式放入write buffer，為了能夠在write buffer就直接得知資料實際的擺放位置，避免挑選到錯誤的victim block。

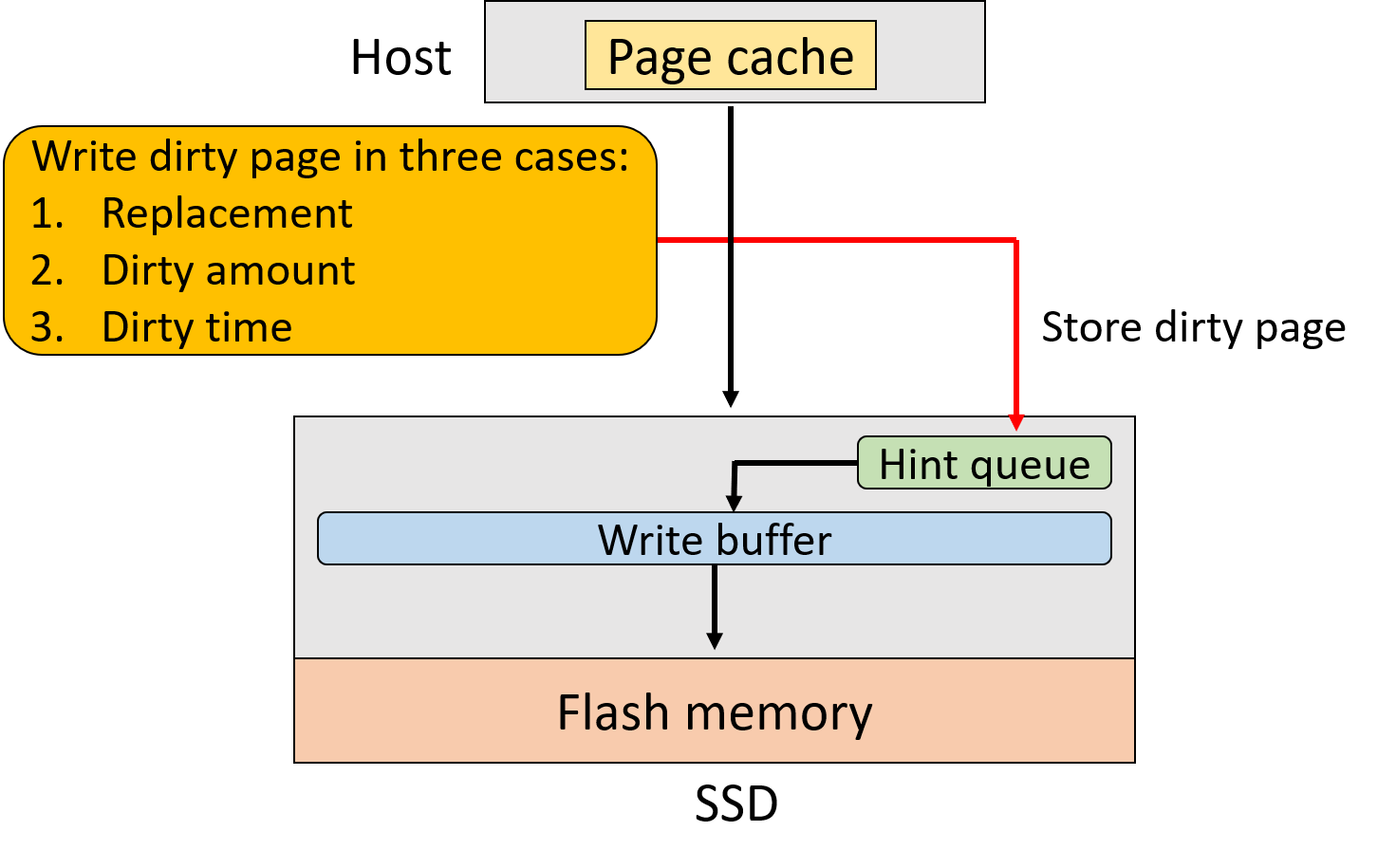


圖 2‑5 Physical Block-Based Write Buffer

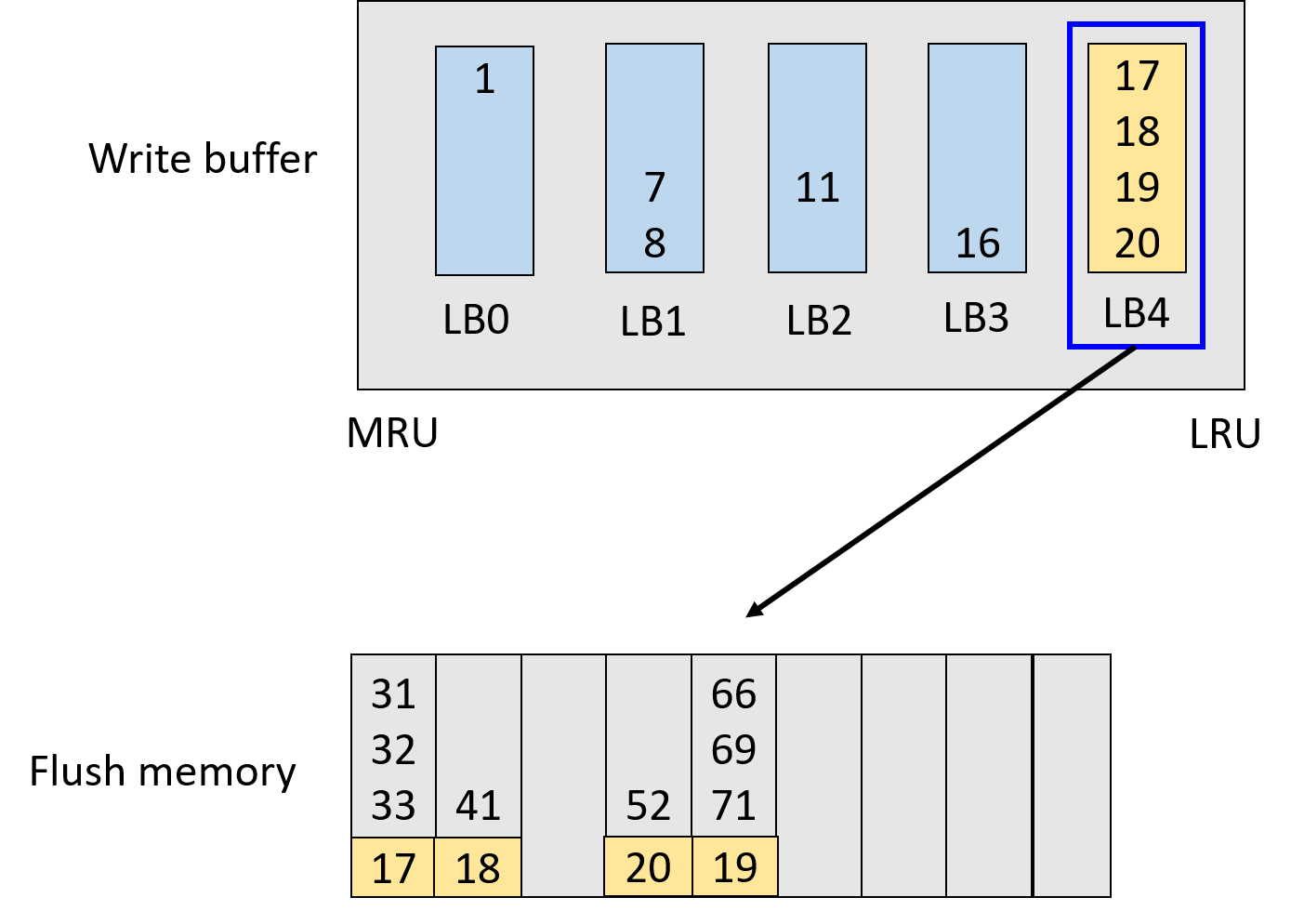


圖 2‑6 Logical Data placement

## Neural network

Neural network，顧名思義就是神經網路，神經網路的目的其實只是在【尋找function】，給它一個input，它要給你一個output，只是這個output必須符合當下任務的答案，也就是說，如果目的是預測股票售價，那這個output就必須是一個價格。但要找出這樣的function以人來說是不可能做到的，因此才需要讓機器幫我們學出來。

以圖2-7為例，我們給的input是x1, x2…xn，假設我們設定為fully connected，也就是每個input feature會對應到每一個Neural。一開始，每個input feature要送進任何一個Neural都會有一個對應的權重，以一個Neural來說，它會算出所有送進來的input feature乘上對應的權重總和再加上bias，也就是)+bias，算出的結果就是yi，bias的作用是為了讓相乘相加的結果為0時，至少有數字可以輸出。但是因為輸出的yi是一個線性函數，線性函數的輸出有所限制，逼近目標函數的能力也因此有所限制，因此我們必須讓Neural Network裡面存在非線性函數，如此才能夠逼近目標函數。

而讓model從線性轉換成非線性的關鍵，就在於圖2-7的後半段【activation function】，經過activation function這個非線性函數，可以讓輸出不受限制，加強Neural Network的表達力，也因此讓這個Neural Network可以透過學習逐步逼近目標函數。

通過activation function後，根據目的不同，輸出的形式也會有所不同，以圖2-7來說，目的是分類，因此輸出就是三維。在輸出後，會根據正確答案與預測的答案，透過Loss function來算出預測值與正確答案之間的差距，再透過Backpropagation將這資訊回傳回去，讓Neural Network調整權重，逼近正確答案。

而我們平常在講的model，其實就是在講【訓練過後找出的目標函數】，訓練的目的單純是為了盡可能逼近目標函數。

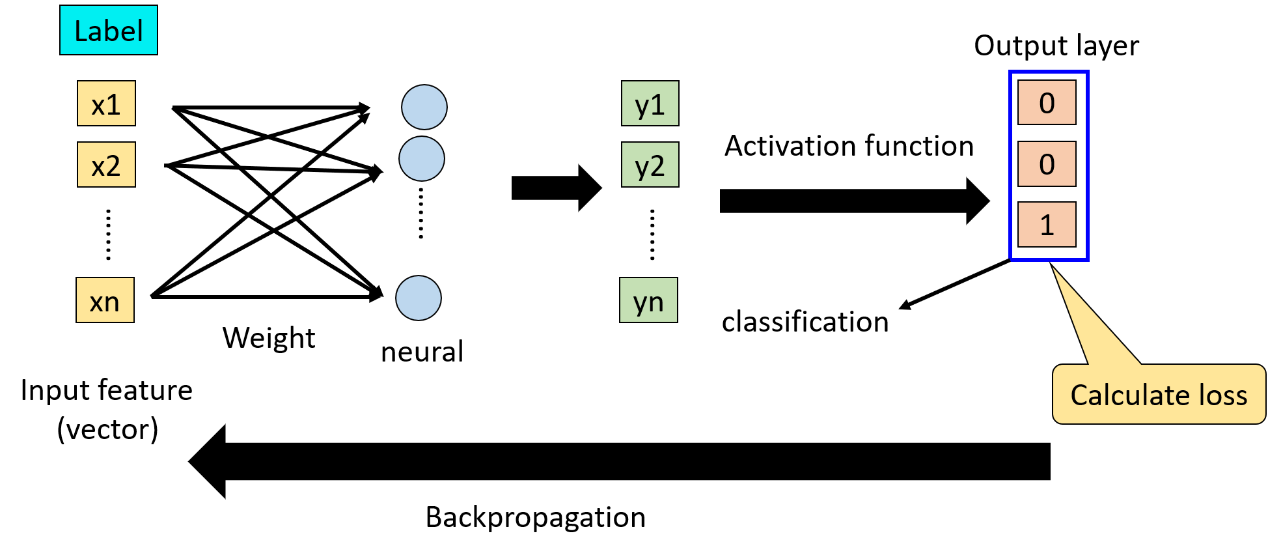


圖 2‑7 Neural Network

## RNN

RNN(recurrent neural network)，主要用於負責需要時間的任務，它相較於一般神經網路的優勢在於能夠記憶，因為一般神經網路不論多麼複雜，都只能接收當下的input，無法透過過去的資訊預測未來的結果。而RNN則是能夠處理【在一定的週期內行為規律能被預測的任務】，利用上次的資訊，去預測下次的資訊，利用下次的資訊，去預測下下次的結果。

如圖2-8，RNN簡單講其實就是能夠記憶的Neural Network，而RNN又分為simple RNN以及，LSTM，這篇論文中，我們主要使用LSTM(Long short-term memory)，因此接下來主要針對LSTM作介紹。

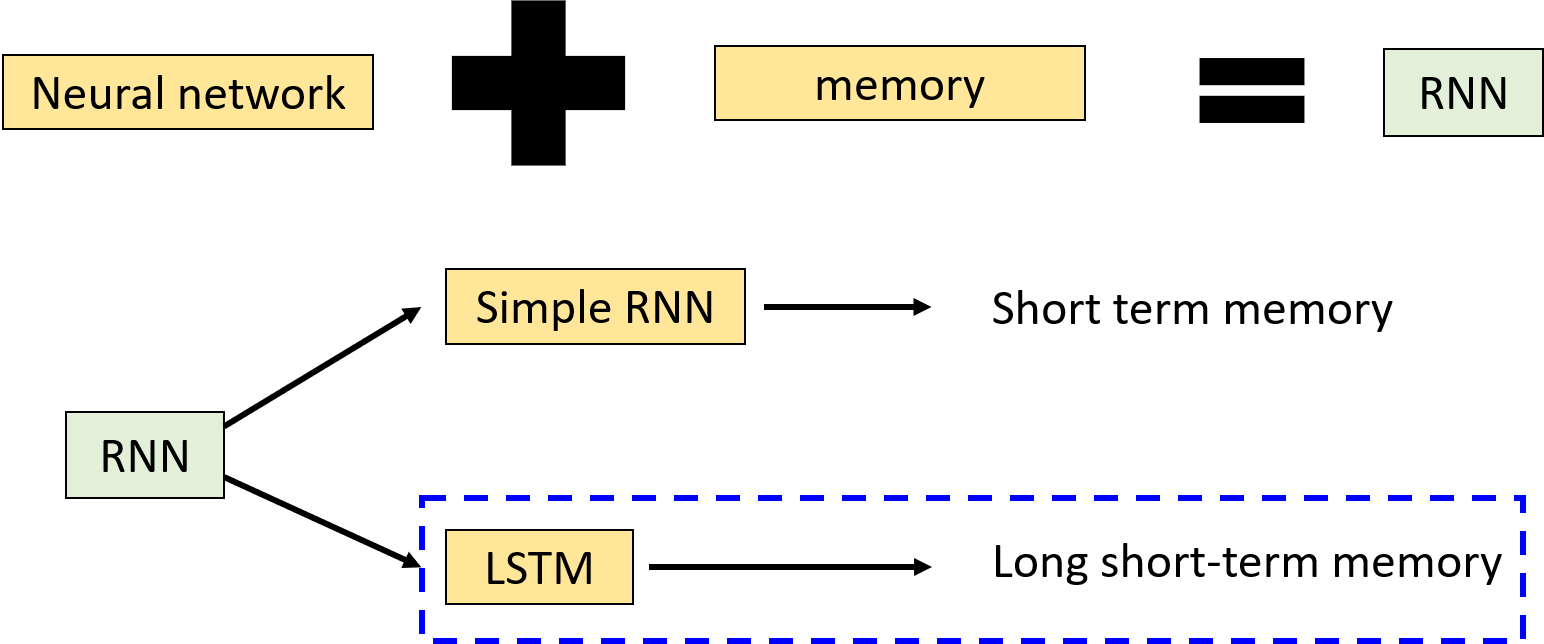


圖 2‑8 Recurrent Neural Network

### LSTM

說到LSTM(Long short-term memory)，其實就是在講RNN(recurrent neural network)，一般而言，RNN分成兩種，一種叫做simple RNN，另一種就是在講LSTM，兩者差別在於，simple RNN只能夠處理需要短期記憶的任務，而LSTM能夠處理長期記憶的任務，因此才會被稱為Long short-term memory。

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2-9，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。

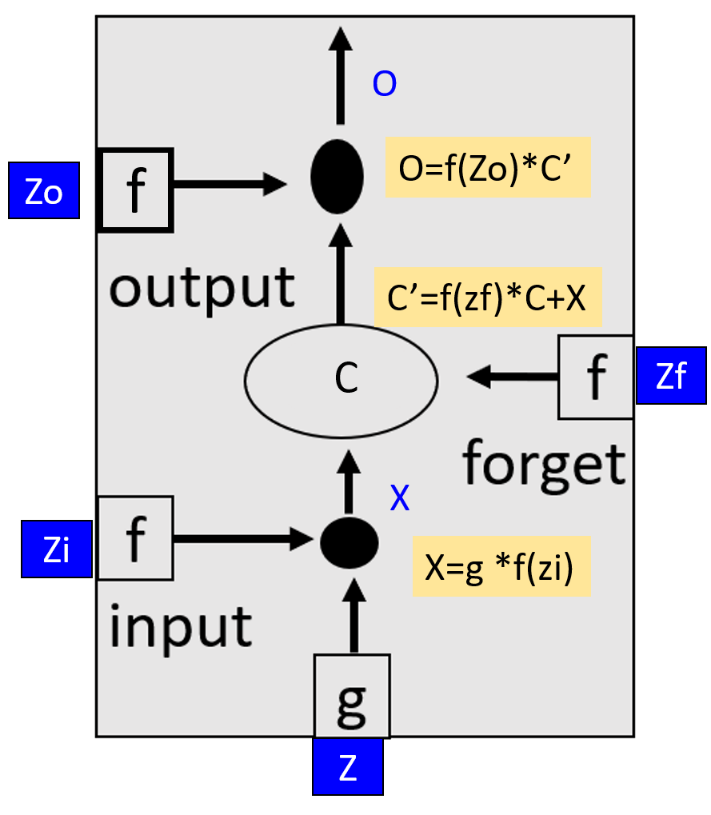


圖 2‑9 LSTM

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，會先加上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。

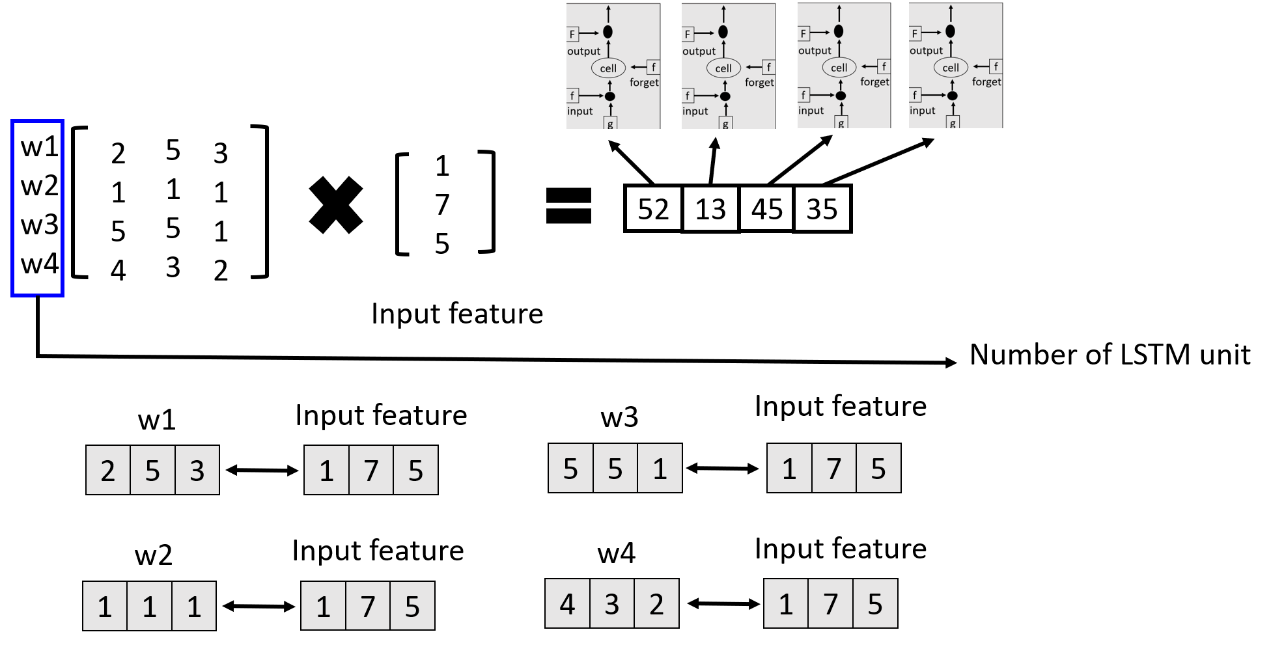


圖 2‑10 LSTＭ input

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2-10，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2-10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2-10有四組權重，那就會是4個純量，而這4個純量，會各自去操控一個LSTM unit的input gate，前面圖2-9，Zi就是4個純量中的其中一個，然後Zo, Zf, Z也是以相同方式算出來的。

由於每個時間點輸入的資訊不同，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，合併為一個向量，以圖2-11來說，input feature X乘上4個不同的權重矩陣後，會得到4個不同的vector Zi, Zo, Zf, Z，分別用來控制LSTM的4個gate，每個時間點這四個vector會同時控制多個LSTM unit，每經過一個time stamp，memory cell的資訊就會被下一個時間點的資訊所取代，有可能會包含上一個時間點的memory cell資訊，也有可能不包含，如圖2-9、2-12，經過了多個時間點，最終每個LSTM unit會output一個純量，以圖2-10為例，就是4個純量，然後將這四個純量合併為一個向量，如圖2-11。

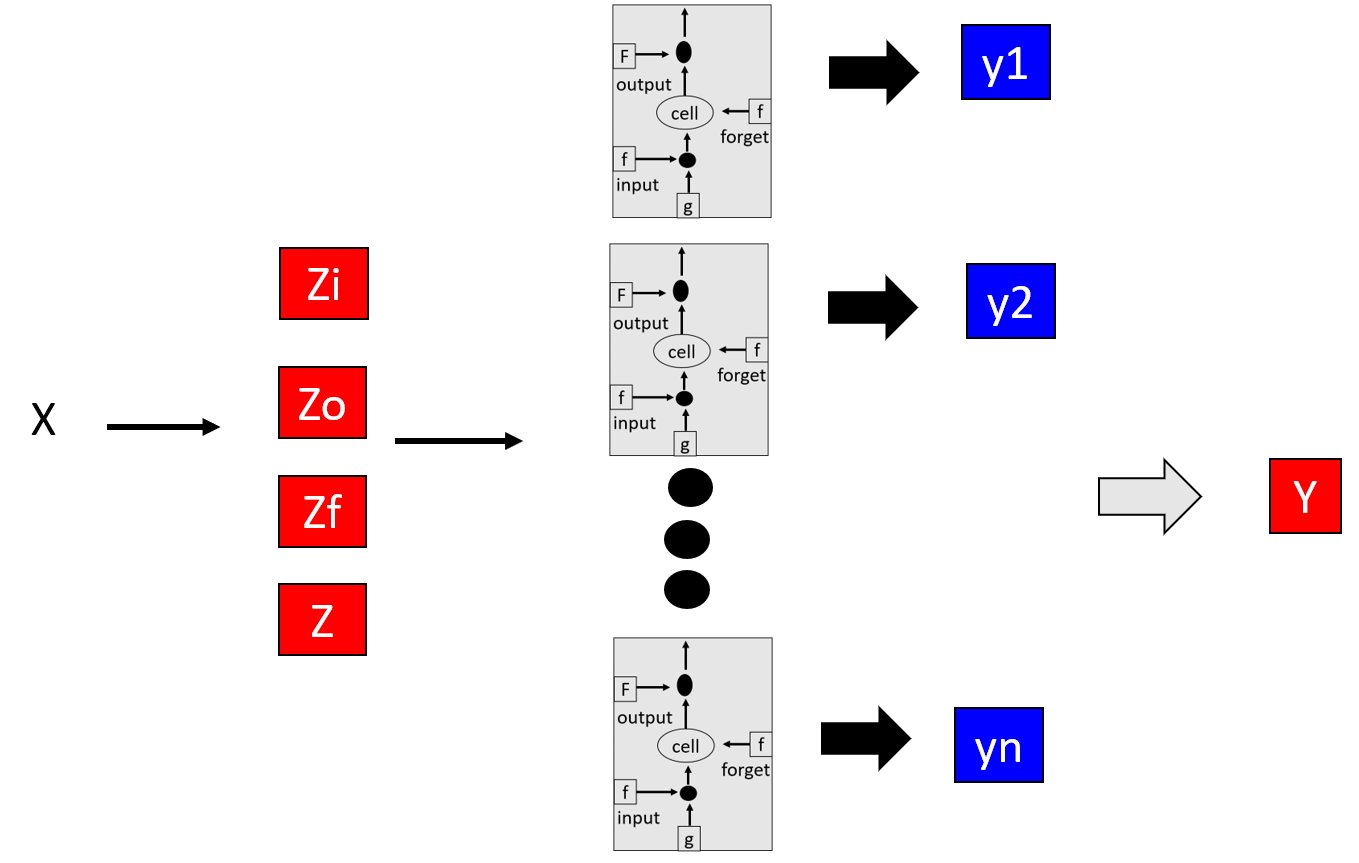


圖 2‑11 LSTM Architecture (多個LSTM單元，在單一時間點內的狀況)

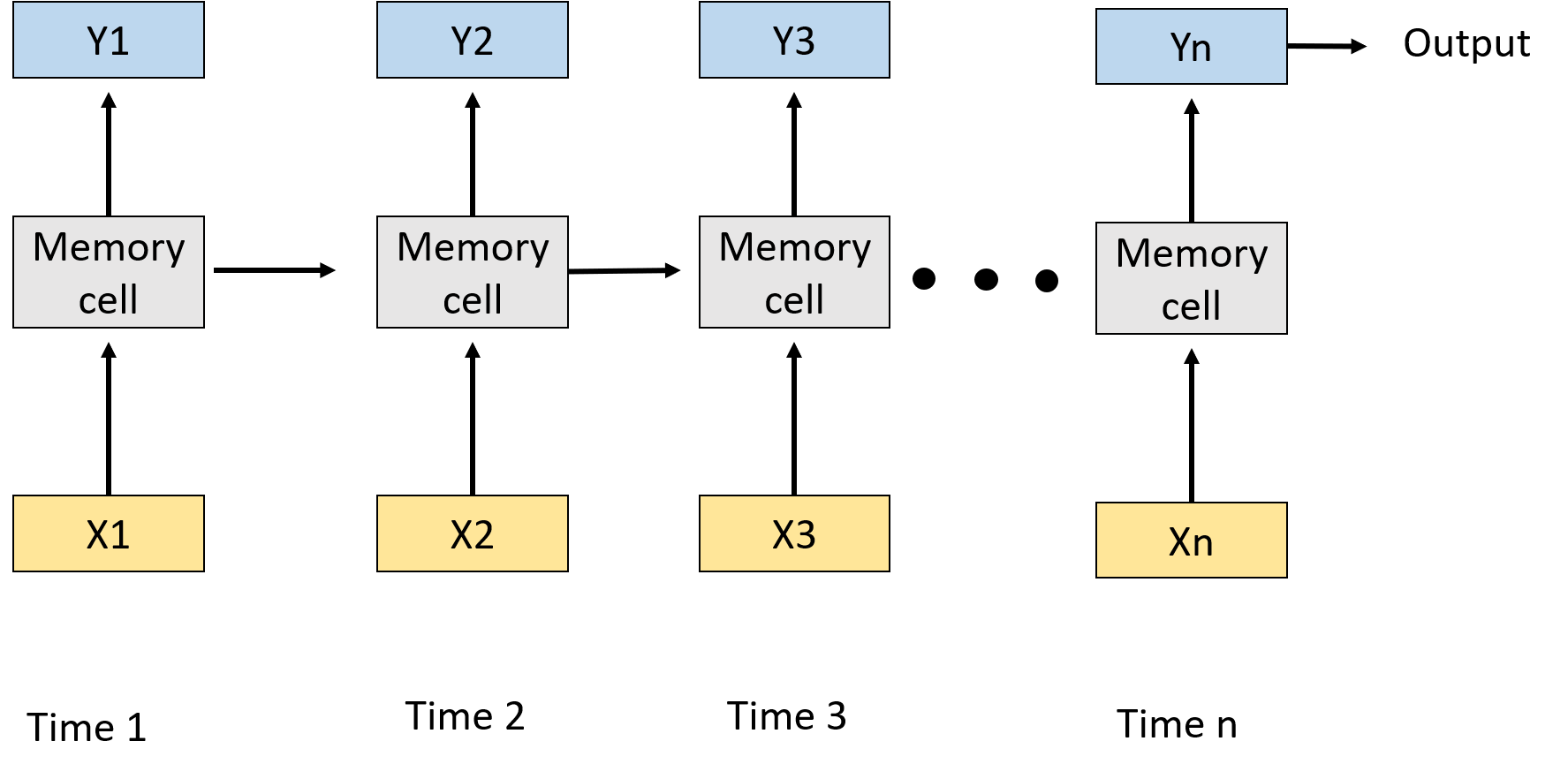


圖 2‑12 LSTM unfold Architecture(一個LSTM單元，多個時間點的狀況)

# 第三章 系統架構與實作方法

## 系統架構

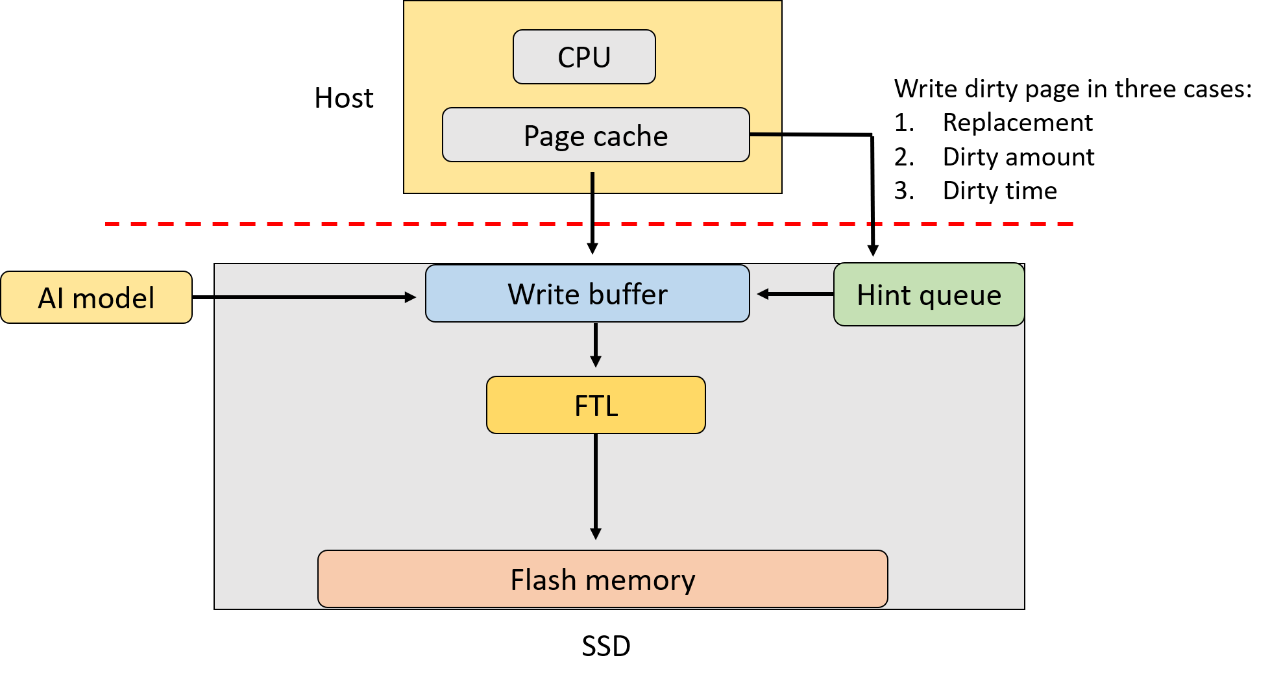


圖 3‑1 系統架構圖

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】【Dirty page ratio在Page Cache待超過一定時間】【Dirty page ratio超過某個threshold】，藉由週期性預測Page Cache的Dirty Page，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

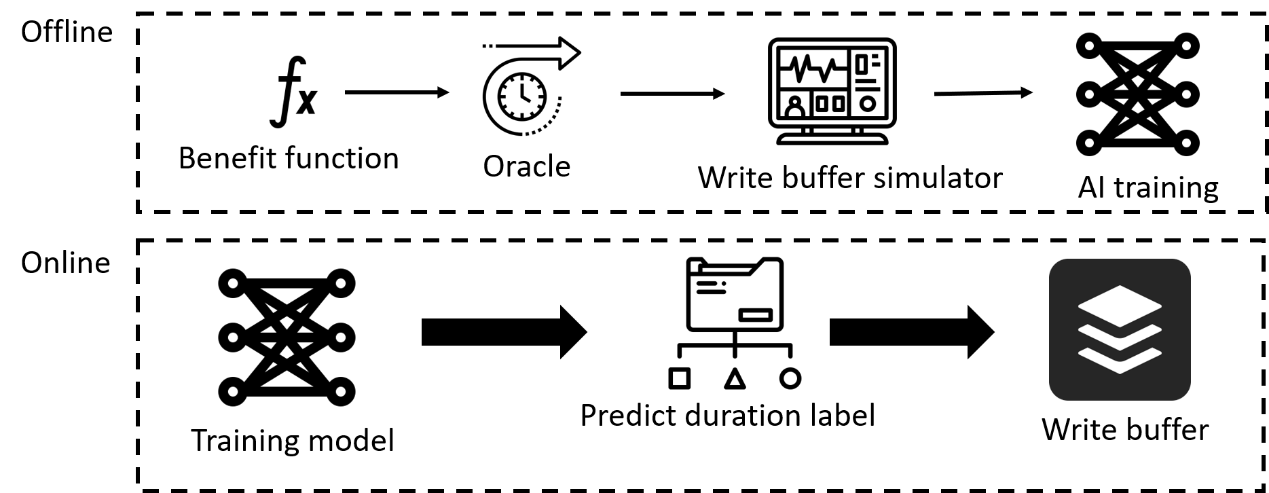


圖 3‑2 AI架構圖

AI 會分成online testing、offline training兩階段，如圖3-2，在offline 部分，會先跑整個trace一次，獲得每個block的benefit value，接著進入write buffer simulator，產生出duration value，先將duration value轉成duration label形式，再將duration label送進AI model當成比對用的label做訓練。

Online部分，會利用訓練好的model預測當下進入write buffer的request，依照soon, mean, late做區分，之後write buffer就可以依照預測出來的duration label來選擇victim block。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，如Late queue，這時候就會藉由Demoting，漸漸將那些不常被存取的block從write buffer踢掉。

### Offline

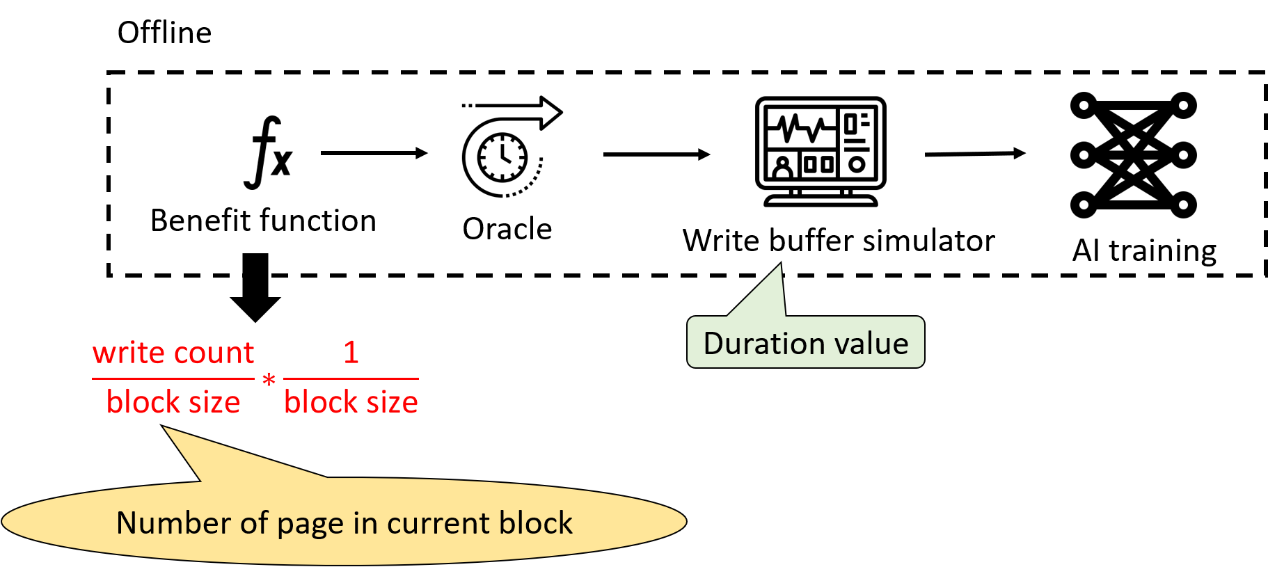


圖 3‑3 Offline架構

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，如圖3-3當中的oracle，我們為了掌握每個block的大小以及被寫入的次數，我們會先跑過整個trace一次，將相關資訊都存起來，為了建構出每個block的benefit value，接著，才會進入到write buffer simulator的部分，在獲得每個block的benefit value 之後(也就是未來的資訊)，我們會在write buffer simulator中跑一次，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，因為request單位是【Page】，因此，只要有request進入write buffer，現存在於write buffer的每一個block的duration value都會被累加。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3-4。

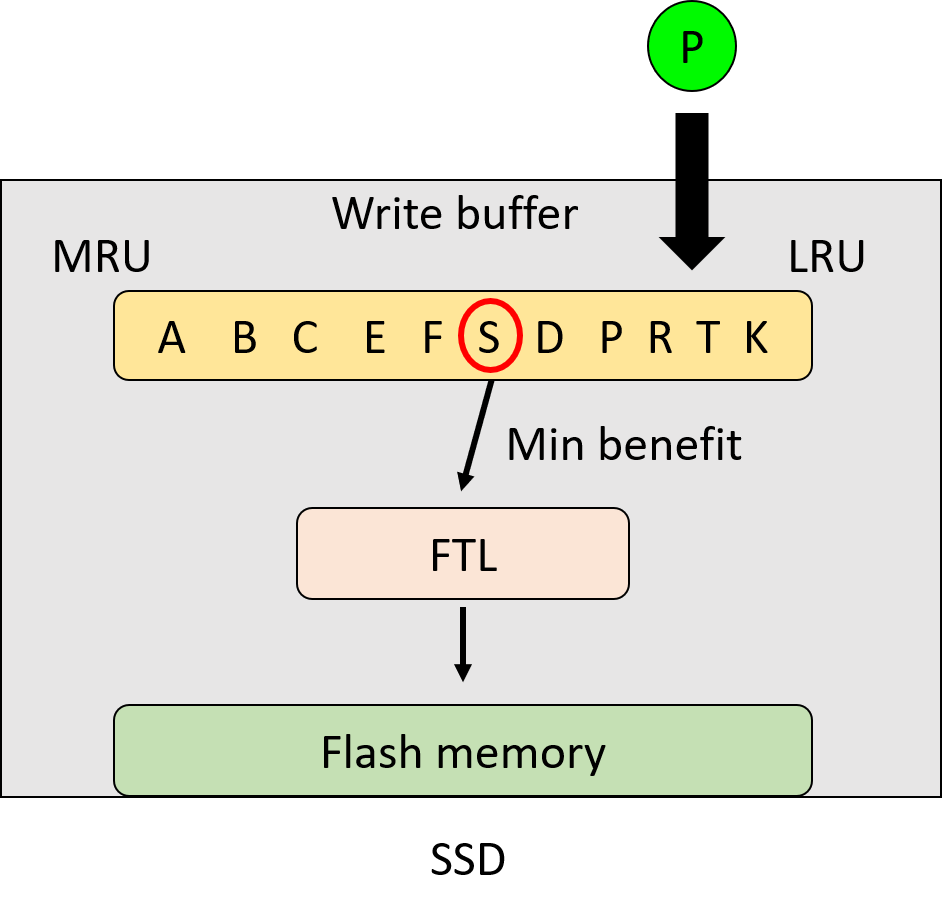


圖 3‑4 Write buffer simulator

以這樣的方式，在write buffer simulator跑完之後，每個block可獲得【Duration value】。

Transform duration value 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3-5，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)。

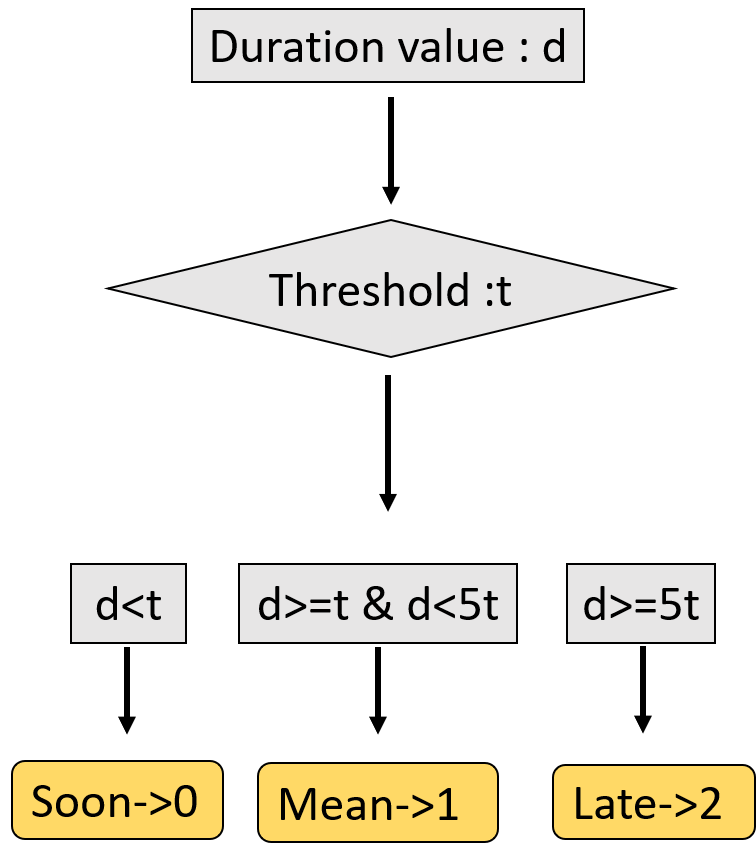


圖 3‑5 Generate Duration Label

將每個block的duration value轉換成label之後，就可以開始進行訓練了。

Offline training

在offline training，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

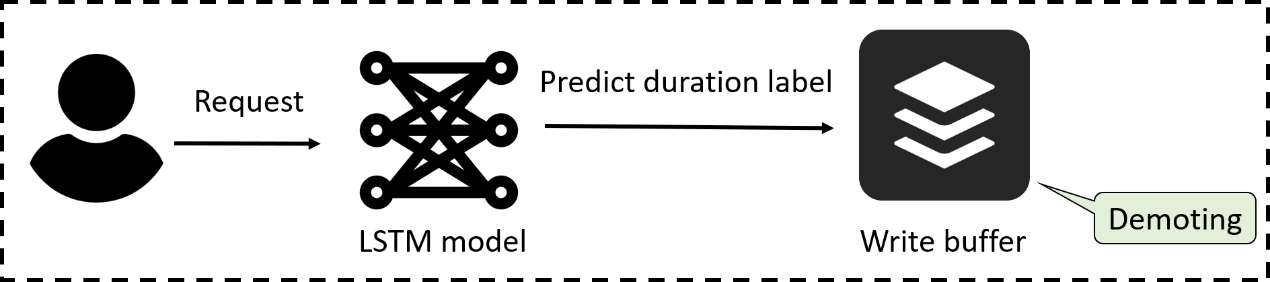


圖 3‑6 Online架構

在online部分，訓練好的model會預測每個進入write buffer的block的duration label，此外，write buffer會監測每個block，來決定是否做demoting。

1. 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，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，由於踢資料會以Soon queue為優先，因此的確可能發生某些資料占用Mean queue和Late queue的空間卻完全沒被使用到的狀況。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)。

如圖3-7，當存在某些block的pass\_req\_count>=threshold時，這時候，只要那些block所在的位置不是Soon queue，就會做Demoting，將Late queue的資料移到Mean queue、將Mean queue的資料移到Soon queue，若原本就在Soon queue，則不用理會，因為位於Soon queue LRU端的資料，原本就是會優先被選擇為victim。

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Demoting後仍然沒有被存取的話，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。

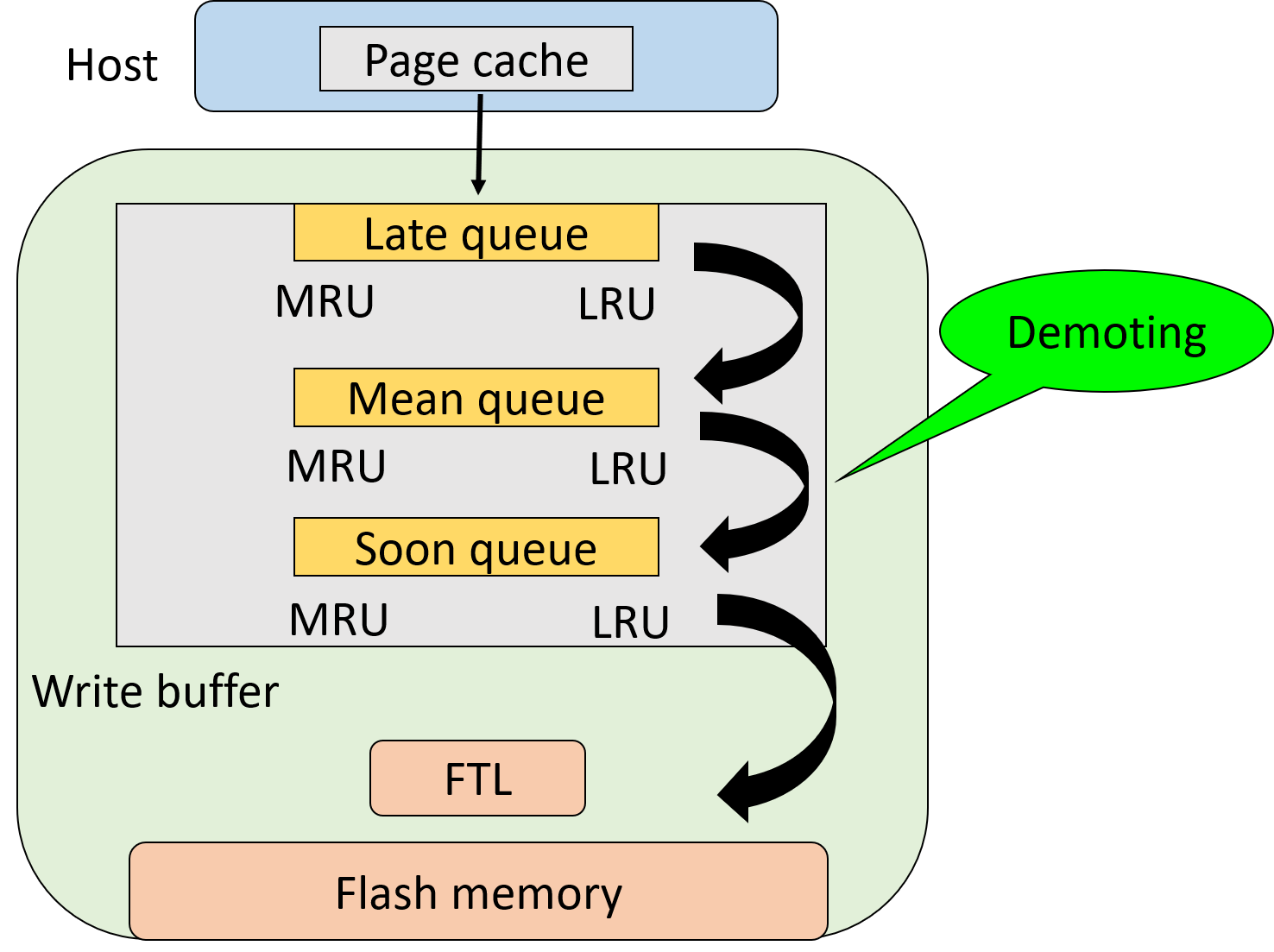


圖 3‑7 Our Write Buffer Data Placement

## 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。Hint資訊，存放的是通過週期性預測，預測即將被踢到SSD的page number，透過Hint資訊，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。

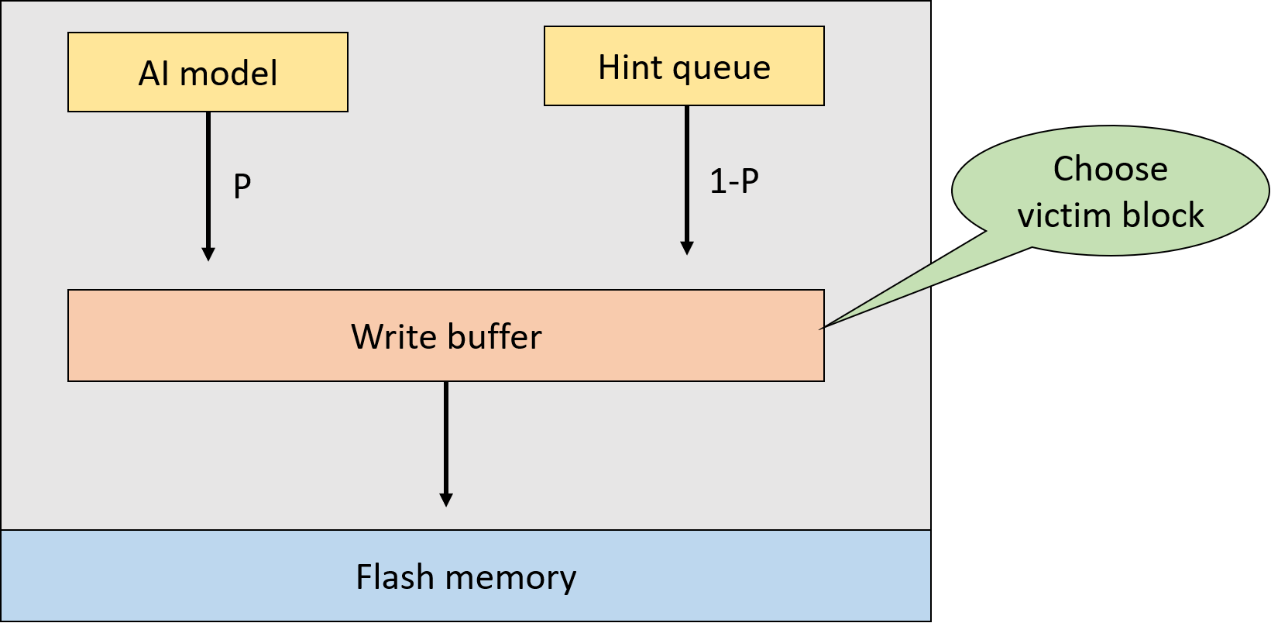


圖 3‑8 AI+Hint架構圖

### 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，我們很多次測試，為每個trace找出hit ratio最高時的p值，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。

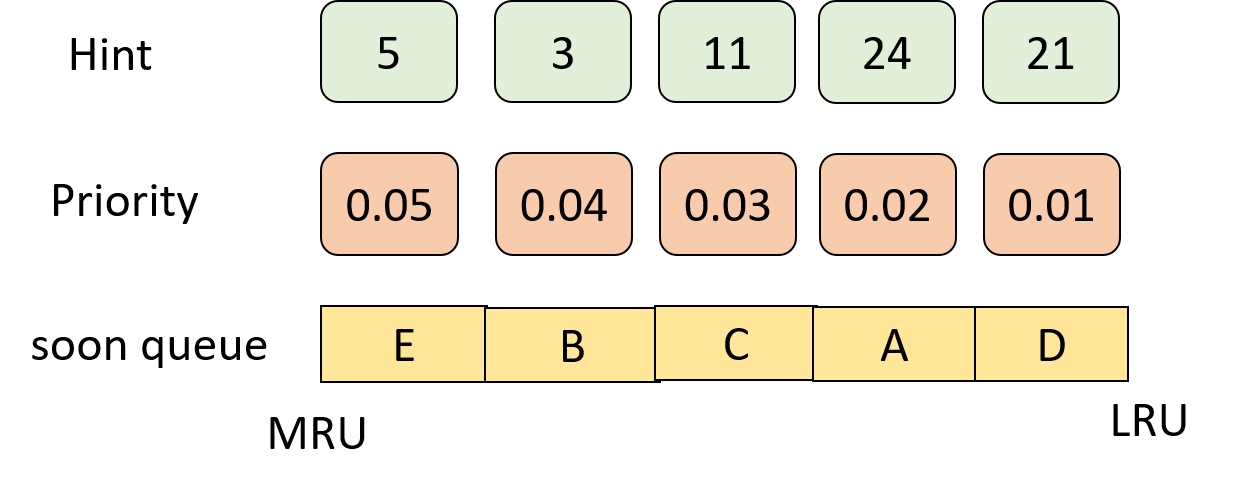


圖 3‑9 AI+Hint資訊

### 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，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。

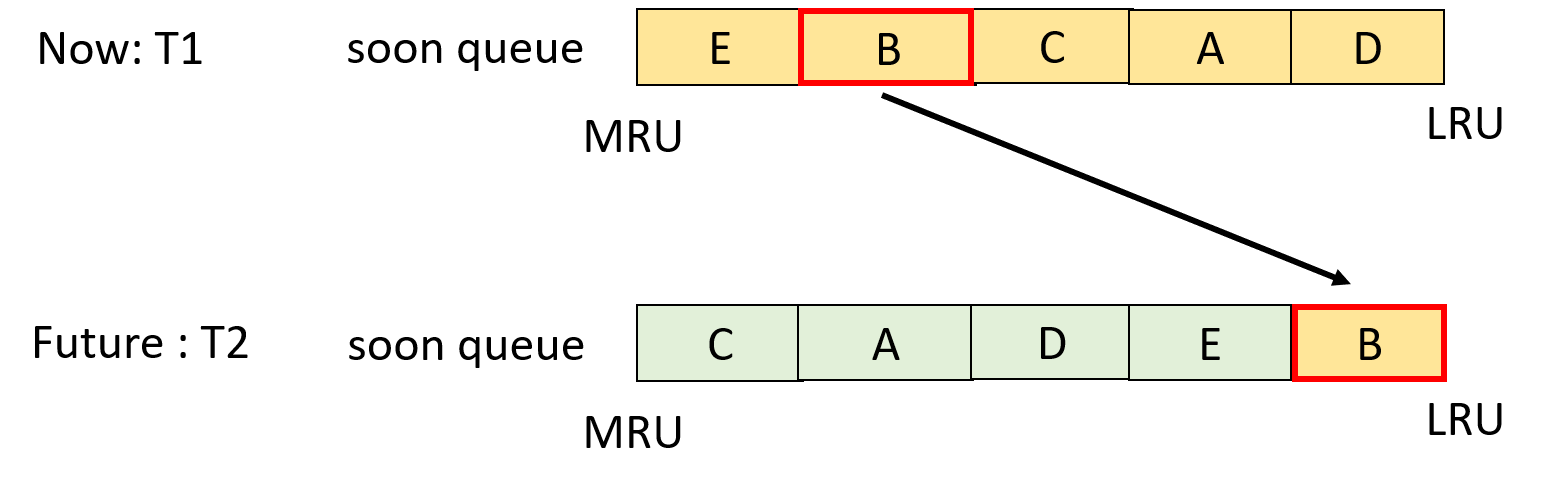


圖 3‑10 誤判的狀況

# 第四章 實驗結果

## 實驗環境

本篇論文使用disksim 4.0來模擬SSD的環境，程式執行在虛擬機上，作業系統為Ubuntu 16.04 32bit，本篇論文是基於Physical block-based write buffer進行改善，因此主要是與Physical block-based write buffer比較。

表4-1為使用的trace，這些trace是從Storage Networking Industry Association(SNIA)上取得的，其中IOzone是透過IOzone benchmark執行產生的trace。

表 4‑1　Trace特性說明

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Trace** | **Number of request** | **Read ratio** | **Write ratio** | **Sequential read ratio** | **Sequential write ratio** |
| UG-fileserver | 560144 | 0.25589 | 0.74411 | 0.456874 | 0.689882 |
| Postmark | 367941 | 0.467635 | 0.532365 | 0.708808 | 0.677919 |
| Financial | 3699197 | 0.8023453 | 0.176547 | 0.016956 | 0.013975 |
| iozone | 1405227 | 0.295923 | 0.704077 | 0.00075 | 0.009433 |
| User-2 |  |  |  |  |  |

## 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. |