

國立交通大學

資訊科學與工程研究所

碩士論文



模擬輔助調整策略固態硬碟高效能緩衝區
A Simulation-Assisted Tuning Strategy for SSD Write
Buffers

研究生：吳翊誠

指導教授：張立平 教授

中華民國 一 百 年 七 月

模擬輔助調整策略之固態硬碟高效能緩衝區
A Simulation-Assisted Tuning Strategy for SSD Write Buffers

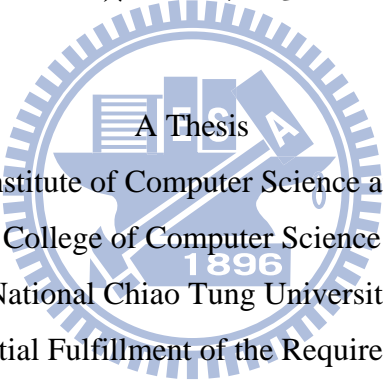
研究生：吳翊誠

Student：Yi-Cheng Wu

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

The logo of National Chiao Tung University is a circular emblem. It features a gear-like outer border. Inside the circle, there is a stylized representation of a building or a structure with the letters 'CSA' and the year '1896' at the bottom. The text 'A Thesis' is written across the center of the emblem.

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 2011

Hsinchu, Taiwan, Republic of China

中華民國九十九年七月

誌謝

沒想到兩年一下就過了，也沒想到我真的可以畢業了。研究所生活對我而言是一個完全全新的體悟跟經驗，一開始真的不太適應腦海中浮現放棄的念頭大概快一百次吧。最要感謝的還是張立平老師，他始終都沒有放棄我在我都放棄自己的時候，讓我能繼續學習不只在知識的追求上在做人做事的態度上的學習更是受益良多。

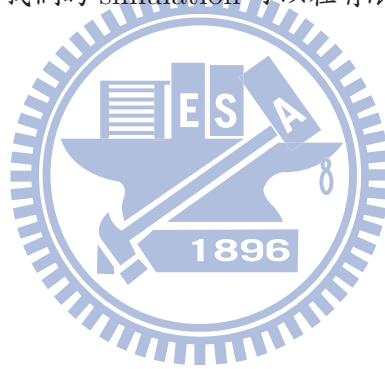
研究所生活中要感謝的人實在太多了，首先要感謝是電鍋 (郭晉廷)、Master(黃偉杰)、義勛 (黃義勛) 這三位學長我在他們身上學到很多也受了他們很多幫助；接著是小節 (李盈節)、薇涵、玟蕙在學業上面受到了你們的照顧；當然還有可愛的學弟妹們陪我度過了不少的快樂時光 (雖然你們叫我學長我都覺得怪怪的)；還有就是我國中、高中、大學的換帖兄弟們很開心在新竹還能和你們相遇，你們總是在我心情低落的給我鼓勵、陪我喝酒；最後要感謝的就是我的父母，感謝他們在背後的鼓勵也很支持我的決定，雖然我總是很衝動很任性你們還是包容著我。

最後能完成碩士的學業對我來說非常的開心和夢幻，謝謝一路上有給過我幫助的人們。時間總是推著我要不斷的前進，我也知道成長不是那麼容易的一件事，我會繼續努力加油，最後謝謝大家。



中文摘要

在 write buffer 的主要功能有兩個1. 吸收時間區域性2. 收集空間區域性來減緩在 NFTL(NAND Flash translation layer) 寫入的成本, 而在設計 write buffer replacement 策略時勢必要根據此兩種區域性來做設計。但由於各種不同 write pattern 裡面的 localities 不盡相同又必須考慮當下硬體的資源 (如 write buffer/Log buffer size), 在不同 situation 下所適用的 write buffer replacement 是不盡相同的。而此篇論文提出利用 simulation 的方式去找出當下情況所應該使用的 write buffer replacement 並能自適性的調整 replacement 策略在不同的 workload 情況。而此篇論文的 simulation 是放在 device(firmware) 端, 因此能夠拿來當作我們 simulation 的資源非常有限, 因此我們必須在拮据的資源下保留少數的資訊去模仿 write buffer/NFTL algorithm 的行為並告訴 write buffer manager 當下所該使用的 replacement 策略。因此我們被須在資源使用量和準確度上面取得妥協, 使得我們的 simulation 可以在有限的資源下, 產生出準確的模擬結果。



Abstract

Write buffer has two major tasks. One is absorbing temporal locality. The other one is collecting spatial locality. Absorbing temporal locality can reduce hot data. Collecting spatial locality can reduce write traffic of block-level and hybrid mapping. When people design write buffer replacement strategy will take two localities as designed metric. But different workloads have different write pattern behavior, and people has to consider current resource hardware can supports. It is hard to strike a balance between temporal locality and patial locality in different workload. This paper proposes a simulation-assisted Tuning strategy for self-adjusting buffer replacement strategy. We put simulation on device(firmware) part, so we have to use few resource to imitate write buffer/NFTL algorithm and inform write buffer manager "what replacement strategy we should use". So we have to make a tradeoff between few resource and simulation's accuracy.

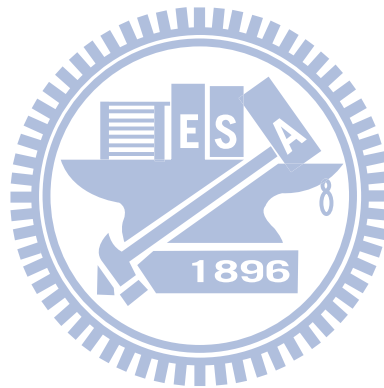


目錄

第一章 Introduction	1
第二章 Problem formulation	3
2.1 FTL	3
2.2 Write Buffer	4
2.3 l-buffer	5
第三章 Simulation-assisted tuning	7
3.1 simulation of Write Buffer	7
3.2 simulation of Log Buffer	12
3.2.1 Committed group management	13
3.2.2 Invalidation handling AND LBA group Shifting	15
3.2.3 log-buffer node data placement	16
3.2.4 early deletion old group	16
3.3 Runtime meta-simulation	16
第四章 Experiment	18
4.1 Experiment Settings	18
4.2 Experiment of vary write buffer size	20
4.2.1 Performance evaluation	20
4.2.2 memory evaluation	21
4.3 Experiment of vary log buffer size	22
4.3.1 Performance evaluation	23
4.3.2 memory evaluation	23
第五章 Conclusion	26
參考文獻	27

表目次

0.1	16M write buffer 在不同 alpha 下 buffer 中平均的 group 數	8
0.2	Environment Setup	18
0.3	l-buffer Simulation methods	19
0.4	l-buffer setting	19
0.5	other buffer methods	19
0.6	Workload setting	20



圖目次

0.1	different alpha setting in insufficient buffer size	5
0.2	different alpha setting in sufficient buffer size	6
0.3	group density	8
0.4	buffer node type	9
0.5	Record Resolution to Bit Resolution	10
0.6	RecordSize	10
0.7	Record and Bit ratio	11
0.8	log bffer group versus log buffer group	12
0.9	committed data in log buffer group	13
0.10	Remerged by First committed ratio	13
0.11	log buffer 架構/操作	14
0.12	Invalidation handling	15
0.13	group Shifting	15
0.14	early deletion	17
0.15	Runtime meta-simulation	17
0.16	Windows NB	21
0.17	Ubuntu	22
0.18	memory evaluation	22
0.19	memory-used ratio	23
0.20	vary logbuffer with early deletion	24
0.21	On-line value changes of α	24
0.22	Memory requirements for simulation vs. log buffer size	25

第一章 Introduction

NAND flash memory 的硬體架構有省電、抗震、體積小等優勢,因此很多電子行動產品(如:行動電話、多媒體播放器、數位相框)使用它來做為作為裡儲存系統的媒介。隨著科技製程的進步,NAND flash memory 價格已慢慢下降,所以 NAND flash memory 容量也慢慢變大,進而慢慢發展出取代機械式傳統硬碟的固態硬碟 (Solid-state disk)。

NAND flash memory 主要的操作可以分成三種 1.read 2.write 3.erase。其中 read/write 是以 page 為操作的單位而 erase 則是以一個 block 為操作單位,跟傳統硬碟相比,SSD 有非常好的隨機讀取效能 (random read),因為 NAND Flash Memory 沒有傳統硬碟需要移動機械手臂到待讀取位置所造成的搜尋時間 (seek delay);而在順序讀取 (sequential read) 和順序寫入 (sequential write),SSD 比起傳統硬碟有相當或者更好的效能;但是 SSD 在隨機寫入 (random write) 下的效能非常差。而在 flash memory 更新資料並不像在傳統硬碟一樣將資料 0 變 1 或 1 變 0,如果 flash memory 要複寫原來已寫過的 data 就必須需經過 抹除 (erase) 這個耗時的操作做完才可以在寫入 (erase before write),在韌體的部分則會有 FTL 介面 (Flash translation layer)[7]來將來 flash memory 模擬成傳統的硬碟,使用者並不用也不需知道 flash memory 的存在和運作。目前常見的 FTL 演算法[1, 5, 10]採用了 out of place 的作法,將新資料寫入預先保留的空間 (log buffer) 中。而在 FTL 裡面會包含當預先保留的空間用完時必須將舊資料回收的管理行為 (garbage collection),而 garbage collection 時必須先在抹除整個區塊前先搬移其中的有效資料頁 (valid Page) 到別的區塊,造成抹除的區塊的成本非常高,尤其是待抹除 區塊包含愈多的有效資料頁時,因此減少 garbage collection 時候的成本是整個 SSD 設計上要考慮的因素。

而最近改善 SSD 效能議題是加入一塊 write buffer [3, 8, 6, 4, 2]來減緩預先保留的空間(log buffer) 消耗和減少做 garbage collection 時的成本。由於 write buffer 的儲存媒介是 RAM,所以並不是 erase before write 類型的儲存媒介對於時間區域性很強的資料可以直接在 write buffer 將此吸收掉避免熱資料消耗了 log buffer 的空間。而當 write buffer 滿要做 Replacement 時,假如是 sequential data 寫到 FTL 此時可做 cost 較低 switch 操作,不用寫到 log buffer 中,但假如稍有破碎的 sequential data 寫到 FTL,此時會將破碎的資料從 FTL 讀上來補成是 sequential data 在寫到 FTL。這樣雖然會有一

些而外的讀取成本但是可以藉由作成本較低的 switch 避免進入 log buffer 作 cost 較高的 full merge, 這個動作稱為 hole plugging。

這篇論文提出一個新的 write buffer 設計方法, 它主要是 simulation-assisted 的方式來決定出目前該使用的 buffer replacement 策略, 我們知道 write buffer 主要是要吸收時間區域性和收集空間區域性, 但是兩方面都需要 buffer 的空間來吸收或收集, 兩者其實是會競爭。而此方法是用 simulation-assisted 根據目前 workload 和 硬體資源多寡去在吸收時間區域性和收集空間區域性之間去取得妥協進而產生最大的效能。而在 block plugging 方面此篇論文採用了 l-buffer 的方式藉由觀察 FTL 中的行為成本去決定要做 logging 還是做 plugging, 藉此得到最大的效能。第二章 Problem formulation 我們會介紹基本的 NAND Flash Translation Layer 架構和 Flash Storage 上的 write Buffer 管理方法設計時需要考量的原則, 並在 Related Work 說明既有在 Flash Storage 上的 Buffer/FTL 演算法及它們的問題; 第三章介紹此篇的 simulation-assisted 方法。



第二章 Problem formulation

2.1 FTL

NAND Flash Memory 的組成, 主要是由多個區塊所組成, 每一個區塊又是由多個 Pages 所組成。NAND Flash Memory 的一次寫入/讀取單位 (read/write unit) 則為一個 Page, 每一個 Page 會有一個 Spare Area, 可以用來儲存 User Area 中資料的 mapping 資訊或者 Error Correcting Code。而刪除單位 (erase unit) 則為一個區塊, 即每次執行 erase 時都是以區塊為單位。NAND Flash Memory 在更新資料時與一般的 block device 不同, 如: disk, 在更新某部份的資料後可直接寫回 其原本的實體位置, 但是 NAND Flash Memory 無法將更改完後的資料直接寫入至該資料原本所在的實體位置, 必須再額外找空間的 Page 寫入並將舊的資料標記成 失效 (invalid), 此動作即稱為 outplace-update。而原本的 data 則視為失效 (invalid), 但也因為如此 NAND Flash Memory 中常常會存有很多 invalid data, 因此 NAND Flash Memory 每隔一段時間即要清除那些 invalid data, 以空出空間提供其他 data 儲存, 此動作稱之為 Garbage Collection。

而大多 FTL 的採取混合式的管理方式, 每個 Logical block 會一對一 mapping 到一個 physical block 我們稱作 data block。而其他沒被對應到的 physical block 我們稱為 spare block 用來吸收 request 的更新, 而 spare block 我們用 page map 來管理, 而 Garbage Collection 的動作把部份在 Log Block 內的資料寫回其應該擺放的 Data Block 中, 在 Hybrid NFTL 中我們稱之為 Merge。採取混合式的管理方式在 block level mapping 和 page level mapping 中取得妥協, 在 mapping table 不會用像 page level mapping 那麼多的記憶體空間, 在 random write 的效能也不像 block level mapping 那般低落。

這邊要介紹幾種混合式管理方式的 FTL。SAST[1](set-Associative Sector Translation) 是將數個邏輯上的 block 綁成一個 group 而這個 group 裡的 logical block 用共相同的 log buffer。SAST 有兩個參數 N 和 K, N 是用來決定將多少個 logical block 綁成一個 group 主要是用來收集空間區域性, 讓 log buffer 使用率提高。另一個參數是用來決定此 group log buffer 的深度主要是用來收集時間區域性。而 N=1 且 K=1 這種情況下是 BAST[10](block-Associative Sector) 它是屬於 1 個 logical block 能吸收跟新的

log block 只有一個。而 BAST 和 SAST 最大的問題是 group 間會有 log block 互相搶奪的現象 (block thrashing)。FAST[5](full-Associative Sector Translation) 為全部的 logical block 共用全部的 log block, 因此不會有 log block 互相搶奪的現象, 但 FAST 的問題在於在 Garbage Collection 關聯到要做 merge 的 Logical block 太多, 造成此次 Garbage Collection 花費很久時間很有可能使得整個架構 time out 掉。

2.2 Write Buffer

write buffer 的主要作用是 1. 吸收時間區域性在減少在 log buffer 空間消耗收集空間區域性在 Hybrid mapping, block level mapping 的 FTL algorithm 下, 可以做 cost 較低 block plugging。大多 buffer replacement 都是針對時間/空間區域性去設計它的 replacement 策略。而不管是吸收時間區域性或者. 收集空間區域都需要 RAM 空間, 也就是如何去調整兩者的比重是一件在 write buffer 設計上重要的問題。

BPLRU[4]的 replacement policy 注重時間區域性, 依 LRU 順序選擇最久沒有被寫入資料的群為 victim, 它目的主要除了吸收熱資料外, 對於 sequential write 也可以收集滿整個群的資料後再寫到 NFTL, 而且使用 plugging 的機制來增加 Switch Merge 的機會。但缺點是因為用 LRU 的方式來管理群, 如果 hot 和 cold 資料同時在群中, 因為 hot 資料一直被 hit 的緣故, 造成 cold 的資料沒有機會 flush 出去而積累在 Buffer 中; 另一個缺點是沒有特別留住資料較少的群, 這些資料是造成 random write 的主要原因, 而且沒有說明什麼時候做 plugging, 如果對這些含較少資料的群也使用 plugging 代價反而會更高。FAB[2]的 replacement policy 注重空間區域性, 優先選擇含最多資料的群為 victim, 它希望從 Buffer flush 出的資料是大筆 sequential 的資料, 而儘量把含較少資料的群留在 Buffer 中。但該方法的缺點是忽略了時間區域性的重要性, 如果某個群所含的資料量很少, 可能會一直留在 buffer 而沒有機會被寫出, 這類資料會佔住 Buffer 的可用空間; 另外對於 sequential write 來說, 大多的群只收集到小部份的 sequential data 就會成為含資料最多的群而先被寫出, 最會導致留在 Buffer 中的都是含資料量少的群。CLC[3]將 write buffer 切成兩個部分, 一部分用來吸收時間區性 (熱區段), 另一部分用來收集時間區性 (冷區段), 再從較冷的區段選擇 victim, 缺點是如何去設定兩個部分的比例難以事前就知道且固定的設定也無法適用到不同的 Buffer size 以及不同的 workload。REF[9]是一個比較不一樣的 replacement policy 它並不是以時間區域性或者空間區域性作為它的主要依據, 他則是會根據 FTL 裡面的情況去決定, 如果 group 在 log buffer 中有長出來的話優先被當作 replace 的對象。優點是在 FAST 方面可以減少做 Garbage Collection

的關連度,SAST 的方面可以增加 log block 的使用率藉此減緩 log block 間搶奪的現象。但缺點是保留不住熱資料一旦熱資料在 log buffer 長出來它都會是 replace 的對象造成 buffer 失去吸收熱資料的功能。

2.3 l-buffer

l-buffer 的 replacement policy 主要是參考了1時間區域性和2空間區域性並定義出 group density($\theta(g) = \frac{1}{a(g)^{1-\alpha} \times s(g)^{1+\alpha}}$), $a(g)$ 代表 group g 在 buffer 中上一次的 access time 而 $s(g)$ 表 group g 當下收集到多少 dirty sector, 並挑出 group density 最小值的 group 來做 replace 而在公式中 $\alpha = 1$ 是用來決定 l-buffer 採用何種 replacement policy, $\alpha = 1$ 時代表 group density 公式只考慮空間區域性以挑選最 group weight 最爲 replace 對象, 而 $\alpha = -1$ 時代表 group density 公式只考慮時間 區域性以挑選最久沒被 reference 的 group 爲 replace 對象。圖0.1、0.2爲 workload 是 notebook trace 在 $\alpha = -1$ 和 $\alpha = 1$ 在充足和不充足的 write buffer size 下的寫出到 log buffer Victim 的 age 和 size 資訊, 在0.1write buffer 較不充足的情況完全考慮時間區域性的情況是要 group 的 age 稍稍變老(大) 就很容易被當作 victim 如左圖, 而在完全考慮空間區域性的情況下因爲 buffer 空間不足沒有很機會收集到 group size 較大 victim; 反觀在0.2buffer 空間較充足的情況下在完全考慮時間區域性時寫出的 victim age 都會是很大的如0.2左圖, 而如果完全考慮空間區域性的話因爲有了足夠的空間所以可以收集到較大 size 的 victim 如0.2右圖。因此 l-buffer 會利用 α 來去適應各種不同的工作環境並根據目前硬體資源所能提供的幫助下發揮出最佳的效能。

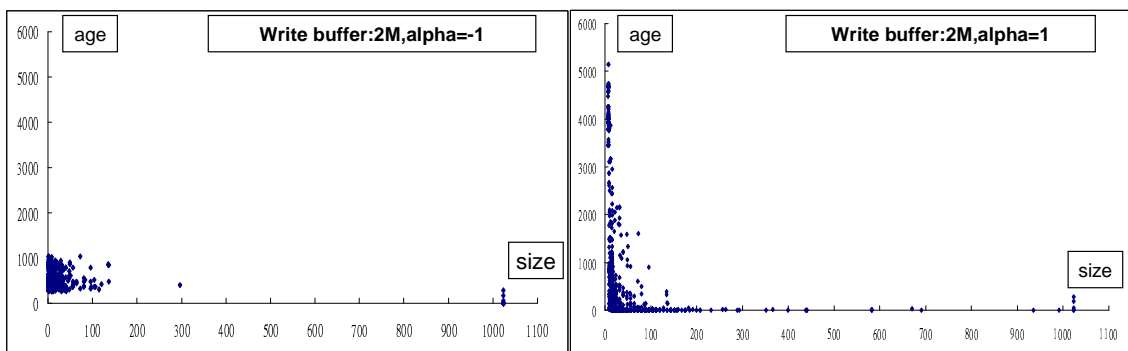


圖 0.1: different alpha setting in insufficient buffer size

當決定要 replace 的對象要進入 FTL 時,l-buffer 會根據 λ 來決定要做 plugging 或 logging, 若大於 λ 做 plugging 反之則做 logging。 λ 會因爲工作環境的樣式不同 和 log

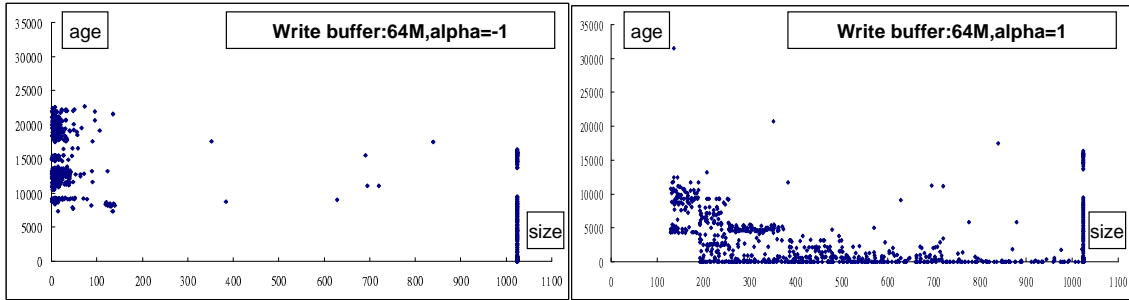


圖 0.2: different alpha setting in sufficient buffer size

buffer 的大小有所改變, 當 log buffer 較小時 λ 會比較小因為此時 log buffer 中做 Garbage Collection 的 cost 會很高倒不如提高做 plugging 的機會以減緩進入 log buffer, 而 log buffer 較大時 λ 會比較大因為 log buffer 較大的話 Garbage Collection 的 cost 會較低會讓多點資料進入 log buffer 且過多的 plugging 反而會拖累整個效能。

α 和 λ tuning 都是會隨著當下的工作環境和目前硬體資源調整, 採用調整的方式是用 session-based 的調整過程及每過一段時間後就會調整 α 和 λ 一次。而 λ 調整在 l-buffer 中它會記錄 FTL 中在每段 session 做 logging/plugging 的成本以算出 λ , 因此在 λ 的調整可以利用成本公式算出來。而 α 的調整並沒有一套成本公式去計算, 而是必須要用 simulation 的方式來幫助去抓取工作環境適合用甚麼樣的 replacement policy 去達到最佳效能。這種 simulation-assisted 的做法如 PB-LRU 是一個 cache 的演算法是用在 multi-disk/cache 並使能源使用率達到最低, 是它必須要有額外的記憶體空間去模擬它的 disk/cache, 而 PB-LRU [11] 是將此 simulation 放在 host 端, 好處是 simulation 有足夠的記憶體空間可以使用, 缺點使用者必須要在他們自己的 host 端裝此 simulation 的程式。而這篇論文並不將 α -simulation 放在 host 端而是放在 firmware 的 device 端。因此會遭遇到的問題就是在 device 端的硬體資源 (RAM) 沒辦法使用很多, 而這篇文提出的方法是利用有限的硬體資源 (RAM) 去做簡化過的 simulation 以達到硬體資源 (RAM) 減少, 但簡化過的 simulation 做出的結果必須和有完整硬體資源 (RAM) 做出的結果精準度要差不多, 是必要在硬體資源 (RAM) 不多和模擬精準度上取的平衡。

第三章 Simulation-assisted tuning

Meta-simulation(device-side simulation) for α -Selection 在 l-buffer 中, α 是 self-tuning 的參數, 用來 fitting 不同的 trace 在時間區域性和空間區域性上取得一個平衡。我們採用 Meta-simulation 方式來挑選 α 。Meta-simulation 是採用 session-based 的方式, 而它是在 back-ground 中決定下一個 session 要採用的 α 值。因為是 back-ground 的 Meta-simulation 勢必要佔掉而外的記憶體空間, 而我們目標是不需要耗損太多的記憶體空間讓整個 Meta-simulation 可以在 firmware 作掉進而放在 device 中。另外一挑戰是 Meta-simulation 精準度的問題, 必須在合理的記憶體空間消耗下還有一定的準確度。根據我們實驗的過程, 必須在記憶體空間和精準度中取的妥協。在記憶體空間的部分, 我們試這根據 write buffer 和 log buffer 的運作特性簡化整個資料結構的設計並搭配較 coarse-grained 的 resolution 去減少記憶體的消耗, 而基於這些有限的資訊去產生精準的模擬結果。所以我們將 meta-simulation 分成兩個部分: 1.write-buffer part 2.log-buffer part。

3.1 simulation of Write Buffer

在 Write Buffer 的 Meta-simulation 方面, 在精準度方面我們必須要能夠 catch 準確的 replacement order, 而左右 replacement policy 方面 l-Buffer 是根據 Group density ($\theta(g) = \frac{1}{a(g)^{1-\alpha} \times s(g)^{1+\alpha}}$), 所以對於每個 LBA group 都必須 keep 住 age(temporal info) and size(spatial info)。而在 memory 的消耗方面每個 group 要紀錄的 information 有 (1)LBA: 此 group 對應到 Logical Block 的 Address(2)temporal info: 這部分比較容易, 只需一個 4KB 的 counter 去記錄。(3)spatial info: 紀錄此 group access state, 因此我們必須要有額外的空間來記錄每一個 sector 在 group 的狀態。表 0.1 是 16 M write buffer 跑 notebook trace buffer 中 group 大概介於 1000~2000 個, 其中 group age 和 group LBA 在 buffer size 16M 的情況下跑 notebook trace 兩個所佔的 memory 總和皆在 20KB 以下。而假如 spatial info 用 bit map 來存的話就要 250KB 對於 Memory 的消耗顯然太大。所以我們針對如何 keep spatial info 提出了 dual-resolution management。

觀察 bit map 的使用狀況大多 group 中 dirty sector 的數量不多大部分都是 clean 的 sector map, 對於這種 group weight 比較小的 group, 我們不用 bit map 的方式來

表 0.1: 16M write buffer 在不同 alpha 下 buffer 中平均的 group 數

trace/ α	-1	-0.5	-0.25	0	0.25	0.5	1
NB	434.22	601.8	765.61	1118.68	1696.47	23048.94	2304.17
PC	368.82	548.86	715.44	1049.17	1836.70	2785.93	3313.29
UBUTU	228.73	346.40	487.36	751.96	1267.11	1777.75	2061.06

紀錄而是用 Record 的方式來紀錄 (計錄 Sector offset)。而對於 group weight 比較大的 group 我們還是利用 Bit Map 紀錄。而 dual-resolution management 主要的概念是要營造出和 l-Buffer 的 replacement 的 order 和資訊並盡可能減少 memory 的消耗量。觀察 Group density 我們可以發現小 group 的 Group density 對於 group weight 的變化非常敏感, 而大 group 的 Group density 對於 group weight 的變化則相對於遲鈍 (譬如 weight $1 \Rightarrow 2$ Group density 就整整變兩倍, 而 weight $1022 \Rightarrow 1023$ 整個 Group density 幾乎沒變)。圖 0.3(a)(b) 觀察將 temporal info 的因素固定, 觀察 group weight 對於 groupDensity 的影響。在圖 0.3(a) 很直覺 $\frac{1}{\text{groupDensity}}$ 會正比於 Group size。而圖 0.3(b) 可以發現當 group Size 小的時候 $\Delta \left(\frac{1}{\text{groupDensity}} \right)$ 對於 group weight 變化影響很大, 反之亦然。這說明了, 當我們比較 weight 較小 group 間的 Group Density, 我們必須用較細微的解析度 (fine-grained) 去管理才可以看出之間的差異。而 weight 較大 group 間的 Group Density, 我們可以用較粗的解析度 (coarse-grained) 去管理以節省記憶體消耗。

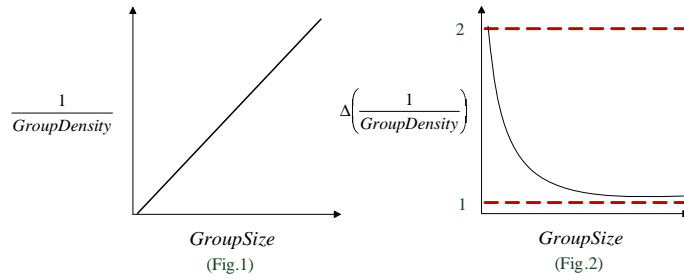


圖 0.3: group density

前面介紹了 dual-resolution management 主要的靈感來源, 接著介紹 dual-resolution 的做法。分類是根據 group 的 weight (dirty sector) 來將分為兩種不同 dirty sector 的擺放方式。小的 resolution 我們稱為 **record Resolution** (fine-grained) 是以 sector 為單位, 它是較精準 data sector map, 它的記錄方式以一個 record 紀錄一個 dirty sector 的

擺放方式。而大的 resolution 我們稱為 **Bit Resolution**(coarse-grained) 是以 sector 的倍數為單位, 它是以 bit map 來做 data 擺放的方式, 但他一個 bit 代表的是數個 sector 被存取的状态。舉例來說若 Bit Resolution = 8, 它一個 bit 就代表 8 sector 的存取的状态。就記憶體的消耗為跟一個 bit 紀錄一個 sector 的 1/8 倍。當然這個方式的精準度會較差, 但成如前面提到較大的 group 其 Group Density 對於 group weight 的變化影響不大。圖 0.4 是不同 map 的表示圖。

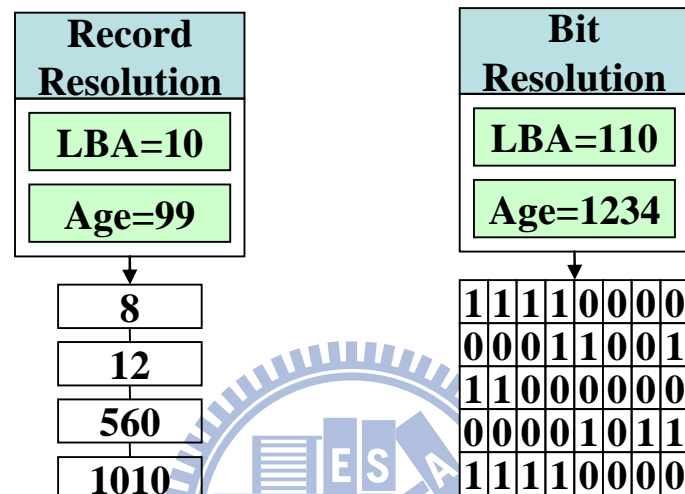


圖 0.4: buffer node type

接著隨這 Group weight 不斷的長大, 我們要如何把 record Resolution 轉換成 Bit Resolution, 這時候我們會利用 **Switch Threshold** 這個參數來做轉換的依據, 假如 record Resolution Group 中的 record 大於 Switch Threshold, 我們就將 record Resolution Group 轉換成 Bit Resolution Group 如圖 0.5 一來是減少 record 間的 search time (如果串太多 record 要 search time 就會增加), 二來是減少 memory 的消耗如公式 (a)。而 Switch Threshold 的選定是非常重要的, 如果太小的話小 record Resolution group 過早被轉換成 Bit Resolution Group 造成沒辦法分辨 weight 小 group 間的差異。太大的話會造成 memory 能 reduce 的不多。在後面實驗介紹的時候會顯示個 trace 的 Switch Threshold 是差不多的。

由於在 write buffer 裡 group 通常 weight 不會很大, 因為 weight 大的 group 很容易被挑選到當成 victim, 所以在 write buffer 裡面 record Resolution Group 通常都是占大多數。根據 Amdahl's Rule 若要在改善記憶體消耗必先從 Record Resolution group 下手。由於一個 Record 只記一個 sector 顯得有些浪費, 再加上 Request 寫入寫

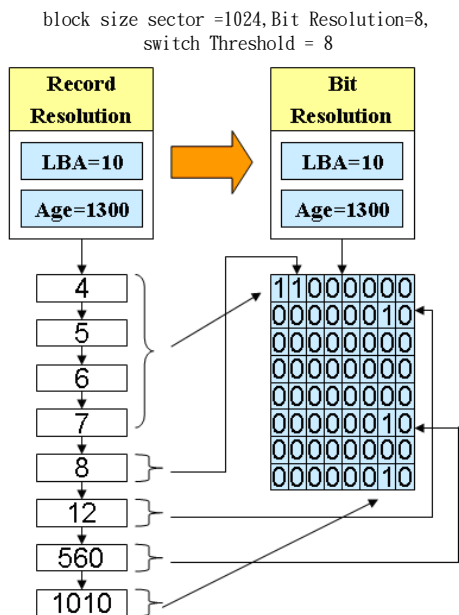


圖 0.5: Record Resolution to Bit Resolution

一個 sector 的比例極低。所以加入 RecordSize 這個機制讓每個 Record 紀錄不只一個 sector, 而是一段連續的 sectors 數。如圖 0.6 在加入 RecordSize 機制後, 某種程度來說它也是 Resolution 的放大。對精準度來講會有一定的 degrade, 所以 RecordSize 不可以太大。根據實驗在各 trace 下使用 RecordSize=2, 精準度還是在不過 Record Resolution map 的記憶體消耗可以減少一半圖 0.7。

$$memory\ reduce \Rightarrow SizePerRecord \times SwitchThreshold - \frac{SectorsPerBlock}{BitResolution} \quad (a)$$

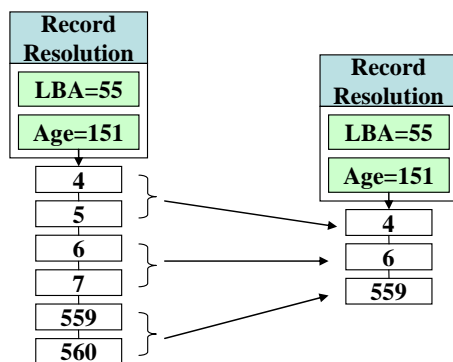
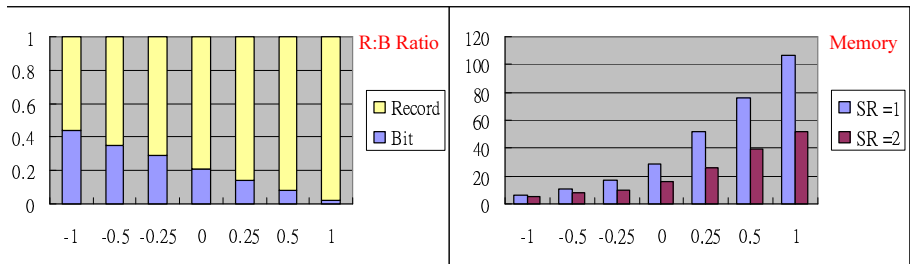
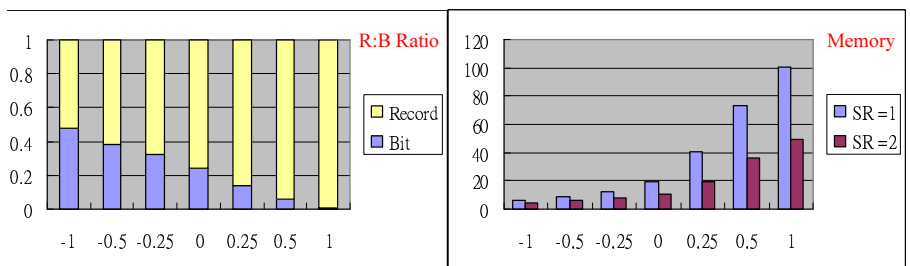


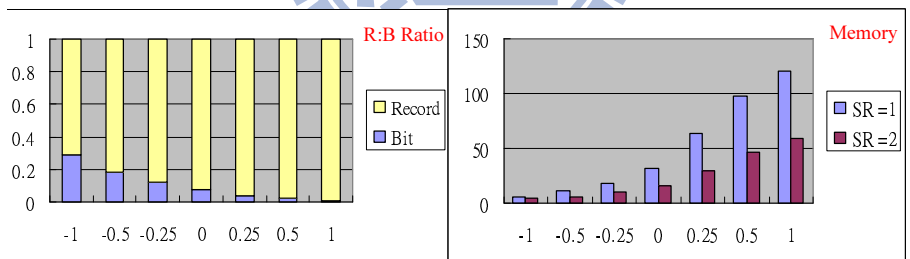
圖 0.6: RecordSize



notebook



Windows desktop



Ubuntu

Fig.7

圖 0.7: Record and Bit ratio

3.2 simulation of Log Buffer

在做 log buffer 模擬時做我們必須要知道 logging/plugging 的 cost 這樣我們才可比較出當下要用哪個 α 值最好, 所以知道 victim block 的 associativity 是非常重要的 information。另一方面是 memory 的考量, 如果將 log block 裡面的資訊全部記下的話, 這樣記憶體消耗會非常可觀。另外 log buffer size 不同於 write buffer size, log buffer size 會遠大於 write buffer size, 所以我們要確保 log buffer 的模擬不會隨這 log buffer size 變大而線性成長。

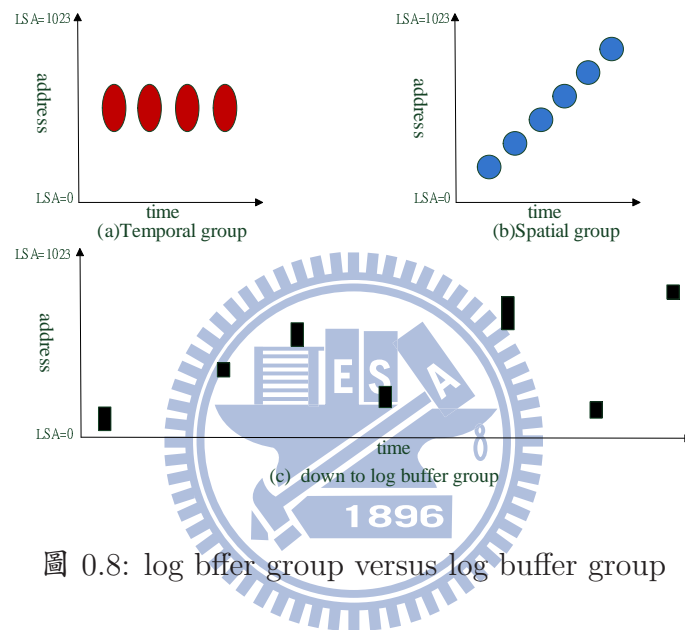


圖 0.8: log bffer group versus log buffer group

前面提到我們要比較不同 α 間的 cost, 就必須知道 log buffer 中 associativity。假如以 Spare 是 64MB, Block size=128KB, Page size = 4KB 的情況下要記下所有 log block 裡面的資訊就要 256KB 這顯然太多, 如果隨這 spare 增大又更恐怖。因此我們不去紀錄全部 log block 中資訊。從 write buffer committed 到 log buffer 的資訊去做觀察, 在前面提到大部份的 write buffer algorithm 都會盡可能去吸收 temporal locality group 如圖0.8(a) 以減少 hot data 不斷的進入 log buffer 造成 log buffer 空間消耗快速。或者會盡可能去吸收 spatial locality group 如圖0.8(b), 這對於 block-level mapping or hybrid mapping 的架構下, 可以有機會做 cost 較低的 Switch merge or Hole plugging。而 l-Buffer 則是會考慮 temporal 及 spatial, 因此真正從 write buffer 中 committed(replace) 下來的資料的特徵1. 會是較 cold 的 data, 因為 hot data group 會被 keep 在 write buffer 繼續攜手。2. 會是相對於較小 size data 進入 log buffer, 因為如果是 size 大的 data 被 flush 下來他會有很大的機會去做 hole-plugging, 並不會進

入 log block 如圖0.8(c)。

3.2.1 Committed group management

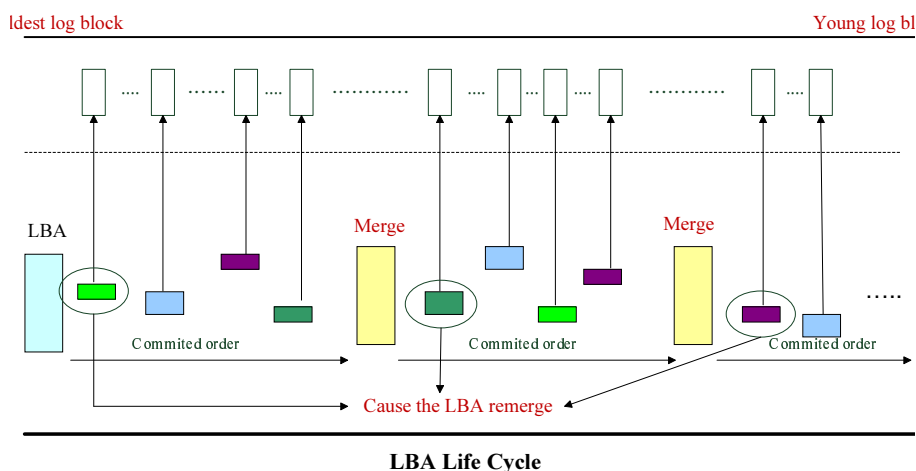


圖 0.9: committed data in log buffer group

size\alpha	-1	-0.5	-0.25	0	0.25	0.5	1
2M	0.670269	0.688902	0.707913	0.733356	0.773737	0.808323	0.842723
4M	0.693169	0.711254	0.731912	0.75943	0.820258	0.859097	0.886756
8M	0.724824	0.746546	0.770826	0.815383	0.895807	0.927305	0.926116
16M	0.746584	0.78272	0.818474	0.885567	0.93382	0.952711	0.953418
32M	0.788257	0.844487	0.861567	0.945644	0.967942	0.958774	0.968021
64M	0.833389	0.904575	0.922887	0.956607	0.978506	0.972099	0.971142
128M	0.909993	0.918945	0.947288	0.984701	0.988842	0.99202	0.952642

(a) notebook log buffer size 64M

size\alpha	-1	-0.5	-0.25	0	0.25	0.5	1
2M	0.713345	0.722475	0.727238	0.743222	0.787633	0.812982	0.800764
4M	0.719086	0.729833	0.738491	0.76516	0.809455	0.841335	0.809745
8M	0.726256	0.735827	0.756009	0.787787	0.835218	0.885267	0.859397
16M	0.734	0.758722	0.781246	0.81588	0.874367	0.911981	0.866143
32M	0.744135	0.776794	0.806683	0.848302	0.91894	0.926968	0.885346
64M	0.753642	0.806746	0.847642	0.901822	0.953025	0.956949	0.901385
128M	0.798777	0.864412	0.884287	0.933624	0.969215	0.975528	0.912162

(b) windows desktop log buffer size 128M

size\alpha	-1	-0.5	-0.25	0	0.25	0.5	1
2M	0.853347	0.856657	0.861805	0.871272	0.886386	0.889993	0.883487
4M	0.857427	0.868245	0.876846	0.886556	0.905228	0.908461	0.901642
8M	0.872552	0.881792	0.890742	0.908524	0.928542	0.945646	0.930093
16M	0.883477	0.895675	0.906894	0.931074	0.957834	0.973381	0.94211
32M	0.893694	0.908823	0.93008	0.959443	0.983758	0.991027	0.951054
64M	0.908222	0.935857	0.959634	0.983335	0.99141	0.992826	0.965086
128M	0.922253	0.960939	0.982089	0.989643	0.995008	0.993407	0.973892

(c) Ubuntu log buffer size 128M

圖 0.10: Remerged by First committed ratio

因此當每個 LBA 從 write buffer 被 committed 到 log buffer 的 Data 很少有機會全部被 invalidate 掉, 在此 LBA remerge 之前, 如圖0.9。而圖0.10是顯示在不同 trace、不同 write buffer size、不同 α 值的情況下 LBA 需要做 merge 時, 引起此次 merge 的 log block 是否為這次 LBA 第一次 committed 的 log block 比例。由圖0.10可以看到大部分引發 merge 的 log block 都為此 LBA 第一次 committed 的 log block, 因此我

們決定只記錄每個 LBA 第一次 committed 的 data, 而不是去記錄整個 log block 裡面的資訊。

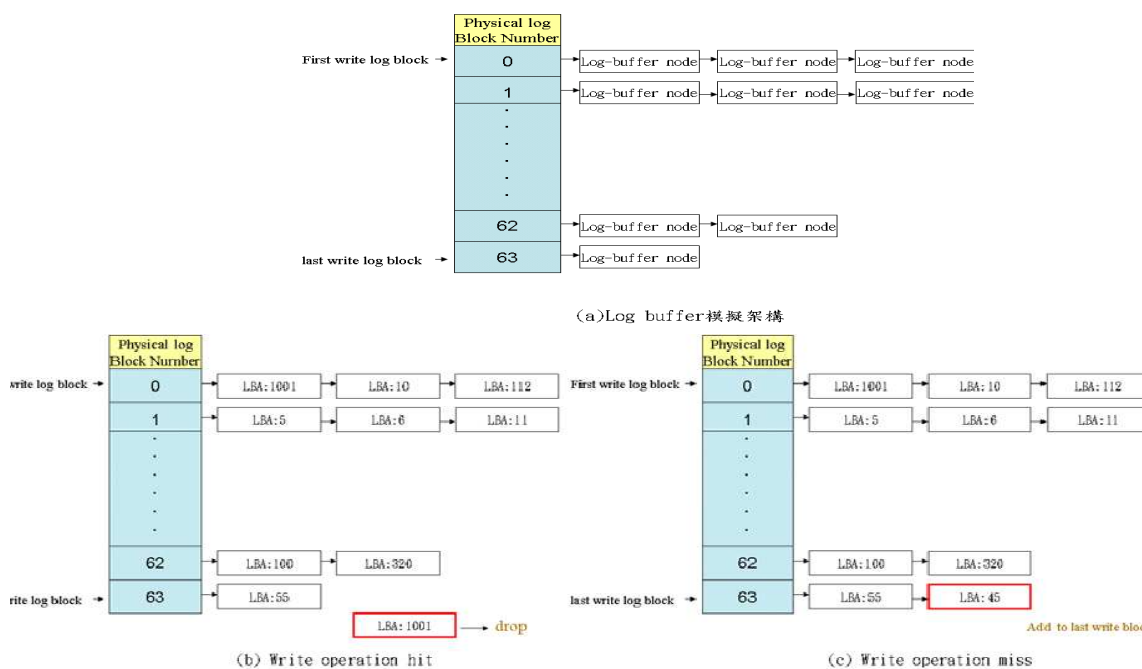


圖 0.11: log buffer 架構/操作

Committed group management 主要的概念, 就是只記錄每個 LBA 第一次 committed 到 log buffer 的資料, 因為大部分的 merge 都是因為第一次的 committed data 所造成的。而至於第一次 committed 之後的 committed 我們就不去記錄它, 因為當 merge 時會將後面所有的資訊都帶走。因此 keep 第一次 committed data 是較有效率維持準確度和減少記憶體消耗有效的方法。在圖0.11(a) 是整個 log buffer 架構, 因為我們只記錄第一次 committed 的 data, 所以每個 LBA 在架構裡只會有一個 node。另外有一個 physical log Block Number 的 index, 而 index 上的 list 紀錄所有 LBA 第一次 committed 時所以寫到的 log block。因此當某一個 log block 被 GC, 我們只需要算它串的 list 有幾個 node 就可以知道 associativity。圖0.11(b) 要表示的是當這次 committed 的 data 不是第一次 committed, 也就是說之前有 committed data 寫到 log buffer, 所以這次的 committed data 我們完全不去記錄他, 以節省記憶體空間, 況且他對 associativity 不會有影響, 會被前面 committed data merge 時帶走。如圖0.11(b) 中 LBA:1001 有一筆 committed data 到來, 可是在 log block 0 上已有 LBA:1001 的 committed 資訊所以我們就不記錄這筆 committed data。而當 committed data 到來對應的 LBA 還沒有在 log buffer 架構長出 node 來, 代表這是第一次的 committed 資料

所以將此筆 committed 放入目前正在寫的 log block index 中。如圖0.11(c)。

3.2.2 Invalidation handling AND LBA group Shifting

在這部分我們要討論的是 invalidation 的機制，雖然第一次 committed data 要完全被 invalidation 掉的機會不高，但我們還是必須要 maintain 如果完全被 invalid 掉時的情況。Invalidation handling 方面，我們一開始都先 keep 第一次 committed data，而後面的 committed data 我們是不處理（儲存）他，但我們會查看他是否能將第一次 committed data invalidate 掉，如果能的話我們就將 group 中儲存的 data 釋放，直到 group 中所儲存的 data 歸零。這樣的做法，我們一方面可以節省記憶體的空間，二方面我可以知道第一次 committed data 完全被 invalidate 掉。如圖0.12

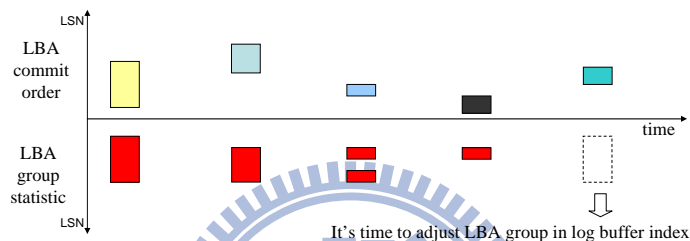


圖 0.12: Invalidation handling

當第一次 committed data 完全被 invalid 時則必須要做 group Shifting 圖0.12，而我們的做法將 group shift 到最後一次 invalidated write log block 中。這對 multi-write 的 total invalidation 會有些微的誤差。因為我們為了記憶體考量直接將 group 移到最後一次 write log block(照理說應該要移到第二次 committed log block, 但要記錄的 info 太多)。不過根據實驗顯示誤差不大，因為占大部份的都是 one write committed data 就可以將第一次 write committed data invalidated 掉。

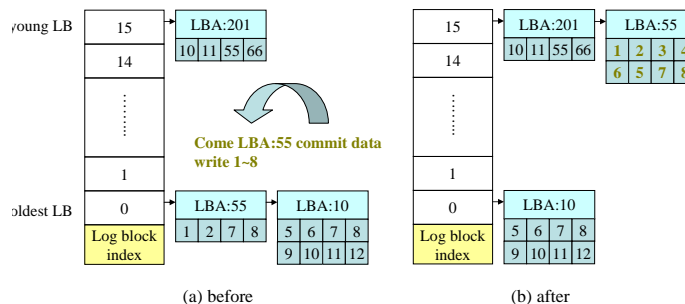


圖 0.13: group Shifting

3.2.3 log-buffer node data placement

我們前面討論巨觀的 memory reduce \Rightarrow committed group management, 談到了只記錄第一次 committed data 來減少不必要的記憶體浪費。而在微觀上面, 我們可以對於第一次 committed data 擺放來達到減少記憶體浪費。在此我們和 write buffer 一樣提出兩種資料的擺放方式: 1. Record \Rightarrow 每個 Record 紀錄一個 LPN。 2. bit map。這邊的兩種資料擺放方式不會影響到整個準確度, 因為我們沒有做 Resolution 的放大, 況且在 log buffer read/write 的單位已經到 page 比起 write buffer 中的 sector Resolution 上面已經大了八倍, 某種程度也是大的 Resolution Map。在這邊我們會有一個轉換的 logbuffer-RecordNum, 當 record 的數目大於此的話我們將 record place 轉換成 bit map place。而如何去設定 logbufferRecordNum 會影響到我們能省到多少記憶體的多寡。根據下面 (a)(b) 的公式如果推導只要讓 Dirty Pages $< \frac{PagePerBlock}{RecordSize}$ 我們用 record 來存會用較少的記憶體空間, 所以將 logbufferRecordNum 設成 $j \cdot PagePerBlock / RecordSize$, 我們將得到較少的記憶體消耗。


$$recordcost = DirtyPages \times RecordSize(bit) \quad (a)$$

$$bitmapcost = PagePerBlock(bit) \quad (b)$$

3.2.4 early deletion old group

在這個部分我們要討論的 group in log buffer 的問題, 因為 log buffer size 往往比 write buffer 大很多, 所以 group 數對於 log buffer memory 消耗會是個問題。我們不希望隨這 spare 的增大造成我們做模擬的 memory 消耗線性增加。因為提出了 early deletion old groups 圖0.14, 它的做法其實就是去限制 log buffer 的 group, 作法就類似 LRU, 我們將那些待在 log buffer 中最 old group 空間釋放出來給新的 LBA group 使用。這樣做的概念是我們給了那些 old group 中很長的時間留在 log buffer 架構裡面, 但他還是沒有辦法將 group 裡面的資料完全 invalidated 掉, 此 group 將來要完全被 invalidate 的機率太低, 所以我們就認定此 group 不會在被移動, 所以將此 group 對應到的 log block index associativity 先加一, 因為我們認定此 group 就會在他目前對應到的 log block 作 GC 時做 merge。

3.3 Runtime meta-simulation

當我們有了 write buffer 和 log buffer meta-simulation 架構之後, 就可以 back-

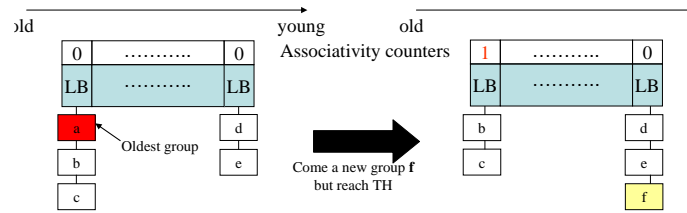


圖 0.14: early deletion

ground 知道當下較好的 α 值是多少，讓我們的真正的 device 可以根據較好的 alpha 值去做運作。而我們採用的調整方法是用 multi-Instance + Session-based 如圖 0.15。multi-Instance 的話我們會選取不同的 α 值 (-1~1) 以 catch 最佳的空間/時間區域性的比例。Session-based: 每經過一 Session 就調整一次，將前一次 Meta-simulation 較好的 alpha 值給這次 session 的 real device 使用，而 Session 的單位我們採用 write request 的次數，這對於個 Instance 較於公平，如果採用 flush group 若偏 BPLRU 的 Instance group 會比較少，反之偏向 FAB 的 Instance group 會比較多。

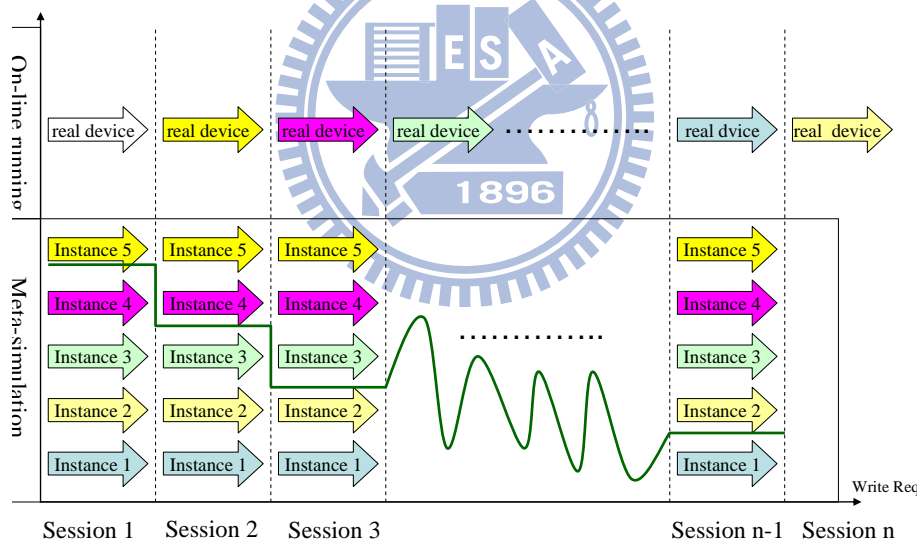


圖 0.15: Runtime meta-simulation

第四章 Experiment

4.1 Experiment Settings

在實驗中我們所使用的 Log buffer size/Write buffer size 以及 NAND flash Memory 規格如表0.2, 其中有 NAND flash Memory 的 Block size、Page size 和抹除及讀寫時間如果沒有特別講就是沿用表0.2。而至於我們在 NFTL 中所使用的演算法是用 FAST [5]來當作底層NFTL 演算法。我們首先是由 l-buffer 這個方法本身對於 α 調整的比較, 而比較的對象有三種 1.offline 2.host-simulation 3. device-simulation 如表0.3,offline 顧名思義他是在 offline 的情況下找出最適合的 α 值但缺點是沒辦法實作不過我們可以來把它當作 optimal 比較的對象。device/host simulation 都是利用 meta-simulation 的方式來找出適合 α 值, 而差異就是所需要用的記憶體空間造成一個可以將 simulator 放入 device 一個則必須在 host 中安裝。而 device-simulation 的設定方面我們所設定的 meta-Instance 是3個, 在 write buffer 的參數設定 *recordSize* 為2個 sector 也就是一個 record 可以紀錄兩個 LSN而在 *bit-resolution* 我們採用為 sector 的八倍大也就是將 map 解析度放大八倍, 至於轉換的 Threshold 為24個 sector。而 log buffer 方面則有兩個參數要設定 1. 在 log buffer 中最多 log buffer group 的個數=3000, 而資料擺放轉換的 Threshold 為4個 page 設定如表0.4。

表 0.2: Environment Setup

write buffer size	2M 4M 8M 16M 32M 64M 128M
log buffer size	64M 128M 256M 512M
Sector size	512 B
Page size	4 KB
block size	512 KB
Time of Page Read	60 us
Time of Page Write	800 us
Time of Page Erase	1500 us

接著會拿剛剛提到的三種 l-buffer α 調整方法跟現有提出的其他 buffer algorithm 作比較,FAB 的 Buffer replacement policy 為每次挑選含最多資料的 Group 當作 victim, 沒有 plugging的機制; 而 BPLRU 的 Buffer replacement policy 則是依 LRU order 來管

表 0.3: l-buffer Simulation methods

	Description
offline	無法實作, 拿來當作 optimal 比較
host-simulation	使用的 memory 多, host 端必須安裝 simulator
device-simulation	用的 memory 少, 須克服精準度的問題

表 0.4: l-buffer setting

instance	3
Write Buffer	parameter setting
recordSize	2(sectors)
bit-Resolution	8(sector 的八倍)
resolution change TH	24(sectors)
Log Buffer	parameter setting
LogbuffernodeTH	3000
ogbufferRecordNum	4

理 Group, plugging 發生的時機當 Group 的資料滿一半時 BPLRU 就會使用 plugging 機制;CLC 的 Buffer replacement policy將 buffer 切成兩個部分 Upper 為 LRU order 用來吸收時間區域性,lower 則用來收集空間區域性而兩個空間比例為 1:9;REF 的 Buffer replacement policy 則是會根據 FTL 的資訊將最近有在 log buffer長出來的 group 優先寫出 buffer 而 plugging 的時機當 Group 的資料滿一半時 BPLRU 就會使用 plugging 機制。而各方法的設定如表 0.5。

而在使用的 workload 的環境我們採用兩種大家比較常用的工作環境 1.windows 2.Ubuntu, 而在 windows 又分為 notebook 和 Desktop 兩種。在 windows 環境使用情況是比較偏有 locality 的操作, 而相較於 Ubuntu 的方面整體的使用會比較 random。而關於 Workload 的設定如表 0.6。在下面的章節會根據不同的 workload、不同的 log buffer size、不

表 0.5: other buffer methods

	Buffer Replacement Policy	Plugging	Default Settings	Reference
BPLRU	Group of Least Recently Used	>0.5		[4]
FAB	Group contains the most sectors	No		[2]
CLC	Upper layer LRU+ lower layer FAB	No	Upper:lower=1:9	[3]
REF	Select the group that Recently evict	>0.5	VW=0.75,VB=3	[9]

表 0.6: Workload setting

	Logical partition	over-provision
Windows NB	20GB	64MB 256MB
Windows Desktop	40GB	128MB 256MB
Ubuntu	40GB	128MB 256MB

同的 write buffer size 來做比較。

4.2 Experiment of vary write buffer size

這部分要探討的是在不同的 write buffer 硬體資源情況下搭配不同 trace 的特性, 而我們的 device-simulation 如何去找出最佳的 α 值去適應不同的 write buffer 硬體資源和不同 trace 的特性。而我們將分為兩個方面去討論 1. Performance evaluation 2. memory evaluation.

4.2.1 Performance evaluation

下面圖 0.16 是 workload 為 Windows notebook 而 over-provision 給 64MB 下的比較數據, 而圖 0.16 左邊那張是 l-buffer 不同 α 值調整策略的比較, offline/host/firmware(device) 這三種是會去調整 α 值而 $\alpha = -1, 0, 1$ 為固定 α 值的策略。首先我們可以看到在 write buffer 比較小時 (2M~8M) 可以看得出來在固定 α 值的策略下在 $\alpha = -1$ 的情況下效果會最好, 因為在 write buffer 小的情況下 write buffer 的空間首要拿來吸收 hot data 也就是時間區域性會是較好的方式, 反觀在 $\alpha = 1$ 下此時因為 write buffer 太小在 buffer 中的 group 很難有機會收集到足夠大的 size 更不用是說做較便宜的 merge:switch/parital merge 而且也沒辦法吸收 hot data 導致在 buffer 小的情況下 $\alpha = -1$ 效能很差; 反觀在 write buffer size 大的時候 (32M~128M) 這時候 $\alpha = -1$ 和 $\alpha = 1$ 的效能會逐漸接近, 因為此時的有了充足的 buffer 空間 buffer 中更有能力來處理/收集空間的區域性。而有能調整 α 值的 offline/host/firmware(device) 在 buffer size 的情況下採用盡量吸收時間區域性所以會和 $\alpha = -1$ 的效能差不多, 但 buffer size 足夠大時又能適度的收集空間區域性, 所以在 buffer size 足夠大能夠調整 α 值的方法會比較好。而在能夠調整 α 值的方法下, 不論是 host-simulation/device simulation 都可以藉由調整 α 去符合當下情況下最該收集的資料使得動態調整 α 值會有較好的效能。另在圖 0.16 右邊那張是在比較 l-buffer 和其他現有的 buffer 演算法做比較, 我們可以看到在每個 write buffer size l-buffer 都比其他方法來的好。圖 0.17 是針對 Ubuntu 這個比較散亂的 workload 作

的效能比較, 圖0.17左邊是 l-buffer 不同 α 值調整策略的比較, 我們先來看固定 α 的效能在 buffer size 小的時候很明顯的在這個 workload 下吸收 hot data 是非常重要的 $\alpha=0,1$ 和 $\alpha=-1$ 有一都有一段不小的差距, 這是因為這個 workload 較為散亂造成在小 buffer size 情況下根本不用考慮到空間區域性的收集, 在 $\alpha=0$ 的情況下多考慮一點空間區域性的收集就使得它的效能和 $\alpha=-1$ 有一段差距。而在能夠調整 α 值的方法下, host-simulation/device simulation/offline 的調整下都有相近的效能且在 buffer size 小的時候接近 $\alpha=-1$ 用以吸收 hot data, buffer 大時適性的考慮多一點空間區域性可以收集空間區域性搭配 plugging 因此效能 在 buffer size 漸大時效能又會比 $\alpha=-1$ 好。而在圖0.17右邊比較了 l-buffer 和他現有的 buffer 演算法, 我們可以看到幾乎每個 write buffer size l-buffer 都比其他方法來的好。而在 buffer size 小時 l-buffer 會和只考慮時間區域性的方法 BPLRU 差不多沒有贏很多的原因是在由於 Ubuntu workload 本身比較零碎加上 write buffer size 小能夠收集到 size 較大的 group 少, 導致能做 plugging 機會不多, 導致 l-buffer 沒辦法從 plugging 來賺取很多的 performance。

綜合上面兩個 workload 的實驗我們可以看到 meta-simulation 有這不錯的調整效果, 這告訴我們動態調整 α 是必要的且不可缺乏的。另外實驗中 device simulation 和 host simulation 的效能沒有多大的落差這說明了我們只存少部分重要的資訊就可以和完整的 meta-simulation 有差不多的效能而在記憶體的使用上又有更多的節省。在下一小節會比較此兩種 meta-simulation 在記憶體使用上的差異。

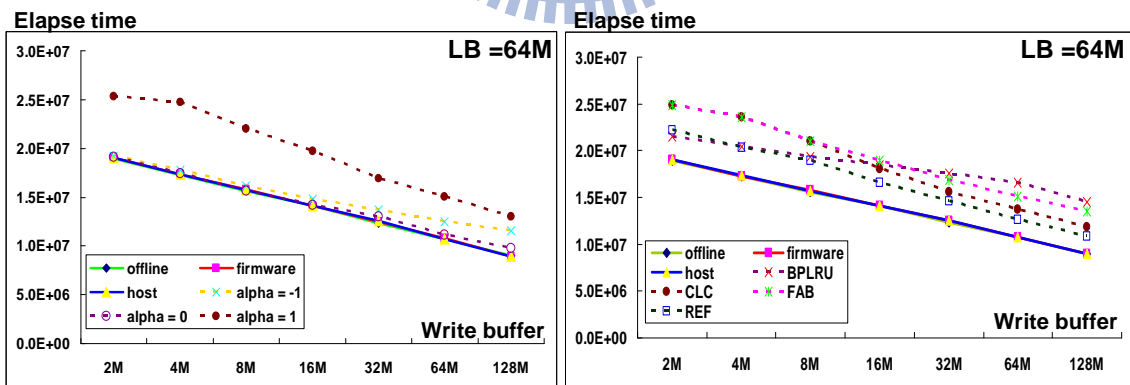


圖 0.16: Windows NB

4.2.2 memory evaluation

在這部分我們將根據上面的實驗來分析 host-simulation 和 device(firmware)-simulation 在記憶體的使用程度上的差別, 圖0.18左邊是 Windows NB 右邊是 Ubuntu workload 從

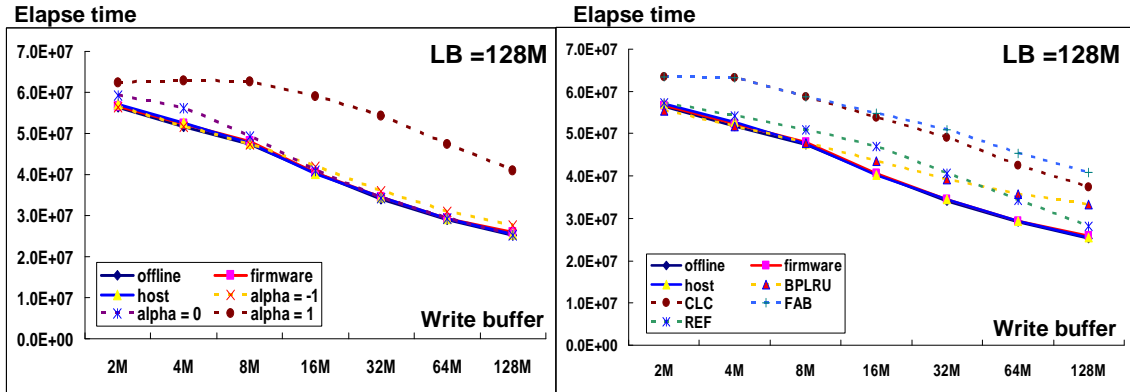


圖 0.17: Ubuntu

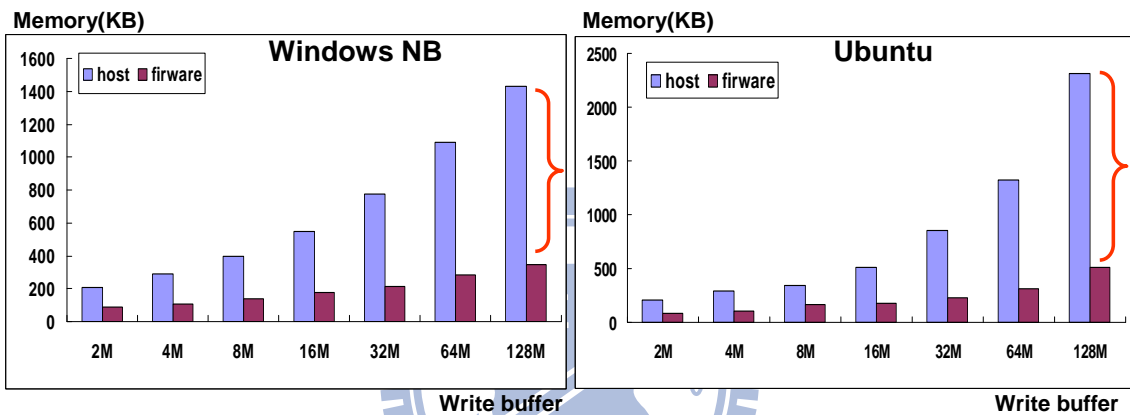


圖 0.18: memory evaluation

圖中我們可以看得出來 device(firmware)-simulation 記憶體的使用量明顯比 host-simulation 來的少, host-simulation 使用量大概為 device(firmware)-simulation 3~4 倍隨著 write buffer 的上升會更多。而主要的原因是我們 dual-resolution 來減少龐大的 sector bit map 以及利用 request 的連續性利用 RecordSize 來紀錄 dirty sector。而圖 0.19 顯示不同 write buffer 下所需要的記憶體空間佔實際 SSD control ram 的比例從圖中看起來並不多, 而另外我們可以發現使用記憶體的多寡和 write buffer size 是成正比的, 因為 write buffer size 一大裡面所要 maintain 的 group 數就多, 不過由於是成正比我們無需特別去對 write buffer group 去做限制。

4.3 Experiment of vary log buffer size

這部分要探討的是在不同的 Log buffer 硬體資源情況下搭配不同 trace 的特性, 而我們的 device-simulation 如何去找出最佳的 α 值去適應不同的 Log buffer 硬體資源和

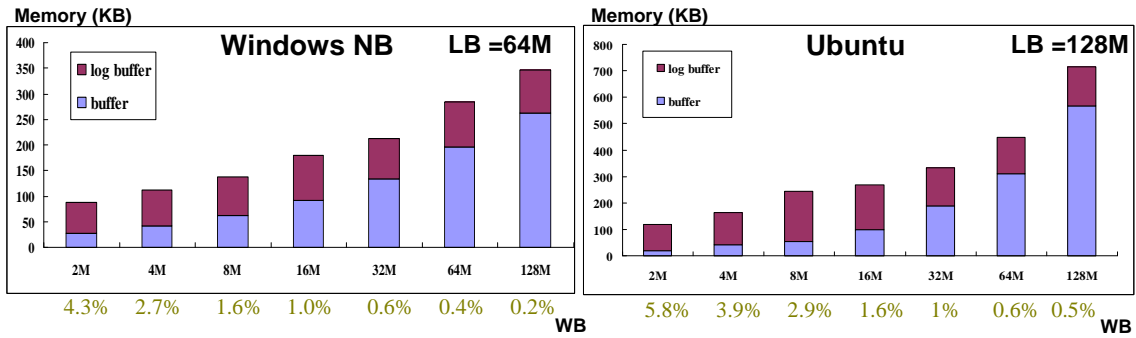


圖 0.19: memory-used ratio

不同 trace 的特性。而我們一樣將分為兩個方面去討論 1. Performance evaluation 2. memory evaluation。但是不一樣的是隨這 log buffer size 增大在 log buffer 中的 group 也會隨之增多,可是此時我們 SSD control ram 並沒有變多,因此當 log buffer 增大時我們就會利用前面提到的early deletion來減少記憶體消耗。

4.3.1 Performance evaluation

這部分我們首先要探討在不同 log buffer size 下搭配不同的LogbuffernodeTH來限制在 log buffer 中的 group 數,圖0.20左邊是 windows Desktop 右邊是 Ubuntu 這兩個 workload 的 logical partition 都是 40GB,所以能摸到的 LBA 數會較多因此我們拿來做實驗。我們首先來看當 LogbuffernodeTH₀2000 下 early deletion 所帶來的效能降級非常的不明顯可以說和沒有限制 log buffer group 的情況下,這說明我們可以藉由 early deletion 的方法來降低 log buffer size 增大時我們使用 SSD control ram 並不會隨他線性增加。而看圖中比較極端的例子當我們將 LogbuffernodeTH 設的很低時候 (200) 我們可以看到效能的方面有呈現降級的現象,因為我們 keep 太少 log buffer group 的資訊造成他的精準度變差。圖0.21是在不同 LogbuffernodeTH 情況下 α 的變化,我們可以看到在 200 的時候 α 的變化跟沒有限制時有這顯著的變化。總之用適當 LogbuffernodeTH 作 early deletion 不僅能減少記憶體消耗在效能方面也不會有太多的降級。

4.3.2 memory evaluation

這部分我們首先要探討在不同 log buffer size 下搭配不同的LogbuffernodeTH所產生的記憶體消耗量。圖0.22我們可以看到如果不對 log buffer group 作限制的話記憶體的消耗是一件恐怖的事,尤其看 Ubuntu workload 幾乎是以倍數再增加,而使用 LogbuffernodeTH 我們可以看到記憶體消耗量方面 windows Desktop 都可以減少 2 成~3 成,在 128M 和 256M 減少的量不明顯是因為在這兩的 log buffer size 下並不會太多因為

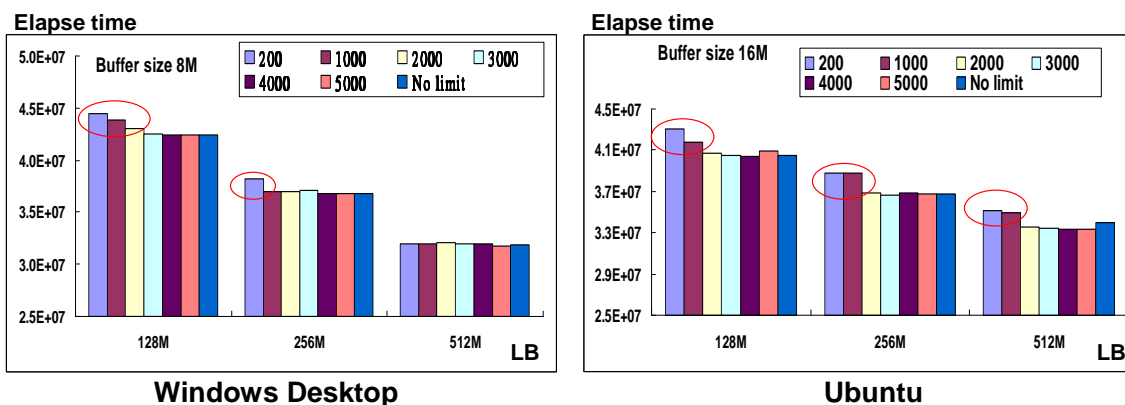


圖 0.20: vary logbuffer with early deletion

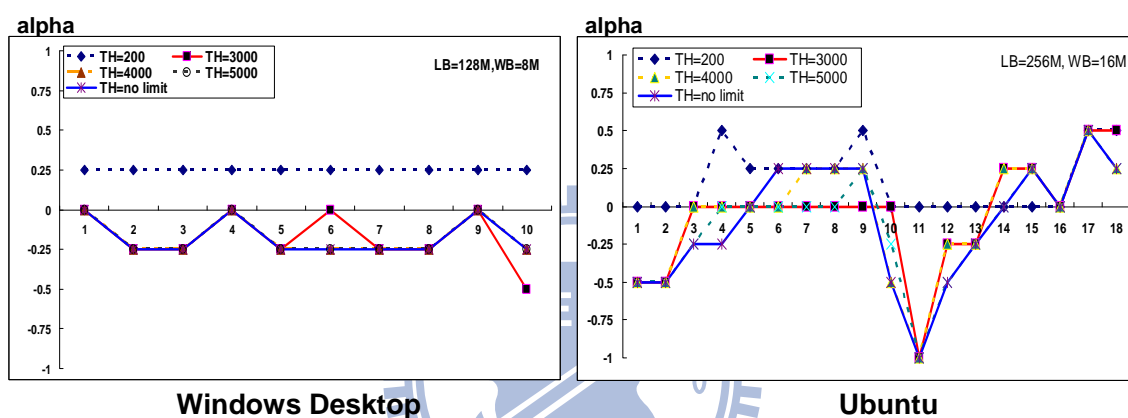


圖 0.21: On-line value changes of α

Desktop 本身是屬於較 locality 的存取行為所以它會摸到的 LBA 比較少；而 Ubuntu 就不一樣了大概可以減少 4 成~5 成，原因是因為 Ubuntu 本身就是屬於較零碎的存取行為所以它能摸到的 LBA 相對多很多造成就算 log buffer size 只有 128M 時我們就有要做 early deletion 的必要。early deletion 能減少的記憶體消耗根據 workload 的存取範圍和存取行為或多或少的節省，對整體而言 early deletion 是必須要做的尤其當 log buffer size 大的時候。

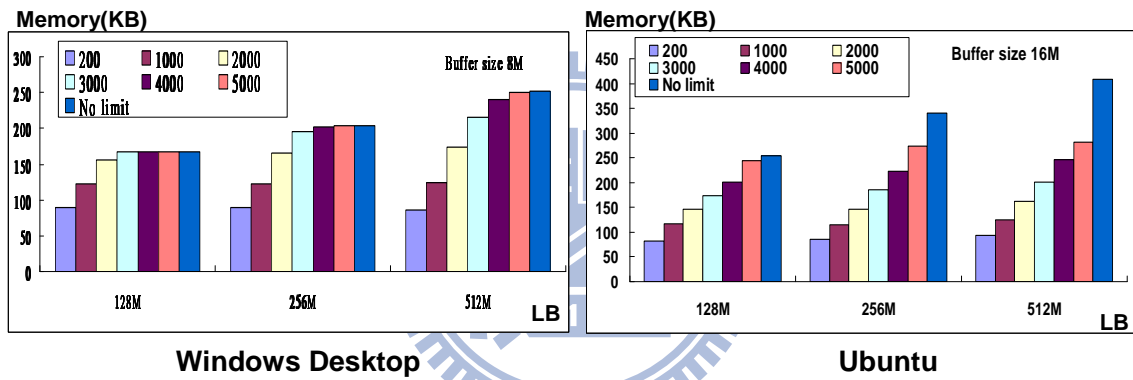


圖 0.22: Memory requirements for simulation vs. log buffer size

第五章 Conclusion

SSD 把資料儲存在 NAND Flash Memory 中, 而由於採用 outplace-update 所以在寫入資料時可能會因為閒置空間不夠而觸發 GC, 在寫入愈 random 的資料時回收 invalid 資料代價愈高, 導致愈慢的寫入效能, 這是 SSD 在一般應用上的效能瓶頸, 長久以來討論固態硬碟上無非是要將從 host 端的 write request 吸收/整理成對底層 NFTL 能夠快速處理的形式。而決定 write buffer 寫到底層 NFTL log buffer 的資料型態就是此 write buffer 的演算法 (換言之也是決定甚麼型態的資料須要留在 write buffer 中) write buffer 替換演算法大多是針對時間或空間的區域性如[4, 2, 3]或者是觀察NFTL 的運作如[9]。BPLRU 是屬於完全考慮時間區域性的演算法搭配 hole plugging, 它的缺點是無法分辨 group 中的冷熱資料及忽略了空間區域性;FAB 則是完全的考慮空間區域性, 它的缺點是假如 write buffer 空間不足難以發揮此演算法優勢及忽略的時間區域性;CLC 方法想綜合時間/空間區域性將 buffer 分成兩個 partition, 因此他有額外的參數要設定; REF的話是優先將在 log buffer 有更新資料的 group 優先當作 victim, 缺點是必須在 log buffer 空間極度缺乏的情況下才會有不錯的效能。此篇論文提出的方法是利用 simulation-assisted 的方式去發掘潛藏在 workload 中的 locality, 畢竟 workload 的 locality 是我們沒辦法掌握, 我們必須在 run-time 的時候根據我們 simulation 的結果去調整我們 write buffer 替換演算法 (也就是改變 α), 我們首要的就是能利用的 resource 不多, 而在此篇論文的提出的方法可以盡可能去減少不必要的資源浪費只存放重要且有代表性的資料而這些資料足以讓我們建構出 write/log buffer 演算法的行為, 並不會造成 simulation 出來的結果不準確也就是成功地在減少 resource 消耗和 simulation 準確度間找到一個平衡點, 讓我們的 simulation 足以放入 device 端。

參考文獻

- [1] H. Cho, D. Shin, and Y. I. Eom. Kast: K-associative sector translation for nand flash memory in real-time systems. In *DATE '09: Proceeding of the 12th conference on Design, Automation and Test in Europe*, pages 507–512, 2009.
- [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. *Consumer Electronics, IEEE Transactions on*, 52(2):485 – 493, may 2006.
- [3] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha. Performance trade-offs in using nvrw write buffer for flash memory-based storage devices. *Computers, IEEE Transactions on*, 58(6):744 –758, jun. 2009.
- [4] H. Kim and S. Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.
- [5] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *Trans. on Embedded Computing Sys.*, 6(3):18, 2007.
- [6] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue. Ccf-lru: a new buffer replacement algorithm for flash memory. *Consumer Electronics, IEEE Transactions on*, 55(3):1351 –1359, aug. 2009.
- [7] M-Systems. *Flash-memory Translation Layer for NAND flash (NFTL)*.
- [8] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee. Cflru: a replacement algorithm for flash memory. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 234–241, New York, NY, USA, 2006. ACM.
- [9] D. Seo and D. Shin. Recently-evicted-first buffer replacement policy for flash storage devices. *Consumer Electronics, IEEE Transactions on*, 54(3):1228 – 1235, aug. 2008.
- [10] S. P. D.-H. L. S.-W. L. Tae-Sun Chung, Dong-Joo Park and H.-J. Song. System software for flash memory: a survey. In *EUC '06: Embedded and Ubiquitous Computing*, pages 394–404, 2006.
- [11] Q. Zhu, A. Shankar, and Y. Zhou. Pb-lru: A self-tuning power aware storage cache replacement algorithm. In *The 18th Annual ACM International Conference On Supercomputing (ICS-04), Saint-Malo, France, June26-july*, volume 1. Citeseer, 2004.