國立交通大學

資訊科學與工程研究所

碩 士 論 文

混合式硬碟緩衝快取區管理機制

Hybrid Disk Aware Buffer Cache Management

研 究 生：曹宗恆

指導教授：張瑞川 教授

中華民國九十六年六月

混合式硬碟緩衝快取區管理機制

學生:曹宗恆　　　　　　　　　　　　指導教授:張瑞川

國立交通大學資訊科學與工程研究所

論文摘要

隨著科技的需要，越來越多的 flash memory 應用在嵌入式系統的環境中，例如 PDA、手機、數位相機、MP3 Player、錄音筆等。眾所皆知的它擁有非常多的好處，例如:非揮發性、容量大、體積小、省電、相較於機械式硬碟具耐震、不受溫度影響等。

最近由 Samsung 推出一種結合傳統硬碟和 flash memory 的混合式硬碟 H-HDD(Hybrid Hard Disk Drive)。在 hybrid disk 中，flash memory 扮演主記憶體與磁碟間 read cache 的角色。當系統想要從硬碟中讀取資訊時，hybrid disk 會先搜尋 flash memory。若有，則不須再去存取磁碟，此舉可以提升整體效能的提升與節省硬碟讀取所消耗的電量。然而，將 flash memory 作為主記憶體與磁碟間的 read cache 並不一定能夠提升系統的效能，主要是因為 flash memory 有一些先天上的限制。我們的研究針對了這些先天設計的缺點，提出一套 hybrid disk aware buffer cache 管理機制包含 buffer cache 置換演算法與 evicted-buffer 過濾機制。經過這樣的系統找出適合放入 flash memory 中的資料。由實驗結果看出，我們成功的找出這些資料並且將之放入 flash memory 當中。

Hybrid Disk Aware Buffer Cache Management

Student: Tzung-Heng Tsao Advisor: Prof. Ruei-Chuan Chang

Computer Science and Engineering College of Computer Science
National Chiao Tung University

ABSTRACT

Because of the demand in technology, Flash memory is popular to be used in embedding systems such as PDA, cell phone, digital camera, mp3 player, and voice pen et cetera. As everyone knows, flash memory has many advantages like non-volatile, low power, and relatively acceptable response time (compare to hard disk) et cetera.

Recently, a novel storage device is called Hybrid Hard Disk Drive (H-HDD) that combines traditional hard disk with flash memory to be publish by Samsung. The flash memory in hybrid disk plays the role of read cache between main memory and hard disk. Hybrid disk will search flash memory firstly when system wants to fetch data form it. It will not do disk I/O if the data is in flash memory. This method can promote system I/O performance and save the power consumption. However, flash memory to be a cache between main memory and hard disk is not sure to promote system performance. The major reason is the flash memory has some innate limits. Our research want to provide a hybrid disk aware buffer cache management focus on its innate limits. We can find the suitable data to put them into the flash memory by our method. According to the performance evaluation, we can cache the suitable data and put them into flash.

致謝

首先感謝我的指導老師 張瑞川教授。讓我能夠順利完成論文。在撰寫論文期間，尤其感謝大緯學長給我許多建議和與我討論論文的細節。也感謝身邊的同學學長的支持與勉勵，讓我能夠承受撰寫論文的壓力。

感謝我的家人，父親、母親、妹妹、以及其他親友，是你們給我完成論文的力量。最後，僅以此論文獻給我最親愛的家人朋友以及老師，由衷地感謝他們。

章節目錄

論文摘要 i
ABSTRACT ii
致謝 iii
章節目錄 iv
圖表目錄 v
第一章 簡介 1
1.1 動機 1
1.2 論文結構 3
第二章 相關研究 4
2.1 Buffer cache replacement algorithm 4
2.2 Second level cache and Cooperative Caching 5
2.3 Flash 的應用 6
2.4 Hybrid Disk 的應用 6
第三章 設計與實作 8
3.1 Hybrid Disk-Aware Buffer Cache 管理機制 8
3.1.1 Buffer Cache 置換演算法 8
3.1.2 Evicted Buffer 過濾機制 11
3.2 管理機制與核心模組的實作 14
3.2.1 Linux's Page Frame Replacement Algorithm: PARA 14
3.2.2 Hybrid Disk Aware Buffer Cache 管理機制之實作 15
3.2.3 Flash Memory Kernel Module 的實作 17
第四章 實驗 20
4.1 實驗環境 20
4.2 Hybrid disk aware buffer cache 管理機制的測試與分析 20
第五章 結論 23
參考資料 24

圖表目錄

圖一 Second level cache....5
圖二 Data 進入 FIFO 和 LRU 的動作....9
圖三 避免資料重覆的作法....9
圖四 置換演算法....11
圖五 判斷 data 是否該搬進 flash memory 的流程....13
圖六 虛擬碼....14
圖七 Linux Page Frame Reclaiming Algorith....15
圖八 演算法參數和 Linux 中 flag 的對應....17
圖九 Kernel module 中 pin table entry 的結構....18
圖十 Kernel module 在 Linux kernel 中的運作....18
圖十一 實作總圖....19
圖十二 Cache and fetch performance(512mbytes ram,512mbytes flash memory)....21
圖十三 Cache and fetch performance(256mbytes ram,512mbytes flash memory)....21

第一章 簡介

1.1 動機

隨著科技的需要，越來越多的 flash memory[25]應用在嵌入式系統的環境中，例如 PDA、手機、數位相機、MP3 Player、錄音筆等。眾所皆知的它擁有非常多的好處，例如:非揮發性、容量大、體積小、省電、相較於機械式硬碟具耐震、不受溫度影響等。

這一兩年來 Flash memory 也應用在 PC 市場上，像是 SSD(Solid State Drive)[33]、Robson technology[14]等，最近由 Samsung 推出一種結合傳統硬碟和 flash memory 的混合式硬碟 H-HDD(Hybrid Hard Disk Drive) [31]。在 hybrid disk 中，flash memory 扮演主記憶體與磁碟間 read cache 的角色[1]。當系統想要從硬碟中讀取資訊時，hybrid disk 會先搜尋 flash memory。若有，則不須再去存取磁碟，此舉可以提升整體效能的提升與節省硬碟讀取所消耗的電量。比較特別的是，在 hybrid disk 的架構下，外部系統無法直接存取 flash memory，而是必須透過數個新增的 ATA 指令[35]:其中 pin 指令可以將硬碟中某一塊區域拷貝一份保存在 flash memory 中，unpin 指令則將 flash memory 的資料設成 invalid。利用此二指令，軟體可以選擇哪些資料應該儲存在 flash memory 中以達到效能的提升或耗電量的降低。

然而，將 flash memory 作為主記憶體與磁碟間的 read cache 並不一定能夠提升系統的效能，主要是因為 flash memory 有一些先天上的限制。首先，flash memory 的資料若要重覆寫入，必須先經過 erase 的動作將資料清除，才能再次寫入。該 erase 的動作跟 flash 讀寫的速度比較起來非常的緩慢且耗電。同時，當 flash memory 在執行 erase 動作時，整個 flash memory 是無法進行讀寫動作的。因此，如果系統需要讀寫 flash memory 中的資訊就要等待 flash memory 做完 erase 的動作，如此一來就會造成系統效能的下降。再者，flash memory 每個 block 都有 erase 次數的限制，為了可以使 flash memory 可以達到其最長的壽命，所以必須以 wear-leveling 的技術來平均地 erase flash memory 的各個 block。因此，軟體在選擇哪些資料需要儲存在 flash memory 中時就必須考慮以上 flash 限制，才有助於系統效能的提升。

目前使用 hybrid disk 系統的只有微軟最新作業系統 Windows Vista 中的 ReadyDrive 這項技術。它將 kernel image 與常用的應用程式資訊 (如應用程式之 directory entries, inodes 與機器碼等)放在 flash memory 中以加速開機及應用程式載入速度。此法雖然考慮到 flash memory 的限制，不會造成太頻繁的 flash update (因為程式本身是唯讀的)，然而我們認為這樣的作法太過保守。因為其只利用 flash memory 來儲存程式本身的機器碼和些許相關資訊，並未儲存程式所使用到的資料。一些應用程式本身的機器碼不大，但可能會存取到大量資料。若我們能將其常用的資料也 cache 在 flash memory 中，將可更進一步提升系統效能。

[1] Hybrid disk 在 low power mode 時硬碟休眠，此時 hybrid disk 可將 flash memory 當做 write cache，將資料直接寫入 flash.。但是在一般情形，hybrid disk 不允許外部直接對其 flash memory 做直接寫入的動作。

在考慮將應用程式常用的資料也 cache 在 flash memory 中時，一個直覺的方法就是將 hybrid disk 中的 flash memory 當作 second level buffer cache，承接從主記憶體 buffer cache 置換出來的資料。一般來說，作業系統會將最近經常使用的磁碟資料存在主記憶體的 buffer cache 中。然而，隨著系統負載增大，主記憶體內會儲存大量的程序資料(process data, heap, stack…等)。此時，buffer cache 的容量會因排擠效應而變小，造成 buffer cache 置換頻率與磁碟 IO 頻率增加。這時若將 hybrid disk 中的 flash memory 當作 second level buffer cache，則觀念上來說即是擴大 buffer cache 的容量，有助於降低磁碟 IO 頻率的增加。

因此，在此論文中，我們將 hybrid disk 內的 flash memory 當作 second level buffer cache，並考慮上述 flash 的限制，提出了一個 hybrid disk aware buffer cache 管理機制包含 buffer cache 置換演算法與 evicted-buffer 過濾機制。當 buffer cache 需要做剔除動作時，此管理機制會挑一些適合放入 flash memory 中的資料，而將其置換到 flash memory 中。因此，這些資料雖然被置換出去，但實際上仍在 second level buffer cache 中，下次取用時仍然不需牽涉到磁碟 IO。

除此之外，此管理機制還具備以下幾個優點:

1. 減少 flash memory update 的次數。如前所述，flash memory update 會造成 erase 次數增加，降低系統效能。因此，藉由避免 cache 一些常 update 的資料在 flash 中，我們的機制可以降低 flash memory update 的次數。
2. 減少主記憶體與 flash 間資料的重複性。我們的管理機制會避免太多資料同時儲存在主記憶體與 flash 中，藉以避免儲存空間的浪費。
3. 避免 cache 連續性資料。我們的管理機制會避免將連續性資料儲存在 flash memory 中。因為硬碟對於連續性資料的存取，效能比 flash memory 更好，所以將一大串連續性資料 cache 在 flash memory 中反而會降低系統效能。

我們更改Linux作業系統的buffer cache置換演算法以實作我們的演算法並且加入我們的過濾機制。同時，我們也實作了一個kernel module來模擬hybrid disk中的flash memory。我們利用web server benchmark: **httpload**，來測量我們方法的效能改進。實驗結果顯示，我們的方法可以將符合我們要求的網頁在被系統從buffer cache剔除時，置換到flash memory當中，並且在之後的存取中被使用到。這樣一來可以預期的，我們可以降低系統的disk I0，使系統的效能有所提升。

1.2 論文結構

在第二章中，我們介紹此篇論文的相關研究。第三章將會詳細的敘述出本篇論文針對現有的問題所提出的探討與解決的方法和說明實作的細節。第四章中將敘述實驗的設計和分析實驗所得到的數據。第五章則是結論以及未來展望。

第二章 相關研究

2.1 Buffer cache replacement algorithm

因為記憶體的容量有限，系統在運作的時候，不可能將所有會使用到的資訊都留在記憶體中，所以關於 buffer cache replacement algorithm 的相關研究從 1960 年開始一直持續被探討著[4][24][10]。

A. Optimal algorithm: Belady's MIN [4]，將最遠的未來才會被使用到的 page 換出記憶體，這樣的方式被證實可以獲得最高的 hit radio，當然這樣的方式在實際系統上不可能做的到，但是依照這個方法可以求出系統利用記憶體時最好的 hit radio，能夠成為所有 replacement algorithm 比較 hit radio 的一個 upper bound。

B. Recency: LRU 演算法將最久不被使用的資料換出記憶體[24][8]。有許多近似或改善自 LRU 的演算法，像是 clock algorithm[6]。LRU 的好處是它考慮資料使用的時間關係，而缺點是完全沒有參考資料所使用的頻率，對於 weak locality 的 access patent ，LRU 的方式反而適得其反。

C. Frequency: LFU 演算法將使用頻率最低的資料換出記憶體[8][1]，它的優點考慮了資料使用的頻率，它的缺點是對於資料被使用的時間順序完全沒有顧及到，遇到 strong locality 的 access patent 將會表現的不理想。

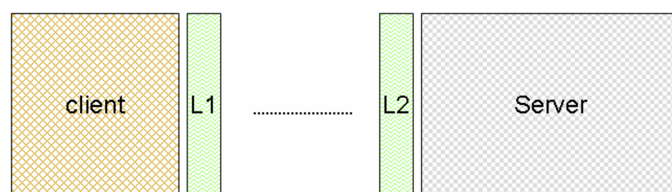
D. 混和式: LRU-K 演算法[29][28]，它基本的概念是換出倒數第 K 次被 reference 時間最晚的資料，這個方式在 K 值較小的時候可以迅速的將不常用的資訊換掉，不僅考慮到 recency 也兼顧到 frequecy。但是儲存這些資訊將會造成 overhead，並且遇到讀取頻率差不多的 access patent 時效果不彰。LIRS(Low Interreference Recency Set)演算法[16]的基本概念是考慮每個 page 從上次被 access 到最近一次被 access 之間其他 pages 被 access 的次數來決定該被置換出去的 page，利用這樣的作法來解決 LRU 只考慮 recency 所造成的問題，但在 worst case 中每次計算總是牽扯 cache 中大部分的 pages 對系統是一項非常沉重的負擔。2Q 演算法[19]是利用 FIFO 和 LRU list 結合，其目的在於將頻率不高的資料不需經過整個 buffer cache 得大小較快將其換出來達到 recency 和 frequecy 兼具的效果，其方法對於系統不需要額外的紀錄 overhead 較小，唯一的缺點就是缺乏自適性無法按照不同的 workload 做出不同的調整。LRFU 演算法[22]結合 recency 和 frequency 兩項因素來作為評分標準，利用一個 λ 值來做參數，能同時考慮兩種參數當然十分的理想但是其演算法找出要換出 page 的時間和需要儲存的資訊也相對的大，因此在實際系統中也很少有人使用這個方式。

E. 自適式: ARC(Adaptively Replacement Cache)演算法[25]，利用一些 rule 來判斷系統現下的 workload 的型態，進而對 recency 和 frequency 的比重做出調整，是一種自適性（self-tuning）的演算法。而 CAR（Clock and Adaptively Replacement）演算法[3]，是將 overhead 較輕、近似於 LRU 的 CLOCK 演算法加入原本的 ARC 來做改進。

上面介紹的演算法，所注重者皆是想要找出最近不會再被使用到的資料將之換出，以求取使用記憶體的最大利益，而我們的研究想要利用 buffer cache replacement algorithm 找出適合放入 flash memory 的資料。因此，除了原本 algorithm 的特性外還需要兼顧找出長期常被使用資訊的能力和處理 flash memory 和 buffer cache 中資料重覆的問題。我們的 Hybrid disk aware buffer cache 管理機制正是針對這些新的需求和問題，希望在演算法中加入我們所需要的新需求和解決新問題的方式。

2.2 Second level cache and Cooperative Caching

隨著科技的進步，processor 和硬碟讀取速度的差別持續的增加，為了減少這個差異，一般利用 main-memory 中的 buffer caches 來降低硬碟 I/O 次數。而這個方式在網路系統中也不例外，需求的資料除了可以在本地端的 cache 取得之外，也可以從 Web server 和其他地方的 cache 取得，現代的大量儲存系統通常都使用了大量的揮發或非揮發的記憶體來做為 buffer cache 進而提升 I/O access 的速度[11][13]，像這種型態的 buffer cache 是一種多層（multilevel）的 buffer cache 架構[27]。舉個例子如圖一，我們稱 client 端的 buffer cache 為 first-level cache(L1)，server 端的 buffer cache 為 second-level cache(L2)，當 client 端想得到 server 端的資料時，首先會去自己的 buffer cache 做尋找，發生 miss 時就對 server 發出 request 而在 server 端也會先去找尋自身的 buffer cache，如果還是沒有找到最後才會進入 server 端的硬碟找尋。但是許多研究發現原本的 buffer cache replacement algorithm 在這樣的架構下得到的表現卻不是很好，因此針對這樣 multi-level cache 的結構出現了如何增進 L1 加 L2 整體 hit radio 的研究[39]。在網路系統的應用上，為了增進 file system 的效能讓讀取遠端資料(remote data）能更快速，產生了 coordinate caching 的研究，此種研究就是將各個 client 端的 caches 分享出一塊空間能夠提供其他 client 要求的資料而不需要依賴 server，進而提升讀取資料的速度並且減低 server 的負擔，但是維持各個分散 cache 資料和 server 的一致性與資料在 client cache 和 server cache 重覆的問題會使 coordinate caching 的效能低落，因此關於這方面有許多的研究來解決或降低這些問題的產生[2][9][16][23][33]。針對資料重覆的問題在 T.M.Wong 和 J.Wilkes 2002 年的研究中[36]提出了 DEMOTE 的想法。我們的研究考慮到 main memory 中的 buffer cache 和 hybrid disk 中 flash memory 的 read cache 也是一個 second level cache 的架構，因此將 DEMOTE 的想法應用在我們的演算法中來降低 flash memory 和主記憶體中資料重覆的問題。



圖一 Second level cache

2.3 Flash 的應用

在 hybrid disk 之前也有許多 flash 的應用使用在不同的地方，像是當作 mp3 的主要儲存裝置時提出減少寫入和 erase 次數的 flash-aware buffer cache 的管理策略[18]和將 flash memory 搭配少量的 RAM 來取代全部用 RAM 來做記憶體的方式[12]還有為 flash disk 制訂不同於硬碟的 log file system[37]，或是針對 flash 本身的限制，例如：壽命、讀寫速度不平衡等缺陷，做出延長使用壽命和增加使用效能的相關研究[7][20]。在 eNVy[38]的研究中提出對於 erase 的策略會直接影響整個 flash device 的效能。如何在 erase 成本和均勻的 erase 每一個 block(wear-leveling)之間找出臨界點也有許多的研究[20]。在手機上面也出現了異質大量儲存裝置(Hard disk 加上 flash memory)，因此有研究提出一個針對不同大量儲存裝置和讀取形態將 main memory 中的 buffer cache 做裝置取向的切割，使其擁有各自的 buffer cache，並且依照各個裝置的效能做 cache 大小調整的 buffer cache 置換演算法[21]。而我們的研究針對了 flash memory 的每個 block 有抹除上的限制和 erase 費時耗電等缺陷，提出只將長期常用的 read-mostly data 存入 flash memory 中的作法，來降低 erase 的頻率，進而提升 flash memory 所帶來的效能和增加其使用壽命。但因為現階段 ATA8 的架構，系統並無法控管 flash memory 本身 wear leveling 的作法，因此我們的研究並不牽涉 hybrid disk 內 flash memory wear-leveling 的作法。

2.4 Hybrid Disk 的應用

目前只有微軟的新一代作業系統 Windows Vista 有利用 hybrid disk 來進行系統效能的提升。Vista 使用了一種叫做 ReadyDrive 的 policy 來利用 hybrid disk 中的 flash memory[26]。

ReadyDrive 將 H-HDD 裡面的 flash disk 劃分成兩區:

Write cache 區:

為了節省硬碟耗電量，Readydrive 會每隔一段時間將硬碟 shut down，shut down 其間發生的資料更新則先寫入 flash memory 中以節省硬碟耗電量。

Image 區:

A. OEM data 存放區:將 OEM 廠商的 application image 放入 flash disk 當中，可以使其更快速的 launching。
B. kernel image 存放區:將系統開機時要用到的 kernel image 先放在 flash memory 中，能夠使開機更加快速。
C. Superfetch 存放區:將統計出常用的 application image 存放到 flash memory 讓 application 執行起來的時間更快。

針對 Image 區的 Superfetch 存放區，如簡介所述，因為其只利用 flash memory 來儲存程式本身的機器碼和些許相關資訊，並未儲存程式所使用到的資料。而我們的方法能將其常用的資料也 cache 在 flash memory 中，更進一步提升系統效能。其他部分的確為系統效能帶來好處，未來希望我們的研究可以和原始 VISTA 的 ReadyDrive 合作，令系

統效能更加提升。

第三章 設計與實作

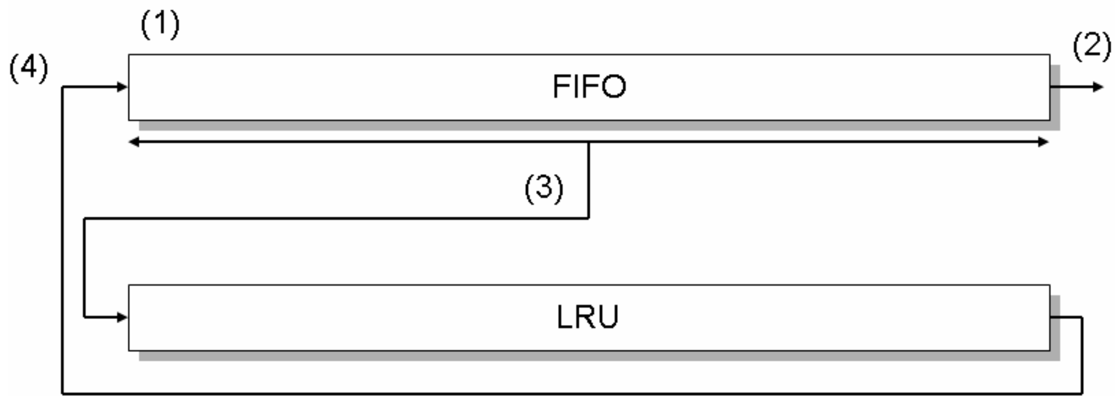
3.1 Hybrid Disk-Aware Buffer Cache 管理機制

在這一節，我們將介紹我們所提出的hybrid disk aware buffer cache的管理機制，包含buffer cache置換演算法與evicted-buffer過濾機制。如前所述，此buffer cache管理機制將hybrid disk內的flash memory當作second level buffer cache。當主記憶體的buffer cache需要置換時，其buffer cache置換演算法會剔除一些buffers，而evicted-buffer過濾機制則會從被剔除的buffers中選擇一些適合放入flash memory內的資料，將其放入flash memory中。

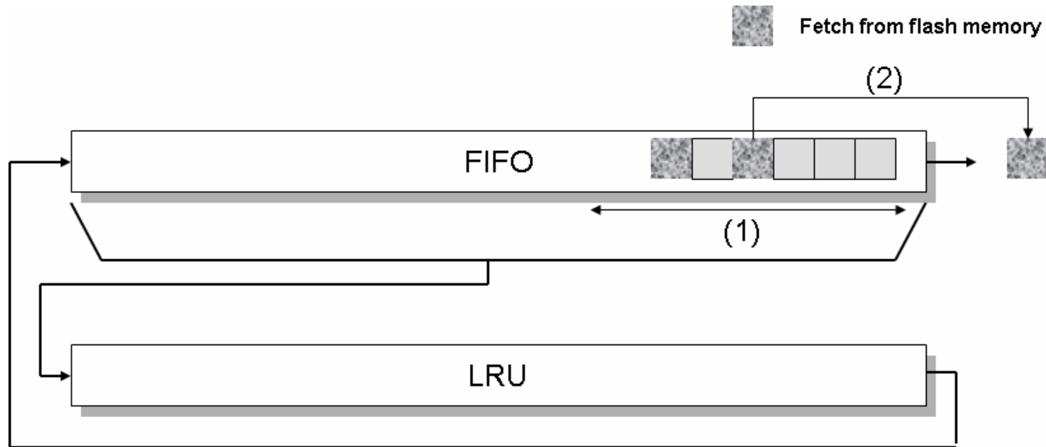
雖然flash memory有著非揮發性和隨機讀寫速度比硬碟快的好處，但它在使用上有著相當多的限制，諸如其erase-before-rewrite的特性及erase次數的上限等。因此，此管理機制在選擇資料放入flash memory時必須考量到這些限制，才得以達到資料存取效能的提升。為了考量這些限制，此管理機制會動態地擷取page存取的特性，並進行進一步的分析，找出硬碟中的哪些資料應該被儲存在flash memory裡面以提升系統效能。以下我們分別描述buffer cache置換演算法與evicted-buffer過濾機制的設計。

3.1.1 Buffer Cache置換演算法

首先，如圖二所示，我們的演算法將buffer cache分成了FIFO list和LRU list。LRU list用來保存最近使用過的資料。然而，眾所周知地，若整個buffer cache只有一個LRU list，則一個大量的sequential access就可以將系統常用的資訊從buffer cache中剔除出去。為了避免這種情況發生，我們多加了一個FIFO list。當資料從硬碟讀取入主記憶體時，它會先被放入FIFO list的前端。在FIFO list中使用頻率高的資料才會被搬入LRU list當中。因此，只使用到一次的資料會存在FIFO list中。當進行buffer cache置換時，FIFO list會從尾端被置換，而LRU list內的資料不會被置換。因此一個大量的sequential access也無法將系統常用的資訊從buffer cache中剔除出去。



圖二 Data 進入 FIFO 和 LRU 的動作



圖三 避免資料重覆的作法

此外，此演算法必須減少主記憶體與flash memory內資料的重複性。所有的second level cache都會有資料重複的問題[36]，當然此處也不例外，當我們不對主記憶體和flash memory中cache的重複資料做妥善處理時，我們可能會面臨兩邊資料過於重複的窘境。這個問題會造成系統在主記憶體找不到資料(即miss)時，在flash memory也沒有的機率變得非常高，如此一來second level cache也就喪失了原本的功能。因此，我們的演算法應該要避免記憶體和flash disk資料的重複。

為了減少main memory和flash間資料大量重複的缺點，我們提出了一個簡單的方式，如圖三，當讀取資料時，若資料是從flash memory中讀取上來，我們利用一個叫做**already_in_flash**的參數記錄這件事情，在buffer cache要剔除 block的時候，我們從FIFO list尾端往前搜尋一定的長度(圖三(1))，優先置換此長度中最尾端的一個從flash memory讀上來的資料(圖三(2))。這樣的做法除了可以降低資料重複率之外，也考慮到因為從flash

memory讀取資料比硬碟快，所以將靠近尾端的一個從flash memory讀上來的資料優先換出，下一次要讀取這份資料時所花費的時間也比從硬碟中拿其他資料比較起來負擔較小。圖四即為我們buffer cache的置換演算法。我們假設LRU.head與LRU.tail分別代表LRU list的頭尾。FIFO.head與FIFO.tail也分別代表FIFO list的頭尾。而demote_length代表要從FIFO list尾端搜尋的長度。Cache的絕對值代表現有在buffer cache的block數目，而Cache max size代表buffer cahce中最多能保存的block數目。首先，我們先假設X是一個系統要存取的block。如果X在LRU list中，我們就將X移至LRU.head。否則若X在FIFO list裡面，我們會將X移去LRU.head。如果X不在兩個list中，表示X是從hybrid disk讀上來的資料，但是因為X原本不在Cache之中，而我們要將X移入FIFO list所以我們先檢查Cache是否有多餘的空間。若沒有，就呼叫evict()這個函式來將FIFO.tail的block剔除。接者將X移到FIFO.head並且檢查X是否是從flash memory中取得。如果是，我們將此block的**already_in_flash**的參數設成1。

在evict()函式中，我們先檢查FIFO list後面一段blocks（長度為demote_length）。如果在這段距離中有任何block的already_in_flash參數為1，我們就將這個block剔除。如果在demote_length的距離中，沒有任何block的already_in_flash參數為1，就將FIFO.tail所對應的block剔除。而在剔除FIFO的block之後，為了維持FIFO list和LRU list的數量平衡，我們利用adjust_list( )這個函式在cache做剔除時去調整FIFO list and LRU list之間平衡，其方式就是汰換一些長久在LRU list之中不被使用的block進入FIFO list中。

```
On accessing a block X:

If X is in LRU then
     move X to LRU.head

                    else if X is in FIFO
     insert X to LRU.head

else
     if (Cache = = Cache max size)
           evict();
     insert X to FIFO.head
     if( X was from flash disk)
           set X.already_in_flash as 1
```

```
evict()
{
     for(i=0;i++;i<demote_length)
           j = FIFO.tail - i
           Y = jth element of the FIFO list
           if Y.already_in_flash is 1
                evict this block
                adjust_list( );
                return
     evict the block corresponding to FIFO.tail
     adjust_list( );
     return
}
```

圖四 置換演算法

3.1.2 Evicted Buffer 過濾機制

如前所述，evicted-buffer過濾機制會從被剔除的buffers中選擇出適合放入flash memory內的資料。明確來說，此過濾機制會將具有下列特性的資料放入flash memory中:

不常修改的資料: Flash memory 資料的更新通常使用non-in-place update方式，將新的資料儲存在其他未使用的page中而將原資料設成無效(Invalid，Garbage data)。若我們搬進的資訊常常被更新，那麼無效資料將隨著資料的更新不斷地增多，可用資料變少，erase的頻率也會隨著提高。另外，hybrid disk在一般模式下不能作為write cache，資料的寫入仍然需要磁碟IO。因此將經常更新的資料放入flash memory中不但無法提升寫入效

能，更需要額外負擔來作資料在磁碟與flash memory間的同步。鑑於此，我們只會將read-mostly的資料搬入flash memory中。

系統長期常用的資料:我們使用flash memory來做為second level cache就是希望讓系統在主記憶體buffer cache找尋不到資料時可以從flash memory裡面獲得資料以減少硬碟的存取。換句話說，flash 本身就是要承接buffer cache移出的資料來保留更多的硬碟資訊。而被移出的資料有三種情況，第一是只被讀取1次的資訊。第二是曾經短期常用的資訊，第三是長期常用的資訊(i.e., 在workload大的時候，這些資訊仍有可能從buffer cache被剔除出去)。因為第一和第二種情況並不能保證這些資料之後將會很常被使用到，如果在flash memory中放入不常用的硬碟資訊不但失去了我們想要達成的目標，還要付出將資料寫入到flash memory的時間並且排擠到其他資料的存放空間。而且因為資料在buffer cache的時間不夠長，我們認為以這樣短的時間來判斷資料是否為read-mostly也不夠充分。經由以上幾點理由，因此我們只預期將第三種長期常用的資料放到flash memory當中。

非大量連續性的資料: 因為rotation time和seek time的影響，所以硬碟在做random access的速度上比flash memory來的緩慢許多。但是當硬碟在做sequential access的時候，如果access的資料量超過170 Kbytes~200 Kbytes，則硬碟的讀取速度就會比flash memory讀取的速度來的快。由上可知，將一個在硬碟中size超過200 Kbytes的連續資料搬進flash memory裡面是不明智的。因此我們需要避免放入連續資料到flash中。

為了判定一個從buffer cache被剔除掉的buffer是否具有以上特性，我們必須與buffer cache的管理機制合作，動態地記錄buffers的特性。以下我們說明在3.1.1節所提及的buffer cache management algorithm中應該做哪些記錄以及如何做這些記錄。

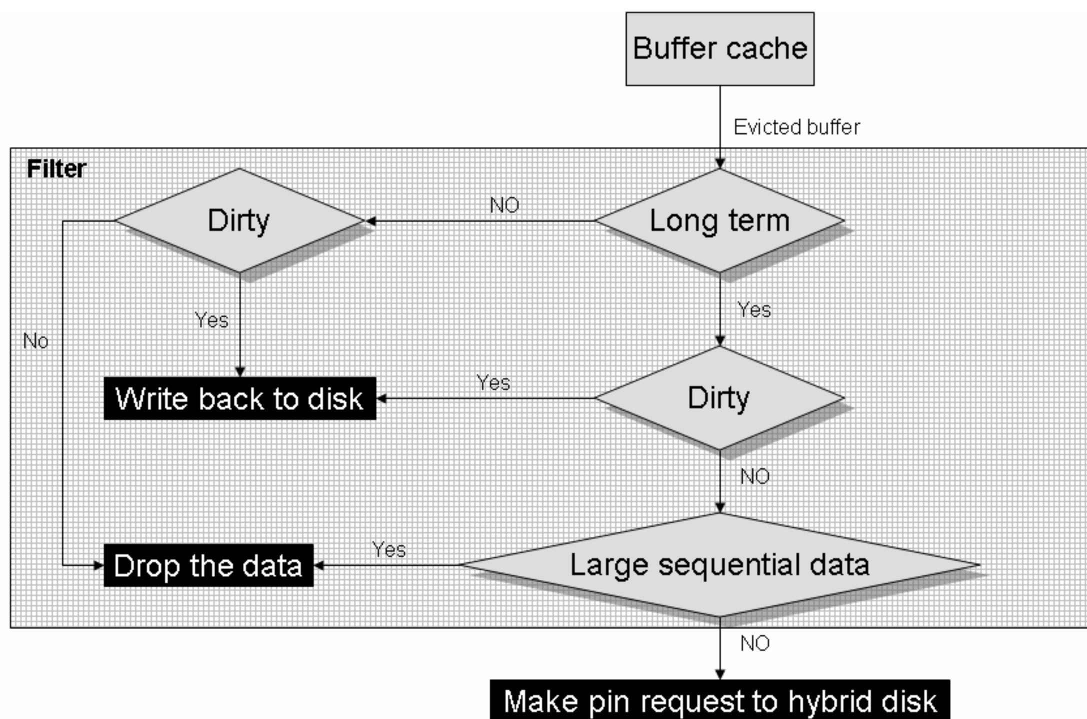
為了判斷一筆資料是否常用，我們會檢視其是否進入過LRU list。如前所述，常用的資料才有機會進入LRU list，而只使用過一次的資料只會在FIFO list中。然而，我們要判定的不只是常用資料，而是長期常用的資料。為達此一目的，我們利用**enter_lru_count**這個參數，當資訊進入LRU list時此參數就會加1，而若是此資料持續不被使用，它將會再一次的回到FIFO list裡面。當這些資料在FIFO list中又被頻繁使用時，會再次回到LRU list。當資料的**enter_lru_count**大於或等於某個threshold N時，表示其已經至少進入LRU list N次，所以我們把此資料判斷成是長期有被系統使用的資料。

為了要選出read-mostly的資訊，我們利用一個叫做**dirty**的參數。當資料被修改過，此參數就會被記錄成1，我們選取從頭到尾**dirty**皆為零的資料來做為read-mostly資料的依據。因為**dirty**為零代表此資料在進入主記憶體後一直未被修改過，所以我們認為其未來修改的機率也不大，放入flash memory中應該不會造成flash 經常性的update。然而，因為我們只觀察一段時間，所以我們不敢保證其永遠不會update，這也是我們稱其為read-mostly，而不是read-only的原因。

如前所述，我們也必須避免屬於大量連續資料的blocks進入flash memory。然而，因

為我們無法得知每個block的資料被讀取進來時是否是屬於大量連續資料的一部分，所以我們用了一個更保守的策略:避免cache大檔案到flash memory中。大部分的連續資料存取是發生在對某一檔案的循序存取，因此此一策略可以避免大量連續資料進入flash memory中。然而，對某一檔案的循序存取並不完全是大量連續資料的存取，有些大檔案的資料分佈仍然是碎裂的(fragmented)，這些碎裂資料即使常用也會被我們目前較保守的策略排除在flash memory外。

圖五顯示我們的過濾機制的流程，圖六則為此過濾機制的虛擬碼(pseudo code)。首先，當一個evicted block被選出時，我們會先利用**enter_lru_count**這個變數來判斷此evicted block是否為長期(long term)被使用的資料，如果**enter_lru_count**小於我們設定的threshold N，就不符合我們的標準，此時我們判斷block是否dirty將它做write back或直接丟棄的處理。當此資料既是long term被使用的資料並且沒有被修改過時，我們檢查此block所代表的檔案大小有沒有超過200Kbytes。如果有，代表其可能屬於大量連續性資料的一部份，不適合放入flash memory中，因此我們就將之丟棄。如果沒有，就代表此一block是符合所以上面我們所提及特性的目標，因此對hybrid disk下pin的指令，將這個block複製一份到flash memory中。



圖五 判斷 data 是否該搬進 flash memory 的流程

```
Evicted data X:

Evicted_buffer_filter()
{
        if( X.enter_lru_count >= N )
            if( X.dirty = 0)
                if( file size of X < 200Kbyte)
                    Pin X to the hybrid disk
                else
                    drop the data
            else write back to disk
        else
            if(X.dirty = 0)
                drop the data
            else
                write back to disk
}
```

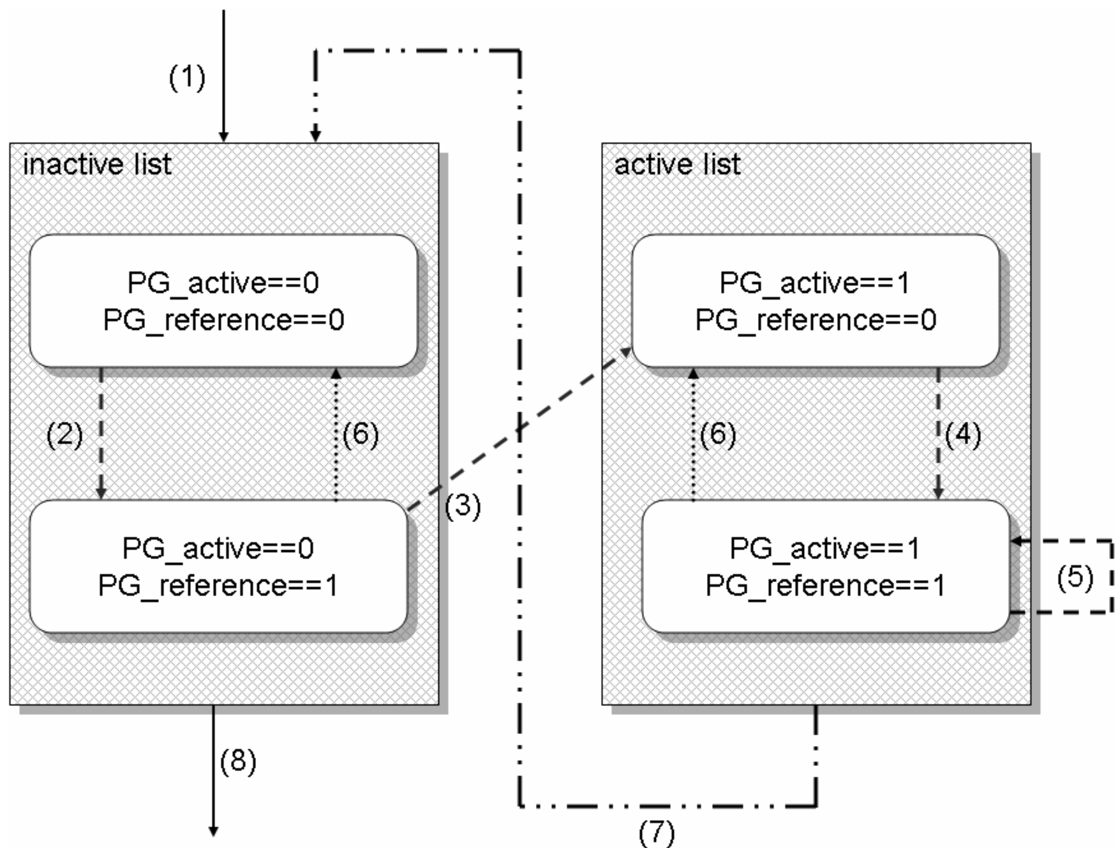
圖六 虛擬碼

3.2 管理機制與核心模組的實作

在此節，我們說明如何將所提出的 hybrid disk aware buffer cache 管理機制實作在 Linux 作業系統中。3.2.1 節我們先介紹 Linux 本身的 page frame replacement algorithm 的運作方式。3.2.2 節敘述如何將我們的演算法和 evicted-buffer 過濾機制整合進入 Linux buffer cache 置換演算法中。因為在這篇論文實作時，Linux 還沒有 hybrid disk 的驅動程式，所以我們無法在 Linux 上直接使用 hybrid disk 來進行相關的實驗。因此，我們寫了一個 kernel module 來模擬 hybrid disk 的各項運作，在 3.2.3 節將會介紹此 kernel module 的架構和其功能。

3.2.1 Linux's Page Frame Replacement Algorithm: PARA

在 Linux 中，所有 memory pages (包含 buffer cache 內之 pages)的 reclaiming 都是經由一個整合的演算法來執行，此演算法稱為 Page Frame Reclaiming Algorithm (PFRA)。在此演算法中，memory pages 會被放入 inactive list 或 active list 內，前者用來保存最近一段時間內未被存取的 pages，而後者用來保存最近常被存取的 pages。



圖七 Linux Page Frame Reclaiming Algorith

Linux 用 page descriptor 資料結構來描述每個 page 的一些基本狀態。該資料結構中有一個稱為 PG_reference 的 flag 用來記錄此張 page 最近是否有被使用到，此 flag 就是 Linux 用來判斷 page 使用頻率高低的依據。如圖七所示，當一張 page 進入 page cache 時，首先進入到 inactive list 中(如圖七(1))，當它被存取到時，系統會檢查其 PG_reference。如果為 0，將其 PG_reference 設成 1(如圖七(2))。如果 PG_reference 為 1，系統會將這張 page 從 inactive list 移到 active list 的前端(如圖七(3))，這是因為這張 page 被 Linux 認為是最近常被使用的 page。在 active list 當中 page 被存取時，也會將 PG_reference 設成 1(如圖七(4)和(5))。當一個 page 有一段長的時間不被使用系統會將它的 PG_reference 設回 0(如圖六(6))。而在記憶體不足時將 active list 裡最近沒被使用的一些 pages 從 active list 尾端移回 inactive list 頭端(如圖七(7))，並且將 inactive list 尾端的 pages 作為 evicted pages 從 cache 中剔除(如圖七(8))。而 Linux 本身也有一套自己的方式決定每次要從 active list 移回到 inactive list 的 page 數目和從 inactive list 剔除的 page 數目來調整兩個 list 之間的大小平衡。Linux 藉由以上這樣的方式來保存系統常存取的 pages 並且剔除系統最近不常用的 pages。

3.2.2 Hybrid Disk Aware Buffer Cache 管理機制之實作

在 3.2.1 裡面我們介紹了 Linux 如何進行其 page frame 置換演算法，在此節我們進一步的介紹如何將我們的演算法和 evicted-buffer 過濾機制修改進入 Linux page frame 置換演算法中。

3.2.2.1 Hybrid disk aware buffer cache 置換演算法之實作

首先，我們的演算法有一個 FIFO list 和一個 LRU list，幸運的因為 Linux 中 inactive list 和 active list 的特性和我們的演算法雷同，因此我們就將之對應。再者，Linux 利用 page descriptor 中的 PG_reference flag 來判斷 page 是否最近經常被使用，因此我們就直接利用此 flag 來當作我們原本判斷資料最近使用頻率高低的依據。接者，在 Linux 的 page descriptor 中有一個 unsigned long 型態名為 **flags** 的變數，裡面包含了 20 個 flags，而我們利用剩下來的 12 個空的 bit 來代替其他在我們演算法中的參數。我們演算法中 **already_in_flash** 的參數，利用一個叫做 PG_from_flash 的 flag 作為代表，當 page 是從 flash memory 取得時，此 flag 就會被設成 1。所以當我們在找尋要從 inactive list 剔除出去的 page 時，我們可以利用檢查 inactive list 的後面一段位置有沒有 PG_from_flash 為 1 的 page，有就先將它置換出去，來達成我們在 3.1.1 中所提到，避免 buffer cache 和 flash memory 中資料過分重覆的方法。

3.2.2.2 Evicted-buffer 過濾機制之實作

接著將介紹在 Linux 中我們如何對應剩下來的其他參數來完成我們的 Evicted-buffer 過濾機制的實作。圖八是我們演算法參數和實作在 Linux 中對應的 flags，以下我們將會介紹在 Linux 中我們怎麼利用這些 flags 來代替我們原本的參數。首先，為了實作的方便，原本我們演算法中的 **enter_lru_count** 參數，在 Linux 裡我們利用取名叫做 PG_ever_active 和 PG_to_flash 這兩個 flags 來代替。PG_ever_active 這個 flag 所表達的意義是『此張 page 是否曾經進入過 active list』，當 page 從 inactive list 移動到 active list 時，我們會將這個 flag 設成 1。接著，PG_to_flash 是當 page 進入 active list 時，我們檢查其 PG_ever_active 是否為 1，若為 1 我們將 PG_to_flash 也設定成 1，用來代表這個 page 不但使用頻率高並且長期會被使用到(因此，在實作時我們認定 2 次進入 active list 的 page 是長期常用之 page)。再者，我們演算法中的 **dirty** 參數，我們利用一個 PG_ever_dirty 的 flag 來代表。Linux 也有一個 PG_dirty 的參數來判斷該 page 與其對應的磁碟區塊是否一致。在做完 write back 的動作後，該 page 之 PG_dirty 就會被清為 0。而我們要記錄的是長期都不被修改的 read-mostly data，因此無法直接利用 PG_dirty 這個 flag。我們利用 PG_ever_dirty 這個 flag，當 PG_dirty flag 被設成 1 時，PG_ever_dirty 也會跟著被設成 1，但是 PG_ever_dirty 在 page 被系統 free 掉之前不會做清除的動作，所以我們可以利用這個 flag 來做為 read-mostly data 的依據。

最後，為了判定一個 page 所屬檔案之大小，我們利用 page descriptor 中的 inode 資訊，找出此張 disk data page 所代表檔案的檔案大小做為讀取大小的判斷，來完成過濾 large sequential data 進入 flash memory 的目的。

<table>
<tr><td rowspan="2">enter_lru_count</td><td>PG_ever_active</td></tr>
<tr><td>PG_to_flash</td></tr>
<tr><td>already_in_flash</td><td>PG_from_flash</td></tr>
<tr><td>dirty</td><td>PG_ever_dirty</td></tr>
</table>

圖八 演算法參數和 Linux 中 flag 的對應

3.2.3 Flash Memory Kernel Module 的實作

因為在撰寫這篇論文的當下，我們沒有 Linux 對於 hybrid disk 的驅動程式。但為求證實我們演算法的效能，因此我們做了一個模擬 flash memory 的 Linux kernel module。我們沒有利用 USB thumb drive 是因為它使用的是 USB 的 interface 不是 IDE 的 interface 並且有其 software protocol 的 overhead，做出來的結果並不會比較準確。而我們在 kernel module 中動態的加入 flash memory 使用時的 time delay 可以更接近 hybrid disk 的架構，以下將以五方面描述此 kernel module 之實作:

1. **Flash memory 的模擬:**

   我們的 kernel module 以主記憶體 DRAM 來模擬 hybrid disk 中的 flash memory。Kernel module 向記憶體要求部份空間當作 flash memory 的空間。我們利用了 Linux 內建要求記憶體的函式__get_free_pages()向系統要求記憶體空間。在插入 kernel module 時，我們以一次配置一張 page 的方式呼叫此函式將整個 flash memory 所需要的記憶體配置出來。

2. **Pin 的動作:**

   我們透過將資料複製到我們 kernel module 所配置之記憶體的方式來模擬 pin 的動作。此外，為了能夠在爾後找到該筆資料，我們將該資料在磁碟上之 block number 與在 kernel module 所配置之記憶體中之位置關聯起來。這些關聯存在一個 pin table 中。圖九顯示每一個 pin table entry 之格式。

3. **供應資料的動作:**

   如圖十，當系統需要的硬碟資訊不在記憶體時，系統會對硬碟發出 request，我們會去攔截這樣的需求，利用 block number 先去比對 kernel module 裡面保留的資料有沒有符合這次需求的資料並且檢查其 invalid bit 是否為 0，如果有這個資料並且 invalid bit 又為 0，就將資料從 kernel module 所配置之記憶體區段中複製一份到需要的 page frame 當中，並且回應系統此次要求已經完成，讓系統繼續運作。如果沒有，則回到系統本來讀取硬碟資料的做法來完成讀取的動作。

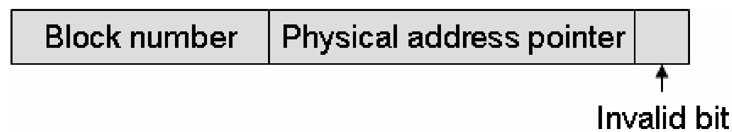
4. **Unpin 的動作:**

   當從 kernel module 裡面取得的資料被修改時，我們會將 pin table 中的相對應的 entry 設成 invalid。

5. **模擬 flash memory 速度的動作:**

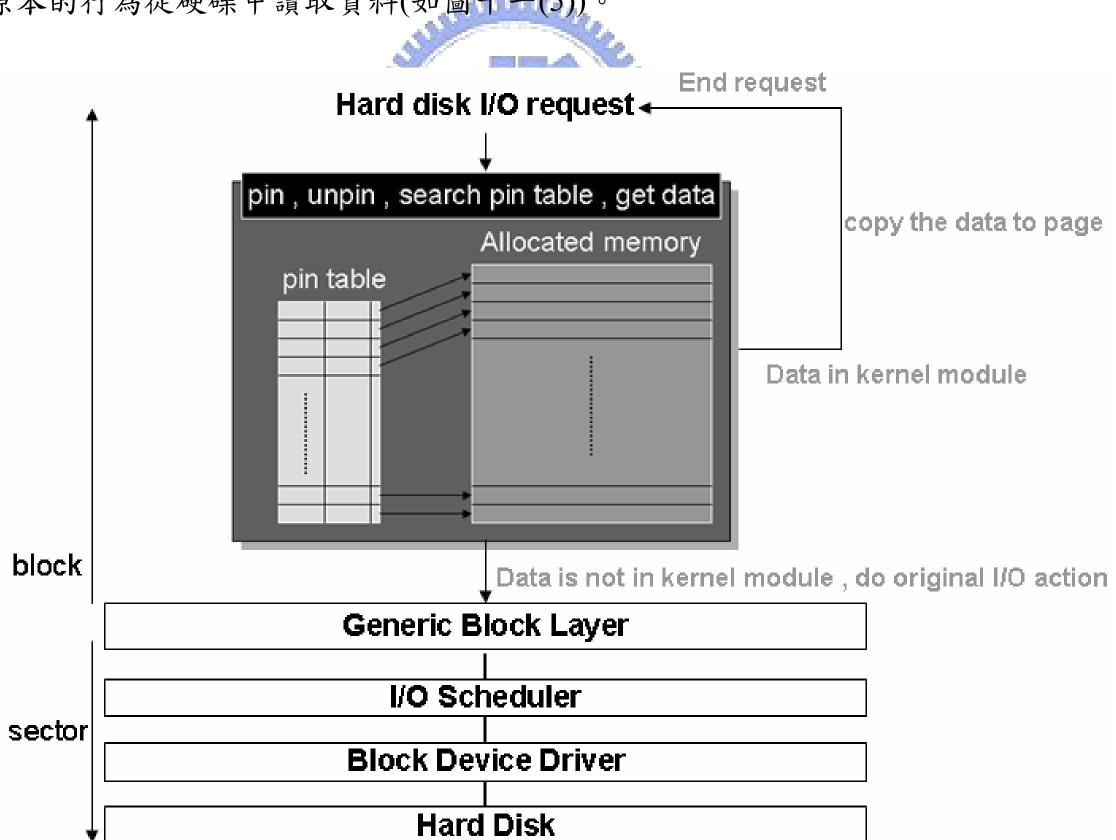
   我們參考了之前 flash memory 方面的文獻[32]，在 kernel module 裡面加入了 time delay 以模擬 flash memory 讀取和寫入，使此模擬更加接近實際的狀況。此外在 flash memory 中進行 erase 運算所造成的 overhead 比較難以模擬正確的時間，因為這牽涉到 cleaning policy 的運作。而此 policy 則是內建於 hybrid disk 的 firmware 中，所以我們在無法得知某一特定的 hybrid disk 所用之 cleaning policy 的情況下，不容易去準確地模擬出其效能。因此，我們採用一個簡單的 cleaning

policy:當 kernel module 沒有任何空間放入複製的 page 時，kernel module 會利用我們的 pin table 找尋在 table 中 invalid bit 為 1 的 entry 將其 block number 設成 0，以代表此區的資料被清除。而在 flash memory 中 erase 的動作是以 block 為單位，一個 block 一般來說是 16KBytes 的大小，而一個 entry 代表一張 page 的資料是 4KBytes。因此，以 4 個 entry cleans 做為一次 erase 計算加入 erase 所花費的時間加在 kernel module 之中。來計算 flash erase 所造成的時間。企圖更加接近實際所花費的時間。

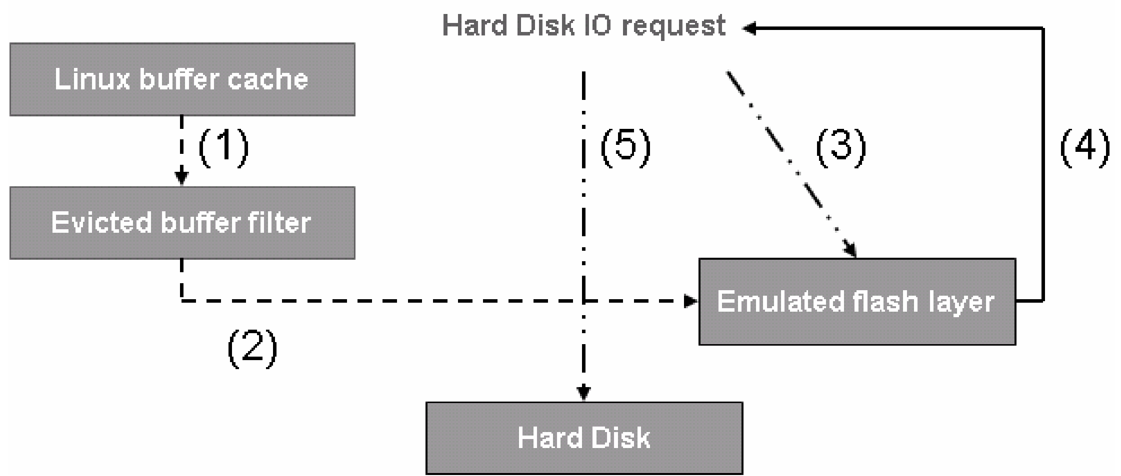


圖九 Kernel module 中 pin table entry 的結構

最後圖十一是資料在我們實作下的整體流程。Linux buffer cache 置換出來的資料(如圖十一(1))透過我們 filter 的挑選，將符合我們需求的資料保存在 kernel module(emulated flash layer)中(如圖十一(2))，在系統對硬碟作 IO request 時，先檢查此 layer 是否有資料(如圖十一(3))，如果有就回應系統需求並且結束此次 IO 流程(如圖十一(4))，如果沒有就按照原本的行為從硬碟中讀取資料(如圖十一(5))。



圖十 Kernel module 在 Linux kernel 中的運作



圖十一 實作總圖

第四章 實驗

本章對於我們所提出的hybrid disk aware buffer cache管理機制的演算法和evicted-buffer過濾機制進行效能測試，4.1節敘述我們的實驗環境，4.2節敘述加入我們機制的效能比較。

4.1 實驗環境

我們選擇Linux做為我們實驗的平台。我們的實驗包含兩台機器，如表一所示，這兩台機器一台當作Web server一台當作client端。我們利用httpload這個Benchmark以測量網路傳送與接收的效能，而因為我們的實驗希望可以更貼近真實的使用方式，因此我們取得了交通大學圖書館的web log做為我們access list。

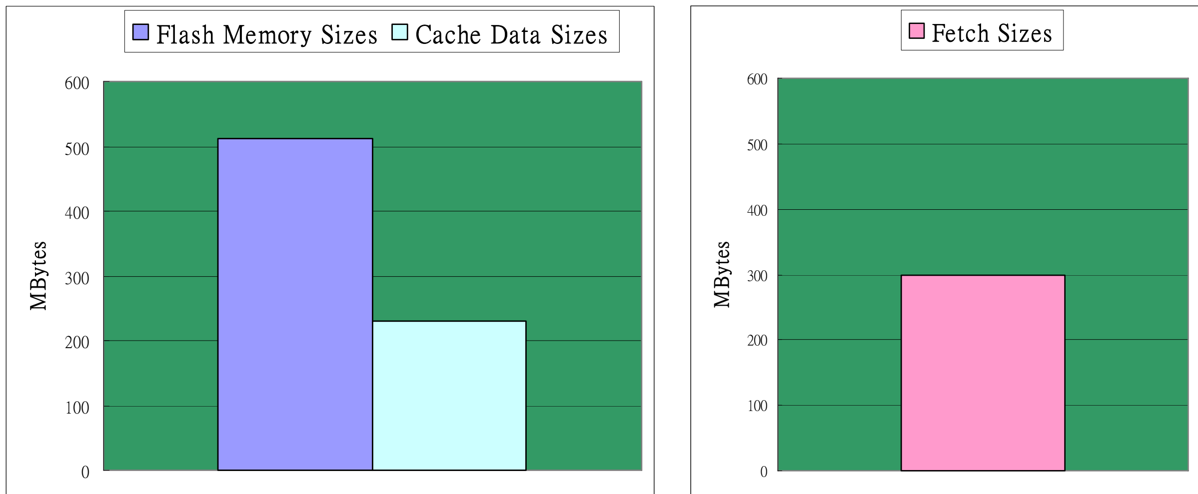
	Server	Client
Hardware	CPU: Intel(R) Pentium(R) 4 CPU 1.60GHz Memory: DDR 400 512MB~2GB Hard disk: Seagate 60GB 7200RPM NIC:Accton Technology Corporation EN-1216 Ethernet Adapter	CPU:Intel(R) Pentium(R) 4 CPU 1.60GHz Memory: DDR 400 256MB Hard disk: Seagate 60GB 7200RPM NIC: Accton Technology Corporation EN-1216 Ethernet Adapter
OS	Linux 2.6.19	Linux 2.4.18

表一 實驗環境

4.2 Hybrid disk aware buffer cache 管理機制的測試與分析

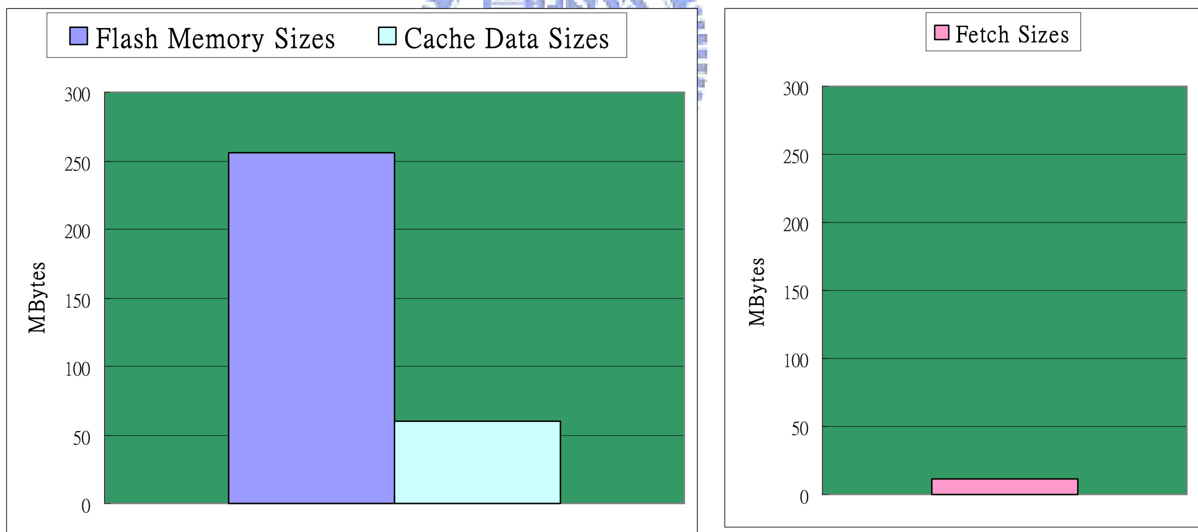
在這一節我們將利用多個web clients對web server的資訊作存取來測試我們的管理機制是否能夠獲得好處。

首先，我們知道當資料流超過buffer cache可以負擔的大小時，buffer cache會依照系統的replacement機制將不常用的資訊搬出，為了加速我們的實驗，因此我們準備了4096個128KB的小檔案和8個64MB的大檔案作為working set，在短時間之內利用多個client去隨機存取這些資料。我們固定系統可以使用的記憶體空間，讓workload超過記憶體的負載，因而開始持續的做buffer cache replacement的動作，以下是以512MB main memory和512MB的flash memory,隨機讀取work load 10次所做出的實驗結果，如圖十二:



圖十二 Cache and fetch performance(512mbytes ram,512mbytes flash memory)

從圖中我們發現，經過 10GB 的 workload 之後我們將 220MB 左右的小檔案抓入 flash memory 當中，而在讀取 10GB 的過程，flash memory 一共提供 300MB 左右的 IO 量，也就是說我們成功減少了 300MB 的 disk IO，但是之後在做一次 10GB 讀取時並沒有顯著的提升 flash memory 中有的資料。對此，我們懷疑是因為記憶體太大，造成絕大部分的小檔案在 buffer cache 裡面充斥著，根本不需要去做 disk IO，因此我們將 main memory 的大小調成 256MB 再做了一次實驗，以下是實驗的結果



圖十三 Cache and fetch performance(256mbytes ram,512mbytes flash memory)

從圖中，我們看到了竟然因為 main memory 大小變小，使得原本 flash memory cache 到的小檔案變的更少。原因是因為 main memory 過小，造成 page 幾乎在 buffer cache 裡面都待的不久，因此 page 很少能夠通過 long-term 的標準，最後造成 block 進入 flash memory 過少的情況。

因此我們發現，此方法在 main memory 太大時，因為根本不需要去做 disk IO 而造成效能低落，在 main memory 過小時，又會發生沒有辦法符合 long-term 標準的缺點，以致於此方法在使用上不論何種情況皆沒有辦法發揮好的效能，因此在設計上可能要再

次做思考。

第五章 結論

從實驗的結果推知，我們的方法雖然可以找出具有 long-term、usually used 和 read mostly 的資料，將之放至 flash memory 裡面來增進系統的 disk IO 效能，但是因為我們的方式可能對於 disk block 的統計只限於 main memory buffer cache 運作的時期，其實並無法長時間的對每個 disk block 的使用做長時間的統計。因此，這樣的方式有可能因為統計時間不長而失去意義。再者，我們為了 hybrid disk 本身的限制所以訂出要 long-term 且 usually used 和 read mostly 的三個標準，造成進入 flash memory 的條件變得非常難以達成，也因為這樣的情況造成 flash memory cache disk block 並不多，而無法有效降低 disk IO 的次數。最後，在實際系統的運作當中，會進入 flash memory 裡面的資訊非常的少量，以至於整個機制在實際的情形下並沒有辦法增進系統效能。

參考資料

[1] A. V. Aho, P. J. Denning, and J. D. Ullman, "Principles of optimal page replacement," J. ACM, vol. 18, no. 1, pp. 80–93, 1971.

[2] M. Aron et al , "Scalable Content-aware Request Distribution in Cluster-based Network Servers" , 2000 Annual USENIX Technical Conference, San Diego , CA , June 2000.

[3] S. Bansal and D.Modha, "CAR: Clock with Adaptive Replacement" , Proceedings of the 3nd USENIX Symposium on File and Storage Technologies, March, 2004.

[4] L. A. Belady,"A Study of Replacement Algorithms for Virtual Storage" , IBM System Journal, 1966.

[5] Daniel P. BOVET, MARCO CESATI , Understanding the LINUX Kernel 3e , Sebastopol USA,November 2005.

[6] W. R. Carr and J. L. Hennessy, "WSClock – a simple and effective algorithm for virtual memory management," in Proc. Eighth Symp. Operating System Principles, pp. 87–95, 1981.

[7] M.-L. Chiang, R.-C. Chang ,Cleaning policies in mobile computers using flash memory , May 1998.

[8] J. E. G. Coffmanand P. J. Denning, Operating Systems Theory.Englewood Cliffs, NJ: Prentice-Hall, 1973.

[9] M.D. Dahlinet al , "Cooperative Caching: Using Remote Client Memory to Improve File System Performance" , OSDI'94 ,1994.

[10] P. J. Denning, "Working sets past and present," IEEE Trans. Software Engineeing, vol. SE-6, no. 1, pp. 64–84, 1980.

[11] EMC Corp., Symmetrix 3000 and 5000 Enterprise Storage Systems, Product Description Guide,1999.

[12] Robert N. Hasbun et al , "Flash EEPROM main memory in a computer system" ,1997

US patent，Dec 1997.

[13] IBM, IBM Enterprise Storage Server, IBM Corp., 1999.

[14] Intel , http://www.intel.com/technology .

[15] Intel , What is Flash Memory? , http://www.intel.com/design/flash/articles/what.htm , 2003.

[16] S. Jiang and X. Zhang, "LIRS: An Effi cient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance", In Proceeding of 2002 ACM SIGMETRICS, pp. 31-42, June 2002.

[17] Song Jiang et al , "A locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance" , the 26th IEEE International Conference on Distributed Computing Systems (ICDCS'06) , 2006.

[18] Heeseung Jo et al , "FAB:Flash-Aware Buffer Management Policy for Portable Media Players" , 2006 IEEE Transactions on Consumer Electronics, April 2006.

[19] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm" , Proceedings of the 20th International Conference on VLDB, pp. 439-450, 1994.

[20] H. J. Kim and S. G. Lee, "An Effective Flash Memory Manager for Reliable Flash Memory Space Management, " IEICE Trans. Information & System, Vol.E85-D, No.6, pp.951-964, June 2002.

[21] Young-Jin Kim and Jihong Kim, "Device-aware cache replacement algorithm for heterogeneous mobile storage devices", 2007.

[22] D. Lee et al, "LRFU:A Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies ", IEEE Trans. Computers, vol. 50, no. 12, pp. 1352–1360,2001.

[23] F. Matias, Cuenca-Acuna , T.Nguyen ,"Cooperative caching middleware for cluster-based servers" , Rutgers University , 2001 IEEE.

[24] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” IBM Sys. J., vol. 9, no. 2, pp. 78–117, 1970.

[25] N. Megiddo and D. Modha, “ARC: a Self-tuning, Low Overhead Replacement Cache”, Proceedings of the 2nd USENIX Symposium on File and Storage Technologies, March 2003.

[26] Microsoft , “Windows PC Accelerators” , November 2006.

[27] D. Muntz, P. Honeyman, “Multi-Level Caching in Distributed File Systems-or-Your Cache Ain’t Nuthin’ but Trash” , Usenix Winter 1992 Technical Conf., pp. 305-314, Jan 1991.

[28] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “An optimality proof of the LRU-K page replacement algorithm,” J. ACM, vol. 46, no. 1, pp. 92–112, 1999.

[29] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K Page Replacement Algorithm for Database Disk Buffering”, Proceedings of the 1993 ACM SIGMOD Conference , pp.297-306, 1993.

[30] M. Rosenblum and J. K. Ousterhout , “The design and implementation of a log-structured file system”, ACM Trans. Computer System,Vol.10, No.1,pp.26-52,1992.

[31] Samsung, http://www.samsung.com/Products/HardDiskDrive/index.htm .

[32] Sandisk, http://www.sandisk.com/SSD .

[33] P. Sarkar and J. Hartman , “Efficient cooperative caching using hint” , OSDI’96 , 1996.

[34] Seagate, http://www.seagate.com/ .

[35] Curtis E. Stevens, “AT Attachment 8 - ATA/ATAPI Command Set” , April 2007.

[36] T.M. Wong and J.Wilkes , “My cache or your cache? Making storage more exclusive”,USENIX’02 , 2002.

[37] David Woodhouse, JFFS : The Journalling Flash File System,2001.

[38] M. Wu, W. Zwaenepoel ,”eNVy:A non-volatile main memory storage system”, Proc. ASPL94 , pp.86-97,1994.

[39] Yuanyuan Zhou, “Second-level buffer cache management” , 2004 IEEE Computer Society ,

2004.