行政院國家科學委員會專題研究計畫 成果報告

嵌入式網路通訊裝置評比技術與工具之研發--子計畫四:嵌入式網路通訊裝置儲存裝置效能評比基準與工具之研發(中心分項)(2/2)

研究成果報告(完整版)

計 畫 類 別 ： 整合型
計 畫 編 號 ： NSC 99-2220-E-009-047-
執 行 期 間 ： 99 年 08 月 01 日至 100 年 07 月 31 日
執 行 單 位 ： 國立交通大學資訊工程學系（所）

計 畫 主 持 人 ： 張立平

計畫參與人員： 碩士班研究生-兼任助理人員：李盈節
碩士班研究生-兼任助理人員：吳翊誠
碩士班研究生-兼任助理人員：林玟蕙
碩士班研究生-兼任助理人員：王薇涵

報 告 附 件 ： 出席國際會議研究心得報告及發表論文

處 理 方 式 ： 本計畫可公開查詢

中 華 民 國 100 年 10 月 31 日

行政院國家科學委員會補助專題研究計畫 ■成果報告
□期中進度報告

嵌入式網路通訊裝置評比技術與工具之研發-子計畫四:嵌入式網路通訊裝置儲存裝置效能評比基準與工具之研發(中心分項)(2/2)

計畫類別：□個別型計畫 ■整合型計畫
計畫編號：NSC 99-2220-E-009-047-
執行期間：2010.08.01 至 2011.07.31

執行機構及系所：交通大學資工系

計畫主持人：張立平
共同主持人：
計畫參與人員：李盈節，吳翊誠，王薇涵，林玟蕙

成果報告類型(依經費核定清單規定繳交)：□精簡報告 ■完整報告

本計畫除繳交成果報告外，另須繳交以下出國心得報告：
□赴國外出差或研習心得報告
□赴大陸地區出差或研習心得報告
■出席國際學術會議心得報告
□國際合作研究計畫國外研究報告

處理方式：除列管計畫及下列情形者外，得立即公開查詢
□涉及專利或其他智慧財產權，□一年□二年後可公開查詢

中 華 民 國 100 年 10 月 30 日

國科會補助專題研究計畫成果報告自評表

請就研究內容與原計畫相符程度、達成預期目標情況、研究成果之學術或應用價值（簡要敘述成果所代表之意義、價值、影響或進一步發展之可能性）、是否適合在學術期刊發表或申請專利、主要發現或其他有關價值等，作一綜合評估。

1. 請就研究內容與原計畫相符程度、達成預期目標情況作一綜合評估

□ ■達成目標

□ 未達成目標（請說明，以 100 字為限）

□ 實驗失敗

□ 因故實驗中斷

□ 其他原因

說明：

2. 研究成果在學術期刊發表或申請專利等情形：

論文：■已發表 □未發表之文稿■撰寫中 □無

專利：□已獲得 □申請中 □無

技轉：□已技轉 □洽談中 □無

其他：（以 100 字為限）

已經發表會議論文兩篇（IWSSPS 2010, CPSNA 2011，如附件），並被邀請投稿至 IEEE embedded systems letter 的一個 special issue

3. 請依學術成就、技術創新、社會影響等方面，評估研究成果之學術或應用價值（簡要敘述成果所代表之意義、價值、影響或進一步發展之可能性）（以 500 字為限）

本計劃成果為固態硬碟的虛擬平台。原則上我們透過產學合作的管道推廣至業界使用，目前廠商的回應都相當不錯。而學術研究方面，基於這個虛擬平台，我們目前得以研究開發新的儲存裝置與主機端的溝通方式，藉以達成更好的效能改善。

Design and Implementation of a Virtual Platform for Solid-State Disks

摘要

本研究提出一個針對固態硬碟之虛擬平台,提供線上的即時模擬環境。該虛擬平台包含兩部分：一個模擬引擎以及一個虛擬磁碟。模擬引擎部分可以對固態硬碟內部的硬體架構與韌體演算法作快速制訂，並進行行為層次的模擬。該模擬引擎具有高度可重組性以及使用上的簡便性。而虛擬磁碟部分則以一個一般的磁碟出現在主機中，並且直接接收來自於主機的讀寫動作，就好像是一個真正的磁碟一般。此虛擬磁碟與模擬引擎之間透過作業系統內部的事件機制互動，計算並模擬讀寫延遲，使得該虛擬平台的效能就像一個真正的固態硬碟一樣。這個虛擬平台不但能夠幫助固態硬碟設計者快速地定下韌硬體的組態，縮短開發測試時間，亦提供研究者一個絕佳的互動式環境，藉以開發主機端與儲存裝置的協同式效能最佳化方法。

關鍵字: 固態硬碟, 快速雛形化, 效能模擬

Abstract

This work presents a virtual platform for solid-state disks (SSDs). This virtual platform consists of a simulation engine and a virtual disk. The simulation engine provides behavioral simulation of hardware architectures and firmware algorithms. SSD designers can use the simulation engine for fast prototyping. The virtual disk appears as a normal disk drive in the host, and accepts read/write requests as if it was a real disk drive. The virtual disk and the simulation engine are integrated into the host operating system and they interact with each other via event-signaling mechanism. Users can have live performance experience when using the virtual platform. The benefits of this virtual platform are twofold: First, the virtual platform is useful to fast prototyping and speeding up the design-and-test cycles. Second, this virtual platform can be useful to researches focusing on cross-layer (i.e., between the host and the storage device) performance optimization techniques.

Keywords: solid-state disk, fast prototyping, simulation

I. INTRODUCTION

近年來行動電腦的儲存裝置由傳統硬碟(Hard Disk Drive, HDD)逐漸被以快閃記憶體為基礎的固態硬碟(solid-state disks, SSDs)所取代。由於 SSD 複雜的硬體架構以及韌體的演算法，使得如何設計高效能的 SSD 成為一項艱鉅的任務。廠商面臨一個實際的問題，在不同的環境或用途下，要如何組合硬體與韌體的設計才能達到最佳的效能。目前有一些離線模擬的工具[1][2][3][4]可以用來測試硬體與韌體的組合。

由於現有的離線模擬工具不易使用，使得 SSD 的研發和測試週期相當耗時，因此廠商強烈要求降低修改和測試週期的時間。另一方面，離線模擬工具有個問題是從 HDDs 收集 workload 的存取紀錄(trace file)時，I/O request 的反應時間會受限於底層的儲存設備，假若從一個慢速設備收集 trace，那麼 I/O request 時間將會增加。因此使用 HDDs 收集 trace 無法完全展現真實 SSD 的 I/O 反應。

本研究提出一種線上 SSD 模擬環境且提供一個快速的硬體-韌體之原型工具為 SSD 設計之用，該模擬環境具有簡單的 programming 介面並有豐富的硬體/韌體設計的選擇。整體而言，該模擬環境包括 sim-engine 與 virtual drive 兩部分，sim-engine 計算 SSD 的 I/O 延遲，virtual drive 在主機端的作業系統創建一個虛擬磁碟，設計者可以透過一般的磁碟存取操作對此虛擬磁碟進行讀寫，virtual drive 會將這些 I/O request 送予 sim-engine， sim-engine 計算這些 request 需要多少 flash

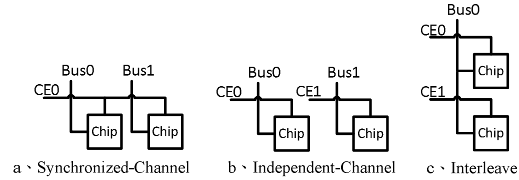


Fig. 1 SSD Inter-chip architecture

Fig. 2 Out-of-place updating data

operations 且花費多少時間，再讓 virtual drive 模擬出這些 I/O 延遲。該工具的目的在於降低除錯的成本，且不需要冗長的 trial-and-error 週期就能找出最佳的設計方案。

虛擬平台有幾項技術上的挑戰如下: 第一，sim-engine 如何提供一個簡單又具共通標準制定的SSD硬體/韌體的抽象方法，讓設計者可以簡單地改變 SSD 的設計。第二，虛擬平台如何與作業系統結合互動才能實現虛擬磁碟的功能。第三，sim-engine 如何準確地計算 I/O 的延遲，virtual drive 如何模擬這些 I/O 延遲。第四，如何利用有限的 RAM 創建出一個很大的虛擬磁碟。

II. HARDWARE/FIRMWARE ABSTRACTION

A. SSD 硬體架構

Figure 3 為 SSD 的硬體架構。Figure 4 (a)稱為"gang"，所有通道由同一條 chip enable line 連接，每個通道必

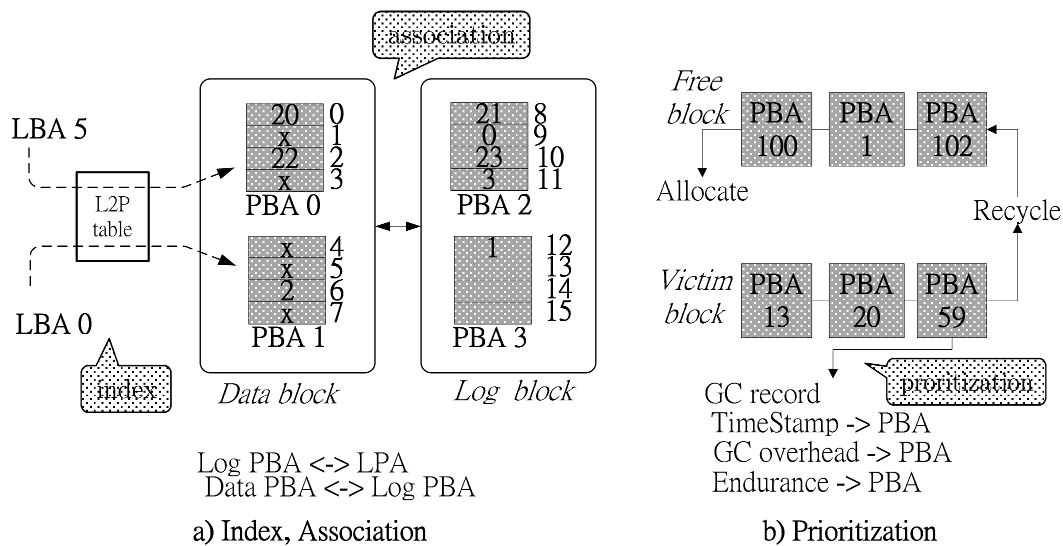


Fig. 7 Abstract Firmware Translation Layer

須做相同的讀/寫操作。若通道不是由同一條 chip enable line 所連接，如 Figure 5 (b)，每個通道可以獨立操作讀/寫。"interleave"類似計算機架構中管線的概念，同一通道中的 chips 可以做不同的讀/寫命令，如 Figure 6 (c)。

在我們的虛擬平台中，我們使用時序引擎(timing engine)來模擬平行的硬體操作，當一個操作完成時會通知其他的模擬模組，換句話說，平行的硬體操作只會計算一次的時間。

B.　韌體演算法

快閃記憶體的最小寫入單位為一個 page，且具有重複寫入相同 page 前，必須要做 erase 的特性，而一個 erase 的單位為一個 block，基於效能的考量，SSD 使用 out-of-place 的資料更新方式(如 Figure 2)，該方法須使用對照表(mapping table)紀錄資料位址的資訊且利用 garbage collection(gc)機制以回收 block。FTLs(Flash Translation Layer)負責 SSDs 的 mapping 與 gc。

在我們的虛擬平台中，我們設計了一套韌體演算法的 APIs，並定義了 FTLs 共用的三種抽象化元素，如 Figure 3。這裡說明了我們如何建立 FTLs 使用它們的行為之模型。

這是 NK[6] FTL，如 Figure 3(a)所示，FTL 利用 Index 處理位址映射，所以我們不只記錄邏輯位址區塊(LBA)與實體區塊位址間(PBA)的關係，還要記錄邏輯分頁位址(LPA)與實體分頁位址間(PPA)的關係。Association 則用來表示 FTLs 之資料集合間的關係，舉例來說，多少個 data block 對映到 log block。Figure 3(b)表示，在 GC 時是以 Prioritization 來選擇出犧牲者。

C.　硬體組態配置範例

我們訂定硬體環境: 4 independent channel, 1 bank, 1 plane, 1 interleave level。而 Flash Chip 的特性如下所示:

Algorithm 1 FW API: BAST FTL

```
    for handle a page do
2:      oldWritePPA ← QueryIndex(pageindex);
        if operation ¡ one page size then
4:          /*handle read modify write*/
        end if
6:      while Write(1 page) < 0 do
            if (have no Current log block then
8:              DoGetOneFreeLogBlock();
                ModifyIndex(pageindex, blockindex);
10:         else
                DoGarbageCollection();
12:             ModifyIndex(pageindex, blockindex);
            end if
14:         AccessBlock ← QueryBlock(blockindex);
        end while
16:     ModifyIndex(pageindex, blockindex);
        AssignGroupUnit(pageindex, blockindex);
18:     if have no free log blocks then
            DoGarbageCollection();
20:     end if
    end for
```

```
NUMBER OF GANG = 1;
CHANNEL PER GANG = 4;
CHIP PER CHANNEL = 1;
PLANE PER CHIP = 1;
hwAPI->SetupFlashChip(Chip
Character);
```

關於韌體的部分，如 Algorithm 1 所示，韌體 API 可以做到: 1) 若寫入量小於 1 page 且此 page 之前已經寫過，則我們執行 read modify write。2) 將此 page 寫到 log block 並透過 API 處理 GC 或是取得新 log block。3) 修改 index，將邏輯分頁位址與實體分頁位址綁在一起。4) 將邏輯分頁位址與其 log block 記下來(association)。5) 若沒有 free space 則執行 GC。

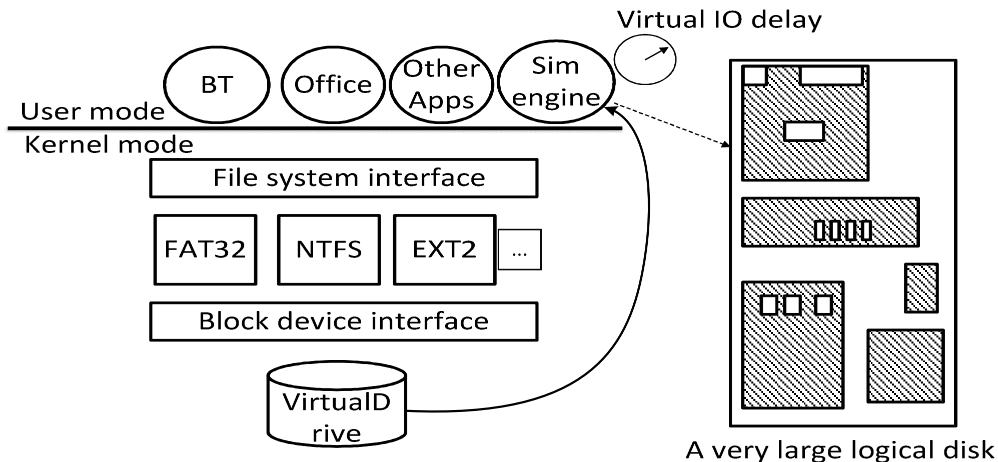


Fig. 8 Virtual Platform: on-line simulation environment

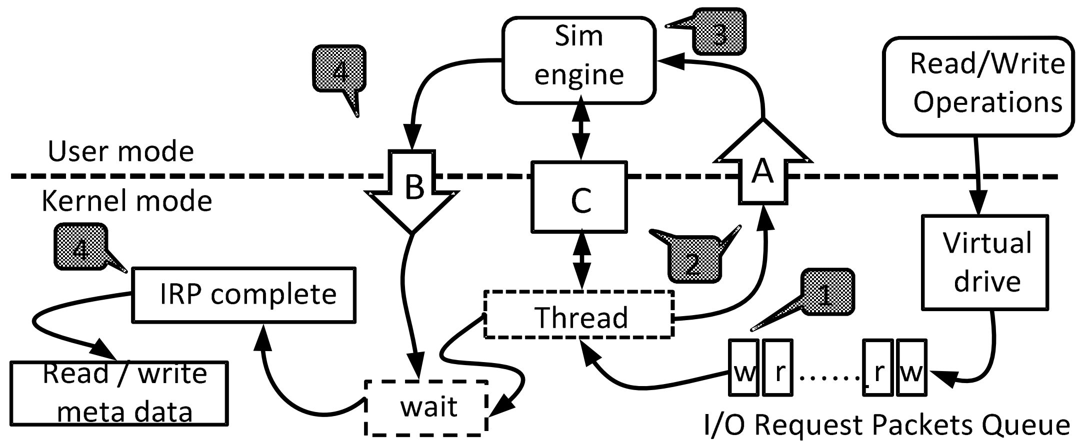


Fig. 9 Sync event flow between kernel mode and user mode

和真實 SDD 開發平台的 BAST FTL 程式碼比較，用我們的韌體 API 可以減少超過 75%的程式碼行數。

III. 虛擬磁碟(virtual drive):線上模擬

我們提出一種線上模擬的構想，如 Figure 4 所示。這是一個作業系統核心模式下的虛擬磁碟，設計者可以建立並控制一個虛擬磁碟如同一個真實硬碟，不像使用者模式的檔案系統[8]只處理使用者資料，在虛擬磁碟上，虛擬平台會產生硬體/韌體結合的 I/O 延遲，設計者可以在任何時間測試並使用虛擬磁碟，用這個方法設計硬體/韌體整合將更加直接且靈敏，可以減少修改和測試的時間。

如 Figure 4 所示，這裡有一些議題: 1) 作業系統的相互影響 2) metadata 的識別，以及 3) I/O 延遲的計算。我們將會在以下的部分討論這些議題。

A. Host 作業系統相互影響

之前提到，在我們虛擬平台上，我們設計了硬體/韌體抽象化 API。為了讓設計 SSDs 硬體/韌體變簡單，這個硬體/韌體抽象化 API 必須保留在作業系統的使用者模式中。

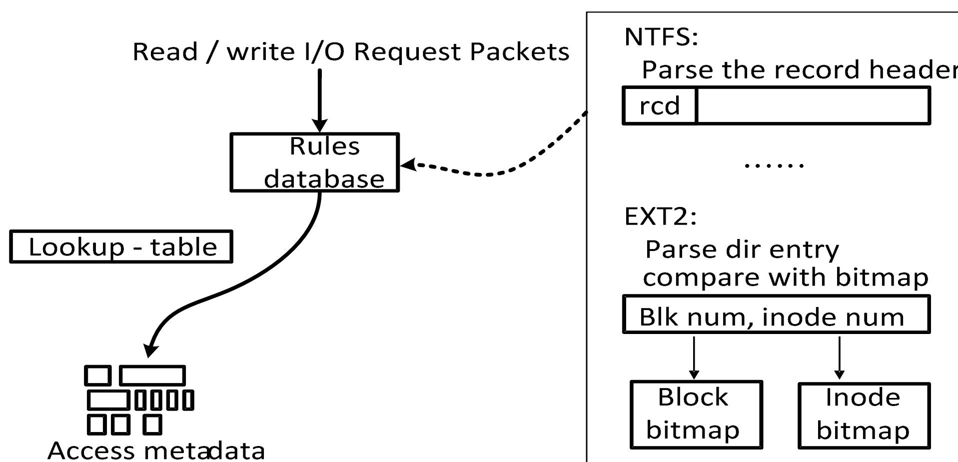


Fig. 10 Rules database of metadata conception

所以我們必須將作業系統的核心模式與使用者模式做同步化動作。如Figure 5 所示，有兩個共用物件 A 和 B 共用了記憶體C，sim-engine 設成"wait"狀態並且等待 A。接下來我們將會用 item 5-1 來解釋 Figure 5 中第一項 item。

虛擬磁碟將會接收從應用程式送出的 I/O request packets(IRPs)，並將它們放在一個 Queue 中，如 item 5-1 所示，接著用執行緒處理這個 Queue。在使用者模式中，執行緒將模擬請求的資訊給 C 並且設置 A 來通知 sim-engine，如 item 5-2 所示，然後它將會設置"wait"狀態。當 sim-engine 被 A 通知，如 item 5-3 所示，它將會啟動並從 C 獲得資訊，並且開始模擬，我們將會計算模擬所用去的時間以及作業系統模式轉換的開銷，當模擬結束，sim-engine 將會設置 B，如 item 5-4 所示，而執行緒將會從記憶體 C 讀取延遲資訊並產生虛擬 I/O 延遲及完成 IRP，如 item 5-5 所示，然後繼續處理 metadata。我們將會在實際 SSD 平台實驗中驗證模擬的 I/O 延遲正確性。關於作業系統執行緒轉換的開銷，我們將在第 IV.章節中解釋。

另一方面，sim-engine 也許會實施排程機制，out-of-order 完成請求。

B. Metadata 之辨識

為了利用大小有限的 RAM 去模擬一個大容量的 SSD，我們提出了一種定義 metadata 的構想。

Metadata 即為用來詮釋資料的一種資料，又可以稱為資料的索引，metadata 只佔所有 data 的一小部分。Disk 中即使只存 metadata ，檔案系統也可以正常運作，並且讓 disk benchmark tools 不要去驗證寫到 disk 上的 data，因此 benchmark tools 可以正常運作在只存有檔案系統 metadata 的虛擬磁碟上。

舉例來說，當我們格式化一個 250GB 的硬碟成 NTFS 的磁區 ，則這個硬碟上的 metadata 只佔了 74.46MB 的硬碟空間，因此我們藉由 metadata 之識別方法來降低 SSD 虛擬平台對 RAM 的使用量。Sivathanu[7]提出一個方法去定義"live data"，但這個方法專注在資料內容的定義，而非 metadata。

要定義 metadata，不同檔案系統有不同的架構，所以我們實作一套"rules database" 在我們的虛擬磁碟中，如 Figure 6 所示，該"rules database"包含許多的不同檔案系統 metadata 定義規則，藉由這個資料庫，我們可以找到並儲存 metadata 到不同的檔案系統上。我們將會在接下來的章節中，來討論 NTFS 與 ext2 中定義 metadata 的方法。

在 NTFS 檔案系統的環境中，最主要的 metadata 都存在 MFT(Master Files table)中。首先，在 disk 的開機磁區中，我們可以知道 MFT 存放的位置，幸運地，每個 MFT 的項目為一筆紀錄的開頭，我們可以透過解析虛擬磁碟上的資料內容來定義這些紀錄，因此我們可以儲存這些紀錄來維持 NTFS 的正常運作，如 Figure 6 所示。

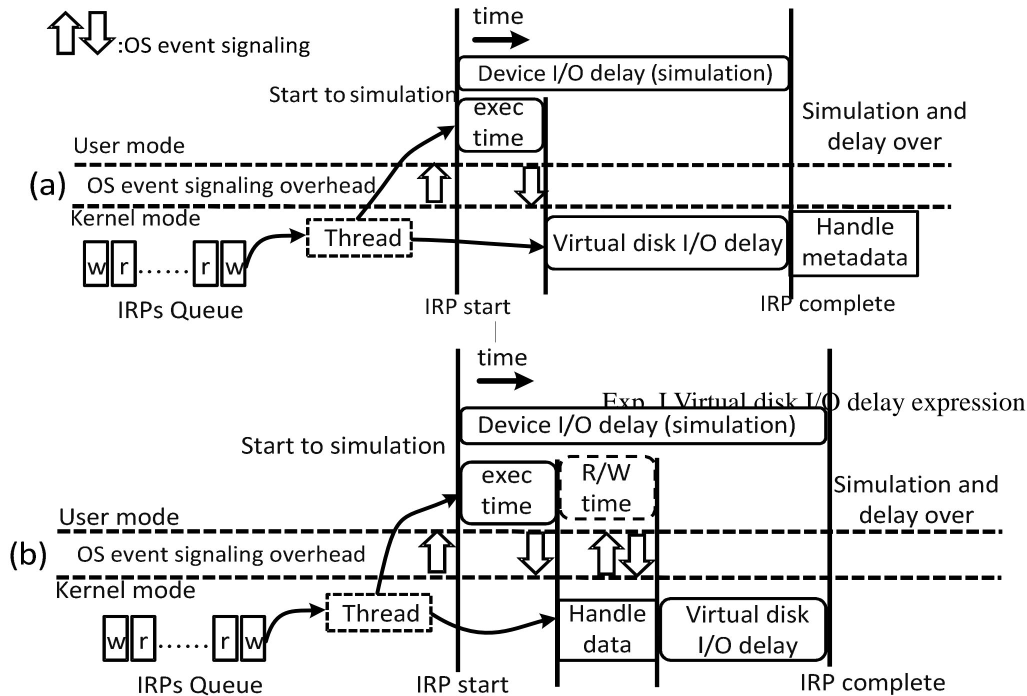


Fig. 11 IO delay simulation

之前的研究[9]中只有辨識 EXT2 檔案系統上固定位置的 metadata，所以無法辨認出 EXT2 中 block group 的 "directory i-node"，因為 EXT2 的目錄不是存在固定的位置。我們的 EXT2 metadata 定義方式可以辨認出 EXT2 的目錄，如 Figure 6 所示。首先，我們可以解析 i-node 資料內容，接著比較 i-node 上的 i-node number 與 bitmap，若 i-node 為一個目錄型的 i-node，則儲存該 metadata 到記憶體中。

IV. I/O 延遲之模擬

虛擬平台有兩個 I/O 延遲模擬的問題：硬體的時間花費，與作業系統的時間消耗。I/O 延遲模擬有兩個模式：一種是只儲存 metadata，另一種則是儲存 metadata 與實際資料。如果我們不考慮作業系統花費的時間，則當 sim-engine 產生一個裝置 I/O 延遲，減去模擬器執行的時間，與 kernel mode 的延遲時間，則為在虛擬磁碟上的 I/O 延遲，如圖 7(a)所示。虛擬平台可以儲存實際的資料，如圖 7(b)所示，裝置 I/O 延遲必須扣除 sim-engine 執行時間與處理資料時間。因磁碟的搜尋時間會破壞虛擬 I/O 的延遲準確度，我們可以用 ram disk 來解決。

由於虛擬平台使用事件訊號去同步 sim-engine 與 virtual drive，所以會產生一些 kernel mode 與 user mode 訊號傳遞的 overhead。另一方面，user mode 的 process 會被排程，排程器將觸發 user process 的 context switch，也會影響虛擬平台模擬的準確度。

為了最小化作業系統模式 switch 的時間消耗與使用者程序的 context

$$\text{OS event signaling overhead} = \text{switch count} \times t_{event}$$

$$\text{sim execution time} = t_{ex}$$

$$t_{dviodelay} = \max \begin{cases} t_{busdly0} + t_{chipbsy} \\ \ldots \\ t_{busdlyN} + t_{chipbsy} \end{cases}$$

$$\text{Virtual disk I/O delay} = t_{dviodelay} - (t_{ex} + t_{event})$$

switch，我們使用一些方法去解決這個問題。首先，我們得到處理器(processor)的時間戳記(TSC)去計算用來發送事件訊號的 CPU cycle time，換句話說，我們使用一個時間校準階段去計算事件訊號的時間消耗。第二，我們加入的虛擬平台 I/O 延遲是在作業系統核心模式，去避免使用者程序的競爭。第三，我們讓 sim-engine 執行在高優先權下，避免虛擬平台被系統的 context switch 所影響。

在計算虛擬 I/O 延遲上符號的意義與方法，如 TABLE I 所示。

T_{event}與T_{ex}皆由TSC計算出來的，$T_{dviodelay}$為多通道架構環境下處理 request 所花的時間。

V. 實驗結果

在這一節，我們有兩個實驗部分。第一個是驗證虛擬平台的準確性，我們會與真正的 SSD(GP5086)平台來跟虛擬平台的模擬結果比較。第二個是

Symbol	Denotation
t_{event}	OS event signaling overhead
t_{ex}	sim-engine execution time
t_{busdly}	bus delay time
$t_{chipbsy}$	flash chip busy time
$t_{dviodelay}$	device IO delay time (simulation)

TABLE I: I/O delay symbol table

在實際 workload 之操作下，虛擬平台展現的硬體/韌體搭配之下的效能。

我們使用業界最常使用的磁碟效能評比工具：IOmeter、ATTO 來驗證我們的虛擬平台與真實的 SSD (GP5086)，並且在虛擬平台安裝 Office 軟體，來測試兩種不同的硬體/韌體組合下的效能差異。

我們已經依照「硬體組態配置範例」章節中，將虛擬平台設定為跟真實平台(GP5086)相同的硬/韌體架構。如同 TABLE II 所示，我們可以觀察到虛擬平台的 I/O 延遲誤差低於百分之五，其誤差的原因來自快閃記憶體晶片的寫入/抹除時間會隨著溫度及電壓的變化而改變。為了要測試我們處理事件通知的時間消耗以及使用者程序間的行程切換的方法，我們使用了 FFT-z 這套工具來增進 CPU 使用率，測試我們的虛擬平台在 CPU 高壓力下的效能，如同 TABLE III 所示。因為行程切換的開銷影響不大，並且 I/O 延遲是在作業系統核心模式，故可以降

Benchmark	GP5080	Metadata	Realdata
IOMeter IOPS	6.47	6.37	6.12
IOMeter IO RespTime(ms)	154.3	155.9	162.6
ATTO 512K SeqWrt(byte)	15070	14519	15209
ATTO 512K SeqRd(byte)	33372	33522	31602
ATTO 8M SeqWrt(byte)	15007	15803	15796
ATTO 8M SeqRd(byte)	33346	33904	32646

TABLE II: Compare a real SSD (GP5086) results with our virtual platform

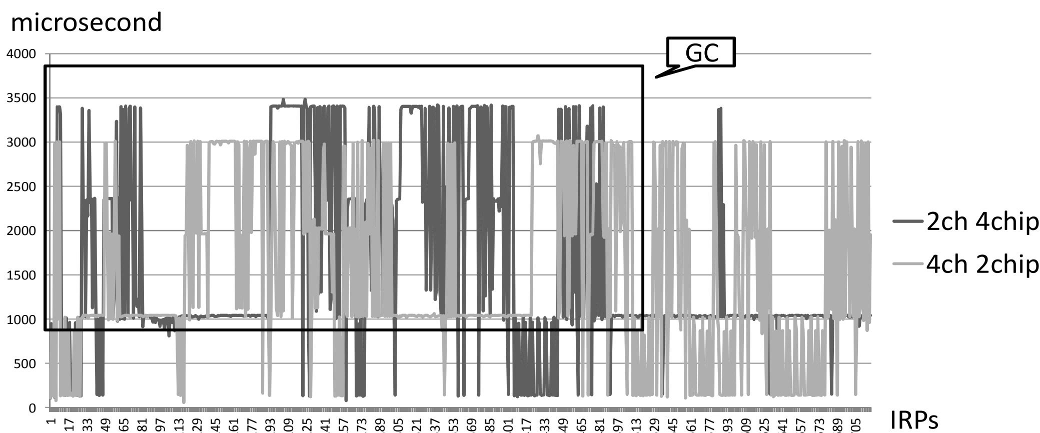


Fig. 12 Installing Office using two different channel architectures of SSDs (hardware)

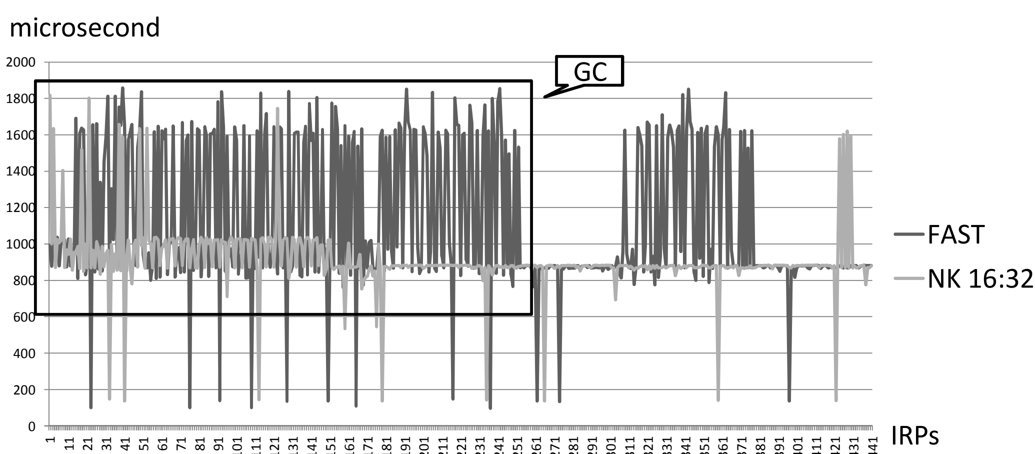


Fig. 13 Installing Office using two different FTL algorithms (firmware)

低使用者程序排程造成的影響，我們的虛擬平台在高度壓力的環境下可以維持虛擬 I/O 延遲的準確性。

為了要比較在兩種不同 SSD 設計下安裝 Office 的效能，我們設定了以下硬體配置組態：32GB 容量，256MB 備用空間。首先來看 Figure 8 所示的兩種不同設計的硬體架構。我們定義了 Flash 晶片數量為八個，並且將通道數量設定為 2 及 4。我們可以發現如果通道數量越多，資料處理可以愈平行地進行，因此在 GC 時有較低的回應時間；但愈多的通道將會分割備用空間，其將導致頻繁的 GC。接著我們比較兩種不同的韌體設計：NK 16:32 以及 FAST，如 Figure 9 所示。FAST 不會限制 log block 的關聯度，因此在 GC 的時候，FAST 的回應時間會高於 NK，這意味著在這段時間，FAST 將會使 SSD 產生較明顯之延遲現象。

VI. 結論

Benchmark	Normal	High stress
IOMeter IOPS	6.37	6.24
IOMeter IO RespTime(ms)	155.92	160.20
ATTO 512K SeqWrt(byte)	14519	14869
ATTO 512K SeqRd(byte)	33522	32483
ATTO 8M SeqWrt(byte)	15803	15779
ATTO 8M SeqRd(byte)	33904	33813

TABLE III: Experiments with/without CPU stress

本研究提出一個針對固態硬碟(SSD)的虛擬平台，並且在 user mode 中設計了一個抽象化的硬體/韌體介面以方便設計 SSD。這個虛擬平台可以做快速的"測試並修改"設計循環並於線上模擬。該虛擬平台可以只儲存 metadata，並在有限的記憶體下，建立大容量的 SSD。在實驗中，我們驗證了時間準確性之誤差相較於真實產品是低於百分之五。此外，我們也比較了兩種不同SSD的硬體/韌體設計來安裝 Office 的效能。

Reference

[1]. N. Agrawal, V. Prabhakaran, T. Wobber, J.D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[2]. J.S. Bucy, J. Schindler, S.W. Schlosser, G.R. Ganger, et al. The DiskSim simulation environment version 4.0 reference manual. Technical report, Technical report cmu-pdl-08-101, carnegie mellon university, 2008.

[3]. Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. Flashsim: A simulator for nand flash-based solid-state drives. In *Proceedings of the 2009 First International Conference on Advances in System Simulation*, pages 125–131, Washington, DC, USA, 2009. IEEE Computer Society.

[4]. Jongmin Lee, Eujoon Byun, Hanmook Park, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Cps-sim: configurable and accurate clock precision solid state drive simulator. In *SAC'09*, pages 318–325, 2009.

[5]. Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*,

[6]. Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7:38:1–38:23, August 2008.

[7]. Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 26–26, Berkeley, CA, USA, 2004. USENIX Association.

[8]. Jun Wang, Rui Min, Yingwu Zhu, and Yiming Hu. Ucfs-a novel userspace, high performance, customized file system for web proxy servers. *IEEE Trans. Comput.*, 51:1056–1073, September 2002.

[9]. Po-Liang Wu, Yuan-Hao Chang, and Tei-Wei Kuo. A file-system-aware ftl design for flash-memory storage systems. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 393 –398, april 2009.

附件（已發表之會議論文）：

1. Ming-Yi Yang, Li-Pin Chang, and Ya-Shu Chen, "Workload-Oriented Benchmarks for Solid-State Disks," International Workshop on Software Support for Portable Storage (IWSSPS), 2009.
2. Chun-Chieh Kuo, Jen-Wei Hsieh, and Li-Pin Chang, "Detecting Solid-State Disk Geometry for Write Pattern Optimization," The International Workshop on Cyber-Physical Systems, Networks, and Applications (CPSNA), 2011

Workload-Oriented Benchmarks for Solid-State Disks

Ming-Yi Yang
Department of Computer Science
National Chiao-Tung University
Hsin-Chu, Taiwan

Li-Pin Chang
Department of Computer Science
National Chiao-Tung University
Hsin-Chu, Taiwan
lpchang@cs.nctu.edu.tw

Ya-Shu Chen
Department of Electrical Engineering
National Taiwan University of
Science and Technology
yschen@ntust.edu.tw

Abstract

A solid-state drive (SSD) uses flash memory as storage media. In the recent years, due to the SSD's ability to conserve power, and to endure shock and vibration, as well as its random access capability, it has started to take the place of the traditional hard drive. However, users' experiences usually do not match the performance claimed by the manufacturers for the SSD. The main reason for this is that most tools used to evaluate the performance of the SSD are the same as those used to test traditional hard drives. The performance cost of the internal management mechanism in SSD is not taken into account by the test methods, so that the apparent results do not represent the true performance of the SSD. This paper proposes a method to test the management efficiency of the SSD based on the disk workload of a real system. The proposed method is able to differentiate the access patterns of an SSD, categorize real workloads into four sets of benchmark suites, and then identify SSD performance bottlenecks.

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Garbage collection; B.3.2 [Memory Structures]: Mass Storage.

General Terms

Design, Performance.

Keywords

Solid-State Disks, Flash Memory, Benchmark.

1. Introduction

NAND flash memory is known for its small dimension, ability to endure vibration and conserve power, and its fast random access capability. SSDs composed of NAND flash memory are already commonly used in personal computers. Unlike traditional hard drives, managing data in SSD is subject to physical constraints of NAND flash memory, such as uneven physical units of read/write and erase, address translation, free-space reclaiming and wear leveling. SSDs use a flash translation layer to simulation block device interface and to hide NAND flash memory's physical characteristics. Manufacturers implement different management strategies in the hardware controller to process SSD management issues and handle performance problems caused by large amounts of data reading/writing.

Apart from the hardware specification, the performance of the SSD is also affected by the management algorithm in the SSD firmware. The test results delivered by such tools do not help users to make a fair evaluation of the SSD. Most current hard drive benchmark tools target traditional mechanical hard drives. They focus on seek, rotate, data transfer and time overhead, which are absent from the management issues of NAND flash memory.

So far, only a few papers have discussed the performance of the SSD and test methods. [1] studied the effects of a variety of design methods on the SSD performance, using by software simulation. However, the test was not conducted on a real product. [2] developed a tool to test SSD resource access patterns, but empirical test results were not reported. [3] defined a complete set of SSD performance evaluation methods including pre-test configuration, post-test configuration and a battery of access pattern tests. The focus of that work was the methodology of performance evaluation, but it failed to identify the reasons for the low performance of SSD management strategies.

The two problems with current test methods are: firstly, the performance metrics of traditional hard drives cannot identify the reasons that cause the SSD performance differential; secondly, tested access patterns do not cover the access patterns actually used by customers. To respond to these difficulties, we propose a new performance metric, Per-Byte-Response. The metric represents the response time of every kilobyte in a single read/write request. The metric emphasizes the overheads imposed by the SSD management activities on each individual request, neglecting the data transfer time. We analyze the spatial distribution and access time distribution of the data's Per-Byte-Response, summarize the typical symptoms of poor resource management, and provide the user with an account of the causes of poor performance. Secondly, we gather the user's real access patterns as a test workload, and conduct an analysis of the characteristic access patterns for different types of workloads. These workloads are categorized into four benchmark suites, which provide users with evaluations of transfer speed, address translation, free-space reclaiming and buffer management. The user can choose a suitable Benchmark suite for the specific SSD or SSD management issue which needs to be addressed, obtain performance metrics and find out what factors are adversely affecting performance.

2. SSD Management

The overall performance of the SSD depends on the hardware architecture and the data management schemes. The hardware architecture includes parallel transmission architecture (Multi-channel or Inter-leaving), controller, types of NAND flash memory, and buffer configurations.

Currently, there are two types of NAND flash memory, SLC and MLC. In order to increase the capacity of the device, many SSDs use MLC as the storage media. However, MLC has a longer read/write time than SLC. Therefore, MLC use has a significant impact on the performance and lifetime of the device.

To improve read/write performance, SSD employs additional RAM as write buffer or read cache. The current SSD buffer management can be categorized into: (1) traditional management schemes, such as FIFO, LRU; (2) new management schemes designed for NAND flash physical characteristics, such as FAB, BPLRU[4]. The former only utilizes the hardware advantage of RAM to shorten access time, while the latter optimize the management scheme costs as well.

The major management issues of the SSD are: address mapping, free-space reclaiming and wear leveling. As the unit of SSD read and write is a page, while the unit of erase is a block, it is necessary to use out-place updates to avoid frequent erasure operations. An address mapping mechanism is needed to translate logical addresses into physical addresses. Most of the current address mapping mechanisms divide the blocks into data blocks and log blocks. All the original data is stored in the data blocks. When each data update arrives, log blocks are used to hold the updated data. It is a design option that how data blocks are associated with log blocks. When a lot of free-space reclaiming actions are taken by a small amount of data written, we can conclude that the address mapping mechanism is not working well, and action needs to be taken [5].

When there is insufficient free-space for data writes, the SSD needs to reclaim free-space by erasing invalid data. However, the minimum unit that can be erased is a block. Garbage collection will trigger a sequence of data moves and erases. The time cost of garbage collection is the major management cost. Generally speaking, garbage collection should be postponed as late as possible, and should erase the block with most invalid pages. When the cold data (rarely updated data) and hot data (frequently updated) are mixed in the same block, the efficiency of garbage collection will be significantly impaired. It is therefore better, where possible, to store hot and cold data in different blocks.

3. Performance Evaluation using Real Workloads

The SSD performance benchmarking proposed in this paper takes the form of a black-box test to evaluate external response time performance. The advantages of this method are easy test environment setup and simple parameters. The disadvantage is the difficulty in diagnosing the reasons for poor performance in a single test. This section will introduce the system configuration of the SSD benchmarking, performance metrics and typical symptoms of suboptimal management strategies.

Our SSD benchmarking method is composed of two steps. The first step is Trace-Collect. It operates in the driver layer of the file system, collecting users' access patterns to hard drives. The second step is called Trace-Replay. It is mainly used to reproduce the data access activities on the SSD that is going to be tested. Because only write requests involve SSD management activities, the benchmarking method proposed in this paper only concerns write requests in the collected traces.

To focus on the impact of the SSD management strategies on performance, we propose the **Per-Byte-Response** (PBR) as a performance metric to eliminate the time overheads contributed by data transfer. For each SSD write request, the PBR is defined by the following formula:

Response time (in seconds) / Request size (in bytes).

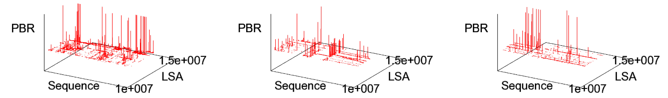


Figure (1) Suboptimal write buffer management **Figure (2) Suboptimal mapping scheme** **Figure (3) garbage collection scheme**

Under optimal management scheme conditions, the management overheads in SSD are kept as low as possible, and no PBRs of requests are noticeably large. However, with suboptimal management schemes, it is possible that a small request introduces lengthy management activities, suddenly increasing its PBR. . General users can not easily identify performance bottleneck of various SSD devices by observing PBR only. To assist users to diagnose the problem, it is necessary to analyze PBR results exhibited by typical symptoms of suboptimal management strategies.

To observe the time and spatial distribution of the PBR values, we used a 3D visualization method, where the X axis represents data reference numbers, the Y axis represents the original logical address and the Z axis represents the PBR value.As shown in Figure (1), there is a significant difference between PBR values. The Figure shows how the data is buffered to improve the performance before the data is written in SSD. When there is free space in the write buffer, the extremely low PBR values represent the time cost of writing data to RAM. When the write buffer is full, data will be written to the SSD, which results in a series of management activities and a significant increase in the PBR value. However, this buffer management scheme is not optimized for the flash translation layer, instead of delivering stable PBR values, the PBR values changes severely. This phenomenon is defined as "suboptimal write buffer management".

As shown in Figure (2), when the PBR values increase dramatically and are randomly distributed over a large range of logical addresses, garbage collection is triggered at a high frequency, even through the total amount of data written is low. The reason is that the suboptimal address mapping scheme results in low utilization rate of the SSD space (i.e., log-blcok threshing). This phenomenon is defined as "suboptimal address mapping".

In Figure (3), the PBR values increase dramatically but only appear densely in a small area and in a short time. This means that certain data a updated frequently, triggering garbage collection. There are two reasons for this phenomenon: (1) extremely high PBR values representing large data relocation cost during garbage collection. (2) high incidence of PBR values, indicating a problematic garbage collection strategy, such as premature garbage collection or improper selection of recycling victims. This phenomenon is defined as "suboptimal garbage collection scheme".

4. Workload Characterization and Benchmark Suites

In this section, we will discuss how to conduct temporal and spatial analysis on the collected workloads, list important characteristics, analyze the relationship between these characteristics and SSD management schemes, and categorize

these workloads into suites of evaluating transfer speed, address mapping, garbage collection and buffer management.

To completely reproduce the data access activities on the SSD to be tested, three parameters need to be recorded for each data item during the trace-collect stage: data write sequence, start address and length of transfer. Each data item is either a write or a rewrite. When the data is written into blank regions, it is known as a write. When the data is written into regions that contain addresses have already been written to, it is called a rewrite. Workloads with large amounts of rewrite requests have high time locality and space locality of data read and write.

According to the start address of each data item, we can further categorize the data into two types: sequential data and random access data. As modern operating systems support multi-tasking environment, sequential data writes or rewrites can be interrupted by write requests from other processes. We determine the sequence of read/write actions by assigning an error value K . Then, the data is defined as sequential data if the write address of the Nth data item and the start address of the $N+K^{th}$ data item are contiguous, where N is an integer. If the data is non-sequential, it is deemed to be random access data. A workload with a high percentage of sequential data has high space availability, which means that it is easy to gather large amounts of invalid data space during garbage collection.

Most data write requests of the file system are for small writes, falling into the size range of 4KB. So request with transfers length smaller than 4KB are treated as small writes.

If a write request's starting address is unable to align with the page boundary of NAND flash-memory pages, then writing a page may require extra overheads of read-modify-write operations. This concerns the performance of the address mapping scheme. To account this, we record whether the data's start address and end address are aligned to the 4KB boundary in sector addresses, and further analyze the ratio of aligned data to the entire dataset.

According to the parameters collected in the trace-collect stage, we directly analyze the rewrite ratio, sequential ratio, alignment ratio and transfer length statistics. This analysis is called macroscopic analysis. For small-scale workloads with simple behaviors, these four characteristics can be categorized and used to test the performance of the SSD management schemes. However, for large-scale workloads with complex behaviors further parameter calculations are required.

Microscopic analysis focuses on the time distribution and space distribution of the access pattern, and understands the formation of its characteristics. To analyze the spatial locality of an access pattern, we derive the traditional performance metric "Seek Distance" to calculate the distance between the end address of the current data item and the state address of the next data item. Even though SSD does not suffer from the cost of the read/write head movements, this metric represents the randomness of the data access. If the variation of seek distances is very large, then the access pattern exhibits random access. This can be used to test the performance of the address mapping scheme.

The temporal locality of an access pattern represents the data rewrite frequency. There are two kinds of data, hot data and cold data, where the temperature of a piece of data is proportional to the frequency that the data is updated. However, for large-scale workloads, hot data are accessed by bursts of variable lengths. Therefore, the time window of accessing hot data must be considered. We define life span and life cycle as follows:

Definition 1：Life Span

Let X be some Logical Sector Address (LSA). Let FIRST_ACCESS(X) represent the request sequence number when X is written for the first time; let LAST_ACCESS(X) represent the request sequence number when X is written for the last time. Life span is defined as:

Life_Span(X) = LAST_ACCESS(X)-FIRST_ACCESS(X).

Definition 2：Life Cycle

Assume Write_Count(X) represents the number of times a LSA address, X, was written. Then

Life_Cycle(X) = Life_Span(x) / Write_Count(x). If Life_Cycle(X)=0, no rewrite has occurred in this address.

Using the life span and life cycle definitions, we are able to observe hotness/coldness differences in the logical address space and understand the mixing level of the hot data and cold data. When the cold data and hot data are separated correctly, the data relocation cost during garbage collection can be reduced significantly. The cold/hot data distribution can be used to test the performance of the garbage collection.

After profiling workloads using the above mentioned indexes, workloads can be match to four benchmark suits, Transfer, Buffer, Mapping and Garbage Collection. The transfer suite is used to evaluate hardware transfer cost. The other three suites are used to evaluate the performance cost of the SSD management schemes. The data mainly composed of sequential write requests can be used in the hardware transfer architecture. Because sequential write requests are less likely to introduce extra copy operations during free-space reclaiming, workloads with a large number of sequential writes are classified as Transfer Suite. When the rewrite ratio of the workload is high and the transfer amount is larger than the write buffer capacity, the write-back mechanism will be triggered. Therefore, workloads with a high rewrite ratio and a large data amount are classified as Buffer Suite. When the rewrite activities of the workloads are random and consist of small writes, or there is unaligned write activity, the space usage rate is low. Therefore, workloads with random data and low alignment ratio are classified as Mapping Suite. When the workload has intensive rewrite activities and the cold and hot data are highly mixed, the garbage collection will be triggered, and such workloads are classified as Garbage Collection Suite. In the next section, workload testing will be described. The characteristics will be analyzed and assigned to appropriate benchmarking suites for SSD performance testing.

5. Experimental Results

5.1 Environment Setup

This section introduces the experiment environment setup, SSD to be tested and collected workloads. The experiment platform is built on a personal computer equipped with Intel Core 2 Dual 1.87GHz,2GB DDR2 memory, and the Windows XP operating system. A total of five SSD devices are tested. In order of price, MTRON and Samsung are high-end products with extra RAM available for write buffering. OCZ is a mid-price product, and TRANSCEND is a low-end product. The device specifications are shown in Table (1).

Table (1) SSD device specifications

Manufacturer	Interface	Memory Unit	Capacity	Controller
MTRON	SATAII	SLC	32GB	MTRON
Samsung	SATAII	SLC	32GB	Samsung
TRANSCEND	SATAII	SLC	16GB	SMI
TRANSCEND	SATAII	MLC	32GB	SMI
OCZ	SATAII	MLC	64GB	JMICRON

We collected user file access patterns from personal computers. User applications are of four types: general application, internet application, operating system installation, and P2P application. The access pattern of each user scenario is given in the table (2).

The first stage, trace-collection, was conducted with Windows XP, using the Diskmon trace tool [6] to collect access patterns and store them in a 16GB independent NTFS hard drive. The second stage, trace-replay, was implemented by using the functions CreateFile() and WriteFile() in the Windows API. Firstly, we used CreateFile() to open the SSD in the device driver mode. Secondly, we used WriteFile() to execute synchronous write activities. Time data from the CPU clock cycle was read by the assembly language function RDTSC().

5.2 Benchmark Suites

To conduct categorization of the workload benchmarking and management mechanism evaluation, a macroscopic analysis was first applied to the four collected workloads that are rewrite ratio, sequential ratio, data length and alignment ratio. The value of each application and its ratio to the overall data transfer rate is given in Table (3). For example, if the update data amount is 100MB and the overall data transferred is 200MB, the rewrite ratio is 50%. The error of the sequential activity is set to 10, which means if the logical addresses of the Nth write and $N+10^{th}$ write are contiguous, the action is considered a sequential write.

Table (2) Workload of user scenarios

Workload	Scenarios
Copy	Copy 200 files from one directory to another
Browser	Use Internet Explorer 5.0 to browse internet for 3 hours
Install Linux	Install Fedora Linux Server 4,the file system is EXT3
eMule	Use eMule 0.48b to download 3 files for 3 hours

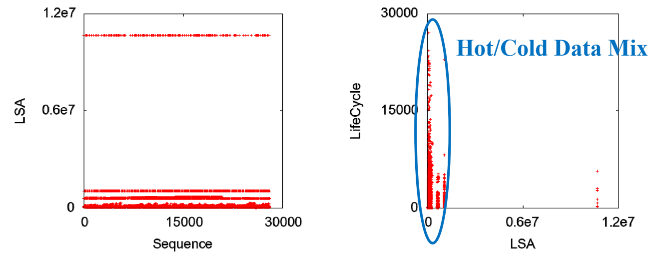
Copy has a low rewrite ratio, and the write request is 64KB sequential write, which can be categorized in the Transfer suite for hardware transfer speed test. Browser has a high rewrite ratio: these are most likely small writes with random access, which can be categorized to the Garbage Collection Suite or to the Mapping Suite. eMule and Install Linux are large-scale workloads. The metric intensities of macroscopic analysis are not very clear, therefore, we also use microscopic analysis to find out the characteristics of eMule and Install Linux, as well as the hot/cold data distribution of Browser.

Figure (4)-(a)(b) are the Browser data logical address distribution and hot/cold data distribution graph. The X axis in Fig. 4 (a) is the data sequence, and the Y axis is the logical sector address. As can be seen from the graph, the random access of the Browser is small and intense. This is because the browser temporarily stores website data on the hard drive to increase the website browsing speed. These temporary files are managed by Index.dat, which is frequently updated. The hot/cold data distribution is interpreted in the Life Cycle analysis. In Fig. 4 (b), the X axis is the logical sector address, and the Y axis is the life cycle (calculated by the definition 2 given in Section 4). As shown in the graph, the hot/cold data are highly mixed. The difference is not clear and hard to identify. Due to the intensity of the hot/cold data mix, we finally categorize Browser to the GC suite.

Table (3) Workload Macroscopic Analysis Results

Workload	Data Transfer	Rewrites Ratio	Sequ-ential Ratio	Data Length	Alig-nment Ratio
Copy	816MB	0.1%	96%	64KB	0%
Browser	477MB	80%	4%	4KB	31%
Install Linux	2387MB	18%	25%	4KB, 128KB	0%
eMule	9437MB	5%	55%	4KB, 512KB	0%

In the macroscopic analysis, the sequential ratio of Install Linux is only 20%. As shown in the LSA distribution graph, Figure (5)(a), its random access is scattered over a wide area. The access addresses are mostly located in group headers –group headers are where the metadata is stored in EXT2/EXT3 file systems. In EXT2/3 default settings, reading data also causes write actions to update the a-time in inode, which causes random writes to be much more common than sequential write. Figure (5)(b) shows the Seek Distance distribution for Install Linux. The X axis is the LSA, and the Y axis is the seek distance. As shown in the figure, the write action of Install Linux is scattered over a large area. Therefore, we categorize Install Linux to the mapping suite to test the address mapping s performance of the SSD.

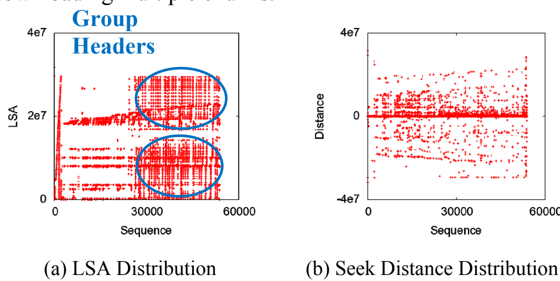


(a) Logical address Distribution (b) Life Cycle Distribution

Figure (4) Characteristics of the Browser workload

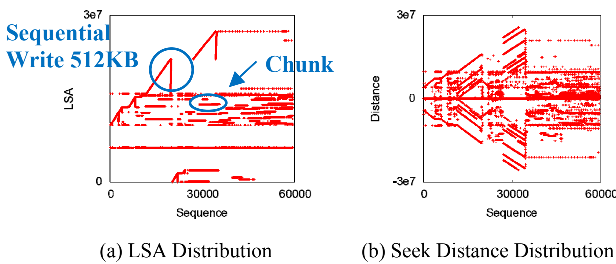
eMule is a popular P2P download software. Its principle is to cut files into several chunks and download multiple chunks simultaneously. A chunk is the minimum download unit, with size 9.28MB and buffer capacity of 128 KB. According to our macroscopic analysis results, 55% of rewrites are sequential and the data length is typically around 512KB. As shown in Figure (6)(a), the first half of eMule is a sequential write, which is caused by the data buffering before downloading each file; the second file is the random write of the chunk download. When a chunk starts to be downloaded, random access will be limited in the addresses of the corresponding chunk. Therefore, every chuck should have

high time locality and space locality. As shown in Figure 6(b), we focused the logic sector address inside a chunk, however, we found the seek distance was large and random. It is suspected that this is because multiple chunks are downloaded simultaneously. When we introduced the life cycle concept, we found the time locality and space locality were indeed concentrated on a small area. The pseudo randomness in a large area created by multiple chunk download may instantly consume the buffer set by eMule and cause frequent rewrites. Therefore, we categorize eMule to the Buffer Suite to test the write buffer management, as well as to determine whether the buffer management mechanism can handle downloading multiple chunks.



(a) LSA Distribution (b) Seek Distance Distribution

Figure (5) Characteristics of the Install Linux workload



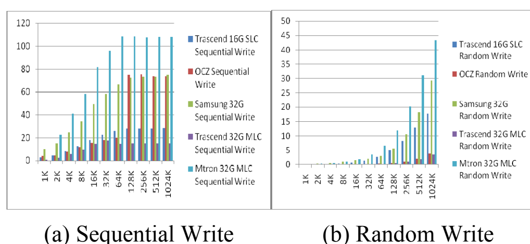
(a) LSA Distribution (b) Seek Distance Distribution

Figure (6) Characteristics of the eMule workload

Based on the macroscopic and microscopic analysis results, we conclude that the real workload and benchmark suite pairs as follows: Copy was matched to Transfer Suite, Browser is tied to GC Suite, Install Linux is categorized to Mapping Suite, and eMule is classified to the Buffer Suite.

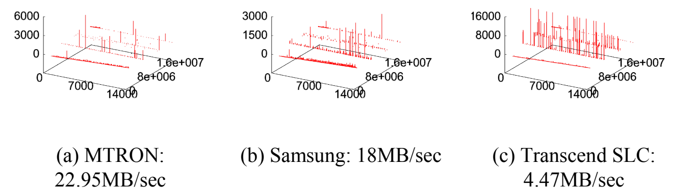
5.3 Benchmark Results

SSD benchmarking is composed of two parts. In Part I, traditional sequential and random pattern test results are used. In Part II, real workloads from the benchmark suites are used to test the SSD. Finally, the impact on SSD management performance is discussed by comparing results obtain in both parts.



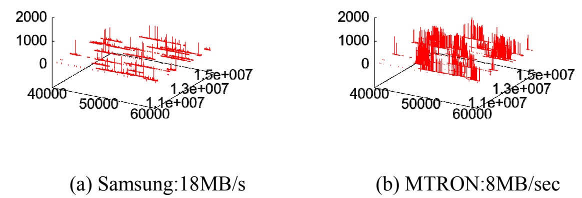
(a) Sequential Write (b) Random Write

Figure (7) IOMeter Sequential and Random Write Test :X axis is data length, Y axis is transfer speed in MB/sec



(a) MTRON: 22.95MB/sec (b) Samsung: 18MB/sec (c) Transcend SLC: 4.47MB/sec

Figure (8) Transfer Suite Test Results



(a) Samsung:18MB/s (b) MTRON:8MB/sec

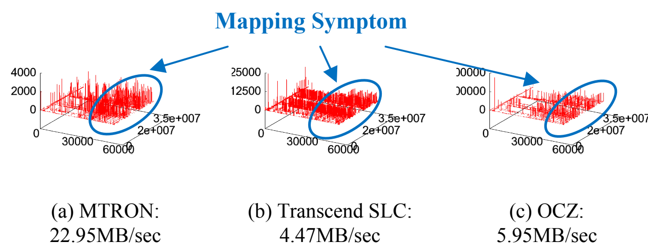
Figure (9) Buffer Suite Test Result

In Part I, the selected SSD devices are tested by the IOMeter sequential and random write patterns. In Figure (7)(a), MTRON has a faster transfer speed than any other competitors. Figure(7)(b) shows that SSD composed of SLC chipset outperform SSD composed of MLC chipset, which also indicates the performance gap of physical characteristics of SLC and MLC. Also, as shown in Figure (7)(b), MTRON and Samsung perform better than Transcend because of the extra write buffer. In addition, MTRON's performance is slightly better than that of Samsung, which shows that the write buffer of MTRON effectively handles the small writes. Given the results, we can conclude that using SLC chipset and extra write buffer will improve the overall performance of SSD.

IOMeter test results match with the product price range. MTRON has the best performance, while Transcend with MLC chipset comes last. Next, we use the benchmark suite to test these devices and use the proposed performance metric Per-Byte-Response (PBR), which is the ratio of response time to data transferred. The experiment results are shown in 3D graphs with X axis representing data sequence, Y axis representing LSA, and Z axis representing PBR. The higher the PBR values are, the higher the management cost is.

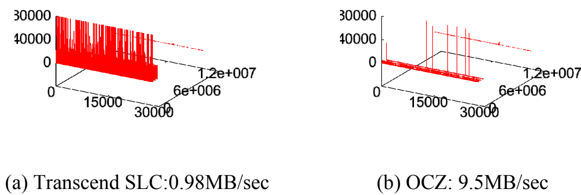
Figure (8) shows the Transfer Suite test results of MRTON and Samsung. Both the performance and the readings match the IOMeter test results. Next, we test the performance of the SSD management mechanism of each device. In Figure (9), we demonstrate the results of MTRON and Samsung by using eMule in the Buffer Suite. We found that the PBR values of MTRON fluctuate considerably. This is probably because the write buffer does not integrate with FTL (File Transfer Layer) design, and is not able to reduce the management cost effectively. The frequent rewrites caused by downloading chunks in eMule triggers address mapping or garbage collection, which reduces performance. According to the Samsung PBR values, the buffer management mechanism effectively reduced the management cost, which indicates its management method integrates with the FTL design.

Figure (10) illustrates the results of the Mapping Suite test on MTRON, Transcend SLC and OCZ MLC by using the Install Linux workload. The results show lack of effective handling on the random rewrites caused by Install Linux. However, the overall performance differs from one device to another depending on whether write buffering is available. MTRON is significantly better than Transcend SLC. Significantly, OCZ MLC outperforms Transcend SLC, even though Transcend SLC has better hardware performance. As shown in Figure (10)(b), OZC has a much lower address mapping cost than Transcend SLC. It is clear that current SSD address mapping mechanisms are not suitable for the Install Linux workload. The reason is that EXT2/EXT3 headers will generate random rewrites over a large LSA area, causing a low space usage problem.



(a) MTRON: 22.95MB/sec (b) Transcend SLC: 4.47MB/sec (c) OCZ: 5.95MB/sec

Figure (10) Mapping Suite Test Result



(a) Transcend SLC:0.98MB/sec (b) OCZ: 9.5MB/sec

Figure (11) GC Suite Test Result

Figure (11) demonstrates the GC Suite test results on Transcend SLC and OCZ by using the Browser workload. The results show the PBR values of Transcend SLD scattered in a large LSA area with high intensities, which indicates that the garbage collection is triggered frequently and the garbage collection mechanism cannot handle the highly mixed hot/cold data and high rewrite ratio associated with this workload. OCZ also has high PBR readings, but the high readings are located at a few LSA, which represents low garbage collection frequency and high garbage collection cost.

According to IOMeter test results, the performance ranking of sequential write is MTRON->Samsung->OCZ->Transcend SLC; while the random access write ranking is MTRON->Samsung->Transcend SLC->OCZ. However, Benchmark Suite test results show that the performance of the SSD is related to the specific workload performed on the device. The reason for this is that the typical access patterns are lack of small data rewrite activities, which does not affect the performance of traditional mechanical hard drive. When the activities have random writes in a large LSA area or high mixed hot/cold data, they will lead to SSD management system bottlenecks.

In the random write test, Transcend SLC has much better performance than OCZ does. However, in the Mapping Suite test and GC Suite test, OCZ outperforms Transcend SLC. This is mainly because the workload demands many small writes. When the small write data is distributed randomly in a large LSA area, the space usage utilization depends on the address mapping mechanism. Proper changes of the ratio of data block and log block can increase the space usage utilization, reduce unnecessary garbage collections and management cost. It is difficult to execute garbage collection effectively if the hot/cold data is highly mixed. Garbage collection should be postponed until enough invalid data has accumulated; at the same time hot data should be separated from cold data. As shown in the results, this was why the OCZ SSD with MLC outperformed Transcend SLC.

It follows from the discussion above that a necessary characteristic of SSD management is the ability to handle small and hot data. This is best accomplished by selecting an SSD device which offers RAM write buffering. In an IOMeter sequential test, MTRON performed significantly better than Samsung. However, when small data has random writes with large LSA area and the data with larger size than the write buffer capacity, the write back is frequently triggered by the buffer management of MTRON, which causes a higher management cost than Samsung's. Although MRTOM has larger write buffer size, the write back mechanism is inappropriate. The time cost of data writes can be only achieved by the access time advantage of the RAM. Therefore, the overall performance still depends on the buffer write back mechanism. A good buffer write back mechanism should integrate with the FTL design to process small hot data and reduce the randomness of the data, rather than the size of the write buffer.

6. Conclusion

This paper discussed the test method for the SSD management performance. We proposed a new performance metric, Per-Byte-Response, to test SSD management performance, and analyzed the typical symptoms of the PBR when the management performance is low. Feedback is given to users so they can diagnose the reason for low performance. Four benchmark suites were used to evaluate each management performance. These benchmark suites, alongside PBR, yielded different results from traditional test methods. This offers considerable insight into the impact of SSD management mechanisms on actual performance.

References

[1] Nitin, A., P. Vijayan, et al. 2008. Design tradeoffs for SSD performance. USENIX 2008 Annual Technical Conference on Annual Technical Conference.

[2] Po-Chun, H., C. Yuan-Hao, et al. 2008. The Behavior Analysis of Flash-Memory Storage Systems. Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing.

[3] Luc Bouganim., et al., uFLIP: Understanding Flash IO Patterns, 4th Biennial Conference on Innovative Data Systems Research (CIDR) January 4-7, 2009, Asilomar, California, USA.

[4] Hyojun, K. and A. Seongjun 2008. BPLRU: a buffer management scheme for improving random writes in flash storage. Proceedings of the 6th USENIX Conference on File and Storage Technologies. San Jose, California, USENIX Association.

[5] Sang-Won, L., P. Dong-Joo, et al. 2007. A log buffer-based flash translation layer using fully-associative sector translation. ACM Trans. Embed. Comput. Syst. 6(3):18.

[6] Diskmon.http://technet.microsoft.com/en-us/sysinternals/bb896646.aspx.

Detecting Solid-State Disk Geometry for Write Pattern Optimization

Chun-Chieh Kuo
National Chiao-Tung University
Hsin-Chu, Taiwan, ROC

Jen-Wei Hsieh
TAIWAN TECH
Taipei, Taiwan, ROC
jenwei@mail.ntust.edu.tw

Li-Pin Chang
National Chiao-Tung University
Hsin-Chu, Taiwan, ROC
lpchang@cs.nctu.edu.tw

Abstract

Solid-state disks use flash memory as their storage medium, and adopt a firmware layer that makes data mapping and wear leveling transparent to the hosts. Even though solid-state disks emulate a collection of logical sectors, the I/O delays of accessing all these logical sectors are not uniform because the management of flash memory is subject to many physical constraints of flash memory. This work proposes a collection of black-box tests can detect the geometry inside of a solid-state disk. The host system software can arrange data in the logical disk space according to the detected geometry information to match the host write pattern with the device characteristic for reducing the flash management overhead in solid-state disks.

1 Introduction

Flash storage is an enabling technology for cyper-physical systems because of its portability, energy efficiency, and small form factors. Because flash memory has some unique physical constraints such as erase-before-write and bulk erase, it exhibits highly asymmetric performance in terms of read and write. Thus, it is important that system software and user applications of cyper-physical systems cope with this performance characteristic for high-performance data access.

Solid-state disks use flash memory as their storage medium. They adopt a firmware layer to enable transparent data access. This firmware layer is usually referred to as *flash-translation layer*, which maps logical sectors to physical flash locations and levels the wear in the entire flash memory. Not surprisingly, the management of flash memory imposes noticeable timing overheads on the processing ordinary read and write requests.

The design of an efficient flash translation layer aims at reducing the overhead of garbage collection, i.e, the extra data copy and flash erasure operations during the reclaiming of free space. Chiang et al. [1] proposed using page-level mapping between logical sectors and flash locations. This approach classifies data into different logical regions according to their update frequencies, and mapping these regions to different flash locations. Lee et al. [2] and Park et al. [4] proposed using hybrid mapping that combines block-level mapping and page-level mapping for a good balance between the mapping-table size and write performance.

In spite of firmware design optimizations, recently researchers started investigating how the host system software can cooperate with the solid-state disk firmware to reduce the flash management overheads inside of solid-state disks. Lee et al. [3] proposed a software layer in the host that converts random write requests into long and sequential write bursts. This method effectively relieves flash storage devices of heavy garbage-collection overheads, especially for those low-end flash storage devices like thumb drives. A similar technique had also been proposed for traditional disk-based storage systems: Schindler et al. proposed aligning file-system extents to disk-track boundaries to enable whole-track pre-fetching and to avoid extra disk-head movement during data accessing [6].

Even though geometry-aware data layout is a promising technique for improving read-write performance, before the host system software can arrange data they must have the geometry information of the storage device (solid-state disks in our case). Such information includes parameters specific to the physical medium like the smallest unit sizes for read/write and flash erasure. There are also logical geometry information such as the unit size of data mapping and the total number of logical sectors that a mapping table can reach. Unfortunately, storage devices will not disclose these information to the host. This prohibits the host software from optimizing data layout for device geometry.

This work proposes a test suite for detecting the geometry information inside of solid-state disks. This method treats solid-state disks as black boxes, and use a set of special read-write patterns to access the storage device and collects the I/O response times during the test. Finally, the distribution of these response times will reveal the desired geometry information.

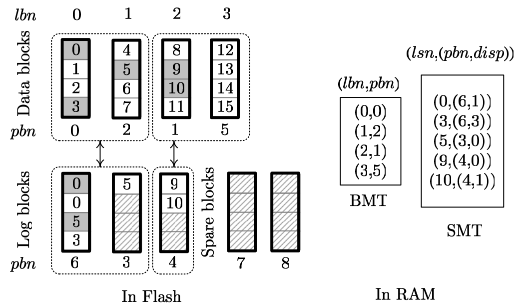


Figure 1. The set-associative mapping scheme whose group size is two. Each data-block group is associated with up to one log-block group.

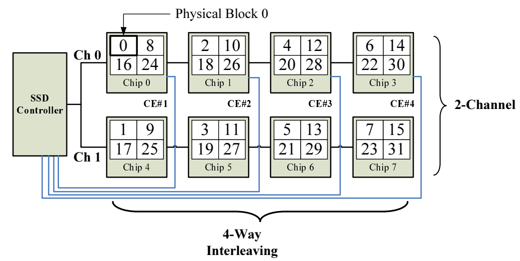


Figure 2. The effective page size is eight times as large as a flash page in a solid-state disk using a two-channel, four-way interleaving architecture. Disk sectors are mapped to flash chips using the RAID-0 style striping.

The rest of this paper is organized as follows: Section II describes the flash characteristics and the fundamentals of flash translation layers. Section III introduces the typical composition of the geometry inside of a solid-state disk, and discuss how the host system software can use these information. Section IV presents a set of tests to detect these geometry information and the test results of several off-the-shelf products. Section V concludes this work.

2 Background

A piece of flash memory is a physical array of *blocks*, and every block contains the same number of *pages*. In a typical flash specification, a flash page is 4096 plus 128 bytes, while a flash block consists of 128 pages [5]. Solid-state disks emulate a collection of logical sectors using a firmware layer called the flash-translation layer (i.e., FTL).

Flash-translation layers update existing data out of place and invalidate old copies of the data to avoid erasing a flash block every time before rewriting a piece of data. Thus, flash-translation layers require a mapping scheme to translate logical disk-sector numbers into physical locations in flash. After writing a large amount of data to flash, flash-translation layers must recycle flash pages storing invalid data by means of block erase. Before flash-translation layers erase a block, it must secure any valid data in this block-to-erase by data copying. *Garbage collection* refers to these internal copy and erase operations.

Flash-translation layers use RAM-resident index structures to translate logical sector numbers into physical flash locations, and mapping resolutions have direct impact on RAM-space requirements and write performance. Solid-state drives for a moderate-level performance requirement usually adopt hybrid mapping for a good balance between the above two factors. Fig. 1 shows a typical design of a hybrid mapping flash-translation layer [4]. Let lbn and pbn in Fig. 1 stand for a logical-block number and a physical-block number, respectively. A logical block is a collection of logical sectors. Hybrid mapping maps logical blocks to physical blocks via a *block mapping table* (i.e., BMT in this figure).

Hybrid mapping uses spare flash blocks as *log blocks* to serve new write requests, and uses a *sector mapping table* (SMT in this figure) to redirect read requests to the newest versions of data in spare blocks. In Fig. 1, term lsn represents a logical-sector number, and $disp$ is the page offset in a physical block. A group of logical blocks can share a number of flash blocks as their log blocks. In this example, a mapping group size has two logical blocks, and a group can have up to two log blocks. Whenever garbage collection is necessary, the flash-translation layer "applies" the updates of sector data in the log blocks to logical blocks, and erases log blocks to reclaim spare (free) blocks. Applying data change is basically a form of garbage collection because it involves data copy and block erase.

3 SSD Geometry Basics

This section introduce the composition of the geometry of solid-state disks and discuss how the system software can use these geometry information for data placement.

3.1 Effective Pages

Flash pages are relatively larger than disk sectors (4096 bytes compared to 512 bytes). The former is the smallest unit for flash read/write, while the latter is the smallest addressable unit in the host software. The effective unit for read and write in solid-state disks can even be several flash pages because many solid-state disks adopt multichannel architectures for parallel data transfer. Fig. 2 shows an example architecture, which uses two channels and 4-way

interleaving: The controller connects the chip-enable lines (i.e., CE's) of two parallel flash chips in the two channels together for synchronized flash operations. The four pairs of flash chips have separate CE lines, but they share the same data path and control path. Thus, during operations, the controller must issue commands to the four chip pairs in turn, and then interleaves the data transfers from/to these flash pairs over time. Logical sectors are striped among flash chips on a RAID-0 basis. Thus, in this architecture an *effective page* is eight times as large as a flash page.

The mismatch between the sizes of disk sectors and effective pages can cause serious performance problems. For example, consider that a disk volume is formatted in Linux ext4 with 4 KB allocation units. When updating a small file, the file system writes 4 KB of data to the underlying block device. Because the effective page size is 32 KB here (eight 4 KB pages), the solid-state disk first retrieves a 32 KB effective page from the flash chips into an internal page buffer, partially updates data in the buffer with new data, and then write the 32 KB of data back to the flash chips. This procedure is referred to as a read-modify-write (RMW) operation. Even worse, if the 4 KB file-system allocation unit happens to be on the boundary between two effective pages, the RMW operations will involves two effective pages. This problem also occurs if a long write burst whose starting sector number is not aligned to a boundary of effective pages.

The RMW overheads not only degrades write performance (2x in the worst case) but also shorten the device lifespan because of writing unmodified data. If the host knows the size of effective pages, then it re-arrange data structures and also de-compose write bursts to align write operations to the boundaries among effective pages.

3.2 Effective Blocks and Mapping Groups

The use of parallel architectures also proportionally enlarges the effective size of blocks. Different from effective pages, the size of effective blocks are related to the behaviors of garbage collection. Fig. 1 had shown that the flash-translation layer allocates log blocks to groups of logical blocks. Let these groups of logical blocks be *mapping groups*. When the host writes to the sectors of a mapping group which do not currently have any log blocks, the flash-translation layer will find new spare blocks as log blocks for this group. If there is not any available spare blocks, the flash-translation layer must trigger garbage collection to reclaim log blocks from other groups.

If the host writes to a collection of logical sectors which are widespread in the entire disk space, then a large number of mapping groups will demand their own log blocks. With a keen competition for log blocks among mapping groups, the flash translation layer will frequently perform garbage collection to reclaim log blocks. As a result, garbage collection can erases a log block before this log block fully utilizes all its pages for serving updates. This problem is referred to as *log-block thrashing*, and a solid-state disk will have very poor write performance when experiencing log-block thrashing.

If the host knows the sizes of effective block and mapping groups, it can place those frequently updated (written) data in the same mapping group. Confining the host write pattern to a small number of mapping groups can effectively avoid log-block thrashing.

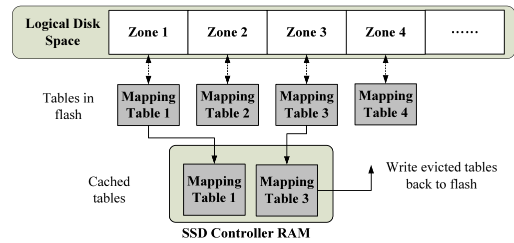


Figure 3. Dividing the logical disk space into zones and each zone uses its own mapping table. The controller can only afford to cache a subset of all these mapping tables.

3.3 Mapping Zones

The controllers in many solid-state disk designs are equipped with a very limited amount of RAM. These controllers even cannot afford to store the entire mapping table in RAM. Thus, many designs divide the entire logical disk space into *mapping zones*, and have these zones use their own mapping tables. Logically, the entire storage device is managed many separate instances of the flash-translation layer. As Fig. 3 shows, the controller can only cache a subset of all the mapping tables, and an instance of the flash-translation layer reloads its table from flash whenever necessary, and stores this table back to flash if its table is evicted from the cache.

If the host frequently accesses a collection of mapping zones and the total size of these zones' tables is larger than the table cache size, then the solid-state disk will spend a noticeable amount of time to reload and commit these mapping tables. Similar to the use of mapping groups, if the host knows the size of mapping zones, then it can place the frequently accessed (not only frequently written but also frequently read) data in the same mapping zone. This increases the hit ratio of the table cache and alleviates the overhead caused by table cache misses.

4 Experimental Results

4.1 Experimental Setup

This section is meant to explore the geometry of SSDs by a series of experiments. The experiments are conducted over a personal computer with Intel Pentium 4 CPU (3.4GHz). The operating system is Windows XP. To eliminate disturbance from the file system, we adopt Windows API, i.e., ReadFile() and WriteFile(), to access underlying storage devices. Using DeviceIoControl() in conjunction with IOCTL_ATA_PASS_THROUGH as parameter, we can send ATA command to storage devices directly. Therefore, we can impose special controls, such as DISABLE READ CACHE, DISABLE WRITE CACHE, or FLUSH WRITE CACHE over SSDs.

We evaluate the management overhead inside SSDs in terms of read/write response time. To achieve a precise measurement, the RDTSC (read time stamp counter) instruction is used to obtain a proper cycle count (which is incremented every clock cycle). Since the response time incurred by a garbage collection varies widely, trigger of a garbage collection is detected based on the throughput. For detection of SSD geometry, we disable read cache or write buffer to precisely assess how FTL adopted in various SSDs operates over underlying NAND flash memory for read/write requests. Table 1 summarizes SSDs evaluated in our experiments. Since MLC SSD is unstable in write performance, we focus on SLC SSD to present our experimental results.

Table 1. Devices under tests.

Brand	Model	Type	Size
Transcend	TS16GSSD25S-S	SLC	16 GB
Transcend	TS32GSSD25S-M	MLC	32 GB
SAMSUNG	MCBQE32G5MPP-0VA	SLC	32 GB
Mtron	MSP-SATA7525-032	SLC	32 GB
Intel	SSDSA2MH080G1GC	MLC	80 GB
OCZ	OCZSSD2-1C64G	MLC	64 GB
OCZ	OCZSSD2-1VTX60G	MLC	60 GB

4.2 Detecting Effective Page Size

4.2.1 Detection Method

When a write request is not aligned with the effective page size, one or two read-modify-write operations might be required depending on amount of the request data. The experiment is conducted by issuing two update requests with adjacent starting addresses to the target SSD iteratively. For

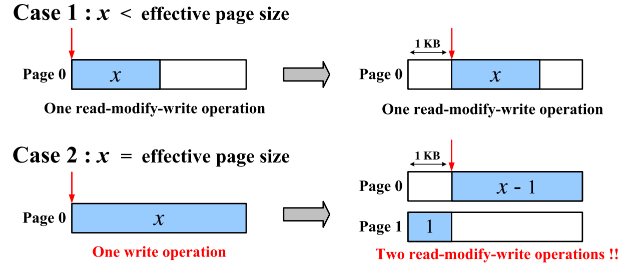


Figure 4. Effective Page Size Detector.

each iteration, amount of the updated data is incremented by 1KB. Once the difference between the response time of requests exceeds a threshold, the effective page size can thus be detected.

As shown in Fig. 4, two possible cases might be encountered as amount of the updated data increased. When amount of the updated data x is smaller than the effective page size of the target SSD, as shown in Case 1, the starting address of update requests either from 0KB or 1KB would have no impact on response time since both of them would require one read-modify-write operation. When amount of the updated data x is equal to the effective page size of the target SSD, as shown in Case 2, the request with its starting address from 0KB requires only one write operation. However, the request with the starting address from 1KB would incur two read-modify-write operations, which is time consuming compared with only one write operation. Thus the effective page size can be detected by comparing response times of two requests with adjacent starting addresses. Note that we must disable write buffer to have a precise measurement.

4.2.2 Detection Results

Fig. 7(a) and 7(b) shows the experimental result of effective page size detection for Transcend TS16GSSD25S-S and Samsung MCBQE32G5MPP-0VA. As shown in the figure, there is an obvious distinguishability on response time of update requests with starting address from 0KB and 1KB when amount of written data is 4KB for Transcend TS16GSSD25S-S and 16KB for Samsung MCBQE32G5MPP-0VA, respectively. We also conduct an experiment for read requests. As shown in Fig. 7(c), since read-modify-write has no impact on read, there is no significant difference on read response times whether we align the request with the starting address of an effective page or not. However, for those target SSDs that cannot have write buffer disabled, we must explore the effective page size from read operations. As shown in Fig. 7(d), a read request aligned with the starting address of an effective page would have a shorter response time for Mtron MSP-SATA7525-032 when data amount of the request is fixed to 8KB. It

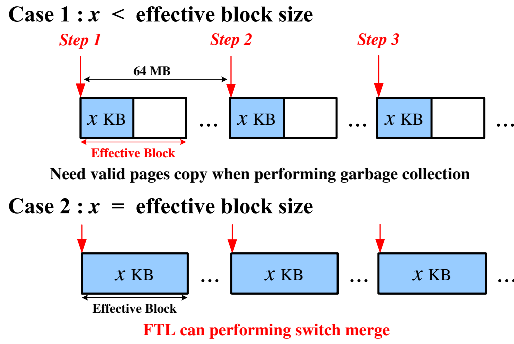


Figure 5. Effective Block Size Detector.

is because such a read request would incur an additional read operation if the request is not aligned with the effective page.

4.3 Detecting Effective Block Size and Mapping Groups

Notably, even though effective blocks and mapping groups are two different things, we use these terms interchangeably here because their difference is insignificant in terms of geometry detection.

4.3.1 Detection Method

For a block-level mapping FTL, overhead of live data copying is inevitable for a partial merge or a full merge [4]. However, when all the data in a data block are sequentially updated, a low-cost switch merge can be performed instead. The experiment is conducted by issuing update requests to the target SSD iteratively. For each iteration, amount of the sequentially updated data is doubled. Once a switch merge is triggered by an update request, the effective block size can thus be detected.

As shown in Fig. 5, two possible cases might be encountered as amount of the sequentially updated data increased. When amount of the sequentially updated data x is smaller than the effective block size of the target SSD, as shown in Case 1, a partial merge is required to reclaim free space. Since a partial merge incurs live data copying, the effective throughput drops. When amount of the sequentially updated data x is equal to the effective block size of the target SSD, as shown in Case 2, a switch merge can be adopted to reclaim free space without any live data movement. Thus the best effective throughput can be achieved.

To ensure that each request is mapped to a different logical block, we separate each subsequent request with enough space, e.g., 64MB in our experiment. As a result, log blocks are consumed quickly and a garage collection would be triggered frequently to reclaim free space for a one-to-one mapping scheme. For a many-to-one mapping scheme, merge operation would be more complex and cost of live data copying for a garbage collection can thus be observed easily.

4.3.2 Detection Results

Fig. 7(e)-7(g) shows the experimental result of effective block size detection. As shown in the figure, there is an obvious distinguishability on throughput improvement under different request sizes. The throughput improves dramatically as the request size increased. The throughput improvement achieves the best performance and becomes steady when the request size exceeds a certain amount of data due to efficiency of switch merge. Therefore, we could conclude that the effective block sizes of Transcend TS16GSSD25S-S, Samsung MCBQE32G5MPP-0VA, and Mtron MSP-SATA7525-032 are 1MB, 4MB, and 4MB, respectively.

4.4 Detecting Mapping Zones

4.4.1 Detection Method

The experiment is conducted by issuing two read requests A and B iteratively. For each iteration, the starting address of the read request A is fixed, while the starting address of the read request B is increased by 1MB. Once the requests access different zones, mapping table thrashing would be incurred. Therefore, the response time of the read request B would be longer afterward.

As shown in Fig. 6, two possible cases might be encountered as the starting address of the read request B increased. When the distance between starting addresses of two read requests is smaller than the zone size, as shown in Case 1, read requests A and B would access the same zone. Thus no mapping table reloading is required. When the distance between starting addresses of two read requests is larger than the zone size, as shown in Case 2, read requests A and B must access different zones. As a result, mapping table reloading is required. In our experiment, we repeatedly issue read requests A and B to trigger mapping table thrashing, from which the overhead of mapping table reloading would be more obvious.

4.4.2 Detection Results

As shown in Fig. 7(h), when the distance between read request A and read request B is shorter than 422MB, the response time of reading 512 Bytes data from address B is obviously better. When the distance between read request A and read request B exceeds 422MB, the response time of reading 512 Bytes data from address B is increased with a

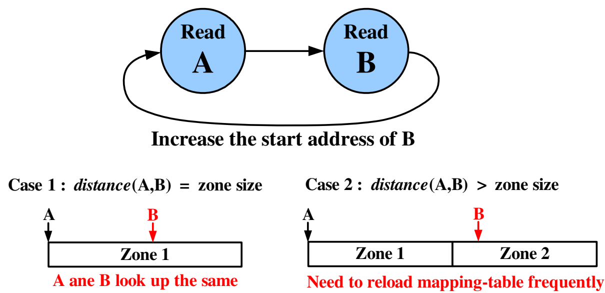


Figure 6. Zone Size Detection.

steady amount due to the overhead of mapping table loading. Therefore, we can conclude the zone size of Transcend TS16GSSD25S-S is 422MB.

5 Conclusion

The management of flash memory in solid-state disks imposes non-uniform response times on random sector accesses. Being aware of the geometry information inside of solid-state disks can help the host system software to change data placement for matching the host write pattern and the storage device characteristics. This work demonstrates a collection of black-box tests that successfully detects the geometry of flash storage devices. We believe that these techniques are beneficial to not only enhancing existing system software but also designing new file systems.

References

[1] M.-L. Chiang, P. C. H. Lee, and R. chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software Practice and Experience*, 29(3):267–290, 1999.

[2] 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.

[3] Y. Lee, J.-S. Kim, and S. Maeng. Ressd: a software layer for resuscitating ssds from poor small random write performance. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 242–243, New York, NY, USA, 2010. ACM.

[4] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–23, 2008.

[5] Samsung Electronics Company. *K9MDG08U5M 4G * 8 Bit MLC NAND Flash Memory Data Sheet*, 2008.

[6] J. Schindler, J. Griffin, C. Lumb, and G. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. In *Conference on File and Storage Technologies*, 2002.

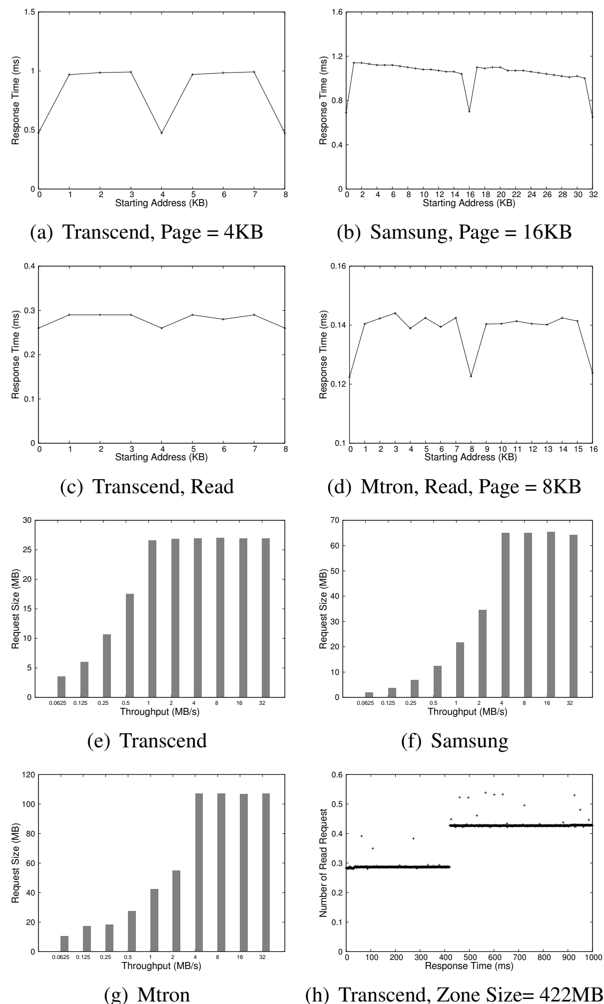


Figure 7. Experimental Results.

國科會補助專題研究計畫項下出席國際學術會議心得報告

日期：100 年 10 月 30 日

計畫編號	NSC 99-2220-E-009-047-		
計畫名稱	嵌入式網路通訊裝置評比技術與工具之研發--子計畫四:嵌入式網路通訊裝置儲存裝置效能評比基準與工具之研發(中心分項)(2/2)		
出國人員姓名	張立平	服務機構及職稱	國立交通大學資訊工程系 副教授
會議時間	12-14 April 2011	會議地點	Chicago, USA
會議名稱	(中文)嵌入式系統程式語言、編譯器、工具以及理論 (英文)ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)		
發表論文題目	(中文）一種低成本的固態硬碟平均磨損演算法 (英文）A Low-Cost Wear-Leveling Algorithm for Block-Mapping Solid-State Disks		

一、 參加會議經過

目的：

本次出席國際研討會 ACM LCTES，會議地點在美國芝加哥，該會議是 CPS Week (Cyber-Physical System) 的會議之一，同時舉行的共有 HSCC, ICCPS, IPSN, LCTES, RTAS，而其中 RTAS 與我個人研究領域也有相關，所以這次我也同時參加了 RTAS 的活動，可謂非常划算的一次行程。

這次參加 LCTES 發表一篇關於快閃記憶體平均磨損技巧的論文。

快閃記憶體目前發展看好，除了當做外部儲存的媒體之外，未來會逐漸對系統軟體以及程式語言的層面發生影響。因此，雖然 LCTES 主軸是嵌入式系統的程式語言支援與技巧，但快閃記憶體相關的議題仍然引起與會學者的莫大興趣，而我亦透過現場的問答得到了許多寶貴的觀點與意見。

過程：

本次會議，我在 LCTES 報告了一篇論文，而時程上相關的研究在同屬 CPS Week 的 RTAS 剛好有錯開，所以我就同時參與了兩個會議，並且進行了一些意見交流。

我這次報告的論文題目，是討論如何設計一個簡單、有效、而又能自我調適的平均磨損演算法。目前業界使用的平均磨損演算法，大多基於靜態平均磨損，其效果不佳。但既有成果之中，效果好的演算法其實作複雜度又頗高，所以這篇研究切入了這個議題，設計出效果又好，實作又簡單的平均磨損方法。而這研究成果亦多加探討關於平均磨損的積極程度，應該要根據寫入儲存裝置的樣式來作自我調整。故這篇論文也探討了這樣的調整應該怎麼做，以及調整的結果如何。

與會學者大多對於該方法的簡單與自我適應能力表示贊同，亦提出了關於多通道架構下的平均磨損該如何處理的問題。也就是說，先進的快閃儲存裝置都會使用多個記憶體通道來平行操作，藉以提升資

料讀寫的速度。而通道架構亦會引起新層面的平均磨損問題，也就是不同通道內的快閃記憶體其磨損頻率將不同。針對這個寶貴意見，我回國後也開發了一些對策，目前已經將該新方法加入會議論文版本，並已經投稿到期刊了。

除了 LCTES 會議內的互動，我也參加了 RTAS 的議程。其中有來自於香港的一些學者，發表關於快閃轉換層中轉址表如何做快取的技術。轉址表需要作快取，主要是因為高解析度位址轉換下（有如作業系統分頁機制下，使用極小的頁），轉址表會變得太大，而必須只能一部分放在快閃儲存控制器內的隨機存取記憶體中。而會後跟幾位研究記憶體相關的學者進行了很多交流，而我在今年六月的 DAC 與十月的 EMSOFT 也再次遇到他們，因此算是建立了一些良好的關係。

二、 與會心得

在會議過程中，與不少國內外學者交換了意見。我個人感覺，快閃儲存裝置內部的快閃轉換層，已經從過去五年韓國學者主導的混合式轉址法又回到頁級轉址法，這是因為先進的固態硬碟控制器設計商，如 Marvell 或 SandForce 等等，開始『捨得』使用硬體規格比較足夠的控制器，故頁級轉址法這兩年又重新獲得重視。此外，過去快閃儲存裝置內部多通道管理方面，大多是沒有特別的策略，也就是

將所有通道綁在一起來同步使用。這次與會之後，深深感受到這邊將是一個火熱的研究題目，值得進入好好探討。

此外，就相關研究領域的發展，目前我個人觀察到除了台灣與韓國的學者之外，目前有一批原本作即時系統或者記憶體系統的香港學者亦開始研究快閃記憶體的議題，而且他們最近在頂級的會議與期刊也有許多斬獲，而美國西岸一些大學以及微軟的研究中心也持續地發表成果。個人覺得，與會過程中接收到這類的資訊，對於將來研究題目的規劃也是有些戰略性的價值。

三、 攜回資料

本次會議攜回 LCTES 論文紙本一本，以及 CPSWeek 論文集光碟一片。

Li-Pin Chang <lpchang@gmail.com>

Your LCTES 2011 Submission (Number 18)

bjorn.desutter@elis.ugent.be <bjorn.desutter@elis.ugent.be> **2010年11月30日上午7:47**
收件者: lpchang@cs.nctu.edu.tw

Dear Prof. Li-Pin Chang:

On behalf of the LCTES 2011 Program Committee, I am delighted
to inform you that the following submission has been accepted
to appear at the conference:

A Low-Cost Wear-Leveling Algorithm for Block-Mapping
Solid-State Disks

The Program Committee worked very hard to thoroughly review
all the submitted papers. Please repay their efforts, by
following their suggestions when you revise your paper.

We will send you more information on the final submission process
and deadlines later this week.

The reviews and comments are attached below. Again, try to follow
their advice when you revise your paper.

Congratulations on your fine work. If you have any additional
questions, please feel free to get in touch.

Best Regards,
Bjorn De Sutter, PC Chair
LCTES 2011

==
LCTES 2011 Reviews for Submission #18
==

Title: A Low-Cost Wear-Leveling Algorithm for Block-Mapping Solid-State Disks

Authors: Li-Pin Chang and Li-Chun Huang

A Low-Cost Wear-Leveling Algorithm for Block-Mapping Solid-State Disks *

Li-Pin Chang
Department of Computer Science, National Chiao-Tung University, Hsin-Chu, Taiwan 300, ROC
lpchang@cs.nctu.edu.tw

Li-Chun Huang
Department of Computer Science, National Chiao-Tung University, Hsin-Chu, Taiwan 300, ROC
kellemes13@gmail.com

Abstract

Multilevel flash memory cells double or even triple storage density, producing affordable solid-state disks for end users. However, flash lifetime is becoming a critical issue in the popularity of solid-state disks. Wear-leveling methods can prevent flash-storage devices from prematurely retiring any portions of flash memory. The two practical challenges of wear-leveling design are implementation cost and tuning complexity. This study proposes a new wear-leveling design that features both simplicity and adaptiveness. This design requires no new data structures, but utilizes the intelligence available in sector-translating algorithms. Using an on-line tuning method, this design adaptively tunes itself to reach good balance between wear evenness and overhead. A series of trace-driven simulations show that the proposed design outperforms a competitive existing design in terms of wear evenness and overhead reduction. This study also presents a prototype that proves the feasibility of this wear-leveling design in real solid-state disks.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Garbage collection; B.3.2 [*Memory Structures*]: Mass Storage

General Terms Design, Performance, Algorithm.

Keywords Flash memory, wear leveling, solid-state disks.

1. Introduction

Solid-state disks are storage devices that employ solid-state memory like flash as the storage medium. The physical characteristics of flash memory differ from those of mechanical hard drives, necessitating different methods for memory accessing. Solid-state disks hide flash memory from host systems by emulating a typical disk geometry, allowing systems to switch from a hard drive to a solid-state disk without modifying existing software and hardware. Solid-state disks are superior to traditional hard drives in terms of shock resistance, energy conservation, random-access performance, and heat dissipation, attracting vendors to deploy such storage devices in laptops, smart phones, and portable media players.

* This work is in part supported by research grant NSC-98-2220-E-009-048 from National Science Council, Taiwan, ROC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

LCTES'11, April 11–14, 2011, Chicago, Illinois, USA.
Copyright © 2011 ACM 978-1-4503-0555-6/11/04. . . $10.00

Flash memory is a kind of erase-before-write memory. Because any one part of flash memory can only withstand a limited number of erase-write cycles, approximately 100K cycles under the current technology [17], frequent erase operations can prematurely retire a region in flash memory. This limitation affects the lifetime of solid-state disks in applications such as laptops and desktop PCs, which write disks at very high frequencies. Even worse, recent advances in flash manufacturing technologies exaggerate this lifetime issue. In an attempt to break the entry-cost barrier, modern flash devices now use multilevel cells for double or even triple density. Compared to standard single-level-cell flash, multilevel-cell flash degrades the erase endurance by one or two orders of magnitude [18].

Localities of data access inevitably degrade wear evenness in flash. Partially wearing out a piece of flash memory not only decreases its total effective capacity, but also increases the frequency of its housekeeping activities, which further speeds up the wearing out of the rest of the memory. A solid-state drive ceases to function when the amount of its worn-out space in flash exceeds what the drive can manage. The wear-leveling technique ensures that the entire flash wears evenly, postponing the first appearance of a worn-out memory region. However, wear leveling is not free, as it moves data around in flash to prevent solid-state disks from excessively wearing any one part of the memory. These extra data movements contributes to overall wear.

Wear-leveling algorithms include rules defining when data movement is necessary and where the data to move to/from. These rules monitor wear in the entire flash, and intervene when the flash wear develops unbalanced. Solid-state disks implement wear leveling at the firmware level, subjecting wear-leveling algorithms to crucial resource constraints. Prior research explores various wear-leveling designs under such tight resource budgets, revealing three major design challenges: First, monitoring the entire flash's wear requires considerable time and space overheads, which most controllers in present solid-state disks cannot afford. Second, algorithm tuning for environment adaption and performance definition requires prior knowledge of flash access patterns, on-line human intervention, or both. Third, high implementation complexity discourages firmware programmers from adopting sophisticated wear-leveling algorithms.

Standard solid-state-disk microcontrollers (controllers in the rest of this paper) cannot afford the RAM space overhead required to store the entire flash's wear information in RAM. Chang et al. [2] proposed caching only portions of wear information. However, periodic synching between the wear information in RAM and in flash introduces extra write traffic to flash. Jung et al. [9] proposed a low-resolution wear information method based on the average wear of large memory regions. Nevertheless, this approach suffers from distortion whenever flash wearing is severely biased. Chang et al. [5] introduced bit-indicated recent wear history. However, recent

wear history blinds wear leveling because recency and frequency are independent in terms of flash wear.

Almost all wear-leveling designs subject wear evenness to tunable threshold parameters [2, 5, 9]. The system environment in which wear leveling takes place includes many conditions, such as sector-translating algorithms, flash geometry, and host disk workloads. Even though the wear-leveling threshold remains unchanged, the results of using a wear-leveling algorithm under various system environments can be very different. Using inadequately tuned parameters can cause unexpectedly high wear-leveling overhead or unsatisfactory wear evenness. Existing approaches require human intervention or prior knowledge of the system environment for threshold tuning.

From a firmware point of view, implementation complexity primarily involves the applicability of wear-leveling algorithms. The dual-pool algorithm [2] uses five priority queues of wear information and a caching method to reduce the RAM footprints of these queues. The group-based algorithm [9] and the static wear-leveling algorithm [5] add extra data structures to maintain coarse-grained wear information and the recent history of flash wear, respectively. These approaches ignore the information already available in sector-translating algorithms, which are firmware modules accompanying wear leveling, and unnecessarily increase their design complexity.

This study presents a new wear-leveling design, called the lazy wear-leveling algorithm, to tackle the three design challenges mentioned above. First, this design does not store wear information in RAM, but leaves all of this information in flash instead. Second, even though this algorithm uses a threshold parameter, it adopts an analytical model to estimate its overhead with respect to different threshold settings, and then automatically selects a good threshold for good balance between wear evenness and overhead. Third, the proposed algorithm utilizes the address-mapping information available in the sector-translating algorithms, eliminating the need to add extra data structures for wear leveling.

The rest of this paper is organized as follows: Section 2 reviews flash characteristics and the existing algorithms for sector translating and wear leveling. Section 3 presents the proposed wear-leveling algorithm, and Section 4 describes an adaptive tuning strategy for this algorithm. Section 5 reports the results of trace-driven simulations, and Section 6 presents an implementation of the proposed algorithm based on a real solid-state disk. Section 7 concludes this paper.

2. Problem Formulation

2.1 Flash-Memory Characteristics

Solid-state disks use NAND-type flash memory (flash memory for short) as a storage medium. A piece of flash memory is a physical array of *blocks*, and each block contains the same number of *pages*. In a typical flash geometry, a flash page is 2048 plus 64 bytes. The 2048-byte portion stores user data, while the 64 bytes is a spare area for storing housekeeping data. Flash memory reads and writes in terms of pages, and it must erase a page before overwriting this page. Flash erases in terms of blocks, which consist of 64 pages. Under the current technology, a flash block can sustain a limited number of write-erase cycles before it becomes unreliable. This cycle limit depends on the type of the flash manufacturing technology: a single-level-cell flash block endures 100K cycles [17], while this limit is 10K or less in multilevel-cell flash [18]. The rest of this paper uses terms "flash blocks", "physical blocks", or simply "blocks" interchangeably.

Solid-state disks emulate disk geometry using a firmware layer called the flash translation layer (i.e., FTL). FTLs update existing data out of place and invalidate old copies of the data to avoid erasing a flash block every time before rewriting a piece of data. Thus, FTLs require a mapping scheme to translate logical disk-sector numbers into physical locations in flash. Updating data out of place consumes free space in flash, and FTLs must recycle memory space occupied by invalid data with erase operations. Before erasing a block, FTLs copy all valid data from this block to other free space. This series of copy and erase operations for reclaiming free space is called *garbage collection*. Reducing data-copy overhead during garbage collection is a priority in FTL designs.

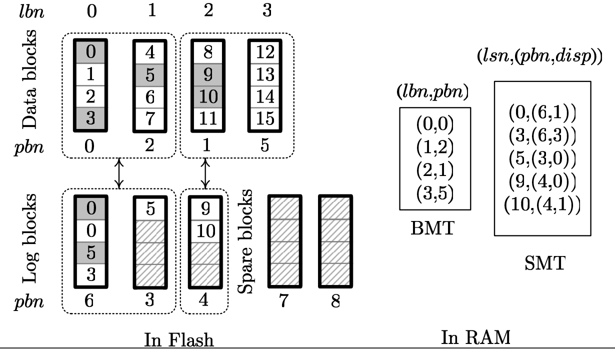


Figure 1. The set-associative mapping scheme whose group size is two. Each data-block group is associated with up to one log-block group.

2.2 Flash Translation Layers

FTLs are part of the firmware in solid-state disks. They use RAM-resident index structures to translate logical sector numbers into physical flash locations. Mapping resolutions have direct impact on RAM-space requirements and write performance. Block-level mapping [21], adopted in many entry-level flash-storage devices like USB thumb drives, requires only small mapping structures. However, low-resolution mapping suffers from slow response when servicing non-sequential write patterns. Sector-level mapping [3, 6, 7] better handles random write requests, but requires large mapping structures, making its implementation infeasible in high-capacity solid-state disks.

Hybrid mapping combines both sector and block mapping for good balance between RAM-space requirements and write performance. This method groups consecutive logical sectors as logical blocks as large as physical blocks. It maps logical blocks to physical blocks on a one-to-one basis using a *block mapping table*. If a physical block is mapped to a logical block, then this physical block is called the *data block* of this logical block. Any unmapped physical blocks are *spare blocks*. Hybrid mapping uses spare blocks as *log blocks* to serve new write requests, and uses a *sector mapping table* to redirect read requests to the newest versions of data in spare blocks.

Hybrid mapping requires two policies: the first policy forms groups of data blocks and groups of log blocks, and the second policy associates these two kinds of groups with each other. Figures 1 and 2 show two FTL designs that use different policies. Let lbn and pbn stand for a logical-block number and a physical-block number, respectively. The term lsn represents a logical-sector number, and $disp$ is the page offset in a physical block. The bold boxes stand for physical blocks, each of which has four pages. The number in the pages indicate the $lsns$ of their storage data. White pages, shadowed pages, and pages with diagonal lines represent pages containing valid data, invalid data, and free space, respectively. The BMT and the SMT are the block mapping table and the sector mapping table, respectively.

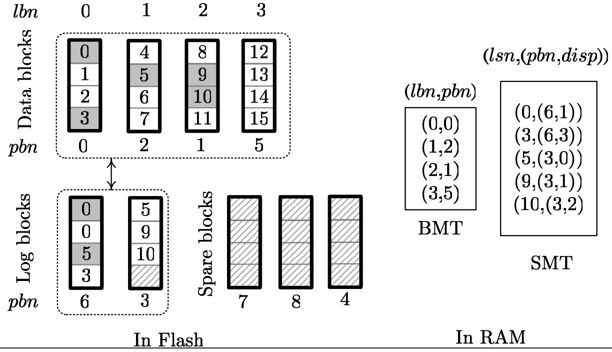


Figure 2. The fully-associative mapping scheme. All data blocks are in one group and all log blocks are in the other.

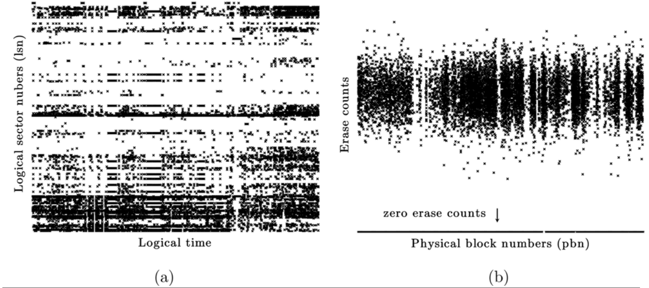


Figure 3. Flash wear in a solid-state disk under the disk workload of a laptop. (a) A fragment of the disk-write workload and (b) the final distribution of flash blocks' erase counts.

Let the group size denote the number of blocks in a group. In Fig. 1, the group size of data blocks is exactly two, while the group size of log blocks is no larger than two. This mapping scheme, called set-associative mapping, associates a data-block group with one log-block group or none. This design has two important variants: set-associative sector translation (SAST), developed by Park et al. [15], and block-associative sector translation (BAST), developed by Chung et al. [22]. SAST uses two variables, N and K, to set the group sizes of data blocks and log blocks, respectively. BAST (Block-Associative Sector Translation) [22] is simpler, fixing N=1 and K=1 always. Figure 2 depicts another mapping scheme, called fully-associative mapping. This method has only two groups associated with each other, one for all data blocks and the other for all log blocks. Fully-associative sector translation (FAST), developed by Lee et al. [12], is based on this design.

2.3 The Need for Wear Leveling

FTLs write new data in log blocks allocated from spare blocks. When they run low on spare blocks, FTLs start erasing log blocks. Before erasing a log block, FTLs collect the valid data from the log block and from the data block associated with this log block, copy this valid data to a blank block, remove the sector-mapping information related to the log block, re-direct block-mapping information to the copy destination block, and finally erase the old data block and log block into spare blocks. This procedure is called either merging operations or garbage collection.

For example, in Fig. 1, the FTL decides to erase the group consisting of log blocks at $pbns$ 3 and 6. This log-block group is associated with the group of data blocks at $pbns$ 0 and 2. The FTL prepares a group of two blank blocks at $pbns$ at 7 and 8. Next, the FTL collects four valid sectors at $lsns$ 0 through 3, and writes them to the blank block at pbn 7. Similarly, the FTL copies valid sectors at $lsns$ 4 through 7 to the blank block at pbn 8. Finally, the FTL erases the physical blocks at $pbns$ 0, 2, 3, and 6 into spare blocks, and then re-maps $lbns$ 0 and 1 to physical blocks at $pbns$ 7 and 8, respectively.

Log-block-based FTLs exhibit some common behaviors in the garbage-collection process regardless of their grouping and associating policies. FTLs never erase a data block if none of its sector data have been updated. In the set-associative mapping illustration in Fig. 1, erasing the data blocks at pbn 5 does not reclaim any free space. Similarly, in the fully-associative mapping illustration in Fig. 2, erasing any of the log blocks does not involve the data block at pbn 5. This is a potential cause of uneven flash wear.

Figure 3(a) shows a fragment of the disk-write traces recorded from a laptop PC's daily use[1]. The X-axis and the Y-axis of this figure represent the logical time and the $lsns$ of write requests, respectively. This pattern biases write requests toward a small collection of disk sectors. Let a physical block's *erase count* denote how many write-erase cycles this block has undergone. After replaying the trace set on a real solid-state disk which adopts an FAST-based FTL (Section 6.1 describes this product in more detail), Fig. 3(b) shows that the final distribution of erase counts is severely unbalanced. The X-axis and Y-axis of Fig. 3(b) represent the $pbns$ and erase counts of physical blocks, respectively. Nearly 60% of all physical blocks have zero erase counts, as the horizontal line at the bottom of Fig. 3(b) shows. In other words, this workload retires only 40% of all blocks, while the rest remain fresh. Evenly distributing erase operations can double the flash lifespan compared to that without wear leveling.

2.4 Prior Wear-Leveling Strategies

This section provides a conceptual overview of existing wear-leveling designs. Static wear leveling moves static/immutable data away from lesser worn flash blocks, encouraging FTLs to start erasing these blocks. Flash vendors including Numonyx [14], Micron [13], and Spansion [20] suggest using static wear leveling for flash lifetime enhancement. Chang et al. [5] described a static wear leveling design, and later Chang et al. [2] showed that this design is competitive with existing approaches. However, the experiments in this study reveal that static wear leveling suffers from uneven flash wear on the long-term.

Hot-cold swapping exchanges data in a lesser worn block with data from a badly worn block. Jung et al. [9] presented a hot-cold swapping design. However, Chang et al. [2] showed that hot-cold swapping risks erasing the most worn flash block pathologically. Cold-data migration relocates immutable data to excessively worn blocks and then isolates these worn blocks from wear leveling until they are no longer worn blocks compared to other blocks. Chang et al. [2] described a design of this idea. This design adopts five priority queues to sort blocks in terms of their wear information and a cache mechanism to store only frequently accessed wear leveling. However, synching the wear information between the cache and flash introduces extra write traffic to flash, and its higher implementation complexity may be a concern of firmware designers.

Unlike the wear-leveling designs above that treat wear leveling and garbage collection as independent activities, Chiang et al. [6] and Kim et al. [11] proposed heuristic functions that score flash blocks with considering garbage collection and wear leveling. In this case, FTLs erase the most scored block. However, erasing a block can require re-scoring all flash blocks. This task excessively stress the controllers and delay ordinary read/write requests.

There are compromises between algorithm concept and implementation, because the controllers can offer very limited resources. Even though different wear-leveling designs are based on

[1] This workload is the NOTEBOOK workload in Section 5.1.

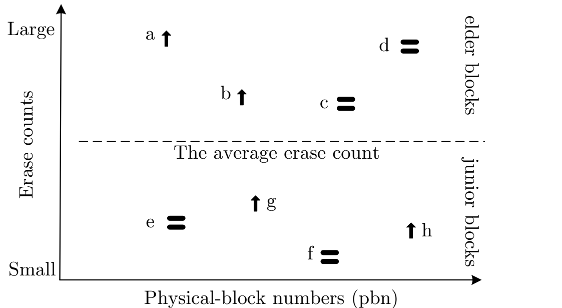


Figure 4. Physical blocks and their erase recency and erase counts. An upward arrow indicates that a block has recently increased its erase count.

the same concept, they could have very different resource demands and performance characteristics. For example, among the different designs of static wear leveling, Chang et al. [5] proposed using a periodically-refreshed bitmap to indicate not recently erased blocks. Differently, the designs from Numonyx [14] and Chang and Kuo [4] store blocks' erase counts in RAM, and involve the block of the smallest erase count in wear leveling.

Lazy wear leveling (the proposed approach) roots in cold-data migration. However, different from the dual-pool algorithm [2], which is also based on cold-data migration, lazy wear leveling adopts the following innovative designs. First, lazy wear leveling does not store blocks' wear information in RAM. It leaves them in flash instead, and utilizes the mapping information available in FTLs to assist wear leveling. In contrast, the dual-pool algorithm requires RAM space to store blocks' wear information and monitor them constantly. Caching the frequently referenced wear information helps to reduce the RAM requirements, but synching wear information between the cache and RAM can add up to 10% of extra write traffic to flash [2]. The second new idea in lazy wear leveling is the ability of self tuning. Wear-leveling algorithms subject wear evenness to a threshold parameter. However, the overhead of wear leveling grows at different rates under different system environments when changing the threshold value. Lazy wear leveling characterizes the overhead as a function of the threshold values, and adaptively tunes the threshold for good balance between the overhead and wear evenness.

3. A Low-Cost Wear-Leveling Algorithm for Block-Mapping FTLs

3.1 Observations

Let the *update recency* of a logical block denote the time length since the latest update to this logical block. If a logical block's last update is more recent than the average update recency, then this logical block's update recency is high. Otherwise, its update recency is low. Analogously, let the *erase recency* of a physical block be the time length since the latest erase operation on this block. Thus, immediately after garbage collection erases a physical block, this block has the highest erase recency among all blocks. A physical block is an *elder block* if its erase count is larger than the average erase count. Otherwise, it is a *junior block*. Notice that block seniority is a relative measure. For example, even though all blocks in a brand-new flash have small erase counts, there will be some elder blocks and junior blocks.

FTLs avoids erasing flash blocks mapped to unmodified logical blocks, because erasing these flash blocks reclaims no free space. Thus, the temporal localities of writing disk sectors can translate into temporal localities of erasing physical blocks. If a flash block has a high erase recency, then this block was not mapped to a static logical block. This flash block will then be mapped to a recently modified logical block. Because of temporal localities of writing disk sectors, recently modified logical blocks will be frequently modified. Therefore, the flash block will be mapped to mutable logical blocks and frequently increases its erase count. Conversely, a physical block loses momentum in increasing its erase count if its erase recency is low.

Figure 4 provides an example of eight physical blocks' erase recency and erase counts. Upward arrows mark physical blocks currently increasing their erase counts, while an equal sign indicates otherwise. Block *a* is an elder block with a high erase recency, while block *d* is an elder but has a low erase recency. The junior block *h* has a high erase recency, while the erase recency of the junior block *e* is low.

A block should keep its erase count close to the average. For instance, the junior blocks *g* and *h* are increasing their erase counts toward the average, while the difference between the average and the erase counts of the elder blocks *c* and *d* is decreasing. However, other than the above two cases, block wear can require intervention from wear leveling. First, the junior blocks *e* and *f* have not recently increased their erase counts. As their erase counts fall below the average, wear leveling has them start participating in garbage collection. Second, the elder blocks *a* and *b* are still increasing their erase counts. Wear leveling should have garbage collection stop further wear in these two elder blocks.

3.2 The Lazy Wear-Leveling Algorithm

This study proposes a new wear-leveling algorithm based on a simple principle: whenever any elder blocks' erase recency becomes high, the algorithm re-locates (i.e., re-maps) logical blocks with a low update recency to these elder blocks. This algorithm, called the *lazy wear-leveling algorithm*, is named after its passive reaction to unbalanced flash wear.

Lazy wear leveling focuses on the wear of elder blocks only, because elder blocks retire before junior blocks. Thus, being aware of recent wear of elder blocks is important. Physical blocks boost their erase recency only when the FTL erases them for garbage collection. Thus, if the FTL notifies lazy wear leveling of its decision on the next victim block, lazy wear leveling can check this victim block's seniority. This way, lazy wear leveling needs not repeatedly check all elder blocks' wear information.

How to prevent elder blocks from further aging is closely related to garbage-collection behaviors: Garbage collection has no interest in erasing a data block if this data block is not associated with any log blocks. A data block does not require any log blocks for storing new updates if the logical block mapped to this data block has a low update recency. Because recent sector updates to a logical block leaves mapping information in the FTL's sector-mapping table, lazy wear leveling selects logical blocks not related to any sector-mapping information as logical blocks with a low update recency. The logical block at lbn 3 in Fig. 1 and 2 is such an example.

Re-mapping logical blocks with a low update recency to elder blocks can prevent elder blocks from wearing further. To re-map a logical block from one physical block to another, lazy wear leveling moves all valid data from the source physical block to the destination physical block. This invalidates all data in the source block and directs the upcoming garbage-collection activities to the source block. Junior blocks are the most common kind of source blocks, e.g., blocks *e* and *f* in Fig. 4, because the storage of immutable data keeps them away from garbage collection. Therefore, selecting logical blocks for re-mapping is related to the wear of junior blocks. To give junior blocks an even chance of wear, it is important to uniformly visit every logical block when selecting a logical block for re-mapping.

Algorithm 1 The lazy wear-leveling algorithm

Input: v: the victim block for garbage collection
Output: p: a substitute for the original victim block v

1: $e_v \leftarrow eraseCount(v)$
2: **if** $(e_v - e_{avg}) > \Delta$ **then**
3:     **repeat**
4:         $l \leftarrow lbnNext()$
5:     **until** $_lbnHasSectorMapping(l)$=FALSE
6:     $_erase(v)$;
7:     $p \leftarrow _pbn(l)$
8:     $_copy(v, p)$; $_map(v, l)$
9:     $e_v \leftarrow e_v + 1$
10:     $e_{avg} \leftarrow updateAverage(e_{avg}, e_v)$
11: **else**
12:     $p \leftarrow v$
13: **end if**
14: RETURN p

The temporal localities of write requests can change occasionally. Disk workloads can start updating a logical block which previously had a low update recency. If this logical block was recently re-mapped to an elder block for wear leveling, then the new updates neutralize the prior re-mapping operation. However, lazy wear leveling will perform another re-mapping operation for this elder block when the FTL is about to erase this elder block again.

3.3 Interaction with FTLs

This section describes how lazy wear leveling interacts with its accompanying firmware module, the flash translation layer. Lazy wear leveling and the FTL operate independently, but the FTL provides some information to assist wear leveling. Algorithm 1 shows the pseudo code of the lazy wear-leveling algorithm. The FTL invokes this procedure every time it erases a victim block for garbage collection. This procedure determines if wear leveling needs intervene in the erasure of the victim block. If so, this procedure looks for a logical block that has not been updated recently, re-maps this logical block to the victim block, and then selects the physical block previously mapped to this logical block as a substitution for the original victim block. Notice that the FTL needs not consider wear leveling when selecting victim blocks. In other words, lazy wear leveling is independent of the FTL's victim-selection policy.

In Algorithm 1, the FTL provides the subroutines with leading underscores, and wear leveling implements the rest. The algorithm input is v, the victim block's physical block number. Step 1 obtains the erase count e_v of the victim block v using $eraseCount()$. Step 2 compares e_v against the average erase count e_{avg}. If e_v is larger than e_{avg} by a predefined threshold Δ, then Steps 3 through 10 will carry out a re-mapping operation. Otherwise, Steps 12 and 14 return the original victim block to the FTL intact.

Steps 3 through 5 find a logical block with a low update recency. Step 4 uses the subroutine $lbnNext()$ to obtain l the next logical block number to visit, and Step 5 calls the subroutine $_lbnHasSectorMapping()$ to check if the logical block l has any related mapping information in the FTL's sector-mapping table. These steps cycle through all logical blocks until they find a logical block not related to any sector-mapping information. As mentioned previously, to give all junior blocks (which are related to logical blocks with a low update recency) an equal chance to get erased, the subroutine $lbnNext()$ must evenly visit all logical blocks. The implementation of $lbnNext()$ can be any permutations of all logical block numbers, such as the Linear Congruential Generator [16]. Using permutations also maximizes the interval between two consecutive visits to the same logical blocks, reducing the probability of re-mapping a logical block with a low update recency from an elder block to another.

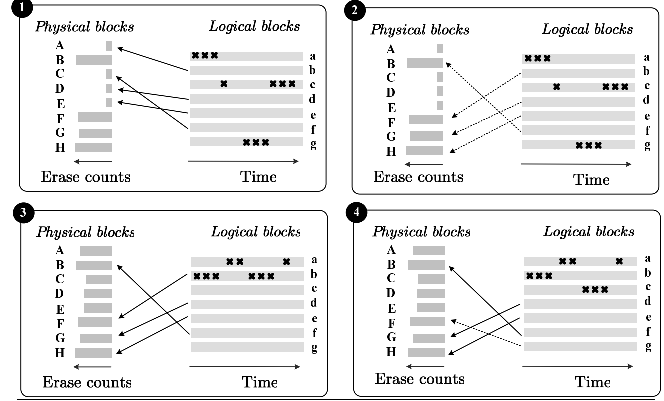


Figure 5. A scenario of running the lazy wear-leveling algorithm. Crosses indicate write requests to logical blocks.

Steps 6 through 8 re-map the previously found logical block l. Step 6 erases the original victim block v. Step 7 uses the subroutine $_pbn()$ to identify the physical block p that the logical block l currently maps to. Step 8 copies the data of the logical block l from the physical block p to the original victim block v, and then re-maps the logical block l to the former victim block v using the subroutine $_map()$. After this re-mapping, Step 9 increases e_v since the former victim block v has been erased, and Step 10 updates the average erase count. Step 14 returns the physical block p, which the logical block l previously mapped to, to the FTL as a substitute for the original victim block v.

3.4 Algorithm Demonstration

Figure 5 shows a four-step scenario of using the lazy wear-leveling algorithm. In each step, the left-hand side depicts the physical blocks and their erase counts, and the right-hand side shows the logical blocks and their updates marked with bold crosses. This example shows only the mapping of logical blocks with a low update recency to elder physical blocks.

Step 1 shows the initial condition. Let the erase counts of the elder physical blocks B, F, G, and H be greater than the average by Δ. Step 2 shows that lazy wear leveling re-maps logical blocks of a low update recency f, b, d, and e to elder physical blocks B, F, G, and H, respectively. As garbage collection avoids erasing physical block with no invalid data, Step 3 shows that physical blocks other than B, F, G, and H increase their erase counts, after processing a new batch of write requests. In this case, the wear of all blocks is becoming even.

In Step 3, the write pattern generates several updates to the logical block b. However, previously in Steps 1 and 2, this logical block had a low update recency, and wear leveling already re-mapped it to the elder physical block F. As previously mentioned in Section 3.2, these new updates to the logical block b will cause further wear of the elder physical block F, making the prior re-mapping operation of the logical block b ineffective in terms of wear leveling. Step 4 shows that lazy wear leveling re-maps another logical block g with a low update recency to the elder physical block F as soon as it learns that the FTL is about to erase the elder physical block F.

4. Adaptive Self Tuning

Tuning the threshold parameter Δ helps lazy wear leveling to achieve good balance between overhead and wear evenness. This tuning strategy consists of two parts: Section 4.1 presents an analytical model of the overhead and wear evenness of wear leveling.

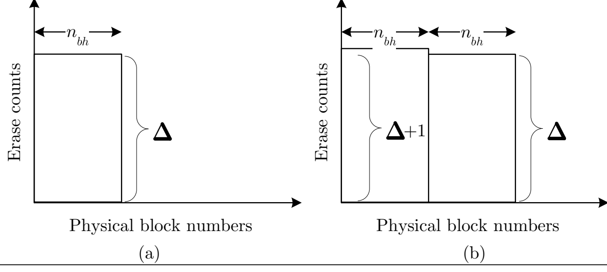


Figure 6. Erase counts of flash blocks right before the lazy wear-leveling algorithm performs (a) the first re-mapping operation and (b) the n_{bh}+1-th re-mapping operation.

Section 4.2 introduces an on-line algorithm that adjusts Δ based on the analytical model.

4.1 Performance Analysis: Overhead and Wear Evenness

Consider a piece of flash memory consisting of n_b physical blocks. Let immutable logical blocks map to n_{bc} among all physical blocks. Let the sizes of write requests be multiples of the block size. Let write requests be aligned to block boundaries. Suppose that the disk workload uniformly writes the mutable logical blocks. Thus, the FTL evenly increases the erase counts of the n_{bh}=$n_b - n_{bc}$ physical blocks.

Let the function $f(x)$ denote how many blocks garbage collection erases to process a workload writing x logical blocks. Consider the case $x = i \times n_{bh} \times \Delta$, where i is a non-negative integer. As all request sizes are multiples of the block size and requests are block-aligned, erasing victim blocks does not cost garbage collection any overhead in copying data. Thus, without wear leveling, we have

$$f(x) = x.$$

Now, consider wear leveling enabled. For ease of presentation, this simulation revises the lazy wear leveling algorithm slightly: instead of comparing the victim block's erase count to the average erase count, the algorithm compares it against the smallest among all blocks' erase counts. Figure 6(a) shows that, right before lazy wear leveling performs the first re-mapping, garbage collection has uniformly accumulated $n_{bh} \times \Delta$ erase counts in n_{bh} physical blocks. In the subsequent n_{bh} erase operations, garbage collection erases each of these n_{bh} physical blocks one more time, and increases their erase counts to $\Delta + 1$. Thus, lazy wear leveling conducts n_{bh} re-mapping operations for these physical blocks at the cost of erasing n_{bh} blocks. These re-mapping operations re-direct garbage-collection activities to another n_{bh} physical blocks. Similarly, Fig. 6(b) shows that, after garbage collection accumulates another $n_{bh} \times \Delta$ erase counts in these new n_{bh} physical blocks, lazy wear leveling again spends n_{bh} erase operations for re-mapping operations. Let function $f'(x)$ be analogous to $f(x)$, but with wear leveling enabled. We have

$$f'(x) = x + \left\lfloor \frac{x}{\Delta} \right\rfloor = x + i \times n_{bh}.$$

Under real-life workloads, the frequencies of erasing these n_{bh} blocks may not be uniform. Thus, $f'(x)$ adopts a coefficient K to take this into account:

$$f'(x) = x + i \times n_{bh} \times K.$$

The coefficient K depends on various system conditions, such as flash geometry, host workloads, and FTL algorithms. For example, dynamic changes in temporal write localities can increase K because the write pattern might start updating the logical blocks which wear leveling has previously used for re-mapping.

Let the *overhead function* $g(\Delta)$ denote the *overhead ratio* with respect to Δ:

$$g(\Delta) = \frac{f'(x) - f(x)}{f(x)} = \frac{i \times n_{bh} \times K}{i \times n_{bh} \times \Delta} = \frac{K}{\Delta}.$$

Because lazy wear leveling compares victim blocks' erase counts against the average erase count rather than the smallest erase count, we use 2Δ as an approximation of the original Δ, and have the coefficient K include the compensation for the error in the approximation. Thus, we have

$$g(\Delta) = \frac{K}{2\Delta}. \tag{1}$$

Notice that, when Δ is small, a further decrease in Δ rapidly increases the overhead ratio. For example, decreasing Δ from 4 to 2 doubles the overhead ratio.

Next, let us focus on the relation between Δ and the wear evenness in flash. Let the metric of the wear evenness be the standard deviation of all blocks' erase counts, i.e., $\sqrt{\frac{1}{n_b}\sum_{i=1}^{n_b}(e_{b_i} - e_{avg})^2}$. The smaller the standard deviation is, the more even the wear of flash blocks is. Provided that wear leveling is successful, $\sum_{i=1}^{n_b}(e_{b_i} - e_{avg})^2$ would be bounded by $n_b \times \Delta^2$. Thus, the relation between the wear evenness and Δ would be bounded by a linear relation.

4.2 On-Line Δ Tuning

As the wear evenness is linearly related to Δ, small Δ values are always preferred in terms of wear evenness. Differently, the relation between the overhead and Δ is non-linear, and decreasing Δ value can cause an unexpectedly large overhead increase. Thus, in spite of limiting the total overhead, setting Δ should consider whether the overhead is worth the wear evenness. This section presents an on-line algorithm that dynamically tunes Δ for balance between overhead and wear evenness. Because there are simple means to limit the total overhead such as adjusting the duty cycle of wear leveling, this study focuses on limiting the overhead growth rate when tuning Δ.

Under dynamic disk workloads, the coefficient K in $g(\Delta)$ may vary over time. Thus, wear leveling must first determine the coefficient K before using $g(\Delta)$ for Δ-tuning. This study proposes a session-based method for Δ-tuning. A session refers to a time interval in which lazy wear leveling contributed a pre-defined number of erase counts. This number is the session length. The basic idea is to compute K_{cur} of the current session and use this coefficient to find Δ_{next} for the next session.

The first session adopts Δ=16, but in theory this initial Δ value can be any number because it will not affect K. Let the current session adopts Δ_{cur}. Figure 7 illustrates the concept of the Δ-tuning procedure: during a runtime session, lazy wear leveling records the erase counts contributed by the garbage collection and wear leveling. At the end of the current session, the first step (in Fig. 7) computes the overhead ratio $\frac{f'(x)-f(x)}{f(x)}$, i.e., $g(\Delta_{cur})$, and solves K_{cur} of the current session using Equation 1, i.e., $K_{cur} = 2\Delta_{cur} \times g(\Delta_{cur})$.

The second step uses $g(\Delta_{next})$=$K_{cur}/(2\Delta_{next})$ to find Δ_{next} for the next session. Basically, lazy wear leveling minimizes Δ values subject to a user-defined limit λ on the growth rate of the overhead ratio (when decreasing Δ). Let the unit of the overhead ratio be one percent. For example, λ=-0.1 means that the overhead ratio increases from x% to (x+0.1)% when decreasing Δ from y to (y-1). Solve $\frac{d}{d\Delta}g(\Delta_{next}) = \frac{\lambda}{100}$ for the smallest Δ value subject to λ. Rewriting this equation, we have

$$\Delta_{next} = \sqrt{\frac{100}{-\lambda}}\sqrt{g(\Delta_{cur})\Delta_{cur}}.$$

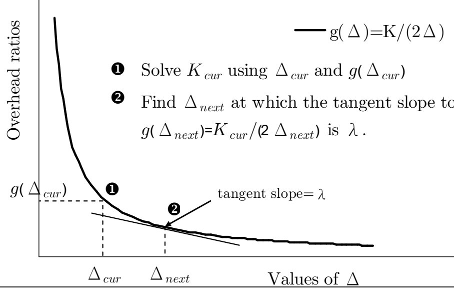


Figure 7. Computing Δ_{next} subject to the overhead growth limit λ for the next session according to Δ_{cur} and the overhead ratio $g(\Delta_{cur})$ of the current session.

For example, when λ=-0.1, if the overhead ratio $g(\Delta_{cur})$ and Δ_{cur} of the current session are 2.1% and 16, respectively, then Δ_{next} for the next session is $\sqrt{\frac{100}{0.1}}\sqrt{2.1\% \times 16} = 18.3$.

The Δ-tuning method adjusts Δ on a session-by-session basis. It requires the session length as the period of adjusting Δ, and λ as a user-defined boundary between linear and super-linear overhead growth rates. The later experiments show that λ=-0.1 is a reasonably good setting, and wear-leveling results are insensitive to different session lengths.

5. Performance Evaluation

5.1 Experimental Setup and Performance Metrics

We built a solid-state disk simulator using System C [8]. This simulator includes a flash module for behavioral simulation on read, write, and erase operations. This flash module can also accept different geometry settings. Based on this flash module, the simulator implements different FTL algorithms, including BAST [22], SAST [15], and FAST [12], which are representative designs at the current time. We tailored the lazy wear-leveling algorithm to accompany each of the FTL algorithm. This simulator also includes the static wear-leveling algorithm based on Chang's design [5]. Static wear leveling is widely used in industry [13, 14, 20] and has been proven competitive with existing wear-leveling algorithms [2].

The input of the simulator is a series disk requests, ordered chronologically. These disk requests were recorded from four types of real-life host systems: a Windows-based laptop, a desktop PC running Windows, a Ubuntu Linux desktop PC, and a portable media player. The user activities of the laptop and desktop workloads include web surfing, word processing, video playback, and gaming, while those of the media player workload are to copy, play, and delete MP3 and video files. These choices include popular options of operating systems (e.g., Linux or Windows), file systems (e.g., ext4 or NTFS), hard-drive capacity, and system usages (e.g., mobile or desktop). Table 1 describes the four disk workloads.

This study adopts two major performance metrics for flash-wear evenness and wear-leveling overhead. The standard deviation of all flash blocks' erase counts (the standard deviation for short) indicates the wear evenness in the entire flash. The smaller the standard deviation is, the more level is the wear in flash. The mean of all flash blocks' erase counts (the mean for short) is the arithmetic average of all blocks' erase counts. The difference between the means of with and without wear leveling reveals the overhead of wear leveling in terms of erase operations. The smaller the mean increase is, the lower is the wear-leveling overhead. It is desirable to achieve both a small standard deviation and a small mean increase.

Unless explicitly specified, all experiments adopted the following default settings: The threshold parameters Δ and TH of lazy

Workload	Operating system	Volume size	File system	Total written
Notebook	Windows XP	20 GB	NTFS	27
Desktop 1	Windows XP	40 GB	NTFS	81
Desktop 2	Ubuntu 9	40 GB	ext4	55
Multimedia	Windows CE	20 GB	FAT32	20
				GB

Table 1. The four experimental workloads.

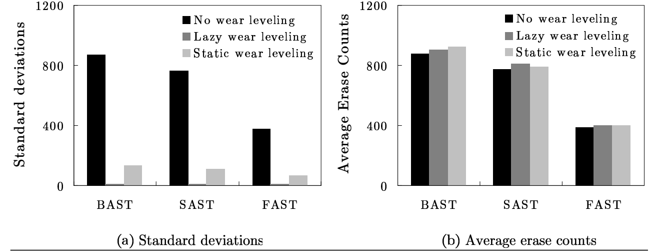


Figure 8. Evaluating lazy wear leveling and static wear leveling with FTL algorithms BAST, SAST, and FAST under the notebook disk workload.

wear leveling and static wear leveling were both 16. TH refers to the ratio of the total erase count to the total number of recently erased flash blocks (i.e., the blocks indicated as one in the erase bitmap). Dynamic Δ tuning will be evaluated in Section 5.3. The flash page size and block size were 4KB and 512KB, respectively, reflecting a typical geometry of MLC flash [18]. The input disk workload was the notebook workload, and the FTL algorithm was FAST [12]. The sizes of the logical disk volume and the physical flash were 20GB and 20.5GB, respectively. Thus, the *over-provisioning ratio* was (20.5-20)/20=2.5%. The experiments replayed the input workload one hundred times to accumulate sufficiently many erase cycles in flash blocks. This helped to differentiate the efficacy of different wear-leveling algorithms. These replays did not manipulate the experiments. Provided that wear leveling is effective, replaying the input disk workload once sufficiently erases the entire flash one time.

5.2 Experimental Results

5.2.1 Effects of Using Different FTL Algorithms

Figure 8 shows the results of using BAST, SAST, and FAST with lazy wear leveling and static wear leveling. The Y-axes of Fig. 8(a) and 8(b) indicate the standard deviations and the means, respectively. First consider the results without using wear leveling. These results show that FAST achieved the smallest mean among the three FTL algorithms. This is because FAST fully utilizes free space in every log bock [12]. On the contrary, BAST suffered from very high garbage-collection overheads, because BAST has poor space utilization in log blocks. These observations agreed with that reported in prior work [12, 15, 22].

Lazy wear leveling consistently delivered low standard deviations under the three FTL algorithms. Its standard deviations were between 10 and 12, almost not affected by FTL algorithms. In contrast, static wear leveling's standard deviations were much larger than that of lazy wear leveling, and was very sensitive to the use of different FTL algorithms. In particular, its standard deviations were 137 and 66 under BAST and FAST, respectively. Regarding wear-leveling overhead, the mean increase of lazy wear leveling was very small, which was no more than 3% in all experiments. Static wear leveling's mean increase was slightly larger, reaching 6%.

Figure 8(b) shows that when the FTL algorithm was SAST, lazy wear leveling introduced a slightly larger mean increase than

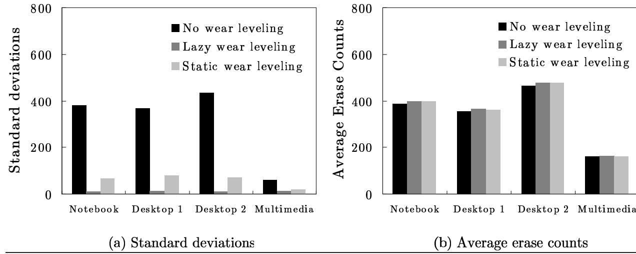


Figure 9. Experimental results of using lazy wear leveling and static wear leveling under the four types of disk workloads.

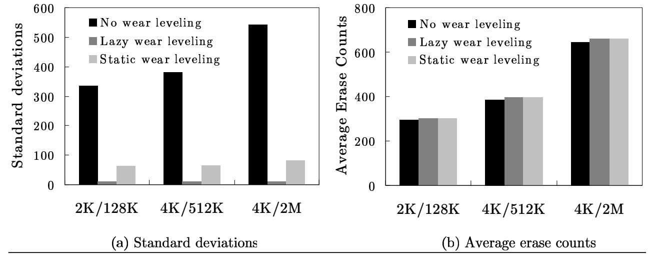


Figure 10. Experimental results under different settings of flash geometry.

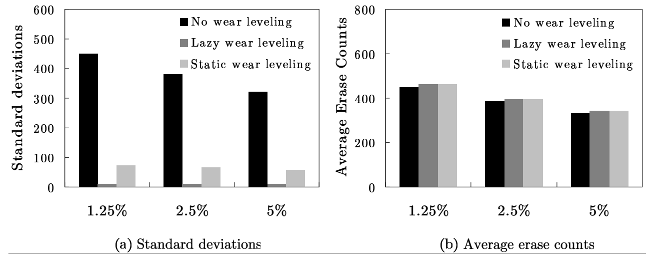


Figure 11. Experimental results under different over-provisioning ratios of flash memory.

static wear leveling. This is due to the different definitions of the threshold parameters of lazy wear leveling and static wear leveling. For a fair comparison, we set $\Delta = 18$ and $TH = 16$ such that the two wear-leveling algorithms produced the same mean increase. Under these settings, the standard deviations of lazy wear leveling was 18, which was much better than 107 in static wear leveling. Section 5.4 provides explanations of the large standard deviation of static wear leveling.

5.2.2 Effects of Using Different Host Workloads

This part of the experiment evaluated wear-leveling algorithms under the four types of disk workloads (as Table 1 shows). The number of times each of the four workload replays was subject to a constant ratio of the total amount of data written into the disk to the logical disk volume size. This ratio was determined by replaying the notebook workload 100 times, i.e., (100×27GB)/20GB=135.

Figure 9 shows that, without wear leveling, the multimedia workload had the smallest mean and standard deviation among the four workloads. This workload consisted of plenty of large and sequential write requests that accessed almost the entire disk space. Therefore garbage collection incurred mild overhead and accumulated erase cycles in all flash blocks at nearly the same rate. On the other hand, the standard deviations and means of using the notebook workload and the two desktop workloads were large. This is because these disk workloads consisted of temporal localities, which amplified the garbage-collection overhead and biased the flash wear as well.

Figure 9 shows that, regardless of the disk workload adopted, lazy wear leveling successfully lowered the standard deviations to about 10. Lazy wear leveling caused only marginal mean increase, no more than 3% under all workloads. On the other hand, even though static wear leveling's increases on the mean were comparable to that of lazy wear leveling, its large standard deviations indicate that it failed to balance the flash wear in all workloads.

5.2.3 Flash Geometry and Over-Provisioning Ratios

Flash geometry and over-provisioning ratios directly affect garbage-collection overhead and the wear evenness in flash. This experiment has two parts. The first part considered three kinds of flash geometry of page size/block size: 2KB/128KB, 4KB/512KB, and 4KB/2MB. The first and the second setups were typical geometries of SLC flash [17] and MLC flash [18], respectively. Advanced architecture designs employ multiple channels for parallel access over multiple flash chips [1, 10, 19]. Thus, the third setting corresponds to the effective geometry of a four-channel architecture. The results in Fig. 10 show that, without wear leveling, adopting coarse-grained flash geometry not only increased the overhead of garbage collection but also degraded the evenness of flash wear. When using lazy wear leveling, the standard deviations and the mean increases were both small. This advantage remained whether the flash geometry was coarse-grained or fined-grained.

The second part of this experiment adopted three over-provisioning ratios: 1.25%, 2.5%, and 5%. The smaller the over-provisioning ratio is, the fewer log blocks the FTL can have. Figure 11 indicates that using small over-provisioning ratios resulted in high overhead of garbage collection. This is because the demand for free space forced the FTL to prematurely copy valid data for garbage collection before these valid data might be invalidated by new write request. Amplified garbage-collection activities also increased the wear unevenness in flash. When using lazy wear leveling, the standard deviations and the mean increases were again small, and its performance was not significantly affected by using different over-provisioning ratios.

5.3 Automated Δ-tuning

This experiment adopted two system configurations C_1 and C_2: the configuration C_1 used the Linux desktop workload with BAST, while the configuration C_2 adopted the notebook workload with FAST. The flash geometry was in both C_1 and C_2 were both 4KB/2MB. The over-provisioning ratios of C_1 and C_2 were 1.25% and 0.625%, respectively.

This experiment consists of three parts. The first part reports the overhead and the standard deviation with respect to different static Δ settings (i.e., dynamic Δ-tuning was disabled) under various system configurations. Figure 12(a) depicts that the relations between Δ and standard deviations appear linear in both C_1 and C_2. This agrees with the analysis of wear evenness in Section 4.1. When Δ was large, the standard deviations of C_1 were larger than those of C_2, indicating that C_1 required more wear leveling than C_2. Figure 12(b) depicts the overhead ratios (see Section 4.1 for definition) for different Δ values. The two solid curves depicts the actually measured overhead ratios in C_1 and C_2. The two dotted lines plot the estimated overhead using $g(\Delta)$ with K=1.2 and K=0.76. The dotted lines and the solid lines are very close, showing that $g(\Delta)$ can produce accurate overhead estimation. The overhead increased faster in C_1 than in C_2, indicating that the cost of wear leveling was higher in C_1.

The second part of this experiment enabled the dynamic Δ-tuning method presented in Section 4.2. The session length for

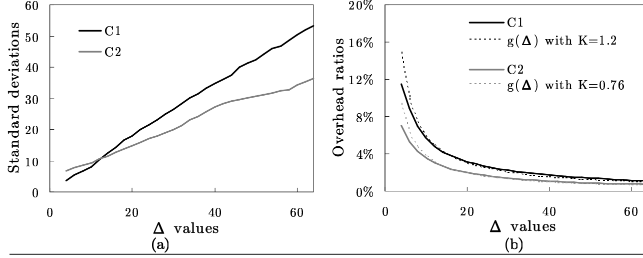


Figure 12. Under system configurations C_1 and C_2, (a) the standard deviations and (b) the overhead ratios with respect to different Δ settings.

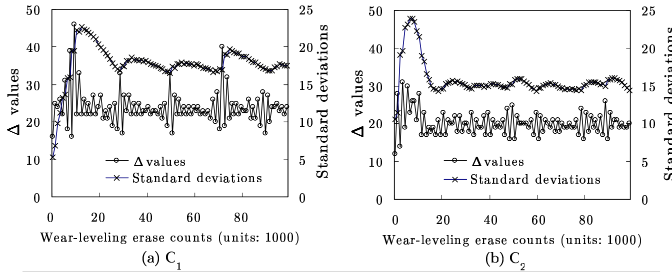


Figure 13. Runtime Δ values and standard deviations in system configurations C_1 and C_2 with the Δ-tuning method enabled. The final overhead ratios of C_1 and C_2 were 2.22% and 1.95%, respectively.

Δ-tuning was 1,000, meaning that Δ adjusted every time after lazy wear leveling erased 1,000 blocks. The value of λ was -0.1. Figure 13 plots the Δ values and the standard deviations session-by-session. The Δ value dynamically adjusted during experiments, and the standard deviations occasionally increased but remained at controlled levels. Overall, even though C_1 requires more wear leveling than C_2 (as Fig. 12(a) shows), the tuning method still refrained from using small Δ values in C_1 because in C_1 the overhead grew faster than in C_2 (as Fig. 12(b) shows).

The third part reports results of using different settings of λ and session lengths. This part used λ=-0.2 in comparison with λ=-0.1 in configuration C_2. When switching λ from -0.1 to -0.2, the overhead ratio increased about 1.7 times (from 1.95% to 3.37%), while the standard deviation improved by only 15% (from 14.46 to 12.28). This is because the overhead growth (when decreasing Δ) can become super-linear when the tangent slope to $g(\Delta)$ is smaller than -0.1 (as Fig. 12(b) shows). Therefore, using λ=-0.2 produced only marginal improvement upon the standard deviation which is not worth the large overhead increase. This part also includes results of using different session lengths. The final standard deviations of C_1 with session lengths 1000, 2000, and 3000 were 14.46, 14.86, and 14.51, respectively. The final overhead ratios with these three session lengths were 1.95%, 2.02%, and 2.05%, respectively. Thus, the efficacy of the Δ-tuning method is insensitive to session-length settings.

5.4 Wear-Leveling Stability

Keeping the standard deviation stable is as important as keeping it low. This experiment observed the change history of standard deviations using different wear-leveling algorithms. The experiment settings here are the same as those in Section 5.2.2. The trace-collecting duration of the notebook workload was one month. Thus, the experimental setting emulated an eight-year session of disk access by replaying the trace 100 times.

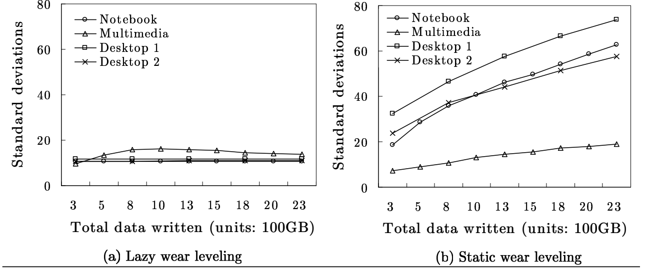


Figure 14. History of changes in standard deviations when using lazy wear leveling and static wear leveling.

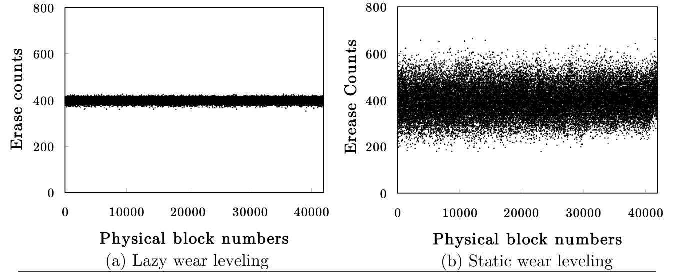


Figure 15. The final distribution of blocks' erase counts under the notebook workload.

Figure 14 shows the standard deviations when using lazy wear leveling and static wear leveling under four types of disk workloads. The X-axes and Y-axes indicate the total amount of data written into the disk and the standard deviations, respectively. Let the *stable interval* of a wear-leveling algorithm be the longest time period $[t, t']$ in which the standard deviations at time points t and t' are the same. A wear-leveling algorithm is *stable* if its stable interval increases as the total amount of data written into the disk increases. Figure 14(a) shows that lazy wear leveling was stable under all workloads. On the contrary, Fig. 14(b) shows that static wear leveling was instable. Figure 15 shows the final distribution of erase counts under the notebook workload. As static wear leveling was instable, the belt of erase counts gradually grew thicker during experiments. A closer inspection of the static wear leveling's results revealed two causes of this instability.

Static wear leveling proactively moves static data away from physical blocks with a low erase recency (called static blocks hereafter), giving static blocks a chance to participate in garbage collection. Erasing a static (physical) block forcibly re-maps the logical block previously mapped to this static block to a spare block. However, static wear leveling conducts this re-mapping regardless of whether the spare block is also static or not. Under the notebook workload, there was a 70% probability that static wear leveling would re-map a logical block of a low update recency from a static block to another static block. This impeded the aging of static blocks only. The second problem is that static wear leveling erases static blocks regardless of their (absolute) erase counts. Under the notebook workload, there was a 50% probability that the block erased by static wear leveling was an elder block. Erasing an elder block does not help wear leveling in any way.

6. An SSD Implementation

6.1 Hardware Architecture

This study reports the implementation of the lazy wear-leveling algorithm in a real solid-state disk. This implementation used Global UniChip Cooperation's GP5086 system-on-a-chip (i.e., SoC) con-

	No WL	Lazy WL	Ratios
Average write IOPS	390	380	-3%
Erase counts			
standard deviation	613	11	-98%
mean	733	751	+2%

Table 2. Evaluation results of the GP5086-based SSD prototype. The average size of write requests was 22 KB.

troller for solid-state disks. The controller includes an 150-MHz ARM7 core, a BCH-based ECC engine, SLC/MLC flash interfaces, and host interfaces including serial ATA and parallel ATA. This controller supports 128KB of embedded SRAM for run-time variables and FTL mapping tables. GP5086 features a four-channel architecture aiming at high sustained data transfer rates. GP5086 erases in terms of four parallel flash blocks in the four channels, while reading and writing do not necessarily involve all the channels. We designed a solid-state disk using GP5086 and four MLC flash chips, with one chip for each channel. The effective page size and block size were 4KB and 2MB, respectively. The GP5086 firmware implemented a SAST-like FTL algorithm optimized for its multichannel architecture. This firmware also included the lazy wear-leveling algorithm for performance evaluation.

6.2 Experimental Results

In this experiment, the over-provisioning ratio was 2.5%, and the threshold parameter Δ was 16. The solid-state disk was connected to a Windows-based PC. A user application ran on this PC and replayed the notebook disk workload one hundred times on the solid-state disk using non-buffered Win32 I/O APIs. To speed up the experiment, the GP5086 firmware replaced its flash-accessing routines with dummy functions.

The results in Table 2 show that enabling lazy wear leveling significantly reduced the standard deviation from 613 to 11, while the mean increase was only 2%. These numbers are consistent with the simulation results. We also measured the time overhead in terms of the average number of write requests completed per second (i.e., the average write IOPS). When measuring IOPS, the firmware switched back to real flash-access routines and the experiment measured the response times of one million write requests. Results show that enabling lazy wear leveling decreased the write IOPS by 3%, which is slightly greater than the 2% mean increase. This is because wear leveling involves extra copy operations in addition to erasing blocks.

7. Conclusion

Successful wear leveling relies on monitoring not only the current wear in flash, but also recent trends in flash wear. Thus, keeping track of blocks' erase frequency (i.e., erase counts) and erase recency is a fundamental design issue. This study presents a simple but effective wear-leveling design called lazy wear leveling. This approach does not require any extra data structures for storing erase counts in RAM. Instead, it borrows the mapping information from the sector-translating algorithm to seek out data that has not been updated recently, and utilizes only in-flash erase counts to identify worn blocks. The timely re-mapping of these data to worn blocks helps even out flash wear.

Lazy wear leveling subjects wear evenness to a threshold variable. This study shows the feasibility of on-line overhead estimating using an analytical overhead model. Based on these estimations, lazy wear leveling can tune the threshold variable for appropriate balance between overhead and wear evenness. A series of trace-driven simulations show the merits of lazy wear leveling, and a prototype proves the applicability of lazy wear leveling in real solid-state disks.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for SSD performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70. USENIX Association, 2008.

[2] L.-P. Chang and C.-D. Du. Design and implementation of an efficient wear-leveling algorithm for solid-state-disk microcontrollers. *ACM Trans. Des. Autom. Electron. Syst.*, 15(1):1–36, 2009.

[3] L.-P. Chang and T.-W. Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *ACM Transactions on Storage*, 1(4):381–418, 2005.

[4] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. on Embedded Computing Sys.*, 3(4):837–863, 2004.

[5] Y.-H. Chang, J.-W. Hsieh, and T.-W. Kuo. Improving flash wear-leveling by proactively moving static data. *IEEE Transactions on Computers*, 59(1):53 –65, jan. 2010.

[6] M.-L. Chiang, P. C. H. Lee, and R. chuan Chang. Using data clustering to improve cleaning performance for flash memory. *Software Practice and Experience*, 29(3):267–290, 1999.

[7] A. Gupta, Y. Kim, and B. Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 229–240. ACM, 2009.

[8] IEEE Standards Association. IEEE Std 1666-2005 IEEE Standard SystemC Language Reference Manual. pages 1–423, 2006.

[9] D. Jung, Y.-H. Chae, H. Jo, J.-S. Kim, and J. Lee. A group-based wear-leveling algorithm for large-capacity flash memory storage systems. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 160–164. ACM, 2007.

[10] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee. A multi-channel architecture for high-performance NAND flash-based storage system. *J. Syst. Archit.*, 53(9):644–658, 2007.

[11] H.-J. Kim and S.-G. Lee. An effective flash memory manager for reliable flash memory space management. *IEICE Transactions on Information and System.*, 85(6):950–964, 2002.

[12] 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.

[13] *Wear-Leveling Techniques in NAND Flash Devices*. Micron® Application Note (TN-29-42), 2008.

[14] *Wear Leveling in Single Level Cell NAND Flash Memories*. Numonyx® Application Note (AN1822), 2006.

[15] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–23, 2008.

[16] K. Rosen. *Discrete mathematics and its applications*. McGraw-Hill New York, 2003. ISBN 0072424346.

[17] *K9F8G08B0M 1Gb * 8 Bit SLC NAND Flash Memory Data Sheet*. Samsung Electronics Company, 2006.

[18] *K9MDG08U5M 4G * 8 Bit MLC NAND Flash Memory Data Sheet*. Samsung Electronics Company, 2008.

[19] Y. J. Seong, E. H. Nam, J. H. Yoon, H. Kim, J.-Y. Choi, S. Lee, Y. H. Bae, J. Lee, Y. Cho, and S. L. Min. Hydra: A block-mapped parallel flash memory solid-state disk architecture. *IEEE Transactions on Computers*, 59:905–921, 2010.

[20] *Wear Leveling*. Spansion® Application Note (AN01), 2008.

[21] *SmartMedia® Specification*. SSFDC Forum, 1999.

[22] 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.

國科會補助計畫衍生研發成果推廣資料表

日期:2011/10/31

<table>
<tr><td rowspan="3">國科會補助計畫</td><td>計畫名稱: 子計畫四:嵌入式網路通訊裝置儲存裝置效能評比基準與工具之研發(中心分項)(2/2)</td></tr>
<tr><td>計畫主持人: 張立平</td></tr>
<tr><td>計畫編號: 99-2220-E-009-047- 學門領域: 自由軟體暨嵌入式系統</td></tr>
<tr><td colspan="2">無研發成果推廣資料</td></tr>
</table>

99 年度專題研究計畫研究成果彙整表

計畫主持人：張立平	計畫編號：99-2220-E-009-047-

計畫名稱：嵌入式網路通訊裝置評比技術與工具之研發--子計畫四:嵌入式網路通訊裝置儲存裝置效能評比基準與工具之研發(中心分項)(2/2)

成果項目			量化			單位	備註（質化說明：如數個計畫共同成果、成果列為該期刊之封面故事...等）
			實際已達成數（被接受或已發表）	預期總達成數(含實際已達成數)	本計畫實際貢獻百分比		
國內	論文著作	期刊論文	0	0	100%	篇	
		研究報告/技術報告	0	0	100%		
		研討會論文	0	0	100%		
		專書	0	0	100%		
	專利	申請中件數	0	0	100%	件	
		已獲得件數	0	0	100%		
	技術移轉	件數	0	0	100%	件	
		權利金	0	0	100%	千元	
	參與計畫人力（本國籍）	碩士生	0	0	100%	人次	
		博士生	0	0	100%		
		博士後研究員	0	0	100%		
		專任助理	0	0	100%		
國外	論文著作	期刊論文	1	0	0%	篇	
		研究報告/技術報告	0	0	100%		
		研討會論文	2	2	100%		
		專書	0	0	100%	章/本	
	專利	申請中件數	0	0	100%	件	
		已獲得件數	0	0	100%		
	技術移轉	件數	0	0	100%	件	
		權利金	0	0	100%	千元	
	參與計畫人力（外國籍）	碩士生	4	4	100%	人次	
		博士生	0	0	100%		
		博士後研究員	0	0	100%		
		專任助理	0	0	100%		

其他成果 (無法以量化表達之成果如辦理學術活動、獲得獎項、重要國際合作、研究成果國際影響力及其他協助產業技術發展之具體效益事項等，請以文字敘述填列。)	期刊論文部分，目前仍在撰寫中。

	成果項目	量化	名稱或內容性質簡述
科教處計畫加填項目	測驗工具(含質性與量性)	0	
	課程/模組	0	
	電腦及網路系統或工具	0	
	教材	0	
	舉辦之活動/競賽	0	
	研討會/工作坊	0	
	電子報、網站	0	
	計畫成果推廣之參與（閱聽）人數	0	

國科會補助專題研究計畫成果報告自評表

請就研究內容與原計畫相符程度、達成預期目標情況、研究成果之學術或應用價值（簡要敘述成果所代表之意義、價值、影響或進一步發展之可能性）、是否適合在學術期刊發表或申請專利、主要發現或其他有關價值等，作一綜合評估。

1. 請就研究內容與原計畫相符程度、達成預期目標情況作一綜合評估

■達成目標

□未達成目標（請說明，以 100 字為限）

□實驗失敗

□因故實驗中斷

□其他原因

說明：

2. 研究成果在學術期刊發表或申請專利等情形：

論文：■已發表 □未發表之文稿 □撰寫中 □無

專利：□已獲得 □申請中 ■無

技轉：□已技轉 □洽談中 ■無

其他：（以 100 字為限）

已經發表會議論文兩篇（IWSSPS 2010, CPSNA 2011，如附件），並被邀請投稿至 IEEE embedded systems letter 的一個 special issue

3. 請依學術成就、技術創新、社會影響等方面，評估研究成果之學術或應用價值（簡要敘述成果所代表之意義、價值、影響或進一步發展之可能性）（以 500 字為限）

本計劃成果為固態硬碟的虛擬平台。原則上我們透過產學合作的管道推廣至業界使用，目前廠商的回應都相當不錯。而學術研究方面，基於這個虛擬平台，我們目前得以研究開發新的儲存裝置與主機端的溝通方式，藉以達成更好的效能改善。