基于片内SRAM的固态硬盘转换层设计

谢长生[1] 李 博[1] 陆 晨[2] 王 芬[1]
(华中科技大学计算机学院 武汉光电国家实验室 武汉 430074)[1]
(辛辛那提大学电子工程系 美国辛辛那提 45221)[2]

摘 要 SSD逐渐成为了存储业界研究的热点。提出基于片内SRAM的flash转换层设计——SBAST,通过SRAM缓存更新的页提高了SSD随机写的效率,并减少了不必要的擦除操作。通过SSDsim的仿真实验,论证了该设计的有效性,给出了后续的计划。

关键词 固态存储器,Flash转换层,擦除操作,随机写

On-chip SRAM-based FTL Design for Solid-state Drive

XIE Chang-sheng[1] LI Bo[1] LU Chen[2] WANG Fen[1]
(National Laboratory for Optoelectronics, College of Computer Seicne and Technlogy, Huazhong University of Science & Technology, Wuhan 430074, China)[1]
(Department of Electrical and Computer Engineering, University of Cincinnati, Cincinnati 45221, USA)[2]

Abstract There are a lot of issues of the SSD arising in the storage industry. This paper proposed a caching FTL design to improve the efficiency of the SSD random write and reduce the amount of erasure operations by on-chip SRAM. By simulation experiments of SSDsim, we demonstrated the efficiency of our design. At the end of this paper, I gave my follow-up plan in this work.

Keywords Solid-state drive, FTL, Erasure operations, Random writes

存储界在不断的技术更新引领下,也尝试着改朝换代。对于传统磁盘式存储器的主导地位,固态硬盘的异军突起成为了存储业界的一个亮点和生力军。

1 固态硬盘的简介

固态硬盘SSD是一类通过固态存储单元构建的存储设备,常规情况下以NAND flash模式为主[1],也有采用SRAM或DRAM的。就传统的存储设备而言,SSD减少了可以移动的机械部件,使其拥有体积小、使用故障率低、功耗小等优势,大量应用到便携式的电子设备中。同时其寻道时间接近0,使得SSD能够提供更好的带宽和I/O特性。表1给出了不同存储设备的性能比较。

表1 多种存储介质的性能比较

Feature	Disk	DRAM	Low Power SRAM	Flash
Read Access	8.3ms	60ns	85ns	85ns
Write Access	8.3ms	60ns	85ns	4~10ms
Cost/MB	$1.00	$35.00	$120.00	$30.00
DataRetention Current/GB	0A	1A	2mA	0A

在表1中,DRAM,SRAM和flash在读写性能上都优于传统的磁盘。SRAM功耗低,但是其价格昂贵,同时需要额外的电力来维持数据;DRAM有卓越的读写能力,其价格也是可以接受的,同样也需要额外的电力来维持数据。虽然我们可以通过DRAM或SRAM同磁盘复合,把其上的数据最终转移至磁盘,但这种设计必然增加实现的复杂性,减小了系统的灵活性[2]。最后可以看出flash综合诸多因素的考虑下成为我们最好的选择。

1.1 固态硬盘的组成

SSD通常的内部构成包含一个外部接口逻辑(Host interface Logic)、SSD控制器、片内RAM和flash packages阵列。图1是SSD一个大致的框图[3]。

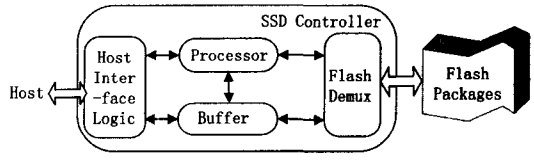

图1 SSD的结构框图

从图中可以知道,外部接口逻辑(Host Interface Logic)主要负责向主机提供可用的连接接口如USB、光纤接口、PCI Express或SATA等。处于SSD中央部分的处理器(Processor)主要用来执行主机提交的指令,同时在其上可通过实现多种优化来适应不同的负载需求。片内Buffer,对实际的存储进行缓冲,这里也是我们算法应用的场所。Flash阵列是SSD实际存储的物理部件,它通过多路复用设备(Flash Demux)发送给处理器和缓冲管理器(Buffer),SSD最终的数据也存放于此。

1.2 SSD读写特性

到稿日期:2009-09-14 返修日期:2009-11-15 本文受国家自然科学基金项目(60933002),863课题(2009AA01A402)和留学生基金委资助。
李 博(1979—),男,博士生,主要研究方向为存储系统、存储系统负载和SSD等,E-mail: libo.ch@gmail.com。

SSD 内部由 flash 阵列构成，图 2 给出了三星 K9XXG08UXM 固态硬盘的 flash 存储单元(Package)的内部结构。

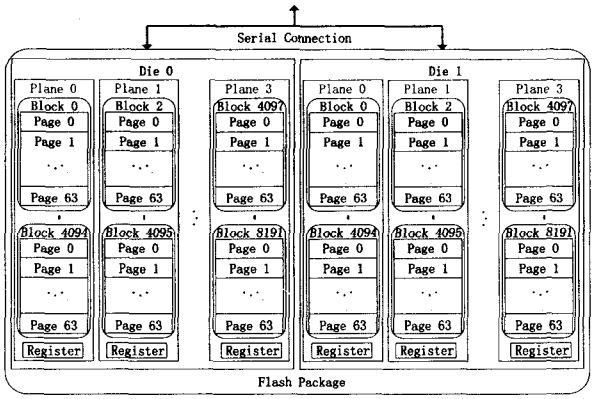

图 2 Flash Package 存储单元

存储单元的容量是 4GB，由两个 dies 组成。每个 die 包含 4 个 planes，每个 plane 包含 2048 个块，而每个块包含 64 个页。在图的上方是一个串行链接负责内部 flash 阵列和外部控制器的数据传输。在每个 plane 内有一个寄存器负责传输响应读写请求的数据。当外部提交请求时，数据首先在 plane 中从页到寄存器进行传输。然后通过外部的串行连接把数据从寄存器传送到外部的控制器。因此当多个 planes 需要传递数据时，就会在串行连接那里形成排队等待。而对于写请求，如果页曾经写入过，那么需要在写之前进行擦除，其写操作的时间就需要加上擦除处理的时间。文献[4]归纳了 SSD 如下读写特征：

1)没有寻道时间，且读取时间相对固定；

2)SSD 的读取性能和传输大小呈正比；

3)数据更新需要先擦后写；

4)擦除块的大小和随机写性能成反比；

5)读写中随机写的比例和 SSD 整体性能成反比。

对于最后一个特征，文献[4]给出了他们的实验结果，如表 2 所列。

表 2 读写比例和 SSD 整体性能的关系

% Writes	Total IOPS	Performance vs 15K SAS Hard Drive
0%	5400	20x Better
5%	252	1.25x Better
10%	130	1.5x Worse
20%	65	3x Worse
50%	26	8x Worse
100%	13	16x Worse

实验的 trace 包含 4000 个读写操作，磁盘对象为一个 SanDisk SATA5000 驱动器。从表 2 可以看出，如果全是读操作，SSD 要比磁盘快 20x；如果全是随机写操作，SSD 要比磁盘慢 16x；如果操作中读写各占一半，其整体性能比磁盘慢 8x。只有在随机写占 10%的情况下，SSD 的整体性能才略微落后于磁盘。

1.3 SSD 的生命期

SSD 的 flash 存储单元内其擦除的次数是有限的，通常是 10 万次左右。然而计算机文件系统的某些文件(如文件分配表)在正常使用中写入次数就可能超出这一极限。同时存储单元一旦擦除失效，其存储空间将无法使用，数据的恢复能力也带有不可逆性，这将大大缩短 SSD 的整体生命期。因此如何有效地延长其整体的生命期成为这方面研究一个亟待解决的问题。

本文首先介绍了 SSD 现在的一些研究热点和相关研究，引出在地址映射方面的几个相关算法，给出算法的初衷。第 4 节具体提到的算法，并对设计中的关键技术进行详细讨论。通过第 5 节的实验结果来验证该算法的可行性。

2 固态硬盘的相关研究

近年来业界涌现出有许多 SSD 方面的研究热点。其中的一些类似于传统磁盘的设计，而另一些关注于 SSD 的系统结构和特性。所有这些设计的不同点使我们有更多的选择来实践 SSD，从系统级的角度，列出主流的热点。

1)写请求的顺序和相应数据的布局。对于 SSD 而言其瓶颈主要是小写和随机写操作。大量的顺序写在合理分布的数据情况下性能要比小写和随机写好很多。因此写的性能和负载的特性紧密相关。

2)并行性，控制器和存储部分可以进行并行操纵来提高效率。例如，在 flash 存储单元内，不同 plane 的寄存器可以并行地执行读写操作。当一个寄存器正在传输数据到该寄存器所在 plane 的某一页面时，其它 plane 内的寄存器可以同时接受来自串行连接的数据。这种并行操作的结果使得不同的操作彼此交叠，优化了 flash package 的整体性能。同样，可以把多个 flash package 分簇，同一簇内的 package 可以协同完成一个多页请求。但是这一并行的代价是必然要在控制器部件内增加更为复杂的控制逻辑，其实现的难度也极大提升[3]。

3)关于 NAND flash 的生命期问题。每个 flash 单元都有固定的擦除次数，存储单元一旦失效，对存储本身是致命的。因此如何延长 SSD 的整体生命期，如何避免不必要的擦除次数，如何均匀地调整擦除分布，如何做到 flash 单元预警等都将是业界不得不考虑的问题。

目前有多种方法在逻辑层实现了 SSD 生命期的延长。平均抹写储存区块技术(Weal leveling)就是其中之一。它的思路是均匀地在整个 flash 单元上分布负载，同时兼顾垃圾回收机制的平衡性。文献[5]提到了清除门限策略(cleaning threshold strategies)。另一方法 quite straightforward 用来直接减少 SSD 的擦除时间。缩短擦除的时间显然可以有效地提供数据更新的效率。我们也试图从这方面入手来做些工作，通过片内 SRAM 来做缓冲，提高随机写效率并减少擦除次数。

3 课题背景

3.1 Flash 转换层

写请求需要考虑两种情况，如果写请求写入的位置空闲，SSD 可以及时写入。但通常的情况下，该位置曾经使用过，那么在完成写入之前，SSD 必须先执行擦除才能后续地写入。这一个过程必然耗费相当多的时间，同时擦除处理是以块为单位进行的，而写是以页为单位进行的[6,7]。最坏的结果是文件系统需要更新一页的内容，而白白擦除了块内的其他页。为此，研究人员定义了一个中间层——flash 转换层(FTL)。当需要进行更新时，它能够保证应用层用户及时透明地把数据定位到一个空闲的页面完成写入[8,9]。

3.2 FTL 地址映射的粒度

对于 FTL 而言，如何快速定位一个页面成为其中一个关键技术。这一位置转换过程分为两个模式：粗粒度和细粒度。

细粒度的映射以页为单位，如图 3 所示[10]。

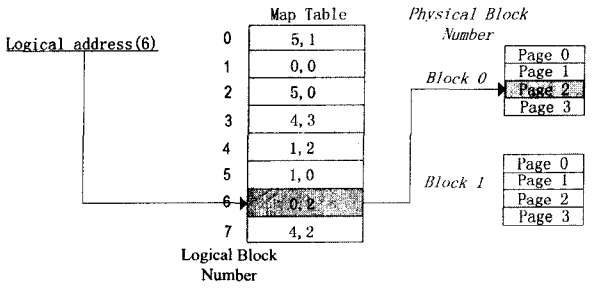

图 3 细粒度的页级映射

由逻辑地址 6 通过 SRAM 上的映射表可以找到对应的内容(0,2)。前者是块号，后者为块内偏移地址。这一翻译过程产生的结果是找到了对应的页，但细粒度的实现是以增加 SRAM 的开销为代价的。例如一个 16MB 的 flash 存储设备，其内页为 512bytes，将需要 64kB 的 SRAM 来存放映射表。其映射的范围同映射表的大小呈线性比例增长[7]，当 flash 的容量为 1GB 时，其映射表的大小将达到 4MB。

粗粒度的映射以块为单位，其过程如图 4[10]所示。

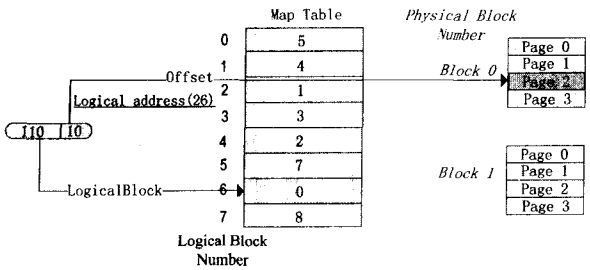

图 4 粗粒度的块级映射

粗粒度机制会把逻辑地址分为两个部分，前者是逻辑的块地址，后者为块内偏移地址。图中逻辑地址 26 拆分成了两部分，110 和 10。前者 110 为逻辑块地址，10 为块内偏移地址。通过 SRAM 上的映射表可以知道 110 对应于物理块 0，通过偏移量 2 可以找到其最终的物理页，完成寻址过程。这一点有些类似于虚拟页地址转换机制[11]。该算法以牺牲映射的灵活性为代价，但是节省了映射表的大小。例如一个 16MB 的 flash 系统只需要 2kB 的 SARM 就能满足，相较细粒度其映射大大节省了 SRAM 资源。这也是容量较大的 SSD 设备通常采用比较大的块的原因[7]。

3.3 BAST 算法

FTL 的另一个作用是可以通过优化算法提高读写效率，减少擦除处理的次数。为此文献[7,12]给出一个置换块算法(replacement block scheme)，但其算法的执行开销过大而不尽如人意。

BAST(block associative sector translation)[7,13]由 Jesung Kim 提出，主要为了解决小写操作和大量的顺序写操作问题。BAST 中的块操作分为两个级别，综合了粗粒度和细粒度的优点，绝大多数的块以块级别进行管理，少数的块以页级别进行管理。在块级别的块命名为数据块，而在页级别的称之为 log 块。log 块主要用于更新操作，若需要更新数据块中的一个页面，系统需要从空闲的块中指派一个 log 块来对应这个数据块。图 5 给出了 BAST 算法的地址映射[10]。

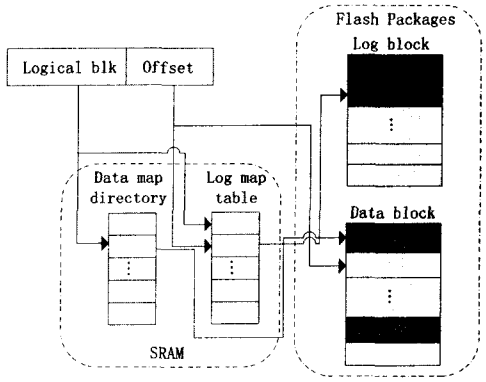

图 5 BAST 的地址映射

BAST 针对不同块采用的映射表是不同的，log 块需要检索 log 地址映射表，而数据块需要检索映射目录。当 FTL 得到上层文件系统发送的逻辑地址时，需要对其进行拆分，前者为逻辑块号，后者为偏移地址。算法 BAST 先判断其需要映射的块是数据块还是 log 块，查找相应的映射表，如果是数据块，物理地址就是检索目录对应的块号加上偏移地址；如果是 log 块，那么通过逻辑块号在 log 地址映射表中找到起始位置加上偏移地址在映射表上找到对应 log 块页地址。BAST 方式实现了更新数据的灵活性，可以见缝插针地把更新数据及时写入空闲的位置中。但是该算法将频于更新的 log 块也保存在 flash 中，这样其存储方式也同样引发更多的写入和擦除操作，其开销因此会更大。

下面总结 BAST 算法的要点：

1)由于综合了粗细粒度匹配算法的优点，BAST 可以更为灵活地更新相应的页面；

2)BAST 相较传统的匹配算法有好的空间利用率，这点在文献[7]进行了试验比较；

3)合并操作类似于 log 结构文件系统采用的 cleaning 机制[14]，其不同在于 BAST 中 log 块内的页面只能对应于相应数据块内的页面；

4)频于更新的 log 块存放在 flash 阵列中，在运行时可能会引发更多的写入和擦除操作。

4 基于 SRAM 的 BAST 机制——SBAST

针对 BAST 算法的不足和可能带来的问题，我们考虑是否将 log 块移入 SRAM 的缓存中。如图 1 所示，我们注意到在现有的 flash 存储设备中，SRAM 已经内置其中，用以保存映射表。我们可以在原有的 SRAM 空间中划出一部分空间来存放 log 块的页面，这样 log 块的更新就可以在 Buffer 中进行，不需要进行擦除处理。虽然这增加了一定的实现难度，但这种代价是值得的。下面给出我们的设计方案。

从图 6 可知，SBAST 把数据的更新页存放在两个地方：SRAM 内的更新链表和 flash 存储单元的 log 块。SRAM 内的更新链表按照 LRU 方式进行排列，为了确定更新的具体存放位置，我们在 log 映射表中增加了一个“place”位。如果该位置位，那么更新页面在 SRAM 的链表内；如果该位未置，那么更新页面在 flash 内的 log 块中。

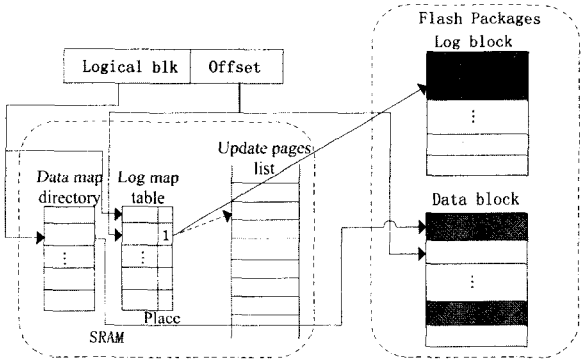

图 6　SBAST 的地址映射

4.1　SBAST 的更新过程

若当数据块内的数据需要更新，则把更新的数据传输至 SRAM 内更新链表。当链表满的时候，需要在链表中置换出一个更新页。那么从 Flash 中确定一个 log 块来匹配置换页所对应的数据块，对 log 块和数据块进行绑定。当且仅当该数据块的更新页再次由 SRAM 的更新链表中置换出，该更新页将追加至对应的 log 块内。

图 7 描述了 log 块、SRAM 更新链表和数据块其内部数据的布局。最右端是数据块，若数据块内页 1 需要更新，则 SRAM 更新链表内的页 1 只需在其位置上更新即可。而当更新页面 1 从 SRAM 更新链表置换出之后，需要写入 log 块页 1 的位置。当且仅当，链表内的页 1 再次置换出，把 log 块中的页 1 置为“不可以用”，然后把新的页 1 追加到 log 块中，这时 log 块中的第 2 页就成为对数据块页 1 的最新更新。

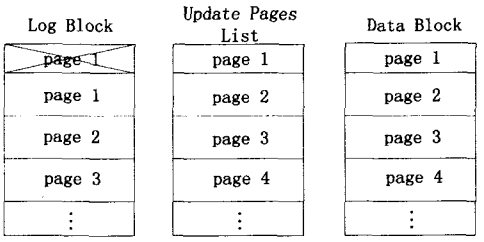

图 7　SBAST 3 类块中更新页的布局

图 7 中数据块页 2 的最新更新页在 log 块的第 3 个位置。由此可以看出 log 块内包含的只是绑定数据块的一系列更新页。

4.2　SBAST 的读取过程

由于加入了更新链表，SBAST 对数据的读取分为 3 种情况：

1）如果读出的数据页没有更新，那么算法根据块级地址映射方式找到数据块内相应的数据页；

2）如果读出的数据页有更新，那么需要通过 log 映射表进行映射。当表内该页 place 为已置，那么更新的数据在 SRAM 内；

3）如果读出的数据页有更新，那么需要通过 log 映射表进行映射。当表内该页 place 为未置，那么更新数据在数据页对应的 log 块内。

从图 6 可以了解这一过程。

4.3　SBAST 的合并过程

若数据块内数据不断更新，使其对应 log 块已满，这时就需要算法进行合并操作，同时生成新的数据块。图 8 给出了这一过程。

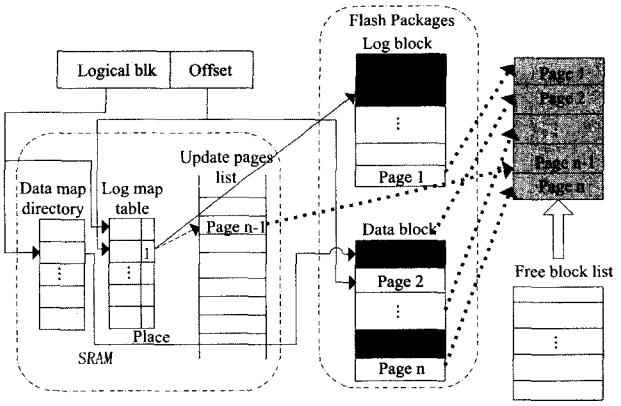

图 8　SBAST 的合并过程

在图 8 中，数据块第 1 页和第 $n-1$ 页已经更新。若更新的页 1 从 SRAM 上的更新链表置换出被写入页 1 对应的 log 块，这时 log 块内已满，则需要合并操作腾出空间。如果页在 SRAM、log 块和数据块内都存在，优先复制 SRAM 内该页的内容；如果该页只在 log 块和数据块中，优先复制 log 块内该页的内容；如果页只在数据块中就直接复制数据块内的内容。图中需要从 SRAM 中复制页 $n-1$，复制 log 块内页 1 和数据块中其余页合并成为一个新的数据块写入 Free 块表中指定的位置。当写入过程完成后，数据块和 log 块内的相应空间被标注，后台将执行擦除操作以回收空间。

5　仿真实验和结果分析

5.1　仿真实验环境

我们的实验利用了 SSDsim[3] 仿真环境，它继承了原来 Disksim[15,16] 的主体框架和构件。具体的 SSD 模拟三星 K9XXG08UXM 固态硬盘，利用 HP 实验室[17] 提供的 snake 和 cello trace 来比较 3 类设计的性能：置换块方案，BAST 复合和我们的 SBAST 算法。两种 trace 分别代表了科学研究和普通用户的日常事务。

5.2　仿真结果和分析

下面的图给出的 snake 和 cello trace 其时间跨度均为一个月，具体分析主要集中在擦除次数、合并次数和平均写响应时间。前者其仿真的 SSD 容量为 4GB，内置 SRAM 为 1.25 MB；后者仿真的 SSD 容量为 8BG，其内置的 SRAM 为 25MB。两者的 SRAM 大小不一样，主要是因为 Cello trace 相对的离散度比较大。

5.2.1　SSD 擦除次数的比较

由图 9 和图 10 可以看出不管在 snake 还是 cello trace 下，SBAST 的擦除都是最小的。图 10 中其 cello 离散度比较大，三者之间的比较也更加明晰。

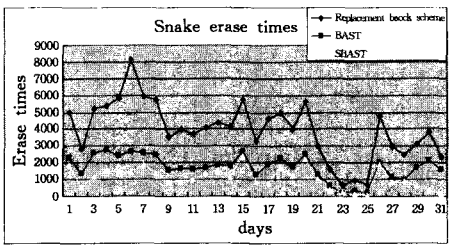

图 9　Snake trace 的擦除次数

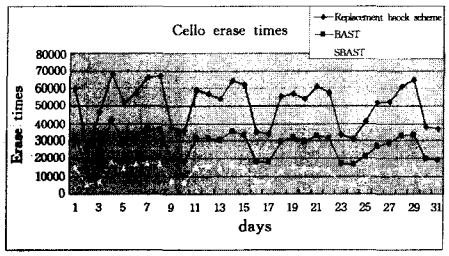

图 10 Cello trace 的擦除次数

5.2.2 SSD 合并次数的比较

合并操作的过程必然会附带擦除操作。其两者成正比关系，即擦除操作越多，合并操作发生的情况也越多。从图 11 和图 12 可知，需要合并操作最多的是置换块策略，其次是 BAST，最少的是我们的算法。两者 SBAST 的合并次数相比较，Snake 次数相对较低，这是因为其离散度的问题。离散度越低，SBAST 的更新页在 SRAM 中更新的可能性就大；反之对于 Cello 由于其离散度偏大，一旦更新页由 SRAM 中换出写至 log 块，其合并操作的可能性就增大。

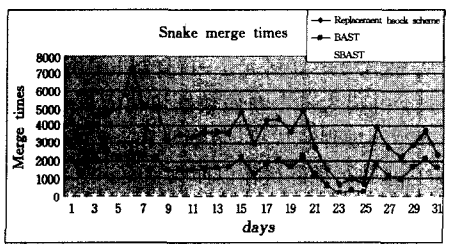

图 11 Snake trace 的合并次数

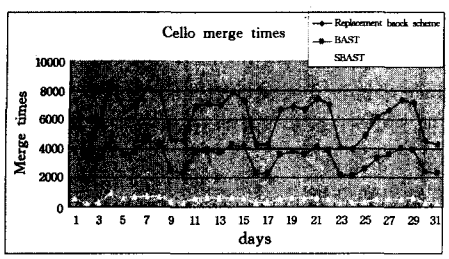

图 12 Cello trace 的合并次数

5.2.3 SSD 写平均响应时间的比较

对于写平均响应时间的比较，SBAST 也是最小的。从上述结果可以得出，我们的设计都取得了一定的优化效果。图 13 和图 14 分别给出了 Snake trace 和 Cello trace 写的平均响应时间。

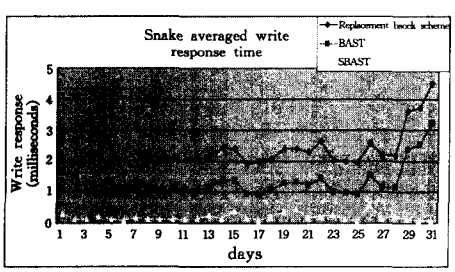

图 13 Snake trace 写的平均响应时间

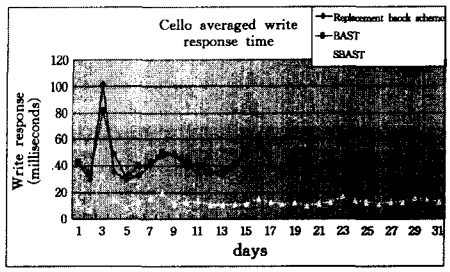

图 14 Cello trace 写的平均响应时间

结束语 本文针对 BAST 可能出现的问题，提出了利用片内 SRAM 来缓存部分的更新写——SBAST 策略。该算法有效地提高了 SSD 随机写效率，同时减少了 SSD 擦除次数。但是其更新的页面只能写入绑定的 log 块内，这可能损耗 SSD 的空间使用率。由此我们今后的工作力图从 SBAST 的空间利用率上作进一步的研究。

参 考 文 献

[1] Wikipedia. Solid-state drive[OL]. http://en.wikipedia.org/wiki/Flash_drive

[2] http://tech.blorge.com/Structure:%20/2008/05/27/samsung-256gb-solid-state-disk-may-change-the-face-of-technology-for-laptops-mp3-and-mobile-devices/

[3] Agrawal N, Prabhakaran V, Wobber T, et al. Design Tradeoffs for SSD Performance[C]//Proceedings of the USENIX Technical Conference. June 2008

[4] Dumitru D. Understanding Flash SSD Performance[OL]. www.storagesearch.com/easyco-flashperformance-art.pdf

[5] Kgil T, Roberts D, Mudge T. Improving NAND Flash Based Disk Caches[C]//Proceedings of the 35th International Symposium on Computer Architecture. 2008:327-338

[6] CompactFlash association. Information about CompactFlash [OL]. http://www.ssfdc.or.jp/

[7] Kim J, Kim J M, Noh S H, et al. A Space Efficient Flash Translation Layer for CompactFlash Systems[J]. IEEE Transactions on Consumer Electronics, 2002, 48:366-375

[8] Intel corporation. understanding the flash translation layer (FTL) specification[OL]. http://developer.intel.com/

[9] MTD. Memory Technology Device (MTD) subsystem for Linux[OL]. http://www.linux-mtd.infradead.org

[10] A Space-Efficient Flash Transition Layer for Compact Flash systems[C]//CS710: Topics in Computer Architecture Embedded Systems. Dept. of Computer Science, KAIST. 2004

[11] Silberschatz A, Galvin P B, Gagne G. Operating System Concepts, 6th ed[M]. John Wiley & Sons, Inc., 2003:330-344

[12] Ban A. Flash File System[P]. United States Patent, 1995, 5: 404-485

[13] Lee S W, Park D J, Chung T S, et al. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation [J]. ACM *Transactions on Embedded Computing Systems*, 2007, 6(3)

[14] Rosenblum M, Ousterhout J. The Design and Implementation of a Log-structured File System[J]. ACM Transactions on Computer Systems, 1992, 10(1):26-52

[15] Ganger G, Worthington B, Patt Y. The disksim simulation environment version 4.0 reference manual[R]. CMU-PDL-08-101. Carnegie Mellon University, May 2008

[16] IOzone. org. IOzone Filesystem Benchmark[OL]. http://www.iozone.org

[17] Ruemmler C, Wilkes J. UNIX Disk Access Patterns[C]//Proceedings of USENIX Winter 1993 Technical Conference. 1993