Multi-bank 闪存文件系统的一种 I/O 调度机制

赵 培 李国徽
(华中科技大学计算机学院 武汉 430074)

摘 要 闪存以其体积小、抗震性强、能耗低、读取速度快等特点,被广泛应用于存储系统中。NOOP是闪存上传统的调度方法,但是NOOP的I/O性能较低,不能满足很多应用程序的要求。根据闪存读取速度快、多个banks(chips)可以并行运行等特点,提出了一种基于闪存文件系统YAFFS的Multi-bank闪存调度方法(简称MBS)。MBS并行地执行请求,且给予读请求更高的优先级。MBS根据AVL-based-tree机制识别出的写请求属性动态地将其分配到合适的bank中。实验结果表明,相比NOOP,MBS调度具有更高的I/O吞吐率、更短的请求响应时间并具有均匀的bank擦除次数和利用率。

关键词 Multi-bank闪存,I/O调度,native闪存文件系统,bank动态分配策略,请求属性的识别
中图法分类号 TP316 **文献标识码** A

Efficient I/O Scheduler over Multi-bank Flash Memory File Systems

ZHAO Pei LI Guo-hui
(College of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract Flash memory has been widely used in storage systems because of its nonvolatile nature, its small size, shock resistance and fast access speed. The traditional scheduler over flash memory storage systems is NOOP scheduler. There is much room for improving the I/O performance, especially over multi-bank flash memory storage systems. Because several banks can operate simultaneously, we proposed a new scheduler called Multi-Bank flash memory Scheduler (MBS) based on the native file system YAFFS to take advantage of the parallelism of multiple banks by considering the high read speed. A flexible bank assignment policy was proposed to assign proper banks for write requests according to the attributes(hot or cold) of requests, which were identified by an AVL-based-tree mechanism. MBS scheduler reorders read and write requests and gives higher priority to reads. The experimental results show that the I/O throughputs and average response time are improved significantly compared with the NOOP scheduler. An even erasable count and capacity utilization were obtained between different banks in the multi-bank storage system.

Keywords Multi-bank flash memory, I/O scheduler, Native flash memory file system, Dynamic bank assignment policy, Identification of request attribute

1 引言

闪存(Flash Memory)以其抗震性强、功耗低、噪音小、体积小、随机读取速度快等优点,被广泛应用于嵌入式系统、实时系统、高性能服务器中[1]。尤其是NAND型闪存以其容量大、价格低等优点被广泛应用于二级存储设备中,已成为磁盘最好的替代品。然而闪存的存储和访问特性与磁盘有很大的不同[2],如重写前需要擦除(erase-before-write)、每个块的擦除次数有限等。因此,原来基于磁盘特性的设计不能直接应用到闪存存储系统中。

NOOP是存储系统中经典的I/O请求调度方式,它将所有的I/O请求简单地放入一个FIFO队列,不区分读写请求。而读请求的响应时间更能影响系统的性能和用户体验,且闪存中的读写操作的性能具有不对称性,其中写操作比读操作消耗更多的时间,而且在执行写操作的过程中可能会引入Garbage Collection[2],因此写请求的响应时间具有不可预期性。文献[3]表明,写请求中引入Garbage Collection以后,写吞吐量将下降50%。故闪存系统中NOOP调度的I/O性能低下,影响了系统的整体性能。另一方面,NOOP调度将写请求按照请求的地址空间分配到确定的bank(或chip)中,造成了不同bank擦除次数严重失衡,从而影响了闪存的使用寿命。

本文提出一种新的基于Native文件系统的Multi-bank闪存调度方法,简称MBS。MBS为读/写请求维护了不同的读/写FIFO队列。由于闪存的读取速度快,且读请求通常会阻塞进程的执行,因此MBS给予读请求更高的优先级。由于

到稿日期:2011-05-26 返修日期:2011-09-28 本文受国家自然科学基金(60873030),国家高技术研究发展计划(863)(2007AA01Z309),国家"十一五"国防预研基金资助。

赵 培(1982-),女,博士生,主要研究方向为基于闪存的存储系统,E-mail:iezhaopei0319@163.com;李国徽(1973-),男,博士,教授,博士生导师,主要研究方向为主动、实时、移动数据库系统理论及集成技术。

闪存内的多个 bank 可以同时并行地运行，因此多个读写请求可以同时被执行。本文提出了一种识别待写入数据(hot/cold)属性的机制，简称 AVL-based-tree 机制。MBS 根据 AVL-based-tree 机制识别的数据属性将写请求动态地分派到不同的 bank 上，使得不同的 bank 具有均匀的擦除次数和容量利用率(capacity utilization，即 bank 中 live 页面与总页面数量的比值)。

2 闪存特性及相关工作

按内部存储矩阵结构不同，闪存可分为 NOR 和 NAND 两种。NOR 型闪存可以按字节存取片内的每个字节数据，读取速度快，但存储密度低、擦除速度慢，因此常用于代码存储，如系统启动代码和内核映像等。相比 NOR 型闪存，NAND 型闪存存储器以高密度、大容量、高数据存储速率以及更快的擦除速度等特点，逐渐成为大容量嵌入式存储设备应用的主流[4]。

NAND 型闪存设备可能包含多个 bank(或 chip)，每个 bank 以块(block)的形式组织，每个块包含固定数量(一般为 32，64 或 128)的页(page)。块是最小的擦除单位，页是读写操作的最小单位，大小一般为 512B 或 2kB。另外闪存在重写前需要执行擦除操作(erase-before-write)[2]，即不支持 in-place 更新，因此闪存一般采用 out-place 更新方式，即将数据的新版本(即更新后的数据)写到闪存其他空闲(Free)的位置，然后将原来的数据标记为失效(Invalid)。闪存的 out-place 更新方式会造成 Live 数据和 Invalid 数据并存于闪存内。当闪存中空闲空间被耗尽后，系统将启动 Garbage Collection(简称 GC)操作来回收空间[2]。此外，由于闪存中每个擦除块具有一定的擦除次数，因此系统需要通过 Wear-leveling策略[2]尽可能地使闪存块具有均匀的擦除次数，以延长闪存的使用寿命。

Multi-bank 闪存系统中，多个 bank 共享 I/O 总线，但是每个 bank 都具有独立的内部寄存器，多个 bank 可以并行地运行。闪存的基本操作(读、写、擦除)包含两个阶段：Setup 阶段和 Busy 阶段。其中 Setup 阶段需要占用 I/O 总线来传输指令和数据，此阶段不能并行。而在 Busy 阶段，各个 bank 内部可以独立执行读、写、擦除操作，因此可以并行地运行。另外，Busy 阶段占用了写操作和擦除操作的大部分时间。表 1 列举了一种典型 NAND 型闪存的各种操作的 Setup 阶段和 Busy 阶段所消耗的时间。

表 1 NAND 型闪存性质(Samsung K9F6408U0A NAND flash)

操作	Setup 耗时(μs)/Busy 耗时(μs)
Read	27/25
Write	27/350
Erase	31/2500

计算机系统中有两种闪存存储系统的实现方式：一种是块设备模拟方式，通过添加闪存转译层[5-8](Flash Translation Layer，FTL)将闪存模拟成和磁盘具有相同接口的块设备。FTL 负责闪存中的存储管理、地址转换和均衡磨损(wear leveling)。用户通过 FTL 透明地访问闪存中的数据，原来的应用程序可以不加修改地运行在闪存系统上。这种方式的优点是代码具有可移植性，与磁盘具有兼容性，但是存储效率较低。另一种方法是 Native 文件系统直接管理闪存页和块，无需转换层的支持，如文件系统 JFFS/JFFS2[9,10]、YAFFS[11]等。此种方法的可移植性差，但是存储效率较高。

目前，研究者已提出很多提高闪存存储效率的方法，如 Wu 和 Zwaenepoel[12]提出的采用 SRAM 作为 Write buffer 及 Cost-Benefit 的 Garbage Collection 策略。Chang 和 Kuo[9]提出的 ASA 策略(Adaptive Striping Architecture)将一个写请求的操作分配到闪存的多个 bank 中，以提高写请求的性能。另外，很多研究者提出采用日志的方法减少细粒度更新所带来的闪存的写操作和擦除操作的次数，从而提高系统性能[13,14]，或建立针对闪存特性的索引结构，提高系统的索引能力[15,16]。

本文提出的 MBS 是一种基于 Multi-bank 闪存的调度机制。MBS 运行在 Native 文件系统 YAFFS 之上，YAFFS 为传统的应用程序提供标准的文件操作，使之可以透明地访问闪存中的数据。

3 I/O 请求队列

I/O 请求包含一组逻辑地址连续的 sector，起始地址为 sector(512Bytes)对齐，I/O 请求长度为 sector 大小的整数倍。逻辑空间中，sector 的地址为 LBA(Logical Block Address)。下文用一个三元组来表示一个 I/O 请求$\langle LBA, size, dir\rangle$，其中 LBA 表示 I/O 请求的起始地址(即首个 sector 的逻辑地址)；$size$ 表示 I/O 请求的 sector 个数；dir 表示请求为读请求还是写请求，$dir=0$ 表示 I/O 请求为读请求，$dir=1$ 表示 I/O 请求为写请求。

3.1 读请求队列

由于读请求的数据已经存储在闪存中，而且可能分布在一个或多个 bank 上，因此无需为读请求分配访问的 bank。若读请求的数据全部位于同一个 bank 内，下文称此请求为 single-bank 请求。若读请求的数据分布在多个 bank 上，则称之为 Multi-bank 请求。系统为 single-bank 请求维护一个 sFIFO_Table 表，每个表项(Table entry)指向属于同一 bank 的 FIFO 读请求队列。例如 sFIFO_Table[i]指向属于第 i 个 bank 的 FIFO 读请求队列。Multi-bank 请求存放在一个简单的 FIFO 队列中，下文称此队列为 mFIFO_list。所有的 Multi-bank 请求按照 FIFO 的顺序插入到 mFIFO_list 中。

当一个新的读请求到达时，首先为读请求分配一个有效期(或者 soft-deadline)，然后根据读请求所属的 bank，将其插入到对应的 FIFO 队列中。如果读请求是 single-bank 请求且属于第 i 个 bank，则按照 FIFO 的顺序将其插入到 sFIFO_Table[i]对应的 FIFO 队列中。若为 Multi-bank 请求，则按照 FIFO 顺序插入到 mFIFO_list 中。

3.2 写请求队列

如果一个 sector 或 LBA 经常被更新，则称之为 hot LBA，否则为 cold LBA。Hot LBA 和 cold LBA 的区分取决于其更新频率，而与读取频率无关。本文提出一种类似于二叉搜索树的方法(下文称 AVL-based-tree 方法)来识别写请求属性，即识别写请求为 hot 写请求、cold 写请求还是 neutrality 请求。

AVL-based-tree 中的每个结点代表一段逻辑地址连续的 sectors，结点数据结构为(LBA，Len，$Counter$)。LBA 为该结点表示的 sectors 的起始地址，也是结点的索引；Len 为 sector 的个数；$Counter$ 为一个具有 C 个 bits 的计数器 C-bit-counter。C-bit-counter 用来记录此结点包含的 $LBAs$ 在一段时间

内被更新的次数。例如结点(100,64,5)代表 *LBA* 100－163 在一段时间内被更新了 5 次。

令 A 代表一个新的写请求 *w_req* 的 *LBAs* 集合，B 代表一个结点 *TNode* 中连续 *LBAs* 的集合，如果 $A \cap B \neq \emptyset$，则称 *w_req* 与结点 *TNode* 相交。设与 *w_req* 相交的 m 个结点为 $N_1, N_2, \cdots, N_i, \cdots, N_m$，这 m 个结点对应的 *LBAs* 集合分别为 $B_1, B_2, \cdots, B_i, \cdots, B_m$，对应的 *Counter* 值分别为 $C_1, C_2, \cdots, C_i, \cdots, C_m$，那么写请求 *w_req* 包含的 *LBAs* 在一段时间内的平均更新次数 $Freq(w\_req)$ 可表示为 $\sum_{i=0}^{m}(|A \cap B_i| * C_i)/\sum_{i=0}^{m}|A \cap B_i|$。若 $Freq(w\_req)$ 大于阈值 UTH，则称 *w_req* 为 hot 请求；若小于阈值 LTH，则称 *w_req* 为 cold 请求；若 $LTH < Freq(w\_req) < UTH$，则称 *w_req* 为 neutrality 请求，其中 LTH 和 UTH 为系统预先设定的参数，并且 $LTH < UTH$。系统分别为 hot、cold 和 neutrality 请求维护一个简单的 FIFO 队列：*hot_list*、*cold_list* 和 *neutrality_list*。如果一个新到达的写请求为 hot/cold/neutrality 请求，那么 MBS 将此请求按照 FIFO 的顺序放入 *hot_list/cold_list/neutrality_list* 中。

AVL-based-tree 的查询、插入和删除操作类似于二叉搜索树。当写请求 *w_req* 到达时，首先按照二叉搜索树的查找方法在 AVL-based-tree 中查找所有与 *w_req* 相交的结点 $N_1, N_2, \cdots, N_i, \cdots, N_m$，若 $A \supseteq B_i$ 则称结点 N_i 是 *w_req* 完全覆盖的。反之，若 $A \cap B \neq \emptyset$，$B - A \neq \emptyset$，则称结点 N_i 是 *w_req* 部分覆盖的。

所有被 *w_req* 完全覆盖的结点的计数器(C-bit-counter)自增 1，表示此结点的 *LBAs* 被更新了一次。每个被 *w_req* 部分覆盖的结点按照如下的方法做出调整。结点 N_i 中没有被覆盖的部分为 $B_i - A$。首先将 $B_i - A$ 从结点 N_i 中删除，然后为其构造一个新结点，重新插入到 AVL-based-tree 中。原结点 N_i 的 *Counter* C_i 自增 1，新构造的结点 *Counter* 保持不变。若某个结点的 *Counter* 达到了最大值，或者自上次衰减以后积累的写请求数达到一个阈值 *Threshold*，则 *AVL-based-tree* 中所有结点的 *Counter* 值将衰减为原来的一半，即 *Counter* 的值向右移一位。下文称 *Threshold* 为衰减周期。

如图 1 所示，当一个写请求 *w_req*(*LBA*=100, *size*=32, *direction*=1) 到达时，与此请求相交的结点有 *A* 和 *C*。其中 *A* 是 *w_req* 完全覆盖的，因此只需将 *A* 的 *Counter* 增加 1 即可。结点 *C* 是 *w_req* 部分覆盖的。首先将结点 *C* 中未被 *w_req* 覆盖的部分(*LBA*=132, *size*=16, *Counter*=3)删除，并作为一个新结点(记为 *C*′)插入到 *AVL-tree* 中，*Counter* 的值与原结点 *C* 保持不变。结点 *C* 中被 *w_req* 覆盖的部分(记为 *C*′)的 *Counter* 值增加 1。

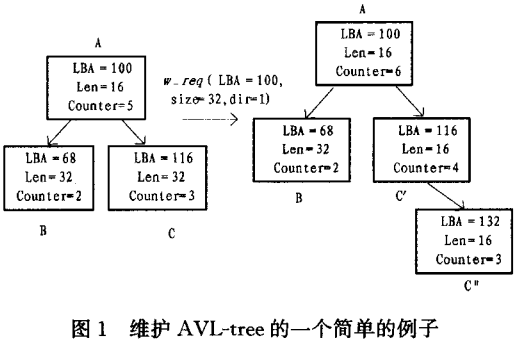

图 1　维护 AVL-tree 的一个简单的例子

AVL-based-tree 中的结点经过多次分裂后，树的高度会随之增加，致使查询效率降低以及 AVL-based-tree 占用 RAM 的空间不断增大。另一方面，AVL-based-tree 中的结点经过若干衰减周期后，*LBA* 连续的两个结点可能具有相同的 *Counter* 值，那么这些结点可以进行合并。AVL-based-tree 的这种合并操作减小了树的高度和占用的内存空间。

4　MBS 调度

闪存支持多个 bank 并行运行，这个特性为同时服务多个请求提供了依据，即 MBS 可以在一个请求返回前，调度其他的请求。MBS 交替调度一组读请求(设含有 *Read_Batch* 个读请求)和一组写请求(设含有 *Write_Batch* 个写请求)。下文称一组读请求和一组写请求的调度过程为一个调度周期。其中 *Write_Batch* 为预先设置的常数，而 *Read_Batch* 的取值依赖于每个调度周期开始时的读请求状态。

在一个调度周期内，MBS 首先调度 *Read_Batch* 个读请求，然后调度 *Write_Batch* 个写请求。MBS 给于读请求更高的优先级，并且 *Read_Batch* ≥ *Write_Batch*。具体地讲，在每个调度周期的开始，如果读请求队列中存在已过期(expired)的请求，则将此周期的读请求 *Read_Batch* 设置为 *Write_Batch* 的 *Write_Starved* 倍(即 *Read_Batch* = *Write_Starved* * *Write_Batch*)，其中 *Write_Starved* 为预先设置的常数。否则 *Read_Batch* = *Write_Batch*。如图 2 所示，设 *Write_Starved*=2，在第 i 个周期开始时没有过期的读请求，则 *Read_Batch*=*Write_Batch*。在第 i+1 个周期开始时存在过期的读请求，则 *Read_Batch*=2 * *Write_Batch*。

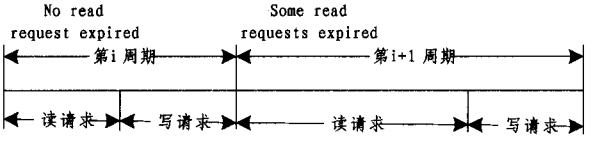

图 2　MBS 的调度周期

当一个请求返回(已完成的请求)时，会触发 MBS 调度一个新的请求，MBS 调度算法如算法 1 所示。设触发 MBS 调度算法的返回请求为 *Req_cpl*，与 *Req_cpl* 相关的所有 bank 的集合记为 α，$\alpha \subseteq \{bank_i \mid 0 \leqslant i < n\}$，$n$ 为闪存系统中 bank 的总数。若 *Req_cpl* 是 Multi-bank 读请求，则 $1 < |\alpha| \leqslant n$；若 *Req_cpl* 为 single-bank 读请求或者写请求，则 $|\alpha| = 1$。全局变量 *g_read* 和 *g_write* 表示一个周期内已完成的读请求和写请求的数量。算法 1MBS 调度中的变量的含义如表 2 所列。

表 2　MBS 调度算法中的变量定义

变量	定义
g_read	在一个调度周期内，目前已完成的读请求的个数
g_write	在一个调度周期内，目前已完成的写请求的个数
Req_cpl	触发 MBS 调度的请求
assBank(req)	与请求 req 相关的所有 bank 的集合
EC(B)	Bank B 的擦除次数(erase count)
CU(B)	Bank B 的容量利用率(capacity utilization)
Avg_EC	闪存内所有 bank 的平均擦除次数
Avg_CU	闪存内所有 bank 的平均容量利用率

由于 Multi_bank 读请求涉及到多个 bank，只有当 Multi_bank 读请求相关的 bank 均为 Idle 状态时，MBS 才服务此 Multi_bank 请求。因此，相比 single_bank 请求，MBS 给予 Multi_bank 请求更长的有效期(或 soft-deadline)。

与 single_bank 请求不同，在服务 Multi_bank 请求前需要对所有相关的 bank 进行加锁。而且，当一个 Multi_bank

请求对某个 bank 加锁后，请求队列中其他请求将不能访问该 bank 或者对该 bank 加锁。如果一个 Multi_bank 请求的相关 bank 被加锁，但是没有被执行，则称此 Multi_bank 请求为 waiting request。另一方面，如果系统中已经存在一个 waiting request，MBS 将不会为其他的 Multi_bank 请求加锁。因此系统中至多存在一个 waiting request，下文将可能存在的唯一 waiting request 记为 *wr_Req*。当所有被加锁的 bank 处于 Idle 状态时，MBS 将优先调度 *wr_Req*。另外，为了避免发生死锁现象(假设系统中不会出现执行不完的请求，即不会出现某个 bank 一直处于 Busy 状态)，一旦 *wr_Req* 被调度或执行(不管该请求是否会成功返回)，MBS 将解锁 *wr_Req* 所有相关的 bank。

在 MBS 调度一组读请求的过程中，如果 mFIFO_list 中存在过期的 Multi_bank 请求(超过了请求的 soft-deadline)，且系统中不存在 waiting request，则将 mFIFO_list 中最早的 Multi_bank 请求设置为 *wr_Req*，并对 *wr_Req* 相关的 banks (即 assBank(*wr_Req*)中的 banks)加锁。

不失一般性，设算法 1 在调度过程中读写请求队列均不为空。

算法 1 MBS()

```
1.  α=assBank(Req_cpl)
2.  IF(Req_cpl 是读请求) {
3.    IF(g_read < Read_Batch){
4.    IF(系统中不存在 waiting request && mFIFO_list 中存在已
        过期的 Multi-bank 请求) {
5.      wr_Req=the oldest request in mFIFO_list
6.      对 assBank(wr_Req)中所有的 bank 进行加锁
7.    }
8.    For(each available bank i∈α && i is not locked){
9.      IF(所有与 wr_Req 相关的 bank 均属于 Idle 状态) {
10.       调度 wr_Req
11.       对 assBank(wr_Req)中所有的 bank 进行解锁
12.     }
13.     Else{
14.       调度 sFIFO_Table[i]中最早的请求
15.     }
16.     g_read ++
17.   }//endFor
18.  }
19.  Else{ // g_read≥Read_Batch
20.    IF(系统中不存在 waiting request){
21.      g_read=0
22.      For(each available bank i∈α)
23.        按照算法 2 调度写请求
24.        g_write ++
25.    }
26.    Else{
27.      IF(所有与 wr_Req 相关的 bank 均属于 Idle 状态){
28.        对 assBank(wr_Req)中所有的 bank 进行解锁
29.        调度 wr_Req
30.        g_read ++
31.      }
32.    }
33.  }
34. }
35. Else{ // Req_cpl 是写请求
36.   IF(g_write < Write_Batch){
37.     For(each available bank i∈α) {//|α|=1
38.       按照算法 2 调用一个写请求
39.       g_write ++
40.     }
41.   }
42. Else{
43.     g_write=0
44.     For(each available bank i∈α){
45.       调用 sFIFO_Table[i]中最早的读请求
46.       g_read ++
47.     }
48.   }
49. }
```

在讲述算法 2 之前，首先讨论写请求的 bank 分配策略。传统的 bank 分配策略按照写请求的逻辑地址分配到确定的 bank 中，例如 bank number=*LBA*/(bank size)，称为静态分配策略。由应用程序访问的局部性原理可知，静态分配策略可能会造成某些 bank 中聚集了大量的 hot 数据或 cold 数据。如果 hot 数据聚集在某个 bank，那么数据的频繁更新将大幅增加 bank 的擦除次数，从而降低了 bank 的使用寿命。另一方面，如果 cold 数据聚集在某个 bank，会增大 bank 的容量利用率。文献[1]表明，当容量利用率增大时，系统性能将大幅下降。例如，当容量利用率从 40%增加到 95%时，闪存能耗将增加 70%～190%，写操作的平均响应时间将增大 30%，闪存 bank 的寿命减少到原来的 1/3。

L. P. Chang 在文献[3]中提出一种动态的 bank 分配策略(即 ASA)，但是 ASA 的调度将一个写请求包含的 *LBA* 动态地分配到存储系统中的多个 bank 上。由于 MBS 建立在 YAFFS 基础之上，YAFFS 总是为写请求分配一段地址连续的闪存空间，因此 ASA 不能直接应用到 MBS 中。本文在 ASA 的基础上，提出一种自适应的 bank 分配方法(下文称 FSA)。为了防止 hot 或 cold 数据聚集在某个 bank 中，FSA 根据第 3.2 节中 AVL-based-tree 方法识别的写请求属性动态地将写请求分配到合适的 bank 中。FSA 的基本思想如下：将 hot 写请求分配到擦除次数小于等于平均值 Avg_EC 的 bank 中，cold 写请求分配到容量利用率小于等于平均值 Avg_CU 的 bank 中，neutrality 写请求分配到擦除次数大于 Avg_EC 且容量利用率大于 Avg_CU 的 bank 中。写请求的具体调度算法如算法 2 所示。

算法 2 wReq_MBS(an available bank i)

```
1. IF(bank i 的擦除次数小于或等于 Avg_EC) {
2.   调度 hot_list 中最早的写请求：hot_list. header
3. }
4. Else{
5.   IF(bank i 的容量利用率小于或等于 Avg_CU) {
6.     调度 cold_list 中最早的写请求：cold_list. header
7.   }
8.   Else{
9.     调度 neutrality_list 中最早的写请求：neutrality_list. header
10.  }
11. }
```

另外，重新排序某些读写请求可能会导致访问冲突。例如，如果两个请求 R_i 和 R_j，其中至少一个为写请求，并且 $[R_i.LBA, R_i.LBA+R_i.size] \cap [R_j.LBA, R_j.LBA+R_j.size] \neq \emptyset$，那么交换 R_i 和 R_j 的执行次序会出现不同的运行结果，R_i 和 R_j 被称为相互冲突的请求，因此冲突请求之间不能改变执行顺序。鉴于此，MBS 在调度没有冲突的读写请求

前，首先按照 FIFO 顺序执行完那些不能改变执行顺序的冲突请求，以保证 MBS 调度的正确性。

5 实验及性能评估

模拟实验建立在 Ubuntu-8.04 操作系统之上，用 MTD NandSim 模拟 256MB 的闪存空间。模拟实验采用的闪存页面大小为 512Bytes，块大小为 16k Bytes，闪存读写性能的参数设置如表 1 所列。加载的文件系统为 YAFF2。我们用 Linux 中的 blktrace 工具获取的一个普通 PC 用户在一周内的所有 I/O 操作，运行的应用程序包括 FireFox，OpenOffice，Eclipse，Transmisson，SMPlayer，Skype 等。表 3 列举了包含 60,000 个请求的 Trace 的基本特征。

表 3 Trace 的基本特性

文件系统	YAFFS2
应用程序	FireFox，OpenOffice，Eclipse，Transmisson，SMPlayer，Skype
读取闪存页面总数	1,614,214 Pages
写闪存页面总数	1,202,367 Pages
I/O 请求总数	60,000
读写请求的比例	52.7% / 47.3%
读写请求的平均长度	51 Pages / 42.3 Pages
I/O 请求的分布	79% of requests access 21% of LBAs

实验设置 AVL-based-tree 机制中结点计数器 Counter 具有 4bits，即 $C=4$；衰减周期 *Threshold* 为 16。判定阈值 $UTH=8$，$LTH=4$。此外 MBS 调度算法中可调节的参数取值如下：*Write_Batch*=16，single_bank 请求和 multi_bank 请求的有效期分别设为 500ms 和 1000ms。实验从以下 3 个方面对 MBS 和 NOOP 进行比较：I/O 吞吐率、读写请求的平均响应时间、不同 bank 的擦除次数和容量利用率分布。

5.1 I/O 吞吐率

首先，闪存设备内的 bank 数量对系统的 I/O 吞吐率具有直接的影响。随着闪存系统中 bank 数量的增加，多个请求的并行度也相应地增加，因此也相应地增加了系统的 I/O 吞吐率。但是 I/O 吞吐率并不总是随着 bank 数量的增加而增加的，当 bank 数量达到一定的数量时，bank 数量的增加对 I/O 吞吐率的影响不大。

由上述可知，闪存操作的 Setup 阶段占用 I/O 总线，因此此阶段不可并行，但属于不同 bank 的闪存操作的 Busy 阶段可以独立并行。因此闪存的操作 *OP*(*OP* 为 Read/Write/ Erase) 最大的并行度 $Max\_Deg_{OP}=\lceil (OP_{Busy}+OP_{Setup})/OP_{Setup} \rceil$。当 bank 的数量大于 Max_Deg_{OP} 时，bank 的增加将不能提高操作 *OP* 的并行度。由表 1 可知，读操作的 Max_*Deg*=2，写操作的 Max_*Deg*=14。因此，当 bank 数量从 1 增加到 16 时，I/O 吞吐率呈线性增加，如图 3 所示。当 bank 的数量大于 16 时，I/O 吞吐率变化不大。另一方面，由于 NOOP 调度采用静态 bank 分配策略，因此 NOOP 引入了更多的擦除操作以及 Live 数据的 Copy，MBS 调度的 I/O 吞吐率比 NOOP 调度高出 10%～40%。

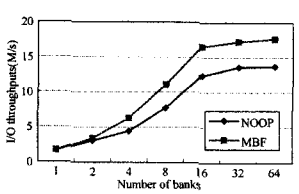

图 3 不同 bank 数量时的 I/O 吞吐率

5.2 读写请求的平均响应时间

由于闪存读操作的最大并行度为 2，因此当 bank 数量从 1 增加到 2 时，读请求的平均响应时间(average response time)约为原来的一半(见图 4)。当 bank 数量大于 2 时，读请求的响应时间变化不大。另一方面，由于 MBS 给予读请求更高的优先级以及访问的局部性原理，读请求在 MBS 调度下的平均响应时间比 NOOP 调度下的平均响应时间降低了 28%～46%。

由于闪存写操作的最大并行度为 Max_*Deg*=14，因此当 bank 数量大于 16 时，写请求的平均响应时间变化不大(见图 5)。另一方面，NOOP 调度下闪存 bank 内容量利用率的不均衡大大增加了擦除操作的开销(大量 Live data 需要 Copy)，从而降低了系统的性能。因此，NOOP 调度下的写请求平均响应时间大于 MBS 调度。然而，当系统中仅有一个 bank 时，由于 MBS 给予读请求更高的优先级，因此 MBS 调度下的写请求平均响应时间稍大于 NOOP 调度。当 bank 数量大于 1 时，MBS 写请求的平均响应时间比 NOOP 减少了 15%～33%。

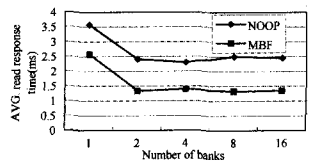

图 4 不同 bank 数量时读请求消耗的平均时间

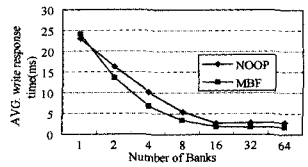

图 5 不同 bank 数量时写请求消耗的平均时间

5.3 擦除次数和容量利用率

由于 NOOP 将写请求按照 LBA 地址静态地分配到确定的 bank 上，由于访问的局部性原理，hot/cold 写请求可能会聚集到某个 bank 上，造成各个 bank 的擦除次数和容量利用率严重失衡。一方面影响了读写请求的平均响应时间，同时大大缩减了闪存的使用寿命。图 6 为不同 bank 分配策略下 bank 的擦除次数和容量利用率。

	Bank0	Bank1	Bank2	Bank3
擦除次数(NOOP)	6978	5999	28097	26159
擦除次数(MBF)	16437	16435	16433	16436
容量利用率(NOOP)	75%	70%	85.8%	83.5%
容量利用率(MBF)	80%	79.5%	79.3%	80.8%

图 6 不同 bank 分配策略下 bank 的擦除次数和容量利用率

结束语 本文提出了一种基于闪存文件系统的调度方式 MBS。MBS 利用 bank 的并行性使得多个读/写请求可以并行地执行，并给予读请求更高的优先级。另外，MBS 根据 AVL-based-tree 方式识别出的写请求属性，动态地将写请求分派到合适的闪存 bank 上，以防止 hot/cold 数据聚集在某个 bank 中，造成擦除次数或者容量利用率过高。实验证明，动态 bank 分配策略减小了读写请求的平均响应时间并延长了闪存的使用寿命。接下来的研究主要针对实时环境下的闪存请求调度。

参 考 文 献

[1] Douglis F, Caceres R. Storage Alternatives for Mobile Computers [C]// Proc. the First Symp. Operating Systems Design and Implementation(OSDI-l 94). Nov. 1994:25-37

[2] Chiang M L, Paul C H, Chang R C. Manage flash memory in

personal communicate devices [C]// Proceedings of International Symposium on Consumer Electronics. 1997

[3] Chang L P, Kuo T W. An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems [C]// 8th IEEE Real-time and Embedded Technology and Applications Symposium(RTAS). September 2002:187-196

[4] Inoue A, Wong D. NAND Flash Applications Design Guide [R]. Toshiba America Electronic Components, Inc, 2003

[5] Kawaguchi A, Nishioka S, Motoda H. A Flash-memory based File System [C]// Proceedings of the 1995 USENIX Technical Conference. January 1995:155-164

[6] Intel Corporation. Ftl Logger Exchanging Data With Ftl Systems [R]. 1995

[7] Intel Corporation. Understanding the Flash Translation Layer Specification [EB/OL]. http://www.intel.com/design/flcomp/applnots/297816.htm, 1998

[8] M-systems. Flash-Memory Translation Layer for NAND Flash (NFTL) [R]. 1998

[9] WoodHouse D. Jffs: The Journaling Flash File System [EB/OL]. http://sources.redhat.com/jffs2/jffs2-html/

[10] Bityutskiy A B. JFFS3 Design Issues. Version 0.32(draft) [EB/OL]. http://www.linux-mtd.infradead.org/tech/JFFS3design.pdf

[11] Alpha One Limited. Yet Another Flash Filing System [EB/OL]. http://www.yaffs.net/yaffs-overview, 2006

[12] Wu M, Zwaenepoel W. eNVy: A Non-volatile Main Memory Storage System [C]// Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems. 1994:86-97

[13] Lee S W, Moon B. Design of flash-based DBMS: An in-page logging approach[C]// Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. Beijing, China, 2007:55-66

[14] 周大,梁智超,孟小峰. HF_Tree:一种闪存数据库的高更新性能索引结构[J]. 计算机研究与发展,2010,47(5):832-840

[15] Wu C H, Kuo T W, Chang L P. An Efficient B-Tree Layer Implementation for Flash-memory Storage Systems [J]. ACM Transactions on Embedded Computing Systems, 2007, 6(3)

[16] 卢泽萍,孟小峰,周大. HV_Recovery:一种闪存数据库的高效恢复方法[J]. 计算机学报,2010,33(12):2258-2266

(上接第 277 页)

际上在人脸识别算法中通用的分类性能度量方法 ROS(rank order statistic)[12],给出了本文提出方法的 CMS(cumulative match scores)即累积匹配分值图。CMS 定义为一个测试度量的实际类别在它的最前 k 个匹配值之间的累积概率,CCR 等价于 $k=1$ 时的 CMS。

表 1 不同算法正确分类率的比较

分类器	算法	CCR(%)
NN	Little et al.[4]	77.5
	Lee et al.[5]	78.25
	本文方法	84.25

表 2 COT 方法的压缩性能比较

算法		Top1(%)	Top5(%)	Top10(%)
Wang 等的方法	STC. NN, No validation	65.00	——	——
	STC, NN, With validation	68.75	——	——
	NED, NN, No validation	65.00	——	——
	NED. NN, With validation	70.00	——	——
BenAbdelkader 等的方法		72.50	88.75	96.25
Collins 等的方法		71.25	78.75	87.5
Philips 等的方法		78.75	91.25	98.75
本文的方法		84.25	90.55	96.5

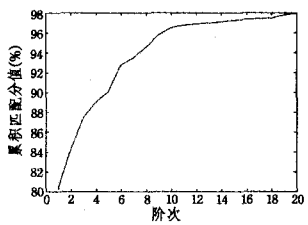

图 5 累积匹配分值图

结束语 本文提出了一种新的鲁棒性的步态识别算法,这种算法基于鲁棒性较强的动力学特征光流,并建立多区域椭圆模型的人体结构模型,将目标人体区域部分按人体结构特点划分为多个子区域,每个子区域通过结合光流特征的椭圆模型进行拟合。使用中国科学院自动化研究所提供的 NLPR 数据库进行实验,实验通过最近邻分类器 NN 进行分类,取得了较理想的识别效果。

参 考 文 献

[1] Liu Z, Sarkar S. Effect of silhouette quality and hard problems in gait recognition[J]. IEEE Trans. Systems Man and Cybernetics. part B: Cybernetics, 2005, 35(2):170-183

[2] Yu S, Tan D, Huang K, et al. Reducing the Effect of Noise on Human Contour in Gait Recognition[C]// Internat. Conf. on Biometrics. 2007:338-346

[3] 方玫. OpenCV 技术在数字图像处理中的应用[J]. 北京教育学院学报:自然科学版,2011(01):7-11

[4] Varadhan K. The NS Manual[EB/OL]. http://www.isi.edu/nsnam/ns/doc/, 2010-02-08

[5] 任亚峰,郑金华. 多目标进化算法鲁棒性实验研究[J]. 计算机应用研究,2011(02):451-454

[6] Chen Chang-hong, Liang Ji-min, Zhao Heng, et al. Frame DifferenceEnergy Image for Gait Recognition with Incomplete Silhouettes[J]. Pattern Recognition Letters, 2009, 30(11):977-984

[7] Lucas B, Kanade T. An iterative image registration technique with an application to stereo vision[C]// Proc of DARPA U Workshop. 1981:121-130

[8] 赵永伟,张二虎,鲁继文,等. 多特征和多视角信息融合的步态识别[J]. 中国图像图形学报,2009,14(3):388-392

[9] McLachlan G, Krishnan T. The EM Algorithm and Extensions [Z]. Wiley Interscience, 1997

[10] 郑帅. CASIA 步态数据库申请说明[EB/OL]. http://www.cbsr.ia.ac.cn/china/Gait%20Database%20CH.asp, 2011-03-03

[11] Chen Chang-hong, Liang Ji-min, Zhao Heng, et al. Frame Difference Energy Image for Gait Recognition with Incomplete Silhouettes[J]. Pattern Recognition Letters, 2009, 30(11):977-984

[12] Dwass M. Simple random walk and rank order statistics[J]. Ann. Math. Statist., 1967, 38:1042-1053