计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2018,27(5):102−111 [doi: 10.15888/j.cnki.csa.006350]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 OpenSSD 的闪存转换算法优化①

陆　政, 范长军, 江云飞

(中电海康集团有限公司, 杭州 310012)

摘　要: 随着信息技术的快速发展, 数据存储的需求日益增长, 人们对硬盘读写性能的要求越来越高. 相比于机械硬盘, 固态硬盘可靠性高、能耗低, 无寻道时间开销, 逐渐取代机械硬盘成为主流的存储介质. 但固态硬盘访问数据时需要经过转换, 对应的闪存转换算法对读写性能影响很大. OpenSSD 项目提供了一个可开发 SSD 固件的平台, 基于此平台本文针对其上的闪存转换算法进行研究并优化, 分析了影响 I/O 读写性能的各类因素, 设计出一种适合 Cosmos OpenSSD 的缓存管理和闪存管理方法, 大幅提高了 Cosmos OpenSSD 的性能.

关键词: 固态硬盘; 闪存转换算法; 缓存管理; 闪存管理

引用格式: 陆政,范长军,江云飞.基于 OpenSSD 的闪存转换算法优化.计算机系统应用,2018,27(5):102–111. http://www.c-s-a.org.cn/1003-3254/6350.html

Improved Flash Translation Layer Algorithm Based on OpenSSD

LU Zheng, FAN Chang-Jun, JIANG Yun-Fei

(CETHIK Group Co. Ltd., Hangzhou 310012, China)

Abstract: With the rapid development of information technology, the need for data storage is growing, and the requirements for I/O performance are getting higher and higher. Compared with the traditional Hard Disk Drive (HDD), Solid-State Disk (SSD) has more advantages such as high availability, low energy consumption, no seek time, etc. Due to these reasons, SSD gradually replaces HDD to become mainstream storage media. The OpenSSD project is an initiative to promote research and education on the recent SSD technology by providing easy access to OpenSSD platform on which open source SSD firmware can be developed. In this study, we investigated the Flash Translation Layer (FTL) algorithm on the Cosmos OpenSSD platform, analyzed all kinds of factors that affect the I/O performance, designed a way to carry out the buffer management and flash management, and finally improved the I/O performance effectively.

Key words: Solid-State Disk (SSD); Flash Translation Layer (FTL) algorithm; buffer management; flash management

硬盘作为主要存储介质, 在计算机中扮演着重要的角色. 相比于传统机械硬盘, 固态硬盘 (Solid State Drive, SSD) 具有读写访问速度快、噪音小、耐用性高等优点, 因此, SSD 的大规模普遍使用是存储技术未来发展的必然趋势.

SSD 由闪存组成[1], 由于闪存具有读写以页为单位而擦除以块为单位, 覆写之前需要擦除的物理特性, 因此一般的文件系统需要通过闪存转换层 (Flash Translation Layer, FTL) 来操作 SSD, 以方便读写 SSD 上的数据.

FTL 固件算法主要用于管理逻辑地址到物理地址的映射[2], 起到地址映射管理的功能. 由于闪存芯片有一定的擦写寿命, 为在使用过程中保持损耗的均衡性, 延长 SSD 的使用寿命, 一个成熟的 FTL 往往还包括内存管理[3]、垃圾回收[4]、磨损均衡[5]等功能.

FTL 算法的优劣是决定 SSD 性能的一个重要因素, 而目前商业的 SSD 都不公开其 FTL 固件的具体实

① 收稿时间: 2017-08-29; 修改时间: 2017-09-15; 采用时间: 2017-09-29; csa 在线出版时间: 2018-04-23

现, 从研究角度而言, 就无法去探索 FTL 的不同设计方案, 以更好地发挥 SSD 的性能, 也无法针对特定的应用场景 (如随机小 I/O 为主的 OLTP 型业务) 对 SSD 的性能进行优化. OpenSSD 项目[6]提供了一个开放的 SSD 研究平台, 以方便对 SSD 的固件进行设计开发, 并验证不同的 FTL 方案对 SSD 性能的影响.

Cosmos OpenSSD 拥有 1GB 内存并具有多通道结构, 其自带的 FTL 算法实现简单, 只能保证基本的 I/O 读写操作. 由于没有充分利用 OpenSSD 的多通道结构实现数据并行存取, 且内存空间使用率低下, 所以其 I/O 读写性能, 特别是针对小粒度随机写负载的性能, 表现并不理想. 本文工作基于 Cosmos OpenSSD 硬件平台进行设计开发, 旨在通过设计并优化其 FTL 算法来提升 SSD 整体存储性能.

1 研究现状

SSD 一般由 NAND Flash 闪存构成, 单个 NAND Flash 的容量和性能有限, 因此每个 SSD 一般由多个闪存整合而成[7], 一个 SSD 的典型结构[8]如图 1 所示.

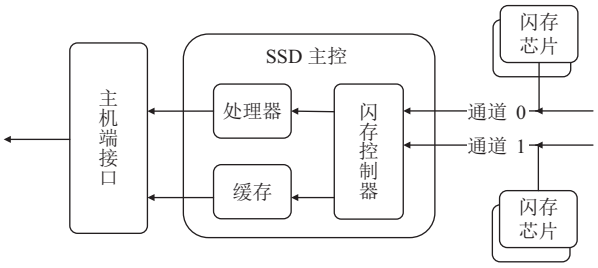


图 1 SSD 结构图

各大硬盘厂商在不断提升闪存容量的同时也注重于主控算法的研究来充分发挥 SSD 的性能. FTL 映射方法可以分为基于页粒度的页映射、基于块粒度的块映射和混合此两种方法的混合映射. 页映射灵活却需要占用较多的资源来保存映射表; 块映射相反, 它只需较少的资源保存映射表, 但映射方法笨拙, 性能不理想; 因此在资源有限的嵌入式系统里最常用的是混合映射方法.

目前最基本的混合映射 FTL 算法是 Log-block 算法[9]. 这种算法大部分块以块为单位映射, 这些块叫做 Data Block; 少部分块以页为单位进行映射, 叫做 Log Block. 上层下发的写 I/O 如果是一个新数据, 则根据偏移写入 Data Block; 如果是已有的数据, 则映射到 Log Block 上. 当 Log Block 全部都写满后, 则进行合并操作来回收 Log Block, 这引起了额外的读、写和擦除操作. 当写 I/O 为主要负载时, 这种映射方式因主要以块映射为主, 对于大部分写请求来说映射太广, 由其引发的垃圾回收操作将会严重影响存取性能.

在 Log-block 算法的基础上, 后续发展出了 BAST、FAST[10]等算法. BAST 算法里 Data Block 和 Log Block 一一对应, 以提高闪存的空间利用率, 但在频繁地随机写 I/O 场景下, Log Block 的利用率很低, 并且频繁进行合并操作, 导致随机访问性能下降. FAST 算法在 BAST 算法的基础上进行改进, 使 Log Block 和 Data Block 比例为 1:N, 大大提高了 Log Block 的利用率, 但是在合并时, 一个 Log Block 里会包含多个 Data Block 里的数据, 垃圾回收会导致多次合并, 影响性能.

虽然混合映射机制折中了页映射和块映射的优缺点, 适合嵌入式场景, 但是仍存在闪存块的利用率以及垃圾回收的效率低和随机 I/O 性能差等问题. 并且随着固态存储技术的发展, SSD 内部通道数越来越多, 容量越来越大, 现在 SSD 内嵌入式系统上拥有更为丰富和灵活的硬件资源, 因此主要研究方向也逐渐从混合映射方式转到页映射方式[5].

在页级映射方法中, Aayush Gupta 等人提出的 DFTL(Demand-based FTL) 算法[11]是典型的代表. DFTL 是一种基于页映射的算法, 它的块分为两类: 一类保存普通数据, 一类保存页映射表信息, 同时在内存中也保存一小部分页映射表信息. 根据时间局部性和空间局部性原理, 上层的 I/O 请求命中时, 性能大幅度提升. 但 DFTL 严重依赖于负载特性、读写比例、数据请求间隔时间等因素, 这使 DFTL 很可能因不适应负载而导致频繁地更新内存中的映射表, 从而造成性能下降.

为解决以上问题, 本文在 OpenSSD 平台下基于页映射的方式设计出一种适合 OpenSSD 的缓存管理和闪存管理方法, 以实现高性能的 FTL 算法.

2 OpenSSD 的特性和结构

Cosmos OpenSSD 是韩国汉阳大学基于 HYU Tiger3 控制器制作出来的一款 PCIe SSD. 如图 2 所示, OpenSSD 上每 4 个芯片叠加成一个 package, 每个

package 拥有一个的独立通道, 该通道在 package 内部与 4 个芯片通过 way 连接. OpenSSD 闪存控制器上含有 4 个通道, 每个通道连接一个 package, 所以最多可以进行 4 通道和 16 条 way 的并行 I/O 处理操作. 其中, 同一个 package 上面的芯片共享一条 I/O 通道, 但通道之间操作互不干扰, 可以并发进行. 除了片选信号, 同一个 package 上的芯片使用同样的命令信号而且每个芯片都有它自己的忙信号.

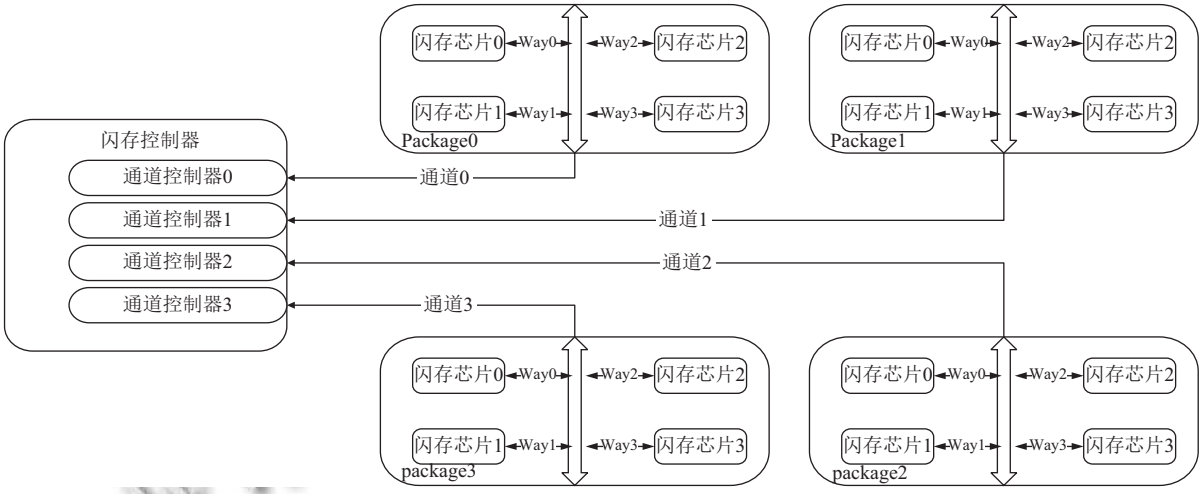


图 2 OpenSSD 闪存层次结构图

如图 3 所示, OpenSSD 数据的传输过程分为两部分: 1) 数据通过 DMA 在主机端和 OpenSSD 的缓存空间之间进行交互; 2) 数据在 SSD 缓存空间与 NandFlash 闪存介质之间进行传输. 由此可知, OpenSSD 的性能取决于主机端与 SSD 缓存之间的 DMA 传输速度以及 SSD 缓存与闪存之间的传输速度.

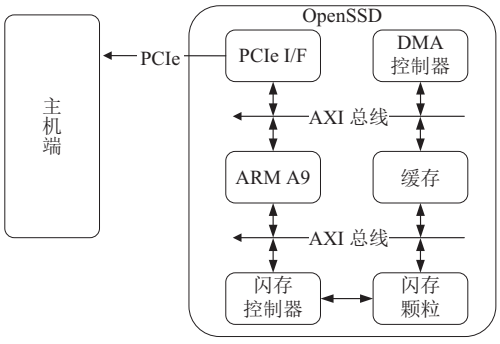


图 3 OpenSSD 数据传输流

经测试, OpenSSD 的 DMA 传输速度如图 4 所示. 通过分析可知, 随着粒度的增大, DMA 传输速度也随之增长.

闪存读写速度如图 5 所示, 由图可知写 I/O 操作进行 way 级别的并行化后, 性能得到很大提升; 但在读 I/O 操作时, 如果 way 属于同一条通道, 那么 way 数目的增长对读性能的提升几乎没有影响, 而属于不同通道的 way 数目增长对于读性能提升很大, 因此读操作 way 级别的并行效果不明显, 而通道级别的并行操作对性能有很大的提升.

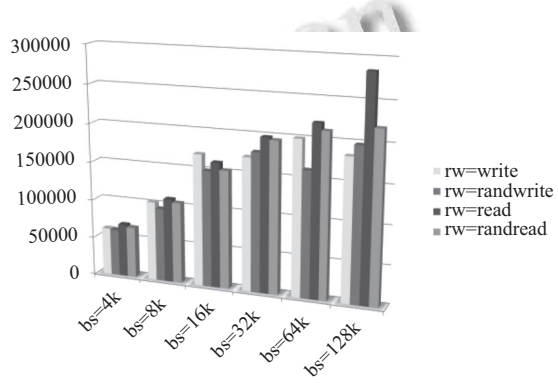


图 4 不同粒度下 DMA 传输速度

在实际设计中, 应该根据这两部分的实际速度来设计合理的 FTL 算法, 以提升 SSD 的存储性能.

3 系统概要设计

OpenSSD 的 FTL 系统概要设计如图 6 所示. 主要包括两个模块: 缓存管理模块和闪存管理模块. 缓存按功能被划分为写缓存 (write buffer)、读缓存 (read

cache) 以及用作 DMA 传输 (DMA buffer) 和垃圾回收的缓存区域 (GC buffer).

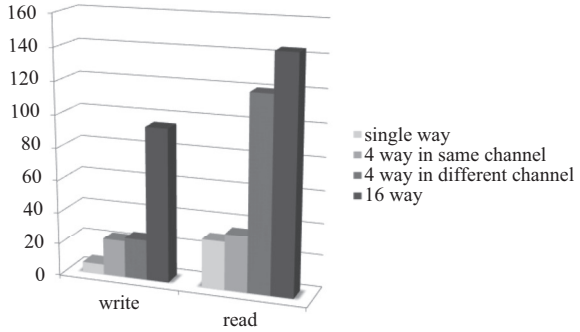


图 5 闪存读写速度 (MB/s)

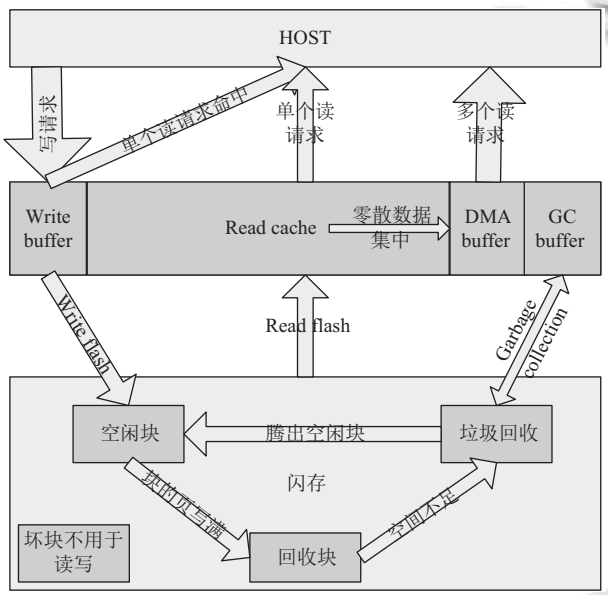


图 6 系统概要设计

如图 6 所示, 当主机端对 SSD 进行写 I/O 操作时, 数据首先会被暂存在 write buffer 中, write buffer 不会立刻将数据写入闪存, 而是将数据在其中保留一段时间并做一些冷热数据区分的处理, 当缓存区域空间达到一定的阈值之后, 再将 write buffer 中的数据并行写回闪存以提升存取性能.

NAND Flash 芯片一般包含若干闪存块, 每个闪存块包括若干页, 闪存芯片的读取和写入以页为单位, 擦除以块为单位, 数据在覆写之前需擦除. 当 write buffer 写回闪存时, 需要根据一定的规则来提供空闲块, 空闲块中应至少含有一个空闲页. 当块中没有空闲页时, 数据无法再进行写入, 块变为无效块, 也就是回收块, 需要等待垃圾回收进行处理. 上述过程的顺利进行, 由相应的闪存管理机制来保证.

闪存管理模块的作用是高效地管理闪存块. 如图 6 所示, 当 write buffer 的数据写回闪存颗粒时, 闪存管理模块提供空闲块来供 write buffer 写回数据, 随着数据的不断写入, 空闲块越来越少, 回收块越来越多, 此时需要对数据块进行垃圾回收操作, 因为 OpenSSD 闪存颗粒没有回拷 (CopyBack) 功能, 所以垃圾回收需要借助于缓存空间 (GC buffer) 来搬运数据. 由于闪存数据块中保存的数据冷热程度不一致, 导致闪存数据块损耗的速度不一致, 因此闪存管理模块除了提供基础的读写功能外, 还需要实现数据块的磨损均衡, 同时随着数据块的不断写入擦除, 当写入擦除达到一定次数之后, 数据块将会损坏, 无法正确读写数据, 此时闪存管理模块需要将这些数据块隔离, 不再参与读写.

读缓存主要用来预读取和临时存储最近访问过的数据, 以提高主机读 I/O 时的命中率. 读 I/O 请求时, 若数据在写缓存中, 则直接读取, 若在读缓存中, 则数据与闪存中保持一致, 也可直接访问. 如果读取的数据在闪存中, 则将数据读取到缓存内再返回主机. 当上层读取大段的数据时, 由于数据在缓存或闪存中是零散分布的, 所以需要将数据集中到 DMA buffer 后再返回给上层.

4 OpenSSD FTL 模块设计

4.1 写缓存模块设计

4.1.1 数据接收模块设计

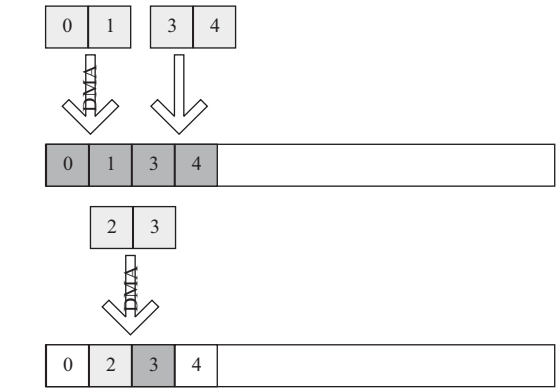
对于闪存颗粒来说, 小粒度的映射有利于提升整体性能和寿命. 这是因为映射粒度越大, 会导致底层无效数据增长越快, 闪存颗粒空间利用率越低, 并引起频繁的垃圾回收操作, 不仅会影响性能, 也会极大降低闪存的使用寿命.

OpenSSD 拥有足够大的缓存空间, 为便于 I/O 性能的发挥和 SSD 寿命的延长, 本文选择 4 KB 大小的粒度来进行映射.

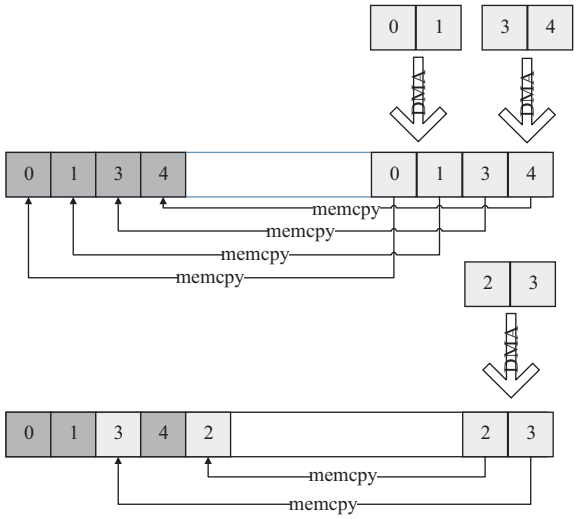
OpenSSD 总共拥有 1 GB 的缓存空间地址, 其中 0x0000_0000 到 0x1000_0000 已经被系统使用, 剩下的缓存空间本文将其划分成 5 部分, 分别为: 元数据信息存储区、写缓存数据存储区、读缓存数据存储区、DMA 数据传输缓存区和垃圾回收数据存储区.

写缓存数据存储区用于接收上层下发的 I/O 数据, 上层下发的 I/O 数据中可能包含之前已经写入缓存的

数据. 如图 7(a) 所示, 假设缓存中已有数据 0、1、3、4, 现收到下发的 I/O 数据 2、3, 则数据 3 命中, 如果将 I/O 数据直接写入缓存命中的地址, 数据 1 将被数据 2 覆盖掉, 导致相邻地址空间的数据被破坏.



(a) 缓存中有效数据1被2覆盖



(b) 覆盖问题解决方案1

图 7 memcpy 相关方案

为了解决这个问题, 提供了以下两个解决方案:

方案一如图 7(b) 所示, 上层数据首先通过 DMA 传输到一个固定的缓存空间, 然后通过 memcpy 操作将数据移动到写缓存数据存储区中.

方案一不仅能够解决写缓存空间的问题, 而且在将缓存写回闪存时, 能够根据数据冷热程度区分选取逻辑页进行回写, 因为即使缓存空间使用情况零散, 在新 I/O 到来时也能用 memcpy 操作把数据从 DMA buffer 拷贝到 write buffer 任意地址, 因此可以将频繁被访问的数据也就是热数据尽量留在内存空间, 而把不常被访问的数据也就是冷数据写回闪存. 把热数据留在内存空间而把冷数据写回闪存, 一方面能够提升数据的缓存命中率, 提升写存储性能; 另一方面由于闪存内保存的是冷数据, 因此更新频率相比热数据更新频率会低很多, 从而减慢垃圾回收的速度, 在提升系统存储性能的同时, 也降低了闪存颗粒寿命衰减的速度.

在方案一中, 许多常用的 FTL 算法都可以借鉴过来使用, 但是局限于 OpenSSD 的硬件特性, 在实际使用场景中并不总是适用. 这是因为 OpenSSD 的 memcpy 的速度只有 60 MB/s, 它与 DMA 的传输速度以及闪存的写速度处于一个数量级, 如果每次操作都要进行缓慢的 memcpy 操作的话, 会极大地影响性能. 当进行小粒度如 OLTP 类的随机写入时, 对应的 DMA 传输速度为 60 MB/s, 这个时候即使缓存命中, 相比直接执行覆写的操作来说, 加入一个 memcpy 的过程意味着缓存命中处理的时间多了一倍; 而当进行大粒度的写请求时, 比如视频播放等业务, OpenSSD 的瓶颈在于闪存写入速度, 即使所有通道和 way 并发, 闪存的写入速度也仅为 90 MB/s 左右, 此时在写入操作时再加入 60 MB/s 的 memcpy 操作, 将会给性能带来巨大的负面影响.

针对以上问题, 本文提出了设计方案二: 如图 8, 采用追加写的方式来接收上层数据, 即每次写入数据的时候接着上次请求的地址开始写入. 如果新数据已经存在于缓存空间, 则无效掉命中的缓存中的旧数据, 并将新数据按照上述方式接着上次请求的地址写入. 如图 8 所示, 之前缓存中已有数据 0、1、3、4, 现收到下发的写 I/O 数据 2、3, 虽然数据 3 命中, 但是将其无效掉, 并将数据 2、3 接着上次请求的地址写入. 这种方式屏蔽了大量的 memcpy 操作, 有利于系统性能的提升.

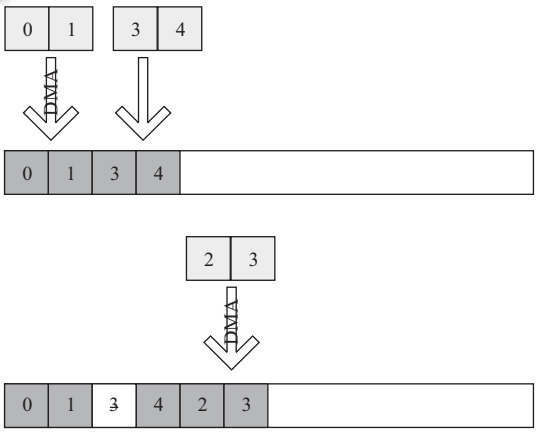


图 8 追加写方式

本文选取的映射粒度为 4 KB, 小于 OpenSSD 的页大小 8 KB, 方案二的这种追加写的方式会导致一些

缓存空间保存的是无效的数据, 而闪存读写只能以页的大小 8 KB 为单位, 导致将缓存数据写回闪存时可能会写回一些无效的数据. 在这种情况下, 写缓存区域越大, 出现无效数据的概率也会越大, 因此写缓存区不宜过大.

同时由于写缓存区的存在, 所以写闪存操作和缓存 DMA 传输可以并发进行. 当 I/O 请求为大粒度写 I/O 时, 因为采用追加写的方式, 当数据命中时会无效掉之前的旧数据, 然后再写入更新的数据, 写缓存空间会被快速消耗, 此时写缓存数据的存储性能瓶颈在于闪存的写入速度, 这表示当写缓冲数据存储区空间不足时, 速度将被闪存写入速度所限制; 如果是单个逻辑页的写请求命中, 则可以通过 DMA 传输直接进行数据覆写, 但是比较图 4 和图 5, 4 K 粒度时 DMA 传输速度比闪存并行的写入速度慢, 8 K 粒度时写入速度持平, 这就代表写入小粒度数据时, 闪存的并行极限速度要比 DMA 传输速度更快, 也就是说因为写回速度更快, 所以上层进行 DMA 传输的时候总能保证有空闲的内存空间可写, 这个时候缓存命不命中对存储性能影响不大.

由上述分析可知, 在方案二的情况下写缓冲数据存储区的大小对存储性能影响不大, 再综合考虑写回无效数据的因素, 写缓冲数据存储区的大小设置得比较小是合适的, 因此本文设计写缓冲数据存储区大小为 4 MB.

4.1.2 写回模块设计

写回模块用于将写缓冲中的数据下盘到闪存颗粒, 其设计包含三个关键问题: 一是写回数据的选择; 二是写回的时机; 三是写回地址的选择.

由于采用追加写的方式来接收上层 I/O 数据, 因此采取 FIFO 的方式写回是比较合适的, 这是因为如果采取区分冷热数据的写回方式, 缓存区的数据会过于零散, 此时管理缓存需要大量进行 memcpy 操作, 会极大地影响性能.

根据缓存写回的时机不同可以分为主动写回机制和被动写回机制, 主动写回机制是指控制器在缓存空间还有空闲的时候主动进行写回操作来清理缓存区的方式, 这种方式的优势是对缓存区空间进行预清理, 以保证后续数据写入时缓存空间足够. 被动写回是指缓存区达到一定阈值之后触发写回操作, 这是必须要实现的机制, 在缓存数据量到达缓存区大小的阈值时, 触发写回操作来清理缓存区供新数据写入.

具体设计如图 9: 上层的新数据写入缓存 FIFO 队列尾端, 当 FIFO 队列大小达到写缓存空间的 50% 时, 触发主动写回操作. 每次触发写回操作时, 从 FIFO 队列头顺序选择 8 KB*16=128 KB 的数据写回, 这样能充分利用通道和 way 的并行. 如果主动写回速度大于上层负载下发 I/O 速度时, 则写缓存空间总是有地址可供写入; 如果上层负载 I/O 比较密集, 则随着数据的不断写入, 写缓存可用空间越来越小, 当写缓存可用空间不足以承载此次 I/O 时, 此时停止接收下发的 I/O, 进行被动写回操作, 直到清理出可供上层 I/O 写入的空间后继续进行写 I/O 操作.

写回数据时涉及到写回地址的选择, 在逻辑地址和物理地址有着一定对应关系的静态地址分配方式下, 并行写回需要额外的管理. 此外, 静态地址分配也不适合不区分冷热数据的缓存管理方式, 这会导致各个闪存颗粒上面的数据冷热不均匀, 使磨损的不均衡性进一步加剧. 因此逻辑地址和物理地址没有固定对应关系的动态地址分配方式成为了首选, 一方面这种方式适合 FIFO 写回方式的并发操作, 逻辑页可以随意写入任意的物理页中, 另一方面动态地址分配也使各闪存颗粒的磨损更均匀, 有利于磨损均衡. 此时逻辑页写回的原则为哪条通道空闲就使用哪条通道, 最大程度地提高了并行度. 当 OpenSSD 将数据从写缓存写回闪存颗粒时, 被选择进行写回的 128 KB 数据将会被随机分配到 4 个通道, 16 条 Way 下, 以达到高效率地并行写回效果.

4.1.3 方案二局限性

当负载拥有小数据量的热数据或是反复写特定的数据时, 采用 FIFO 式的追加写方式管理缓存数据无法有效区分冷热数据, 所以无法达到数据的高缓存命中率, 此时性能表现略不理想, 同时这也不利于对闪存块进行高效的磨损均衡管理.

写缓冲数据存储区大小设置为 4 MB 也会存在一些局限性, 当大粒度写 I/O 请求不是很频繁时, 更大的缓冲区能够缓存更多的数据, 具有大缓冲区的方案性能会更胜一筹.

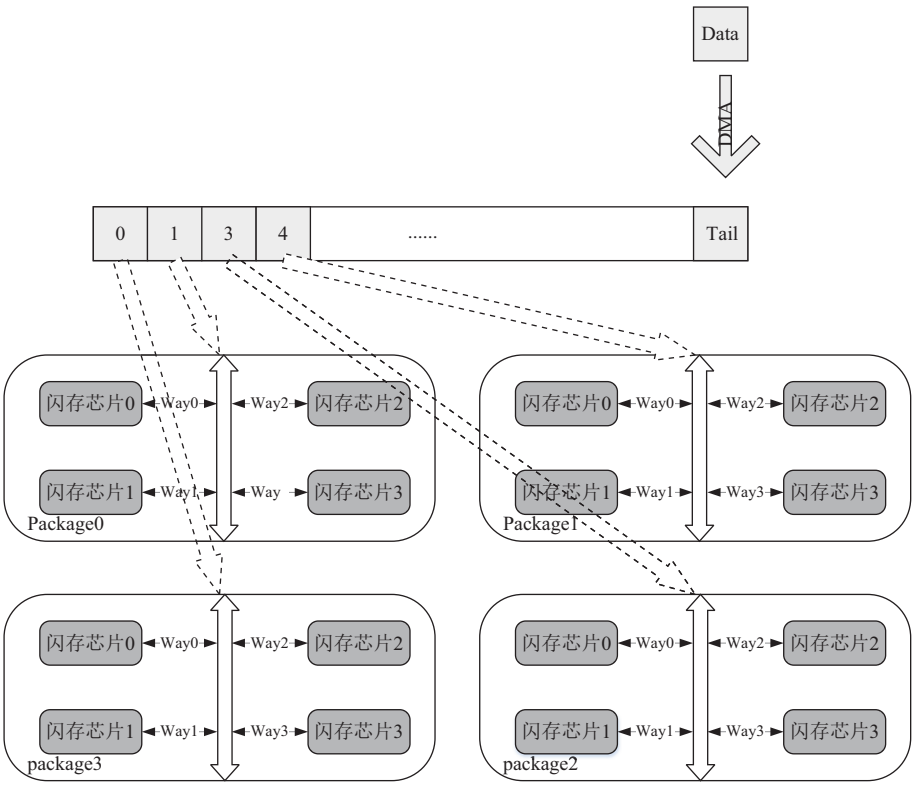


图 9 数据写回示意图

4.2 读缓存模块设计

读缓存数据存储区的缓存单元与逻辑地址的映射关系由红黑树结构来管理. 红黑树是一种自平衡二叉查找树, 和 AVL 树类似, 在进行插入和删除操作时通过特定操作保持二叉树的平衡, 从而获得较高的查找性能. 在实际应用中, 红黑树是高效的, 它可以在 O(log$n$) 时间内执行查找、插入和删除操作, 这里的 n 是树中节点的数目.

为了区分冷热数据, 让热数据在内存空间停留的时间尽可能长, 而冷数据尽量保存在闪存中, 读缓存区需要一个缓存管理算法来管理冷热数据. 由于 OpenSSD 的处理器性能不是太强, 过于复杂的管理算法会导致较大的时间开销从而影响存储性能, 因此选择易于实现、代价小且性能还不错的 LRU 算法.

读缓存区域被划分成许多小单元, 设置每个单元的大小与闪存页大小相同. 这些小单元分为已使用和未使用两种状态, 未使用的缓存单元链接成一个 FIFO 队列, 当从闪存读数据到缓存空间时, 从 FIFO 队列中取缓存页使用; 已使用的缓存单元链接成一条链表, 使用 LRU 算法管理.

当上层下发读 I/O 命令时, 如果数据在闪存中, 则需要将数据从闪存页读取到缓存空间, 此时读缓存从 FIFO 队列中分配缓存空间以供数据从闪存读入, 读取的数据根据 LRU 算法插入已使用单元的队列中, 并在红黑树中保存逻辑地址和内存地址的映射关系, 当缓存空间不足时, 将 LRU 链表尾部的冷数据清除, 腾出的缓存单位插入未使用的 FIFO 队列中以继续供读缓存使用.

如图 9 所示, 写闪存时可以进行并行数据处理, 同理在读闪存时也可以做一些并行处理. 根据图 5 的测试结果可知, 读可以进行通道级别的并行, 因此在主机端读取数据时, 若有通道空闲就从这个通道预读一些数据. 预读的策略是检测读取数据的相邻几个逻辑页数据是否存在于空闲通道的闪存颗粒中, 如果存在, 则将其一并读取到缓存中, 这能显著提升小粒度的顺序读性能. 因为逻辑页粒度为 4 K, 而闪存物理页大小为 8 K, 因此上层下发的一个小于 4 K 的读 I/O 请求会额外读取到另一个逻辑页, 在一定程度提高了缓存命中率.

4.3 数据一致性设计

在引入缓存区之后, 需要保证数据在缓存和闪存

中的完整性和一致性, 本文进行以下设计.

写 I/O 数据时, 数据类型为以下几种情况之一:

1) 若数据为全新数据, 将其写入 write buffer, 此时数据只存在于 write buffer 中;

2) 若数据本身已存在于 write buffer 中, 则无效之前 write buffer 中的旧数据, 并将新数据追加写入 write buffer 中;

3) 若数据只存在于闪存中, 直接将闪存中的数据无效, 并将数据写入 write buffer 中;

4) 若数据在 read cache 中, 此时数据在 read cache 和闪存中一致, 需要同时无效掉 read cache 和闪存中的数据, 并将数据写入 write buffer 中.

读 I/O 数据时: 当数据已被预读取到 read cache 中时, 保留闪存中的备份, 这样在清除读缓存空间时无需将数据再次写闪存.

4.4 闪存管理模块设计

闪存管理模块主要用于管理闪存块的使用. 闪存块可以分为 3 类: 空闲块、当前使用块和待回收块. 空闲块是指其中还有空闲页可供写入数据的块, 当前使用块是指当前正在使用的块, 待回收块是已经写满数据且在回收链表中的块.

空闲块以二叉堆的结构进行管理, 以擦写次数为关键字, 这可以保证每次寻找新块时都选择擦写次数最少的块来写入数据.

当闪存中没有空闲块可写时, 就要进行垃圾回收操作. 为了获取最优的存储性能, 此处使用贪婪垃圾回收算法, 即每次都选取有效页最少的块进行回收, 这样不仅可以进行最少量的数据读取从而减少垃圾回收对存储性能的影响, 同时也能最大限度地清理出闪存空间来供新数据写入.

由于每个块内有 256 个页, 故用 257 条链表分别表示无效页数目从 0 到 256 的块, 每条链表的首尾元素由垃圾回收元数据信息表的 head 和 tail 来记录, 如图 10 所示, 当一个块内的某个页因为映射关系更新而被无效时, 该块的无效页数目就会增长, 此时就需要把它从旧的链表中移动到新的链表中, 这样每条链表都链接着具有相同无效页数目的块.

当要进行垃圾回收时, 从无效页最多的链表开始遍历, 直到找到一个含有无效页数目最多的块来进行垃圾回收. 垃圾回收的过程如下: 首先将回收块里面的有效数据统一搬运到内存空间, 然后转移到该晶圆的预留块内, 此时回收块内的有效数据转移到了预留块中, 预留块已经被使用成为了普通的块, 所以将回收块擦除之后作为新的预留块, 至此垃圾回收结束.

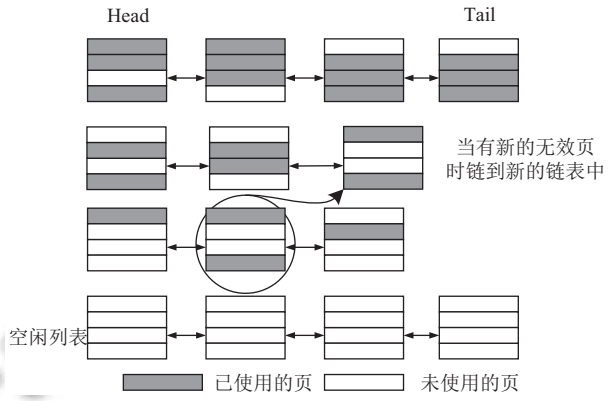


图 10 回收块链表示意图

为了充分利用 OpenSSD 多通道结构, 在某个晶圆触发垃圾回收时, 其余晶圆无需等待它垃圾回收处理结束再继续工作, 而是每个晶元都选取一个回收块进行垃圾回收, 因为读写擦除都可以并行操作, 这就相当于在原来回收 1 个块的时间内回收了 16 个块, 在单位时间腾出了更多的可用存储空间, 垃圾回收效率大大提高.

5 实验评估

5.1 测试参数设定

下述实验基于 OpenSSD 平台, 采用 fio 作为测试工具. fio 是一个非常灵活的 I/O 性能测试工具, 它可以通过多线程或多进程模拟各种 I/O 操作. fio 的工作过程很简单, 通过写一个 job 文件来描述要访真的 I/O 负载. 运行性能测试时, fio 从 job 文件读取这些配置参数, 并根据这些参数描述启动相关仿真进程 (线程).

在实际应用中, 通常以吞吐量和 IOPS 这两个指标作为测试基准. 吞吐量指数据传输的速度, 常用在顺序读写基准测试中; IOPS 表示每秒读写操作同样大小的数据块 (如 4 KB) 的次数, 常用于随机读写基准测试中.

fio 测试的主要相关设置参数如下: rw 参数根据测试项目改变, 随机写为 randwrite, 随机读为 randread, 顺序写为 write, 顺序读为 read; ioengine 分为同步和异步, 同步引擎可以通过多线程来增加请求密度, 异步引擎则同时受线程数和队列深度的影响, 此处统一选择同步引擎 sync; 因为要测试裸盘性能, 故跳过操作系统的

页缓存 (page buffer), 选择 Direct I/O 的方式, 设置相关配置项 direct 为 1; 多线程测试时根据实际效果选择 256 线程即可.

随机 I/O 性能测试的粒度大小设置为 4 KB; 由于驱动的限制, 顺序 I/O 性能测试的粒度设置为 120 KB 大小.

随机读写和顺序读写的 job 配置文件大同小异, 下面给出随机写的 job 文件:

```
[rand-write]
ioengine=sync
filename=/dev/pcissda
rw=randwrite
bs=${BS}
direct=1
size=4 g
numjobs=${NUMJOBS}
iodepth=1
runtime=300
group_reporting
```

5.2 优化前后性能测试和分析

(1) 随机写性能测试

随机写测试 IOPS 结果如图 11.

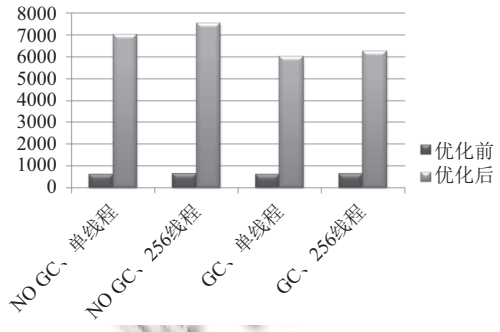


图 11 优化前后随机写 IOPS 性能对比

测试结果分析: 由图 11 可知, 优化后 FTL 对于小粒度随机写负载处理能力大幅度提高, 其中在不触发垃圾回收的情况下, 单线程 IOPS 性能由原来的 602 提升到 7007, 性能提升 11.6 倍, 多线程 IOPS 性能由原来的 636 提升到 7538, 性能提升 11.8 倍, 在触发垃圾回收的情况下单线程 IOPS 性能由原来的 596 提升到 6014, 性能提升 10 倍, 多线程 IOPS 性能由 631 提升到 6241, 性能提升 9.9 倍.

随机写性能的大幅度提升是因为优化后 FTL 充分利用了通道级并行和芯片级并行, 但是优化后性能只有理论 IOPS 的一半左右, 这是因为 CPU 运行代码会占用大量的时间, 从而增加了 I/O 请求的处理时间.

(2) 顺序写性能测试

顺序写带宽测试结果如图 12, 单位为 KB/s.

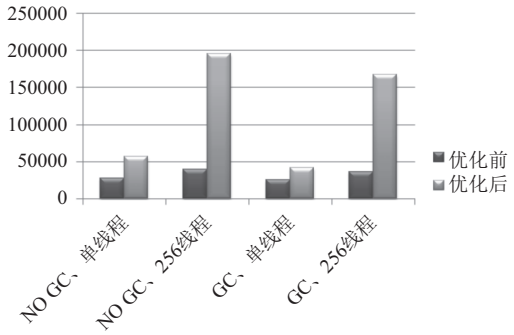


图 12 优化前后顺序写带宽性能对比

测试结果分析: 顺序写性能和随机写性能一样也得到大幅度提升, 根本原因在于充分利用了通道和芯片的并行传输. 其中不触发垃圾回收情况下单线程顺序写带宽由 27 908 KB/s 提升到 56 374 KB/s, 性能提升 2 倍, 256 线程下带宽由 39 090 KB/s 提升到 195 336 KB/s, 性能提升 5 倍; 触发垃圾回收情况下单线程带宽从 25 802 KB/s 提升到 41 385 KB/s, 性能提升 1.6 倍, 256 线程条件下带宽由 35 734 KB/s 提升到 167 748 KB/s, 性能提升 4.69 倍.

(3) 随机读性能测试

随机读 IOPS 测试结果如图 13.

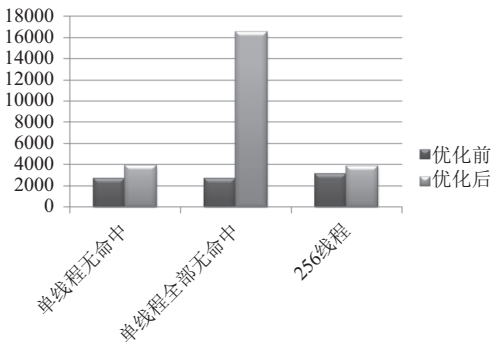


图 13 优化前后随机读 IOPS 性能对比

由图 13 可知, 在单线程无命中的情况下, 读 IOPS 由原来的 2700 提升到 3902, 全部命中的情况下

读 IOPS 由 2700 提升到 16 482, 这表示如果缓存命中的话, 读取速率将大幅度提升. 在 256 线程实际测试中, 读 IOPS 由 3148 提升到 3847, 提升幅度不大, 这是因为 fio 并没有模拟出具有冷热区分度的数据, 所以缓存命中不多, 性能提升不大.

(4) 与其余通用算法比较

在方案一的基础上, 本文选取了几种通用 FTL 算法, 针对小粒度随机写负载进行测试, 并与本文方案进行比较, 结果如图 14 所示.

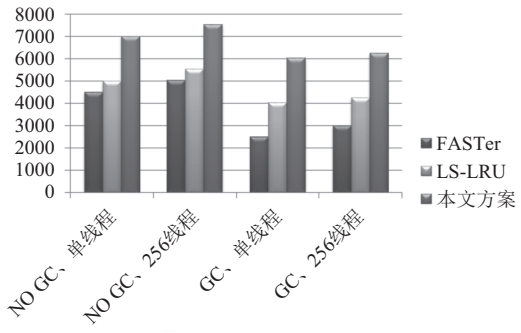


图 14 本文方案和几种通用 FTL 算法比较

由图 14 可知, 针对缓存冷热数据进行处理的通用 FTL 算法在 OpenSSD 上表现的实际性能与本文提出的方案存在一定差距, 主要原因是因为大量的 memcpy 操作抵消了缓存命中带来的好处.

6 总结

本文分析了 OpenSSD 平台的硬件特性, 考虑了影响其 I/O 读写性能的各类因素, 提出了一种适用于该平台的 FTL 方案. 通过与几种通用 FTL 算法的对比实验, 证明该方案有效提升了 OpenSSD 的系统性能, 特别地, 针对小粒度写 I/O 负载的性能提升最为明显.

参考文献

1 Van Houdt B. On the power of asymmetry and memory in flash-based SSD garbage collection. Performance Evaluation, 2016, (97): 1–15. [doi: 10.1016/j.peva.2015.11.002]

2 Hu Y, Jiang H, Feng D, *et al*. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. Proceedings of the 26th Symposium on Mass Storage Systems and Technologies. Incline Village, NV, USA. 2010. 1–12.

3 Lee JH, Jung BS. High performance NAND flash memory system with a data buffer. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 2013, E96.A(12): 2645–2651. [doi: 10.1587/transfun.E96.A.2645]

4 Chang LP, Liu YS, Lin WH. Stable greedy: Adaptive garbage collection for durable page-mapping multichannel SSDs. ACM Transactions on Embedded Computing Systems (TECS), 2016, 15(1): 13.

5 Wei DB, Deng LB, Zhang P, *et al*. A page-granularity wear-leveling (PGWL) strategy for NAND flash memory-based sink nodes in wireless sensor networks. Journal of Network and Computer Applications, 2016, (63): 125–139. [doi: 10.1016/j.jnca.2015.12.010]

6 OpenSSD Project. http://www.OpenSSD-project.org/wiki/The_ OpenSSD_Project. [2015-09-15]

7 Kim J, Seo S, Jung D, *et al*. Parameter-aware I/O management for Solid State Disks (SSDs). IEEE Transactions on Computers, 2012, 61(5): 636 –649. [doi: 10.1109/TC.2011.76]

8 He D, Wang F, Jiang H, *et al*. Improving hybrid FTL by fully exploiting internal SSD parallelism with virtual blocks. ACM Transactions on Architecture and Code Optimization (TACO), 2014, 11(4): 43.

9 Seo D, Shin D. Recently-evicted-first buffer replacement policy for flash storage devices. IEEE Transactions on Consumer Electronics, 2008, 54(3): 1228–1235. [doi: 10.1109/TCE. 2008.4637611]

10 Kwon SJ, Ranjitkar A, Ko YB, *et al*. FTL algorithms for NAND-type flash memories. Design Automation for Embedded Systems, 2011, 15(3-4): 191–224. [doi: 10.1007/s10617-011-9071-9]

11 Gupta A, Kim Y, Urgaonkar B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. ACM SIGARCH Computer Architecture News, 2009, 37(1): 229–240. [doi: 10.1145/2528521]