SSD Cache 加速技术白皮书

Version 1.0

2014.08

目录

1. 应用背景

1.1. 前言

在计算机系统中，CPU 中有 L1，L2、甚至有 L3 Cache；Linux 有 Page Cache，Mysql 有 BufferCache/QueryCache；IO 系统中 Raid 卡/磁盘也有 Cache；在大型互联网系统中，数据库前面一般也都有一层 MemCache。Cache 是容量与性能之间取得平衡的结果，以更低的成本，获得更高的收益，是系统设计应遵循的原则。

1.2. 应用分析

随着时间的流逝，网站、企业数据、用户数据一直不停的积累，从 MB，到 GB，再到 TB。如果将这些数据全部放到大容量的 SATA、SAS 传统的普通硬盘山，会发现性能（响应时间）不够；如果将这些数据全部放到 SSD，则会发现成本太高。即使能够将数据全部放到 SSD 上，经过一段时间，会发现大量的数据中只有一小部分是经常访问的（如热点新闻、热播电视剧等），而大部分的其他数据只有在很少的情况下才会被访问，或者是几乎不被访问。而这些很少被访问，或者几乎不被访问的数据放到 SSD 上，实际是一种浪费。

借助前面提及到的 Cache 概念，我们可以很自然的想到，利用传统的普通硬盘容量大的优点，将所有的数据存放在普通硬盘上；利用 SSD 的性能优势，将大量数据中的热点数据（经常访问的数据）存放在 SSD 上；这样既可以满足大数据量的需求，同时也可以满足对于热点数据的性能需求。

2. 基本原理

2.1. 基本架构

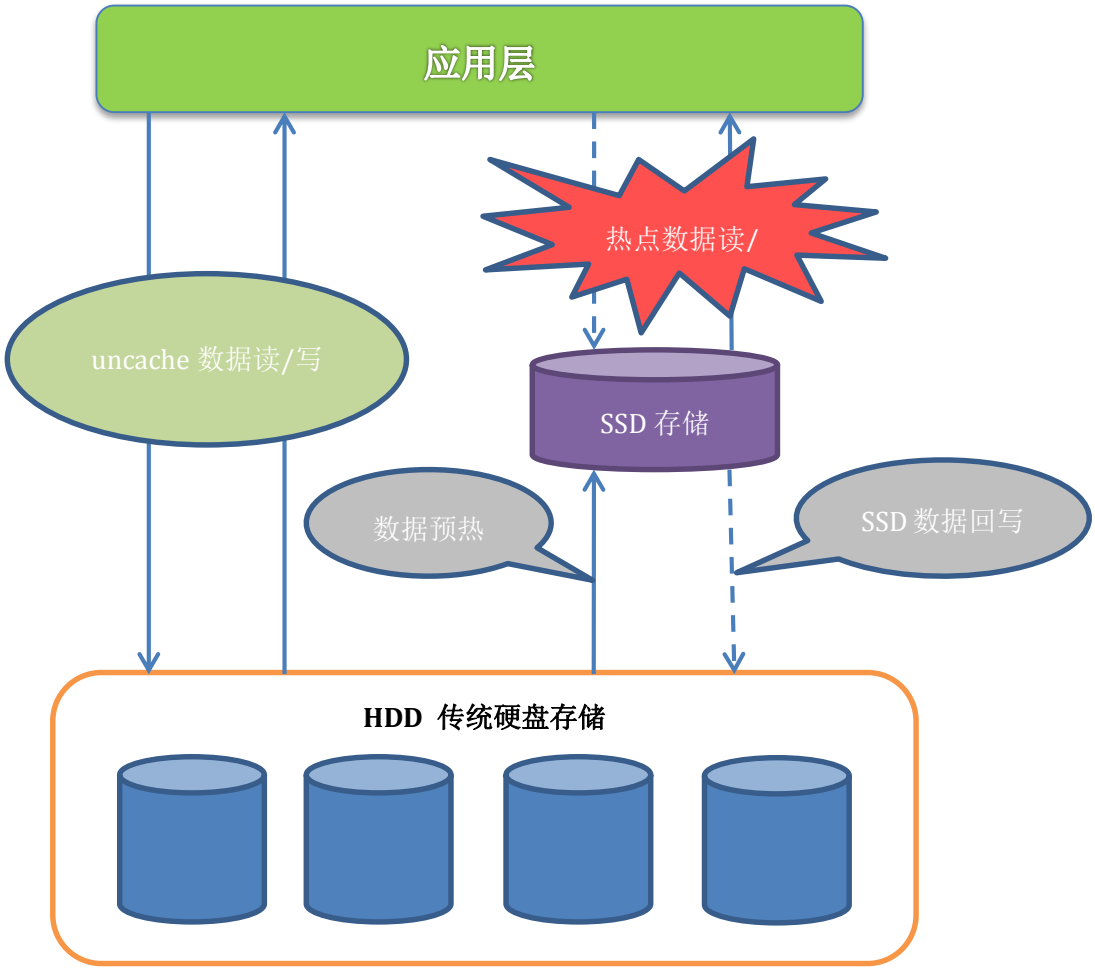

图 1.1 cache 架构图

从上面的架构图中，我们可以看出，将热点数据存储在 SSD 设备中，可以大大的提高数据读写的响应时间。然后，针对于具体的业务，如读多写少的互联网业务，或者是写多读少的业务，我们有对应的缓存模式。

2.2. 缓存模式

缓存模式基本上可以分为 3 种：write-through(透写模式),write-around(read-only,只读模式)，write-back(回写模式)。

2.2.1. write-through 模式

write-through 模式：对于读数据，全部缓存到高速设备 SSD 设备上；对于写操作，也缓存到高速设备 SSD 中，同时也写入到 HDD 中。这种模式比较适合读多写少，并且写的数据是最近访问的热点数据场景。这种模式因为写也同时写到 HDD 中，所以写的性能并不是最优，但是数据最安全。读写流程如下图：

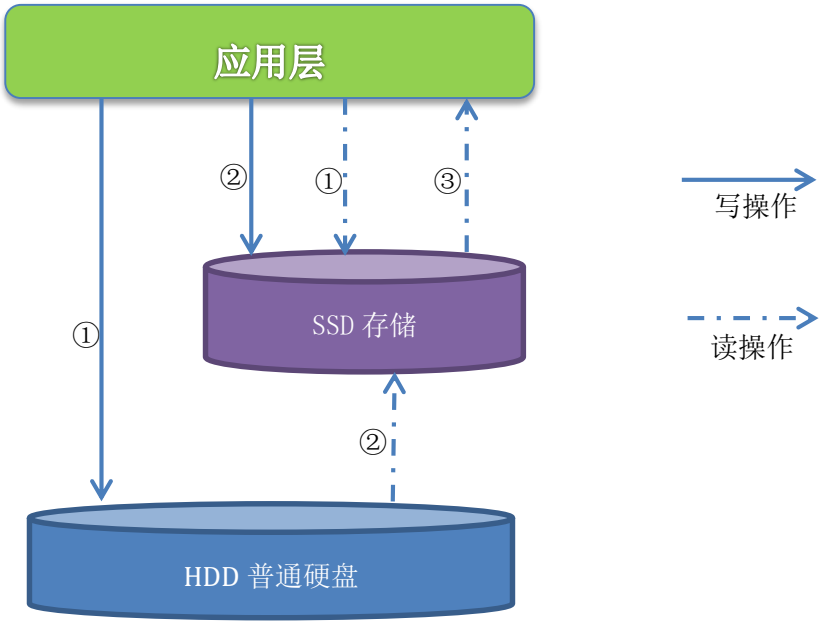

图 2.1 Write-Through 读写流程示意图

- 写流程：在 write-through 模式下，SSD 与 HDD 均需要写入；所以写①与写②操作都是需要的。
- 读流程：读流程是需要分为以下 2 种情况：

  1) 读的数据缓存在 SSD 设备中，数据可能是之前读或者写的数据（缓存到 SSD 中的）,此时可以直接将 SSD 数据返回用户程序。流程为读③；

  2) 读的数据没有缓存在 SSD 设备中，数据可能是第一次读取，也可能是之前读或

者写的数据但是已经被覆盖，此时，首先需要将数据从普通磁盘上读取到 SSD 中，然后从 SSD 中将数据返回。流程为读①→读②→读③。

2.2.2. write-around(read-only)模式

write-around(read-only)模式：只对读操作进行缓存，缓存的流程同 write-through 模式；写操作不缓存。这种模式非常适合读多写少的场景。读写流程如下：

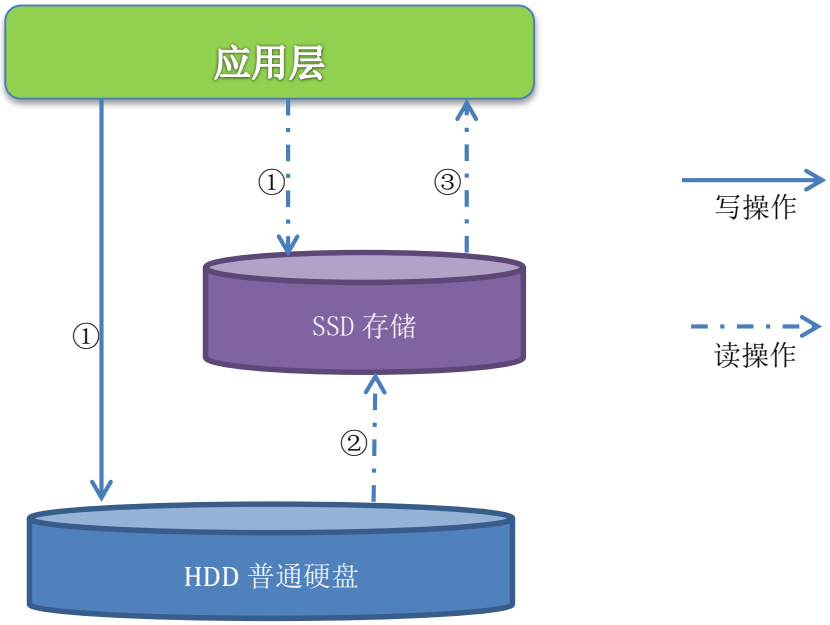

图 2.2 Write-around 读写流程示意图

- 写流程：在 write-around 模式下,写操作不缓存，则直接写 HDD 普通硬盘。即为：写①；
- 读流程：读流程是需要分为以下 2 种情况：
  1) 读的数据缓存在 SSD 设备中，数据可能是之前读或者写的数据（缓存到 SSD 中的）,此时可以直接将 SSD 数据返回用户程序。流程为读③；
  2) 读的数据没有缓存在 SSD 设备中，数据可能是第一次读取，也可能是之前读的数据但是已经被覆盖，此时，首先需要将数据从普通磁盘上读取到 SSD 中，然后从 SSD 中将数据返回。流程为读①→读②→读③。

2.2.3. write-back 模式

write-back 模式：对于读数据，全部缓存到高速设备 SSD(如果 SSD 空间不够，则根据替换策略，将 cache 之中的数据同步到普通硬盘上；对于写操作，同样也会缓存到 SSD 中；但是当 SSD 中的脏数据(与 HDD 中不一致的数据)达到一定阀值时候，会将脏数据同步到 SSD 中。

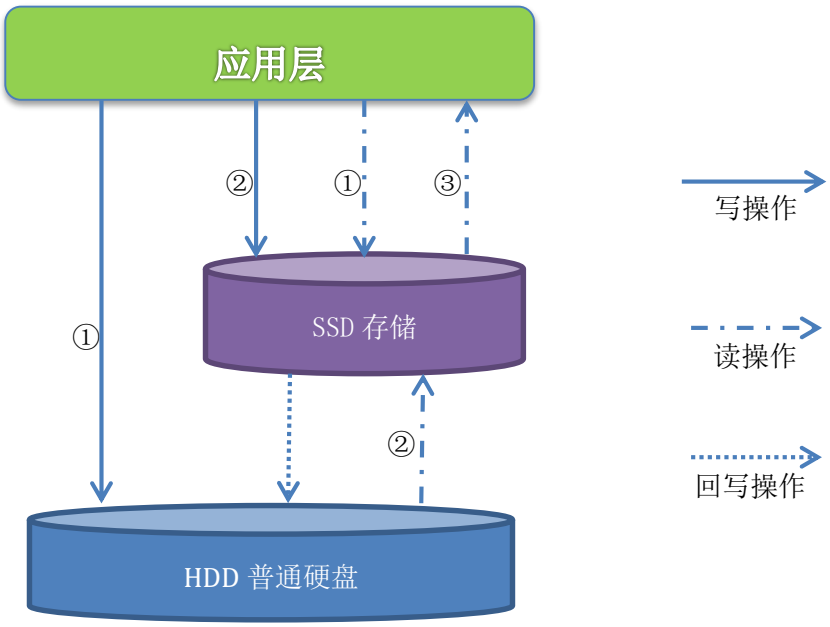

图 2.3 Write-back 读写流程示意图

- 写流程：写流程分为以下 3 种情况：
  1) 写的数据满足缓存要求，此时只需要将 SSD 中的数据改写为最新的数据即可。流程为写②；
  2) 写的数据不满足一些特定的要求（如不对齐等情况），此时则直接将数据写入到 HDD 普通硬盘中；流程为写①；
- 读流程：读流程是分为以下 2 种情况：
  1) 读的数据缓存在 SSD 设备中，数据可能是之前读或者写的数据（缓存到 SSD 中的）,此时可以直接将 SSD 数据返回用户程序。流程为读③；
  2) 读的数据没有缓存在 SSD 设备中，数据可能是第一次读取，也可能是之前读的数据但是已经被覆盖，或者是之前写的数据但是被同步到 HDD 普通硬盘上，

此时，首先需要将数据从普通磁盘上读取到SSD中，然后从SSD中将数据返回。流程为读①→读②→读③。

在 write-back 模式中，不得不提及到数据回写的过程。数据回写指的是将 SSD 中的脏数据刷新到 HDD 普通硬盘中，这个只有在 write-back 模式中才有。一般数据回写会有一些系统参数，如阀值（即脏数据达到一定的比率才会开始执行数据回写）；时效时间（数据超过一定时间无操作，则可以执行数据回写）；数据回写的速率（因为从 SSD 中奖数据写入到 HDD 中，速度较慢，需要考虑回写的速率限制）等参数。回写数据的时机与速率会很大程度上影响写的性能，如果阀值设置的非常高，则当到达阀值后，数据回写的速度会远远低于数据写入到 SSD 中的速度，这就会造成 SSD 中并没有充足的空间给新写入的数据，导致写会直接写入到 HDD 普通硬盘中，无法实现写加速。所以在设置数据回写相关的参数时，需要根据实际的情况适当的调整参数。

在数据回写的过程中，我们还需要注意一个问题，就是选择那些 block 回写到 HDD 普通硬盘上。一个好的置换策略，就是能够最大限度的保证热点数据存在 SSD 上，将不常访问的数据回写 HDD 普通硬盘上。具体的置换策略在下一小节会仔细介绍。

2.3. 置换策略

置换策略只有在 write-back 模式中才会使用，因为只有在 write-back 模式中，才会出现 SSD→HDD 回写数据的情况。通常使用的数据置换策略有如下几种：

- FIFO(First In First Out)：先进先出策略，这是一种最简单的置换策略。这种策略的实质就是总是选择在 SSD 中停留时间最长的 block 置换，即新进入 SSD 的 block, 先替换。理由是：最早被写入 SSD 中的 block，其不再被读/写的可能性最大。这种策略只是在按线性顺序访问地址空间时才是理想的，否则效率不高；
- LRU(Least Recently Used)：最近最久未使用策略。该策略的实质是，当需要置换 SSD 中的 block 时，选择在最近一段时间里最久没有使用过的 Block 予以置换。LRU 策略是与每个 Block 最后使用的时间有关的。由于该策略实现复杂，有一种 LRU 近似的策略 NUR(Not Recently Used)最近未使用算法；
- LFU(Least Frequently Used)：最少使用置换策略。该策略的实质是，当需要置换 SSD 中的 block 时，选择最近时间使用最少的 Block 替换。LFU 策略是与每个 Block

使用的次数相关的；

- RAND：随机置换策略。该策略的实质就是，当需要置换 SSD 中的 block 时，用软硬件的随机数产生 SSD 要替换 Block 的序号。

在实际使用过程中，用户可以根据具体的情况选择合适的置换策略，尽最大的可能性使热点数据最大程度的保留在 SSD 中，使 SSD 的性能优势发挥到最大！

3. 技术实现

目前将 SSD 作为普通硬盘 cache 的实现方式很多，下面只对两种实现方式 DM-Cache 与 IO 拦截进行进行说明。DM-Cache 具体的代表主要是 Flashcache；而 IO 拦截是先的代表则是 leadio.

3.1. dm-cache

在介绍 DM-Cache，我们首先来熟悉下 Linux 中的 Device mapper 技术。

Device mapper 是 Linux 2.6 内核中提供的一种从逻辑设备到物理设备的映射框架机制，在该机制下，用户可以很方便的根据自己的需要制定实现存储资源的管理策略。

Device mapper 在内核中是作为一个块设备驱动被注册的，它包含 3 个重要的对象概念。，mapped device、映射表、target device。

- Mapped device 是一个逻辑抽象，可以理解成为内核向外提供的逻辑设备；
- 映射表描述的是从 Mapped device 到 Target device 之间的映射关系。从 Mapped device 到一个 target device 的映射表由一个多元组表示，该多元组由表示 mapped device 逻辑的起始地址、范围、和表示在 target device 所在物理设备的地址偏移量以及 target 类型等变量组成。
- Target device 表示的是 mapped device 所映射的物理空间段，对 mapped device 所表示的逻辑设备来说，就是该逻辑设备映射到的一个物理设备。

Device mapper 中这三个对象和 target driver 插件一起构成了一个可迭代的设备树。如下图：

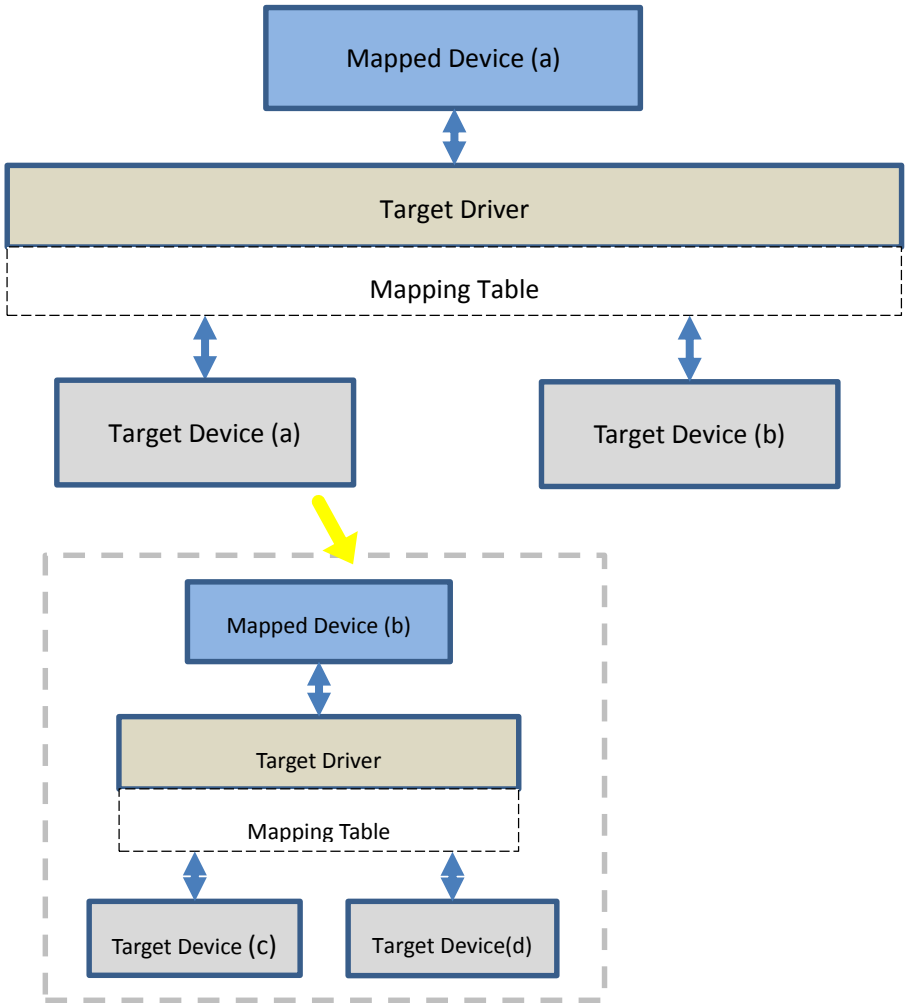

图 3.1 Device Mapper 内核对象映射层次

从上图可以看出，在该树型结构中的顶层根节点是最终作为逻辑设备向外提供的 mapped device，叶子节点是 target device 所表示的底层物理设备。最小的设备树由单个 mapped device 和 target device 组成。每个 target device 都是被 mapped device 独占的，只能被一个 mapped device 使用。一个 mapped device 可以映射到一个或者多个 target device 上，而一个 mapped device 又可以作为它上层 mapped device 的 target device 被使用，该层次在理论上可以在 device mapper 架构下无限迭代下去。

Device mapper 本质功能就是根据映射关系和 target driver 描述的 IO 处理规则，将 IO 请求从逻辑设备 mapped device 转发相应的 target device 上。

而 DM-Cache 正是利用了 Device Mapper 这一次原理。将 HDD 普通磁盘与 SSD 磁盘作为 DM 中的 Target Device，聚合成为一个虚拟设备（Cache 设备），同时将 HDD 普通磁盘与 SSD 磁盘 Block 进行映射；Dm-Cache 可以根据映射,可以将对 HDD 普通磁盘的操作转换

为对 SSD 设备的操作，从而实现将热点数据存储在 SSD 中的功能。HDD 普通磁盘对 SSD 设备 Block 的映射，可以为简单的线性映射，也可以采用 Hash 映射。

3.2. IO 拦截

首先我们需要了解下 Linux 下的 IO 流向，如下图：

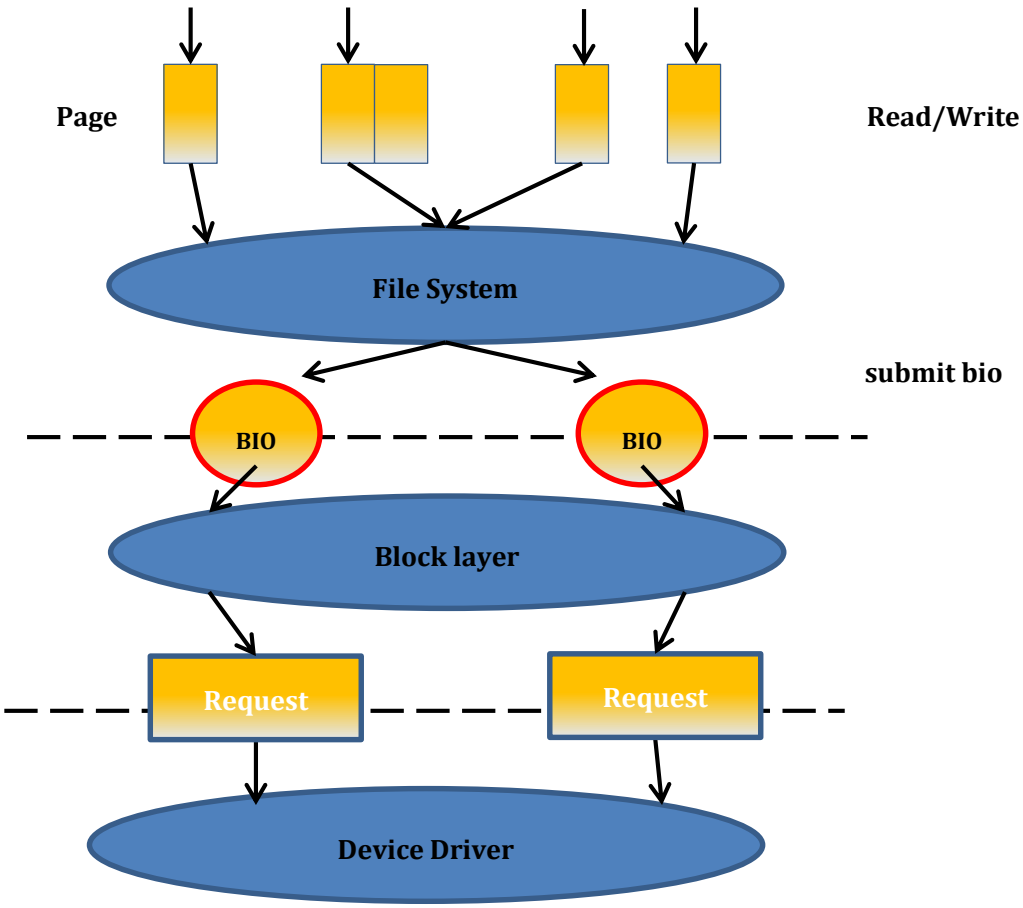

图 3.2 Linux IO 流

为了屏蔽不同文件系统的差异性，我们可以对 bio 请求拦截，在 Block Layer 实现某些分发函数的钩子函数，同时也需要将普通磁盘 Block 与 SSD 磁盘 Block 进行映射；根据映射规则以及操作类型，决定是否将普通磁盘操作转换为对 SSD 磁盘的操作。如在 write-around 模式中，则只需要对读操作进行处理：根据映射规则，首先查看 Block 是否存在 SSD 中，如果不存在，则从 HDD 中读取，并写入到 SSD 中，然后返回；如果已经存在 SSD 中，则直接将读请求转换为对 SSD 映射 Block 的读请求。

4.性能评测

这里选用了 Flashcache 作为 Cache 软件进行了测试。主要测试随机写与随机读两方面。

- **随机写测试**

  采用 Flashcache+pblaze3 测试效果如下：

数据量	热点数据	IOPS	I/O 带宽	状态	write hit%
50GB	10GB	107,899	866MB/s	8k 随机写	95

  同样配置下的，普通 HDD 8k 随机写的 BW=1859.4KB/s,IOPS=232

- **随机读测试**

  采用 Flashcache+pblaze3 测试效果如下：

数据量	热点数据	IOPS	I/O 带宽	状态	read hit%
50GB	10GB	210,000	1.50GB/s	8k 随机读	95

  同样配置下，普通 HDD 硬盘的 8k 随机读测试 BW=2331.8KB/s，IOPS=291

对比图如下：

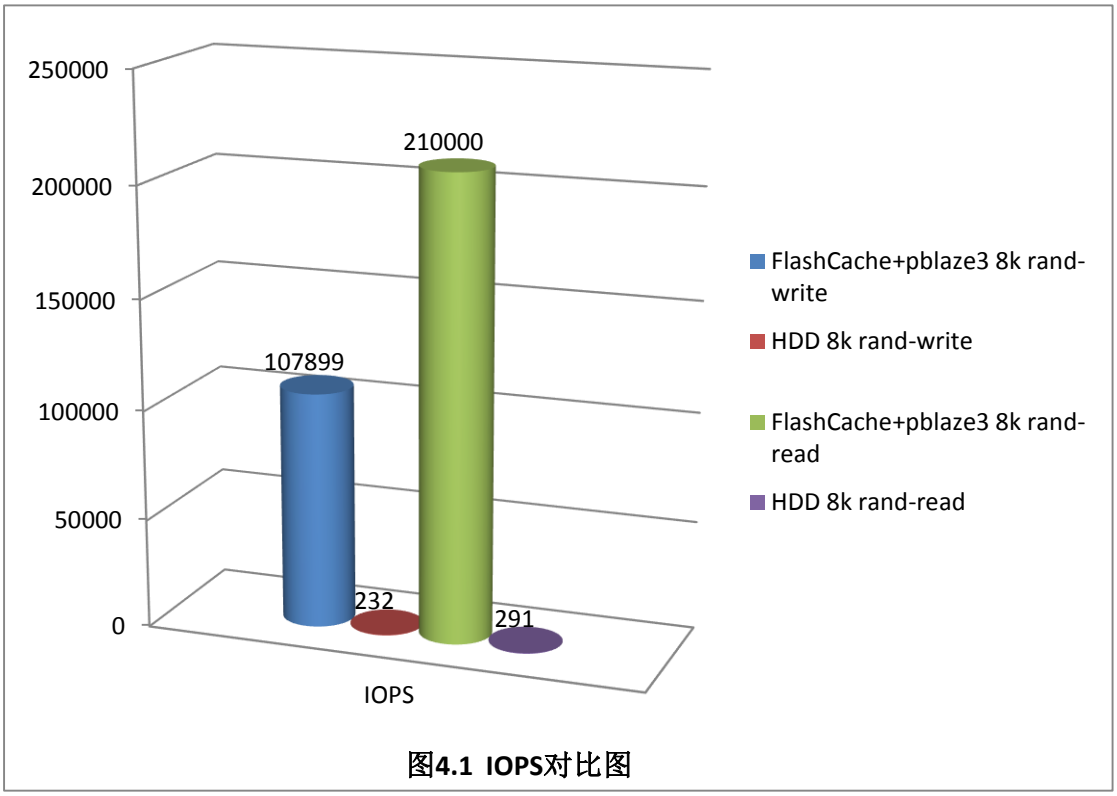

图4.1 IOPS对比图

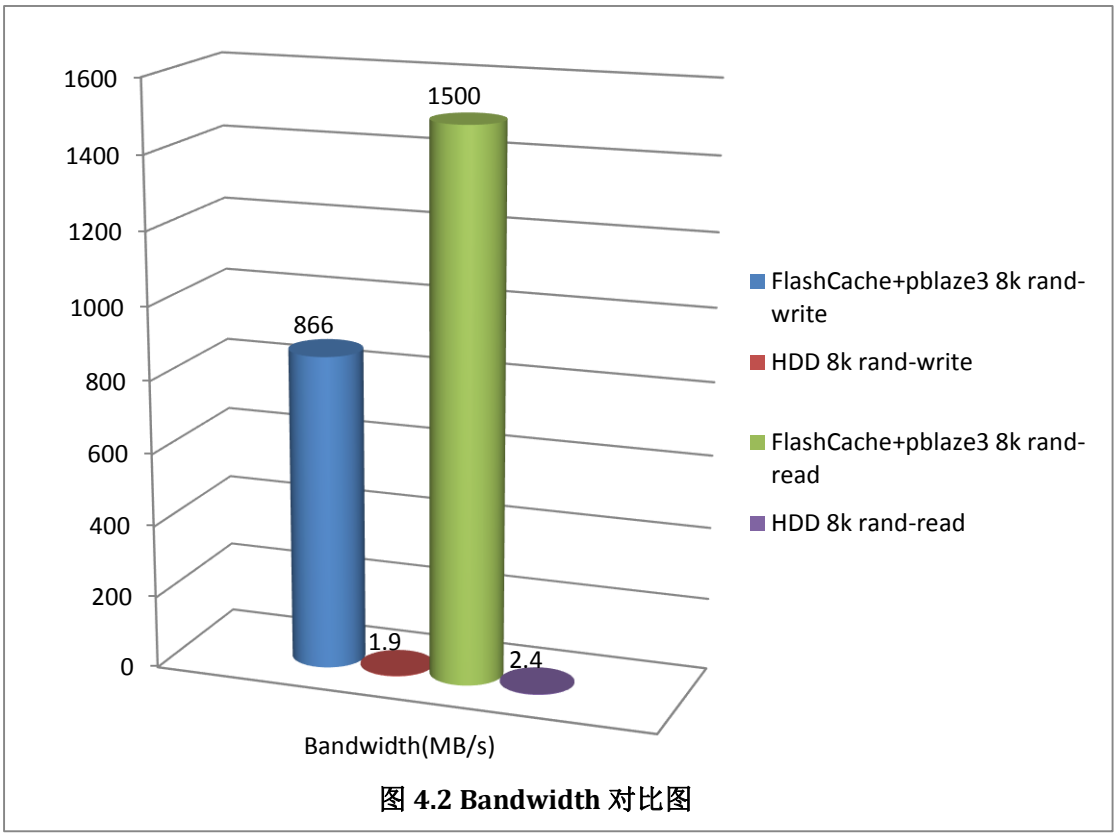

图 4.2 Bandwidth 对比图

从上述的数据中，我们可以非常清楚的看出采用了 cache 软件后，读/写性能的提升！

5. 最佳实践(flashcache)

目前有很多的 cache 软件，如 bcache,flashcache,leadio 等软件。这里选用 flashcache 软件，来说明在应用中的一些最佳实践。

flashcache 是由 Facebook 技术团队开发，最初是为加速 MySQL 设计的。Flashcache 是在 Linux 层面上的，它是一个非常不错的软件（准确的说是一个 Linux 的模块），可以动态加载。当前最稳定的版本是 v3.1.1，可以从 https://github.com/facebook/flashcache/中下载。

5.1. 安装 flashcache

在安装 flashcacache 之前，需要确认 Linux 系统中已经安装 kernel-devel 模块，并且与 linux 的 kernel 版本一致，否则安装 flashcache 会出现错误。

1) 将 flashcache-stable_v3.1.1.zip 放到任一目录下,解压。

unzip flashcache-stable_v3.1.1.zip

2) 如果是在 Centos6.4/6.5 或者是 Redhat6.4/6.5 系统,需要修改 flashcache 中的部分代码，否则编译出错。 修改处为 src/flashcache_conf.c 文件的 1749 和 1754 行。

1749 行修改为如下：

#elif ((LINUX_VERSION_CODE >= KERNEL_VERSION(3,0,0)) && (LINUX_VERSION_CODE < KERNEL_VERSION(3,9,0))) || (defined(RHEL_RELEASE_CODE) && (RHEL_RELEASE_CODE >= 1538) **&& (RHEL_RELEASE_CODE <= 1540)**)

1754 行修改为如下：

#elif (LINUX_VERSION_CODE >= KERNEL_VERSION(3,9,0)) ||

(defined(RHEL_RELEASE_CODE) && (RHEL_RELEASE_CODE >= 1541))

回到 flashcache 根目录 make && make install

3) 进入 src 目录 insmod flashcache.ko

5.2. 创建 flashcache 设备

创建 flashcache 设备是利用 flashcache_create 命令创建的。

flashcache_create [-v] -p back|around|thru [-s cache size] [-b block size] cachedevname ssd_devname disk_devname

参数说明：

-p: 创建 cache 设备采用的模式；

(back---write back,around---write around,thru---write through)

-b: flashcache 设备的块大小。默认是 4k.

(4k 对于大多数应用是最合适的选择)

假设：/dev/memdiska 是 ssd 加速卡，/dev/sda3 是需要加速的硬盘。实例如下：

flashcache_create -p back –b 4k cachedev /dev/memdiska1 /dev/sda3

5.3. 加载 flashcache 设备

对于不同的缓存模式，加载方式是不同的。如果是 write-around 和 write-through 模式，则不需要特殊的加载命令，只需要重复执行 flashcache_create 命令即可；

对于 write-through 模式需要执行 flashcache_load 命令。

flashcache_load ssd_devname [cachedev_name]

具体实例如下：

/sbin/flashcache_load /dev/memdiska1

通常我们为了保证系统重启之后就可以加载了对应的 flashcache 设备，我们可以将对应的加载命令写到/etc/rc.local 文件中。

5.4. 删除 flashcache 设备

对于不同的缓存模式，删除 flashcache 设备也不是不同的。对于 write-around 模式与 write-through 模式，可以执行 dmsetup remove 命令删除 flashcache 设备。

dmsetup remove cachedev

其实也可以不执行该命令，系统在 shutdown 时会自动将 flashcache 设备删除。

对于 write-back 模式，因为在 SSD 设备中含有一些元数据以及脏数据，删除 flashcache 设备需要执行如下两个步骤：

1) 执行 dmsetup remove 命令,删除 flashcache 设备，并回写脏数据

dmsetup remove /dev/mapper/cachedev

如果之前进行过大量的写操作，此时可能会耗费较长时间，需要将 cache 中的脏数据回写到 HDD 上。但是如果不需要将 cache 中的脏数据回写到 HDD 上，可以设置 flashcache 的 sysctl 参数 fast_remove.

2) **执行 flashcache_destroy 命令，删除加速卡上存储的 metadata.**

flashcache_destroy /dev/memdiska1

5.5. 监控 flashcache 设备

flashcache 设备在运行期间，我们也需要随时监控对其运行状态，可以根据实际情况进行参数优化。对 flashcache 的监控可以有以下两种方式：

- **通过 flashstat 监控**

flashstat 是 flashcache 提供的工具，在终端中执行 flashstat 命令，可以获取实时的 cache 读写的情况。

flashstat 的用法：

指定监控时间间隔（-i，默认为 1s）,监控次数（-c 默认 0 表示一直监控），Flashcache 设备(-d 默认为/dev/mapper/cachedev)即可。如果需要重定向到文件，则建议关闭 ANSI 颜色显示，使用-n 选项即可。

需要注意的，flashstat 只能监控 write-back 模式下的 cache 设备，监控的 flashcache 默认设备名是/dev/mapper/cachedev，如果在创建 flashcache 设备时候，创建的 cache 设备并不是 cachedev，则需要对 flashstat 脚本进行修改，将其中 cache 设备名修改为创建时的 cache 设备名；

- **通过 dmsetup status 获取信息**

在终端执行 dmsetup status，可以获取 cache 中读写的次数，以及读写命中 cache 的情况。

如果想清空之前的记录信息，可以通过 flashcache 的 sysctl 参数 zero_stats 修改，重新记录。

5.6. 数据一致性

作为存储的一种方案，我们就需要考虑数据的一致性。这个是与缓存模式紧紧关联的。

write-through 模式，由于是同时写 SSD 与 HDD，这样是保证了数据一致性的。即使是异常断电或者 SSD 设备出现故障，HDD 普通磁盘中的数据同样也会保证数据的一致性；

write-around 模式，由于只是进行读缓存，这样在各种情况下，数据都不会出现错误，能够保证数据的一致性。

write-back 模式，由于是将写操作先缓存在 SSD 设备中，在出现关机或者是异常断电的情况下，数据仍保留在 SSD 设备中，在再次加载 cache 设备的时候，能够将 SSD 设备中的数据回写到 HDD 普通硬盘上，这样也能够保证数据的一致性；但是如果 SSD 设备出现故障，在 HDD 普通磁盘中的数据可能会出现错误，无法保证数据的一致性；其实这种情况下，我们可以用两块 SSD 设备做 Raid1，降低出现问题的概率。而且，我们还可以在系统闲暇时，将 SSD 设备中的脏数据回写到 HDD 普通磁盘中，这样也能减少出现问题的概率。

5.7. 参数优化

根据不同的场景，我们可以修改 flashcache 的一些配置参数，使性能达到最优。flashcache 的配置参数可以通过命令 sysctl –a| grep flashcache 查看。设置对应的参数可以直接 sysctl key=value;如：sysctl dev.flashcache.memdiska1+sda1.do_sync=1

```
[root@localhost ~]# sysctl -a | grep flashcache
dev.flashcache.memdiska1+sda1.io_latency_hist = 0
dev.flashcache.memdiska1+sda1.do_sync = 0
dev.flashcache.memdiska1+sda1.stop_sync = 0
dev.flashcache.memdiska1+sda1.dirty_thresh_pct = 20
dev.flashcache.memdiska1+sda1.max_clean_ios_total = 4
dev.flashcache.memdiska1+sda1.max_clean_ios_set = 2
dev.flashcache.memdiska1+sda1.do_pid_expiry = 0
dev.flashcache.memdiska1+sda1.max_pids = 100
dev.flashcache.memdiska1+sda1.pid_expiry_secs = 60
dev.flashcache.memdiska1+sda1.reclaim_policy = 0
dev.flashcache.memdiska1+sda1.zero_stats = 0
dev.flashcache.memdiska1+sda1.fast_remove = 0
dev.flashcache.memdiska1+sda1.cache_all = 1
dev.flashcache.memdiska1+sda1.fallow_clean_speed = 2
dev.flashcache.memdiska1+sda1.fallow_delay = 900
dev.flashcache.memdiska1+sda1.skip_seq_thresh_kb = 0
dev.flashcache.memdiska1+sda1.clean_on_read_miss = 0
dev.flashcache.memdiska1+sda1.clean_on_write_miss = 0
dev.flashcache.memdiska1+sda1.lru_promote_thresh = 2
dev.flashcache.memdiska1+sda1.lru_hot_pct = 75
dev.flashcache.memdiska1+sda1.new_style_write_merge = 0
```

1) 过滤大量顺序读**(skip_seq_thresh_kb)**

因为 SSD 设备最大的优势是在于随机读/写，在大量顺序读的情况下，如果数据在 SSD 中没有命中，这时候的性能会比直接读取 HDD 普通硬盘更差， HDD 普通机械硬盘在大量顺序读的情况下，更有优势。

这时候可以设置参数**skip_seq_thresh_kb**.skip_seq_thresh_kb 参数的含义则是如果超过这个参数的值，则 flashcache 会忽略该数据，直接交给 HDD 普通硬盘处理。用户可以根据场景中顺序读的大小，设置该参数，过滤顺序读，使性能优化；

2) 替换策略的选择**(reclaim_policy)**

flashcache 只实现了两种替换策略，FIFO 与 LRU 算法。正如第二章所提到的，如果是在大量的线性读写操作的场景下，使用 FIFO 更加合适；而在随机读写的场景下，选择 LRU 则更合适；挑选替换算法，是通过参数 **reclaim_policy** 调整的。reclaim_policy=0 则表示选择的是 FIFO 策略，如果 reclaim_policy=1,则表示选择的是 LRU 策略。

3) write-back 模式下脏数据的比率**(dirty_thresh_pct)**

在 write-back 模式下，因为对写操作的缓存，在 SSD 中存在与 HDD 不一致的脏数据，当 SSD 中达到脏数据的阀值时，SSD 就会将脏数据回写到 HDD 普通硬盘中。默认的脏数据比率是 20%，如果存在短期的大量写操作，可以将脏数据比率阀值适当的提高，使更多的写操作发生在 SSD 上；而在没有写操作的情况下，可以调整 dirty_thresh_pct 值，使数据回写到 HDD 上，以便为以后的读写操作提供更多可用的 cache 空间。如果是一致持续的写操作,dirty_thresh_pct 设置要根据 cache 的空间而定，一定不要太低也不要太高，太低的会造成持续不断的回写操作，导致性能下降；而设置的太高，在前期，写操作性能会很好，但是一旦达到阀值之后，由于设置的值太高，会导致 SSD 中的脏块数据较大，而回写的速度较慢，这会导致写的性能立即下降，抖动及其厉害。

4) write-back 模式下脏数据的时效性**(fallow_deley)**

write-back 模式下，数据回写的情况不仅仅是在脏数据达到设定的阀值之后才会出现，在脏数据达到一定的时效性时候，也会出现。fallow_delay 设置的正是脏数据 Block 的有效时间；如果想使写的脏数据尽量的保存在 SSD 中，则可以设置 fallow_delay=0，则不会检查脏数据的时效性；如果是为了保证 SSD 中有持续可用的空间，同时保证 SSD 与 HDD 数据的一致性，可以设置 fallow_delay 参数。具体的值还需要根据实际情况而定。

5) write-back 模式下 clean 操作的限制**(max_clean_ios_set 与 max_clean_ios_total)**

每次执行 clean(数据回写)操作时候，处理的 cell set 需要满足<=max_clean_ios_set,而清理的总 cell 是需要<=max_clean_ios_total;这两数据会影响每次 clean 的数据量；flashcache 中的 cell set 默认是 512。当系统闲暇时，并且 SSD 中的脏数据量很多，我们可以将这两个参数提高，加速数据回写的速度；而在大量读写的时候，可以将参数设置的低一些，保证正常的操作。

6) write-back 模式下强制执行/停止回写操作**(do_sync 与 stop_sync)**

通过 sysctl 设置 do_sync=1,则可以强制执行数据回写操作；而设置 stop_sync=1，则强制停止回写操作；如果系统处于空闲状态，可以设置 do_sync，使 SSD 数据回写到普通硬盘上，保持数据一致性；而一旦出现大量读写操作，则可以设置 stop_sync 停止回写操作，以免影响正常性能。

7) flashcache 设置 black list 与 white list **(cache_all)**

flashcache 可以屏蔽默写特定进程的读写操作；如果 cache_all 设置为 1，则 flashcache 对 blacklist 中进程的读写不缓存其他进程缓存；如果 cache_all 设置为 0，则 flashcache 对 whitelist 中的进程读写缓存，其他进程不缓存；flashcache 增加 blacklist/whitelist,是通过 ioctl 操作完成的。

FLASHCACHEADDBLACKLIST：增加 pid 到 blacklist;

FLASHCACHEDELBLACKLIST: 从 blacklist 中删除对应的 pid;

FLASHCACHEDELALLBLACKLIST:清空 blacklist;

FLASHCACHEADDWHITELIST: 增加 pid 到 whitelist;

FLASHCACHEDELWHITELIST: 从 whitelist 中删除对应的 pid;

FLASHCACHEDELALLWHITELIST: 清空 whitelist;