论文阅读与前期工作总结

姓名：张昊,张浩轩, 严萌,秦晨曦

学号：17343148,17343149, 17343134,17343096

前期工作

使用示意图展示普通文件IO方式(fwrite等)的流程，即进程与系统内核，磁盘之间的数据交换如何进行？为什么写入完成后要调用fsync？

```
application buffer
    |
clib buffer
    |
page cache
    |
IO queue
    |
  drive
    |
disk cache
    |
  disk
```

1. 应用程序先将内容写入应用程序的application buffer
2. 调用fwrite将内容写入CLib buffer
3. 然后此时flush函数把数据从CLib buffer 拷贝到page cache中
4. 从page cache刷新到磁盘上可以通过调用fsync函数 最后调用fsync函数是为了将文件从内存同步到硬盘，如果不调用的话，程序意外崩溃，数据就会发生丢失。

简述文件映射的方式如何操作文件。与普通IO区别？为什么写入完成后要调用msync？文件内容什么时候被载入内存？

1. Linux提供了内存映射函数mmap, 它把文件内容映射到一段内存上(准确说是虚拟内存上), 通过对这段内存的读取和修改, 实现对文件的读取和修改
    1. 进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域
    2. 调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系
    3. 进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝
2. 与普通io的区别
    1. 文件映射没有频繁的数据拷贝，只有发生缺页时才会有数据拷贝，由于mmap()将文件直接映射到用户空间，所以中断处理函数根据这个映射关系，直接将文件从硬盘拷贝到用户空间，只进行了一次数据拷贝
    2. 常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝

3. 因为此时数据还保留在内存上，掉电之后数据丢失，因此需要保存在硬盘上
4. 调用msync()函数(显示同步)时和结束时调用munmap()后自动调用msync()函数进行同步

参考Intel的NVM模拟教程模拟NVM环境，用fio等工具测试模拟NVM的性能并与磁盘对比（关键步骤结果截图）。

首先建立nvm环境,然后是将nvm挂载到了/mnt/pmemdir目录下

```
mount -o dax /dev/pmem0 /mnt/pmemdir
```

然后安装fio,在ubuntu下的命令是:

```
apt-get install fio
```

首先我们测试nvm盘的速度，这里是指令

```
sudo fio -filename=/mnt/pmemdir/test -direct=1 -iodepth 1 -thread -
rw=randrw -ioengine=psync -bs=16k -size=200M -numjobs=10 -runtime=1000 -
group_reporting -name=fiotest
```

这里比较重要的参数是rw=randrw读写都是随即读写和-size=200M测试文件大小是200M.首先运行在nvm环境下,做了三组测试:

- 随机读

```
     lat (nsec): min=1917, max=23998k, avg=10039.44, stdev=283421.54
    clat percentiles (usec):
     |  1.00th=[    3],  5.00th=[    4], 10.00th=[    4], 20.00th=[    5],
     | 30.00th=[    5], 40.00th=[    5], 50.00th=[    5], 60.00th=[    5],
     | 70.00th=[    5], 80.00th=[    5], 90.00th=[    6], 95.00th=[    6],
     | 99.00th=[   11], 99.50th=[   11], 99.90th=[   20], 99.95th=[   61],
     | 99.99th=[16057]
  lat (usec)   : 2=0.01%, 4=19.16%, 10=79.62%, 20=1.12%, 50=0.05%
  lat (usec)   : 100=0.01%, 250=0.01%, 500=0.01%
  lat (msec)   : 4=0.01%, 10=0.01%, 20=0.02%, 50=0.01%
  cpu          : usr=4.33%, sys=36.92%, ctx=111, majf=0, minf=44
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwt: total=128000,0,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=11.6GiB/s (12.4GB/s), 11.6GiB/s-11.6GiB/s (12.4GB/s-12.4GB/s), io=20
00MiB (2097MB), run=169-169msec

Disk stats (read/write):
  pmem0: ios=0/0, merge=0/0, ticks=0/0, in_queue=0, util=0.00%
```

- 随机写

```
     lat (usec): min=2, max=28019, avg=11.67, stdev=291.81
    clat percentiles (usec):
     |  1.00th=[    4],  5.00th=[    4], 10.00th=[    4], 20.00th=[    4],
     | 30.00th=[    5], 40.00th=[    5], 50.00th=[    5], 60.00th=[    6],
     | 70.00th=[    6], 80.00th=[    6], 90.00th=[    8], 95.00th=[   11],
     | 99.00th=[   11], 99.50th=[   11], 99.90th=[   86], 99.95th=[ 3064],
     | 99.99th=[16057]
  lat (usec)   : 4=21.44%, 10=72.29%, 20=6.08%, 50=0.08%, 100=0.02%
  lat (usec)   : 250=0.02%, 500=0.01%, 750=0.01%, 1000=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%, 10=0.01%, 20=0.03%, 50=0.01%
  cpu          : usr=9.80%, sys=37.16%, ctx=401, majf=0, minf=0
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwt: total=0,128000,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=9524MiB/s (9986MB/s), 9524MiB/s-9524MiB/s (9986MB/s-9986MB/s), io=20
00MiB (2097MB), run=210-210msec

Disk stats (read/write):
  pmem0: ios=0/0, merge=0/0, ticks=0/0, in_queue=0, util=0.00%
```

- 混合随机读写

```
     |  1.00th=[    4],  5.00th=[    4], 10.00th=[    5], 20.00th=[    5],
     | 30.00th=[    5], 40.00th=[    6], 50.00th=[    6], 60.00th=[    6],
     | 70.00th=[    6], 80.00th=[    7], 90.00th=[    8], 95.00th=[   11],
     | 99.00th=[   12], 99.50th=[   13], 99.90th=[   90], 99.95th=[ 2638],
     | 99.99th=[14484]
  lat (usec)   : 4=14.87%, 10=81.10%, 20=3.78%, 50=0.10%, 100=0.05%
  lat (usec)   : 250=0.02%, 500=0.01%, 750=0.01%, 1000=0.01%
  lat (msec)   : 2=0.01%, 4=0.01%, 10=0.02%, 20=0.02%, 50=0.01%
  cpu          : usr=6.99%, sys=37.28%, ctx=429, majf=0, minf=0
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwt: total=63811,64189,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=4553MiB/s (4774MB/s), 4553MiB/s-4553MiB/s (4774MB/s-4774MB/s), io=99
7MiB (1045MB), run=219-219msec
  WRITE: bw=4580MiB/s (4802MB/s), 4580MiB/s-4580MiB/s (4802MB/s-4802MB/s), io=10
03MiB (1052MB), run=219-219msec

Disk stats (read/write):
  pmem0: ios=0/0, merge=0/0, ticks=0/0, in_queue=0, util=0.00%
```

然后我们测试自己的硬盘，由于我电脑是SSD，所以速度较快.

- 随机读

```
     |  1.00th=[  247],  5.00th=[  281], 10.00th=[  302], 20.00th=[  338],
     | 30.00th=[  379], 40.00th=[  416], 50.00th=[  457], 60.00th=[  502],
     | 70.00th=[  553], 80.00th=[  627], 90.00th=[  725], 95.00th=[  824],
     | 99.00th=[ 1037], 99.50th=[ 1106], 99.90th=[ 1303], 99.95th=[ 1401],
     | 99.99th=[ 2737]
   bw (  KiB/s): min=31360, max=33408, per=10.01%, avg=32369.33, stdev=367.74, s
amples=120
   iops        : min= 1960, max= 2088, avg=2023.08, stdev=22.98, samples=120
  lat (usec)   : 250=1.25%, 500=58.31%, 750=31.86%, 1000=7.24%
  lat (msec)   : 2=1.32%, 4=0.01%, 10=0.01%
  cpu          : usr=0.49%, sys=2.83%, ctx=128072, majf=0, minf=40
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwt: total=128000,0,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=316MiB/s (331MB/s), 316MiB/s-316MiB/s (331MB/s-331MB/s), io=2000MiB
(2097MB), run=6331-6331msec

Disk stats (read/write):
  sda: ios=124259/4, merge=0/4, ticks=0/0, in_queue=60112, util=98.64%
```

- 随机写

```
     | 30.00th=[  709], 40.00th=[  840], 50.00th=[  865], 60.00th=[  930],
     | 70.00th=[  947], 80.00th=[  971], 90.00th=[ 1123], 95.00th=[ 1287],
     | 99.00th=[ 1860], 99.50th=[ 4293], 99.90th=[11469], 99.95th=[24249],
     | 99.99th=[26084]
   bw (  KiB/s): min= 9760, max=28704, per=10.04%, avg=18118.16, stdev=3554.04,
samples=219
   iops        : min=  610, max= 1794, avg=1132.31, stdev=222.13, samples=219
  lat (usec)   : 50=0.01%, 100=6.02%, 250=0.54%, 500=0.55%, 750=24.34%
  lat (usec)   : 1000=51.81%
  lat (msec)   : 2=15.82%, 4=0.40%, 10=0.36%, 20=0.09%, 50=0.07%
  cpu          : usr=0.53%, sys=5.95%, ctx=255745, majf=0, minf=0
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwt: total=0,128000,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
  WRITE: bw=176MiB/s (185MB/s), 176MiB/s-176MiB/s (185MB/s-185MB/s), io=2000MiB
(2097MB), run=11350-11350msec

Disk stats (read/write):
  sda: ios=0/125931, merge=0/6, ticks=0/408, in_queue=9016, util=80.24%
```

- 混合随机读写

```
     | 70.00th=[ 1762], 80.00th=[ 1975], 90.00th=[ 2278], 95.00th=[ 2540],
     | 99.00th=[ 3130], 99.50th=[ 3490], 99.90th=[ 6980], 99.95th=[10421],
     | 99.99th=[25822]
   bw (  KiB/s): min= 2789, max= 6957, per=10.02%, avg=5205.89, stdev=707.69, sa
mples=390
   iops        : min=  174, max=  434, avg=325.26, stdev=44.27, samples=390
  lat (usec)   : 100=4.64%, 250=4.06%, 500=0.35%, 750=1.71%, 1000=7.17%
  lat (msec)   : 2=59.90%, 4=21.76%, 10=0.34%, 20=0.02%, 50=0.04%
  cpu          : usr=0.53%, sys=3.78%, ctx=256616, majf=0, minf=0
  IO depths    : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >=64=0.0%
     submit    : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     complete  : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >=64=0.0%
     issued rwt: total=63811,64189,0, short=0,0,0, dropped=0,0,0
     latency   : target=0, window=0, percentile=100.00%, depth=1

Run status group 0 (all jobs):
   READ: bw=50.4MiB/s (52.9MB/s), 50.4MiB/s-50.4MiB/s (52.9MB/s-52.9MB/s), io=99
9MiB (1045MB), run=19768-19768msec
  WRITE: bw=50.7MiB/s (53.2MB/s), 50.7MiB/s-50.7MiB/s (53.2MB/s-53.2MB/s), io=10
03MiB (1052MB), run=19768-19768msec

Disk stats (read/write):
  sda: ios=63436/63896, merge=0/11, ticks=396/0, in_queue=28636, util=85.39%
```

使用PMDK的libpmem库编写样例程序操作模拟NVM（关键实验结果截图，附上编译命令和简单样例程序）。

安装PMDK的步骤很简单，基本就是如下几步:

```
git clone https://github.com/pmem/pmdk.git
cd pmdk
make
sudo make install
```

make install 会把.so文件安装到/usr/lib的系统目录下,会把头文件安装到/use/local/include下，如果不加sudo权限不够
下面是**编译样例** 这里我们编译的是pmdk/src/examples/libpmem/full_copy.c，使用如下命令

```
ubuntu@ubuntu-TM1604:~/DBMS$ gcc full_copy.c -o full_copy-pthread -lpmem
ubuntu@ubuntu-TM1604:~/DBMS$ ls
DataBase  full_copy.c  full_copy-pthread  homework  pmdk
```

这里需要链接两个动态链接库，分别是多线程的动态链接库和libpmem.so文件,然后我们测试一下

```
ubuntu@ubuntu-TM1604:~/DBMS$ sudo ./full_copy-pthread full_copy-pthread /mnt/pme
mdir/full
ubuntu@ubuntu-TM1604:~/DBMS$ ls /mnt/pmemdir/
full
```

论文阅读

总结一下本文的主要贡献和观点(500字以内)(不能翻译摘要)。

首先背景是随着科技的发展，出现了许多新型内存技术，不同于闪存，SCM它具有非易失性的特性， 具备超越闪存的潜力。然而对于SCM，写操作会慢于读操作，所以之前主要以闪存为基础而设计 的b-树会不再适用于SCM，因此需要重新设计一种树来利用SCM的功能。在此之前已经有了CDDS b树 ，wBTree,NV-Tree,bsigut,但在内存泄漏和数据恢复等方面表现得不尽如人意。而本文中所提出的 FBTree却是利用SCM来实现良好的数据结构性能。

FBTree的设计原则有四条：

- 利用指纹识别，降低搜索时间，提高性能。
- 利用在SCM中存储主数据，在DRAM中存储非主数据，降低在访问数据时所需要的时间，同时这也导致了FBTree能够快速的恢复。
- 通过对不同状态选择不同的并发方案来达到有选择并发性，通过选择性的执行工作，防止HTM在并发时执行其他的操作。
- 良好的编程模型。其中数据的回复和内存泄漏问题任然没有很好的得到解决。

通过测验，FBTree的内存使用更少，数据恢复时间更快，性能开销更小，同时FBTree对于SCM的高延迟具有较好的弹性，在数据恢复方面也保证了不会丢失信息。

SCM硬件有什么特性？与普通磁盘有什么区别？普通数据库以页的粒度读写磁盘的方式适合操作SCM吗？

SCM硬件有什么特性?

- SCM(Storage Class Memory) 存储类内存, 也叫做**persistent memory,** 它最大的特点就是 **persistent**, 也就是非易失性, 通俗来讲就是断电后原有的数据还在, 并不会像RAM一样丢失.
- 另外SCM还有容量大的特性, 一般而言, SCM的容量可以达到1TB甚至更高, 并且价格合理, 能够让大部分人负担得起.
- SCM具有快速**读取**的特性, 读取速度仅仅与DRAM相当, **写入**速度较DRAM而言则相差10到100倍以上.
- 论文中的话是

> SCM combines the economic characteristics, capacity, and non-volatility property of traditional storage media with the low latency and byte-addressability of DRAM.
>
> SCM has a potential that goes beyond replacing DRAM: it can be used as universal memory, that is,as main memory and storage at the same time.

SCM与普通磁盘有什么区别?普通数据库以页的粒度读写磁盘的方式适合操作SCM吗?

- 读写不对称, 在SCM上, 读取的速度是明显大于写入的速度.
- SCM的读写速度虽然比不上DRAM, 但是比磁盘速度快, 一般来讲, SCM的速度介于DRAM与SSD之间.
- SCM具有作为NVM设备的致命缺陷, 也就是说写入的次数有限, 写入几百万次时可能会造成永久失效的问题. 而磁盘不同, 就HDD来讲, 寿命非常长, 一般可以使用10年以上, 就SSD来讲, 一般可以写入1万次到10万次以上, 寿命比HDD短一些, 但是这里的写入是指一个颗粒的写入次数, 而不是磁盘整体的写入次数, 所以SSD的寿命还是可以接受的. 在这一点上SCM与HDD的区别较大, 与SSD有相似点.
- 低能耗, 与磁盘相比, 无论是读取还是写入, SCM的能耗更低.

结合上述特点, 特别是写入次数而言, SCM的写入次数非常**expensive**, 而且论文中也多次提到

> Chen et al. proposed to use unsorted nodes with bitmaps to decrease the number of **expensive writes** to SCM.
>
> Leaf nodes however keep keys unsorted and use a bitmap to track valid entries in order to reduce the number of **expensive SCM writes.**

所以我们秉持的宗旨之一就是尽量减少对SCM的写入次数,而且由于SCM是按字节寻址的, 如果和普通数据库读写磁盘一样以页的方式, 那么就会产生不必要的读写次数, 从而减少了SCM的寿命. 因此我们应该把计算操作与持久存储操作分离, 只存储必要的数据, 使得对SCM的写入尽量减少, 延长使用寿命.

操作SCM为什么要调用CLFLUSH等指令？

- 由于SCM的存储,访问是一个长链,需要很多的硬件,它的写操作是必须要软件协作的.CLFLUSH指令在处理器缓存层次结构（数据与指令）的所有级别中，使包含源操作数指定的线性地址的缓存线失效。失效会在整个缓存一致性域中传播。如果缓存层次结构中任何级别的缓存线与内存不一致（污损），则在使之失效之前将它写入内存。而FENCE 指令，也称内存屏障（Memory Barrier），起着约束其前后访存指令之间相对顺序的作用。

- 我们使用CLFLUSH等命令来将那些高速缓存中的写变成持久性的,并且由于他们并不删除cache line中的内容,仅是将其中内容写回,这将极大的提高cache line再次使用时的效率.论文中也提到:

  > Contrary to CLFLUSH, CLWB does not evict the cache line but simply writes it back, which can lead to significant performance gains when the cache line is re-used shortly after it was written back.

  写入后不调用，发生系统崩溃有什么后果?

  如果没有调用这些函数,系统便发生崩溃,那么将造成数据的

FPTree的指纹技术有什么重要作用？

在SCM中，对于未排序的叶子节点，采用线性扫描需要花费大量时间，使用指纹技术， 可以提前过滤掉不合适的值，从而提高FBTree的检索效率。

指纹技术在原文中是这么定义的

> To enable better performance, we propose a technique called Fingerprinting. Fingerprints are one-byte hashes of leaf keys, stored contiguously at the beginning of the leaf as illustrated in Figure 2. By scanning them first during a search, fingerprints act as a filter to avoid probing keys that have a fingerprint that does not match that of the search key.

在我的理解中就是用一个哈希表保存FBTree中的数据的个数，该表放在FBtree的首端， 在检测数据时，先在该表中进行过滤，即可提高检索效率。

为了保证指纹技术的数学证明成立，哈希函数应如何选取？

(哈希函数生成的哈希值具有什么特征，能简单对键值取模生成吗？)

> Fingerprints are one-byte hashes of leaf keys, stored contiguously at the beginning of the leaf as illustrated in Figure 2. By scanning them first during a search, fingerprints act as a filter to avoid probing keys that have a fingerprint that does not match that of the search key.
>
> We assume a hash function that generates uniformly distributed fingerprints.

也就是说哈希函数要生成一个均匀分布的fingerprints, 而简单地对键值取模生成的哈希值很明显不是均匀分布的, 所以哈希函数不能对键值取模生成.

持久化指针的作用是什么？与课上学到的什么类似？

- 持久化指针和持久化分配器配合来保证数据的一致性.由于FPTree中的叶子节点与内节点分别存储在SCM与DRAM中,所以在重启应用时的数据恢复和使用需要持久化指针的帮助.
- 持久化指针包含8byte的文件ID,以及8byte的偏移量.文件ID对应的文件是由持久化分配器分配的.而虚拟指针也通过持久化分配器来进行虚拟指针和吃计划指针的相互转换.持久化指针在程序崩溃或者其他异常的时候仍然是有效的,可以用来在重启时更新虚拟指针,使得程序的重启顺利进行.