

# Flash cache 的实现问题

# agenda

- Device mapper
- Flash cache

# Device mapper

- 块驱动的实现机制
- Bio的转发机制
- Expose的接口信息

# 块驱动的实现机制

- 注册逻辑磁盘 mapped device

drivers/md/dm.c:

```
static struct mapped_device *alloc_dev(int minor)
{
    struct mapped_device *md = kzalloc(sizeof(*md), GFP_KERNEL);
    md->queue = blk_alloc_queue(GFP_KERNEL);
    blk_queue_make_request(md->queue, dm_request);
    md->disk = alloc_disk(1);
    sprintf(md->disk->disk_name, "dm-%d", minor);
    add_disk(md->disk);
}
```

- 用户层向逻辑磁盘提交IO请求，也就是说逻辑磁盘时用户读写接口，用户不直接读取实际磁盘。

# 块驱动的实现机制

- `drivers/md/dm.c`描述了这个DM框架，也就是DM的块驱动程序。
- 通过用户层工具`dmsetup`注册块设备和驱动，块设备即[mapped device](#)

作为一个逻辑上连续的磁盘

其映射的实际磁盘由一个

表来查取（`dm_table`），这个表表示一种逻辑磁盘上面的每一个扇区

到实际磁盘上面扇区的映射关系，并且`dm_table`组织成B-tree，加快了映射的查找速度。

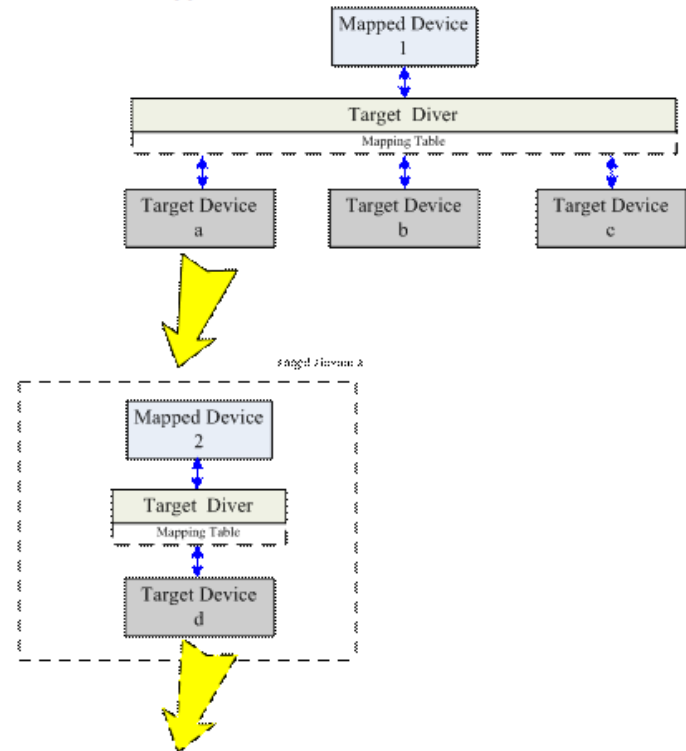
```
struct mapped_device {  
    struct request_queue *queue;  
    struct gendisk *disk;  
    struct workqueue_struct *wq;  
    struct dm_table *map;  
    struct kobject kobj;  
    . . .  
}
```

# 块驱动的实现机制

- Dm\_table描述了某一个mapped\_device的映射表
- 包括了一个dm\_target组成的数组
- 每一个数组包含了一个target
- Dm\_target描述了mapped\_device的某一个扇区映射到的设备，即可以是一个实际的磁盘设备也可以是一个mapped\_device。DM设备的映射是一种递归的实现。

```
struct dm_table {  
    struct mapped_device *md;  
    /* btree table */  
    unsigned int depth;  
    struct dm_target *targets;  
    . . .  
}
```

图2 Device mapper 内核中各对象的层次关系



# 块驱动的实现机制

- 通过dm\_table查找逻辑设备mapped\_device映射的设备dm\_target, dm\_target描述了一个设备, 这个块设备映射为mapped\_device中的某一段。Dm\_target包含了一个target\_type, 描述设备的类型, target\_type描述了如何将对mapped\_device请求的bio转换为对dm\_target (dm\_garget的void\* private指向具体的块设备) 指向的设备的请求的bio的方法。


```
struct dm_target {  
    struct target_type *type;  
    sector_t begin;  
    sector_t len;  
    /* target specific data */  
    void *private;  
    sector_t split_bio;  
    . . .  
}
```

```
struct target_type {  
    dm_ctr_fn ctr;  
    dm_map_fn map;  
    . . .  
}
```

# 块驱动的实现机制

- Private指向设备，比如线性映射设备的：

- `ioctl`:

```
static ioctl\_fn lookup\_ioctl(unsigned int cmd){  
    {DM\_DEV\_CREATE\_CMD, dev\_create},  
    {DM\_TABLE\_LOAD\_CMD, table\_load},  
    . . .  
}  
{drivers/md/dm-ioctl.c}
```
  - ```
struct linear\_c {  
    struct dm\_dev *dev;  
    sector\_t start;  
}
```
  - ```
struct dm\_dev {  
    struct block\_device *bdev;  
    fmode\_t mode;  
    char name[16];  
}
```
- 



# Bio的转发机制

```
__map_bio  
r = ti->type->map(ti, clone, &tio->info);  
if (r == DM_MAPIO_REMAPPED) {  
    generic_make_request(clone);  
}
```

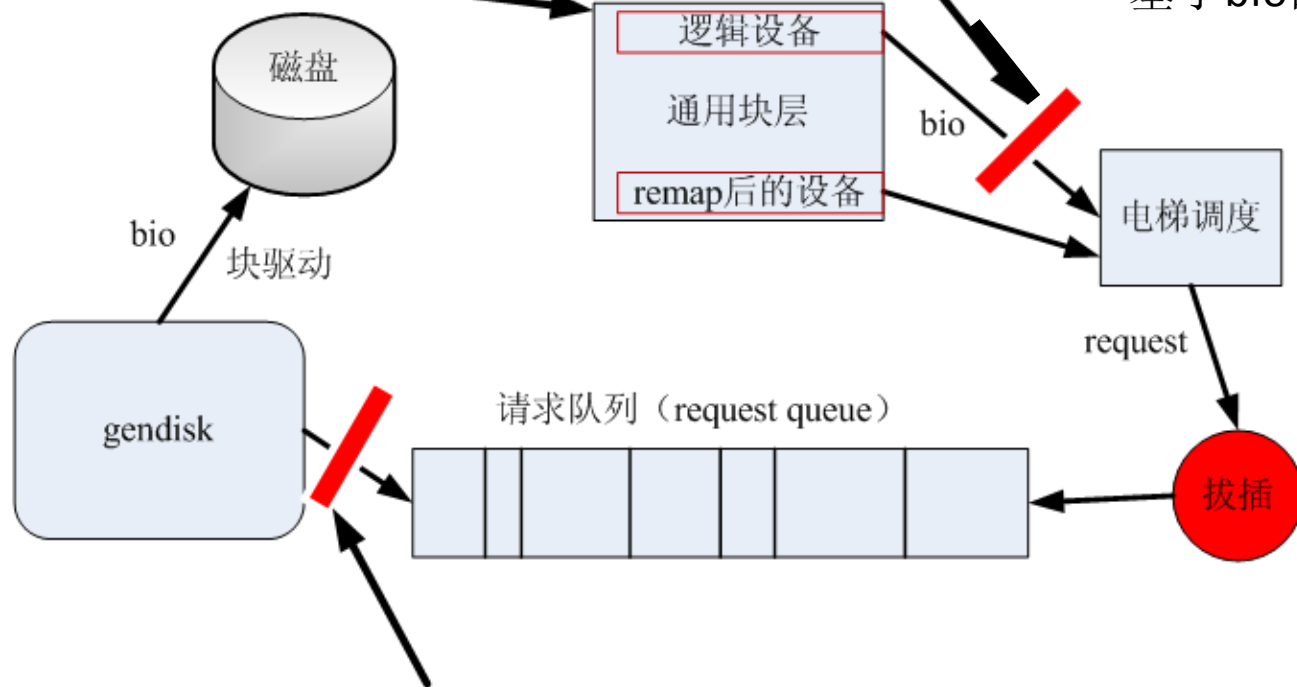
\_\_clone\_and\_map

\_\_split\_and\_process\_bio

dm\_request

Generic\_make\_request > q->make\_request\_fn

基于bio的转发



DM将每一个bio拆开，按目标设备的块大小构建新的bio，也就是说，提交到目标设备的bio最大为一个目标设备逻辑block。

Unplug & q->request\_fn[Blk\_init\_queue(dm\_request\_fn, NULL)]  
2.6.30中没有这个实现，在2.6.33中有

# 基于request的转发

- 块驱动程序中，底层服务由[blk\\_unplug\\_timeout](#)函数依靠内核负责块驱动的工作队列（kblockd）执行具体块设备的q->request\_fn来执行请求。基于request的转发30内核还没有实现。
- 如果在generic\_make\_request的时候直接处理掉bio而不是利用bio通过I/O调度器构建request，则就是基于bio的转发，如果构建了request插入到块设备的请求队列中，则会激发请求队列的request\_fn函数。这样就是基于request的转发。

```
[root@WS09091401 program]# ps -e |grep kblockd
186 ?      00:00:00 kblockd/0
187 ?      00:00:00 kblockd/1
188 ?      00:00:00 kblockd/2
189 ?      00:00:00 kblockd/3
190 ?      00:00:00 kblockd/4
191 ?      00:00:00 kblockd/5
192 ?      00:00:00 kblockd/6
193 ?      00:00:00 kblockd/7
```

# 接口

- `include/linux/device-mapper.h`
- `include/linux/dm-io.h`
- `include/linux/dm-kcopyd.h`

# device-mapper.h

- 包含了需要定义的struct [target type](#)中各种函数的原型，还包括了操作dm\_table的方法

```
dm\_ctr fn ctr;  
dm\_dtr fn dtr;  
dm\_map fn map;  
dm\_map\_request fn map\_rq;  
dm\_endio fn end\_io;  
dm\_request\_endio fn rq\_end\_io;  
dm\_flush fn flush;  
dm\_presuspend fn presuspend;  
dm\_postsuspend fn postsuspend;  
dm\_preresume fn preresume;  
dm\_resume fn resume;  
dm\_status fn status;  
dm\_message fn message;  
dm\_ioctl fn ioctl;  
dm\_merge fn merge;  
dm\_busy fn busy;
```

```
struct dm\_target {  
    struct dm\_table *table;  
    struct target\_type *type;  
  
    /* target limits */  
    sector\_t begin;  
    sector\_t len;  
  
    /* FIXME: turn this into a mask, and merge with io_restrictions */  
    /* Always a power of 2 */  
    sector\_t split\_io;  
  
    /*  
     * These are automatically filled in by  
     * dm_table_get_device.  
     */  
    struct io\_restrictions limits;  
  
    /* target specific data */  
    void *private;  
  
    /* Used to provide an error string from the ctr */  
    char *error;  
};
```

```
int dm\_table\_add\_target(struct dm\_table *t, const char *type,  
    sector\_t start, sector\_t len, char *params);
```

# dm-io.h

- 区域映射，可以针对被映射的某一个区域直接进行bio的提交

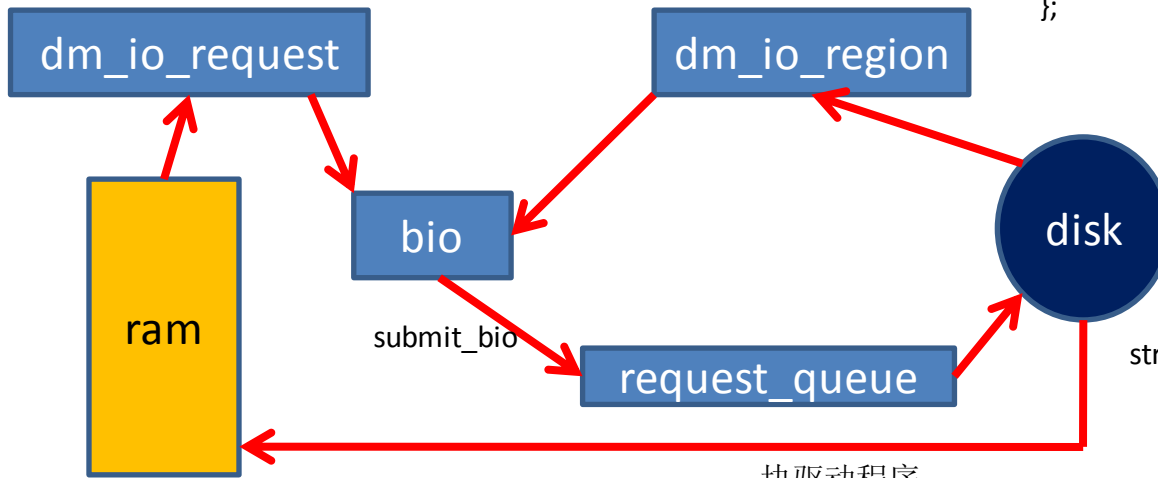
```
int dm_io(struct dm_io_request *io_req, unsigned num_regions,  
          struct dm_io_region *region, unsigned long *sync_error_bits);
```

构造bio请求需要的信息：保存磁盘数据的内存地址，异步通知函数等

dm\_io

磁盘上的某一个区域的3元组

```
struct dm_io_request {  
    int bi_rw;  
    struct dm_io_memory mem;  
    struct dm_io_notify notify;  
    struct dm_io_client *client;  
};
```



```
struct dm_io_region {  
    struct block_device *bdev;  
    sector_t sector;  
    sector_t count;  
}
```

# linux/dm-kcopyd.h

- kcopyd provides a simple interface for copying an area of one block-device to one or more other block-devices, either synchronously or with an asynchronous completion notification.
- 提供了一种底层的拷贝数据的方法。

```
int dm_kcopyd_copy(struct dm_kcopyd_client *kc, struct dm_io_region *from,  
                  unsigned num_dests, struct dm_io_region *dests,  
                  unsigned flags, dm_kcopyd_notify_fn fn, void *context);
```

# 利用DM实现自己的逻辑设备模型

- 类型定义（数据结构）：

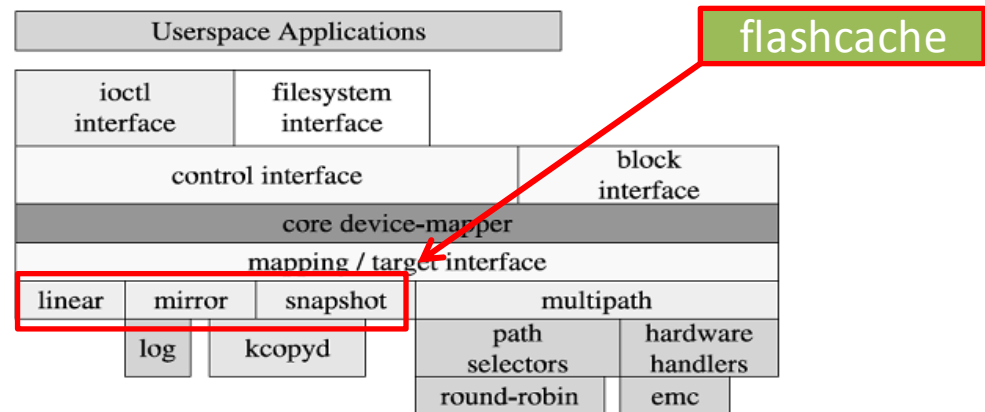
- Private设备：

```
struct cache_c {  
    struct dm_target    *tgt;  
    struct dm_dev        *disk_dev; /* Source device */  
    struct dm_dev        *cache_dev; /* Cache device */  
    ...  
}
```

- dm\_target->target\_type

```
static struct target_type flashcache_target = {  
    .name = "flashcache",  
    .version= {1, 0, 1},  
    .module = THIS_MODULE,  
    .ctr = flashcache_ctr,  
    .dtr = flashcache_dtr,  
    .map = flashcache_map,  
    .status = flashcache_status,  
    .ioctl = flashcache_ioctl,  
};
```

## Device Mapper Kernel Architecture



## 转发逻辑

- **map**函数负责将对逻辑设备提交的**bio**映射为对物理设备的**bio**

```
int  
flashcache_map(struct dm_target *ti, struct bio *bio,  
               union map_info *map_context)  
{  
    if (bio_data_dir(bio) == READ)  
        return flashcache_read(dmc, bio);  
    else  
        return flashcache_write(dmc, bio);  
}
```

- 利用DM自动转发**bio**（**map**函数修改**bio**属性），或者使用**dm\_io**函数进行转发。



# Flash cache

- 原理
- 性能

# Cache映射策略

MD: 逻辑设备

SSD: cache设备

SATA: 源设备

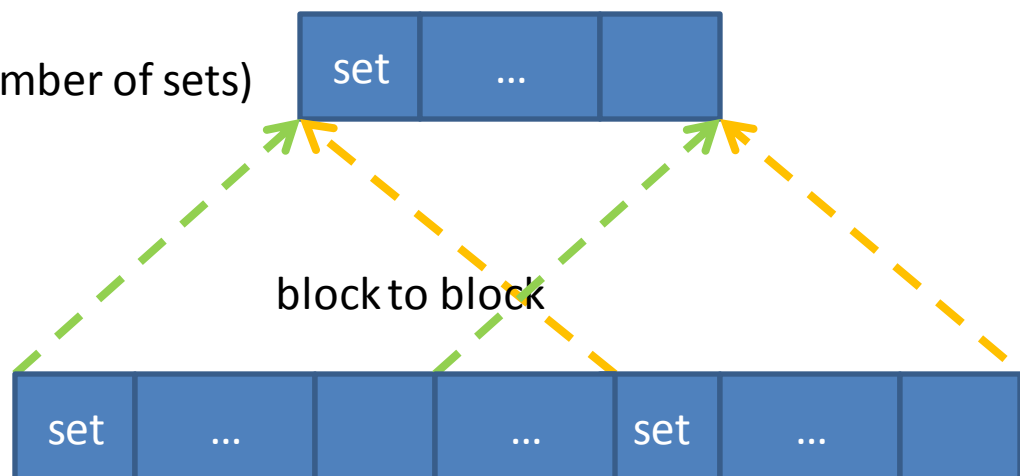
将源设备按SSD大小分为N份，在每一份内，SSD和SATA分为同样大小的M个set，每个set默认为512个block（这个block为SSD的逻辑块大小，并且用来初始化目标设备(target\_device)的split\_bio字段，由dm负责将上层的bio拆分成基于split的大小的block，然后再传给flashcache。所以会存在SATA上面的文件系统（ext4等）格式化的块大小和逻辑设备不一致的情况。

$\text{sector}(\text{logic}) = \text{sector}(\text{sata})$

$\text{dbn} = \text{sector}(\text{logic})$

$\text{target set} = (\text{dbn} / \text{block size} / \text{set size}) \% (\text{number of sets})$

回忆: bio中只涉及设备的扇区号，并不知道块的存在，块是文件系统组织数据的基本单位。



# Cache内存数据结构

- struct cache\_set和struct cacheblock 是在内存中的保存的cache信息，每一个SSD的块都对应一个cacheblock，每一个SSD中的set都对应一个cache\_set，cache\_set中还有lru链表指针等信息。Cache\_c为逻辑设备md中的private指针指向的似有设备，也就是整个cache框架。依次可以计算出比如NG的SSD需要的内存量为：  
$$N * 2^{30} / \text{blocksize} * (\text{sizeof}(\text{cacheblock}) + \text{sizeof}(\text{cache\_set}) / \text{set\_size})。$$
- $\text{Sizeof}(\text{cacheblock}) = 24$
- $\text{Sizeof}(\text{cache\_set}) = 20$
- $\text{Set\_size} = 512$
- $\text{Blocksize} = 4096$  30G SSD大约需要180MRAM

# Cache读策略

- 对于一个读bio处理方法为：
  - 得到bio中的对应SATA的扇区号，找到SSD上面的set，在set内部查找这个块是否被缓存过
  - 如果缓存过，则将bio中的扇区号转换为SSD上的扇区号，提交bio到SSD
  - 如果没有缓存过，则将bio提交到SATA，并且克隆一个bio，将该bio的操作改为写，在提交给SATA的bio完成之后，将克隆的bio提交给SSD，从而将内存中的刚刚读取的数据复制到SSD上面，完成缓冲的功能。
  - 文档中说会出现跨block的bio，但是我觉得如果block号对齐的话应该不会出现这种bio，因为dm已经将bio按照split大小分开了。

# Cache写策略

- 如同页缓冲，内核并没有真正的将数据写入磁盘，而是写入SSD。
- 内核写策略：当脏页超过一个门限（默认40%），则启动pdflush将脏页写回磁盘，写回操作可以在多种情况下被触发。
- Flashcache写回操作同样设置了一个门限（默认20%）
- 避免torn pageproblem:
  - Sector大小=block大小（这个可以理解）
  - Shadow page（不懂是什么意思）

# Cache替换策略

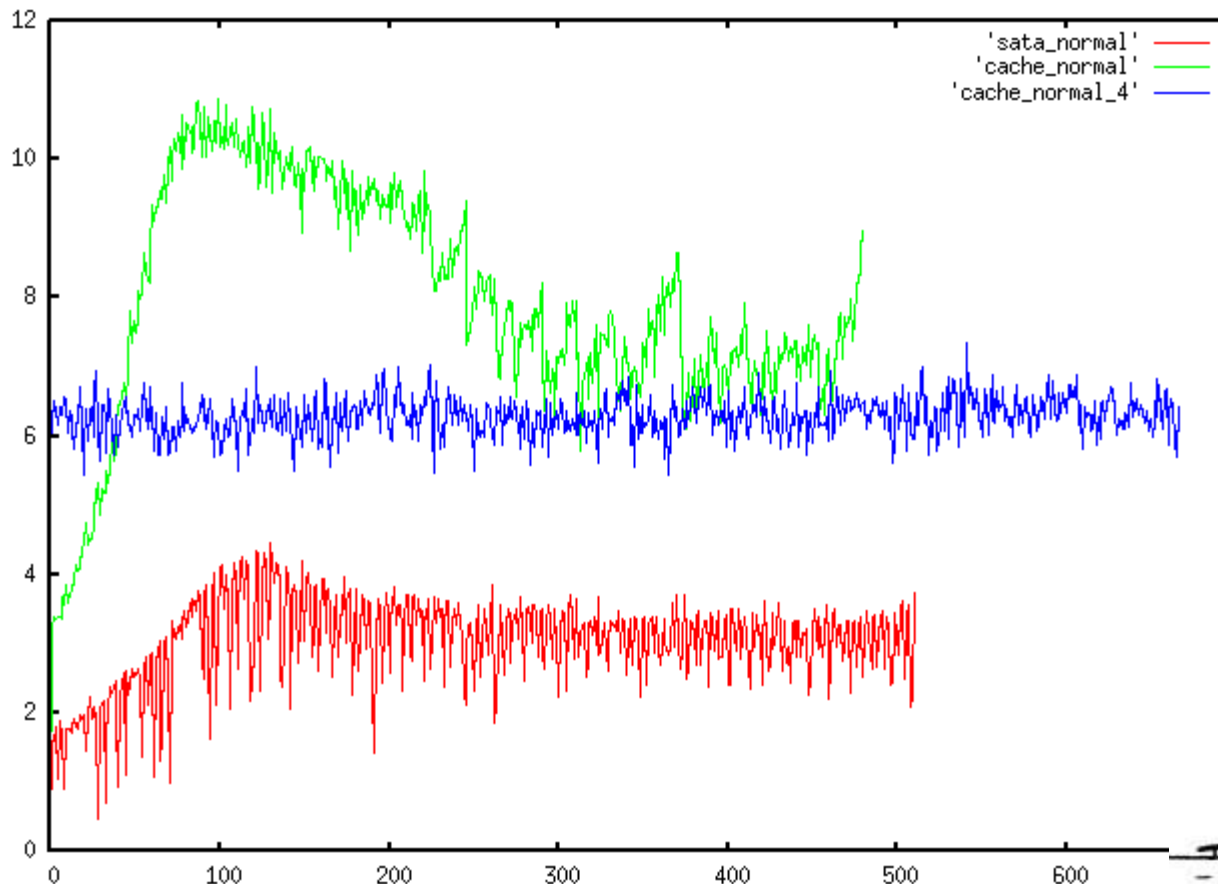
- 当某一个bio请求映射到的set内所有block都被占用（state = vaild）的时候，启动替换策略（FIFO或者LRU）。
- 选择被替换的页
- 如果是脏页，则等待脏页操作完成将其刷回SATA
- 默认FIFO，可以通过  
/proc/sys/dev/flashcache/reclaim\_policy设置（0默认FIFO）

# 性能测试

- 读数据方案：
  - 5K-20K文件
  - (0-1) 正态分布读取和随机读取
  - 300G SATA分区，240G数据
  - 30G SSD 和60G SSD 分区，测试不同CACHE大小的影响
  - 文件系统：ext4
  - 读取速度：极限测试（200线程）
  - 测试时间连续读取1.5小时以上

# sata Vs. cache

- 在正态分布读情况下，FIFO策略的60G cache并不比30G cache好，观察命中率变化（/proc/flashcache\_stat）发现在正态分布读在30G和60G的情况下基本相同。

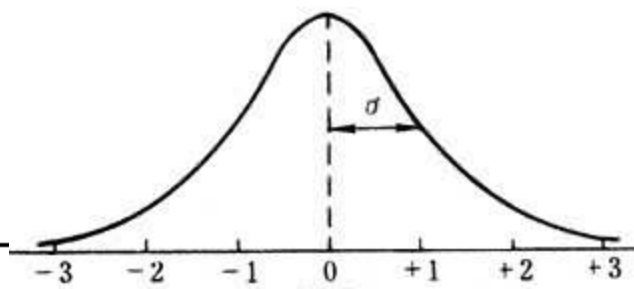


3.4% -> 7.8%

20% -> 50%

43% -> 80%

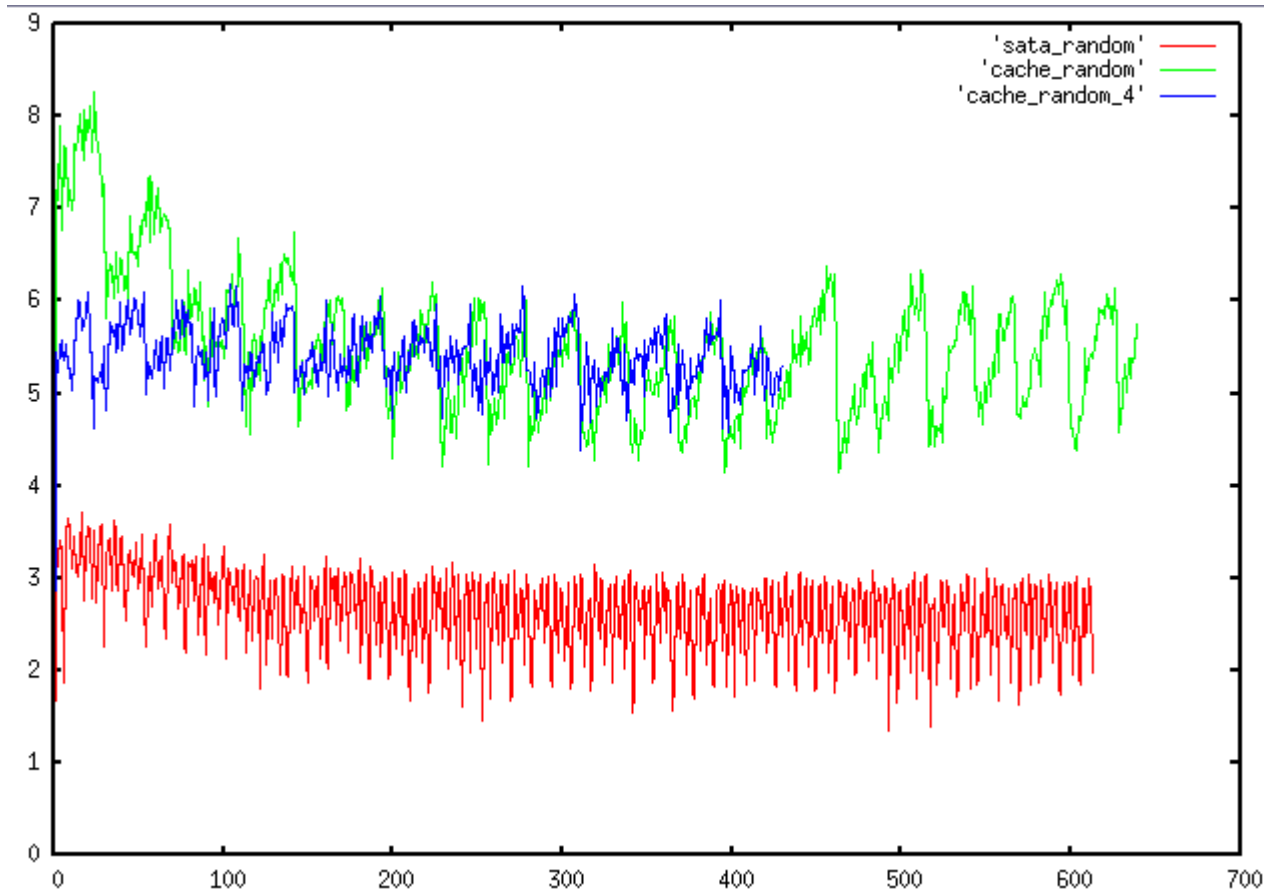
80%读取集中在43%文件



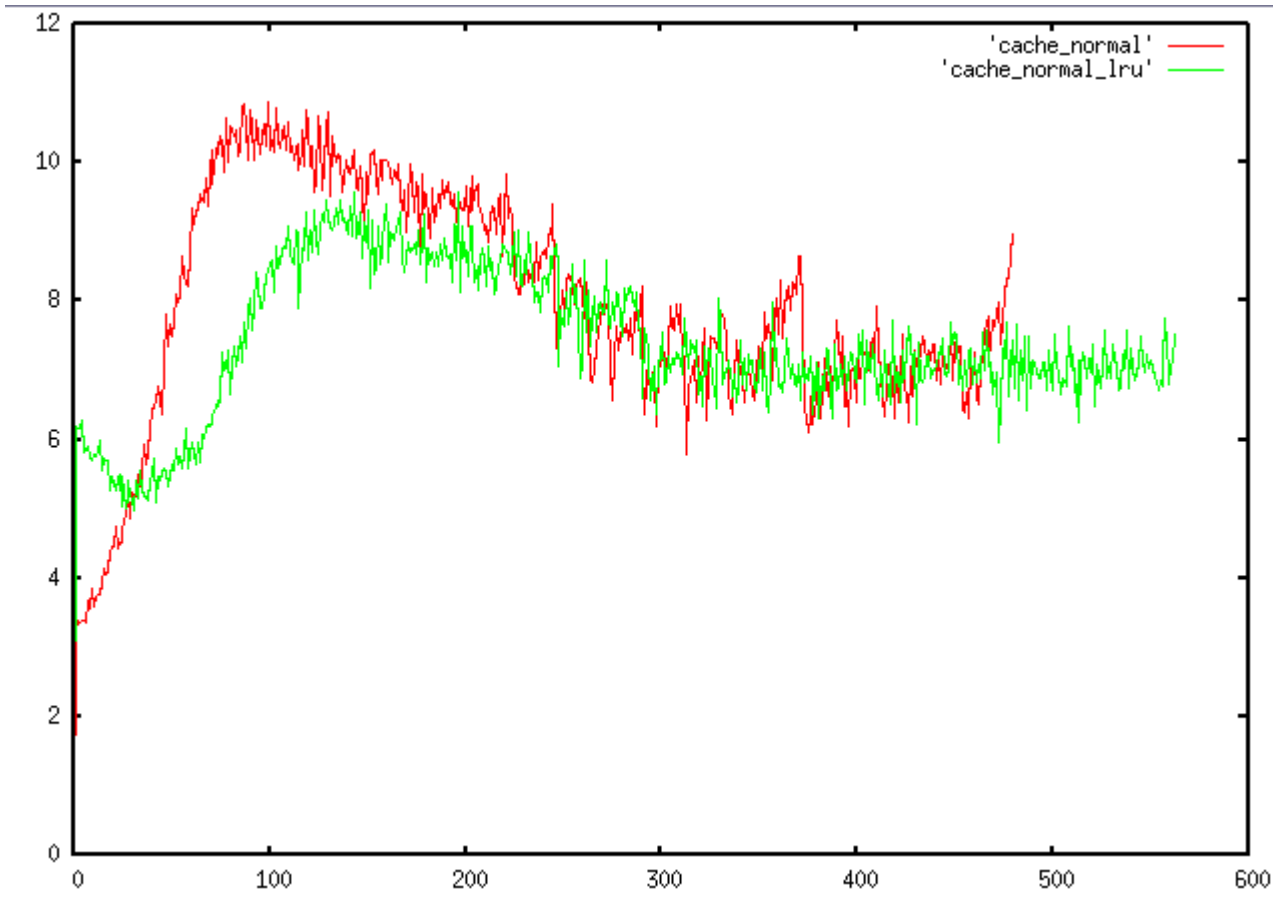


# sata Vs. cache

在随机读取的情况下，cache还是有明显的性能提升，但是30G和60G的性能还是没有什么差距，60G表现为抖动比较小。

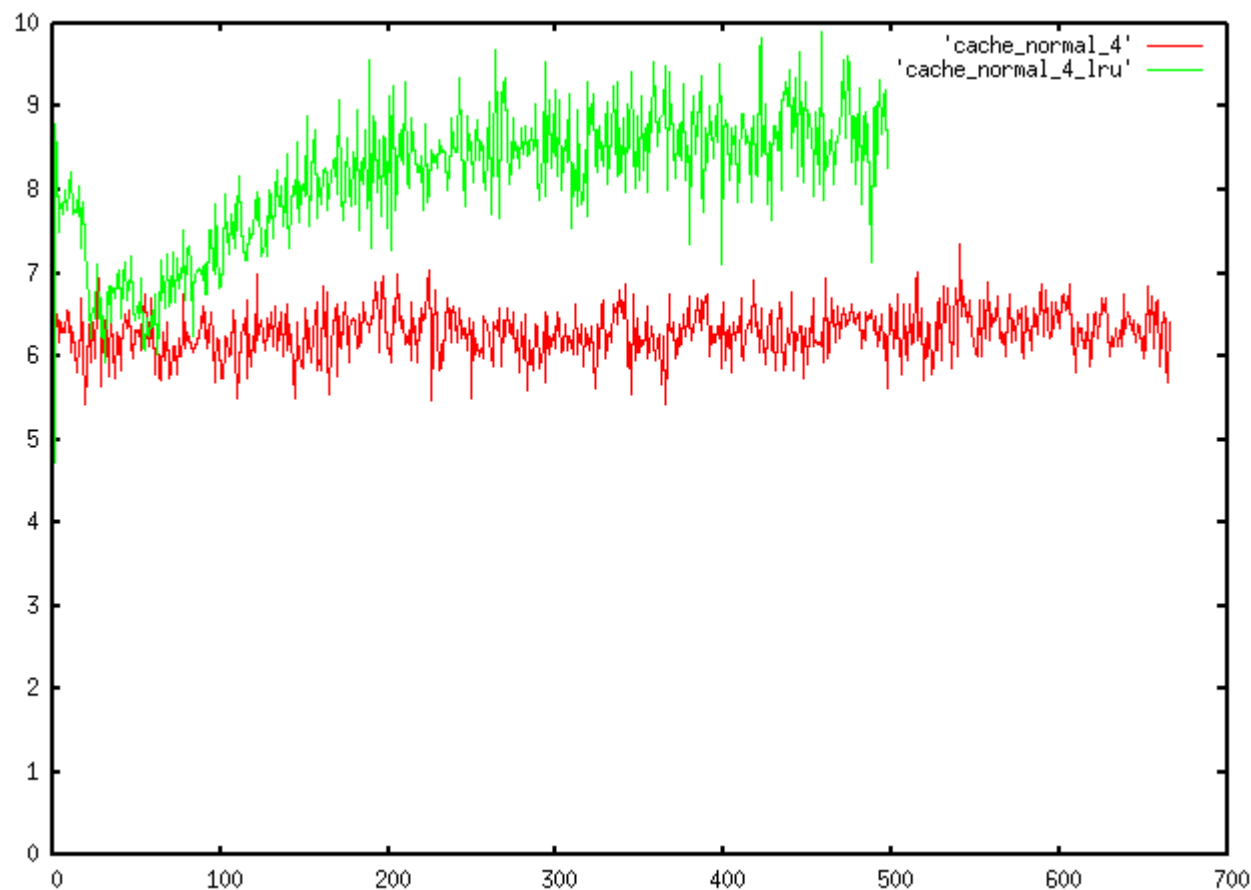


# FIFO Vs. LRU



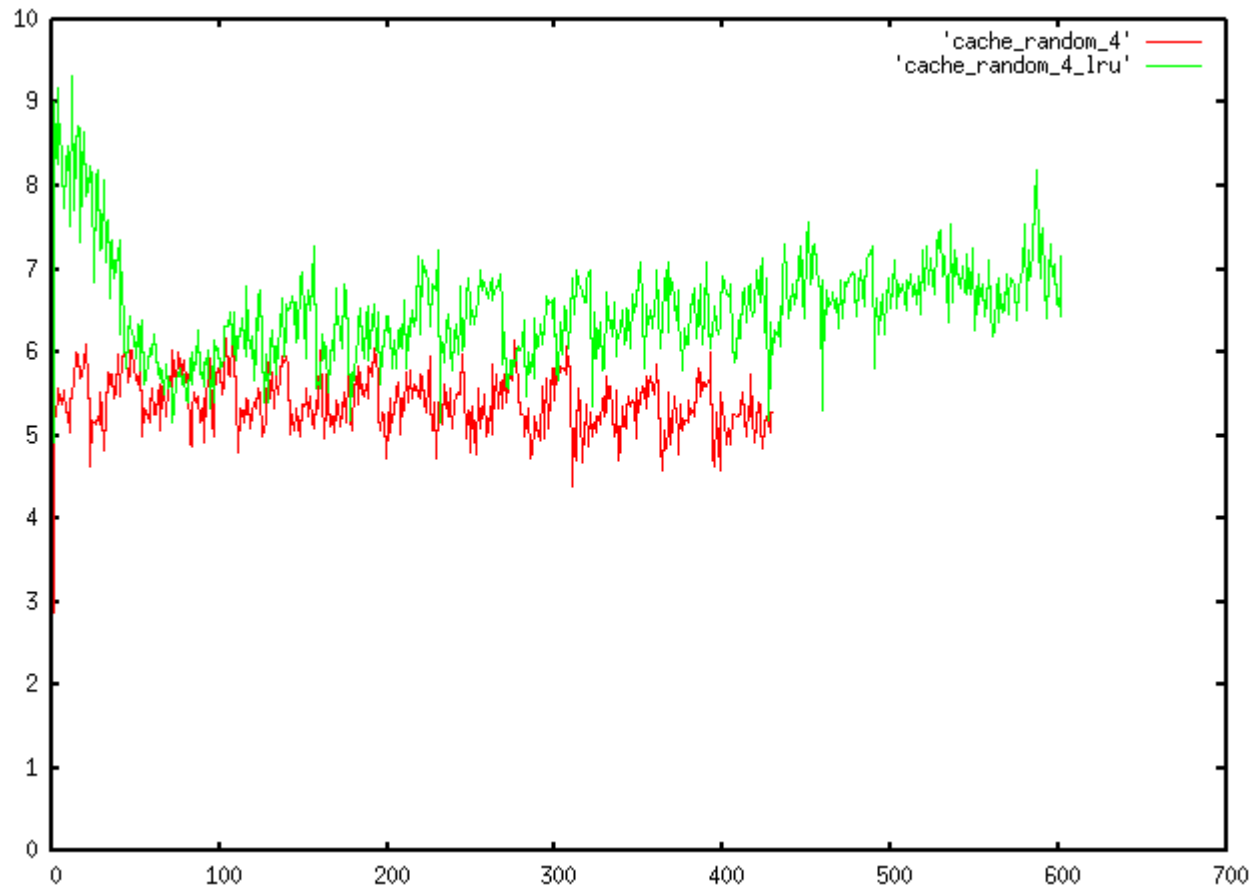
30G情况下lru  
相对于Fifo没  
有明显提升，  
但是抖动明显  
减小。

# FIFO Vs. LRU



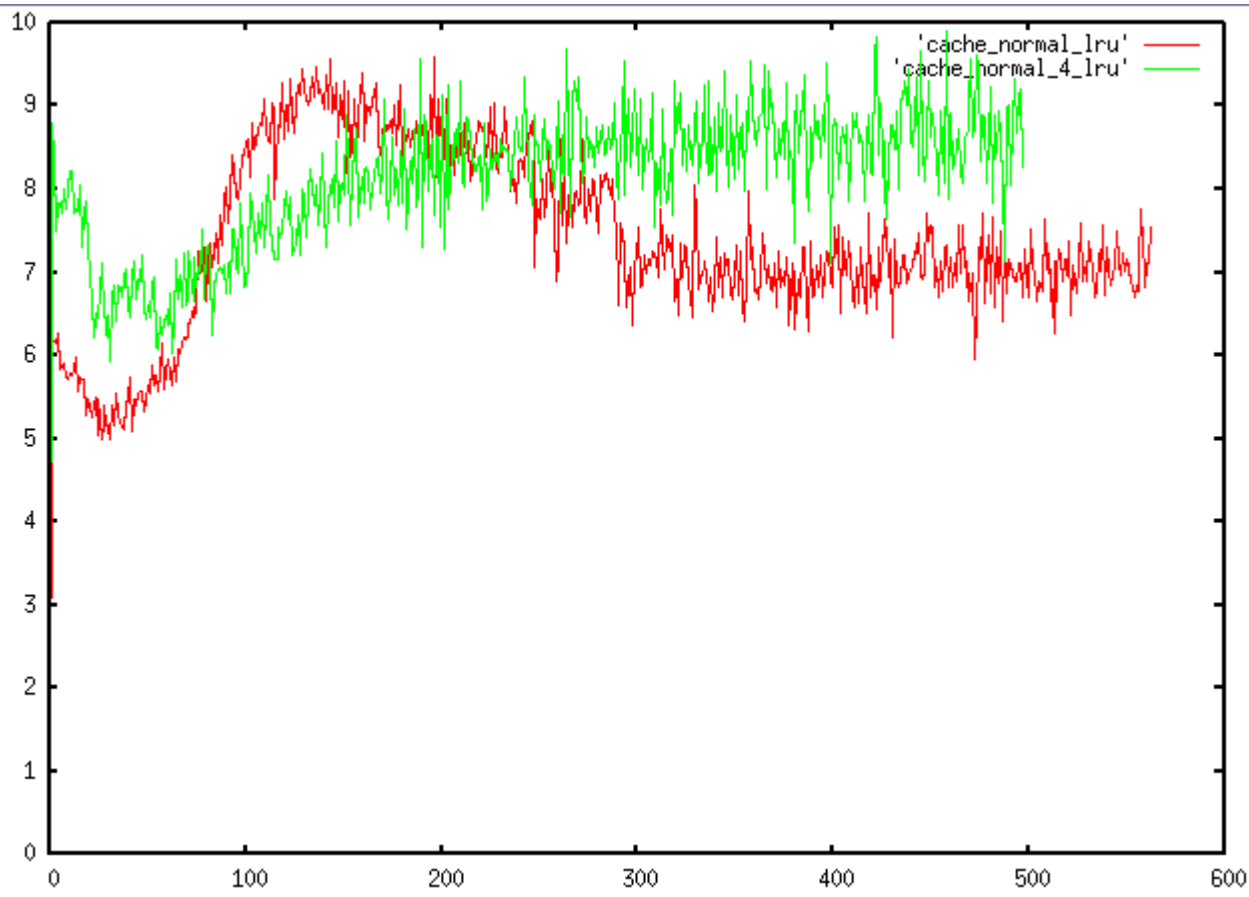
正态分布读，60G情况下，LRU较FIFO有明显提升，平均速率可以达到8Mbyte/s以上

# FIFO Vs. LRU



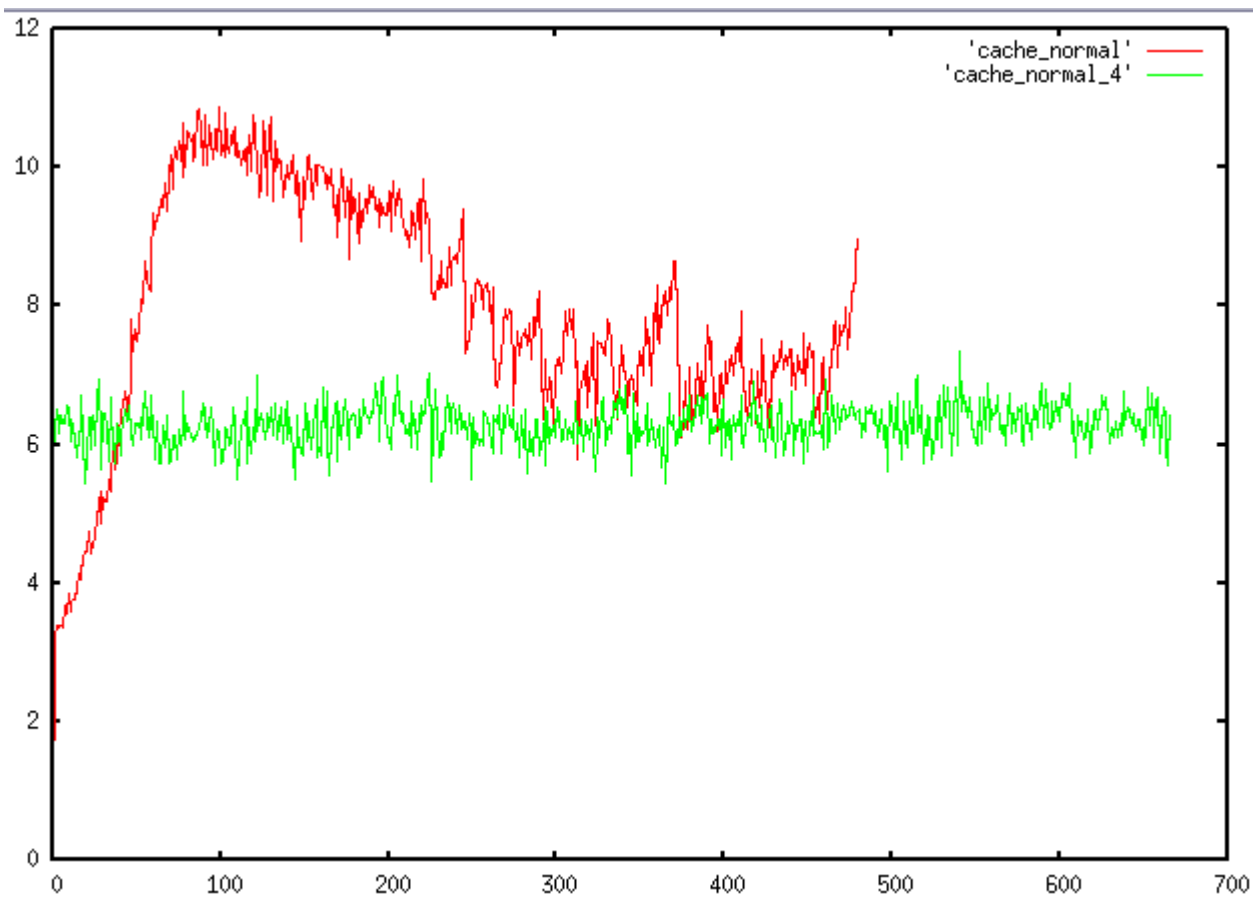
随机读取情况下，lru  
较fifo提升不大

# 30G Vs. 60G



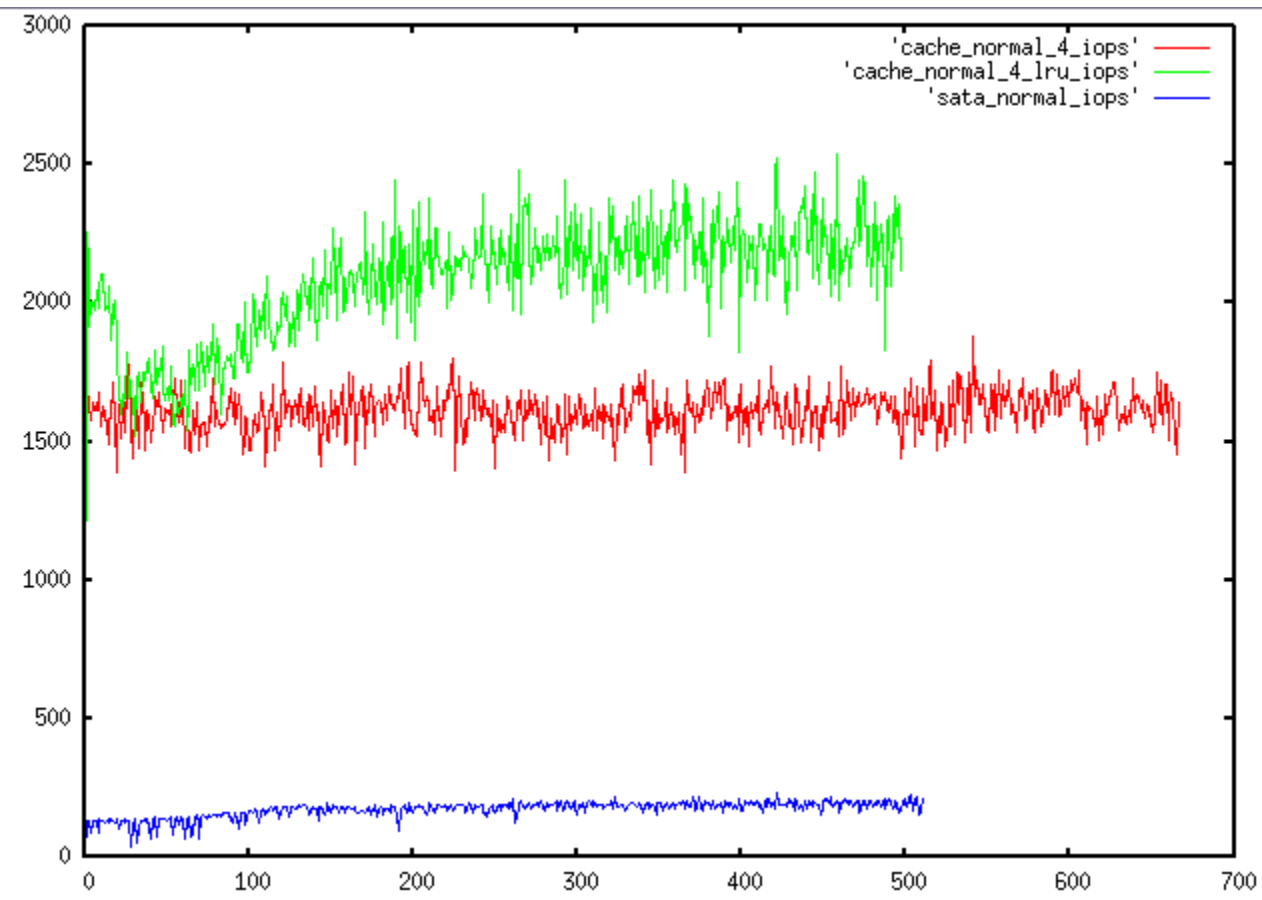
正态分布读，  
60G，LRU情况下和  
30G对比还是有明显  
提升，但是达到稳定  
状态的过程不同，30G  
达到最高点后有明显  
下降趋势，不知道是  
什么原因

# 30G Vs. 60G



正态分布读，FIFO策略  
情况下，60G比较平稳，  
但是没有30G读取速率高

# iops



同样60GLRU策略获得了较好的效果。相对于SATA有很大提高。

# conclusion

- 替换策略LRU，60G情况下性能最优



# Flashcache操作注意事项

- 如果想要destory某一个设备，不要先运行flashcache\_destory，先运行dmsetup--remove []。
- 设备文件在/dev/mapper/\*
- 设备在/sys/block/dm-\*下
- 用户接口/proc/sys/dev/flashcache
- Cache状态/proc/flashcache\_stat,/proc/flashcache\_error
- 格式化设备可以通过逻辑设备/dev/mapper/\*，也可以直接格式化设备文件。比如：
  - mkfs.ext4 /dev/mapper/cache\_sdb1\_sdd3
- 针对某一个具体磁盘的多个逻辑cache设备可以同时存在，但不要同时访问。