## Flash FS

F2FS维护六个主要日志区域来进行冷热数据的分离，冷热数据有三个温度(hot/warm/cold)，目录项的node和data都是hot，文件直接节点和用户数据是warm，间接节点和cleanning清掉的数据/用户标记的冷数据/多媒体文件都是cold。通过预期的更新频率进行冷热数据的分离。

就是N+n是做了cp后这个数据更新的点，因为有特殊的flag，所以我们可以找到它，记为N+n。然后recovery的时候，使用cp，cp有个稳定的记录，能得到这个数据在cp之前的稳定的状态，记为N-n，进行对比，不同就使用最新的那个。

Metadata

direct pointers

or inline data

Inline xattrs

Single-indirect

Double-indirect

Triple-indirect

Wandering Tree 是一个在以 LFS 为代表的使用 Copy on Write 技术的文件系统上都存在的 一个问题，当文件的数据块被更新的时候是写到 log 的末尾，该数据块的直接指针也因为数据位置的改 变而更改，然后间接指针块也因为直接指针块的更新而更新。

F2FS 引入了一个新的表，Node Address Table(NAT)。首先需要了解 F2FS 的文件结构，在传统的 文件系统中，indirect node block 的表项是指向 direct node block 的，但是在 F2FS 并不 是这样的，indirect node block 是指向 NAT 的，这样每当一个 data block 被污染的时候，会 更新 direct block 和 NAT 中的表项，这样就防止了在 indirect node block 中的传播。

## NVM

PMFS 引入了一个硬件原语：pm\_wbarrier，用所谓细粒度的 log 来实现 consistency。在可以使用原子写的地方，PMFS 会率先使用原子写。对于 meta data 而言 PMFS 使用了 undo journal 的方式，因为 meta data 的 size 比较小。对于真正的 data 会使用 copy on write 的方式来保证一致性。在 PMFS 没有被 umount 的时候断电， PMFS 会在下一次 mount 的时候进行 recovery。对于 PMFS 的 allocator 的 consistency，会遍历文件系统的 B 树来进行 recovery。这几方面综合就是 PMFS 为了解决 consistency 问题做的事情。

Copy on write 类似于 F2FS 中的对 NAND Flash 的做法，在写的时候会复制一个数据副本，然后在副本上进行修改，最后再将指针指向副本所在的地址。Journaling 的典型实现是 EXT 4，在存储数据之前，会先将数据元数据等存放在 log 中，在事务提交后才将数据真正写入。在 recovery 的时候会把 log 里的所有事务重放一遍。Log-structured updates 是 F2FS 基于的 recovery 方式，相比于 journaling 的多次写，LFS 只需要一次追加写。

clflush: 将cache中修改后的数据写进memory system中(但不一定能写进nvm）flush the cacheline

sfence: 保证存储操作的完成 ensure the completion of store

pm\_wbarrier: 保证数据一定能写进nvm中。 ensure the durability of every store to PM

引入pm\_wbarrier是为了解决clflush不能保证一定将数据写入nvm的问题，三条指令共同保证durability

redo: 新数据被写入日志中并持久化，直到事务成功提交才写入文件系统中。

undo: 旧数据被写入日志中并持久化，新数据直接写入文件系统中，若事务失败则通过重写日志中的旧数据来回滚。

比较：undo实现简单,对事务中的每个log entry都需要一次pm\_wbarrier;redo对一次事务操作只需要两次pm\_wbarrier,但对事务中的所有读操作都带来额外的overhead。PMFS采用undo。

## Recovery

RAMCloud管理replica是通过，在中DRAM只保存每个object的一个备份，而大量的冗余备份是位于二级存储中的（如disk或flash）。 通过这种方式备份会带来两个问题

将数据放入disk，必须要同步地对disk写入-->非常慢！

在宕机恢复的过程中，由于需要从二级存储读取数据来重构-->还是慢！

既然把数据备份在二级存储中慢，那备份到DRAM中呢？这就需要DRAM不能断电，而且会产生高昂的费用！ 因此，RAMCloud还是选择把数据备份到二级存储中。

如上面图片所示，RAMCloud的存储的过程是当一个master接收到写请求，它将要写入的object append到内存里的log中，然后再将log entry通过网络分发到各个backup去。backup将接收到的信息buffer到内存中，然后立即向master返回一个RPC请求，而不需要立即写入到disk。master接收到backup的请求之后，向client返回。对于backup，在buffer满了之后才将buffer的数据写入disk。

可是考虑到一点，log是被buffer到backup的DRAM中的，那么一旦断电，就会导致数据的丢失---->为了解决这个问题，RAMCloud采用DIMM的方式，加入电池来保证buffer可以flush到disk上去。

注意到Hash table，hash table在RAMCloud是以<tableid, objectid>做key来定位内存中log的位置，在每次appen log的时候，都会在hash table建立索引--->hash table支持了对数据的随机访问

那么log-structured logging有如下特点：

由于backup是采用buffering的方式存储log（这是异步的、批量化的），这样避免了对disk的同步地写

只有在backup的buffer满了之后，才将数据flush到disk上，这样做就不会浪费大量的时间在等待磁盘写入

log的结构是一致的，这句话的意思就是disk和DRAM的log结构一致，因此不需要额外的管理开销

使用了hash table实现了数据的随机访问

RAMCloud采用了Partitioned Recovery策略，来进行错误恢复。思路也十分简单，RAMCloud将crashed master的object分割到若干recovery master上，让它们分开处理。并且，这些recovery master把这些数据整合到自己的log和hash table中。这样就解决了CPU/disk/network对性能带来的瓶颈，它的核心方法就是利用了系统的scaling。而这种方法的缺点就是需要与所有的host进行通信。

## Multiprocessor, Concurrency & Locking

原因是non-scalable lock哪怕在N个核的时候表现不错，也很有可能在N+1或者N+2个核的时候突然collapse。

scalable lock解决的就是这个问题，优点就是不会产生这种突然的大幅度的collapse，锁的性能不会随着核的数量增加而下降太大。缺点就是要对现有的源代码做改动，但是改动并不复杂。

这是因为在ticket lock中，会需要记录两个变量，一个是now\_serving表示正在使用lock的ticket（一个整数），另一个是next\_ticket记录着当前最后一张ticket（就是拿票等待的核的号码）。当任何一个核去拿锁的时候，都会需要读取now\_serving这个变量判断跟自己的ticket是否相等。这样一来，每个核都会对now\_serving做cache，一旦这个锁被释放，ticket lock中的now\_serving就会增加1，这个操作会invalidate所有核的cache里的now\_serving，这会触发所有的核来重新读取now\_serving这个值所在的cacheline，论文说明了在现有的架构中，这个read会被串行化处理，一个一个来，这就导致消耗的时间与等待锁的核的数量呈线性增长关系。

因此随着 core 的增加，在 serial section 很小的时候，Sk = 1/(s + ck/2) 中 k 对 Sk 的贡献越大，因此越容易受到 k 的影响。这也就是为什么，在 serial section 很小的时候，只是多了很少一些在等待锁的 cores，就出现了性能的雪崩。

在 ticket lock 中，所有的 core 都去依靠同一个变量来获得锁，而在 MCS lock 中，前一个 core 在 release 的时候会修改下一个 core 的 locked 变量，通知下一个 core 来获得锁。这样每个锁都只依靠属于自己的一个变量，这样的实现是缓存一致性无关的，有着更好的 scalability。

MCS Spinlock 申请操作描述如下：

申请者 B 使用原子交换操作将自旋锁 mcs\_lock 指向自己的mcs\_lock\_node 结构以确定在链表中的位置，并返回 mcs\_lock原来的值 pre\_mcs\_lock。即使多个执行线程同时申请锁，由于交换操作的原子性，每个执行线程的申请顺序将会被唯一确定，不会出现不一致的现象。

如果 pre\_mcs\_lock 为 NULL，表明锁无人使用，B 立即成为锁的拥有者，申请过程结束。

如果 pre\_mcs\_lock 不为 NULL，则表明 pre\_mcs\_lock 指向申请者 B 的直接前驱 A 的 mcs\_lock\_node 结构，因此必须通过pre\_mcs\_lock 来修改 A 的 next 域指向 B 自己，从而将链表构建完整。

然后 B 一直在自己的mcs\_lock\_node 结构的 waiting 域上自旋。当 B 的直接前驱 A 释放自旋锁时，A 只须通过 next 域将 B 的 waiting 域修改为 0 即可。

MCS Spinlock 释放操作描述如下：

释放自旋锁时，锁的拥有者 A 必须十分小心。如果有直接后继 B，即 A 的 mcs\_lock\_node 结构的 next 域不为 NULL，那么只须将 B 的 waiting 域置为 0 即可。

如果 A 此时没有直接后继，那么说明 A “可能”是最后一个申请者（因为判断是否有直接后继和是否是最后一个申请者的这两个子操作无法原子完成，因此有可能在操作中间来了新的申请者），这可以通过使用原子比较-交换操作来完成，该操作原子地判断 mcs\_lock 是否指向 A 的 mcs\_lock\_node 结构，如果指向的话表明 A 是最后一个申请者，则将mcs\_lock 置为 NULL；否则不改变 mcs\_lock 的值。无论哪种情况，原子比较-交换操作都返回 mcs\_lock 的原值。

如果A 不是最后一个申请者，说明中途来了新的申请者 B，那么 A必须一直等待 B 将链表构建完整，即 A 的 mcs\_lock\_node 结构的 next 域不再为 NULL。最后 A 通过 next 域将 B 的 waiting 域置为 0。

## Transactional Memory/DB

优先级反转 一个高优先级任务间接被一个低优先级任务所抢先(preemtped)，使得两个任务的相对优先级被倒置

护航 多线程程序中相同优先级任务反复竞争同一个锁。当一个持有锁的进程被解调度了，可能是因为调度时间到了，发生了一个页错误，或者是其他一些终端。当有类似终端发生时，其他本该可以运行的进程却无法运行。

死锁 两个或以上任务在执行过程中，由于竞争资源造成阻塞

本文是基于嗅探的缓存一致性协议修改来的，文章中在硬件上实现这一点的时候加入了一个新的 L1 Cache，叫做 Tx Cache。因此非事务修改走 regular cache，事务走 Tx Cache。

事务缓存有四个标签，Normal, XCommit, XAbort, Empty。事务提交的时候 XCommit -> Empty, XAbort -> Normal。事务 Abort 的时候，XAbort -> Empty, XCommit -> Normal。

它其实是利用了现存的缓存一致性协议，对于冲突的判断，都是用它来完成的。要理解这篇论文就必须对缓存一致性有比较清楚的认识。而因为实现利用了 Cache，所以在发生 Context Switch 等需要 Flush Cache 的情况下时，事务是一定会被 Abort 的，所以这篇文章的实现主要面向的场景是 short critical section，同时没有 IO 读写。

不过，实现因为是完全基于现有的实现（缓存一致性），因此可以跟非事务性内存的同步操作一起进行，因为其他不在事务中的操作是一样会触发缓存操作的。这算是一个这样实现带来的好处，很关键。

孤儿事务 是指在被废弃（aborted）之后继续执行的事务（例如，在另一个已经提交过的事务更新了自己的read set之后）update: 孤儿事务是指本身事务在执行时产生了错误（而非读写数据被其他事务提前修改）导致的abort，以至于这个事务会一个retry占据资源无法commit/rollback.

事务在执行过程中利用VALIDATE指令来确保自己所读取的值是正确的，防止读取过期的数据，即保证了数据的一致性。

## NoSQL

Table 就是用户可见的表，它有 Row，Column，和不同 Version 的 Value。Table 会从逻辑上划分为多个 Tablet 以方便存储。

Tablet 是一个存储单元。在 BigTable 的架构中，是由一个 Master Server 和多个 Tablet Server 组成的。Tablet 的读写是由 Tablet Server 来处理的。

SSTable 是一种文件的格式，全称是 "Sorted Strings Table"，是指按照 Key 的排序在文件中存储 <Key, Value> 对。在逻辑上 SSTable 会包含多个 Block，每个 Block 为了方便寻址会有一个 index，所有的 Block index 会写在 SSTable 文件的最后，每当 SSTable 文件被 open 的时候，会将索引加载到内存里，这样每次 Lookup 的时候只有一次硬盘读取。SSTable 也支持全部读取到内存里，这样在 Lookup 的时候没有任何硬盘的读取。

在进行写操作的时候，Tablet Server 会先做一些检查，保证请求的合法以及权限问题。权限的检查是通过检查一个在 Chubby 中的列表进行的，这个列表会被 Client Library 缓存住。一次被允许的写操作会先进入 Commit Log，在处理 Log 的时候采取了批处理来提高吞吐。在操作被 Commit 后，它的内容会被插入 MemTable 里，当 MemTable 的 size 超过一个阈值的时候，会让当前的 MemTable 进入一个 frozen 的状态，随后创建一个新的 MemTable，Frozen 的 MemTable 就可以以 SSTable 的形式写入 GFS。

在进行读操作的时候，Tablet Server 会在做了一些检查保证合法后，在 MemTable 和 SSTable 的一个 merge 后的 view 中来进行读操作，这样可以保证可以读到最新的值。

对于那些需要强一致性，需要同步更改多行数据的应用来说，BigTable是不合适的。

## Distributed Transaction

（spanserver执行部分）对要读的数据，向对应的group leader拿读锁（如果拿不到就放弃，重新拿，即wound-wait）；执行本地读写（外部不可见）；修改完成，开始2PC。选择coordinator group，将修改发送给coordinator leader和non-coordinator-participant leader；（每个non-coordinator-participant leader执行部分）收到txn修改内容后，选择本地最新成功的txn commit timestamp作为"prepare timestamp"返回给coordinator leader；（coordinator leader执行部分）获得每个leader相应的写锁；等待所有participant leader的"prepare timestamps"，选择最大的"s"。再将s与TT.now().latest和本地最新成果的txn commit timestamp比较，取最大的作为commit timestamp，赋值给"s"；

持续调用TrueTime获取interval，等待s < TT.now().earliest，即TT.after(s)，确保所有在s之前的txn都全局生效；以s为commit timestamp提交当前txn，并反馈client；

释放锁。

调用TrueTime API，将右区间作为只读事务的版本号,记作Sread, 如果读事务涉及到的副本不够新，那么读会阻塞。每个副本会维护一个时间戳Tsafe, 如果Sread <= Tsafe ，则这个副本够新。Tsafe取两个时间戳的更小值，第一个是副本所在的paxos group中已经apply到状态机的时间戳，第二个是paxos group中leader目前正在参与的还没有commit的分布式事务在该paxos group的最小的prepared timestamp - 1。

还有一种方法，可以不用客户端等，方法是Sread通过客户端和所有涉及到的paxos group的leader进行协商，每个leader返回他们的最后一次事务的commit时间戳给客户端，然后客户端取最小的版本号作为Sread即可。

## RDMA-enable Transaction Processing

事务需要事先知道write-set与read-set

无序存储是HTM/RDMA友好的, 而有序存储则没有对RDMA做过优化

当有机器失效时, 考虑Durability而不是可用性

读写事务提交后, 需要写回, 并释放写锁, 如果此时, 此机器挂了, 其锁永远无法被释放

读写事务冲突检测:

先本地/再本地: 由HTM保证原子性

先本地/再远程: 远程的RDMA请求, 会打断当前机器的HTM. 当前事务, 只能等待远程RDMA加的写锁释放后再继续进行. 如果远程RDMA加的是读锁, 则本地HTM可以同时执行

先远程/再本地: 远程的RDMA会先加锁数据, 本地HTM要去访问时, 如果发现数据上有写锁, 则abort, 如果有读锁, 则可并行执行

先远程/再远程: 远程的RDMA会先加锁数据, 另一个远程RDMA来访问时, 如果发现已经有写锁, 则中止, 如果发现是读锁, 则取其中止时间为自己锁的中止时间, 然后正常执行

## RDMA-friendly Key/value Store

## Low Latency

为了防止两个 server 都执行，client 可以引入一个小延迟，而不是同步发两个请求

Micro-partitions, Selective Replication, Latency-induced Probation, Large Information Retrieval

Systems(Canary)

## NFV

普通的路由器的实现只需要通过push操作就可以实现了，但是只实现push操作对于click来说，会带来包调度方面的巨大的复杂性，而通过实现pull操作，就可以通过一个简单的pull元素或者pull元素与队列的组合实现复杂的调度策略，pull操作对于click来说是必须的。

## Graph Query

Full-history 就是说所有的历史记录都会被记录下来。之前是只记录一次的。之所以可以这样做是因为一方面 RDF 的查询都不会有太多步，而且 RDMA 在低于 2K bytes 的时候性能都是差不多的，所以 Wukong 可以这样做。

## Bugs

如果执行第二步时得不到准确的结果，那么会漏报一些unstable code；

如果执行第一步时得不到准确的结果，就会产生误报(false warning / false positive)。

目前 STACK 给出的 undefined behavior pattern 可能不齐全。

STACK 为了更好的 scalability，它会用 LLVM 来内联函数，并且在内联后的 individual 函数上进行检查。

在设计上，STACK为了使可扩展性更高，在计算 Δ = ∀e:Reach(e) → ¬Undef(e) 的时候做了一些近似运算，使最后得到的结果可能会漏掉一些unstable code。