# Report for ScaleFS

## Summary of major innovations

ScaleFS想要打造一个具有多喝可扩展性的一个文件系统，他结合以往的内存文件系统的高可扩展性和磁盘文件系统的持久性等特点，创新性的提出了将这两个文件系统分割开来，并且利用类似于日志记录关键操作的方式来完成两个文件系统的一致性，从而达到了多核性能和正确持久化的一个绝佳平衡。

1. 将两个文件系统解耦合，分别完成各自的角色
2. 使用时间戳线性化来保证容灾安全和高吞吐量
3. 对Cross-directory rename 操作进行了详细分析

## Problems the paper mentioned

主要就是做一个具有多核可扩展性的一个文件系统。传统的Linux File System其实没有很好的在文件系统设计的时候考虑多核扩展性问题，所以虽然很多情况下理论上File System是可以做到Scalability，但是由于Cache-line contention或者Global Lock Require等问题在实际系统中并没有做到比较好的可扩展性。

当然在保证可扩展性的同时，作为一个最终需要持久化的文件系统，在这之间我们还需要满足Crash Safety（容灾安全）以及较好的磁盘性能。这里的容灾安全其实是指保持正确的语义，就是当我调用fsync最终能够将内容刷到磁盘，并且不会给用的人员造成困扰。

## Related works/papers

这里其实我们需要简单的对文件系统有一些了解，实际上这里文中也提到了很容易被混淆的概念，一个是in-memory file system（内存文件系统），on-disk file system（磁盘文件系统），in-memory data structure, on-disk data structure以及in-memory copy of on-disk data structure。这些概念需要弄清楚，首先on-disk data structure比较好理解，就是真正存在磁盘上的用来组织文件系统的数据结构（比如inode，block等等），还有in-memory copy of on-disk data structure，这个是因为cpu没有办法直接往磁盘上直接修改也不划算，所以会现在内存上有个和磁盘上一模一样的结构，然后先修改在内存上的备份，然后再扔给磁盘驱动程序来将这一块内容完完整整刷到磁盘的对应位置。而in-memory data structure则可以是完全不一样的结构，他可以为了专门进行设计

其实在老师上课讲的那一篇Commutor里面就已经对于内存文件系统的扩展性的bottleneck以及如何设计来规避它都已经将的比较清楚了。然而这个对应的文件系统并没有考虑到如何写到持久性存储器，很多设计都只是基于内存文件系统而言，忽略了很多I/O设计以及容灾比如日志等等。

而其他的部分文件系统也是想考虑实现可扩展性，然而他们依然还是单纯的在磁盘文件系统上进行改进，使用Per-inode的日志形式。另外的一些设计则都是属于小修小补的方式，或者干脆就是针对特定应用场景或者特定数据的优化。

## Intriguing aspects of the paper

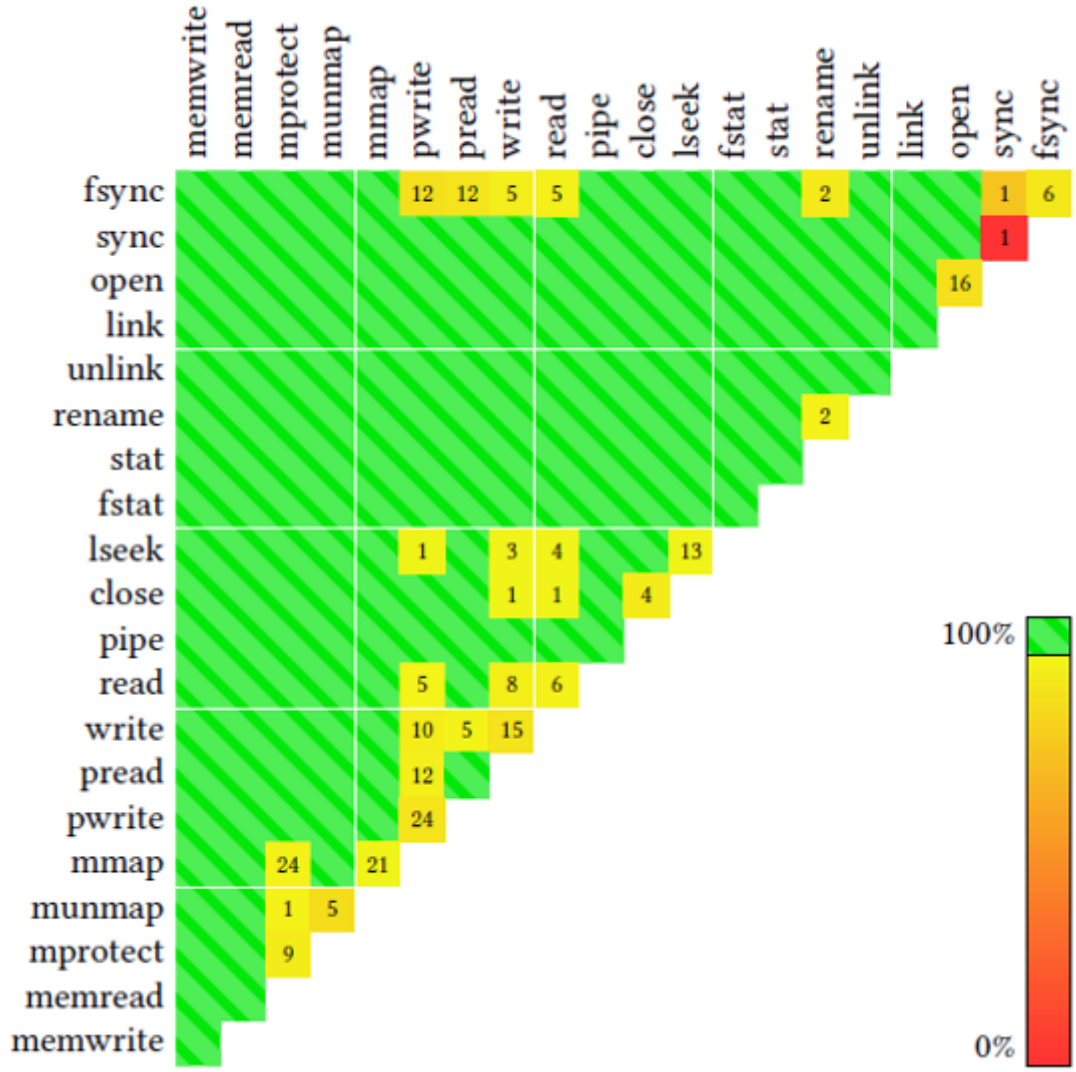
1. 通过分离两个文件系统的想法其实之前也有类似的思想，比如如dcache，乃至数据库里其实也很久以前就有这个想法。然而之前的很多想法分离的都不是很彻底，往往对于内存数据在进行写操作的时候，还会对磁盘数据在内存的表现形式进行写操作。这样尽管读的效果能够有很好的保证，但是写的时候依然还是受限于磁盘的存储形式（无法很好适应多核的格式）。所以这边通过日志形式的彻底分离（不到fsync不同步写操作）的方式使得在绝大多数操作都可以停留在内存里（写oplog也是在内存），从而天生就比较容易进行定制in-memory data structure，而在这个时候是

不需要去特地考虑磁盘文件I/O等特性，做了彻彻底底的分离，使得设计权衡内容变少，更加专注Scalability。而对于on-disk file system他则可以更加的去关注高吞吐量，也可以考虑复用之前的文件系统

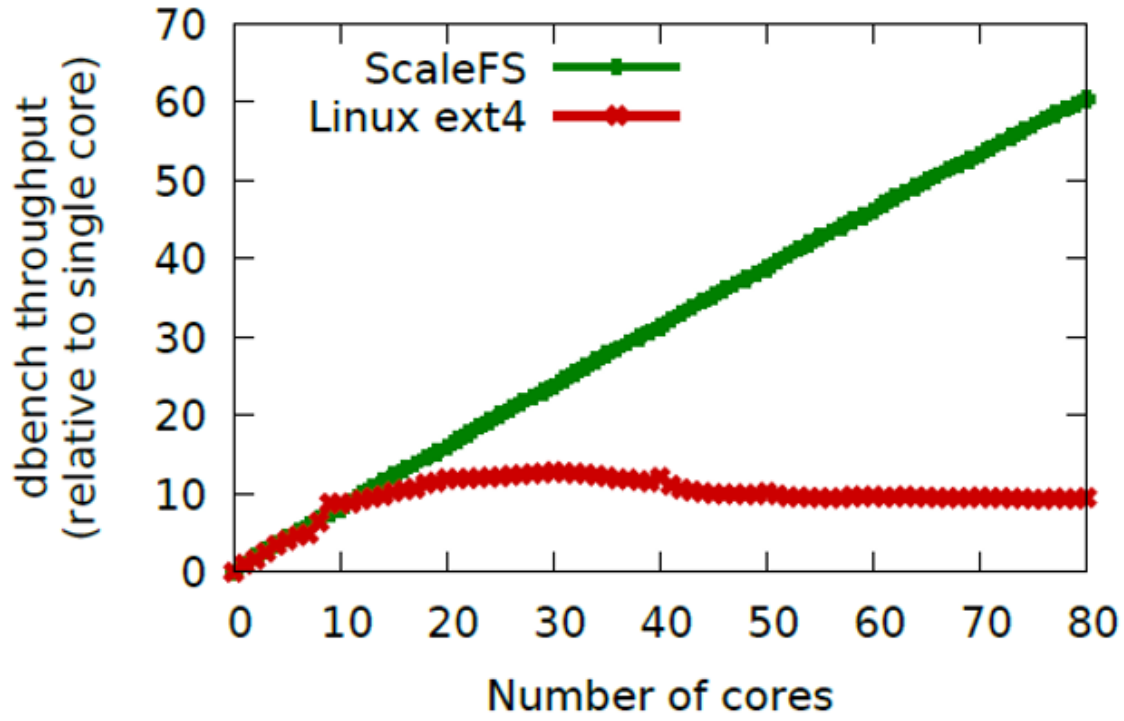
2. 不过分离本身并不吸引人，真正比较有价值的是，分离之后两个系统各司其职，分别完成性能和持久化的目标，这个是我觉得比较有价值的地方
3. 另外文中也依然非常仔细的探讨了各种设计来达到比较好的Scalability，也考虑到了Cache-line contention，所以在MemFS设计的时候，重新设计了一套mnode的机制，在创建的时候不同于inode，大概就是独立自增所以可扩展性还可以，然后同时mnode作为mfile的时候，会使用radix tree来进行存储Page Cache，这个数据结构类似于页表的结构，所以可以比较好的适应动态增长，并且也不是很占用大量空间，同时如果可以很好的利用原子指令来进行lock-free访问，因为每次访问radix的时候基本上到后面几层的时候几乎是不会相关的。而对于目录而言，则是使用了并行哈希（Entry-level lock）来避免并行竞争问题
4. 而他使用的Oplog则是主要用于记录修改了MemFS的mnode的操作（主要是指mdir的link, unlink, rename）操作等，而这些被记录的操作则是能够在之后fsync刷回到DiskFS（需要进行合并整理等等），而为了Scalability，需要被设计为per-core的log。然而为了正确性，论文会使用全局的Timestamp来标记每个操作发生的时间或者也称作线性点，保证了读写的顺序最终在MemFS和DiskFS上的是可以一致的。因此在fsync对操作进行合并的时候，按照时间排序即可。紧接，Oplog作为一个桥接MemFs和DiskFS的系统，之后需要做的事情，就是将这些排好序的操作给最终整理成on-disk data structure上面的所需要修改的块，并且更新在DiskFS中的in-memory presentation of on-data structure上，接下来交给DiskFS来继续刷新操作。注意的是Oplog在往DiskFS书写日志的时候也可以有各种兼并策略。还有个问题就是就算有了Timestamp作为线性点，但是由于不同的操作锁在MemFS上耗费时间不相同，从而可能会存在的临时状态，Timestamp在后面的操作日志先完成与Timestamp在前面的，这个本来不用担心，因为线性点保证了两者一致，然而对于刚才情况，fsync可能因为发生在前面的记录还没写完从而忽略，这样就造成了理论上后发生的内容先被刷回去了，这样就造成了不一致，给使用者带来很大不方便。解决方案就是记录当前cpu运行到哪个操作开始中（线性点）了，然后保证发起fsync的那个时间是比所有核当前记录的时间都小才说明该刷回去的都刷回去了
5. 然后DiskFS已经将需要更新的操作写到了本地的内存备份中，接下里就是要往磁盘写东西了，这个其实很类似正常的日志文件系统，只写Metadata的日志，在之前就已经将数据刷回到磁盘上了。然后这里的可优化的点就是多磁盘或者多通道并发，这个其实也有个per-core journal，然后为了顺序和依赖正确性，这个交由给Oplog生成事务的时候就对需要变化的，同时也进行了等待直到之前依赖的事务已经被刷回去了
6. 然后文中就考虑Cross directory rename的合理性和一致性，简单来说，就是防止由于跨越目录的重命名导致文件丢失，所以当同步源文件的父目录的时候，一定要同时同步目的文件的父目录，放在同一个fsync内完成。另外，对于fsync只能同步自己和自己的孩子，而不能顺便同步父目录，也会导致MemFS和DiskFS看到cross-directory rename的表现不一样从而造成文件夹循环包含，这个也是通过记录从父亲目录到根目录整个路径，当同步自己的时候也需要这条路径都同步一下

# Experiments: test/compare/analyze

实现是通过两方面来进行验证的，首先算是理论上的验证，利用SOSP13的Commutor来进行算是形式化的验证其的Conflict-free程度，如下图：




很明显，一片绿色通过的样子，基本上按照他这样实现方式，能够做到绝大部分情况都能保证Conflict-free，该没有冲突的情况都能做到没有冲突。 然后除了在理论上进行了验证，也通过一些真实的benchmark来做了一些测试，实验比较多在这里展示也没意思，就挑选了个DB（本人相关）的case如下图



可以看到他的throughput很明显比ext4在核多的时候表现的非常好，而且整体也基本上成正比关系，最重要的是在核数比较少的情况下，他也和ext4有一较之力。

# Places the reseach can be improved

1. 首先就是fsync的第一个语义感觉可以加强一些和之前想的不太合理，毕竟修改文件的话，其实不需要同步目录，只有新增文件的时候应该也把目录同步一下，而非让他成为独立的节点（不知道这边不考虑同步父亲目录的原因是什么）。而且这个应该比Cross-directory rename好实现很多，甚至应该作为FS的可选项配置。
2. 感觉这个思想很好，其实可以做成模块化的东西，就是既然DiskFS可以作为任意的底层文件系统，其实上层的MemFS也应该可以有多重实现方式，可以不仅仅为了Scalability，还可以用来其他的功能，而这个模块化可替换的就很好，主要就是实现为两边定制的Oplog
3. 就如他自己也提到的，可以考虑新硬件的情况，可能Bottleneck会发生一些变化，而且考虑DiskFS和Oplog其实需要的内存相比于MemFS的Page Cache基本上不是重点，可以考虑将DiskFS和Oplog放到NVM上，这样的话，很多可持久化的操作可以更加推迟，比如可以只到Oplog差不多要满了再去刷，因为反正这里的掉电也不会损失什么，最多最多fsync就把dirty file page cache刷回去做个时间标记就好了。

# What I would do if i wrote this paper

我会比较在意日志里面的正确性和合理性

以及这里面很多名词进行更加细致的区分，并且强调每个操作和过程到底是从MemFS到oplog，从oplog到DiskFS(in-memory representation of on-disk structure)，从DiskFS最终刷到Disk

而且这边的整个刷回过程其实也让人会有点费解，可能会添加一些专门图来讲一个最极端尽可能包括各种情况的case从头到尾走一遍。

# Survey Paper Lists

[1] Bhat S S , Eqbal R , Clements A T , et al. [ACM Press the 26th Symposium - Shanghai, China (2017.10.28-2017.10.28)] Proceedings of the 26th Symposium on Operating Systems Principles, - SOSP "17 - Scaling a file system to many cores using an operation log[J]. 2017:69-86.

[2] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding manycore scalability of file systems. In Proceedings of the 2016 USENIX Annual Technical Conference, Denver, CO, June 2016.

[3] C. Gruenwald, III, F. Sironi, M. F. Kaashoek, and N. Zeldovich. Hare: a file system for non-cache-coherent multicores. In Proceedings of the 10th ACM EuroSys Conference, Bordeaux, France, Apr. 2015.

[4] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP), pages 1–17, Farmington, PA, Nov. 2013.