关于崩溃一致性问题研究的综述

许平华 2017202110059

### 简介

一致性历来是文件系统设计中的重要问题。设想当用户使用文本编辑器处理完文档并按下保存键后，电脑突然断电，那么用户的文档还在吗？类似这样的问题就被称为“崩溃一致性（crash consistency）”。这样的场景是很常见的，因此在诸如断电、系统故障等情况下如何保持一致性就成为了文件系统设计中的一个挑战。

本文介绍了在目前主流系统中常用的日志机制（journaling）[1][2]和在此基础上开发的较为先进的崩溃一致性文件系统（crash consistency file system，简写为ccfs）[3]

### 日志机制

许多应用为了使它们的用户级数据结构能在一次故障中维持一致性，使用了严密的更新协议来修改数据。更新协议以一连串的系统调用（例如写文件操作、重命名操作）来更新文件或目录，并保证可恢复性。例如，考虑一个简单的数据库管理系统，它将用户的数据存放在单个数据库文件中。为了在一次系统故障时能保证事务的原子性，该数据库管理系统可以使用一种日志更新协议（update protocol）。如果故障发生，数据库管理系统会执行恢复协议（recovery protocol）：如果数据库文件只有部分被更新了，那么系统会从日志文件（journal）重新执行完整的更新操作。

在文献[1]中有对日志机制及其改进的详细介绍，应用这种机制的文件系统是在更新文件状态前先写数据，且这两者有着严格的时间先后顺序。现在讨论Linux ext3如何在文件系统中加入日志机制，ext3的大多数磁盘上的结构与ext2类似，磁盘被分成了多个块，每个块都有一个索引号（inode）和数据位图（bitmap）。那么最关键的问题就是日志本身，它在现有的划分中占据一些小的空间或被放在其他设备上。现在假设日志被放置在相同的文件系统视图中，图1是带有日志的ext3文件系统。

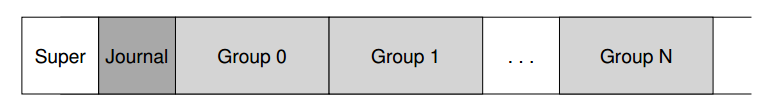


图1 带有journal的ext3示意图

假设现在要往磁盘里面写入inode，bitmap以及一个datablock，那么为了确保安全，我们先将这些数据写入磁盘的日志中，结果如图2所示。整个的写入操作的过程分为两个步骤：1.写日志，将结果TxB和TxW包装的数据写入磁盘的日志区域，并且等候操作的完成；2.检查点，将数据写入到最终地址。

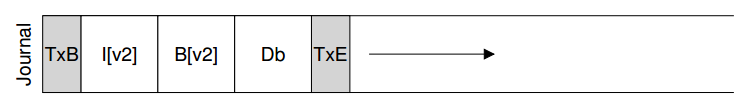


图2 journal数据写入示意图

但这样的操作可能会出现错误，因为在真实写操作时，磁盘内部可能因为调度而将写的顺序改变。这样一来，可能出现磁盘除datablock外的结果已写完，但在写datablock时发生错误。为了解决这个问题，文件系统将事务写分为了两步：1.将除TxE以外的所有块写入日志，等候完成；2.写TxE，等候完成。

传统的日志机制会带来重复写的问题，数据会先被写入日志空间，然后才会被写入文件中。解决这个问题对于维持数据管理应用的版本一致性是至关重要的。在文献[2]中，Chen Jie等人基于传统的日志机制提出了sJournal，其架构如图3所示。通过引入sJournal可以有效地保证上层文件系统的一致性。

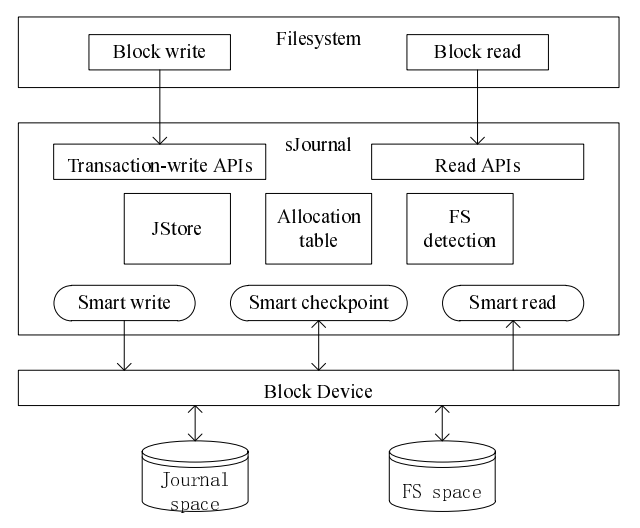


图3 sJournal框架图

sJournal从上层文件系统级接管I/O流，然后在文件系统空间和日志空间之间智能地将其重定向。为了实现这个功能，sJournal设计并引入了两个关键的数据结构：Allocation Table和JStore。

其中，Allocation Table维持所有文件系统空间中的块在硬盘上的分配情况，因此sJournal可以判断一次写操作是否重复。而JStore维持登记在日志中的块向它们在文件系统空间中固定位置的映射，因此sJournal可以在目标块还未做检查点操作时将读流重定向到日志。如图4所示，JStore以B树的结构组织数据，共包含3棵B树：根树、重定向树以及分配树。

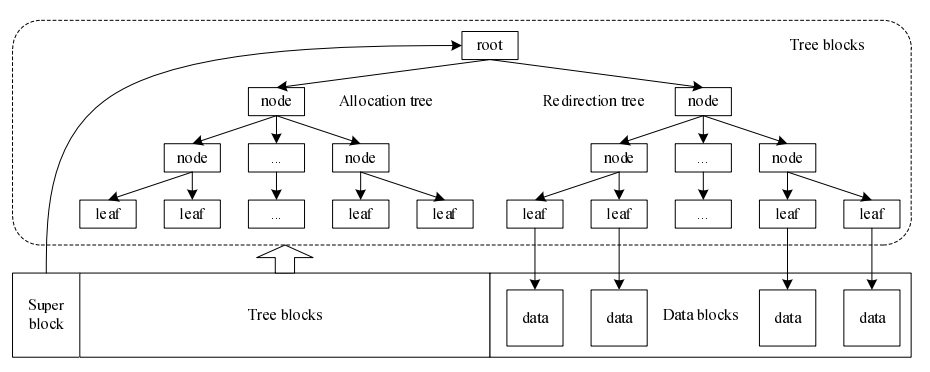


图4 JStore层次结构

JStore将日志空间分为三部分：超级块、元数据组和数据组。超级块总是被存储在日志空间的某一固定位置（常常是第一个块），它记录了日志空间的基本使用信息以及根树的根块的地址。元数据组被用来存储所有的树块，而数据组被用来存储所有的被登记的数据。

当一个数据块需要被登记到日志时，JStore首先为其在数据组中分配一个新的空块，然后将数据写重定向到分配好的地址，最后在重定向树中插入新的键值对。

sJournal通过start\_transaction、make\_dirty\_buffer、commit\_transaction这3个写事务接口为上层文件系统提供日志更新保护机制。文件系统在声明写或更新操作前应当调用start\_transaction；当文件系统需要清空所有的更改时应当调用commit\_transaction；在一次事务中，每当文件系统缓存被更改，都应当通过make\_dirty\_buffer来通知sJournal。当一个事务被提交，所有的被标记的文件系统缓存都会被一个智能的逻辑处理。

图5描述了智能写过程。对于每一个被标记文件系统缓存，智能写逻辑会通过查询Allocation Table来判断写操作是否重复。如果写操作未重复，数据将被添加到文件系统空间的写队列中，否则它将通过JStore被登记到日志空间，然后写请求将会被添加到日志空间的写队列中。在3种条件下，触发器可以清空队列：1.队列长度超过限制长度；2.超时；3.调用检查点。当所有的被标记文件系统缓存都已经被发布，JStore中被更改的块将会被添加到日志的写队列。当所有的清洗操作已完成，智能写逻辑会将JStore超级块冲洗到硬盘。当超级块到达了硬盘，文件系统将会新建一个一致性视图。最后，FS检测将会根据来自上层文件系统的新的持久化分配信息更新Allocation Table。

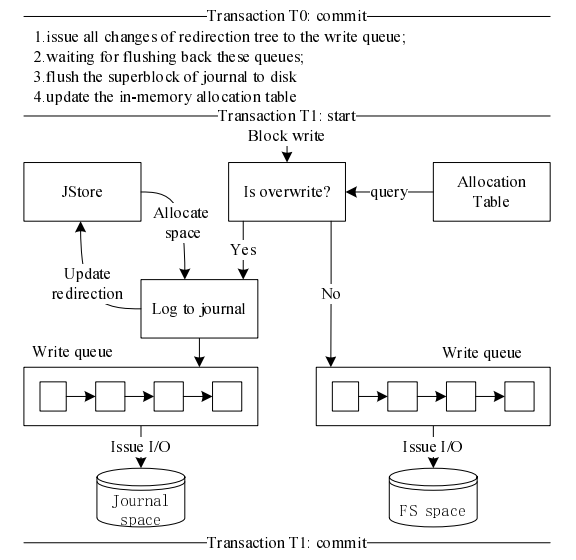


图5 智能写逻辑

图6描述了智能读过程。智能读逻辑在收到一个读块请求后，首先会判断是否要通过查询JStore将请求块重定向到日志空间。如果目标块在日志空间中，读请求会被重定向到日志空间。

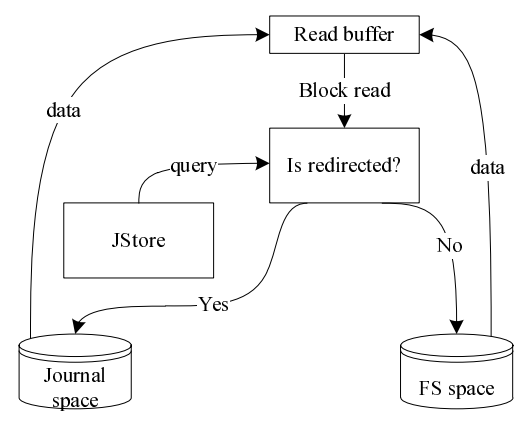


图6 智能读逻辑

图7描述了智能检查点过程。先从日志空间读被登记的数据，然后将其写回在文件系统空间中的固定位置。与Ext3中不同，登记在日志中的数据可以再内存中被释放，因此检查点的时间比较自由。检查点的唯一硬性条件是日志的空余空间小于限额。另外，通过向文件系统空间添加被正确登记的数据，普通的写操作流可以与检查点合并。

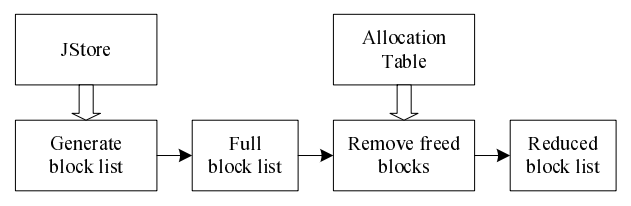


图7 智能检查点逻辑

正确地实现崩溃一致性协议已经被证明是十分困难的。首先，正确性与更新协议的准确语义关系紧密。因为文件系统先将写操作缓冲在内存区中，然后再将它们送到磁盘上，从一个应用的角度来说，这些系统调用的顺序可能会被打乱。如图8所示，数据库管理系统先将数据写入磁盘中的日志中，确保写入完成之后再将数据写入到它们最终的地址同时更新状态表明写入成功。

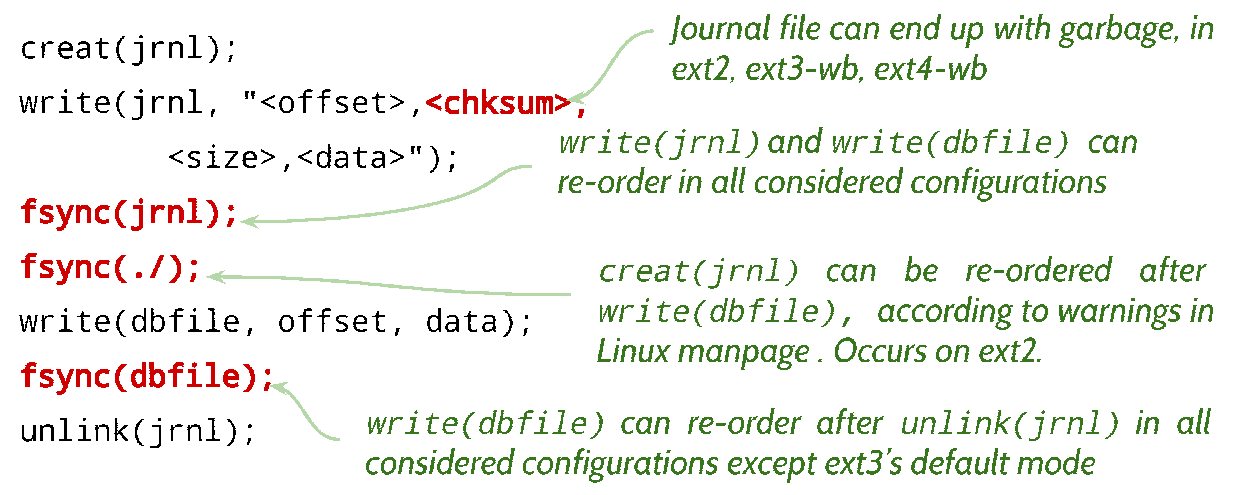


图8 Journaling更新协议示例

但是，对日志文件的unlink操作可能会因为调用顺序被打乱而在更新数据库文件之前被执行。但在图8中，系统调用fsync被用来在unlink操作前同步更新到磁盘。而且，这些系统调用的语义可能会因文件系统的不同而发生改变。例如，上述的乱序情形会发生在ext2、ext4和xfs中却不会发生在ext3中。

另外，恢复协议必须考虑到故障发生时各种可能的状态。由于在每次对数据存储的修正中，该协议都会被用到，因此应用的开发者尝试使恢复协议能够尽量应付更多不同的状况。

最后，和并发性机制相似，产生崩溃的条件无法确定，崩溃一致性协议难以被验证。由于协议的实现的有效性与应用的数据结构关联紧密，在整个应用中复用单个确定的协议的实现是不现实的。要保持崩溃一致性是如此的复杂，以至于许多知名开发公司的产品中同样存在缺陷。

### Ccfs

在文献[3]中，Pillai等人提出了文件系统ccfs（Crash-Consistent File System），试图在保证正确性的同时维持较高的工作效率。首先，他们提出了如下两个假设：

1. 有序性假设。在一个有序的、弱原子性的文件系统上存在更新和恢复协议能够正确地工作。
2. 一个有序的、弱原子性的文件系统可以和其他文件系统具有同样的工作效率。

他们认为应用级更新协议的缺陷来自于对文件系统提供的保障的依赖。第一个是在磁盘上将系统调用持久化的顺序必须与应用声明的顺序一致，一次系统崩溃不会使系统调用产生乱序。第二个较为次要的是当一个应用声明了一定类型的系统调用后，在一次系统崩溃中，这些调用保持原子性。第二个保证又叫做弱原子性，特别是被执行文件操作的系统调用所要求。如果一个系统调用向文件末尾增加数据，不仅增加了文件大小，而且要向新增加的部分写入数据，这两者都应具有原子性。

保证有序性可以大大降低更新协议设计的难度，因为它使崩溃产生时可能的状态数同时也是恢复协议需要处理的状态数大大减少。例如，考虑一个需要向文件中简单地复写n个扇区的更新协议。如果文件系统能保持有序性和弱原子性，那么，只会存在n种崩溃状态；反之，会有种可能的状态。保持有序性可以使恢复变得更加容易。

大多数文件系统都保持了有序性，但目前常用的文件系统均允许重序。虽然保持更新的有序性对崩溃一致性十分重要，但现有的一般观点认为保持有序性会导致性能的下降。不过Pillai等人认为在保持有序性的前提下维持高效率是可行的。例如，在日志型文件系统中，所有的文件系统更新首先会被写入到日志中，一旦在日志中更新被提交，更新可以被传播到它们的目标地址。然而，虽然日志型文件系统保证了完全的写有序，但在这种机制中，每个数据项都必须被写两次，牺牲了很多性能。出于这个原因，大多数的日志型文件系统只使用日志记录元数据。最理想的情况下，一个文件系统应该既有只对元数据使用日志模式的性能，也有对所有数据使用日志的有序性。

因此，ccfs的目标有两个：保持更新的有序性和弱原子性、达到和广泛使用的非有序文件系统类似的性能。Ccfs由ext4改进得到，它拓展了ext4的日志系统，从而以一种高效的方式来保证有序性和弱原子性。在ccfs中，关键的概念是将每个应用都分隔到一个流（stream）中，只在每个流中保持有序性，而对于不同流的写操作可以有重序。要实现这个概念会遇到两个问题：如何在流之间分隔元数据结构和日志机制、如何在每个流中高效地保持有序性。Ccfs在不影响已有的文件系统优化的前提下，解决了上述的两个问题。

Ccfs引入了流的抽象概念，每个应用通常都和一个单独的流相关。来自不同流的写操作可以存在重序，但在一个流中，必须保证有序性以应对系统崩溃。流的概念可以很容易地被应用到一半的工作流中。例如，在用编辑器修改了一个文本文件的同时，从互联网上下载一个二进制文件，随后这两个文件都被加入到集群服务器（VCS）仓库中。最开始，文本编辑器和下载器必须能够操纵它们各自的流A和B，与A相关联，与B相关联。因为对和的位置不做任何限制，所以它们可以被放在同一文件夹下。并且，VCS也能够操作另一个流C来修改和。在这样一个方案中，流的抽象概念应该保持有序性来保证崩溃一致性，同时允许足够的重序来达到尽可能好的性能。因此，在CCFS中流是瞬时的，不会特别地和特定的文件或文件夹绑定。在一个流中被修改的文件之后可能会在另一个流中被修改。由于这样的灵活性，每个流都能互不影响地独立提交，但对于两个处理逻辑相关数据的流需要特别关注。例如，考虑一个被流A创建的文件夹，在该文件夹中有一个文件被流B创建。

从应用的角度来说，文件的创建和文件夹的创建被重序在逻辑上是说不通的。因此，当多个流进行逻辑相关的操作时，文件系统需要给予足够的重视，这样有序性才能被保持。对于“相关”的定义是很宽松的，因此在较短时间内，在不同的流中，相关操作不会频繁发生，否则可能会使文件系统的性能下降。例如，Pillai等人不认为在一个文件夹中创建不同的文件夹是相关的（这经常发生的，例如两个应用在同一个文件夹中创建文件）。流借接口允许所有属于同一应用的进程或线程共享同一个流，但在必要情况下，也允许一个线程在不同流中切换。在这篇文献中，一个新的调用被用来将当前线程和流s联系起来，该线程所有的未来的更新都会被分配到流s中。当执行fork操作时，子线程将继承父进程的流。

Pillai等人采用了一个很简单的方法来分别保持每个流的有序性，Ccfs改进了日志机制来维持多个在内存中运行的事务，每个事务均对应一个流。每当一个出现一个同步系统调用，只有一个对应的流的正在运行的事务会被提交。一个流中所有的变动都和该流正在运行的事务相联系，因此保证了流中的有序性。同时运行多个事务将产生一些麻烦，如果提交其中一个事务，没有提交余下的事务本来就会导致元数据的重序。但是，内部的文件系统一致性依赖于元数据操作间的全局有序性。所以应使不同流中的元数据变动保持逻辑上的独立性，而这些流分别对应正在运行的事务。

在保留现有ext4的优化的前提下，ccfs引入了一系列技术来解决这些问题。前文中描述的日志机制是针对块粒度的：整个块与运行的事务相联系并且提交的事务记录了整个块的变更。但ccfs使用了混合粒度日志，与事务相联系的可以是一系列字节而不需要是整个块，但事务的提交仍然是块粒度的。Ccfs要求字节粒度的日志是因为由不同流所变更的不同元数据结构可能存在于相同的块中。例如，不同应用使用的两个文件的索引号可能包含在同一个块中，在块粒度日志中，将不同流的运行事务与索引号联系起来是无法实现的。不过在块粒度日志中，可以实现许多字节粒度日志无法实现的优化，例如在检查点操作时合并数据。图9解释了混合粒度日志的机制。混合粒度机制在内存中进行日志记录时采用字节粒度，使得流之间能保持独立，并且ext4的延迟日志优化不会受到影响。提交和检查点操作则采用块粒度，因此保留了延迟检查点操作。

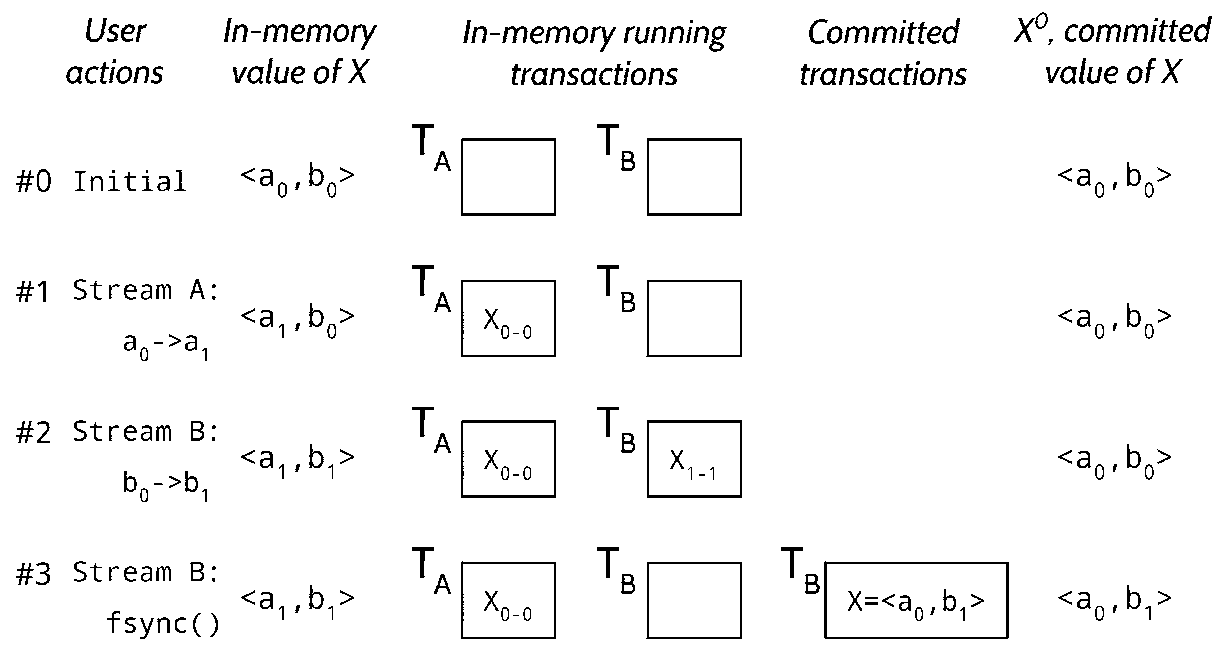


图9 混合粒度日志示例

为了简化字节序列与运行事务的联系，ccfs允许连接特定字节序列上准确的变更，这个技术也被叫做delta日志，在元数据结构确实被不同的流共享时会发挥作用。在ext4中，需要delta日志的共享元数据结构有空索引计数（free inode count）和空块计数（free block count）它们都与文件系统的全局状态有关。

文件系统中的元数据经常使用链表和树等包含内部节点的数据结构，这会使一个流中的元数据操作同时更新其它流中的相关联节点。例如，删除一个链表的表头会要求将其下一个节点更新为表头，这就可能会和其他流产生联系。Ccfs通过采用交替的数据结构来避免这种跨流的更新操作。

Ccfs对文件系统空间分配的管理很严格，因此流之间重序的空间重分配不会影响到文件系统的一致性。论文中就此举了一个例子。假设流A删除了一个文件并释放了索引节点，流B试图创建一个文件，ext4通常会将A释放掉的索引节点分配给B。然而，若B在A之前提交了，就会产生冲突，被恢复的状态会包含两个被分配了一样的索引节点的无关文件。为了解决块分配中的这种问题，ext4只有在释放块的事务已经完全提交的情况下，才会重新使用该块。该篇论文的作者在ccfs中借鉴了这种方法来处理索引节点。

对于维持流内的有序性，下面介绍三种较常用的做法。Ext4仅对元数据使用日志，并允许文件内容添加和重写产生重序。数据日志（data journaling）数据都能保持有序性，但却会显著降低性能，因为它经常会将数据写两次。选择性数据日志（selective data journaling）是一种混合型方法，它只登记重写的文件数据及文件内容添加对应块的指针，并能够保证元数据和文件数据的有序性。由于在现在的工作中，文件内容添加占了绝大部分，因此与登记所有更新相比，选择性数据日志要高效得多。Ccfs采用了第三种方法。但与ext4默认的日志系统相比，在实际生产环境中，ccfs却处于劣势。经ccfs作者分析，是因为ccfs采用了选择性数据日志，但却使一个很关键的优化——延迟分配（delayed allocation）——失效了。如图10所示，若没有延迟分配机制，当一个应用向文件添加内容时，数据块会被立刻分配，块指针也会被立刻分派给文件；反之，文件系统不会立刻分配块，对于多次写的块分配会被延迟并在最后一起分配，大大提高速度，但一般的延迟分配却会破坏选择性数据日志机制的有序性。因此ccfs的作者采用了有序性条件下的延迟分配（order-preserving delayed allocation）技术，使得在保证程序有序性的前提下，系统能够允许延迟分配。

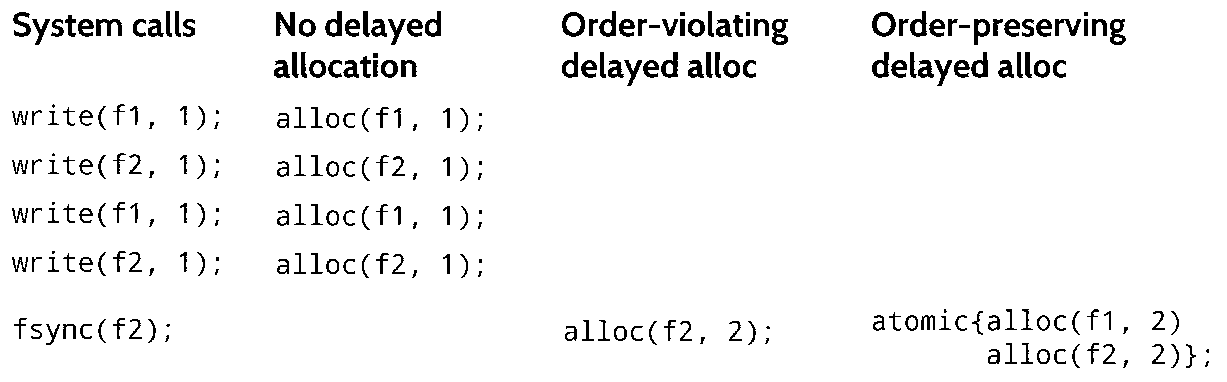


图10 延迟分配机制说明

在ccfs的测试阶段，Pillai等人提出了四个问题，并进行了相关实验：

1. Ccfs是否改善了应用的崩溃一致性？
2. Ccfs是否通过流解决了写依赖的瓶颈？
3. Ccfs在运行单个流的文件系统基准上表现如何？
4. 在真实应用上保证有序性的前提下，ccfs的性能如何？

Pillai等人首先测试了5个在ext4中会出现不一致问题的应用，实验结果如图11所示。在实验中ext4产生了一些问题：当插入键值对时，LevelDB无法维持有序性；在Git和Mercurial的仓库中可能会产生有问题的数据；ZooKeeper可能会无效。但对于ccfs，只在Mercurial中出现了不一致问题。

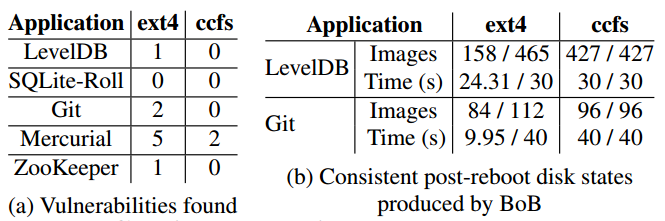


图11 ccfs与ext4的对比试验结果

接着Pillai等人又在三个基准网络上分别测试了ext4，单流的ccfs，多流的ccfs。实验结果如图12所示，与维持全局有序性的文件系统相比，在使用多流的ccfs时，性能良好，可以有效地避免写依赖瓶颈，但是，在单个流中，写依赖会造成很明显的瓶颈。

Pillai等人通过实验说明了ccfs能够在保证有序的前提下，维持高效的性能。同时，ccfs是修改了部分ext4源代码得到的，Pillai等人提出了流的抽象概念并实现了其API，其他人在使用时不会遇到太多开发上的问题。

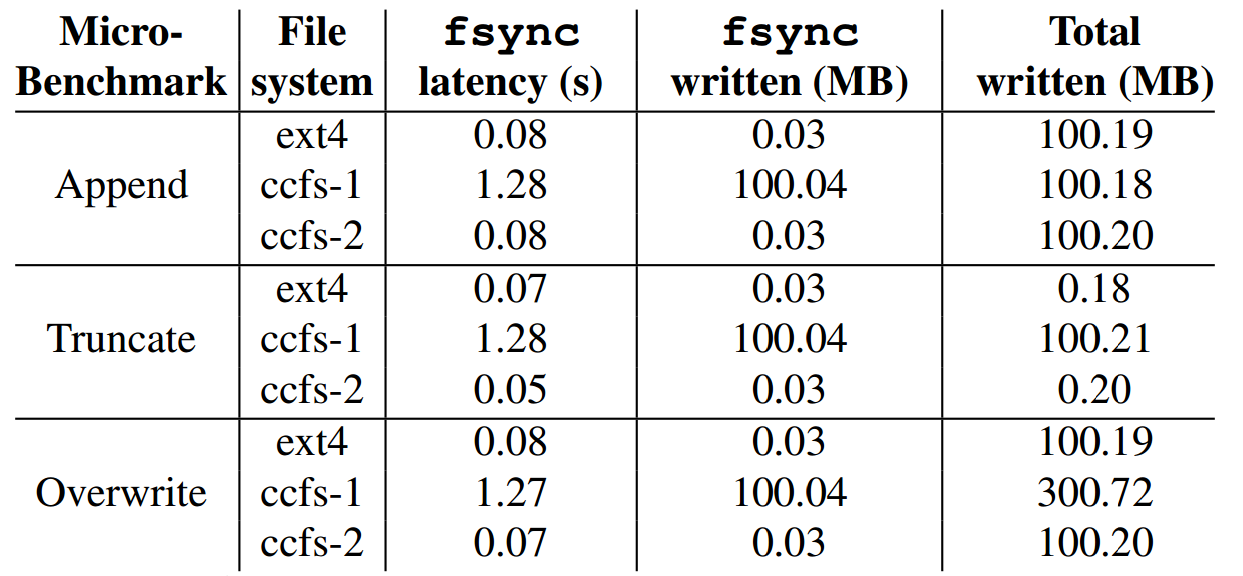


图12 ext4、单流ccfs和多流ccfs的对比试验结果

## 总结

在文件系统中，一致性和性能是相互矛盾的。但在很多时候，提升其中一个的表现，并不会使另外一个的表现折损太多，因此在多数时候，常用的文件系统都是在这两者之间谋求一个折中的方案，使得两者都“牺牲”一点，换回一个“共赢”的局面。日志机制已经发展得比较成熟，被很多常用的文件系统所使用。Pillai等人提出的ccfs使得原始的ext4在一致性上有了较大的提升，但也不可避免地牺牲了少量的性能，但将两者综合考虑，ccfs的总体表现还是非常好的。

## 参考文献

1. Lee E, Yoo S H, Bahn H. Design and Implementation of a Journaling File System for Phase-Change Memory[J]. IEEE Transactions on Computers, 2015, 64(5):1349-1360.
2. Chen J, Tan Z, Wu F, et al. sJournal: A New Design of Journaling for File Systems to Provide Crash Consistency[C]// IEEE International Conference on Networking, Architecture, and Storage. IEEE, 2014:53-62.
3. Arpaci-Dusseau R H, Arpaci-Dusseau R H, Arpaci-Dusseau R H, et al. Application Crash Consistency and Performance with CCFS[J]. Acm Transactions on Storage, 2017, 13(3):19.