文件系统异步持久化的验证

摘要

文件系统是存储和检索永久数据的基石，然而文件系统的设计和实现的复杂性使得其容易存在缺陷，从而不能保证在计算机可能随时宕机的情况下数据的一致性。

形式化验证是目前已知的唯一的保证系统没有编程错误的方法[1]，目前在形式化验证领域有一些杰出的工作,这些工作展示了用形式化验证的方法编写复杂系统的可行性,其中FSCQ[2] 就是一个形式化验证过的具有崩溃安全性的文件系统。FSCQ 文件系统最初的设计和实现的对象是一个同步的文件系统(下文简称 FSCQ-SYNC)，后续为了优化性能，又引入了元数据前缀规范来支持异步持久化(下文将 FSCQ 的后续工作简称为后续工作[20])。然而，后续工作区分元数据和数据的异步持久化策略，本文则统一作为文件数据异步写入磁盘。在这篇工作中，我们以发布的 FSCQ-SYNC 作为基准，重新支持并验证了文件系统的异步持久化。本文提出的 AsyncFSCQ,在 FSCQ-SYNC 的基础上,(1)增加文件系统异步持久化的接口支持(即 fsync)。后续工作在支持异步持久化时修改了文件系统内部所有函数的接口,而我们则利用原有接口实现异步持久化,通过预先中止事务的机制来解决遇到的实现的问题。(2)提出异步持久化接口应该具有的精确的规范。后续工作中为了支持更多的优化(包括区分元数据和数据的异步持久化策略),引入了元数据前缀的方法来描述规范,我们不区分元数据与数据，依然按照原来的规范框架书写规范。(3)证明对于异步持久化的实现符合定义的规范,从而保证 AsyncFSCQ 不会在任何可能的崩溃情况下丢失数据。相比于 FSCQ 后续工作中对于异步持久化的支持，我们通过预先中止事务机制，可以在保留大部分原有实现和证明的基础上支持异步持久化，在保证性能优化的同时减少对于系统带来的修改以及重新证明的复杂度。本文在最后给出了 AsyncFSCQ 测试结果,从结果可见 AsyncFSCQ 的性能相比于 FSCQ-SYNC 已经有了有显著提升。

关键词：形式化验证，文件系统，异步持久化，霍尔逻辑

VERIFICATION OF ASYNCHRONOUS PERSISTENCE IN FILE SYSTEM

ABSTRACT

The file system is the cornerstone of storing and retrieving permanent data, but they are complex enough to have bugs that might cause data loss, especially in the face of system crashes. How to write a bug-free file system remains to be studied.

Formal verification is the only known method to guarantee that a system is free from errors. Currently there are many outstanding works in the area of formal verification, which shows the feasibility of using formal verification method to compose complex systems. FSCQ is one such file system which is verified to be crash safe. The later work of FSCQ add support for deferred durability. To study the formal verification method adopted in FSCQ, we choose sosp15 branch of FSCQ as my start point and implement my version of asynchronous persistence in file system. In this paper, AsyncFSCQ, based on FSCQ, aims to (1) support deferred durability by adding a interface, i.e. fsync(). Compared to FSCQ, they modify all the internal function interfaces to support deferred durability and we use the original interfaces to do that. (2) A precise specification of fync API is brought up. In FSCQ, they use metadata-prefix specification to support more complex optimizations and we adhere to the original specification framework. (3) The implementation of asynchronous persistence is proved to meet its specification. So we can ensure that AsyncFSCQ will not lose data under any sequences of system crashes. The evaluation shows that AsyncFSCQ outperforms FSCQ by an order of magnitude.

Key words: formal verification, file system, deferred durability, Hoare Logic

目 录

第一章 引言------1
1.1 崩溃安全性------1
1.2 规范框架------2
1.3 规范框架------2
1.4 异步持久化------3
1.5 规范框架------3
1.6 概述------4
第二章 背景介绍------5
2.1 相关工作------5
2.1.1 查找和修复文件系统中的漏洞------5
2.1.2 文件系统的形式化验证------5
2.1.3 形式化验证系统软件------5
2.1.4 系统崩溃的验证------6
2.1.5 异步持久化的支持------6
2.2 文件系统基础------6
2.2.1 数据缓存区------6
2.2.2 文件和索引节点------6
2.2.3 目录和路径------6
2.2.4 分配位图------6
2.3 日志协议和优化------7
2.3.1 基本的日志协议------7
2.3.2 一些优化------7
2.4 Coq 编程------8
2.4.1 Coq 编程语言------8
2.4.2 用 Coq 证明------8
2.4.3 库里霍华德同构------8
2.4.4 Ltac 语言------9
2.5 程序正确性------9
2.5.1 霍尔逻辑------9
2.5.2 分离逻辑------9
第三章 异步持久化规范------11
3.1 磁盘模型------11
3.2 崩溃条件------11
3.3 逻辑地址空间------13
3.4 恢复执行语义------15
3.5 异步持久化的规范------16
3.5.1 应用模式分析------16

3.5.2 异步持久化规范的讨论----17
3.5.3 异步持久化的规范说明----17
第四章 证明异步持久化规范----19
4.1 证明自动化----19
4.2 证明崩溃条件规范----20
4.2.1 程序分解----20
4.2.2 谓词蕴涵----21
4.3 证明恢复语义规范----21
4.4 异步持久化的证明----21
第五章 构建一个文件系统----22
5.1 概述----22
5.2 支持异步持久化----22
5.3 预先中止----23
第六章 性能测试----24
6.1 应用性能测试----24
6.1.1 测试设置----24
6.1.2 结果分析----24
6.2 漏洞讨论----25
6.3 规范正确性----25
第七章 结论以及未来的研究方向----26
7.1 提取出高性能的本地代码----26
7.2 验证具有崩溃安全性的应用程序----26
7.3 支持并发----26
参考文献----27
谢辞----29

第一章 引言

本文用形式化验证的方法构建并且验证文件系统的异步持久化功能，基于 FSCQ-SYNC 提出了 AsyncFSCQ，AsyncFSCQ 具有崩溃安全性并且相比于 FSCQ 具有较大的性能提升。

本章接下来的部分会介绍崩溃安全性，这是保证文件系统正确性的关键属性，解释了为什么构建既安全又易于 I/O 的文件系统是困难的，并提出了本文如何在 FSCQ-SYNC 的基础上实现并验证异步持久化。

1.1 崩溃安全性

应用需要依靠文件系统来持久化数据，但即使是仔细编写的文件系统也会存在漏洞，尤其是在系统可能随时崩溃的情况下。当遭遇断电或者软件故障时，系统会由于失败停止运行，紧接着重启。因为应用完成某个操作可能涉及到多次写磁盘，在这多次写磁盘的时间里，系统可能随时发生崩溃，有可能一个写操作只完成一部分，也有可能所有的写操作已经完成，崩溃之后，磁盘会处于一个中间状态，这个状态下数据可能不一致，因此再重启之后通常需要运行一个恢复程序，检查磁盘的状态，将其恢复到一个数据一致的状态，以便将来能够继续正确执行程序，文件系统的这种属性称为崩溃安全性。

保证崩溃安全性对文件系统来说很重要，因为不一致的磁盘状态会导致数据的丢失。例如，在调用重命名这个系统调用的时候，通常会涉及到删除原来的文件，建立新的文件，如果计算机在中间崩溃，有可能新的文件还没有建立，但旧的文件已经删除了，这时候就发生了数据的丢失。

实现崩溃安全性具有挑战性，因为不知道系统会在中间的哪个时刻突然崩溃，想通过穷举所有可能的崩溃场景来验证文件系统的正确性几乎不可能。并且，现代的硬盘驱动通常会增加一层缓存来优化 I/O 实际写入磁盘的顺序，通过写操作的重排序来优化写的性能，文件系统的开发人员必须在适当的位置植入屏障，这个屏障会保证之前代码里面所有的写操作持久化到磁盘，以此来保证顺序性，但屏障是非常耗时的操作，需要尽量的避免，这种异步的 I/O 使得论证崩溃安全性变得更加复杂。

现在比较通用的实现崩溃安全性的方法是使用预写式日志。这时写操作不会直接修改文件系统相应部分的数据结构，而是会先将修改记录在磁盘上的日志区域，当所有操作记录完成后，会进行标志，然后再将数据从日志区域写到实际的磁盘区域，再把日志区域的记录擦除。假如系统在中间发生了崩溃，恢复程序会检查标志是否存在，假如存在，恢复程序会先把所有数据重新写一遍到磁盘上，这样就可以解决之前数据可能只写了一部分的问题。假如不存在，说明之前的数据并没有往磁盘上写，因此可以直接将日志区域的数据擦除，仿佛这个操作完全没有发生一样，写标志这个时间点成为了操作是否发生的时间点，即使在运行恢复程序的过程中系统再次发生崩溃，重启之后再次运行恢复程序依然能正确的恢复磁盘状态。

虽然概念上很简单，但预写式日志会增加许多额外的磁盘写入和磁盘屏障操作，因此实际的文件系统会实现很多优化来尽量增加磁盘吞吐量，这些优化包括将多个事务合并成一个事务以及延迟将缓存中的数据写入磁盘。

目前保证文件系统具有崩溃安全性的办法主要有测试，程序分析和模型检查，虽然他们可以有效的发现文件系统的漏洞，但他们却不能保证文件系统完全没有漏洞。而本文正是要

解决这个问题，如何构建一个具有崩溃安全性的文件系统，并且又能有较好的性能。

1.2 规范框架

用形式化验证的方法编写系统，程序员需要指定系统需要满足的规范，实现系统，然后书写实现符合规范的证明，书写出来的证明可以通过定理自动证明工具来检验是否成立。目前已经验证的系统包括编译器，微内核，内核扩展以及简单的远程服务器，但是在这些系统的规范框架里面都没有包括对系统崩溃的规范。

包括崩溃的规范框架需要解决几个重大的挑战，（1）系统崩溃的规范需要能够描述系统在崩溃发生前系统的状态以及崩溃发生之后恢复系统时面临的状态，同时，还需要考虑系统可能随时发生崩溃的中间状态。（2）这套规范框架必须能够真实的反映硬件的特征，比如磁盘的异步写入，以便能实现良好的 I/O 性能，（3）框架必须能够支持模块化的开发，开发人员可以分开验证系统的各个模块，然后能够将这些模块组合，（4）最后，框架要能支持自动化的证明，当需要对某些定义进行修改时，不需要手动重新实现所有证明。

本文将沿用 FSCQ 中提出的 CHL（Crash Hoare Logic）框架，CHL 框架是对标准的霍尔逻辑的扩展，在其中包括了对崩溃条件的描述，这套框架除了要求程序员指定在一段程序执行的过程中，程序需要满足前置条件和后置条件，同时还要求指定可能发生崩溃之前系统的状态应该符合的不变量。CHL 框架下定义的是一个现实的磁盘模型，在发生崩溃之前可能存在对某个地址一系列的写操作，在没有磁盘同步之前，这些写操作可能一部分的被应用到了磁盘上，因此在崩溃之后，磁盘上可能存在不是最新写入的值。CHL 通过逻辑地址空间的概念来支持以模块化的方法构建系统，并使用不变量来向上层隐藏底层的细节。CHL 中还包括了对于系统恢复的描述，指定了一段程序在运行过程中，假如系统突然宕机，系统恢复时应该运行的恢复程序，既考虑到了正常执行也考虑到了恢复执行的情况。最后采用 CHL 框架书写的规范广泛使用了分离逻辑，分离逻辑有助于证明过程的高度自动化。

1.3 异步持久化

我们首先必须清晰的知道文件系统应有的行为，然后才能为文件系统指定应有的规范。但是，对于文件系统来说，却没有精确地规范存在，比如 POSIX[3]标准里面虽然定义了文件系统的每个系统调用接口的功能，但是对于可能发生崩溃的情况下，这些系统接口究竟能够提供怎样的原子性和有序性的保证，POSIX 标准并没有明确说明，实际上，由于文件系统存在不同程度的优化，使得不同的文件系统的默认实现不尽相同，即使是同一个文件系统在不同的配置下也会有不同的行为。比如对于 ext3 文件系统，具有数据日志和写回两种配置选项，数据日志模式下所有的写操作会保证有序的发送给磁盘进行写入，而在写回模式则没有有序性的保证，应用程序需要自行插入大量的磁盘同步操作来保证顺序，一方面，这使得应用程序难以不做任何修改的在不同文件系统之间移植，另一方面，即使对应用程序进行了仔细的修改，我们依然会遇到很多意想不到的丢失数据的情况。

本文就文件系统异步持久化的接口，fsync()，提出了精确的规范。在 AsyncFSCQ 中，采用组提交的方式来处理事务，所有事务在完成之后，其造成的修改会先缓存在内存中，不会将修改立马写到磁盘里，这样完成单个事务的速度会加快，因为不需要在每次提交之后发起磁盘同步的命令，而是等到多个事务完成之后将修改一起写回到磁盘里，通过这种方法来有效的提高文件系统的 I/O 性能。我们提出的 fsync 接口采用一种简化的设计，它不需要任何的参数，会将缓存中累积的所有修改都写回到磁盘。

有了行为的明确定义之后，应用需要自己在必要的地方通过调用 fsync 来完成数据的持久化。使用我们的 fsync 接口，应用也能够对其自身的正确性进行验证。

1.4 系统架构

本文使用 CHL 框架，在 FSCQ 的基础上实现了 AsyncFSCQ，一个支持异步持久化的的具有崩溃安全性的文件系统，系统使用的是形式化验证领域广泛使用的 Coq 定理证明器，Coq 是一门编程语言，并且同时提供了实现和验证的功能，图 1-1 为系统架构。

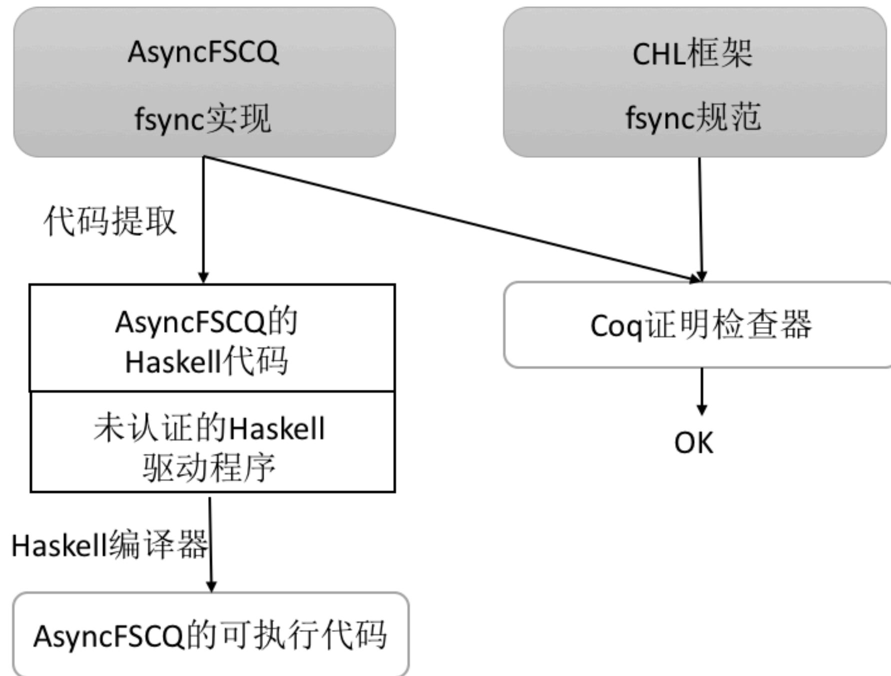

图1-1 系统架构

CHL 框架是用 Coq 实现的一个用来书写规范的框架，它允许文件系统开发人员编写包含崩溃条件和恢复过程的规范，并且提供了较好的证明自动化的支持。

本文使用 CHL 框架实现和验证 AsyncFSCQ 文件系统，采用 CHL 为 fsync 这个函数接口编写规范，fsync 用 Coq 来实现，并证明了其实现是符合定义的规范。CHL 降低了证明的负担，因为它可以自动将函数内部的证明链接起来。尽管 CHL 框架能够提供证明的自动化，但是我们发现，编写规范和证明仍然花费了主要的时间。

我们使用 Coq 的提取代码的功能，提取了 AsyncFSCQ 的 Haskell 实现，我们将这个实现与一个小型的用 Haskell 写的未认证的驱动程序结合使用作为 FUSE 用户级的文件服务器，我们可以在这个文件系统上运行未经修改的 Unix 应用程序。

为了验证系统的实现确实能提高文件系统的 I/O 性能，我编写了小型的基准测试，使用 FUSE 在 Linux 上运行 AsyncFSCQ，实现结果表明，相比于 FSCQ，AsyncFSCQ 显著提升了文件系统的 I/O 性能。

AsyncFSCQ 也有一些不足之处，它有较高的 CPU 开销，因为它从 Coq 中提取出 Haskell 代码来生成可执行的实现，对于异步持久化的优化也只是采用了简单的处理，AsyncFSCQ 还有更多的优化空间，具体将在下文介绍。

1.5 本文贡献

本文主要构建了一个具有高 I/O 性能的具有崩溃安全性的文件系统，我们独立的基于 FSCQ 的 sosp15 版本，重新的实现并验证了异步持久化功能，不同于 FSCQ 的后续工作，我们不区分元数据和数据，使得实现更加简明，本文具体有如下的一些贡献：

- 相比于后续工作对原系统内部函数接口做尽可能小的修改，实现了文件系统的异步持久化功能，即 fsync()接口，fsync()的实现采用了组提交的方式,单一事务在提交之后，数据并没有实际写到磁盘，而是累积多个事务之后异步持久化的磁盘上。
- 精确定义了 fsync()的规范，fsync()会把所有的未持久化的数据和元数据一起刷到磁盘里，应用需要通过加入 fsync()调用来保证关键数据的持久化，后续工作还实现了 fdatasync()接口，并使用元数据前缀方法对规范进行说明。
- 证明了 fsync()的实现符合定义的规范，保证了在调用 fsync()的过程中系统突然崩溃也不会产生数据的丢失，具有崩溃安全性。
- 通过测试验证了支持异步持久化之后，AsyncFSCQ 相比于 FSCQ 具有显著的 I/O 性能提升。

1.6 概述

本文的剩余部分将深入的描述如何实现并验证文件系统异步持久化的功能。

本文首先在第二章描述相关工作以及背景介绍。

然后在第三章更深入的介绍 CHL 规范框架，我们首先介绍 CHL 框架假设的异步的磁盘模型，接着展示如何使用 CHL 框架的崩溃条件，逻辑地址空间以及恢复程序执行语义来书写程序的规范，接着给出 fsync()的规范并说明。

在第四章，我们会给出 fsync()的实现，并说明如何证明实现符合规范，以及如何使用自动化的方法来减轻证明过程中的负担。

接着在第五章，我们会说明 AsyncFSCQ 文件系统对 FSCQ 系统的修改，包括如何尽量减少对原有系统的影响来增加新的功能，以及一些其他需要注意的细节和在构建 AsyncFSCQ 的过程中的教训和经验。

在第六章，我们会评估 AsyncFSCQ 的性能，正确性。

最后在第七章对本文进行一个总结并且提出一些未来可以探索的方向。

第二章 背景介绍

本章首先对构建一个具有崩溃安全性的文件系统的相关工作进行介绍，为了更好的理解增加异步持久化功能中面对的挑战，接着会介绍现代文件系统的一些基础知识，描述一个简单的日志协议和一些优化，然后会介绍 Coq 编程语言相关的背景和函数式编程的基本思想，最后会介绍霍尔逻辑和分离逻辑。

2.1 相关工作

2.1.1 查找和修复文件系统中的漏洞

以前的工作研究了文件系统中的漏洞[4]和应用中对底层文件系统产生不一致的假设的例子[5],最近的一个例子是 2013 年比特币拿出一笔奖励，用于找出一个能损坏存储在 LevelDB 数据库中的财务事务的严重漏洞[6]。

众所周知，应用程序开发人员在确保应用程序即使遇到系统崩溃也能不丢失数据方面很容易犯错。比如，ext4 文件系统曾作出一个修改，改变了应用可以观察到的崩溃行为，应用由于没有及时调用 fsync，导致很多应用在系统崩溃后丢失数据。然而，ext4 的开发人员认为，ext4 从未保证能将所有的写操作一定持久化到磁盘上，因此这个事件可以认为是由于应用的漏洞触发的。类似的行为也出现在了不同文件系统以及其不同的配置选项中，这导致了不同的崩溃行为。

模型检查[7]和模糊测试[8]在检测文件系统漏洞方面是很有效的。他们枚举可能的用户输入和磁盘状态，注入崩溃，然后检查文件系统哪里出现了问题或者违反了不变量，这些方法的确能找出漏洞，我们也会使用它们对 AsyncFSCQ 进行测试。然而，这些方法却不能检查所有可能的执行路径，因此无法保证系统没有任何错误。

2.1.2 文件系统的形式化验证

构建正确的文件系统一直是有吸引力的目标，有许多文献，使用不同的规范语言来形式化证明文件系统，包括 ACL2[9]，Alloy[10]，Athena[11]，Isabelle/HOL[12]，PVS[13]，SSL[14]，Z[15]，KIV[16]，以及他们的组合[17]。大多数的规范说明没有考虑系统崩溃的情况，除了康克和杰克逊的工作，但是他们并没有将规范说明和实现结合起来。

之前的工作大多致力于给出文件系统接口的规范，这些规范采用一种轨迹的方式来书写，然而这种规范的定义却不适合用来定义文件系统的内部结构。

Cogent[18]能为文件系统生成高效的可执行代码，并支持延迟写入，但缺少整个文件系统的规范和证明。由于 AsyncFSCQ 是从 Coq 提取出 Haskell 代码，再进一步获得可执行的文件系统，因此具有较高的 CPU 开销，AsyncFSCQ 可以借鉴 Cogent 中的 DSL 的方法，从而产生更有效的代码，并减少可信基的大小。

2.1.3 形式化验证系统软件

过去十年在形式化验证系统软件方面取得了重大的进步，这也激发了我对于形式化验证领域的尝试。其中 CompCert[21]编译器是一个使用 Coq 验证了的编译器，作为编译器，它并不处理崩溃的情况，但是 FSCQ 中采用了 CompCert 的方法来验证缓存替换算法。

sel4[22]项目使用了 Isabelle 辅助证明工具，由于 sel4 是一个微内核，文件系统不是 sel4 内核中的一部分，sel4 也没有持久状态。

Verve[23]，Bedrock[24]，Ironclad[25]和 CertiKOS[26]都表明，证明自动化可以极大减轻证

明的负担。其中 Bedrock，Verve 和 Ironclad 中都是用了霍尔逻辑的验证自动化。

2.1.4 系统崩溃的验证

系统崩溃是分布式系统需要考虑的核心问题。TLA[27]可以用于证明在节点和网络可能发生故障的情况下，分布式系统依然能遵守某些不变量，然而 TLA 是纯粹关于设计的，而不是可执行代码。Verdi[28]可以将 Coq 编写的分布式系统提取出可执行代码，然而 Verdi 的节点故障模型是对系统重新启动时应该维持的状态的高级描述，被认为是正确的，Verdi 必须依靠其他软件（如文件系统）来提供崩溃状态的证明。FSCQ 提出的 CHL 框架则是对这一问题的解决，通过加入崩溃条件来描述系统崩溃时的状态。

2.1.5 异步持久化的支持

在 FSCQ 的后续工作中，给原先同步的文件系统增加了异步持久化的支持。在异步持久化的状态下，磁盘可能是一系列的状态，为了描述这种情况下的磁盘状态，他们提出了元数据前缀规范的描述方法，在元数据前缀规范下，对元数据修改的操作依次作用在磁盘上，造成了一系列演化的磁盘状态，这一系列状态用磁盘序列这个概念来描述，同时对于这一系列状态下的不变量用磁盘关系来描述。为了支持元数据和数据有着不同异步持久化策略，他们提出了这些规范描述方法，而在本文中不区分元数据和数据的区别，统一作为文件数据保存在内存状态中，累积多个事务之后再一起写入到磁盘，因此可以避免因为元数据和数据的问题而改变原有的内部函数接口和规范描述框架。

2.2 文件系统基础

文件系统的目的是组织和存储数据，在一个类 Unix 系统下，数据存储在一个文件中，一个文件是一个字节数组，几个文件可以放在一个目录里，当然一个目录里也可以有子目录，他们会形成一个树状的分层结构，文件系统则负责维持存储在磁盘上的这些数据结构，例如记录一个文件存储在哪些块上，磁盘上哪些区域是可用的，来提供目录以及文件这些抽象的表示。下面会一一说明。

2.2.1 数据缓存区

访问磁盘比访问内存要慢几个数量级，因此需要在内存中维护一块磁盘的数据缓存区。在读取磁盘时，先从数据缓冲区中查找想要读取的块是否已经缓存了，如果没有的话才会到磁盘去把数据块取回来，在写磁盘时，也会先把数据写到数据缓存区。如果数据缓存区已经满了，缓存区通常会选择把最不常访问的数据驱逐出缓存区，为新的数据腾出空间。数据缓冲区在写时，可以选择直接将数据同步写到磁盘里（直写），或者将写延迟，直到数据块被驱逐的时候再写（写回）。

2.2.2 文件和索引节点

文件的磁盘表示被称为索引节点，它保存文件的重要元数据，例如文件的大小和时间戳，以及属于该文件的数据块的索引列表。文件系统通常会分配一个连续的区域来存储所有的索引节点，每个索引节点具有相同的大小，所以很容易根据索引节点的编号访问到具体的索引节点，这个索引节点编号称为索引值。我们为根目录保留一个特殊的索引值。

2.2.3 目录和路径

目录被实现为一类特殊的文件，其内容是目录条目的序列，每个目录条目由一个名称和一个指向另一个文件或目录的索引值组成。文件系统的用户可以使用路径名来找到目录树中的文件或者目录，该路径名根据目录的路径连接每个父目录的名称。

2.2.4 分配位图

文件系统中使用分配位图来跟踪空闲块和索引节点的使用情况，例如，使用块位图，文件系统可以快速的定位到哪些块是空闲的，而不需要遍历整个目录树。

要了解这些概念如何协同工作，我们可以通过一个具体的例子，比如我们需要将文件

“/path/src”拷贝到“/path/dst”，文件系统首先需要从目录树中定位源文件的位置，为此，它在根目录条目中找到“path”（通常已经缓存在数据缓存区里），拿到path的索引值ipath，将path的目录条目加载到数据缓存区中，定位和名称“src”相对应的索引值isrc，然后文件系统通过读取isrc索引节点的文件内容。要创建目标文件，文件系统将分配一个新的索引节点idst，并在inode的分配位图中标记为已经使用，然后根据src文件的大小来给dst文件分配新的磁盘块，在idst中记录分配的块号，并且在块的的数据位图中将其标记为已使用，再将src的内容写入到新分配的块中。最后，文件系统将一个新的目录条目（“dst”，idst）添加到path的目录条目中，并将目录的新内容写回到，磁盘里。注意，此处的顺序并不能保证崩溃安全性，具体将在下文讨论。

2.3 日志协议和优化

2.3.1 基本的日志协议

文件系统中最有趣的问题之一是如何在系统崩溃之后恢复文件系统，并且不丢失数据。许多文件系统操作涉及多个对磁盘的写。例如，创建文件至少涉及两个写操作，分配未使用的索引节点，并将索引节点添加到父目录中。如果系统在两次写之间崩溃，则索引节点在索引节点位图中可能被标记为已经使用，但却不属于任何一个目录。

文件系统一般会实现预写日志来解决这个问题，以确保系统崩溃重新启动的时候，磁盘上的数据结构处于一致的状态。磁盘通常会分配一块固定的区域给日志，日志通常由两部分构成，一部分是连续的日志条目，每个日志条目记录希望对磁盘进行的写操作，另一部分是一个单一的提交快，记录着日志条目的数量。

一个基本的日志协议工作如下：（1）日志记录系统为文件系统的一个操作开启一个事务。（2）对于该操作引发的每一个磁盘写入的地址和值，日志记录系统向日志条目附加一个新的地址/值对。（3）当日志记录系统提交事务时，它会发出一个磁盘同步来保存磁盘上的日志条目，然后更新提交块以反映日志的长度，然后再次同步磁盘以完成这个事务。（4）日志系统将日志中的修改应用到实际的磁盘位置，并发出磁盘同步将更改持久化到磁盘上。（5）最后，将零写入提交块来截断日志，并同步到磁盘上。（6）无论何时系统崩溃并重新启动，都会运行恢复过程。它根据提交块的长度读取日志；如果日志不为空，则跳到步骤(4)重新执行。

上述协议的正确性依赖于一个重要的假设：更新提交块是原子的同步的操作。原子性确保了提交块上要么是原长度，要么是新长度，同步则保证了磁盘控制器不会对磁盘的写操作进行重新排序，将提交块的写和其他的写调换顺序，这是通过在更新提交块之前和之后发出两个屏障（磁盘同步操作）来强制执行。在这个协议下，在事务中途，还没有更新提交块之前系统崩溃的话，提交块的长度为零，此时磁盘的数据没有被写，这个事务就像没有执行一样，数据处于一致的状态；提交之后但在应用完成之前的系统崩溃将导致提交块长度非零，并且日志中包含相同数目的有效条目，这时就只需要重新执行提交操作，会达到事务执行完的状态。

这个基本的日志协议会增加磁盘写操作以及耗时的磁盘同步操作。例如，写入单个磁盘块的操作将需要完成四次磁盘写入（两次为提交块，两次为块的内容）和四个磁盘同步（在两次提交块更新之前和之后）。现代的成熟的文件系统会实现一些优化来减少磁盘操作的数量。例如，在Linux的ext4文件系统中实现了延迟应用日志，组提交，日志校验码和日志旁路，如下所述。

2.3.2 一些优化

（1）延迟应用日志

日志系统提交事务后，可以延迟应用日志条目，即将（4）推迟执行，等待随后的事务

将更多的条目附加到日志中，当日志填满时，可以一次性应用所有条目，这样多个事务的应用日志操作组合在一起，共用后面的（4）（5）操作，就可以将每个事务的磁盘同步由四个降为两个，同时将写磁盘的次数也由四次降为两次。

（2）组提交

日志系统可以在内存中累积多个系统调用的写入，将他们合并到单个事务中，即将（3）延迟，单个事务的修改先在内存中保存，等累积多个操作后再进行（3）操作，为合并的事务发出一个提交，组提交在多个小事务之间摊销磁盘同步数，并能通过合并多个事务的写入来减少磁盘写入次数。

（3）日志校验和

如果提交块包含所有相关日志条目的校验和，则可以在写日志条目和更新提交块之间省略一次磁盘同步。日志校验和进一步减少磁盘同步，从每个事务两次到只有一次。恢复过程可以通过计算日志条目的校验和并将其与存储在提交块中的校验和进行比较来确定日志条目是否完整从而确定是否需要提交或者终止事务。

（4）日志旁路和延迟写入

使用日志旁路，文件系统可以将文件内容直接写入文件数据块，而不是通过日志预写系统，而元数据（文件属性和目录条目）的更新仍然通过日志。当与回写数据缓存区结合使用时，日志旁路会放松文件系统一致性的保证：通过使用 fsync 和 fdatasync，这样可以独立于其他文件和元数据刷新每个文件的数据。这样对于就地修改数据的应用来说可以提供很好的性能。因为组提交和日志旁路延迟更新磁盘（分别用于元数据和非元数据），所以我们使用一个泛泛的术语来表示它们：延迟写入。

AsyncFSCQ 目前只实现了组提交优化并对正确性进行了证明，后续工作则实现了所有的优化，我们计划在接下来的工作中实现其它的优化。优化的实现是基于 Coq 编程语言，并且对优化证明也需要对优化给出一个精确的规范，用到的 CHL 规范框架是基于霍尔逻辑和分离逻辑，因此在下文中我们将分别介绍这两部分。

2.4 Coq 编程

Coq 是一个证明助手，它旨在开发数学证明，特别是编写形式化的规范和程序，然后证明程序符合它的规范。Coq 可以自动的从代码中提取出可执行的程序，例如 OCaml 或者 Haskell 的源代码。

属性、程序和证明都以相同的语言形式化，这称为构造演算（CIC），Coq 中所有的逻辑判断实际上是类型判断，Coq 的核心实际上是一种类型检查算法。

2.4.1 Coq 编程语言

Coq 的对象分为两类，一类是 Prop 类型，另一类是 Type 类型。其中 Prop 是针对命题的种类，比如$\forall A\ B: Prop, A \wedge B \rightarrow B \vee B$。Type 则是用于表示数据类型和数学结构的类型，比如函数$Z \Rightarrow Z * Z$，Coq 的一个重要特征在于 Prop 类型和 Type 类型都可以使用递归的方式来定义。从编程的角度来看，Coq 很类似于 Haskell 和 OCaml，是一门函数式编程语言，包括的特征如闭包和高阶函数、惰性计算、递归、没有“副作用”和不修改状态等等。

2.4.2 用 Coq 证明

Coq 里面证明的开发是用户通过使用策略语言来完成的，用户可以按照需要对证明的目标使用策略语言来化简完成证明。

2.4.3 库里霍华德同构

在 Coq 里，程序即证明，我们要证明的对象也具有一个类型，我们在书写证明的时候，实际上是在用策略语言将要证明的对象构造出来的过程，这就是库里霍华德同构表达的，我们书写的程序实际上也是对程序所具有的类型的证明。

2.4.4 Ltac 语言

Ltac 是 Coq 里面的一个特定领域语言，使用 Ltac 我们可以编写能够自动搜索证明，形成决策的策略，从而去自动的解决一些证明问题。这里需要注意的是对 Ltac 的使用也是一个权衡，Ltac 虽然能够帮助证明自动化，但是证明搜索的时间相对于手动证明会更长，我们往往需要在其中找到一个平衡点。

2.5 程序正确性

霍尔逻辑[18]是一个经典的对程序正确性进行论证的系统，分离逻辑[19]则是对霍尔逻辑的扩展，以支持对于存储状态的谓词。

2.5.1 霍尔逻辑

霍尔逻辑的规范通常表示为霍尔三元组，他描述了一段代码的执行是如何改变计算的状态，霍尔三元组定义为：

$$\{P\}\,\mathbb{C}\,\{Q\} \qquad (3\text{-}1)$$

其中 P 对应于在程序 C 运行之前应该保持的前提条件，Q 是后置条件，为了证明一个规范是正确的，我们必须证明，假设 P 在调用 C 之前是成立的，我们必须证明 C 执行之后，状态 Q 是成立的。对于霍尔逻辑，我们有两个基本的推理规则：

$$\frac{\{P\}\,\mathbb{C}_1\,\{Q\} \quad \{Q\}\,\mathbb{C}_2\,\{R\}}{\{P\}\,\mathbb{C}_1;\mathbb{C}_2\,\{R\}} \qquad (3-2)$$

$$\frac{P_1 \Rightarrow P_2 \quad \{P_2\}\,\mathbb{C}\,\{Q_1\} \quad Q_1 \Rightarrow Q_2}{\{P_1\}\,\mathbb{C}\,\{Q_2\}} \qquad (3-3)$$

（3-2）是能够连接两个顺序过程的组合规范，（3-3）则是推论规范，允许加强前置条件或者削弱后置条件。（⇒表示逻辑蕴涵）

2.5.2 分离逻辑

分离逻辑用于组合分别作用在不相交存储空间（例如磁盘）的谓词，可以对存储空间更轻松的做出谓词描述。存储被定义为从地址到值映射的偏函数，存储区域 d 是一个有效的地址集合，这个地址集合里面的每个地址的值都有定义：

$$\mathrm{dom}\ \mathrm{d} = \{\mathrm{a} \mid \mathrm{d(a)} = \mathrm{Some\ v}\} \qquad (3-4)$$

分离逻辑中的基本谓词是（1）空断言，表示的是地址处没有有效的值；（2）点到关系，写成$\mathrm{a} \rightarrow \mathrm{v}$，它断言地址 a 是唯一有效的地址，地址存储的值是 v；（3）分离连接算子（*），假如有两个谓词 P 和 Q，那么 P*Q 表示的就是存储空间可以分为两个互不相交的部分，其中一部分满足 P，另一部分满足 Q。它们的数学表示见下：

$$\mathrm{d} \vDash \mathbf{emp} \iff \mathrm{dom\ d} = \emptyset \qquad (3-5)$$

$$\mathrm{d} \vDash \mathrm{a} \rightarrow \mathrm{v} \iff \mathrm{dom\ d} = \{\mathrm{a}\} \wedge \mathrm{d(a)} = \mathrm{v} \qquad (3-6)$$

$$\mathrm{d} \vDash \mathrm{P} * \mathrm{Q} \iff \exists d_1 d_2, \mathrm{dom\ d} = d_1 \cdot d_2 \ \wedge\ d_1 \vDash \mathrm{P} \ \wedge\ d_2 \vDash \mathrm{Q} \ \wedge\ d_1 \perp d_2 \qquad (3-7)$$

其中$\mathrm{d} \vDash \mathrm{P} * \mathrm{Q}$表示的是存储空间 d 符合 P 和 Q 这两个谓词，$d_1 \perp d_2$表示的是$dom\ d_1 \wedge dom\ d_2 = \emptyset$，“·”是函数联合运算符。

除了霍尔逻辑（3-2）（3-3）的标准推理规则外，分离逻辑还能够支持结构规则：

$$\frac{\{P\}\,\mathbb{C}\,\{Q\}}{\{P * R\}\,\mathbb{C}\,\{Q * R\}} \qquad (3-8)$$

结构规则规定，如果一个程序在满足前置条件 P 和后置条件 Q 的一部分存储空间上运行，那么它也可以安全的运行在一个更大的前置条件满足$P * R$的存储空间上运行，运行之后原来的存储空间依旧满足 Q，额外的那部分空间没有任何改变，依旧满足 R。

结构规则能够很好地支持局部的推理，这是模块化证明的关键属性，我们可以分别证明操作磁盘的不同部分的代码（比如，索引节点区域和块位图区域），然后安全的把它们组装

起来证明建立在它们上面的模块（比如，文件层）。

分离逻辑还可以实现高度的证明自动化，比如，以下的一些规则能够允许我们对推理进行重写，或者从推理的两边同时取消一些项。

$$\frac{}{P \iff P * emp} \qquad \frac{}{P * Q \iff Q * P} \qquad \frac{}{P * Q * R \iff P * (Q * R)}$$

$$\frac{P \Rightarrow Q \quad R \Rightarrow S}{P * R \Longrightarrow Q * S} \qquad \frac{P \Rightarrow Q}{P * R \Longrightarrow Q * R} \tag{3-9}$$

第三章 异步持久化规范

形式化验证的核心是能够证明文件系统在正常操作期间能够正常工作，并能够从任何可能的崩溃中恢复到一致的状态。在正常执行的过程中，文件系统可能会由于忘记清除新分配的目录或间接块的内容，从而导致数据损坏，或者没有使用足够的磁盘同步导致磁盘内容来崩溃后无法恢复。以前的工作表明，Linux 内核中成熟的文件系统在正常运行和系统崩溃时都会包含漏洞。

为了能够证明文件系统的实现完全符合规范，我们首先必须能够写出正确的规范，AsyncFSCQ 中使用 CHL 框架来明确异步持久化的规范，CHL 框架由 FSCQ 中提出，它是基于霍尔逻辑的扩展，扩展部分包括了异步磁盘模型，崩溃条件，逻辑地址空间和恢复执行语义，正是这些功能允许我们声明精确的规范，并且实现证明他们，本章的第一到第四节将介绍 CHL 框架，然后在第五节描述我们如何使用 CHL 框架。

3.1 磁盘模型

CHL 专用于证明操纵假想磁盘状态的域专用语言的属性，依照分离逻辑的假设，磁盘被定义成从整数块号到内容的偏函数，其中超过磁盘范围的块号不会映射到任何东西。

在定义域专用语言的时候，一个有趣的问题是如何构建异步磁盘模型。现代的硬盘驱动器通常具有内部写缓冲区。发给磁盘的写入不会立即持久化，最近的写往往会被缓存在内部缓存中，并且由硬盘驱动器来决定什么时候会将这些未完成的写刷到磁盘。更重要的是，硬盘驱动器可以对这些写重新排序进行写入以提高磁盘吞吐量。假如系统没有崩溃，一个读总是能够读到最新写得值，但是假如系统崩溃并重新启动，则可能会发现崩溃之前稍后发出的写持久到了磁盘，但是之前的写却没有。硬盘驱动器还提供了写屏障操作，称为磁盘同步，这会强制将写的缓存全部刷到磁盘上，磁盘同步操作可以用来保证不会有未完成的写操作，之前所有的写都会保留在磁盘上。但是磁盘同步非常的耗时，应当尽可能的避免。

为了捕获这种异步的性质,CHL 用一个值集来描述一个块的内容,而不是一个单一的值，值集是一个非空的块值的集合，用（v,vs）来表示，其中 v 代表着这个地址最近写入的值，而 vs 是一系列的之前写入的值，假如系统突然崩溃，v 和 vs 中的任何一个值都可能出现在磁盘上。而磁盘同步操作意味着 vs 为∅，最新的值会出现在崩溃之后，否则，这个磁盘块就是未同步的。

CHL 的磁盘模型假设块是原子写的，也就是说，在系统崩溃之后，块必须包含最后写入的值或者是先前值中的一个，并且不允许部分块写入。这是文件系统一个共同的假设，只要每个块都是一个扇区，并且这和很多实际中的磁盘的行为相匹配，现代的磁盘大都有 4KB 的扇区。

3.2 崩溃条件

给定了异步磁盘模型，假如没有系统崩溃的情况，开发人员就可以使用霍尔逻辑和分离逻辑写出关于磁盘状态的谓词，但如果考虑系统崩溃，我们需要一种手段来描述可能崩溃的中间状态。

霍尔逻辑的三元组{P}C{Q}中并没有说明可能的崩溃情况，我们的过程 C 会由一系列的最底层的磁盘操作组成（例如，磁盘读，磁盘写，磁盘同步），在其中散布着计算，这些操

作会在磁盘上操作持久化的状态，崩溃可能导致 C 在其中的任一点处停止，这时后置条件 Q 可能并不成立。

为了描述可能存在崩溃情况的程序，CHL 将霍尔逻辑扩展成一个具有崩溃条件的四元组，写为：

$$\{P\}\mathbb{C}\{Q\}\{R\} \tag{4-1}$$

其中 Q 依然是后置条件，描述了 C 在没有崩溃情况下的执行结束时的状态，而 R 是描述 C 执行过程中可能发生的中间状态的崩溃条件。

例如，图 3-1 中展示的是 FSCQ 中磁盘写的规范，在使用 CHL 时，这些规范是用 Coq 的代码写的，我们这里展示的是一个更容易阅读的版本。对应于（4-1）中的四元组。

SPEC $\quad disk_write(a, v)$
PRE $\quad disk \vDash F * a \rightarrow (v_0, v_s)$
POST $\quad disk \vDash F * a \rightarrow (v, \{v_0\} \cup v_s)$
CRASH $\quad disk \vDash F * a \rightarrow (v_0, v_s) \vee$
$\quad F * a \rightarrow (v, \{v_0\} \cup v_s)$

图 3-1 FSCQ 中写的规范

规范由四个部分组成，其中第一行对应的是我们推理的过程名称，$disk_write$；其前置条件声明的是磁盘上地址 a 的地方指向的值是一个集合，v_0是最新写入的值，v_s则表示之前写入的值的序列，除 a 以外的地址满足的谓词是 F；其后置条件说的是 a 以外的值依旧满足谓词 F，没有修改，而地址 a 指向的值的集合中，最新的元素是v，旧的值则是由v_0加上v_s组成的序列；由于磁盘写是一个原子操作，因此它要么完成了值的写入，要么没有完成，因此可能发生系统崩溃时有两种状态，一是前置条件的状态，二是后置条件的状态，通过将前置条件和后置条件并起来则表示了系统崩溃时可能的所有状态，即系统崩溃时必须满足的谓词。

磁盘写的规范说明还表明了真实磁盘的两个重要行为，磁盘实际 I/O 可以异步发生，磁盘的写可以重新排序，这样磁盘才能有良好的性能。之所以能真实的描述磁盘的这种特性在于一个地址会指向一组值而不是一个值，指向一组值表明系统崩溃恢复的时候，磁盘上的值可能是之前的某个值，而不同地址之间可能会呈现出不一致的状态，这也表明了磁盘对写进行了重新排序。假如我们用一个地址指向一个值来表达我们的规范，这将表明我们的磁盘在以同步的模式运行，所有的写不会写入缓存中，直接写到磁盘上，但这样子磁盘的性能会大大降低。

CHL 的崩溃条件有一个微妙的地方，他们描述的是系统崩溃之前磁盘的状态，而不是系统崩溃之后。在崩溃之后，CHL 的磁盘模型认为，每个块将从崩溃之前的值的集合中选择一个值，例如在图 3-1 中的前置条件中表明，磁盘的 a 地址指向的可能是一系列值的集合，而不是一个值。选择将崩溃条件描述为崩溃之前的值有一个显著的好处就是，崩溃条件会跟前置条件和后置条件类似。例如，在图 3-1 中，崩溃条件就是前置条件和后置条件合并起来的。而描述崩溃之后的状态需要更复杂的谓词。而且，使得崩溃条件与前置条件和后置条件类似能够有利于证明自动化。

与其他基于霍尔逻辑编写规范的方法类似，CHL 要求开发人员为每个过程（包括内部程序）编写完整的规范（例如，从空闲位图分配对象）。这需要说明准确的前置条件和后置条件。在 CHL 中，开发人员还必须为每个过程编写崩溃条件。实际上，崩溃条件通常比前置条件和后置条件更简单。

3.3 逻辑地址空间

上面的例子说明了 CHL 如何指定关于磁盘内容的谓词，但文件系统通常需要在其它抽象级别表达相似的谓词。考虑对于 Unix 里面的 pwrite 系统调用，想要描述它的规范应该类似于 disk_write，除了它应该描述文件内容中的偏移量和值，而不是磁盘上的块号和块值。直接在磁盘内容上表达这个规范是很乏味的。例如，描述 pwrite 可能需要说明，我们从位图分配了一个新的块，生成了 inode，可能分配了一个间接块，并修改了一些恰好对应于文件中某个偏移量的磁盘块。编写这种复杂的规范也很容易出错，这可能导致试图证明一个错误的规范，从而消耗巨大的人力。

为了以简洁的方式表达这种高级的抽象，我们观察到，这些抽象中的许多地方涉及到逻辑地址空间。例如，磁盘是从磁盘块号到磁盘块内容的地址空间；文件索引层是从文件索引号到文件索引的地址空间；每个文件里，是从该文件中的偏移量到数据的地址空间；目录是从文件名到索引节点号的地址空间。基于这个观察，CHL 扩展了关于磁盘推理的分离逻辑，使得其适用性更广泛，来以同样的方式表达诸如文件，目录或逻辑文件内容的更高级的地址空间。

作为逻辑地址空间的实例，我们考虑 inc_two 的简单过程。inc_two 表达的是一个在一个操作中需要同时更新两个或多个块的文件系统调用的实质，我们先读取两个块的内容，然后将这两个块的内容加一，再写回。该过程使用日志系统提供的 log_begin，开始一个事务，用 log_read 来读取数据，然后用 log_write 来写数据，最后用 log_commit 来提交这个事务。假如在执行 inc_two 的过程中发生了系统崩溃并且重新启动，首先系统会运行 log_recover 这个恢复程序，如果在恢复过程中依然发生系统崩溃，则重新启动后，恢复过程将再次运行。原则上来说恢复过程可能运行若干次。尽管如此，只要 log_recover 完成，日志记录系统就能保证两个写入要么都发生，要么都没有发生，不管经历了多少次系统崩溃。

```
Def inc_two(a1,a2)
    log_begin()
    v1 = log_read(a1)
    v2 = log_read(a2)
    log_write(a1,v1+1)
    log_write(a2,v2+1)
    log_commit()
```

图 3-2 inc_two 的伪代码

为了简化说明，我们假设现在的日志系统只是采用前面描述的基础的日志协议，而没有任何优化的加入。

inc_two 的规范如图 3-3 所示，展示了如何使用地址空间来书写简洁的规范。规范中引入了新的地址空间，start_state 和 new_state，这两个状态是日志系统提供的逻辑的地址空间，这些逻辑地址空间的引入使得开发人员可以陈述一个简明的规范，这个逻辑地址空间展现的是一个同步的系统接口，因此，高层的开发人员可以大大忽略底层的异步磁盘的细节。

SPEC	$inc_two\ (a_1, a_2)$
PRE	$disk \vDash log_rep(NoTxn, start_state)$
	$start_state \vDash F * a_1 \rightarrow (v_x, vs_x) * a_2 \rightarrow (v_y, vs_y)$
POST	$disk \vDash log_rep(NoTxn, new_state)$
	$new_state \vDash F * a_1 \rightarrow (v_x + 1, \emptyset) * a_2 \rightarrow (v_y + 1, \emptyset)$
CRASH	$disk \vDash log_would_recover(start_state, new_state)$

图 3-3 inc_two 的规范

具体而言，在前置条件中，$a_1 \rightarrow (v_x, vs_x)$适用于日志系统提供的逻辑磁盘的地址空间，在后置条件中，$a_1 \rightarrow (v_x + 1, \emptyset)$适用于逻辑磁盘的新的内容。就像物理磁盘一样，这个地址空间也是从地址到值的偏函数。与物理磁盘不同的是，通过日志系统写入逻辑磁盘的操作展现的是一个同步的接口，即整个操作完成之后，能保证要写入地址的值被刷到磁盘上，因此在后置条件中，我们看到的是单个同步的值，这层抽象做的具体的事情就是隐藏更下层的异步的接口，这也是引入地址空间的好处。

为了使得此规范更加简明，我们必须描述在逻辑地址空间的地址 a 有 v 值的意义，我们在 log_rep 将逻辑地址空间和实际的物理磁盘连接起来。例如，我们指定起始状态如何存储在磁盘上，以及如何逻辑的构造新的状态，比如将日志内容应用到起始状态。通过 log_rep 的这层转化，对逻辑地址空间的描述将会变成对物理磁盘的描述，并且隐藏了一些琐碎的细节。

log_rep 将会把逻辑地址空间作为参数，日志系统中有好几个可能的状态，NoTxn 表示当前磁盘没有事务活动，日志系统中还可能存在的状态包括事务处于 ActiveTxn 的状态，活跃状态中的事务所有的修改只会发生在内存中，然后是 FlushedUnsync 状态，表示此时开始进入提交事务的过程，首先把内存中的修改刷到数据缓冲区的日志记录的区域中，未同步表示还没有刷到磁盘上；然后是 Flushed 状态，表示此时已经有了一次磁盘同步，我们将日志的数据区的内容刷到了磁盘上；然后是 CommittedUnsync 状态，表示我们现在将提交块写到了数据缓冲区里，同时还没有同步到磁盘里；然后是 Committed 状态，表示提交块被同步到了磁盘里，这个点之前和之后作为事务是否完成的分界点；然后是 AppliedUnsync 状态，表示现在数据缓存区里将日志的内容实际应用到数据区，最后是 AppliedSync 状态，表示日志应用的工作已经完全反映到了磁盘上，整个事务至此完成。

log_rep 结合日志系统的状态和逻辑磁盘状态准确的反映出物理磁盘上的状态。在语法上，我们用符号$las \vDash predicate$表示逻辑地址空间 las 匹配特定的谓词。就一个具体的例子而言，我们考虑 NoTxn 的表示，如图 3-4，一个无事务的状态下，提交块必须包含 0，start_state 必须为空。

$$
\begin{aligned}
log_rep(NoTxn, start_state) :=\ & \\
& CommittedBlock \rightarrow (0, \emptyset) * \\
& start_state.equal(\mathrm{empty}_state)
\end{aligned}
$$

图 3-4 log_rep 的 NoTxn 状态的代码

inc_two 的崩溃条件使用 log_would_recover 描述了所有可能的崩溃状态从这些崩溃状态要么恢复到 start_state，要么恢复到 cur_state，使用 log_would_recover，我们考虑的所有状态即前文所述的日志系统的八种状态，这些状态的产生源于这个操作深层次的调用中任何可能的崩溃点（比如，在 log_commit 过程里面的崩溃情况）。

3.4 恢复执行语义

计算机遇到崩溃并重新启动后，它会在恢复正常操作之前先运行恢复程序（如log_recover)，崩溃条件和地址空间允许我们指定计算机在执行过程中可能会崩溃的状态，但是，我们还需要一种证明恢复过程正确性的方法，包括在恢复期间的系统崩溃。霍尔逻辑没有提供一种形式，让我们能够表达一个正常的程序在执行的过程中，恢复过程随时可能需要运行。

比如，我们想要证明一个事物能够提供有或无原子性，如果在调用 log_commit 之前，inc_two 崩溃，将不会应用对 log_write 的调用，如果在 log_commit 中崩溃，要么所有的写都反应在磁盘上，要么都没有。要实现这个属性，日志系统必须在每次崩溃之后运行 Log_recover 来将已经提交的事务继续做完，包括在运行 log_recover 本身发生崩溃之后。

log_recover 过程的规范如图 3-5 所示，它指出，从与 log_would_recover(last_state, committed_state)相匹配的任何状态开始，log_recover 会将事务回滚到 last_state 或者将提交的事务继续执行到 committed_state。此外，log_recover 的规范是幂等的，因此它的崩溃条件意味着前置条件，这将允许 log_recover 在执行的过程中崩溃多次。

SPEC	$log_recover()$
PRE	$disk \vDash log_would_recover(last_state, committed_state)$
POST	$disk \vDash log_rep(NoTxn, last_state) \vee$
	$log_rep(NoTxn, committed_state)$
CRASH	$disk \vDash log_would_recover(last_state, committed_state)$

图 3-5 FSCQ 中 log_recover 的规范

为了表达 log_recover 在遇到崩溃之后运行，CHL 提供了恢复执行语义，一般的执行语义下，一个程序的执行要么产生错误状态，崩溃状态或者正常完成状态，恢复执行语义则考虑的是两个程序执行，其中一个是正常程序，另一个是恢复程序，并且要么产生一个错误，一个完成状态（正常程序完成执行）或者一个恢复状态（恢复程序完成执行)。这种模型描述了正常程序尝试完成，但如果系统崩溃，它将开始运行恢复程序，如果再回复过程中系统崩溃，有可能需要多次运行恢复程序，最终达到恢复状态的过程。

具有恢复执行语义的 CHL 规范可以表示为:

$$\{P\}\mathbb{C} \bowtie \mathbb{R}\{Q\} \quad (3-1)$$

其中⋈表示程序 C 和 R 联合执行。该规范指出，假如 C 在满足前置条件 P 下执行，并且每当系统在 C 或 R 本身内部崩溃时执行恢复过程 R，则当 C 或 R 终止时，条件 Q 将成立。

图 3-6 展示了如何使用⋈符号扩展 inc_two 规范以包括恢复执行，后置条件表示，如果 inc_two 在没有崩溃的情况下完成，则两个块都已经更新，或者两者都没有更新，这时一定是因为发生了崩溃，运行恢复程序之后达到了原状态。状态变量 status 指示系统是否达到完成状态或者恢复状态。

SPEC	$inc_two(a_1, a_2) \bowtie log_recover()$
PRE	$disk \vDash log_rep(NoTxn, start_state)$
	$start_state \vDash F * a_1 \rightarrow (v_x, vs_x) * a_2 \rightarrow (v_y, vs_y)$
POST	$disk \vDash log_rep(NoTxn, new_state) \vee$
	$(status = recovered \wedge log_rep(NoTxn, start_state))$
	$new_state \vDash F * a_1 \rightarrow (v_x + 1, \emptyset) * a_2 \rightarrow (v_y + 1, \emptyset)$

图 3-5 inc_two 的恢复执行规范

一些需要说明的情况是，区分一般完成状态和恢复完成状态，将能使我们做出更强的规范说明。另外，inc_two 的恢复执行规范没有崩溃条件，因为 inc_two 总会成功，也许在运行多次 log_recover()之后。

在这个示例中，恢复过程 R 只是 log_recover()，但是建立在日志系统之上的层次的恢复过程可能由几个恢复过程组成，例如，文件系统中的恢复过程包括首先从磁盘读取超级块来查找日志的位置，然后再运行 log_recover()。

3.5 异步持久化的规范

在前面我们介绍了延迟写入的优化会将对磁盘的写先缓存在内存里，并且给用户提供 fsync 和 fdatasync 的接口来发起磁盘同步操作。延迟写入会导致很多可能的崩溃状态，当计算机崩溃的时候，磁盘的状态可以反映自上次调用 fsync 之后系统的状态。因为应用程序必须使用 fsync 来实现自身的崩溃安全性，所以我们需要一种在文件系统的接口级别将延迟写入形式化的方法。

3.5.1 应用模式分析

首先我们来看一下 fsync 和 fdatasync 的使用场景，图 3-6 展示了一个典型的使用 fsync 和 fdatasync 的应用。

```
tmpfile = “crashsafe.tmp”

Def update(filename)
    f = open(tmpfile, “w”)
    …(write files)
    f.close()

    fdatasync(tmpfile)
    rename(tmpfile, filename)
    fsync(dirname(filename))

Def recover()
    unlink(tmpfile)
```

图 3-6 应用以崩溃安全的方式更新文件的伪代码

update(filename)能确保在系统崩溃后，文件名为 filename 的文件将具有其旧内容或者新数据，它不会处于既含有旧的数据又含有新的数据这种中间状态。为了确保这个属性，update(filename)首先将新数据写入临时文件，我们假设文件系统启用了日志旁路和回写缓存，所以将新数据写到文件的数据块的操作不会经过日志，并且新的数据可能尚未写到文件的数据块。

一旦 update(filename)完成向临时文件写入数据，他将调用 fdatasync 强制文件系统将缓冲的临时文件数据块的更改从写回缓存刷新到磁盘并且发出一个磁盘写入屏障。

fdatasync 返回后，update(filename)将使用重命名替换原始文件与新的临时文件。由于文件系统的重命名对于崩溃是原子的，临时文件的内容已经在磁盘上，如果系统在此时崩溃，应用程序将会观察到原始内容或者新内容，最后 update(filename)使用 fsync 将其更改刷新到目录，以便在返回时，应用程序可以确保新数据能够在崩溃后保留下来。

如果在执行 update(filename)的时候系统崩溃，则必须先执行文件系统的恢复代码，在这个应用模式中，假如临时文件存在的话，恢复代码只需要删除临时文件，因为如果临

时文件存在，那么 update(filename)必须在中间崩溃，因此原始文件仍然有其旧的内容。

这种应用模式能够允许较高的性能，因为除了 fsync 和 fdatasync 之外的所有系统调用都可以是异步的。这允许文件系统推迟将磁盘的目录和文件修改，从而允许批量的更新。因此文件系统的高效的实现只需要向磁盘发出两个磁盘同步屏障，一个用于 fdatasync 的调用，另一个用于 fsync。

3.5.2 异步持久化规范的讨论

从规范的角度来看，一个干净朴素的接口可能是允许应用程序将所有写磁盘系统调用封装到单个事务中，该事务负责将所有的写写入到磁盘日志中，虽然这简化了应用程序开发人员的工作，但是能够以安全的方式更新的数据量受到了限制，因为整个事务必须能够放入磁盘上的日志区域，但是文件系统通常有固定大小的日志空间，通常占据总磁盘空间的一部分，实际上，应用程序很可能发出一个不能装入日志空间的写操作，则文件系统必须将这个写操作分解成多个事务，从而导致实际上暴露出非事务的写入。

POSIX 标准里面并没有明确的定义 fsync 和 fdatasync 应该具有的行为，一个尤其容易出错的地方在于 fsync 对于目录的处理。比如考虑下面的例子。

假设一个应用调用了 rename(“d1/f”，“d2/f”)，紧接着，应用接着调用了 fsync(“d1”)将数据持久化，为了性能的考虑，一个符合 POSIX 标准的实现应当只把特定目录的内容刷到磁盘（比如 d1)，这样文件系统就可以不同文件和目录上并行的进行 fsync 的调用。然而，在这个例子里面，我们又可以遭遇到文件 f 的内容被丢失的问题，因为 d1 被同步到了磁盘上，但是可能还没有出现在 d2 目录里面，因为 d2 还没有被同步。因此，如果 fsync 要求接受一个参数目录，然后把这个目录相关的数据持久化到硬盘上，文件系统可能能够获得好的性能，但是要想使应用通过文件系统获得崩溃安全性就很困难了。

因此为了能够使得开发者能够有效地开发崩溃安全性的应用，文件系统可能提供一个没有参数的 fsync 接口，就像，传统的 BSD 系统的语义一样，所有的元数据操作立马同步到磁盘上，而文件块的内容则采用异步的方式同步到磁盘上。这样应用程序就不需要担心在目录上调用 fsync 的问题了，但可以想见，这种处理办法存在的问题是，对于频繁操作元数据的应用，它的性能会很差（比如解压一个 tar 包）。

因此从这个例子看出的是，我们需要解决 fsync 调用在目录上的问题，需要在性能和保持崩溃安全性方面达成一个平衡。

后续工作中提出的规范如下：

（1） fdatasync(f)将 f 的数据内容刷到磁盘上。

（2） fsync 的作用是 fdatasync 的一个超集，既能够将元数据刷到磁盘上也能把数据刷到磁盘上，同时，fsync 会将所有元数据都刷新到磁盘上，即，假如我们调用 fsync（d)，那么我们不会考虑参数 d，而直接将所有的元数据的刷新到磁盘上。

（3） 总而言之，文件系统能够对任何文件的数据块或者是所有的元数据进行刷磁盘操作。这允许文件系统对数据的写入顺序进行乱序。在崩溃之后，元数据的更新表现出来一定是一致的，而数据块的更新可能表现的乱序。

3.5.3 异步持久化的规范说明

在本文中，我们考虑实现一个简单版本的异步持久化。对于文件系统的每一个系统调用，我们都默认每个操作采用组提交的方式，在每个操作提交的时候，对于磁盘的修改只保留在内存里，既没有刷到数据块缓冲区，更没有刷到磁盘里， 这样所有操作对磁盘的修改可以累积起来，真正的刷磁盘的过程在应用手动调用 fsync 的时候，这时候会将之前累计的所有事物的修改一起提交到磁盘上，提交的过程就和最基础的提交协议一样。

我们的规范与后续工作的规范相比，具有的优点在于：

（1） 规范简单，应用在使用 fsync 对数据进行持久化时，语义更明确了，不需要担心会出现元数据或者数据不一致的情况。

（2） 不需要对现有的数据结构进行大的修改，在规范声明和实现的时候可以复用之前的代码，这样也可以尽可能的减少对证明的修改，具体将在下一章介绍。

当然，我们的规范存在以下一些缺陷：

（1） 不能支持应用细粒度的对文件的数据进行持久化以，以及应用需要手动控制重要数据的持久化。

（2） 在应用手动调用持久化之前，所有的事务的修改只存在于内存中，因此假如系统崩溃，那么之前所有的事务都会丢失。这也是异步持久化为了提高性能而做出的折中。

下图是我们对 fsync 指定的规范。

SPEC $fsync\,()$
PRE $disk \vDash log_rep(NoTxn, last_state, deferred_state)$
POST $if\ result = true, disk \vDash log_rep(NoTxn, new_state, empty_state) \vee$
$if\ result = false, disk \vDash log_rep(NoTxn, last_state, empty_state)$
$new_state = replay(last_state, deferred_state)$
CRASH $disk \vDash log_would_recover(last_state, new_state)$

图 3-7 fsync 的规范说明

在 fsync 的规范中，前置条件指出，在调用 fsync 之前，累积了很多个事务对磁盘的修改，这些修改统一放在了 deferred_state 中，这些修改目前存储在内存里，没有持久化。当前磁盘的状态处于 last_state 状态。后置条件里面，对 fsync 的调用结果进行了分类讨论，假如 fsync 成功了，那么 deferred_state 里面累积的修改全都会被持久化到磁盘里，将磁盘变为一个新的状态 new_state，假如 fsync 没有成功，那么磁盘还会保持原来的状态，但是存储在内存中的数据将会丢失，即对应 empty_state。最后在崩溃条件里面，磁盘要么恢复到 last_state，要么恢复到 new_state。

由于在提交的过程中，日志空间的大小有限，为 512 个块，因此累积的事务对磁盘的修改不能超过 512 个磁盘块，一旦超过，就会导致事务提交无法成功，这时候程序就需要手动的调用 fsync 将数据刷到磁盘里，从而腾出空间。

第四章 证明异步持久化规范

前一章描述了我们如何书写异步持久化的规范，本章将介绍我们证明实现符合其规范。在 CHL 框架中，一个很重要的设计原则是利用证明自动化来减少开发人员的证明负担。因此本章首先介绍 CHL 如何使得证明自动化。

即使有证明自动化的帮助，我们仍然需要大量的手动工作来进行验证。因为有很多方面不能完全自动化，接下来我们会说明如何对异步持久化进行证明。

4.1 证明自动化

为了从直观上给出 CHL 如何进行证明自动化，我们首先考虑图 4-1 示例的一个简化的 fsync 的代码以及 4-2 表示的证明过程。

```
Def fsync()
      log_begin()
      log_flush()
      log_apply()
      log_commit()
```

图 4-1 简化的 fsync 实现

图 4-1 里面展示的是用 log_begin 开始一个事务，然后进行事务的提交操作，首先 log_flush 会将内存状态中累积的修改刷到磁盘上的日志区域，然后 log_apply 将日志应用到实际的地址处，最后 log_commit 完成事务的提交。

图 4-2 则是这个程序的控制流，最外围的圆角矩形对应于 fsync()，fsync()有前置条件，后置条件和恢复条件，恢复条件是指经历系统崩溃，运行恢复程序之后达到的状态。小的圆角矩形对应于上层的系统调用用到的程序（比如 log_flush），每个这样的小程序都有前置条件，后置条件和崩溃条件，在图中，控制流是串行的。更复杂的程序可能有更复杂的控制流。包括循环。每一个上层的系统调用都需要一个恢复程序，比如这里的 log_recover，恢复程序是在这个顶层系统调用遇到崩溃时，系统重启之后首先调用的便是恢复程序，恢复程序也有着前置条件，后置条件和崩溃条件。因为崩溃条件可能在任何一个子过程遇到系统崩溃的时候调用，我们用红色的线表示。

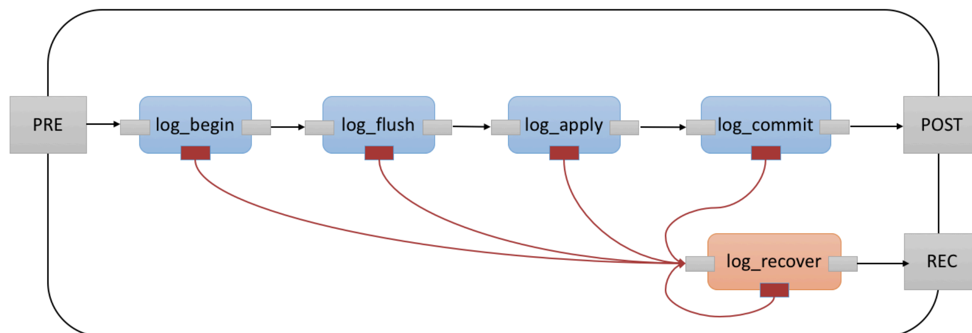

图 4-2 证明一个控制流的例子，圆角矩形表示应用的规范，灰色的矩形表示前置条件、后置条件或者恢复条件，红色的矩形表示崩溃条件，箭头表示控制流和逻辑蕴涵。

为了证明这样一个顶层的程序 C 是正确的，我们需要证明如果 C 要执行，并且在执行之前 C 的前置条件成立，那么要么（1）C 的后置条件的正常运行之后成立，要么（2）恢复条件在恢复程序运行之后成立。

对于第一种正常执行的情况，我们必须证明顶层程序的前置条件蕴含第一个子过程的前置条件，第一个子程序的后置条件蕴含着接下来要执行程序的前置条件，并这样一直下去。

对于第二种设计崩溃的情况，我们则必须证明每个过程的崩溃条件都蕴含这恢复程序的前置条件，恢复程序的后置条件蕴含着顶层程序的恢复条件。

在这两种情况下，逻辑蕴涵关系都按着控制流的方向传递，这允许我们进行证明的自动化，在 CHL 框架里面，有用策略语言写好的证明搜索的 Ltac 程序，假如前置条件很容易被之前程序的后置条件蕴涵，那么我们的 Ltac 程序可以自动完成这个过程，开发人员不需要进行手动的证明，在实际中，通常是这样的情况，开发人员只需要证明程序的不变量的成立。在接下来的章节会更深入的介绍两种情况下的证明过程，并且明确表示出什么需要手动证明。

4.2 证明崩溃条件规范

证明 CHL 规范的过程是重复对现有的目标应用推理规则来不断化简产生新的目标的过程。与霍尔逻辑和分离逻辑类似，CHL 能够支持以下三种规则：组合规则，推论规则和结构规则。

$$\frac{\{P_1\}\,\mathbb{C}_1\,\{P_2\}\{R_1\} \qquad \{P_2\}\,\mathbb{C}_2\,\{P_3\}\{R_2\}}{\{P_1\}\,\mathbb{C}_1;\mathbb{C}_2\,\{P_3\}\{R_1 \vee R_2\}} \tag{4-1}$$

$$\frac{P' \Rightarrow P \qquad \{P\}\,\mathbb{C}\,\{Q\}\{R\} \qquad Q \Rightarrow Q' \quad R \Rightarrow R'}{\{P'\}\,\mathbb{C}\,\{Q'\}\{R'\}} \tag{4-2}$$

$$\frac{\{P\}\,\mathbb{C}\,\{Q\}\{R\}}{\{P * S\}\,\mathbb{C}\,\{Q * S\}\{R * S\}} \tag{4-3}$$

CHL 在证明的时候可以自动运用以上的规则，现在我们具体考虑如何证明崩溃条件下的规范，分为程序分解和谓词蕴涵两个步骤。

4.2.1 程序分解

CHL 的第一个阶段是使用组合规则将证明目标分解成一系列的证明义务，这一系列的证明义务将在下一阶段得到证明。

具体来说，CHL 会考虑 C 中的每个步骤，并查看每个步骤的前置条件和后置条件，我们假设每个小的步骤已经有一个证明好了的规范，比如最基本的磁盘读写，同步操作。CHL 首先假设 C 的前置条件成立，并且为每一个步骤产生两个证明义务，（1）当前的条件，要么是 C 的前置条件，要么是前一个步骤的后置条件都蕴涵这个当前步骤的前置条件，以及（2）当前步骤的崩溃条件蕴涵 C 的崩溃条件。对程序 C 的最后一个小步骤，会产生最后一个步骤蕴涵 C 的后置条件的证明义务。

比如，考虑图 3-2 中的 inc_two 函数，在图 3-3 中给出了它的具有崩溃条件的规范，首先，CHL 会考虑对 log_begin 的调用，log_begin 的规范在图 4-3 中给出，这会产生两个证明义务，一是 inc_two 的前置条件要蕴涵 log_begin 的前置条件，并且 log_begin 的崩溃条件要蕴涵 inc_two 的崩溃条件。

SPEC log_begin ()
PRE $disk \vDash log_rep(NoTxn, start_state)$
POST $disk \vDash log_rep(ActiveTxn, start_state, start_state)$
CRASH $disk \vDash log_rep\ (NoTxn, start_state)$

图 4-3 log_begin 的规范

4.2.2 谓词蕴涵

有些程序分解之后产生的证明义务很容易证明，比如 inc_two 的前置条件蕴涵 log_begin 的前置条件，CHL 根据一阶逻辑会立即证明这个蕴涵成立。

对于更复杂的情况，CHL 会依赖于分离逻辑的结构规则来证明。当 inc_two 第一次调用 log_write 时，我们需要证明$F' * a_1 \to (v_x, vs_x) * a_2 \to (v_y, vs_y)$蕴涵 $F * a \to (v_0, vs_0)$，CHL 会自动将$a_1 \to (v_x, vs_x)$与$a \to (v_0, vs_0)$匹配，并将这些项从证明义务的两边同时去掉，然后将 F 匹配为$F' * a_2 \to (v_y, vs_y)$，这样会将这个证明义务完成，然后将$a_2 \to (v_y, vs_y)$的信息带入到接下来的证明义务中。

有一些证明义务并不那么直观，需要开发者手动的证明，通常这种情况来自于多个层次的抽象的匹配，一个谓词用到的是一个更高层的表示，另一个谓词用到的是稍微低层次的表示。

4.3 证明恢复语义规范

前面叙述了具有崩溃条件的规范是如何自动化证明的，对于具有恢复条件的规范也同样可以通过证明自动化的方法完成，比如图 3-5 所示的规范，如果恢复程序是幂等的，CHL 可以使用崩溃条件的规范直接来证明具有恢复条件的规范，证明恢复规范的规则称为恢复规则：

$$\frac{\{P\}\,\mathbb{C}\,\{Q\}\{R\} \qquad \{R\}\,\mathbb{R}\,\{S\}\{R\}}{\{P\}\,\mathbb{C} \bowtie \mathbb{R}\,\{Q \vee S\}} \qquad (4-1)$$

恢复规则说的是假如恢复过程的前置条件既是$\mathbb{C}$的崩溃条件又是自身的崩溃条件，如果$\mathbb{C}$和$\mathbb{R}$的后置条件分别是 Q 和 S，那么将$\mathbb{C}$和$\mathbb{R}$一起运行的结果就是$Q \vee S$。

4.4 异步持久化的证明

对于异步持久化的证明也是先证明崩溃条件规范，再证明恢复语义规范。

在崩溃语义规范的证明中，由于 fsync 的实现直接调用日志层次的 gcommit，gcommit 的实现类似于日志的 commit 的实现，做的事情就是将内存状态中累积的修改刷到磁盘上，在实现上也与 commit 类似，都是先检查是否有足够的日志空间，如果空间足够，就按照 2.3.1 中基本日志协议里面的步骤将修改持久化到磁盘中。

这里假设 gcommit 里面的每一个过程调用的规范都已经得到了证明，因此经过程序分解之后，我们会产生一系列的证明义务，即前一个步骤的后置条件蕴涵当前的前置条件，崩溃条件蕴涵 gcommit 的崩溃条件。然后我们会使用用 Ltac 编写的 hoare 策略对目标进行化简，hoare 策略实际上在重复的进行模式匹配，假如蕴涵关系十分简单，hoare 策略可以自动完成证明，证明不了的会作为没有解决的目标，此时就需要手动进行证明。

需要手动证明的部分主要是对于不同层次的谓词的匹配，往往需要我们将定义解开，运用已经证明的一些定理，和上下文中的假设进行证明。

证明了崩溃过程的规范之后，我们可以直接运用恢复规则来证明恢复过程规范，只要我们的恢复过程的崩溃条件蕴涵自身的前置条件并且 fsync 的崩溃条件也蕴涵恢复过程的前置条件，那么证明就可以完成。

第五章 构建一个文件系统

这一章会介绍 AsyncFSCQ，为了提供异步持久化功能，遇到的一些实现上的问题，以及我们为了解决这些问题对 FSCQ 做的修改。

5.1 概述

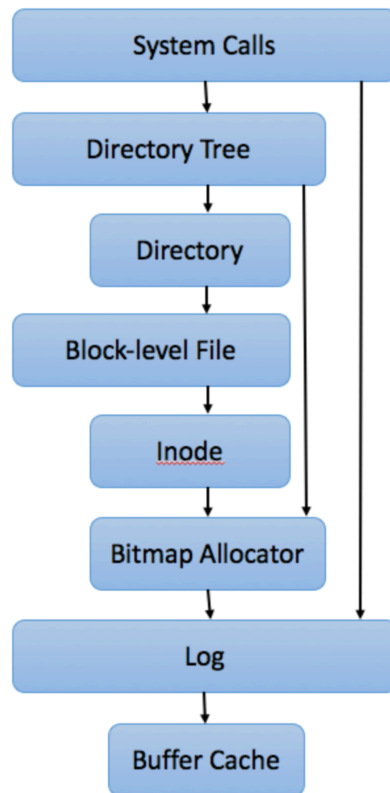

图 5-1 AsyncFSCQ 的系统组成图

图 5-1 展示了 AsyncFSCQ 整体的架构，箭头表示了各部分之间的依赖。最底层的 BufferCache 模块提供了一个磁盘块的写回缓存，所有对于磁盘的访问（包括日志系统的访问）都会经过缓冲块这个模块，日志模块给以上的模块提供了标准的文件抽象的接口。Bitmap Allocator 会实现一个位图分配器，这个位图分配器既用于块的分配，也用于索引节点的分配。Inode 模块实现了索引节点这个层次，Inode 会调用快分配器来分配直接和间接块，Block-level file 会暴露给上层一个块文件的接口，Directory 会在块文件的基础上实现目录，Directory Tree 会将目录和块文件组成一个目录树的结构，它会调用位图分配器来分配或者回收索引节点，最顶层会以事务的方式实现整个文件系统。

图 5-2 展示的是 AsyncFSCQ 的磁盘分布，超级块里包含的是系统里面其他部分在磁盘上的信息，并由 mkfs 初始化。

5.2 支持异步持久化

在 FSCQ 的 sosp15 的版本中，并没有支持异步持久化，所有的事务都用日志层的 begin

和 commit 包裹。

这里需要提及的是在函数式语言的编程中，没有全局变量，而对每个事务而言都需要使用到内存状态和缓存状态这两个全局状态，在 FSCQ 中的处理办法是，所有的函数调用都包括了内存状态的缓存缓存状态。内存状态指的是与每个事务相关的修改的内存的键值映射，而缓存状态指的是磁盘块的缓存。每个新的事务开始之前内存状态都会被置成空，在事务提交的时候会把内存状态中的键值修改全部同步提交到磁盘中。

在支持异步持久化的时候，我们需要累积多个事务对于内存状态的修改，解决这个问题有以下几种方法：

一是我们可以增加一个新的全局变量，这个变量记录累积的内存状态的修改，原来的内存状态变量只记录当前事务造成的内存状态的修改，这从设计上来说是最合理的，然而存在的问题是，这需要我们为所有的函数增加一个参数，这个参数就是全局的内存状态修改的累积，这样一来我们就需要对原来所有的规范和证明进行修改。另一方面，假如出现了另一个需要全局记录的变量，我们依然用这种参数传递的方式加入到我们的所有函数中，这显然不是一个良好的设计思想，最终我们的函数将会有过多不相关的参数，并且不利于后续的开发和维护。

二是我们可以将所有的全局变量都放到一个结构里面，这个结构采取嵌套的定义方法，每个函数只需要一个参数，这个参数包含了从最顶层到最底层需要的所有的全局变量，这也是 FSCQ 的后续工作中所采取的办法，但这样一来需要对所有的函数和规范进行修改，对于一个入门的学习者来说，这值得在更加熟悉证明之后再尝试，因此我也没有选择这样的方法。

三是我们还是用原来的参数，但这个参数表示的含义是累积的内存状态的修改。这样一来我们不需要对原来的所有函数接口进行修改，可以依然使用之前的接口，需要修改的是部分的函数的规范，原先的每个事务开始的时候原本是没有事务的状态，内存状态为空，可现在这个内存状态可能包含之前修改的值。但这种方法存在的问题是，无法处理事务中止的情况，假如一个事物手动调用 abort 接口，想要中止当前的事务，但是将内存状态置为空会导致之前的事务对状态的修改也丢失掉。我们将在下面一个小节提出对这个问题的解决。

5.3 预先中止事务

为了能够支持当前事务的中止，并且不会导致之前事务对内存的修改状态也被消除掉，我们需要在每个可能中止的系统调用之前记录在之前的系统内存状态值，这样才能保证我们可以对当前事务中止并接着使用原来的内存状态继续运行，我们称之为预先中止。

当一个系统调用因为访问到不存在的块，复制一个不存在的文件或者其他问题时，导致当前事务无法继续进行时，我们会捕捉到这个情况，对当前事务进行预先中止，在预先中止的时候，我们会用之前记录的运行当前事务之前的内存状态替换全局的参数，这样一来程序将继续执行在之前的内存状态上，当前事务对内存的修改不会反应在变量上，这样来达到中止事务的目的。

第六章 性能测试

文章会回答关于 AsyncFSCQ 的下列问题，（1）AsyncFSCQ 能达到怎样的性能。（2）AsyncFSCQ 能避免怎样的漏洞。（3）AsyncFSCQ 中异步持久化的规范是否正确。

6.1 应用性能测试

AsyncFSCQ 本身能够运行各种软件，包括邮件服务器等，实际使用中，我们可以在 AsyncFSCQ 上运行 GNU 的核心工具，比如 ls，grep 等，编辑器，比如 vim，emacs，软件开发工具，比如 git，gcc，make 等等。

6.1.1 测试设置

我们分别进行了微基准测试与实际测试，测试是运行在四核联想 G400 机器上，CPU 是 Intel i5-3230M 2.60GHz,12GB DRAM,运行在 Ubuntu 16.04 上,512GB SSD,我们的 AsyncFSCQ 编译在 GHC 8.0.1 上。

我们的测试分别针对 FSCQ 和 AsyncFSCQ 两个文件系统，FSCQ 作为测试基准，在 FSCQ 里面，每次写磁盘的操作都是同步的，而在 AsyncFSCQ 中，默认的事务采用组提交。我们写了一个简单的程序，在这个程序里面我们会总共发出 8192 次写文件的调用，每次写 64 字节，程序采用两层循环，外面一层循环每次会调用刷磁盘操作，里面一层循环相当于每次合并的写操作的次数。并且保证了外面一层循环次数(OUT_ITER)*内部循环次数(IN_ITER)=8192。我们分别将内部循环次数设置为 1，2，4，8，16，32，64，128，256，测试得到的结果如下图 6-1。

我们还测试了典型的软件开发流程，包括首先使用 git 克隆一个项目，编译项目，搜索关键字以及删除文件，测试结果如图 6-2。

6.1.2 结果分析

图 6-1 展示了写文件的微基准测试的结果。

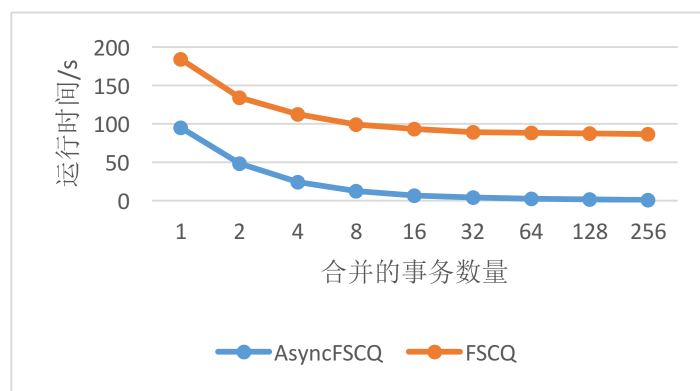

图 6-1 写文件时间测试

结果展示了 AsyncFSCQ 相对于 FSCQ 的 I/O 性能有显著的提升。图的横坐标表示了经过多少个写操作进行一次提交操作，图的纵坐标表示了完成所有写操作所需要的总时间。首先随着每次组合在一起的事务数变多，对于 AsyncFSCQ 来说，所需的时间每次接近减少了一般，原因在于每次调用的 fsync 的次数减少了一半，这说明 fsync 是时间耗费的主要来源。对于 FSCQ 来说，由于每次写操作都会进行同步的写磁盘，但是把多个写操作合并在一起，

可以减少文件的创建操作，因此时间消耗也会减少。

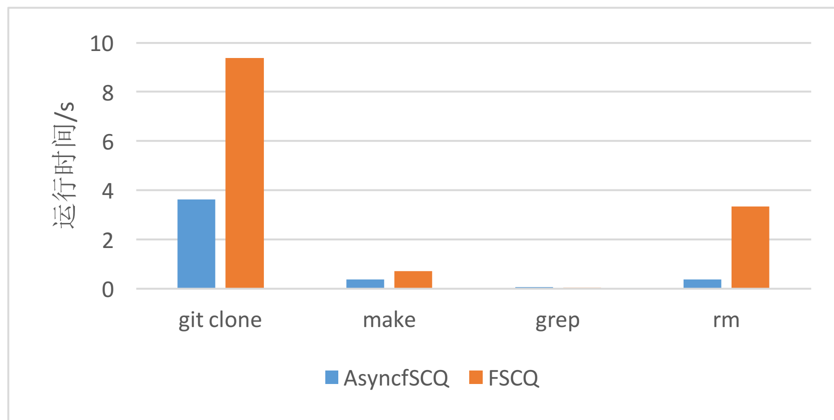

图 6-2 软件开发流程时间测试

图 6-2 展示了软件开发流程的时间测试，这个测试也表明了 AsyncFSCQ 相比于 FSCQ 具有较大的性能提升。由于 grep 操作只涉及到查询操作，没有磁盘的写，因此两者的时间接近。

值得一提的是，对于 AsyncFSCQ 还有性能提升的空间，我们目前只采用了组提交的优化，还有更多的优化方式进一步减少 fsync 的次数。

6.2 漏洞讨论

我们将从两方面讨论 AsyncFSCQ 的实现没有漏洞存在。一是我们通过案例研究，了解了在 linux 的 ext4 文件系统中存在的系统漏洞。二是，我将通过 AsyncFSCQ 的开发过程来阐述来证明正确性的时候，我们能发现漏洞。

通过研究 ext4 文件系统的 git 日志，我收集并非类了在提交中解决的漏洞，通过研究这些漏洞，发现 AsyncFSCQ 包含了这些漏洞可能出现的场景，并且这些漏洞 AsyncFSCQ 都能通过规范来预防掉。并且 AsyncFSCQ 可以通过定理的证明来排除掉这些可能出现的漏洞。

在开发 AsyncFSCQ 的过程中，我经常遇到一些无法证明的规范或者定理，无法证明的原因则来自于实现的问题或者规范的问题。比如在证明日志系统里面相关系统调用的定理成立时，有时候我的前置条件里面缺少了相关的对内存状态的描述，而在后置条件中包含了对内存状态的描述，导致我无法证明定理的成立，从而发现了这个隐藏在规范中的漏洞。

6.3 规范正确性

为了证明异步持久化的规范确实是正确的，我进行了一些实验。

首先，通过运行设计的程序来检查异步持久化的功能是否符合设想。比如通过写一些小程序，在写操作之后没有持久化数据就退出，重启磁盘后看能否恢复原来的数据，只要调用了 fsync 之后才能保证数据的持久化，从而证明了实现符合想要的功能，并且规范和实现一致。

其次，通过手动构造一些定理，比如常用的原子性提交数据的定理，对定理的证明成功，则表示了我们的规范能够允许我们得到想要的性质，从而证明了规范的正确。

第三，通过规范的简洁性，我们可以发现异步持久化的规范是正确的，fsync 的规范总共只有 5 行代码，相比于复杂的实现，我们更容易确认 fsync 的规范是正确的。

	PRE	POST	CRASH	Total
fsync	2	2	1	5

表 6-1 fsync 规范的代码行数

第七章 结论以及未来的研究方向

复杂的文件系统长期以来都有着各种微妙的漏洞，这些漏洞会导致数据的丢失或者数据的泄露，AsyncFSCQ 是一个形式化验证过的文件系统，并且通过支持异步持久化来有效的提高其 I/O 性能，我们在这个系统中对异步持久化相关的系统调用 fsync 提出了规范说明，并且证明了它的实现需要规范说明。

为了实现这个目标，我们首先需要使用 CHL 框架对 fsync 的规范进行精确的描述，我们采用了组提交的思想来实现我们的 fsync，为了减少证明的负担，我们利用了证明自动化，同时在设计 fsync 的实现的时候，我们尽可能的避免对原有的规范的大的修改。

接下来，值得研究的方向还包括以下一些。

7.1 提取出高性能的本地代码

尽管 AsyncFSCQ 能提供较好的 I/O 性能，提取出的 Haskell 的代码导致了巨大的 CPU 开销并且将 Haskell 编译器以及运行时加入到了我们系统的可信基中，我们希望能够产生可信的执行代码，这些代码能够利用较低层的优化，并且将 Haskell 运行时从我们的可信基里面去除。最近有一些工作，比如 COGENT 就在朝这个方向努力，我们希望采用同样的思想来改善这个问题。

7.2 验证具有崩溃安全性的应用程序

在本文中我们提出了一个崩溃安全性的更新文件的方式，但是我们仍然需要在 AsyncFSCQ 的基础上去证明这些应用的正确，并且证明应用的正确仍然需要不小的努力，对于更大的应用如何证明这些应用基于我们的文件系统也具有崩溃安全性的也是一个值得研究的问题。

7.3 支持并发

所有的实际的文件系统都运行在多用户的环境下，并且利用并发来实现良好的性能，目前 AsyncFSCQ 并不支持并发，并且不对任何的共享状态建模，我们需要考虑两种主要形式的并发，（1）I/O 并发，其中一个进程的 I/O 操作和另一个进程的 I/O 操作重合。（2）并行性，不同的核在计算的时候会访问到共享的内存中的状态，（比如对一个缓冲数据块的读写）。证明并发的程序目前还是一个开放的问题，如何先从一个有限的并发来着手解决这个问题也许是一个比较好的着手点。

参考文献

[1] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 207-220.

[2] Chen H, Ziegler D, Chajed T, et al. Using Crash Hoare logic for certifying the FSCQ file system[C]//Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015: 18-37.

[3] IEEE (The Institute of Electrical and Electronics Engineers) and The Open Group. The Open Group base specifications issue 7, 2013 edition (POSIX.1-2008/Cor 1-2013), April 2013

[4] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST), pages 31–44, San Jose, CA, February 2013.

[5] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In Proceedings of the 11th Symposium on Operating Systems Design and Implementa- tion (OSDI), pages 449–464, Broomfield, CO, October 2014.

[6] Robert Escriva. Claiming Bitcoin's bug bounty, November 2013. http://hackingdistributed.com/2013/11/27/bitcoin-leveldb/.

[7] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI), pages 273–287, San Francisco, CA, December 2004.

[8] Dave Jones. Trinity: A Linux system call fuzz tester, 2014. http://codemonkey.org.uk/projects/trinity/.

[9] William R. Bevier and Richard M. Cohen. An executable model of the Synergy file system. Technical Report 121, Computational Logic, Inc., October 1996.

[10] EunsukKangandDanielJackson.FormalmodelingandanalysisofaFlashfilesystem in Alloy. In Proceedings of the 1st Int'l Conference of Abstract State Machines, B and Z, pages 294–308, London, UK, September 2008.

[11] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In Proceedings of the 6th International Conference on Formal Engineering Methods, Seattle, WA, November 2004.

[12] Markus Wenzel. Some aspects of Unix file-system security, August 2014. http://isabelle.in.tum.de/library/HOL/HOL-Unix/Unix.html.

[13] WimH.HesselinkandM.I.Lali.Formalizingahierarchicalfilesystem.InProceedings of the 14th BCS-FACS Refinement Workshop, pages 67–85, December 2009.

[14] Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In Proceedings of the 23rd European Symposium on Programming, pages 169–188, Grenoble, France, 2014.

[15] WilliamR.Bevier,RichardM.Cohen,andJeffTurner.AspecificationfortheSynergy file system. Technical Report 120, Computational Logic, Inc., September 1995.

[16] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jorg Pfähler, and Wolfgang Reif.

Verification of a virtual filesystem switch. In Proceedings of the 5th Working Conference on Verified Software: Theories, Tools and Experiments, Menlo Park, CA, May 2013.

[17] Miguel Alexandre Ferreira and Jose Nuno Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In Proceedings of the 12th Brazilian Symposium on Formal Methods, August 2009.

[18] C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10):576–580, October 1969.

[19] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science, pages 55–74, Copenhagen, Denmark, July 2002.

[20] Chen H. Certifying a Crash-safe File System[D]. Massachusetts Institute of Technology, 2016.

[21] Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, 52(7):107–115, July 2009.

[22] Klein G, Elphinstone K, Heiser G, et al. seL4: Formal verification of an OS kernel[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 207-220.

[23] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verifi- cation of a type-safe operating system. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 99–110, Toronto, Canada, June 2010.

[24] Adam Chlipala. Mostly-automated verification of low-level programs in compu- tational separation logic. In Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 234–245, San Jose, CA, June 2011.

[25] ChrisHawblitzel,JonHowell,JacobR.Lorch,ArjunNarayan,BryanParno,Danfeng Zhang, and Brian Zill. Ironclad Apps: End-to-end security via automated full-system verification. In Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI), pages 165–181, Broomfield, CO, October 2014.

[26] Ronghui Gu, Jeremie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In Proceedings of the 42nd ACM Symposium on Principles of Programming Languages (POPL), Mumbai, India, January 2015.

[27] Leslie Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16(3):872–923, May 1994.

[28] Wilcox J R, Woos D, Panchekha P, et al. Verdi: a framework for implementing and formally verifying distributed systems[C]//ACM SIGPLAN Notices. ACM, 2015, 50(6): 357-368.

[29] GitHub – mit-pdos/fscq at sosp15 https://github.com/mit-pdos/fscq/tree/sosp15

谢辞

我首先想感谢陈海波教授，很高兴有他作为我的导师，他是一位系统领域的通才，在学术研究的各个方面都要求严格我们，是他指引我从事形式化验证方面的研究，这个领域目前还存在很多未解的问题，在这个领域做研究很有挑战性，要学习很多知识与方法，感觉在从另一个角度来认识系统。

我想感谢 IPADS 实验室，这个实验室里的每一位成员，尤其是我们实验室的老大臧院长，在这个实验室里学习成长感觉非常棒，大家总是可以轻松的交流起来，从学习到生活，大家都非常厉害，能力都很强，并且都很平易近人，每当我遇到问题时，总可以找到帮助我的人，这让我倍感压力，同时又不断激励着我。臧老大使我们实验室的灵魂，这个实验室由他一手建立起来，他严谨而又幽默，实验室的方方面面都在他的关怀下显得特别有条不紊而又温馨。

我想感谢我的室友，杜东，他也是 IPADS 实验室的成员，我和他当初一起决定进入这个实验室，我们也一起选择了留在实验室读博士。他对我在形式化验证这个领域的起步起着重要的作用，虽然我们的研究方向不一样，但每当我遇到无法理解的问题时，他都会耐心的倾听，充当我的“小黄鸭”，然后给出他的理解和意见，每次带着问题找他，总能得到满意的结果，有他这样一位同行者，使得科研这条艰深枯涩的道路也充满了乐趣。

当然，我还想感谢我的父母和我的女朋友，感谢他们对我的无条件的爱和支持，有他们在，我遇到再大的困难都有人倾诉，都能重新鼓起勇气，他们给了我生活和学习的目标。

VERIFICATION OF ASYNCHRONOUS PERSISTENCE IN FILE SYSTEM

The file system is the cornerstone of storing and retrieving permanent data, but they are complex enough to have bugs that might cause data loss, especially in the face of system crashes. How to write a bug-free file system remains to be studied.

Formal verification is the only known method to guarantee that a system is free from errors. Currently there are many outstanding works in the area of formal verification, which shows the feasibility of using formal verification method to compose complex systems. FSCQ is one such file system which is verified to be crash safe. The later work of FSCQ add support for deferred durability. To study the formal verification method adopted in FSCQ, we choose sosp15 branch of FSCQ as my start point and implement my version of asynchronous persistence in file system. In this paper, AsyncFSCQ, based on FSCQ, aims to (1) support deferred durability by adding a interface, i.e. fsync(). Compared to FSCQ, they modify all the internal function interfaces to support deferred durability and we use the original interfaces to do that. (2) A precise specification of fync API is brought up. In FSCQ, they use metadata-prefix specification to support more complex optimizations and we adhere to the original specification framework. (3) The implementation of asynchronous persistence is proved to meet its specification. So we can ensure that AsyncFSCQ will not lose data under any sequences of system crashes. The evaluation shows that AsyncFSCQ outperforms FSCQ by an order of magnitude.

In detail, implementing write-ahead logging is the common way to make sure that the on-disk data structure is in a consistent state when the system crashes and restarts. For a naïve logging protocol, many disk writes and expensive disk barriers are imposed. For example, an operation that writes to a single disk block would require 4 disk writes and 4 disk syncs to complete. Sophisticated file systems implements several optimizations to reduce the number of disk operations. For example, Linux ext4 employs deferred apply, group commit, log checksum and log bypass. This dissertation describes how we use group commit to batch multiple transactions so we can amortize number of disk syncs across multiple small transactions and potentially reduces the number of disk writes by merging writes from multiple transactions.

In POSIX specification, the behavior of fsync() and fdatasync() are not well specified. To simplify the question, our fsync() are defined without arguments. All transactions are default to be batched and synced to disk when applications invoke fsync(). If, unfortunately, system crashes before applications are able to call fsync(), the data written to disk after last fsync() are lost. In this way, we will have a clear understanding of what fsync should do in the face of crashes and we specify this specification using the CHL (Crash Hoare Logic) framework. CHL is an extension of Hoare Logic, which is brought up in FSCQ. Hoare Logic is written as a triple, which describes how the execution of a piece of code changes the state of the computation. CHL extends Hoare Logic with a crash condition that allow the developers to specify the state of the disk before the crash and CHL also contains a recovery semantics that allow the developer to specify the recovery procedure

to run if the system were to crash. In addition, CHL abstracts an asynchronous disk model so the disk driver can cache and reorder the writes to disk. To express similar predicates at different level of abstractions, CHL uses logical address space to hide the low level details.

After writing the specification for fsync, we need to prove that the implementation meets its specification. A key challenge is how to reduce the proof burden for the developers. As an example, we consider a simplified version of fsync. It consists of a sequence of call to log_begin, log_flush, log_apply and log_commit. To prove that the specification of fsync is correct, we need to show that if fsync is executed and the precondition is held before execution, either the postcondition holds or the recovery condition holds after recovery finishes. For the first case, we must show that the precondition of fsync implies the precondition of log_begin, and the postcondition of log_begin implies the precondition of log_flush, and so on. Similarly, for the second case, we need to show that the crash condition of all procedures implies the precondition of the recovery procedure. And the crash condition of the recovery procedure also needs to imply its precondition. After successfully executing the recovery procedure, we will reach the recovery condition of fsync.

In this way, we can follow the control flow of the program or function and divide it into a sequence of procedure calls, supposing the specifications of which we have already proved. We then generate a series of proof obligations like described. Some obligations generated are trivial and can be proved automatically. Others involve complicated predicate implications. These predicates are descriptions on disk states and different levels of logical address space. For the same level of address space, predicate implications can be done automatically and the predicates on both side of the implication can be "canceled out". What is left usually involves different levels of predicate implications and this needs to be finished by the developers.

We evaluate AsyncFSCQ and show that its I/O performance is better than FSCQ by an order of magnitude. We write a microbenchmark, which consists of 8192 writes to disk. Each time we write 64 bytes. The program uses a two-tier iteration. Each outer iteration will invoke disk sync, so the number of inner iterations is equivalent to the number of transactions batched. We alter the number of inner iterations and draw several conclusions. First, for the same number of batched transactions, AsycFSCQ outperforms FSCQ by an order of magnitude on average. Second, more transactions we batch, better performance we get. Specifically, as we double the number batched transactions, the execution time of the program decreases to half, which shows that fsync is the bottleneck in our program. And by carefully using fsync in our program, we can extract more performance. We also run AsyncFSCQ with real world software, such as GNU coreutil (rm, grep, etc.) and software development tools (git, make and so on). The experience of running such software shows that for software involving writes to disk, AsyncFSCQ is way faster than FSCQ.

Evaluation of performance is not enough. We also need to give proofs that AsyncFSCQ supports deferred durability correctly. To do that, we conduct a bug study on ext4 file system. By categorizing the bugs found, we manually check whether these bugs can happen in AsyncFSCQ and whether AsyncFSCQ prevents them. The results exhibits that AsyncFSCQ is complicated to have all the categories of the bugs and it is able to prevent all of them except concurrency bugs, as concurrency is not supported in AsyncFSCQ.

Formal verification can prove that out implementation meets the specification and in a way shift the possible bugs from implementation to specification. Whether our specification is correct also need to be evaluated. There are several ways to do that. The most straightforward way is to manually checking the specification. Since our specification only contains 2 lines of precondition,

2 lines of postcondition and 1 line of crash condition, it is easy to spot any possible errors in it than tens of hundreds lines of implementation. Also, we can compose theorems on the basis of our specifications. If we end up in proving these theorems, we can safely say that the specification has the properties shown in the theorems.

At last, I'd like to share a few words on development efforts. This project takes me into the world of formal verification. From nothing known about this area, I pick this project as my graduation project in January. The first few months have seen a great efforts spent in learning the basic of Coq programming and proving in Coq. In March, I start to dig into the FSCQ project and come up with the idea of supporting deferred durability. And another month is spent in following this project, how it designs the domain-specific language, how the thinking of functional programming is used in this project, how to write Haskell, how the CHL framework works and so on. And now in June, I successfully add a simple functionality in FSCQ. The process shows that using formal verification to compose systems is manageable. And once stepping into this world, there is more to learn and more I can do. For future work, we can implement more optimizations in AsyncFSCQ and add concurrency support for our file system.