文件系统优化的几种方法

摘要

现代存储系统因为存储访问和存储系统的改进而变得更加迅速，而与此同时，快速的存储系统和主机与存储系统的交互不畅导致的延迟和缓慢逐渐成为了新的主要矛盾来源。因此，针对交互不畅的问题，近年来有不少的研究专攻了这项问题，并且提出了较多的解决方法，归纳起来就是：内核级文件系统方法，用户级文件系统方法和固件级文件系统方法。本文将对较近取得突破的用户级文件系统方法和固件级文件系统方法做出较为具体的介绍，并且对于最新的将三种方法有机组合的跨层文件系统进行详细的介绍。并在文章的最后分析这几种方法对于性能的提升具体有多少。

介绍

现如今，存储访问行为本身的延迟已经被压低到了毫秒级别[1]，而应用程序也在设计上尽量避免造成 I/O 冲突导致并行性降低，与此同时，存储系统的访问策略也在时刻影响着 I/O 的并行性（例如系统调用开销，粗粒度并发控制和无法利用硬件级别的并发性）。针对这个问题，一些内核级，用户级和固件级的文件系统都从 CPU 并行性[2]，直接存储访问[3]或者存储固件[4]三个方向进行入手，设计了许多提高并行性的办法，这些方法都显著地提升了 I/O 的并行性能。然而，这些方法都是孤立设计的，对于超高速的存储硬件的利用并不使人满意。

内核级文件系统（kernel FS）作为一种最传统的设计方案，满足了文件系统的基本需求，例如完整性、一致性、持久性和安全性。它的结构如图 1 所示。作为最简单最传统的方案，它在不断改进之后仍然存在三个主要的限制：第一，应用程序必须通过操作系统的系统调用来执行 I/O，这使得每次调用都会产生 1-4μs 的延迟[4]，而且最近关于这个问题导致的安全漏洞[5]进一步增加了这里的延迟。第二，即便是最为先进的方法，在线程进行访问文件中不相交数据部分的操作的时候，也会强制执行不必要的序列化（比如 iNode 级别的读写锁）从而导致了高并发访问开销。第三，内核文件系统的设计未必能充分利用硬件级别的功能，例如计算、数千个 I/O 队列和固件级调度，最终影响 I/O 密集型应用程序的 I/O 延迟、吞吐量以及并发性。

基于这些问题，为了解决内核级文件系统的用户级文件系统（user FS）应运而生，它通过绕过操作系统进行直接存储访问的方式来规避上述三种瓶颈[4]。然而，这种方式在面对一些无法信任的用户的时候很难保证一些基本的文件系统功能。因此，最终得到了图 2 这样的结构。这些设计在近几年的研究当中取得了关键性的进步并且一度是当前最前沿的水准，这些设计有些是仅仅只支持数据平面绕过操作系统进行直接操作，换言之，也就是不能满足数据的共享。另外一些

设计则通过回避或者继承跨线程和进程的粗粒度与次优并发控制，甚至损失正确性的方式来实现直接访问。不过大多数用户级文件系统的设计并没有能够充分利用现代存储的硬件功能。

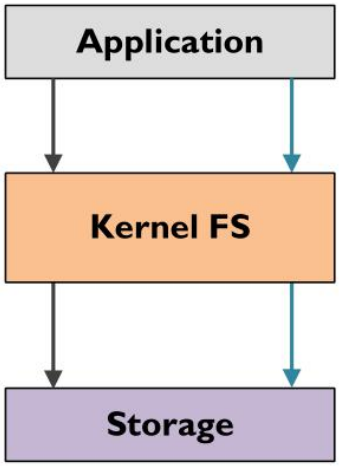


图 1：内核级文件系统结构示意

而另外一种不同于用户级文件系统的设计——固件级文件系统（firmware FS）也是一个非常具有代表性的解决方案，它的做法是将文件系统整个嵌入到设备固件当中，以便进行直接的访问[6]，在存储器当中的固件级文件系统负责满足基本的文件系统的各种要求。图 3 粗略展示了这种结构。虽然这个思路是利用存储系统本身计算能力的开创性贡献，但当前的设计并不能很好的利用主机的多核 CPU 的并行性质，导致了效率的降低；此外，这些设计还继承了以 iNode 为重心的请求队列、并发控制和调度设计，导致了 I/O 的可扩展性较差。

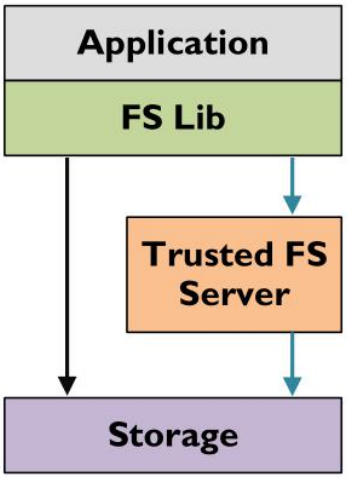


图 2：用户级文件系统结构示意

简而言之，当前的用户级别文件系统、内核级文件系统和固件级文件系统的设计重点在对自身的模型进行优化，缺乏横跨用户、内核以及固件三层的协同设

计，这种协同设计在不损害基本文件系统的功能的情况下对实现直接存储和扩展并发 I/O 性能有着巨大的帮助。

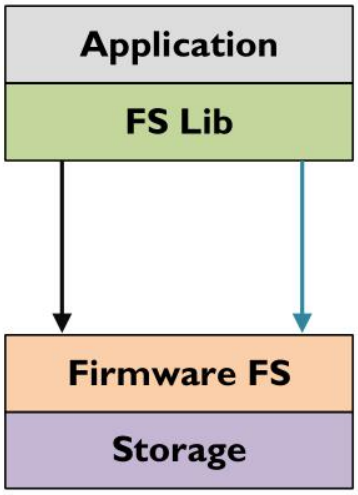


图 3：固件级文件系统结构示意

由是，跨层文件系统（CrossFS）的思路营运而生，这种系统的扩展性强，具有高并发访问吞吐量和较低的访问延迟。

跨层文件系统通过将文件系统分散在用户层、固件层和操作系统层的方法来达到取长补短，应用每一层好处的效果。固件组件（FirmFS）是文件系统的核心，它能够让应用程序在不损害基本文件系统的功能的情况下直接访问存储，固件组件利用存储硬件的 I/O 队列，运算能力和 I/O 调度能力来提高 I/O 性能。用户级库组件（LibFS）对可移植操作系统接口（POSIX）兼容，兵使用主机级 CPU（host CPU）处理并发控制和冲突解决。操作系统组件在固件组件和用户级库组件创建初始接口，并将软件级访问控制转换为硬件安全控制。

对于并发性控制的设计，仅仅只是将文件系统分散在三层是不足以实现 I/O 的可扩展性。I/O 的可扩展性要求重新访问文件系统并发控制，降低日志记录成本，并设计与并发控制相匹配的 I/O 调度。可以注意到，在大多数并发应用程序当中，文件描述符是是一个对访问的自然抽象，其中线程和进程使用独立的文件描述符来访问和更新共享的或者私有的文件的不同区域（比如 RocksDB 维护的 3.5K 开放文件描述符）。因此，在跨层文件系统当中，针对 I/O 的可扩展性问题，引入了基于文件描述符的并发控制，允许线程同时更新火访问非冲突的文件块。

接下来，文件描述符将被一一映射到专门划出的硬件 I/O 队列，来利用存储硬件的并行性和细粒度的并发控制机制。使用文件描述符发出的所有非冲突请求（即对不同块的请求）都会被添加到特定的文件描述符队列当中。另一方面，对于冲突的请求则是通过使用单个队列来进行排序的。这样的设计为设备 CPU（device-CPUs）和固件组件提供了在主机和设备 cpu 几乎没有同步的情况下并发

执行调度请求的机会。对于跨文件描述符冲突的解决和块的更新次序，跨文件系统才用的是单索引节点间隔树（per-iNode interval tree），间隔树读写信号量（interval tree rw lock）和全局时间戳进行并发控制，只不过，与当前的文件系统在请求完成之前都必须保持线程的 iNode 级锁不同，跨层文件系统只获取间隔树的读写锁，来对请求队列（fd-queue）进行请求排序。简而言之，跨层文件系统的并发设计将文件同步问题转化为了队列排序问题。

在这种思路的指引之下，设计出来的跨层文件系统在跨线程和进程共享数据的并发访问基准测试当中，相比于其他算法的吞吐量增加了大致 4.87 倍，而在没有数据共享的多线程的 file bench 和 macro benchmark 工作负载的吞吐量则提高了 3.58 倍，在现在市面上广泛运用的应用程序，例如 RocksDB 和 Redis 当中，则分别有 2.32 倍和 2.35 倍的吞吐量提升。

背景与相关工作

现代存储系统随着以相变存储器等非易失性存储（NVM）为代表的高速存储设备的出现而出现了革命性的改变，这些高速存储设备提供了远超硬盘的带宽（8-16GB/s）和相比于硬盘低了两个数量级的访问延迟（<20μs）[7]，这主要归功于高速存储设备普遍能够提供更快的存储硬件和访问接口（比如支持 PCIe support），支持在存储器当中进行运算（具有 4-8 个 CPU、4-8GB 的 DRAM、64K 的 I/O 队列）、基于电容器的故障保护[8]以及存储器件对编程行为的逐渐支持。而文件系统也在近年来发展到了可以基于高性能的现代存储设备进行开发，很多的工作都为这些开发做出了不可磨灭的贡献。在开发的过程当中，合理地利用现代存储设备的快速与可计算功能，成为了提高文件系统效率的关键，所以，在开发文件系统的过程当中，可扩展并发性和支持高效的数据共享成为了至关重要的标准。接下来，我们将讨论内核级、用户级、固件级的文件系统进行的一些设计和意义。

最传统的做法就是内核级文件系统（Kernel-FS），这种做法依然是模仿原先的操作系统对文件系统的管理——无论是控制还是数据都通过操作系统进行统一的调度，最终再访问存储系统。基于这种结构体系，为了应对全新的高速存储设备，则需要设计心得内核文件系统[7]。例如，F2FS 利用 SSD 的多个队列，摒弃采用日志结构设计。轻量 NVM 则将 FTL 固件代码移动到了主机并定制请求调度[7]，而对于 PMFS、DAX 和 NOVA 这样的文件系统则利用 NVM 的字节寻址能力进行加载和存储的操作。或者有另外的系统例如 ext4 印度读写信号量或者利用 NVMe 的 I/O 命令队列来进行直接的 I/O 操作。可以看到这些设计要么不能支持 POSIX 接口，要么不能够很好的进行块操作。

内核级文件系统的缺陷使得绕过操作系统，直接从用户级库访问存储硬件的用户级文件系统（User-FS）的研究有了明显的意义。不过，用户及文件系统需要

解决的首要问题就是用户级库文件是一个不安全的文件，如果让它能够直接访问并且能够修改元数据的话，那么极有可能因为用户程序的错误或者通过用户数据进行攻击而导致内核的数据被修改，进而使得整个文件系统损坏；除此之外，还需要解决原子性和崩溃一致性的问题。这里我们将介绍一下 ZoFS 方法[9]。

这种办法通过将存储具有相同权限的文件的 NVM 页面组成一个容器，这个容器在论文当中被称之为 coffer，每个 coffer 中的所有页面都有一个单独的权限。这些 coffer 当中存储的是一个包含元数据的根页面，并且这个 coffer 会把这个根页面只读映射到用户空间库当中，根目录下有很多文件，这些文件存储在同一个 coffer 当中，由于这些文件目录可以直接在用户空间库当中查阅（可读的查阅），所以其管理不需要内核的参与。

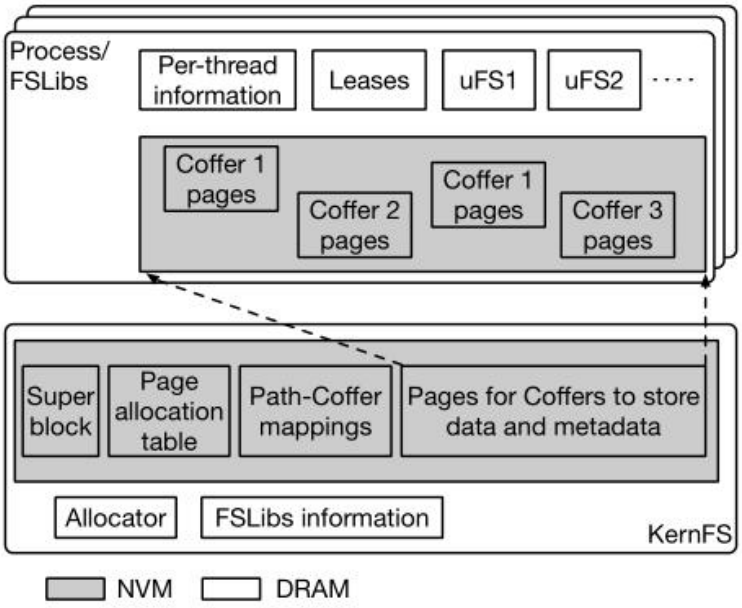


图 4：coffer 有两个部分，分别在内核模块（kernelFS）和用户模块（FSLib）内核模块在文件系统中维护页面分配信息和路径映射。FSLib 包含一组 FS 库（μFS）和一些辅助工具，coffer 内部的结构由相应的μFS 维持。

通过这些 coffer，可以构建一个 NVM 系统体系结构 Treasury，图 4 显示了这种架构。这种结构由两个部分组成：内核空间模块（kernel-FS）和用户空间库（FSLibs），这种结构将文件系统的保护和管理分离开来，让用户空间库通过 coffer 来对存储在 NVM 当中的数据进行控制，而对于存在于不同的 coffer 当中的数据的通信，则引入内核，由内核来进行管理以达到管理和保护的目的。Treasury 的核心部分则记录所有 coffer 的元数据，并负责检查所有来自用户空间文件系统库请求相对于 coffer 的权限，决定是否令其访问，而内核（kernel-FS）就仅仅只负责全局的空间管理，而不涉足 Treasury 内部具体的管理，而 Treasury 的管理则交由 FSLib 当中的文件库（称之为μFS）进行，这些文件库当中存储有 coffer 的管理数据与相关数据结构，并负责与应用程序进行交互，这些μFS 可以有多个，不同

的μFS 可以针对不同的应用背景依据不同的方法来组织 coffer 当中的文件数据与元数据。

可以看到 ZoFS 是通过使用虚拟内存来作为缓冲区来解决用户级文件系统的安全性问题，为控制和数据平面操作直接访问存储系统提供了基础，但也可以看到，这种架构需要极多的文件库进行维持，还需要不同的方法来构建 coffer，开销极大，而且并没有彻底解决控制平面直接访问的问题（依然需要进行内核操作）。

所以，进一步，有些研究采用了更加激进的研究办法，即固件文件系统（Firmware-FS）。这种文件系统的要义在于将整个文件系统或者其中的一个部分装载到设备固件当中，利用存储器当中的运算功能实现对主机 CPU 的减负，下面将介绍 INSIDER 设计办法[10]。

INSIDER 设计办法的核心在于将部分的文件系统的运算装载到固件当中，因此要选好在固件当中的计算组件，对于文件系统的性能将会有很大的提升，为此，在这个固件文件系统当中，对于计算组件的要求有三个：1，更高的程序化，让计算组件能够处理更多的工作；2，更高的并发性，能够提供更多的带宽，为并发的程序同时处理提供更好的平台；3，高能源利用率，更少的散热做更多的处理，避免过热情况。

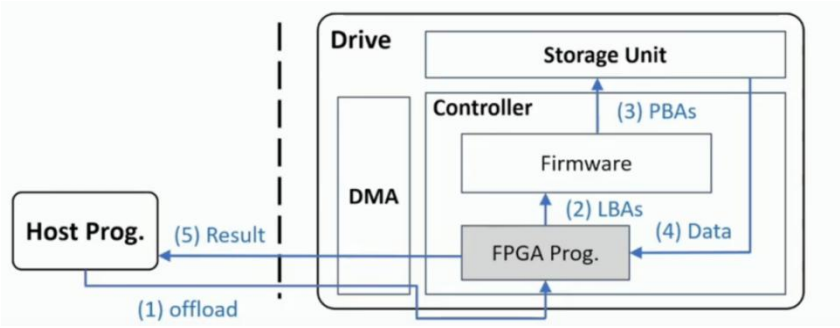
对于市面上常用的计算组件，图 5 展示了这些计算组件对于三个条件的满足情况。

		GPU	ARM	X86	ASIC	FPGA
Programmability		Good	Good	Good	No	Good
Parallelism	Data-Level	Good	Poor	Fair	Best	Good
	Pipeline-Level	No	No	No	Best	Good
Energy Efficiency		Fair	Fair	Poor	Best	Good

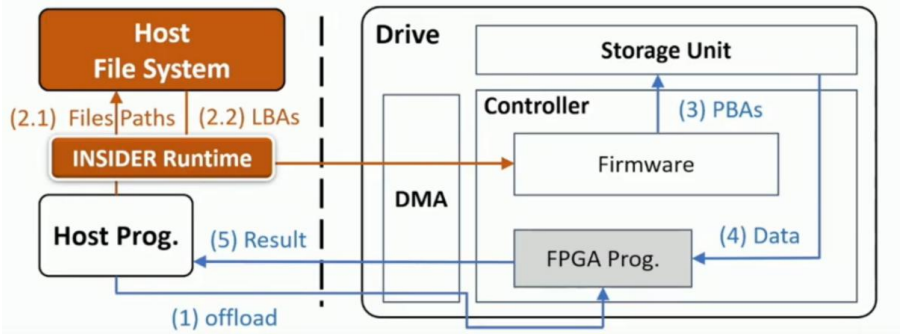
图 5：不同计算组件对三种条件的适应情况

对于这三种条件，可以看到只有 ASIC 与 FPGA 符合的比较好，ASIC 在除了程序化上都有远超 FPGA 的性能，但是因为 ASIC 在程序化上的拙劣表现，所以最终还是选择 FPGA 为佳。

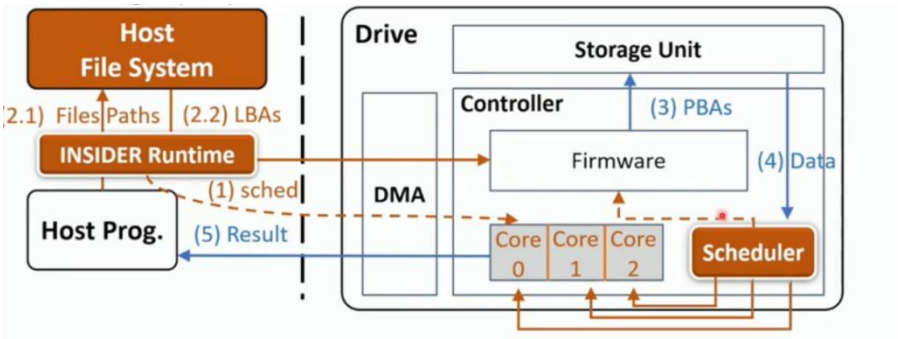
在选定计算组件以后，要着手开始搭建固件文件系统的框架，首先是将计算组件纳入驱动器控制器当中，此时，在驱动控制器当中，计算处理器负责将主进程输入的虚拟地址呈送给固件，并且将存储器当中查询到的数据进行计算，送回主程序。



可以看到在这个过程当中，计算处理器既要处理地址又要处理数据，可能带来数据冲突以及安全隐患，为了解决这个问题，可以将传输虚拟地址的操作交给主 CPU 来进行，也就得到了下面的情况：



但是这种情况下会出现新的问题，也就是主机的文件系统和计算结构之间的速度可能存在不匹配的问题，于是解决的方案就是在驱动器当中加入更多的计算处理器，并且为处理器分配一个组织结构，负责将要处理的数据交给空闲的 CPU。



在这种体系的构建下，INSIDER 的性能要比传统的内核级文件系统要好很多，然而，无论是 INSIDER 还是 ZoFS 都没有办法很好的解决强制序列化的问题，导致线程内部本身没有冲突访问的操作因为需要序列访问 iNode 锁导致不能并行。因此，这里可以有更大的改进，接下来，将介绍 CrossFS 设计[11]。

CrossFS 设计的思想

上述问题出现的主要原因在于仅仅只是改进文件系统架构的一个地方，没有做到用户层、设备固件层以及操作系统层的协同运作，而协同设计对在不损害基础文件属性的情况下实现直接存储访问和扩展并发 I/O 性能至关重要。

为了解决上述问题，提出了 CrossFS，这种文件系统是可以跨层直接访问的文件系统，这个文件系统提供了可扩展性，高并发访问吞吐量和较低的访问延迟，CrossFS 通过在用户层、设备固件层和操作系统层分解文件系统来实现这些目标，

从而利用每一层的好处。固件组件（FirmFS）是文件系统的核心，它的作用是能够在没有损害元数据的风险的情况下直接进行存储访问，具体而言，FirmFS 利用的是存储硬件的 I/O 队列、计算组件和 I/O 调度能力来提高 I/O 性能。用户级库组件（LibFS）提供的是 POSIX 兼容性，并利用主机 CPU 处理并发控制和冲突问题。操作系统组件则是在 LibFS 和 FirmFS 之间设置初始接口（例如初始化 I/O 队列），并且吧软件级别的访问转化为硬件安全控制。

下面，将具体地对 CrossFS 的结构进行讲解。

图 6 是对 CrossFS 结构的示意图。

可以看到，在跨层文件系统当中，用户层的请求首先进入主机 CPU 进行处理，主机 CPU 可以支持可移植操作系统接口（POSIX），然后由主机 CPU 将 I/O 命令加载到 I/O 队列当中去，并且在这个阶段，进行确保并行的处理；而中间的核心组件（Kernel Component）则是用来在固件当中初始化 I/O 队列的设施这可以有效地保证程序隔离与安全，因此，只有当初始化的时候，才需要启动这个核心组件，其余时候，在初始化完成之后，便只需要调用相应的 I/O 队列，实现了软件开销的减小。而最下方的固件文件系统（FirmFS）则是负责处理 I/O 队列当中的 I/O 请求，并且处理数据和元数据，支持数据的交互与再次对要求许可的检查。

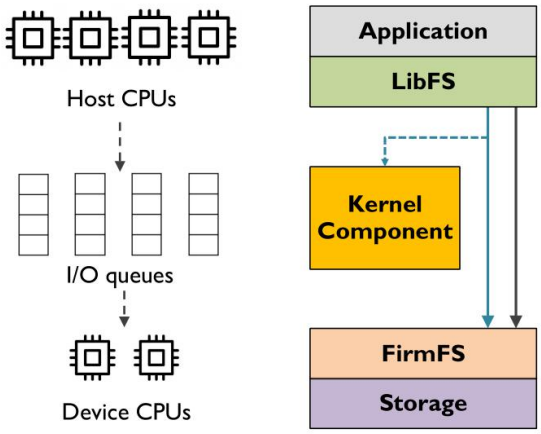


图 6：CrossFS 的结构示意图与相应的硬件支持

下面通过一个例子来说明这个系统是如何解决并行性问题的。

如图 7，当整个进程开始运行的时候，核心组件（Kernel Component）在 NVM 当中划出一个区域，并在这个区域当中创建 I/O 指令队列（I/O command-queue）与 DMA 访问的数据区域（DMA-able NVM region），也就是左边的数据区域（data buffer）。

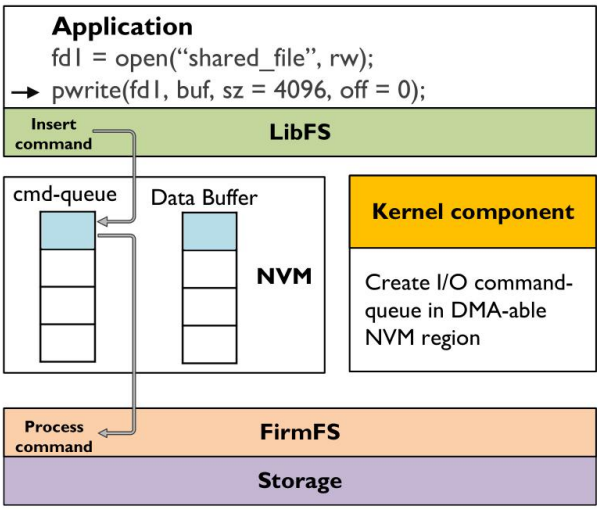


图 7：CrossFS 执行程序的具体过程

当用户程序开始执行指令的时候，用户文件库（LibFS）将相应的指令转化为可移植操作系统接口（POSIX）下的文件系统 I/O 系统调用，然后依据这个系统调用，将指令插入在 NVM 当中初始化的请求队列，其中包括请求的指令（插入 cmd-queue）和相应要键入的数据（插入 data buffer），然后固件系统当中的文件系统（FirmFS）将 cmd-queue 当中的指令装载到固件当中，形成可以执行的进程指令，直接在固件当中进行操作。

在固件文件系统将指令加载到固件当中之后，固件系统会对当前的指令进行检查，确保指令的安全性。

而在增加了一个 I/O 队列以后，可以大规模的减少因为强制序列化而导致的并行性降低的问题，这种问题可以通过对上述例子的进一步分析来进行解释：

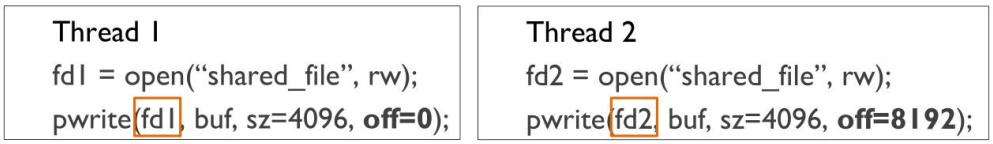


图 8：两个线程的例子

如图 8，这两个进程访问两个块，一个块的位置在 0，另一个块的位置在 8192，块的大小都是 4096，说明两个块是并不冲突的，但是由于强制序列化的问题，两个线程必须先后持有 iNode 锁，先后进入内核进行访问，这样的现状使得本来可以并行的两个线程被迫串行地执行程序，而且这种情况在强制序列化的线程执行当中是普遍存在的，也就是说，为了少数的冲突不得不牺牲大量的并行机会，如果能够解决这些少数的冲突，将大量的并行机会利用起来的话，那将会是一个巨大的提升。

而跨层文件算法则依据自身的性质解决了这个问题，图 9 展示的就是将图 8 的例子放入跨层文件系统当中进行操作。

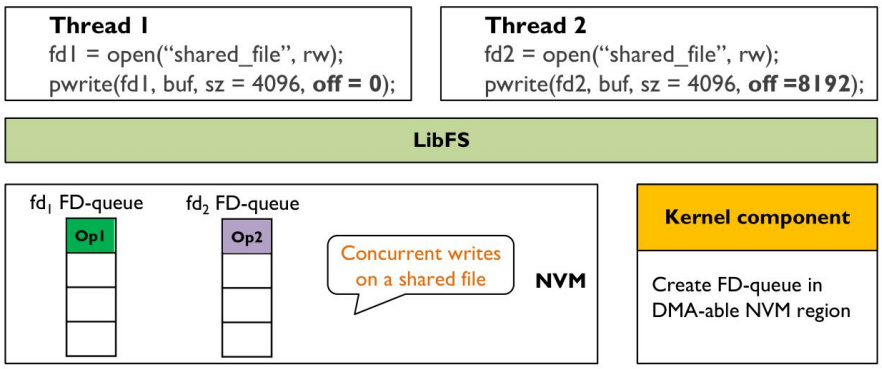


图 9：跨层文件系统处理多个线程

在跨层文件系统当中，I/O 队列承担了预处理两个线程指令的工作，两个线程的指令分别被读入预先由核心组件设置好的两个 I/O 队列当中，在没有冲突的情况之下（也就是例子当中的情况），在预处理读入 I/O 队列之后，可以将 I/O 队列当中所有的指令视为一个线程，直接送入固件当中进行相应的处理，如此便解决了因为强制序列化导致的并行不佳的情况，不过这个例子仅仅只讨论了线程访问的块不重合的情况，对于访问块重合的情况，目前这种机制又引来了冲突的问题，需要进行一下改进。

具体的改进方式是，在用户层文件系统当中，引入一个间隔树的数据结构，这个数据结构在叶节点上可以存储当前的块是否已经被之前的指令所访问，如果后继存在一个指令也要访问节点上已经有指令要访问的块的话，则启动冲突处理机制，这种机制下，先前的指令将会不被执行（因为在逻辑上，存储块中保存的数据取决于最后一个执行的指令），之后更新间隔树叶节点标识的指令为最新的指令，并将该指令送入队列的下一个空格当中。

因此，最终，可以看到跨层文件系统的具体结构变成了图 10 的模样。

在这个结构当中，当用户层在不断预处理指令的时候，节省掉了逻辑上原本不需要执行的指令，而且这种预处理仅仅只需要在队列中进行插入和删除的操作，相比于传统的 iNode 强制顺序化而言，也节省了一个写指令的时间，可以说在大量的操作之下，这种节省将会给性能带来非常大的提升。

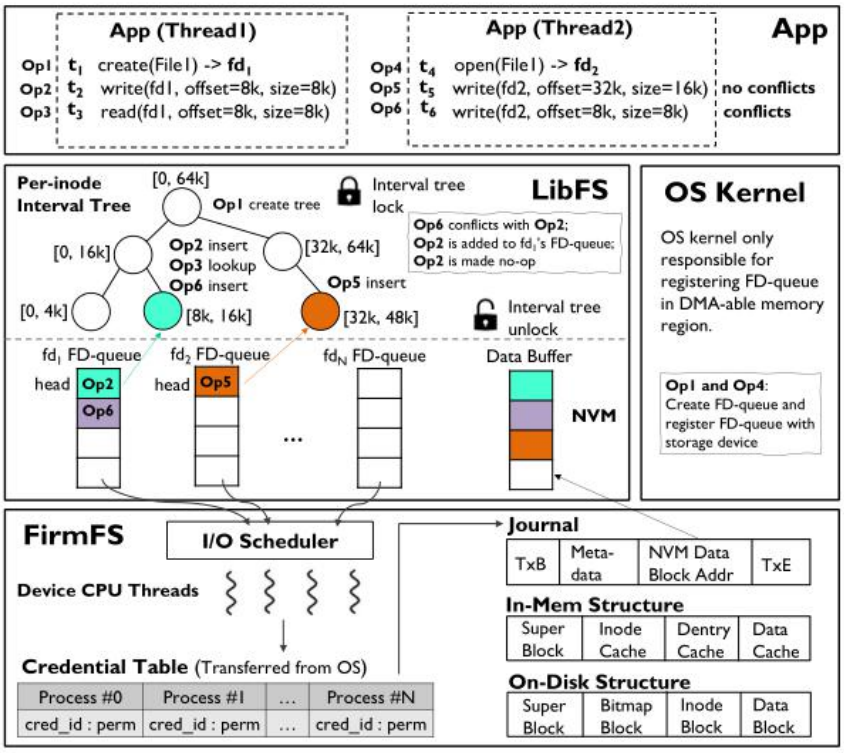


图 10：跨层文件系统的执行流程

性能测试

在进行各种结构改良之后，用户型文件系统和固件型文件系统的性能都优于之前传统的内核型文件系统，而跨层文件系统则在并行上展现出了比用户型文件系统与固件型文件系统更加优异的特质。

对于 ZoFS 而言，其多线程性能明显优于传统的各种文件系统，如图 11。

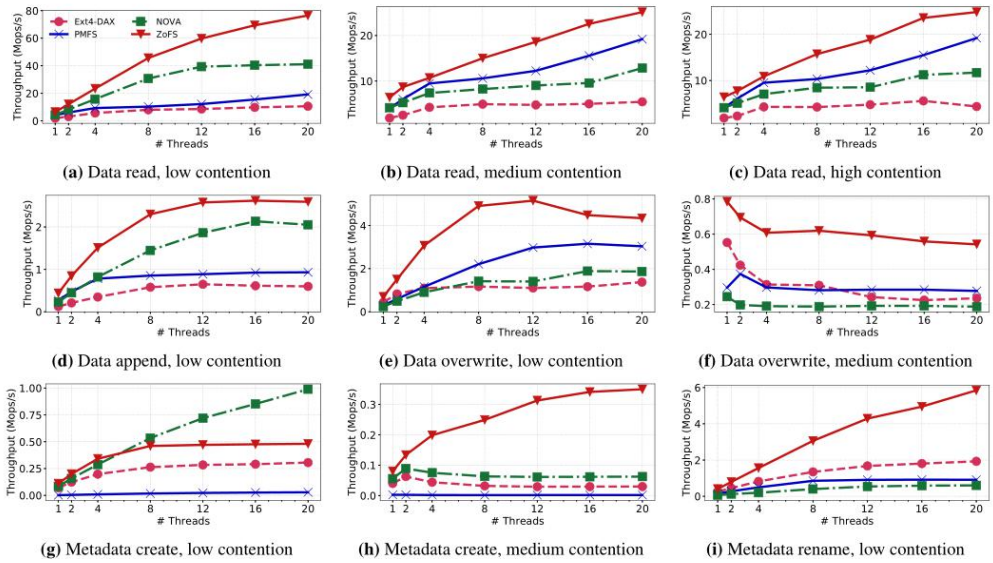


图 11：ZoFS 与其他内核级文件系统的性能比较

而对于 INSIDER 而言，和内核级文件系统相比，在实际的应用当中，性能提升平均而言高达 12 倍，具体的性能提升可以参看图 12。

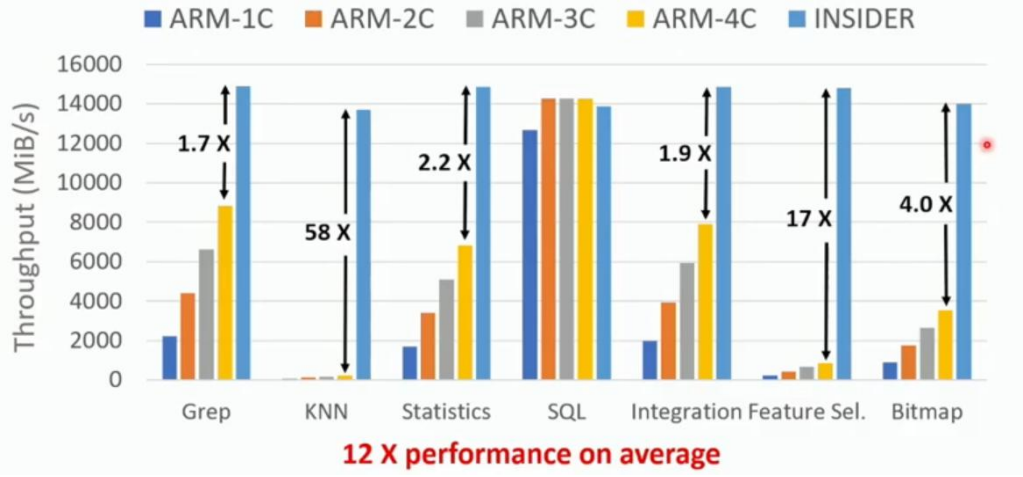


图 12：在不同的应用程序下 INSIDER 对吞吐量的提升

而对于 CrossFS 而言，性能的提升相对于前两者而言更加显著，而且在多线程的读和写测试当中，其他文件系统的吞吐量在线程数量增加的时候吞吐量并未有显著改善甚至还出现了下降的情况，唯有 CrossFS 保证了基本线性的增长，具体的情况可以参照图 13。

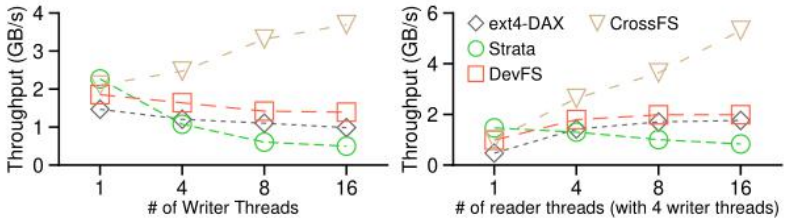
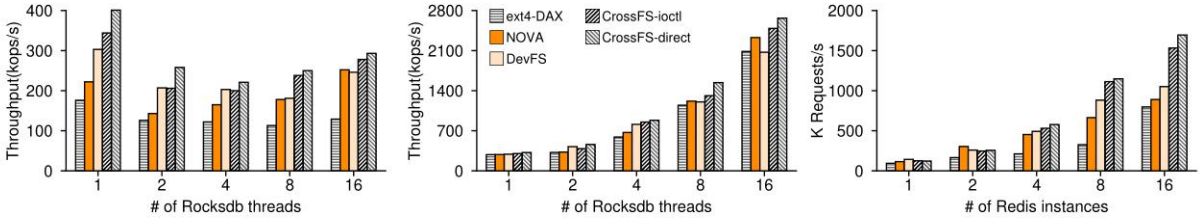


图 13：读写线程并行对各种文件系统的影响

而在实际程序测试当中，CrossFS 也对于其他的文件系统有较为显著的提升，虽然因为实际程序的复杂性线性增长的特质并不能体现，具体情况参照图 14。



(a) Rocksdb random writes (b) Rocksdb random reads (c) Redis SET benchmark

图 14：CrossFS 的实际测试

结论

文件系统在大批量的数据处理当中是非常重要的一个环节，优化文件处理系统会带来极好的性能提升，在优化方案当中，对于用户端和并行性的两个方向是最容易解决的方向，也是首先突破传统结构的方向，但是协同设计可以进一步的挖掘文件系统的潜力，不失为一个非常巧妙的思路。

参考文献

[1] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports,Doug Woos, Arvind Krishnamurthy, Thomas

Anderson,and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI’14, Broomfield, CO, 2014.
[2] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS ’19, pages 427–439, New York, NY, USA, 2019. Association for Computing Machinery.
[3] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (*SOSP '19*). Association for Computing Machinery, New York, NY, USA, 494–508. DOI:https://doi.org/10.1145/3341301.3359631
[4] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File SystemwithDevFS. InProceedingsofthe16thUSENIX Conference on File and Storage Technologies, FAST’18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.
[5] P. Kocher *et al*., "Spectre Attacks: Exploiting Speculative Execution," *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA, 2019, pp. 1-19, doi: 10.1109/SP.2019.00002.
[6] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File SystemwithDevFS. InProceedingsofthe16thUSENIX Conference on File and Storage Technologies, FAST’18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.
[7] Matias Bjørling, Javier González, and Philippe Bonnet. LightNVM: The Linux Open-channel SSD Subsystem. In Proceedings of the 15th Usenix Conference on File and Storage Technologies, FAST’17, Santa clara, CA, USA, 2017.
[8] Y. Son, J. Choi, J. Jeon, C. Min, S. Kim, H. Y. Yeom, and H. Han. SSD-Assisted Backup and Recovery for Database Systems. In 2017 IEEE 33rd International Conference on Data Engineering (ICDE), pages 285– 296, April 2017.
[9] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (*SOSP '19*). Association for Computing Machinery, New York, NY, USA, 478–493. DOI:https://doi.org/10.1145/3341301.3359637
[10] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In 2019 USENIX Annual Tech- nical Conference (USENIX ATC 19), pages 379–394, Renton, WA, July 2019. USENIX Association.
[11] Yujie Ren and Changwoo Min and Sudarsun Kannan.CrossFS: A Cross-layered Direct-Access File System.In 2020 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pages 137--154, November 2020. USENIX Association