班级 030731 班

学号 03073013

本科毕业设计（论文）

外文资料翻译

毕业设计题目 基于 Linux GUI 的视频播放器实现

外文资料题目 A 64-bit, Shared Disk File System for Linux

学　　院 计算机学院

专　　业 教育技术学

学 生 姓 名 朱春来

指导老师姓名 姚勇

目录

Linux 中的 64 位共享磁盘文件系统

Kenneth W. Preslan, Andrew P. Barry, Jonathan E. Brassow, Grant M. Erickson, Erling Nygaard, Christopher J. Sabol, Steven R. Soltis, David C. Teigland, and Matthew T. O'Keefe Parallel Computer Systems Laboratory Department of Electrical and Computer Engineering University of Minnesota

摘要

今天的计算机系统，速度和响应性常常由网络和存储器子系统性能决定。像纤维管道和高速以太网这样更快更具有适配性的网络化接口为更高性能设备的实行提供了关卡，但新的问题是要求机器和网络激活存储设备进行交互。

我们已经有了一个叫 GFS（全局文件系统）Linux 文件系统允许多台 Linux 机器进行存取和共享磁盘，并且通过纤维管道或者 SCSI 存储网络备份。我们计划通过 IP 运输包格式化的 SCSI 命令来扩展 GFS，使任何激活了 GFS 的 Linux 机器可以存取共享网络设备。GFS 在作为本地文件系统、传统的通过 IP 运行的网络文件系统和通过像纤维管道这样的存储网络运行的高性能集群文件系统都工作良好。GFS 设备共享为 Linux 提供了一把打开集群技术的钥匙，帮助 Linux 提供可用性、规模可伸缩性和平衡机器资源。

我们的目标是建立一个可伸缩（在客户端和设备的数量，容量，连通性以及带宽），单一服务文件系统并使之综合为建立在 IP 基础上的网络附加存储（NAS）和建立在纤维通道上的存储区域网络（SAN）。我们把这个结构叫存储区域以太网络（SAINT）。它提高了 SAN 集群的速度和伸缩性能，以及客户端伸缩性能和 NAS 机器的网络操作能力。

我们通过 Linux 端口显示 GFS 结构是可以在各种平台上移植的，并且现在正通过 NetBSD 的一个端口工作着。GFS 源码是自由通用的开源软件（GPL），网址是 http://gfs.lcse.umn.edu.

第一章 简介

传统本地文件系统通过创建在磁盘块与一连串文件、文件名、目录名之间的映射表来提供一个持续名字空间。这些文件系统把设备看成本地的，设备不能共享，因此没有必要这这些文件系统中增加共享功能。相反，它的重点是存储高速缓存和通过要求每个文件系统操作尽可能少的磁盘存取数量来聚集文件系统操作，从而提高性能[1],[2]。

新的网络化技术允许多个机器共享相同的存储设备。允许机器在这些共享设备上同时加载和存取文件的文件系统叫做共享文件系统[3],[4]。和传统描述的文件系统比，共享文件系统提供了更少的服务选择，因为它的重点是数据共享。插图 1 显示，机器通过存储局域网络直接把设备作为自己的附属。[5],[6],[7],[8]

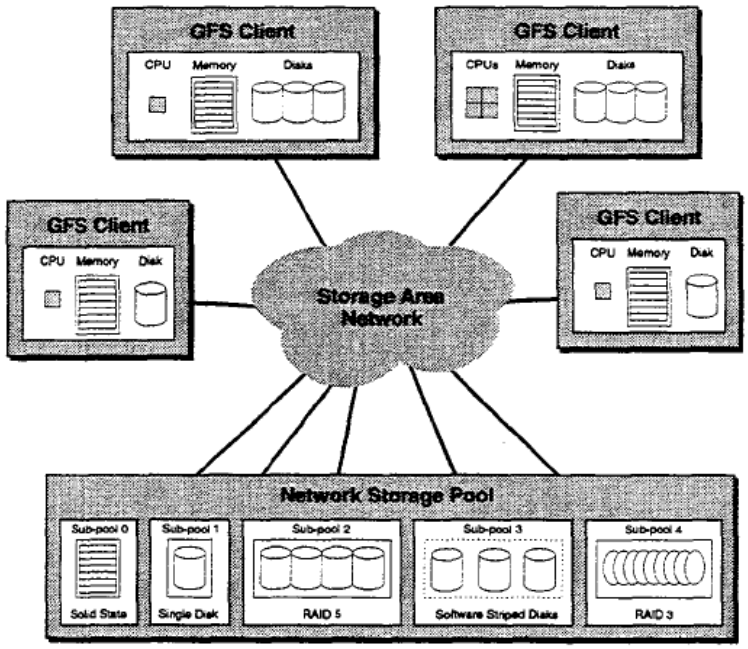

Figure 1: A Storage Area Network

共享文件系统通过建立在存储设备和机器间的网络上，从而具有多个优点：

1.伸缩性可以增加，因为当单个客户端失败时，其它客户端也许想继续工作，在共享文件系统中它可以访问那个操作失败的客户端文件。

2.在多个客户端共享磁盘时，简单的通过客户端功能对任何磁盘的资料组的任何端口进

行快速存取，从而平衡加载的最大工作量。

3.共用存储设备为同意的磁盘符号，系统对所有机器进行相同的访问变得可能。

4.在像设计为中央服务的网络文件系统 NTF 一样，天生就可以在容量、连接性和带宽方面进行无限制伸缩。

在 1995 年的夏天我们发展了我们自己的共享文件系统，叫做 GFS-1（全局文件系统，版本号为 1）。那个时候，我们首先对在硅图行（SGI）硬件上开拓纤维通道来传送大量科技资料组[9]很感兴趣。在一个更快的纤维通道网络上允许机器共享设备，这要求我们为 IRIX（SGI 的 System V Unix 变体）自己编写一个共享文件系统，在里面我们开始努力给出一个原型描述[10]。这次实施使用了与 SCSI 相似磁盘使 SCSI 保留和释放命令同步。保留和释放锁住了整个设备，使得它不可能支持同时对一个磁盘进行文件元数据访问。很明显，这是无法接受的。

通过给 SCSI 开发一个更精确的锁命令，从第二个原型 GFS-2 中删除了瓶颈。我们在 1998 年的论文[4]和联合论题[11]中对这个原型进行了描述，并且给出了四个客户端在纤维通道交换机和 RAID-3 磁盘阵列的使用性能结果。由于频繁的锁争持而没有达到三个客户端通过的比例。甚至，更大的文件需要更高的性能和伸缩性，因为文件元数据和文件数据都是缓存在客户端的。

1998 年的春天，我们开始移植代码到开发源码的 Linux 操作系统上。我们这样做有很多原因，但最主要的是 IRIX 是封装代码的，使得我们很难完全的是 GFS 成为内核的一部分。同时，Linux 已经得到 64 位和 SMP 支持，并且在数字化设备企业（DEC）最初平台上比我们的 IRIX 桌面化机器更快也更便宜。

而且，我们已经不再把有限的精力放在大型数据程序上而是扩大到设计一个常规意义上的由简单桌面化机器到大型激活不同网络共享设备的文件系统上。因为我们有核心源码我们可以支持元数据和文件数据缓存，但这要求对锁规格细节进行改变[12]。这个在 Linux 中的 GFS 端口包含了 GFs-2 的重要变化，因此我们把它当作 GFS-3。接下来的部分我们将描述 GFS-3（在这篇论文中我们把它当作 GFS），当前的实行包括我们的 Linux 端口的细节，新的伸缩性目录和元数据数据结构，初步性能报告和将来的工作。

第二章 GFS 背景

对 GFS-2 的详细描述看附录[4]，对 GFS-1 的描述看附录[10]。在这部分我们对这个文件系统的关键特性进行总结。

2.1 设备锁

设备锁是 GFS 用来促进文件系统元数据相互排斥的技术。设备锁也在元数据被多个客户端缓存时帮助元数据保持一致性。这些锁应用在存储设备（比如磁盘），并用 SCSI 设备锁命令 Dlock 来访问。Dlock 命令独立于所有其他的 SCSI 命令，所以设备支持这些锁对本来资源是否被锁没有感知。这个文件系统提供一个文件到设备锁的映射表。

在最初的设计中，每个设备锁从根本上说是测试和设置锁。每一个 GFS 客户端获得锁，读取数据，修改数据，写回数据，释放锁。这样允许文件系统对元数据进行完全操作，对于对这些元数据的其他操作询问得到的回答则是“原子的“。

每一个设备锁都有一个“版本号”与之关联。当一个客户端要对一部分元数据进行 读取-修改-写回 操作时，它获得锁，然后进行 读取-修改-写回 操作，并且通过“unlock“增加 1 行为来释放锁。当一个客户端只是想读取元数据，它获得锁，读取数据并且通过 unlock 行为来释放锁。如果所有的客户端都遵循这个方案，一致性可以通过比较锁行为返回的版本号和已经占有的锁的版本号来检查。如果版本号相同，则没有客户端可以修改被锁保护的数据，这样可以保证有效性。版本号也用于锁描述管理器 Vaxcluster[6]的高速缓存中。

2.2 网络存储池

网络存储池（NSP）设备容量支持抽象化的 GFS 客户端单个统一存储地址空间。NSP 应用在 SCSI 设备和纤维通道设备的最顶层-设备传动层。这样设备从文件系统的逻辑地址空间转换到各个设备的地址空间。对 NSP 子区划分为包含相同设备类型的组合，这些类型继承了底层设备和网络连接点的物理属性。

2.3 资源组

GFS 根据网络存储池各方面来分类元数据，而不是把所有的都放入一个单个类别。多个资源组被用来区分成包括数据、目录节点位图和数据块等元数据的单独的组，用以增加客户端并发性和文件系统伸缩性，避免瓶颈和平衡典型元数据搜索操作的平均值。一个设备也许存在一个或多个资源组，或者一个资源组包括多个设备。

资源组类似于 Linux 的 Ext2 文件系统中的组块。和资源组一样，组块通过允许单个计算机的多个进程分配和释放数据块来保证并发性和伸缩性；GFS 资源组允许多个客户端同时进行这样的操作。

GFS 包含一个单独的超级块，它包含元数据的摘要而不是通过资源组来分类（超级块也许可以提供性能和易于恢复）。这些元数据包括文件系统加载的客户端书目、计算每个客户端独特标识符的点阵图、设备被加载到文件系统的那个地方和文件系统块的大小。超级块还包含了描述本地每个资源组和其他设置信息的资源组静态索引。

2.4 目录节点元素

一个 GFS 目录节点就是进入文件系统块的入口，因为通过多个客户端引起有含义的争持而占有元数据来共享单个块。为了统计内部分裂结果，我们设计了目录节点元素，它允许把文件系统信息和真实数据都包含到文件系统目录节点块中。如果文件大小超过了目录节点中储存到数据块或间接数据块的指针数组的数据部分。否则就是文件系统目录节点信息被储存而占有了文件系统数据后，文件系统块剩下的部分。对于客户端访问由单个块填充的文件的请求，路径名中的每个目录需要有读取目录文件的特征对于目录查找特别有用。

考虑到一个文件系统块大小是 4KB，而假定目录节点头信息需要 128 个字节。没有填充的情况下，1 字节大小的文件需要总共 8KB 和至少 2 次磁盘运输来读取目录节点和数据块。在填充的情况下，1 字节大小文件只需要 4KB 和 1 次读取请求。在 GFS 清空目录节点前，文件大小可以达到 4KB-128 字节，即 3968 个字节。

GFS 根据每个目录节点在磁盘上的地址来分配目录节点号。目录包含文件名及其节点号。一旦 GFS 查找操作匹配一个文件名，GFS 根据有关联的节点号来定位目录节点。GFS 通过分配到节点号的磁盘地址来同步被分配的节点和空闲的块组。

2.5 直接文件结构

GFS 像插图 2 中显示的那样使用了直接的指针树结构。目录节点的每个指针在元数据树中都指向相同的高度（所有的指针都是直接指针或都是间接指针，或者都是双链接的）。元数据树的高度大小正好可以占有文件。

大部分常规 UFS 文件系统目录节点都有一个固定的直接指针号、一个间接指针、一个双链接指针和一个三个间接指针的组合。这意味着一个 UFS 文件大小被限制了。然而，对于小型文件，UFS 目录节点指针树需要较少的间接迂回。像 SGI 的 EFS 文件系统，或者 SGI 的 XFS 文件系统的 B 树路径，提供了基本长度配置的其它选择。GFS 元数据的当前结构是一个应用选择，并且这些选择在将来的研究中值得探究。

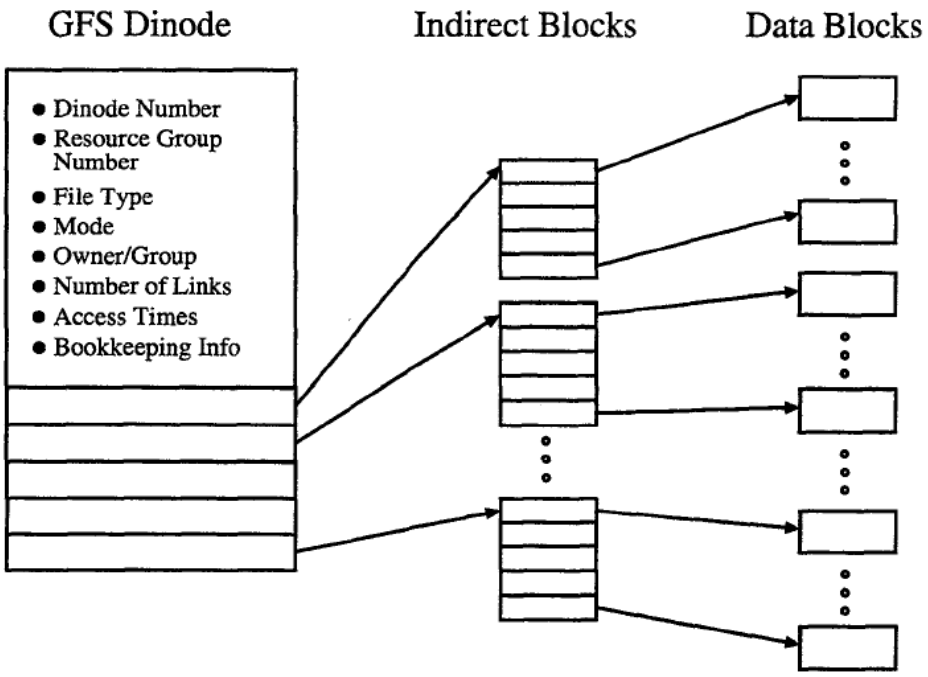

Figure 2: A GFS dinode. All pointers in the dinode have the same height in the metadata tree.

第三章 Linux 中的 GFS

GFS 最初工作在 SGI 的 IRIX 操作系统上。IRIX 是最佳的大型数据开发环境，而且它为 GFS 的开发提供了一系列工具。IRIX 缺乏的两样是核心接口文件和简单通用的核心代码。

为了使文件系统模块和核心的其它部分交互，文件系统的设计者需要理解到核心其他部分的接口。通过阅读接口的文档或者实施接口的源代码中任何一个方法都很容易完成（有人认为源代码是唯一正确的文档）。没有一个功能是 IRIX 具有的。

像 Linux 这样的开发源码操作系统是开发新的核心代码的好主意。源代码是完全自由的。所有的核心接口都可以通过小型测试和思考来掌握。因为源代码是完全自由的，文档就可以通过第三方来编写[13],[14]。无论如何，我们认为在将来 Linux 会克服处理大型数据方面的不足。GFS 主要是在 Linux 上开发。

3.1 IRIX 与 Linux

IRIX 和 Linux 的虚拟文件系统（VFS）层次之间有很大的区别。IRIX 使用标准的 SVR4 VFS/Vnode 接口，而 Linux 使用自己的途径。

3.1.1 VFS 缓存

不管是为 VFS 层次提供一套系统调用还是类似操作都需要文件系统进行特别编码。当然方法不同。SVR4 VFS 层次从一开始就准备支持网络文件系统[15]。相反，Linux 文件系统更多的是面向优化本地文件系统。

SVR4 VFS 层次与文件系统特殊编码之间的界限是很明显的。每当 VFS 层需要从文件系统特殊层获得信息时，它使用一个与文件系统中这个层信息有关的函数调用。它对前面的请求几乎不记录信息。

这对于网络文件系统是非常有用的。一台机器可以改变文件系统的数据，而不必担心其它机器的 VFS 层会缓存这些数据。VFS 层总是从文件系统特殊层获得信息。文件系统特殊层能一直提供这些数据的元数据。

在另一方面，Linux 的 VFS 层可以获得很多文件的访问信息。包括文件大小、权限和链接数之类的复制信息。对于本地文件系统，这些功能是很棒。所有的磁盘访问存取都是通过

VFS 层，所以它可以很好的缓存使用的数据。本地文件系统非常块，因为 VFS 层通过调用相应的函数和等待文件系统特殊层定位和重新编码请求信息，这样可避免重复载入数据。它只读取自己备份的数据。本地文件系统也比 SVR4 类似功能更简单。Linux 的 VFS 层自动进行权限检查。本地文件系统的设计者不必关心 UNIX 怎么管理权限的具体细节。

然而，这使得比在 Linux 中设计和实施网络文件系统更困难。网络文件系统尤其是共享磁盘文件系统不控制缓存，会在机器之间产生矛盾。Linux 的 VFS 层提供了文件系统特殊代码的调用来处理陈旧的数据，但这很笨拙也没有 SVR4 VFS 高效。在对特殊代码调用前，有很多地方 Linux VFS 要对文件元数据（比如权限）进行检查。很多时候，为了避免忽略条件，文件系统特别代码进行了重复的检查。SVR4 VFS 不会在第一个地方进行检查，所以只进行一次（在文件系统特殊层）。

Linux 是一个开发源码软件，并且开发组对于新主意也持开发态度。GFS 组的目标之一就是对 VFS 层进行修改，以便使所有常规文件系统尤其是 GFS 提供更高效。

3.1.2 目录输入输出

IRIX 有一个叫目录输入数的东西。它允许文件系统直接在磁盘和用户缓冲区进行数据复制。这样不用内存复制，就像普通输入输出缓冲区从磁盘读取数据到高速缓冲区，然后复制到用户缓冲区。目录输入输出对于大型文件存取提供了更高的速度。

这个时候，Linux 还没有和目录输入输出同样的东西。所有的磁盘输入输出都要通过高速缓存区。Stephen Tweedie 为 Linux 写了一个实施目录输入输出的补丁[16]，但直到版本 2.3 才被正式写入内核。

3.2 支持纤维通道的 Linux

目前，纤维通道（FC）设备可以在 Linux 中使用[17]。它是在 New Hamphshire 大学(UNH）关于 Qlogic ISP2100 的 Inteeerability 实验室完成的 [18]。设备被综合到 Linux 设备组织中，所有在文件系统中出现的 FC 设备都被看作是标准的 SCSI（sd）设备。QLA2100 预设的防火墙适配器目前只支持 FC 循环。更新防火墙的构造也许可以获得支持。目前为止，我们还没有测试防火墙的结构支持。对设备的新工作就是获得结构支持（对纤维通道技术包括循环结构需要很好的描述，可以从 Benner 的书中获得[17]）。

第四章 文件系统改进

对文件系统和元数据结构的许多改进在参考文献[4]和[10]有详细描述。我们确信这些改进会显著提高 GFS 的伸缩性。

4.1 目录和可延长散列法

传统文件系统处理不好的一点就是大型目录。大部分早期和部分中期文件系统把目录存储为没有排序的线性目录入口列表。这对于小型目录还是令人满意的，但对大型目录就太慢了。平均下来，文件系统要找到一个入口要搜索一般的目录。不但耗费了 CPU 时间，还引起磁盘的过度输入输出。大型目录会占有数兆的磁盘空间。

GFS 用可延长散列法[19],[20]作为目录结构。可延长散列法（Ex-Hash）提供一种存储目录数据的方法使得对于寻找详细的入口都非常的迅速。大量的搜索操作不太需要了。

比如，用 4096 位的块和 280 位的目录入口，一个 GFS Ex-Hash 目录可以记录 1700 个文件并在读取一个块就可以找到入口。一个 Ex-Hash 目录可以包含 910000 个文件并读取 2 个块可以寻找到任一个文件。相比之下，使用未排序线性表平均要成 62000 个块中搜索一半才可以找到一个目录入口。4.1.3 节讲述我们是怎么统计这些数据的。

4.1.1 Ex-Hash 工作机制

Ex-Hash 基础是每个文件名进行多位散列。一个哈希位的子集用来作为索引指向一个哈希表。哈希表的指针指向的“叶块“包含目录条目入口本身.Ex-Hash 目录中的搜索一个特定的项使用以下步骤：计算文件名的 32 位散列，从哈希左侧 X 未知位，查找叶块磁盘在哈希表的地址，读叶块，搜索目录项入口的叶块。

这个散列计划的诀窍是，哈希表的大小可以随着目录的增加而增加。哈希表的大小始终是 2 的幂次方。这个 2 的幂次方决定哈希表多少位被用作索引。当哈希表变得太小，无法容纳的目录时，哈希表的大小需要增加一倍，并且有一位的哈希已经被使用。这使得添加大型目录项时，不用对叶块连接表进行重新排序。

与给每个哈希表入口分配一个叶块不同的是，一个叶块可以指向多个哈希表项。这使得哈希表指针可以和只有少量目录项的哈希指针分享叶块。这提高了 Ex-Hash 的内存使用效率。

每个叶块总有 2 的幂次方个哈希表指针指向它。当目录项被添加到一个“满“的叶块时，叶块分割为两个独立的叶块。一半的哈希表指针指向原来的叶块，一半指向新叶块。目录项分布在两个叶块中，这样指针可以指向正确的哈希表元素。当叶块需要被分割而只是一个哈希表指针指向它时，该哈希表的大小增加了一倍。这样就有两个指针，叶块可以正常分割。

图 3-5 是 Ex-Hash 目录表的一个例子。它显示了一个规模大小是 4 的小型哈希表。每个文件的前两个哈希位被用来在表中查找。在这个例子中，每一片叶子块拥有三个目录条目。

GFS 的哈希表数据存储为目录中普通文件录，并且通过标准 vnode 输入输出例程存取。这使得哈希表可以持续增长却仍然对任何特定的指针存取保持最佳访问时间。叶块经常通过目录分配程序来存储到普通文件数据块外部。

GFS 有两种目录操作模式。当目录数量足够小，适合在通常的目录节点部分保存为元数据的指针，它们就像小型普通文件填满目录节点一样存储在这里。如果目录已经打开，那么目录节点已经存储在内存中，不用读取任何数据来查找目录。

4.1.2 可扩展哈希表

当目录转换为 Ex-Hash 格式后无法存储时，哈希表大小开始变成文件系统块的一半，然后存放到目录节点。要找到任何一个入口，需要从填满的哈希表中查找叶块地址并读取叶块数据。由于目录节点（和哈希表）已经在内存中，任何目录项都可以通过读取一个块信息寻找到。

当哈希表翻倍时，它不能继续增加数据，并将相关文件资料存储在目录中。任何条目可以在读取一个哈希表块数据，加上目录节点和哈希表之间的间接块，再读取叶块数据后找到。

这可能是一个防止哈希表增长太快的好办法。由于哈希是 32 位，通过一个叶块指针（8 字节），哈希表可能占用 2 的 32 次方个字节。一个具有相同散列的文件集添加到同一个目录是可能的（但既不可能）。这将导致哈希表很快增长到它的最大极限。对于一个目录来说，32 千兆字节太大了（至少现在是这样），GFS 有一个编译时常量，可以防止哈希表大小超过一定规模（默认情况下是 16 兆字节）。如果目录需大小需要超过这个值，叶块将在连接表中被拴在一块。这会增加了目录查找的平均访问时间，但增长非常缓慢，目录的大小每增加一倍就要增加一次磁盘访问。 （在这一点上，目录拥有 2900 万个入口，因此倍增不是很可能）。

添加和删除目录项消耗目录搜索一样的时间。因为每个块都要读取，因此读取整个目录

的内容就和读取线性目录内容一样慢。其它任何操作都要比用线性方式存储目录更快。

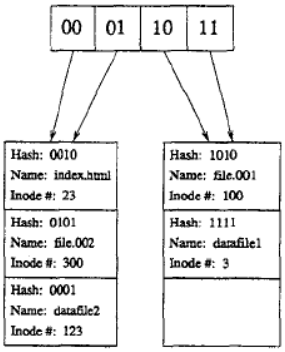

Figure 3: An ExHash directory: The hash table has a size of four and there are two leaf blocks.

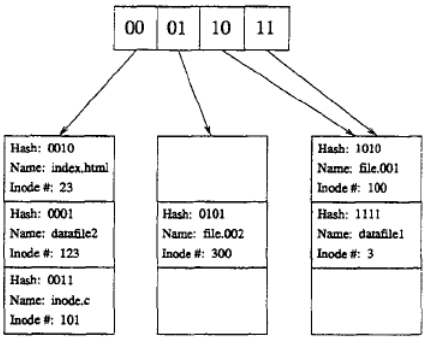

Figure 4: The directory from Figure 3 after the file "inode.c" was added. The addition forced the leftmost leaf block to be split.

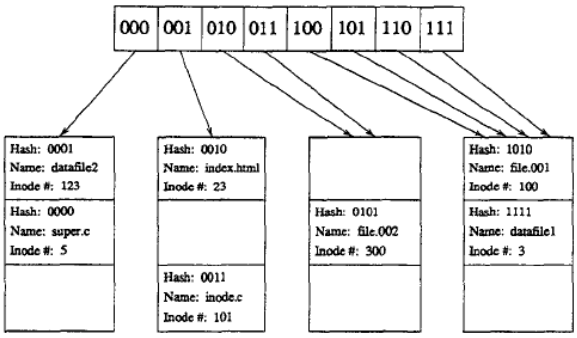

Figure 5: The directory from Figure 4 after the file "super.c" was added. The addition forced the hash table to be doubled and the leftmost leaf block to be split.

4.1.3 访问时间

假设一个块是4096字节且一个目录项是280字节，一个Ex-Hash目录可以容纳1700个文件并读取一个块信息就可以找到任何一个文件。在这情况下，哈希表是填满了的。一个填满了的哈希表至少有一半指针能匹配FS块。指针是8个字节，所以一个FS块可以容纳512个指针。这样哈希表有256项，而每个叶块有32位的头信息，所以在叶块中可以匹配的目录项书目是 （4096-32）/280=14.5。假设一个Ex-Hase目录在翻倍之前只有一半填满（一个超过合理的假设），它可以容纳1/2×14 ×256 =1792项。目录节点（即哈希表项）打开后就一直存在内存中，所以要做的就是读取适当块的信息，即“读“。

同理，一个块大小是4096字节的Ex-Hash目录，在需要保存任何间接块时，可以拥有131072个哈希表项，它进行两次块访问可以找出任何条目：1/2× 14 ×131072=917504

4.1.4 B-树比较

与B -树进行比较[21]是有道理的。B-树是一种常用的组织目录项的树模式方法。d是树的序号，每个块都指向2d+1号块。

树安排有这样一种方式，它从根开始一直到叶子进行目录入口的搜索，该条目会在搜索路径中被找到。搜寻时间（取决于块的数量）在0和O（logd（n）之间。搜寻时间更可能接近上限，因为有更多的条目是接近叶块的。

作为比较，假设一个GFS目录块大小4096位且目录项大小280位。从拥有469万个项的Ex-Hash目录中进行目录项搜寻要读取三个块信息（包括一个间接元数据块、一个散列块和一个叶块）。类似的B-树目录高度至少是6.平均下来，Ex-Hash目录在所有目录大小范围内要比B-树的存取时间至少快两倍。

将足够的I/O设备放到目录中也是一种有效措施。Ex-Hash和B-树常见的例子是没有溢出的加法（即块有足够的空间保存条目）。对于Ex-Hash搜索目录需要相同的时间。B-树总是将条目添加到叶子，这意味着添加一个节点，树的高度就增加。

当块没有空间存储项目是，更复杂的情况发生了。一个Ex-Hash目录将叶块拆分为两个。这包括访问两个叶块并改变到哈希表中的指针。B-树溢出要转移项目到临块或父块，甚至要分配一个新的块。 两项计划的比较大致就这样。

对于Ex-Hash来说最耗费资源的是当叶块不能被分割，而哈希表要增加一倍的时候。该哈希表必须被读入内存，并返回两次原来大小的数据（然后，叶块被分割）。由于哈希变得

更大而越来越昂贵，但幸运的是，这很少发生。

空间利用率是另一个问题。目录所使用的空间实际上用来装目录项的百分比是多少？ B 树总是保证在至少一半。但是也有一些保证三分之二满的变种。他们也有非常小的组织开销。在目录中使用的每块保存目录条目。唯一的存储组织数据是 n +1 个数据，每块存储对每一个 n 项的指针。

有没有什么样限制目录的分配的空间实际使用存放目录条目的百分比。该空间目录的效率依赖于质量的哈希函数（见第 4.1.5）。 ExHash 目录也有块没有持有目录条目（即，哈希表）。这降低了空间效率目录有点，但一般需要哈希表不到百分之一的叶块的大小。

我们认为，平均而言，提供了可扩展 hashing 目录组织的 B 树更好的方法。这是速度比 B 树，但多一点点空间消耗了。鉴于可用磁盘空间急剧增加，在存取时间慢得多下降，我们觉得这是正确的权衡。

4.1.5 散列函数

哈希函数在这个算法中使用是很重要的。它必须提供均匀分布的哈希键。如果不是均匀分布的话，目录的空间利用率会下降。由于使用更大的哈希表，在目录节点和哈希表数据之间的目录节点块增加，导致每次查询中读取的块数增加，使得空间转换效率较低，并且访问时间变慢。

一个要考虑的重要事情是哈希键的设置。一般的哈希函数设计为通过字典单词来很好的工作。在这些哈希键中各个键值之间没有太多相同之处。GFS 哈希函数的键的处理更加的结构化。一个很好的例子是数字时域通过时间步后产生成千上万以时间有关命名的文件（如“timestep.00001”, ” timestep.00002” , ”timestep.00003”,等等）。日期是存放在大型目录中文件文件名的另一个共同内容。。最常见的“折叠和散列“函数[20],[22]键位除重叠外就这么多。

GFS 为它的哈希函数提供了 32 位的空间效率测量（CRC）。由于 CRC 设计为对单一的位进行错误检查，因此提供了很多统一的哈希键。在两个名字中的少数位的改变会在 CRC 中通过提供一个数字显示出很大的区别。我们的结果显示 CRC 在很大程度上比别的文件类型键的哈希，甚至比目录类型键的哈希执行的更好。作者目前所了解，之前没有使用过 CRC 方法作为可延长哈希目录的哈希函数。

GFS 目录代码是仪器装备的，因此当目录填充后可以提供空间效率测量。这项措施被定义为：

eff = 入口数量/块数量（每个入口）　　（1）

其中，效率的数字表示的分配给该目录的空间分数，实际上包含目录条目。

作为测试，一个目录中充满了 45402 个以字典中的单词命名的文件。这样做两次。一次 GFS 使用传统的哈希（参考[20]），一次使用 CRC 哈希。从插图 6 中结果可以看出，公式（1）中的“每个入口（EntrysPerClock）“值约为 14.6 。

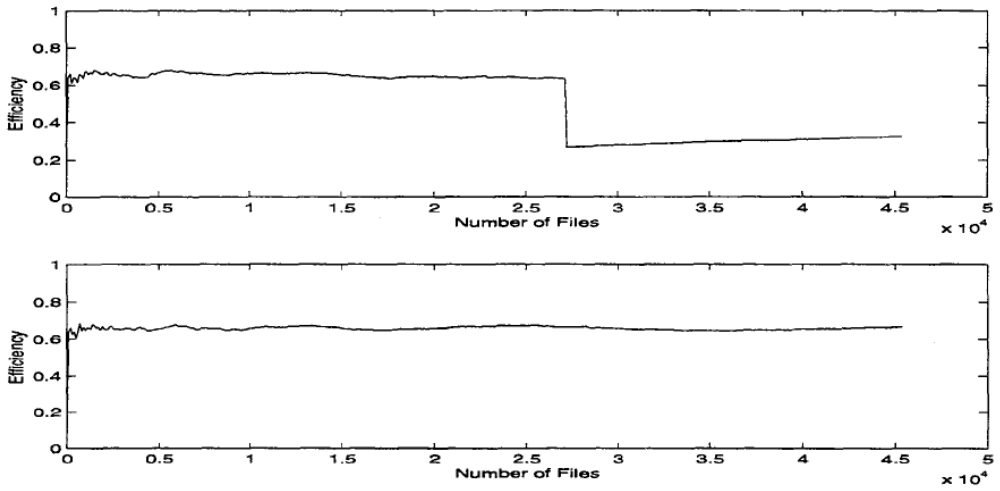

Figure 6: The space efficiency of a GFS ExHash directory as 45,402 files with names from the dictionary are created. The top graph shows the performance of a conventional hash. The bottom graph shows the performance of a CRC hash. The efficiency measure indicates the fraction of the space allocated for the directory that actually contains directory entries. (i.e. – An efficiency of one is optimal)

结果显示直到约第 27000 项前传统的哈希效率一直很高。在这里，传统的哈希产生了太多的几乎相同的哈希键。新的目录引起了叶的溢出。因为哈希键几乎相同，无法在两个新叶中平分条目。事实上，它们都被存放到一个叶子。叶的再次溢出导致第二次平分。溢出一直持续到只有一个指针从哈希表指向叶时。这样，溢出导致哈希表翻倍。之后可以进行叶的平分，但又会产生一个导致哈希表翻倍的溢出。

总之，哈希表很快达到其最大大小 16 MB。哈希表需要的所有空间减少了目录的空间利

用效率。与此相反，CRC 提供了更加均匀分布的哈希散列键。这提高了 CRC 哈希目录的效率并使它更加一致。它还可防止级联溢出的情况。数百万入口的目录通过 CRC 哈希创建，而重复溢出从来就不是一个问题。

随着哈希表的增长，访问目录的时间也有所增加。当哈希表很大时，访问目录需要访问间接块（回忆一下，哈希表作为一般文件数据存储）。此外，对于只有一个指针的叶溢出问题通过建立一个叶的连接表得到解决。因此，大型哈希表不断需要一个目录的空间利用效率，还需要速度。

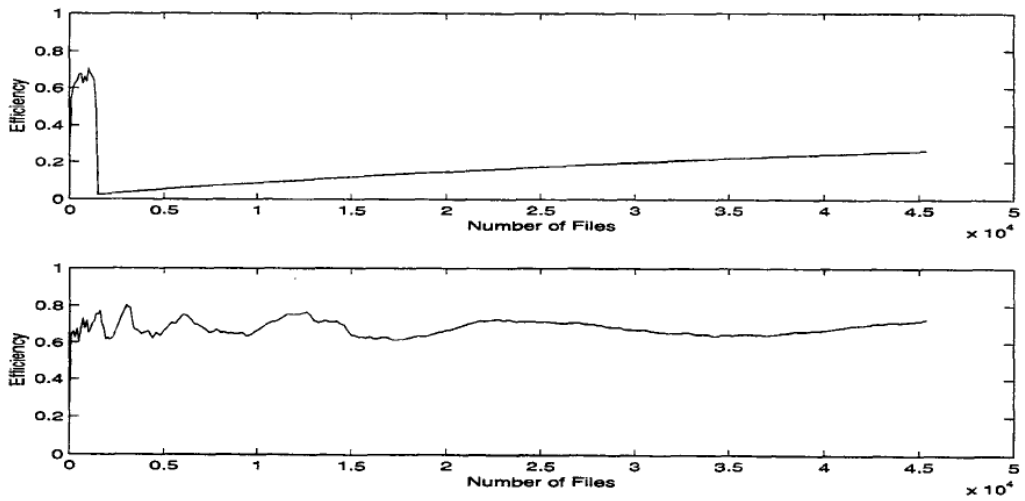

Figure 7: The space efficiency of a GFS ExHash directory as 45,402 files with names "file.0000000000" through "file.0000045401" were created. The top graph shows the performance of a conventional hash. The bottom graph shows the performance of a CRC hash.

对 45402 个文件类型名字做同样的测试（我们使用 file.0000000000,file.0000000001,file.000000002 等等）。从插图 7 的结果可以看出，前 1500 项导致哈希表翻倍达到最大容量。该 CRC 哈希效率超过它使用字典中的单词，但仍然是一个合理的哈希表大小。

总之，CRC 哈希比传统的哈希函数产生大量更好的结果。而且 CRC 哈希更快速。该方案实施在基本操作（与，移位，异或等）和 1KB 大小的查找表中查询。CRC 哈希比传统的哈希相比需要更多的时间来计算。

4.2 GFS 一致性

当元数据存取和更新时，必须十分小心。如果正确的块不是在适当的时候进行操作，很容易导致元数据损坏。目前大部分 GFS 的主要工作放在确保所有的锁都在正确的位置。

这种新增的锁也相应增加了死锁。文件系统在很多地方执行操作要同时保持多个锁。例如，查找操作就同时需要两个锁。查找操作需要一个目录和目录中的一个文件名，返回文件的节点。这个操作需要这两个锁：当读取目录和进行文件节点号确认时必须有一个锁。当读取文件节点时需要一个锁。这两个锁必须同时锁住，否则，当这个机器上有相同的进程进行查找操作时，要求释放这两个锁会产生竞争条件情况。

还是有一些地方发生两个或多个锁死锁的情况。期望进行块回收是很困难的，因为块被分割成目录结构的各个部分。防止文件系统树一部分死锁的命令，可能会导致另一个地方的死锁。

GFS 处理这个问题的办法是后备取舍和重复尝试。如果一个客户端已经占有了一个块却仍需要其他块资源，会占有这个块一段时间。如果这个块资源没有获取到，就构成了产生死锁的条件。它释放第一个锁，睡眠一个随机时间，之后尝试整个操作。这样可以避免死锁，但不是最好的方案。该协议的最新版本允许 Dlock 客户端之间相互交谈，在必要时可以应用单独的公平共享协议。

另外一个新特性是现在可以递归获得 Dlock。这通过增加文件系统和 NSP 设备值之间的一个层来实现，其中 NSP 用来在每个命令发出前进行检查。如果 Dlock 一般被具有相同进程 id 的进程占有，则计数器增加并把命令传递给文件系统，好像进程成功实施了锁设备。如果提出申请的是另一个进程，这这个进程会一直睡眠到占有块进程释放这个块资源。如果块没有被进程占有，这个命令一直传到设备池和实际的 Dlock 设备。

当解锁命令发出后，计算器递减。当计算器是 0 时，命令传到设备池和实际设备。

该算法的一个十分有趣且有用的副作用是可以防止对同一个机器的同一个设备的同步锁。如果一个进程占有一个块，而另一个进程来申请这个块要睡眠到前一个进程结束。这最大程度的减少了网络流量。

这个递归锁层对于下一代 GFS 十分重要。在这个新版本中，GFS 目前占有块的时间会更长些。这样就可以写入缓存和减少时间延迟的影响。递归锁块只需要很小的变动代码就可以使得占有块的时间更长。

为了激活缓存，当一个 DLock 第一次被申请，它的“锁的次数“被设置为 2（而不是 1）。从这点到之前的代码会想平常一样申请和释放锁。不同点是锁和解锁都是文件系统内

部命令，而不是锁设备的。当文件系统要释放锁设备的锁时，它至少调用一次解锁例程。“被锁次数“计数器递减到 0 并对锁设备调用解锁命令。

4.3 应用高速缓存缓冲区

高速缓存缓冲区是现代 UNIX 操作系统的重要组成部分[23],[2]。为了防止对磁盘的过多访问，将最近操作的文件作业保存到内存部分叫“高速缓存缓冲区“的磁盘块中。因为不用进行磁盘访问，之后对文件数据的访问可以快速完成。如果请求的数据不在高速缓存缓冲区，会从磁盘中把数据复制到缓存缓冲区中再返回个用户程序。这对元数据和文件块都适应。

不像 IRIX，Linux 目前没有提供绕过高速缓存缓冲区的“直接“磁盘访问。IRIX 没有缓存区对于大型磁盘读取是有益的，因为在操作系统中只有一个内存拷贝，而不是两个。不过，在两个平台上的普通用法特点是通过小文件的频繁读取。在这个情况下，通过高速缓存缓冲区比读取磁盘要大大提高性能。对于大型文件来说，因为不用反复引用间接块，使用元数据缓存大大提高了性能。

在 GFS 中使用高速缓存缓冲区是比较复杂的，因为要提供多个客户端同时访问和缓存同一个磁盘块的能力。当客户端检测到磁盘上数据改变（由 Dlock 的计算器表示），需要把其缓存区的数据设为无效并对数据进行重新读取。在 GFS 中对每个块的缓存缓冲区最近更新进行跟踪，以便在必要的时候把它设置为无效。这样可以使用缓存缓冲区来读取、对小文件反复请求和加快大型文件的访问速度。过去没有这个能力，缓存区在读取后就立即失效了。缓存的权限使用很困难且在 GFS 中无法提高，直到最新的 Dlock 规范使用后后[12]。

Linux 使用任何备用内存空间作为缓存空间，且在内存空间不足时可以回收这些空间。这把问题进一步复杂化，GFS 不应尝试把被操作系统回收的缓存区设置为无效。在缓存区的头部插入一个函数指针，使得 GFS 可以删除一个离开缓存去头部的链接。通过 Linux 例程管理缓存的变化是可行的，因为内核代码是可用的。IRIX 的源代码在将来会提供类似 GFS 的改进。

现在可以支持对缓存区进行位、块的预读。在这种方法中，一个块的请求结果在同一时间会在下几个块中访问，并存储到缓存区中。本地程序使得其后的块读取和预读可以提高整体性能。

4.4　空闲空间管理

GFS 当前的空闲空间管理方案是基于位图的。对于文件系统的每一个块，在给定的资源组中有数据表示是否空闲。当文件系统填满数据，为了找到需要的空闲空间需要增加搜寻的位数，这会降低效率。这很耗费资源，每增加一个字节，要验证每一个条件，更耗费资源的是要搜寻每一位。

新的方法是建立一个计划，这会耗费一定的资源，但可以提高性能。与把每个文件系统队列看作一个资源组不同，我们自己对空闲块进行限制。因为资源组中的每一组空闲文件系统块，都会引用队列的起始块和块的数量，如插图 8 所示。当创建一个文件系统，在每一个资源组中都会有一个引用。当增加文件时，只需要改变引用的起始地址。当一个文件删除，如果不能增加到一个存在的引用中，就要创建一个引用。如果文件系统高度分离，那么保存的引用占有的空间就会很大。

这个方法有两个很明显的优点。首先，这没有必要去为了找到空闲的块而对整个块图进行搜寻。当我们找到空闲块队列后，工作的重点就是在组中找到一个足够大可以存放文件的空闲块。第二个优点是这个计划可以把块设为“守门“块，就是其他块放在这个块后面。这样我们可以努力在磁盘上把组元数据和数据合在一起。这种方法需要更多的时间来通过引用搜寻到最靠近“守门“块的块，这在将来可能会减少磁盘潜力。

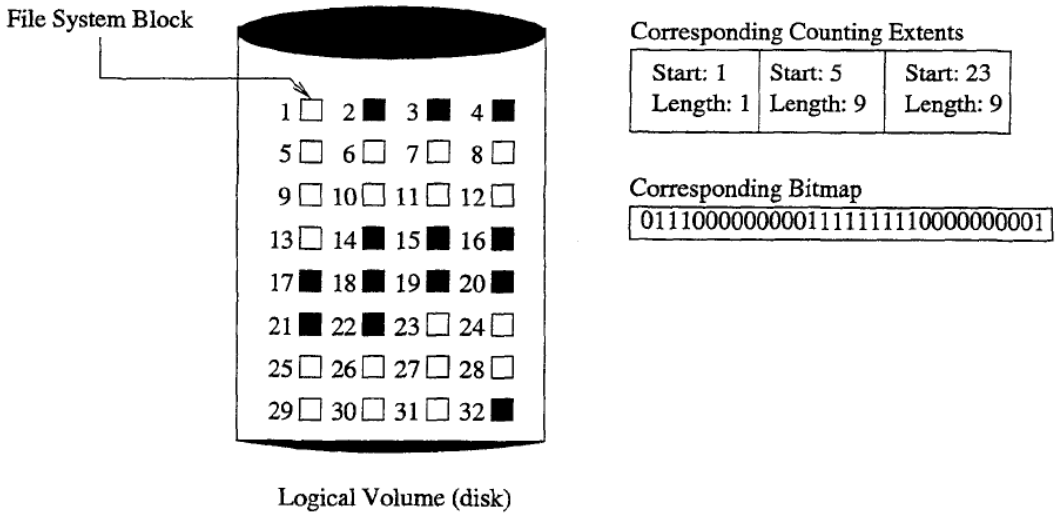

Figure 8: The extent and bitmaps encodings of 32 blocks, including free and in-use blocks.

4.5 网络存储池

池设备联合一个由收集不同成分形成的共享存储为单一值称为网络存储池。池设备是建立在 SCSI 和纤维通道之上，类似于 SCI 的 xlv 和 linux 的 md。它可以拆开多个设备并为 GFS 提供一个块池，隐藏实施细节。设备可能通过特殊的特性拆分为子池。

目前池设备的最近变化使得它和标准 Linux 块设备接口更适配。池设备变体中的设备锁选项同样支持从失败的锁请求中指数级回放，努力获得锁，并放弃块。我们开发了一个可以运行在所有 IP 网络上的 Dlock 服务守护程序。这个能力对于 GFS 的发展很有用，还可以帮助那些不支持 Dlock 的磁盘设备（GFS 可以通过单一机器模式运行在不支持 Dlock 设备上，意味着 GFS 可以和 Linux 其它桌面文件系统一样高效处理任何种类的磁盘）。其它池设备条件包括在核心同步更新池的用户层用具，和建立在池参数的同步创建文件系统。

Ptool 是一个用户层工具，可以通过用户编写的参数文件对池设备进行设置。池名字，子池定义，子池设备（包括磁盘部分），大小和范围，甚至 Dlock 设备都可以在参数文件中设定。这些参数通过标签在池设备中的任何磁盘使用前都会被读取使用。Ptool 只需要一个客户端在池中创建就可以被所有客户端访问。

Passmble 是一个用户层程序，它扫描一个客户端的所有设备来决定存在哪个设备并可以被使用。共享设备上的所有标签（由 Ptool 编辑）被读取来形成逻辑上的池定义。这些信息之后被传给核心，再被添加到自身的管理池列表中。早期的一个重要改进是从池标签中删除了主要和辅助号，因为不同的用户会定义不同的设备。特制标签的需要条件部分也被删除了。新的存储池可以被编辑、组合和添加到动态核心。Passmble 必须在一个池设备被创建后，由一个用户启动时运行。池设备文件可以被 Passmble 创建和删除，正如存储池添加和销毁。

4.6 Dlock 新特性

新版本的 Dlock 协议特性使得 GFS 执行的更好，也更可靠。新的锁协议定义在文献[12]。主要添加的有：

1.Dlock 超时

现在每个 Dlock 都有一个时间限制。如果一个锁在锁住状态过长时间的话，就会自

动失效并解锁。一个客户端想使用锁更长时间的话，可以在重设锁的时间后发送“touch lock“命令。

这个新特性解决了 GFS 的一个很大性能问题。它可以编辑缓存区。过去，客户端需要花费一定的时间进行锁检测来发现失败的客户端；当锁没有活力后要手动从新设置。这意味着一个锁可以占有最大的时间。

在新版本中，当 Dlock 超时后，锁设备决定哪个客户端失败。一个客户端可以持有锁更长的时间周期，并且其他客户端不能读取或编辑这个锁。这意味着锁不必同步写回信息，使得广泛的编辑缓存变得可能。

2.返回客户确认号

在新的锁方案中，每个 GFS 客户端指定一个独一无二的四位整数。客户 ID 在 Dlock 命令的 SCSI 命令描述块中传到到 Dlock 设备。

当有其它客户端占有这个锁时 Dlock 命令失败，并且返回这个客户端在机器中的客户号。这使得仍然保持 GFS 锁中心，同时客户端之间可以交流。

这同样帮助客户端持有锁更长的时间。当一个客户端想获得一个被其他客户端持有的锁时，可以通过发送 SCSI 信息给这个客户端。这个信息是可以询问对方能否释放锁，或者询问是否可以授权访问第三方到或从磁盘翻译。

3.Reader/Writer 锁

很多数据尤其是元数据，总是经常读取却很少写入。这种类型的存取模式使得很好的使用 Reader/Writer 锁。Reader 需要多个读取锁中的一个。Writer 需要一个 reader 锁和其它 writer 锁。新协议中的每个锁都是 reader/writer 锁。这对于想 root 目录一样高度拥挤的区域有很大的帮助。

第五章　性能报告

插图 9 显示 Linux GFS 的 11 种单个用户 I/O 带宽。这个测试是实施在 533 MHz Alpha 和拥有 512 MB 的 RAM ，使用 Linux
2.2.0-pre7 核心。机器通过一个Qlogic QLA2100 主机设配卡循环连接到 8 个存储STFC纤维通道设备。在写这篇论文的时候，GFS已经可以读取缓存。读取头部和编辑缓存还没有得到实施。这个测试使用一个 4096 位的块（使用一个 8192 位大小的块会提高 10%的性能，但并不是所有的Linux都有这么大的块）。

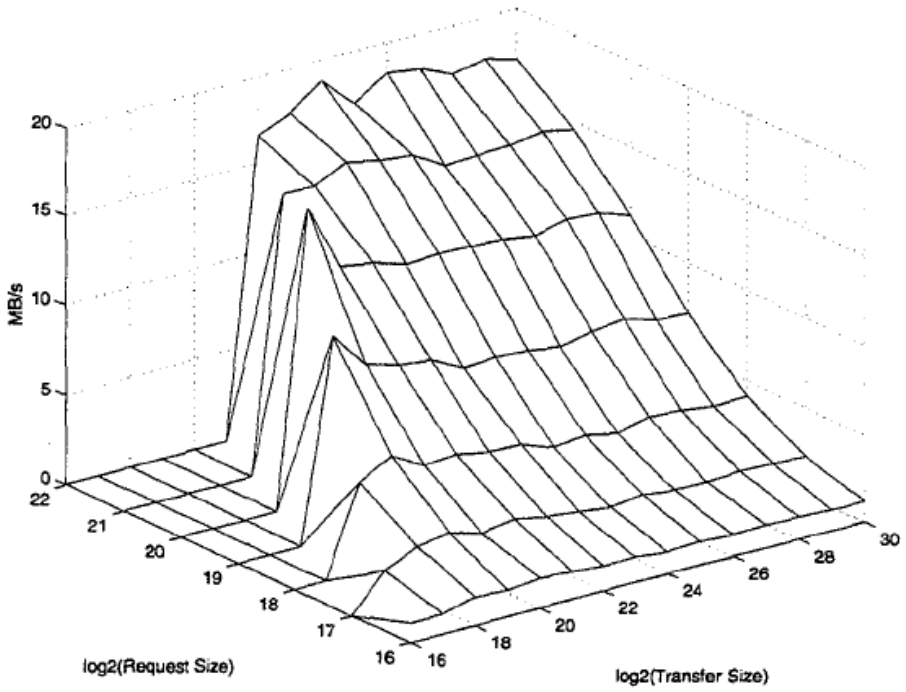

Figure 9: Linux GFS create bandwidth

如插图 9 显示，第一次创建的带宽是 17.5MB/s。使用更小的块，GFS性能失去了什麽似的只有 1.3MB/s。当实施了编辑缓存后这会得到很大的提高。编辑缓存可以把小型块和大型

块捆绑在一起，从而整体上提高性能。现在，Dlock对于每一个请求都可以获取和释放。实施编辑缓存需要Dlock持有锁更长的时间。没有了请求锁的开销，小型I/O请求应该会得到提高。

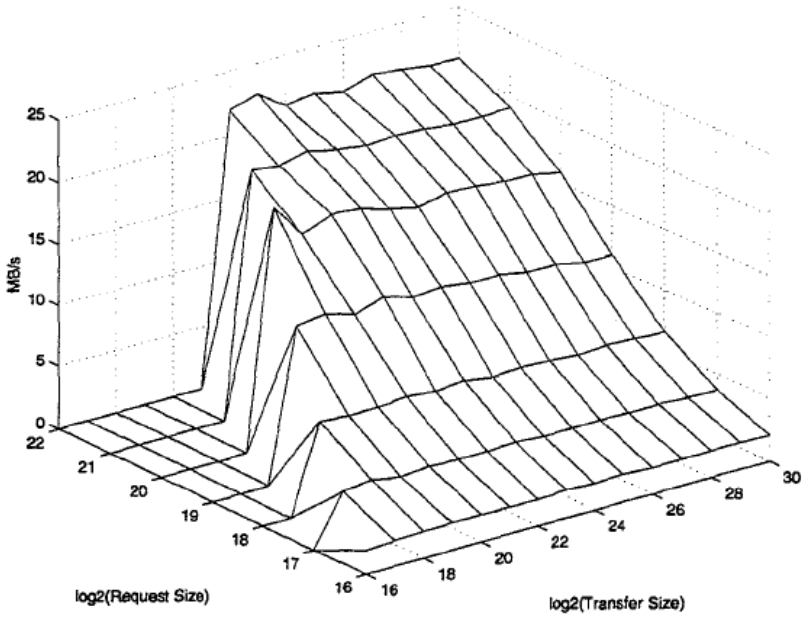

Figure 10: Linux GFS preallocated write bandwidth

插图 10 显示了在相同机器上预分配编辑的带宽情况。为了预分配编辑，必须快速缓存元数据（和本地数据块），并尽可能快的编辑数据。没有必要去分配块，从而使得文件系统速度增加 25%。在最终的IRIX释放中，预分配编辑是创建速度的两倍[24]。实际上，在创建和预分配编辑上只有很小的区别，这得益于Linux GFS的常规新块实施和读取缓存机制。

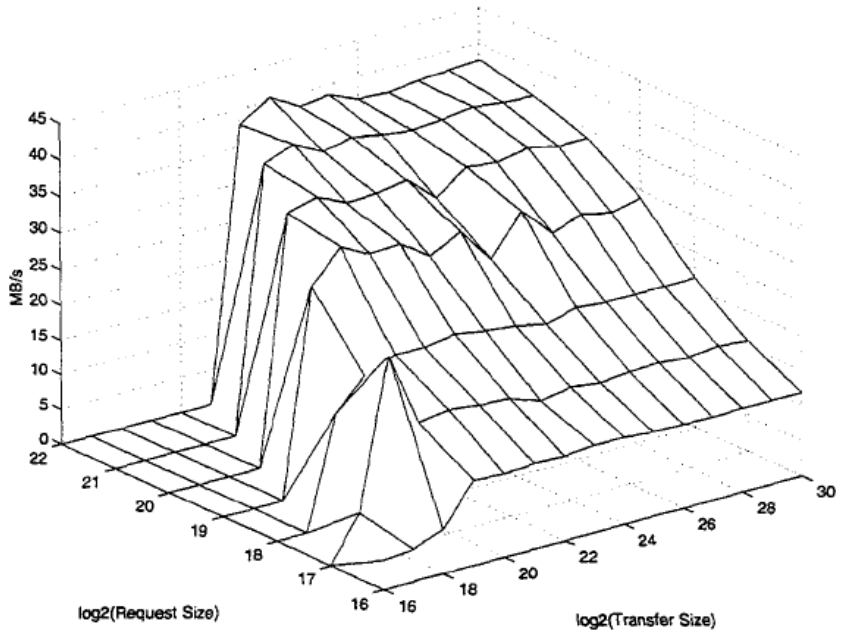

Figure 11: Linux GFS read bandwidth

插图 11 显示读的带宽是 38MB/s。这比IRIX GFS的缓存I/O更快（它只有 23MB/s）。在小型I/O请求方面比IRIX GFS提高很多。在IRIX GFS上小型请求的带宽是 1～2MB/s（Linux GFS在可以读取缓存前也是这样）。通过读取缓存，Linux GFS对小型请求I/O是 12MB/s。当实施了读取头部和提高Dlock访问例程后，提高的速度还会更多。

所有的数据显示Linux GFS的缓存I/O比IRIX GFS缓存I/O性能更好。当然，IRIX GFS间接缓存I/O使得速度得到很好的提高。对于大型文件的创建带宽是 35MB/s，对于读取是 55MB/s（在之前的IRIX GFS，目录I/O没有对小型文件的速度有很多提高）。在 3.1.2 节，Linux还没有Direct I/O。一旦克服这个障碍，创建和读取的带宽都会提高。

第六章　下一步工作

GFS在过去的三年有了很好的发展，但还有很长一段路要走。当前和未来的工作将在下面描述。

6.1 错误处理

错误处理在共享磁盘文件系统中是非常重要的。所有的客户端可以立即处理元数据，所以失败时客户端可以在任何不合理的地方离开元数据。以后，如此多的机器同时访问磁盘，当一个客户端结束时不用每次进行文件系统检查（fsck）是很重要的。当文件系统在线时，这对于简单修复由失败客户端引起的错误很重要。

我们目前研究了一系列的方法来进行快速检查和处理。这包括截图、写入日志和存入消息记录。最终论文将描述 GFS 的可剥离元数据记录计划。

6.2　共享操作系统池

从 GFS 到 linux 的端口使得可以在 IRIX 和 Linux 客户端之间共享文件系统（对以后其他平台的 GFS 也一样）。没有使用文件系统或池设备来做这个工作。目前，为了支持 Alpha 和 x86 Linux 客户端，位排序和结构包例程添加到了文件系统和池设备。在 GFS 可以在不同的操作系统共享之前的最后一个工作是常规磁盘分离格式的不足。

为了专注于存储数组，最简单的办法是取消分离格式。当处理一个由数百个磁盘组成的磁盘阵列，不用把磁盘分成一块块的。用不同的方式组织磁盘，系统需要提供设置能力。

很多时候尤其是建立在 IP 上 SCSI，跨平台分离格式就很有用。微软的分区格式很常用，如果没有被限制的话会是一个好选择。当然，很多其它的标准也要被支持。

如果 GFS 在 Linux 上成功了，就可以为其它程序员编写共享文件系统提供一个实施参考。如果别的操作系统使用 GFS 协议并把元数据实施共享文件系统，这样就可以进行相互操作且避免共享文件系统的不一致性。

6.3 扩展文件系统

当设备被添加到存储网络，文件系统应该能力进行同步扩展并使用这些空间。扩增文件系统的那一部分是第一步工作。这通过为空间增加一个子池来完成（限制子池的可拆除性）。把新数据传到核心和增加子池到核心内部结构一个简单的过程。最复杂的是使用 ptool 和 pasemble 来同步改变池定义标签和从新池正确集合池的定义，正在使用或没有使用的池设备都属于集合池。在文件系统层，一个程序需要更新超级块并且重新读取在磁盘中的索引，再提醒每个客户端的文件系统重新读取可以使用的新空间。我们努力改进文件系统使它可以这样扩展。

6.4 一个 BSD 的 GFS 端口

GFS 的工作目标是由不同成分组成的工作站簇。这个认为有 GFS 到 BSD UNIX 的端口完成。

6.5 应用 IP 的 SCSI

通过编写代码使 SCSI 命令和数据遵循 IP 网络，这扩充了存储局域网络的想法[25]。一个像 GFS 这样的共享设备文件系统可以在更大的硬件范围内存储数据。和网络附上存储被限制在磁盘数组不同，一个电脑可以把本地磁盘引导给 IP 网络，实际上成为了一个网络存储设备。

自从任何机器都可以成为网络附上存储设备，更新到更加适合 SAN 的硬件就不用那么急迫了。一个 GFS 的安装可以通过以太网硬件实施。因为需要增加 I/O 带宽，纤维通道可以增加到存储局域以太网络（如插图 12）。GFS 从以太网和纤维通道网络存储数据都是一样灵活，但使用新的设备会更快些。

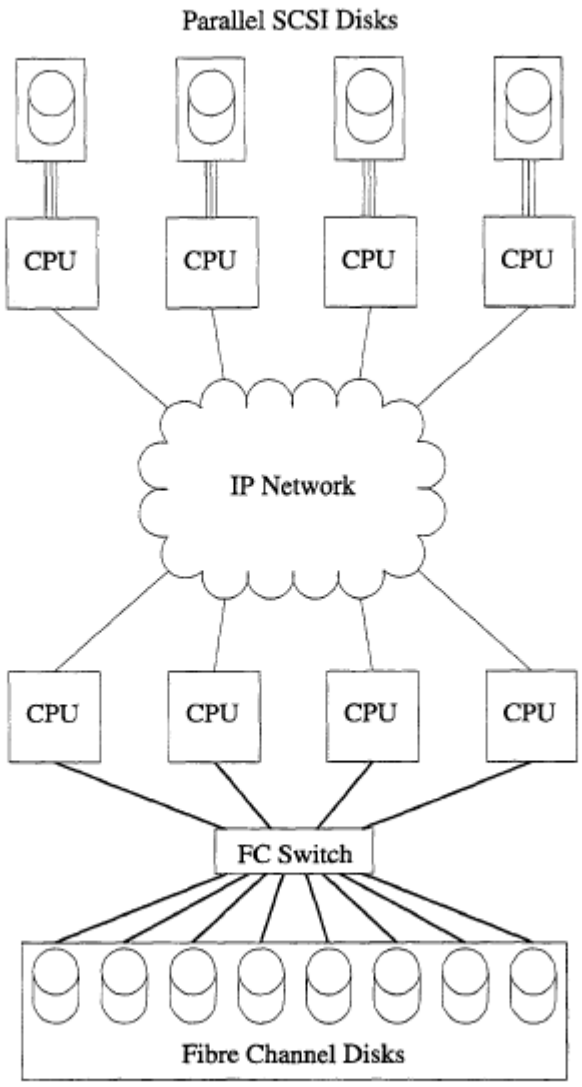

Figure 12: A Storage Area InterNetwork – Because all the CPUs are exporting their disks with SCSI over IP, all machines can access all disks.

SCSI 应用 IP 之上的关键是两个软件，客户端和服务器端。一个服务器端守护程序等待连接。当连接建立后，守护程序接收从网络上转换来的 SCSI 命令。然后对这些命令进行重新打包并通过本地 SCSI 总线传递给本地磁盘（同样可以传到纤维通道磁盘上）。服务器端从磁盘获得应答，打包之后通过 IP 传回给客户端。

客户端通过一个到操作系统的接口而看起来像一个 SCSI 磁盘。当它从更高层次获得请求，将命令打包并通过网络传到服务器机器。再把从服务器端传回的应答传给更高层次。

打包 SCSI 命令并通过专线或网络传递的技术已经是 SCSI-3 的一部分。所有这些都要实施设备。这应该是直接前进的。Van Meter 已经实施了一个计划并显示让 SCSI 速度超过以太网[25]。

服务器这边同样可以发送 SCSI 命令。服务器将查看哪种类型的 SCSI 命令被转换了。如

果是一个特殊命令，服务器可以自己保存并不经过磁盘给客户端返回一个信息。其它命令直接传给磁盘。

Dlock 命令也可以这样实施。命令先在进程中进行标准化，直到它足够宽能用在 SCSI 设备，服务器程序发送它。

第七章 致谢

这些年很多人为 GFS 共享了代码和想法。作者要感谢这些人：

来自 Minnesota 大学的 Benjamin I. Gribstad, Steven Hawkinson, Thomas M. Ruwart, Aaron Sawdey,

来自 Seagate 技术公司的 Dave Anderson, Jim Coomes, Gerry Houlder, Nate Larson, Michael Miller,

来自 NASA Ames 研究中心的 Alan Poston, John Lekashman

来自Ciprico公司的Edward A. Soltis

作者还用感谢Sam Coleman审阅了这篇论文。

参考

[1]L. McVoy and S. Kleiman. Extent-like performance from a unix file system. In Proceedings of the 1991 Winter USENIX Conference, pages 33-43, Dallas,TX, June 199 1.

[2]Uresh Vahalia. Unix Internals: The New Frontiers.Prentice-Hall, 1996.

[3]Matthew T. O'Keefe. Shared file systems and fibre channel. In The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems, pages 1-16, College Park, Maryland, March 1998.

[4]Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Tom Ruwart. The design and performance of a shared disk file system for IRE. In The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fpeenth ZEEE Symposium on Mass Storage Systems, pages 41-56, College Park, Maryland, March 1998.

[5]Roy G. Davis. VMCluster Principles. Digital Press,1993.

[6]N. Kronenberg, H. Levy, and W. Strecker. VAX-Clusters: A closely-coupled distributed system.
ACM Transactions on Computer systems, 4(3): 130-146, May 1986

[7] K. Matthews. Implementing a Shared File System on a I-IiPPi disk array. In Fourteenth IEEE Symposium on Mass Storage Systems, pages 77-88, September 1995

[8]G.Pfister. In Search Of Clusters. Prentice-Hall,Upper Saddle River,NJ,1995.

[9] Aaron Sawdey, Matthew O'Keefe, and Wesley Jones.A general programming model for developing scalable ocean circulation applications.In Proceedings of the 1996 ECMWF Workshop on the Use of Parallel Processors in Meteorology,pages 209-
225,Reading,England,November 1996.

[10]Steven R. Soltis,Thomas M.Ruwart,and Matthew T. O'Keefe. The Global File System.In The Fifth NASA Goddard Conference on Mass Storage Systems and Technologies,volume 2,pages 319-342,College Park,Maryland,March 1996.

[11]Steven R.Soltis.The Design and Implementation of a Distributed File System Based on Shared Network Storage.PhD thesis,University of Minnesota,Department of Electrical and Computer Engineering,Minneapolis,Minnesota,August 1997.

[12] Matthew T.O'Keefe,Kenneth W.Preslan,Christopher J.Sabol,and Steven R.Soltis.X3T10 SCSI committee document T10/98-225R0-Proposed SCSI Device Locks.http://ftp.symbios.com/ ftp/ pub/ standards/ io/ x3t10/ document.98/ 98-225r0.pdf,September 1998.

[13] M.Beck ,H.Bohme,M.Dziadzka,U.Kunitz,R.Magnus,and D. Verworner.Linux Kernel Internals.Addison-Wesley,second edition,1998.

[14]Alessandro Rubini.Linux Device Drivers.O'Reilly & Associates,1998.

[15] S.R.Kleiman.Vnodes:An architecture for multiple file system types in Sun UNIX.In

Proceedings of the Summber 1986 USENIX Technical Conference, pages 238-247, June 1986.

[16] Stephen Tweedie. PATCH: Raw device IO for 2.1.131. http://www.linuxhq.com/ lnxlists/ linux-kernel /1 k_9812_02/ msg00686.html, December 1998.

[17] Alan F. Benner. Fiber Channel:Gigabit Communications and I/O for Computer Networks. McGrawHill, 1996.

[18]Chris Loveland and The InterOperability Lab of the Unixersity of New Hampshire. Linux driver for the Oiolic ISP2100. http://www.iol.unh.edu/ consortinums/ fc/fc_linux.html, August 1998.

[19] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible Hashing-a fast access method for dynamic files. ACM Transactions on Database Systems, 4(3):315-344, September 1979.

[20] Michael J. Folk, Bill Zoeilick, and Greg Riccardi. File Structures. Addison-Wesley, March 1998.

[21] Douglas Comer, The ubiquitous B-Tree. Computing Surverys, 11(2):121-137, June 1979.

[22] Aaron Sawdey. The Sawdey Hash. Email Message, April 1998.

[23] Maurice Bach. The Design of the UNIX Operating System. Prentice-Hall, 1986.

[24] Grant M. Erickson. The design and implementation of the global file system in silicon graphics' irix. Master's thesis, University of Minnesota, Department of Electrical and Computer Enginerring, Minneapolis, Minnesota, March 1998.

[25] R. V. Meter, G. F. Finn, and S. Hotz. VISA:Netatation's Virtual Internet SCSI Adapter. In Proceedings of the 8th International Conference on Architechural Support for Programming Language and Operating Systems(ASPLOS-VIII), San Jose, CA, October 1998.

[26] X3T10 SCSI committee. Draft proposed X3 technical report-Small Computer System Interface-3 Generic Packetized Protocol(SCSI-GPP). http://ftp.symbios.com/ ftp/ pub/ standards/ io/ x3t10/ drafts/ gpp/ gpp-r09.pdf, January 1995.

[27] John Lekashman. Building and managing high performance, scalable, commodity mass storage systems. In The Sixth Goddard Conference on Mass Storage Systems and Technologies in cooperation with the Fifteenth IEEE Symposium on Mass Storage Systems, pages 175-179, College Park, Maryland, March 1998

所有的GFS的出版吾和源码可以从这个网址找到：http://gfs.lcse.umn.edu