华中科技大学

硕士学位论文

并行文件系统缓存技术研究

姓名：林运章

申请学位级别：硕士

专业：计算机软件与理论

指导教师：韩宗芬

20040412

摘　要

作为集群的 I/O 子系统，并行文件系统实现对分布在集群内各节点上的文件、设备和网络资源的全局访问，为集群服务器系统设计一个高性能的并行文件系统是很有必要的。利用主存速度和网络带宽、磁盘带宽之间的差异，在主存中实现缓存是提高并行文件系统性能的主要手段之一。

PVFS（Parallel Virtual File System）是一个开放源码的并行文件系统。针对 PVFS 文件访问速度慢、可靠性差等缺点，HPVFS（High performance Parallel Virtual File System）以 PVFS 为基础平台，实现了多项高可用、高可扩展、文件高速访问方案，其中包括缓存系统。HPVFS 的缓存系统包括元数据缓存、计算节点数据缓存和存储节点缓存三种，旨在实现元数据和数据的高速访问。

元数据缓存通过改进元数据访问网络协议等措施，对命中的元数据减少一次网络通信，使各有关操作速度得到提升；此外，将脏元数据直接写回元数据服务器，并迅速检查和纠正元数据异常，保证元数据缓存的一致性。

采用共享内存方式实现计算节点缓存。每个计算节点都运行一个守护进程申请和管理共享内存缓存空间，应用进程把共享内存映射到自己的内存空间便可以直接访问缓存，减少了进程通信时的内存拷贝次数，实现了节点内进程间数据的真正共享。

存储节点缓存关注多个存储节点同时向一个用户提供数据时可能出现的一个严重问题：单一存储节点缓存单独缺失会降低其它存储节点缓存效率。采用存储节点缓存协同置换算法 *CMQ*（Coordinated Multi-Queue）把一些经常同时被访问的数据块归为一个访问组，并对访问组施行整体置换策略，使各存储节点缓存的命中率趋于均匀，从而提升并行文件系统存储节点缓存的整体命中率。

测试结果表明，元数据缓存可以使元数据相关操作速度提升一倍以上；计算节点缓存可以使并行读和并行写性能提高 30%以上。仿真结果表明，与 LRU，LFU 等传统置换算法相比，用于存储节点缓存的置换算法 CMQ 命中率提升可达 125%以上。

关键词：并行文件系统，缓存，元数据缓存，计算节点缓存，存储节点缓存

Abstract

As the I/O subsystem of cluster server, Parallel file system realizes the globally accessing of all files, devices and network resources distributed on the nodes of cluster. It is necessary to provider a high performance parallel file system to cluster server. Making full use of the distance among the memory speed, network bandwidth, and disk bandwidth, building caches in the system is one of the major methods to enhance the file system performance.

PVFS (Parallel Virtual File System) is an open source parallel file system, which has poor file access performance and poor scalability. Aimed at these problems, HPVFS (High performance Parallel Virtual File System) implements multiple high scalability, high availability, and high I/O throughput rate strategies on basis of PVFS. Cache system is one of these strategies, which realize rapid access of metadata and data. The cache system of HPVFS consists of metadata cache, compute nodes cache and storage nodes cache.

With strategy of simplifying metadata access protocol, the metadata cache reduces one communication with metadata server once the cache hits. Because the metadata can be timely written back to the metadata server, and the metadata exception can be quickly checked out and corrected, the metadata cache eliminates the consideration of metadata consistence.

The compute node cache is implemented using shared memory. A daemon which is in charge of applying and administrating the shared memory cache space begins to run when the cache system startup. The application processes attaches the shared memory to its own memory space, then they can access the cache directly. The shared memory cache reduces the times of copying data between tow processes, and realizes data truly shared among the processes in a node.

The design of storage nodes cache pays attention to a serious problem: a storage node cache single missed may break the parallel data accessing of the system, and degrade the efficiency of the others caches. The Coordinated Multi-Queue (CMQ) Algorithm relieves this problem efficiently. CMQ algorithm classifies the blocks which often be accessed together as an access group, and replaces the blocks of an access group in bulk. CMQ algorithm homogenizes the cache hit-ratios of all the storage nodes caches, and improves the global cache hit-ratios of the cluster file system.

The test results indicate that the metadata cache can raise the metadata operation speed

for above 100%, and the computing nodes cache can enhance the system performance of parallel read and parallel write for more than 30%. The simulation results show that the hit-ratios promotion of CMQ algorithm can come up to 125%, compared with traditional algorithms such as LFU and LRU etc.

Key words: Parallel file system, Cache, Metadata cache, Compute nodes cache, Storage nodes cache

独创性声明

本人声明所呈交的学位论文是我个人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除文中已经标明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

学位论文作者签名：

日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保留、使用学位论文的规定，即：学校有权保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权华中科技大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本论文属于

保密□，在_____年解密后适用本授权书。

不保密□。

（请在以上方框内打“√”）

学位论文作者签名：　　指导教师签名：

日期：　　年　　月　　日　　日期：　　年　　月　　日

1 绪论

1.1 课题背景

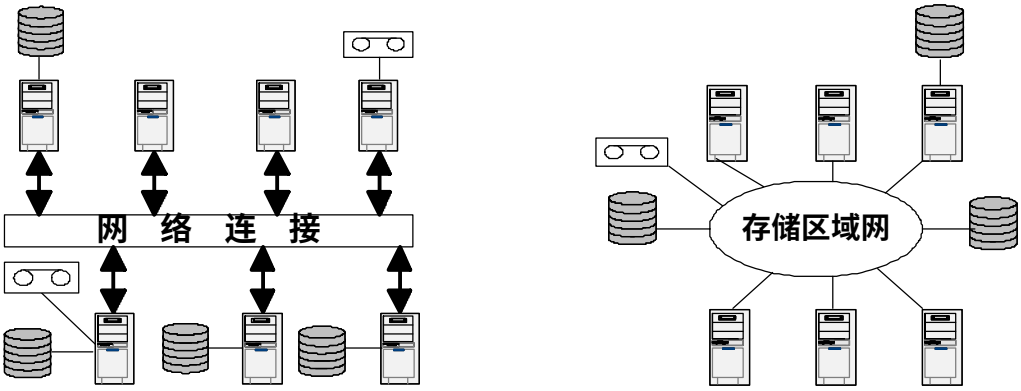
随着计算机技术和网络技术的飞速发展，数据量和服务量都呈几何级数爆炸式增长，这种增长速度在未来几年内都无法预测。各大企业及科研机构所配备的服务器在应对剧增的工作负荷时显得越来越力不从心。大多数企业和机构都无法承受采用传统MPP（Massively Parallel Processing）和SMP（Symmetric Multiple Processor）超级计算机处理业务所需的巨额费用。集群技术的适时出现为用户提供了一种处理迅速、I/0吞吐量高、容错性能好、可靠性高和可扩展性好的新型解决方案。集群服务器相对于其所能提供的计算能力可以说是非常廉价的。随着集群在提高可扩展性、提供完全的单一系统映像等方面做的越来越好，集群有可能在不久后取代MPP、SMP等机器[1]。近年来，集群计算正成为许多领域内的研究热点，广泛的应用于气象、基因数据处理、多媒体、I/O密集型数据库和语音图像识别等研究领域。

集群作为一种并行或分布式处理系统，由很多连接在一起的独立的计算机组成，像一个单独集成的计算资源一样协同工作[2]。集群需要通过一个并行文件系统实现对分布在集群内各节点上的所有文件、设备和网络资源进行全局访问，并且为用户呈现出单一系统映像。无论数据存储在集群中哪个节点上，任何用户（远程或本地）都可以通过并行文件系统实现访问，甚至在应用程序从一个节点迁移到另一个节点后，应用程序仍然可以透明地访问。因此，为集群系统设计一个高性能的并行文件系统有着非常重大的意义。

1.2 并行文件系统研究现状

根据数据存储的方式，并行文件系统可分为基于消息传递的文件系统和共享存储的文件系统，这两种文件系统结构如图1.1所示。基于消息传递的文件系统中各客户节点享用单一的文件名字空间，使用共享语义来保证文件的一致性。基于消息传递的文件系统的系统结构决定了它具备很好的可扩展性和可移植性。当各存储节点被均匀访问时，基于消息传递的文件系统能显示出很好的性能。但是基于消息传递的文件系统很难做到负载均衡，并且性能受到各节点的网络带宽的限制。共享存储的文件系统

中，任何一台机器都可以统一访问所有的存储设备，任何客户节点都可以进行任何的文件系统操作，而不会影响到其它客户节点。共享存储的文件系统不存在单一失效点，任何一台机器的故障都不会影响到系统中的其它机器，存储设备的带宽可以被所有节点充分利用。共享存储的文件系统是真正无服务器的，很容易做到负载均衡，可扩展性较好，不过存储设备较为昂贵。



(1) 基于消息传递文件系统结构　　(2) 共享存储模式文件系统结构

图 1.1　两种并行文件系统结构

用于集群环境中的并行文件系统也叫做集群文件系统。通常，基于消息传递的并行文件系统把文件数据分布存储在高速互联的多个存储节点上，计算节点利用文件系统把对本地逻辑文件读写语义转化为多个物理块的 I/O 请求。并行文件系统充分利用集群内部存储资源和网络资源，并通过集群内部各节点充分合作得到性能提高。

文件系统作为系统的 I/O 子系统在很多情况下会成为系统的一个明显瓶颈。这是基于这样一个事实：一方面微处理器性能每年提高 50%到 100%，而另一方面磁盘技术的发展主要是增加它的容量，而很难提高它的性能[3]。对于并行文件系统来说，网络带宽也是影响系统性能的一个关键因素。当网络通信过多，或网络传输速率与磁盘存取速率相当，系统性能瓶颈便体现在网络带宽上。因此，设计一个高性能的并行文件系统应该从两个方面考虑：一方面尽可能的减少对磁盘的访问，另一方面尽可能减化网络访问协议，减少网络访问。

目前，国内外设计的分布式、并行文件系统的主要代表有 Coda[4]、DPFS（Distributed Parallel File System）[5]、Inter-Mezzo[6]、NFS（Network File System）[7,8]、GFS（Global File System）[9]、xFS（Serverless File System）[10]和 OPIOM（Off-Processor I/O with Myrinet）[11]、COSMOS[12]、Galley[13]、GPFS（General Parallel File System）[14]、Clusterfile[15]、DAFS（Direct Access File System）[16]、Lustre[17]

和 PVFS [18]等。

上述并行文件系统的设计和研究主要集中在以下几个方面：

（1）单一系统映像

并行文件系统建立在本地文件系统上，在集群中处于单一系统映像层[2]。在用户看来，实现了单一系统映像的文件系统应当和单机文件系统没什么区别，其文件的组织是单一的树型结构，文件全局存取，不需要用户去直接关心数据的物理存储和访问方面的问题。

（2）数据放置策略

并行文件系统为了给用户提供并行服务，以达到聚合 I/O 带宽的目的，通常将文件数据分片存储在集群的众多存储节点上。数据的分片放置策略有连续存放（Round-Robin）、基于 hash 存放和 RAID 等。

（3）缓存与预取

由于网络带宽、磁盘与内存速度之间的巨大的差距，根据文件数据访问的局部性原理，利用缓存机制来提高系统的 I/O 性能。然而，缓存并不能提高第一次被访问的数据块的访问速度，这就需要使用预取策略从磁盘中预取。

（4）数据访问负载均衡

集群服务器数据分布在很多存储节点上，用户在访问数据时有可能造成存储节点的负载不平衡，有些存储节点访问过热，而有些存储节点却很少被访问。数据访问负载均衡还体现在对某一个文件，或某一文件中的具体数据块的访问上。解决数据访问负载不平衡的方法有数据的动态迁移、热点数据备份等。

（5）元数据管理策略

元数据是描述文件属性的特征信息，是文件系统中最重要的数据信息，访问非常频繁。元数据以何种方式在文件系统中存放，采用集中式元数据管理还是分布式元数据管理都对并行文件系统性能有着至关重要的影响。

（6）高可用和容错策略

文件系统需要通过自检操作来恢复因异常关机断电造成的文件数据丢失，保证文件系统的前后一致性。文件系统的高可用和容错策略主要体现在对元数据和数据的高可用和容错方面。最常用的高可用策略包括数据备份存储、日志管理以及设立检查点等。

1.3 并行文件系统缓存

文件系统缓存是一种尽量减少文件系统访问磁盘次数的机制，用提高文件系统性能的方法来弥补磁盘性能的不足。缓存设计遵循了计算机系统设计的最广泛的准则——加快经常性事件[19]，基本思想是将应用程序所需的文件块保存在内存中。这样，被请求的文件块只有在第一次被访问时需要从磁盘中读取，以后的访问就可以直接从缓存中读取，读操作的性能因此得到了大幅度提高。此外，由于被应用程序修改后的数据块只需保存在缓存中，由系统在适当时候再将它们写回磁盘，写操作的性能也因此得到了提高。

与本地文件系统把数据存储在本地磁盘上不同，并行文件系统把数据最终存储在远端磁盘上，并行文件系统的数据检索层次也因此与本地文件系统不同，计算节点缓存、存储节点缓存和存储节点磁盘构成了并行文件系统的三级检索和存储层次。计算节点把访问过的数据块缓存在本地内存中，力求减少应用程序访问存储节点的次数，将网络通信开销降至最低。集群中众多的存储节点都有自己的主存，并行文件系统在存储节点上实现缓存，可以进一步推迟对低速磁盘的访问。

并行文件系统同样也可以把文件的元数据缓存起来，实现元数据的高速访问。不过，在没有非常严格的一致性保证策略情况下，多数并行文件系统都不在计算节点上实现元数据缓存。

随着高速网络的出现，网络访问速率大大超过了磁盘存取速率，许多分布式、并行文件系统都在客户端即计算节点建立了合作式缓存。设计合作式缓存主要基于以下几点考虑[20]：尽可能推迟对存储节点的访问，以免存储节点访问过热；利用众多计算节点的内存资源比建立一个巨大的存储节点缓存在性价比方面更为合算；合作式缓存大小可以伸缩，使用起来更为灵活、方便。但是，合作式缓存在保证数据的一致性，数据的置换策略，以及避免缓存访问热点等方面显得更为复杂，这几个方面都是合作式缓存开销的主要来源。

最著名的合作式缓存算法是 N-Chance 转发算法[21]，它最早实现在 xFS（Serverless Network Filesystem）[10]中。N-Chance 算法把每个客户节点的缓存分成可以动态改变大小的本地部分和全局部分。本地部分主要用于缓存本地节点应用程序所需的文件块，全局部分则主要用于缓存其他节点所需的但不能缓存在本地的文件块。N-Chance

还设立了一个管理节点来管理块的位置信息和维护缓存一致性，客户节点对缓存块的修改都需要通知管理节点，由管理节点通知其它客户节点抛弃其持有被修改数据块。N-Chance 本地部分缓存使用的置换算法是 LRU，抛弃在缓存中有多个备份的数据块，只有一个备份的数据块则转发到全局缓存中保存。

GMS（Global Memory Service）[22]是另一种使用很广泛的缓存合作算法。GMS 也使用了一个管理节点来管理数据块的位置信息，这与 N-Chance 一样，不过 GMS 中管理节点并不负责维护缓存的一致性。使用 GMS 算法时，每一个缓存数据块都有一个属性标志该数据块在缓存中是否唯一，数据块在缓存中的备份数目由管理节点管理。GMS 直接抛弃在缓存中超过一个备份的数据块，一旦数据块的拷贝减少到一个，管理节点便通知持有该数据块的客户节点将该数据块标志为唯一，该数据块在置换时便会被转发到全局缓存中。

N-Chance 和 GMS 都极大提高了缓存命中率，不过使用这两种算法时，本地缓存和全局缓存中都可能存在一个数据块的多个副本，一致性维护代价很高，并且管理节点负担过重，是系统的一个单一失效点。针对这个缺点，Hint-based 算法[23]取消了管理节点，把管理节点的功能分布到各客户节点。使用 Hint-based 算法时，把每一个刚从磁盘中取来的数据块都标志成 mastcopy。置换时，Hint-based 直接抛弃不是 mastcopy 的数据块，mastcopy 则被转发到其它客户节点的全局缓存中，并由持有 mastcopy 的客户节点负责维护该数据块的一致性。不过 Hint-based 需要在每个客户节点上维护一张 mastcopy 位置表，当 mastcopy 较多时，维护代价也是很高的。

目前，国内外对文件系统缓存做了很多研究工作，这些工作既包括文件系统原始设计时所作的缓存研究，也包括基于已有文件系统所作的缓存实现。大多数都是针对客户端即计算节点数据缓存的研究，对存储节点缓存和元数据缓存所作的工作较少。

Coda[5]文件系统在客户端专门设立了一个叫做“Venus”的进程来管理磁盘缓存。“Venus”功能上实际上相当于一个缓存代理服务器：在 VFS 中不能处理的文件系统请求都被送往位于应用层的“Venus”，由“Venus”检索所请求的文件是否在本地中有缓存，一旦找不到所需文件，Coda 便由“Venus”通过远程过程调用 RPC 把整个文件从存储服务器拷贝过来存储在本地的一个文件中。Coda 文件系统在本地磁盘中缓存的实际上是整个文件，其弊端是很严重的：首先，“Venus”进程处理一切请求，必然造成很长的等待队列；其次，Coda 文件系统用文件保存远程数据，其速度显然不如把数据直接保存在内存中；另外，Coda 以文件级粒度维护数据一致性，对文件每

一部分的修改都要通知服务器，引起的通信量是非常大的。

Lustre[17]是由“集群文件系统公司”开发的一个开放源代码并行文件系统，同时实现了元数据缓存和计算节点数据缓存。Lustre 基于一种叫做“基于对象的磁盘（object-based disk）”的特殊存储设备设计，这种存储介质不能使用现有的接口访问，而要由内核提供专门的接口来访问。Lustre 的元数据缓存只缓存新写入的元数据，通过缓存汇聚来减少通信和提高并行性。不过 Lustre 没有提供特别的措施保证元数据在写回到元数据服务器前的可靠性，一旦丢失，对文件系统的影响很严重。Lustre 的真实文件系统 I/O 都由 OSTs（Object Storage Targets）负责。Lustre 每一个节点上都有一个 OSTs，负责各节点之间的通信连接。因此，Lustre 的合作式缓存建立在 OSTs 上，维护缓存一致性的锁机制也由 OSTs 管理。可以看出 Lustre 是基于特殊存储设备的并行文件系统，Lustre 的技术包括缓存策略等都被限制在这种特殊设备上，通用性不强。

COSMOS[24-25]是由中科院计算所设计的用于曙光系列超级服务器的并行文件系统，建立在 IBM AIX 操作系统平台上。COSMOS 是在 xFS 的基础上研究的，它的很多缓存策略都跟 xFS 相同。不过，COSMOS 在 xFS 上增加了一些不同策略，例如 COSMOS 在 xFS 的块粒度级一致性协议的基础上增加了文件级粒度。COSMOS 称这种一致性维护协议为双粒度协议，具体采用何种粒度由一个专门的节点管理。当客户节点打开的文件没有被写共享时，管理节点给这个文件分配一个文件级粒度的一致性管理权限，当出现其它用户打开了该文件时，管理节点负责通知第一个打开该文件的用户，该文件的文件级粒度缓存一致性管理权限就被取消了，取而代之的是块粒度的缓存一致性管理权限。

这些文件系统原始设计的缓存多趋向于对缓存策略的研究，对缓存的实现方法并没有多少提及。Vilayannur 等基于 PVFS 专门介绍了在内核实现计算节点缓存的方法[26]。具体实现方法是：所有 PVFS 应用程序发出的系统调用都被定向到一个内核动态可加载模块中，该模块负责检索本地缓存数据，并承担本地缓存缺失时与存储节点通信的任务。这种内核级的实现方法对应用程序完全透明。不过，在内核中实现缓存要占据大量的内核存储空间，并且要占用大量的内核处理时间，有可能影响其它任务的处理。另外，内核级缓存还会引起大量的用户态和内核态进程切换和内存拷贝，开销很大。

国内外对位于存储节点的文件系统缓存所作的研究并不多。Yuanyuan Zhou 在她的博士论文[27]中专门论述了存储节点缓存的访问模式。她的研究结论是：相对于计算

节点缓存，存储节点缓存的访问具有更长的时间间隔；存储节点缓存中各数据块的访问频率大小分布很不均匀。在研究了访问模式之后，Zhou 还设计了一种专门用于存储节点缓存的置换算法 MQ（Multi-Queue），主要思想是：用多队列管理缓存数据块，访问频率高的数据块放入到高队列中，访问频率低的数据块则放入到更低级的队列中，并在访问过程中对数据块所属队列进行动态调整，被置换的数据块的块号和访问频率也由一个 FIFO 队列管理，以便再次利用。从她的仿真结果中可以看出 MQ 可以获得很高的缓存命中率。

1.4 并行虚拟文件系统概述

PVFS[18,28-30]是由美国 Clemson 大学开发的一个开放源码并行文件系统，旨在为 Linux 集群服务器提供一个高性能并行文件系统。目前 PVFS 已经有了两个版本：PVFS1 和 PVFS2，本文下文出现的 PVFS 都是指 PVFS1。PVFS 被设计成客户/服务器结构，任何客户节点上的创建、删除和读写文件等请求都必须送到服务器方进行处理。PVFS 采用了元数据和文件数据相分离的结构，系统中存在管理元数据的元数据服务器和存储文件数据的存储节点。元数据服务器仅仅负责管理元数据，如超级块、i 节点等；文件数据 I/O 则由存储节点处理。PVFS 系统中可以配置多个存储节点，但只允许存在一个元数据服务器。PVFS 的系统结构如图 1.2 所示。

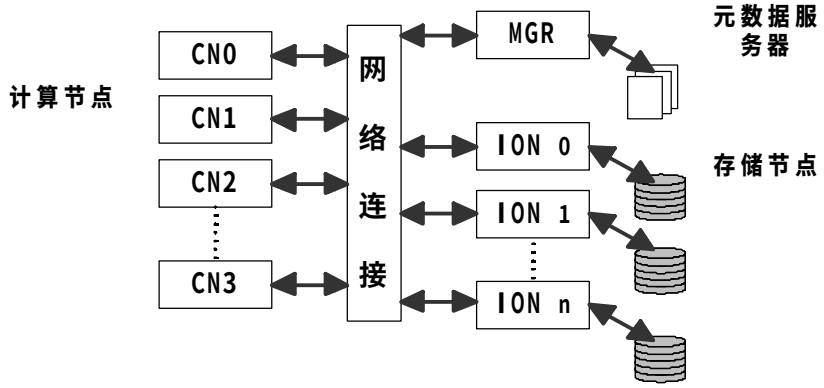
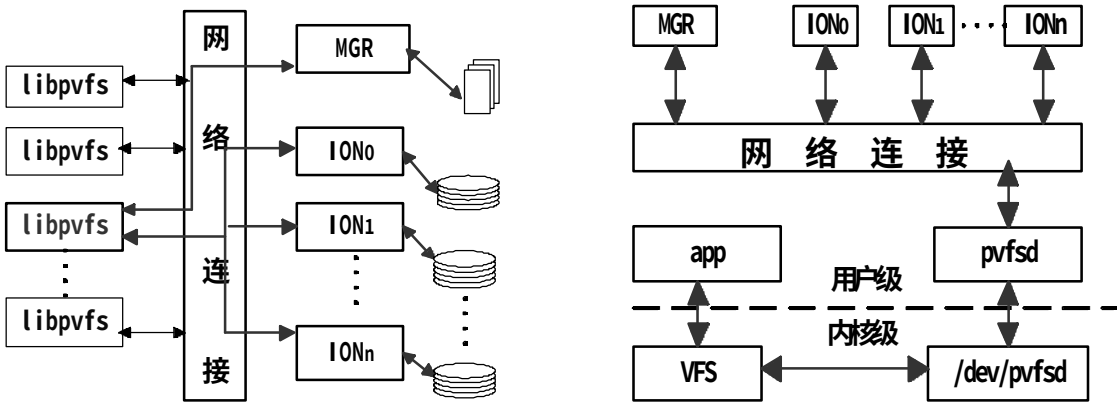


图 1.2 PVFS 系统结构图

PVFS 客户端、元数据服务器端和存储节点端都是用户级实现。在元数据服务器端和存储节点端都运行一个守护进程，分别是 mgr 和 iod，用于不断接收和处理来自客户端的访问请求。在客户端，PVFS 为用户提供了大量的接口函数，这些接口函数完全遵守 UNIX 文件系统语义，用户可以利用这些接口函数编写自己的 PVFS 函数。

这些接口函数实际上是一些网络通信函数——用户的一个文件操作函数通过这些接口函数转化成网络通信请求直接发给服务器，在服务器得到处理之后又通过这些函数转化成文件操作结果，返回给用户。这些网络操作函数都是在用户级实现，不经由 Linux VFS。同时 PVFS 实现了一个核心模块以提供与 Linux VFS 之间的接口，从而支持把 PVFS 文件系统挂载在 Linux 的某个目录之下。PVFS 核心模块通过队列与客户端通信。由于 PVFS 客户端是在用户级实现的，PVFS 建立了一个字符设备文件（/dev/pvfsd），客户端通过读写该字符设备来进入核心。PVFS 元数据服务器、数据服务器与客户端的通信请求都是首先写入到该设备文件，在设备文件中形成一个队列，核心模块直接对该队列进行存取。PVFS 两种实现如图 1.3 所示。



(1) PVFS 库函数实现　　(2) PVFS VFS 接口实现

图 1.3　PVFS 两种实现机制

PVFS 以文件形式存储元数据，由一台元数据服务器管理。PVFS 采用分片的形式存储数据，数据分片均匀连续的存储在设有编号的存储服务器 ION 上，这样可以实现文件数据的并行存取，为用户提供聚合带宽。

PVFS 只是对文件系统功能作了简单实现，还远远达不到一个高性能并行文件系统所需具备的高可靠性、高可扩展性和高吞吐率的要求。总结起来，限制 PVFS 性能提高的主要原因有：

（1）采用集中式元数据管理。元数据服务器是文件系统的单一失效点，并且在文件处理负荷较重时会成为文件系统的性能瓶颈，限制了系统的可扩展性。

（2）元数据以文件形式存放，每一次元数据操作都要访问元数据文件，限制了元数据的访问速度。

（3）PVFS 中不存在任何形式的缓存。应用程序每一次文件访问都要直接访问元

数据服务器和存储服务器上的低速磁盘，并带来大量的网络通信开销。

（4）没有任何故障恢复机制，一旦故障出现无法挽回损失。

（5）没有数据负载均衡机制，可能出现部分存储节点访问过热，部分存储节点过冷的现象。

不过，PVFS 的设计充分体现了并行文件系统的特征，并且源代码开放，仍不失为一个很好的原型系统。目前国内外有很多研究人员基于 PVFS 设计自己的文件系统。比较有名的如韩国釜山国立大学基于 PVFS 设计的并行多媒体文件系统 PMFS（Parallel Multimedia File System）[31]等。

1.5 系统主要工作

针对 PVFS 存在的问题，我们设计了高性能并行文件系统 HPVFS（High performance Parallel Virtual File System）。HPVFS 是对 PVFS 的功能扩充，在其上实施了多项高可靠性、高扩展性和高速文件访问等策略，主要工作包括：

（1）针对 PVFS 使用单台元数据服务器集中管理元数据，HPVFS 采用主从元数据服务器互为热备份的方式管理元数据、并添加了元数据故障恢复机制，避免元数据服务器成为系统单一失效点。

（2）针对 PVFS 用文件形式存放元数据，元数据访问低速缺点，HPVFS 采用寄生式元数据管理方式，把元数据存放在文件的 inode 中。

（3）PVFS 没有任何缓存。HPVFS 则添加了元数据缓存、计算节点数据缓存和存储节点数据缓存，为系统构建多级数据检索和存储层次，实现元数据和数据的高速访问。

（4）PVFS 没有数据访问负载均衡机制，HPVFS 采用数据迁移与副本相结合的方式，分散访问热点。

1.6 本文工作与组织

HPVFS 的缓存系统是本文的主要研究内容。元数据缓存主要工作是分析 PVFS 元数据访问操作函数，并在计算节点实现一种无一致性问题的元数据缓存。计算节点缓存的主要工作是分析共享内存的优点，采用共享内存实现缓存。存储节点缓存的主要工作则是对一种存储节点缓存系统置换算法的研究。

本文后续章节是这样组织的：

第二章是 HPVFS 框架及其缓存系统。介绍了 HPVFS 的总体结构、具体功能模块划分及其文件访问流程，最后简要介绍了 HPVFS 缓存系统的构成。

第三章是元数据缓存。分析了实现元数据缓存的原因，提出了元数据缓存实现的关键技术和注意事项，之后对 PVFS 文件系统的元数据访问网络协议进行了详细分析，并阐述了通过在 PVFS 客户端实现元数据缓存来简化元数据访问网络协议的方法，最后对 PVFS 元数据缓存进行性能测试和结果分析。

第四章是计算节点缓存。叙述了采用共享内存实现计算节点缓存的优点和实现方法，并对实现实例进行性能测试和结果分析。

第五章是存储节点缓存。首先阐述了存储节点缓存的访问模式，以及实现存储节点缓存的技术；然后对并行文件系统中存储节点为计算节点提供并行服务的特性进行了分析，发现存储节点间的缓存应该进行协同置换，提出了一个协同置换算法，并对该算法进行了仿真实验；最后是对协同置换算法中访问组的统计方法的初步研究和设想。

第六章是结束语。概括了本文研究成果和创新点，并说明研究中存在的问题和未来的研究方向。

最后是本文引用的参考文献和致谢。

2　系统框架及其缓存系统

在 PVFS 基础平台上，HPVFS 实施了多种高可用、高可靠和文件高速访问策略，对 PVFS 进行了多项功能扩充。本章介绍 HPVFS 的总体结构、模块结构和文件访问流程及其缓存系统的构成。

2.1　系统总体结构

HPVFS 旨在为集群服务器提供高性能的并行文件系统，实现对分布在集群各节点的所有文件、设备和网络资源的全局访问，并为用户呈现出单一系统映像。HPVFS 以并行虚拟文件系统 PVFS 为基础平台，主要研究内容包括：元数据的高速访问和高可用性设计；多级缓存机制；数据高可用和容错策略；存储服务器访问负载均衡；故障恢复技术等。

HPVFS 把集群内部节点从功能上划分为计算节点、元数据服务器和存储节点，这些节点通过集群内部网络进行互联。HPVFS 的总体结构如图 2.1 所示。

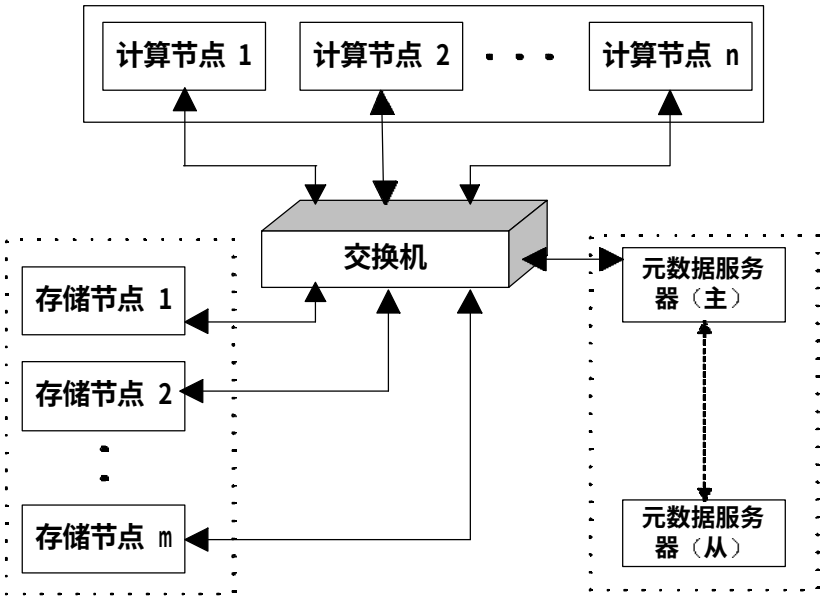


图 2.1　HPVFS 总体结构图

HPVFS 具有多个计算节点，它们是并行文件系统的入口。计算节点为用户提供本地文件系统视图，并为用户提供统一的编程接口。用户任务到达集群服务器后，通

过一定的调度策略分配到具体的计算节点处理。计算节点负责把任务转化成集群内部文件系统请求，从元数据服务器和存储节点获取所需数据，并把处理结果返还给用户。

HPVFS 把文件数据分片存储在很多存储节点上，为用户提供聚合带宽。系统初启时，每个存储节点都运行一个守护进程 iod 负责接收来自计算节点的数据访问请求。HPVFS 把文件分成很多子文件，每一个子文件都由存储在同一个存储节点的该文件的所有数据分片构成，这样所有数据分片就可以使用本地文件系统调用直接访问。

元数据服务器负责存储和管理描述文件特征信息的元数据。计算节点读写任何文件之前都需要先从元数据服务器获得该文件的访问权限、大小及数据分布情况等信息。HPVFS 采用主从元数据服务器形式管理元数据，主元数据服务器失效之后，由从元数据服务器立即接管，避免出现单一失效点。主从元数据服务器互为热备份，容易实现故障迅速恢复。

2.2 系统模块结构

HPVFS 的模块结构如图 2.2 所示。除去 PVFS 的原有守护进程模块和函数接口模块的实现，HPVFS 的主要模块有：双元数据服务器高可用模块、数据副本模块、数据负载均衡模块、寄生式元数据管理模块和缓存模块等。

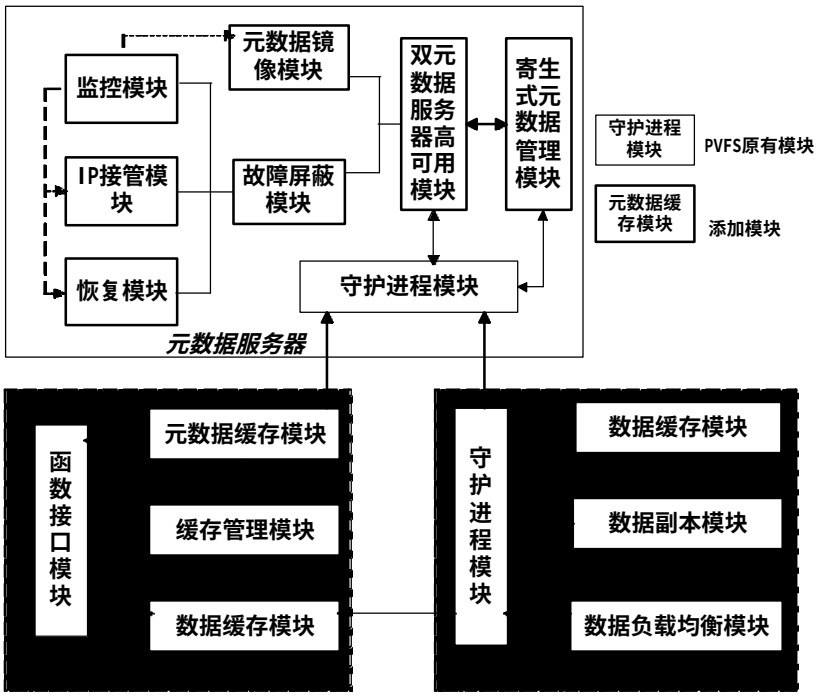


图 2.2 HPVFS 模块结构图

双元数据服务器高可用模块实现主从元数据服务器管理，由两个二级模块构成：元数据镜像模块和故障恢复模块。元数据镜像模块负责将主服务器上的元数据所在目录文件实时镜像到从服务器一个镜像目录中，当主服务器失效后从服务器将用镜像目录中的元数据提供服务。元数据镜像模块又由若干个三级模块组成，包括主、从服务器上的处理模块和网络模块。故障屏蔽模块实现完全自动和透明的接管，保证为用户提供连续服务。故障屏蔽模块由监控模块、IP 接管模块和恢复模块等三个三级模块组成。监控模块是实现故障屏蔽的前提和基础，也是整个系统运行的控制中心所在。监控模块采取 heartbeat 的检测机制，在活动状态下主、从服务器互相发心跳信息来给对方，以此来判断对方的状态。主元数据服务器失效后，从元数据服务器通过 IP 接管模块接管主元数据服务器的 IP，从而实现对元数据处理功能的接管。主服务器失效之后通过故障恢复模块进行恢复，主从元数据管理服务器互为镜像，主服务器故障恢复所需的信息由从元数据服务器提供。

HPVFS 在一定时间间隔内统计访问热点，并根据统计结果把访问热点在整个系统中转移，这部分工作由数据负载均衡模块完成。对于访问实在太频繁的数据块，无论如何迁移都会产生新的热点，HPVFS 采用对这些数据块进行备份的方式来解决这个问题，这由数据副本模块完成。

HPVFS 的缓存模块包括元数据缓存模块，计算节点缓存模块，计算节点缓存管理模块和存储节点缓存模块等构成。HPVFS 的原型系统 PVFS 上没有实现任何形式的缓存，每一次文件访问都需要访问元数据服务器和存储节点的低速磁盘。利用主存、网络和磁盘三者之间的速度差异，在计算节点和存储节点上分别建立缓存，尽可能推迟访问磁盘。

2.3 缓存系统构成

缓存系统是本文的主要研究内容。HPVFS 的缓存系统由元数据缓存、计算节点缓存和存储节点缓存三部分组成。元数据缓存只缓存元数据。计算节点缓存和存储节点缓存则只缓存文件的真实数据，它们和存储节点磁盘一起构成了 HPVFS 的三级检索和存储层次。HPVFS 缓存系统结构如图 2.3 所示。

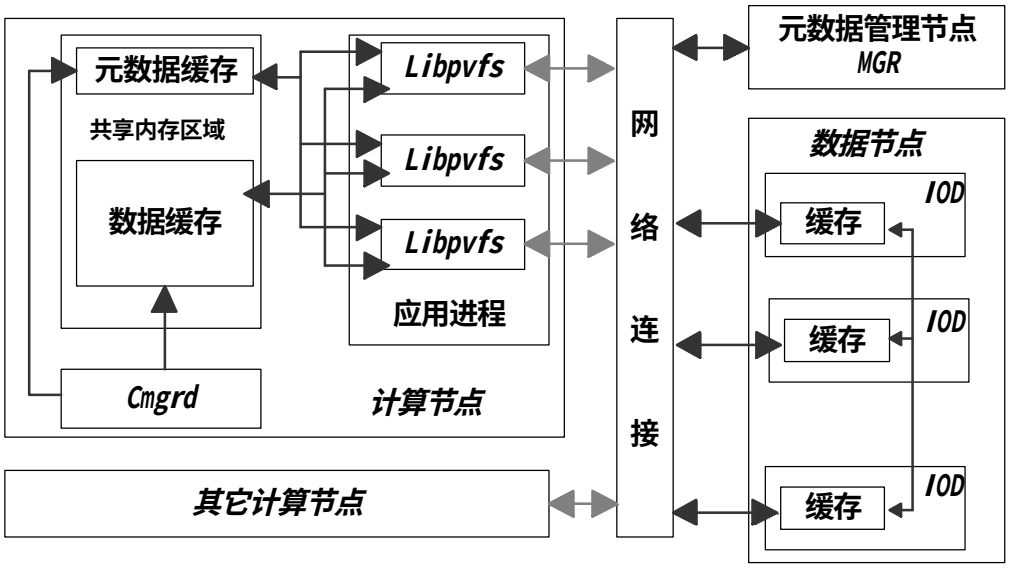


图 2.3　HPVFS **缓存系统总体结构图**

元数据是文件系统中最为重要的数据。在文件系统中对元数据的操作是非常频繁的。PVFS **每一次元数据操作都至少需要访问两次元数据服务器，带来了大量的网络通信开销。元数据缓存旨在简化这种烦杂的元数据操作网络协议，实现元数据的高速访问。采用共享内存方式在计算节点上实现元数据缓存，元数据缓存的申请和管理都由一个一直运行的守护进程“**Cmgrd**”负责。**HPVFS **新元数据在写到元数据缓存的同时，直接写回元数据服务器，确保元数据的正确性和同步更新。另外，由于元数据的至关重要性，在建立元数据缓存之后，**HPVFS **每一次元数据操作仍然需要访问一次元数据管理服务器，及时纠正元数据异常，保证元数据的一致性。**

计算节点是建立并行文件系统缓存最直观和效果最明显的位置。HPVFS **的计算节点缓存也采用共享内存实现，同样也是由守护进程“**Cmgrd**”负责向操作系统申请和管理。共享内存缓存方便计算节点上的应用进程直接读写，可以减少进程间通信的内存拷贝次数，实现进程间数据的真正共享。**

存储节点缓存是并行文件系统二级数据检索和存储层次。并行文件系统存储节点缓存存在一个潜在的严重问题：当多个数据节点同时向计算节点上的一个进程提供并行访问服务时，存在一个潜在的严重问题：可能出现个别数据节点缓存块缺失，而大部分数据节点缓存都命中了所需数据块的情况，影响文件系统数据的并行访问。基于此，HPVFS **各数据节点在置换时进行协同，对经常出现被同时访问的数据块施行整体置换策略，以此降低单一存储节点缓存单独缺失的概率。**

2.4 小结

本章介绍了 HPVFS 的总体结构、模块结构、工作流程和缓存系统的结构。HPVFS 以 PVFS 为基础平台，在其上实现了多种高可用、高可扩展和文件高速访问等策略。与 PVFS 一样，HPVFS 的节点功能上可以分为计算节点、元数据服务器节点和存储节点。除了 PVFS 原有的模块，HPVFS 的主要设计了五大模块。其中双元数据服务器高可用模块实现的主从元数据服务器是提高 HPVFS 的可用性和可扩展性的主要手段；缓存模块实现的三种缓存是提高文件系统高速读写的主要方法。在用户看来，访问 HPVFS 文件就像访问本地文件系统文件一样，HPVFS 内部文件访问流程也是非常清晰的。

HPVFS 的缓存系统是本文的主要研究内容，主要包括三种类型的缓存：位于计算节点的元数据缓存、位于计算节点的数据缓存和存储节点缓存。元数据缓存实现元数据高速访问；计算节点数据缓存、存储节点缓存和存储节点磁盘构成了 HPVFS 的三级数据检索和存储层次。元数据缓存和计算节点数据缓存都采用共享内存技术实现，它们都由计算节点上一个守护进程“Cmgrd”申请和管理。存储节点缓存主要关注一个协同置换算法的实现。本章对这种三种缓存没有展开讨论，将在后续几章中详细论述。

3 元数据缓存

元数据是文件系统中最为重要的特征信息，应用进程对元数据的操作非常频繁。实现高速元数据访问是提高文件系统性能的关键因素之一。在计算节点上建立元数据缓存可以有效减少访问元数据服务器带来的大量网络通信开销。本章论述元数据缓存的关键技术，分析 PVFS 的元数据操作网络访问协议，提出了 PVFS 平台上实现元数据缓存的方法，此方法保证了元数据的一致性。

3.1 元数据缓存的必要性

3.1.1 元数据及其存储方式

“文件”是指按一定的组织形式存储在物理介质上的信息，实际上包含两方面的信息：存储的数据本身以及有关该文件的组织和管理信息，前者是文件的数据，后者就是该文件的元数据。因此，从操作对象看，文件系统的操作可以划分为两种：数据操作和元数据操作。数据操作是对真实用户数据的读、写和执行，元数据操作则是对文件系统中文件属性结构的浏览和修改。元数据被称为“数据的数据”。在文件系统中，元数据是指用来描述一个文件系统文件的特征数据。一个文件的元数据包括其创建和修改时间、权限许可、属主、组、大小、文件逻辑存储和物理存储等信息。在 UNIX 文件系统中，整个文件系统就像一棵树，文件系统的根目录对应与树的根，每个目录都是树中的一个节点，文件则对应树的叶子。文件的逻辑存储由文件所在的目录指定。文件的其它元数据信息大部分都存储在文件的索引节点中，每个文件的索引节点都唯一的对应着一个节点号。文件系统的超级块中存储了整个文件系统的元数据信息，它指定了文件系统的挂接根目录，记录了文件系统的使用信息。

集群是由很多通过高速网络互联在一起的计算机的集合。集群中很多节点都配置了磁盘，为了利用集群这种结构特性，提高系统访问并行性，集群文件系统通常将文件数据分片存储在这些配备有磁盘的存储节点上。用户在读写一个文件时需要知道文件数据被分片存储在哪些存储节点的磁盘上，以及这个存储节点在集群中的位置。因此，集群文件系统的文件元数据信息比本地文件系统元数据信息要多一些，文件的元数据还包括文件数据分片存储的位置。

并行文件系统的元数据存储有两种方式：一种是将元数据直接存储在磁盘上，由文件系统提供访问函数接口，这种存储方式的好处是访问起来极为高速，但是需要提供及其复杂的访问接口，且需要改动内核，实现起来比较麻烦；另一种是直接以本地文件的形式存储在元数据服务器上，实现起来比较简单，但是每一次访问元数据都需要转化为对元数据文件内容的访问，在访问速度上比起第一种要差很多。

3.1.2 元数据密集型应用

服务器处理的任务从数据访问特性上可以分为两种：数据访问密集型应用和元数据访问密集型应用。数据访问密集型应用，例如科学计算、FTP 文件传输等，通常在访问一次元数据之后，便不断进行大量数据的存取和计算。随着网络技术的快速发展，出现了越来越多的元数据访问密集型应用。例如网页浏览、文件搜索、邮件服务和电子商务等。这些应用的共同特点是服务器需要打开大量的小文件，并且这些小文件经常被不同的用户重复访问。例如，网页浏览服务会涉及到巨大量的文件——门户网站要提供数十万乃至数百万的文件给用户浏览。这些文件不仅相对较小（从 1k 到几百 k 不等），并且经常发生变化[32]。在任意给定时间，都有很多文件被创建、读、写或删除，所有服务器上处理的基本上是这类任务。又如 UNIX 文件系统中，“ls”是运行的最多的命令，用户不断通过“ls”命令获取目录中的文件或目录的属性信息，“ls”命令就是不断访问文件的元数据来获取文件属性的。另一运行得非常多的 UNIX 文件系统命令是“cd”，也需要访问目录的元数据信息。在这些元数据密集型应用中，提高元数据访问速度是提高文件系统性能的关键。

3.1.3 元数据缓存及必要性

对于分布式环境中的并行文件系统，元数据管理是管理数据的关键。首先，元数据是最重要的系统数据。应用程序在访问一个文件的数据之前，首先需要定位数据的存储位置。在并行文件系统中，用户在访问一个文件时，首先向元数据服务器发出请求，元数据服务器把要访问的文件的元数据应答给用户。用户在元数据中获取文件所在的存储节点位置信息，从而可以和这些存储节点建立起连接。因此从系统稳定性和可靠性的角度来说，文件元数据的正确性和可靠性是至关重要的。其次，元数据文件的访问速度影响着整个文件系统的访问性能。在并行文件系统中，对元数据的访问是非常频繁的。用户对文件的任何操作都需要事先访问文件的元数据，对文件所做出的任何改动都必须记录在文件的元数据中。因此，提高元数据访问的速度，对提升文件

系统的性能非常重要。

提高元数据访问的速度有两种方法。一种是提高元数据的管理效率。例如改进元数据的存储方式，变集中式元数据管理方式为分布式元数据管理方式，即在系统中设立多个互为镜像的元数据服务器。这样，这些服务器既可以同时向不同的用户提供服务，以避免单个元数据服务器造成的访问瓶颈；同时由于这些元数据服务器互为镜像，也可以提高元数据的可靠性。另一种方式是简化元数据的访问网络协议。在并行文件系统中，元数据存放在元数据服务器上，由元数据服务器统一管理，用户需要与元数据服务器进行网络通信来获取元数据。因此，过于复杂的元数据访问网络协议会严重影响元数据的访问速度，从而成为系统的又一个性能瓶颈。

在计算节点建立元数据缓存是简化元数据访问协议的方法之一。文件的元数据本身占用的空间并不大，通常一个文件的元数据所占用的空间大概只有一百个字节到两百个字节，因此，在计算节点建立元数据缓存并不会占用太大的内存空间。当应用程序访问一个文件时，首先试图从本地元数据缓存中获取被访问文件的元数据，一旦命中，将无需与元数据服务器进行网络通信，即使没有命中，由于访问本地内存的速度远高于网络带宽，对于系统性能也不会造成太大的影响。

在本地文件系统中，元数据的访问同样频繁。Linux 本地文件系统的文件元数据是寄生在 inode 中的。要访问一个文件时，一定要通过文件的 inode 才能知道文件类型、文件组织形式、文件大小、文件数据存储位置以及其下层的驱动程序等必要的信息[33]。为了提高文件系统的时空性能，Linux 为已分配的 inode 构造了缓存。文件 inode 信息实际上就是文件的元数据，因此 VFS inode 缓存可以称为 Linux 本地文件系统的元数据缓存。

3.2 元数据缓存关键技术

元数据缓存作为文件系统缓存的一部分，与数据缓存在很多实现策略上是一致的。这些关键技术包括：置换算法、预取算法等等。但是，基于元数据在文件系统中的重要地位，元数据缓存在一些关键技术上所采取的策略和要求与一般意义上的数据缓存有所不同。

3.2.1 完整性和一致性维护

文件系统为了提高自身的可靠性和可用性，大都实现了文件系统故障后重新恢复

的功能。最著名的文件系统恢复工具“fsck”就是通过检测文件系统中所有的元数据，以试图修复文件系统的超级块，从而达到修复文件系统的目的。日志文件系统也是通过记录元数据操作信息来修复系统的。例如 JFS（Journal File System）便是使用原子事务记录文件系统元数据上执行的操作（即原子事务）信息。如果发生系统故障，可通过重放日志并对适当的事务应用日志记录信息，使文件系统恢复到故障前的一致状态。一旦元数据出现错误，系统故障后将无法恢复到正确的状态。因此，在元数据操作期间，系统必须确保元数据完整正确地写入到元数据服务器的磁盘上。在客户端添加元数据缓存之后，应用程序修改元数据应该立即写回到元数据服务器的磁盘上。这与一般意义上的数据缓存有很大区别——数据缓存为了提高文件系统的写性能，避免“小块写”问题，通常使用缓存聚集要写的文件数据，直到产生大的块再执行写操作、保存奇偶校验信息，以推迟写磁盘。

元数据的一致性要求是非常严格的。由于元数据记录了文件的修改时间、权限属性以及文件数据的分布情况，因此元数据的正确性是确保文件操作正确性的前提。在客户端添加元数据缓存之后，必须时刻保证元数据的一致性，保证缓存中元数据得以同步更新。如何保证元数据缓存的一致性是实现元数据缓存的关键难点。

3.2.2 置换与查找策略

作为缓存的关键技术之一，缓存数据块大小的取定会对系统性能造成很大的影响。缓存块大小应该适中，数据块太大会造成主存空间的浪费，在缓存块中造成很多空洞；数据块太小则会造成一次文件读写的数据需要很多缓存数据块才能容纳，增加了缓存系统查找开销，如果出现读写所需的数据在缓存中部分命中的情况，还会引起大量的“小块”读写的问题。对于数据缓存来说，缓存块大小的选定取决于具体的应用环境。对于小文件访问居多，例如网页访问、邮件服务器等应用来说，缓存块不能过大。而对于大文件访问居多，例如科学计算等应用来说，缓存数据块的选定就应该大一些。PVFS 缺省状态下，文件数据分片大小是 64KB，适合于大文件的访问。Linux 本地文件系统 EXT3 为了适应不同应用的需求，提供了从 512B 到 32KB 的七种数据缓冲区长度。

元数据作为文件的描述信息，虽然包括很多文件属性，但是一个文件的元数据本身所占用的空间却是很小的。元数据缓存块不能太大，对于一个给定的文件系统来说，元数据缓存块的大小通常是固定的，与描述一个文件所需的元数据大小相当。本地文件系统的元数据信息寄生在文件的 inode 中。在 Linux 中，一个 VFS inode 所占用的

空间大约只有为 300 个字节，不过所描述的文件属性却是非常丰富的。PVFS 采用文件的形式存放元数据，一个元数据文件所占用的空间是 112 个字节。因此，相对于存储容量日益增大的节点主存来说，一个元数据缓存块大小可以说微不足道。一个容量为 1MB 左右的元数据缓存可以容纳上万块缓存数据块，一个元数据缓存块对应着一个文件的元数据，从而可以缓存上万个文件的元数据。David kotz 等人在 Intel iPsc/860、Intel Paragon、IBM SP-2 上收集了 156 个小时的文件系统 traces 才发现有 63779 个文件打开操作[34]。因此，对于一般应用来说，如果把元数据缓存容量配置更大一些的话，元数据缓存块几乎没有被置换出去的机会。对于元数据缓存来说，置换算法的好坏不是影响缓存性能的最关键因素，不过缓存的查找算法却显得非常重要。在 EXT3 文件系统中，就利用 inode 所指向的超级块的指针值和该 inode 中 i_ino 值作为 hash 参数来建立 inode 缓存哈希表来加快查找速度。

3.3 元数据缓存实现

在计算节点上建立元数据缓存可以简化 PVFS 非常繁琐的元数据操作网络访问协议，实现元数据的高速访问。

3.3.1 元数据访问函数分析

PVFS 设计时充分考虑了提高文件系统的并行性。PVFS 在系统初始化时，为所有的存储节点连续分配了唯一的存储节点号。文件数据分片采用轮转（Round-Robin）的方式存储在这些节点号连续的存储节点上。在同一存储节点上的同一个文件的所有数据分片构成了该文件的一个物理子文件，如图 3.1 所示。数据分片信息作为文件元数据存储在元数据服务器上。PVFS1 采用的是集中式元数据管理，系统中唯一的一台元数据服务器承担着所有计算节点的访问任务。

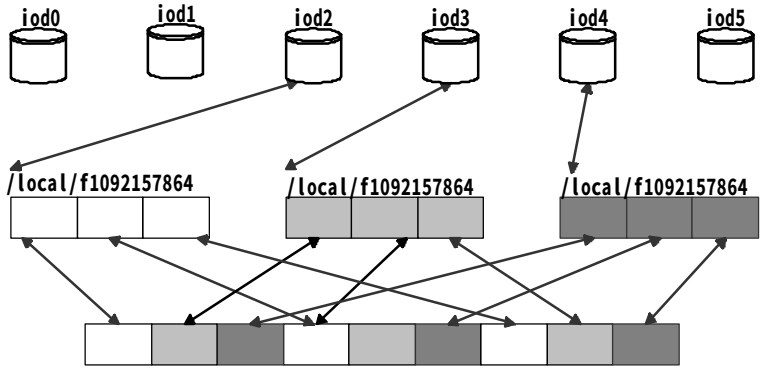


图 3.1 PVFS 文件分片

PVFS 有关元数据的访问操作网络协议非常繁琐，用户在获取文件的元数据时，至少需要访问两次以上的元数据服务器，给元数据服务器带来了极大的负担，同时访问延时也是很大的。以文件打开操作“open”为例。在客户端看，PVFS 应用进程在打开一个文件可分解为三步，如图 3.2 所示。第一步，应用进程对将要访问的文件做路径检查。如果是本地文件，则调用本地文件操作函数对文件进行处理；否则，PVFS 把要访问的文件的路径转化为并行文件系统的真实全局路径名，随后与元数据服务器建立连接，发出查询请求“detect”以确认所访问的文件是否存在和所访问的文件是否为目录，这是 PVFS 在整个文件打开操作的第一次元数据访问。第二步，对于被确认为存在的 PVFS 文件，PVFS 才真正的向元数据服务器发出打开请求“open”。元数据服务器在接收到客户打开文件请求后，先通知相关存储节点打开存储在其上的物理子文件，在得到确认后，把元数据应答给用户，这是 PVFS 文件打开操作的第二次元数据访问。第三步，客户收到元数据服务器的应答之后，发出一个空请求“NOOP”，与各存储节点建立连接。至此，文件的打开操作才算完成。可以看出，PVFS 的文件打开操作总共涉及到两次客户端与元数据服务器的网络通信，一次与各相关存储节点的网络通信，还包括一次元数据服务器与各存储节点的网络通信，所引起的网络通信开销是很大的，严重限制了系统性能的提高。

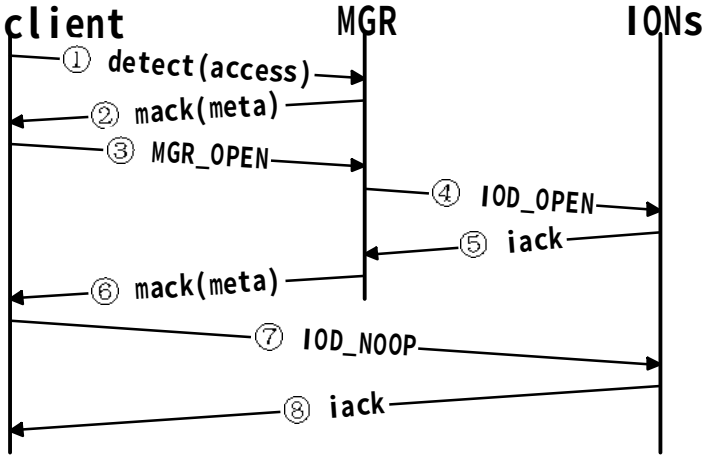


图 3.2　PVFS 文件打开操作网络协议图

详细分析 PVFS 文件打开操作过程，可以发现 PVFS 文件打开操作网络协议是可以简化的。PVFS 文件打开操作第一次访问元数据服务器相当于本地文件系统的“stat”操作。由于“stat”操作需要从存储节点收集文件状态信息，会造成大量的网络通信量，因此 PVFS 改为向元数据服务器发出“access”请求，元数据服务器把除了文件

大小的其它的状态信息应答给客户。PVFS 的网络通信协议完全能够简化。

3.3.2 元数据访问网络协议改进

有两种方法可以简化 PVFS 文件打开操作的网络通信协议。一种是把第一次元数据访问和第二次元数据访问合并，即在第一次就直接向元数据服务器发出“open”请求。元数据服务器接到客户端的请求后，先在服务器对相应元数据文件做本地“stat”操作，以确保要打开的文件存在，然后再通知各对应存储节点打开物理子文件。这种方法需要较大地改动 PVFS 的源代码，且所作的改动仅仅针对提高“open”操作速度。另一种方法便是在客户端构建元数据缓存。

在客户端构建元数据缓存不仅仅限于提高“open”操作的性能。仔细分析 PVFS 其它涉及到元数据访问的操作，可以发现它们都无一例外需要访问两次元数据服务器，这些操作包括：chmod、chown、lstat、mkdir、stat、rename、rmdir、statfs、truncate、unlink、utime、access、stat64 和 truncate64 等，几乎囊括了所有的文件系统操作语义。实现元数据缓存之后，上述操作的性能都可以得到很大的提升。改进后的 PVFS 文件打开操作网络协议如图 3.3 所示。

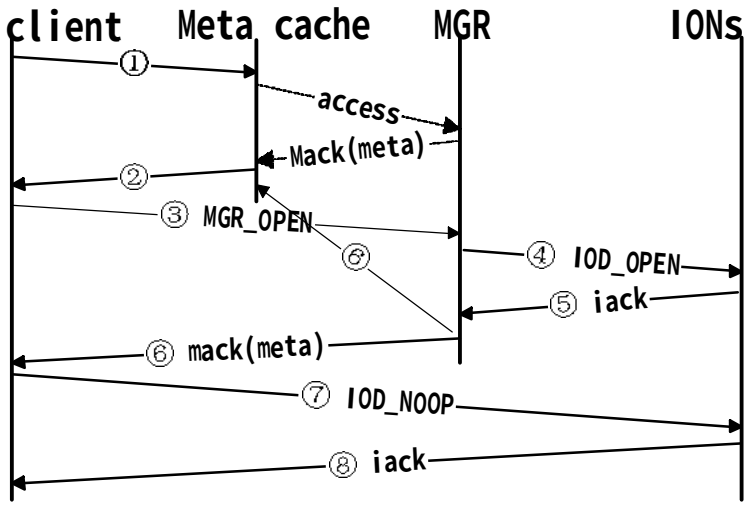


图 3.3 改进后的 PVFS 文件打开网络协议图

改进后，文件打开操作步骤如下：

（1）将文件状态信息应答给应用进程。

（2）在获得文件的状态信息之后，发送文件打开请求到元数据服务器。

（3）元数据服务器发送子文件打开命令到存储节点。

（4）存储节点打开子文件之后，把应答信息反馈到元数据服务器。

（5）元数据服务器把元数据发给应用进程，同时也对元数据缓存进行更新。

（6）应用进程发送空请求到存储节点，与存储节点建立连接。

（7）存储节点应答应用进程，表明连接已经建立，文件打开完毕。

从文件打开操作工作流程可以看出，元数据缓存在命中之后可以省略了一次访问元数据服务器。其它元数据相关操作也是如此，这极大的减少了元数据访问的网络开销。

由上文分析可见，PVFS 中几乎所有元数据第一次访问都是调用函数“pvfs_detect(4)”来检查文件并与元数据服务器通信的。因此，在 PVFS 中客户端可以直接修改“pvfs_detect(4)”函数，在函数中加入元数据查找、插入等函数来实现元数据缓存。

下面是实现元数据缓存部分代码：

```
int pvfs_detect(char *fn, int64_t *fs_ino_p)  /* fn是文件路径名，文件元数据
                                                 文件的inode号由fs_ino_p返回*/
{
   if(search_fstab(fn) ==0)                    /* 表示是本地文件 */
          return 0;
   if((meta = meta_find(fn)) != NULL))         /* 找到元数据*/
   {
                fs_ino_p = meta->fs_ino;
                return 1;
    }
    send_mreq_saddr(mreq, mack);               /*发送请求到元数据服务器*/
    meta_add(mack);                            /* 将应答元数据加入到缓存队列中 */
    fs_ino_p = mack->ack.access.meta.fs_ino;
    return 1;
}
```

3.3.3 一致性问题

文件的元数据在访问过程中也是经常改变的，由此产生了元数据缓存一致性问题。元数据的重要性决定了元数据缓存的一致性要求比一般数据缓存的一致性更为严格。图 3.3 所示的元数据缓存结构可以确保元数据一致性。

文件元数据一致性是由图 3.3 中用户第二次访问元数据服务器来保证的。从图 3.3 中可以看出，用户第二次访问元数据服务器时，并没有考虑直接从本地的元数据缓存中读取。用户打开一个文件或对文件做其它元数据相关操作时，仍然至少需要访问一次元数据服务器。这是因为用户打开一个文件必须通知元数据服务器，时刻保证文件元数据的同步更新；另外，存储节点只接收来自元数据服务器的物理子文件打开请求。这确保了文件元数据的一致性。假设某一用户在缓存中持有一个文件的元数据期间，

元数据因其它用户而发生了改变，导致了元数据出现不一致情况，能够迅速检测出来并纠正。仍以“open”为例，第一次元数据访问即“access”请求的目的只是为了检查目标文件是否存在和检查目标文件系统是否已经挂载，这种检查也可以出现在第二次元数据访问即真正打开请求（mgr_open）时的异常检查中，无非是推迟检查而已。元数据缓存并不服务第二次元数据访问，相反，第二次元数据访问所获得的元数据还将更新元数据缓存，由此解决已经出现的不一致问题。

3.3.4 性能测试和结果分析

（1）测试环境

为了验证元数据缓存的效果，先后测试了 PVFS 添加元数据缓存前后的各种文件系统操作性能。测试硬件环境如表 3.1 所示。测试采用的 PVFS 版本是 1.5.4，配置如下：存储节点 8 个，其中有一个存储节点同时也是元数据服务器。由于存储节点和计算节点互不影响，这些存储节点同时也是计算节点。适应于 PVFS 元数据文件的大小，元数据缓存块的大小定为 112 个字节。元数据缓存总大小定为 1MB。相对于节点内存大小 256MB 来说，1MB 大小的元数据缓存对节点运行其它程序不会构成丝毫的影响，但是却可以容纳 10280 个元数据文件，对于多数应用来说，元数据缓存块几乎没有被置换出去的机会。在这种情况下，对于 PVFS 元数据缓存来说如何快速查找数据块是影响缓存性能的关键。PVFS 元数据缓存所采用的置换算法是基于 hash 的 LFU-DA（Least Frequently Used of Dynamic Aging），该算法的目的就是为了加快查找速度。本章没有对置换算法做具体研究，有关 LFUDA 算法将在第四章中阐述。

表 3.1 测试硬件环境

CPU	Intel PⅢ 550MHz
主存	256MB
网卡	100MBps D-LINK DFE-530X
交换机	100MBps D-LINK DESS-3226
操作系统	Redhat Linux 7.1 （**版本号**：2.4.7-10）

（2）测试方法和结果分析

测试所使用的 benchmark 是在 Lmbench 的基础上修改的。Lmbench 的目的是测试整个计算机系统，由一系列小的 benchmark 组成，其中有相当一部分是用于测试文件系统的性能。但是 Lmbench 所使用的是标准文件 I/O 库函数，而所要测试的是 PVFS 自带的库函数的元数据操作用时，因此需要对 Lmbench 进行修改，把所有的标准文件

I/O 库函数都改为相对应的 PVFS 库函数。另外，Lmbench 的 lat_fs 仅仅是用来测试文件创建、删除小文件的吞吐率，远远达不到所要测试的操作的种类，因此需要在 lat_fs 中添加了大量其它如 chmod、chown 等元数据相关操作测试。

先关注元数据缓存对系统性能的影响。首先测试没有元数据缓存情况下各元数据相关操作耗费的时间，然后测试添加了元数据缓存后各操作在缓存命中和未命中时的有关操作耗费的时间。测试结果如表 3.2 所示。可以看出在添加元数据缓存之后，在缓存没有命中的情况下，各操作耗费的时间比没有添加元数据缓存时的相同操作耗费的时间要稍多一些，这是由添加元数据缓存之后数据块查找开销引起的，但差距是非常微小的，这表明即使要查找的元数据不在缓存中，对系统性能也不会有多少影响。一旦元数据缓存命中，文件操作用时便呈倍数的减少。文件操作性能的提高尤以“open”操作和“creat”操作为甚，提高倍数达到 13.5 倍。性能提高最小的是“truncate”操作。该操作是对文件大小进行重新设定，需要与存储节点通信，改进效果不明显，只有不到 10%。除去“open”操作和“create”操作，其它操作用时平均提高 102%。

表 3.2　PVFS 在添加元数据缓存前后有关操作耗时（ms）

耗时		Open	chmod	chown	lstat	mkdir	Rmdir
添加缓存前		38.00	2.45	1.96	2.95	3.07	2.56
添加缓存后	未命中	38.42	2.50	2.16	3.06	2.87	2.61
	命中	2.85	1.92	1.28	2.29	error	1.11
耗时		Creat	access	rename	stat	unlink	Truncate
添加缓存前		34.56	2.21	2.48	3.13	3.04	2.97
添加缓存后	未命中	35.60	2.29	2.76	3.04	3.23	3.03
	命中	2.93	1.59	1.48	2.39	1.33	2.85

本文 3.3.4 节曾提及，元数据缓存不一致情况可以在用户第二次访问元数据服务器的时候得以解决。为了验证元数据缓存对文件系统异常检查时间的影响，本文还测试了添加元数据缓存前后元数据异常的检查时间。测试可以分为两步：第一步，没有元数据缓存的情况下，访问一个不存在的文件或目录，记录报错的时间；第二步，在文件系统中添加元数据缓存，先访问一个存在的文件或目录，使得该文件或目录缓存在元数据缓存中，然后在其它计算节点把该文件删除，人为造成元数据不一致情况，再第二次访问该文件或目录，记录报错时间。测试结果如表 3.3 所示。

表 3.3　添加元数据缓存前后相关操作异常检查耗时（ms）

一致性检查	open	chmod	chown	lstat	rmdir
添加前耗时	1.33	1.03	1.85	1.26	1.18
添加后耗时	1.82	1.72	2.63	1.74	1.45
一致性检查	truncate	access	rename	stat	unlink
添加前耗时	1.25	1.19	1.07	1.21	1.23
添加后耗时	1.74	1.64	1.92	1.75	1.81

分析表 3.3 的测试结果可以发现在添加元数据缓存之后，一致性异常检查比正常检查多出 0.5ms 左右。这一结果大概是正常检查时间的 40%。这对于大多数用户来说是可以接受的。元数据出现严重异常，例如文件被删除，或文件权限被其它用户更改时，程序的异常处理通常都是退出程序的执行或者把错误记录下来，向用户报告。对于用户来说，推迟 0.5ms 知道程序出现问题是可以忍受的。对于元数据出现的一般不一致情况，例如文件大小被改变、文件修改时间被改变时，元数据不一致情况将在第二次访问元数据服务器时得以修复。

3.4　小结

元数据是描述文件数据的特征信息。提高元数据的访问速度对于并行文件系统来说非常重要，特别是对于运行元数据密集型应用的集群系统来说至关重要。由于元数据在文件系统中的重要性，元数据缓存的一致性要求非常严格，不能出现丝毫影响文件访问正确性的不一致问题。PVFS 文件系统元数据访问操作网络协议非常繁琐。通常，一次 PVFS 元数据有关操作至少需要访问两次元数据服务器，网络开销非常大。客户端过多的访问元数据服务器，也给元数据服务器带来了沉重的负担，有可能造成元数据服务器成为系统性能瓶颈。在 PVFS 计算节点上实现元数据缓存可以简化元数据访问网络协议。由于 PVFS 元数据访问操作的特点决定了在 PVFS 客户端建立元数据缓存不必担心一致性问题。经过测试，在添加元数据缓存之后，缓存命中时，各有关操作速度有了成倍甚至数十倍的提升。即使没有命中也不会对操作速度造成很大影响。由于文件元数据占用空间非常小，通常添加 1MB 以上大小的元数据缓存后，被访问过的文件的元数据几乎没有机会被置换出去。测试还表明，对元数据不一致异常推迟检查不会影响系统性能。

4 计算节点缓存

计算节点是实现文件系统缓存的最直接、效果最明显的位置。计算节点缓存的命中直接消除了计算节点到存储节点间的网络消耗，对系统性能的提高是最明显的。本章主要讨论采用共享内存方式实现计算节点缓存的技术和相关算法，并进行性能测试与分析。

4.1 共享内存缓存

共享内存是一种进程间通信方式，具有效率高、速度快的特点。采用共享内存实现计算节点缓存可以减少进程间的内存拷贝，并且实现节点内的数据共享。

4.1.1 共享内存

共享内存，顾名思义就是两个或更多个进程可以访问同一块内存空间，任何进程对这块空间中的某个单元内容的改变可以为其它进程所“看”到[35]。图 4.1 描述了共享内存的含义。共享内存通过物理内存映射建立。进程间使用共享内存通信时，由一个进程向系统申请一块物理内存区域，其他进程通过一个共同的键值取得这块内存区域的标识号；在得到一个共享内存区的标识号以后，每个进程可以将此共享内存区映射到自己的虚存空间，然后就可以像访问自己内存空间一样去访问这块共享内存区域。AT&T Unix 系统 V 提供了相关的系统调用建立共享内存，它们是：shmget()，获得或创建一个 IPC 共享内存区域，并返回相应的标识符；shmat()，建立共享内存区的映射；shmdt()，撤销共享内存区的映射；shmctl()：对共享内存区控制、管理和删除。

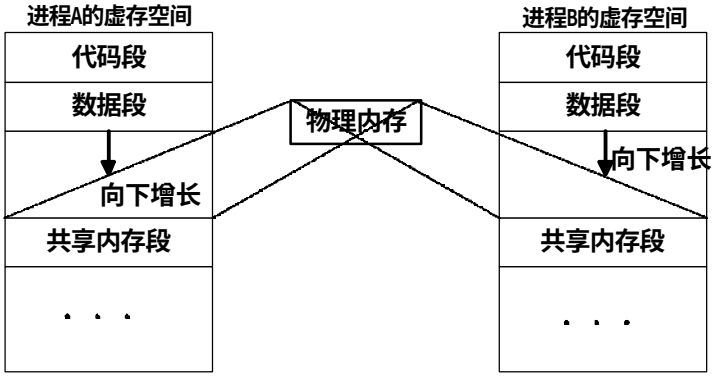


图 4.1 共享内存的含义

由于共享内存区域由多个进程之间共享，必然需要某种同步机制，以避免几个进程同时修改一个内存单元。实现共享内存区域进程间的互斥的方法有很多，一般的方法是对要修改的区域加互斥锁，或者设立一个信号量，进程在修改共享内存区域前首先查看信号量值。

共享内存可以说是最有效率的进程间通信方式，也是最快的 IPC 方式，这是由两个原因决定的。第一，各通信进程可以直接读写内存，不需要向管理进程申请。管道和消息队列等通信方式，需要在内核和用户空间进行四次的数据拷贝，共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。第二，共享内存中的内容往往是在解除内存映射之后才写回文件。进程之间在共享内存时，并不总是读写少量的数据就解除映射，有新的通信时，再重新建立共享内存区域。一般的做法是保持共享区域，直到通信完成为止，这样，数据内容一直保存在共享内存中，并没有写回文件。可见，比起其它通信方式，共享内存通信效率是非常高的。

4.1.2 缓存实现方法

Linux 可以在两个地方实现文件系统缓存：内核空间和用户空间。内核空间中的缓存好处是所有的用户进程和内核线程都可以直接访问，能够实现同一节点内的缓存合作，并且在内核空间中实现缓存对应用层进程完全透明，应用层程序不需要任何修改。然而，在内核中实现缓存受到很多限制。首先，内核存储空间由所有的进程共享，在内核空间实现缓存会占用大量的存储空间，并且经常读写缓存会占据大量的内核态时间，影响系统处理其它任务。其次，在内核空间实现缓存开销太大。这包括两种开销：进程切换开销和应用层和内核层之间的内存拷贝开销。用户进程每一次读写缓存都需要进入到内核中，会引起大量的进程在内核态和用户态之间相互切换，每一次读写缓存都是一次应用层和内核层间的内存拷贝，开销太大。另外，在内核空间中实现缓存，实现起来也较为复杂困难。一种在并行文件系统中实现内核空间缓存的方法是使用 Netfilter。由于内核空间缓存对应用程序是透明的，应用程序在读写文件数据时仍然像往常一样向存储节点发出读写请求，这种读写请求实际上就是网络通信请求。在 Linux 中，所有的网络通信都需要经过内核层，因此可以在内核层运行一个截获数据包的线程。这包括两次数据包截获：截获上层应用进程的文件读写请求数据包和截获数据服务器的读写应答数据包。首先，应用层发来的所有文件读写请求数据包都被截获下来；然后在内核缓存中查找所需的文件数据，如果找到所需数据，便在内核中

伪装来自存储节点的数据包，把数据应答给应用进程，如果没有找到所需数据，仍然向服务器发送请求；服务器发来的读写应答数据包在送往上层应用进程的同时也要被截获下来，以更新缓存数据。

应用层文件系统缓存比较实现起来比较简单，可以减少进程在用户态和内核态的切换次数，但是不能做到对应用进程透明，需要对应用程序进行修改，通常的实现方法有两种。第一种方法很简单，每个应用进程都在自己的内存空间中设立缓存，但是这种缓存只能为自己服务，各进程之间明显缺乏合作和协调，同时对内存空间也是一种浪费。第二种方法是运行一个用户守护进程，管理本机的文件系统缓存。当一个进程需要读写数据时，便向这个守护进程发出读写通信请求，由守护进程把数据发给应用进程。第二种方法不能避免内存的多次拷贝，当应用进程读写请求比较频繁时，会引起很长的请求等待队列。

4.1.3 共享内存缓存模型

HPVFS 文件系统缓存使用共享内存实现，模型如图 4.2 所示。用共享内存实现文件系统缓存的好处是可以避免进程间内存的过多拷贝，实现同一节点内进程间数据共享。

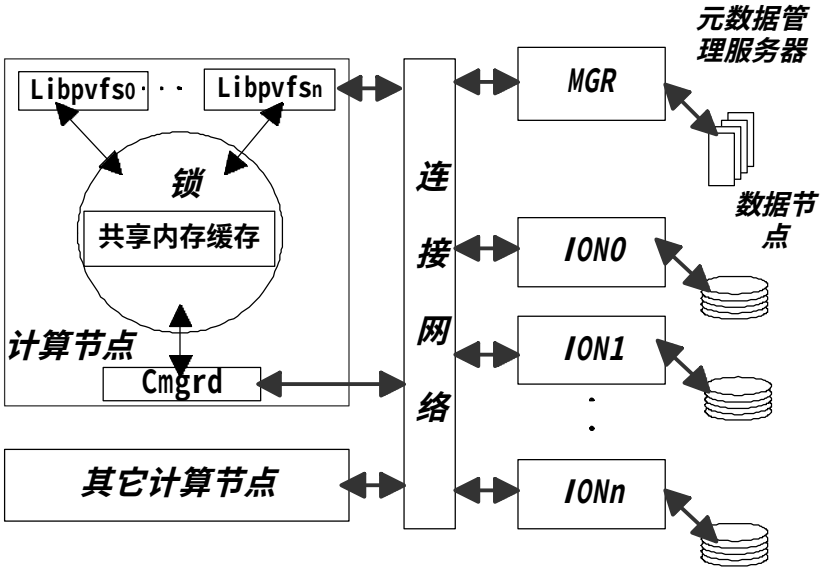


图 4.2 HPVFS 共享内存缓存模型

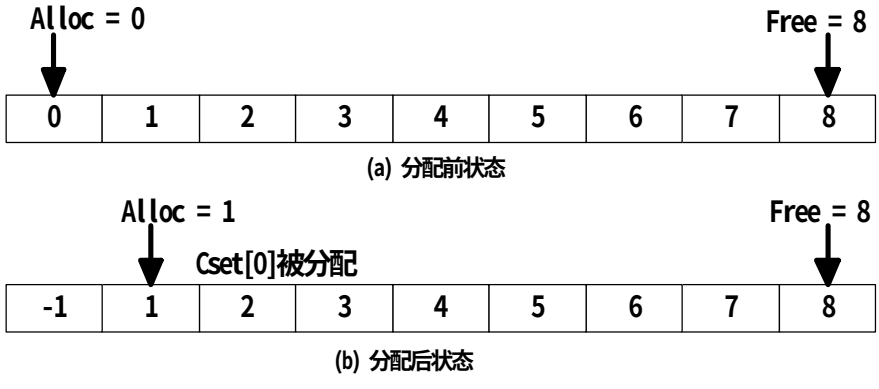
系统首先运行一个用户守护进程“Cmgrd”，该进程负责共享内存空间的申请和管理，但并不负责处理其它进程的读写请求。应用进程将共享内存映射到自己的虚拟内存空间，在需要读写数据时，应用进程就可以直接访问共享内存缓存了，从而避免了

进程间的大量通信和内存的多次拷贝。在 Linux 中，在文件中“/proc/sys/kernel/shmmax”规定了系统一次申请共享内存的最大字节数，文件“/proc/sys/kernel/shmmni”中规定了一次申请共享内存的最小字节数。因此，利用共享内存实现文件系统缓存需要注意系统对共享内存空间的限制，不过规定值可以通过手工进行修改。

4.1.4 块的分配与释放

用共享内存实现文件系统缓存，缓存空间是在系统启动时预先静态分配的。计算节点在系统初启时运行一个守护进程“Cmgrd”，申请缓存所需的共享内存空间。在文件系统缓存中，数据块的申请和释放是非常频繁的，需要采取一定策略加快速度。“Cmgrd”在初启时一次性的向系统申请一大块共享内存空间，按照缓存块大小把该空间分成很多物理上连续小数据块，用一个块数组 Cset[n]（n 表示缓存的块数）管理。系统另外维护一个与 Cset 同样大小的整形数组 free_table，指示当前可分配块数组项。free_table[i]的值代表可分配的块数组项的下标：free_table[i]的值为-1 时，表示其所代表的块已分配；free_table[i]的值为 m 时，表示当分配到 free_table[i]时，其所代表的数据块 Cset[m]将被分配。系统初始时，所有缓存块都是空闲的，free_table[i]的值为 i。为了方便管理，采用两个数值指示块数组中可供分配空闲项：“Alloc”指示第一个可供分配数组项，“Free”指示最后一个空闲数组项，也代表块回收时将被插入到 free_table 数组中的位置，如果“Alloc”与“Free”相等，表示已无空闲缓存空间，需要通过置换来腾出空间。

图 4.3(a)说明了第一次分配数据块前的状态。free_table 数组的各项值与下标相等：“Alloc”为 0，free_table[Alloc]的值为 0，指明第一个可分配的数据块为 Cset[0]；“Free”为 8，指明最后一个可分配的数据块为 Cset[8]。Cset[0] 被分配后的状态如图 4.3(b)所示：free_table[0]值为-1，“Alloc”向前推进、值变为 1，表示下一次可分配块为 cset[free_table[1]]。



(a) 分配前状态

(b) 分配后状态

图 4.3 块的分配

图 4.4(a)描述了数据块第一次被释放前的状态。在此之前，前 4 块已被分配；图 4.4(b)描述了数据块被释放后的状态：在 Cset[2]所指示的数据块被释放后，按照“Free”的指示，将 free_table[0]的值修改为 2，表示最后一个可分配块为 Cset[2]同时“Free”循环向前推进、值为 0。

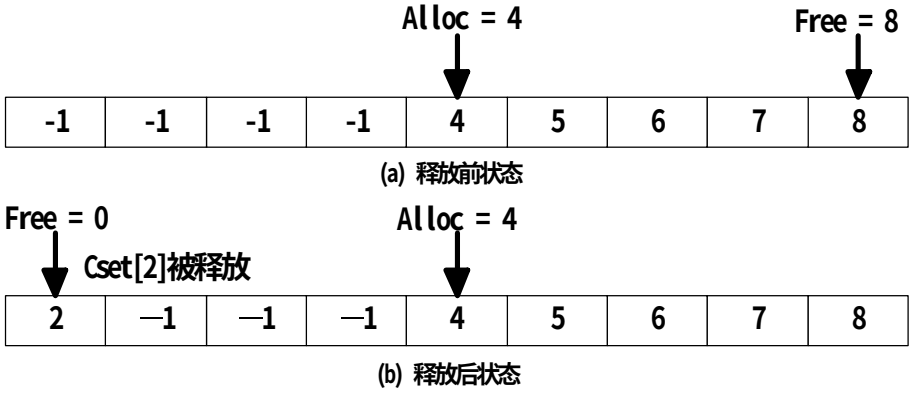


图 4.4　块的释放

4.2　置换算法

置换的目的是为了让缓存空间资源达到最佳利用状态。置换策略决定缓存中哪一数据块将被移出缓存，它是缓存系统性能指标之一。可以说，研究缓存系统，必然要先研究置换算法。为了提高缓存块命中率，国内外对置换算法做了大量的研究和实现。

LFU-DA（LFU with Dynamic Aging）是 LFU 的改进算法[36]。LFU 算法只考虑到访问了很多次的数据块有很大机会在不久被重新访问，却忽略了这些数据块的最近一次访问时间。事实上，系统经常在处理完一个任务之后，转向处理其它的任务。那些在前一次任务中访问非常频繁的数据块有可能再也没有机会在新的任务中被重新访问，但是这些占据了大量缓存空间的数据块的访问次数却是很大的，采用 LFU 算法没有可能把这些数据块置换出去，极大的降低了缓存空间的利用率，这种现象称为缓存污染。LFU-DA 算法对缓存数据块加入了“年龄”属性，并且在访问过程中不断加以调整。LFU-DA 算法并不是每次都调整所有的数据块的“年龄”，而只是在访问的时候动态调整。每访问一次缓存中的数据块，或者向缓存中加入新的数据块，都要为数据块计算“年龄”。数据块的“年龄”由数据块的访问次数和上次被置换的块的“年龄”共同求得，它的计算公式是：$Ki=Ci * Fi + L$，其中，Ki 是新的“年龄”；Fi 是数据块被访问的次数，第一次加入到缓存中的数据块的 Fi 值为 1；Ci 是一个可变参数，

可以根据应用情况做具体调整；*L* 是一个全局变量，当每次有数据块被置换出去时，都把 L 的值调整为被置换块的“年龄”。LFU-DA 每一次都把“年龄”最小的数据块置换出去。LFU-DA 常常被用作 Web 代理缓存服务器的置换算法，可以使缓存服务器获得很高的字节命中率。实际上，由于 LUF-DA 同时考虑了“访问时间”和“访问频率”两个方面，用作文件系统缓存置换算法也能获得很好的性能。当文件系统缓存容量很大时，LFU-DA 算法有一个弊端：缓存队列过长，导致应用程序在缓存中查找数据耗时太长。

本文对 LFU-DA 算法进行了改进，具体的做法是：将大的缓存队列分成几个小队列，根据数据块的块号和数据块所属的文件的 inode 号求 hash 值，然后根据 hash 值把数据块插入到对应的队列中。每一个 hash 队列都是 LFU-DA 队列。这样就可以加快缓存的查找速度，证明方法很简单：假设 LFU-DA 算法的平均查找时间是 t，系统缓存根据 hash 值被分成 k 个队列，那么改进后算法的平均查找时间就是 t/k。改进后的算法可称为基于 hash 的 LFU-DA（hash-based LFU with Dynamic Aging）算法，实现伪码如下：

```
Struct Cache_block
{
      int64_t f_ino;                                  /*文件inode号 */
      int64_t blk_no;                                 /* 块号 */
      char data[BLKSZ];                               /*数据 */
      struct Cache_block* next_block;                 /* 指向下一块*/
      int key;                                        /*块的“年龄”*/
      int frequency;                                  /*访问频率 */
};

void access_Lfuda(struct Cache_block* cb, int L)
/* L是最近一次被置换块的key值 */
{
      int queue_value;
      cb->frequency ++;
      cb->key = cb->frequency + L;
      queue_value = hashfunc(cb->f_ino, cb->blk_no); /*求队列号*/
      insert(queue_value, cb);       /* 把块cb插入到相应的队列中*/
}
```

4.3 锁粒度

由于应用进程可以直接访问共享内存缓存，可能出现几个进程在同一时刻访问同一缓存单元的情况，这就需要加锁来实现进程间的访问互斥。锁粒度的大小对缓存性能影响很大，粒度过大会造成太大的一致性和互斥开销，粒度太小则会带来太大的锁

管理开销。缓存锁粒度一般分为块粒度和文件级粒度两种。使用块粒度锁策略的好处是不影响其它块的访问，但是需要为每个数据块设立一个锁，锁管理开销显然是巨大的。文件级锁粒度是为每个文件设立一个锁，对文件中任何一个数据块的访问都要对整个文件加锁，必然影响到其它进程的访问。

本文缓存的锁粒度是队列级的。如上节所述，缓存队列根据 hash 值被分成很多个小队列，为每个小队列都设立了一个读写互斥锁，这是由于：本文的缓存不与其它计算节点共享，加锁只是为了保证节点内进程间的互斥。使用队列级锁粒度策略易于管理，守护进程“Cmgrd”在文件系统初启时负责预先为每个队列静态申请一个锁，之后所有锁也是由“Cmgrd”管理和撤销。本文锁采用信号量实现，信号量的数目与队列数目对应，信号量的值代表一个临界区即其对应队列的读写资源是否可用。

4.4 性能测试与分析

测试实验所用的硬件环境如表 4.1 所示。测试程序是 PVFS 自带的一个程序 pvfs-test。该程序测试了调用 PVFS 库函数的并行读、写速度，采用 MPI-IO 实现，包含了大量的文件数据的读写：创建一个对所有测试进程可见的新文件，测试进程并行地读写数据该文件的各个区域。读写带宽由该程序取所有进程读写操作时间的最大值来计算。在所有的测试中，每个计算节点从文件大小为 2N MB 的连续区域读、写数据，其中 N 是存储节点的个数，例如如果使用 8 个存储节点，则每个测试进程的读写大小为 16MB。本文改变 PVFS 存储节点的个数，计算节点的个数和读写数据的总大小，进行分别测试。测试时，PVFS 数据分片大小缺省设为 16KB，适应于分片大小，缓存块的大小也为 16KB，缓存的总容量设定为 8MB。

表 4.1 测试硬件环境

CPU	Intel PⅢ 550MHz
主存	256MB
网卡	100MBps D-LINK DFE-530X
交换机	100MBps D-LINK DESS-3226
操作系统	Redhat Linux 7.1 （版本号：2.4.7-10）

图 4.5 是并行读性能。从图中可以看出使用 8 个、4 个、2 个存储节点时并行读带宽都比 PVFS 原型系统有了较大幅度的提高。使用 8 个存储节点时，读带宽最大可达 98.58MB/s，PVFS 原型系统对应值为 65.4MB/s；使用 4 个存储节点时，读带宽最大值

为 78.36MB/s，PVFS 原型系统对应值为 46.78MB/s；使用 2 个存储节点时，读带宽最大值为 45.73MB/s，PVFS 原型系统对应值为 26.21MB/s。从图中可以看出，各条曲线代表的数值都是先呈线性的增长，到达到一定的值后，便平稳下来，变化不是太剧烈。由于条件有限，本文测试没有使用更多的计算节点，不过从图中仍然可以看出，读带宽在计算节点达到一定数量之后，有下降的迹象，这是由于在存储节点一定的情况下，计算节点太多会造成在存储节点上有较长的等待队列，影响了响应速度，同时网络带宽限制也是一个原因。从图中还可以看出实现缓存之后的曲线图在达到稳定值后，曲线并不像没有实现缓存之前的性能曲线图那么平整，出现了比较多的起伏现象，这大概是由于缓存中数据块的置换给系统带来了一定的开销。

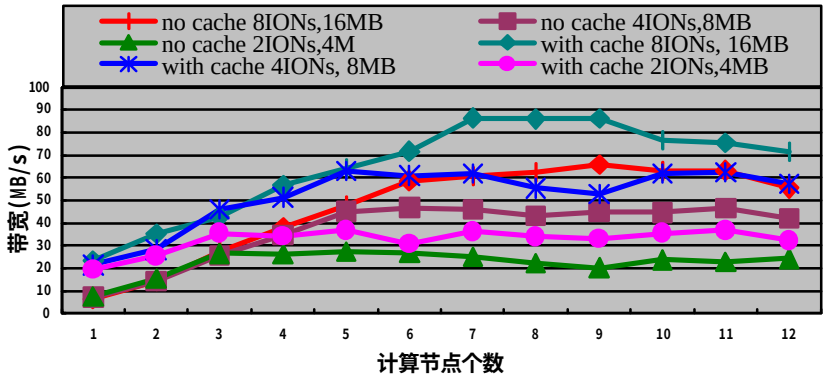


图 4.5　并行读性能

图 4.6 是并行写性能。与读性能相比，缓存的并行写性能稍差，这是由于测试程序先进行写数据引起的。使用 8 个存储节点时，写带宽最大可达 86.33MB/s，PVFS 原型系统对应值为 65.92MB/s；使用 4 个存储节点时，写带宽最大值为 62.91MB/s，PVFS 原型系统对应值为 46.61MB/s；使用 2 个存储节点时，写带宽最大值为 36.78MB/s，PVFS 原型系统对应值为 27.45MB/s。从图中可以看出缓存写性能在达到稳定之后，曲线出现了更多的波动，这是由于写带来了更多的块置换的缘故。

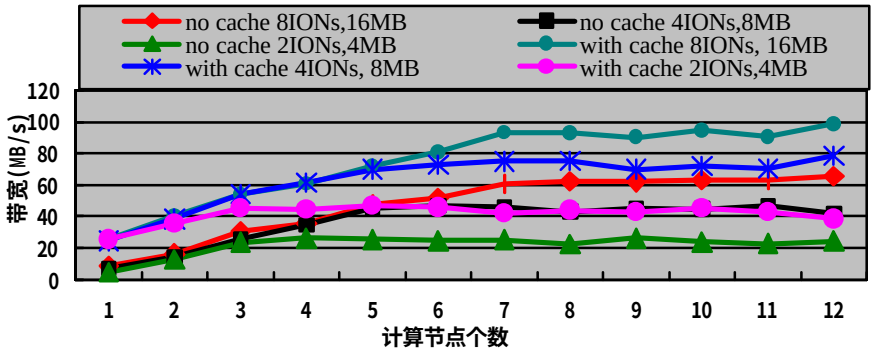


图 4.6　并行写性能

4.5 小结

在计算节点上实现文件系统缓存是最直接，效果也是最明显的。计算进程可以直接从本地缓存中获取数据，从而可以省略计算节点与存储节点之间的通信开销，尽可能的推迟对存储节点磁盘的访问。共享内存是最有效率的进程间通信方式，也是进程间最快的通信方式，使用共享内存方式进行通信，可以省略至少两次进程间的内存拷贝。采用共享内存实现文件系统计算节点缓存，应用进程在映射共享内存空间之后，可以像自己的内存空间一样直接访问共享内存缓存，实现同一计算节点上多个进程对缓存数据的共享，这种共享称为节点内的缓存合作。测试结果表明，用共享内存实现缓存，文件系统数据读写速度的提升是很明显的。

5　存储节点缓存

二级缓存建立在存储节点上，提供了又一个数据存储层次。当一个用户同时向很多存储节点发出数据访问请求时，二级缓存存在一个潜在严重问题：可能出现某一存储节点上的缓存单独缺失的情况，将对文件系统数据并行访问造成不良影响。本章专门研究了这个问题，提了一种新的二级缓存置换策略，即节点间协同置换策略，并为此设计了一个二级缓存协同置换算法。

5.1　二级缓存

集群作为一种分布式多处理机，其内部具有非常丰富的数据存储层次。集群内部节点可以分为负责处理任务的计算节点和负责存储的存储节点。存储节点有可能有很多，每个节点都配置有一个磁盘；也有可能是由一个节点后提供大容量的存储服务，在其后配备有如 SAN、磁带等，该节点充当数据管理服务器的角色。一台机器可以既是计算节点也是存储节点，不过在大型的集群服务器中，通常是各司其责。集群内部计算节点和存储节点都带有丰富的主存资源，除了在计算节点建立客户端缓存之外，为了提高存储节点的数据响应速度，经常在存储节点上也建立一个更大容量的数据缓存。为了区别不同节点上的缓存，把计算节点上的缓存称为一级缓存，把存储节点上的缓存则称为二级缓存。典型的集群数据存储层次如图 5.1 所示。

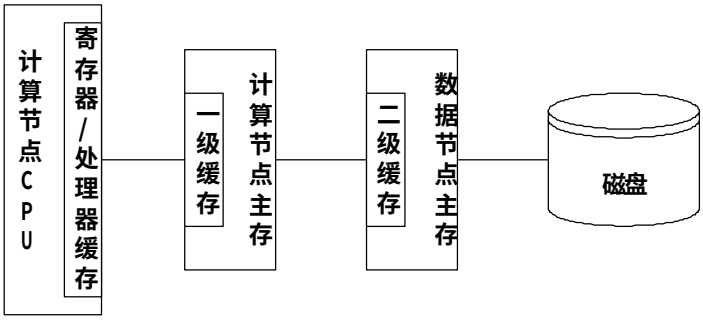


图 5.1　并行文件系统数据存储层次

位于存储节点上的二级缓存具有和位于计算节点上的一级缓存不同的访问模式。这种不同主要表现在访问“时空”特性的不同。对计算节点上的一级缓存的访问具有高度的时间和空间局部原理特性，即最近被访问的数据块极有可能被重新访问、极有可能马上访问刚刚访问的数据块的邻接块[37]。因此，具有高度的时空局部特性的数据

块应该保留在缓存中。这种特性使得考虑访问时间优先的算法如 *LRU* 等在一级缓存中具有很高的命中率。然而，二级缓存并不具有特别明显的时间局部特性，这主要是由二级缓存与一级缓存的访问次序不同造成的。在建立两级缓存之后，应用程序在访问数据时，首先去查找的是位于客户端的一级缓存，只有在一级缓存中找不到所需的数据块时，才去位于存储节点上的二级缓存中查找；在二级缓存中取得数据之后，数据被保存在一级缓存中，并且要持续保存一段时间。这就使得刚刚在二级缓存中被访问的数据块不会被同一个用户马上访问，特别是在客户端具有合作式缓存时，将不会被所有的客户端马上访问到。这说明在二级缓存中使用以访问时间为优先的置换算法不能获得很好的性能。二级缓存和一级缓存的访问模式的不同还体现在块的访问频率上。研究表明[38]，对整个系统而言，数据块的访问频率是极不均匀的：只有很少的数据块被非常频繁访问，这些块称为访问“热点”；一些块被访问次数一般，称之为访问“暖点”；而大多数的数据块几乎从来没有被访问，称为访问“冷点”。同样是由于一级缓存的关系，二级缓存中保留的数据块在整个系统的访问中多属于“暖点”。“热点”数据块由于访问频率大而保留在一级缓存中，在二级缓存中的访问次数也较少，“冷点”数据块本身就很少被访问到，因此二级缓存中的保留长时间的数据块多是那些在客户端看来属于访问“暖点”的数据块。

5.2 二级缓存协同置换

5.2.1 访问组

在高性能计算等并行应用中需要访问大量的非连续数据块[39]， 其中一些数据块经常同时出现在同一个应用的多次访问中。例如顺序访问模式下，连续的几个数据块就经常被同时访问。这种现象在并行计算中表现得尤为明显。采用 MPI（Massage Passing Interface）编程环境进行并行计算时，将任务划分为很多块，同时在一个机器上或多个机器上启动多个进程处理。伴随着任务的划分，应用程序处理的大块数据也被划分成了很多块分在不同的进程上同时进行处理。根据文件数据经常被重复访问的特性，这些划分出来的数据块经常被同时访问到。矩阵运算[39]是一种常见的科学计算，要涉及到大量的数据。矩阵运算的特点是经常整行、整列的元素同时参与运算，例如矩阵相乘运算就是一个矩阵的一行元素与另一个矩阵对应的一列相应元素的计算，矩阵相加则是行与行的相加，因此矩阵的行和列经常作为一个整体出现在运算过程中。

这些经常同时在一个应用中被同时访问的数据块组成了应用的一个访问组。访问模式非常清楚的应用中，访问组的划分很明显，可以根据访问特性在程序中直接指定。如上面所说的矩阵运算，这种应用可以称为访问组已知的应用。在很多应用中，访问模式并不是特别明显，它的访问组则需要在处理过程中根据以往的访问记录进行统计。

5.2.2 节点间协同置换

在数据分片存储的并行文件系统中，多个存储节点并行地向一个或多个用户提供服务。用户的一次需要同时从几个存储节点上获取数据，此时，单个存储节点缓存单独缺失会破坏整个系统的访问并行性，使得其它数据极点的缓存命中变得毫无意义。如图 5.2 所示，用户在一次操作中需要同时访问数据块 B0~B4，这些数据块分别存储在存储节点 Node0~Node4 上。系统进程很快在 Node0、Node1、Node3 和 Node4 的缓存中找到了所需数据块 B0、B1、B3 和 B4，并及时发送给用户。而存储节点 Node2 的缓存中却没有所需的数据块 B2，此时 Node2 只能从磁盘中读取数据块 B2。在同步读写操作中，用户只有等到收到 B2 才算是真正完成本次操作。我们知道，从磁盘中读取数据要远慢于从内存中读取数据，这样 Node0、Node1、Node3 和 Node4 的缓存命中将变得毫无意义。假设 B0~B4 经常被同时访问，构成应用的一个访问组，如果能够使各存储节点缓存同时保存这些数据块的话，无疑会加快数据的读写速度；即使不能同时保留这些数据块时，也应该同时置换出这些数据块，使得各存储节点缓存能留出更大的空间来装载其它数据块。这种考虑了访问组的置换称为存储节点间缓存协同置换。

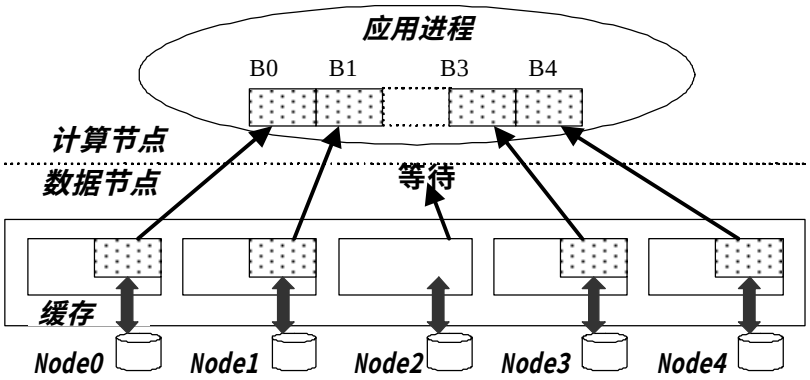


图 5.2 单个存储节点缓存缺失破坏系统访问并行性

5.3 协同置换算法

上面分析可以发现好的并行文件系统二级缓存的置换算法应该综合遵循以下三条基本原则：

（1）利用最多的缓存空间来保存访问“暖点”数据块；

（2）遵循基于访问时间的频率优先，综合考虑访问时间和访问频率；

（3）各存储节点对置换进行协同。

现有的置换算法未能完全考虑这三点：*LRU* 和 *MRU* 等基于访问时间考虑的置换算法缺乏对数据块访问频率的考虑；*LFU* 等基于访问频率考虑的置换算法则缺乏对访问时间的考虑，在访问过程中没有考虑缓存污染问题；*FBR*、*MQ* 等算法综合考虑了访问时间和访问频率，在只有单个存储服务器的系统中可以获得非常好的缓存命中率和缓存读写响应速度，但是没有考虑在集群环境下各存储节点间的协同置换，其整体命中率仍然不够理想。

针对上述问题本文设计了一种并行文件系统二级缓存协同置换算法——*CMQ*（*Coordinated Multi-Queue*）算法[40]。该算法基于 *MQ* 算法设计，在置换时，对位于不同存储节点上的同属于一个访问组的数据块给予整体考虑，即对于一个访问组要么整体置换，要么整体保留。和传统的数据块单独置换算法相比，*CMQ* 算法减小了二级节点缓存中出现单个节点单独缓存缺失的概率，并且各节点的缓存命中率更趋于平均。

5.4 算法描述

CMQ 算法是在 *MQ* 算法的基础上设计的。在访问规律不清楚的应用中，*CMQ* 算法需要根据过去的访问记录来统计应用的访问组。*CMQ* 算法以访问组中数据块的访问频率作为协同置换的依据，各存储节点在协同置换一个访问组时需要获知位于其他存储节点上同一个访问组中数据块的访问频率。为了减少各存储节点之间的通信，*CMQ* 算法在每个存储节点上维护所有存储节点缓存信息队列。同时为了维护各存储节点缓存信息队列的一致性，*CMQ* 算法要求每个存储节点都处理来自用户的所有请求，对所有的数据块访问请求都执行置换算法。用户把每一次访问请求完整地发给每个存储节点，在存储节点上对请求进行拆分，对本地数据块访问请求和非本地数据块访问请求执行不同的处理。在每个存储节点上维护所有存储节点的缓存信息队列，并

不会带来过大的开销，这是因为 *CMQ* 算法对非本地数据块请求只执行空置换，并没有从其他存储节点的磁盘上存取数据，非本地缓存队列各数据项中只记录块的访问信息，其指向数据块的指针都为空。只有对本地数据块请求时才从本地磁盘上存取数据，属于本节点的缓存信息队列各数据项指向数据块的指针指向位于本地内存中的数据块。

CMQ 算法对属于每个存储节点的缓存信息都使用多队列来管理，即分别用 m 个 *LRU* 队列来管理属于每个存储节点的缓存信息。根据数据块的访问频率来决定把其插入到哪一个 *LRU* 队列中。每个存储节点的缓存信息被分成 m 个部分，从而能够很快地从缓存队列中找出最冷的一块进行置换。这样，在每个存储节点上都要维护 $n*m$ 个缓存信息队列，这些队列可用一个 $n*m$ 矩阵来表示：$|Q_{i,j}|_{n*m}$，每一行代表属于一个存储节点的 m 个缓存信息队列。其中 n 代表存储节点的个数，m 代表管理属于每个存储节点的缓存信息的 *LRU* 队列个数，*Q(i,j)*表示属于节点 i 的第 j 个缓存队列。*Q(i,j)*中的数据块允许在缓存中的不被访问的时间（生存期）比 *Q(i,j-1)*中的数据块更长。*Q(i,j)* 称为 *Q(i,j-1)*的高一级队列。*Q(i,j-1)*则称为 *Q(i,j)*的低一级队列。

CMQ 算法根据数据块在缓存中未被访问的时间来动态调整数据块的访问频率。高一级队列中的数据块可能被调整到低一级队列中，最低一级队列中的数据块则有可能被调整出缓存。*CMQ* 算法把数据块从缓存中置换出来并非简单的丢弃。*CMQ* 算法为属于每个存储节点的 m 个缓存信息队列都提供了一个 *FIFO* 队列 $Q_{out}(i)$来记录被丢弃块的历史访问频率和块标记号，一旦被丢弃的数据块被重新访问，即可从中找到它的历史访问频率。

对一个访问组实行整体置换时，需要经常查找同一个访问组中其他数据块的访问频率。为了减少查找时间，*CMQ* 算法把同一访问组中所有的数据块都链接在一起。

CMQ 算法可分成三部分：数据块的访问、数据块队列的动态调整和置换对象的选取。其中数据块的访问是算法的主体，由它调用算法的其他两部分。

5.4.1 数据块的访问

假定用户需要访问位于节点 i 的数据块 a。数据块 a 被命中时，只需将 a 的访问频率加一，并把 a 从当前缓存队列中移到相应新队列 *Q(i，k)*的尾部。如果缓存未命中，则分两种情况处理：1. 缓存中还有空间，则无需置换；2. 缓存已满，需要从缓存队列中选出一个数据块进行置换。查找 Q_{out}（i）队列，如果发现 a 的历史访问记录，

则 a 的访问频率设为历史访问频率加一，并把 a 的历史访问记录从 Q_{out}（i）删除；如果没有发现 a 的历史访问记录，a 的访问频率便设为 1。如果 a 非本地数据块，即当前节点不是节点 i，则不进行真正的数据存取；否则从本地磁盘中取得块 a，并插入到相应新队列 $Q(i，k)$的尾部。

数据块 a 的新队列号 k 由数据块的访问频率求得：$k=QueueNum(f)$，f 是 a 的访问频率。$QueueNum(f)$在不同的背景下可以有不同的定义。在后面的仿真实验中，$QueueNum(f)$定义为 $lg(f)$。

为了便于以后动态调整数据块的访问频率，*CMQ* 算法设定了数据块的生存期限 *expireTime*。当一个数据块被插入到新队列时，数据块 *expireTime* 设为 *currentTime+LifeTime*，*currentTime* 表示当前时间，*lifeTime* 表示数据块的生存期，即该数据块允许在缓存中不被访问的时间，均用访问缓存的次数衡量。下面是 *CMQ* 算法访问数据块 a 的伪码：

```
Accessblock(a，i)
{
    if（a 命中）
            把a从当前队列中移出;
    else
    {
        if( 缓存已满 )
            victim = EvictBlock( i );
        if ( a 在Qout (i) 中 )
            从Qout (i)中取得a的历史访问频率
         else
            a的访问频率为0;
        如果块a位于本节点上，从磁盘中取出，
        否则执行空置换;
    }
    a.reference++;
    a.Queue = QueueNum ( a.reference );
    把a插入到新队列的尾部
    a.expireTime = currentTime +lifetime;
    Adjust(i);
}
```

5.4.2 数据块队列的动态调整

为了消除过去一段时间访问很频繁但最近很久没有被访问过的块，一些高队列中的数据块会被调整到低队列中去，直至丢弃。缓存中的每一块都有一个生存期限 *expireTime*。*CMQ* 算法在每一次访问缓存时，都把当前时间 *currentTime* 增一。如果一个数据块的 *expireTime* 小于 *currentTime*，则表示该数据块已超出了生存期限，应该被

调整到低一级队列中或者丢弃。下面是 *CMQ* 算法动态调整块队列的伪码：

```
Adjust(i)
{
    currentTime++;
    for ( k=0; k<m; k++ )
    {
        c = head of Q_ik;
        if(  c.expireTime < currentTime  )
        {
            把c插入到Q_i(k-1)的尾部;
            c.expireTime = currentTime + lifetime;
        }
    }
}
```

5.4.3 置换对象的选取

缓存空间不足时，需要从缓存信息队列中选取一块实行置换。*CMQ* 首先选择最低一级非空队列的头部数据块作为置换对象。例如，*Q(i,0)*不空，*CMQ* 便选择 *Q(i,0)* 的头部数据块实行置换；否则，选取 *Q(i,1)*的头部数据块实行置换，以此类推。被置换出去的数据块的历史访问记录由 *Qout(i)*记录。

CMQ 算法直接置换出不属于任何访问组的数据块。对于属于某一个访问组的数据块，根据该访问组中所有数据块的访问频率作出是否整体置换的决定。如果选取的置换对象的访问频率在本访问组中是最大的，则协同换出本访问组中所有的块；否则，将本访问组中所有块的访问频率统一为本组中块的最大访问频率，并把它移到相应新队列尾部。*CMQ* 继续选择下一块作为置换对象。下面是置换对象选取伪码：

```
EvictBlock(i)
{
    Victim是存储节点i的最低一级非空队列Q_ij的头一块;
l1: if ( victim不属于任何访问组 )
        把 victim 从 Q_ij中丢弃;
    else
    {
        取得本访问组中块的最大访问频率ref;
        if ( ref<=victim.reference )
            丢弃本访问组的所有块;
        else
        {
            调整本访问组中所有块的访问计数为ref，
            并插入到相应队列尾部;
            victim=q.next;
            goto l1;
        }
    }
    Qout(i)记录下victim的访问记录;
    return victim;
}
```

5.5 算法仿真与结果分析

5.5.1 仿真方法

验证算法效果的方法是：用合成 trace 仿真实验比较 *LRU*、*MQ*、*LFU* 和 *CMQ* 算法的命中率。现阶段还没有对访问组进行细致的研究，而是在 hp disk trace 基础上模拟并行访问的特征，在原有 trace 数据中规定了 20%的访问组。在所有的仿真实验中，块的大小都是 8KB。属于每个存储节点的缓存信息队列用 8 个 *LRU* 队列来管理，即上文队列矩阵中 m＝8。数据块的生存期 *lifeTime* 随缓存大小作出调整。Q_{out} 队列长度是缓存大小的 4 倍。Q_{out} 队列只记录数据块编号和历史访问频率，因此 Q_{out} 队列所占的空间非常小。多次仿真实验发现 *QueueNum(f)*函数定义成 *lg(f)*时效果是最好的（*f* 是块的访问频率）。

为了验证合成 trace 本身的效果，首先利用合成 trace 仿真了只有一个存储节点提供服务的时 *LRU*、*LFU* 和 *MQ* 算法的命中率，目的是为了消除访问组的影响，发现所得的命中率结果与 Zhou[41]等人利用 hp disk trace 仿真所得结果非常相似。这说明加入访问组之后，trace 的效果并没有发生多少改变。表 5.1 列出了单个存储器下两者对 *LRU*、*LFU* 和 *MQ* 仿真实验的命中率百分比。

表 5.1　Hp disk trace 与 合成 trace 下 LRU 和 MQ 命中率百分比

算法	Trace	16MB	32MB	64MB	128MB	256MB
LRU	Hp disk trace	14.5	22.1	41.4	62.5	77.8
	合成 trace	13.6	23.8	41.2	60.6	75.2
LFU	Hp disk trace	20.2	29.6	43.6	57.3	71.5
	合成 trace	18.4	29.2	45.2	59.3	68.1
MQ	Hp disk trace	22.0	36.4	54.2	68.9	78.5
	合成 trace	21.2	38.4	54.2	68.6	77.8

5.5.2 仿真结果与分析

仿真方法是模拟 5 个存储节点向 5 个客户提供并行服务。图 5.3 列出了缓存大小从 8MB 至 256MB 的 *LRU*、*LFU*、*MQ* 和 *CMQ* 算法的访问命中率。从图 5.3 可以看出，相同大小的缓存下，*CMQ* 算法的命中率是最高的。*CMQ* 和 *MQ* 的命中率要明显优于 *LRU* 和 *LFU*。

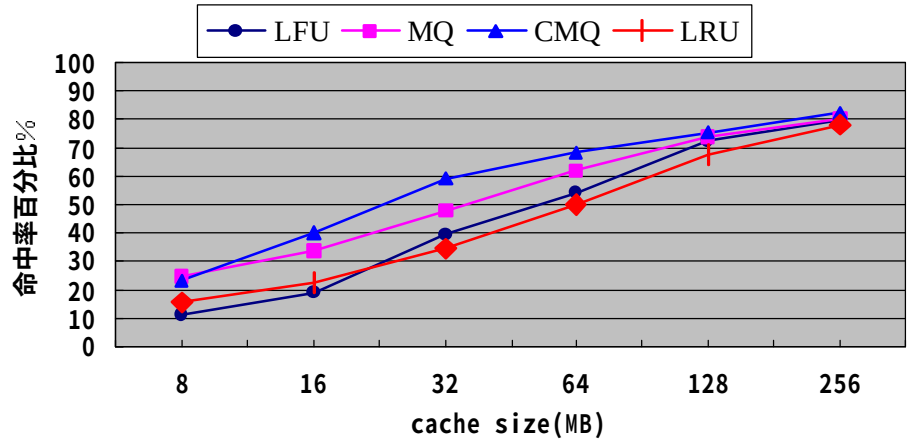


图 5.3　命中率百分比

尤其是在缓存容量为 8MB 时，*CMQ* 的命中率是 *LRU* 的 2.09 倍。这是由于缓存容量较小时，*LRU* 无法把访问热度相对较大的数据块长时间保留在缓存中。从图 5.3 中还可以看出，缓存容量较小时，*LRU* 的命中率要优于 *LFU*，但随着缓存容量的增大，*LRU* 的命中率却要比 *LFU* 差。相对于 *MQ*，*CMQ* 在命中率上也有较大的提高,他们之间最大差距是 125%（32MB 时）。*CMQ* 算法命中率提高的原因是减少了各存储节点中出现单一节点缓存缺失的机会。各存储节点对同一访问组中的块进行了协同置换，提高了缓存效率，进而提高其他热点块的命中率。

为了进一步比较 *MQ* 算法和 *CMQ* 算法，还计算了执行 *MQ* 和 *CMQ* 置换算法之后的各存储节点的缓存命中率方差。从表 5.2 可以看出，*CMQ* 的命中率方差总体上要小于 *MQ*。这表明实行了协同置换之后，各存储节点的缓存命中率趋于平均。其原因是各存储节点实行协同置换之后，同一访问组中的块的命中率趋于相等。

表 5.2　各存储节点缓存命中率方差

算法	8M	16M	32M	64M	128M	256M
MQ	1.14	9.31	4.38	6.28	3.68	0.11
CMQ	0.31	2.56	2.13	1.14	0.69	0.15

5.6　关于访问组的统计

可以肯定的是访问组统计的准确性会对协同置换算法的性能产生很重要的影响。对于访问模式不清晰的应用，访问组的统计会产生一定的前期开销。一种可能有效的访问组统计方法是在为数据块之间引入一个访问相似度的概念。同时根据访问的“空

间局部特性”，可以考虑依远近程度给予数据块的邻近块一定的相似度。本文对此做了一定的研究，以下是一些定义和推导。

定义 1　数据块 A 对数据块 B 的访问相似度为 j ,那么 $0 \le j \le 1$。

定义 2　数据块对其本身的相似度为 1。

定义 3　两个数据块之间的相互相似度是相等的。

定义 4　如果两个数据块的相似度大于给定值 $x(0 \le x \le 1)$，那么这两个数据块可以划分到同一个访问组中。

下面给出数据块 A 和数据块 B 的访问相似度求解公式。

假设数据块 A 总访问次数为 K，数据块 B 的总访问次数为 J，数据块 A 和数据块 B 同时被访问的次数为 m，数据块 A 和数据块 B 之间的间隔为 d，那么数据块 A 对数据块 B 的访问相似度 j 可以用下面公式求解：

$$j = \frac{2m}{K+J} * l + e^{-|d|} * m \quad （0 \le l \le 1，\ 0 \le m \le 1，\text{都为设定的参数}） \quad (5.1)$$

推导 1：由于 A 和 B 是同一个数据块时，那么 $2m = K = J, d = 0$， $j = 1$，所以 $l + m = 1$。

推导 2：当 A 和 B 没有被同时访问到时，那么 $m = 0$，所以 $j = e^{-|d|} * m$，此时 j 只与 A 和 B 之间的间隔有关；

推导 3：当 A 和 B 之间的间隔为无穷大时，即 $|d| \to +e$ 时，那么 $e^{-|d|} = 0$，所以，$j = \frac{2m}{K+J} * l$，此时 j 只与 A 和 B 被同时访问的次数 m 有关。

l 代表了应用的访问频率相关性，m 则代表了应用的空间局部相关性，它们可以在不同的应用中取不同的值。在给定了上述定义和求解公式（5.1）之后，访问组统计的准确性就取决于 l 和 m 的取值以及 x 的确定了。

值得注意的是，访问组中各数据块之间具有很大的关联性，因此访问组统计算法可以从单机文件系统的缓存预取算法中得到借鉴，或者访问组的统计将对文件系统的预取算法有所帮助。本文对此没有做进一步研究。

5.7　小结

利用众多存储节点汇成的丰富主存资源建立并行文件系统存储节点缓存有利于

进一步推迟对磁盘的访问，加快文件读写速度。

并行文件系统存储节点缓存中特有的一种现象：集群某一或几个存储节点的缓存单独缺失将破坏并行文件系统的读写访问并行性，使得其它存储节点缓存命中变得没有意义，降低了那些数据命中节点的缓存效率。针对这个问题，本章设计了一种存储节点间的协同置换思想，即各存储节点对由经常被同时访问的数据块组成的访问组施行整体置换策略。*CMQ* 算法的设计就是在 *MQ* 算法中加入协同置换策略。仿真结果表明，*CMQ* 算法能够获得很高的系统整体命中率，采用 *CMQ* 算法后，各节点的缓存命中率也趋于平均。

其它置换算法也可以改进成为协同置换，*LRU* 算法可以改进成 *CLRU*，*LFU* 算法可以改进为 *CLFU* 等，都能更适用于集群文件系统数据缓存。

对访问组的统计所做的一些定义和求解推导，虽然没有深入研究求解公式中两个参数 l 和 m 的取值和 x 的确定问题，但为进一步研究工作奠定了基础。

6 结束语

本文的研究成果是在国家 863 项目“集群服务器功能软件”基金资助下取得的。并行文件系统在国内外都属于研究得比较成熟的课题，许多研究小组和企业公司研究、开发的大量专用或通用文件系统都试图从各种角度、采用各种先进硬件设备来达到高性能、高可用和高可扩展性。在现有条件下，要设计更高性能文件系统，可以说是困难重重。

本文论述的缓存系统是高性能并行文件系统 HPVFS 的一部分，所作主要工作是改进 PVFS 文件系统，在 PVFS 平台上扩充元数据缓存、计算节点数据缓存和存储节点缓存，并对元数据缓存和数据缓存的实现细节、计算节点数据缓存和存储节点缓存的访问模式进行深入研究，具体包括：

（1）研究了元数据缓存的实现技术，在 PVFS 上设计了没有一致性问题的元数据缓存，有效提高元数据访问速度；

（2）针对传统计算节点缓存实现方法开销大，内存拷贝过多的问题，提出了用共享内存实现计算节点缓存，提高了数据并行读写性能；

（3）发现多个存储节点中个别节点缓存缺失会影响其它多数节点缓存效率，提出了多个存储节点之间缓存协同置换的思想，设计了一种协同置换算法 *CMQ*，有效提高存储节点缓存的整体命中率，并初步研究了 *CMQ* 算法所需的访问组统计方法。

今后还可以从以下几个方面进行深入研究：

（1）访问组的统计算法。明确访问组统计求解公式中具体应用与参数的取值之间的关系，本文只对此做了初略分析，没有深入研究，这将是下一步研究的重点。

（2）缓存预取策略。文件系统缓存并不能提高数据块首次被访问的速度，预取策略有助于解决这个问题。可以借鉴上文所说的访问组统计算法进行预取策略研究。

（3）探讨缓存的其他实现方法。磁盘与主存是两种矛盾的存储层次：磁盘容量大、安全，但速度慢；主存则高速，但容量小且不安全。因此，可以找寻介于两者之间的存储介质来实现缓存。闪存就是一种这样的存储设备，并且闪存特别有利于实现元数据缓存。这就需要研究闪存的存储特性，文件系统与闪存之间的底层接口特性等。

致　谢

能顺利完成这篇硕士论文，真的要感谢大学七年间我的所有老师和同学们！他们给予了我无穷的求知欲望和无尽的生活乐趣。

首先要感谢的是我的导师韩宗芬教授！自进入华工以来，韩老师便无时无刻给予我关心，这种关心如慈母般地无微不至。韩老师对学术的严谨、对研究的认真和执着都一直深深地感动着我。感谢韩老师在学业上对我的悉心教诲，感谢韩老师总是在我前进的道路上指引方向，感谢韩老师在我懈怠时给予的敦促和鞭策，感谢韩老师以各种方式让我明白了很多事理。这种感激是无以鸣谢的。临走了，我只能在心里祝愿韩老师永远年轻快乐。

感谢金海教授为我树立了一个学术上仰望的目标。金老师的渊博知识、对学科前沿的准确把握让我大开眼界。感谢金老师为我提高了良好的科研环境，感谢金老师为我们进步所付出的辛苦工作。感谢课题负责导师石柯副教授的悉心指导，他在课题方向的把握及在研究过程中给予的指导都让我受益匪浅。感谢庞丽萍教授、李胜利教授、章勤副教授以及实验室所有的老师在学习和生活上对我的教导和帮助，他们对科研工作的投入，对工作的认真负责，对学生的关心呵护都是我学习的目标。

感谢师兄岳建辉和师姐徐婕博士对我的一切帮助。从课题项目开展以来，和他们一起讨论、一起学习使我受益匪浅。虽然期间经历了很多挫折，现在回想起来还是充满快乐的，和他们在一起使我第一次懂得了如何做研究。特别祝愿徐婕博士和她的宝贝健康幸福。感谢何飞跃、许俊、杨俊杰、程斌等同学，和他们一起进步的日子是让我终生难忘的。

感谢我的挚友和同学陈汉华在学习生活上的关心。感谢我的室友余潮、程念华和前室友张明虎，和他们朝夕相处真的很快乐，和他们开玩笑是我生活中不可缺少的乐趣。感谢王锦龙、储杰、赵美平、谭鹏柳、吴敏娜、张绮、罗贞、鄢娟、李运发等同学。特别感谢我的本科同窗向双平，高中同窗陈国强、李亮辉。

我最需要感谢的是我的父母兄嫂，他们一直都默默的支持着我，鼓励着我。近二十年的学业，父母对我的付出不是用数字可以来衡量的。父母对我的关爱我终生无法报答。

最后感谢我从小学到硕士十八年来的所有老师和同学！

参考文献

[1] K. Hwang, H. Jin, and E. Chow et al. Designing SSI Clusters with Hierarchical Checkpointing and Single I/O Space. IEEE Concurrency, 1999, 7(1): 60~69

[2] R. Buyya **编**. **高性能集群计算**: **结构与系统**. **第** 1 **版**. **郑纬民**, **石威**, **汪东升等译**. **北京**: **电子工业出版社**, 2001(1): 2~31

[3] D. A. Patterson and K. L. Hennessy. Computer Architecture: A Quantitative Approach. Informed Prefetching and Caching. In: Fifteenth ACM Symposium on Operating Systems Principles. Cooper Mountain, Colorado, USA. 1995. New York: ACM Press, 1995. 79~95

[4] M. Satyanarayanan. The Evolution of Coda. ACM Transactions on Computer Systems (TOCS), 2002, 20(2): 85~124

[5] X. Shen, and A. Choudhary. DPFS: A Distributed Parallel File System. In: International Conference on Parallel Processing (ICPP' 01). Valencia, Spain. 2001. Washington, DC, USA: IEEE Computer Society Press, 2001. 533~541

[6] P. Braam, M. J. Callahan, and P. l. Schwan. The Intermezzo file system. In: Proceedings of the 3rd of the Perl Conference. Monterey, USA. 1999. USA: O' REILLY Press, 1999. 14~25

[7] D. Ellard, J. Ledile, and P. Malkani et al. Passive NFS of Email and Research Workloads. In: Proceedings of the Second USENIX Confererce on File and Storage Technologies (FAST' 03). San Francisco, USA. March 2003. Berkeley: USENIX Association, 2003. 203~216

[8] B. Pawlowski, S. Shepler, and C. Beame et al. The NFS Version 4 Protocol. In: Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000). Maastricht, the Netherlands. 2000. Netherlands: NLUUG, 2000. 34~51

[9] K. W. Preslan, A. P. Barry, and J. E. Brassow et al. A 64-bit, Shared Disk File System for Linux. In: the 6th IEEE Mass Storage Systems Symposium held jointly with the Seventh NASA Goddard Conference on Mass Storage Systems & Technologies. San Diego, California, USA. March 1999. Washington, DC, USA: IEEE Computer Society Press, 1999. 351~378

[10] T. E. Anderson, M. D. Dahlin and J. M. Neefe et al. Serverless Network File Systems.

ACM Transactions on Computer Systems, 1996, 14(1): 41~79

[11] Geoffray, and Patrick. OPIOM: Off-Processor I/O with Myrinet. Future Generation Computer Systems, March 2002, 18(4): 491~199

[12] **贺劲**, **徐志伟**, **孟丹等**. **基于高速通信协议的** COSMOS **机群文件系统性能研究**. **计算机研究与发展**, 2002, 39(2): 129~135

[13] N. Nieuwejaar, and D. Kotz. The Galley Parallel File System. In: International Conference on Supercomputing Proceedings of the 10th international conference on Supercomputing. Philadelphia, Pennsylvania, USA. New York: ACM Press, 1996. 374~381

[14] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In: Proceedings of the First Conference on File and Storage Technologies. Monterey, USA. January 2002. Berkeley: USENIX Association, 2002. 231~244

[15] F. Isaila, and W. F. Tichy. Clusterfile: A Flexible Physical Layout Parallel File System. In: 3rd IEEE international Conference on Cluster Computing. Newport Beach, CA. 2001. Washington, DC, USA: IEEE Computer Society Press, 2001. 37~46

[16] K. Magoutis, S. Addetia, and A. Fedorova et al. Structure and Performance of the Direct Access File System. In: Proceedings of the 2002 USENIX Annual Technical Conference. Monterey, CA, USA. June 2002. Berkeley: USENIX Association, 2002. 57~72

[17] P. J. Braam. The Lustre Storage Architecture. Medford, USA: Cluster File Systems, Inc. 2004. 14~422

[18] P. H. Carns and W. B. Ligon Ⅲ. PVFS: A Parallel File System for Linux Clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference. Atlanta, USA. 2000. Berkeley: USENIX Association, 2000. 317~327

[19] D. A. Patterson, J. Hennessy. Computer Architecture A Quantitative Approach. **北京**: **机械工业出版社**, 1999. 14~39

[20] F. M. Cuenca-Acuna, and T. D. Nguyen. Cooperative Caching Middleware for Cluster-Based Servers. In: the 10th IEEE international Symposium on High Performance Distributed Computing. San Francisco, USA. 2001. Washington, DC, USA: IEEE Computer Society Press, 2001. 303~314

[21] M. Dahlin, R. Wang, and T. E. Anderson et al. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In: the Proceedings of the First

Symposium on Operating Systems Design and Implementation. Monterey, CA, United States. 1994. Berkeley: USENIX Association, 1994. 267~280

[22] G. Voelker, E. Anderson, and T. Kimbrel et al. Implementing Cooperative Prefetching and Caching in a Globally Managed Memory System. In: ACM SIGMENRICS Performance Evaluation Review, Proceedings of the 1998 ACM SIGMENRICS joint international conference on Measurement and modeling of computer systems. Madison, Wisconsin, United States. 1998. New York: ACM Press, 1998. 33~43

[23] P. Sarkar and J. H. Hartman. Hint-based Cooperative Caching. ACM Transactions on Computer Systems, 2000, 18(4): 387~419

[24] J. Wang, and Z. Xu. Cluster file systems: a case study. Future Generation Computer File systems, 2002, 18(3): 373~387

[25] **杜聪**, **徐志伟**. COSMOS **文件系统性能分析**. **计算机学报**, 2001, 24(7): 710~715

[26] M. Vilayanur, M. Kandemir, and A. Sivasubramaniam. Kernel-Level Caching for Optimizing I/O by Exploiting Inter-Applicaion Data Sharing. In: Proceedings of 2002 IEEE International Conference on Cluster Computing. Chicago, Illinois, United States. 2002. Washington, DC, USA: IEEE Computer Society Press, 2002. 425~433

[27] Y. Zhou. Memory Management for Networked Servers. A PhD's Dissertation. library of Princeton University, November 2000

[28] A. Ching, A. Choudhary, and W. Liao et al. Noncontiguous I/O through PVFS. In: Proceedings of 2002 IEEE International Conference on Cluster Computing. Chicago, Illinois, United States. 2002. Washington, DC, USA: IEEE Computer Society Press, 2002. 405~414

[29] S. Garg, and J. Mache. Performance Evaluation of Parallel File System for PC Clusters and ASCI Red. In: 3rd IEEE International Conference on Cluster Computing (CLUSTER'01). Newport Beach, United States. 2001. Washington, DC, USA: IEEE Computer Society Press, 2001. 172~177

[30] W. Ligon III, and R. Ross. PVFS: Parallel Virtual File System, Beowulf Cluster Computing with Linux. T. Sterling ed. Boston: MIT Press, November 2001. 391~430

[31] S. Park, S. Park, G. M. Kim, and K. D. Chung. Design and Implementation of Parallel Multimedia File System Based on Message Distribution. In: International Multimedia Conference. Proceedings of the eighth ACM international conference on Multimedia. California, United States. 2002. New York: ACM Press, 2000. 422~425

[32] K. A. Smith. Workload-Specific File System Benchmarks, A PhD's Thesis. Library of

Harvard University, Cambridge, Massachusetts, United States, January 2001

[33]**李善平**, **刘文峰和李程远等**. Linux **内核** 2. 4 **版源代码分析大全**. **北京**: **机械工业出版社**, 2002. 1: 245~249

[34]N. Nieuwejaar, D. Kotz and A. Purakayastha et al. File-Access Characteristics of Parallel Scientific Workloads. IEEE Transactions on Parallel and Distributed Systems, October 1996, 7(10): 1075~1089

[35]**胡希明**, **毛德操**. Linux **内核源代码情景分析**(**上册**). **第一版**. **杭州**: **浙江大学出版社**, 2001: 687~ 829

[36]J. Dilley, and M. Arlitt. Improving Proxy Cache Performance Analyzing Three Cache Replacement Polices. Technical Report HPl-199-142, HP Laboratories Palo Alto, October 1999

[37]V. Milutinovic, and M. Valero. Guest Editors' Introduction-Cache Memory and Related Problems: Enhancing and Exploiting the Locality. IEEE Transactions on Computers, February 1999, 48(2): 97~99

[38]T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In: Proceedings of the International Conference on Parallel Architecture and Compilation Techniques. Barcelona, Spain. 2001. Washington, DC, USA: IEEE Computer Society Press, 2001. 3~14

[39]R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In: the 7th Symposium on the Frontiers of Massively Parallel Computation. Maryland, United States. 1999. Washington, DC, USA: IEEE Computer Society Press, 1999. 182~189

[40]**韩宗芬**, **林运章**, **金海等**. **一种集群文件系统二级缓存协同置换算法**. **计算机工程与科学**, **录用编号**: 23290

[41]Y. Zhou, J. F. Philbin, and K Li. The Multi-Queue Replacement Algorithm for Second Level Buffer caches. In: Proceedings of the 2001 USENIX Annual Technical Conference. Boston, Massachusetts, USA. 2001. Berkeley: USENIX Association, 2001. 91~104

附录 1　攻读学位期间发表论文目录

[1] **韩宗芬，林运章，金海，岳建辉，徐婕．一种集群文件系统二级缓存协同置换算法．计算机工程与科学．录用编号**：23290．**署名单位**：**华中科技大学计算机学院**

并行文件系统缓存技术研究

作者：林运章
学位授予单位：华中科技大学