

日志结构事务缓存技术研究

培养单位：清华大学计算机系

专 业：计算机科学与技术

本 科 生：周 枫

指导教师：郑纬民 教授

二〇〇〇年六月

清华大学毕业设计论文

摘 要

本文提了一个以“数据库/事务处理”环境和“办公室/工程”环境为应用背景的新的磁盘子系统的改进方案——日志结构事务缓存技术，这一技术能够在很低的成本下，提供比传统文件系统缓存技术更好的性能和更高的可靠性，并且可提高服务器的可用性，减少维护时间。本文主要内容如下：

1. 提出日志结构事务缓存技术，分析磁盘物理特征，指出采用存储在磁盘上的日志结构的缓存，可以充分利用磁盘高速连续写入的特点，大大提高磁盘的写性能。引入与日志结合的 IO 事务的概念，指出采用日志结构的 IO 事务缓存可以解决磁盘数据一致性问题，提高数据可靠性。
2. 基于对磁盘、处理器、内存等设备的发展情况分析，以及与其它方法的比较，论证日志结构事务缓存在目前具备应用条件，且有良好的发展前景。
3. 使用基于跟踪记录（Trace）的磁盘子系统模拟方法对其性能进行模拟，初步结果证实其对性能的提高相当大。
4. 进行了 Linux 下的具体实现的设计，完成了程序结构、算法、数据结构、操作等设计。在设计过程中，充分考虑了结构的合理性、模块化、可移植性等问题。对于一些可能遇到的技术问题及其解决进行了初步的讨论。

清华大学毕业设计论文

ABSTRACT

This paper proposed a new approach called LSTC, Log-Structured Transaction Cache, for the purpose of improving I/O performance. Mostly for the “Database / Transaction” and “Office / Engineering” environment, LSTC technology provides better performance as well as better reliability than traditional file system buffer cache at very low extra costs. It also adds to system availability and reduces maintaining time. The essential contents of this paper are listed as follows:

1. Based on the physical characteristics of hard disks, we proposed the LSTC technology. We point out that using on-disk log as the cache for write operations, the system can exploit the high continual transfer rate of modern disks and boosts writing performance of file system by magnitudes. We also introduce the concept of IO transaction into LSTC, and explain that using a log-structure transaction cache, we can solve the problem of maintaining meta-data integrity, which has existed in Unix-like systems for a long time.
2. After summarizing the recent development of hard disk, processor and DRAM, we conclude that most computers today have the configuration for running LSTC to achieve better performance and reliability.
3. Simple trace-driven simulations are carried out to evaluate the new architecture. Results show that it improves performance considerably.
4. An implementation design under the Linux operating system is presented in the form of architecture, algorithms, data structures and scenarios. Hierarchy, modularity and portability are kept in mind when carrying out the design. Several problems, which we may encounter in actual implementation, are also discussed.

清华大学毕业设计论文

目 录

第 一 章 绪 论	1
1.1 背景	1
1.2 方法概述	2
1.3 本文的主要工作及内容安排	3
第 二 章 日志及其在文件系统中的应用	5
2.1 日志概念及分类	5
2.2 日志在文件系统中的应用	5
2.3 实例分析	7
2.3.1 日志文件系统 (LFS)	7
2.3.2 ext3 文件系统.....	8
2.3.3 磁盘缓存磁盘 (DCD)	8
第 三 章 日志结构事务缓存	10
3.1 设计考虑	10
3.2 LSTC 结构.....	10
3.3 日志基本结构	13
3.4 写操作	15
3.5 读操作	16
3.6 检查点操作	16
3.7 故障恢复	17
第 四 章 LSTC 在 Linux 下的实现设计	19
4.1 结构	19
4.1.1 文件系统 TAFS 结构	19
4.1.2 与相关系统实现方式的比较	21

清华大学毕业设计论文

4.2 LSTC 块设备.....	22
4.3 数据结构	22
4.3.1 lstc_buffer 类型	23
4.3.2 lstc_atom 类型.....	23
4.3.3 lstc_transaction 类型	24
4.3.4 lstc_buffers[].....	25
4.3.5 lstc_fops (file_operations 类型)	26
4.3.6 lstc_gendisk (struct gendisk 类型)	26
4.4 基本操作	26
4.4.1 lstc_init.....	26
4.4.2 lstc_cleanup	27
4.4.3 lstc_open_atom.....	27
4.4.4 lstc_close_atom(atom).....	27
4.4.5 lstc_log_write(atom, buffer_head).....	27
4.4.6 direct_write(buffer_head).....	28
4.4.7 lstc_fops → write = lstc_write.....	28
4.4.8 lstc_fops → read = lstc_read.....	28
4.4.9 lstc_fops → ioctl = lstc_ioctl	28
4.4.10 lstc_commitd.....	28
4.4.11 lstc_checkpointd.....	29
4.4.12 lstc_recovery.....	30
第 五 章 性能评估方法及初步结果	32
5.1 性能评估方法评述	32
5.2 工作负载取得	33
5.3 基于跟踪记录 (Trace) 的事件驱动磁盘模拟器.....	34
5.4 初步结果	35

清华大学毕业设计论文

第 六 章 结论及进一步工作	37
参考文献	38
致 谢	39

清华大学毕业设计论文

第一章 绪 论

1.1 背景

随着信息化与互联网的不断发展，海量信息的存储、处理、检索成为计算机与网络应用的重要领域。新的网络应用需求，例如大型 WEB 站点、数字图书馆、科学数据中心等的出现，导致要求系统存储、管理的信息量的急剧膨胀，对作为计算机系统中信息存储方式的文件系统提出了如下要求：

1. 可靠性：即在软硬件错误的情况下最大限度地保证数据安全；
2. 可扩展性：随着需求的增加，容量应能自由增加；
3. 性能：往往要能针对特定的应用模式，提供高性能；
4. 可用性：要方便管理，发生错误及时检测、快速恢复。

而传统文件系统在这几个方面都遇到了问题：

1. 性能

由于磁盘的机械运动本质，磁盘的性能在过去一段时间内缓慢的提高。一般来讲，硬盘的速度比主存速度要低三到四个数量级而且其访问速度的加快（每 10 年快 1/3），带宽的变大（25%每年）要比内存的进展要慢。

以往解决这一速度差异的主要用两种办法，第一种是改变硬盘子系统的结构，RAID 是硬盘结构最重要的革新之一，它最主要是通过并行性提高了数据的吞吐率，对访问延迟并没有大的改进，同时对于大量的小数据块的写操作，其性能非常的差。要解决这一问题，主要的方法是使用大容量的缓冲存储器，由于要保证写操作的可靠执行，因此必须使用不受系统掉电或故障影响的存储器，即昂贵的非易失性存储器（NVRAM, Non-Volatile RAM）。NVRAM 高昂的价格使其无法在所有系统中使用，人们希望能有更好的解决办法。

清华大学毕业设计论文

第二种是通过优化文件系统来提高小块数据 I/O 操作的性能,比如优化文件系统中块的分配算法,将相关的文件分配在较接近的磁盘块中,比如在 BSD FFS 文件系统中就采用了这样的方法;或者使用主机的 RAM 作为 I/O 操作的缓存。后者被广泛地采用,但由于主机 RAM 在掉电后即无法恢复,因此一方面这个缓存大小不能太大,因且要定期写到磁盘,导致性能受影响,同时也给文件系统的可靠性带来影响。

2. 可靠性与可用性

在文件系统中引入缓冲在提高性能的同时,很大程度地降低了系统的可靠性,因为一旦系统掉电或发生故障,缓冲中的数据将不能及时写回磁盘,从而丢失数据。然而更糟糕的是,这会造成文件系统数据结构的 inconsistency,从而在系统重新开始运行前,必须对整个文件系统进行完整性检测。在海量数据存储的条件下,这一过程需要非常长的时间。例如,在 linux 的 ext2 文件系统下,一个 100G 的文件卷进行一遍 fsck 过程需要 10 分钟到 10 小时不等的时间,这极大地掉低了系统的可用性,对于大部分应用是不可接受的。

为了解决这个问题,传统文件系统中一般采用对关键数据使用同步写的方法,比如系统消息日志和数据库的事务提交大都采用同步写,而一些文件系统对元数据(目录、i-节点等)也采用同步写,而由于元数据大都是小块数据,是最需要进行缓存的数据,因此这样做导致系统整体性能下降很多。

1.2 方法概述

为了解决这些问题,综合前人关于日志系统的研究成果,经过分析与实验验证后,我们提出一种提高磁盘子系统可靠性与性能的新方法——日志结构事务缓存技术(Log-Structure Transaction Cache, LSTC)。采用这种技术以后,磁盘子系统可以在保证数据可靠及文件系统一致性的前提下,提供更好的 IO 性能。

我们的想法,是利用日志结构的磁盘写性能很高的特点,用日志磁盘作磁盘

清华大学毕业设计论文

的写入缓存，大幅度提高系统的写入性能。其中，将日志结构与块结构结合起来，数据磁盘用块结构存储，而利用一个小容量磁盘，以及相等容量的主存 DRAM 缓存，构造了一个非对称（对两者访问速度而言）、持久（掉电后数据仍然存在）、高速（与 DRAM 速度接近）的缓存，利用磁盘连续写入时数据传输速率很高的特点，可以使组合的缓存达到与一般 DRAM 缓存相近的速度，而持久性又是通过磁盘的持久性得到的。

采用这一缓存后，所有的 IO 写操作，相对数据磁盘而言，都可以采用异步写（延迟写）的方法，相对同步写而言，异步写的响应时间要低数个数量级（有实验表明，2 个到 4 个）。一般 IO 操作中，新建文件、删除文件、移动文件等都带来很多同步写操作，用户写操作也可能是同步的（由用户自己指定），这些操作往往很慢。消除同步写后，使得用户程序总的运行时间减少。

另有一个思想是将 IO 操作事务化(Transactionize)，确保文件系统数据一致性，使得文件系统一致性检查没有必要。具体做法是将逻辑上是一个整体，但要分数次写操作完成的操作，比如创建文件，组成一个原子操作，而若干个这样的操作组成一个事务（这样组合纯粹是为了效率，每个原子操作一个事务效率过于低下），写入数据磁盘时，保证同一事务中所有原子操作都能顺利完成。实际上，写入 LSTC 时，数据就是按事务组织的。当数据从 LSTC 写入数据磁盘（称为检查点操作）时，直到一个事务更新完毕，才将这个事务从 LSTC 中去掉。

如果系统意外停止工作（比如掉电），LSTC 中就会有检查点或正在进行检查点的数据，下次系统启动时，文件系统会首选对 LSTC 中所有的事务进行检查点操作，以确保数据正确写到数据磁盘。这一操作称为日志前滚（Roll Forward），相对于数据库中的日志滚回（Roll Back）而言。

1.3 本文的主要工作及内容安排

日志方法是很早就得到广泛运用的一种方法，在操作系统、数据库系统、分

清华大学毕业设计论文

布计算中，日志都扮演着很重要的角色。而将日志技术直接运用到文件系统中，则是在 80 年代末期，从那时到现在，已经有不少的基于日志的文件系统方面的新方法提出，比如日志文件系统（LFS）¹，磁盘缓存磁盘（DCD）²，另外还有若干商业文件系统中使用了日志技术，比如 SGI 公司的 XFS，IBM 公司的 JFS 等，这些都表明日志技术的确能改进文件系统。

本文提出的方法是文件系统的附加或改进，很适合对响应时间要求较高的“办公/工程”环境和对数可靠性要求较高的“数据库/事务处理”环境，适合于这些环境下的工作负载特征，能够用较低的额外花费提供比原有文件系统好得多的可靠性和性能。

对这一方法，我使用模拟的方法进行了初步的验证，证实它确实是有效的，模拟的结果与预期的基本一致。

目前已经完成了在 Linux 操作系统下的实现的概要设计，完成了日志结构及内存数据结构的设计，对实现的层次、如何与现有代码配合、具体实现中可能遇到的一些问题都作了考虑。

本文按先后顺序阐述了以下内容：

1. 第二章是对日志技术的简单介绍以及对其以前在文件系统中应用的介绍。
2. 第三章分析日志结构事务缓存（LSTC）的原理，包括对一些设计问题的说明和对性能的预测。
3. 第四章解释在进行具体实现设计中遇到的一些问题及解决，实际上，进行实现设计是遇到问题最多也是花费时间最多的工作。
4. 第五章是对所采用的性能评估方法和结果的说明，磁盘系统的模拟是一个很复杂的问题，限于时间与水平，本文写作时仅仅实现了较简单的模拟，进一步的工作正在进行中。结果说明方法是有效的。
5. 第六章给出结论，提出进一步的工作。

清华大学毕业设计论文

第二章 日志及其在文件系统中的应用

2.1 日志概念及分类

我们在这里讨论的日志，是指程序在某种操作过程将信息中记录到一个流式的媒介中，在以后正常或非正常的情况下，由程序再次阅读的内容。因此我们讨论的日志有以下几个特征：

1. 由程序阅读：而不是由人阅读，这是与一般所说日志（比如 Unix 下的系统日志）的不同，我们讨论的是类似数据库系统中所使用的日志；
2. 流式写入的：日志的写入是顺序的，即具有类似磁带一样的性质，写入过程不能随机进行。这是日志与一般文件的区别。但日志可以随机读出。

这样定义的日志从使用方式上分一般可分两类：

1. 回滚日志：将进行的操作记录在日志后，以后某一时候可以利用日志将系统恢复到记录日志前的状态。数据库系统中一般使用这种日志完成事务处理功能，如果一个事务不成功，系统将使用日志回滚对系统信息进行恢复。
2. 前滚日志：有时又称为预写（Write-ahead）日志，通过将操作先都记录在日志中，并不应用到实际的系统上，而当需要时，再一次性将所有操作作用到系统上。这种日志经常使用在 IO 系统中，有时也使用在分布系统或并行系统中本地进程对全局数据的操作上，用来保证数据的一致性，以及尽量减少进程间的通信。

2.2 日志在文件系统中的应用

先让我们来看文件系统一般的物理介质——硬磁盘的一些性能参数，及它们的特点。

清华大学毕业设计论文

硬盘性能一般由如下参数：³

- 磁头定位时间
- 旋转延迟
- 数据传输率

磁头定位时间和旋转延迟占了响应时间的大部分，但它们主要由机械运动速度决定。这导致硬盘虽然密度、容量不断增加，但转速、磁头寻道时间的改进却是不大的。过去 30 年间，磁盘容量由几 M 扩大到几百 G，但转速仅仅由几千 RPM 变为一万多 RPM，而寻道时间仅由 20 多 ms 缩短为 5-10ms。而数据传输率由数据密度和转速共同决定，由于存储密度的增加，在转速提高不多的情况下，磁盘的连续数据传输速率提高较快，目前的高速硬盘连续数据传输速率在 20MB/S 以上。因此，可以说目前的硬磁盘性能是不平衡的，有很大一部分的 IO 操作随机性较强，受寻道、旋转延迟的影响而性能很差。

为了解决磁盘子系统的性能问题，人们提出了各种方法，如前所述，RAID 是其中的一种，改变文件系统组织结构是另一种方式。比如很早的时候，BSD 系统中的 FFS 文件系统就试图通过将磁盘分成多个分配区，并尽量将文件的相关数据（文件内容、目录、i 节点）都放在同一个分配区内，以期减少寻道时间，提高性能。

而在此之后，有学者改变了文件系统传统的 I-结点，数据块位置固定的做法，将日志结构引入文件系统中。日志有这样几个特性，使其可能被应用于文件系统：

1. 日志的连续写入特性，可以弥补磁盘的性能缺陷。因为磁盘连续写入速度很高，所以将随机的 IO 请求转化为连续的日志写入磁盘，速度会有数量级的提高；
2. 利用日志结构，有可能将逻辑上有关系的数据靠近存放，有利于提高读取性能；
3. 日志可以将一些操作组合成整体，保证操作的原子性，对于保证文件系统完整性有利。

清华大学毕业设计论文

目前使用日志技术的文件系统根据使用日志的目的与方式可分为这样几种：

1. 文件系统结构完全按日志格式组织，比如日志文件系统（LFS）
2. 将日志作为元数据的暂存处，以保证可靠性和文件系统完整性，比如 IBM 公司的 JFS，Linux 下的 ext3 等
3. 将日志与主存缓存结合，作为文件系统缓存的一部分，以期提高性能，如磁盘缓存磁盘（DCD）

下面，简单介绍一下这三类中的各一个。

2.3 实例分析

2.3.1 日志文件系统（LFS）

1988 年 U.C.Berkley 的 John Ousterhott 和 Fred Douglass 提出的日志文件系统（Log-Structured File System）¹，是一个完全基于日志的文件系统，称其可对小块数据的写操作提供两个数量级以上的性能提高，同时将垃圾收集（去除无用的日志）的开销控制在可接受的范围内。1992 年他们对如何用 LFS 实现一个传统的文件系统作了完整的论述⁴。

LFS 将磁盘看成是一个很长的环型日志，对文件的写操作（修改、追加）的内容连续地写到日志的最后。因为多次修改可以通过一大块日志来写，所以 LFS 对小块数据的写操作性能很高。而 i-节点的改变也和文件一样，写在日志的最后，这里就有与传统文件系统不一样的地方，因为 i-节点的位置不定，必须要在写出 i-节点的同时，写出 i-节点的位置信息。这在 LFS 称为 i-节点图块(i-node map block)，i-节点图块也写到日志最后，而所有的 i-节点图的位置都记录在磁盘的固定位置，定期更新。

LFS 中另一个重要问题是空闲块的收集，因为日志信息是冗余的，一段时间之后，要对日志进行紧缩，收为冗余的日志空间。这个工作后来被认为是 LFS 在一些环境下性能不佳的主要原因。

清华大学毕业设计论文

LFS 声称可以得到比传统文件系统高得多的性能，然而，后来 Margo Seltzer 和 Keith A. Smith 指出，LFS 的性能并不是在所有的应用模式下都是好的，与一些采用了集簇块分配策略的文件系统（比如 BSD FFS, ext2）相比，在负载很重或磁盘空间紧张时，LFS 的性能会变得很差。

2.3.2 ext3 文件系统

ext3 是 linux 下的一个新文件系统，目前正在开发中，有可能要成为 linux 将来使用的文件系统。ext3 使用日志主要是为了保证文件系统一致性，ext3 将对磁盘的操作写到数据磁盘的同时，还写到一个日志文件中，这样保证了系统崩溃之后重新启动时，不需要全面检查磁盘就可以保证数据一致性。

ext3 通过修改内核源码，在 Linux 的主存缓存旁增加了一个日志文件，对磁盘的所有写操作先以写到日志中，同时也留在主存缓存中。系统有更新线程定期将数据写到数据磁盘中，从而将日志中的内容作废。日志文件在正常情况下是只写的，只有当系统失败重新启动时，才从其中读数据来恢复到数据磁盘中。

ext3 对所有的写操作都做日志，因此每次 IO 操作都要做两次，因此性能较差。而且 ext3 大量修改了 Linux 内核源码，移植性较差。

2.3.3 磁盘缓存磁盘（DCD）

磁盘缓存磁盘（Disk Caching Disk）是罗德岛大学的 Yiming Hu, Qing Yang 在 1996 年提出的使用日志技术提高磁盘写性能和可靠性的技术。

DCD 使用一个容量较大（几十到几百兆）的日志磁盘（log disk），作为 RAM Cache 的扩展，来保存文件的改变（写操作）。实际上是在 RAM Cache 和数据磁盘之间，加了一层日志磁盘作为缓存。在 RAM Cache 中的数据在系统一有空闲时就写到数据磁盘（data disk）中。

对日志磁盘的写入按整块数据进行，磁头几乎不需移动，因此速度很快，写

清华大学毕业设计论文

完后写操作即可立即返回。同时日志磁盘可以很大（数十到数百 MB），可以发挥程序访问磁盘的局部性。DCD 不象 RAM Cache 那样存在可靠性问题，因为系统崩溃重新启动后可以由日志磁盘的内容完全重建没有写入到数据磁盘中的数据。

DCD 使用虚拟硬盘设备的方式来实现，因此与文件系统无关，可移植性较好。

但是，由于 DCD 对上层完全无知，操作以块为单位进行，因此对一些逻辑上具原子性的操作无法保证原子性，比如创建文件操作，往往要做建 i-节点，改目录表，写文件内容等几个块的写操作，但如果在没有完成时，系统崩溃，那磁盘结构就会不一致。DCD 无法避免这一点，因此还要借助完全扫描的方法来保证文件系统一致。

DCD 的另一个较大问题是当负载较重时，与 LFS 一样，将数据从日志磁盘回写到数据磁盘的操作会很费时间，而且，由于要从日志磁盘的其它地方而不是末尾读取数据，导致日志磁盘来回寻道，对性能影响非常大。故 DCD 一般仅适用于小或中等负载的环境。

清华大学毕业设计论文

第三章 日志结构事务缓存

3.1 设计考虑

日志结构事务缓存（LSTC）的设计目的是通过使用日志技术，来尝试解决硬盘子系统在写性能，特别是小文件的写操作性能上的问题。从 LFS、DCD 出现以来的 5-10 年来，硬件方面的发展有几点是与 LSTC 的设计相关的：

1. 动态 RAM 价格大幅下降。计算机的主存大小迅速增加，目前一般 PC 及工作站都有 100M 以上主存，小型服务器普遍有数百 M 主存。这在五年前是无法想象的。
2. 硬盘存储密度突飞猛进，硬盘传输速率成倍增加。5 年间一般磁盘容量从 1G 增加到 100G。传输速度从几 MB/S 上升到几十 MB/S。

硬盘传输速率的增加，使得采用日志技术带来的效率增益提高。而由于主机内存的大量增加，我们认为 DCD 中采用的主存与日志缓存的上下层结构已经没有必要，因为实际上数十 M 至 100M 的缓存一般已经能够满足要求，而主存已经能够提供这样的容量。DCD 将日志缓存设计为读写式设计，在系统负载大时，日志磁盘要大量寻道，性能低下。

基于这样的考虑，我们认为在新的硬件条件下，设计新的基于日志的文件系统缓存技术是可行的也是有很大意义的。

3.2 LSTC 结构

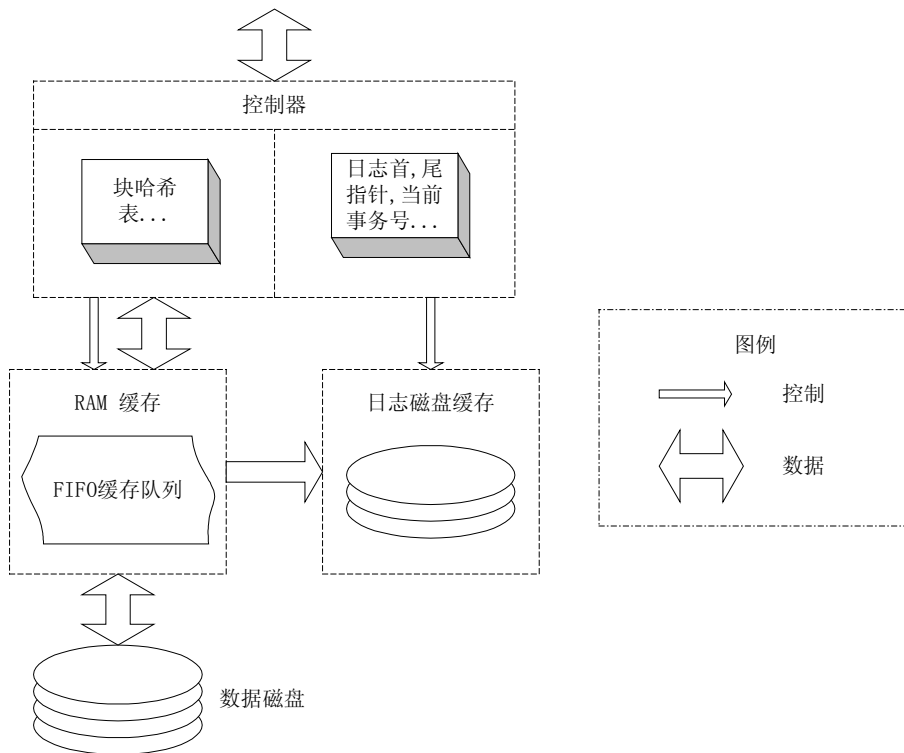
日志结构事务缓存（LSTC）采用一种非对称的结构，通过 RAM 与日志结构的磁盘的不同特点互补来组成一个可靠、高速的缓存。不考虑 LSTC 内部结构，LSTC 从外部看起来是一个具有接近 DRAM 的写入/读出速度，可以随机读写，非

清华大学毕业设计论文

易失性的缓冲存储器。

根据文件系统操作的特点，为了保证文件系统数据的一致性，LSTC 中数据的写入以**原子**为单位进行，一个原子是若干个磁盘块的集合，同一个原子的操作具有事务性，当文件系统处于稳定状态时，要么所有块都已写入磁盘，要么所有块都没有写入磁盘。这样的事务性的特点，使得文件系统完整性检查成为没有必要的事情。

LSTC 由三部分组成：控制器、RAM 缓存、日志磁盘缓存，其结构如图表 1 所示。



图表 1 LSTC 结构

RAM 缓存和磁盘缓存的内容几乎完全相同，除了部分刚刚写入 RAM 的数据还未写入磁盘缓存以外，其它数据都相同。

正常情况下，磁盘缓存是只写的，控制器将 RAM 缓存中的新数据不断地写入日志磁盘缓存的最后，因为磁头几乎不需要寻道，因此数据传输速率很高，几乎

清华大学毕业设计论文

可以达到磁盘的最高连续传输速率。

RAM 缓存在正常情况下负责所有的随机读写操作，它实际上是一个 FIFO 的队列，新进入的数据加在队尾，已经写入数据磁盘的数据就从队头淘汰出去。当要读取数据时，控制器访问 RAM 缓存，实现在这个队列上的随机访问。

控制器是 LSTC 的核心，主要任务是控制在三个存储体（RAM 缓存，磁盘缓存、数据磁盘）间在适当的时候转移数据，定位最新的数据在哪个存储体并读取，以及进行错误恢复。在控制器中，对应磁盘缓存有一些数据结构，例如**日志头、尾指针，当前事务号**；对应 RAM 缓存也有一些数据结构，最重要的是**块哈希表**，用来查找对应某一磁盘地址的最新数据的指针。

由于日志缓存磁盘上的内容在 RAM 缓存中都有，因此它起到一个完整的备份作用，数据保存在其中仅仅是为了一旦系统崩溃，RAM 缓存中数据丢失时，能利用日志中的信息，重建所有的未写到数据磁盘的信息。

前面已经提到，若干次逻辑上属于同一操作的写操作称为一个**原子**，对 LSTC 而言，原子是最小的写操作单位。若干个原子组成一个**事务**（这样组合使得操作粒度变粗，纯粹是为了效率，每个原子操作作为一个事务效率过于低下），写入数据磁盘时，保证同一事务中所有原子操作都能顺利完成。实际上，写入时，数据就是按事务组织的。当数据从 LSTC 写入数据磁盘时，直到一个事务更新完毕，才将这个事务从 LSTC 中去掉。

另外，LSTC 并不缓存所有的写请求，对于一些非元数据的大块数据请求，将其直接送往磁盘，这样有助于提高效率。

下面为了说明方便，引入几种数据状态和数据操作。

由于引入了日志磁盘，因此数据状态比一般的 IO 缓存中干净（Clean）和脏（Dirty）两种状态要多一种，就是已提交（Committed），其意义是已经写入日志磁盘，但是还未写入数据磁盘，相应的，脏（Dirty）数据指的是写入 RAM 缓存，但还未写入日志缓存的数据。

注意，这里的数据状态与系统文件系统中的缓冲中数据块状态没有关系，当

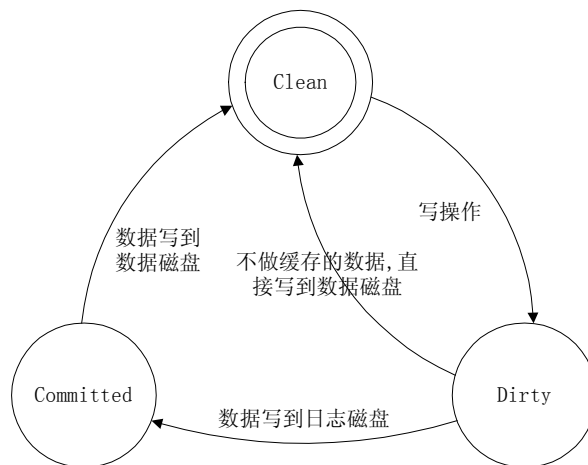
清华大学毕业设计论文

写入请求到达 LSTC 并且返回之后,文件系统缓冲中数据块的状态就已经是 Clean,而实际上在 LSTC 中,这时数据还可能是 Dirty 或 Committed。

改变数据状态的操作可能有:

- **写入 (Write):** 将数据由文件系统层写到 LSTC 层, 根据是否进行缓存, 分别进行不同的操作;
- **提交 (Commit):** 指将数据从 RAM Buffer 中写到日志磁盘中, 写入以事务的形式进行, 即必须一个或多个完整的原子操作才能提交;
- **检查点操作 (Checkpointing):** 指将数据在日志中标为过时, 数据所占的日志空间可以重新利用, 当一个事务的数据已经完全写到数据磁盘时, 日志就已经没用了, 可以做检查点操作。

数据状态与操作的关系由图表 2 显示:



图表 2 数据状态与操作

3.3 日志基本结构

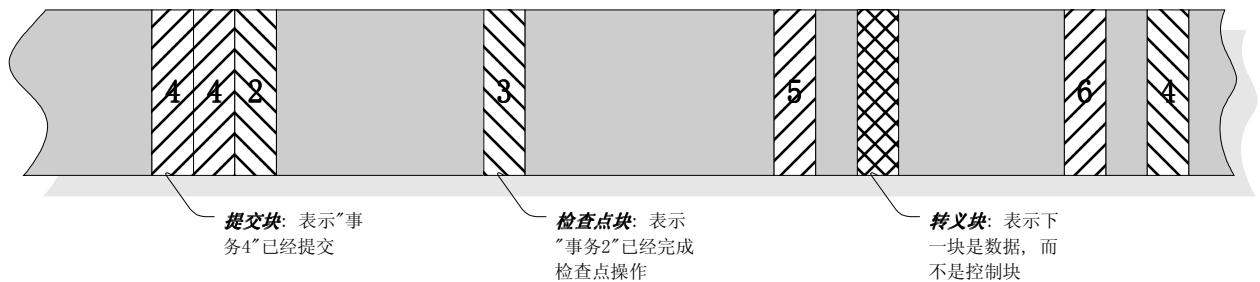
磁盘日志的用途是在发生意外后恢复出没有写入数据盘的数据。日志存放在一片连续的磁盘区域中, 简单做法, 可以直接存放在一个磁盘分区中。

清华大学毕业设计论文

日志要满足如下的要求：

1. 能够在写入操作意外中断的情况下，重新启动后识别出上次哪些内容是完整的，哪些是不完整的，以便进行恢复操作。
2. 日志应该是写优化的，写日志时应该是“流式”的，即连续写入的，也就是说按照类似磁带的方式来访问。而且提交一个事务完成后，不需要到另一个地方去写标志。但日志不需要为读取做优化。

示意性的日志结构如图表 3：



图表 3 日志的典型结构

说明：

1. 图中显示的是日志的一部分，横向表示块地址的延伸，当日志超过磁盘结束后，就绕回开始端；
2. 在日志的最前端（块 0），有一超级块（在图上未标出），存贮日志的一些基本信息和日志是否正常结束的标志。
3. 灰色的是**数据块**，表示实际的数据。斜线指示“**控制块**”，“控制块”有三种：提交块、检查点块和转义块，意义如下：
4. **提交块**表示从上一提交块到此提交块间的数据都属于同一事务，而且这一事务已经提交完毕，提交块中同时包含这些数据块的磁盘地址，所以根据数据块数量多少，提交块有可能会占几个磁盘块。
5. **检查点块**表示某一个事务的检查点操作已经完成，即这个事务的数据已经完全写到数据磁盘上，日志中的这一事务实际上已经没有用了。
6. **转义块**表示下一块是数据，而不是控制块，需要转义块是为了保证数据块

清华大学毕业设计论文

与控制块的正确区别。因为控制块一般通过开头几个字节的特定值来区别（称幻数），对于开头几字节也是这个值的数据块来说，就需要用转义块。一个要注意的问题是，转义后的数据块开头的值不能为幻数，因为如果日志绕回后，刚好覆盖了转义块，则还是会造成错误。

3.4 写操作

进行写操作时，LSTC 控制器首先通过数据的类型、大小等参数判断是否进行日志，对于元数据一定要进行日志，较大块的连续数据不进行日志，小块数据也可进行日志以提高性能。这是基本的判据，但也可以有更复杂一些，比如当 CPU 很忙或日志盘快满时，可以只对元数据进行日志，以减轻系统负担及对日志空间的需求。

对于不进行日志的数据，控制器直接将其交给磁盘驱动程序处理。

对于其它的数据，控制器首先将其送入 RAM 缓存的 FIFO 队列的队尾，在修改若干数据结构之后，这时有两种选择，立即返回或等到数据写入日志后返回，我们分别称为**提交前返回**和**提交后返回**，很显然，前者性能更好，后者更安全。

数据写入 RAM 缓存之后，**提交**操作将数据由 RAM 缓存写到日志中，这时一方面要保证数据及时写到日志中，因为在 RAM 缓存中的数据是不安全的，另一方面为了提高性能要求每次写的数据块较大。我们通过一个较小的延时 T 和一个较大的缓存空间限制 M 来达到这样的目的，进行提交操作的条件是：

(上一次写到现在的时间 $\geq T$) 或 (未提交的数据大小 $\geq M$)

T 较小（比如数十 ms），而 M 较大（比如数百 K）使得这一策略既能在系统较空闲的时候（写操作间时间较长），保证数据及时写入日志，又能在系统很忙的时候，采用较大的写块来提高性能。

如果采用提交后返回， T 的值将对性能影响较大，这时写操作的响应时间受到 T 的很大影响，过大的 T 导致低负荷时响应时间太长，而过小的 T 导致高负荷时

清华大学毕业设计论文

写入日志的块过小，影响性能。可能使用动态的 T 是更好的选择，这需要进一步的实验来验证。

3.5 读操作

读操作比较直接，对于文件系统的 Cache 中没找到的块，LSTC 在块哈希表中查找，如果找到，就将 RAM 缓存中的内容返回，如果没有找到，就将直接从数据磁盘中读出数据并返回。

这里有一个问题是 RAM 缓存与文件系统缓存很多时候内容是重复的，造成内存空间浪费。这个问题在具体实现中，可以通过 RAM 缓存与文件系统缓存的更紧密结合来解决，即对于同一块数据可以共享缓存的内容，而只是分别保存指针。

3.6 检查点操作

因为在 LSTC 中，日志缓存是当成 Cache 使用的，因此其中的数据最终得写到数据磁盘中；而且日志磁盘的空间是有限的，系统运行一段时间后，日志空间就会出现不足。检查点操作将数据从日志中写到数据磁盘中，同时将这些数据占用的日志空间释放出来，以便再次利用。

检查点操作具体实施时比较简单，控制器从 RAM 缓存的 FIFO 队列的最后开始取数据，按事务为单位进行，一次将一个事务的所有块写入数据磁盘中，然后在日志中写出检查点块，将这个事务标为过时，同时可以将 RAM 缓存中的数据释放，并修改相应数据结构。经检查点之后，日志尾部前进，日志中的空闲空间变大。

检查点操作与提交操作不同，因为日志中的数据是安全的，不会因为掉电而消失，因此检查点操作采用相对消极的策略进行。只在日志盘较满（比如空闲空间小于 20%）、系统空闲较长一段时间（比如数秒）后或系统正常关闭前才进行。检查点操作由后台进程完成，不影响其它前台的工作。

清华大学毕业设计论文

在检查点操作中，一个可以考虑的性能改进是进行磁头调度，即将一个事务中的所有操作重新排列顺序，使得完成所有操作，磁头的寻道距离变短，一般使用所谓“电梯算法”，即类似电梯的调度方法，保证磁头单向运行，在一次运行过程中，进行尽量多的操作，到达边界后，再反向单向运行，如此重复。磁头调度是充分发挥磁盘性能的重要手段，有研究表明，磁头调度能使磁盘实际表现从最大性能的 7% 上升到 25%。⁵

但是，在 Unix 类系统的磁盘驱动中，一般已经含有磁头调度算法。看起来在检查点操作中再做磁头调度没有意义。

事实上并不完全是这样，由于考虑通用性的原因，磁盘驱动中的请求队列一般较短（比如在 Linux 中，IDE 磁盘写请求队列长度是 85，在源码中是 $NR_REQUEST * 2 / 3$ ）。而研究表明要达到最大性能的 25%，需要对长度为 1000 的队列进行调度，因此在检查点操作中，如果一个事务中的请求数足够多，进行磁头调度可能是有效的。假设一个事务有 1MB 大小，其中每个请求 1K，那就有 1000 个请求，而这样的情况在系统负载较大时会很常见，目前的 CPU 计算能力很强，应该可以轻松承受对这 1000 个块进行排序操作。

3.7 故障恢复

LSTC 的故障恢复是指当系统没有正常关闭的情况下，下次系统启动进入工作状态之前，将没有写到数据盘的数据从日志取出，写入数据盘中。

故障恢复中最主要的问题是如何找出日志的首尾。确定首尾之后，只要将这段日志读入内存，重建 RAM 缓存和块哈希表等数据，就完成了恢复工作。在前面的基本日志结构之上，找出日志首尾是比较容易的。因为日志中的信息是冗余的，因此可能有多种方法确定这一信息，最简单的方法，可以做两次日志扫描：

1. 第一遍扫描，找出所有控制块，分析出已提交的最大事务号（有溢出绕回的问题，下面讨论），和已进行检查点的最大事务号，其间的即是要恢复

清华大学毕业设计论文

的。

2. 第二遍扫描，进行实际的恢复工作，将数据读进内存，建立相应的数据结构即可。

当然这样做速度会比较慢，一个 100M 的日志可能要几十秒要几分钟时间，实际上一次扫描或不用完全扫描都是有可能，可以进一步优化。

清华大学毕业设计论文

第四章 LSTC 在 Linux 下的实现设计

4.1 结构

我们将首先在 Linux 下实现 LSTC，并使其与文件系统结合起来运行。由于涉及操作系统内核及磁盘驱动程序，所以设计中遇到不少与应用层程序不同的问题，在接口、结构及 LSTC 所处位置等问题上也经过反复考虑和讨论。目前完成的工作是结构、数据结构、基本算法的设计。由于时间限制，还没有进行实现。

LSTC 要与文件系统结合才能使用，因此我们计划从 Linux 的 ext2 文件系统修改而得到一个与 LSTC 一起使用的文件系统（TAFS, Tsinghua Advanced File System），由于 Linux 内核有模块支持，可在不改动内核的情况下，动态挂接新的模块，因此新的文件系统和 LSTC 的安装使用将会很方便。

下面是基本的实现设计。

4.1.1 文件系统 TAFS 结构

很多当前的 Unix 环境，都提供对多种文件系统的支持，多个文件系统是作为独立的模块加入操作系统内核的，因此，一般说来，要实现一个新的文件系统，只要按照一个定义好的接口，实现这个系统的功能即可。

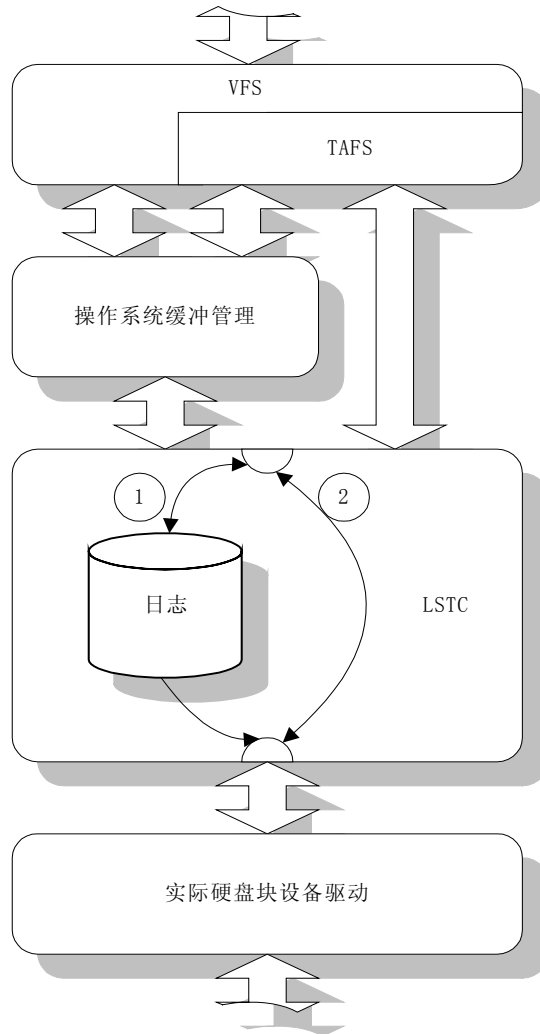
Linux 下这种对多种文件系统的支持是基于虚拟文件系统（VFS）的设计，VFS 定义了一套虚拟的文件系统操作，一个新文件系统只要实现了这些操作，就可以挂接到 Linux 之上。按照这种思路，我们也可以这样来实现 TAFS。

剩下唯一的问题是 VFS 所定义的操作及层次实际上比较高，所有的文件系统都共用 Linux 的缓冲管理机制（实现在 fs/buffer.c 等一批文件中），而实际上我们需要改变缓冲管理的部分机制，比如写入缓冲区的数据，到底何时应该写入数据

清华大学毕业设计论文

磁盘，应该由 LCFS 来控制。但 VFS 没有包含这一接口。

我们的办法是将 LSTC 作为一个设备驱动来实现，而 TAFS 在需要的时候越过 VFS 的接口，直接调用 LSTC 的接口。具体的调用结构如图表 4 所示。



图表 4, Linux 实现的结构

从上图可以看出，TAFS 上层与 VFS 接口，对下层与 Linux 的缓冲接口，与其它文件系统并列。

LSTC 则是一个块设备驱动，并且声明自己是一个磁盘，它一方面调用实际的设备驱动完成读写操作，另一方面又实现了日志缓存，定义了一套 IO 控制操作，

清华大学毕业设计论文

来实现对 IO 事务的管理，例如，新建事务、提交事务等。

采用这一结构，我们认为有以下优点：

- 充分利用 OS 已有功能；
- 可移植性好，Solaris、FreeBSD 都有 VFS 或类似的结构；
- 不需要或很少量改变操作系统内核；
- 良好的模块化，TAFS 和 LSTC 是不同层次的功能，应该在 OS 的不同层来实现；
- 现有文件系统，应该都可以稍加修改，使用 LSTC 而提高可靠性和性能。

在此结构中，LSTC 完成以下功能：

- 提供与一般磁盘块设备驱动程序相同的 IO 操作接口；
- 提供事务化的 IO 写操作接口；
- 提供对上层透明的日志结构缓存，以加速 IO 操作。

为达到这些需求，LSTC 采用虚拟设备驱动的方式来实现，即在现有磁盘块设备驱动的外层，包裹上自己的代码，一部分传统的功能通过直接调用原有设备驱动来实现，一些独有的功能，则通过自己的代码来实现。

这一实现方法实际上相当于面向对象编程中的继承方法，即 LSTC 是原有设备驱动的子类，继承了原有设备驱动的一些功能，同时又扩展了原有设备驱动。当然，这只是概念上的继承，实际内核并不是用具有面向对象功能的语言实现，所以编程中一些实现继承的手段要自己编写。

4.1.2 与相关系统实现方式的比较

其它几个同类的系统的实现形式不同。

- DCD(Disk Caching Disk)，采用设备驱动的形式实现了日志结构的 Cache，但没有 IO 事务的概念。DCD 完全用设备驱动的形式实现，因此可以在任何现有文件系统上实现，但因为缺少 IO 事务的概念，因此磁盘元数据一

清华大学毕业设计论文

致性无法保证，系统掉电后还是需要运行 fsck 程序。

- Ext3，是一种使用日志技术的文件系统，在文件系统层实现具有事务功能的日志，使用了公用的缓冲管理代码，大量修改了内核数据结构，以保证缓冲能够支持 IO 事务。

相比之下，LSTC 作为一个设备驱动实现，但又有一些特殊的接口，是 DCD 与 ext3 两种实现方式的折衷。

4.2 LSTC 块设备

通过对内核块设备驱动的机制分析，要实现这样一个设备驱动，我们要做的事情有：

- 注册新的主设备号 (MAJOR number)；
- 对系统中需要使用 LSTC 的所有磁盘，建立相应的文件系统结点（也就是 /dev/中的设备节点文件），并为它们创建相应的内部数据结构，这些设备之间，用从设备号来区别。
- 在内部数据结构中，要保存以下内容：指向实际设备驱动的指针、当前所有活跃的事务的表、当前所有已经结束但还未提交的事务、所有已经提交当还未过检查点的事务。
- 将一部分 IO 请求直接交给原设备驱动做；
- 另一部分 IO 请求用日志中实现；
- 建立一个新的内核线程 (lstc_commitd)，负责对数据进行提交工作；
- 建立一个新的内核线程 (lstc_checkpointd)，负责定期进行数据的检查点操作。

4.3 数据结构

下面列出 LSTC 中的主要数据结构，对每个数据结构，首先说明其用途，然

清华大学毕业设计论文

后是其简要结构，这里没有列出其 C 语言形式的详细结构，而只是简单描述。

根据 linux 内核代码的一般习惯，所有公开的接口数据结构和函数前都有模块名为前缀，而所有内部私有数据结构与函数都没有前缀。这里遵守这一习惯，公开的结构前都有 `lstc_` 前缀（比如 `lstc_buffer`），私有的则没有（比如 `log_write`）。

在具体实现时，要仔细考虑竞争情况的处理，所以实际数据结构可能会加上一些锁定标志，等待队列等。

4.3.1 `lstc_buffer` 类型

`lstc_buffer` 是一块数据在缓冲区中的形式，有以下内容：

- 数据块指针；
- 数据块在磁盘上的地址（主、子设备号、块号等）；
- 双向链表指针，指向上一个下一个 `lstc_buffer`；
- 所属的 `lstc_atom` 的指针。

4.3.2 `lstc_atom` 类型

`lstc_atom` 称为**原子操作**，是一些 `lstc_buffer` 的集合体。对于事务化的 IO 写操作来说，`lstc_atom` 表示了一个最小的操作单位，即，一个 `lstc_atom` 所写的几个块，要么全部成功，要么全部失败而丢失。通过 `lstc_atom` 实现了磁盘数据结构的一致性。每次进行事务化 IO 时，首先要取得一个 `lstc_atom` 结构，然后利用此结构进行 IO 操作。

`lstc_atom` 的结构如下：

- ID；
- 所有 `lstc_buffer` 组成的链表的头指针。
- 状态，也就是内含 `buffer` 的状态，可以有：`dirty`, `committed`, `clean` 三种；
- 双向链表指针，指向上一个和下一个 `lstc_atom`；

清华大学毕业设计论文

- 所属的 `lstc_transaction` 的指针。

4.3.3 `lstc_transaction` 类型

`lstc_transaction` 即是 **IO 事务** 对应的数据结构，一个 `transaction` 是若干 `lstc_atom` 的集合，`lstc` 在进行检查点操作时，按照 `lstc_transaction` 为单位来进行。从逻辑上说，检查点操作可以按 `lstc_atom` 为单位来进行，即每次确认一个 `lstc_atom` 从日志中写到了数据盘上。但由于检查点操作涉及很多数据块的修改，而 `lstc_atom` 一般只包含很少的数据块，因此按 `lstc_atom` 为单位做检查点操作效率太低。所以引入 `lstc_transaction` 是基于性能考虑。

`lstc_transaction` 结构如下：

- ID，此标识符是一个持久的单增值，下一个值记录在日志的超级块中，也就是说下一次系统重新启动时，此值将从上次的值继续增加。用一个 32 位无符号整数，应该可以保证能正确处理绕回的情况；
- 事务状态，可以是：`active`, `committing`, `committed`, `checkpointed` 四种。意义分别是：

`active`: 正在接受数据的那个事务

`committing` 正在向日志中提交数据的那个事务

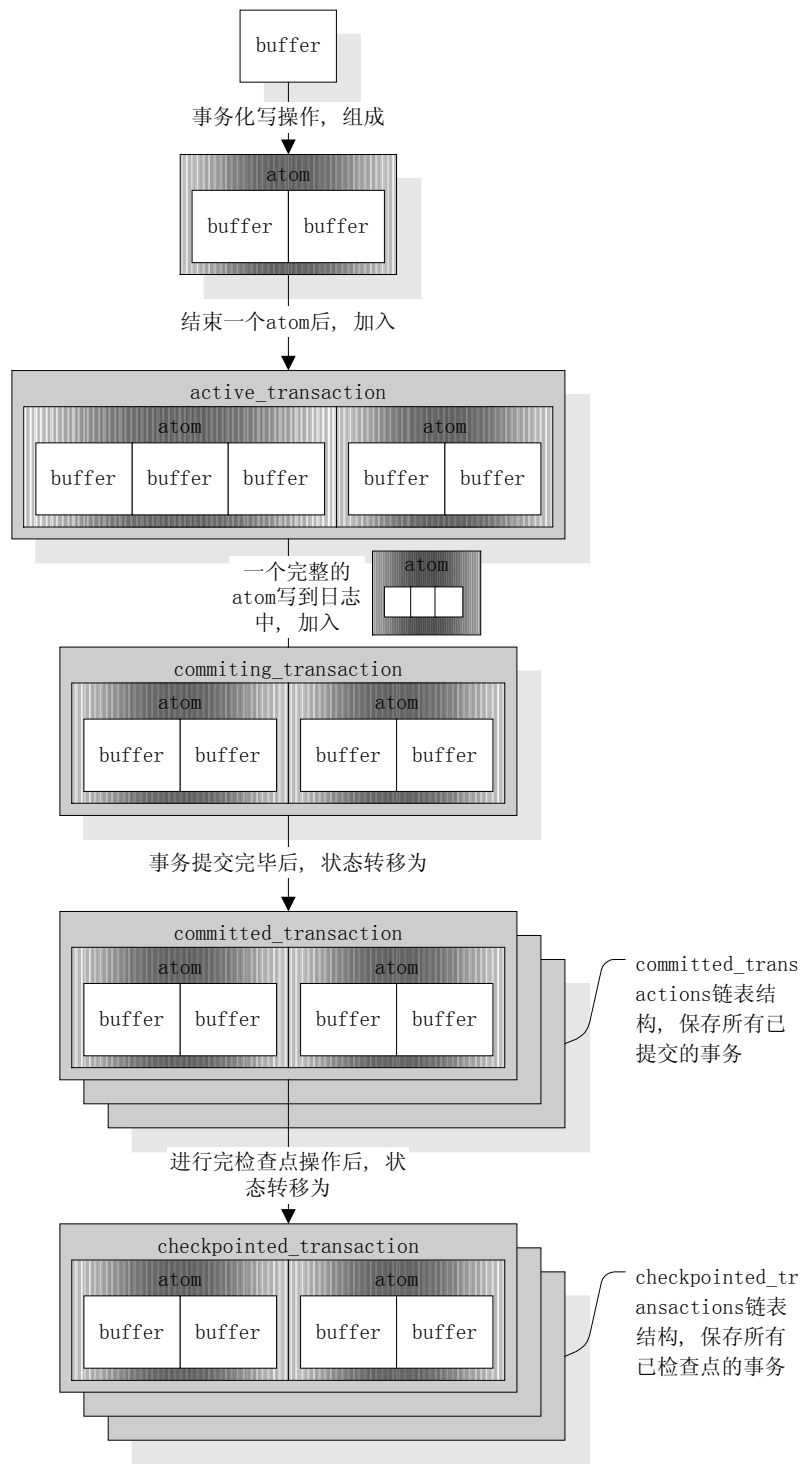
`committed` 已经向日志中提交完毕，但数据还未写到数据盘中的事务

`checkpointed` 数据已经写入数据盘中的事务，实际上含有的都是干净的数据，留在内存中只是为了作为 `cache`，提高读取速度；

- 所有 `lstc_atom` 组成的链表的头指针；
- 双向链表指针，指向上一个和下一个 `lstc_transaction`

以上三种类型 `lstc_buffer`, `lstc_atom` 和 `lstc_transaction` 是整个 `lstc` 的基础，它们的相互关系和大致的变化过程如下页图所示。

清华大学毕业设计论文



图表 5, 内存数据结构关系及操作

4.3.4 lstc_buffers[]

清华大学毕业设计论文

`lstc_buffers[]`是一个哈希表的数组，对应每一个子设备号。完成从数据块地址到 `lstc_buffer` 的映射，上层要读取数据时，先在 `lstc_buffers[]`中找，如果有缓存的数据，则直接返回，否则，调用下层的实际设备驱动读取。

要注意的是，`lstc_buffers[]`中，每一个块地址只保存最新的一个 `lstc_buffer`，而在整修缓存空间中，一个地址是可能对应多个 `lstc_buffer` 的，在不同的事务中，允许有相同地址的块存在。

4.3.5 `lstc_fops` (`file_operations` 类型)

`file_operations` 结构用来指明一个块设备能够完成的操作，`lstc_fops` 没有什么特别，与一般磁盘的 `file_operations` 一样。

4.3.6 `lstc_gendisk` (`struct gendisk` 类型)

用来完成分区表的识别。`gendisk` 结构是内核提供给块设备驱动使用的完成分区识别功能的系统数据结构。内核通过一些子设备命名与子设备号间的对应规则，来实现分区的自动识别功能，在这过程中，就需要在 `lstc_gendisk` 中设定的一些值及函数指针。

4.4 基本操作

下面列出 LSTC 中的主要操作，对每个操作，首先说明其目的及使用场所，然后是操作实现的简单描述。

4.4.1 `lstc_init`

初始化 `lstc` 块设备驱动：

- 注册设备号 (MAJOR)；

清华大学毕业设计论文

- 初始化块设备信息，包括 blk_dev（注册 lstc_request 函数），blk_size, blksize_size, hardsect_size, read_ahead;
- 注册 lstc_commitd 和 lstc_checkpointd

4.4.2 lstc_cleanup

卸载 lstc 块设备驱动:

- 清除设备信息
- 注销设备号
- 等待 lstc_commitd 将所有原子全部提交
- 等待 lstc_checkpointd 将所有事务全部进行全检查点操作
- 杀死 lstc_commitd 和 lstc_checkpointd

4.4.3 lstc_open_atom

开始一个 IO 写的原子操作。它分配一个 lstc_atom 结构，然后返回之。

4.4.4 lstc_close_atom(atom)

当一个原子中的所有 IO 写调用完成后，使用者调用 lstc_close_atom，它将这个 lstc_atom 加入 active_transaction 结构中，然后唤醒 lstc_commitd 线程来将 active_transaction 中的原子提交。

4.4.5 lstc_log_write(atom, buffer_head)

事务化的写操作。atom 是这个操作所在的原子，buffer_head 是内核的一个通用数据类型，内含一个或多个缓冲区。log_write 将 buffer_head 中的数据复制到 atom 中，并且在 lstc_buffers[] 中作相应登记，然后立即返回。

清华大学毕业设计论文

4.4.6 `direct_write(buffer_head)`

内部使用的非事务化的写操作。直接调用磁盘驱动完成。

4.4.7 `lstc_fops → write = lstc_write`

公开的数据写操作，由 VFS 和 AFS 调用。如前文（节[错误！未找到引用源。](#)）中所述，可根据一定算法判断，或者调用 `lstc_log_write`，或者调用 `direct_write` 来完成。

4.4.8 `lstc_fops → read = lstc_read`

读操作，首先在 `lstc_buffers[]` 中查找，如果找到则直接返回。否则调用磁盘驱动完成操作。

4.4.9 `lstc_fops → ioctl = lstc_ioctl`

设备相关的 IO 控制接口。目前没有，所以直接调用实际磁盘驱动的 `ioctl`。

4.4.10 `lstc_commitd`

`lstc_commitd` 是一个内核线程，在系统初始化时即被启动，任务是将数据从内存缓冲中写到日志中，并管理事务由 `committing` 状态到 `committed` 状态的转移（参见 4.3.3 *lstc_transaction* 类型）。

如前所述（见 3.4），在提交数据时采用相对积极的策略，在 `lstc_commitd` 中可以这样做：

- 当 `lstc_commitd` 被唤醒时，判断未提交的原子的数量，如果大于某一阈值，则立即进行提交。如果小于，即判断系统空闲时间，如果大于某一阈值，

清华大学毕业设计论文

则也进行提交。其中，后一判断是为了保证脏数据不在内存保留太久。

- 当 `committing_transaction` 中的原子数量大于一定阈值，或一个 `committing_transaction` 存在时间超过某一阈值后，`lstc_commitd` 应该写出提交块，同时将此事务状态标为 `committed`，再生成一个新的事务作为 `committing_transaction`。

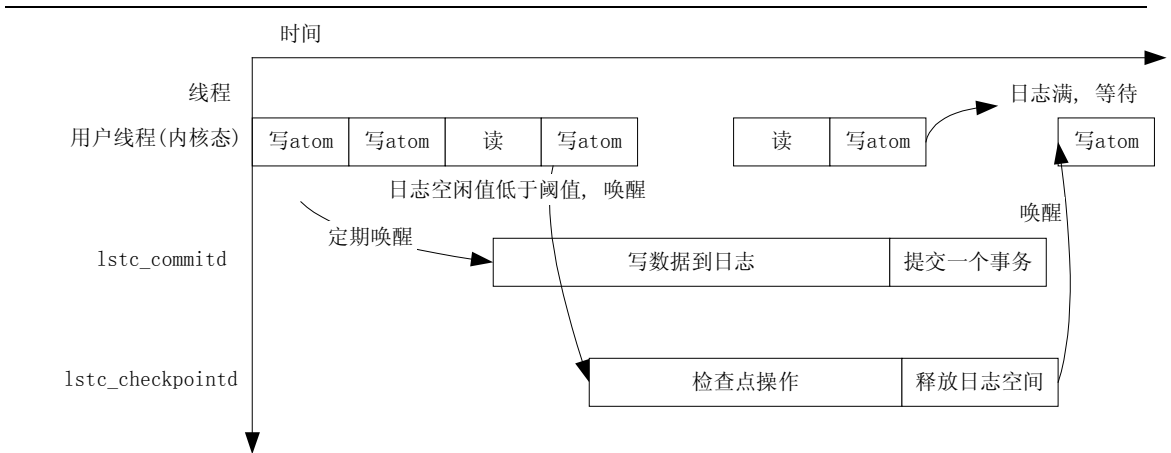
4.4.11 `lstc_checkpointd`

与 `lstc_commitd` 一样，`lstc_checkpointd` 也是在系统初始化时启动的内核线程，它的任务是将数据从日志中写到数据磁盘。并管理事务由 `committed` 状态到 `checkpointed` 状态的转移（参见 4.3.3 *lstc_transaction 类型*）。

与 `lstc_commitd` 不同，由于没有丢失数据的危险，`lstc_checkpointd` 在进行检查点操作时相对消极，只在必要的时候进行。比如采取这样的策略：

- 在其它操作（比如分配缓冲区）中发现系统内存小于某一阈值时，唤醒 `lstc_checkpointd`；
- 当 `lstc_commitd` 写入日志时发现日志空间小于某一阈值时，唤醒 `lstc_checkpointd`；
- 当 `lstc_commitd` 发现已经没有日志空间时，唤醒 `lstc_checkpointd` 并在其等待队列上等待，直到有足够的空间。
- `lstc_checkpointd` 定期醒来，如果发现系统已经闲置时间超过某一阈值，则开始进行检查点操作。

清华大学毕业设计论文



图表 6 工作线程间关系

图表 6 显示了用户进程所进行的 IO 操作与 lstc_commitd 和 lstc_checkpointd 间一次典型的交互关系:

4.4.12 lstc_recovery

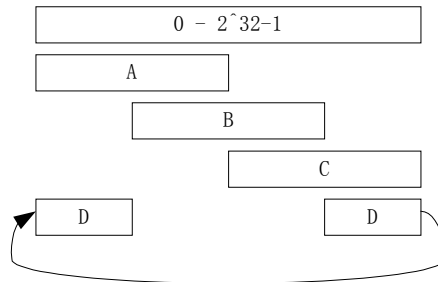
如果系统正常关机, 会在日志的第 0 块(超级块)中写入正常关机标志, 而当系统运行中, 这一标志是清除的。因此, 如果系统非正常停止工作后再启动, 通过这一标志可以识别出来。这时, 调用 lstc_recovery, 进行从日志恢复数据的工作。

lstc_recovery 采用前面所述的恢复方法, 具体内容见 3.7 故障恢复。

其中有一个问题就是事务号溢出的问题, 由于事务号是持久的, 随着系统不断运行, 最终会溢出。如果事务号是 32 位, 假设每秒有 100 个 IO 事务, 那么它将在系统运行 11930.5 小时(497 天)后溢出, 所以溢出的问题我们必须考虑。

清华大学毕业设计论文

实际上，由于日志空间的限制，同时在日志中存在事务号范围是非常小的，



图表 7 事务号绕回

这样我们可以假设事务号只能同时存在于其 32 位一半的范围内，即有如图的四种可能（A, B, C, D）。

由编号是可绕回的，因此判断“大小”有些复杂。判断 a, b 两数大小时，具体做法是：首先取其最高两位，如果相同，比如其它位即可，如果不同，按上图分组的规则来判断。

清华大学毕业设计论文

第五章 性能评估方法及初步结果

在进行实际系统的开发之前，我们希望能对其性能有初步的预测和掌握，同时掌握不同的算法取舍、参数调整对结果的大致影响。又因为涉及的是操作系统内核，进行改动和分析改动的效果都比较麻烦，因此更需要预先对其有一个把握。磁盘系统是一个相当复杂的多因素的系统，进行解析的分析是不太可能的，因此我们采用模拟的方法，通过对用户访问、LSTC、磁盘分别进行建模或跟踪，组成综合的模拟系统，来考察其性能。

这一工作目前正在进行中，模拟器已经设计完成并部分实现，已经能给出一些简单情况下的结果，但我们的最终目的是要模拟真实的用户访问条件下的系统行为，到这一目标还有一段路要走。

这个模拟器不仅可以用于分析 LSTC，将来也可用户关于磁盘系统、文件系统的其它研究中。

5.1 性能评估方法评述

对磁盘存储系统的性能模拟一直是一个研究的热点，一般来说，人们有这样几种方法：

1. 利用人工合成的访问请求，在人工建模的虚拟磁盘上进行操作；
2. 利用跟踪记录（Trace），即是实际环境下的磁盘访问的日志记录，在人工建模的虚拟磁盘上进行操作；
3. 建立系统级模拟器，模拟整个操作系统的行为，让实际的应用程序在其中运行，比如 Stanford 大学的 SimOS 系统⁶；

显然这三种方法实现的复杂性不一样。第三种方法当然能取得最好的效果，但构建模拟器的工作量巨大，而且另一个问题是它要求所模拟的访问程序是实际可

清华大学毕业设计论文

运行的，也就是必须是一个基本已经实现的程序才能利用系统级模拟器，这在算法的早期评估中往往是做不到的。因此第一、二种方法还是有其生命力，尤其是其于 Trace 的方法。

简单的基于 Trace 的方法是将所有对磁盘的访问请求依次记录下来，在模拟的磁盘上进行重放。由于磁盘特性的改变，这时的各请求响应时间会不一样，在简单的基于 Trace 的方法中，不考虑这一改变对上层的设备驱动、文件系统和应用程序的影响，一般假设请求间的间隔不变，而顺次将后一请求提前与推迟。这实际上会导致较大的误差，因为在多进程并发的条件下，一旦磁盘响应时间发生变化，请求间的次序也会发生变化。

在我们的模拟器中，对此稍作改进，仅仅认为同一进程的请求次序是固定的，通过在 Trace 中记录进程发出请求的时间，以及模拟磁盘驱动程序的特性，实现了高一个层次的模拟，即从 Trace 中抽取出各进程分别对磁盘进行访问的序列，对它们进行并发的模拟，这应该能更好的反映实际情况。

模拟系统有如下部分组成：

1. Trace 取得工具，通过修改 Linux 内核，加入生成 Log 的代码来实现；
2. 事件驱动的控制，这完成与 OS 中的任务调度器类似的工作；
3. LSTC 的基本算法实现；
4. 磁盘模型；
5. Trace、模拟结果数据的可视化程序；

目前 1、4、5 已经完成，其中 4（磁盘模型）使用了 DiskSim 模型⁷，这是据称目前最准确的磁盘模型，对 10 种目前主流 SCSI 磁盘的模拟响应时间误差小于 0.3%。2、3 正在进行中。

5.2 工作负载取得

工作负载以跟踪记录（Trace）的形式取得，通过修改操作系统内核，在 IO 操

清华大学毕业设计论文

作例程中加入目录操作，记录操作类型、操作时间和其它参数来完成。在 Linux 下，直接使用内核日志功能实现。下面是生成的某一个 Trace 的一部分

```
Jun 16 19:19:32 zf kernel: 961154372.920540:2126:newrequest: type=1 ,current=3, sector=1311968, count=8
Jun 16 19:19:32 zf kernel: 961154372.920575:2126:start: current=3, LBAsect=1353359
Jun 16 19:19:32 zf kernel: 961154372.929142:2126:endwrite
Jun 16 19:19:38 zf kernel: 961154378.872423:2127:newrequest: type=0 ,current=671, sector=581840, count=8
Jun 16 19:19:38 zf kernel: 961154378.872463:2127:start: current=671, LBAsect=623231
Jun 16 19:19:38 zf kernel: 961154378.885561:2127:endread
Jun 16 19:19:38 zf kernel: 961154378.886283:2128:newrequest: type=0 ,current=671, sector=581936, count=8
Jun 16 19:19:38 zf kernel: 961154378.886313:2128:start: current=671, LBAsect=623327
Jun 16 19:19:38 zf kernel: 961154378.892578:2128:endread
```

在这过程中，遇到的问题有：

- 因为日志数据量较大，缺省的内核日志缓冲区过小，经常溢出而丢失数据，将其由 16K 改成 1.6M 后解决；
- 日志操作要耗用一定的 CPU 时间和 IO 操作，采用日志后是否影响原有程序行为？通过将日志写到不同磁盘或网络上，以及将日志写方式从同步 IO 改为异步 IO，基本消除了这个影响。实验证明改进后的 Trace 提取对原 IO 操作基本没有影响。比如一个同样的 IO 操作，在打开和关闭日志的情况下，分别需要 29 和 28 秒完成。

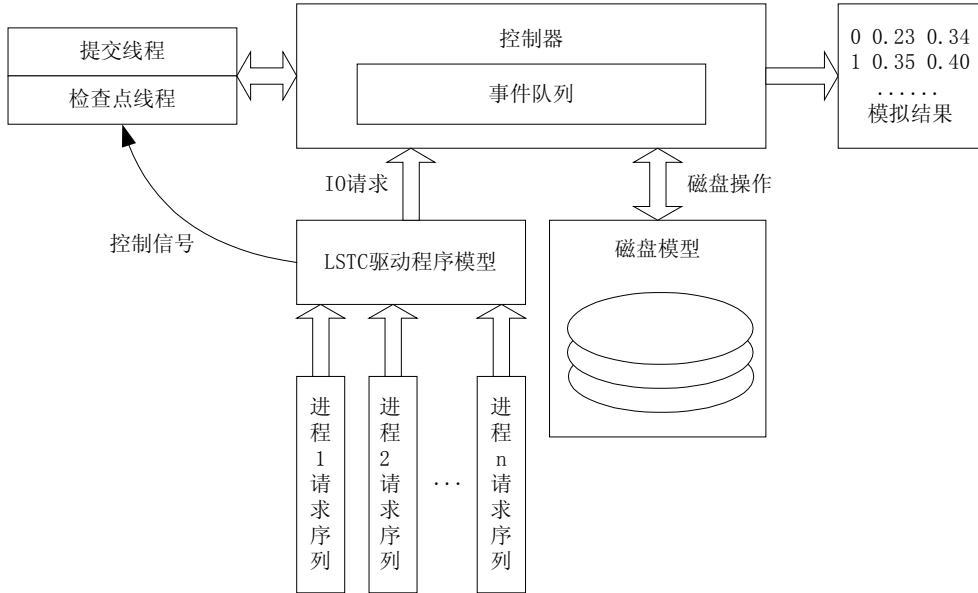
5.3 基于跟踪记录（Trace）的事件驱动磁盘模拟器

模拟器结构如图表 8 所示。

控制器维护一个事件队列，并通过此队列模拟操作系统的进程调度。所有进程的下一次请求经驱动程序排队后，下一个请求进入事件队列；而提交线程、检查点线程的下次可能的操作也作为事件，进入事件队列；磁盘模型运行所需的一些内部事件，也进入事件队列。控制器每次选择最早的事件执行。这样就可以在一个单一进程的模拟器中，模拟多任务的实际运行环境。

模拟器的输出结果是一个与输入格式相近的 Trace 文件，将通过上面提到可视

清华大学毕业设计论文



图表 8 基于 Trace 的事件驱动磁盘系统模拟器

化程序，变成图形显示输出，可以方便地分析比较不同情况下系统的行为特性。

5.4 初步结果

由于模拟系统中的核心部分 LSTC 还没有完成，所以目前不能给出模拟的完整结果。但已经有一些极限情况下的结果，比如下表显示了两个不同的工作负载，在日志磁盘空间无限、不考虑检查点操作的影响的假设下，对所有 IO 操作的平均响应时间：

工作负载	不使用 LSTC	提交前返回的 LSTC	提交后返回的 LSTC
cpsync.log	5.71	0	0.63
untar1.log	4.92	0	0.60

时间单位：毫秒 (ms)

其中工作负载 cpsync.log 是对一个 10M 大小的小文件较多的目录进行同步写

清华大学毕业设计论文

的复制操作，`untar1.log` 是解开一个 82M 的 tar 文件（Linux 源码包）的日志。这两个工作负载比较典型地反映了 LSTC 对密集的写操作的性能提高。模拟器选择的磁盘模型是 Quantum atlas10k，这是一个 10000RPM 的 SCSI 接口硬盘。

提交前返回的 LSTC 响应时间为 0，因为日志空间无限，所以不需要检查点操作，所有的写操作以内存和 CPU 速度进行，而不是磁盘速度，写日志磁盘的操作都在后台进行。实际运行中，日志空间不足引起的检查点操作会对性能有一些影响，但提交前返回的 LSTC 的响应时间依然应该很低，因为检查点操作也是在后台进行的。

提交后返回的 LSTC 在前台将数据直接写到磁盘，需要一定的响应时间。但由于将大部分寻道时间去掉，因此响应时间比原来要低近一个数量级。相比提交前返回的 LSTC，提交后返回的 LSTC 的可靠性要高得多，可以用在数据库和事务处理系统中，因此，由于能有这么大的性能提高，提交后返回的 LSTC 也是有广阔应用前景的。

清华大学毕业设计论文

第六章 结论及进一步工作

本文主要是关于一个新的基于日志的文件系统性能改进——日志结构事务缓存技术，这一技术能够在“办公/工程”环境及数据库及事务处理环境下提供比传统文件系统的缓存技术更好的性能和更好的数据可靠，并且可提高服务器的可能性，减少维护时间。本文的主要工作有：

1. 提出日志结构事务缓存技术，引入基于 IO 事务的日志的概念，以解决磁盘数据一致性问题，利用磁盘高速连续写入的特点，来大大提高磁盘的写性能。基于对磁盘、处理器、内存等设备的发展情况分析，以及与其它方法的比较，论证其具有更好的性能，更高的可靠性。
2. 使用基于跟踪记录（Trace）的磁盘子系统模拟方法对其性能进行模拟，初步结果证实日志结构事务缓存是有效的。
3. 进行了 Linux 下的具体实现的设计，完成了程序结构、算法、数据结构、操作等设计。在设计过程中，充分考虑了结构的合理性、模块化、可移植性等问题。对于一些可能遇到的技术问题及其解决进行了初步的讨论。

从一个纸上的方案到实际的实现有很长的路走，因此要使其成为一个可用的系统，还有很多工作要做。下一步的工作主要有：

1. 目前的 LSTC 设计主要用于单机系统，是否可以将其与其它技术相结合，并应用到机群、硬盘阵列上，需要进一步探讨。
2. 目前的模拟分析比较粗浅，工作负载也不是实际工作负载，因此需要在实际的不同类的工作负载上进一步验证算法，对于不足之处进行改进。当然，这需要有更好的模拟环境，因此要完成并不断改进模拟器。
3. 要将 Linux 下的设计方案变成原型实现，通过这个原型系统的实际运行来实际验证其性能与可靠性。

清华大学毕业设计论文

参考文献

- 1 J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: A case for log-structured file systems," *tech. rep., Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkley, Oct. 1988*
- 2 Yiming Hu, Qing Yang, "DCD-Disk Caching Disk: A New Approach for Boosting I/O Performance", *Annual International Symposium on Computer Architecture May 22-24 1996*
- 3 Satoru Kaneko, "Evolution of magnetic disk subsystems", *Journal of Magnetism and Magnetic Materials 134(1994) 217-222*
- 4 Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems 10, 1 (February 1992), 26-52.*
- 5 Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout, "Disk Scheduling Revisited," *Proceedings of the Winter 1990 USENIX Technical Conference, (January 1990).*
- 6 "SimOS, the complete machine simulator," <http://simos.stanford.edu>
- 7 "The DiskSim Simulation Environment", <http://www.ece.cmu.edu/~ganger/disksim/>
- 8 John H. Howard, Michael L. Lazar, ... "Scale and Performance in a Distributed File System", *ACM transactions on Computer Systems, Vol.6, No. 1, February 1988, Pages 51-81*
- 9 Murthy Devarakonda, Bill Kish, and Ajay Mhindra, "Recovery in the Calypso File System", *ACM Transactions on Computer Systems, Vol. 14, No. 3, August 1996, Pages 287-310*
- 10 Daniel Stodolskym Mark Holland, William V. Courtright II, and Garth A. Gibson, "Parity-Logging Disk Arrays", *ACM Transactions on Computer Systems, Vol. 12, No. 3, August 1994, Pages 206-235*
- 11 Tycho Nightingale, Yiming Hu, and Qing Yang, "The Design and Implementation of a DCD Device Driver for Unix"

清华大学毕业设计论文

致 谢

衷心感谢我的导师郑纬民教授，没有郑老师的悉心指导，不会有本文的完成。郑老师广博深厚的学术素养、踏实严谨的治学精神和勤奋忘我的工作作风都令我景仰，也将是我治学为人的楷模，更将在我读研期间激励我的学习工作。

衷心感谢杨广文老师，靳超、刘炜等学长，以及王小川等同学，是你们的帮助，使我能在这么短的时间内溶入教研组的集体中，对所从事的事情熟悉起来，并体会到与人合作的快乐。

周 枫

2000 年 6 月

于清华园

清华大学毕业设计论文

- ¹ J. Ousterhout and F. Douglis, “Beating the I/O bottleneck: A case for log-structured file systems,” *tech. rep., Computer Science Division, Electrical Engineering and Computer Sciences, University of California at Berkley, Oct. 1988*
- ² Yiming Hu, Qing Yang, “DCD-Disk Caching Disk: A New Approach for Boosting I/O Performance”, *Annual International Symposium on Computer Architecture May 22-24 1996*
- ³ Satoru Kaneko, “Evolution of magnetic disk subsystems”, *Journal of Magnetism and Magnetic Materials 134(1994) 217-222*
- ⁴ Rosenblum, M., Ousterhout, J., “The Design and Implementation of a Log-Structured File System,” *ACM Transactions on Computer Systems 10, 1 (February 1992), 26-52.*
- ⁵ Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout, “Disk Scheduling Revisited,” *Proceedings of the Winter 1990 USENIX Technical Conference, (January 1990).*
- ⁶ “SimOS, the complete machine simulator,” <http://simos.stanford.edu>
- ⁷ “The DiskSim Simulation Environment”, <http://www.ece.cmu.edu/~ganger/disksim/>