**Research and Implementation of the Optimization Method of Virtual Machine Images Storage**

**Based on Deduplication**

A Dissertation Submitted for the Degree of Master

**Candidate：Guoyu Zhao**

**Supervisor：Prof. Limin Xiao**

School of Computer Science & Engineering

Beihang University，Beijing，China

**中图分类号：TP309.3**

**论文编号：10006SY1006510**

硕 士 学 位 论 文

基于重复数据删除技术的虚拟机镜像存储优化方法的研究与实现

作者姓名 赵国玉 申请学位级别 工学硕士

指导教师姓名 肖利民 职 称 教授

学科专业 计算机系统结构 研究方向 文件系统

学习时间自 年 月 日 起至 年 月 日止

论文提交日期 年 月 日 论文答辩日期 年 月 日

学位授予单位 北京航空航天大学 学位授予日期 年 月 日

关于学位论文的独创性声明

本人郑重声明：所呈交的论文是本人在指导教师指导下独立进行研究工作所取得的成果，论文中有关资料和数据是实事求是的。尽我所知，除文中已经加以标注和致谢外，本论文不包含其他人已经发表或撰写的研究成果，也不包含本人或他人为获得北京航空航天大学或其它教育机构的学位或学历证书而使用过的材料。与我一同工作的同志对研究所做的任何贡献均已在论文中作出了明确的说明。

若有不实之处，本人愿意承担相关法律责任。

学位论文作者签名： 日期： 年 月 日

学位论文使用授权书

本人完全同意北京航空航天大学有权使用本学位论文（包括但不限于其印刷版和电子版），使用方式包括但不限于：保留学位论文，按规定向国家有关部门（机构）送交学位论文，以学术交流为目的赠送和交换学位论文，允许学位论文被查阅、借阅和复印，将学位论文的全部或部分内容编入有关数据库进行检索，采用影印、缩印或其他复制手段保存学位论文。

保密学位论文在解密后的使用授权同上。

学位论文作者签名： 日期： 年 月 日

指导教师签名： 日期： 年 月 日

摘 要

桌面虚拟化是一种建立在服务器虚拟化基础上的典型云计算应用，是建立在服务器虚拟化基础上的云应用。虚拟桌面架构通过服务器虚拟化，集中管理、统一配置，使得计算和数据更安全；同时，便于精简配置，降低IT系统成本、能耗以及管理开销。但虚拟桌面架构面临服务器虚拟化所带来的存储代价和性能上的挑战。

针对目前桌面虚拟化存储的现状，本文研究并设计了一套基于在线重复数据删除技术的用户态文件系统，对虚拟机镜像进行优化，取得的主要成果如下：

1. 基于FUSE架构，以重复数据删除技术为基础，实现了一套用户态的文件系统，实现对现有虚拟机的镜像存储进行优化。对于常用的虚拟机镜像文件实现了将磁盘占用空间降低30%以上的效果，并且保证了虚拟机的性能。
2. 实现了基于分布式Hash的多个节点之间的指纹和数据块并行存储方法。实现了在多个节点上并行存储指纹和数据的方法。在此基础上实现了一套基于重复数据删除技术的分布式文件系统，使得在不同宿主机节点之间的虚拟机镜像也可以进行重复数据删除。
3. 基于虚拟机磁盘I/O访问跟踪预测的自适应数据预取方法，在基于FUSE上层虚拟机的I/O访问进行跟踪和预测，在此基础上实现了对指纹数据和相应的数据的预取，以提高虚拟机的读写性能。

关键词：桌面虚拟化，存储优化，重复数据删除，文件系统

Abstract

Desktop Virtualization, as a typical application of cloud computing, is built upon the server virtualization. Through server virtualization, unified configuration and virtual desktop infrastructure make the computing and data more secure while reducing the cost of IT systems, energy consumption, as well as administrative overhead. However, virtual desktop infrastructure is facing the challenge of storage cost and performance generated by server virtualization.

To address the storage pressure by the rapid growing of virtual machine image , we had deveoped a user level inline deduplication file system based on the FUSE framework. The main research contents and achievements are as follows:

1. We have issused an user level inline deduplication file system to addressable the storage pressure by the rapid growing of numbers of virtual machines. Compared to an ordinary file system without deduplication, we show that our file system can save at least 30% of space of single VM image and even more of multiple VM images while achieving considerable run time performance.
2. We implemented the multi-nodes storage of fingerprints and data block with distibuted Hash table. A distributed file system aimed at inline deduplication was implemented based on this techolotgy, so that data and fingerprints on different nodes can be accessed simultaneously. And virtual machine images on different nodes can be deduplicated.
3. An Adaptive prefetching method was developed out based on the I/O patten we get from the file operations of FUSE. So that data block and fingerprints can be prefetched dynamicly to improve I/O performance of out system.

**Key words**: Desktop Virtualization, Storage Optimization, Deduplication, File System

目 录

[第1章 绪论 1](#_Toc342973328)

[1.1 研究背景与意义 1](#_Toc342973329)

[1.2 课题来源与研究目标 2](#_Toc342973330)

[1.3 研究内容 3](#_Toc342973331)

[1.4 论文主要研究成果 3](#_Toc342973332)

[1.5 论文组织结构 4](#_Toc342973333)

[第2章 国内外相关研究及发展趋势 5](#_Toc342973334)

[2.1 存储优化方法概述 5](#_Toc342973335)

[2.1.1 自动精简配置存储技术 5](#_Toc342973336)

[2.1.2 分层存储技术 7](#_Toc342973337)

[2.1.3 Diskreduce存储技术 7](#_Toc342973338)

[2.1.4 重复数据删除存储（Data deduplication） 8](#_Toc342973339)

[2.1.5 各种存储优化技术的比较 8](#_Toc342973340)

[2.2 重复数据删除技术 8](#_Toc342973341)

[2.2.1 基本概念 8](#_Toc342973342)

[2.2.2 相同数据检测技术 10](#_Toc342973343)

[2.2.3 备份系统 14](#_Toc342973344)

[2.2.4 虚拟机镜像存储优化 15](#_Toc342973345)

[2.3 FUSE文件系统架构 16](#_Toc342973346)

[2.4 本章小结 18](#_Toc342973347)

[第3章 系统设计 19](#_Toc342973348)

[3.1 概述 19](#_Toc342973349)

[3.2 系统设计 19](#_Toc342973350)

[3.2.1 设计思路 19](#_Toc342973351)

[3.2.2 镜像存储优化方法的实现层次分析 20](#_Toc342973352)

[3.2.3 总体模块架构 22](#_Toc342973353)

[3.2.4 总体工作流程 24](#_Toc342973354)

[3.3 基于重复数据删除技术的用户态文件系统架构 25](#_Toc342973355)

[3.3.1 系统整体架构 25](#_Toc342973356)

[3.3.2 客户端部分设计 26](#_Toc342973357)

[3.3.3 服务器端设计 30](#_Toc342973358)

[3.4 基于分布式Hash的并行存取方法 33](#_Toc342973359)

[3.4.1 指纹数据的分布与存储方法 33](#_Toc342973360)

[3.4.2 分块数据的分布与管理方法 34](#_Toc342973361)

[3.4.3 多节点之间的网络通信模块 34](#_Toc342973362)

[3.5 指纹数据的预取与缓存策略 35](#_Toc342973363)

[3.6 本章小结 35](#_Toc342973364)

[第4章 系统实现 37](#_Toc342973365)

[4.1 原型系统实现概述 37](#_Toc342973366)

[4.2 镜像文件存储方式 37](#_Toc342973367)

[4.3 详细实现细节 38](#_Toc342973368)

[4.3.1 getattr函数 38](#_Toc342973369)

[4.3.2 create函数 39](#_Toc342973370)

[4.3.3 open函数 40](#_Toc342973371)

[4.3.4 read函数 41](#_Toc342973372)

[4.3.5 write函数 42](#_Toc342973373)

[4.3.6 truncate函数 45](#_Toc342973374)

[4.3.7 缓存模块的实现 46](#_Toc342973375)

[4.4 本章小结 47](#_Toc342973376)

[第5章 系统的测试与分析 48](#_Toc342973377)

[5.1 测试环境 48](#_Toc342973378)

[5.1.1 硬件环境 48](#_Toc342973379)

[5.1.2 软件环境 48](#_Toc342973380)

[5.2 功能测试 48](#_Toc342973381)

[5.2.1 测试目标 48](#_Toc342973382)

[5.2.2 测试内容 48](#_Toc342973383)

[5.2.3 虚拟机镜像优化存储优化效果测试 51](#_Toc342973384)

[5.2.4 测试结果分析 52](#_Toc342973385)

[5.3 性能测试 52](#_Toc342973386)

[5.3.1 测试目标 52](#_Toc342973387)

[5.3.2 测试内容 52](#_Toc342973388)

[5.3.3 测试结果 53](#_Toc342973389)

[5.3.4 测试结果分析 58](#_Toc342973390)

[5.4 本章小结 59](#_Toc342973391)

[总结与展望 60](#_Toc342973392)

[参考文献 62](#_Toc342973393)

[攻读硕士学位期间取得的学术成果 65](#_Toc342973394)

[致 谢 66](#_Toc342973395)

图目录

[图 1 桌面虚拟化架构研究目标与内容 1](#_Toc342973396)

[图 2 自动精简配置原理 6](#_Toc342973397)

[图 3重复数据删除技术 9](#_Toc342973398)

[图 4 完全文件检测（WFD）技术 11](#_Toc342973399)

[图 5 基于FSP算法的相同数据检测技术 12](#_Toc342973400)

[图 6 基于CDC算法的相同数据检测技术 13](#_Toc342973401)

[图 7 基于滑动分块的相同数据块检测技术 14](#_Toc342973402)

[图 8 Linux文件系统模型 18](#_Toc342973403)

[图 9 FUSE文件系统[2] 18](#_Toc342973404)

[图 10 基于FUSE实现的重复数据删除系统 22](#_Toc342973405)

[图 11 系统总体模块架构 23](#_Toc342973406)

[图 12重复数据删除工作流程 24](#_Toc342973407)

[图 13 系统整体架构 26](#_Toc342973408)

[图 14 文件系统接口层（客户端）结构 27](#_Toc342973409)

[图 15 优化后的镜像文件存储结构 28](#_Toc342973410)

[图 16 基于CAS方法的数据读取流程 30](#_Toc342973411)

[图 17 基于CAS方法的数据写入流程 30](#_Toc342973412)

[图 18 分块文件管理 31](#_Toc342973413)

[图 19 服务器端整体架构 33](#_Toc342973414)

[图 20 Hash指纹映射方法 34](#_Toc342973415)

[图 21 指纹序列预取窗口 35](#_Toc342973416)

[图 22 Windows7虚拟机运行效果 49](#_Toc342973417)

[图 23 WindowsXP 虚拟机运行效果 50](#_Toc342973418)

[图 24 Red Hat Enterprise Server 6.0虚拟机运行效果 51](#_Toc342973419)

[图 25 原生虚拟机的文件读取性能 53](#_Toc342973420)

[图 26 优化存储之后的虚拟机文件读取性能 53](#_Toc342973421)

[图 27 原生虚拟机的写入性能 54](#_Toc342973422)

[图 28 优化存储之后的虚拟机文件写入性能 55](#_Toc342973423)

[图 29 原生虚拟机的重复读性能 56](#_Toc342973424)

[图 30 优化存储之后的虚拟机的重复读性能 56](#_Toc342973425)

[图 31 原生虚拟机的重复写性能 57](#_Toc342973426)

[图 32 优化存储之后的虚拟机的重复写性能 57](#_Toc342973427)

[图 33 镜像优化前后的读写性能对比 58](#_Toc342973428)

表目录

[表 1 各种存储技术优缺点对比 8](#_Toc342973429)

[表 2 实现层次对比分析 21](#_Toc342973430)

[表 3 CAS存储优化效果 51](#_Toc342973431)

# 绪论

## 研究背景与意义

鉴于用户正在变得越来越多和应用程序越来越大的事实，接触每一个桌面，为最终用户的系统安装、设置和更新应用程序是一项非常繁重的任务。另一方面，用户日益要求他们的PC尽可能快速地运行，无论是在线还是离线都要快速地运行，同时还要保证他们的计算环境的安全，这也是一个难点。而桌面虚拟化应用则可以满足这一需求。

桌面虚拟化作为一种典型的云计算应用，是建立在服务器虚拟化基础上的云应用。如图 1所示，桌面虚拟化基础架构在云中为用户提供远程的计算机桌面服务；服务提供者在数据中心服务器上运行用户所需的操作系统和应用软件，然后采用桌面显示协议将操作系统桌面视图以图像的方式传送到用户端设备上。同时，服务器对用户端的输入进行处理，并随时更新桌面视图的内容。用户通过任意可以接入Internet的终端，通过授权认证即可进行桌面计算。在桌面虚拟化计算模式下，所有计算都可以放在数据中心的服务器上，对终端设备的要求将大大降低，类似于平板电脑、智能手机、掌上电脑以及其他智能的手持设备这样的瘦终端就可以满足更广泛、更灵活的应用需求。桌面虚拟化架构通过服务器虚拟化，集中管理、统一配置，使得计算和数据更安全；同时，便于精简配置，降低IT系统成本、能耗以及管理开销。



图 1 桌面虚拟化架构研究目标与内容

桌面虚拟化旨在使工作场所(最终用户的应用程序、数据和设置)与它下面的架构层隔离开，为企业提供新的管理和部署的选择。Gartner在2008年发表的《Gartner拥有总成本报告》称，部署桌面虚拟化最多能够减少2800美元(包括大约1200美元的硬件/软件成本)并且避免了大部分技术支持的成本。此外，桌面虚拟化技术在提高硬件使用寿命、增加数据的安全性和容错性以及提高桌面使用的灵活性等方面均具有较大优势[1]。

桌面虚拟化面临着巨大的技术挑战。目前大多数大企业的IT部门要支持的桌面比服务器（无论是虚拟的还是物理的）要多。由于全球PC数量超过10亿台，所以桌面虚拟化存在巨大的机会。但是，更多的桌面虚拟镜像也代表着更多的数据存储开销，同时，PC向数据中心迁移的过程中，PC的环境需要保留。目前，在桌面虚拟化技术中，为了数据的安全，通常采用备份的存储原则：一份存储在本地，其他复本存储在不同的远程服务器。这样带来的就是巨大的存储空间支出与浪费。同时，在桌面虚拟化镜像中，未使用的存储空间全为0，并且相同的软件有相同的存储数据等，这些也带来了存储是巨大的空间支出与浪费。比如说有5000个桌面，每个都内置60GB的硬盘，成本将会无法承受。因此，对于数据的存储优化将是会是研究重点。考虑到不同的桌面虚拟镜像中存在很多重复的数据，如虚拟化的系统，虚拟化的应用程序等。现有技术通过自动精简配置和重复数据删除等技术，以降低存储成本、优化资源。

本课题来源于核高基重大专项“服务器操作系统研发及产业化”课题的“服务器操作系统性能优化技术研究”子课题中的“虚拟机镜像存储优化”子任务。

本文主要集中研究在桌面虚拟化应用过程中如何对数据存储进行优化，以减少存储空间的支出，提高存储效率，解决在桌面虚拟化的应用中的大规模虚拟机镜像带来的存储压力。

## 课题来源与研究目标

本课题来源于核高基重大专项“服务器操作系统研发及产业化”课题的“服务器操作系统性能优化技术研究”子课题中的“虚拟机镜像存储优化”子任务。

本文主要集中研究在桌面虚拟化应用过程中如何对数据存储进行优化，以减少存储空间的支出，提高存储效率，解决在桌面虚拟化的应用中的大规模虚拟机镜像带来的存储压力。

论文的研究目标主要是解决目前桌面虚拟化应用中大规模的虚拟机镜像文件带来的存储挑战。虚拟桌面架构面临服务器虚拟化所带来的存储代价和性能上的挑战。由于几十或上百个虚拟桌面整合到一台物理服务器上，随着用户数目的增长和用户需求的提高，服务器的存储空间需求会迅速扩展；如果一台物理服务器上有100个虚拟桌面，每个40 GB的容量就需要创建和维护100个虚拟磁盘镜像，总共需要4TB的存储容量。另外，一台物理服务器往往需要同时启动和运行多个虚拟机，如何优化虚拟机I/O访问来提高存储性能也是亟待解决的问题，利用虚拟磁盘镜像中存在大量数据冗余的特性；本文应用重复数据删除技术来优化虚拟桌面架构的存储空间利用率。

正对上述描述，本文的主要研究目标是：研究基于重复数据删除技术的虚拟机镜像存储优化方法，以减小大规模虚拟机镜像带来的存储开销。

## 研究内容

本论文的研究目标为：研究基于重复数据删除技术的虚拟机镜像存储优化方法，以减小大规模虚拟机镜像带来的存储开销。主要包括以下研究内容：

1. 基于在线重复数据删除技术的用户态文件系统架构；
2. 基于分布式Hash的多个节点之间的指纹和数据块并行存储方法；
3. 基于虚拟机磁盘I/O访问跟踪预测的自适应数据预取方法；

## 论文主要研究成果

针对目前桌面虚拟化应用中大规模虚拟机镜像文件所带来的磁盘开销，提出一种基于在线重复数据删除技术的虚拟机镜像存储优化方法，设计并实现了一套基于重复数据删除技术基础的用户态文件系统，实现了在对虚拟机监控器透明的情况下，降低虚拟机镜像文件所带来的磁盘开销，具体成果如下：

1. 基于在线重复数据删除技术的用户态文件系统架构

论文研究并实现了一个技术在线重复数据删除技术的用户态文件系统，实现对虚拟机的透明存取，利用内容寻址存储的方式进行重复数据删除，达到了很高的重复数据删除率，对于单个虚拟机镜像实现了将磁盘占用空间降低30%以上，并将虚拟机的读写性能损失控制在10%的范围内。

1. 基于分布式Hash的多个节点之间的指纹和数据块并行存储方法

实现了在多个节点上并行存储指纹和数据的方法。在此基础上实现了一套基于重复数据删除技术的分布式文件系统，使得在不同宿主机节点之间的虚拟机镜像也可以进行重复数据删除，并且由于经过重复数据删除之后，镜像文件的数据位于不同的节点上，从而不同节点之间的数据可以并行的读取和写入，提高了系统的性能。

1. 基于虚拟机磁盘I/O访问跟踪预测的自适应数据预取方法

在基于FUSE框架提供的I/O操作接口的基础上，对上层虚拟机的I/O访问进行跟踪和预测，以此为基础上实现了对指纹数据和相应的数据的预取，以提高虚拟机的读写请求响应时间，优化读写性能。

## 论文组织结构

本文后续部分的内容如下：

第一章为绪论，首先介绍了论文选题的背景、研究意义与课题来源，其次对论文的研究目标和研究内容做了概括性的介绍，最后对论文的贡献做了简要介绍。

第二章为国内外相关研究及发展趋势介绍，首先是对现有的存储优化方法进行概要的介绍及优缺点分析，其次重点介绍了重复数据删除技术以及其在存储优化中的应用。

第三章为系统设计，从整体的角度对系统进行了分析与设计，提出了预期的目标。首先对系统进行了整体模块划分，之后介绍了系统的整体流程，最后对本论文设计的核心研究点分别进行了详细地介绍。在此基础上逐步设计出一个完成的系统。

第五章对原型系统的实现进行了深入介绍，包括各个操作和模块的功能。

第六章对原型系统进行了深入的测试和验证，分别从存储优化效果和虚拟机的I/O性能两个方面进行了测试并的出了结论。

以上六章构成论文的主体，总结与展望部分对论文工作进行了深入总结，并指出今后的研究方向和需进一步完成的工作。

# 

# 国内外相关研究及发展趋势

## 存储优化方法概述

为了有效降低虚拟机镜像的存储开销，国内外研究人员开展了大量的研究。目前在桌面虚拟化应用领域主要的厂商有微软的基于操作系统的桌面虚拟化（VDI），思杰的Citrix XenDesktop ，VMware的view等[1]，但是现有的存储优化技术无法满足桌面虚拟化中大规模的虚拟机镜像数据的存储需求。目前在桌面虚拟化应用的存储优化技术方面，主要研究热点是将过去的自动精简配置、分层存储以及冗余数据删除技术等方法应用到桌面虚拟化领域中，如VMware view 4.5的重复数据删除引擎和分层存储。目前多采取的存储优化技术主要有自动精简配置、分层存储、DiskReduce技术以及重复数据删除技术等。

### 自动精简配置存储技术

在传统的存储系统中，当某项应用需要一部分存储空间的时候，往往是预先从后端存储系统中划分出一部分足够大的空间预先分配给该项应用，即使这项应用暂时不需要使用这么大的存储空间，但由于这部分存储空间已经被预留了出来，其它应用程序无法利用这些已经部署但闲置的存储容量。这种分配模式一方面使闲置的存储数量不断增加，系统总体拥有成本升高；另一方面用户不得不购买更大的存储容量，才能适应环境，成本进一步加大[3]。自动精简配置是一项新的容量分配的技术，不会一次性的划分过大的空间给某项应用，而是根据该项应用实际所需要的容量，多次的少量的分配给应用程序，当该项应用所产生的数据增长，分配的容量空间已不够的时候，系统会再次从后端存储池中补充分配一部分存储空间。

自动精简配置的核心原理是“欺骗”操作系统，让操作系统认为存储设备中有很大的存储空间，而实际上的物理存储空间则没有那么大。除了操作系统，应用也认为有很多剩余空间，尽管其中大部分空间都是虚拟出来的。而随着应用写入越来越多的空间，物理存储利用率也会越来越高。自动精简配置能够减少已分配但未使用的存储空间。其功能在于对数据块按需分配，而不是对所有存储空间需求进行预先分配。这项技术不会出现早先的有大量空间闲置，却还无法使用的现象。自动精简配置技术可以减少几乎所有空白空间，以避免利用率低下的情况出现，目前该技术能够降低10%的磁盘开销，并避免出现分配大量存储空间给某些独立服务器，却一直没有使用的情况。

如图 2所示，服务器识别存储设备的时候，看到的并不是真实空间，而是由自动精简配置虚拟出来的卷。而且只有当服务器写入存储的时候，真实的空间才会被分配。所有的虚拟空间都是零填充的缓冲区。当然，随着越来越多的数据被写入，实际存储空间也可以随需扩展。



图 2 自动精简配置原理

自动精简配置技术的优势在于，可以自动扩展已经分配的存储卷。同时，这种方式可以使用户使用接近100%的存储空间。因为所有的真实空间分配都与应用写入数据一一对应。另一个自动精简配置的优点在于和远程镜像结合的时候，可以节省镜像的空间和带宽。采用传统镜像方法，在最初部署块级别远程镜像的时候，整个磁盘卷都会被WAN传输。而使用了自动精简配置，管理员则可以只传输被使用了的空间。这可以极大的减少建立镜像的时间，同样的，自动精简配置也可以使用低成本的通信线路来传输数据，而且由于传输的数据空间减小，带宽的占用也有所降低。

空间利用率的提升：费用节省首先源于存储空间利用率的提高，所以只要简单的计算利用率具体提高了多少，就可以知道节省了多少存储费用。具体来说，你可以在部署自动精简配置之前，先计算一下某个应用占用的空间，然后于自动精简配置之后再计算一下占用的实际空间，这个空间的节省量就是自动精简配置能够达到的效率提升。一般而言，一个典型的自动精简配置能够将存储空间利用率从60%提升到80%，大约减少20%左右的存储费用。

自动精简配置在部署过程中可能遇到的挑战有以下两个方面[4]：第一个挑战在于，部署了自动精简配置之后最可能遇到的问题就是达到了最大存储容量。应用永远都以为还有很多容量，但实际上物理容量可能已经填满了。所以，一种可能的状况是，当实际空间已经没有空余，在新磁盘被添加之前，卷访问将被锁定；另一个挑战在于，并不是所有的数据库和文件系统都能收回已经删除了的空间。一些数据库和文件系统总是分配新的空间，而不是删除之后释放的空间。当频繁删除和建立文件的时候，很可能出现问题——自动精简配置系统将会持续要求更多的物理空间，而同时一些删除了数据的空间却不可用。所以我们最好在自动精简配置系统下对应用做一下测试，看看实际存储利用率到底有多大。我们也应该问问厂商，它有没有测量实际存储资源利用率的工具，或者有没有程序来阶段性的恢复已经删除的空间。

### 分层存储技术

分层存储（Tiered Storage），也称为层级存储管理（Hierarchical Storage Management），广义上讲，就是将数据存储在不同层级的介质中，并在不同的介质之间进行自动或者手动的数据迁移，复制等操作[5][6][7]。实际上，分层存储其实是一种在高速小容量层级的介质层与低速大容量层级的介质层之间进行一种自动或者手动数据迁移、复制、管理等操作的一种存储技术及方案。作为存储技术来说，分层存储与存储虚拟化也是相互交叉、互为因果的一种关系。分层存储的实现需要存储虚拟化的支撑，这样才能将分层的实现与具体应用分离。而虚拟化技术的实现又有分层技术的影响，所以分层存储是目前桌面虚拟化应用存储的主要研究技术。

分层存储是将不同类别的数据分配到不同类型的存储介质上，目的是提高存储效率，减少总使用成本(TCO)。存储分类基本上是根据应用程序的服务层面的要求，包括可用性、性能、保存要求、使用频率以及其他因素等。由于大量的日益增长的电子存储数据、最佳实践政策、自动持续性分配软件、特定数据匹配以及设备特点等因素，分层存储可以非常复杂。

分层存储可以采取多种形式，而且经常是存储架构增长的自然结果。通过将高速缓存分配给不同数据，以及/或通过使用不同特点的物理上分开的存储阵列，分层存储可以在一个阵列内建立（使用不同容量的或不同性能的磁盘驱动器）。分层存储号称可以节省多达50%的存储使用成本，使得它成为无差别容量配置的一种有吸引力的方案。

### Diskreduce存储技术

Disckreduce[8]技术主要针对HDFS存储优化而设计。在HDFS的数据存储中，为了数据的高可用性和高容错性，通常采取“一式三份”的存储方法，即一份本地，两份远程复制。这样就浪费了大量的存储空间。从本质上来说，DiskReduce是为HFDS设计的一种RAID技术。Diskreduce技术为了减少复本以降低存储开销，采取一份奇偶校验码和两份复本（一份本地，一份远程复本）的存储方式，可以很好的降低存储容量。在提高集群的存储利用率的同时，DiskReduce采用尽可能的延迟编码时间来提高多个副本的数据的写入速度。

### 重复数据删除存储（Data deduplication）

重复数据删除技术是目前虚拟化存储领域的研究重点，一般来说，重复数据删除是在子文件级发现冗余数据的一种方法，它用一个指针代替重复的数据，这项技术可以有效减少磁盘存储空间需求，并降低数据传输的网络带宽需求。

### 各种存储优化技术的比较

为了更清楚的对比各项存储优化技术的优缺点，表 1对上文中所提到的各种存储优化技术进行了直观的比较。

表 1 各种存储技术优缺点对比

|  |  |  |
| --- | --- | --- |
| 存储技术 | 优点 | 缺点 |
| 传统“一式三份” | 数据高可用性和高安全行性 | 磁盘占用空间太大 |
| 重复数据删除 | 有效的减少了备份时的冗余数据量，降低了存储空间 | 会带来一定的性能损失，容错性有所下降 |
| DiskReduce | 降低了备份数量，减少了存储量 | 没有对数据本身进行优化，容错方较“一式三份”差 |
| 分层存储 | 提高了数据的操作效率 | 没有降低实际的数据量 |
| 自动精简配置 | 提高了磁盘空间的利用率 | 没有降低实际数据量 |

## 重复数据删除技术

### 基本概念

重复数据删除技术（De-duplication）是一种数据缩减技术，通常用于基于磁盘的备份系统，旨在减少存储系统中使用的存储容量。它的工作方式是在某个时间周期内查找不同文件中不同位置的重复可变大小数据块。重复的数据块用指示符取代。高度冗余的数据集（例如备份数据）从数据重复删除技术的获益极大；用户可以实现10比1至50比1甚至更高的缩减比。而且，重复数据删除技术可以允许用户的不同站点之间进行高效，经济的备份数据复制。重复数据删除技术的方法如图 3所示。



图 3重复数据删除技术

重复数据删除技术是一种基于数据冗余度的数据缩减技术[10]。它首先利用一定的数据分块策略将大的数据对象划分为若干较小的数据块，并用数据块的加密Hash值作为指纹来标识它们；通过比对数据块的指纹，可以提高存储系统的空间利用率和网络系统的带宽利用率。按重复数据删除操作时机的不同，可以分为在线重复数据删除和离线重复数据删除。在桌面虚拟化架构中，由于即时缩减数据能够避免不必要的数据传输和写操作，使得在线重复数据删除更具有优势。按重复数据删除操作粒度的差异，可以分为文件级、块级重复数据删除。

随着重复数据删除技术的发展，该技术大量应用于存储备份和归档系统中，该系统中的重复数据删除模块负责对数据内容进行比对分析，查找出冗余数据，并将其元数据反馈给存储服务接口，最后将不重复的数据存入到存储介质中。重复数据删除技术主要分为以下两大类：[10]

1. 相同数据检测技术[11][12][13]。相同数据检测技术主要分为文件级相同数据检测和数据块级相同数据检测两种。完全文件检测（whole-file-detection，简称WFD）主要通对文件进行hash运算，通过比较hash值检测并消除内容相同的文件；细粒度的相同数据块检测主要通过固定分块（fixed-sized partition，简称 FSP）检测技术、可变分块（content-defined chunking，简称 CDC）检测技术、滑动块（sliding block）技术进行重复数据的查找与删除。
2. 相似数据检测和编码技术[14][15]。利用数据自身的相似性特点，通过 shingle 技术、bloom filter 技术和模式匹配技术挖掘出相同数据检测技术不能识别的重复数据；对相似数据采用 delta 技术进行编码并最小化压缩相似数据,以进一步缩减存储空间和网络带宽的占用。

### 相同数据检测技术

相同数据检测计数是通过将数据进行划分，找出相同的部分，并以指针代替相同 数据存储，达到减小存储空间的目的。

1. 完全文件检测技术

WFD技术是以文件为粒度进行重复数据删除的方法[16]。如图 4所示，首先对整个文件进行Hash计算，然后将计算结果的到的Hash值同已存储的Hash值进行比较，如果检测到相同的值，则仅将文件用指针代替，否则存储新的文件，并更新Hash库。将Hash算法引入到重复数据删除技术中是为了利用Hash值可以唯一标识文件内容的特性，通过Hash计数，文件数据被标识为一个固定大小的Hash值，比较该Hash值就可以判别重复数据。目前MD5和SHA-1是应用最广泛的Hash算法，两者的抗冲突性都比较好。

WFD技术有两个优势：（1）在普通硬件条件下计算速度很快，研究表明，SHA-1是83M/S，而MD5是227MB/S；（2）可以检测到完全相同的文件，能节省较大的存储空间。但该方法也存在以下缺点：（1）对于较大的数据集，需要比较的范围太大，耗时太多；（2）不能检测文件内部的相同数据。因此，在实际应用中，WFD技术很少被使用。



图 4 完全文件检测（WFD）技术

1. 固定分块检测计数（FSP）

完全文件检测技术不能检测文件内部存在的内容相同的冗余数据，块级重复数据监测应运而生，FSP算法是通过把文件分割成大小相同的数据块，然后通过对数据块进行Hash来判断相同数据的方法[18][19][20][21]。如图 5所示。该方法分为3个步骤：（1）提供一个已经预先定义好的数据块大小，所有文件均按照这个固定大小进行划分；（2）每个划分好的数据块均通过Hash算法（MD5或SHA-1）计算出一个指纹来唯一代表这块数据；（3）将计算的Hash值与已存储的Hash序列进行比较，如果检测到相同的值，则删除其所代表的数据块，否则，存储新的数据块并更新Hash值。



图 5 基于FSP算法的相同数据检测技术

基于FSP算法的重复数据删除技术已经应用在很多领域，并具有以下特征：（1）缩减存储空间；（2）减少网络传输的数据量，如虚拟机镜像存储优化系统，通过该技术加速了在低带宽网络上的数据传输并改进了内存的性能。此外，基于FSP算法的重复数据删除技术还可以提供很高的处理速度，适合于在交互性的环境中应用。FSP算法的主要局限性在于：对编辑和修改的序列很敏感，对于插入问题和删除问题的处理十分低效。不能智能地根据文件自身内容的变化和文件之间的关联关系进行调整与优化。

1. 可变分块技术

针对FSP算法在插入和删除数据时的局限性，研究者们提出了可变块大小的划分的检测技术。CDC算法是应用Rabin指纹[11][20][21]将文件分割成长度大小不一的分块策略。与固定分块不同的是，他对文件的划分是基于文件内容的，因此数据块的大小是可变的，如图 6所示。其过程分为两步：（1）一个文件按照CDC算法分为若干数据块。CDC算法首先从文件头开始，将固定大小的（互相重叠）的滑动窗口中的数据看成是文件的各个组成部分。在窗口的每个位置，该窗口中的数据的一个指纹被计算出来，鉴于Rabin指纹的计算高效性以及Rabin指纹函数的随机性（对任意数据呈现均匀分布），因此经常被用来计算滑动窗口内容的指纹。当指纹满足某个条件时，则把此时的窗口位置作为分块的边界，重复这个过程直到整个文件数据都被分块。（2）对于划分出的每个块用Hash函数计算出它的指纹并与已存储的Hash指纹数据进行比较，若检测到相同的Hash指纹，则删除其代表的数据块，否则，存储新的数据块并更新已存储的Hash指纹。



图 6 基于CDC算法的相同数据检测技术

根据CDC的算法特性，无论是插入还是删除一小部分字节都只会影响到两个分块，其余的保持不变，及CDC算法在两个相似的对象之间（中间相差几个字节）可以检测出更多的冗余数据。但CDC算法也存在一定的局限性，划分的粒度大部分取决于期望块的设定，如果该值设置的比较小，虽然粒度较细，重复数据的查找也较为精确，但每个块的额外信息存储会造成很大开销；反之，若该值设置的比较大，则粒度过粗，影响重复数据删除的效果。因此，如何权衡精确查找和额外开销是CDC算法的一个难点。

1. 滑动块检测技术

滑动块检测技术结合了固定分块和可变分块的优点[22][23][24][25]，块大小固定，管理简单。对大的分块，CDC的重复数据检测性能较好，而滑动分块技术对细粒度匹配更适用。如图 7所示，基于滑动分块的额相同数据监测技术共有4步：（1）利用校验和（checksum）函数和固定大小的滑动窗口来计算文件对象每个重叠块的求和校验值；（2）对于每个分块，比较校验值与先前存储的校验值；（3）若匹配，则利用更严格的MD5算法对数据块进行Hash运算，并将MD5 Hash值与先前存储的Hash值进行比较，从而进行冗余数据监测。如果检测到冗余，则将其记录后，滑动窗口越过这个冗余块继续前移。而且先前已被划分的块和最近被检测到的冗余之前的这个碎片需要被记录并且存储；（4）若校验和或者Hash值不能匹配，则滑动窗口继续前移。如果滑动窗口已经滑动了一大块大小的距离，仍然不能够匹配到任何已经被存储的块，则需要对这个块进行求校验和和MD5 Hash计算，并且存储在各自的表中，用作以后的块的比较。



图 7 基于滑动分块的相同数据块检测技术

滑动分块检测技术的特点是对插入和删除问题很高效，并且可以检测到更多的冗余数据。对于插入问题来说，如果一小部分字节被插入到一个对象当中，则只有周围的块会发生改变，其他的分块仍然可以被识别出来并匹配，并且，一个长度等于插入的字节数的碎片会产生出来；对于删除问题来说，删除一小部分字节也会产生一个长度等于块大小减去被删除部分字节长度的碎片，其他块不受影响。但滑动分块也存在一个不足：在插入和删除问题中都会引入碎片。

### 备份系统

重复数据删除技术目前大量应用于数据备份与归档系统，因为对数据进行多次备份后，存在大量重复数据，非常适合这种技术[27]。事实上，重复数据删除技术可以用于很多场合，包括在线数据、近线数据、离线数据存储系统，可以在文件系统、卷管理器、NAS、SAN中实施。重复数据删除技术也可以用于数据容灾、数据传输与同步，作为一种数据压缩技术可用于数据打包重复数据删除技术可以帮助众多应用降低数据存储量，节省网络带宽，提高存储效率、减小备份窗口，节省成本。重复数据删除技术很好的解决了数据备份过程中重复数据占用大量的存储空间的问题。

Venti [20] 是最早采用内容寻址存储的思想（Content Addressable Storage，简称CAS）的思想来实现的一个基于重复数据删除的归档数据备份系统，它采用一个唯一的Hash指纹来标记数据块的内容。然而，Venti本身不是一个完整意义上的文件系统，而只提供块级别的存储接口。Venti系统中的分块大小是固定的，但实际的分块大小可以根据应用的需要改变，分块大小可设置为512B至56KB之间。Hash指纹采用160字节长度的SHA-1算法。

Foundation[12]是建立在Venti基础之上的另一个基于CAS的备份系统。它采用一个Bloom Filter来加速判断新产生的指纹数据和对应的数据块。Foundation主要是面向个人用户的。通过对整个磁盘生成快照（snapshot）来保存和恢复用户数据。相对于Venti，Foundation只需要很低成本的硬件资源。作者指出，Foundation生成快照时的吞吐率可以达到21 MB/s，在恢复快照时的吞吐率可以到到14 MB/s。并且，在同等的硬件条件下可以比Venti实现更好的性能。

C. Policroniades等[2]比较了基于WFD、FSP以及应用Rabin指纹的CDC三种算法在重复数据删除中的压缩效果。Hash算法均采用SHA-1。对于 300 MB 的文件，在 500MHz Pentium 3 上，依次花费 62s（4.8 MB/s）、71s（4.2 MB/s）、340s（<1 MB/s），其中 Rabin fingerprint 包含了确定边界后计算 SHA-1 的时间。从实验结果来看基于Rabin指纹的CDC算法的压缩比最高。

Data Domain Deduplication File System [13]（以下简称为DDFS）是Data Domain公司开发的商用重复数据删除系统，采用变长分块技术，可以打到很高的压缩比和吞吐率。该系统的基本存储单元不是传统意义上的数据块，而是把一系列数据块进行分装后的容器，称为Container[13]，一个Container包含有元数据和数据部分，并且Container的大小是固定的。针对内容空间无法容纳所有指纹数据的问题，DDFS采用三级查询机制：Bloom Filter过滤、Hash缓冲查询和Hash文件查询。首先在内存中的Bloom Filter中进行查找。一个Bloom filter用一个m位向量来概括在块索引中n个块指纹值的存在信息。如果Bloom filter指出这个块不存在，则这个块一定不存在。如果Bloom filter指出该数据块存在，表明该数据块可能存在，再到Hash缓存中进行查找，如果存在则说明该数据块存在，否则再到磁盘上去查询。对于数据在磁盘上的组织采用了基于流的块排列技术，以有效利用数据的局部特性，提高缓存的命中率。利用局部性的缓存来减少随机I/O对系统性能带来的损害。数据块按照之前的顺序被存储在同个Container当中，以保持一定程度的空间局部性。利用空间局部性做缓存的预取，每次把Container的元数据预取到Cache当中。

HYDRAstor [14] 和 HydraFS[15] 共同构成了一个存储系统，分别属于后端（block-level store）和前端（file system）。HYDRAstor 是第一个正式商用的，可实现重复数据删除的后端存储系统，具备可扩展性和高性能，并且具备数据容错恢复功能。HydraFS则是在HYDRAstor技术上实现的一个完整意义上的能够实现重复数据删除的文件系统。

### 虚拟机镜像存储优化

目前，重复数据删除技术已被广泛应用于虚拟机环境。通过分析虚拟机磁盘镜像的重复数据删除效果，发现重复数据删除技术能够大量地缩减虚拟机磁盘镜像存储空间[28][29][30]。

Nath等研究了基于CAS的存储系统对虚拟机镜像存储的影响，并且发现在基于虚拟机的存储系统中进行数据块级别的重复数据删除能够充分利用存储和网络资源[31][32][33]。然而他们的工作主要集中在对虚拟机镜像的存储效率和网络负载的研究，并没有涉及在访问虚拟机镜像时的读写吞吐率，也没有对经过CAS优化后的虚拟机的读写性能与原生虚拟机读写性能进行对比。

Liguori等在Venti的基础上实现了一个可用于KVM虚拟机的重复数据删除系统[15]。通过重新实现QEMU的block层的API[30]，使之可以直接调用Venti提供的读写接口。在此基础上对不同操作系统的虚拟机磁盘镜像进行重复数据删除效果比较分析，并测试重复数据删除技术对虚拟机启动和读性能的影响[15]。然而，他们的工作主要集中于研究CAS存储对虚拟机镜像的优化效果，以及加入CAS存储后对虚拟机启动时间段的影响。由于Venti自身的局限性，在读写吞吐率方面并没有实现很好的性能。

Zhang等人研究了重复数据删除技术对虚拟机在线迁移的加速效果[32]。

LiveDFS[7]是香港中文大学开发的在OpenStack中部署的虚拟机镜像在线重复数据删除系统。它采用4KB大小的固定分块，Hash算法使用MD5。LiveDFS基于Ext3文件系统实现，并作为一个内核模块运行在内核中。因此，LiveDFS支持在线重复数据删除，并且是POSIX兼容的。在Ext3文件系统中，若干Block被分成一个Group。同样的，为了减少磁头的偏移带来的系统开销，LiveDFS采用把指纹和数据放在同一个Group中的方案，并采用指纹预取策略，当一个Group中的指纹被访问时，整个Group中的指纹数据都会被加载到内存中。LiveDFS在单机上实现了很好的性能，尤其是写入的性能，可以达到和本地Ext3文件系统同样的性能。由于是基于Ext3文件系统实现，LiveDFS只能实现在单机上的重复数据删除，并且只能挂载到特定分区，并且在内存中建立指纹过滤器，因此需要较长的挂载时间。基于Ext3实现的内核文件系统在很大程度上要依赖于内核的数据结构，因此并不能很好的适应内核数据结构的变化。由于实现为单个节点上的文件系统，LiveDFS无法对不同节点上存储的虚拟机镜像进行重复数据删除。即无法检测不同节点上存储的虚拟机镜像文件之间的重复数据。

## FUSE文件系统架构

FUSE[2]（用户空间文件系统）作为类UNIX系统平台上可加载的内核模块，允许非特权用户创建功能完备的文件系统，而不需要重新编译内核，Linux文件系统模型如图 8所示。VFS（虚拟文件系统转换器）是整个Linux文件系统模型的关键，VFS层是对所有具体文件系统的一个抽象表示，每个具体的文件系统都会实现VFS提供的标准文件操作接口。在VFS中，每个文件都用一个全局唯一的i-节点（inode）来表示。FUSE作为Linux内核的一个具体文件系统实例，FUSE模块仅仅一个提供Kernel模块的接入口，而本身的主要实现代码位于用户空间中。对于读写虚拟文件系统来说，FUSE是个很好的选择。FUSE包含包含一个内核模块和一个用户空间守护进程，将大部分的VFS调用都委托一个专用的守护进程来处理。FUSE用户空间文件系统与真实的文件系统不同，它的supper block（文件系统超级块），inode，dentry（目录项）等都是由内存虚拟而来，具体在物理磁盘上存储的真实文件结构是什么，它不关心，且对真实数据的请求通过驱动和接口一层层传递到用户空间的用户编写的具体实现程序里来，这样就为用户开发自己的文件系统提供了便利，这也就是所谓的“用户空间文件系统”的基本工作理念。

FUSE起初是为了研究AVFS(A Virtual Filesystem)而设计的，而现在已经成为SourceForge的一个独立项目，目前适用的平台有Linux,，FreeBSD，NetBSD，OpenSolaris和Mac OS X。

FUSE具有的特点如下：

1. 库文件和 API简单，极大地方便了用户的使用
2. 安装简便，不需要加补丁或者重新编译 kernel
3. 执行安全，使用稳定
4. 高效，相对于其它用户态文件系统实例
5. 非特权用户可以使用

FUSE的文件操作处理流程如图 9所示。



图 8 Linux文件系统模型



图 9 FUSE文件系统[2]

使用FUSE，可以像可执行二进制文件一样来开发文件系统，它们需要链接到FUSE 库上。换言之，这个文件系统框架并不需要了解文件系统的内幕和内核模块编程的知识。

## 本章小结

本章针对论文的研究内容，对国内外相关研究现状进行了介绍。主要介绍了现有的存储优化方法，并重点对重复数据删除技术技术进行了深入介绍。之后又介绍了重复数据删除技术在备份系统以及虚拟机镜像存储方面的应用。最后介绍了Linux内核文件系统组件以及FUSE用户态文件系统架构。

# 系统设计

## 概述

为了节省硬件的采购成本和经常性的管理开销，虚拟化技术被广泛应用于企业数据中心的服务器资源管理，使得每台物理服务器能够整合几十或上百台虚拟机，从而提高IT资源的利用率。这使得每台服务器需要管理大量的虚拟磁盘镜像，而服务器的本地磁盘不足以提供如此大的存储容量，需要将这些磁盘镜像存放到一个集中的大容量存储池，并由多台服务器所共享。当服务器需要启动某个虚拟机时，需要从共享存储系统中将虚拟机磁盘镜像读到服务器的本地磁盘来运行。

目前桌面虚拟化产品中大规模虚拟机镜像存储带来了数据重复存储的问题，增加了共享存储系统的存储空间压力，需要在现有的虚拟机镜像存储基础上，设计并实现针对虚拟机镜像文件的存储优化方法，以避免重复数据的多次存储。本论文以重复数据删除技术为基础，采用内容寻址存储（CAS）的方式，建立面向虚拟机镜像的存储优化方法，通过检测数据的内容相似性以合并存储重复数据，达到降低虚拟机镜像存储开销的目的。本论文在基于FUSE开源文件系统架构的基础上，开发出一套完整的用户态文件系统，因此，优化方法对虚拟机监控器和虚拟机操作系统来说是透明的。在此基础上，对文件系统进行了并行性和缓存、预取方面的优化，在降低存储开销的同时可以保证不对虚拟机的性能造成显著的影响。

## 系统设计

### 设计思路

目前，重复数据删除技术已被广泛应用于虚拟机环境。通过分析虚拟机磁盘镜像的重复数据删除效果，发现重复数据删除技术能够大量地缩减虚拟机磁盘镜像存储空间。Nath等人发现在基于虚拟机的存储系统中进行块级重复数据删除能够充分利用存储和网络资源[34]，Liguori等人对不同操作系统的虚拟机磁盘镜像进行重复数据删除效果比较分析，并测试重复数据删除技术对虚拟机启动和读性能的影响[15]，Zhang等人研究了重复数据删除技术对虚拟机在线迁移的加速效果[32]。本系统主要研究如何利用重复数据删除技术来优化虚拟桌面存储系统的存储利用率和I/O性能。

典型的存储系统都是对数据进行分块，分块大小可以是固定大小或者是变长的。大部基于CAS的备份系统都是对数据进行变长分块，以最大限度的发掘重复数据块，因此可以打到很高的重复数据删除比例。然而，本论文主要研究和实现基于CAS的方式实现对虚拟机镜像的重复数据删除，并采用FSP算法对文件进行固定长度分块。主要是基于以下方面：

1. 与备份归档系统不同，虚拟机镜像文件很少会发生在文件中间插入或者删除数据块而导致大规模的偏移量改变的情况。并且，对于FSP算法来说，修改数据块的操作只会影响被修改的数据块，而不会对其他不相关的数据块产生影响。
2. 对于虚拟机镜像的存储来说，CDC算法和FSP算法的重复数据删除效果差别很小[16]。对于虚拟机镜像文件来说，FSP算法的重复数据删除效率甚至会高于基于CDC算法的方法。并且，由于分块大小固定，FSP算法在分块和检测重复数据的性能上有着绝对的优势，因此本系统采用基于FSP的算法来实现对虚拟机镜像的重复数据删除。从而使得本论文所研究的优化方法既可以保证对虚拟机镜像的重复数据删除效果，又可以使得优化后的虚拟机系统性能得到保证。
3. 绝大多数的CPU（例如，Intel x86等）只支持固定大小的内存分页，因此大部分的文件系统也都是采用固定大小的分块以最大化的优化内存页面的使用以提高效率。
4. 由于本论文研究的虚拟机镜像存储优化方法支持在线重复数据删除功能，对数据块的分配和删除操作会频繁发生。因此，需要对空闲的数据块进行管理以便于重新分配从而最大化磁盘空间的利用率。基于FSP的固定分块算法在对镜像文件进行切分之后产生的数据块大小是固定的，因而，在空闲块的管理和利用方面具有更大的优势和更高的效率。

综合以上论述，FSP算法在针对虚拟机镜像存储优化方面比基于内容的CDC算法更具有优势，因此，本文所研究的基于重复数据删除技术的虚拟机镜像存储优化方法采用FSP算法来实现对镜像文件进行定长分块。为了与CPU的内存页面大小保持一直以最大化的提高性能，数据分块大小确定为4KB，即一个内存页面的大小。Hash算法采用128位的MD5算法，MD5算法具有运算速度快，抗冲突性好等特点（运算速度可达250MB/s），并且MD5的位数是2的整数次幂，因此，有利于内存对齐以及缓存方面的优化。

### 镜像存储优化方法的实现层次分析

为了实现对虚拟机镜像的存储优化，需要改变虚拟机的I/O操作流程。即，需要在虚拟机的整个I/O操作的层次上进行修改，加入基于重复数据删除技术的存储逻辑，以实现对虚拟机镜像存储的优化。一般来说镜像存储优化的实现层次可以分为三种：文件系统上层、文件系统接口层和文件系统存储层。表 2详细比较了在各个层次上实现优化方法的难度和优势。

表 2 实现层次对比分析

|  |  |  |
| --- | --- | --- |
| 实现层次 | 优点 | 缺点 |
| 文件系统上层 | 与所使用的文件系统和存储系统无关，便于移植 | 需要修改虚拟机监控器的I/O层，并且需要处理I/O请求的转发，同时容易形成单一结点瓶颈 |
| 文件系统接口层 | 实现在计算结点，避免单一结点瓶颈问题；有效管理本地缓存以提高CAS读写的性能 | 需要根据具体的文件系统，实现客户端接口 |
| 文件系统存储层 | 对接口层和应用程序层透明 | 需要修改具体的文件系统，可移植性差 |

综合考虑三种实现方法的实现难度和优势，本系统在文件系统接口层实现重复数据删除存储过程，采用开源文件系统架构FUSE[2]来实现对相关功能的封装。

为了保证原型系统的可移植性，降低系统开发的复杂度，本系统基于FUSE框架实现对CAS存储虚拟机镜像过程的封装，此过程对共享存储系统和上层的应用程序均保证透明性，以最大限度的提高系统的可移植性。基于FUSE的重复数据删除系统如图 10所示。

原型系统以FUSE架构为基础在宿主机上实现访问共享存储系统的客户端，对于虚拟机镜像文件和存储系统中的数据块之间的对应管理均在该客户端实现。同时，宿主机具有缓存功能，存储最近访问的数据块和虚拟机镜像文件对应Hash值序列。虚拟机镜像文件的Hash序列和数据块均以文件的形式存储在共享存储系统中，客户端通过访问共享存储系统的接口远程读写相应的数据。

宿主机的虚拟机（VM）实例由远程的桌面客户端启动并操作，在此过程中，每个VM通过虚拟机监控器向宿主机所挂载的共享存储目录中的相应文件读写数据，对应的I/O系统调用所产生的相应VFS调用在内核空间被FUSE的内核模块捕捉；然后，VFS调用被转发到用户态的FUSE接口程序中，该程序重新实现了每种VFS调用的对应应答流程，在此过程中按照CAS的存储思路与远程的共享存储系统交互，重点完成了从虚拟机镜像文件到实际在共享存储系统中数据存储之间的转换，即用户所看到的每个虚拟机镜像文件（如数据盘文件、内存文件等）被划分成多个文件存储（每个文件存储一个或多个CAS数据块）。

在上述过程中，从用户的角度看，用户是在操作本地的桌面系统；从虚拟机监控器的角度看，所操作的虚拟机文件和原存储方案（如直接存储在NFS、Ceph中）相同；从共享存储系统的角度看，所存储的虚拟机镜像文件被分成若干小的文件分别存储。

在宿主机上实现的基于FUSE的用户态文件系统布局和I/O操作流程如图 10所示，图中的加粗方框中的Deduplicaion是重复数据删除逻辑所要实现的地方，程序全部位于用户态，因而本文所研究的基于重复数据删除技术的虚拟机存储优化方法具有很好的可移植性以及通用性。



图 10 基于FUSE实现的重复数据删除系统

### 总体模块架构

本文所研究的基于重复数据删除技术虚拟机镜像存储优化方以开源架构FUSE框架为基础，实现了一套完整意义上的用户态文件系统。系统的总体模块架构如图 11所示。与现有的并行文件系统与分布式文件系统类似，本系统主要分为客户端、服务器和网络通信三个部分。其中，客户端与服务器可以运行在相同或者不同的物理机器上，客户端与服务器之间通过网络进行通信和交换数据。虚拟机运行在安装有客户端的机器上，并且虚拟机镜像需要复制到用户态文件系统所挂载的目录当中。经过CAS方法分块的数据存储在服务器当中。



图 11 系统总体模块架构

1. 客户端模块设计

客户端的设计主要包括对FUSE提供的文件操作接口进行实现。在FUSE提供的文件操作接口中实现基于重复数据删除技术的存储逻辑。包括基于CAS的存储过程，对经常访问的指纹和分块数据进行缓存，通过网络与服务器进行交互（分块数据的读取和写入），以及对指纹数据进行预取等功能。

1. 服务器端模块设计

服务器端的设计主要涉及到对分块数据的存储和管理。包括：建立指纹以及对应数据块之间的对应关系，对指纹数据进行索引，判断重复数据，对分块数据进行管理（包括写入，修改，删除等功能），通过网络与客户端机器进行通信，响应客户端的读写请求等功能。

1. 网络通信模块设计

网络通信模块的设计主要涉及到提供一种能在客户端和服务器端进行数据传输的高效方法。包括指纹和数据块在各个服务器之间的分布方法。

### 总体工作流程

随着数据存储量的快速增长，当前测存储优化方法难以满足降低存储开销的需求。本论文针对桌面虚拟化应用中存储开销大的问题，需要研究并实现基于重复数据删除技术的虚拟机镜像存储优化方法。

需要研究并实现一种在线的重复数据删除系统以无缝的应用在现有的桌面虚拟化架构中，从而解决虚拟机镜像带来的存储空间开销的问题。同时在应用重复数据删除技术进行去重的过程必须保持对上层虚拟机应用的透明。在此基础上需要对重复数据删除的各个环节进行优化以保证系统的性能。基于重复数据删除技术的镜像存储优化方法需要客户端、服务器以及网络通信部分协调完成。总体的的工作流程如图 12所示。



图 12重复数据删除工作流程

客户端需要完成的工作包括对镜像文件进行分块，计算分块的指纹（MD5值），并对本地的指纹序列进行管理。在计算出数据指纹之后，根据指纹和对应的数据块对服务器发出相应的读写请求。服务器端完成的工作包括根据指纹对数据进行存取，即重复数据删除过程。

在服务器端，首先需要根据客户端发来的指纹数据对指纹数据库进行检索，如果对应的指纹已经存在则直接更新相应数据块的引用计数即可，若在指纹数据库中没检索到对应的指纹，则在服务器存储池中新分配一个数据块，新分配的数据块被用来存储对应的数据，并建立相应的“指纹——数据”对应关系，存入服务器端的指纹数据库中。

## 基于重复数据删除技术的用户态文件系统架构

目前桌面虚拟化产品中大规模虚拟机镜像存储带来了数据重复存储的问题，增加了共享存储系统的存储空间压力，需要在现有的虚拟机镜像存储基础上，设计并实现新的存储优化方法，以避免重复数据的多次存储。本论文以CAS为基础，建立面向虚拟机镜像的存储优化方法，通过检测数据的内容相似性以合并存储重复数据，实现重复数据删除的目的，以达到降低虚拟机镜像存储开销的目的。

在降低存储开销的同时不能对虚拟机的性能造成显著的影响。并且，优化方法对虚拟机监控器和VM 操作系统来说应当是透明的。因此，需要基于重复数据删除技术实现一套文件系统。考虑到内核文件系统的复杂性，本论文采用基于FUSE实现一套基于重复数据删除技术的用户态的文件系统。实现对虚拟机的透明访问。

### 系统整体架构

基于重复数据删除技术的用户态文件系统的整体架构分为客户端和服务器两个部分以及客户端与服务器之间的通信模块。客户端和服务器端可以运行在相同或者不同的计算机节点上。客户端和服务器端之间通过以太网进行通信。客户端运行在运行虚拟机的节点上。在客户端实现了基于FUSE的用户态文件系统，因此保证了在对虚拟机镜像文件进行重复数据删除的同时对虚拟机监控器和虚拟机操作系统透明的。整个系统结构如图 13所示。



图 13 系统整体架构

### 客户端部分设计

#### 客户端整体架构

客户端的主要功能是要实现对上层应用的透明存取，并在此基础上对由应用层发出的I/O请求进行处理。客户端的整体结构如图 14所示，客户端实现方面的主要工作体现在以下几个方面：

1. 整体框架

利用FUSE实现对标准I/O调用封装，以实现对上层虚拟机的透明对各个I/O系统调用进行重新实现。关键的文件系统操作包括文件读取，文件写入，文件打开，文件关闭等（图 14中对应的read，write，open，close等）。基于CAS方式的文件操作主要体现在文件的读取和写入操作方面。在文件读取和写入时，首先要对元数据文件进行操作。首先，从元数据文件中获得文件的大小信息，其次，根据文件的读写请求的偏移量信息，计算出对应数据块的指纹在元数据文件中的位置，最后，从元数据文件中读出指纹序列，按照CAS的方式对数据进行读取和写入，在写入过程中进行重复数据删除技术操作。

1. 元数据文件管理

元数据包含文件大小以及相应的指纹序列，元数据以普通文件的形式存储在客户端本地磁盘中，具体的存储格式如图 15所示。元数据文件包含的信息主要有原始镜像文件大小以及镜像文件被分块以后计算出的Hash序列。元数据需要提供Hash序列初始偏移量等信息，在文件读取和写入操作时，可以根据Hash序列初始偏移量结合读写请求偏移量等信息确定对应数据块的Hash所存在的位置，从而可以对Hash序列进行读取写入以及修改等操作。

1. 根据读写请求偏移量以及大小等信息动态决策从服务器预取数据的策略以加快存储速度。具体做法如下：

设置Hash序列的预读窗口，每次读取Hash序列时，把整个窗口内的Hash序列一块读出到用户态的缓冲区中，下次读取是可进行判断，若对应位置的Hash已存在于缓存中，则直接读取。由于内核本身具有文件页面缓存的功能，即对经常读取的文件最大限度的缓存到内存里面。



图 14 文件系统接口层（客户端）结构

#### 元数据文件存储结构

经过优化之后的镜像文件有两部分组成，一部分是与源文件同名的Hash序列文件（元数据文件），元数据文件的结构如图 15所示。为了加快对Hash数据的读取，结合Hash序列文件一般体积较小的特点，把元数据文件直接存储在宿主机的磁盘中。在读取Hash序列的时候就可以省去对远程服务器的读写，因此可以大幅度提高元数据操作以及Hash序列读取和写入的效率；另一部分则是文件的实际数据，文件的实际数据被分割成大小相同的数据块存储在服务器端，在服务器端保存有Hash和数据之间的映射关系（指纹数据库），运行虚拟机的宿主机通过安装在宿主机上的客户端与存储实际数据的服务器端进行交互，这样可以根据Hash值对相应的数据进行操作。

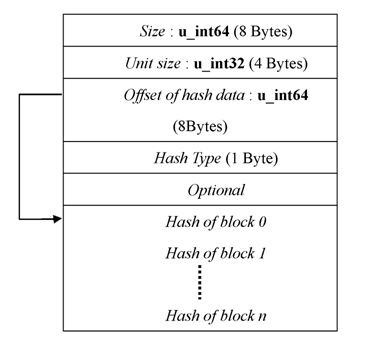


图 15 优化后的镜像文件存储结构

元数据文件中的Hash序列的偏移量和对应数据偏移量之间的对应关系由如下公式表示：

其中，为Hash值在元数据文件中的偏移量，表示原始数据块在原始文件中的偏移量（读写请求的给出的偏移量，由函数参数给出），表示分块的大小（本系统中为4KB，即为4096），表示Hash序列在元数据文件中的起始偏移量，考虑到内存页面对齐，本系统中值固定为4096。

#### CAS存储过程

利用CAS进行重复数据删除的基本思想为：用唯一的指纹来表示数据的内容，并且指纹相对于原始数据只有很小一部分。本论文采用的Hash算法为用通用的MD5算法，用MD5算法生成数据块的指纹，并用128位作为Hash来唯一表示相应数据块的内容。由于本系统采用分块文件则保存在服务器的共享存储系统当中，并由服务器端程序进行管理。服务器端保存有全局的“指纹——数据”映射关系。

根据Hash值查找数据文件时，可直接在共享目录中查找是否存在指定文件名的文件。若存在则直接将其共享次数在原有的基础上加1，若不存在则创建新的文件并把共享次数标记为1。

删除文件时，只有当文件的共享次数达到0才把文件真正从磁盘上删除，否则只是将其对应的共享次数减1即可。

在客户端部分的基于重复数据删除的读取和写入流程如图 16和图 17所示。



图 16 基于CAS方法的数据读取流程



图 17 基于CAS方法的数据写入流程

### 服务器端设计

服务器最要功能被设计为用来响应客户端发来的读写请求。服务器主要根据客户端发来的读写请求实现指纹数据的检索、分块数据的存储以及分块数据的管理和高校存储等功能。具体一下几个方面的工作：

1. 分块文件管理

经过CAS优化后的数据文件表现为大量大小相等的数据块，一种直观可行的方案是将数据块当成单个小文件，直接以其对应的指纹来命名，这是对内容寻址存储思想的直接体现，然而，现有的文件系统并没有针对大量小文件的情况进行特殊的优化。因此，随着小文件数量的增长，文件的检索性能将会极具下降。因此，需要研究和开发一种合理的方案来存储分块数据，以适应海量分块数据的应用场景。

本系统采用的是一种将经过重复数据删除后的分块文件聚合为大文件的方案，把数据块连续地存在磁盘的大文件中，利用数据块在文件中的偏移量进行寻址。在释放数据块时，文件中间可能产生空闲区域。对于文件中间可能产生的空闲块，采用动态数组方式链接起来，每次写入新数据块需要分配空间时，优先分配文件中间的空闲块。分块文件管理方案如图 18所示。



图 18 分块文件管理

整个空闲空间的索引分为两级，用于存储一级索引的数组在系统初始化时直接分配。二级索引数据则根据需要进行分配。每个二级索引数组都是采用堆栈的方式进行操作，即后进先出，最新产生的空闲块会最先被分配。空闲数据块的总数记录在相应的数据结构中。

当新的空闲块产生时，根据当前空闲块的总数计算出当前偏移量应该存在二级索引中的位置，如果二级索引数据中有足够的空间容纳新产生的空闲块地址，则直接写入二级索引数组对应的位置。若当前二级索引数据空间已满，则需要申请新的数组作为二级索引，并将数组地址链接到一级索引当中。

当有新的数据块需要写入时，首先查看文件中间是否有空闲区域，若有则直接取二级索引栈顶位置的元素作为偏移量进行数据写入。若当前文件中间不存在空闲区域，则新写入的数据块可直接追加到文件尾部。

1. “指纹——数据”映射关系

基于CAS的存储方式，在读取和写入时都需要根据相应的指纹找到对应的数据块然后进行操作。在服务器端需要维护全局的“指纹——数据”映射关系。

“指纹——数据”是典型的key-value关系，指纹是全局唯一的，因此本系统把指纹和数据之间的映射关系以key-value的形式存储于高性能的Berkeley DB中，其中key为指纹本身，value为指纹所对应的数据块在具体文件中的偏移量和数据块的共享计数等信息。系统针对对经常访问的指纹数据以及共享计数比较大的指纹数据都进行了缓存，以此提高指纹操作的效率。

1. 读写请求队列

根据指纹值不同设置多个读写队列，以达到在多个读写队列之间可以对指纹和数据进行并行读写。



图 19 服务器端整体架构

## 基于分布式Hash的并行存取方法

### 指纹数据的分布与存储方法

在重复数据删除系统中，指纹是最频繁访问的数据。因此指纹数据的分布与查找的性能是决定重复数据删除性能的关键，同时也是制约重复数据删除技术应用的一个主要方面。当前的重复数据删除技术主要应用于备份系统或用于虚拟机镜像的离线存储。典型的重复数据删除系统都把指纹数据全部加载到内存中，这样以来指纹的查找便会成为全部在内存中进行的操作，可以达到较好的性能，但同时也占用这巨大的内存空间。

本系统采用基于分布式Hash的指纹和数据的存储方案，在服务器节点内部，指纹数据存在Berkeley DB中，并且对经常访问和引用次数较大的指纹数据进行缓存，因此，在兼顾内存占用率的同时可以达到很好的性能。

同时，指纹数据映射到不同的服务器节点上，不同节点之间的指纹的操作相互独立，彼此之间不会冲突，因此可以实现不同节点之间对指纹和相应数据的并行读取和写入。在客户端根据对数据计算出的指纹信息，确定相应的数据所在的服务器的信息，即建立“指纹——服务器”之间的映射关系。通过映射关系，能够实现在客户端和服务器之间的高效通信，和数据存取。

Hash指纹的具体映射方法如图 20所示，整个128位的Hash被分为两部分，最低4个字节和高12个字节。最低四个字节可以确定一个无符号整数，通过对整数进行均匀的Hash运算即可以把Hash指纹映射到不同的服务器上。



图 20 Hash指纹映射方法

### 分块数据的分布与管理方法

将经过重复数据删除后的分块会根据其各自的指纹被映射到不同的服务器节点上，分块数据在服务器上会被聚合为大文件（例如每个文件大小为4GB），对于文件中间可能产生的空闲块区域，采用动态数组方式链接起来，每次产生新数据块需要分配空间时，优先分配文件中间的空闲块。这样可以充分的减少对内存的占用，又可以达到很高的空闲快分配性能。

### 多节点之间的网络通信模块

网络通信模块主要实现在多个客户节点和服务器节点之间实现指纹和数据并行存取，在客户端对指纹进行简单Hash运算，映射到不同的服务器节点上。利用指纹（md5）数据本身的均匀分布特点达到很好的负载均衡的效果。

本系统采用将指纹和对应的数据存储在同一服务器节点上的方案，以此来减少请求数据时需要进行网络通信的次数。当客户端需要从服务器端读取数据时，只需要给出对应数据的Hash指纹，便可以利用一次网络通信把数据从服务器端传输到客户端，以此来提高系统的整体读写性能。

## 指纹数据的预取与缓存策略

对于计算机系统来说，磁盘操作通常要比CPU、内存等操作的速度慢好几个数量级，因此在大部分系统中，磁盘操作都是整个系统的瓶颈所在，通常情况下，缓存和预取数据是减少磁盘I/O次数从而提高系统性能的有效方法。在Linux操作系统中，经常访问的文件会被最大限度的缓存到内存中，以此来提高系统整体性能。然而，页面缓存存在于内核态的地址空间当中，用户态的程序每次需要读取数据时仍然需要系统调用从而进入内核态，系统调用涉及到CPU权限的检查以及切换，寄存器的切换等，频繁的系统调用会对系统性能带来很大开销。

本系统设计并实现了一种用户态的指纹数据缓存和预取方法。设置用户态的指纹序列预取窗口，并最大化的根据局部性原理预取和缓存当前窗口中的指纹序列，可以最大限度减少每次读指纹序列带来的系统调用切换开销，从而大幅度提高指纹数据操作的效率。具体的预取窗口设置如图 21所示，根据根据需要设置一个或多个预取窗口以适应文件在不同的位置进行读取和修改的需求。



图 21 指纹序列预取窗口

## 本章小结

本章主要论述了论文所论述方法的总体设计。首先对镜像优化方法的实现层次进行了深入分析，确定了基于FUSE实现基于重复数据删除技术的用户态文件系统方案。之后，对系统进行了整体模块的划分，并对各个模块的功能进行了论述。在此基础上给出了基于重复数据删除技术的虚拟机镜像存储优化方法的总体工作流程。之后，对本论文的核心研究点进逐个进行了详细深入地介绍。对论文要实现的核心系统——基于重复数据删除技术的用户态文件系统进行了深入讨论以及详细的设计及实施方案，包括各个模块的划分以及各个模块在实现时所要解决的关键问题及给出相应的解决方案。介绍了基于分布式Hash的并行存取方法的实现。最后，对指纹数据的换取和预取的实现方案进行了详细设计。

# 系统实现

## 原型系统实现概述

本论文所研究的基于重复数据删除技术的虚拟机镜像存储优化方法最终实现为一套基于内容寻址存储方式的用户态文件系统。主要涉及到客户端与服务器端以及网络通信模块的开发。在客户端方面主要工作为对基于FUSE提供的文件操作接口进行重新实现，以适应CAS的存储方式，包括基于CAS的读写方式，元数据文件的管理，指纹数据的预取和缓存策略等。服务器端主要设计到“指纹——数据”之间的映射关系的存储和管理，分块数据的高效存储方法的实现等。网络通信模块主要为客户端和服务器之间提供通信接口，本系统采用基于TCP的网络通信方案进行设计。系统涉及到的全部代码都在Linux系统上开发完成。

为了保证原型系统的可移植性，降低系统开发的复杂度，本系统基于FUSE框架实现对CAS存储虚拟机镜像过程的封装，此过程对共享存储系统和上层的应用程序均保证透明性，以最大限度的提高系统的可移植性。本系统在支持FUSE架构的操作系统之上均可以运行。

宿主机的虚拟机实例（VM）由远程的桌面客户端启动并操作，在此过程中，每个VM通过虚拟机监控器向宿主机所挂载的共享存储目录中的相应文件读写数据，对应的I/O系统调用所产生的相应VFS调用在内核空间被FUSE的内核模块捕捉；然后，VFS调用被转发到用户态的FUSE接口程序中，该程序重新实现了每种VFS调用的对应应答流程，在此过程中按照CAS的存储思路与远程的共享存储系统交互，重点完成了从虚拟机镜像文件到实际在共享存储系统中数据存储之间的转换，即用户所看到的每个虚拟机镜像文件（如数据盘文件、内存文件等）被划分成多个文件存储（每个文件存储一个或多个CAS数据块）。

在上述过程中，从用户的角度看，用户是在操作本地的桌面系统；从虚拟机监控器的角度看，所操作的虚拟机文件和原存储方案(如直接存储在NFS、Ceph中)相同；从共享存储系统的角度看，所存储的虚拟机镜像文件被分成若干小的文件分别存储。

## 镜像文件存储方式

针对KVM虚拟机镜像文件的特点，原来的文件将用来存储文件大小信息和Hash序列，而文件中的实际数据则以分块文件的形式保存在共享存储区域中。原文件的存储格式参照图 15所示。

镜像文件中首先存储原镜像文件的大小信息，数据类型为8字节无符号整数。接下来就是文件数据的Hash信息。在共享存储系统中，分块文件将直接以Hash序列来命名，对应的分块被共享的次数以整数的形式存在分块文件的扩展属性中。

Hash算法使用通用的MD5算法，并用128位作为Hash值保存在原文件中。分块文件则保存在指定的共享目录（共享文件系统）中。分块文件直接用其Hash值作为文件名。

根据Hash值查找数据文件时，可直接在共享目录中查找是否存在指定文件名的文件。写入操作发生时，现在内容中计算出数据不分的Hash值，然后根据Hash创建的分块文件，若同名的分块文件已经存在则直接将其共享次数在原有的基础上加1，若不存在则创建新的文件并把共享次数标记为1。

同理，当删除文件时，只有当文件的共享次数达到0才把文件真正从磁盘上删除，否则只是将其对应的共享次数减1即可。

## 详细实现细节

本系统以FUSE和CAS技术为基础，程序的核心其实是一组针对FUSE所提供的用户态文件系统的接口的具体实现。核心程序位于../src/fuse\_core.c文件中,程序中用到的辅助函数（如write\_block\_file、unlink\_block\_file、md5\_to\_path等）位于../src/fuse\_io\_utils.c文件中，Hash算法md5位于../src/md5.c文件中，程序中用的数据结构和变量说明如下。

* int g\_block\_size 用于表示分块文件的大小
* int init\_offset 用于表示镜像文件中Hash序列的起始偏移量
* int hash\_size 用于表示Hash值的长度（以字节为单位）
* char \*dest\_path 用于表示分块文件的存储路径

### getattr函数

static int fuse\_getattr(const char \*path, struct stat \*stbuf)

{

int res;

res = lstat(path, stbuf);

if (res == -1)

return -errno;

if (stbuf->st\_mode & S\_IFREG)

{

res = open(path, O\_RDONLY);

if (-1 == res)

return -errno;

else

{

pread(res, &(stbuf->st\_size), sizeof(off\_t), 0);

close(res);

}

}

return 0;

}

此函数用于实现获取文件的属性操作，对应于文件操作系统调用中的stat/lstat函数，属性存于stbuf所指向的结构体中。在调用了系统lstat函数之后进行文件属性的判断，若文件为普通文件则从镜像文件的头部读出原文件的大小，其他的属性则保持不变。若文件为非普通文件（如目录等）则函数直接等同的系统的lstat调用。

### create函数

static int fuse\_create(const char \*path, mode\_t mode, struct fuse\_file\_info\* fi)

{

off\_t st\_size = 0;

int fd = open(path, O\_CREAT | O\_EXCL | O\_RDWR, mode);

if(fd != -1)

fi->fh = fd;

else

{

perror("create file error:");

return -errno;

}

pwrite(fd, &st\_size, sizeof(off\_t), 0);

return 0;

}

文件的创建函数，对应于系统调用creat。在调用系统函数open创建文件之后，把文件描述符保存与fuse\_file\_info \*fi 结构体中，以备文件操作函数read write 等调用。之后在文件头部写入文件大小信息。文件大小用8字节无符号整数表示。

### open函数

static int fuse\_open(const char \*path, struct fuse\_file\_info \*fi)

{

int res;

if(fi->flags & O\_TRUNC)

{

printf("O\_TRUNC flag is set\n");

fuse\_truncate(path, 0);

fi->flags &= ~O\_TRUNC;

}

fi->flags &= ~O\_WRONLY;

fi->flags |= O\_RDWR;

res = open(path, fi->flags);

if (res == -1)

return -errno;

fi->fh = res;

return 0;

}

文件打开函数，对应于系统调用open，由于在对文件经行写入操作时也需要先读取镜像文件中的Hash序列，所以当文件以O\_WRONLY方式打开时需要将其打开标志修改为O\_RDWR。如果打开文件中有truncate 操作，需要调用对应的fuse\_truncate 函数，而不是系统的truncate调用。在调用open函数打开文件之后将文件描述符保存在fuse\_file\_info \*fi 所指定的结构体中，以供read、write等函数使用。

### read函数

static int fuse\_read(const char \*path, char \*buf, size\_t size, off\_t offset, struct fuse\_file\_info \*fi)

{

int fd, bfd, err = 0;

int res = 0, rdcount = 0;

off\_t i = offset, st\_size = 0;

char src\_block\_path[256] = {0};

unsigned char \*hashdata=(unsigned char\*)(src\_block\_path + strlen(dest\_path) + 20);

strcpy(src\_block\_path, dest\_path);

fd = fi->fh;

res=pread(fd, &st\_size, sizeof(off\_t), 0);

if(res == -1)

{

perror("read filesize error:");

return -errno;

}

st\_size = min(st\_size,offset+size);

while (i < st\_size/\*offset+size\*/)

{

err = pread(fd, hashdata, hash\_size, i / g\_block\_size \* hash\_size + init\_offset);

if(err == -1)

{

perror("read md5 error");

return -errno;

}

md5\_to\_path(src\_block\_path + strlen(dest\_path));

bfd = open(src\_block\_path, O\_RDONLY);

if (bfd == -1)

{

return -errno;

}

rdcount += pread(bfd, buf + i - offset, min(g\_block\_size,

size - i + offset), (i & (g\_block\_size - 1)));

close(bfd);

i+= min(size-i+offset,g\_block\_size-

(i&(g\_block\_size-1)));

}

return rdcount;

}

读取文件函数，对应于系统调用read。read函数操作流程如下：

1. 从fuse\_file\_info \* fi所指定的结构体中获取文件描述符fd
2. 从fd指向的文件中读取文件大小信息
3. i / g\_block\_size \* hash\_size + init\_offset为Hash序列的位置，读出相应的Hash序列后存于hashdata数组中。
4. 调用md5\_to\_path 函数吧Hash序列转换成分块文件实际的路径
5. 读取分块文件中的数据添加到buf中
6. 更新文件指针，若还有数据要继续读则转步骤3，若无则转7
7. 返回实际所读的字节数

### write函数

static int fuse\_write(const char \*path, const char \*buf, size\_t size, off\_t offset, struct fuse\_file\_info \*fi)

{

int fd, bfd;

int res = 0;

off\_t st\_size = 0, i = offset;

int err = 0;

char src\_block\_path[256], dest\_block\_path[256], md5char[36] = { 0 };

unsigned char data[BLOCK\_SIZE], \*hashdata = (unsigned char\*) md5char + 20;

fd = fi->fh;

if (fd == -1)

return -errno;

res = pread(fd, &st\_size, sizeof(off\_t), 0);

if (res == -1) {

perror("read size error:");

return -errno;

}

while (i < offset + size)

{

memset(data, 0, sizeof(data));

if (i < st\_size)

{

res = pread(fd, hashdata, hash\_size,

i / g\_block\_size \* hash\_size + init\_offset);

if (res > 0)

{

md5\_to\_path(md5char);

sprintf(src\_block\_path,"%s%s",dest\_path, md5char);

bfd = open(src\_block\_path, O\_RDONLY);

pread(bfd, data, g\_block\_size, sizeof(int));

close(bfd);

ulink\_block\_file(src\_block\_path);

}

}

memcpy( data + (i & (g\_block\_size - 1)),

buf + i - offset,

min(size - i + offset, g\_block\_size - (i & (g\_block\_size - 1))));

md5(data, g\_block\_size, hashdata);

err = pwrite(fd, hashdata, hash\_size,

i / g\_block\_size \* hash\_size + init\_offset);

if (err == -1)

return -1;

md5\_to\_path(md5char);

sprintf(dest\_block\_path, "%s%s", dest\_path, md5char);

err=write\_block\_file(dest\_block\_path,data,

g\_block\_size);

if (err == -1) {

perror("error @ offset %d");

return -1;

}

i += min(size - i + offset, g\_block\_size - (i & (g\_block\_size - 1)));

}

st\_size = st\_size < offset + size ? offset + size : st\_size;

res = pwrite(fd, &st\_size, sizeof(off\_t), 0);

if (res == -1)

{

perror("update size error:");

return -1;

}

return size;

}

文件的写入函数，对应于系统调用write。函数的操作流程如下：

1. 从fuse\_file\_info \*fi 所指向的结构体中获取文件描述去fd
2. 从文件头读出文件大小信息保存在st\_size变量中
3. 根据偏移量计算出Hash序列的位置并读出Hash序列保存于hash\_data中
4. 调用md5\_to\_path将Hash序列转换成对应分块文件的路径，并读出分块文件中的数据保存于data中，调用unlink\_block\_file 释放对应的分块文件
5. 将buf中的数据写入data中，并计算data数据的Hash值
6. 根据新的Hash序列，调用md5\_to\_path 函数得到文件路径，并调用辅助函数write\_block\_file将data中的数据写入对应的分块文件中
7. 更新文件指针，若还有数据要写入则转步骤3，若无则转步骤8
8. 将offset+size 和 st\_size 值中较大的作为新的文件大小并写入文件头
9. 返回实际写入的字节数

### truncate函数

static int fuse\_truncate(const char \*path, off\_t size)

{

int fd, res = 0;

char src\_block\_path[256] = {0};

off\_t st\_size=0,i=(size & (g\_block\_size - 1)) ? size + g\_block\_size - (size & (g\_block\_size - 1)) : size;

fd = open(path, O\_RDWR);

if (fd == -1)

{

return -errno;

}

strcpy(src\_block\_path, dest\_path);

res = pread(fd, &st\_size, sizeof(off\_t), 0);

if(res == -1)

{

return -1;

}

while (i < st\_size)

{

pread(fd, src\_block\_path + strlen(dest\_path) + 20, hash\_size, i / g\_block\_size \* hash\_size + init\_offset);

md5\_to\_path(src\_block\_path + strlen(dest\_path));

ulink\_block\_file(src\_block\_path);

printf("unlink path is %s\n", src\_block\_path);

i += g\_block\_size;

}

res = pwrite(fd, &size, sizeof(off\_t), 0);

close(fd);

return 0;

}

fuse\_truncate函数实现截断文件的功能，对应于truncate/fruncate系统调用。

函数实现流程：

1) 以读写方式打开镜像文件

2) 读取镜像文件大小保存于st\_size 变量中

3) 根据文件指针大小计算对应Hash序列的位置

4) 从镜像文件中读出Hash序列，并调用md5\_to\_path得到对应分块文件的路径，并调用unlink\_block\_file函数释放分块文件

5) 若已经到达文件结尾则转步骤6，否则转步骤3

6) 将size写入文件头以更新文件大小信息（注：对源文件的截断操作实际并没有真正调用truncate操作，只是更新了文件头部的文件大小+-信息）

### 缓存模块的实现

缓存模块使用数据块的Hash值来作为缓存索引，缓存区建立于模块的私有内存空间里，由缓存模块维护。当文件系统需要进行读写操作时，直接调用缓存模块提供的API进行命中检查或者记录缓存。

1. 缓存区结构

缓存区采取1024行8路组相连结构，每一个缓存块包含以下内容：

hash\_arr[hash\_size]用于记录特定缓存块的hash值的数据结构

block\_arr[g\_block\_size]用于保存缓存块数据的数据结构

life\_arr用于记录缓存块的生命值，供替换策略使用的数据结构

1. 缓存模块的函数

读缓存操作

size\_t cache\_read(unsigned char\*hash,void\*buf,size\_t size);

通过hash值确定行，再在组内遍历，若命中则返回并复制数据块内容，并返回复制的大小，否则返回0。

传入参数：

unsigned char\*hash：hash值数组的头指针（方向in）。

void\*buf：用于存储数据块的空间头指针（方向out）

size\_t size：需要复制的大小（必须小于全局块大小g\_block\_size，方向in）

返回值：

返回已复制的大小

写缓存操作

void cache\_write(unsigned char\*hash,void\*buf,size\_t size);

通过hash值确定行，遍历寻找生命值最小的缓存块替换。

传入参数：

unsigned char\*hash：hash值数组的头指针（方向in）。

void\*buf：数据块的头指针（方向in）

size\_t size：数据块的大小（必须小于全局块大小g\_block\_size，方向in）

返回值：无返回值

## 本章小结

本章对系统的实现进行了深入的解释和说明。包括主要的系统调用函数的重新实现和缓存模块的设计。

# 系统的测试与分析

## 测试环境

### 硬件环境

2台节点机，配置如下：

* CPU：Intel® Xeon® 5160 3.00GHZ\*4
* 内存：4GB DDR2 800
* 硬盘：80GB
* 网卡：Intel Gigabit

### 软件环境

* 操作系统：Redhat 6.0企业版
* Linux内核版本：2.6.32 x86\_64
* Fuse版本： Fuse-2.9.2
* 虚拟机软件：qemu-kvm

## 功能测试

### 测试目标

对系统的正确性进行测试，采用不同的操作系统进行测试，确保所有系统均可正确运行。

### 测试内容

采用不同的操作系统进行测试，确保所有系统均可正确运行。测试用到的操作系统有以下几种：

* Windows7
* Windows XP professional
* Red Hat Enterprise Server 6.0

测试结果：

* Windows7：

运行命令: /usr/libexe/qemu-kvm 7.img –boot c –m 1024 –usbdevice tablet –vnc :2 &

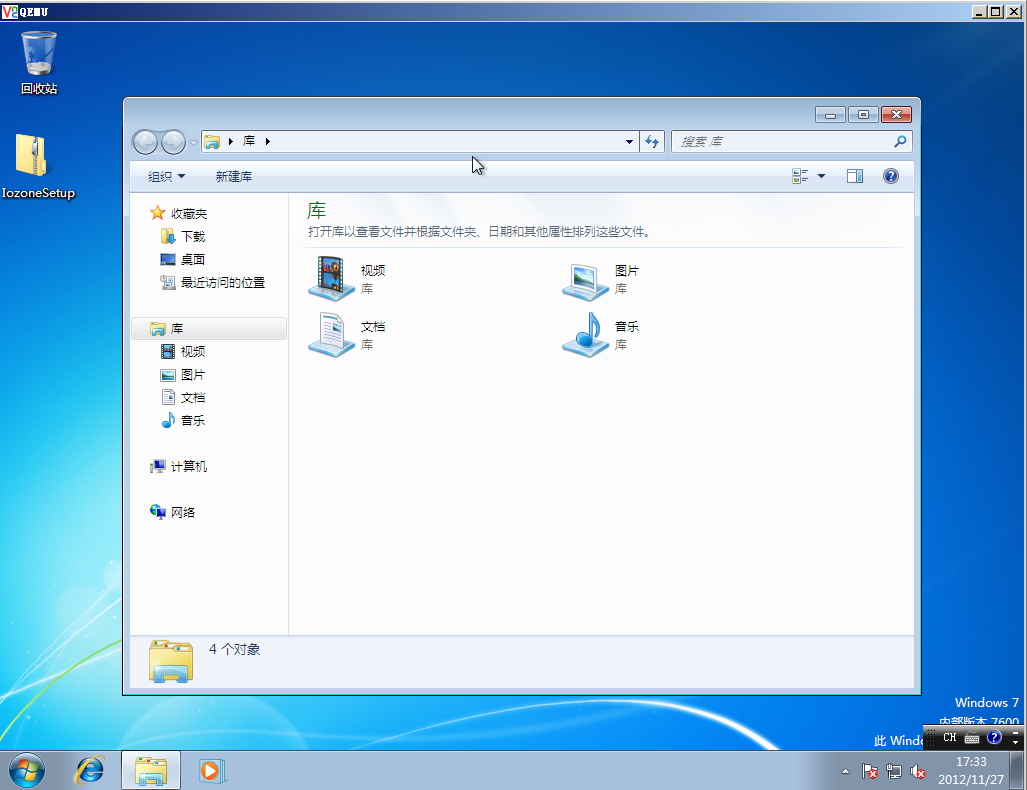


图 22 Windows7虚拟机运行效果

如图 22所示，Windowx7的虚拟机可以正常启动和运行。

* Windows XP professional：

运行命令： /usr/libexe/qemu-kvm xp.img -boot c -m 1024 -usbdevice tablet -vnc :2 &



图 23 WindowsXP 虚拟机运行效果

如图 23所示，WindowxXP的虚拟机可以正常启动和运行。

1. Red Hat Enterprise Server 6.0:

运行命令：/usr/libexe/qemu-kvm redhat.img -boot c -m 1024 -usbdevice tablet -vnc :2 &

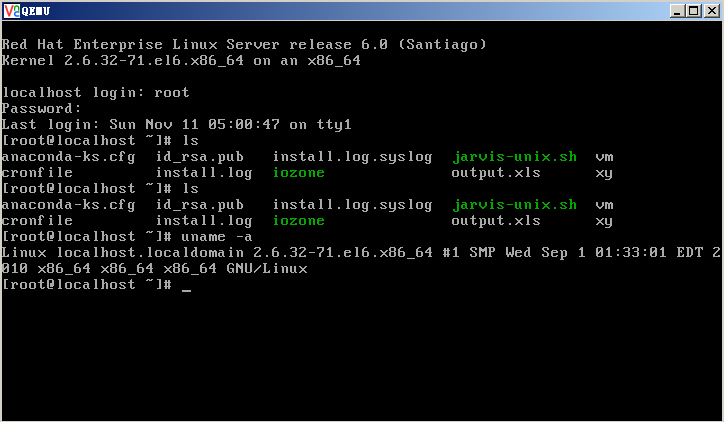


图 24 Red Hat Enterprise Server 6.0虚拟机运行效果

结果：如图 24所示，Red Hat Enterprise Server 6.0虚拟机可以正常启动和运行。

如以上结果所示，本系统分别对运行Windows和Linux的虚拟机进行了正确性测试，测试结果表明本系统对运行Windows和Linux的虚拟机都可以正确的启动和运行。因此，在对虚拟机镜像文件进行重复数据删除后没有对虚拟机的正常运行产生影响，因此，系统的正确性得到验证，因而本系统具有可用性。

### 虚拟机镜像优化存储优化效果测试

重复数据删除技术的核心目标是降低虚拟机镜像文件的磁盘占用开销，因此需要对优化之前和优化之后的镜像所占存储空间进行对比，得出镜像存储优化在存储空间降低方面的实际效果效果。

所用到的镜像文件包括一下几种：

* Windows7 (7.img)
* WindowsXP Professional (xp.img)
* RedHat Enterprise Server 6 (redhat.img)
* Fedora15 (fedora.img)

测试效果：在实际拷入以上4个镜像的前提下，优化之后镜像所占空间为优化之前的49.5%。具体优化效果如表1.1所示。

表 3 CAS存储优化效果

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 镜像文件 | 原始大小(GB) | 优化后大小(GB) | 原始镜像占用总空间(GB) | 优化后所占空间(GB) |
| Redhat | 20.0 | 4.0 | 80.0 | 39.6 |
| Fedora | 20.0 | 10.6 |
| WindowsXP | 20.0 | 12.0 |
| Windows7 | 20.0 | 13.0 |

### 测试结果分析

镜像文件本身在保存内存快照或者磁盘的地方可能存在大片区域的相同部分，例如，没有使用的内存或者磁盘空间均以“0”来填充。而镜像文件之间也存在着很多相同的部分，运行相同操作系统或者相同操作系统的不同版本的虚拟机镜像文件之间都会存在大量相同的部分，而重复数据删除技术优化方法在于重复数据只保存一份，因此，可以针对虚拟机镜像的上述特点达到降低镜像文件磁盘占用率的效果。如表 3中所示的测试结果，镜像优化部分达到预期的效果。对于常用的虚拟机镜像，本系统基本可以实现将磁盘空间占用率降低30%甚至更高。

## 性能测试

### 测试目标

测试优化后的虚拟机内部的I/O性能，主要对磁盘文件读写方面进行测试，并与优化之前的虚拟机内部的I/O性能进行对比，通过分析对比得出系统的具体开销及其对虚拟机I/O性能可能产生的影响。

### 测试内容

在虚拟机中运行IOZone基准测试程序测试虚拟机I/O性能，并通过优化后和优化之前的数据进行对比得出结论。主要测试内容如下：

* 测试不同的块传输大小（4KB–16MB之间）
* 测试不同的文件大小（4KB – 2GB之间），得出测试数据
* 测试所用虚拟机操作系统：WindowsXP Professional sp3
* 测试所用命令：iozone –Rab –g 2G

### 测试结果

图 25 原生虚拟机的文件读取性能

图 26 优化存储之后的虚拟机文件读取性能

由图 26可见，在文件大小或文件分块较小时，顺序读取性能很高，这是因为系统缓存的缘故，文件较小时，磁盘的读写速度基本接近于内存速度，因而较高。随着文件和分块的逐渐变大，读取性能开始出现急剧下降，因此，此时的读取性能才是真正意义上的磁盘读取性能。

通过存储优化之后和优化之前的比较可见，经过镜像优化之后的顺序读性能基本接近优化之前的性能，甚至有些区域要好于优化之前的性能。这是因为在存在大量相同数据读取的情况下，CAS本身就是对读取性能的优化，利用缓存功 能对相同的数据只读取一次到内存中，之后，相同的数据块就会的读取操作就会发生在内存中。

图 27 原生虚拟机的写入性能

图 28 优化存储之后的虚拟机文件写入性能

由图 28可以看出，顺序写的性能要低于顺序读的性能，并且相比于顺序读，顺序写的性能更加稳定，但也会随着文件的增大而降低。

CAS优化前后的虚拟机顺序写性能对比如下表所示。在顺序写方面，CAS优化前后的性能基本一致，某些点上CAS优化之后的性能要远远好于优化之前的性能。这是因为CAS优化之后相同的数据快至写入硬盘一次，而优化之前却必须全部写入磁盘，因而出现CAS优化之后顺序写性能在某些地方优于优化之前的现象。

图 29 原生虚拟机的重复读性能

图 30 优化存储之后的虚拟机的重复读性能

从重复读（Re-Read）的测试结果来看，重复读的性能变化趋势基本和顺序读一致，但Re-Read要整体好于顺序读的性能，同样是因为操作系统对读取过的文件产生缓存的原因。同样，当测试文件大小或分块达到一定程度后，性能出现急剧下降，在此之后整体保持在基本稳定的状态。

通过对比可以看出，CAS优化之后的虚拟机在重复读方面表现较好，这是因为CAS的缓存机制从本质上说优于系统本身的缓存方法，系统自身缓存可以命中的操作经过CAS之后也一定会命中，而CAS缓存还对相同内容的数据块进行缓存，因此从缓存命中率来看CAS缓存效果更好。

图 31 原生虚拟机的重复写性能

图 32 优化存储之后的虚拟机的重复写性能

由图 31 与图 32可以看出，重复写的性能要优于顺序写的性能，同样是系统缓存的缘故。并且，重复写和顺序写在随着文件和分块大小的性能变化上基本一致。在文件较小时，性能较高，而随着文件和分块的变大，性能开始下降并最终趋于稳定值。

为了更好的比较镜像优化前后虚拟机的I/O性能，下面给出优化前后平均吞吐率的对比图，优化前后的虚拟机I/O性能对比如图 33所示。

图 33 镜像优化前后的读写性能对比

### 测试结果分析

整体上看，无论是优化之后的虚拟机还是原生态的虚拟机在读写小文件时性能较好，而随着文件的逐渐变大，读写性能出现了急剧的下降，这是因为在读写小文件时，系统的缓存功能对读写性能起到了很大的提升作用，而随着文件的变大，操作系统的缓存逐渐失去效果，而此时才是真正意义上对磁盘读写性能的测试。

从I/O测试结果以及形成的对比来看，经过存储优化之后的虚拟机读写性能和原生态虚拟机读写性能相比，性能损失小于的10%。

在某些数据大小和分块大小上面，经过优化的虚拟机性能甚至比原生态的虚拟机性能更高，原因在于CAS对于相同的数据只写入一次。加上缓存模块之后，对于大量相同内容的数据进行读取使得缓存命中率提高，也有助于提高虚拟机的读写性能。因此，性能方面经过CAS优化之后的性能相比于优化之前的性能损失在10%以内。

## 本章小结

本章对基于重复数据删除技术的虚拟机镜像存储优化系统进行了全方位的测试和验证，并对测试结果进行了深入分析。包括功能测试和性能测试。功能测试主要针对虚拟机的正确运行进行测试。结果表明，优化之后的虚拟机可以正确的运行，说明了优化系统能保证镜像文件的正确性。性能测试主要针对优化之后的虚拟机I/O性能进行测试。从测试结果来看，经过镜像存储优化之后的虚拟机的I/O性能也达到了很好的效果。

总结与展望

针对目前桌面虚拟化应用当中，大规模的虚拟机镜像文件带来的存储开销，文章首先分析了现有的存储优化技术及重点对重复数据删除技术做了详细和深入的介绍，之后介绍了重复数据删除在备份系统与虚拟机镜像存储优化系统中的应用。在现有研究的基础上，本文研究并实现了一种技术重复数据删除技术的虚拟机镜像存储优化方法。具体做法是：采用CAS方法，对虚拟机镜像文进行分块后计算指纹序列，通过指纹序列判断重复数据。对于内容相同的数据块只保留一份，并且在每一个分块文件中记录对应被共享的次数。原来的文件中实际保存的是指纹序列，数据部分保存在共享存储系统里面。为了保证对虚拟机监控器的透明存取及系统的可移植性，采用基于FUSE框架将对虚拟机镜像的操作过程封装到相应的系统调用应答接口的实现中，以服务FUSE内核模块所获取上层用户操作虚拟机所产生的I/O请求，从而将虚拟机镜像文件的直接存储转换为基于CAS方式的存储。在此基础上，分别对存储优化效果和性能等两个方面对系统进行了测试和验证。

整个系统完全实现在Linux操作系统上，以FUSE架构为基础，开发了一套基于重复数据删除的分布式用户态文件系统。从而验证了在线重复数据删除技术在桌面虚拟化应用中的可行性。整个系统的实现涉及到Linux编程的方方面面，特别是文件操作与网络及多线程编程。需要对Linux编程有深入的了解。从阅读文献到系统设计，再到原型系统的实现，以及最后的测试和验证，整个论文的完成过程是艰辛且充满挑战的。。但是，论文的完成过程却引领我真正深入到文件系统和操作系统内部。另外，论文的完成过程也是对研究生阶段所学知识的一个综合利用与总结。因此，本论文的设计研究与实现虽然具有一定难度，但却是十分有意义的。

目前，论文所实现的原型系统和存在一定的缺陷。特别是性能和容错性方面还有很大的提升空间。因此，论文的后续工作将是进一步优化系统的性能以及系统的容错性。首先可以在现有的技术上，进一步优化指纹数据的管理，更好的挖掘指纹的访问模式，以便更好的进行缓存和预取，减少磁盘I/O对系统性能的影响。其次，可以在现有系统的基础上，加入日志的功能，以便随时记录各个操作的过程，在系统突发断电等情况下可以利用日志来恢复损坏的数据。组后，对于引用计数较大的分块文件，可以采用多个副本的存储方式，防止因为一个数据块损坏而导致很多虚拟机无法启动的现象发生。除以上两点之外，系统在扩展性方面还有进一步优化的空间，例如采用一致性散列Hash算法对指纹在多个节点上的分布进行管理，这样便可以在增加或删除存储节点时，只影响一小部分节点，而不是全部的数据都要重新分布。。当然，对系统性能的改进和优化并不仅限于所列的几个方面。在未来进一步的研究工作中，新的优化和改进思路也将被不断提出。

参考文献

1. Desktop Virtualization. http://datacenter.watchstor.com/desktopvirtualization-138310.htm
2. File System in User Space. http://fuse.sourceforge.net/
3. C. Policroniades and I. Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In Proceedings of the 2004 USENIX Annual Technical Conference, Boston, MA, USA, June 2004.
4. Partho Nath, Michael A. Kozuch, David R. O’Hallaron. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. Annual Tech ’06: 2006 USENIX Annual Technical Conference.
5. C. Ungureanu, B. Atkin, A. Aranya, S. Gokhale, S. Rago, G. Calkowski, C. Dubnicki, and A. Bohra. HydraFS: a High-Throughput File System for the HYDRAstor Content Addressable Storage System. In Proc. of USENIX FAST, 2010.
6. C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M.Welnicki. Hydrastor: A scalable secondary storage. In Proc. USENIX FAST, 2009.
7. Chun-Ho Ng, Mingcao Ma, Tsz-YeungWong. Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud. Middleware 2011, 2011 – Springer.
8. Bin Fan, Wittawat Tantisiriroj, Lin Xiao , Garth Gibson. DiskReduce: RAID for data-intensive scalable computing. Proceedings of the 4th Annual Workshop on Petascale Data Storage. November 14-14, 2009, Portland, Oregon.
9. Philip Carns, Walter Ligon III, Robert Ross. BMI: A Network Abstraction Layer for Parallel I/O.
10. 敖 莉, 舒继武, 李明强. 重复数据删除技术. Journal of Software, Vol.21, No.5, May 2010, pp.916−929
11. Michael O. Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
12. B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In Proc. USENIX FAST, 2008.
13. S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In Proc. USENIX ATC, 2008.
14. D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In Proc. IEEE MASCOTS, pages 1–9. IEEE, 2009.
15. B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, 1970.
16. K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In Proc. ACM SYSTOR, 2009.
17. E. Kruus, C. Ungureanu, and C. Dubnicki. Bimodal content defined chunking for backup streams. In Proc. USENIX FAST, page 18. USENIX Association, 2010.
18. A. Liguori and E. Van Hensbergen. Experiences with Content Addressable Storage and Virtual Disks. In WIOV’08, 2008.
19. M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In Proc. USENIX FAST, 2009.
20. D. Meister and A. Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In Proc. IEEE MSST, 2010.
21. Offline Deduplication for Btrfs. http://www.spinics.net/lists/linux-btrfs/msg07818.html.
22. S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In Proc. USENIX FAST, 2002.
23. Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. 7th USENIX Conference on File and Storage Technologies.
24. Nawab Ali, Philip Carns, Kamil Iskra, Dries Kimpe, Samuel Lang, Robert Latham, Robert Ross, Lee Ward, P. Sadayappa. Scalable I/O Forwarding Framework for High-Performance Computing Systems. 2009 IEEE.
25. A. Clements, I. Ahmad,M. Vilayannur, and J. Li. Decentralized deduplication in SAN cluster file systems. In Proc. USENIX ATC, 2009.
26. S. C. Rhea, K. Liang, and E. Brewer. Value-Based Web Caching. in Proceedings of the 12th International Conference on World Wide Web (WWW), Budapest, Hungary, May 2003, pp.
27. Muthitacharoen A, Chen B, Maziéres D. A low-bandwidth network file system. In: Proc. of the 18th ACM Symp. on Operating System Principles (SOSP 2001). New York: ACM Press, 2001. 174−187.
28. Karissa Miller, Mahmoud Pegah. Virtualization: virtually at the desktop. Proceedings of the 35th annual ACM SIGUCCS fall conference Pages 255-260.
29. Chunqiang Tang. FVD: a High-Performance Virtual Machine Image Format for Cloud. USENIXATC'11 Proceedings of the 2011 USENIX conference on USENIX annual technical conference.
30. Fabrice Bellard. QEMU, a fast and portable dynamic translator. Proceedings of the annual conference on USENIX Annual Technical Conference, p.41-41, April 10-15, 2005. Anaheim, CA.
31. A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux Virtual Machine Monitor. In Proceedings of the Linux Symposium, pages 225-230, 2007.
32. Xiang Zhang, Zhigang Huo, Jie Ma, Dan Meng. Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration. Cluster Computing (CLUSTER), 2010 IEEE International Conference on.
33. Partho Nath, Bhuvan Urgaonkar, Anand Sivasubramaniam. Evaluating the usefulness of content addressable storage for high-performance data intensive applications. Proceeding HPDC '08 Proceedings of the 17th international symposium on High performance distributed computing 2008.

攻读硕士学位期间取得的学术成果

1. **Zhao Guoyu**, Tang Haofu, Xiao Limin, and Li Xiuqiao. Efficient Inline Deduplication on VM Images in Desktop Virtualization Environment[C]. International Conference on Mechatronics and Computational Mechanics. (已录用)
2. 肖利民, **赵国玉**, 李秀桥等. 一种基于内容寻址存储的虚拟机镜像存储优化方法[P]. 中国专利: 201210491092.9
3. 龚韬, 肖利民, **赵国玉**等. 一种应用于内容解析存储的基于内容的缓存方法[P]. 中国专利: 201110365027.7

致 谢

在本论文即将完成之际，谨此向我的导师肖利民教授致以衷心的感谢和崇高的敬意！本论文的工作是在肖老师的悉心指导下完成的。肖老师以他敏锐的洞察力、渊博的知识、严谨的治学态度、精益求精的工作作风和对科学的献身精神给我留下了刻骨铭心的印象，这些使我受益匪浅，并将成为我终身献身科学和献身事业的动力。

在攻读硕士的这两年半时间里，导师不仅为我创造了优越的科研和学习环境，使我得以在计算机科学领域中自由翱翔，同时在思想上、人生态度和意志品质方面给予了谆谆教诲，这些教益必将激励着我在今后的人生道路上奋勇向前。

感谢阮利老师在本论文的写作过程中对我的指导和帮助。

真诚感谢实验室的李秀桥博士师兄，他不仅在学术上给我指引，而且在生活上予以帮助，从他们身上我学到很多知识。感谢项目组成员在项目开发中的互助合作，正是集体的努力才使得项目进展顺利。

由衷感谢我的室友以及实验室的各位同学，他们开创性的研究拓展了我的学术视野，无数次的争论和探讨使我的研究工作有了长足的进展。

衷心的感谢我的父母和其他亲朋好友对我的关心、支持和理解，没有他们对我的关心、鼓励和支持，我无法完成现在的硕士学业。

最后，感谢曾经教育和帮助过我的所有老师。衷心地感谢为评阅本论文而付出宝贵时间和辛勤劳动的专家和教授们！