

中國人民大學

硕士 学位 论 文

(中文题目) 云端存储性能隔离关键技术研究

Research on Key Technologies
of Cloud Storage Performance

(英文题目) Isolation

作者学号: 2017100929

作者姓名: 王传雯

所在学院: 信息学院

专业名称: 计算机系统结构

导师姓名: 柴云鹏
云存储服务；分布式存储；混

论文主题词: 合存储；性能隔离

论文提交日期: 2020 年 4 月 30 日

独创性声明

本人郑重声明：所呈交的论文是我个人在导师的指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写的研究成果，也不包含为获得中国人民大学或其他教育机构的学位或证书所使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

论文作者：_____ 日 期：_____

关于论文使用授权的说明

本人完全了解中国人民大学有关保留、使用学位论文的规定，即：学校有保留送交论文的复印件，允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存论文。

论文作者：_____ 日 期：_____

指导老师：_____ 日 期：_____

摘要

大数据时代的来临，使得社交网络、电子商务、企业、科研等领域带来了海量增长的数据。云端存储服务的便捷性和高性价比为这些海量数据的储存、分析、处理提供了最佳选择。这些服务提供商向使用者做出服务质量（Quality of Service, QoS）保障，如带宽、延迟、空间，来满足不同用户的需求。性能隔离技术是云端存储平台服务质量保障必须攻克的难关。这是由于云端存储经常存在多租户共享同一块存储介质或者同一个存储系统的情形。共享带来了资源竞争的问题，如果不加以隔离，会出现存储带宽、延迟分配不均的情况。当资源竞争发生时，必然存在一些租户难以达到服务提供商保证的服务质量，严重影响服务购买者的体验。合格的性能隔离技术需要保证每一种用户均能达到预先设定的性能目标，同时保证最大化利用系统资源。然而，目前云存储的性能隔离仍然面临一些问题：1) 云存储类型多样。单机存储，如混合存储、虚拟磁盘等；分布式存储，如 Ceph, Lustre 等。存储平台应用了不同的隔离技术，需要分开研究设计。2) 现有的性能隔离技术存在问题，达不到预期的隔离目标。

本文的主要工作包括介绍了云端存储性能隔离的研究背景、意义和面临的挑战，分别针对典型的单机节点存储和多机节点存储平台上的性能隔离技术进行调研。针对这两种存储类型，提出了更加准确和稳定的性能隔离算法，并在实际系统上用真实工作负载进行实验评估。本文主要贡献总结如下：针对典型的**单机节点存储——混合存储**：1) 定量分析了经典隔离技术 Linux Control Group 性能隔离不准确的原因。2) 设计并实现了自适应精确节流算法。3) 进行了详实的对比实验，证明了自适应精确节流算法在隔离准确性和稳定性方面均优于 Linux Control Group。针对典型的**多机节点存储——Ceph**：1) 设计并实现了基于时间窗口的 WinRWClock 隔离算法 2) 与经典性能隔离算法 dmClock 算法进

行大量实验对比，证明 WinRWClock 算法在隔离准确性上更优。

关键词：云存储服务，分布式存储，混合存储，性能隔离

Abstract

The advent of the era of big data has brought massive data growth in the fields of social networks, e-commerce, enterprises, and scientific research. The convenience and high cost performance of cloud storage services provide the best choice for the storage, analysis and processing of these massive data. These service providers will provide users with quality of service (QoS) guarantees, such as bandwidth, delay, and space, to meet the needs of different users. Performance isolation technology is a difficult problem that the cloud storage platform service quality assurance must overcome. This is because cloud storage often has multiple tenants sharing the same storage medium or the same storage system. Sharing brings about the problem of resource competition. If it is not isolated, there will be uneven storage bandwidth and delayed allocation. When competition for resources occurs, there must be some tenants who are difficult to achieve the service quality guaranteed by the service provider, which seriously affects the experience of service buyers. Qualified performance isolation technology needs to ensure that each user can achieve the preset performance goals, while ensuring maximum utilization of system resources. However, the current performance isolation of cloud storage still faces some problems: 1) Cloud storage types are diverse. Stand-alone storage, such as hybrid storage, virtual disk, etc.; distributed storage, such as Ceph, Lustre, etc. In this way, different isolation technologies are applied to the storage platform, which requires separate research and design. 2) There are problems with the existing performance isolation technology, and the expected isolation target cannot be achieved.

The main work of this paper includes an introduction to the research background, significance and challenges of cloud storage performance isolation, and research on performance isolation technology on typical single-node storage and multi-node stor-

age platforms. Aiming at these two storage types, a more accurate and stable performance isolation algorithm is proposed, and an actual system is used to evaluate the experiment with real workload. The main contributions of this article are summarized as follows: For typical **stand-alone node - storage-hybrid storage**: 1)Quantitatively analyzes the cause of the inaccurate performance isolation of the classic isolation technology Linux Control Group. 2)Designed and implemented an adaptive precise throttling algorithm. 3)Conducted a detailed comparison experiment and proved that the adaptive precision throttling algorithm is superior to the Linux Control Group in terms of isolation accuracy and stability. For typical **distributed storage - Ceph**:1)Designed and implemented WinRWClock isolation algorithm based on time window.2)A lot of experimental comparisons with the classic performance isolation algorithm dmClock algorithm prove that the WinRWClock algorithm is superior in isolation accuracy.

Key Words : Cloud Storage Service, Distributed Storage, Hybrid Storage, Performance Isolation

目录

第 1 章 绪论	1
1.1 研究背景	1
1.1.1 云存储平台概述	1
1.1.2 单机节点存储的性能隔离	2
1.1.3 分布式存储性能隔离	3
1.2 本文的研究内容和组织结构	4
1.2.1 主要研究内容	4
1.2.2 主要创新点	4
1.2.3 组织结构	5
第 2 章 相关工作	7
2.1 云服务	7
2.2 云存储	9
2.2.1 单节点上的云存储	11
2.2.2 多节点上的云存储	12
2.3 资源隔离技术和算法	13
2.3.1 单节点上的资源隔离技术	13
2.3.2 多节点上的资源隔离技术	15
2.4 本章小结	16

第 3 章	单结点的性能隔离——以混合存储为例	17
3.1	研究动机	17
3.2	自适应精确隔离算法 (Self-Adaptive Accurate Throttling Algorithm, SAAT 算法)	19
3.2.1	单用户情况	19
3.2.2	多租户情况	21
3.2.3	算法设计与实现	21
3.3	实验设计与算法评估	23
3.3.1	实验配置	23
3.3.2	整体效果	24
3.3.3	不同 Trace 的影响	25
3.3.4	扩展性评估	26
3.4	本章小结	26
第 4 章	分布式架构下的性能隔离——以 Ceph 为例	28
4.1	问题分析	28
4.1.1	Ceph 分布式系统架构介绍	28
4.1.2	Ceph 系统中的性能隔离机制	32
4.2	基于时间窗口的时钟隔离算法 <i>WinRWClock</i> 算法设计与实现	39
4.2.1	算法设计	39
4.2.2	系统实现	44
4.3	实验评估	50
4.3.1	实验配置	50
4.3.2	整体效果	50
4.3.3	只设定预留目标的情况对准确度的影响	51
4.3.4	预留目标与权重目标共同作用对准确度的影响	53

4.3.5 预留目标变化对准确度的影响	53
4.3.6 I/O 粒度对准确度的影响	54
4.4 本章小结	55
第 5 章 总结与展望	56
5.1 本文总结	56
5.2 未来展望	56
参考文献	61
致谢	62

插图

图 2.1 <i>Cgroup</i> 工作示意图	14
图 3.1 混合存储设备 <i>Cgroup</i> 作用效果图	18
图 3.2 混合存储设备 <i>Cgroup</i> 不准现象及其原因	18
图 3.3 混合存储设备 <i>Cgroup</i> 不准现象及其原因	19
图 3.4 混合存储设备 <i>Cgroup</i> 不准现象及其原因	20
图 3.5 算法流程图	22
图 3.6 单用户：平均吞吐量差值 <i>DAT</i> 和	25
图 3.7 双用户：平均吞吐量差值 <i>DAT</i> 和平均波动值 <i>Average Jitter</i>	25
图 3.8 不同 <i>Trace</i> 组合：平均吞吐量差值 <i>DAT</i> 和平均波动值 <i>Average Jitter</i>	26
图 3.9 多用户：平均吞吐量差值 <i>DAT</i> 和平均波动值 <i>Average Jitter</i>	27
图 4.1 Ceph 架构图（汉化自 [7]）	31
图 4.2 Ceph I/O 算法图 [49]	31
图 4.3 Ceph I/O 调度示意图	33
图 4.4 dmClock 算法流程图	35
图 4.5 dmClock 算法的表现（配置一）	38
图 4.6 dmClock 算法的表现（配置二）	39
图 4.7 时间标签处理流程图	41

图 4.8 RequestTag 类	45
图 4.9 ClientInfo 类	45
图 4.10 ClientRec 类	46
图 4.11 I/O 队列方法类	48
图 4.12 I/O 时序图	49
图 4.13 只设定预留目标的情况对准确度的影响	52
图 4.14 预留目标与权重目标共同作用对准确度的影响（二）	53
图 4.15 预留目标变化对准确度的影响	54
图 4.16 工作负载粒度对准确度的影响	55

表格

表 1.1 本文组织结构	6
表 3.1 实验中采用的工作负载	24
表 4.1 dmClock 算法效果实验 (R,W,L) 配置信息	37
表 4.2 WinRWClock 实验参数配置	51
表 4.3 500K 粒度下不同配置 DAT 变化百分比	52
表 4.4 1M 粒度下不同配置 DAT 变化百分比	52

第 1 章 绪论

1.1 研究背景

1.1.1 云存储平台概述

过去 10 年间，计算机学术界和产业界推出了各种针对大数据分析、处理的应用、架构和基础平台。现如今，层出不穷的应用、框架运行在各种云环境 (Google Cloud Platform[14], Amazon EC2[2], Microsoft Azure[26] 等等)。云环境为企业及学术界提供了一种廉价且便于扩展的计算、存储资源。截至 2019 年，全球范围内数据中心的数据容量已经高达 1644 EB (1EB = 1000 PB)，是 2016 年的 1.8 倍；预计 2021 年，这个数据会达到 2300 EB。[38] 大规模的数据量访问给云端存储平台提出了高并发、可扩展的需求，许多分布式存储系统应运而生，如 Google File System[16]，Lustre[19]，Ceph[6] 等。

与此同时，共享 (Co-location)、邻居噪声 (Noisy Neighbor)、网络拥塞 (Network Congestion) 等多种原因，给分布式存储平台提供持久、稳定、可靠的性能提出了严峻挑战。服务质量 (Quality of Service) 保证——即租户能否得到云服务商保障的性能 (带宽、延迟等) 成为学术界不可忽视的课题。

性能隔离 (Performance Isolation) 是云平台必须要实现的技术之一。虽然云平台的各种使用者在逻辑上互不影响，但是在底层会共享内存、CPU、网络、存储等物理资源。这会导致不同使用者之间出现资源竞争的情况，使用者的性能会互相影响。性能隔离目标是保障每个虚拟资源租户得到相应的 CPU、内存、存储、网络等资源，同时避免用户之间的性能相互影响。例如，一种广泛使用的资源隔技术是 Linux Control Group (简称 Cgroup)。作为容器化技术的基础，Cgroup 可以隔离、限制、记录不同进程组的 CPU、内存、网络、存储资源的使用。再比如，广泛应用于网络的令牌桶 (Token Bucket) 算法也属于隔离算法的

一种，通过令牌个数限制发包数量，从而限制不同用户的网络流量。对于存储资源来讲，租户希望拿到稳定的带宽和延迟。但是与 CPU，内存等静态资源不同，存储平台由于存在多种类型存储设备构成缓存设备或者存储设备上构建了复杂的软件层（如文件系统、块设备层等）的情况，系统性能是动态变化的。因此存储平台不能在运行前给诸多用户分配好资源，这为存储平台的性能隔离增加了难度。

云端可租用的存储资源多种多样。可以按照结点类型分为单机节点上的存储和多机节点上的存储。这两个类型性能隔离技术面临的挑战是不同的，接下来章节**1.1.2** 单机节点存储的性能隔离和子章节**1.1.3** 分布式存储性能隔离分别介绍了两种典型存储类型混合存储和分布式对象存储的特点、性能隔离技术及其面临的问题。

1.1.2 单机节点存储的性能隔离

1.1.2.1 应用前景

云存储上的单机节点负责接收和处理用户请求的，在单机节点上我们只需要关注不同的用户请求到达单机节点后，怎样提高性能和降低成本。混合存储就是云存储供应商常用的解决方案。混合存储是慢速设备（Slow Device，简称 SD）与快速设备（Fast Device，简称 FD）的组合，这样既可以利用快速设备的高吞吐量也可以利用慢速设备高性价比的空间。同时，混合设备也给服务商不同级别的用户提供了多种销售选项。可以将 SSD 设置为 HDD 的缓存，甚至可以将新兴的非易失性内存（NVM）设备设置为 SSD 的缓存，以提供不同级别的服务。混合设备的另一个优点是屏蔽了内部细节，使用者无需对应用软件进行修改，直接使用常见的块设备访问接口即可访问。一种常见的混合方法是在操作系统级别将不同的设备组合到一个逻辑设备中。例如，Facebook 的 Flashcache[33]，利用 I/O 重定向技术 device mapper[20]，在内核里实现了缓存优化算法。但是 Linux 系统仍然把此混合设备当成普通的单一设备向用户提供服务。

1.1.2.2 混合存储性能隔离面临的挑战概述

设备映射器（Device Mapper）技术 [20] 的存在，使得两个设备可以当作一个逻辑设备访问。因此，广泛应用的性能隔离技术 Linux Control Group（简

称 Cgroup) 可以不做任何修改地应用到混合设备上。但是通过实验我们发现 Cgroup 技术在混合设备上效果并不好，即共享混合设备的租户们无法拿到目标性能。在章节③“单个结点的性能隔离——以混合存储为例”中，本文通过实验和定性分析揭露了相关原因：由于混合设置存在缓存不命中的情况，在命中率的时刻，慢速设备无法支持用户需求的吞吐量，而快速设备的带宽用不满，从而导致实际的平均吞吐量达不到目标，服务质量难以保证。

1.1.3 分布式存储性能隔离

1.1.3.1 分布式存储系统的应用前景

新型分布式文件系统采用基于对象的存储的体系结构，其中传统的硬盘已被智能的对象存储设备（OSD）取代。该设备将 CPU，网络接口和本地缓存与基础磁盘或 RAID 结合在一起。OSD 用一种取代传统的块级接口，客户端可以在其中读取或写入更大范围（且大小通常可变）的对象（Object），将低级块分配决策分配给设备本身。客户端通常与元数据服务器（MDS）交互以执行元数据操作（打开，重命名），直接与 OSD 进行通信以执行文件 I/O（读取和写入），从而大大提高了总体可扩展性。采用这种模式的系统由于元数据工作负载很少或没有分配，因此可扩展性受到限制。如果依赖传统的文件系统原理，如分配列表和索引节点表，以及不愿意将自主性分配给 OSD，将进一步限制了可扩展性和性能。

Ceph[6] 就是采用了对象存储的分布式文件存储系统。它是一种具有出色性能和可靠性，同时也具有无与伦比的可扩展性。其架构基于以下假设：PB 量级系统内在是动态变化的；大型系统不可避免地是增量构建的，节点故障是正常现象，而不是例外情况。Ceph 通过将数据访问操作和元数据操作解耦合，用计算文件位置直接定位目标 OSD 代替搜索文件分配表定位 OSD。同时，Ceph 充分利用了 OSD 的自主性来实现数据访问、序列化更新、多副本、故障监测和恢复等功能。Ceph 利用高度自适应的分布式元数据集群架构，极大地提高了元数据访问的可扩展性，并因此提高了整个系统的可扩展性。

1.1.3.2 分布式存储系统性能隔离面临的挑战概述

分布式文件系统上的经典性能隔离算法是 2010 年提出的 dmClock[36] 算法。其提出了预留（Reservation）、限制（Limit）、权重（Weight）三个目标。希望一个存储系统，能在预留给用户目标带宽资源的基础上，按照指定的权重分配带宽资源，同时满足其限制条件。此目标的实现，需要保证预留资源和权重资源的优先级顺序，即先保证预留目标达成，再按权重瓜分剩余的系统资源。dmClock 算法使用了多时钟标签调度的方法，通过两阶段调度来限制不同标签的调度顺序。Ceph 系统就在其 I/O 调度层实现了 dmClock 算法队列，以实现用户的服务质量保证。但是，dmClock 并不能完美地实现性能隔离目标。章节 4 “[分布式架构下的性能隔离——以 Ceph 为例](#)” 通过实验和算法分析展示了 dmClock 性能隔离效果及相关原因：dmClock 多时钟标签调度的设计存在问题，无法保证用户之前资源调度的优先级顺序。

1.2 本文的研究内容和组织结构

1.2.1 主要研究内容

本文分别针对两种存储类型，提出了更加稳定和准确的资源隔离算法。具体来说，针对混合存储，提出了**自适应精确隔离（SAAT）** 算法，解决了经典隔离技术 Cgroup 不准确的问题；针对分布式存储系统，提出了基于时间窗口的多时钟调度算法 **WinRWClock**，并在典型的分布式文件系统 Ceph 上予以实现。

1.2.2 主要创新点

本文的主要创新点如下：

- 1) 定量分析了经典性能隔离技术 Cgroup 性能隔离不准确的原因。
- 2) 设计并实现了**自适应精确隔离（SAAT）** 算法。
- 3) 在实际混合存储设备上进行了详实的对比实验，证明自适应精确隔离(SAAT) 算法在准确性优于 Cgroup，平均提升 4.62%。

- 4) 在实际分布式存储平台 Ceph 上进行实验，展示了经典性能隔离算法 dm-Clock 的效果，并从原理出发分析了其表现不好的原因。
- 5) 设计并实现了基于时间窗口的多时钟调度算法 **WinRWClock**。
- 6) 在 Ceph 平台上的大量实验证明，WinRWClock 算法在隔离准确性上优于 dmClock 算法，平均提升 75.98%。

1.2.3 组织结构

本文共分为五章，组织结构框架如表1.1所示。

在第 1 章绪论部分中，主要介绍了云端存储资源性能隔离的研究背景及意义和面临的挑战，并对本文的主要研究内容和主要创新点进行了阐释，说明了本文的组织结构。

在第 2 章相关工作部分，首先从单节点和多节点两方面梳理了当前云存储平台的发展状况，接着分别对单节点和多节点上的资源隔离技术进行了总结梳理。

第 3 章、第 4 章分别选取了单节点和多节点典型的存储类型——混合存储和分布式存储进行研究。其中，第 3 章首先展示了混合存储上性能隔离技术 Cgroup 的问题，并定量分析其原因，接着介绍本文提出的**自适应精确隔离（SAAT）**算法，最后与 Cgroup 进行了对比实验。第 4 章首先进行问题分析，介绍了 Ceph 分布式平台架构和其上应用的性能隔离算法 dmClock 原理，分析了 dmClock 的表现和原因；接着，介绍**基于时间窗口的多时间标签隔离算法 WinRWClock**设计与实现；最后，展示了在 Ceph 平台上进行实验评估的过程。

第 5 章为总结与展望，对本文提出的自适应隔离算法和基于时间窗口的隔离算法进行了总结，并指出了本文的不足之处和未来存储资源隔离的研究方向。

主要内容	章节结构	内容说明
本文研究基础	第一章 绪论	主要介绍本文的研究背景、意义、创新点和研究现状。
	第二章 相关工作	
云端存储性能隔离算法研究	第三章 单个结点的性能隔离	展开了以混合存储为例的研究。定量分析混合存储上经典隔离技术 Cgroup 无法实现性能隔离目标的原因；自适应精确隔离算法设计与实现；实验设计与算法评估。
	第四章 分布式架构下的性能隔离	以分布式存储系统 Ceph 为例展开研究。Ceph 性能隔离机制问题分析；基于时间窗口的多时钟隔离算法 WinRWClock 算法设计与实现；实验评估。
本文总结	第五章 总结与展望	总结了本文的研究成果和不足，并对未来的该领域的研究工作做出展望。

表 1.1 本文组织结构

第 2 章 相关工作

2.1 云服务

云服务因为大幅度降低了成本和增加了使用的方便程度，在当今时代得到了广泛的应用。根据 NIST 的定义，云计算可以简便和按需分配地访问通过网络共享的可配置计算资源，云计算可以被视为提供上述功能的模型。可配置计算资源包括网络、服务器、存储资源、应用和服务等等。云计算上的资源能快速地分配和释放，最小化管理成本和服务供应商的参与。[\[44\]](#)

具体来说，云计算相当于为用户提供要通过网络访问的无限可定制资源。用户可以随时更改自己需要资源的数量和质量。过去，一家企业起步阶段需要花费大量成本购买和配置基础设施。现在，新企业只需要通过云服务就可以低价租用少量资源，并且根据企业的发展程度不断调整使用云资源的数量。等到企业发展的步上正轨，企业获得了足够的流水，可以决定是继续使用云服务还是配置自己的基础设置或者私有云。

云服务提供的服务按照资源类型主要分为三类：Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)) [\[51\]](#)。IaaS 是指将整个框架作为服务提供给用户，通常是以虚拟机的形式，例如 Amazon EC2[\[2\]](#), GoGrid[\[8\]](#) 和 Flexiscale[\[12\]](#)。PaaS 是指将操作系统或者软件开放框架等平台资源作为服务提供给用户，例如 Google App Engine[\[13\]](#), Microsoft Windows Azure[\[26\]](#) 和 Force.com [\[24\]](#)。SaaS 是指将应用作为服务提供给用户，例如 Salesforce.com[\[24\]](#), Rackspace [\[9\]](#) 和 SAP Business By Design [\[25\]](#)。大量的应用服务供应商会购买框架和平台服务供应商的产品作为基础提供他们的应用服务。

云服务也可以按照云的访问权限分类，公有云是开放给其他用户访问的，通常是公司或者公益组织等为用户提供的服务；私有云则被放在公司的防火墙内部，主要由公司内的人员使用，保证公司业务和数据的安全性；混合云则是既

使用私有云资源，也使用公有云资源，从而更全面的满足用户的需求。

根据 [47]，云服务的生命周期可以分为五个环节：

- 1) Selection: 用户选择自己需要的云资源或者性能需求。有一些可配置计算资源非常好定量，例如用户可以指定自己需要一个配置为 4 核 1G CPU, 4G 内存, 50GB SSD (固态硬盘) 和 1TB HDD 的 VM (虚拟机)。但也有一些可配置计算资源难以定量，例如网络资源，用户的目的只是获得他需要的带宽，而满足这个带宽需要为他分配多少的网络服务器时长或者更多的内部细节用户难以了解，大量用户共享的复杂网络环境也让具体的配置无法提前确定。这种情况下用户只要说出自己的性能需求，也常被称为 SLA (service level agreements)，即用户和服务供应商达成一致的用户对服务质量的需求。还有一些情况是用户既有对资源定性的需求，也有对性能的需求。比如用户想租用云存储服务，需要 1TB 的存储空间，以及不低于 100MB/s 的带宽。
- 2) Configuration: 云平台将用户所需要的资源或者性能需求转化为云平台上的资源配置。这一步看上去简单，实际上非常复杂。因为云环境上的资源大多数情况下是不一致的。比如用户说要 1TB 的存储空间，平台上不仅有 NVM, SSD, HDD 等多种类型的存储磁盘，每种类型的磁盘可能也存在不同的种类和质量，低端的 SSD 和高端的 SSD 速度相差超过一个数量级。使用时间等因素也会影响设备的性能和可靠程度。以 SSD 为例，使用时间越长，擦除次数越多，设备的可靠性就会越低。在这种情况下，如何既满足用户的需求，又最小化服务供应商的成本，变成了一个异常复杂的问题。
- 3) Deployment: 云平台根据资源配置需求将资源实际分配给用户。
- 4) Monitoring: 云平台对用户对资源的使用情况进行实时监控。为了节约成本，大多数情况下云服务供应商不会傻傻地将用户需要的资源以预留的方式完全分配给用户使用。例如，用户说自己的虚拟机需要 4G 的内存，实际上用户只用了 1G，剩余的 3G 供应商可以用来供给其他用户。这就需要云平台监控用户的使用情况，发现资源闲置就通知其他模块进行资源重分配。监控最复杂的部分是，有些情况下很难判断资源是否闲置。以内存为例

例，Linux 的操作系统有虚拟内存功能，内存既可以是真实的 DRAM，也可以将磁盘作为虚拟内存，将 DRAM 无法承载的内容写在磁盘中，在需要的情况下再载入到内存。换言之，用户需要 4G 内存的时候，在磁盘的空闲空间足够的情况下，既可以给用户 3G 的 DRAM，也可以给用户 1G 的 DRAM，只是越小的 DRAM，就会有越频繁的磁盘加载操作，会导致性能越慢。假设用户的对内存的需求不变，供应商只能观察到随着给的 DRAM 容量降低，用户的虚拟机获得的性能降低的情况，但是供应商难以确定性能降到哪个程度刚好是用户能接受的下限。而现实中用户对内存的需求是不可能不变的。供应商既难以预测用户对内存需求的变化，也难以准确地定性分析在某个时间点，更改内存供应导致的性能变化情况，更无法知道用户对性能的满意程度。这就导致想要“监控”用户使用的资源是一个不可能完美完成的任务。

- 5) Control: 根据监控结果，云平台对资源进行调配，如果资源存在闲置，可以将用户不需要的资源分配给其他需要的用户；如果用户资源不足，就分配给用户更多的资源。严格来说，云服务的出现并不算划时代的新技术，因为它只是将各种现存的技术进行组合（虚拟化技术、互联网技术、分布式技术等等）。但是，云服务在应用不断增加的过程中，遇到了大量的技术挑战，吸引了科研界和企业界的注意。例如可靠性问题。云环境是前所未有的复杂环境，不仅包括了各种各样不同配置和不同地区的硬件和网络设施，也包含了前所未有的不确定因素。云上的任何一个节点都可以突然掉电、断网或者发生硬件故障。这对用户的服务质量和数据可靠性的保障造成了极大的威胁。供应商需要在成本和可靠性上选取一个适当的平衡。然而对云环境进行可靠性评估都很难做到。[34]

2.2 云存储

如今是大数据时代，社交网络和电子商务由于广泛的使用产生海量的数据。这些数据需要保证准确性和私密性，以满足用户和企业的需求。同时，数据本身蕴含了巨大的价值，电子销售的网站可以通过用户的历史购买信息分析用户的购买习惯，为用户准确推送商品。企业本身和科研等领域也带来了海量增长的数据。这些数据的存储、使用和管理成为了巨大的挑战。全球范围的数据规模

每两年翻一倍，2013年的全球数据规模是4.4ZB($1\text{ZB} = 1024\text{EB} = 1048576\text{ PB}$)，2020年初这个数字已经达到44ZB。[\[42\]](#)大数据需要访问便捷、价格低廉、管理方便、安全可靠的存储媒介，云存储应运而生。

云存储的目标是提供按需分配、按照使用付费的存储服务，用户可以根据自己实时的存储需求变化随时增加或者减少使用的存储空间，最小化用户存储数据的经济成本。云存储除了更低的价格之外，还需要满足数据的高可用性、耐久性、低延迟、数据安全和数据一致性等。数据的高可用性是指数据在任何情况下都是可用的。如果服务器发生掉电，内部的数据在通电后可恢复，但是掉电期间无法访问，掉电的时间段服务器内的数据处于不可用状态。数据的高可用性要求云存储对数据进行多备份和容灾备份，可以应对各种突发情况。数据具备耐久性，就是说数据不会因为长时间不被访问就产生错误或者丢失。很多存储设备是通过电属性的变化存储数据信息，在长时间不通电的情况下会因为缓慢放电等现象，数据产生错误和丢失。数据的耐久性可以通过数据定期维护来实现。数据安全是指其他人不能通过任何手段知道用户要求保密的数据信息。云存储的数据安全涉及到很多方面，网络、大量存放数据的节点、管理数据的节点等等，任何一个环节有漏洞，都可能让黑客趁虚而入，窃取用户的数据。数据一致性是与事务有关的概念，一次事务的执行前后数据都是一致的，事务执行到一半就不执行也不回滚到执行前的状态就是违背了数据的一致性。。事务的一致性就是要保证在任何情况下，数据都是一致的。当数据在云存储中进行多备份的情况下，保证数据的一致性和保护用户得到的性能通常是矛盾的，需要根据具体情况选择合适的数据一致性等级和性能保障。

提供云存储服务的有阿里云[\[30\]](#)、腾讯云[\[28\]](#)、百度云[\[27\]](#)、提供块存储服务的 Blob Storage[\[4\]](#)、提供云盘服务的 Google Drive[\[17\]](#)、提供对象存储的 Google cloud storage[\[14\]](#) 和提供关系数据库服务的 Azure RDS[\[21\]](#) 等等。即使已经有了如此多的商用云存储服务，云存储并不成熟，仍然存在很多隐患和挑战。例如，2018年8月，腾讯因为人为操作失误和硬盘失误造成部分云存储数据丢失，无法找回，发布公开道歉[\[29\]](#)。此外，目前云存储的发展趋势是不同云之间进行连接，如何尽可能减少甚至完全不需要用户对于不同的云存储供应商进行复杂的操作和调整也成为了重要的研究方向。随着遍布世界各地的存储节点互相连接，网络延迟也成为影响云存储的重要因素。云存储上数据可能被存储在多个在物理上相隔很远的节点上，严格保证数据的一致性会导致很长的数据访问延

迟，极大地降低云存储的性能，所以云存储的一致性和性能的权衡取舍也成为了学术界和工业界共同关注的问题，需要根据用户需求和实际的节点分布等云存储的配置情况进行调整。

云存储可以分为两个层次，在单结点上的数据存储和在多节点上的分布式数据存储。单结点上的数据存储是只看发生在一个物理机上的存储问题，多节点则主要关注用户请求如何被分配到了不同的节点。单结点云存储是多节点云存储的基础。[2.2.1](#)介绍单节点上的云存储的情况，[2.2.2](#)介绍多节点上的云存储的问题。

2.2.1 单节点上的云存储

在云存储的结构中，每个用于存储数据的节点会存储属于不同用户的数据，当相应的请求被分配给这个节点的时候，节点需要及时处理用户的请求并且返回结果。也就是说，单节点上的云存储的目标就是接受和处理上层分发的请求。在单个节点上，最重要的问题是提升性能、资源隔离和安全性问题。如果节点上的性能不足，处理请求时间过长，用户获得的整体性能就会变差。资源隔离是指不同用户之间不会相互影响。资源隔离的详细内容在[2.3](#)中介绍。安全性问题要保证不同用户不能访问对方的数据，保证黑客无法从单个节点中窃取用户的私密信息。

混合存储是云存储的存储节点上常用的存储配置，用于提升存储性能。混合存储是指将快速设备（例如新型 NVM 盘和 SSDs）和慢速设备（多为 HDDs）包装为一个逻辑设备供上层访问。通过将比较热的数据放在快速设备，比较慢的数据放在慢速设备，既提供了近似快速设备的性能，又大幅降低了价格。混合存储的常用技术有 dm-cache[\[40\]](#)，Flashcache[\[33\]](#) 和 bcache[\[5\]](#) 等。

dm-cache[\[40\]](#) 是基于内核 device mapper 机制实现的一种包装快速设备和慢速设备成一个混合设备的方法。通过动态地将数据移动到快速设备，可以达到提升性能的目的。dm-cache 要求用户创建混合设备时，指定一个原设备 (origin device)，一个缓存设备 (cache device)，一个元数据设备 (metadata device)。原设备一般是容量大，速度慢的慢速设备 (如机械硬盘)；缓存设备一般是容量较小，速度快的快速设备 (如固态硬盘)；元数据设备用来存储缓存中数据块信息 (如是否是脏块等信息)。

Flashcache[\[33\]](#) 是 Facebook 为 InnoDB 数据库加速而实现的混合存储模块。

与 dm-cache 类似,这项技术也是基于 Linux 内核自带的 device mapper 功能。其要求用户指定缓存设备和源设备。常见情况是固态硬盘 (SSD) 和机械硬盘 (HDD) 的混合方案。除此之外, 其他快速设备和慢速设备的混合方案如 NVM+SSD, NVM+HDD 都是支持的。

bcache[5] 是 Linux 内核块设备层 cache, 与 Flashcache 相比更加灵活, 支持固态硬盘作为多块机械硬盘的共享内存, 而且可以在运行中动态增加, 删除缓存设备和后端设备。其使用 B+ 树来维护索引, 虽然比 Flashcache 复杂, 也已经进入内核主线, 但是还存在不稳定的情况, 与 Flashcache 相比不太成熟。

2.2.2 多节点上的云存储

单个节点可以实现对多用户请求的处理之后, 我们把关注点转移到多节点上, 关注用户从发出请求到请求来到某个节点之间发生了什么。首先, 云存储要为用户提供一套定价标准, 明确不同的存储容量和带宽延迟安全性等性能需求对应的价格, 让用户从中选择。接下来, 像2.1中介绍的, 用户的性能需求被云存储系统的管理模块转化为实际给用户分配的资源。用户实时产生的数据请求也会由管理模块按照指定的规则分发到相应的节点上。

多节点需要完整可靠的用户身份认证机制, 既保障用户的数据安全和服务质量, 也保护自己的资源不会被未付费的黑客用户盗用。目前云存储中常见的用户认证技术有密码学的密钥技术。云存储中的数据放置问题与安全、性能密切相关。如果云存储用一套公开的规则或者索引计算数据放置的位置和方式, 例如 Ceph[6][48], 这让盗取数据并对应到不同的用户变得更容易。但与此同时, 这样的技术大大降低了数据访问的延迟, 提升了性能, 所以 Ceph 作为分布式存储系统得到了广泛应用。在可靠性高的私有云环境下, Ceph 具备相当高的吸引力和实用性。除了安全性之外, 云存储系统要考虑数据放置的平衡问题。不同的数据的热度, 即在单位时间内被访问的次数, 是不同的。如果数据放置不平衡, 一个节点的数据都比较热, 节点的承载能力不足, 剩下的节点则因为数据过冷, 导致闲置, 系统的空闲能力就被浪费了。因此, 云存储系统会通过对数据流量的监控, 适时地进行资源迁移, 保证流量的均衡。

多节点上还有一个重要的功能, 就是资源隔离。当一个用户的访问请求超过他购买的性能时, 要对用户的访问进行限流。但是假如用户的访问请求没有超过他们的需求, 且系统的总承载能力足够的情况下, 如何进行流量引导以便

每个用户都能得到他们应得的流量，不同用户之间不会互相干扰，是本文关注的重点。[2.3](#) 将会对这一问题和目前的解决方案进行详细的阐释。

2.3 资源隔离技术和算法

资源隔离是指不同用户之间使用资源不会互相影响，即使在同一个资源池上运行，用户也感觉自己独占一个更小的资源池，不会被其他用户的负载影响。资源隔离也可以分成单节点上的资源隔离和多节点上的资源隔离。[2.3.1](#) 介绍单节点上的典型资源隔离技术，[2.3.2](#) 介绍多节点上的资源隔离技术。

2.3.1 单节点上的资源隔离技术

云计算最常用的技术是虚拟化，无论节点实际上使用什么类型的资源，云服务都可以通过虚拟化的方式让用户可以使用他们想要的环境。以 docker[\[11\]](#) 为代表的容器技术因为性能好，得到了广泛应用。Docker 的资源隔离主要基于两个技术，Namespace[\[18\]](#) 和 Cgroup[\[37\]](#)。

2.3.1.1 Namespace

Namespace[\[18\]](#) 是 Linux 内核提供的资源隔离技术。Namespace 为用户指定命名空间，用户只能使用自己命名空间内的资源，进而实现资源隔离。Namespace 一共支持 6 种资源的隔离，包括 IPC、Network、Mount、PID、UTS 和 User。IPC 是进程间通讯。Linux 系统中有很多种类的进程通讯，例如 Posix 消息队列和共享内存。IPC Namespace 让不同 namespace 下的相同标识符对应不同的消息队列，通讯时消息标识符要在自己的 namespace 下寻找对应的消息队列，不同 namespace 下的进程无法进行进程通讯。Network Namespace 隔离网络资源，每个 Network Namespace 有且只有一个自己的网络设备，用户可以自行配置自己的网络设备。Mount Namespace 隔离文件系统挂载点，每个用户挂载自己的文件系统，并且只能看到自己的文件系统，其他用户在自己的 Namespace 下进行任何操作都不会影响到自己的文件系统。PID Namespace 用于隔离进程 ID，每个 Namespace 内部有自己专门的进程 ID 表，不同 Namespace 下可以有相同的 PID，实际对应不同的进程。UTS Namespace 的用途是隔离主机名和域名。Network Namespace 已经隔离了 IP 地址，但是主机名可以代替 IP 地址实现网络访问。UTS Namespace

隔离了主机名和域名，确保不同用户之间不会互相干扰。User Namespace 是通过隔离用户 ID 和组 ID 对用户权限进行管理。用户可以在自己的命名空间内拥有 root 权限，但是对命名空间外的资源则不具备此权限。虽然 Namespace 提供了对很多种类资源的隔离能力，但是它并不能将 host 机的所有资源都进行隔离。Cgroup(control groups)^[37]是 Linux 内核自带的资源隔离工具，提供更加完善和强大的资源隔离功能。

2.3.1.2 Linux Control Groups

Linux Control Group (Cgroup) ^[37]是目前资源隔离领域最主流的工具，由 Google 开发，目前已经被集成进 Linux 内核。Cgroup 可以用来限制、控制与分离一个进程组的资源，如 CPU 时间，系统内存，I/O 和网络带宽。这项技术被广泛应用于云环境、虚拟化容器，来管理、分配、限制不同应用的资源使用情况。Cgroup 的工作情况如图所示。图2.1中将不同进程，比如数据库进程或者虚拟机进程，划分进不同的组 (Group)，并且给每个组指定资源分配情况——按照指定格式写进 Cgroup 的配置文件，可以只限定一种或几种资源。为了控制这些资源，Cgroup 会对每个组中进程使用资源的情况进行监视、记录、统计。这样，不同组之间资源就被隔离开，达到资源隔离的目的。

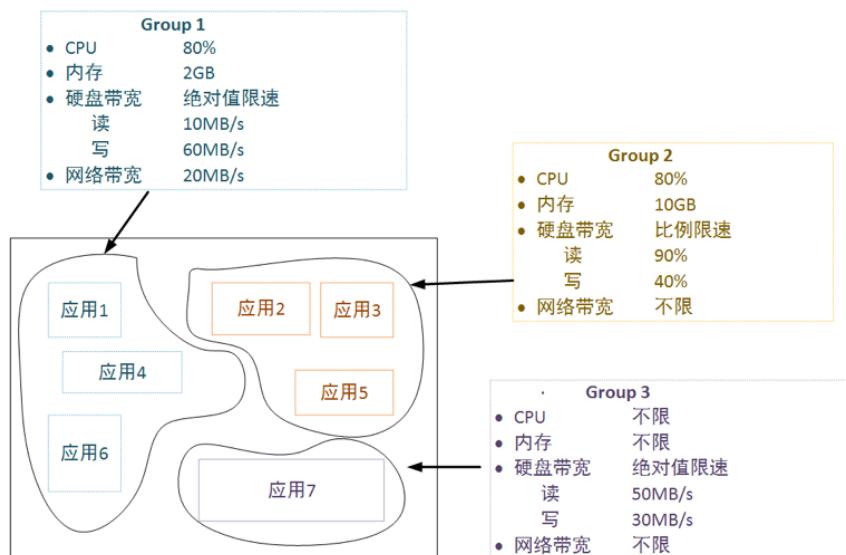


图 2.1 Cgroup 工作示意图

Cgroup 中限制 I/O 资源有两种方式：比例权重限速和绝对带宽/IOPS 限速

如图2.1所示，Group2是比例权重限速，Group1和Group3是绝对值限。比例权重限速是通过设置不同权重值来达到限速的目的。绝对带宽/IOPS限速是给定一个固定带宽或IOPS的限定值最为上限，Cgroup控制此用户组访问该设备时的带宽/IOPS不会超过此上限。混合设备上可以用绝对带宽/IOPS限速的方式来限速；但是由于比例权重限速是基于Linux内核中I/O调度层的CFQ调度算法实现的，而上述提到的基于Flashcache等工具搭建的混合设备由于在逻辑设备上跳过了I/O调度层的代码，因此无法利用Cgroup的比例权重限速功能。

2.3.2 多节点上的资源隔离技术

在云存储中，一个用户的数据存储访问服务可能由多个存储节点共同完成，这些节点可能分布在相隔万里的物理机器上。不仅如此，去中心化的云存储服务很可能没有与这些节点均相连的中心管理节点。所以，如何去监督与管理用户在每个节点上得到的服务质量，是多节点下的资源隔离技术的重大挑战。很多资源隔离技术通过引入中心管理节点的方式简化了挑战，但是中心管理节点与去中心化的云计算趋势相违背，增加了整个系统的成本负载，降低了系统性能。`dmClock`[36]技术巧妙地在不引入中心管理节点的情况下解决了上述问题，成为了代表性的多节点下资源隔离技术方案。

2.3.2.1 dmClock技术

`dmClock`技术的目标是在分布式存储的场景下，即使不同的节点承担的负载不平衡，也能保证系统总的reserve, weight和limitation都符合要求。`dmClock`将用户希望得到的性能和资源隔离目标细化为三个指标：reserve, weight和limitation。Reserve是用户希望得到的最小资源总量。系统的服务能力是波动的，系统需要保证自己的最低服务能力不低于所有用户总reserve量，否则系统就无法满足用户的需求。Weight是用来衡量不同用户的权重的，当用户在reserve范围内的请求数量都被满足之后，系统的能力按照weight的份额分配给各个用户。Limitation是每个用户能得到的最高性能。`dmClock`要求所有用户预先定义好自己的reserve, weight和limitation的值。虽然没有中心节点，但是用户端是和所有要服务的节点相连的，从用户的角度来看，用户端实际上是这个用户特有的中心节点。所以`dmClock`让用户端的接口负责监督和管理这个用户的流量情况。`dmClock`的用户端只需要在向每个服务节点发送的数据访问请求中附带几个字节表示其他节点的服务完成情况，就可以让每个服务节点知道其他节点的情况。

此外，每个服务节点处理好数据访问请求后发送给用户端的信息中也附带几个字节表示自己的服务完成情况即可。dmClock 用这样的模式最小化了多节点下资源隔离的开发成本、通讯成本和性能负载。dmClock 是通过标签机制满足 reserve, weight 和 limitation 的目标的。标签机制的具体内容在章节4.1.2.2中具体解释。

2.4 本章小结

本章介绍了论文的相关工作，首先介绍了云服务的基本概念和发展现状，然后在此基础上介绍了云存储的概念和现状，以单节点和多节点作为分界分别介绍了两种环境下的主要挑战和代表性的技术。最后，本章介绍了资源隔离技术在单节点和多节点上的资源隔离技术。

第3章 单结点的性能隔离——以混合存储为例

本章节以混合存储为例研究了单节点的性能隔离机制。章节3.1首先介绍了混合存储的基本工作原理，接着详细介绍了Linux Control Group在混合设备上的表现，并定量分析了原因。3.2根据混合设备内部工作原理，推导得出

3.1 研究动机

混合存储设备结合了快速设备与慢速设备的优势，同时由于对用户来说是同一个逻辑设备，便于在平台间移植。在混合设备进行限制时，Cgroup作为广泛使用的资源限制工具，可以直接作用在混合设备上，如图混合存储是指将不同类型的存储设备组合在一起，以保护内部细节。一种常见的方法是在操作系统级别将不同的设备组合到一个逻辑设备中（例如，Facebook的Flashcache [33]）。近年来，一些研究[46, 50, 41]重点是在混合存储中应用基于SSD的缓存以实现高I/O速度。对于通常具有写持久性限制[31]的SSD或NVM设备，其他一些研究提出了一些写高效的缓存替换算法，以平衡缓存命中率和缓存设备生存期[35, 45, 39, 32]。

但是，当我们使用Cgroup设置进程组的吞吐量或IOPS的上限时，我们会发现，在混合设备上不断执行I/O请求时，目标吞吐量和测量的吞吐量之间存在差距。如图3.1所示。

混合设备上的吞吐量节流不准确的原因在于，由于时间上的命中率较低，混合设备中的SD会在某个时间段内承担大多数请求，并且它无法提供足够大的带宽来满足租户的目标吞吐量，如图3.2所示。

在这种情况下，尽管FD可以提供比目标端口更高的吞吐量来提升平均带宽，但是FD的带宽受到Cgroup的严格限制。如果我们可以将吞吐量限制提高到适当的值，则混合设备的平均吞吐量可以达到用户的指定值。

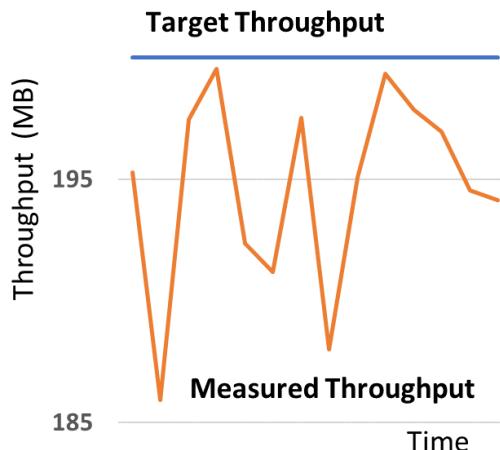


图 3.1 混合存储设备 Cgroup 作用效果图

因此，精确吞吐量调节的一种可行解决方案是在混合设备上适当地促进 Cgroup 的指定吞吐量调节，以使 FD 和 SD 的平均总吞吐量达到用户指定的目标。但是，混合设备中的内部行为很复杂，并且状态不断变化（例如，高速缓存命中率，FD 和 SD 的工作时间重叠等），因此在混合设备上实现精确的吞吐量调节是一个挑战性的问题。为了在混合存储设备上实现准确的性能隔离，我们提出了一种称为自适应精确节流（SAAT）的新方法。

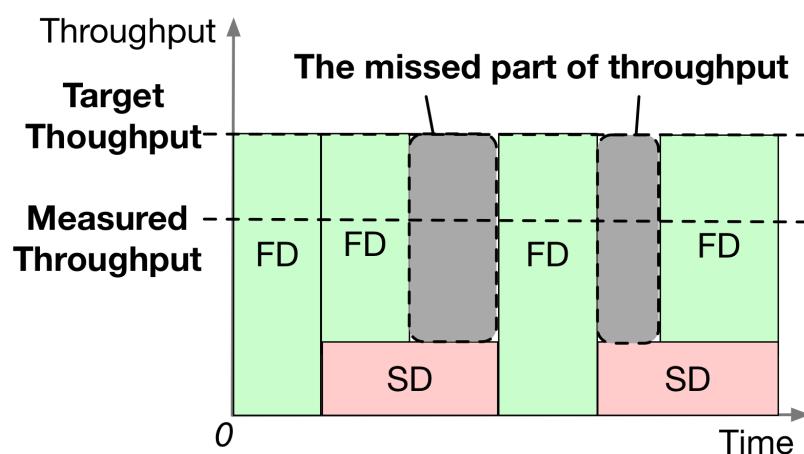


图 3.2 混合存储设备 Cgroup 不准现象及其原因

如图 3.3 所示，Cgroup 用于限制租户的读取和写入带宽。假设目标读写吞吐

量值为 $Th_{R1}^T, Th_{W1}^T, \dots, Th_{Rn}^T$ 和 Th_{Wn}^T 。 n 个租户，原始 Cgroup 将其用作分配的吞吐量限制值。通过限制 Cgroup，这些租户的实际测量的吞吐量值（即 $Th_{R1}^M, Th_{W1}^M, \dots, Th_{Rn}^M$ 和 Th_{Wn}^M ）通常小于分配的值（即 $Th_{R1}^A, Th_{W1}^A, \dots, Th_{Rn}^A$ 和 Th_{Wn}^A ）。

在这种情况下，我们提出的 SAAT 方法负责为所有租户确定一组适当的分配的吞吐量值，这些值通常大于目标吞吐量，以使测得的吞吐量尽可能接近目标。通过收集最新的统计信息以及实际吞吐量与指定吞吐量之间的差异，SAAT 可以定期为 Cgroup 提供适当的已分配吞吐量值。在以下各部分中，我们将介绍如何确定单租户情况和多租户情况下的分配吞吐量设置。

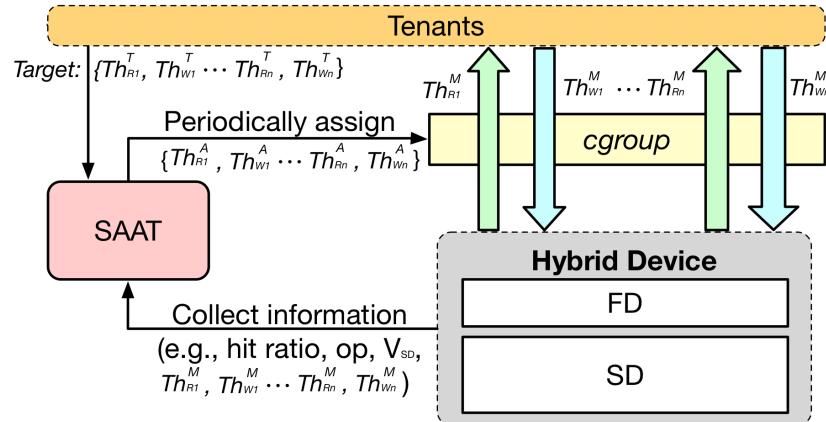


图 3.3 混合存储设备 Cgroup 不准现象及其原因

3.2 自适应精确隔离算法 (Self-Adaptive Accurate Throttling Algorithm, SAAT 算法)

3.2.1 单用户情况

单租户情况意味着只有一个用户正在使用混合设备。这是多租户案例的基础。我们假设缓存在写回模式下工作，并且有足够的 I/O 请求，即不少于目标和分配的吞吐量，即 Th^T 和 Th^A 。如图所示，所有写请求将首先写入 FD，而那些未命中的缓存将触发将受害者写回到 SD。

对于读取的请求，不同之处在于错过的请求不会导致 FD 正常工作。因此，即将到来的请求 req_f 的总字节数等于 Th^A 乘以 FD 的服务时间（即 t_f ）。在请

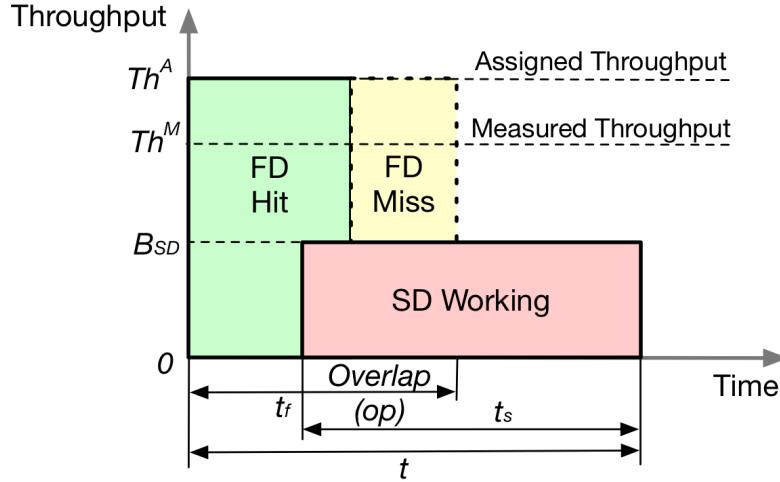


图 3.4 混合存储设备 Cgroup 不准现象及其原因

求处理期间，FD 和 SD 可以并行工作。因此，我们用 t_s 表示 SD 工作的时间，用 t_f 表示 FD。由于 FD 和 SD 通常并行工作，因此我们使用 op 表示 t_f 中重叠时间的比例。测得的吞吐量 Th^M 等于即将到来的请求的总字节数 req_f 除以给定的时间段 t ，如等式(3.1)所示，用于读写请求。

$$Th^M = \frac{Th^A \times t_f}{t_s + t_f - op \times t_f} \quad (3.1)$$

t 期间内所有高速缓存未命中都会触发 SD 工作。由于后端操作不会受到 Cgroup 的限制，并且 SD 通常是瓶颈，因此 SD 通常会以其全带宽（即 B_{SD} ）处理 I/O 请求。这样我们就可以得到等式(3.2)。

$$t_s \times B_{SD} = R_{miss} \times req_f \quad (3.2)$$

通过在等式中消除 t_f , t_s 和 req_f 。(3.1) sim 式(3.2)，我们得到 Th^M , Th^A , R_{miss} 和 B_{SD} 之间的关系，如等式(3.3)所示。

$$\frac{1}{Th^A} = \frac{1}{1 - op} \times \left(\frac{1}{Th^M} - \frac{R_{miss}}{B_{SD}} \right) \quad (3.3)$$

3.2.2 多租户情况

当多个租户共享一个设备时，对于 $tenant_i$ ，测量的吞吐量 Th_i^M ，分配的节流阀 Th_i^A ，FD 服务时间 t_{f_i} ，SD 服务时间 t_{SD}^i 和整个混合设备的重叠率 op 也符合等式(3.1)。

在此期间，SD 将处理所有租户的遗漏请求，因此我们可以基于(3.2)获得等式(3.4)用于基于等式的多租户情况。

$$t_s \times B_{SD} = \sum_1^n R_{miss_i} \times Th_i^A \times t_f \quad (3.4)$$

通过求解 N 元方程，我们可以计算每个租户的分配吞吐量，即 Th_i^A ，如公式3.5所示。

$$\frac{1}{Th_i^A} = (1 - op)^{-1} \left(\frac{1}{Th_i^M} - \frac{\sum_1^n R_{miss_i} Th_i^M}{B_{SD}} \right) \quad (3.5)$$

等式(3.5)反映了测量的吞吐量，分配的限速值，缓存未命中率以及 FD 和 SD 的重叠率之间的关系。根据此等式，当分配的限速值 Th_i^A 升高时，测得的吞吐量也会增加，这与实际情况相符。如果错过的请求总数 $\sum_1^n R_{miss_i} Th_i^M$ 增加，则(3.5)建议增加 Th_i^A 以允许更多请求进入，这也是有道理的。当 op 上升时， Th_i^A 应该较小，因为我们应该在设备繁忙时减轻压力。

3.2.3 算法设计与实现

3.2.3.1 算法设计

SAAT 算法的过程如 Alg 1 所示，其中初始分配的 Th^A 等于目标 Th^T 。当请求到达时，我们启动守护程序来监视每个租户的吞吐量，并收集包括未命中率和当前时段的已测量吞吐量的信息。然后，我们使用函数 f_1 计算 op ，并使用函数 f_2 预测下一个分配的吞吐量 Th_{next}^A 。 f_1 和 f_2 可以从公式(3.5)得出。

改序号程序开始，首先 1) 设定周期 T，2) 收集时间 T 内的统计信息，具体收集信息内容在 d) 部分详细介绍。[4] 根据公式和收集的统计信息计算下一周期的限速值，具体公式在 e) 部分进行详细介绍。最后，控制修改 RIT 配置文件，使最新的限速值生效。

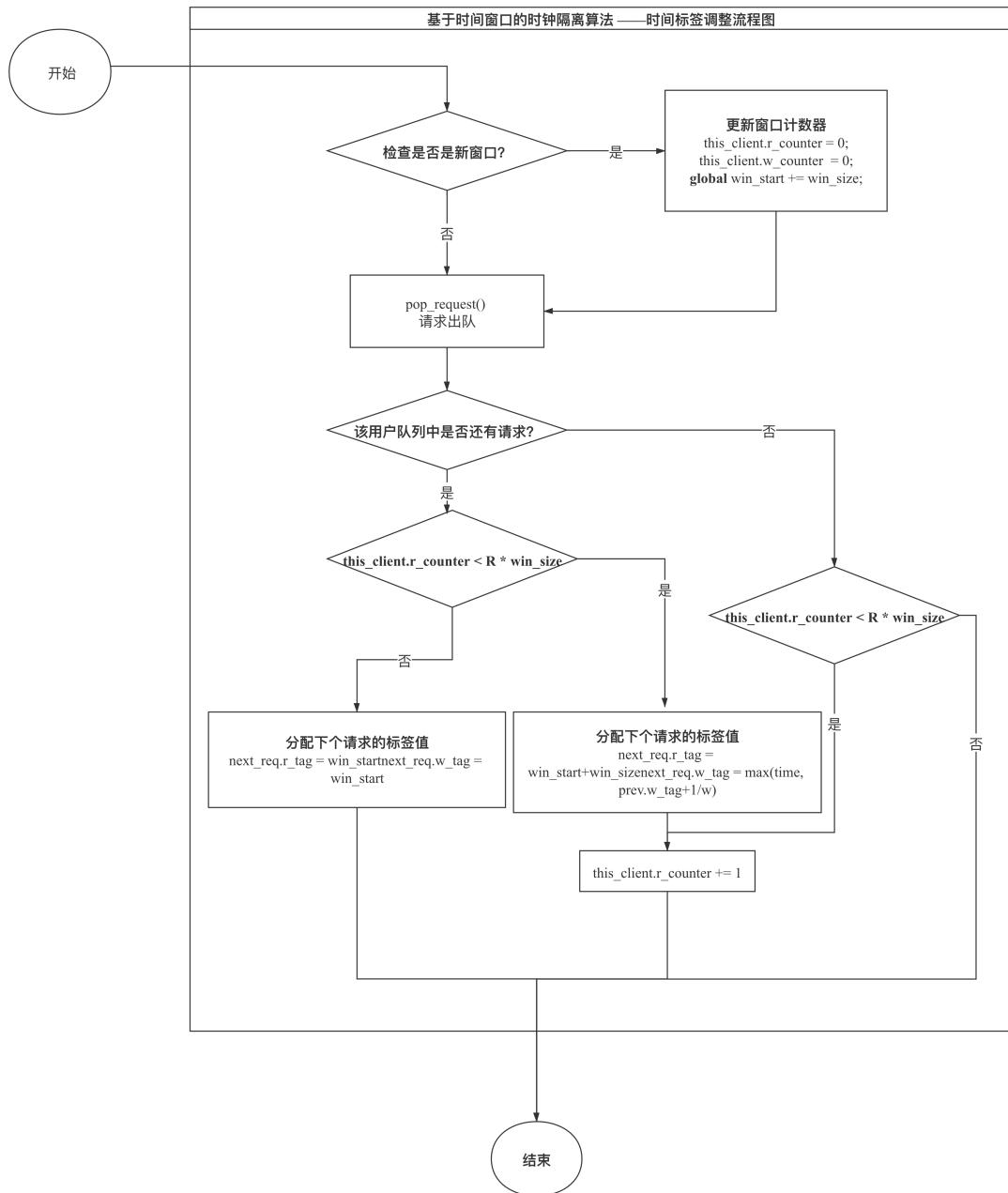


图 3.5 算法流程图

Algorithm 1 SAAT Algorithm

```

 $Th^A \leftarrow Th^T$ 
while working do
     $(R_{miss}, Th_{cur}^A) \leftarrow CollectStatistics(Period)$ 
     $op \leftarrow f_1(Th^M, Th_{cur}^A, R_{miss}, B_{SD})$ 
     $Th_{next}^A \leftarrow f_2(op, Th^T, R_{miss}, B_{SD})$ 
end while

```

3.2.3.2 算法实现

算法I用 Python 语言实现。1. 信息收集模块。2. 监控计算模块。3. 调整限速值模块。

3.3 实验设计与算法评估

3.3.1 实验配置

在本节中，我们将评估 SAAT 和原始版本 *Cgroup* 在混合存储设备上性能隔离的准确性。实验是在装有 8 核 Intel i7-6700 CPU @ 3.4GHz 和 16GB DRAM，运行 64 位 Linux 内核 v4.4.62 的 PC 上进行的。我们首先通过 Flashcache [33] 构建了一个混合设备，其中包括一个 100 MB 的永久内存作为快速设备 (FD) 和一个 250GB 的固态驱动器作为慢速设备 (SD)。我们修改了 Flashcache 的统计信息模块，以收集每个进程的未命中率。部署了 *Cgroup* 和 SAAT 来在混合存储设备上进行性能隔离。

我们采用两种指标来评估性能隔离的准确性。第一个度量是测得的一个 Th^M 与目标 Th^T 之间的平均吞吐量之差，即 DAT。第二个是平均抖动，即每个周期 P 中测得的吞吐量 Th_P^M 与目标吞吐量 Th_P^M 之间的差的平均值。第一个指标反映了很长一段时间内的总体准确性，而第二个指标则反映了所测得的吞吐量在目标周围浮动的稳定性。

重放了从典型企业数据中心收集的 Microsoft Research (MSR) 剑桥跟踪 [43]，以代表我们实验中的不同租户。如表3.1所示，我们的评估包括十种不同类型的应用程序。

表 3.1 实验中采用的工作负载

Trace	Application Type	Request Count	Read Ratio
<i>usr</i>	User home directories	12,873,274	72.14%
<i>prn</i>	Print server	17,635,766	19.79%
<i>hm</i>	Hardware monitoring	8,985,487	32.65%
<i>rsrch</i>	Research projects	3,254,278	11.22%
<i>src</i>	Source control	14,024,860	16.78%
<i>stg</i>	Web staging	6,098,667	31.76%
<i>ts</i>	Terminal server	4,216,457	25.89%
<i>web</i>	Web/SQL server	9,642,398	53.59%
<i>mds</i>	Media server	2,916,662	29.61%
<i>wdev</i>	Test web server	2,654,824	27.3%

3.3.2 整体效果

为了比较不同目标吞吐量设置下的 *DAT* 和 *Average Jitter*，我们使用它们与目标吞吐量值的比率。例如，如果我们将目标吞吐量设置为 200 MB / s，而测量的差异为 10 MB / s，则 *DAT* 或 *Average Jitter* 被视为 5 %。

我们执行了 14 组实验，涵盖了各种跟踪组合，不同的目标吞吐量值和不同数量的租户的情况。每组运行五次，平均结果被我们的实验采用。在这 14 组中，SAAT 平均可以将 *DAT* 从 6.56% 降低到 1.94 % (即 3.38 倍)，并将 *Average Jitter* 从 6.62% 降低到 5.25 %。

在单租户情况下，我们对五条迹线采用了目标吞吐量设置，范围从 150 MB / s 到 220 MB / s，并计算了五条迹线的平均结果。图??显示，SAAT 算法可以将 *DAT* 从 5.87% 降低到 1.7%，吞吐量减少了 3.39 倍。此外，原始 Cgroup 案例的 *DAT* 会随着目标吞吐量的增加 (即，从 2.96% 从 8.05%) 继续增加。相反，在这些情况下，SAAT 可以维持稳定且较小的 *DAT* (即 0.98% ~ 2.13%)。对于平均抖动，SAAT 比原始 Cgroup 方法高出 1.26 倍 (即 5.87 % 与 4.63 %)，如图??所示。请注意，平均抖动的减少低于 *DAT*，因为前者在每个周期都需要精确的吞吐量调节，比后者要难得多。平均抖动的改善更加困难，因为该算法必须在每个周期内给出准确的估算值。在多租户的情况下，我们将两个租户设置为相同

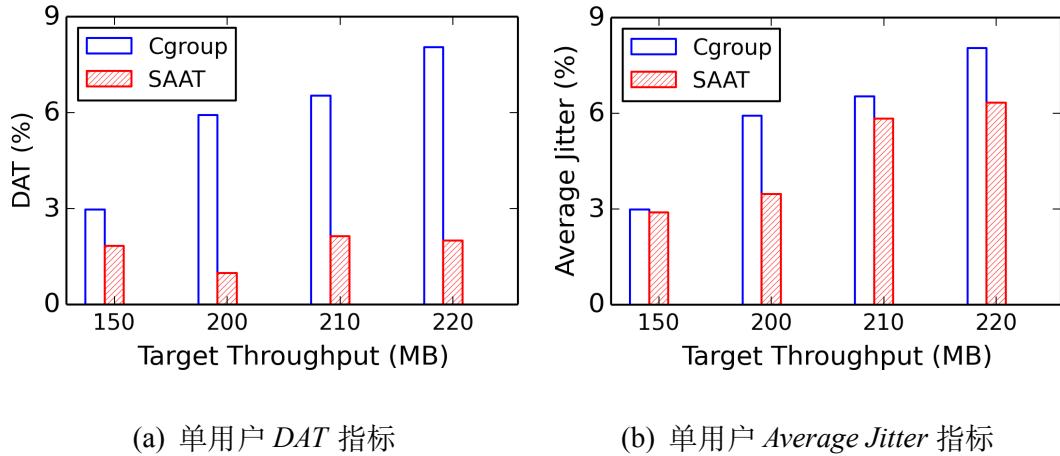


图 3.6 单用户：平均吞吐量差值 DAT 和

的目标吞吐量值，范围从 150 MB / s 到 250 MB / s。对于每个测试，我们计算了两个租户的平均 DAT 或 Average Jitter。结果表明，在所有吞吐量设置中，面对多个租户时，SAAT 始终优于 Cgroup。如图??和图??所示，SAAT 将 DAT 从 10.61% 降低到 3.85%，并将平均抖动从 10.62% 降低到 9.48%。

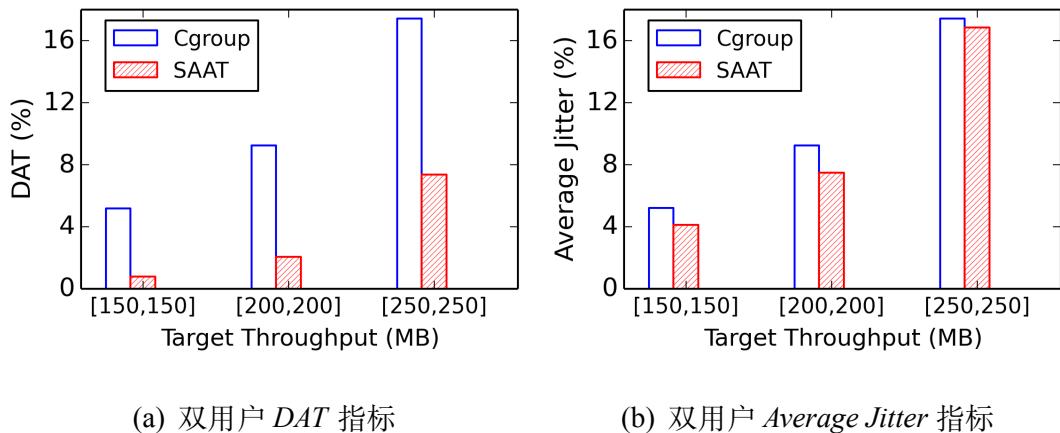


图 3.7 双用户：平均吞吐量差值 DAT 和平均波动值 Average Jitter

3.3.3 不同 Trace 的影响

为了评估各种工作负载下的 SAAT，当部署两个租户时，我们使用四种不同的组合和八条迹线进行了四组实验。

在所有情况下，我们将目标吞吐量设置为 200 MB / s。如图??所示，SAAT 在 DAT 的每组 Trace 组合中都优于 Cgroup。对于平均抖动，图??显示 SAAT 平均比 Cgroup 好 1.24 倍。

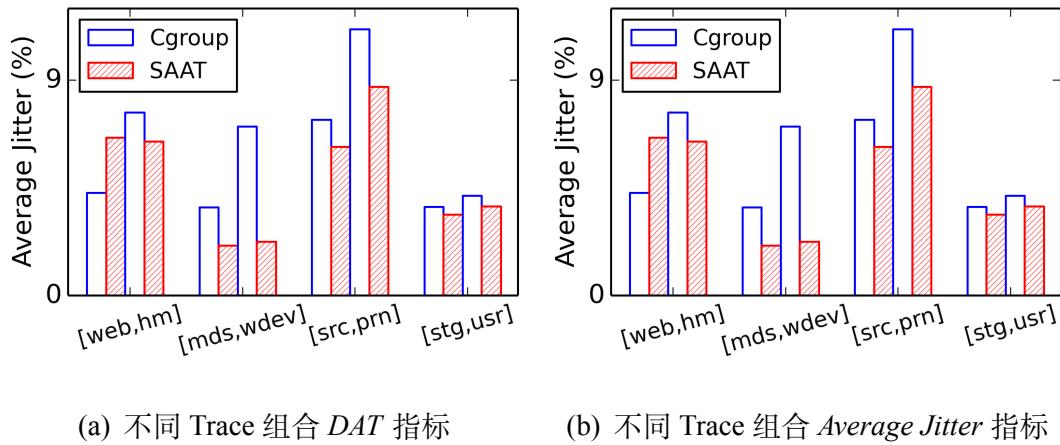


图 3.8 不同 *Trace* 组合：平均吞吐量差值 DAT 和平均波动值 Average Jitter

3.3.4 扩展性评估

当租户数分别为 2、3 或 4 时，我们测量了 SAAT 的性能。% 目标吞吐量的设置根据经验值而变化。结果表明，随着租户数量的增加，SAAT 始终实现的吞吐量变化要小得多。SAAT 提高了每个租户的 DAT，如图??所示，与 Cgroup 相比平均提高了 3.78 倍。当四个租户共享同一混合设备时，SAAT 可以将 DAT 从 3.91% 降低到 0.452%，是改进的 8 倍。图??表示平均抖动从平均 4.245% 降低到 2.293%。

3.4 本章小结

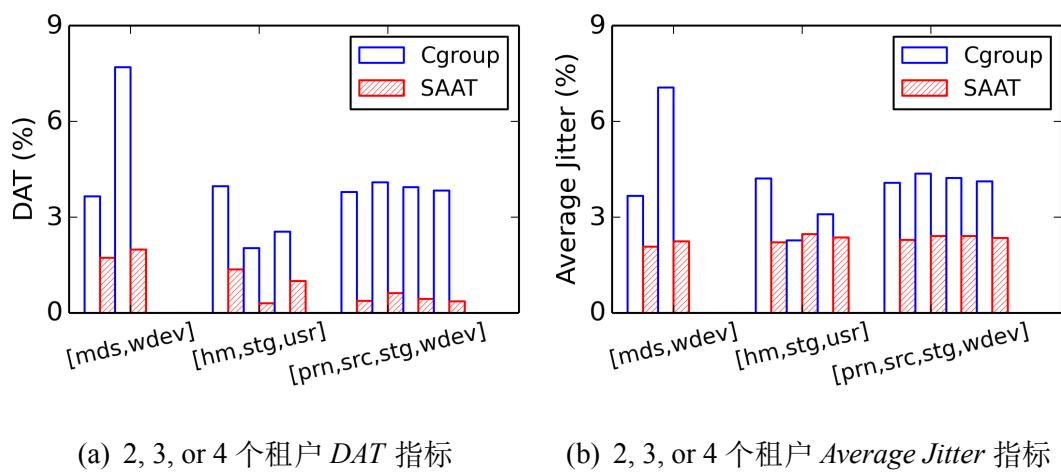


图 3.9 多用户: 平均吞吐量差值 DAT 和平均波动值 Average Jitter

第4章 分布式架构下的性能隔离——以 Ceph 为例

Ceph [6] 是目前广受欢迎的开源分布式存储平台，被广泛应用于虚拟化、公有云/私用云、容器化、高性能计算、大数据和视频/CDN 等。[7] 从 2010 年发布第一个版本 Argonaut 起，至今已经发布了 15 个版本，逐渐成为一款成熟的分布式存储技术。其特别的架构设计给使用者提供了无与伦比的可扩展性，为海量数据（EB 级别）的存储、访问提供了解决方案。根据 Ceph 社区用户数据调查 [8]，超过八成用户表示使用 Ceph 的经历令人极其满意。可以预测的是，在未来使用 Ceph 作为分布式存储方案的群体会逐年增加。作为典型的分布式存储技术，Ceph 平台非常重视用户的服务质量保障，提供秒级的吞吐量保障。本章选择 Ceph 的性能隔离技术进行研究，具有代表性。

本章的组织结构如下。章节4.1分为两个章节：子章节4.1.1介绍了 Ceph 的整体架构，并着重介绍了 I/O 调度层设计；子章节4.1.2详细介绍了 Ceph 目前使用的 I/O 性能隔离技术 dmClock 作用原理，并设计实验展示了其效果，分析了其无法实现质量保障目标的原因。章节4.2展示了 WinRWClock 算法设计与实现。章节4.3设计实验验证了 WinRWClock 算法的效果，并与 dmClock 算法进行了详细对比。

4.1 问题分析

4.1.1 Ceph 分布式系统架构介绍

1) 基本组件

Ceph 的整体架构和基本组件已经在图4.1中简要地展示出来了。Ceph 架构的基础是 RADOS 系统 [49]。RADOS 系统提供了构建非常好的可扩展性的分布式存

储集群的方法，RADOS 的节点可以进行自主地管理，很好地应对节点故障等情况，上层不需要考虑管理底层节点等技术细节。RADOS 管理的存储节点中，主要运行两种进程：OSD 守护进程和 Monitor 监护进程。

OSD 进程非常强大，可以直接与用户进行连接，处理用户的数据访问请求。此外，OSD 进程也会定期向 Monitor 监护进程汇报自己和邻居 OSD 节点的状态，让 Monitor 监护进程及时更新 OSD 的状态，在故障出现的情况下对系统进行调整。RADOS 为 Ceph 上层的访问提供了坚实的下层基础。在存储节点上，还有两个重要的数据存储单位，pool 和 PG (placement group)。

Pool 是在逻辑上对存储的对象进行分区的单位，在一个 pool 内的存储对象具有相同的数据安全级别，包括数据冗余类型和副本分布策略。一个用户的存储对象可能因为安全级别不同需要被分布到不同的 pool 上，重要的数据需要更多的备份和更安全的纠删码，没有那么重要的数据则可以减少备份和纠删码级别来获得更好的性能或者更便宜的价格。

PG 是 Ceph 上存储对象放置的最小单位，在一个 PG 内的对象都会被放置到相同的硬盘上。如果对象是三备份的，那么一个 PG 内的不同对象的三个备份会被放置在相同的三个硬盘上，主备份和副备份的设置也完全相同。不同用户的存储对象不可能属于同一个 PG。Ceph 中对数据放置进行调整，让不同节点达到均衡的最小单位是 PG。如果因为节点故障或者其他问题，需要对数据进行恢复，数据恢复的最小单位也是 PG。

在 RADOS 的基础上，Ceph 系统为用户提供了四种不同的访问方式：直接访问、对象存储、块设备和文件系统。

直接访问是通过 librados 的库进行访问。应用可以调用 librados 库中的函数和接口对下层 RADOS 节点进行访问。Librados 库支持 C, C++, Java, Python, Ruby 和 PHP 等多种语言，多种语言的支持让开发者可以自由地选择适合自己应用的语言对 Ceph 系统进行访问。和其他访问模式相比，直接访问的方式更加自由，开发者可以根据自己的数据和访问模式特点定制自己的存储和访问方式，从而提高存储系统的性能。

除了让应用直接调用，Librados 库也会被对象存储和块设备系统调用，为它们提供底层支持。对象存储被越来越多地应用于云环境。Ceph 系统提供的对象存储接口是 radosgw，用户与 radosgw 之间通过 FastCGI 接口进行访问。CGI (Common Gateway Interface) 可以被翻译为通用网关接口，为 HTTP 服务器和

其他机器上的服务提供通讯服务。传统的 CGI 接口性能不足，无法满足日益增长的云服务的通讯需求，因此 Ceph 系统用了更高性能的 FastCGI。FastCGI 中，动态语言的解析不需要在 HTTP 服务器上运行，从而大幅提高了服务器的性能。Radosgw 支持 Amazon S3[3] 和 Openstack Swift[22] 两种访问的权限鉴定机制，并且程序使用两种机制兼容的 API 访问的情况下，可以实现两种权限鉴定机制的兼容，用一种机制更新的数据，可以用另一种机制访问。

块设备存储对用户来说，提供的是虚拟的存储磁盘。因为块设备访问简单，支持快照技术，比较常用于虚拟机和 host 机。Ceph 在 Linux 内核中实现了 rbd 模块，用于挂载、访问和驱动虚拟块设备。Rbd 模块驱动的块设备既可以被虚拟机直接访问，也可以被用户态的用户通过 librbd 进行访问。QEMU 是一个可以让用户定制自己的虚拟机，提供更好的性能或者模拟用户需要的硬件和环境的模拟器。用户如果使用的是基于 QEMU 模拟的虚拟机，用户会在用户态访问块设备。所以 rbd 模块通过 librbd 对使用 QEMU 的用户进行块设备访问支持。

用户同样可以通过熟悉的文件系统的方式访问 Ceph。Ceph 提供的文件系统是与 POSIX 兼容的。Ceph 提供的文件系统是 CephFS，是与 POSIX 兼容的。CephFS 的用户通过专门的库 libcephfs 访问下层，Libcephfs 实际上是通过 librados 库对 rados 数据存储节点进行访问。CephFS 提升性能的核心是将元数据专门分离到单独的节点，进行单独的访问和管理。用户有以 ls, cd 为代表的很多操作只会访问和更改元数据，不会影响到存储数据。专门隔离的元数据节点 MDS 处理这些操作，就大大降低了 OSD 的开销，提升了整体性能。

2) Ceph I/O 调度

根据前面对于 Ceph 的四种访问模式的说明，可以看出，所有模式下的访问都会调用 librados 库访问 rados 层，在 rados 层上不同的访问模式是统一的。图 4.2 以文件系统为例展示了在 rados 层上 Ceph 内部的 IO 路径。

用户和 OSD 之间的连接是 Monitor 监控进程通过 cephx 认证机制进行用户身份鉴定之后建立的。Cephx 的认证机制原理是先让 Monitor 进程和用户通过安全的网络连接共享用户密钥，这个密钥是独属于每个用户用来识别该用户身份的。当用户想要访问 Ceph 系统上的数据时，用户需要和 OSD 建立一次连接，这次连接被称为会话。在这次会话中，Monitor 会为其生成一个会话密钥，并且用用户密钥加密会话密钥，将加密后的信息发送给用户。用户收到信息后用自己的密钥解密信息，就获得会话密钥，并用此密钥对之后发布的所有信息进行某

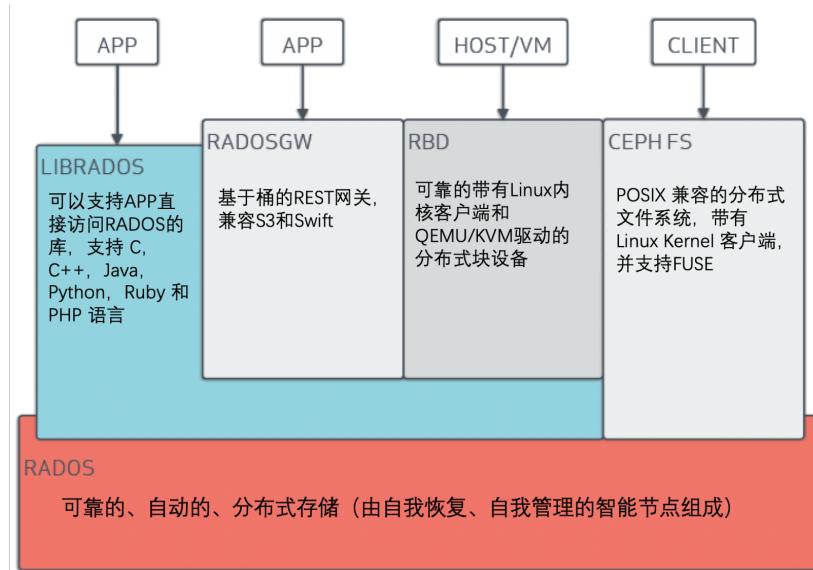


图 4.1 Ceph 架构图 (汉化自 [7])

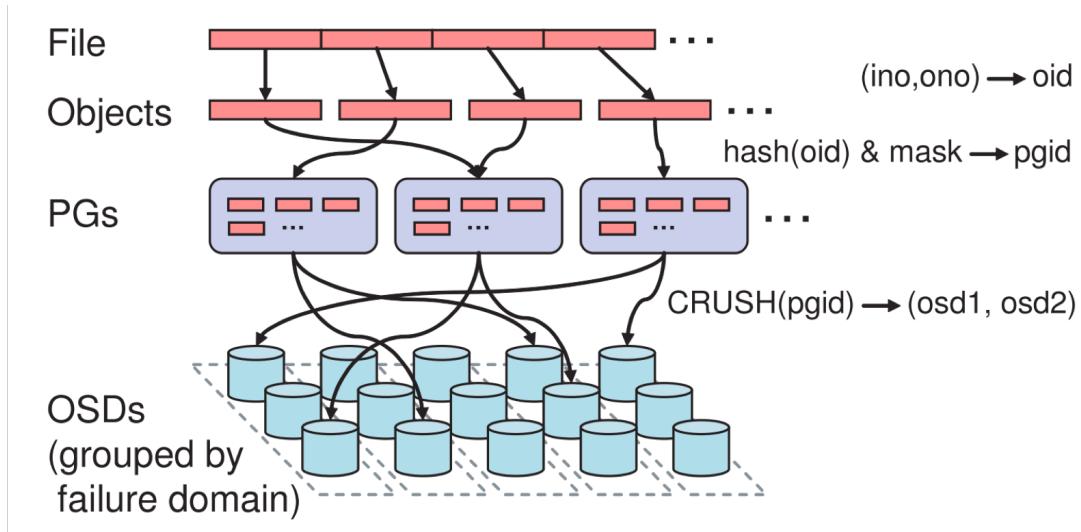


图 4.2 Ceph I/O 算法图 [49]

种加密或者验证操作，证明自己的身份。Monitor 和 OSD 等进程和节点在需要校验的部分判断用户是否持有正确的会话密钥，如果用户的会话密钥正确，说明用户同样拥有正确的用户密钥。黑客无法通过监听用户与服务器的信息传播内容，破解会话密钥和用户密钥，保证了用户身份认证的有效性。

在确认了用户身份之后，Ceph 系统需要将上层用户要访问的文件位置转化为存储在内部 rados 层上的 OSD 中的某个硬盘上的某个位置的数据。这个转化过程如下。用户的每个文件都有对应的 ino(inode id)，识别每个文件。Ceph 的文件系统会通过某些映射机制将文件内容切分成对象，并映射到对象存储空间中。文件中的某个内容通过文件识别码 ino 和对象顺序 ono(object number)，经过某种函数计算，就得到了对象存储空间中统一的对象编码 oid(object id)。对象存储空间中的对象基于 oid 可以计算出 PGid，也就是把这个对象放置到确定的唯一一个 PG 上。被划分到同一个 PG 中的对象，被存储到相同的硬盘上。

随后，PGid 基于 CRUSH 算法，可以计算出实际被存储到哪些 osd 上。CRUSH 算法是 Ceph 机制的核心。因为上述所有过程的算法和需要的数据结构都是公开的，所以用户端可以基于自己要访问的文件位置直接计算出自己需要连接哪些 OSD，直接将请求发送给对应的 OSD 进行处理。这种去中心化的机制符合云计算的模式，大大提升了 Ceph 分布式存储系统的性能。需要注意的是，用户的数据，即使是同一个文件，也可能会被映射到不同的 PG 和 OSD 上，所以用户可能需要与多个 OSD 建立连接。

当用户的请求被定位到 OSD 之后，相应的 OSD 就会接收到用户发送的数据访问请求。一个 OSD 可能会接收到很多个用户发送的请求。为了让不同用户之间不会相互干扰，让 OSD 更好地满足用户请求，OSD 内部会用调度队列对这些用户的请求进行调度。Ceph 支持 prio(PrioritizedQueue)、wpq(WeightedPriorityQueue)、dmClock 队列。Prio 队列是基于令牌桶机制的系统。当令牌数量足够的时候，令牌会先给高优先级的请求；否则，会先给低优先级的请求。wpq 队列是有权重的优先级队列，会将不同队列的优先级进行对比，防止某些队列优先级太低的时候无法被调度。Dmclock 队列的情况会在 4.2 中进行详细介绍。

4.1.2 Ceph 系统中的性能隔离机制

Ceph 中 I/O 质量保障机制借用 dmClock 算法实现。dmClock 根据 2010 年的论文 [36] 在 Ceph 中实现，2018 年 3 月发行了一个版本 [10]。该算法结合了多

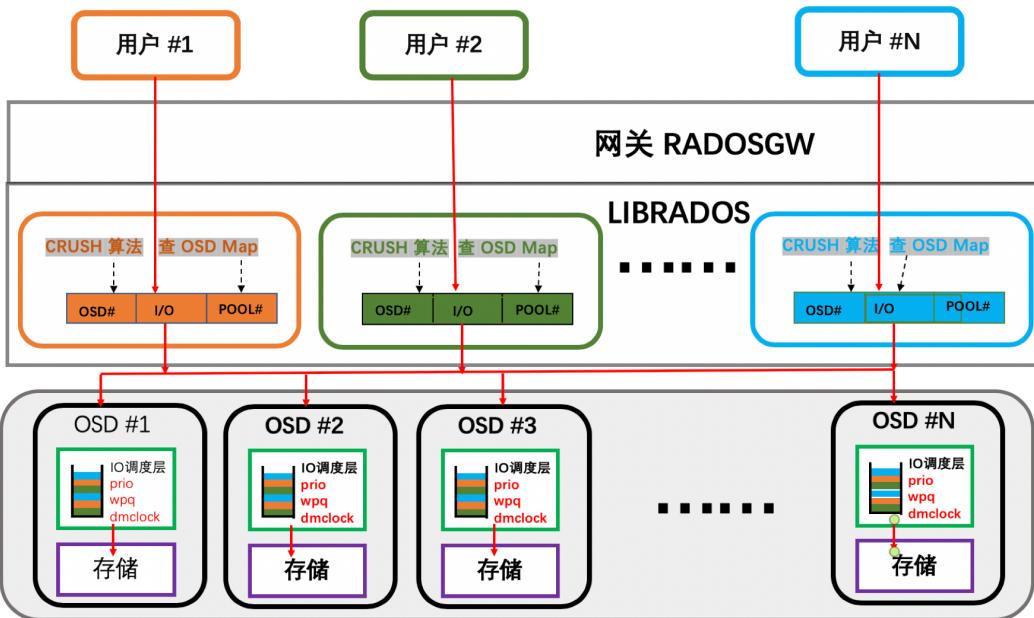


图 4.3 Ceph I/O 调度示意图

个实时时钟和动态时钟选择的原理在上述 IO 调度层作用，实现对不同用户 IO 分配的目标。dmClock 在根据用户不同的分类方法，有三种不同实现。根据 IO 类型分类的 mClockOpClass，把客户端前台请求与 Ceph 后台服务（如容错恢复、备份等）的请求分开调度；mClockClient 按照客户端不同的 IO 类型进行调度；mClockPool 按照访问不同池的 IO 进行调度。在接下来的部分中，会详细介绍 dmClock 算法的原理，应该实现的隔离目标，并实际搭建了 Ceph 系统，检验了 dmClock 的隔离效果。

4.1.2.1 分布式存储的隔离目标

共享在分布式存储中十分常见，而且不同用户类型的需求是多样的，因此云服务平台需要满足多样的服务目标。

假设现在有三种服务在共享 Ceph 平台资源，远程桌面服务、数据迁移服务和在线事务处理服务。很明显，这三种服务的工作负载对响应速度和吞吐量的需求不同。对于远程桌面服务，需要的带宽资源低、响应速度快；数据迁移服务需要占用大量带宽资源，但是不需要及时响应；在线事务处理既需要大量的

带宽资源，也需要低延迟。

设系统的吞吐量为 1000 IOPS，基于上述需求，我们可以给远程桌面服务和在线事务处理服务各预留 200 IOPs 资源，其余的 600 IOPS 系统资源按照 1:2:3 共享。对于数据迁移服务，为了防止其占用过多资源，我们设置 150 IOPS 的限制。这样的分配方法可以满足分享存储资源的工作负载的多种需求。三种服务按照比例得到的带宽资源分别是：远程桌面服务， $\frac{1}{1+2+3} \times 500$ IOPS，即 100 IOPS；数据迁移服务 200 IOPS，但是由于 150 IO；在线事务处理服务 300 IOPS。注意，数据迁移服务被设置了 140 IOPS 的限制，剩余的 60 IOPS 资源被远程桌面服务和在线事务处理服务按照 1:2 的服务分配。

综合三种分配目标，我们可以计算出每种服务在给定目标下的理想吞吐量：远程桌面服务得到预留资源和按比例分配的资源 320 IOPS，在线事务处理服务得到 440 IOPS，数据迁移服务由于限制，得到 150 IOPS 的带宽资源。

由上面的例子，我们可以使用如下目标来描述一个用户期望得到的质量保证：

- 1) **预留 (Reservation)** 系统保证用户一定能拿到的资源。
- 2) **权重 (Weight)** 用户之前按照比例分配的资源。
- 3) **限制 (Limit)** 用户可以得到的带宽资源上线。

假设系统能力是 C ，那么用户 i 应该得到目标吞吐量应为公式 4.1 所示。

$$Th_i^T = \max(R_i + \frac{W_i}{\sum_1^n W_k} \times C, L_i) \quad (4.1)$$

因此，对于给定的隔离目标，我们可以衡量用户实际得到的带宽资源是否符合以上公式计算得出的理论目标值。

4.1.2.2 dmClock 算法原理

基于时间标签的调度是许多基于公平分配的调度器的做法：所有的请求按照被分配好的标签进行调度。比如，算法按照 $\frac{1}{w_i}$ 的间距给依次用户 i 的所有请求打标签；如果请求按照标签值进行调度，那么次用户被服务的请求就按照 w_i 比例分配。同时，为了同步空闲用户和活跃用户的标签，算法需要维护一个全局虚拟时钟 **dmClock** 算法扩展了这个思想，采用多个标签控制来实现目标，同时也同步了空闲用户和活跃用户的时钟。如图 4.3 所示，dmClock 算法给每个用户被分配三个标签，预留标签 R ，按照比例分配的共享标签 W ，限制标签 L 。使

用不同的时钟来完成对这三个目标的控制，动态地选取这三个标签之一来完成预留阶段或权重阶段的调度。如图4.4所示，dmclock设计的调度器主要分为三个阶段，标签分配阶段；标签调整阶段；请求调度阶段。

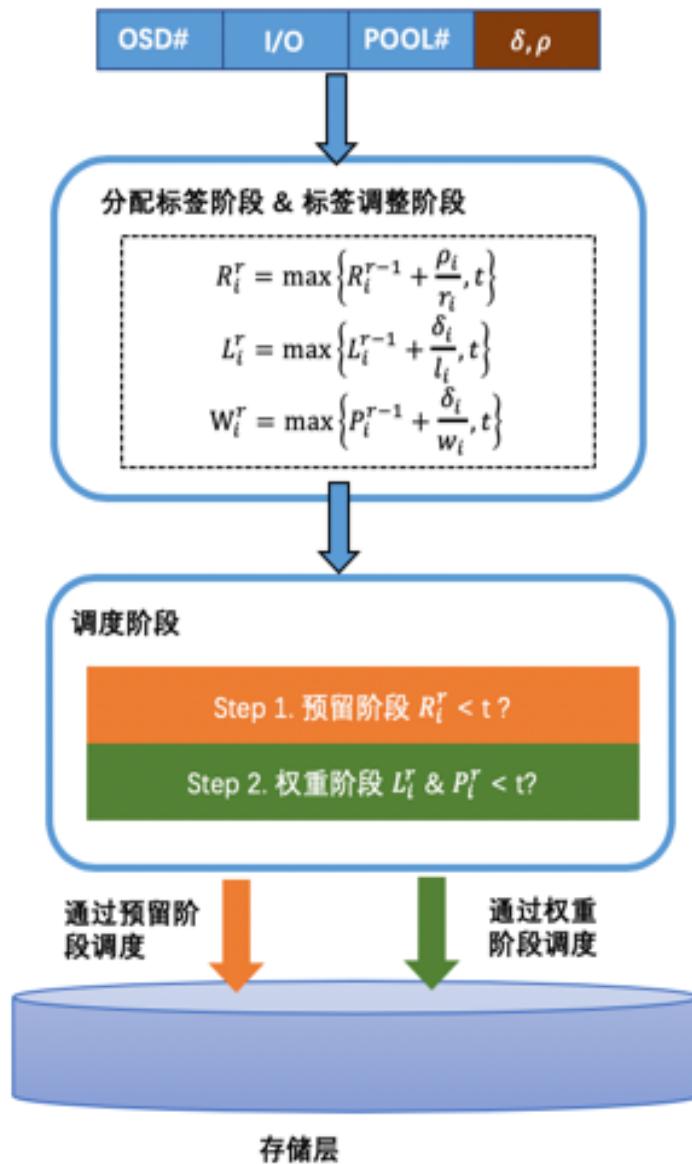


图 4.4 dmClock 算法流程图

标签分配阶段：这个过程会给来自在时间 t 到达的用户 i 的请求 r 分配 R , L 和 W 标签。所有的标签按照统一的原则进行分配，在此用分配 R 标签的方法展示。此请求的 R 标签取到达时间与该用户前一个请求的 R 标签 $+ \frac{1}{r_i}$ 如公式4.2所

示：

$$R_i^r = \max(R_i^{r-1} + \frac{1}{r_i}, CurrentClock) \quad (4.2)$$

由此公式可以得知标签的两个特性，第一，若用户的 R 标签以 $\frac{1}{r_i}$ 间隔分布，在时间 T 内可以被调度 $T \times r_i$ 个请求；第二，如果当前的时间比该用户 i 计算得出的 R 标签大，说明此用户已经很长时间没有活跃了，那么此用户的 R 标签应该等于当前时间。这样以，用户不会因为空闲得到好处。类似的，对于用于比例分配的 W 标签来说也是取当前时间与 $w_i(r - 1) + \frac{1}{w_i}$ 的最大值，紧跟着的请求被接着 $\frac{1}{w_i}$ 的间距分配标签值。用于限制的 L 标签取当前时间与 $l_i(r - 1) + \frac{1}{l_i}$ 的最大值。L 标签的间隔是 $\frac{1}{l_i}$ 。如果队列里排在首位的请求 L 标签小于当前时间的话，此用户被调度的请求还在限制值之下，因此这个请求可以被调度。如果 L 标签大于当前时间，说明用户调度过的请求已经达到了限制值，此请求不可被调度。

标签调整阶段：标签调整是校准给按照比例分配的标签的当前时间。如果一个客户闲置太久，它的权重标签会非常小。这时，直接进行调度的话，此闲置用户会比其他用户占优势。为了使每个用户公平地按照比例分配，算法应该调整所有用户的权重标签，使其站在同一起跑线。

请求调度阶段：标签分配、调整好之后，算法就可以按照标签进行请求调度。首先，检查是否有用户的 R 标签小于当前时间。如果有，持有最小 R 标签的请求会被调度。这个阶段被称为预留阶段。如果，所有用户请求的 R 标签都比当前时间大，那么进入基于权重标签的调度阶段。在这个阶段，调度器会将系统资源按照预定的比例分配。调度器会在满足限制条件的请求中，选取 W 标签最小的请求进行调度。满足限制条件的请求意味着其 L 标签小于当前时间。当用户 I 的请求在权重阶段被调度后，此用户 i 的剩余所有请求的 R 标签会减 $\frac{1}{r_i}$ 。R 标签递减的操作目标是避免在权重阶段调度请求的用户在预留阶段难以被调度。

按照如上三个阶段调度，dmClock 应该满足用户 i 如公式 4.1 描述的目标。

4.1.2.3 dmClock 效果分析

为了验证 dmClock 算法在 Ceph 系统中资源隔离是否有效，本文编译安装部署了 Ceph 系统，进行了多组实验。

1) 实验配置

实验是在装有 8 核的 Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz 和内存 377G 的服务器上进行的。操作系统版本是 CentOS 7.5.1804，内核版本是 4.14.146。本实验采用 Ceph 12.1.0 版本。我们使用 Dell Express Flash PM1725a 1.6TB AIC 型号的 NVMe 磁盘。工作负载采用 Ceph[6] 官方社区提供的 benchmark 工具 radosbench[23]。在 Ceph 的传统文件系统 (filestore) 创建 5 个池 (Pool)，每个池含有 32 个放置组 (PG)。该工具会根据配置产生不同粒度、规定时间长度的读写工作负载。IO 调度层的队列 op_queue 设置为 mclock_pool。其他 Ceph 系统配置详见见章节4.3。

Ceph 可以将访问不同池的请求识别出来，所以本实验把不同的池看作不同用户。针对这两组用户测试了两种具有代表性的 (R,W,L) 配置，如表格4.1所示。配置一给用户 1 配置了高额读的预留资源 (IOPS)，其余用户的预留资源很少，权重资源配置相同。此配置目的是为了观察算法是否能给用户分配足够多的预留资源。配置二给出了权重和预留资源均配置的情况，可以用来观察权重大的用户对预留资源多的用户的影响。

用户编号	配置一	配置二
1	(400, 1, 0)	(200, 1, 0)
2	(2, 1, 0)	(2, 3, 0)
3	(1, 1, 0)	(1, 5, 0)
4	(3, 1, 0)	(3, 2, 0)
5	(4, 1, 0)	(200, 1, 0)

表 4.1 dmClock 算法效果实验 (R,W,L) 配置信息

2) 衡量指标

如4.1.2.1小节所述，本节通过用户 i 实际测量到的平均吞吐量 Th^M_i 与按照公式4.1计算出的目标吞吐量 Th^T_i 的差值来衡量 dmClock 算法的实际隔离效果。

该差值记为：

$$DAT_i = Th_i^M - Th_i^T \quad (4.3)$$

3) 效果分析

图4.5和图4.6展示了两种配置下 dmClock 算法的表现。柱状图展示了用户实际得到的吞吐量 Th_i^M ，折线图标记了按照当前系统能力 $\sum_1^5 Th_i^M$ 计算出的目标吞吐量值 Th_i^T 。

图4.5中，系统的当前能力是 1067 IOPS，但是用户 1 只得到 395 IOPS，距离目标吞吐量少 136 IOPS。图4.6中，系统的当前能力是 1080 IOPS，用户 1 和用户 5 均没有得到应得的 I/O 资源，分别是 200 和 199，差距分别是 56 和 57，相应地，其他 3 个用户得到的资源比实际多。在两个配置中，只考察按比例分配的资源，即配置一中的用户 2~5 和配置二中的 2~4，发现用户实际得到的资源基本按照设置的比例分配。

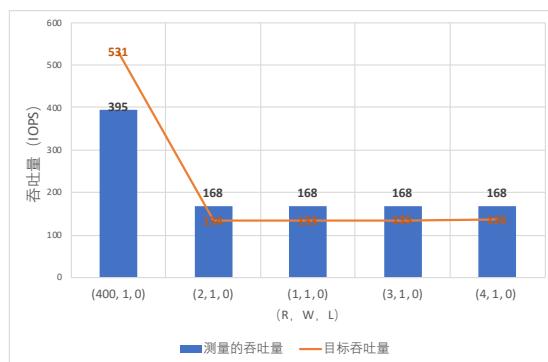


图 4.5 dmClock 算法的表现（配置一）

综合分析发现，dmClock 无法保证那些设定预留目标的用户需求。这是不合理的。

4) 原因分析

为了解释差距出现的原因，我们需要深入算法内部进行分析。回忆4.1.2.2小节介绍的 dmClock 算法原理。用户的请求要被调度需要满足以下条件，1) R, W, L 中某个标签小于当前时间。2) 该标签在所有用户请求中是最小的。为了保证先满足用户的预留目标，调度器严格按照先比较 R 标签，再比较 W, L 标签的顺序执行。这个严格的执行顺序仅仅保证了同一个用户，不同请求之间的顺序。两个请求会隔时间差 $\frac{1}{R_i}$ 进行调度。考虑在配置一的这样一个情况，用户 1 的请求 r 在时间 t_0 ，预留阶段被调度，下个请求 r+1 需要再等待 $\frac{1}{R_i}$ 的时间才

能被调度。在时间 t_1 ($t_1 < t_0 + \frac{1}{R_i}$), 由于用户 1 的 r+1 请求不符合调度条件, 调度器只能调度用户 1 ~ 4 中符合权重阶段的请求。尽管用户 1 有充足的高优先级请求等待被调度, 但是由于未达到调度时间, 其他用户低优先级的请求提前占用系统资源。用户之间权重阶段占用预留阶段资源的做法会导致设定大预留目标的用户很难拿到目标吞吐量。



图 4.6 dmClock 算法的表现 (配置二)

4.2 基于时间窗口的时钟隔离算法 WinRW Clock 算法设计与实现

基于章节4.1.2.1提出的隔离目标, 以及章节4.1.2.2对 dmClock 效果不好的原因分析, 本文设计了基于时间窗口和多时钟的算法。子章节4.2.1介绍了算法设计思想, 子章节4.2.2介绍了在 Ceph 系统里面如何实现的。

4.2.1 算法设计

1) 设计目标

WinRW Clock 算法在分布式存储平台上完成以下目标:

- 保证预留资源与权重资源不互相影响。
- 保证预留的资源能达到目标。
- 保证剩余的系统资源按照比例分配。
- 在毫秒级的时间尺度上, 完成上述目标的保证。

为了达到上述目标，WinRWClock 提出了新的多时钟标签分配方法，引入了时间窗口的机制帮助请求平稳调度。

2) 算法流程和伪代码实现

WinRWClock 算法中，请求仍然采用**两阶段调度法**思想，先判定预留 R 标签，再判定权重 W 标签；且一个请求是否被调度取决其所处阶段和该阶段所用时间标签与当前时间的大小。重要的问题在于，如何通过改变分配时间标签的方式，在满足目标的同时，使两阶段不受影响？

新算法取消了 dmClock 算法中请求之前时间标签的间隔 $(\frac{1}{r_i}, \frac{1}{w_i})$ ，改用时间窗口加统计的方法控制请求调度。给每个用户设置一个预留计数器，用来确保未达到预留目标的用户请求先行被调度。考虑在4.1.2.3 “原因分析” 中的例子，由于请求之间存在 $\frac{1}{r_i}$ 间隔，用户 1 的第 $r+1$ 个请求需要再等待 $\frac{1}{R_i}$ 的时间才能被调度。在等待过程中，其他用户，即使处于优先级低的权重阶段，也会被调度，导致**抢占**此用户的资源。如果消除用户 1 请求之间 $\frac{1}{r_i}$ 的时间间隔，保证请求调度不会被延迟，这样便避免了“抢占”的问题。

图4.7展示了给某个用户分配时间标签的流程。分配标签算法作用在系统决定选取哪个用户的请求出队的时候，即使用此次出队的请求计算等待在队列中的请求标签。因此，流程图中的“开始”代表系统进入出队函数。**第一步**，判定是否是处于新窗口。如果“是”，更新是用户的预留计数器；“否”则不更新计数器 $r_counter$ 和窗口起始时间 win_start 。**第二步**，出队一个请求。注意，进入出队函数之前，程序已经判定队列中至少有一个请求，所以在此无需处理。**第三步**，确认此用户队列中是否有其他请求，如果“是”，准备给队列中的下一个请求分配标签。公式4.4和4.5分别给出了决定用户 i 的第 $r+1$ 个请求的 R 标签和 W 标签的逻辑。每个窗口被服务的理论请求个数应该是 $win_size \times$ 预留目标值 $target_r$ 。 $r_counter_i$ 记录了本时间窗口在用户 i 预留阶段服务的请求个数，如果计数器超过理论值，说明该时间窗口内的请求已经被服务完毕，可以进入权重阶段。分配 R 标签时，如果处于预留阶段，其 R 标签等于窗口的起始时间 win_start ，这可以保证此请求无需等待，优先调度；否则，其 R 标签等于下个窗口的起始时间。分配 W 标签时，如果处于预留阶段，说明应按照 R 标签调度，理论上 W 标签可以取任何值，这里我们分配窗口起始值；否则，W 标签取

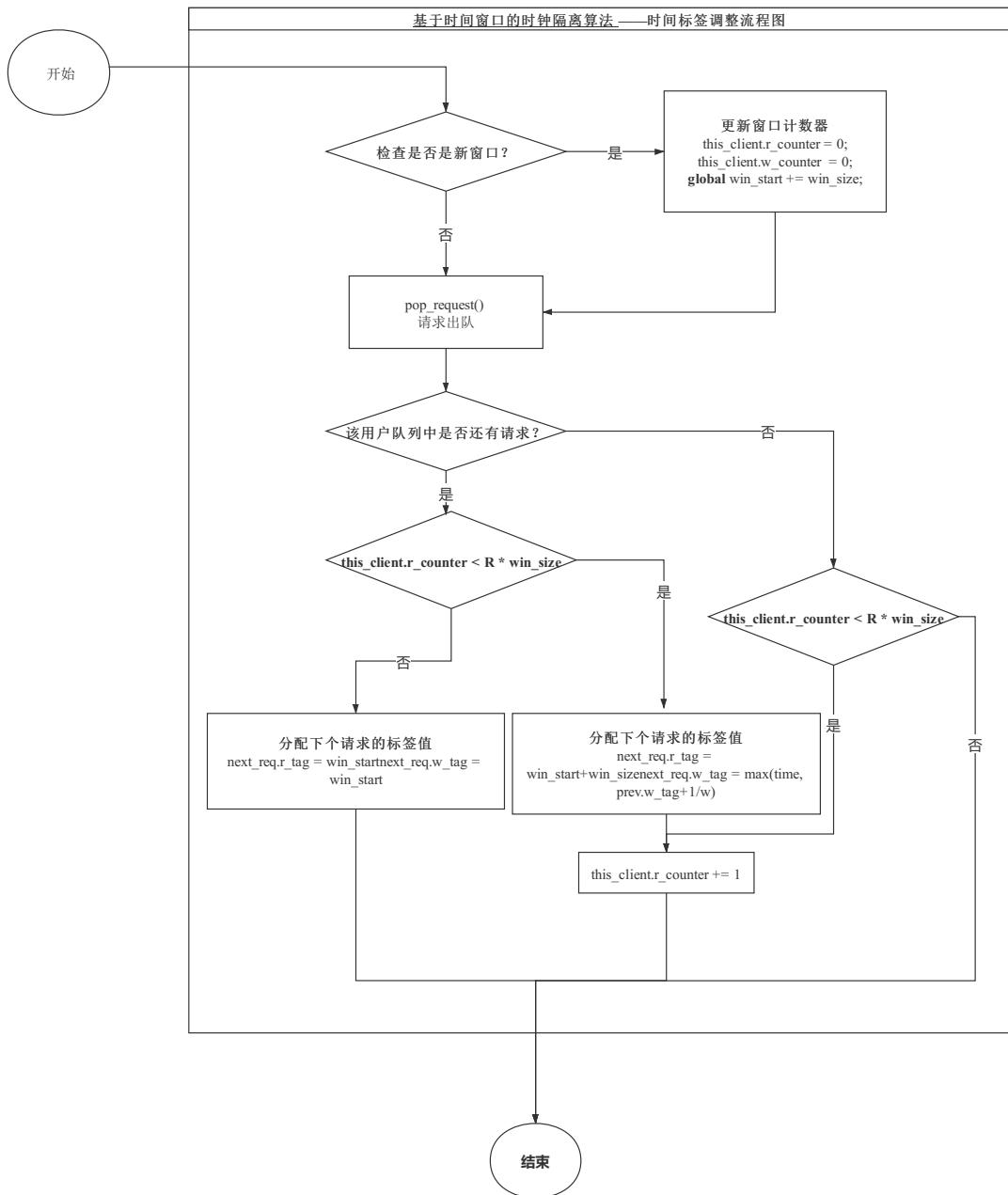


图 4.7 时间标签处理流程图

前一标签 $+ \frac{1}{w_i}$ 与当前时间的最大值。这样可以保证用户得到应得的权重资源。

$$R_i^{r+1} = \begin{cases} \text{win_start} & r_counter_i < target_r \times \text{win_size} \\ \text{win_start} + \text{win_size} & r_counter_i \geq Target_R \times \text{win_size} \end{cases} \quad (4.4)$$

$$W_i^{r+1} = \begin{cases} \text{win_start} & r_counter_i < target_r \times \text{win_size} \\ \max(\text{time}, W_i^r + 1/w_i) & r_counter_i \geq target_r \times \text{win_size} \end{cases} \quad (4.5)$$

分配完标签之后，在预留阶段，计数器增一。第三步中，如果队列中不含任何请求，则无需分配标签，但是计数器仍然要计入本次在预留阶段出队的请求。最后，此流程结束，请求进入下层存储层处理。

算法2展示了 WinRW Clock 的伪代码实现。**procedure Dequeue** 中的逻辑对应了流程图4.7。其中分配标签的逻辑由 **procedure AssignTag** 实现。此外，算法2还展示了流程图4.7没有涉及到的方面，即如何确定第一个请求的标签值。**procedure Enqueue** 入队函数处理了确定第一个请求标签的情况。请求入队时，首先会初始化第一个时间窗口，并且会给队列为空时入队的请求分配标签。

Algorithm 2 WinRW Clock

```

1: WinStart  $\leftarrow 0$ 
2: WinCounter  $\leftarrow 0$ 
3: WinSize  $\leftarrow 1$ 
4: procedure PopRequest( $c_i$ ) ▷ 从用户 i 的请求队列中取出一个请求
5: end procedure
6: procedure PushRequest( $r, c_i$ ) ▷ 向用户 i 的请求队列中添加请求
7: end procedure
8: procedure AssignTag( $request_r, time_t, client_c_i$ )
9:   if  $t - WinStart > WinSize$  then ▷ 新窗口
10:     $WinStart \leftarrow \max(WinSize + WinStart, t)$ 
11:     $c_i.r\_counter \leftarrow 0$ 
12:   else
13:     if  $c_i.r\_counter < c_i.reservation$  then ▷ 预留阶段
14:        $R_i^r \leftarrow WinStart$ 

```

```

15:       $P_i^r \leftarrow P_i^r - 1$ 
16:       $L_i^r \leftarrow \max(L_i^r - 1, t)$ 
17:       $c_i.r\_counter \leftarrow c_i.r\_counter + 1$ 
18:    else                                ▷ 权重阶段
19:       $R_i^r \leftarrow WinStart + WinSize$ 
20:       $P_i^r \leftarrow P_i^r - 1 + 1/w_i$ 
21:       $L_i^r \leftarrow \max(L_i^r - 1, t)$ 
22:    end if
23:  end if
24: end procedure
25: procedure Enqueue( $request_r, time_t, clientc_i$ )
26:  if  $c_i$  has no request then          ▷ 该用户队列为空
27:    if  $WinCounter == 0$  then            ▷ 第一个时间窗口
28:       $WinStart \leftarrow t$ 
29:       $WinCounter \leftarrow WinCounter + 1$ 
30:    end if
31:    AssignTag( $r, t, c_i$ )
32:  end if
33:  PushRequest( $r, c_i$ )
34: end procedure
35: function Dequeue( $time_t, clientc_i$ )
36:   $r \leftarrow PopRequest(c_i)$            ▷ 取出队首请求
37:  AssignTag( $r + 1, t, c_i$ )           ▷ 分配下一个请求的标签
38:  return  $r$ 
39: end function

```

3) 算法总结

根据前面的目标介绍和算法设计介绍，最后总结一下为什么 WinRWClock 的算法设计可以满足设计目标。目标 a 是保证预留资源与权重资源不互相影响。因为 WinRWClock 在每个窗口内都将预留资源那部分的标签值设为窗口内最小值，确保预留资源会先被执行，所以不会被权重资源影响。目标 b 是保证预留的资源能达到目标。因为预留资源先执行，并且每个窗口都按照用户设置参数

和窗口大小进行计算，所以可以达到目标。目标 c 是保证剩余的系统资源按照比例分配。剩余的资源是按照 dmClock 算法类似的标签值进行计算的，能达到按照比例执行不同用户请求的效果。并且只有系统把预留资源都服务之后才会走权重资源，所以可以将剩余的系统资源按照比例分配。目标 d 是在比毫秒级的时间尺度上，完成上述目标的保证。因为 WinRWClock 的时间窗口大小可以根据需要的是时间尺度自行设置，并且算法的运行时间很短，只要时间窗口不降低到毫秒级以下，就不会影响到前面三个目标的正确性。基于上述理论分析，我们在 Ceph 内部实现了 WinRWClock，实现细节在章节^{4.2.2}中介绍。

4.2.2 系统实现

WinRWClock 算法用 C++ 语言在 Ceph 12.0.1 版本中实现，本章节详细展示了 WinRWClock 实现中用的数据结构和过程。前面图^{4.3}介绍过，在 Ceph 系统中，用户可以与 OSD 通过 libRados 直接通信，所以质量保证算法在每个 OSD 上的 IO 调度层实现。OSD 通过一个叫 ShardedOpWQ 的工作队列对所有请求进行管理。这是一个虚拟复合队列，通过继承机制，实现了不同的子队列类型，dmClock 就是其中的一个队列类型。I/O 请求从 ShardedOpWQ 出队后，会通过 ObjectStore 的接口给下层存储设备处理。实现 WinRWClock 算法本质上是实现 ShardedOpWQ 的工作队列成员用的一个实例。在配置过程中，只要把 op_queue 选项设置成 WinRWClock 队列即可。

1) 时间标签处理。

图^{4.8}展示了记录请求标签的类 RequestTag，其类成员 arrival，reservation，proportion，reservation 分别记录了请求到达时间，ready 布尔变量标志一个请求是否在限制值以内，可以被调度。预留 R 标签，权重 W 标签和限制 L 标签。第二和第三个构造函数，通过重载机制分别实现在预留阶段和权重阶段分配标签的算法。

2) 用户请求队列管理

a. 用户类实现

图^{4.10}所示的用户类 ClientRec 包含了一个用户所需的所有信息：请求队列 requests，预留计数器 r_tag_counter，权重计数器 w_tag_counter，分布式参数 delta，pho，上一个调度的请求标签值 prev_tag，质量保证目标 info。其成员函数包括处理请求队列 requests 的函数，如请求入队函数 add_request()，出队函数

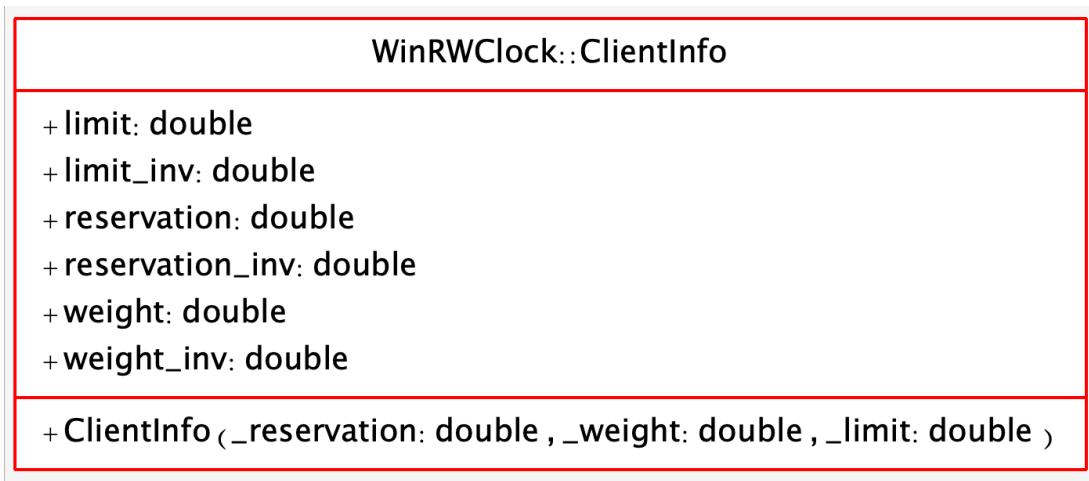


图 4.8 RequestTag 类

pop_request() 等。请求队列中的元素是含有成员 RequestTag 的类。

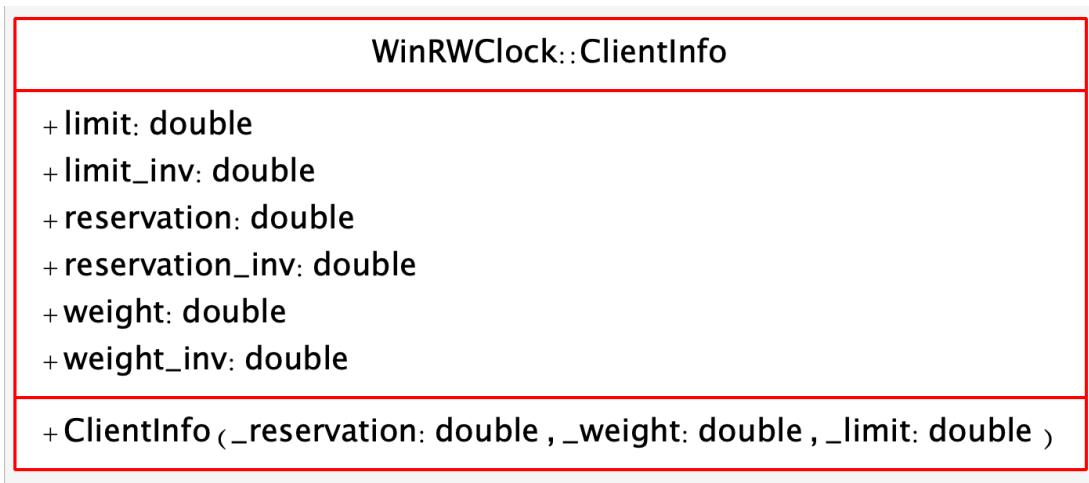


图 4.9 ClientInfo 类

b. 快速获取含有最小 R 标签或者 W 标签的用户

WinRWClock 算法需要快速找到持有最小标签的用户请求进行调度，采用最小堆（完全二叉树结构）进行管理。二叉树的节点是如图4.10所示的用户类 ClientRec。ClientRec 中的类成员 requests 是一个先进先出队列（std::deque），包含此用户的所有请求。完全二叉树的比较元素是 ClientRec 实例中 requests 队列里的第一个请求的标签值。但是，算法要求快速得到用户中 R 标签或者 W 标

```
WinRWClock::PriorityQueueBase::ClientRec

+ cur_delta: uint32_t
+ cur_rho: uint32_t
+ idle: bool
+ info: const ClientInfo *
+ last_tick: Counter
+ r_tag_counter: uint32_t
+ w_tag_counter: uint32_t
- client: C
- prev_tag: RequestTag
- prop_delta: double=0.0
- requests: std::deque

+ assign_unpinned_tag_(lhs: double &, rhs: const double )void
+ ClientRec(_client: C, _info: const ClientInfo *, current_tick: Counter )
+ add_request_(tag: const RequestTag &, client_id: const C &, RequestRef)void
+ next_request() ClientReq &
+ pop_request()void
+ remove_by_req_filter_(filter_accum: std::function , visit_backwards: bool , bool
+ remove_by_req_filter_bw(filter_accum: std::function , bool
+ remove_by_req_filter_fw(filter_accum: std::function )bool
+ update_req_tag_(_prev: const RequestTag &, _tick: const Counter &)void
+ get_req_tag() const RequestTag &
+ has_request()bool
+ next_request() const ClientReq &
+ request_count() size_t
- lim_heap_data() c::IndIntruHeapData
- ready_heap_data() c::IndIntruHeapData
- reserv_heap_data() c::IndIntruHeapData
```

图 4.10 ClientRec 类

签的最小值，因此需要用两个完全二叉树结构进行管理，分别按照 R 标签和 W 标签排序。两个二叉树中节点中的 ClientRec 是共享的，用 `std::make_shared` 实现。这样保证通过 R 标签调度走的请求可以在 W 标签二叉树中被删除，反之亦然。完全二叉树用变长数字存储，由于是比较基础的数据结构，实现过程在此不赘述。

c. WinRWClock 类封装

WinRWClock 队列作为 Ceph 工作队列类 ShardedOpWQ 构造函数的参数传入。WinRWClock 必须提供自己的 `enqueue` 和 `dequeue` 函数的实现。这个类用 **PriorityQueueBase** 实现，如图4.11 所示。此类成员中包括伪代码4.2 中定义的全局时间窗口起始变量 `win_start`，窗口大小 `win_size` 和窗口计数器 `win_count`。此外，`ready_heap`, `resv_heap`, `limit_heap` 分别实现了 b 中提到的最小堆实例，通过对这三个对数据的操作可以获取当前持有最小标签的用户请求。`client_map` 实现了 `client id` 到 `client` 实例的映射，便于检索。成员函数中的 `do_add_request` 和 `pop_process_request` 分别实现了添加请求 `enqueue` 功能和删除请求 `dequeue` 功能，分别对应算法4.2 中的 **procedure Enqueue** 和 **procedure Dequeue**。

现在以出队函数为例，介绍如何使用上述类的实例，实现算法中的逻辑。图4.12 展示了出队函数 `pop_process_request` 的序列图，展示了 4 个类 `PrioritQueueBase`（队列管理），`ClientRec`（用户），`RequestTag`（请求标签）和 `IndIntruHeap`（最小堆）的关系。白色柱形表示此类的实例的存在期，比如 `PrioritQueueBase` 实例一直存在，因为 `pop_process_request` 是此类的一个成员函数，函数调用过程中，此类的实例必然一直存在；而 `RequestTag`, `ClientRec` 的实例会出现结束的情况，这是由于在函数调用过程中，暂时使用过这些类的实例。因此，通过图4.12，我们可以清晰得到此函数使用这几个类的方式以及这几个类之前的关系。首先，函数会从指定的堆中取出堆首 `ClientRec` 实例，并调用 `next_request` 函数获得该用户 FIFO 队列中的队首元素。紫色区域 ALT 判断是否是新的窗口，如果是更新窗口的起始时间，并遍历堆中的所有用户，清零其计数器。`pop_requests` 操作从 `ClientRec` 实例中的请求队列出队一个请求。接着，`has_requests` 操作判断队列中是否有其他请求，红色区域 ALT 判断此请求出队后，是否有后续请求，如果是，则准备给下个请求打标签。`get_client_info` 函数获得当前用户的预留、权重、限制目标。紫色 ALT 区域说明在时间窗口内部，紧接着黄色 ALT 区域判断处于预留阶段还是权重阶段，分别按照算法逻辑更新时间标签，可以看到此

```
WinRWClock::PriorityQueueBase

# is_dynamic_cli_info_f: constexpr bool=U1
# allow_limit_break: bool
# anticipation_timeout: double
# check_time: Duration
# clean_mark_points: std::deque<Duration>
# cleaning_job: std::unique_ptr<std::function<void()>>
# client_info_f: ClientInfoFunc
# client_map: std::map<ClientID, ClientInfo>
# data_mtx: mutable std::mutex
# erase_age: Duration
# finishing: std::atomic_bool
# idle_age: Duration
# limit_break_sched_count: size_t
# limit_heap: c::IndIntruHeap<RequestRef>
# prop_sched_count: size_t
# ready_heap: c::IndIntruHeap<RequestRef>
# reserv_sched_count: size_t
# reserv_heap: c::IndIntruHeap<RequestRef>
# tick: Counter
# win_count: size_t
# win_size: Time=1.0
# win_start: Time=0.0

+request_sink(RequestRef, void)
+remove_by_client(client: const C &, reverse: bool, accum: std::function<void()>)
+remove_by_req_filter(filter_accum: std::function<void()>, visit_backwards: bool, bool)
+update_client_info(client_id: const C &)
+update_client_infos()
+client_count() size_t
+display_queues(out: std::ostream &, show_res: bool, show_lim: bool, show_ready: bool, show_prop: bool, void)
+empty() bool
+get_heap_branching_factor() uint
+request_count() size_t
# min_not_0_time(current: const Time &, possible: const Time &, const Time &)
# PriorityQueueBase<_client_info_f: ClientInfoFunc, _idle_age: std::chrono::duration<Time, std::ratio<1>>, _erase_age: std::chrono::duration<Time, std::ratio<1>>, _check_time: std::chrono::duration<Time, std::ratio<1>>, _delete_from_heap: ClientRecRef &, heap: c::IndIntruHeap &)
# delete_from_heap(client: ClientRecRef &, heap: c::IndIntruHeap &)
# pop_process_request(heap: IndIntruHeap &, process: std::function<void()>, now: Time, void)
# delete_from_heaps(client: ClientRecRef &)
# do_add_request(RequestRef, client_id: const C &, req_params: const ReqParams &, time: const Time, cost: const double, void)
# do_clean()
# do_next_request(now: Time, NextReq)
# reduce_reservation_tags(client: ClientRec &)
# reduce_reservation_tags(client_id: const C &)
# ~PriorityQueueBase()
# get_cli_info(client: ClientRec &)
const ClientInfo *
```

图 4.11 I/O 队列方法类

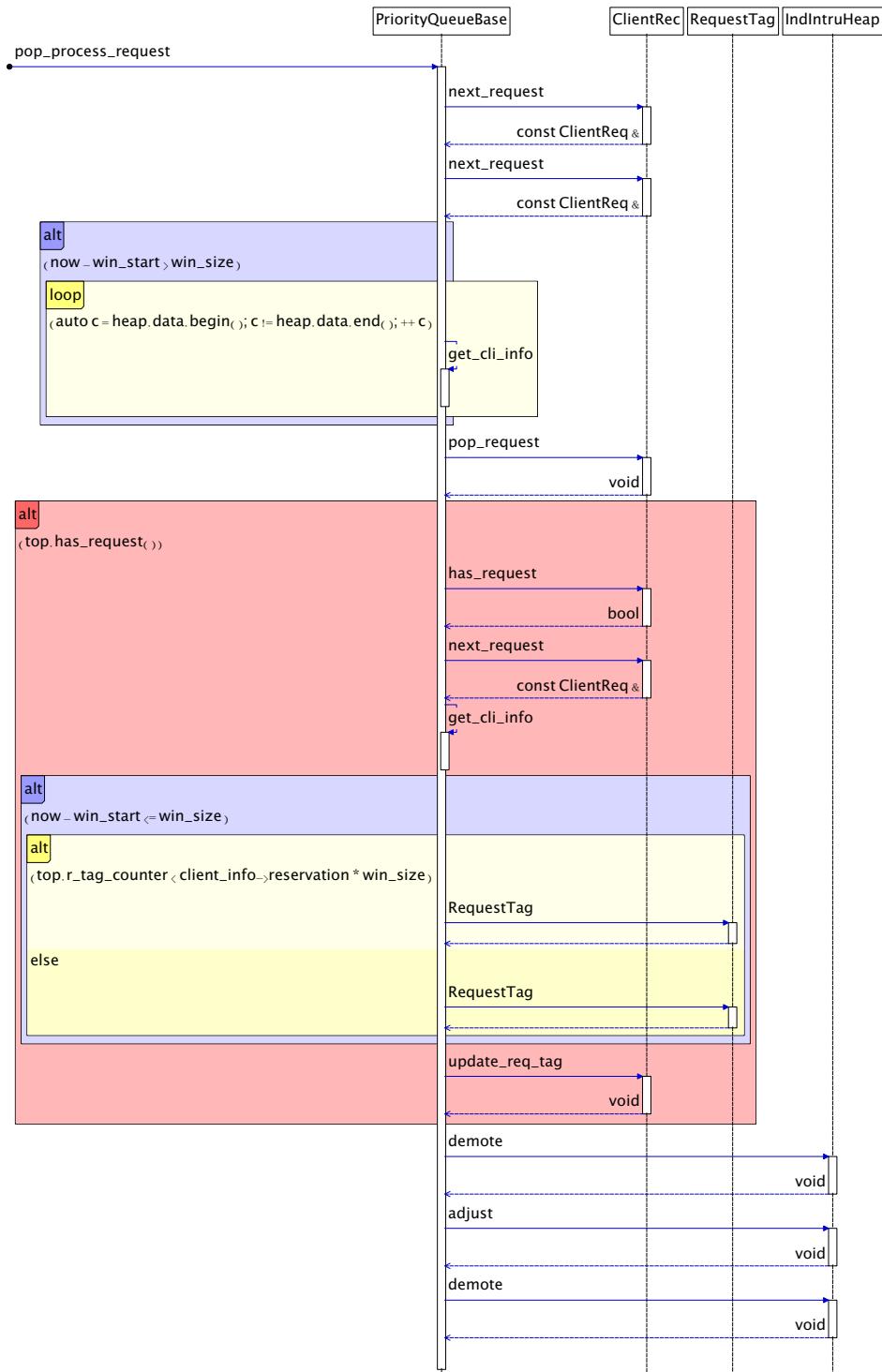


图 4.12 I/O 时序图

处与 RequestTag 类实例交互，这时在更新相应用户请求的标签。在更改标签后，需要重新调整最小堆的顺序，保证堆顶的标签是最小的。

同理，本文也使用上述提到的数据结构对出队函数进行了实现。

4.3 实验评估

本章安装部署 Ceph 系统，并对实现的 WinRWClock 算法进行了测试，与系统原有的 dmClock 算法进行对比分析。

4.3.1 实验配置

机器环境 实验是在装有 8 核的 Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz，内存 377G 和 1Gbps 网卡的服务器上进行的。内核版本是 4.14.146，操作系统版本是 CentOS 7.5.1804。

Ceph 平台搭建 本实验采用 Ceph 12.1.0 版本。使用 Dell Express Flash PM1725a 1.6TB AIC 型号的 NVMe 磁盘作为底层存储。文件系统类型使用 Ceph 的传统文件系统 (filestore)。创建 5 个池，每个池含有 32 个 PG 组。把这 5 个池看作 5 个用户，在现有版本的实现中，对池的访问会被标记成不同的用户。工作负载采用 Ceph[6] 官方社区提供的 benchmark 工具 radosbench[23]。使用该工具可以随机产生访问不同池、多种粒度的工作负载。

实验参数配置 如表格4.2中所示，本实验采取了六组服务质量目标配置。配置一，只给用户 1 配置最大的预留目标，权重值相同，这是为了验证算法在只有预留目标情况下的效果。配置二用户 1 和用户 5 配置了高权重目标 200 IOPS，五个用户的权重配比是 1:3:5:2:1。配置三与配置二对比，预留目标设置一致，权重目标五个用户一致。配置二与配置三结合，可以对比得出权重和预留目标共同作用下的效果。配置三至五，把用户 3 的权重值从 200 提高至 400，可以观察在某个用户预留目标的变化对算法效果的影响。

4.3.2 整体效果

按照章节4.1.2.1提出的目标，以及公式4.3所给出的 **DAT** 计算方法，我们可以分别计算出 WinRWClock 和 dmClock DAT 值，记为 $DAT_{winrwclock}$ 和 $DAT_{dmClock}$ 。

用户编号	配置一	配置二	配置三	配置四	配置五	配置六
1	(400, 1, 0)	(200, 1, 0)	(200, 1, 0)	(2, 5, 0)	(2, 5, 0)	(2, 5, 0)
2	(2, 1, 0)	(2, 3, 0)	(2, 1, 0)	(1, 5, 0)	(1, 5, 0)	(1, 5, 0)
3	(1, 1, 0)	(1, 5, 0)	(1, 1, 0)	(200, 1, 0)	(300, 1, 0)	(400, 1, 0)
4	(3, 1, 0)	(3, 2, 0)	(3, 1, 0)	(3, 5, 0)	(3, 5, 0)	(3, 5, 0)
5	(4, 1, 0)	(200, 1, 0)	(200, 1, 0)	(4, 5, 0)	(4, 5, 0)	(4, 5, 0)

表 4.2 WinRWClock 实验参数配置

在本章节中, 我们采用 WinRWClock 对比 dmClock DAT 值变化百分比 $\frac{|DAT_{winrwclock}| - |DAT_{dmClock}|}{|DAT_{dmClock}|}$ 来整体衡量算法的效果。如果百分比为负值, 说明 WinRWClock DAT 值小, 距离服务质量目标更近, 效果好; 如果为正值, 说明 WinRWClock DAT 值大 dmClock, 距离服务质量目标更远, 效果差。表格 4.3 给出了 500K 粒度下, 六组配置 WinRWClock 与 dmClock 变化百分比。在 500K 粒度下, 系统的写性能约为 1475 IOPS。在所有配置下, WinRWClock 均减小了与质量保证目标的平均差距。六组配置, 五个用户功平均减少了 73.29% 的差距。最好的情况出现在配置三中, 与 dmClock 相比, WinRWClock 使五个用户距离目标的差值平均减小了 99.65% 的误差, 这意味着新算法基本消除了与服务质量目标之间的误差; 最差的情况出现在配置五中, 与 dmClock 相比, WinRWClock 使五个用户距离目标的差值平均减小了 30.90%。出现正值的情况有两处: 配置五中, 用户 2 的 DAT 值增加了 3.88 IOPS; 配置六中, 用户 1 的 DAT 值增加了 1.29 IOPS。从绝对值来看, 14 IOPS 的误差是可以接受的。表格 4.4 给出 1M 粒度下, WinRWClock 算法与 dmClock DAT 值变化百分比。六组配置, WinRWClock 比 dmClock 平均减少 78.67% 的差距。最好的情况出现在配置一中, 平均减少 99.41% 差距, 这意味着 WinRWClock 算法基本与质量保证目标相同; 最差的情况发生在配置四, 平均减少 -39.79% 差距。唯一的正值出现在配置四用户 4 中, WinRWClock 比 dmClock 增加了 4.4 IOPS, 这个差距在实际应用中是可以接受的。

4.3.3 只设定预留目标的情况对准确度的影响

表 4.2 中给出的配置一是为了衡量在有一个用户设定较高的预留目标, 其他用户设置较低的预留目标, 而权重值相同的情况下, 两种算法的效果比较。

配置类型	用户 1	用户 2	用户 3	用户 4	用户 5	平均值
配置一	-83.42%	-64.59%	-61.39%	-73.92%	-62.74%	-69.21%
配置二	-98.81%	-94.04%	-92.05%	-93.51%	-98.82%	-95.45%
配置三	-99.92%	-99.49%	-99.87%	-99.49%	-99.49%	-99.65%
配置四	-96.59%	-96.89%	-98.95%	-98.58%	-98.43%	-97.88%
配置五	-30.23%	+35.96%	-92.32%	-39.36%	-28.55%	-30.90%
配置六	+13.80%	-33.72%	-93.13%	-65.75%	-54.29%	-46.62%

表 4.3 500K 粒度下不同配置 DAT 变化百分比

配置类型	用户 1	用户 2	用户 3	用户 4	用户 5	平均值
配置一	-74.46%	-65.35%	-66.30%	-97.64%	-64.0%	-73.57%
配置二	-99.70%	-99.54%	-99.72%	-98.40%	-99.70%	-99.41%
配置三	-96.61%	-96.26%	-97.32%	-95.18%	-95.93%	-96.26%
配置四	-47.41%	-53.65%	-92.51%	+34.21%	-39.58%	-39.79%
配置五	-63.30%	-65.03%	-81.58%	-64.23%	-58.58%	-66.54%
配置六	-97.40%	-97.66%	-99.86%	-90.78%	-96.69%	-96.48%

表 4.4 1M 粒度下不同配置 DAT 变化百分比

图 4.13 展示了距离目标平均差值 DAT 的结果。dmClock 算法中，用户 1 实际的得到的平均吞吐量值比目标吞吐量值少了 136.93 IOPS，而在 WinRW Clock 算法下，用户 1 纸币目标值少 22.1 IOPS，提升了 114.23 IOPS。相应地，用户 2~5 被 dmClock 算法多分配的吞吐量资源也减少了。由此可见，WinRW Clock 至少可以保证单一用户的预留资源目标。

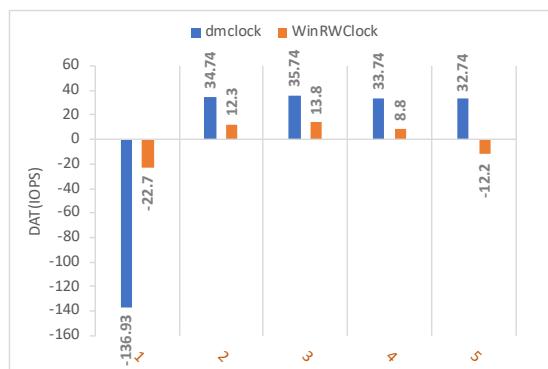


图 4.13 只设定预留目标的情况对准确度的影响

4.3.4 预留目标与权重目标共同作用对准确度的影响

配置二和配置三给用户 1 和用户 5 设置了较大的预留目标值。同时，配置二给五个用户设置了不同的权重值，而作为对照配置三给五个用户设定了相同的权重值。相同的预留目标、不同的权重目标的设置可以对比得出权重值设置对算法准确度的影响。同时，也可以得到算法是否可以在两个阶段相互作用下仍然效果良好。

图4.3.4和4.3.4分别给出了两种配置下的 DAT 结果。在配置二中，dmClock 性能隔离使得用户 1 和用户 5 比预期目标分别少占用了 56.25 和 56.58，而用户 2, 3, 4 分别多占用了 33.58, 58.75 IOPS 和 20.5 IOPS。而在配置三中，dmClock 使得用户 1 和用户 5 比预期目标各少占用了 118.4 IOPS，用户 2, 3, 4 分别多占用了 78.93, 79.93 和 77.93 (IOPS)。上述两组结果对比可知权重目标设置不同可以略微修正 dmClock 带来的偏差。在 WinRWClock 算法的性能隔离作用下，配置二中的五个用户分别距离预期目标 +0.67, +2, -4.67, +1.33, +0.67 (IOPS)；配置三中的五个用户分别距离预期目标 -0.1, 0.4, -0.1, 0.4, -0.6 (IOPS)。可以看出，WinRWClock 基本完成了隔离目标，与 dmClock 算法相比，获得了大幅提升。

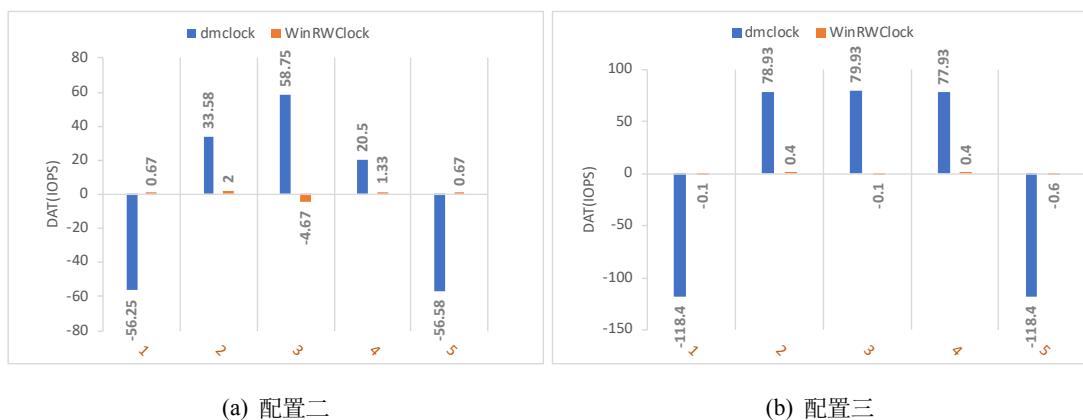


图 4.14 预留目标与权重目标共同作用对准确度的影响 (二)

4.3.5 预留目标变化对准确度的影响

配置四至配置六在设置了同样的权重目标，用户 3 的预留目标由 200 IOPS 增加至 400 IOPS。这三组配置对比可以得出预留目标变化对算法准确度的影响。

图4.15三个图分别给出了配置四至配置五的 DAT 结果。只考察用户 3, dmClock 算法在三种配置下效果类似，分别距离目标值 -40.89, -36.84 和 -35.39 (IOPS)。由此可见，预留目标值的变化并不会大幅影响算法准确度。WinRWClock 算法将用户 3 的 DAT 绝对值分别降低至 0.43, 2.83, 和 2.43 (IOPS)。通过三组实验对比可知，预留目标值的变化不影响 WinRWClock 算法比 dmClock 算法的优越性。

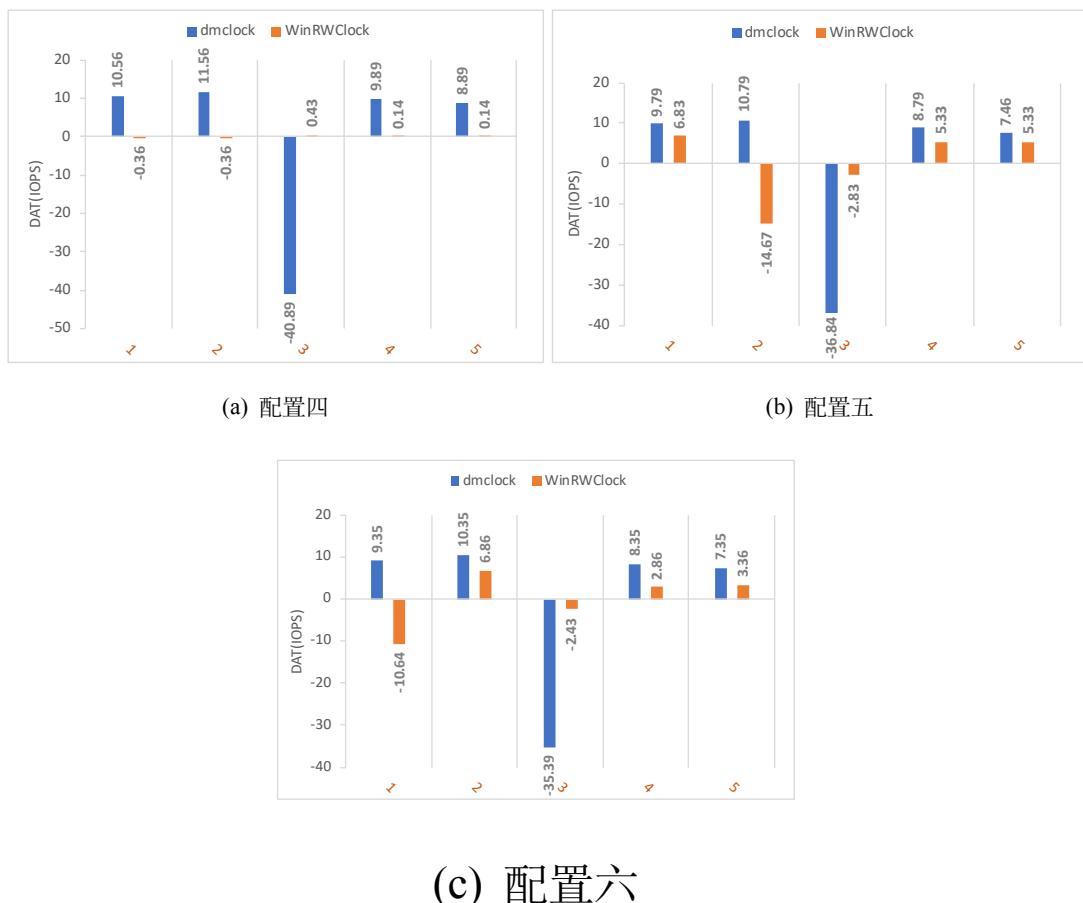


图 4.15 预留目标变化对准确度的影响

4.3.6 I/O 粒度对准确度的影响

本子章节评估算法在不同 I/O 粒度下的准确度变化。图4.16展示了 1M 粒度下的 DAT 情况。其中图4.3.6给出了配置一的情况，1M 粒度下，dmClock 给用户 1 带来的差值分别是 -188.33 和 -48.1 (IOPS)。图4.13中，500 K 粒度下，对应

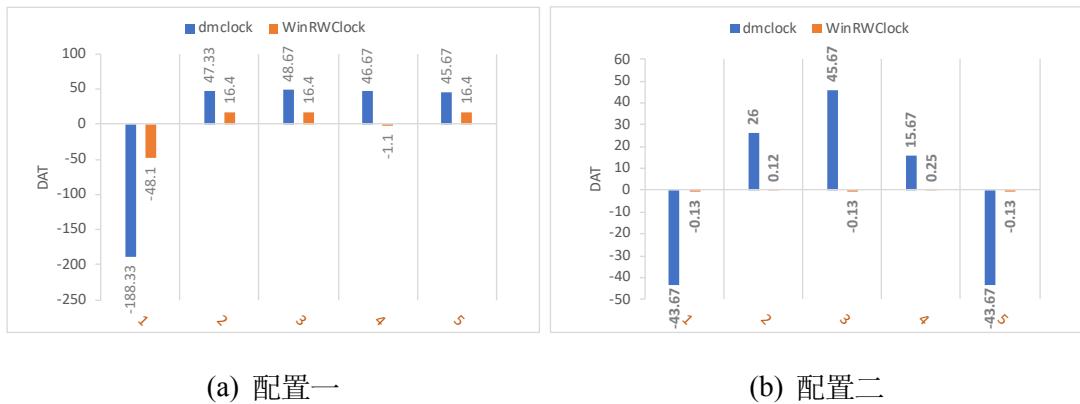


图 4.16 工作负载粒度对准确度的影响

的值分别是-136.93 和-22.7 (IOPS)。与 dmClock 算法相比，WinRWClock 算法还是有大幅提升。除此之外，这还说明两种算法在大的预留值，IO 粒度大的情况下表现不好。[4.3.6](#)给出了配置二的情况。dmClock 和 WinRWClock 给用户 1 带来的差值分别是-43.67 和-0.13 (IOPS)。500 K 粒度下，对应的值分别是-56.25 和 0.67 (IOPS)。说明在较小预留值和不同权重值的作用下，WinRWClock 算法的效果稳定不变，分别比 dmClock 提升 65.86% 和-79.81%。

4.4 本章小结

本章研究了典型的分布式存储平台性能隔离机制。剖析了 Ceph 的 I/O 调度流程，介绍了 dmClock 在 Ceph 中的作用原理；并实验得出经典性能隔离算法 dmClock 在 Ceph 平台上的效果，剖析其表现如此的原因；之后详细介绍了基于时间窗口的多时钟隔离算法 WinRWClock 的设计与实现；最后设计实验，从多个方面对比了 WinRWClock 的效果。经过大量实验表明，WinRWClock 算法比 dmClock 算法准确度平均提升 75.98%。

第5章 总结与展望

5.1 本文总结

5.2 未来展望

参考文献

- [1] 2018 年 ceph 用户使用调查结果. <https://ceph.com/ceph-blog/ceph-user-survey-2018-results/> Accessed, JULY 17, 2018.
- [2] Amazon elastic computing cloud. <aws.amazon.com/ec2>.
- [3] Amazon s3. <https://aws.amazon.com/cn/s3/>.
- [4] Azure blob storage. <https://azure.microsoft.com/en-us/services/storage/blobs/>.
- [5] bcache kernel documentation. <https://evilpiepirate.org/git/linux-bcache.git/tree/Documentation/bcache.txt>.
- [6] Ceph. <https://ceph.io>.
- [7] Ceph 官方架构图. <https://docs.ceph.com/docs/master/architecture/>.
- [8] Cloudhosting, cloud computing and hybrid infrastructure from gogrid. <http://www.gogrid.com>.
- [9] Dedicated server,managed hosting, web hosting by rackspace hosting. <http://www.rackspace.com>.
- [10] dmclock 在 ceph 中的实现。. <https://github.com/ceph/dmclock/releases>.
- [11] Docker. <https://www.docker.com/>.
- [12] Flexiscale cloud compand hosting. www.flexiscale.com.

- [13] Google app engine. <http://code.google.com/appengine>.
- [14] Google cloud storage. <https://cloud.google.com/storage>.
- [15] Google drive. <https://www.google.com/drive/>.
- [16] Google file system. https://en.wikipedia.org/wiki/Google_File_System.
- [17] Google storage. <https://cloud.google.com/storage/>.
- [18] linux namespace. https://en.wikipedia.org/wiki/Linux_namespaces.
- [19] Lustre. <http://lustre.org>.
- [20] Lustre. https://en.wikipedia.org/wiki/Device_mapper.
- [21] Microsoft azure remote desktop services. <https://docs.microsoft.com/en-us/windows-server/remote/remote-desktop-services/welcome-to-rds>.
- [22] Openstack swift. <https://wiki.openstack.org/wiki/Swift>.
- [23] Radosbench. https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/1.3/html/administration_guide/benchmarking_performance.
- [24] Salesforcecrm. <http://www.salesforce.com/platform>.
- [25] Sap business bydesign. www.sap.com/sme/solutions/businessmanagement/businessbydesign/index.epx.
- [26] Windowsazure. www.microsoft.com/azure.
- [27] 百度云服务. <https://cloud.baidu.com>.
- [28] 腾讯云服务. <https://cloud.tencent.com>.

- [29] 腾讯云犯下低级错误导致创业公司丢失大量数据. <https://tech.sina.com.cn/i/2018-08-20/doc-ihhxafa9715074.shtml>.
- [30] 阿里云服务. <https://cn.aliyun.com>.
- [31] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. *Proceedings of FAST 2010: 8th USENIX Conference on File and Storage Technologies*, pages 115–128, 2019.
- [32] Y. Chai, Z. Du, X. Qin, and D. A. Bader. Wec: Improving durability of ssd cache drives by caching write-efficient data. *IEEE Transactions on Computers*, 64(11):3304–3316, 2015.
- [33] Facebook. Flashcache: A general purpose, write-back block cache for linux. <https://github.com/facebookarchive/flashcache>.
- [34] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.
- [35] B. Gregg. L2arc. <https://blogs.oracle.com/brendan/entry/test> Accessed, 2008.
- [36] Ajay Gulati, Arif Merchant, and Peter J. Varman. Mclock: Handling throughput variability for hypervisor IO scheduling. *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010*, pages 437–450, 2010.
- [37] Tejun Heo. Control group v2. <https://www.kernel.org/doc/Documentation/cgroup-v2.txt> Accessed October, 2015.
- [38] Arne Holst. Data center storage capacity worldwide from 2016 to 2021, by segment. <https://www.statista.com/statistics/638593/worldwide-data-center-storage-capacity-cloud-vs-traditional> Accessed October, 2015.

- [39] S. Huang, Q. Wei, J. Chen, C. Chen, and D. Feng. Improving flash-based disk cache with lazy adaptive replacement. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2013.
- [40] Heinz Mauelshagen Joe Thornber and Mike Snitzer. dmcache kernel documentation. <https://www.kernel.org/doc/Documentation/device-mapper/cache.txt>.
- [41] T. Kgil, D. Roberts, and T. Mudge. Improving nand flash based disk caches. In *2008 International Symposium on Computer Architecture*, pages 327–338, 2008.
- [42] Yaser Mansouri, Adel Nadjaran Toosi, and Rajkumar Buyya. Data storage management in cloud environments: Taxonomy, survey, and future directions. *ACM Computing Surveys (CSUR)*, 50(6):1–51, 2017.
- [43] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage*, 4(3), November 2008.
- [44] Peter Mell and Tim Grance. On-demand self-service. *Nist*, 15:10–15, 2009.
- [45] Timothy Pritchett and Mithuna Thottethodi. Sievestore: A highly-selective, ensemble-level disk cache for cost-performance. *SIGARCH Comput. Archit. News*, 38(3):163–174, June 2010.
- [46] Mohit Saxena, Michael Swift, and Yiying Zhang. Flashtier: A lightweight, consistent and durable storage cache. *EuroSys ’12 - Proceedings of the EuroSys 2012 Conference*, 05 2012.
- [47] Denis Weerasiri, Moshe Chai Barukh, Boualem Benatallah, Quan Z. Sheng, and Rajiv Ranjan. A taxonomy and survey of cloud resource orchestration techniques. *ACM Comput. Surv.*, 50(2), May 2017.
- [48] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. CEpH: A scalable, high-performance distributed file system. *OSDI*

- 2006 - 7th USENIX Symposium on Operating Systems Design and Implementation, pages 307–320, 2006.
- [49] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RA-DOS: A scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Petascale Data Storage Workshop, PDSW '07, held in Conjunction with Supercomputing '07*, pages 35–44, 2008.
 - [50] Q. Yang and J. Ren. I-cash: Intelligently coupled array of ssd and hdd. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 278–289, 2011.
 - [51] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

致谢