doi:10.3772/j.issn.1002-0470.2020.02.001

面向强一致性的分布式对象存储的I/O并行性优化①

史 骁②*** 宋永浩* 郑晓辉*** 唐宏伟* 于 雷* 赵晓芳③*

(*中国科学院计算技术研究所 北京 100190)

(**中国科学院大学 北京 100049)

摘 要 在提供数据强一致性保障的分布式对象存储系统中,其I/O并行性受到系统I/O调度算法(主副本优先调度)的限制。本文提出了一种简单高效的I/O调度策略,其基于多副本的主从模型,可在保障强一致性的同时充分挖掘I/O并行空间。其包括如下3个主要步骤:第一,I/O请求被发送至主副本节点进行负载合并;第二,这些请求被送至数据相关性检测器根据相关性分配优先级;第三,根据I/O优先级及负载分布,将I/O请求尽可能均衡地转发至各副本节点上并行处理。本文实现了一个分布式对象存储系统原型用于验证该策略的有效性。实验分别对本文策略的各个环节进行了评估,实验结果表明,较主副本优先调度策略,本文策略使得GET请求吞吐量最大提升41.8%,GET请求平均延迟最大降低42.5%,GET请求99.9th延迟最大降低15.8倍,这使得系统性能达到最终一致性下基准调度策略C3的水平。

关键词 分布式对象存储;I/O并行性;多副本;强一致性

0 引 言

分布式对象存储是云计算存储的主要形式之一,其可有效满足数据中心存储服务的高扩展性需求[1-4],常用于存储大规模的非结构化数据,包括文本、图片、视频、训练集、机器学习模型等。为充分发挥存储系统性能、提高存储系统利用率,良好的系统I/O并行性是一个基本条件[5]。

系统设计人员常常通过降低系统的数据一致性级别来提高系统的I/O并行性,但弱一致性级别及相关I/O并行优化技术无法满足具有强一致性需求的应用。最终一致性是分布式对象存储中常采用的一种弱一致性模型[4],其特点是:对象读(GET)请求可以在任意可用的副本节点执行。但是,当应用的I/O模式具备流水线特征时,最终一致性模型所引发的数据不一致会影响应用计算的准确性及存储系统利用率[1]。针对弱一致性模型,现有工作提出了一系列借助负载均衡优化I/O并行性的方法[5-10]。例如,C3[5]提出根据存储节点主动反馈的I/O请求队列长度,在存储客户端建立存储节点的实时负载模型以供调度参考,从而实现负载均衡。其他针对一致性的研究多集中在探讨如何协调复杂应用的一致性需求及存储系统所提供的一致性服务能力[11,12],或如何高效、实时地对服务中出现的数据不一致性进行检测[13]。这些研究工作通常以较弱的一致性模型为基础,这使得其难以移植到强一致性的场景中。

为满足强一致性需求,工业界提出了一系列提高一致性保障的措施。例如,Google的分布式对象存储服务利用Spanner提供跨数据中心的全局强一致性[1],但其独特的软硬件设施(其在数据中心部

① 国家重点研发计划(2018YFB0904503)和国家自然科学基金(61672499)资助项目。

② 男,1990年生,博士生;研究方向:分布式存储系统优化;E-mail:shixiao@ict.ac.cn

③ 通信作者,E-mail:zhaoxf@ict.ac.cn

(收稿日期:2019-03-07)

署了 GPS、原子钟及相应的时间管理 API[14])不具备可复制性;Amazon S3 通过保障桶信息的“写后读”相关性[3]提高一致性级别,但其数据一致性保障粒度仍然较粗,仍以牺牲对象的一致性来换取 I/O并行性的提升。

目前,为了实现强一致性,系统设计普遍采用主副本优先调度策略。基本的多副本机制要求系统对任一数据对象维护指定数目的副本数据。同一数据对象在系统中的多个副本可构成完整的“副本组”。用于存储某一副本的存储节点称为“副本节点”。在此基础上,主从多副本机制规定副本组内的某个副本为该组的“主副本”,而其他副本为“从副本”。相应地,主副本所在的存储节点可称为“主副本节点”。在副本主从关系的基础上,主副本优先调度策略要求各主副本节点执行副本组中数据对象的所有 I/O 请求。该策略假设 I/O 并行性可以借助数据分布的随机性实现。但实际应用中,由于数据访问热度并不均衡,该策略极易使得系统负载分布失衡[5-7],从而影响 I/O 并行性。这在第 5 节所述实验中同样得到了验证。

针对强一致性下 I/O 并行性优化技术尚不完善的现状,本文的研究目标是在提供强一致性保障的同时,优化系统的 I/O 并行性,以提高强一致性存储服务的性能。本文提出了一种基于多副本的 I/O 调度策略,适用于常见的分布式对象存储系统,其主要包括 3 个步骤。第一,I/O 请求被发送至数据所在的主副本节点进行 I/O 负载合并以及强一致性关系确认。本文策略利用主从多副本机制,所有 I/O 请求均需被发送至所在副本集合的主节点上,并等待该节点实施有效的 I/O 调度。为减少相关性冲突,主副本节点需要在调度前结合当前上下文中的请求执行负载合并,并确定 I/O 请求之间的一致性关系。第二,针对每个 I/O 请求,分析当前环境中其可用的 I/O 并行决策空间,设置相应的执行优先级。优先级一方面保障数据相关性,另一方面保障请求时间窗口的顺序性。该机制支持来自同一副本集合的 I/O 请求在任意一个副本节点上进行优先级比较。第三,均衡 I/O 负载。经过预处理后的 I/O 请求在主副本节点等待调度。根据多副本主从模型,所有写请求由主副本节点协调执行。具备相关性的 I/O 请求在主副本节点有序执行。其余读请求则可完全并行执行。

本文实现了一个分布式对象存储系统原型,其支持多副本的主从模型以及基于哈希的数据分布定位机制。为进行对比实验,在原型系统中分别实现了主副本优先调度、随机调度、C3[5]、本文方法这 4 种调度策略。实验结果表明,本文策略可以在保障强一致性的同时,充分释放系统 I/O 并行性,提高系统资源利用率。其具体包括:负载合并可有效提高存储资源利用率,使得系统 GET 吞吐量最大提升 6.4%;较常用强一致性保障策略(主副本优先调度),本文策略使得 GET 请求吞吐量最大提升 41.8%,GET 请求平均延迟最大降低 42.5%,GET 请求 99.9^{th} 延迟最大下降 15.8 倍,使系统性能达到最终一致性下基准调度策略 C3 的性能水平;数据相关性导致吞吐量下降幅度不超过 7.2%。

本文的主要工作包括:

(1) 设计实现了强一致性下可执行 I/O 并行性优化的调度框架。

(2) 基于调度框架设计实现了负载合并器、数据相关性检测器以及负载均衡器。

(3) 在原型系统上实验验证了本文工作的有效性。

本文组织结构如下:第 1 节对本工作研究背景及问题挑战进行了阐述;第 2 节阐述了支持强一致性的 I/O 并行优化调度框架;第 3 节描述了框架内各子模块的输入输出及核心算法;第 4 节简述了原型系统的实现;第 5 节阐述了实验结果;第 6 节介绍了多副本调度以及一致性模型的相关研究;第 7 节对本文进行了总结。

1 背景及挑战

1.1 系统架构因素

分布式对象存储系统对数据实行扁平化的管理,具备以下特点:对象在整个系统中被唯一标识;对象无差异地隶属于桶或容器中;桶中的对象数量不受限制;对象是数据操作的最小单元,即标准 API

(application programming interface)仅支持针对对象整体的读(GET)、写(PUT)、删除(DELETE)[1,3]。

分布式对象存储中,以下 2 项架构因素影响本研究工作的开展:(1)数据可靠性机制:多副本或纠删码;(2)对象定位机制:哈希或元数据。数据可靠性机制影响 I/O 并行化的决策空间。对象定位机制影响调度器的架构、交互方式及功能定义。本文工作针对采用多副本及一致性哈希定位的分布式对象存储系统。

多副本机制在诸多分布式存储系统中得到广泛应用,包括 Ceph[2,15]、Lustre[16]、HBase[17]、Cassandra[18]以及 Openstack Swift[19]等。其要求系统分散地维护多个数据副本,一致性保障依靠主从模型或者 QUORUM 等模型。主副本优先调度便是面向主从模式的多副本机制而设计。在强一致性前提下,系统可保障任意对象的操作满足“写后读”一致性要求。

一致性哈希使用对象标识作为哈希函数的输入,计算对象的存储位置。基于一致性哈希定位机制,系统可以分布式地计算对象的副本节点集合,更加灵活地支持调度框架的设计。

1.2　I/O 并行化的决策空间

I/O 并行化的决策空间,或称 I/O 调度的决策空间,是指在满足一致性级别的前提下,可执行给定 I/O 请求的存储节点构成的集合。在主从多副本机制下,写请求的调度决策空间仅包含主副本节点;读请求的调度决策空间受到数据一致性模型影响。例如,最终一致性模型下,读请求的调度决策空间为副本组内的所有节点。

I/O 决策空间的计算设计可划分为 2 个子任务。

(1) 选择计算架构。计算可采取中心化或分布式的架构。由于 I/O 调度服务需实时处理高并发负载,本文研究中采用分布式架构。结合主从多副本机制的特点,由主副本节点执行调度决策操作,一方面满足分布式计算架构需求,另一方面为 I/O 并行性空间的计算提供了基础。

(2) 确立计算模型。计算模型可兼顾多项因素,例如可靠性机制、负载分布、网络状态等。有关分布式系统任务调度的相关研究常利用多副本的并行化空间进行负载均衡,将调度决策转化为根据模型计算最优决策的问题,包括负载均衡预测模型[5,8]、探索-利用模型[9]等。本文方法的计算模型的设计结合了负载均衡因素。

1.3　挑战:强一致性对调度决策空间的限制

强一致性要求具备相关性的 I/O 请求按照时间序列关系线性执行,约束了每个 I/O 请求的调度决策空间,从而影响了 I/O 并行性。同时,为提高 I/O 并行性,将 I/O 请求分配至多个节点上并行执行,若后续 I/O 请求与已经分配的 I/O 请求之间存在相关性,则系统需要跨节点维护、追踪相关性信息。在分布式系统架构中动态实时地维护相关性信息难度较大。

表 1 展示了常见分布式存储系统中调度策略与一致性模型的对应关系。可见,为实现强一致性,分布式对象存储系统普遍采用主副本优先调度策略,而这一策略未充分地挖掘调度决策空间,无助于提高 I/O 并行性。在主副本优先调度中,所有属于相同副本集合的 I/O 请求均在主副本节点进行有序的排队。例如,一同排队的 < GET, obj1 > 和 < GET, obj2 > 请求,它们之间不存在任何数据相关性,响应顺序完全不影响二者操作的正确性。这种所有I/O

表 1　常见分布式存储系统的多副本调度策略

名称	类型	多副本调度策略	一致性模型
Ceph Rados	分布式对象存储	主副本优先	强一致性
		随机调度	最终一致性
		最近节点优先	
Lustre	分布式文件存储(底层为分布式对象存储)	基于历史负载状态	强一致性,需要动态地维护副本的状态、可用性
HDFS	分布式文件存储	最近节点优先	强一致性,假设一写多读
HBase	分布式 KV 数据库	主副本优先	强一致性
		批量请求所有副本节点	最终一致性或强一致性
Cassandra	分布式 KV 数据库	多种策略	多种一致性模型
		C3	最终一致性
OpenStack Swift	分布式对象存储	随机调度	最终一致性

请求在主副本节点上有序执行的方式严重浪费了多副本所提供的并行 I/O 空间。实际上,通过本文工作可以看出,强一致性与主副本优先调度并不存在绑定关系。

表 1 中其他调度策略虽然可以利用多副本的并行 I/O 空间,但不能保障强一致性。国内外相关研究多聚焦于使系统提供与应用需求所适宜的数据一致性模型。例如,文献[12]和文献[13]的作者探讨了在存储系统中同时提供多种一致性模型所面临的挑战。

总之,强一致性下的 I/O 并行性优化技术发展需要新的调度策略。

2 调度框架

本节提出了支持强一致性的 I/O 并行性优化调度框架,如图 1 所示。不同于以往的调度策略,本框架以各主副本为中心,要求客户端将 I/O 请求发送至主副本节点,由各主副本节点独立实施调度。调度器由以下 3 部分构成。

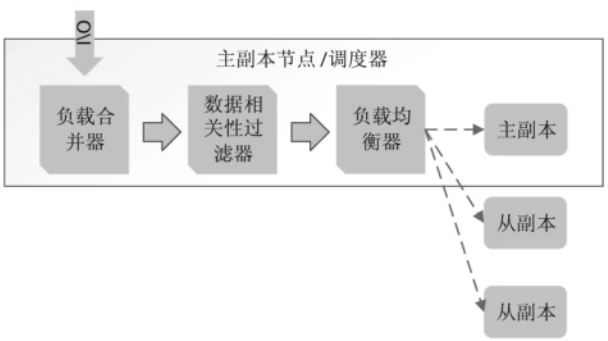

图 1 调度框架

(1) 负载合并器。合并不同客户端之间的同质 I/O 请求,以节省 I/O 处理开销。所谓同质 I/O 请求,指的是针对同一对象的同类型 I/O 请求。I/O 序列中相邻的同质请求可得到相同的响应结果。由于对象存储的编程接口十分简明(主要包含 GET、PUT、DELETE 等),且操作以对象为最小单元,这为合并同质 I/O 请求提供了基础。

(2) 数据相关性检测器。数据相关性检测器完成数据相关性的过滤以及优先级分配。

(3) 负载均衡器。负载均衡器负责将预处理后的 I/O 请求序列在所属的副本集合节点之间进行均衡。

2.1 架 构

根据多副本主从模型的特点,本文工作选择将主副本节点设置为数据相关性检测器,作为保障强一致性的核心模块。在主副本节点实现数据相关性检测具备以下优势:

(1) 架构天然地利用数据分布的随机性,对 I/O 的数据相关性负载进行了分组。由于强一致性的数据相关性检测以对象为单位,这要求系统提供统一视图,以便无遗漏地比较不同客户端之间的 I/O 请求。但实际运行环境中出现的高负载现象要求采用分布式的数据相关性检测器。若采用独立的节点进行相关性检测,客户端之间将引入大量点到点的网络通信,其通信规模与客户端的数量以及 I/O 负载强度成正比。而利用副本主节点实现分布式检测,一方面可以减少不必要的通信,另一方面可以利用现有机制避免额外的负载均衡机制。

(2) 有利于 I/O 请求的数据相关性定序。数据相关性关系确定后,系统需保障相关 I/O 的线性响应顺序。如确保将具备相关性的 I/O 请求正确发送给副本节点,需要客户端或检测器与副本节点之间进行额外的通信。在本架构下,检测器可保障相关的 I/O 请求顺序插入本地 I/O 请求队列。

(3) 减少通信开销。在本架构下,调度框架对客户端透明,客户端将 I/O 请求发送出去后即完成任务,避免了大量低效的通信。而调度器与其他副本节点之间的通信采用批量发送 I/O 的技术,可大幅度节省通信开销。

在进行相关性检测之前,本框架通过负载合并器对同质的 I/O 请求进行合并。我们定义同质的 I/O 请求为针对同一个对象的同类型操作。这样,冗余的 I/O 操作可以被有效避免。例如,<client1, obj1, GET>, <client2, obj1, GET> 可合并为 <(client1, client2), obj1, GET>。该类 I/O 请求执行完毕后需向所有请求客户端进行响应。

数据相关性检测器将标记好优先级的 I/O 请求转发给负载均衡器。负载均衡器负责保持本副本组内各节点的负载均衡。需要注意的是,不同副本组

的节点集合之间存在交集。各个副本组的负载均衡器实现的是来自本副本组内 I/O 请求的均衡。负载均衡器通过批量调度,按各个设备的 I/O 处理能力对请求分布进行均衡,并将同目标节点的 I/O 请求打包后发送给目标副本节点。

2.2　存储客户端作用域

对于存储客户端而言,I/O 调度过程对其完全透明,其职责基本未发生改变。但不同于其他策略的是,存储客户端将 I/O 请求提交至主副本节点,而实际的请求响应节点可能是副本组内的任意一个节点,如图 2 所示。

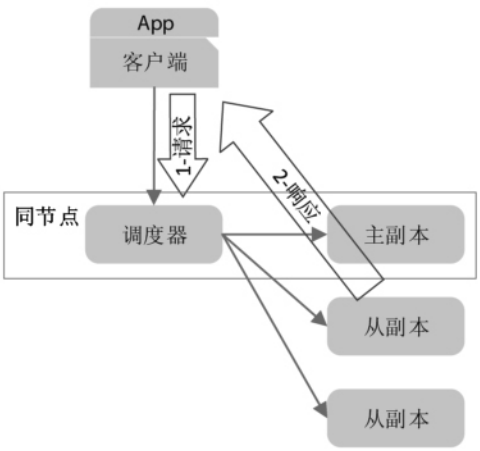

图 2　客户端视角

2.3　调度器作用域

调度器内数据通信通过线程通信完成,但是被调度至其他副本节点的 I/O 请求需要通过一次网络通信进行转发。调度器与其他副本节点通信时,采用批量方式转发待处理的 I/O 请求,如图 3 所示。

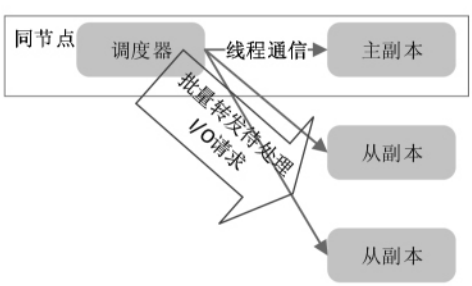

图 3　调度器视角

2.4　副本节点作用域

副本节点执行调度器提交的 I/O 请求序列,其优先级队列需严格参照调度器中所分配的优先级。这样,同一副本节点中,来自同一副本组的 I/O 操作的优先级可保持一致性。

3　调度模块

本节对调度器主要模块的功能进行了定义。

3.1　负载合并器

负载合并器主要实现 2 个任务。首先,负载合并器周期性地收集一组待处理的 I/O 请求。每个周期形成一个时间窗口。负载合并器分别记录 I/O 请求的目标对象以及 I/O 类型。然后,负载合并器对该组请求内的同质(同对象、同 I/O 类型且其间不夹杂同对象、其他 I/O 类型)I/O 操作进行合并。这 2 个任务完成后,待处理 I/O 请求的强一致性关系即得到确定。本文统一采用了向前合并的策略处理 GET、PUT、DELETE 操作,这里分别对它们合并的方式及影响进行说明:

(1) GET 合并。GET 合并遵从的重要原则是减少用户应用程序不必要的重试和过时数据的读取,包括桶以及对象的读取操作等。执行合并后的 GET 请求置于第一个被合并 GET 请求的位置。

(2) PUT 合并。PUT 请求写入或更新对象数据,按照强一致性的时间序列化关系,合并后的 PUT 请求写入最后一个到达的 PUT 请求的内容,并将其置于第一个被合并 PUT 请求的位置。

(3) DELETE 合并。DELETE 请求删除对象数据,其操作完全冗余。DELETE 请求合并后置于第一个被合并的 DELETE 请求的位置。

这种局部的重排序之后,后续的操作将严格遵守此时确定的数据相关性关系,如图 4 所示。这里所讨论的负载合并同操作系统内核 I/O 调度中的 I/O 合并(向前或向后并)[19] 的区别在于:

(1) 处理阶段不同。本文负载合并在用户态进行;内核 I/O 负载合并在内核态执行。

(2) 影响范围不同。本文负载合并可影响多个节点,在分布式系统层面减少 I/O 负载量;内核 I/O 负载合并局限于单点。

(3) 目标不同。本文负载合并的一个重要目的

是简化数据相关性的分析及一致性的管理；内核 I/O 负载合并不存在这一目标。

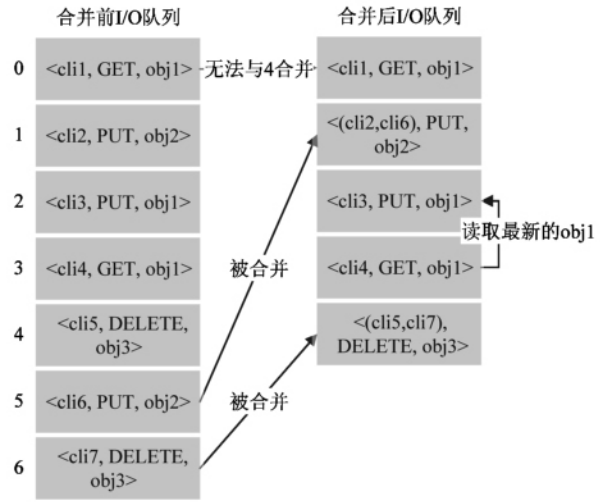

图 4　负载合并示意图

3.2　数据相关性检测器

负载合并器确定了一个时间窗口内的数据强一致性关系，并将待调度 I/O 请求序列转发至数据相关性检测器。数据相关性检测器一方面维护当前序列中的数据相关性，另一方面需要维护不同时间窗口之间请求序列的数据相关性。读操作包括系统内不同粒度的 GET 操作，写操作包括系统内不同粒度的 PUT、DELETE 操作。这里针对写后读、读后写、写后写的处理进行说明。

（1）写后读相关。写后读相关的存在不受时间窗口的限制，这种相关性维护需要在时间线上进行宏观的管理。

（2）读后写相关。读后写相关的存在同样不受时间窗口的限制。

（3）写后写相关。写后写相关仅存在于不同时间窗口的写、写请求之间。

数据相关性检测器将数据相关性转换为确定的优先级关系。优先级以整数形式表示 I/O 的相对执行顺序，其数值越大，则优先级越低。优先级的分配需考虑时间窗口及数据相关性关系。为方便优先级编码，每个时间窗口内的优先级起始编码为常数 * 时间窗口号。这表示在一个时间窗口内，针对同一对象的 I/O 操作间最多可具备常数个相关性关系。本文称之为相关性系数。当优先级个数不够用时，将 I/O 推迟至下一时间窗口处理。在每个时间窗口内，检测器负责分配多个连续的优先级，所有在相关性中需要优先执行的 I/O 请求、无相关性的 I/O 请求被分配较高优先级，其他 I/O 请求被分配较低优先级，按照相关性逐级加 1，如算法 1 所示。

算法 1：数据相关性检测及优先级分配

输入： I/O 序列 *IOSeq*，
时间窗口编号 *WID*，
数据相关性记录表 *rel _ map*，
优先级累加记录表 *prio _ map*，
相关性系数 *R*

输出： 标记了优先级的 I/O 序列 *IOSeq*

```
1:  Priorities = {WID * R, WID * R + 1, ···}
2:  for Req in IOSeq:
3:    // assign lower priority
4:    if rel _ map.find(Req. obj):
5:      Req. priority = max(prio _ map[Req. obj] + 1,
6:  WID * R);
7:      prio _ map [Req. obj] = Req. priority;
8:    else
9:      Req. priority = WID * R;
10:   // for future checking
11:   if Req. type = = PUT || Req. type = = DELETE:
12:     rel _ map. insert(Req. obj);
```

不同副本组内分配的优先级不具有可比性，但同副本组内分配的优先级在任一副本节点上都可比较。各副本节点的 I/O 队列遵守该优先级关系进行 I/O 处理。例如，当主副本节点将写操作下发至某个从副本节点进行数据同步时，其优先级依旧保持原有级别，当前从副本节点中若存在优先级较高但尚未执行的 I/O 请求，则该写操作仍然需要等待高优先级操作执行完成之后方可执行。这样，相关性即可依靠优先级进行保障，从而实现强一致性。

为维护一致性关系，数据相关性检测器接受调度器的反馈，当 I/O 请求被插入 I/O 队列后，相应的相关性约束可被释放。数据相关性检测器将分配好优先级的 I/O 请求序列转发至负载均衡器。

3.3　负载均衡器

根据多副本的主从模型，所有写请求都需要遵从先主后从的执行顺序。因此，相关的写后读、写后写请求均直接在主副本节点上继续排队。

对于其他 I/O 请求,负载均衡器按照负载分布状况进行调度。由于对象存储系统的 I/O 操作单元以对象为最小单位,因此,本文认为各节点的负载状况可以参考 I/O 队列长度进行判断。由于负载均衡算法并非本文研究的重点,这里使用了较为简单的负载均衡策略(算法 2 所示)——轮询方式配合提升 I/O 并行性。在每次负载均衡调度时,先将具备相关性的I/O操作分配给主副本节点,其他 I/O 操作则优先分配至当前时间窗口内负载量最少的节点。

算法 2:负载均衡策略

输入: I/O 序列 *IOSeq*,
相关性系数 *R*
输出: 各副本节点 *I/O* 序列 *NodeSeq*

```
1:  // Schedule I/Os in data relativity first
2:  for Req in IOSeq:
3:    if Req.priority%R == 0:
4:      if Req.type == (PUT | DELETE):
5:        node = current;
6:    else:
7:      node = current
8:    NodeSeq[node].insert(Req);
9:  // Schedule other I/Os.
10: for Req in IOSeq:
11:   if Req.priority%R == 0:
12:     if Req.type ! = (PUT | DELETE):
13:       node = argmin_node(NodeSeq(node).size());
14:     NodeSeq[node].insert(Req);
```

4　系统原型实现

为了验证本文方法的有效性,本研究实现了一个分布式对象存储系统的原型。该对象存储系统包含了所需的基本功能。其数据可靠性机制采用了主从多副本机制;数据分布机制使用一致性哈希方法;客户端模块支持 GET、PUT 对象操作接口;底层的对象存储利用本地文件系统实现;后端存储节点模块支持优先级 I/O 队列,并支持本文调度器中的优先级规则。

除本文调度策略之外,还在系统原型中支持了 3 种调度策略:

(1) 主副本优先调度策略(MAIN 调度)。所有 I/O 请求均提交至主副本节点处理。该策略支持强一致性模型,被广泛使用,其作为强一致性下的负载均衡基准算法。

(2) 随机调度策略(RANDOM 调度)。调度器为读请求随机地选择一个可用的副本节点。该模式仅支持最终一致性模型。

(3) C3[5]。调度器根据存储结点 I/O 队列长度的实时反馈及并发队列补偿以模拟负载分布状况。该调度策略提供了较好的负载均衡能力,但仅支持最终一致性模型,其作为最终一致性下的负载均衡基准算法。

以上前 2 种调度策略均在客户端实现,C3 需要客户端及各个存储节点的交互,而本文方法在主副本节点模块中实现。在实际工作过程中,MAIN 和 RANDOM 调度策略均不需要借助任何网络通信,仅通过一致性哈希来定位和选择 I/O 请求的目标执行节点。C3 需要借助网络通信更新队列长度信息,该通信开销可通过 I/O 响应信息来捎带传递。在本文调度策略中,一个 I/O 请求的调度,至少需要 1 次网络通信,最多需要 2 次网络通信。第 1 次网络通信由客户端将 I/O 请求发送至主副本节点,第 2 次网络通信由主副本节点将 I/O 请求根据负载均衡需求批量发送至其他副本节点。后续实验表明,网络通信的开销未明显增加 I/O 请求的处理延迟。

5　实　验

本节介绍了验证本文策略实际效果的方法及实验结果。实验测试集群包含 5 台存储服务器, 30 块 HDD 硬盘以及独立的客户端服务器。系统存储池中包含 512 个副本组。测试开始前,测试存储池中预先保存了 10 万个大小均为 4 kB 的对象。

测试中的 I/O 负载利用 COSBench 工具[20] 生成。实验中根据模型系统的接口修改了 COSBench 的功能。该工具通过多线程客户端模拟并发负载场景。模拟程序可调节 2 个参数:负载并发度以及I/O 读写比例。负载并发度指客户端同时等待的 I/O 请求的总个数,该指标可探索系统的吞吐量上限。I/O读写比例指的是所有客户端所产生的 I/O 请求

中读请求和写请求的个数的比值。实验中调度器模块的时间窗口设置为 500 μs,相关性系数设置为 5。时间窗口间隔设置较小,主要为了避免时间窗口内 I/O 的过量堆积,从而影响响应延迟。受此影响,相关性系数为 5 基本可以满足用于标记一个时间窗口内的优先级关系的需求。测试中所有对象的大小均为 4 kB。在此基础上,对不同 I/O 负载下的调度策略所展现的系统性能进行了度量。

实验结果表明,本文策略可以在保障系统强一致性下较好地发挥 I/O 并行性,实现系统利用率等的提升。

5.1 负载合并模拟

本节对负载合并的效果进行了模拟实验。实验通过调节固定并发度下(并发度为 64),同质 I/O 请求比例进行。由于无法量化应用中的实际同质 I/O 请求比例,模拟针对同质 I/O 比例为 0 ~ 20% 下负载合并的收益进行了统计。其收益具体表现为实际 I/O 并发度的降低。通过记录 I/O 负载合并器所提交的 I/O 序列长度和,可得图 5 所示合并效果,其他策略由于不具备负载合并策略,而保持恒定并发度。由图 5 可见,得益于同质 I/O 请求合并,实际的 I/O 并发度得以降低,从而有助于系统资源的节省和利用率的提升。

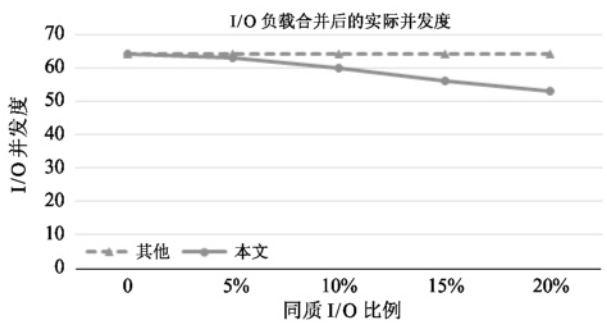

图 5 负载合并效果

5.2 负载均衡性能

本节对本文策略的 I/O 并行性优化效果进行测试。由于各基准算法的一致性保障级别不同,而强一致性会引起一定的性能损耗,因此在本节评估中,通过采用 100% 的 GET 操作集合来避免数据相关场景及其可能引发的开销。同时,为进一步度量负载合并的作用,实验分别测试了不启用及启用负载合并功能的本文策略的效果。本节使用“本文-负载合并”表示未启用负载合并功能的策略。

该负载模式下,通过提高客户端 GET 请求的并发度(16 ~ 2048),测试分布式对象存储系统的性能和服务质量及相应上限。测试中关注的主要指标包括 GET 请求吞吐量、GET 请求处理延迟(平均延迟、99 百分位延迟、99.9 百分位延迟)。实验表明,本文策略对于提高分布式对象存储系统存储资源利用率、降低请求服务延迟、减少请求延迟分布的波动情况有明显效果。

本文策略有助于提高分布式对象存储系统的资源利用率。如图 6 所示,在不同 GET 请求并发度下,利用本文多副本调度算法的分布式对象存储系统表现出良好的 GET 请求吞吐量。随着 GET 请求并发度的不断提高,4 种算法均提高了整体 GET 吞吐量,但是,MAIN 远未达到系统吞吐量上限,RANDOM 在较高并发度下接近了系统吞吐量上限,C3 和本文-负载合并在并发度达到 64 后达到了系统上限,而本文策略较本文-负载合并进一步提高了系统吞吐量,说明负载合并效果带来了明显的收益。在并发度为 64 时,本文策略所产生的 GET 请求吞吐率较 MAIN 提高了 41.8%,较 RANDOM 提高了 29.1%,较 C3 提高了 6.4%。这说明,本文策略的负载均衡效果明显优于其他基准算法。

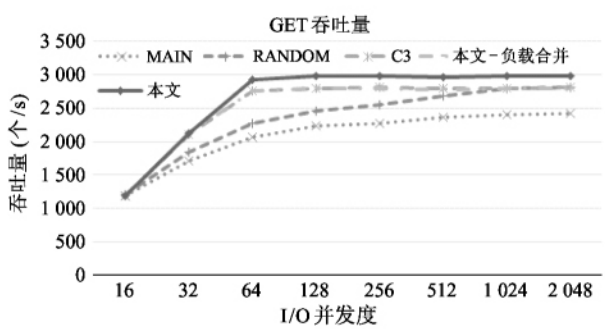

图 6 不同策略的 GET 请求吞吐率

本文策略有助于降低 GET 请求平均延迟。如图 7 所示,在不同 GET 请求并发度下,尤其是随着并发度的提高,利用本文策略下所得的 GET 平均处理延迟明显低于其他策略。并发度为 64 时,本文策略较其他策略优势最明显,其 GET 请求平均延迟较 MAIN 降低了 42.5%,较 RANDOM 降低了 29.7%,

与 C3 基本持平。

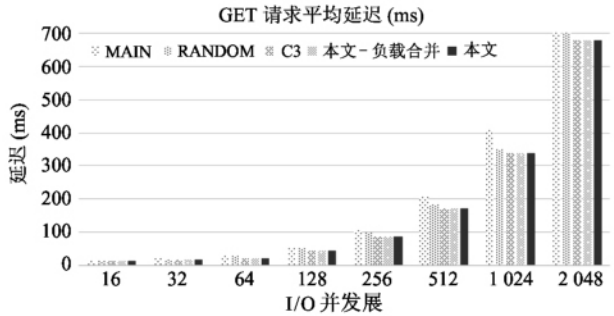

图 7　不同策略的 GET 请求平均延迟

本文策略有助于降低 GET 请求延迟分布的波动状态。如图 8 所示,在不同 GET 请求并发度下,利用本文策略所得的 GET 请求 99.9th延迟明显低于其他策略。当 I/O 并发度达到 256 时,本文策略效果的优势最为显著,其 99.9 百分位延迟较 MAIN 策略降低了 10.3 倍,较 RANDOM 策略降低了 15.8 倍,与 C3 基本持平。这说明,本文策略有助于提供可靠的、可预测的 I/O 服务。如图 9 所示,在不同 GET 请求并发度下,本文策略可以有效地保障延迟分布的平稳。

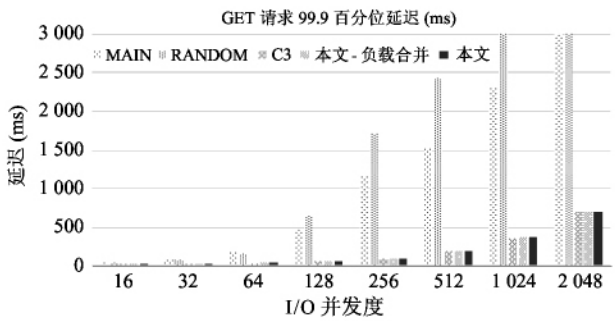

图 8　不同策略的 GET 请求 99.9th延迟

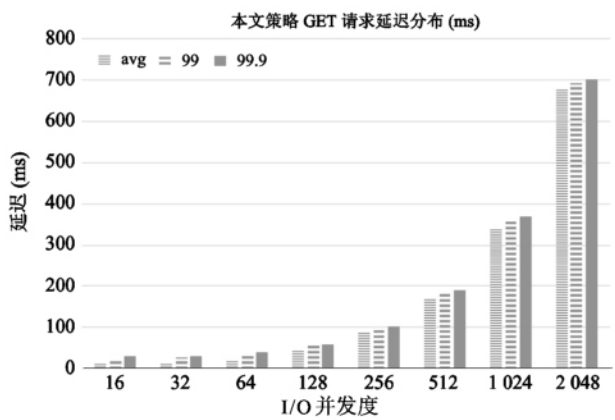

图 9　本文策略的 GET 请求延迟分布

5.3　数据相关性影响

本小节对数据相关性所带来的影响进行分析。由于其他多副本调度策略仅适用于单一的数据一致性级别,其各自效果不受数据相关性的影响,本节仅针对本文策略的表现进行了展示。实验表明,本文策略可以有效利用受到数据相关性影响的 I/O 调度决策空间,避免相关性产生类似于 MAIN 策略的巨大性能限制。

与 5.2 节负载不同,本小节使用读写比例为 7:3 的负载模式,并区分为无数据相关以及有数据相关的模式。由图 6 可知,当 GET 请求并发度达到 64 时,基本上逼近了所布署系统的吞吐量上限,因此本节测试设置 I/O 并发度为 64,读写比例侧重读多写少场景(读写比例为 70% ~100%)。例如,在读写比例为 7:3 的场景下,其中读并发度为 45($\approx 64 \times 0.7$),写并发度为 19($=64-45$)。

如图 10 所示,受读写比例逐步提高的影响,GET 请求并发度逐步下降,相应地,在数据相关性的影响下,系统吞吐量出现下降。在 70% GET 请求负载的场景下,GET 请求吞吐率下降比例达到最大,产生了 7.2% 的性能下降。相较于强一致性的 MAIN 策略,其对系统性能的约束远强于本文策略。因此,本文策略在克服数据相关性影响上取得了良好效果。

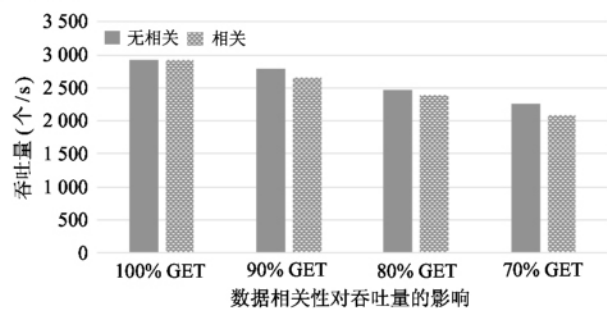

图 10　相关性对 GET 吞吐率的影响

6　相关工作

多副本调度是分布式系统中负载均衡研究的重要话题。C3[5]利用 I/O 响应捎带的存储节点待处理 I/O 队列信息在客户端建模存储节点的负载分布。CFA[8]通过提取广域网的网络特征,利用相似特征匹配的方式预测不同副本的负载状况,以实现

实时的副本选择。Pytheas[9] 通过强化学习方法自主地学习副本上负载的分布状况,继而执行实时的副本选择。这些方法代表了多副本选择时的不同模型,但它们都默认在最终一致性模型下工作,未涉及强一致性的讨论。

在多副本系统中提供多种兼容的一致性具有重要意义。文献[11]中讨论了用户期望的数据一致性模型与系统提供的一致性模型之间的差距,并指出降低一致性级别常常并未完全收获预期的收益。文献[12]设计实现了一个新型的多副本事务范式,以允许用户针对数据而非事务层面制定一致性级别。文献[13]实现了一个可以提供自适应的一致性保障的多副本系统原型。以上工作均未就强一致性的保障提供性能上的优化。

7 结 论

针对保障强一致性的分布式对象存储系统其 I/O 并行性受到已有 I/O 调度算法的限制,本文设计实现了在强一致性要求下优化 I/O 并行性的调度策略。其通过在架构设计上结合多副本主从模型的特点,简化了调度过程中的一致性确定、请求定序、协作通信等过程。其负载合并器将单位时间窗口内 I/O 请求的一致性需求进行确定、简化,提交至数据相关性检查器。数据相关性检查器记录并确定 I/O 请求的执行优先级,转交至负载均衡器。负载均衡器根据副本集合节点的性能模型将 I/O 请求分散执行,并使其满足数据相关性要求。利用原型系统,我们通过实验证明本文策略可以在保障强一致性的同时优化 I/O 并行性,提高系统资源利用率并取得高度可预测的存储服务质量。其中,负载合并可有效提高存储资源利用率,使得系统 GET 吞吐量最大提升 6.4%;较常见一致性保障策略(主副本调度策略),本文策略可使得 GET 请求吞吐量最大提升 41.8%, GET 请求平均延迟最大降低 42.5%,GET 请求 99.9th延迟最大降低 15.8 倍,使系统性能达到最终一致性下基准调度策略 C3 的性能水平;数据相关性导致吞吐量下降幅度不超过 7.2%。

参考文献

[1] Yao Z. How google cloud storage offers strongly consistent object listing thanks to Spanner [EB/OL]. https://cloudplatform. googleblog. com/2018/02/how-Google-Cloud-Storage-offers-strongly-consistent-object-listing-thanks-to-Spanner. html: Google, 2018

[2] Weil S, Leung A, Brandt S, et al. Rados: a scalable, reliable storage service for petabyte-scale storage clusters [C]// Proceedings of the 2nd International Workshop on Petascale Data Storage, Reno, USA, 2007: 35-44

[3] Introduction to Amazon S3 [EB/OL]. https://docs. aws. amazon. com/AmazonS3/latest/dev/Introduction. html: Amazon Web Service, 2019

[4] Arnold J. Openstack swift: using, administering, and developing for swift object storage [M]. Sebastopol: O'Reilly Media, Inc., 2014

[5] Suresh L, Canini M, Schmid S, et al. C3: Cutting tail latency in cloud data stores via adaptive replica selection [C]// Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, Oakland, USA, 2015: 513-527

[6] Vulimiri A, Godfrey P, Mittal R, et al. Low latency via redundancy [C]// Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies, Santa Barbara, USA, 2013: 283-294

[7] Wendell P, Jiang W, Freedman M, et al. Donar: Decentralized server selection for cloud services [C]// Proceedings of the ACM SIGCOMM 2010 Conference, New Delhi, India, 2010: 231-242

[8] Jiang J, Sekar V, Milner H, et al. CFA: a practical prediction system for video QoE optimization [C]// Proceedings of 13th USENIX Symposium on Networked Systems Design and Implementation, Santa Clara, USA, 2016: 137-150

[9] Jiang J, Sun S, Sekar V, et al. Pytheas: enabling data-driven quality of experience optimization using group-based exploration-exploitation [C]// Proceedings of 14th USENIX Symposium on Networked Systems Design and Implementation, Boston, USA, 2017: 393-406

[10] Ford D, Labelle F, Popovici F, et al. Availability in globally distributed storage systems [C]// Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, Vancouver, Canada, 2010: 61-74

[11] Wada H, Fekete A, Zhao L, et al. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective[C]//Proceedings of the 5th Conference on Innovative Data System Research, California, USA, 2011: 134-143

[12] Kraska T, Hentschel M, Alonso G, et al. Consistency rationing in the cloud: pay only when it matters[J]. *Proceedings of the VLDB Endowment*, 2009, 2(1): 253-264

[13] Lu Y, Lu Y, Jiang H. Adaptive consistency guarantees for large-scale replicated services[C]//Proceedings of 2008 IEEE International Conference on Networking, Architecture, and Storage, Chongqing, China, 2008: 89-96

[14] Corbett J, Dean J, Epstein M, et al. Spanner: Google's globally distributed database[C]//Proceedings of the 10th USENIX Symposium on Operating System Design and Implementation, Hollywood, USA, 2012: 251-264

[15] Weil S, Brandt S, Miller E, et al. Ceph: a scalable, high-performance distributed file system[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, USA, 2006: 307-320

[16] Schwan P. Lustre: building a file system for 1000-node clusters[C]//Proceedings of the 2003 Linux Symposium, Ottawa, Canada, 2003: 380-386

[17] Wang Y, Li C, Li M, et al. HBase storage schemas for massive spatial vector data[J]. *Cluster Computing*, 2017, 20(4): 3657-3666

[18] Laksham A, Malik P. Cassandra: a decentralized structured storage system[J]. *ACM SIGOPS Operating Systems Review*, 2010, 44(2): 35-40

[19] Pradeep K, Huang H. Falcon: scaling IO performance in multi-SSD volumes[C]//Proceedings of the 2017 USENIX Annual Technical Conference, Santa Clara, USA, 2017: 41-53

[20] Zheng Q, Chen H, Wang Y, et al. COSBench: A benchmark tool for cloud object storage services[C]//Proceedings of 2012 International Conference on Cloud Computing, Honolulu, USA, 2012: 998-999

An approach to improve I/O parallelism for strong consistent distributed object storage

Shi Xiao[*,**], Song Yonghao[*], Zheng Xiaohui[*,**], Tang Hongwei[*], Yu Lei[*], Zhao Xiaofang[*]

([*]Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)

([**]University of Chinese Academy of Sciences, Beijing 100049)

Abstract

For distributed object storage which guarantees strong consistency, its I/O parallelism is restricted by current I/O scheduling (primary replica-first) method. This paper presents a simple but effective I/O scheduling method based on master-slave model in replication. It guarantees strong consistency and fully exploits decision space of I/O parallelisms. The three main procedures are: First, all I/Os are sent to master replica for workload merge; Second, all requests are forwarded to data relativity checker to assign appropriate priorities. Third, according to priorities and real-time load status, I/Os are calculated for the final destination for load balancing. A prototype of distributed object storage for evaluation is implemented. Experiments show that, compared with primary replica first, the proposed approach can maximumly increase the GET throughput by 41.8%, maximumly decrease average GET latency by 42.5%, and maximumly decrease 99.9th percentile GET latency by 15.8×, similar to the state-of-the-art C3 which cannot provide strong consistency.

Key words: distributed object storage, I/O parallelism, replica selection, strong consistency