

生产实习报告

**专业班级：**计算机科学与技术1506班

**学 号：** U201514631

**姓 名：** 王 晶

**指导教师：** 肖 亮

**实习单位：** 华中科技大学

**实习时间：** 2018.07.09-2018.08.10

**计算机科学与技术学院**

目录

[1 实习概要 1](#_Toc523131688)

[1.1 实习目的 1](#_Toc523131689)

[1.2 实习内容 1](#_Toc523131690)

[1.2.1 调查和记录 1](#_Toc523131691)

[1.2.2 生产实践 2](#_Toc523131692)

[1.3 实习单位及实习岗位简介 2](#_Toc523131693)

[1.3.1 实习单位 2](#_Toc523131694)

[1.3.2 实习指导老师简介 4](#_Toc523131695)

[1.3.2 实习岗位 4](#_Toc523131696)

[1.3.3 实习计划 4](#_Toc523131697)

[2 实习报告 6](#_Toc523131698)

[2.1 国内外发展概述 6](#_Toc523131699)

[2.2 基本原理分析 6](#_Toc523131700)

[2.2.1 HPC应用的I/O特征 6](#_Toc523131701)

[2.2.2 Burst Buffer 7](#_Toc523131702)

[2.2.3 Burst Buffer File System 9](#_Toc523131703)

[2.3 实习内容 15](#_Toc523131704)

[2.3.1 阅读相关文献 15](#_Toc523131705)

[2.3.2 BurstFS环境搭建 17](#_Toc523131706)

[2.3.3 UnifyCR代码学习 20](#_Toc523131707)

[2.4 实习工作总结 35](#_Toc523131708)

[2.5 实习建议 36](#_Toc523131709)

[3 实习问题解答 37](#_Toc523131710)

[参考文献 40](#_Toc523131711)

# 1 实习概要

## 1.1 实习目的

生产实习是高等工科院校教学计划中实践教学的重要环节之一，是理论与实际相结合的有效方式。

通过生产实习环节，学生可以利用所学的理论知识去分析实习中所看到的实际生产技术，从而将科学的理论知识加以验证、深化、巩固和充实。生产实习还能培养学生进行调查、研究、分析和解决实际问题的能力，扩展学生的知识面，既体会到学习书本知识的必要性，又提高解决实际工程问题的能力。此外，生产实习也是学生接触社会、了解社会的重要途径。

为后续专业课程的学习、课程设计和毕业设计打下坚实的基础。通过生产实习，还可以拓宽学生的知识面，增加感性认识，把所学知识条理化、系统化，学习一些从书本上学不到的专业知识，并获得本专业国内、国外科技发展现状的最新信息，激发学生向实践学习和探索的积极性，为今后的学习和将从事的技术工作打下坚实的基础。

培养学生工程技术素养、实现理论联系实际的重要措施。通过生产实习，是学生把所学的理论知识与生产实际相结合，重点培养学生分析问题和解决实际问题的能力。

## 1.2 实习内容

生产实习内容包括两个方面：

1. 调查和记录
2. 生产实践

### 1.2.1 调查和记录

1. 调查和记载实习单位的管理方式和经营情况，包括：

· 单位的组织结构

· 技术部门结构及人员配备情况

· 生产的产品或销售规模、产值、产量及新产品的研制情况

· 实习单位的各种规章制度，包括规章制度名称和重要规定

2. 记录产品研制、开发的方式及过程，包括：

· 计算机硬件、软件的组成、结构和性能、测试机维修等情况，用学过的表示方法加以描述、绘制和记载

· 记载产品生产和装配的工序、工艺及各个环节上的重要处理技术

· 产品开发或研制的技术特点及技术性能的先进性

3. 收集整理产品开发的技术资料，包括：

· 硬部件的线路图及技术参数

· 各种软件设计文档

· 产品性能指标及参数

· 产品使用说明书

以上内容应严格按照生产实习单位的工作要求有选择地记载，具体内容可包括各种资料的类型、格式、图样及全部或部分内容，主义生产实习单位保密工作要求，所记载内容均写在“生产实习日记”中。

### 1.2.2 生产实践

根据所在实习单位的具体条件，完成下列一至多项工作并作详细记载：

1. 了解计算机、计算机网络和计算机新技术的实际应用情况

2. 熟悉或参加某个软件项目的开发工作，包括：软件开发方法、软件开发工具、软件项目管理及文档资料整理工作。

3. 参与系统总体调试工作

4. 掌握软件开发工具及多媒体软件的应用

5. 了解计算机网络的体系结构，高层功能分化以及新技术的应用。

## 1.3 实习单位及实习岗位简介

### 1.3.1 实习单位

华中科技大学信息存储与数字媒体课题组，隶属于华中科技大学武汉光电国家实验室信息存储研究部。

华中科技大学信息存储实验室，源于1974年华中工学院（现华中科技大学）创办的计算机外部设备专业，2000年被批准为信息存储系统教育部重点实验室，2005年底实验室整体进入武汉光电国家实验室、2011年成为光电信息存储与显示功能实验室，2006年被批准为数据存储系统与技术教育部工程研究中心。该实验室有50余名教师，其中长江讲座教授2名，长江特聘教授2名，国家杰出青年基金获得者1名，万人计划“中青年科技创新领军人才”1人，青年千人2 名，楚天学者2名，教育部“新世纪优秀人才支持计划”获得者2名，此外，实验室还有300余名博士和硕士研究生。实验室团队于2008年获批教育部“长江学者和创新团队发展计划”“信息存储系统与技术”创新团队，并因评估为优秀，于2015年获得滚动支持。是中国在信息存储领域研究人员最多、水平一流研究单位和重要的人才培养基地。

本实验室成员的专业方向主要分布于计算机系统结构、微电子、光学工程属多学科交叉。所在的计算机系统结构学科是国家重点学科，并是国家“211”工程和“985”工程重点建设学科。

实验室的主要学术方向及研究内容有：

（1）并行存储系统，研究高可用、高性能、大容量、自组织的并行存储技术、对象存储系统、分布式存储技术等；

（2）云存储及其服务保障技术，包括存储虚拟化、数据去重技术、数据长程保存技术等；

（3）超高密度存储技术，研究采用新原理的下一代存储技术，大幅度提高存储密度和速度。主要有近场光存储、瓦式记录磁盘技术和光磁混合存储技术；

（4）存储安全与高可用存储系统，研究数据备份、灾难恢复、数据保护技术；

（5）新型非易失主存技术，研究闪存、相变存储、忆阻器等新存储器件特性及相应的体系结构，数据组织、持久化内存文件系统等；

（6）固态存储器技术，研究内容包括相变存储器和闪存融合的固态存储技术，高速固态盘技术等；

（7）非易失存储技术研究，从原理、材料、器件工艺、集成、测试等方面开展忆阻器、相变存储器等新型存储器技术的研究，解决存储器中的能耗、速度、可靠性等一系列关键的科学问题。

近年来，实验室承担了包括973、863、国家自然科学重点基金、面上基金、国防预研项目等40多项国家级项目，在科研、研究生培养和国内外学术交流方面取得了一系列重要的成果。成果获得过多项国家级和省部级奖，并在国际存储挑战竞赛中获得决赛奖。在国际和国内学术期刊和会议上发表论文500余篇，获得发明专利200余项。

实验室具有良好的研究环境和实验条件，建有PB级网络存储试验平台和齐全的信息存储试验环境，拥有国际先进水平的测试仪器和开发工具。

实验室瞄准世界一流水平，争取在信息存储科学和技术领域取得重大成果，并利用自主知识产权核心技术服务于我国存储工业，以期成为我国信息存储工业的技术源头。

### 1.3.2 实习指导老师简介

万继光，男，教授，博士生导师，工学博士，1972年7月出生，在美国中佛罗里达大学访问学习1年。研究方向为新型存储系统软件研究，主要是基于Linux平台的存储系统软件开发，包括：分布式存储系统软件，KV键值存储系统，面向新型存储器件的文件系统，面向云计算虚拟机的分布式块存储系统，并行存储系统。

### 1.3.2 实习岗位

Burst Buffer分布式文件系统组。该组由一名博士学长徐鹏带领，4名实习生共同参与。个人负责Burst Buffer分布式文件系统的原理与代码学习。

### 1.3.3 实习计划

（1）第一阶段：基础知识学习

阅读、学习关于应用于高性能计算的Burst Buffer以及运用在Burst Buffer上的文件系统BurstFS的相关文献，理解BurstFS的关键设计，完成论文阅读文档。

（2）第二阶段：安装配置BurstFS

从Github官网上下载UnifyCR（BurstFS）的源代码，编译安装，并成功运行测试程序。

（3）第三阶段：阅读理解代码

阅读UnifyCR源代码，学习、理解UnifyCR运行的方式与流程、关键部分的设计，完成源码阅读文档。

（4）第四阶段：探讨交流

实习生交流代码阅读的收获与心得，探讨UnifyCR的实现原理。

# 2 实习报告

## 2.1 国内外发展概述

1. Burst Buffer

目前在高性能计算环境中有两种代表性的burst buffer体系结构：本地节点burst buffer和远程共享burst buffer。在本地节点burst buffer体系结构中，burst buffer存储位于单个计算节点上，因此聚合burst buffer带宽随计算节点数量线性增长。这种可伸缩性的好处在最近的文献中得到了很好的证明。它还解决了可扩展的元数据管理策略来维持分布在所有burst buffer节点上的数据的全局命名空间的需求。在远程共享burst buffer架构中，burst buffer存储位于计算节点和后端存储系统之间的少量I/O节点上。计算节点和burst buffer节点之间的数据移动需要通过网络。在I/O节点上设置burst buffer有利于burst buffer服务的独立开发、部署和维护。因此，目前已经有几款知名的商业化软件来管理这种类型的burst buffer，如DataWarp和Infinite Memory Engine。

由于burst buffer的重要性，burst buffer已被广泛地应用于顶级超级计算机。例如，圣地亚哥超级计算机中心的DASH、东京工业大学的Tsubame、美国阿贡国家实验室的Theta和Aurora、美国能源部下属橡树岭国家实验室的Summit、和劳伦斯利弗莫尔国家实验室的Sierra等超级计算机安装了本地节点burst buffer。广州超级计算中心的天河2号、洛斯阿拉莫斯国家实验室的Trinity、劳伦斯伯克利国家实验室的Cori等超级计算机安装了远程共享burst buffer。

## 2.2 基本原理分析

### 2.2.1 HPC应用的I/O特征

HPC中，科学应用在计算阶段的IO操作很少，而在每个计算片段结束时，集中地IO，更新分析文件和CP文件。各种各样的应用同时运行时，不同应用的IO阶段交叠，导致了很高的IO峰值。

因此，HPC应用的I/O特征之一是周期性的浪涌式I/O。

以OLCF Spider存储系统为例，其24小时的I/O（写）记录如图2.1所示。

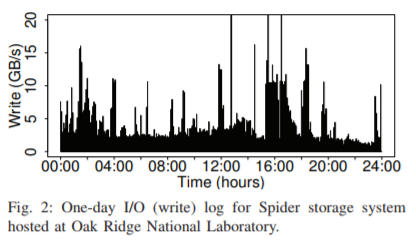


图2.1 Spider存储系统24小时I/O（写）记录

如图，大部分时间，写吞吐量低于10GB/s，某些时间写吞吐量超过10GB/s甚至超过20GB/s。I/O负载比存储系统能提供的性能指标低很多。

以运行在超级计算机Titan上的科学应用Community Earth System Model为例，每个月保存一次气候数据到永久存储系统中，每3个月进行一次记录CheckPoint。其I/O记录如图2.2所示。

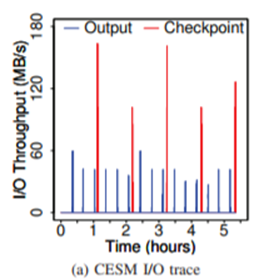


图2.2 CESM I/O记录

### 2.2.2 Burst Buffer

Burst Buffer的出现是为了满足高性能计算中对更好的I/O性能的要求。采用Burst Buffer，可以改进两个场景和组件的I/O性能。1、应用程序可用的总带宽。带宽越高，优化良好的应用程序可以更快地读写大量数据。2、文件系统的IOPS（Input/Output Operations per Sencond）。许多应用程序发起大量小I/O操作，在这种情况下，IOPS成为了性能的限制因素之一。

改进I/O性能能在多方面帮助科学应用：

1. 提高应用程序的可靠性（通过更快的checkpoint-restart）。
2. 加速应用程序的小块传输和文件分析的I/O。
3. 为核心外部应用程序提供快速临时空间。
4. 为需要大量输入文件的作业创建一个中间区域或耦合模拟作业之间的持久快速存储。
5. 大型仿真数据的后处理分析。
6. 传输途中的数据可视化和分析。

以安装在Cori上的Burst Buffer为例，图2.3说明了Burst Buffer的概念结构。Burst Buffer在物理位置上是处于计算和存储节点之间的一层Flash层，在Cray DataWarp中它驻留专门的XC40节点上，它是I/O计算节点系统（采用Aries互联）和存储Fabric的桥梁。SSD安装在Burst Buffer节点中，通过Scheduler和DataWarp软件堆栈来支持HPC计算作业。

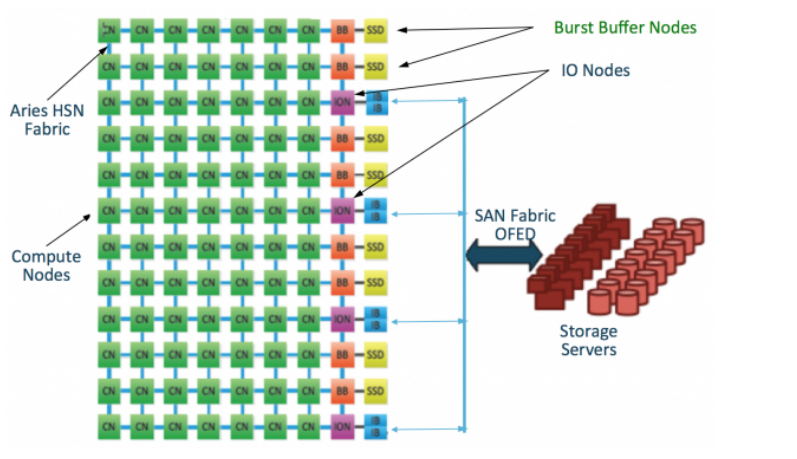


图2.3 Cori中Burst Buffer架构

用于DataWarp的闪存依附在Burst Buffer节点。两个Burst Buffer节点在一个刀片上。每个Burst Buffer节点包含：1个Xeon 64GB DDR3内存处理器、2个3.2TB NAND闪存SSD模块（连接2个PCIe gen3 x8接口）。每个Burst Buffer节点通过PCIe gen3 x16接口与Cray Aries网络互联。每个Burst Buffer能提供连续读写峰值约6.5GB/s。其结构图如图2.4所示。

Burst Buffer利用闪存及SSD技术以及运用其上的软件技术，其速度介于DRAM和HDD之间，容量介于HDD和DRAM之间，用以吸收超级计算机上科学应用程序产生的浪涌式I/O，有效缓解了底层存储系统带宽不足在高性能计算上的瓶颈问题。

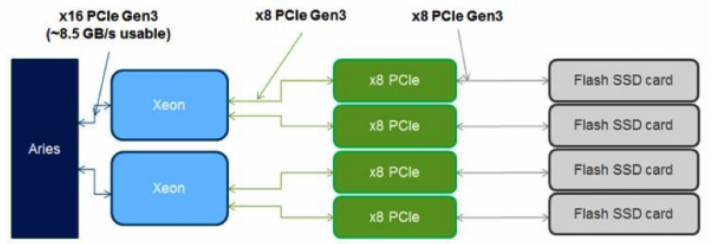


图2.4 单个Burst Buffer节点结构

### 2.2.3 Burst Buffer File System

Burst buffers已成为大规模超级计算机上不可或缺的硬件资源，用于缓存科学应用中突发性的I/O。但是，burst buffers在批量提交的作业和跨不同批次作业的应用中高效共享，缺乏软件支撑。此外，burst buffers需要解决密集型数据科学应用中许多挑战性的I/O模式。

Burst Buffer File System（BurstFS）是由LLNL（劳伦斯利佛摩国家实验室）提出的一个临时性的文件系统，支持可扩展和高效的I/O带宽聚合，且与批量作业具有相同的生命周期。

LLNL基于BurstFS正在开发一个用户级文件系统UnifyCR。

#### 1. 背景知识

在目前的HPC系统中，checkpointing达到了总I/O量的75%~80%。且普遍认为checkpoint的数据大小将增加，checkpoints之间的时间间隔将减小。大文件和短间隔的checkpointing需要更快的存储带宽。

如图2.5，N-N I/O，每个进程读写一个独立的文件。

在N-1 I/O，所有进程读或写一个共享的文件，N-1 I/O可以进一步分类成两种模式：N-1 segmented和N-1 strided。

N-1 segmented I/O：每个进程访问一个不重叠、连续的文件区域。

N-1 strided I/O：进程之间相互重叠它们的I/O。

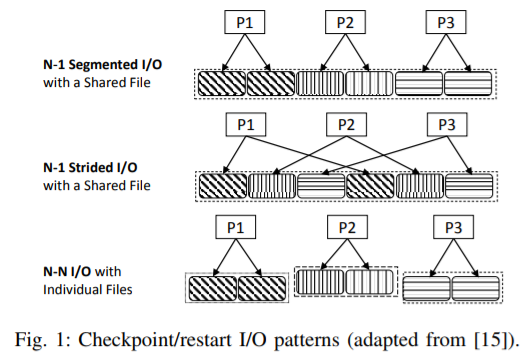


图2.5 Checkpoint/restart I/O模式

HPC中科学应用的另一个I/O特点是数据往往是多维的。科学应用中，多维变量以一特定次序写入，但每次用到的只是数据的一部分，常以不同的次序读、分析、可视化。

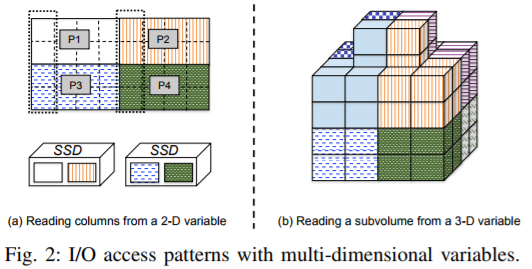


图2.6 多维数据的I/O

如图2.6所示，一个进程可能需要变量的其中一部分，但存储在不连续的块中。因此该进程需要发起多个较小的读请求。

#### 2. BurstFS

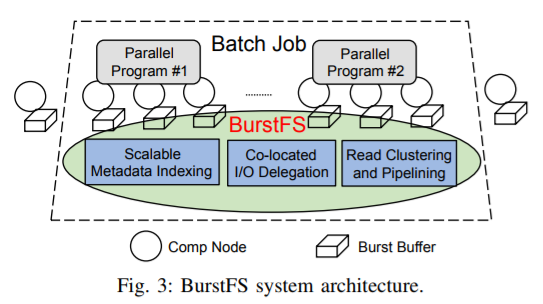


图2.7 BurstFS系统架构

BurstFS是一个临时的文件系统，与HPC作业具有相同的生命周期。设计目标是支持数据密集型模拟、分析、可视化、checkpoint/重启的，分布式的、node-local存储上的可扩展I/O操作的聚合。

BurstFS实例在一批作业开始时启动，为作业中的所有应用提供数据服务，并在作业结束时终结。如Fig.3为BurstFS系统架构。

所有分配的资源和节点会在作业执行结束后清理，以便重新使用，避免了结束的资源对其他作业造成干扰或对文件和存储系统的操作造成不可预见的问题。此外，同一批作业中的并行程序可以共享数据，大大减少了对后端持久文件系统的需求。

BurstFS将元数据组织到分布式KV存储中，它支持可扩展性元数据索引从而可以快速生成数据的全局视图，以便于快速读取操作。它也提供了一种延迟同步方案来降低元数据更新的成本和频率。此外，BurstFS还支持可扩展、可回收的I/O管理。此外，还引入了一种server-side读聚集和流水线机制，来提高读性能。

A. 可扩展元数据索引

N-1 I/O模式的一个难题是访问分散在所有节点上的元数据。这导致每个进程都需要从所有节点收集元数据。

1）元数据的分布式K-V存储：通过分布式K-V store和数据分段的日志写。它利用MDHIM来构建分布式K-V存储，也提供额外的有效处理突发读写操作的特征。

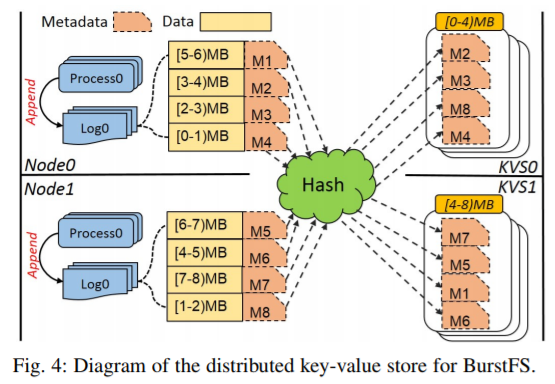


图2.8 BurstFS的分布式key-value存储

图2.8展现了BurstFS数据和元数据的组织。每个进程将数据存到本地的burst buffer，作为data logs，并被组织成数据段，新数据总是追加到data logs（如通过日志写的方式）。通过日志写，一个进程的所有片段被存到一起，而不管他们相对于其他进程数据的全局逻辑位置。

当并行程序的进程创建一个全局共享文件时，会为每个segment创建一个KV对（如M1、M2）。

Key: 文件ID（8字节hash值）和共享文件中segment的逻辑偏移量组成。

Value: 描述了该segment的真实位置，包括hosting burst buffer、包含该segment的log（多进程在一个节点上不止一个log）、log上的物理偏移和长度。

所有segments的KV对（KVP）提供共享文件的全局layout。并行应用中的多个进程能快速检索元数据并形成全局layout的视图。

2）Lazy Synchronization：每个进程提供一个小内存池，来存放写操作的元数据KVPs，并且，在可配置的间隔结束时，KVPs被定期存储到分布式K-V存储中。

BurstFS利用MDHIM中的批处理，将这些KVPs在几个round-trips中一起传输，来最小化单个put操作的延迟。内存池中连续的KVPs，且映射到同一个server，会合并成一个KVP。当每个进程发出的多个数据segments逻辑上连续时，Lazy synchronization能明显减少所需KVPs的数量。（如N-1 segmented和N-N写）

3）Parallel Range Queries：要开始读操作之前，BurstFS先查找分布式数据segments的元数据。要为不同的读操作检索请求的元数据条目，需要支持大量对KV存储的范围查询（range queries）。但是MDHIM不直接支持范围查询。

为MDHIM客户端与服务器端提供并行扩展。客户端，将传入的范围请求转换成多个小范围请求，并送到hash一致的服务器端。这些小请求被并行地送到所有范围内的服务器去检索所有的KVPs。服务器端，对于小范围查询，所有在范围内的KVPs使用顺序扫描检索。

B. 共置I/O委托

BurstFS读操作需要从远程burst buffers传输到一个启动读取的进程。在BurstFS中，使用一个通过co-located I/O delegation的可扩展的读服务。在每个节点中启动一个I/O代理委托进程。委托为作业中的所有应用提供数据服务。

共置I/O委托示意图如图2.9，该示意图中有三个节点，每个节点上运行两个进程。

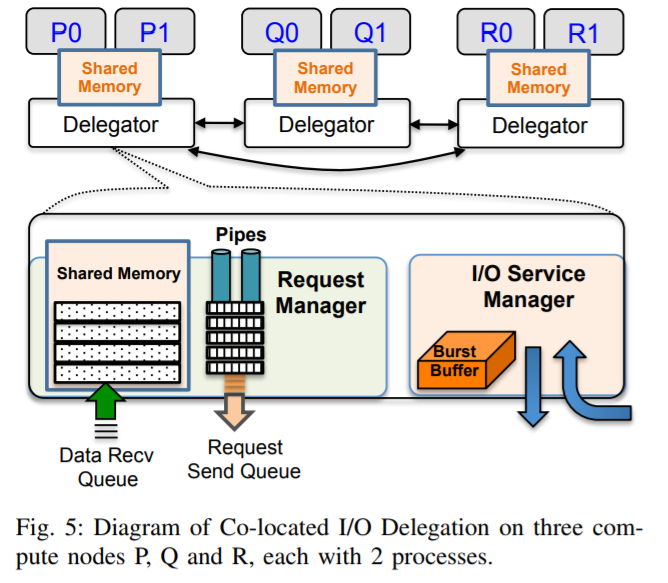


图2.9 共置I/O委托示意图

每个委托有两个主要部件：request manager和I/O service manager。在这种方式下，I/O服务的传统CS模型转换成了所有委托间的p2p模型。单个进程不再直接与I/O服务器直接通信，而是通过I/O代理，从而大大减少了网络通信的通道和相关的资源。

I/O service manager专门用来处理从对等方peer来的读请求。I/O service managers寻找合并请求的机会，从本地委托进行数据检索，并将数据传回请求的委托。

Request manager有两个主要数据结构组成：request send queue和data receive queue。

Request send queue是一个入口数可配置的循环列，当队列没满时，它接收所有客户进程通过命名管道的读请求。

Data receive queue驻留在委托和客户进程同一节点的共享内存中。对于每个I/O请求，在receive queue中创建一个未完成的请求条目。从远程委托返回的数据，直接存到共享内存池中，receive queue中查找匹配的未完成的条目，找到匹配项时，将未完成标记为已完成。再通过管道向客户进程发送确认以使用数据。

Request manager监控共享内存池的使用级别。当超出了阈值（默认75%），委托通知进程尽快使用他们的数据，并向远程委托限制请求量。Request manager还监控发送队列中基于接收数据的入口带宽，当流进的带宽饱和时，request manager为发送请求和接收数据创建额外的网络通信通道。

C. 服务端的读请求聚集和流水线

问题：一个进程会发起许多小的、不连续的读请求。

已有的库或工具（POSIX lio\_listio、OrangeFS），虽然能合并小请求，但这些技术主要在客户端实现，并且依靠底层存储系统，如disk scheduler。缺少分布式文件系统能够全局优化这些批量读请求。

在I/O service manager引入SSCP（server-side read clustering and pipelining），以提高BurstFS的读性能。

SSCP要解决同时、有时冲突的目标：（1）在读请求中检测空间局部性并合并成大的连续读；（2）尽快执行on-demand读请求。

如图2.10，SSCP提供了两个关键部分来实现这些目标：2级读请求队列、3级快速移动数据流水线。

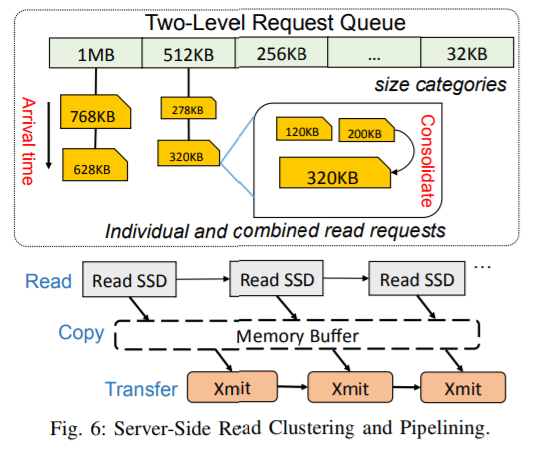


图2.10 服务端读聚集与流水线

在two-level request queue中，SSCP先创建几个请求大小的分类，覆盖从32KB~1MB。到达的请求会插入到合适的分类中，当和其他请求连续时，则合并。以到达时间来排队，合并的请求用最初成员的到达时间。为了最好的调度效率，优先服务size最大的读请求。BurstFS强制每个类别的等待时间（默认5ms）。若有类别超过该阈值等待时间没有被服务，则选取其中最早的读请求服务，并重置时间。

I/O service manager创建一个临时缓冲发出数据的内存池。这有利于网络传输的数据segments的重新组织，也支持了流水线的建立。如Fig 6，3级流水线：reading、copying、transferring。Reading stage：I/O service manager从request list中挑选请求，从本地burst buffer读取数据到memory buffer。Copying stage：准备memory buffer中的数据，拷贝到network packet中，memory buffer中的数据可能需要被分成发往不同远程委托的多个回应包。Transferring stage：I/O service manager打包一个或多个发往同一个远程委托的network replies成一个network message（最大1MB），再发送出去。

## 2.3 实习内容

### 2.3.1 阅读相关文献

阅读与Burst Buffer和Burst Buffer File System相关的论文与资料，学习什么是Burst Buffer、Burst Buffer的作用、Burst Buffer File System的设计架构等内容。

相关文献包含但不限于：

“Burst Buffer Architecture and Software Roadmap”；

《Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior》；

《Accelerating Science with the NERSC Burst Buffer Early User Program》；

《BurstFS: An Ephemeral Burst Buffer File System for Scientific Applications》；

《BurstMem: A High-Performance Burst Buffer System for Scientific Applications》；

《A 1 PB/s File System to Checkpoint Three Million MPI Tasks》

其中Burst Buffer与Burst Buffer File System基本原理于2.2节中阐述，此处不再赘述。

《Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior》介绍了一种应用于Burst Buffer的有效的刷写策略。高性能计算在应用中通常发生浪涌型I/O。为了加快应用的速度，通常会提供永久性的存储系统来处理这些突发IO。非易失性RAM被用来作为Burst Buffer，来吸收这些大量突发数据，并减少对永久性存储系统的IO需求。但是，在没有对Burst Buffer合理的刷写情况下，突发IO被传到底层存储系统，将导致严重的IO争用问题。

为了最大限度降低IO供应需求、解决IO突发问题，提供了一个主动draining 调度策略来管理分布式节点-本地突发缓存的刷写过程。此外，还开发了一个IO供应模型来预测对永久存储系统的最小IO供应需求。评估结果表明，应用proactive draining scheme在保证底层存储系统的IO性能的同时，很大程度的减少了对IO供应需求。

《Accelerating Science with the NERSC Burst Buffer Early User Program》介绍了NERSC用于Burst Buffer的早期用户程序。NERSC拥有一个由700多个不同项目的6500多个用户组成的多样化群体，涵盖了多种科学计算应用。因此，NERSC的Burst Buffer用例也相当广泛、多样。论文介绍了从NERSC的Burst Buffer早期用户程序获得的性能测试和经验教训，该程序选择了一些研究项目来获得对Burst Buffer的早期访问，并利用它的能力来实现新的科学进步。这是Burst Buffer第一次被强调在多样化、真实的用户工作负载下，因此这些经验将对改进Burst Buffer在HPC中心的应用有相当大的好处。

《BurstMem: A High-Performance Burst Buffer System for Scientific Applications》介绍了一个高性能突发缓冲系统BurstMem的设计。尽管burst buffer有很多优势，但现在的研究大多集中在建模和仿真上，少有设计与实现的文档。

本论文系统地设计了一个burst buffer系统，名为BurstMem。BurstMem提供了一个具有高效存储和通信管理策略的存储框架。它提供了一个简单的接口，允许应用程序快速转储checkpoint数据，并异步地将数据刷写到PFS，而不干扰应用的计算。通过一组存储管理策略，它有效地利用了存储设备的容量和带宽，并减少了总的I/O时间。此外，BurstMem还设计了一种新的基于数的索引技术，可以支持两阶段的快速数据刷写。最后，BurstMem实现了可移植的快速通信层，使其能移植到不同网络配置的系统上。

在尖端缓存系统Memcached上改造、扩展功能实现BurstMem。BurstMem是一个轻量级的、分布式的、基于DRAM的缓存系统。尽管不是为了科学应用设计的，但它包含了分布式缓存管理的所有特征。它对于复杂应用有快速的存储解决方案、良好的扩展性。修改了Memcached的数据放置策略、通信层和存储管理模块。此外，还设计了协调数据洗牌和刷写到PFS的机制。通过实验测试，BurstMem能够使运行在顶尖计算机系统上的科学应用I/O性能加速达8.5倍。

《A 1 PB/s File System to Checkpoint Three Million MPI Tasks》介绍了一个用户空间基于主存的文件系统的架构与设计思路。其探讨了多种设计方案以及它们各自的优劣，如系统I/O拦截的方案，一种方案是使用FUSE（Filesystem in User Space）模块，但这种方案的缺点是不适用于HPC（一些HPC不用Linux，或不加载某些内核模块），对于checkping工作流的I/O性能较差。另一种方案是使用一些wrapper函数封装POSIX标准I/O函数，这种方案能在程序链接时完成，所有控制在用户空间完成，开销较小，但缺点是实现的代码工作量大。又如在主存中文件存储的方案等。

### 2.3.2 BurstFS环境搭建

在64位CentOS 7上安装burst buffer文件系统“UnifyCR”。UnifyCR是LLNL（劳伦斯利弗莫尔国家实验室）正在开发中、基于BurstFS一个用户级文件系统。

UnifyCR需要的前提环境包括MPI、LevelDB、GOTCHA。

1. 安装MPICH

1.1 下载MPICH 3.2.1源码

http://www.mpich.org/downloads/

1.2 编译安装

1）指定MPICH安装路径为“/home/[username]/mpich-install”

$ mkdir /home/wj/mpich-install

$ ./configure --prefix=/home/wj/mpich-install 2>&1 | tee c.txt

2）编译

$ make 2>&1 | tee m.txt

3）安装

$ make install 2>&1 | tee mi.txt

4）设置环境变量

修改~/.bashrc文件，在末尾添加：

PATH=/home/wj/mpich-install/bin:$PATH;

export PATH

使用命令：source .bashrc使之生效。

5）检测

通过which指令检测环境是否配置成功，或使用mpich-3.2.1/examples目录下的样例程序。

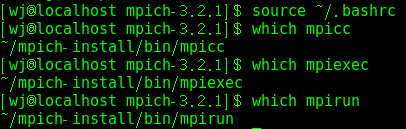


图2.11 MPI安装检测

1. 安装SPACK

2.1 从github下载spack

$ git clone https://github.com/spack/spack

2.2 通过spack下载安装leveldb

$ ./spack/bin/spack install leveldb

2.3 通过spack下载安装gotcha 0.0.2版本

$ ./spack/bin/spack install gotcha@0.0.2%gcc

此处若默认安装最新版本（1.0.2）会导致后续UnifyCR运行失败

2.4 通过spack下载安装environment-modules

$ ./spack/bin/spack install environment-modules

2.5 在终端加载leveldb、gotcha

$ spack load leveldb

$ spack load gotcha

1. 安装UnifyCR

3.1 从github下载UnifyCR

$ git clone https://github.com/LLNL/UnifyCR.git

3.2 安装openssl

$ yum install openssl-devel

3.3 生成配置文件

$ sh autogen.sh

3.4 运行配置文件

$ ./configure

3.5 编译

$ make

3.6 编译测试检查

$ make distcheck

由于各种原因，可能前面某步没有正确进行，均会导致UnifyCR安装出现问题，即使安装完成，也无法正确运行。此步可以检查编译是否正确。编译正确会出现图2.12结果，11项内容全部正确通过。

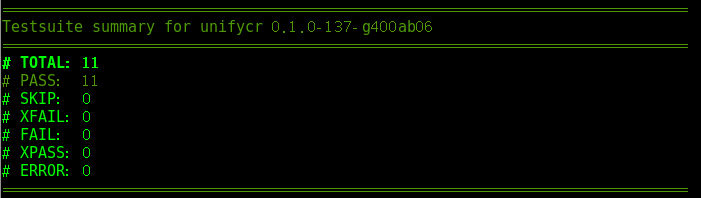


图2.12 编译检查

没有发生错误会生成发布软件包，如图2.13所示。



图2.13 生成发布软件包

3.7 安装

# make install

注意，从此步开始需要超级用户权限，故切换到root模式。

1. 测试UnifyCR
   1. 编译测试程序

进入到UnifyCR客户端的测试程序路径：

# cd client/tests

编译MPI程序：

# mpicc -o test\_write `unifycr-config –pre-ld-flags` test\_write.c `unifycr-config –post-ld-flags` -lunifycr\_common

4.2 运行测试程序

需要先解决依赖库的问题，否则运行程序会提示找不到某些链接库。可采取的方式有建立符号软链接、修改etc下的配置文件等。还需要创建运行时路径、配置文件、运行状态文件等。使用mpirun命令运行程序：

$ mpirun ./test\_write

程序运行，但提示错误：unifycr\_mount call failed

查看源码，修改其中的DEBUG条件使其能输出DEBUG信息。得知需要先配置相关参数、启动服务端、挂载文件系统路径等。

4.2.1 修改参数

参数由命令行参数设置，为了简便起见，直接在源码中将参数写好。将UnifyCR的挂载路径设置为“/mnt/ssd”。

4.2.2 为挂载路径添加硬盘

为虚拟机添加一块硬盘，安装文件系统，并挂载到指定路径。

# fdisk /dev/sdb

n

# mkfs -t ext4 /dev/sdb1

# mkdir /mnt/ssd

# mount /dev/sdb1 /mnt/ssd

4.2.3 启动UnifyCR服务端进程

# unifycrd

4.2.4 在终端中添加环境变量

# export UNIFYCR\_SPILLOVER\_DATA\_DIR=/mnt/ssd

# export UNIFYCR\_SPILLOVER\_META\_DIR=/mnt/ssd

4.2.5 运行测试程序

# mpirun -n x ./test\_write

其中x为MPI并行的进程数，x=1时，结果如图2.14所示。



图2.14 test\_write运行结果（并行度=1）

x=2时，结果如图2.15所示。



图2.15 test\_write运行结果（并行度=2）

### 2.3.3 UnifyCR代码学习

UnifyCR是一个用户级文件系统，通过gotcha接管系统I/O调用以及C语言标准输入输出库中的I/O。该文件系统是一个临时性的文件系统，与运行的一批作业具有相同的生命周期，当作业运行结束时，该文件系统也取消挂载。该文件系统的I/O主要在Burst Buffer节点或DRAM上进行。

应用程序可以使用本地节点burst buffer作为共享文件的缓冲区。UnifyCR是为支持检查点/重新启动而设计的，检查点/重新启动是HPC最重要的I/O工作负载，也是其他常见的I/O工作负载。有了UnifyCR，应用程序可以像编写并行文件系统一样轻松地写入快速、可伸缩的节点本地突发缓冲区。

UnifyCR支持来自burst buffer的I/O带宽的可扩展和高效聚合，同时具有与批作业相同的生命周期。它基于可扩展的元数据索引、共置I/O委托、服务端读聚集和流水线，有效地提高了科学I/O速度。

#### 1. UnifyCR基本架构

关于UnifyCR的read、write操作主要涉及到两大模块——client与server。

其中server又分为两个主要部件——request manager、service manager。其基本结构如图2.16所示。图中假设三个计算节点P、Q、R，每个节点上运行了两个客户进程（如P0、P1）。委托Delegator属于服务端。

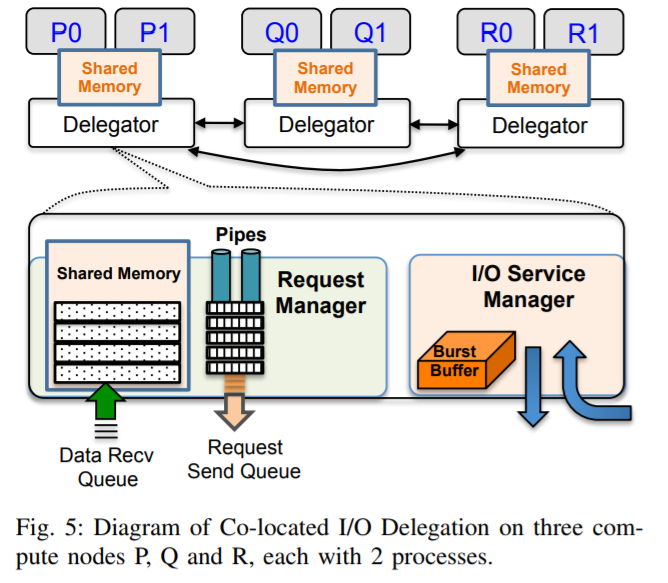


图2.16 UnifyCR基本架构

以上部分的源码目录结构如下：

client/src:

unifycr.c // 客户端挂载UnifyCR及基本文件操作

unifycr-sysio.c // 接管系统调用的I/O函数定义

unifycr-stdio.c // 接管stdio的I/O函数定义

unifycr-fixed.c // 写操作用到的写实现函数

unifycr-internal.h // 客户端用到的部分数据结构定义

server/src:

unifycr\_init.c // server端启动、挂载

unifycr\_service\_manager.c // service manager

unifycr\_request\_manager.c // request manager

unifycr\_sock.c // 与客户端进行socket通信的函数

unifycr\_cmd\_handler.c // 接受客户端连接，处理客户端命令

meta/src:

mdhim.c // mdhim的K-V存、取函数

client/test/:

test\_read.c // 读测试程序

test\_write.c // 写测试程序

#### 2. UnifyCR启动、挂载

Server端：

服务端启动流程如图2.17所示。

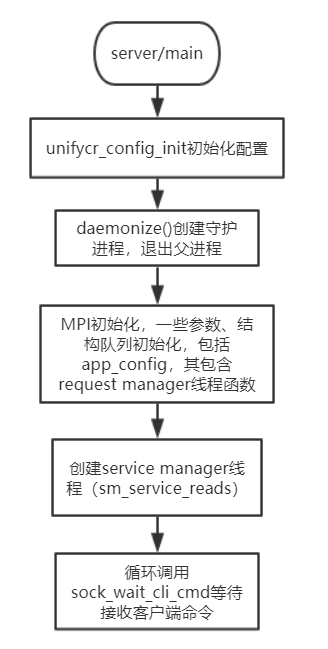


图2.17 server端启动流程

服务端启动时，初始化配置后，调用daemonize()创建守护进程。再通过MPI初始化，创建分布式多节点服务环境。初始化客户端程序队列、服务线程队列、socket等。随后创建服务线程（sm\_service\_reads），初始化MDHIM的K-V存储。之后开始循环接收（sock\_wait\_cli\_cmd）、处理（delegator\_handle\_command）客户端发来的命令。

Client端：

每个客户程序调用unifycr\_mount挂载UnifyCR。

unifycr\_mount初始化配置后调用unifycrfs\_mount。unifycrfs\_mount，初始化socket、初始化与server端的共享文件内存、将挂载路径加到列表中。具体细节见代码分析。

I/O调用接管：

挂载UnifyCR后，应用程序的I/O调用，包括POSIX系统调用和stdio的I/O调用均被UnifyCR接管，使用UnifyCR的函数替代。UnifyCR使用GOTCHA来包装原函数。

使用宏定义来命名包装函数。如“UNIFYCR\_WRAP(open)”将会被替换成“\_\_wrap\_open”，而原函数则使用UNIFYCR\_REAL(name)命名。该宏定义代码如下：

/\* we define our wrapper function as \_\_wrap\_<iofunc> instead of <iofunc> \*/

#define UNIFYCR\_WRAP(name) \_\_wrap\_ ## name

/\* gotcha maps the <iofunc> call to \_\_real\_<iofunc>() \*/

#define UNIFYCR\_REAL(name) \_\_real\_ ## name

#### 3. 数据通信方式

##### a) client与server命令通信

客户端与服务端之间通过socket进行命令的通信。命令包括UnifyCR挂载（COMM\_MOUNT）、元数据（COMM\_META）、读（COMM\_READ）、取消挂载（COMM\_UNMOUNT）等。定义如下：

typedef enum {

COMM\_MOUNT, /\*the list of addrs: appid, size of buffer, offset of data section, metadata section\*/

COMM\_META,

COMM\_READ,

COMM\_UNMOUNT,

COMM\_DIGEST,

COMM\_SYNC\_DEL,

} cmd\_lst\_t;

客户端或服务端通过向socket文件write方式，将命令发送到对方。另一方使用poll函数接收。

##### b) client与server数据通信

client与server数据的通信采用共享内存的方式。数据主要包括读请求、请求的数据。

如客户端的请求数据的接收缓冲区shm\_recvbuf，通过mmap，将文件“%app\_id-recv-%client\_id”映射到内存shm\_recvbuf。其中“%app\_id”代表客户程序号，“client\_id”一般表示客户程序进程的rank号。在客户端的unifycr.c的unifycr\_init\_recv\_shm函数中：

snprintf(shm\_name, sizeof(shm\_name), "%d-recv-%d", app\_id, local\_rank\_idx);

recvbuf\_fd = shm\_open(shm\_name, MMAP\_OPEN\_FLAG, MMAP\_OPEN\_MODE);

…

shm\_recvbuf = mmap(NULL, shm\_recv\_size, PROT\_WRITE | PROT\_READ,

MAP\_SHARED, recvbuf\_fd, 0);

而在服务端的委托通过mmap将每个客户端的接收数据缓冲区文件“%app\_id-recv-%client\_id”映射到app\_config->shm\_recv\_bufs。在unifycr\_cmd\_handler.c中的attach\_to\_shm函数中：

sprintf(shm\_name, "%d-recv-%d", app\_id, client\_side\_id);

tmp\_fd = shm\_open(shm\_name, MMAP\_OPEN\_FLAG, MMAP\_OPEN\_MODE);

……

app\_config->shm\_recv\_bufs[client\_side\_id] =

mmap(NULL, app\_config->recv\_buf\_sz, PROT\_READ | PROT\_WRITE,

MAP\_SHARED, tmp\_fd, SEEK\_SET);

#### 4. UnifyCR读（read）流程

客户程序发起读请求，需要发送到服务端进行处理、读取数据再传回客户程序。其简要流程如图2.18所示。

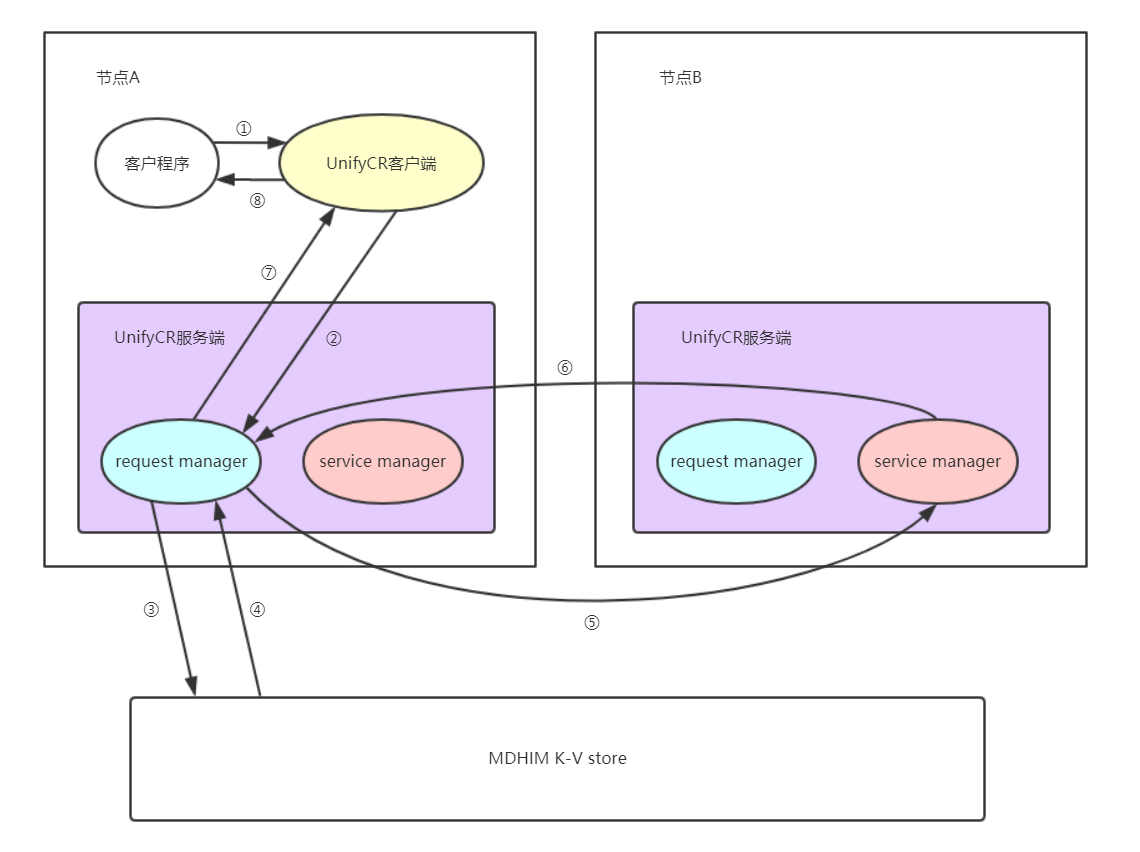


图2.18 UnifyCR简要读流程

各步骤分别为：

1. 客户程序调用read函数，被UnifyCR利用gotcha拦截。
2. UnifyCR客户端将读请求通过共享内存的方式发送给本地节点服务端，并通过socket发送读命令到服务端，由服务端request manager开始处理。
3. request manager向MDHIM分布式K-V存储查询读请求的目标位置节点及其他元数据。
4. MDHIM分布式K-V节点返回查询的value。
5. request manager向目标远程委托（节点A与节点B也可能是同一节点）的service manager通过MPI发送读请求。
6. service manager获取需要的数据，通过MPI将数据发回远程委托的request manager。
7. request manager将数据通过共享内存的方式发送给UnifyCR客户端，并通过socket发送命令让客户端使用数据。
8. UnifyCR客户端返回read操作的结果

更具体的流程图如图2.19所示。图中浅黄色部分为UnifyCR的客户端（非客户程序），蓝色部分为UnifyCR的server端的request manager，红色部分为server端的service manager。该流程省略了许多细节，仅关注重要核心部分。

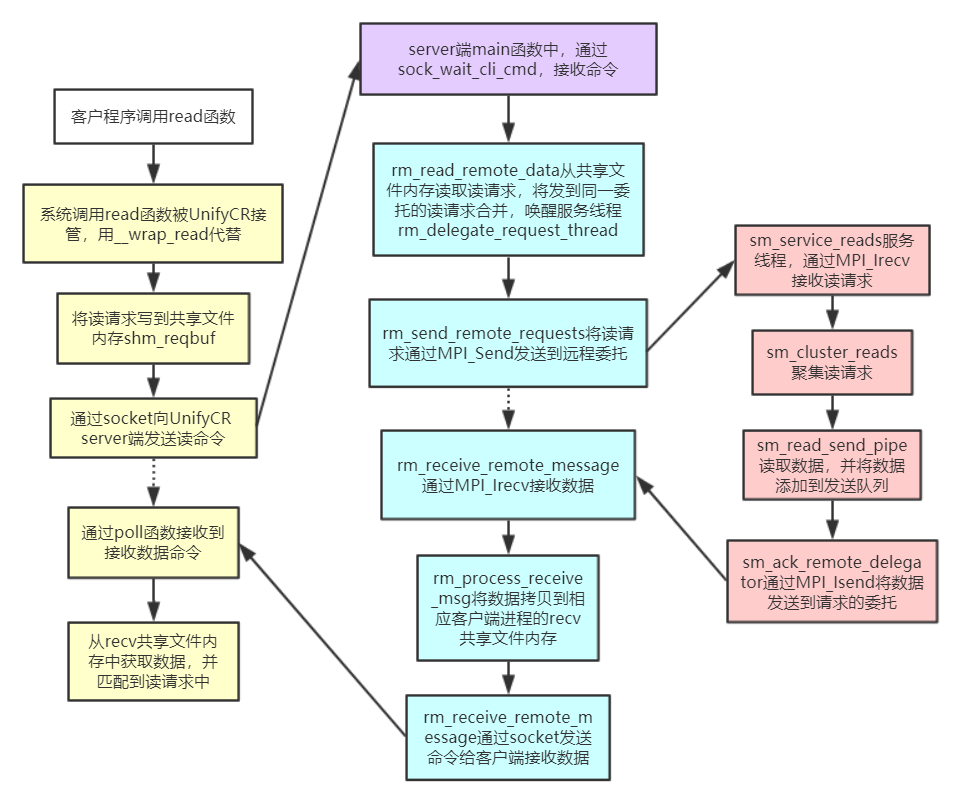


图2.19 UnifyCR读（read）流程

客户端程序通过read函数发起读操作。UnifyCR首先通过文件的路径来判断是否使用拦截该文件的I/O，若文件处于挂载点路径及其子目录下，则接管该文件的I/O操作，否则使用普通的系统调用，此处仅关心接管后的I/O操作。read函数调用unifycr\_fd\_logreadlist函数，将读请求写到共享文件shm\_reqbuf，再通过socket发送读cmd。

服务端main函数中启动service\_manager线程sm\_service\_reads，并通过sock\_wait\_cli\_cmd接收到cmd，调用函数delegator\_handle\_command处理cmd，使用rm\_read\_remote\_data函数处理读cmd。rm\_read\_remote\_data函数从对应客户进程的共享文件读取读请求元数据，将发到同一委托的请求合并，然后唤醒服务线程（rm\_delegate\_request\_thread）。

rm\_delegate\_request\_thread先通过函数rm\_send\_remote\_requests将读请求通过MPI\_Send发送到远程委托。之后通过rm\_receive\_remote\_message读数据。

sm\_service\_reads中，通过MPI\_Irecv接收读请求，通过sm\_cluster\_reads聚集读请求，通过sm\_read\_send\_pipe读取、发送数据。sm\_read\_send\_pipe调用sm\_wait\_until\_digested，最终调用sm\_ack\_remote\_delegator函数，通过MPI\_Isend将数据发送到请求数据的远程委托中。

rm\_receive\_remote\_message通过MPI\_Irecv接收数据，调用rm\_process\_received\_msg将数据拷贝到相应客户端进程的recv共享文件内存中。再通过socket发送命令让客户端接收数据。

#### 5. UnifyCR写（write）流程

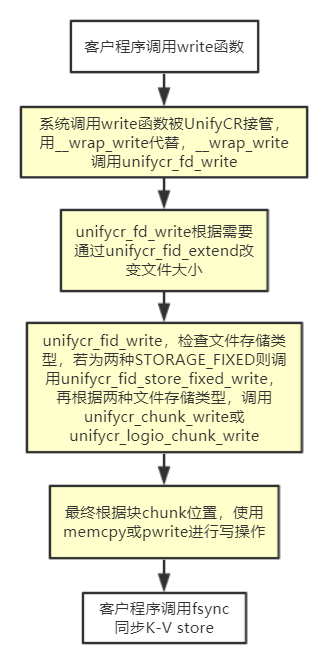


图2.20 UnifyCR写（write）流程

UnifyCR的write流程如图2.20所示。

不同于UnifyCR的read，UnifyCR写为局部（local）写，写操作本身不和UnifyCR服务端交互。

在客户程序完成write后，还需要调用fsync，接管的fsync会将app、进程及文件的相关数据保存到K-V store中。

#### 6. 部分核心函数解析

从个人源码阅读文档中，摘选部分核心函数。以伪代码形式列举、描述UnifyCR读、写流程中的部分核心函数。

**test\_read的main函数**

1. MPI初始化

2. 通过getopt函数，用命令行参数初始化参数。

参数含义：每个进程读/写seg\_num个block，每个block size为blk\_sz，每次操作读/写tran\_sz。

3. 通过unifycr\_mount进行挂载。

4. 开始读操作

每次读/写操作，根据N-1模式或N-N模式，计算偏移量offset。使用read函数进行读取。

5. 计算时间、读带宽。

**\_\_wrap\_open**

函数原型：int \_\_wrap\_open(const char \*path, int flags, ...)

1. 如果flags包含O\_CREAT，则读取可变长参数open的mode

2. 通过函数unifycr\_intercept\_path(path)检查是否拦截这个文件（unifycr初始化且该文件位于mount点，则拦截）

a) True

i. 调用unifycr\_fid\_open打开文件

ii. 分配一个空闲的unifycr文件描述符并与文件连接

iii. 返回fid+unifycr\_fd\_limit（加上偏移，避免和系统中活动的fd冲突）

b) False

i. 普通方式打开文件

**unifycr\_fid\_open**

函数原型：int unifycr\_fid\_open(const char \*path, int flags, mode\_t mode, int \*outfid, off\_t \*outpos)

说明：打开特定（mount）路径下的新文件id

1. 调用unifycr\_get\_fid\_from\_path检查该文件是否已存在，获取fid

若文件不存在（fid < 0）：

a) 调用unifycr\_get\_global\_fid(path, &gfid)获取gfid

b) 调用get\_global\_file\_meta(gfid, &ptr\_meta)从K-V存储获取元数据指针ptr\_meta

i. ptr\_meta == NULL

fid = -1

ii. ptr\_meta != NULL（其余进程已经创建了该文件，但文件的属性没有缓存到本地，分配一个file id slot）

1. 调用unifycr\_fid\_create\_file(path)创建文件，返回fid；

2. 调用unifycr\_fid\_store\_alloc(fid)初始化该文件的存储；

3. 调用unifycr\_get\_meta\_from\_fid(fid)初始化全局元数据

2. 根据文件是否存在：

a) 若fid < 0（步骤1.b中ptr\_meta为空）

若flags设置了O\_CREAT：

i. 调用unifycr\_fid\_create\_file(path)创建文件，返回fid；

ii. 调用unifycr\_fid\_store\_alloc(fid)初始化该文件的存储；

iii. 将新创建文件的属性送到key-value存储中，调用set\_global\_file\_meta、ins\_file\_meta

b) 文件已存在：

i. 检查文件打开是否存在错误

3. 设置出口参数outfid、outpos（0或追加写的size）

**\_\_wrap\_read**

函数原型：ssize\_t \_\_wrap\_read(int fd, void \*buf, size\_t count)

1. 通过unifycr\_intercept\_fd(&fd)检查是否拦截该文件（检查fd是否大于unifycr\_fd\_limit，

并将fd还原为原始fd（减去unifycr\_fd\_limit））

a) True

i. 调用unifycr\_get\_filedesc\_from\_fd(fd)获取该文件描述结构的指针

ii. 准备read\_req\_t，调用unifycr\_fd\_logreadlist(&tmp\_req, 1)进行读操作

b) False

普通方式读

**unifycr\_fd\_logreadlist**

函数原型：int unifycr\_fd\_logreadlist(read\_req\_t \*read\_req, int count)

说明：从委托获取读请求的数据，参数read\_req为数组，个数为count

1. 将read\_req中的fid转换为global fid（减去unifycr\_fd\_limit）

2. 使用qsort，根据read\_req的fid、offset排序（后续match中会用于比较）

3. 调用unifycr\_coalesce\_read\_reqs合并连续的读请求，并将超过限定大小（unifycr\_key\_slice\_range）的请求分成小请求

4. 将上步处理后的读请求队列中的每个读请求元数据结构体（shm\_meta\_t）写到shm\_reqbuf中（与服务端委托通信）。

5. 将这一批读请求信息（读标志、读请求个数）写到文件cmd\_fd.fd（客户端socket）中

6. while (tot\_sz > 0)

a) 通过poll函数同步

b) 从shm\_recvbuf中获取读请求及数据，并通过函数unifycr\_match\_received\_ack(read\_req, count, &tmp\_read\_req)匹配客户端的读请求和接收到的读请求，将tmp\_read\_req（shm\_recvbuf）中数据memcpy到read\_req的buf中

c) 修改tot\_sz等参数，回到6

**unifycr\_fd\_write**

函数原型：int unifycr\_fd\_write(int fd, off\_t pos, const void \*buf, size\_t count)

说明：从buf写count个字节到文件fd的pos偏移处，如果需要的话分配新的字节并更新文件大小，将gaps填充0

1. 检查fd是否可写、pos+count是否溢出等

2. 根据需要通过unifycr\_fid\_extend改变文件大小

3. 将filesize和pos间的gap填充为0

4. 最终调用unifycr\_fid\_write(fid, pos, buf, count)将数据写入文件

**unifycr\_fid\_store\_fixed\_write**

函数原型：int unifycr\_fid\_store\_fixed\_write(int fid, unifycr\_filemeta\_t \*meta, off\_t pos, const void \*buf, size\_t count)

说明：将数据写到fixed-size chunks类的文件

1. 计算写入位置的chunk\_id和chunk\_offset。

2. 根据文件类型用不同方式将数据写入chunk（可能当前chunk剩余空间不够，需往后续chunk写）

a) meta->storage == FILE\_STORAGE\_FIXED\_CHUNK

调用unifycr\_chunk\_write(meta, chunk\_id, chunk\_offset, buf, count)写入chunk

b) meta->storage == FILE\_STORAGE\_FIXED\_LOGIO

调用unifycr\_logio\_chunk\_write(fid, pos, meta, chunk\_id, chunk\_offset, buf, count)写入chunk

**unifycr\_chunk\_write**

函数原型：static int unifycr\_chunk\_write(unifycr\_filemeta\_t, \*meta, int chunk\_id, off\_t chunk\_offset, const void \*buf, size\_t count)

说明：从user buf写数据到指定chunk

1. 判断chunk的位置

a) 位于内存（CHUNK\_LOCATION\_MEMFS）

使用memcpy复制数据

b) 位于spill over设备（CHUNK\_LOCATION\_SPILLOVER）

调用pwrite进行写操作

**unifycr\_logio\_chunk\_write**

函数原型：static int unifycr\_logio\_chunk\_write(int fid, long pos, unifycr\_filemeta\_t \*meta, int chunk\_id, off\_t chunk\_offset, const void \*buf, size\_t count)

1. chunk位于内存（CHUNK\_LOCATION\_MEMFS）

a) 调用memcpy从buf拷贝count字节到指定chunk\_buf

b) 查找并更新文件属性（size）

c) 调用unifycr\_split\_index，将大于特定值（unifycr\_key\_slice\_range）的写请求分成小请求

d) 若原写请求队列最后一个能和新请求第一个合并（同一文件、连续，且属于同一slice）则合并

e) 将新请求添加到unifycr\_indices中

2. chunk位于spillover（CHUNK\_LOCATION\_SPILLOVER）

a) 调用\_\_real\_pwrite进行写操作

b) 查找并更新文件属性（size）

c) 调用unifycr\_split\_index，将大于特定值（unifycr\_key\_slice\_range）的写请求分成小请求

**rm\_read\_remote\_data**

函数原型：int rm\_read\_remote\_data(int sock\_id, int req\_num)

说明：获取读请求数据，唤醒读请求服务线程

1. 通过sock\_id获取app\_id、client\_id及服务线程thrd。

2. 调用meta\_batch\_get，从key-value store中获取读请求的位置。

3. 调用qsort，根据send\_msg\_t的目标委托rank排序。再合并发送到同一委托的发送请求。

4. 通过pthread\_cond\_signal唤醒服务线程（rm\_delegate\_request\_thread）

**rm\_delegate\_request\_thread**

函数原型：void \*rm\_delegate\_request\_thread(void \*arg)

说明：线程函数，处理读请求，每个客户端程序的进程对应绑定一个该线程，具有参数cli\_signature\_t，其具有客户端app\_id和sock\_id。

1. 通过参数的app\_id、sock\_id获取线程控制块thrd\_ctrl\_t。

2. 调用rm\_send\_remote\_requests(thrd\_ctrl, thrd\_id, &tot\_sz)向远程委托发送请求。

3. 调用rm\_receive\_remote\_message(app\_id, sock\_id, tot\_sz)接收请求的数据。

**rm\_send\_remote\_requests**

函数原型：int rm\_send\_remote\_requests(thrd\_ctrl\_t \*thrd\_ctrl, int thrd\_tag, long \*tot\_sz)

说明：向远程委托发送读请求

1. 调用rm\_pack\_send\_requests将一批目标相同的请求打包，并累计总大小tot\_sz

2. 使用MPI\_Send将请求发送到对应的远程委托

**rm\_receive\_remote\_message**

函数原型：int rm\_receive\_remote\_message(int app\_id, int sock\_id, long tot\_sz)

说明：通过MPI\_Irecv接收请求的数据，调用rm\_process\_received\_msg函数。之后通过socket向客户端发送命令吸收数据

1. 循环接收数据（while (tot\_sz > 0)）

a) 使用MPI\_Irecv从远程委托接收数据

b) 调用rm\_process\_received\_msg处理接收的数据

2. 调用sock\_notify\_cli向客户端发送命令，使用数据。

**rm\_process\_received\_msg**

函数原型：int rm\_process\_received\_msg(int app\_id, int sock\_id, char \*recv\_msg\_buf, long \*ptr\_tot\_sz)

说明：分析接收到的数据，通过memcpy将recv\_msg\_buf中数据拷贝到app\_config的shm\_recv\_bufs中

1. 通过app\_id获取对应的app\_config

2. 若app\_config的接收共享内存空间不足，则调用sock\_notify\_cli通知客户端吸收数据

3. 将数据拷贝到app\_config->shm\_recv\_bufs中

**sm\_service\_reads**

函数原型：void \*sm\_service\_reads(void \*ctx)

说明：线程函数，处理从requesting委托接收到的读请求

1. 调用sm\_init\_socket()初始化socket

2. 分配READ\_BUF\_SZ（1GB）的mem\_buf

3. 初始化read\_task\_set、service\_msgs、pended\_reads、pended\_sends、send\_buf\_lst、rank\_ack\_task。

4. while(!flag)循环

a) 调用MPI\_Irecv（非阻塞）从任意来源接收读请求消息，使用MPI\_Test检测接收状态（irecv\_flag）。若接收未完成，则持续接收读请求直到预期bursty行为结束。while(!irecv\_flag)：

i. usleep等待

ii. 当认为bursty行为结束（等待时间超过了bursty\_interval）：

1. 调用sm\_cluster\_reads，基于文件offset和ages聚集读请求

2. 调用sm\_read\_send\_pipe，通过read、copy、send流水线读取并发送数据

iii. MPI\_Test

b) 调用sm\_decode\_msg(req\_msg\_buf)解码请求消息，或判断是否结束退出循环

**sm\_cluster\_reads**

函数原型：int sm\_cluster\_reads(task\_set\_t \*read\_task\_set, service\_msgs\_t \*service\_msgs)

说明：基于文件的offset和ages聚集读请求。

read\_task\_set：包含聚集读请求的读任务列表；service\_msgs：接收的读请求消息列表

1. 使用qsort对service\_msgs的msg排序，根据msg的app\_id、client\_id、offset排序。

2. 将service\_msgs的读请求转到read\_task\_set。属于同一个app\_id、client\_id，offset连续的读请求，且总size小于READ\_BLOCK\_SIZE则合并，并以其中最早到达的时间作为合并后的读请求arrival\_time。最终的read\_task以app\_id、client\_id为单位，且size小于1MB。

read\_task\_set中的read\_task（read\_task\_t）的start\_idx、end\_idx为service\_msgs->msg的下标

**sm\_read\_send\_pipe**

函数原型：int sm\_read\_send\_pipe(task\_set\_t \*read\_task\_set, service\_msgs\_t \*service\_msgs, rank\_ack\_task\_t \*rank\_ack\_task)

说明：通过read、copy、send三级流水线读取并发送数据。

read\_task\_set：读任务列表；service\_msgs：接收的读请求列表；rank\_ack\_task：发向请求委托的发送任务列表

1. 使用qsort对read\_task\_set的read\_tasks排序，基于size、arrival\_time排序

2. 对read\_task\_set中的每个task循环处理：

a) 当read buffer满了（buf\_cursor + task.size > READ\_BUF\_SZ）时，调用sm\_wait\_unitl\_digested(read\_task\_set, service\_msgs, rank\_ack\_task)等待

b) 根据请求数据的位置：

i. 全部在shared memory（tmp\_offset + task.size <= app\_config->data\_size）：从app\_config和tmp\_offset获取请求数据的起始位置，通过memcpy拷贝到mem\_buf中。调用batch\_ins\_to\_ack\_lst将数据放到rank\_ack\_task网络队列中。

ii. 部分请求的数据在shared memory（tmp\_offset < app\_config->data\_size）：shared memory中的数据通过memcpy拷贝到mem\_buf中，剩余数据（ssd中）使用pread从app\_config->spill\_log\_fds指示的fd读取到mem\_buf中。

iii. 所有请求的数据都在ssd：将读操作添加到pended\_reads列表，并通过aio\_read异步读取数据

3. 调用sm\_wait\_until\_digested(read\_task\_set, service\_msgs, rank\_ack\_task)等待队列中数据发送完毕

**sm\_wait\_until\_digested**

函数原型：int sm\_wait\_until\_digested(task\_set\_t \*read\_task\_set, service\_msgs\_t \*service\_msgs, rank\_ack\_task\_t \*read\_ack\_task)

说明：等待直到所有数据读取并发送。传入read\_ack\_task的实参为rank\_ack\_task

1. 将所有pended\_reads中的数据通过batch\_ins\_to\_ack\_lst添加到发送发送任务队列中。重

置pended\_reads

2. 对read\_ack\_task中的所有task，调用sm\_ack\_remote\_delegator通过MPI\_Isend向远程委托发送ack（包括请求的数据）

3. 对所有pended\_sends中的ack使用MPI\_Test检测发送，直到所有ack都已发送。重置pended\_sends

4. 重置read\_ack\_task中的所有数据

## 2.4 实习工作总结

虽然这次实习项目较新、涉及到非常多没接触过的知识，也遇到了许多困难，但最终逐一解决了问题后，我本人也收获了很多。以下是我对本次实习工作的总结与体会。

1. 阅读论文及相关资料。

阅读学习关于Burst Buffer以及BurstFS的相关论文与资料，写阅读文档，了解本项目的基本背景知识，包括高性能计算中超级计算机的I/O特点、存在的瓶颈问题，Burst Buffer的目的、优势，以及BurstFS的设计等等。然而，这其中还涉及到了许多本科生阶段未学习的知识，因此需要自行查阅涉及到的相关知识，如并行MPI、Infiniband等。

1. BurstFS环境搭建

虽然这个文件系统是用户级文件系统，不需要重新编译系统内核。但因其工作环境的特殊性、缺少资料等原因，其编译安装过程也不容易。按照官方给的简易安装教程并不能正确地安装该文件系统。后来我们找到了该文件系统在Github上的Travis CI，按照上面的步骤，发现了问题并帮助我们成功编译。同时编写安装文档。

安装过程中遇到了非常多的问题，例如安装UnifyCR需要先安装leveldb、gotcha、MPI等环境，运行UnifyCR的测试程序前需要先添加一些指定路径的文件夹和配置文件等。

1. 学习UnifyCR代码

结合运行调试与论文，阅读学习UnifyCR的代码，理解UnifyCR的工作机制与流程，并编写整理UnifyCR代码阅读文档。

作为一个文件系统，UnifyCR的代码量很大，并且还是一个主要运用于HPC中间层存储的临时性分布式文件系统，难度也比较大。但好在论文提供了文件系统的部分设计理念，以及代码中作者写了不少注释，对于梳理代码有不小的帮助。

经过5周的实习，我个人受益匪浅。首先实习接触的这个课题较新颖，让我对HPC中的Burst Buffer有了简单的了解，这种结构对提升高性能计算的I/O性能非常有帮助。Burst Buffer也是对传统计算机冯诺依曼结构的一种革新。BurstFS则是专门用于Burst Buffer的一种临时性文件系统，为Burst Buffer提供软件支撑。此外，我还对存储有了更多的了解。但受限于专业知识以及实习时间，我对分布式的了解依然很少，实习期间能学到的知识也只是存储的冰山一角。

## 2.5 实习建议

1. 实习期间应多与他人交流，每个人的实习经验都非常宝贵，应相互交流、学习。

2. 虽然在学校实验室实习的主要目的不是为找工作提供经验，但也应注重锻炼实习生的工程能力。因为本科期间学的知识主要是基础知识，但对这些知识在项目中的运用能力还很薄弱，实习阶段是一个很好的锻炼机会。

3. 本科生实习不必等到大三暑期再进行。更早地接触科研、项目很有好处，当然这也与学校的培养计划有关。

# 3 实习问题解答

问题：

Burst Buffer有什么不足之处，有什么优化改进方法？

回答：

HPC（高性能计算）在应用中通常发生浪涌性I/O。百亿亿级计算的时代，非易失性RAM被用来作为Burst Buffer，来吸收这些大量突发数据，并减少对永久性存储系统的I/O需求。但是，在没有对Burst Buffer合理的刷写情况下，突发I/O被传到底层存储系统，将导致严重的I/O争用问题。

HPC中，科学应用在计算阶段的I/O操作很少，而在每个计算片段结束时，集中地I/O，更新分析文件和CP文件。各种各样的应用同时运行时，不同应用的I/O阶段交叠，导致了很高的I/O峰值。如图3.1，该图显示了OLCF Spider存储系统一天的I/O记录。

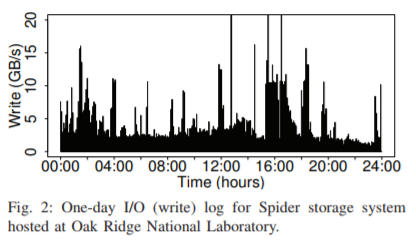


图3.1 Spider存储系统一天的写记录

大部分时间，写吞吐量低于10GB/s，某些时间写吞吐量超过10GB/s甚至超过20GB/s。I/O负载比存储系统能提供的性能指标低很多。尽管现代存储系统利用了各种缓冲来解决I/O峰值，但如果没有恰当的下发，I/O峰值仍然会出现在永久存储系统中。

即便使用了Burst Buffer，如果没有特殊设计的刷写方案，浪涌式I/O对存储的利用率仍然不高。采用Burst Buffer后，一种直观的策略是反应性刷写（reactive draining），即当数据到达Burst Buffer时，对数据被动地进行刷写。

如图3.2（a），使用reactive draining，burst buffer很快地将数据刷写到底层存储系统，导致将密集的I/O传递到底层存储系统，依然在底层文件系统产生突发性I/O问题。

一种改进的策略是使用proactive draining，即主动控制将数据引流到底层永久存储系统来平缓I/O峰值。这种策略利用了科学应用I/O的周期性特性。因为科学应用的I/O性能主要取决于Burst Buffer而不是永久存储系统，故proactive draining将数据分成小块，在I/O间隔内分散I/O的刷写，而不影响程序的I/O性能。

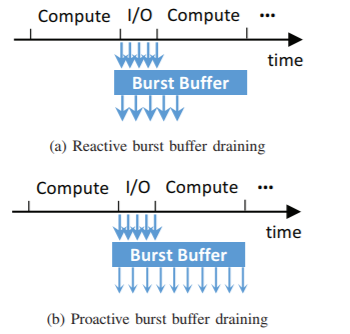


图3.2 reactive draining和proactive draining示意图

在HPC存储系统中，OST（Object Storage Targets）通常以循环的方式分配给应用程序。如图3.3（a），OST 1~8分配给应用程序1，OST 9~16分配给应用程序2。应用程序1的每个文件跨4个OST，应用程序2的每个文件跨2个OST。

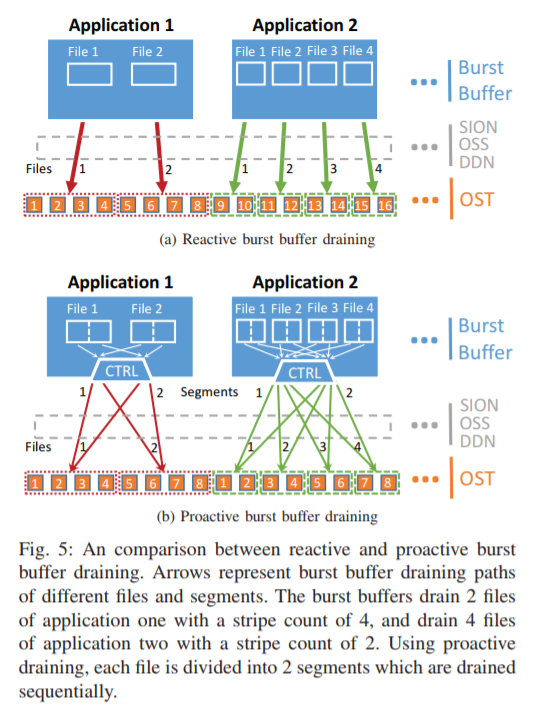


图3.3 reactive和proactive draining的对比

在reactive draining，当burst buffer中的数据可用时，将数据刷写到底层永久存储系统，当burst buffer得到分配的OST，将OST添加到等待队列中。而采用proactive draining策略，如图3.3（b）所示，每个文件被分成2个片段。Busrt Buffer刷写控制器先刷写每个文件的第一个片段，再刷写每个文件的第二个片段。Burst Buffer向分配的OST发起一段较长时间的刷写请求。刷写的片段周期性地到达同一组OST。如图3.3（b）所示，应用程序2的刷写片段与应用程序1的刷写片段在同一个OST上交叠。

在proactive draining中，片段是调度和刷写的基本单元。对不同应用刷写片段的I/O请求不能互相重叠，否则，来自不同应用的顺序读写变成了随机读写，导致OSSs（Object Storage Servers）上请求竞争、OSTs上资源竞争。为了协调不同应用的下发过程和避免共享OSTs上的竞争，proactive burst buffer刷写控制器需要计算下发不同应用的片段的最大时间跨度。然后，控制器以此将数据划分成合适的片段，周期性地刷写这些片段。

# 参考文献

[1] TechRprt@NESRC\_Burst Buffer Architecture and Software Roadmap\_2018 <http://www.nersc.gov/users/computational-systems/cori/burst-buffer/burst-buffer/>

[2] Burst buffer – Wikipedia <https://en.wikipedia.org/wiki/Burst_buffer>

[3] Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior, 2017 IEEE 25th International Symposium, Kun Tang, Ping Huang, Xubin He, et al

[4] Accelerating Science with the NERSC Burst Buffer Early User Program, 2016-01-01 LBNL Bhimji, W Bard, D Romanus, M et al.

[5] BurstFS: An Ephemeral Burst Buffer File System for Scientific Applications, T. LLNL 2016-01-28, Wang, W. Yu, K. Sato, A. Moody, K. Mohror.

[6] BurstMem: A High-Performance Burst Buffer System for Scientific Applications, Teng Wang, Sarp Oral, Yandong Wang, et al, 2014 IEEE International Conference on Big Data

[7] A 1 PB/s File System to Checkpoint Three Million MPI Tasks, Raghunath Rajachandrasekar, Adam Moody, Kathryn Mohror, Dhabaleswar K.