中国科学技术大学

博士学位论文

纠删码容错存储系统的编解码流程优化研究

作者姓名：　徐亮亮

学科专业：　计算机软件与理论

导师姓名：　许胤龙 教授　吕敏 副教授

完成时间：　二〇二二年五月十六日

University of Science and Technology of China

A dissertation for doctor's degree

Study on Coding Workflow in Erasure-coded Storage Systems

Author: Liangliang Xu

Speciality: Computer Software and Theory

Supervisors: Prof. Yinlong Xu, Assoc. Prof. Min Lyu

Finished time: May 16, 2022

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文，是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。本人提交的电子文档的内容和纸质论文的内容相一致。

控阅的学位论文在解密后也遵守此规定。

☑ 公开 ☐ 控阅（____ 年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

摘　要

随着互联网应用的快速发展，用户的数据呈指数型增长，存储系统对容量以及性能的需求越来越高。保证高可靠性是存储系统的基础功能，多副本和纠删码是存储系统常用的两种容错存储策略。多副本通过多倍的冗余来保证存储系统的高可靠性。相对于多副本，纠删码能够用低存储开销提供高可靠性，但是在数据读写、降级读以及故障修复等过程中，需要大量的跨节点数据传输和编解码计算，所以网络和计算常成为性能瓶颈。一般来说，纠删码存储系统关注容错能力、读写性能、降级读性能以及修复性能等指标。本文主要关注于高可靠存储系统在不同的应用场景和需求下，对纠删码存储策略进行编解码流程的设计与优化，以满足系统对关键指标的要求。具体包括基于纠删码存储系统的数据布局设计来优化故障修复性能、基于纠删码存储系统的故障修复任务调度设计来均衡修复负载以及基于分离内存系统架构的纠删码流程设计来提供高可靠/高性能存储系统。其主要研究内容和贡献点如下：

(1) 基于纠删码存储系统的数据布局设计：

分布式存储系统中常采用随机数据布局来保证存储上的均衡，但是在故障修复过程中会导致大量的跨机架流量和分批修复的负载不均衡，从而显著地降低了修复性能。另外，分布式存储系统中常常部署混合纠删码来满足多样性的用户需求，这会进一步加剧上述问题。为此，本文提出了一种均匀数据布局 PDL（PBD-based Data Layout）来优化分布式存储系统中的故障修复性能。PDL 是基于成对均衡设计（一种具有均衡数学特性的组合设计工具）构造的，因此能够为混合纠删码提供均匀的数据分布。基于提出的数据布局 PDL，本文提出了相应的负载均衡的故障修复方案 rPDL。该修复方案通过选择替代节点和源节点，有效地减少了跨机架流量，并提供了近似均衡的跨机架流量分布。本文在 HDFS 3 中实现了 PDL 和 rPDL，与 HDFS 现有的数据布局和修复方案相比，rPDL 实现了更高的修复吞吐率，分别达到了单节点故障的 6.27 倍，多节点故障的 5.14 倍以及单机架故障的 1.48 倍。除此之外，rPDL 将降级读延迟平均降低了 62.83%，并减轻了在故障修复时对前端应用的影响。

(2) 基于纠删码存储系统的故障修复任务调度设计：

纠删码策略通常以低存储成本为数据提供高可靠性。一旦发生故障，丢失的块将会被批量修复。由于一批修复的故障条带数量有限，批次内的数据布局是不均匀的。再加上修复任务的源节点和替代节点的随机选择，节点间的修复负载在一个批次内是不均衡的，这严重减慢了故障修复的速度。为了解决这个问题，本文提出了一个修复任务调度模块 SelectiveEC，它为基于纠删码的大规模

存储系统提供了可证明的网络流量和修复负载均衡。首先，它依赖二分图来模拟节点之间的修复流量。然后，它动态地选择任务以形成批次，并使用完美或最大匹配以及 k-正则子图等理论，仔细确定地选择源数据块或存储修复块的位置。SelectiveEC 支持单节点故障和多节点故障修复，并且可以部署在同构和异构网络环境中。本文在 HDFS 3 中实现了 SelectiveEC，并在 18 节点的本地集群和 50 个虚拟机实例的 AWS EC2 中评估其修复性能。在同构网络环境中，与最先进的故障修复方法相比，SelectiveEC 将修复吞吐率提高了 30.68%。在异构网络环境中，由于均衡调度避免了负载过重的节点，它进一步实现了 HDFS 的平均 1.32 倍修复吞吐率和 1.23 倍的前台任务吞吐率。

(3) 基于分离内存系统架构的纠删码流程设计：

在分离内存系统中，纠删码冗余策略能够以低内存成本提供高可靠性。然而，随着单边 RDMA 延迟可以达到微秒级，不同于传统存储系统中网络和磁盘 I/O 等资源是瓶颈，编解码计算成为分离内存系统部署纠删码的新瓶颈。为了在分离内存系统中实现纠删码高效部署，本文先通过对编解码计算和 RDMA 传输的工作流程详细的分析得到了三个关键的系统发现。然后，本文提出了 MicroEC，它通过缓存优化重新设计了编解码函数栈，并利用高效的流水线来协同优化编解码计算以及 RDMA 传输。本文实现了一个具有一般操作支持的系统原型，例如写/读/降级读/修复。实验表明，MicroEC 显著降低了编解码延迟，与单边 RDMA 的低延迟相匹配，尤其是对于大于 1MB 的大型对象。与最先进的纠删码和三副本技术相比，它还分别实现了高达 2.08 倍和 1.74 倍的写入吞吐率。

关键词：可靠性；分布式存储系统；分离内存；纠删码；数据布局；故障修复

ABSTRACT

The rapid development of internet applications brings exponential growth of data volume, which makes storage systems with higher and higher demand for capacity and performance. Ensuring high reliability is the foundation of storage systems. In general, replication and erasure coding are two common storage strategies to provide fault tolerance in storage systems. Replication ensures the high reliability of storage systems through multiple redundancies. Compared with replication, erasure coding (EC) can provide high fault tolerance with low storage cost, but in the process of data read and write, degraded read and failure recovery, needs a lot of cross-node data transmission and encoding/decoding operations, so network and computing often become the performance bottleneck. In general, erasure-coded storage systems focus on multiple indicators such as fault tolerance, read and write performance, degraded read performance, and recovery performance. This paper mainly focuses on the coding workflow designs and optimizations of erasure coding under different application scenarios and requirements in highly reliable storage systems to meet the systems' requirements for key targets. Specifically, it includes a data layout design for erasure-coded storage systems to optimize failure recovery performance, a recovery task scheduling design in erasure-coded storage systems to balance recovery load, and a workflow design of erasure coding in disaggregated memory systems to provide high-reliability and high-performance storage systems. Its main research contents and contributions are as follows:

(1) Research on a data layout design for erasure-coded storage systems

In distributed storage systems, random data layout is commonly used to ensure balanced storage, but the traditional random data placement induces massive cross-rack traffic and imbalanced load during failure recovery batch by batch, which degrades the recovery performance significantly. In addition, various erasure codes coexisting in a DSS exacerbate the above problems. In this paper, we propose PDL, a uniform data layout, to optimize failure recovery performance in DSSes. PDL is constructed based on Pairwise Balanced Design, a combinatorial design scheme with uniform mathematical properties, and thus presents a uniform data layout for mixed erasure codes. Then we propose rPDL, a failure recovery scheme based on PDL. rPDL reduces cross-rack traffic effectively and provides a nearly balanced distribution of cross-rack traffic by uniformly choosing replacement nodes and retrieving determined available blocks to recover the lost blocks. We implement PDL and rPDL in HDFS 3. Compared with the existing data

layout and recovery scheme in HDFS, experimental results show that rPDL achieves much higher recovery throughput, 6.27× on average for single-node failures, 5.14× for multi-node failures and 1.48× for single-rack failures, respectively. It also reduces degraded read latency by an average of 62.83%, and provides better support to front-end applications under failures.

(2) Research on a recovery task scheduling design in erasure-coded storage systems

Erasure coding has been commonly used to offer high data reliability with low storage cost. Upon failures, the lost blocks are recovered in batches. Due to the limited number of stripes, the data layout within a batch is non-uniform. Together with the random selection of source and replacement nodes for recovery tasks, the recovery load among surviving nodes is skewed within a batch, which severely slows down failure recovery. To solve this problem, we present SelectiveEC, a new recovery task scheduling module that provides provable network traffic and recovery load balancing for large-scale erasure-coded storage systems. It relies on bipartite graphs to model the recovery traffic among surviving nodes. Then, it intelligently selects recovery tasks to form batches and carefully determines where to read source blocks or to store recovered ones, using theories on a perfect or maximum matching and k-regular spanning subgraph. SelectiveEC supports single-node failure and multi-node failure recovery, and can be deployed in both homogeneous and heterogeneous network environments. We implement SelectiveEC in HDFS 3, and evaluate its recovery performance in a local cluster of 18 nodes and AWS EC2 of 50 virtual machine instances. SelectiveEC increases the recovery throughput by up to 30.68% compared with state-of-the-art baselines in homogeneous network environments. It further achieves 1.32× recovery throughput and 1.23× benchmark throughput of HDFS on average in heterogeneous network environments, due to the straggler avoidance by the balanced scheduling.

(3) Research on a workflow design of erasure coding in disaggregated memory systems architecture

In disaggregated memory systems, erasure coding can provide high reliability with low memory cost. However, as the latency of one-sided RDMA goes down to the microsecond level, coding computation becomes the new bottleneck in disaggregated memory with EC, instead of the expensive network and disk I/Os as in traditional storage systems. To enable efficient EC in disaggregated memory, we first present three key insights for guiding the design by thoroughly analyzing the workflows of coding and RDMA transmission. We then develop MicroEC, which redesigns the cod-

ing stack with cache optimizations and also leverages efficient pipeline to optimize RDMA transmission. We implement a prototype with general operations support, such as write/read/degraded read/recovery. Experiments show that MicroEC significantly reduces the coding latency, which matches the low latency of one-sided RDMA, especially for large objects of size greater than 1MB. It also achieves up to 2.08× and 1.74× write throughput, compared with the state-of-the-art EC and 3-way replication polices, respectively.

Key Words: Reliability; Distributed Storage Systems; Disaggregated Memory; Erasure Coding; Data Layout; Failure Recovery

目　录

第 1 章　绪　　论

1.1　高可靠存储系统概述

1.1.1　数据存储的意义

随着 5G、大数据、人工智能等新兴领域的快速发展，使得用户的数据呈指数型增长。根据国际数据公司 IDC 预测 [1]：全球用户产生的数据量将从 2018 年的 33ZB 增长到 2025 年的 175ZB，年增长速度约为 30%；另外中国用户的数据量将以超全球 3% 的增速排在世界首位，中国用户的数据量将在 2025 年占全球数据量的 27.8%。据中国互联网络信息中心在 2022 年 2 月 25 日发布的第 49 次《中国互联网络发展状况统计报告》[2]，图 1.1 显示了我国网民规模已经超过 10 亿，全民互联网的普及率达到 73.0%。随着数字化时代的快速发展，互联网的用户数以及数据量都会呈增长的趋势。

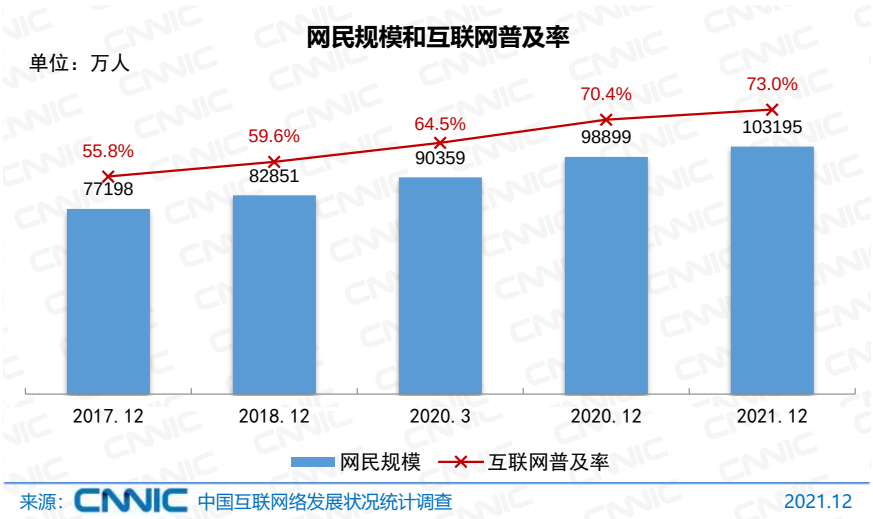


图 1.1　中国网民规模和互联网普及率

随着社会经济各领域数字化建设的发展，使得数据成为经济发展的战略资源，数据存储的需求也呈现指数级增长。国务院发布《“十四五”数字经济发展规划》并对“十四五”期间我国数字经济发展做出了整体性部署：在 2025 年，我国数字经济将迈向全面扩展期，数字经济产业增加值占 GDP 比重达到 10%。2022 年 2 月，国家发展改革委宣布“东数西算”工程全面启动，将在全国 8 地建设国家算力枢纽节点以及规划 10 个国家数据中心集群，全国一体化大数据中心建设将推动数字经济发展。目前，国家的信息化发展成就显著，互联网行业发展迅速。

当用户数量以及用户产生的数据量逐渐增加，存储系统是数据存在和发挥价值的基础平台：一方面需要保证大容量数据存储的可靠性，另一方面需要保证用户对数据的高效访问性能。存储系统也是信息化时代以及数字化经济下国家

建设的基础设施。

1.1.2　分布式存储系统

随着新业务的产生以及增长，存储系统性能的需求也将越来越高，容量与性能的双重增长，导致存储系统面临新的挑战。分布式存储是一种数据存储技术，简单来说，就是将数据分散存储到多个存储服务器上，并将这些分散的存储资源构成一个虚拟的存储设备，并在用户使用时无感知，但是实际上数据分散地存储在服务器的各个角落。我们将这一类的系统称作分布式存储系统 [3-6]。作为代表性的开源分布式存储系统，HDFS [5] 具有主从的架构，主节点服务器 NameNode 管理文件系统命名空间并控制客户端对文件的访问。从节点由许多 DataNode 组成，用于存储数据以及服务用户请求。

分布式存储系统通常部署树型的拓扑结构，它由多个机架组成，每个机架包含一些存储节点。如图 1.2 所示，一个机架中的所有存储节点都通过一个架顶（TOR）交换机连接，而所有机架通过一个核心交换机连接，形成一个树型结构的拓扑结构。由于大量的读写操作访问不同机架的数据，可用的跨机架带宽面临着激烈的竞争：通常来说，每个节点可用的跨机架带宽仅为机架内带宽的 1/20 到 1/5 [7-8]。因此，跨机架带宽通常被认为是比机架内带宽更稀缺的资源，对于分布式存储系统中的访存性能至关重要。

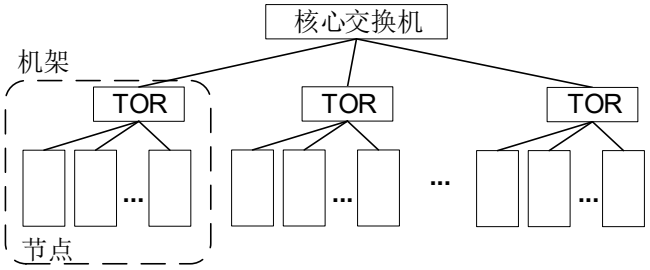


图 1.2　分布式存储系统的树型拓扑结构

1.1.3　分离内存存储系统

分布式存储系统虽然能够通过横向扩展 (增加物理节点) 或纵向扩展 (升级集群内部分节点的某些硬件设备) 来提升自己的处理能力，但随着应用的多样化和复杂化，传统存储系统面临着新的挑战：(1) 资源的过度配置。应用的异构性导致运行过程中资源需求的严重不均衡，存储系统通常按照峰值需求配置机器资源，导致资源的过度配置，造成存储系统成本很高而又有大量闲置资源。(2) 远程使用代价高。存储系统中单个节点可能用到其他节点的系统资源，借助于高速网络虽然能远程使用内存等空闲资源，但访问路径长以及效率低，并不能从本质上解决远程资源的高效动态使用。(3) 设备故障影响大。当前存储系统最小的任务分配单元是节点，当某个节点的部分组件发生故障时，该节点会完全丧失功能，造

成所有的资源都不能使用。由于高速网络技术的发展，例如远程直接内存访问（Remote Direct Memory Access, RDMA），网络传输的带宽和时延都接近内存访问性能。例如，200 Gb/s Mellanox ConnectX-6 IB RNIC（RDMA NIC）的吞吐量非常接近 DDR3-1600 DRAM 的 4 通道的本地内存访问 [9]。因此，可以使用弹性扩展的方式设计具有微秒级访问时延的共享大容量内存池，并且持久内存 [10-11] 的商用也进一步推动了其发展。

为此，分离内存系统应运而生，它通过解耦计算和内存来提高资源利用率。计算池包含一个内存非常有限的计算服务器集群，而内存池通过使用 DRAM 或持久内存提供大的共享内存空间。图 1.3 展示了分离内存的典型系统架构图，它将计算资源和内存资源分为计算池和内存池。一般来说，内存池中计算能力有限，所以计算服务器负责计算任务和用户服务。另外由于计算池端本地内存有限，所以用户数据存储在内存池端，计算服务器必须访问远端内存服务器的数据。为了在内存池中不涉及内存服务器算力的情况下同时提供低访问延迟，计算服务器和内存服务器通常通过高速网络连接，并采用远程访问技术 [12-13]，例如远程直接内存访问（RDMA）或 Gen-Z。本文重点讨论广泛使用的单边 RDMA 技术：也就是 RDMA `WRITE` 和 `READ`，它可以提供与本地内存访问类似的微秒（μs）级延迟以及吞吐率 [9]。

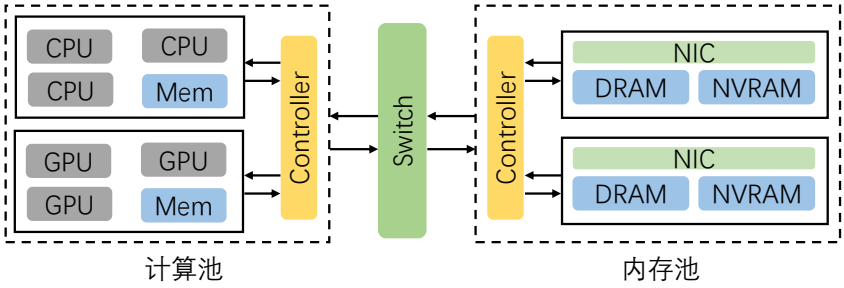


图 1.3　分离内存存储系统架构图

1.1.4　存储系统可靠性

高性能的存储系统通常由许多的服务器或者磁盘组织而成，包括软件、硬件、网络连接和电源等各个方面，所以故障事件是经常发生的，比如磁盘损坏、节点失效、网络故障以及集群宕机等 [7]。图 1.4 描述了谷歌公司存储集群在三个月内的故障事件分布 [7]，可以看到故障事件的种类是繁多的，而且随着时间的推移，故障事件的发生也是没有规律的。Facebook 公司也分析了一个真实的 3000 节点集群（存储 2300000 个块，每个块 256MB）一个月的故障节点数，发现每天有 20 个或者更多节点故障的情况是很常见的 [14]。

存储系统的故障常常会带来各方面的损失。2021 年 3 月，欧洲最大云服务器厂商 OVH 在法国斯特拉斯堡的数据中心遭受到史无前例的火灾，造成了 OVH

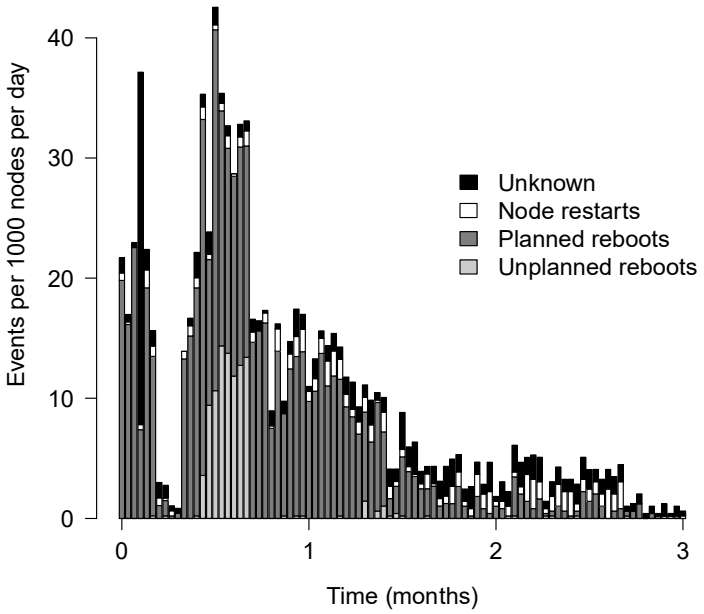


图 1.4　谷歌公司存储集群故障事件分布

许多全球的客户服务中断，导致了超过 350 万个网站下线，影响涉及到了政府、银行等 [15]。中国许多云服务厂商也遇到过许多故障事件从而造成损失 [16]，比如腾讯云在 2018 年 7 月出现广州一区主备两条运营商网络链路同时发生故障，2018 年 8 月出现北京某企业在腾讯云服务器上包括备份的数据全部丢失；阿里云在 2019 年 3 月出现大规模宕机，影响了华北地区的许多互联网公司；以及华为云在 2020 年 4 月出现历史上首次云服务大面积瘫痪。存储系统的故障事件往往是不可避免的，所以高可靠存储系统保证数据的容错是非常重要的。

1.2　高可靠存储系统容错策略

1.2.1　存储系统容错策略

存储系统为容忍故障从而来保证数据可靠性，通常需要牺牲一些存储开销，并在多个节点上存储冗余数据。但是，冗余数据不仅仅带来额外的存储成本，冗余数据的产生、元数据的管理以及故障修复时的访存 I/O 等都会造成存储系统的性能下降。所以，高效的冗余存储策略对存储系统来说非常重要。一般来说，现有存储系统的容错策略包括两类：多副本技术 [17-20] 和纠删码技术 [21-23]。

多副本技术：多副本技术通过对相同数据拷贝多份的方式来提供容错，但是它的存储成本很高，比如通常使用的三副本有额外 200% 的存储开销。但是由于多副本的读、写以及修复阶段的实现较为简单，许多系统默认采用多副本技术，比如 HDFS [20] 和 Ceph [4]。目前业界最通用的实现方法是主从副本 (PBR) 实现方式 [17-20]，该方法的主副本提供主要的用户请求，其他的副本提供容错需求。

多副本产生的高额副本开销不仅仅是存储开销，冗余的副本也会进一步给网络带宽、内存占用等方面带来额外的资源占用。对于信息化时代的海量数据存储来说，多副本的高存储开销是不可忍受的。

纠删码技术：纠删码技术是一个天然的代替多副本的存储策略。纠删码技术可以提供与多副本相同级别的容错，并且存储开销要低得多。一般业界关于纠删码的配置中，额外的存储开销不会超过 50% [24-25]。然而，由于纠删码需要额外编解码计算开销以及更为复杂的数据条带化的逻辑，纠删码会导致存储系统其他的开销，比如高修复开销 [26-29]。但是可以通过算法以及系统实现等对纠删码系统进行性能提升。下一小节将对纠删码技术从多方面来进行介绍。

1.2.2　纠删码技术简介

1. 纠删码家族

Reed-Solomon (RS) 码 [7, 14, 24, 27, 30] 作为一种代表性的纠删码，具有两个正整数参数 k 和 m，RS(k,m) 码将 k 个数据块编码为 m 个额外的校验块，并把它们分别存放在 $k+m$ 个独立的存储设备上，$k+m$ 个块中的任何一个都可以通过其他数据/校验块重构得到。所有的 $k+m$ 数据/校验块形成一个条带，$k+m$ 被称为条带大小。因此，(k,m) 码允许任何 m 块丢失，达到了最大距离可分（Maximum Distance Separable, MDS）属性 [31]。在实际部署中，最常用的 MDS 码是线性的。也就是说，任何块都是同条带中任意 k 个其他块的线性组合。RS 码也是一种线性 MDS 码，本文主要以 RS 码为例来介绍纠删码存储系统的多方面优化。

除此之外，还有一些流行的纠删码种类。例如，旋转的 RS 码 [32] 通过降低修复过程中读取的数据量来改善降级读性能。再生码 [33-34] 存储成本比 MDS 码稍大，但是通常修复所需要的源数据量较少。Clay 码 [35] 也是 MDS 码，它通过 MDS 码来构造出再生码，具有最小的修复带宽和磁盘 I/O。蝴蝶码 [36] 是提供双重故障的系统再生码。本地可修复码 [14, 25] 在大多数故障情况下可以比 RS 码减少修复流量，能够提供更好的修复性能同时保证相同的容错能力；但是本地可修复码具有更高的存储开销，它们也不是 MDS 码。

2. 基本性能指标

虽然纠删码在数学上有很好的性质以及形式化定义，但是部署在实际的存储系统中涉及到存储、计算以及网络等各个模块，所以真正在存储系统中部署纠删码需要关注于许多方面的性能指标。一般来说，纠删码存储系统主要关注以下基本性能指标：

- **存储效率：**原始数据占整个条带数据的比例。例如：RS(k,m) 码的存储效率为 $k/(k+m)$。
- **容错能力：**系统能够保证数据不丢失时最大容忍故障的能力。例如：RS(k,m)

码可以容忍任意的 m 个节点同时发生故障。

- **编码性能：**纠删码执行原始数据块编码生成校验块的性能。例如：RS(k, m)码将原始 k 个数据块编码得到 m 个校验块的性能。
- **解码性能：**使用可用数据块执行解码得到原始数据块的性能。例如：RS(k, m)码使用任意 k 个块解码得到原始的 k 个数据块。

同时保证存储效率高以及容错能力高，需要设计更为复杂的编解码算法，但是也带来了更高的编解码复杂度。不同种类的纠删码以及差异化的参数可以提供不同维度的性能保证，但是目前不存在一种纠删码能够保证存储系统在各个维度上都能做到最优。

3. 主要操作类型

除服务通常用户的正常读写操作之外，纠删码因其校验块可以提供更多的功能以及操作类型。一般来说，纠删码存储系统需要支持以下主要操作类型：

- **正常读：**本地用户读取远端节点上的一个可用的块或者完整的文件。
- **写：**本地用户向远端节点写入一个条带数据（包含数据块以及校验块）。
- **降级读：**本地用户读取远端节点上的临时不可用的一个数据块。
- **故障修复：**系统修复单个或者多个故障节点上的数据。
- **校验块更新：**本地用户更改了原始数据块时，需要将该数据块关联条带中的校验块对应更新。

纠删码存储系统的操作类型往往涉及到系统的各个环节，不同操作类型的性能瓶颈也不同，需要从系统的各个方面协同优化。下一小节，我们详细的拆解基于纠删码存储的系统流程，从而更深刻的理解纠删码部署的关键环节。

1.2.3　基于纠删码存储的系统流程

纠删码的形式化定义较为简单，但是它的操作 I/O 流程涉及到存储系统的存储、计算以及网络，所以真正在存储系统中部署纠删码面临着许多性能瓶颈以及挑战。我们以故障修复作为典型代表来介绍纠删码系统部署时的工作流程，其他的操作（读、写以及降级读）与故障修复有着类似的或者相反的 I/O 操作。当修复一个故障块时，它首先需要选择一个替代节点，该节点没有可用块（非故障块），然后需要选择 k 个源节点参与修复，如图 1.5 所示，总的故障修复流程共包括四个阶段：

(1) **读取源节点磁盘中的辅助数据：**将参与修复的源数据块从源节点的磁盘中读到内存中；
(2) **跨节点传输源数据：**替代节点跨节点读取源节点内存中的 k 个源数据块；
(3) **替代节点执行解码：**替代节点基于源数据块来执行解码计算，解码完成的块保存在本地内存中；

(4) **将解码完成的数据写到替代节点磁盘中：**替代节点将解码完成的块写到本地的磁盘中，完成整个修复过程。

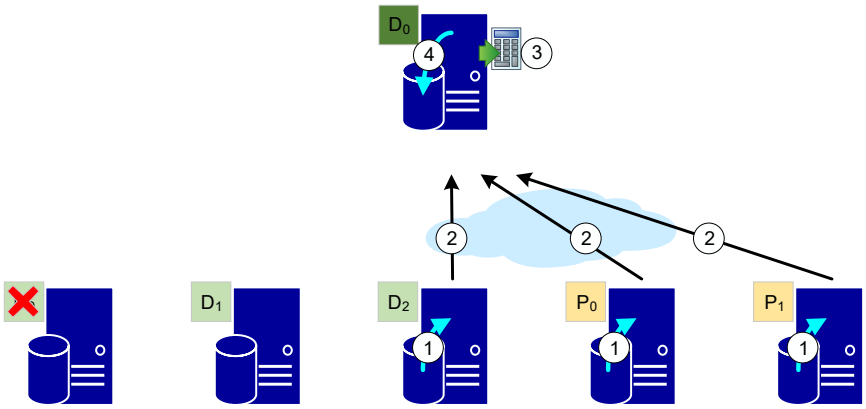


图 1.5　纠删码故障修复流程分析

我们将上述各阶段花费的时间分别表示为 t_1, t_2, t_3 以及 t_4。设块大小为 B，源节点 s 的读磁盘带宽为 $B^s_{I/O}$，替代节点 r 的写磁盘带宽为 $B^r_{I/O}$，网络带宽为 B_w。修复时间可以估计为 $t = t_1 + t_2 + t_3 + t_4 = \underset{s}{max}(\frac{B}{B^s_{I/O}}) + \frac{kB}{B_w} + t_{decoding} + \frac{B}{B^r_{I/O}}$。通过上述的分析，我们发现纠删码部署涉及到存储系统多个节点的的存储、计算以及网络等阶段，所以任何环节的阻塞将带来整个系统流程的低效，纠删码的系统部署面临着许多的挑战。

1.3　高可靠存储系统相关研究工作

近些年来，学术界以及工业界关注于高可靠存储系统并开展了许多的研究以及提出了许多的优化策略。我们下面将从数据布局优化，修复任务调度优化以及分离内存系统优化三个方面来作归纳与总结。

1.3.1　数据布局优化

数据布局是在磁盘或服务器上放置数据/校验块或者副本的方式，这在阵列存储系统（RAID）和分布式存储系统的性能和可靠性中都起着至关重要的作用。有一些工作可以通过优化数据布局 [37-41] 来提高 RAID 性能。Muntz 和 Lui [39] 首次提出 Parity declustering [37]，它主要利用平衡不完全区组设计（BIBD）的组合工具来提供负载均衡的访存性能，它需要额外的几块幸存磁盘来存储修复后的数据，可以保证数据修复时的读取负载在所有幸存磁盘之间是均衡的。Holland 和 Gibson 在实际系统中进一步实现了 Parity declustering [37]。SODP [41] 是对 Parity declustering 的进一步优化，可以更好的保证关联故障事件。Parity declustering 将修复后的数据存储到幸存的磁盘中，但是数据修复仍然受到替代磁盘的写入速

度的限制。RAID+ [38, 42] 利用正交拉丁方的组合工具将数据修复时的读和写操作均匀扩展到所有磁盘上，因此它进一步加快了磁盘阵列存储系统的修复速度。

对于分布式存储系统中的数据布局，EAR [43] 是一种有利于从多副本到纠删码有效转换的数据放置算法。对于可扩展的存储系统，还存在一些基于哈希的数据放置方法。[44] 使用伪随机哈希函数在动态变化的存储区域中均匀分布和有效定位数据。Chord [45] 使用一致性哈希 [46] 的变体为存储节点分配密钥，以有效适应系统中节点的动态变化。SCADDAR [47] 使用伪随机算法为所有磁盘分配块，并且在系统磁盘扩容情况下最小化迁移数据量。RUSH [48] 和 CRUSH [49] 利用基于伪随机的哈希函数将副本的对象映射到存储设备的可扩展集合。当新的存储设备添加到系统中时，这些方法在概率上均匀地分布数据并最大程度地减少数据迁移。这些基于哈希的放置策略旨在提高分布式系统中的可扩展性。

1.3.2 修复任务调度优化

近些年来,有一些关于分布式系统中负载均衡的副本修复调度的工作 [18, 50-51]。Dayu [50] 是一种贪心的调度算法，用于基于多副本的分布式存储系统中的数据修复。Dayu 只需在每个故障块的一个源节点（即存放其中一个副本的节点）和一个替代节点之间创建连接，所以它不适用于基于纠删码的环境。因为在部署 (k, m) 纠删码的分布式存储系统中修复故障块时，需要进行 k 个源节点与一个替代节点的连接调度，调度开销太大。

此外，还有一些关注于纠删码存储系统的修复流量调度。S3 [52-53] 是一种考虑修复任务具有修复期限的在线修复调度算法。HDFS [24] 文件系统采用随机的调度修复算法，也就是随机选择源节点和替代节点参与故障修复。CAR [54-55] 提出了一种贪心调度算法，主要考虑可以使用备份节点进行纠删码故障修复的存储系统，首先从任务队列中依次初始化一批任务，然后逐一为每个任务选择源节点，从而可以避免在有限次迭代中出现读负载过重的节点。该算法可以均衡来自源节点的跨机架修复流量，从而加快修复过程。ECPipe [56] 和 PPR [57] 主要考虑优化降级读场景或者单块修复任务的场景。而对于单节点故障修复场景，这些方法主要通过逐个添加任务并贪心地选择当前 k 个负载最轻节点做为源节点，将任务打包为一批来执行修复。

1.3.3 分离内存系统优化

当前工业界对分离内存架构的研究已取得一定进展，越来越多的企业针对自己面临的业务场景的特性，开始关注分离内存架构 [58]。Intel 提出了一种新的基于 “片上光互联” 技术的机架式架构设计 (RSA) [59]，它将机架内的服务器、网络以及存储等部件通过光纤等设备和软件调度进行整合，分离成各类资源池，根

据具体应用需求为每个节点动态分配所需资源，从而有效提升资源的利用率。微软公司通过利用硬件 FPGA 来取代部分软件工作，并配置一种高度灵活的可重组的网络互联技术 Catapult [60]，解耦硬件加速器资源，为应用配置专门的硬件加速器资源从而起到加速作用并减轻服务器的计算压力。Gen-Z [13] 标准在 2016 年发布，初始有 AMD、ARM、Broadcom 以及华为等 12 家公司加入，旨在提出一种开放系统互连的内存访问技术。在该技术下，网卡、交换机以及其他的设备可以通过直连的方式简化数据链路上的软件栈，从而保证高效的内存访问。

在学术界，分离内存架构也得到国内外许多学者的关注。针对集群内存资源竞争激烈和总体利用率低下的问题，Lim 等人 [61] 提出了分离内存的思想。INFINISWAP 架构 [62] 通过在集群中每台服务器节点上设置专门的空闲内存资源管理程序，通过利用 RDMA 网络技术，仅通过更改软件层面就实现了分离内存。在国内，中国科学院计算所提出了一种 Venice 松耦合数据中心架构 [63]，该架构通过使用 PCIe 协议和芯片技术来构建多节点系统，从而实现内存、网卡以及 GPU 等各类资源的共享，提升了内存访问带宽及访问延迟性能。除此之外，之前的工作研究了分离内存的多个方面，包括内存管理 [9, 64-69]、网络优化 [13, 70-75]，架构设计 [61, 76]，操作系统设计 [77-78] 以及数据库索引设计 [79] 等方面。Hydra [80] 是为远程内存提供高可用的纠删码部署，主要关注于小对象的场景并以换页机制读取远程内存数据。

1.3.4 现有工作的不足

尽管现有大量工作提出了许多方法来优化高可靠存储系统，但是随着多用户以及多应用场景的新需求，使得这些方法在均衡数据布局、修复任务调度以及分离内存架构下纠删码部署等方面存在不足。

均衡数据布局技术的不足：尽管现有许多的研究工作关注于数据布局的优化，但是他们主要的场景是关注于磁盘阵列存储系统，而在分布式存储系统中关于均匀的数据放置策略较少。然而磁盘阵列存储系统中的磁盘组织是扁平的，而分布式存储系统中通常用树型结构的拓扑组织存储节点。另外分布式存储系统中的网络带宽是异构的，其中机架内带宽足够，而跨机架带宽经常被超额使用。此外，在分布式存储系统中，每个服务器都具有存储资源和计算能力，因此每个服务器都可以参与编码/解码计算，而对于磁盘阵列存储系统，只能在连接磁盘的单个服务器上执行计算。分布式存储系统与磁盘阵列存储系统的架构以及场景不同，关于故障修复性能的需求以及瓶颈是不同的，因此磁盘阵列存储系统的数据布局优化工作不能直接应用到分布式存储系统中。

修复任务调度技术的不足：尽管之前的一些工作关注于修复任务调度，但是他们主要考虑一些特殊场景的修复调度，比如关注副本修复调度以及同构网络

拓扑下的调度。现有的分布式存储系统的故障事件往往是复杂的，常常存在前端的工作负载造成异构的环境，也存在并发多节点故障。现有的研究工作无法涵盖存储系统复杂的应用场景，对于多用户需求的分布式系统不友好，在同构和异构网络环境中缺少通用的修复调度解决方案来支持单节点故障和多节点故障，所以难以保证系统故障修复性能。

分离内存架构下纠删码部署：在分离内存架构下，大家暂时关注于内存管理、网络优化以及架构设计等方面，没有仔细的研究纠删码部署所带来的问题。传统外存或者内存存储系统中，主要工作集中在降低跨节点网络流量和磁盘 I/O 的高开销等方面，或者利用纠删码的天然特性，例如低成本和小块的高并发特性，用于具有特殊应用需求的场景中。但是在分离内存的架构下，低网络时延可能带来纠删码部署的新瓶颈。另外，新的系统架构需要重新设计纠删码工作流程。使用纠删码来保证分离内存系统的可靠性将面临着编解码与网络友好的读写流程、元数据管理以及故障修复等亟待解决的问题。

1.4　本文的主要研究内容

本文关注于纠删码容错存储系统的编解码流程优化研究，主要研究内容以及各部分间的关系如图 1.6 所示，主要包括：基于纠删码存储系统的均匀数据布局设计、基于纠删码存储系统的故障修复任务调度设计以及基于分离内存架构的纠删码流程设计。具体内容概括如下：

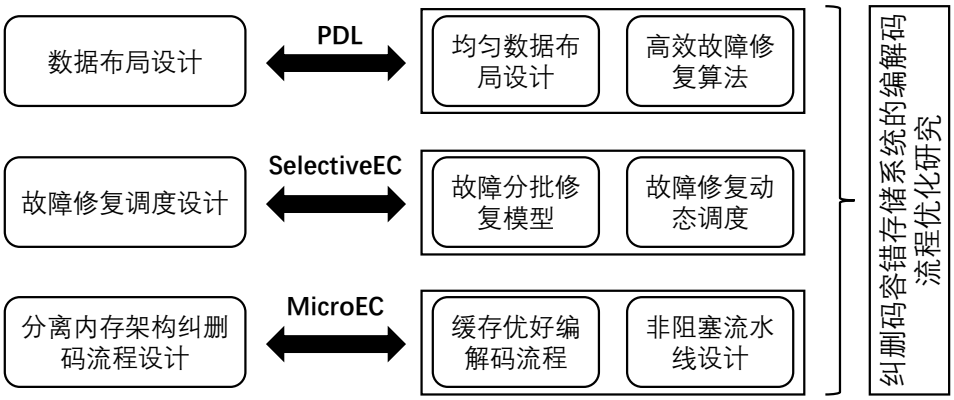


图 1.6　本文主要研究内容

1.4.1　基于纠删码存储系统的均匀数据布局设计

分布式存储系统中故障修复速度受限于以下因素：(1) 大规模分布式存储系统通常使用树型拓扑结构组织节点在多个机架中，导致存储系统中可用的跨机架带宽比机架内带宽少得多。繁重的跨机架流量严重减慢了修复过程。(2) 系统将同一条带的块随机放置到尽量多的机架上，以保证最大的机架级容错。它在具有大量条带的分布式存储系统中能实现负载均衡。但是，由于内存、I/O 和 CPU

等资源有限，我们只能逐批修复数据，每批由有限的条带组成。通常，一个批次内的修复工作量是严重不均衡的。(3) 为了满足用户不同的可靠性需求以及异构应用的工作负载，多个纠删码策略共存于分布式存储系统中，导致数据分布更加复杂，故障修复更加困难。针对上述分布式存储系统面临的三个方面问题，我们研究并提出了纠删码策略的均匀数据布局 PDL 以及相应的修复方案 rPDL。

具体来说,我们首先基于数学上的组合设计工具成对平衡设计（PBD, Pairwise Balanced Design）提出了一种用于混合纠删码存储的均匀数据布局 PDL（PBD-based Data Layout)。PDL 实现了机架级数据的均匀分布，并且容忍多节点故障或者单机架故障，从而满足不同应用的需求。然后，我们提出了一种基于 PDL 的的修复方案 rPDL，均匀选择替代节点并读取确定的可用块以修复丢失块。rPDL 在修复过程中实现了有限条带内的读写 I/O 均衡性和高并行度。最后，我们进一步基于 rPDL 设计了机架内解码策略，该策略大大减少了修复过程的跨机架流量，从而缓解了跨机架带宽的竞争并加速了修复。由于 PDL 以及 rPDL 具有均匀的块分布，通过修复过程中几乎均衡的读写 I/O 以及设计的机架内局部解码，rPDL 实现了高并行度的故障修复，并大大减少了跨机架流量。因此，与传统的随机数据放置相比，它明显加快了修复过程。

1.4.2　基于纠删码存储系统的故障修复任务调度设计

现有分布式存储系统常随机分布数据块/校验块，在故障修复过程中随机选择源节点和替代节点。一方面在系统上实现简单；另一方面，分布式存储系统中有大量的数据块/校验块，从统计的角度可以达到节点之间数据分布和工作负载的均衡。但由于内存容量、网络带宽、CPU 计算能力等方面的限制，故障盘中数据的修复过程是分批次执行的。现有的修复方法将序列号连续的一些待修复的数据打包成一个批次。因为每个批次中待修复的数据量有限，随机分布数据块/校验块会导致每批次内各节点负载严重不均衡，而随机选择源节点和替代节点则会加剧这种不均衡性。此外，异构的存储系统环境，动态变化的工作负载和流量也会对此产生影响，最终这种不均衡会明显拖慢修复过程。在分布式存储系统中，数据修复工作仅能使用有限的网络带宽，另外不同节点上故障修复的工作负载是不均衡的，这也会减慢整体修复。基于前面的负载不均衡问题，我们从修复流量调度的角度出发，设计了修复调度模块 SelectiveEC。

具体来说，我们首先分析了常用的分批修复机制，发现它可以加速故障修复，但在批处理中面临严重的修复负载不均衡。然后我们提出了一个二分图模型和一个批处理算法来将修复任务分批处理并选择源节点来均衡节点之间的上游修复流量，主要思想是不采用将连续序列号的多个条带打包到一个批次的方式，而是从大量待修复的数据中挑选一些，将这些条带打包到一个批次，结合对源节

点的确定性选择算法，达到从各个节点读取源数据的均衡。最后，我们使用另一个二分图模型和最大匹配算法来选择替代节点以均衡其下游修复流量和解码负载，可以做到确定性地选择替代节点，使得各个节点间解码的工作量以及已修复的数据量达到均衡。均衡调度模块 SelectiveEC，可以在分布式存储系统中进行有效的故障修复，并支持修复多节点故障，在异构环境以及网络和磁盘 I/O 带宽等资源动态变化的情况下同样适用。SelectiveEC 一方面可以加快故障修复的速度，另一方面可以在故障修复时对前端应用的性能干扰更小。

1.4.3　基于分离内存架构的纠删码流程设计

在传统基于磁盘的系统中广泛使用多副本策略来提供简单的容错，但它带来了高存储成本。在分离内存系统下，DRAM 和持久化内存都比传统磁盘更昂贵，另外由于写入冗余的数据副本，它还引入了额外的写入时延。纠删码是一种替代方案，提供了相同级别的容错能力和更小的存储开销。但是，当在分离内存系统中部署纠删码时，随着单边 RDMA 实现微秒级的低时延，由于编解码的高计算成本，我们面临着新的挑战和瓶颈。通过我们仔细分析了将纠删码部署在分离内存中的编解码函数栈和 RDMA 传输过程，我们发现有多种因素导致高写入/读取时延，它们在降低编解码计算和网络传输时延提供了优化机会。首先，由于数据已经在计算服务器的内存中，通过逻辑上拆分数据进行细粒度的数据条带化，使其与 CPU 上 L1 cache 对齐，可以大大加快编解码速度。其次，通过重用缓存中第一个条带的编解码辅助数据，例如编解码矩阵和乘法表，重新设计编解码函数栈，也可以大大减少后续条带的编解码时间。最后，编解码计算和 RDMA 传输的最佳适配性能很大程度上取决于对象的大小，因此仔细协调可以使这两种操作达到最高的速度，并使它们以流水线方式良好地工作。基于上述分析，我们设计了一种新的纠删码方案 MicroEC。

具体来说，我们首先从缓存效率以及 RDMA 传输的角度，彻底分析了纠删码部署的软件栈。我们发现了在分离内存系统中部署纠删码时的关键性能瓶颈。我们提供了三个关键系统发现，用于在系统级别优化编解码和 RDMA 传输工作流程。然后我们通过利用缓存对齐的条带化和重用编解码辅助数据来加速编解码，重新设计了编解码函数栈。进一步还提出了有效的数据结构来支持新的设计。最后，我们设计了非阻塞流水线进行编解码和 RDMA 传输，并仔细调整编解码和网络传输对象的大小，使得流水线达到最大化的并行度。通过仔细实现这些技术和多项优化，MicroEC 大大减少了编解码时延并使其与单边 RDMA 速度相匹配，实现了比现有纠删码和副本技术更低的访存时延。

1.5　本文的组织结构

本文总共分为五章，章节组织结构如下：第一章是绪论，介绍了高可靠存储系统的相关背景以及本文的主要研究内容。第二章研究了纠删码存储系统的数据布局，通过设计纠删码存储策略下的关于系统正常与故障状态下的均匀数据布局，从而达到负载均衡的修复负载。第三章研究了纠删码存储系统的故障修复任务调度，通过设计修复调度模块，从而达到分批任务负载均衡的修复状态。第四章研究了分离内存系统架构的纠删码流程设计，通过设计编解码计算与网络传输友好的纠删码工作流程，从而系统达到部署纠删码后的高效访存性能。第五章总结了本文的研究内容，并指出了本文的不足之处以及对未来工作的展望。

第 2 章　基于纠删码存储系统的数据布局设计

2.1　引言

为满足指数级增长的数据存储需求，由众多存储节点组成的分布式存储系统（Distributed Storage System, DSS）应运而生，例如谷歌的 GFS [3]、Ceph [4]、Apache 的 HDFS [5] 和微软的 Azure [6]。由于预算有限，分布式存储系统通常使用商用设备构建，这会导致故障的频繁出现 [3, 5, 7]。多副本策略是一种提高可靠性的传统方法，通过在 N 个不同存储节点中保留相同数据副本来提供容错。多副本策略的实现简单，并且可以提供对相同数据的并行访问，但是 N 副本的存储开销是 N 倍，这对于海量的数据存储需求来说是不可忍受的。纠删码作为多副本的一种替代策略，它能够提供同级别的容错能力，但存储开销要低得多。(k, m) 最大距离可分码 (MDS 码) 将 k 个数据块编码为 m 个校验块，同时 $k + m$ 个块形成一个条带。(k, m) MDS 码在存储开销为 $(1 + m/k)$ 倍的情况下最多可以容忍 m 个并发的故障。然而纠删码会导致高修复成本，比如 (k, m) MDS 码必须访问 k 个源数据块，这会产生大量修复流量，从而导致更长的数据修复时间。缓慢的修复过程增加了数据不可靠的时间窗口大小和数据二次故障的可能性，并且在很长一段时间内也会降低系统正常的访存性能。因此，加速故障修复对于部署纠删码的存储系统来说至关重要。

由于下面的一些原因，分布式存储系统中的故障修复过程通常很慢：(1) 大规模分布式存储系统通常使用树型拓扑架构，将节点组织在多个机架中。这些分布式存储系统中可用的跨机架带宽比机架内带宽少得多。大量的跨机架流量严重减慢了修复过程。(2) 系统将同一条带的块随机放置到尽可能多的机架上，以保证最大的机架级容错。它在具有大量条带的分布式存储系统中可以实现存储上的负载均衡。但是，由于存储系统 I/O 和 CPU 等资源有限，我们只能逐批修复数据，每批由有限的条带组成。通常，一个批次内的修复工作量是严重不均衡的。(3) 为了满足用户不同的可靠性需求，适应异构应用的工作负载，通常会在分布式存储系统中部署多个纠删码策略，这会导致数据分布更加复杂，故障修复更加困难。

本章针对上述三个方面分布式存储系统中面临的问题，借助于组合设计工具成对均衡设计（PBD, Pairwise Balanced Design），为具有混合纠删码的分布式存储系统提出了一种均匀数据布局：*PDL*（PBD-based Data Layout）和相应的修复方案：*rPDL*。得益于块的均匀分布，修复过程中几乎均衡的读写 I/O 以及设计的机架内局部解码策略，rPDL 实现了高并行度的故障修复，并大大减少了跨机架流量。因此，与传统的随机数据放置相比，它显然加快了修复过程。本章贡献

可以总结如下：

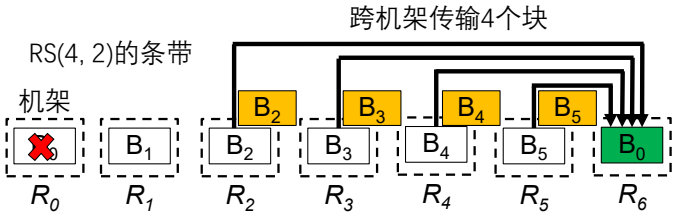
- 基于组合设计工具成对均衡设计（PBD）提出了一种用于多纠删码的均匀数据布局 PDL。该数据布局实现了机架级数据的均匀分布，并且容忍多节点故障和单机架故障，从而满足了不同应用的需求。
- 提出了一种基于 PDL 的高效修复方案：rPDL，它通过确定性地选择替代节点以及可用的源数据块来修复故障块。rPDL 在修复过程中实现了有限条带内的读写 I/O 均衡性和高并行度。
- 设计了机架内局部解码算法，大大减少了修复过程的跨机架流量，从而缓解了跨机架带宽的竞争并加速了修复过程。
- PDL 和 rPDL 在 HDFS 3 中实现，并在 28 节点的服务器集群上进行了实验。与 HDFS 现有的数据布局相比，PDL 将单个不可用数据块的降级读延迟平均减少了 62.83%。对于单节点故障修复，平均减少了 63.73% 的跨机架流量，实现了更好的跨机架流量负载均衡。与此同时，它平均实现了 6.27 倍的单节点故障修复吞吐率。此外，rPDL 在多节点故障和单机架故障的修复中达到了 HDFS 的 5.14 倍和 1.48 倍吞吐率。几个代表性的 MapReduce 工作任务也显示了 PDL 和 rPDL 对前端应用程序有着更好的支持。

本章其余部分组织如下：我们首先在第 2.2 节对数据布局的非均衡性问题进行了描述以及分析；然后在第 2.3 节提出了基于树型拓扑的纠删码均匀数据布局 PDL 的设计；在第 2.4 节介绍了基于 PDL 的负载均衡的故障修复方法 rPDL 的设计；在第 2.5 节和第 2.6 节分别介绍了 PDL 以及 rPDL 的系统实现细节以及性能评估；最后，在第 2.7 节对本章总结。

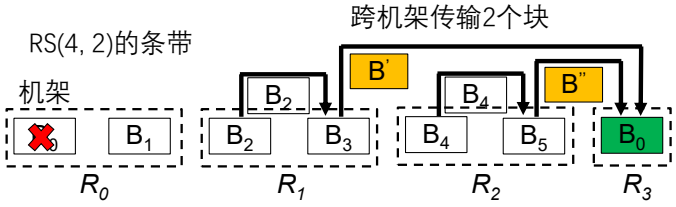
2.2　数据布局的非均衡性问题描述

当存储系统发生节点故障时，需要修复该节点上的所有数据块与校验块，这会导致非常高的 I/O 负载和修复流量。除了用于故障修复的必要的 I/O、流量和 CPU 工作负载外，以下因素进一步减慢了分布式存储系统中的故障修复过程。

用于故障修复的大量跨机架流量：如果系统的机架数量足够多，现有的做法会对每一个条带采用 “一个机架一个块” 的方式，从而达到最大的机架级别的容错。然而，这样的存储方式也会导致故障修复时产生大量的跨机架流量。如图 2.1(a)，假设 6 个块 $B_0, B_1, B_2, B_3, B_4, B_5$ 形成一个 $(4, 2)$ 码的条带，每一个块都存储在单独的机架中，假设机架 R_i 中对应存放 B_i 块，$i = 0, \cdots, 5$。如果存储 B_0 的节点出现故障，则会从不同于 R_i（$i = 0, \cdots, 5$）的机架中选择一个替代节点 N。节点 N 将跨机架读取 B_1, B_2, B_3, B_4, B_5 中的 4 个块来修复 B_0。假设故障节点上原本有 2TB 的数据，则整个修复过程将导致总共 8TB 的跨机架流量。



(a) 优化前的策略



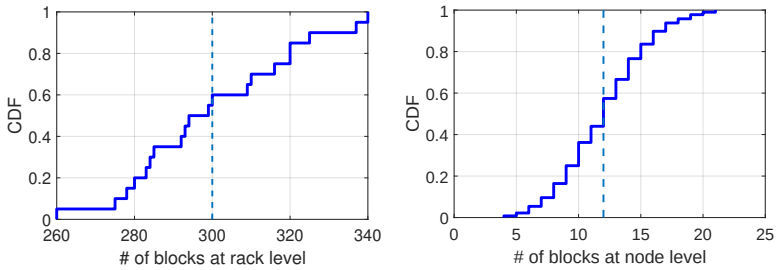
(b) 优化后的策略

图 2.1　跨机架流量示意图

考虑到实际的分布式存储系统中很少发生两个机架并发故障，我们可以放宽一个机架中只有一个块的限制，将一个条带的多个块放在同一个机架中。因此，现在的主要目标是如何在系统中放置条带中的 $k+m$ 块，从而最大限度地减少跨机架修复流量，同时容忍单机架故障和 m 个并发节点故障。如图 2.1(b)，我们可以将 6 个块 $B_0, B_1, B_2, B_3, B_4, B_5$ 存储到 3 个机架中，每个节点一个块。假定 B_0 和 B_1 存放在机架 R_0 中，B_2 和 B_3 存放在机架 R_1 中，B_4 和 B_5 存放在机架 R_2 中。在修复 B_0 块时，如果我们在不同于 R_i（$i = 0, \cdots, 2$）机架中选择一个替代节点 N，并在 R_1 和 R_2 中采用局部解码得到对应的聚合块，节点 N 只需要跨机架读取两个聚合块，节省了一半的跨机架流量。也就是说，修复故障节点的 2TB 数据只会导致 4TB 的跨机架流量。一般而言，对 (k, m) 码的条带按上述方式放置并采用局部解码，单节点故障修复的跨机架流量大约可以减少到随机数据布局的 $\frac{1}{m}$。

块的不均匀分布：从概率的角度来看，随机的数据分布在存储条带数量足够多的时候会达到近似均匀。但是由于 I/O、网络带宽和计算资源的限制，待修复数据是被分批修复的。每个批次由有限的条带组成，条带的数量通常与节点数量处于同一数量级。由于一批内的条带数量相对较少，其批次内的块分布很难均匀分布。图 2.2 显示了分布式存储系统中块分布的累积分布函数图（CDF），我们将 1000 个条带（HDFS 中默认设置为节点数的两倍）分配给拥有 500 个节点和 20 个机架的分布式存储系统。图 2.2(a) 中展示了块在机架级的分布情况，负载最重的机架有 340 个块，比负载最轻的机架 260 块多 30.77%。图 2.2(b) 展示了节点级的块分布，其中负载最重的节点中有 21 块，而负载最轻的节点仅有 4 个块，二者差距有 5.25 倍。由此可见，节点级的块分布比机架级的更加不均衡。HDFS 中默认块大小为 128MB，假设系统使用 1Gbps 的交换机和 1/20 × 1Gbps 的可用

跨机架带宽 [7-8]，负载最重的机架修复用时（共 5324.8s）比负载最轻的机架慢 $(340-260)\times 128\text{MB}/(1/20\times 1/8\times 1000\text{MB/s}) = 1638.4$ 秒。因此，每一批修复过程中，随机数据分布会导致数据布局的严重不均匀，从而拖慢修复过程。此外，具有不同条带大小和不同容错能力的多种纠删码混合会使得系统更难达成均匀的块分布。



(a) 机架级块分布　　(b) 节点级块分布

图 2.2　模拟 HDFS 系统的机架级以及节点级块分布的 CDF 图

用于故障修复的跨机架流量不均衡: 在故障修复过程中，每批需要参与修复的替代节点的选择在负载均衡中也起着至关重要的作用。如果替代节点聚集在一些机架中，那么这些替代节点同时读取数据将大大增加它们的架顶交换机（ToR）的跨机架下行链路负载。HDFS 默认选择故障节点附近的节点作为替代节点，使得这种流量不均衡的情况非常容易出现。此外，如果源数据块主要集中在少量机架中，那么将从这些机架读取大量数据参与故障修复，这会严重增加它们架顶交换机（ToR）的跨机架上行链路负载。在性能评估一节中，通过统计各个机架跨机架修复的读（或写）流量的最大值与平均值的比，记作 λ_r（或 λ_w），从而来评估跨机架修复的读（或写）的负载均衡程度。对于 HDFS 默认的单节点故障修复，$\lambda_r = 1.4278$ 以及 $\lambda_w = 5.4592$，而 rPDL 则可以达到 $\lambda_r = 1.0447$ 和 $\lambda_w = 1.2955$，其中 $\lambda_r = 1$ 和 $\lambda_w = 1$ 代表着所有机架之间读写的完美均衡。

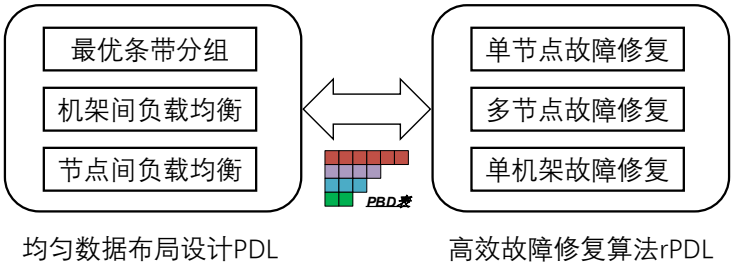


图 2.3　PDL 和 rPDL 的整体架构

综上所述，我们将首先设计一个均匀的数据布局 PDL，支持分布式存储系统中的多纠删码策略；然后设计一种高效的修复方案 rPDL，从而在故障修复时减少以及均衡跨机架修复流量。图 2.3 描述了 PDL 以及 rPDL 的整体架构，主要包括*数据布局设计 PDL* 以及*高效故障修复算法 rPDL* 两部分。首先，为了减少以及均衡修复时跨机架流量，*数据布局设计 PDL* 将一个条带的所有块分成若干组，

利用 PBD 表来将条带分好的组放置到机架上，并将同一组中的块放置到同一机架的节点上，从而实现了机架间以及机架内均匀的块分布。然后，为了最小化修复时的跨机架流量以及均衡修复负载，高效故障修复算法 *rPDL* 一方面可以在故障修复过程中在机架内部署局部解码，从而减少跨机架流量；另一方面可以在故障修复过程基于 PDL 来达到均匀的修复负载，并且支持单节点故障修复、多节点故障修复以及单机架故障修复。我们将在下面的章节中详细的介绍这两部分的内容。

2.3　一种基于树型拓扑的纠删码均匀数据布局设计

在本节中,我们首先介绍了纠删码的线性性质,然后介绍成对均衡设计（PBD）和均衡不完全区组设计（BIBD）的基本概念和属性，它们是实现本节数据布局的基础工具。

2.3.1　预备知识

纠删码线性性质： (k,m) 码的线性性质使其具有进行局部解码的可能性。以图 2.4 中 $(4,2)$ 码条带为例。假设 $B_0 = c_1B_1 + c_2B_2 + c_3B_3 + c_4B_4$，（其中 c_i 为纠删码特定的解码系数，这里的 $i = 1,2,3,4$）。由于加法满足结合律，我们可以先将 $c_1B_1 + c_2B_2$、$c_3B_3 + c_4B_4$ 局部解码为 B^* 和 B^{**}，然后得到 $B_0 = B^* + B^{**}$。因此，同一条带中被读取到的多个块可以在机架内进行局部解码，然后再跨机架传输块以减少整体的跨机架流量。

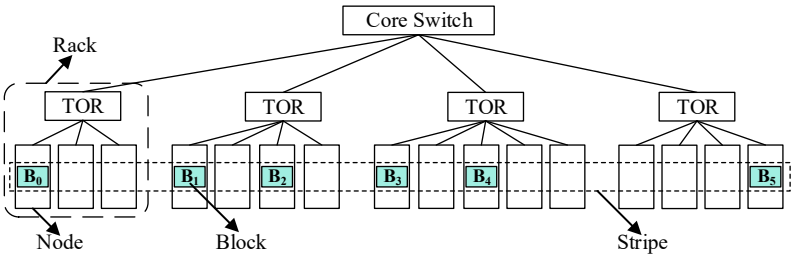


图 2.4　分布式存储系统中一个 $(4,2)$ 码条带的块分布示例图

PBD 与 BIBD： PBD（Pairwise Balanced Design）和 BIBD（Balanced Incomplete Block Design）是组合理论中的两类重要设计，被广泛用于软件测试以及存储系统数据布局等领域 [81-83]。本章中，我们使用 PBD 设计存储系统的纠删码数据布局，并通过多个 BIBD 来构造 PBD。BIBD 的定义如下:

定义 2.1　给定 5 个正整数：b,v,κ,r,λ，以及 v 个元素的集合 $S = \{s_0,s_1,\cdots,s_{v-1}\}$，一个基于 S 的 (b,v,κ,r,λ)-BIBD 是集合 S 的 b 个子集（也称为元组）构成的子集簇 $\mathcal{T} = \{T_0,T_1,\cdots,T_{b-1}\}$, 满足以下的条件:

(1) $T_i \subseteq S$ 以及 $|T_i| = \kappa$（$0 \leqslant i \leqslant b-1$），

(2) 每个元素恰好出现在 r 个元组中，

(3) 每一对的元素恰好同时出现在 λ 个元组中。

$(v, \mathcal{K}, \lambda)$-PBD 是一组 v 个元素的子集簇，使得任何两个元素都恰好出现在 λ 个元组中。它与 BIBD 的不同之处在于元组大小来自一组正整数 $\mathcal{K}$，而不是一个固定的正整数。PBD 的定义如下：

定义 2.2　给定两个正整数 v, λ，以及一个集合 $\mathcal{K} \subset \{k : k \geqslant 2\}$，一个 $(v, \mathcal{K}, \lambda)$-PBD 是一个 v 元集合 S 的子集簇 $\mathcal{A}$，满足下面的条件：

(1) $|S| = v$，

(2) $A \subseteq S$ 以及 $|A| \in \mathcal{K}$，这里所有 $A \in \mathcal{A}$，

(3) 每一对的元素恰好同时出现在 λ 个元组中。

$(v, \mathcal{K}, \lambda)$-PBD 可以由具有相同参数 v 的一组 BIBD 构造。例如，$(b_1, v, \kappa_1, r_1, \lambda_1)$-BIBD 和 $(b_2, v, \kappa_2, r_2, \lambda_2)$-BIBD 形成一个 $(v, \{\kappa_1, \kappa_2\}, \lambda_1 + \lambda_2)$-PBD，并且每个元素都在 $r_1 + r_2$ 个元组中。在图 2.6(b) 中，$(15, 6, 4, 10, 6)$-BIBD 和 $(10, 6, 3, 5, 2)$-BIBD 形成一个 $(6, \{3, 4\}, 8)$-PBD，每个元素出现在 15 个元组中，并且任意两个元素同时出现在 8 个元组中。我们可以使用 PBD 来设计多个纠删码的数据布局。PBD 中的 v 个元素对应于分布式存储系统中的机架，集合 $\mathcal{K}$ 中的元素对应于使用不同纠删码编码的条带分组数。PBD 中的每个元组对应于单个条带，用于条带中的块放置在机架的编号寻址。一个 PBD 可以看作是地址映射表。根据 PBD 的定义和构造，所有的机架可以分配到相同数量条带的数据，提供了机架级均匀的数据布局。另外，参数 λ 表示任意两个机架同时分配到 λ 个条带中的数据，能够保证在故障修复时均匀的流量分布。

2.3.2　最优条带分组

对于采用 (k, m) 码的条带，为保证能够容单个机架或者任意 m 节点故障，系统放置块需要满足以下的规则：

- 同一条带的最多 m 个块被分配到同一机架以容忍单机架故障；
- 同一条带的任意两块都不会被放置到同一节点以容忍 m 个并发节点故障。

为了最大化局部解码的收益，条带的组数应当尽量的小。考虑到上述的第一个约束条件，得到 $N_g(k, m) = \lceil \frac{k+m}{m} \rceil$ 是容忍机架故障时的最小组数。我们称一个包含恰好 m 个块的组是满的，否则为未满的。为了使块分布均匀，需要将所有块都进行分组，使得任意两组中的块数量最多相差 1 个。因此，如果 $m \mid k$，例如 $k = tm$，则最小的组数 $N_g(k, m) = (k + m)/m = t + 1$，并且所有组都已满。否则，如果 $k = tm + q$ 且 $0 < q < m$，则 $N_g(k, m) = \lceil (k + m)/m \rceil = t + 2$，这时有 $m - q$ 个未满组，每组内有 $m - 1$ 个块，其余组是满的 [①]。

[①]实际上，如果某个组最多包含 $m - 2$ 个块，则 $m \geqslant q + t + 3$，即 $m \geqslant 5$。例如，$(22, 7)$ 码的条带有 5 个

一个 RS(10, 4) 码条带的组数为：$N_g(10,4) = \lceil \frac{10+4}{4} \rceil = 4$，分组示例如图 2.5 所示，其中 $\mathbf{G_0}$ 和 $\mathbf{G_1}$ 是满的，而 $\mathbf{G_2}$ 和 $\mathbf{G_3}$ 是未满的。

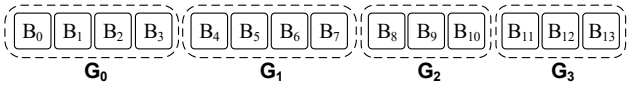


图 2.5　RS$(10, 4)$ **码的条带分组**

基于上述分组规则，对于 (k, m) 码，PDL 最多可以容忍 m 个并发节点故障或单机架故障。而 “一个机架一个块” 的放置策略最多可以容忍 m 个并发节点故障或 m 个并发机架故障，所以 PDL 牺牲了多机架级别的容错能力。然而在实际系统中，机架故障比节点故障少得多，而并发的多机架故障非常罕见，通常来说单节点故障在所有故障事件中占主导地位，占比超过 90% [7, 84]。实际上，代表性的 HDFS 系统默认配置三副本仅仅用来防止单机架故障 [85]，所以 PDL 满足分布式存储系统大多数应用的可靠性需求。

2.3.3　机架以及节点间的均匀数据分布

假设要在部署了 τ 种不同纠删码的分布式存储系统中放置块，其中第 i 种纠删码参数为 (k_i, m_i)，$0 \leqslant i \leqslant \tau - 1$。例如图 2.6 中使用 4 种纠删码：RS(4, 3), RS(6, 3), RS(10, 4) 和 RS(12, 4)。现在我们根据上一小节中讨论的分组规则构造一个 PBD 表，并根据 PBD 表将这些组分配到物理机架中。

对于 (k_i, m_i) 码，其条带分组的最小组数为 $\kappa_i = N_g(k_i, m_i)$。令 $\mathcal{K} = \{\kappa_i = N_g(k_i, m_i) | 0 \leqslant i \leqslant \tau - 1\}$ 是所有纠删码最小组数的集合。请注意，不同纠删码的最小组数可能相同。例如，RS(4, 3)、RS(6, 3)、RS(10, 4) 和 RS(12, 4) 的最小组数分别为 3, 3, 4, 4，因此集合 $\mathcal{K} = \{3, 4\}$。

假设分布式存储系统中有 v 个机架，我们可以根据 $(b_i, v, \kappa_i, r_i, \lambda_i)$-BIBD 来执行 (k_i, m_i) 码的分配，如图 2.6(c)。因此，如果分布式存储系统中有 6 个机架，我们可以为 RS(4, 3) 和 RS(6, 3) 选择 (10, 6, 3, 5, 2)-BIBD，以及为 RS(10, 4) 和 RS(12, 4) 选择 (15, 6, 4, 10, 6)-BIBD。将 $\kappa_i \in \mathcal{K}$ 的所有 $(b_i, v, \kappa_i, r_i, \lambda_i)$-BIBD 组合起来，将如图 2.6(b) 所示得到一个 PBD 表。

我们将 $b = \sum b_i$ 个条带分配到一个批次中，称为 *PBD 周期*，并使每个机架放置了 $r = \sum r_i$ 个组。请注意，在 PBD 表中，$\lambda = \sum \lambda_i$ 表示任何一对机架都包含在一个 PBD 周期的 λ 个组中。在图 2.6 中，一个 PBD 周期包含 25 个条带。为实现均匀的数据布局，我们根据 PBD 表将各个组分配到物理机架，然后将组内的块分配给机架内的节点。

(1) *将组映射到机架*。我们首先根据 PBD 表中的元组大小 $\kappa_i = N_g(k_i, m_i)$ 来为

未满组，每个组分别包含 6, 6, 6, 6 和 5 块。然而，在实践中，分布式存储系统通常使用带有 $m \leqslant 4$ 的 (k, m) 码来容忍最多 4 个节点的故障。所以下面我们只考虑每组包含 m 或 $m - 1$ 块的情况。

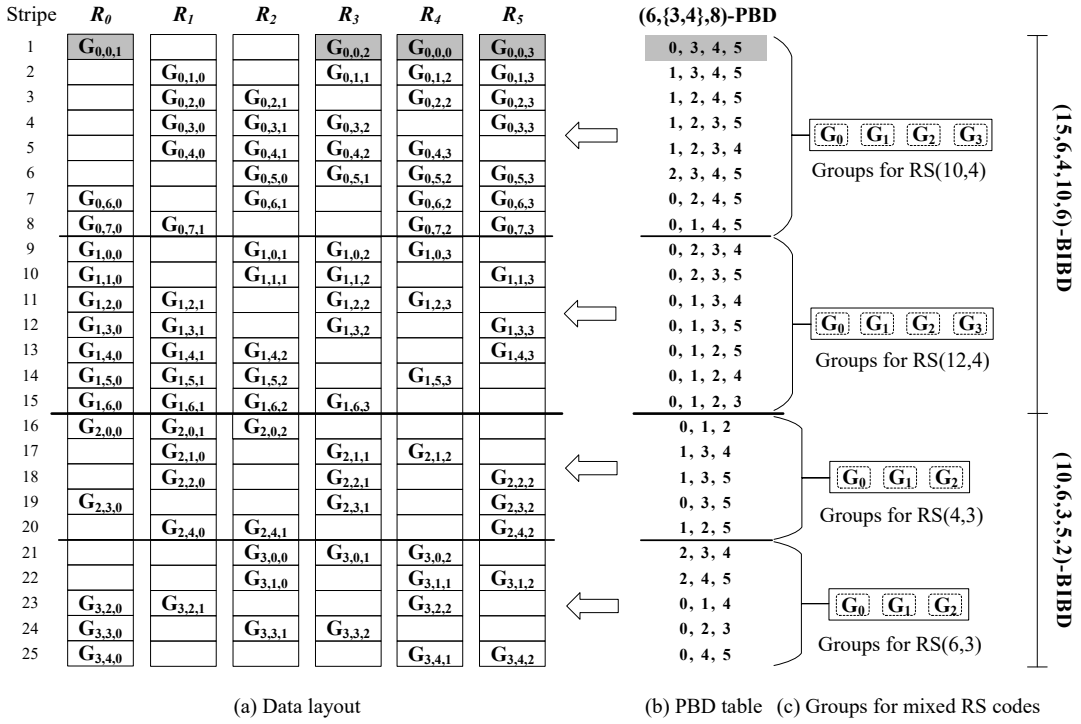


图 2.6　基于 $(6,\{3,4\},8)$-PBD 的混合纠删码数据布局

分组后的 (k_i, m_i) 码分配物理机架。纠删码的条带以循环的方式对应于 PBD 的元组。我们通过随机映射将一个条带的 κ_i 个组分配到相应元组指定的机架，每个组分配到一个机架。以图 2.5 所示的 RS(10, 4) 码条带为例。首先我们找到图 2.6(b) 中的第一个元组 $(0, 3, 4, 5)$，它代表 $\mathbf{G_0}, \mathbf{G_1}, \mathbf{G_2}$ 和 $\mathbf{G_3}$ 这 4 个组可以分配到 4 个机架 R_0、R_3、R_4 和 R_5。系统采用随机映射以决定各组分配到哪个对应机架。例如，$\mathbf{G_0}, \mathbf{G_1}, \mathbf{G_2}$ 和 $\mathbf{G_3}$ 四个组可以分别对应 R_4, R_0, R_3 和 R_5，并各自标记为 $\mathbf{G_{0,0,2}}$, $\mathbf{G_{0,0,0}}$, $\mathbf{G_{0,0,1}}$ 和 $\mathbf{G_{0,0,3}}$，如图 2.6(a)。源机架 R_3 和 R_5 是未满机架，因为它们包含未满组 $\mathbf{G_2}$ 和 $\mathbf{G_3}$。在图 2.6(a) 中，$\mathbf{G_{i,j,l}}$ 是第 j 个条带的第 l 组，使用第 i 种纠删码进行编码，其中 $i = 0, 1, 2, 3$，分别对应 RS(10, 4), RS(12, 4), RS(4, 3) 和 RS(6, 3) 码。从图 2.6(a) 中可以发现，所有的组都均匀分布到所有机架上。

(2) 将块映射到节点。在每个机架内，同一组中的块以轮询方式分配给节点，因此块到机架内节点的分布是均匀的，即一个机架内任意两个节点的块数相差不超过 1 个。

通过上述两个步骤，PDL 实现了基于 PBD 表的机架间和基于轮询规则的节点间均匀的数据布局。由于分布式存储系统常采用树型结构的拓扑结构，机架级别的负载均衡对修复性能的作用比节点级别的负载均衡更重要。在本章下面的讨论中，我们主要关注机架级别的修复流量的负载均衡。

2.4　一种基于均匀数据布局的高效故障修复算法设计

基于上一节中提出的均匀数据布局 PDL，本节提出了针对单节点、多节点和单机架故障的修复方案 rPDL。

2.4.1　单节点故障修复算法

假设机架 R_f 中的节点 N_f 出现故障。如上节所述，系统将 (k,m) 码的条带分成 $\kappa = N_g(k,m)$ 组，每组由 m 或 $m-1$ 块组成，分别称为已满组或未满组。我们将每个组分配到一个单独的机架，并将与满（未满）组对应的机架称为满（未满）机架。给定 (k,m) 码的条带 S，我们将存有条带 S 的块的机架称为源机架，其他机架称为非源机架。如果除 R_f 之外的所有源机架都已满，我们将条带 S 称为已满。如果除 R_f 之外存在某些源机架未满，则我们将 S 称为未满的。单节点故障的修复方案 rPDL 包括三个步骤，即 (1) 选择替代节点，(2) 选择可用块来修复故障块和 (3) 解码故障块。

选择替代节点: 为了最大化修复任务的并行性并加快修复过程，我们从所有节点中为每个条带随机选择一个替代节点。但为了容忍单机架故障，任何机架都不应该存储超过 m 个块。所以我们不能从一个满的源机架中选择一个替代节点。此外，我们不会从包含故障节点的机架中选择替代节点。原因如下：如果机架 R_f 中的一个节点发生故障，则 R_f 是所有故障条带的源机架，但其他机架都是一部分故障条带的源机架。因为机架的数量通常远大于实际分布式存储系统中纠删码条带最小分组的数量，如果我们随机地选择一个源机架并从中来选择一个替代节点，那么 R_f 将遭受比其他机架更重的跨机架流量。所以我们不会从 R_f 中选择替代节点。

基于上述讨论，为了容忍单机架故障，本章提出了以下选择替代节点的方案。此外，此替代节点的选择方案有助于最小化跨机架流量，参考定理 2.2。

- 如果条带 S 已满，则从非源机架中随机选择其替代节点。
- 否则，条带 S 的替代节点从 R_f 以外的未满的源机架中随机选择。

选择可用块以修复故障块: 修复算法不会从 R_f 读取可用数据块以达到负载均衡（与上述选择替代节点的方案理由类似）。下面的引理表明除 R_f 之外的所有源机架都将参与故障修复，即替代节点从除 R_f 之外的所有源机架读取可用块。

引理 2.1　除 R_f 外的所有源机架都将参与 rPDL 下的单节点故障修复。

证明　由于最优条带分组策略，所以每个组包含不超过 m 个块，因此任何两组中的块总和会超过 m。如果有一个源机架 $R(R \neq R_f)$ 不参与修复，那么 R 和 R_f 中的所有块（总共至少 $m+1$ 个块）都不会参与修复。所以最多 $(k+m)-(m+1) = k-1$ 个块可用于修复，这与故障块只能从 k 个块修复相矛盾。 ∎

解码故障块：对于 (k, m) 码，其任意块都是其他 k 块的线性组合，如 $B_0 = \sum_{i=1}^{k} c_i B_i$，其中 $c_i (i = 1, 2, \cdots, k)$ 是该纠删码指定的解码系数。为了减少修复故障块产生的跨机架流量，我们可以通过在机架内的源节点上进行局部解码来聚合同一机架中参与修复的源数据块，将执行这样局部解码的源节点称为*局部解码节点*。替代节点从除 R_f 之外的源机架接收聚合后的块并修复故障块 B_0。

图 2.7 中给出了两个示例。在图 2.7(a) 中，从非源机架中为 RS(6, 3) 码条带的故障块选择替代节点，该条带已满并从除 R_f 外所有源机架读取可用块。将存储 B_4 和 B_6 的两个源节点分别作为各自机架的局部解码节点，然后执行局部解码并将聚合块 $\mathbf{B}^*$ 和 $\mathbf{B}^{**}$ 传输到替代节点。图 2.7(b) 展示了另一种情况，其中替代节点是从未满源机架中选择的，用于未满的 RS(10, 4) 码条带。

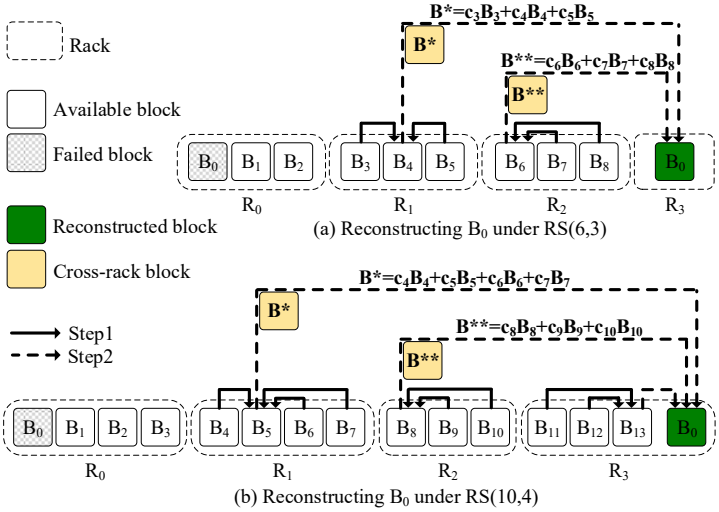


图 2.7　rPDL 局部解码修复一个故障块

为了方便描述，我们用 *CRRT*（*cross-rack reconstruction traffic*）表示以块为单位的跨机架修复流量。当从源机架跨机架读取块到替代节点时，CRRT = 1，并会导致来自源机架的*跨机架修复读*（*cross-rack-reconstruction read, CRRR*）和到替代节点所在机架的*跨机架修复写*（*cross-rack-reconstruction write, CRRW*）。例如图 2.7(a) 中，修复 B_0 需要跨机架传输 $\mathbf{B}^*$ 和 $\mathbf{B}^{**}$ 到 R_3，即 CRRT = 2，R_1 和 R_2 各有一个 CRRR，并且有两个 R_3 的 CRRW。对于已满条带，rPDL 通常修复故障块的 CRRT = $N_g - 1$（未满条带为 CRRT = $N_g - 2$）。结合引理 2.1，我们有以下定理。

定理 2.2　当单个节点发生故障时，rPDL 会为每个故障条带的修复引入最小的跨机架流量。

证明　根据我们的分组方案，对于每个故障的条带 S，如果它是满的，那么除 R_f 之外的所有源机架都是满的，并且包含 m 个块，这意味着 $k = tm$ 或 $tm - 1$（R_f 只能提供 $m - 1$ 或 $m - 2$ 个可用块）。这时替代节点只能来自非源机架，则从 $\lceil \frac{k}{m} \rceil = t$ 个源机架读取数据就足够了。而如果从 R_f 中选择的替代节点，这时候不

会节省跨机架流量，主要原因如下：R_f 中选择的替代节点在机架内读取 $m-1$ 或 $m-2$ 块，这时需要 $\lceil\frac{tm-(m-1)}{m}\rceil=\lceil\frac{(t-1)m+1}{m}\rceil=t$ 或 $\lceil\frac{(tm-1)-(m-2)}{m}\rceil=\lceil\frac{(t-1)m+1}{m}\rceil=t$ 个块的跨机架读取。如果 S 未满，则 $k=tm+q, 0<q<m$ 并且每个组包含 x 或 $x+1$ 个块，其中 $x=\lfloor\frac{k+m}{t+2}\rfloor$。如果替代节点是从未满的源机架中随机选择的，它会读取机架内的所有可用块，至少 x 个块，而其余的块跨机架读取。如果替代节点来自 R_f，它最多读取 R_f 内的 x 个块，并且跨机架读取相同甚至更多数量的块。因此，我们的替代节点选择方案实现了最小 CRRT。 ∎

对于 PBD 周期中的所有待修复条带，按照 PBD 定义，有 λ 个条带同时关联 R_i 和 R_f 机架，所以任何机架 $R_i \neq R_f$ 都是 λ 个待修复条带的源机架。因此，对于 PBD 周期中的所有条带，非源机架均匀分布在所有幸存的机架中。图 2.8 说明了 CRRR 和 CRRW 的均匀分布。为了修复 R_0 中故障节点的 15 个块，每个机架作为源机架 8 次，产生 7 个 CRRR，并作为替代节点机架 3 次，产生 7 个 CRRW。rPDL 在任意数量的条带内很难实现 CRRR（或 CRRW）的完全均匀，下一小节中我们从概率分布的角度来分析 rPDL 的近似均匀性。

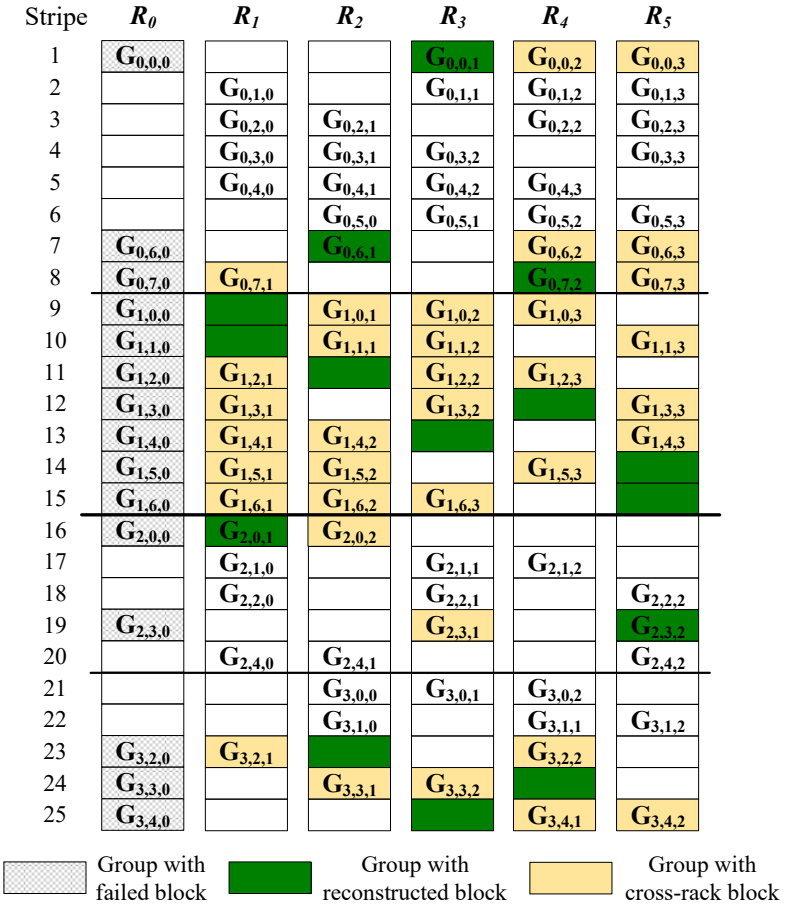


图 2.8　基于图 2.6 的 rPDL 修复流量分布（假设 R_0 中的故障节点导致 R_0 中的所有组都具有一个故障块。RS(12, 4) 和 RS(6, 3) 码的条带已满，其他条带未满）

2.4.2　故障修复流量的均衡性分析

下面的定理 2.3 展示了在有限数量的条带下，所有机架进行单节点故障修复时，rPDL 修复故障块的渐近均匀分布和 CRRR（或 CRRW）的负载均衡情况。

定理 2.3　在 rPDL 中，令 X_1 与 X_2 为随机变量，分别代表来自机架中已满和未满的条带的修复块的数量，μ_1、μ_2 是对应的期望值。假设 b 是一个 PBD 周期中的条带数，而 n_c 是 PBD 周期数。则有：

$$P(|\frac{X_1-\mu_1}{\mu_1}| \geqslant \delta) \leqslant 2e^{-\frac{c_1 bn_c\delta^2}{3(v-1)}}, \tag{2.1}$$

$$P(|\frac{X_2-\mu_2}{\mu_2}| \geqslant \delta) \leqslant 2e^{-\frac{c_2 bn_c\delta^2}{3(v-1)}}, \tag{2.2}$$

其中 $0<\delta<1$, c_1 和 c_2 分别是来自已满条带和未满条带的修复块所占比例，满足 $0 \leqslant c_1, c_2 \leqslant 1$ 且 $c_1+c_2=1$。

证明　我们以不等式 (2.1) 证明为例，不等式 (2.2) 可以类似地证明。假设在第 j 个 PBD 周期中有 s_{j1} 个满条带，其中 $j=1,2,\cdots,n_c$。因为我们从一个随机选择的非源机架中选择一个替代节点，比如说 R_c，并且分布式存储系统中有 v 个机架，所以从 R_c 中选择一个替代节点的概率是 $p_1=\frac{s_{j1}}{(v-1)b}$。因此，从 R_c 中选择替代节点的变量服从二项分布 $B(b,p_1)$，即 $X_{j1} \sim B(b,p_1)$。所以在第 j 个 PBD 周期中 X_1 的期望是 $E_j=bp_1=\frac{s_{j1}}{v-1}$，并且在 n_c 个周期中的期望是

$$\mu_1=\sum_{j=1}^{n_c}E_j=\frac{\sum_{j=1}^{n_c}s_{j1}}{v-1}=\frac{s_1}{v-1}$$

其中 s_1 是 n_c PBD 周期中满条带的数量。令 $c_1=s_1/bn_c$ 为 n_c 个 PBD 周期中的满条带数量。通过 Chernoff Bound，我们有

$$P(|\frac{X_1-\mu_1}{\mu_1}| \geqslant \delta) \leqslant 2e^{-\frac{c_1 bn_c\delta^2}{3(v-1)}}$$

∎

接下来用一个例子来说明定理 2.3 中的近似均匀分布。根据图 2.8，一个 PBD 的周期中有 $v=6$ 个机架和 $b=25$ 个条带。令 $c_1=c_2=0.5$，$\delta=0.1$。如果 $n_c=360$，则

$$P(|\frac{X_i-\mu_i}{\mu_i}| \geqslant 0.1) \leqslant 0.0996, i=1,2$$

从概率的估计来看，rPDL 需要大量的条带才能达到均匀分布，但是实验结果表明，20 个 PBD 周期足以达到近似均匀的修复流量分布。此外，源机架近似

均匀的分布保证了 CRRR 的均匀。而对于 CRRW，要修复已满（或未满）条带的故障块，包含替代节点的每个机架需要 $N_g - 1$（或 $N_g - 2$）个 CRRW。因为已满（或未满）条带有不同的 N_g，所以 CRRW 无法达到与 CRRR 相同的均衡程度。但如性能分析一节的实验所示，CRRW 仍然比现有的随机数据布局更加均衡。

值得指出的是，我们可以通过独立使用 BIBD 来达到每种纠删码都有均匀分布的修复块。但对于所有纠删码，与 PBD 相比，独立使用 BIBD 需要更多的条带才能同时达到修复块的均匀分布。例如，如果对 RS(10, 4) 的条带使用 (15, 6, 4, 10, 6)-BIBD（见图 2.6）需要 15 个条带来完成一个循环。类似地，RS(12, 4)、RS(4, 3) 和 RS(6, 3) 分别需要 15、10 和 10 个条带。因此，为了达成与 PBD 中相同的均匀程度，需要使用两倍数量的条带。由于在分布式存储系统中通常部署多种纠删码以满足多样的用户需求，因此使用 PBD 可以更有效地实现修复块的均匀分布。

2.4.3　多节点故障修复算法

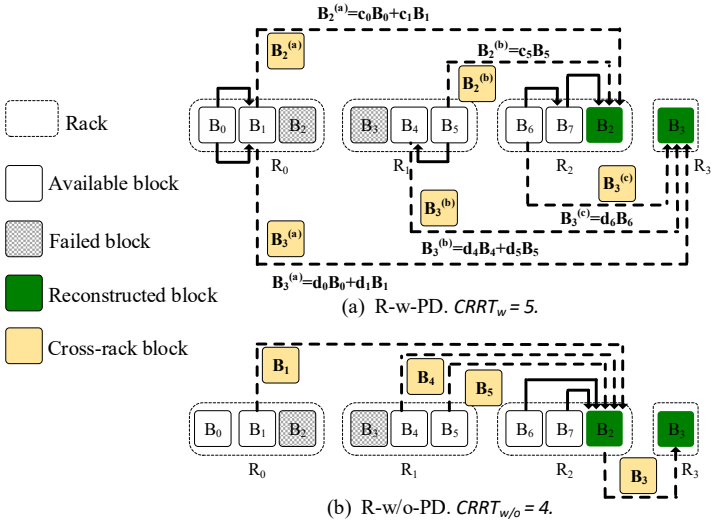


图 2.9　修复 RS(5, 3) 码的双故障条带，采用与不采用局部解码的示例图

多节点故障可能导致条带中的一个或多个块不可用。以下是修复具有多个故障块条带的主要挑战：

1. **替代节点的选择：**我们需要为故障的条带选择多个替代节点，每个替代节点存储一个修复的块，目的是尽量减少跨机架的修复流量并尽可能保持原始的布局。替代节点首先从未满源机架中选择，然后根据需要从非源机架中选择。这些情况比单节点故障更加的复杂，因为不同的故障模式导致源机架的分布不同（未满/满）。
2. **是否采用局部解码：**局部解码可以有效减少单节点故障的跨机架修复流量，但不一定适用于多节点故障，因为即使它们是用相同的数据/校验块修复的，解码不同故障块的解码系数也是不同的。也就是说，对于每个故障的块，提供源数据块的机架需要传输一个单独的聚合块。因此，当有多个故障块需

算法 2.1　多节点故障修复替代节点选择算法

Input: 条带 *stripe* 关联的 l 个故障块集合 L 以及源机架集合 SR
Output: 一个替代节点的集合 RN

```
1  Initialize 一个不满的没有故障块的源机架集合 NSR; s = NSR 集合的大小; RN = null
2  if l⩽ s then
3      从 NSR 中随机选择 l 个机架;
4      for 在 L 中的每一个故障块 do
5          随机地从一个不满的源机架中选择一个节点，并且该节点上没有 stripe 的
             源数据块，把该节点加入到 RN 中;
6  else
7      for rack in NSR do
8          随机地选择一个节点，并且该节点上没有 stripe 的源数据块，把该节点加
             入到 RN 中;
9      随机地选择一个非源机架 R′;
10     随机地选择 R′ 中的 l-s 个替代节点并把它们加入到 RN 中;
11 Return RN;
```

要许多机架来提供源数据块时，局部解码不再具有优势。例如，如图 2.9 所示，在故障的 RS(5, 3) 码条带中有两个不可用的块，这两个故障块出现在两个满的源机架中。局部解码修复（缩写为 *R-w-PD*）产生 CRRT=5。而不采用局部解码修复（缩写为 *R-w/o-PD*）产生较少的 CRRT，只需要跨机架传输 4 个块。所以是否选择局部解码，取决于哪个方案消耗更少的跨机架流量。故障模式的多样性使其选择更加困难。如果选择局部解码，为最小化跨机架修复流量，我们仍然需要从源机架中仔细选择源数据块参与修复。

基于上述的挑战分析，本节提出了具有多节点发生故障时故障条带修复的方案，只有单个故障块的条带可以通过前面的单节点故障方案来修复。在多故障块的情况下，条带中可能有多个块不可用，通常使用不同的解码系数进行修复。如果用局部解码修复多块故障的条带，其每个不可用块都应当生成一个对应的局部聚合块，并将它们发送到机架之外的替代节点上。因此，局部解码在某些情况下不会减少跨机架修复的流量，反而会增加机架间的流量。只有在减少跨机架修复流量时，才应当选择局部解码。对于这种具有多个故障节点的修复情况，本节首先介绍如何选择替代节点，然后分析修复时采用以及不采用局部解码情况，最后给出具体算法来执行多故障修复。

A. 选择替代节点

为了尽量减少跨机架修复流量并尽可能保持原有布局，rPDL 倾向于从没有故障节点的未满源机架中选择替代节点，然后从非源机架中选择替代节点。

算法 2.1 展示了如何为多节点故障修复选择替代节点。假设在条带中有 l 个块不可用，则对应的需要 l 个替代节点。假设故障条带有 s 个未满源机架（第 1 行）。如果 $l \leqslant s$，我们从 s 个未满源机架中随机选择 l 个机架，并在每个选择的机架中，随机选择一个没有该故障条带中的数据/校验块的节点作为替代节点（第

2-5 行)。如果 $l > s$，我们首先从 s 的每个未满源机架中随机选择一个没有该故障条带中的数据/校验块的节点作为替代节点（第 7-8 行），然后随机选择一个非源机架并从中随机选择 $l - s$ 个替代节点（第 9-10 行)。算法 2.1 的开销主要来自尽可能从未满源机架中选择替代节点的循环，它是 $\mathcal{O}(\min\{l, s\})$。

B. 不使用局部解码的修复

不使用局部解码的修复方法简写为 *R-w/o-PD*。我们考虑一个 (k, m) 码，其修复包括三个步骤：(1) 选择一个主替代节点，(2) 从 k 个源节点读取源数据到主替代节点进行解码，(3) 保留一个修复块，发送剩余修复块到辅助替代节点。如果从未满源机架中选择主替代节点，那么该机架拥有 $m - 1$ 个源数据块将会使 CRRT 减少 $m - 1$，所以是更优的选择。另外，k 个被选择的源数据块应该尽可能均匀地分布在各个源机架中。

接下来我们推导出通过 R-w/o-PD 修复具有多故障块条带的 CRRT（记作 $CRRT_{w/o}$)。假设在待修复条带中有 l 个不可用的块，其中 $l \leqslant m$。

(1) 条带已满：由于没有未满源机架可供选择替代节点，我们在非源机架中随机选择一个机架，从中随机选择 l 个节点作为替代节点，并任意指定其中一个节点作为主替代节点 N_{pri}。将 k 个源数据块读取到 N_{pri} 会产生 k 个 CRRT，而主替代节点将解码完成的块发送到机架内的辅助替代节点，没有产生 CRRT。综上，$CRRT_{w/o} = k$。

(2) 条带未满：由于至少有一个未满源机架，因此将未满源机架中的任意一个替代节点指定为主替代节点 N_{pri}。因此 N_{pri} 所在的机架中有 $m-1$ 个源数据块，这意味着只需要将 $k-(m-1)$ 个源数据块跨机架读取至 N_{pri}。N_{pri} 用 k 个源数据块修复所有的故障块，并将 $l-1$ 个修复块传输到不同机架中的其他 $l-1$ 个替代节点，产生 $l-1$ 个 CRRT。即 $CRRT_{w/o} = k-(m-1)+(l-1) = k+l-m$。

综上所述，R-w/o-PD 的 CRRT 为

$$CRRT_{w/o} = \begin{cases} k, & \text{修复一个已满条带;} \\ k + l - m, & \text{修复一个未满条带。} \end{cases} \tag{2.3}$$

C. 使用局部解码的修复

这种方法（缩写为 *R-w-PD*）类似于单节点故障的修复方案（R-w-PD 的 CRRT 记作 $CRRT_w$)。首先在尽可能少的机架中选择 k 个源数据块，然后在每个选择到的机架中，局部解码节点通过对每个故障块都进行局部解码得到一个聚合块，并将生成的 l 个聚合块分别发送到 l 个替代节点。各个替代节点通过合并聚合块来修复对应的故障块。

假设至少有 N'_g 个机架提供用于修复的源数据块，每个机架都向对应的替代节点发送一个聚合块，因此 $CRRT_w = l \times N'_g$。但下面两种情形都能减少 $CRRT_w$：

情形 1. 一些替代节点位于未满源机架中。如果替代节点 N 在一个未满源机架 R_r 上，该机架包含 $m-1$ 个块。R_r 可以将其中所有源数据块机架内传输到节点 N 上，减少一个聚合块的跨机架传输。也就是说，如果 s 个未满源机架中各自有一个替代节点，$CRRT_w$ 可以减少 s。

情形 2. 有一个源机架仅仅包含一个源数据块，且另一个非源机架中至少有两个替代节点。假设源机架 R_1 提供一个块 B，非源机架 R_r 包含 $l'(\geqslant 2)$ 个替代节点。R_1 将 B 跨机架发送至 R_r 中的一个替代节点，之后该节点可在 R_r 机架内将 B 传输至其他 $l'-1$ 个替代节点，这样 $CRRT_w$ 可以减少 $l'-1$。

综上所述，R-w-PD 的 CRRT 为：

$$CRRT_w = \begin{cases} l \times N'_g - s, & \text{情形 1;} \\ l \times N'_g - l' + 1, & \text{情形 2;} \\ l \times N'_g, & \text{其他情形。} \end{cases} \tag{2.4}$$

D. 实例探究

本节将介绍是否使用局部解码进行修复的实例探究。由于分类情况很多，此处只提出分析结果，具体证明内容参考本文末的附录。

(a) 两故障： 假设 $k = tm + q, 0 \leqslant q \leqslant m-1$，其中 $m \geqslant 2$ 并且 $k > m$。我们根据 $q = 0$ 和 $q > 0$ 分类讨论两种情况，并将后者进一步细分为 $m = 2$、$t = 1$ 以及 $m \geqslant 3, t \geqslant 2$ 三种情况。通过我们的分析，发现 $CRRT_w < CRRT_{w/o}$ 在大多数情形下均成立，也就是采用局部解码能避免更多的跨机架流量，但除去以下特例：

(1) $k = 2t$ 或 $k = 2t+1$。

(2) $(4,3)$ 码，并且两个故障块在同一个未满源机架中或者每个故障块在一个单独的未满源机架中。

(3) $(5,3), (7,4)$ 或 $(8,3)$ 码，每个故障块都在一个单独的已满源机架中。

(4) $(5,4)$ 码，每个故障块都在一个单独的已满机架中。

(b) $l(3 \leqslant l \leqslant m)$ **故障：** $l(3 \leqslant l \leqslant m)$ 故障可以使用类似方法来分析 $CRRT_w$ 并确定修复的方式。但是 $l(\geqslant 3)$ 个并发故障的情况非常少见，因此省略分析结果。

E. 多故障修复算法

算法细节： 算法 2.2 展示了如何在 R-w-PD 和 R-w/o-PD 之间做出选择。根据 B 和 C 中的分析，我们首先准确地得到 $CRRT_{w/o}$ 和 $CRRT_w$（第 2-6 行）。由于定性分析的结果，我们可以很容易地得到 $CRRT_{w/o}$ 和 $CRRT_w$ 的确定值，避免计算 CRRT 的开销。然后，我们根据 $CRRT_w$ 和 $CRRT_{w/o}$ 中的更小值来选择 R-w-PD 或 R-w/o-PD（第 7 行）。在确定修复方法后，我们需要从幸存的源节点中选择 k 个源节点（第 8-19 行）。如果修复方法是 R-w-PD，我们采用贪心算法来选择 k 个源节点（第 8-16 行）。此外，局部解码节点在参与修复的源机架中随

算法 2.2　R-w-PD 或 R-w/o-PD 的选择算法

Input: 条带 *stripe* 关联的 l 个故障块集合 L 以及源节点集合 S
Output: 修复方法 *Approach*; 已选的源节点集合 *SN*; 局部解码节点集合 P
1 **Initialize** $CRRT_{w/o} = CRRT_w = 0$, *Approach* = *null*, *SR* = 源机架, 需要的源节点数 $sum_s = 0$
2 $CRRT_{w/o}$ = 修复一个满的条带? k : $k + l - m$;
3 **switch** *R-w-PD* 的情况类型 **do**
4 　情形 1: $CRRT_w = l \times N'_g - s$; break;
5 　情形 2: $CRRT_w = l \times N'_g - l' + 1$; break;
6 　其他情形: $CRRT_w = l \times N'_g$;
7 *Approach* = $CRRT_{w/o} > CRRT_w$? R-w-PD : R-w/o-PD;
8 **if** *Approach* == *R-w-PD* **then**
9 　通过幸存的源节点数来降序排列 *SR*;
10 　**for** R_i *in SR* **do**
11 　　**if** $sum_s + R_i$ 中的源节点数 $> k$ **then**
12 　　把 R_i 中随机选择的 $k - sum_s$ 个源节点加入到 *SN*;
13 　　break;
14 　　**else**
15 　　把 R_i 中的源节点加入到 *SN*;
16 　　sum_s + = R_i 中的源节点数;
17 　　随机地从 R_i 加入 *SN* 的节点中，选择一个节点作为局部解码节点，并加入到 P;
18 **else**
19 　随机地从 S 中选择 k 个源节点;
20 Return *Approach*, *SN*, P;

机选择（第 17 行）。如果修复方法时 R-w/o-PD，则从幸存的源节点中随机选择 k 个源节点（第 18-19 行）。结合算法 2.1，我们完成了多故障的修复算法。

算法开销：算法 2.2 的开销主要是排序（第 9 行）以及循环（第 10-17 行），它们的开销是 $\mathcal{O}(N_g log N_g + N_g)$。因为故障修复的时间一般是秒级的，所以算法的开销是忽略不计的。

2.4.4　单机架故障修复算法

如果单个机架出现故障，每个 (k, m) 码的故障条带中将有 m 或 $m-1$ 个不可用块，因此选择替代节点以及修复的方法类似于多节点故障修复。R-w/o-PD 在单机架故障修复中是首选方案，具体分析如下：

(1) $k = tm$，有 $t+1$ 个组，每组 m 个块。每个故障条带都是已满的。故障机架在故障条带中产生 $l = m$ 个待修复的块，并且 t 个幸存的源机架提供 k 个块用于修复，因此 $CRRT_w = tm$，$CRRT_w = CRRT_{w/o}$。

(2) $k = tm + q(1 \leqslant q \leqslant m-1)$，并且故障机架（假设为 R_f）已满。一个条带中有 $t+2$ 个组，包括 $m-q$ 个未满组，所以除了 R_f 之外的 $t+1$ 个机架提供用于修复的可用块。由于故障条带中有 $l = m$ 个待修复块和 $m-q$ 个未满的源机架，因此在未满源机架中有 $m-q$ 个替代节点。由 R-w-PD 的情

形 1，有 $CRRT_w = m(t+1) - (m-q) = tm + q = k$。此时故障条带未满，$CRRT_{w/o} = k + l - m = k$，因此 $CRRT_w = CRRT_{w/o}$。

(3) $k = tm + q(1 \leqslant q \leqslant m-1)$，并且故障机架（假设为 R_f）未满。条带中有 $t+2$ 个组，包括 $m-q$ 个未满组。所以除了 R_f 之外的 $t+1$ 个机架提供用于修复的可用块。因为故障条带中有 $l = m-1$ 个待修复块和 $m-q-1$ 个未满机架，因此可以在未满机架中选择 $m-q-1$ 个替代节点。由 R-w-PD 的情形 1，有 $CRRT_w = m(t+1) - (m-q-1) = tm + q + 1 = k + 1$，而 $CRRT_{w/o} = k$ 或 $CRRT_{w/o} = k + l - m = k - 1$ 取决于故障条带是否已满。因此，$CRRT_w > CRRT_{w/o}$。

当 $CRRT_w > CRRT_{w/o}$ 或者 $CRRT_w = CRRT_{w/o}$ 时，R-w/o-PD 不会比 R-w-PD 产生更多的跨机架流量。基于上述对三种情况的分析，单机架故障要么是 $CRRT_w > CRRT_{w/o}$ 要么是 $CRRT_w = CRRT_{w/o}$。因为 R-w/o-PD 总能达到较小的跨机架流量或较小的计算开销较低（无需局部解码计算），所以对于单机架故障修复，R-w/o-PD 是首选的方法。

2.4.5　故障修复后均匀数据布局管理

rPDL 从未满源机架或非源机架中选择各个故障条带的替代节点，这可能会破坏 PDL 的均匀布局。为了维护 PDL 的布局，系统不可避免地需要迁移数据，但该过程可以在负载较轻的时段后台完成。对于一个 (k, m) 码的故障条带，迁移的目标是根据分组规则确保修复后的条带中每组包含 m 或 $m-1$ 个块。下面将根据未满组的数量，详细分析单节点故障修复后的迁移情形。多节点和单机架故障修复的分析类似，此处不再赘述。假设 $k = tm + q$，$0 \leqslant q \leqslant m-1$。

(1) $q = 0$，此时所有组都是已满的，并且从非源机架中选择了一个新的替代节点，因此需要将修复块迁移到 R_f。

(2) $m - q \geqslant 2$，此时至少有两个未满组，无论故障块在已满组还是未满组中，都有一个未满源机架根据 rPDL 放置修复块，系统仍然维持了近似均匀的条带分布，不需要迁移。

(3) $m - q = 1$，此时条带正好有一个未满组。如果故障块在唯一的未满组中，rPDL 将从非源机架中选择替代节点，需要把修复块迁移到 R_f，类似于情形 (1)。

否则，rPDL 在未满源机架中选择替代节点，不需要迁移，类似于情形 (2)。

显然，只有特定的情况需要迁移，并且单节点故障下 (k, m) 码需要迁移的块所占比例仅为 $\frac{1}{k+m}$。除 R_f 以外，需要迁移的修复块在机架之间的分布是近似均匀的。因此，迁移可以并行执行，迁移引起的跨机架流量在除 R_f 外的机架间近似均匀。为了进一步减轻迁移带来的干扰，当系统负载较轻时，rPDL 会逐批执行迁移，例如将多个 PBD 周期作为一批处理。

2.5　系统实现

我们在 Hadoop 3.1.1 中的 HDFS 纠删码模块（HDFS EC）实现了 PDL 和 rPDL，并通过修改 HDFS EC 默认的块放置策略和纠删码工作流程来评估 PDL 以及 rPDL 的性能。图 2.10 展示了 HDFS 中实现 PDL 的整体架构。

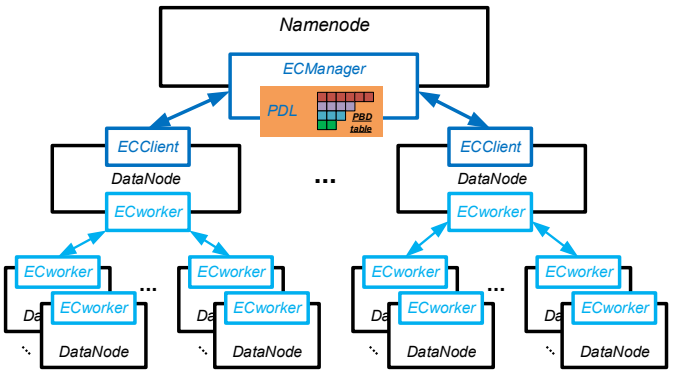


图 2.10　PDL 的系统实现概要图

HDFS EC 的实现：ECClient 是 HDFS 客户端从多个源节点读取数据时作为替代节点的实现。NameNode 中的 ECManager 主要监控节点状态以识别故障节点以及协作故障修复等过程。添加的 ECWorker 线程监听来自 ECManager 的修复命令，通过从各个 DataNode 拉取数据、执行编码以及修复故障块等来满足这些修复请求。默认情况下，对于多节点或单机架故障修复，HDFS 会先选择一个主替代节点执行修复，在主替代节点中保留一个修复块，并将其余的修复块发送至其他替代节点。

PDL 与 rPDL 的实现：PDL 和 rPDL 实现在 ECManager 的数据放置策略模块中。ECManager 的辅助功能包括对条带的块进行分组、将组分配给机架以及将组内的块分配给机架内的节点。系统正常状态下，它通过 PBD 表来为条带寻址对应的机架和节点。在修复状态下，系统根据是否采用局部解码来选择替代节点和源节点。rPDL 分两步实现局部解码，第一步将每个修复任务拆分为子任务，每个子任务在源机架中将机架内的源数据块发送到相应的 DataNode 上执行局部解码。第二步，替代节点上读取局部解码块从而进一步修复故障块。rPDL 是否使用局部解码取决于在多节点故障和单机架故障两节中讨论的最小化跨机架流量的规则。

PDL 的额外开销：为了实现 PDL，分布式存储系统需要在元数据服务器（HDFS 中的 Namenode）中存储一个 PBD 表。PBD 表的大小取决于系统的机架数、纠删码策略、以及 PBD 周期等参数，其大小在 HDFS 中最多为 KB 级别。此外，额外的 CPU 开销（例如寻址 PBD 表）等和存储开销相比可以忽略不计。

2.6　性能评估

2.6.1　实验设置

我们的实验在 28 节点的本地集群上进行，每台机器都配置了四核 3.4 GHz Intel Core i5-7500 CPU、8GB RAM 和 Seagate ST1000DM010-2EP102 1TB 7200-RPM SATA HDD。所有机器都运行 Ubuntu 16.04.3 LTS 操作系统。一台机器被分配为 HDFS 的 NameNode，其余 27 台机器作为 DataNode。所有机器分布在两层交换机连接的多个机架中，每个交换机的带宽为 1 Gbps，其树型拓扑结构与图 2.4 类似。

传统分布式存储系统（例如 HDFS）随机地分配块，在有限的一批条带内很难实现修复负载均衡。此外，这些系统在故障节点附近选择替代节点，导致替代节点负载过重。本章认为数据布局以及替代节点和源节点的选择都有助于提高故障修复的性能。为深入理解数据布局和故障修复算法对修复性能的贡献，分别部署以下四种方案对系统修复进行了实验：(1) 数据布局基于 PDL 的修复算法（用 rPDL 表示）；(2) 默认的 HDFS 修复算法及其默认数据布局（用 HDFS 表示）；(3) 随机选择替代节点和源节点以及采用随机数据布局（用 RDP 表示）；(4) 对条带中的块进行分组，将组随机分配到机架（即不使用 PBD）并使用局部解码修复算法（用 NonPBD 表示）。

所有实验设置都容忍单机架故障或者 m 节点故障，详细说明见表 2.1。系统中配置了 6 个机架，机架依次拥有 $6,5,4,4,4,4$ 个节点，并将块大小设置为 16MB。将 RS$(3,2)$、RS$(6,3)$ 和 RS$(10,4)$ 的条带以 1 ： 1 ： 1 的比例写入系统，随后随机使一个节点离线来模拟单节点故障。为有效地利用分布式存储系统的修复能力，实验放宽了分批处理修复任务的限制。基于 HDFS 的默认参数 `dfs.datanode.ec.reconstruction.threads = 8`，每一批修复任务的数量最多能够达到 8 倍幸存节点的数量。因此有 26 个节点的集群最多执行 8×26 个修复任务，修复之前写入条带的数量被限定为不超过 500，以便能够一批处理故障节点上的块。每组实验数据为五次运行的平均结果，实验的标准差相对较小，在图中忽略。

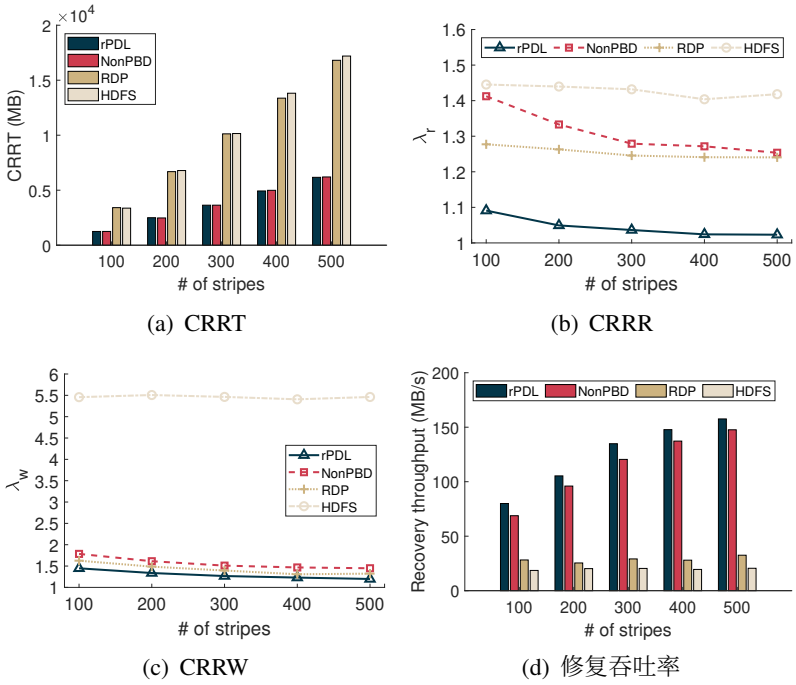
表 2.1　rPDL 以及比较对象的属性

修复方案	块特征	替代节点选取	被检索块选取	局部解码
HDFS	分散，随机	就近	随机	否
RDP	随机	随机	随机	否
NonPBD	分组，随机	随机	随机	是
rPDL	分组，均匀	均匀	均匀	是

2.6.2　单节点故障修复性能评估

这组实验评估了 rPDL 以及比较对象在 *CRRT*、*CRRR* 与 *CRRW* 的负载均衡、修复吞吐率以及降级读延迟等方面的性能。

CRRT：首先评估 CRRT，定义为通过核心交换机进行修复的数据量，反应了核心交换机在修复过程中的负载，实验结果展示在图 2.11(a) 中。可以看到，相比于 HDFS，rPDL 将 CRRT 降低了 62.90% 至 64.32%，因为 rPDL 将条带中的块分组并进行局部解码，理论上将 (k, m) 码的 CRRT 降低到传统修复方法的 $1/m$ 左右，实验也进一步验证了这一结论。



(a) CRRT　(b) CRRR

(c) CRRW　(d) 修复吞吐率

图 2.11　单节点故障的修复性能

CRRR 与 CRRW 的负载均衡： CRRR 的负载均衡程度使用 $\lambda_r = r_m/r_a$ 来进行评估，r_m 和 r_a 分别是单个机架 CRRR 的最大值与平均值。λ_r 的值越接近 1，CRRR 越均衡。类似的，CRRW 的负载均衡程度用 $\lambda_w = w_m/w_a$ 评估。从图 2.11(b) 和图 2.11(c) 可以看出，由于均匀的数据布局以及替代节点和源节点的选择，rPDL 实现了 CRRR 和 CRRW 最好的负载均衡。rPDL 的 λ_r 始终比其他三种方案小 20%，并在 500 个条带时以 $\lambda_r = 1.02$ 达到最佳读取均衡。rPDL 与 NonPBD、RDP 和 HDFS 相比，λ_w 平均减少了 17.14%、9.13% 和 76.27%。HDFS 将修复任务分配给附近的 DataNode，大多数修复流量都集中到 R_f，因此 rPDL 远好于 HDFS。NonPBD 的 λ_r 和 λ_w 比 RDP 更差，因为与 RDP 相比，尽管 NonPBD 的分组和局部解码会降低 CRRT，但也会导致更严重的负载不均衡。rPDL 的 λ_w 高于 λ_r，因为它需要更多的 PBD 周期才能使每个 BIBD 中都达到饱和，从而达到未满条带替代节点的均匀分布。

修复吞吐率：接下来评估系统修复吞吐率（定义为每秒修复的数据量）。图 2.11(d) 显示，rPDL 的修复吞吐率平均为 HDFS、RDP 和 NonPBD 的 6.27 倍、4.35 倍和 1.10 倍。RDP 随机选择替代节点，所以其吞吐率比 HDFS 更高。由于部署了局部解码，NonPBD 的表现明显优于 RDP。与 NonPBD 相比，rPDL 由于均匀的数据布局以及替代节点和源节点的均匀选择，显示出更高的修复吞吐率，尤其是在跨机架带宽有限的情况下。从实验中还发现，当条带数量增加时，CRRR 和 CRRW 更加均衡（见图 2.11(b) 和 2.11(c)），rPDL 的修复吞吐率也会变得更高。

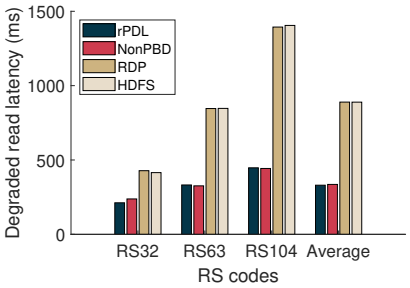


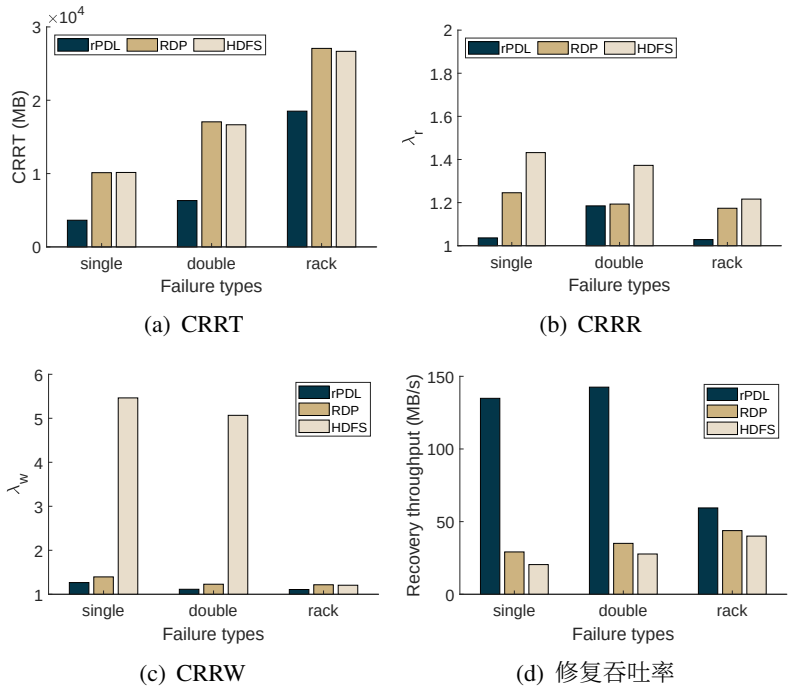
图 2.12　降级读延迟

降级读延迟：本实验中，系统首先为 RS(3, 2)、RS(6, 3) 或 RS(10, 4) 各写入一个条带，而后随机下线一个有数据块的节点，客户端以修复的方式获取该数据块。图 2.12 显示 rPDL 和 NonPBD 实现了几乎相同的降级读延迟，HDFS 则与 RDP 的接近。rPDL、NonPBD 与 HDFS、RDP 相比，使用 RS(3, 2)、RS(6, 3) 和 RS(10, 4) 码的条带的降级读延迟分别下降了大约 48.92%，60.80% 和 68.16%，因为 rPDL 和 NonPBD 都采用局部解码，有效地降低了 CRRT。如果对于大比例的纠删码，也就是条带中的数据块和校验块更多，rPDL 降级读延迟的减少幅度更大，因为此时的并行修复效率更高。

2.6.3　多节点故障和单机架故障的修复性能评估

本组实验对 300 个纠删码条带（RS(3, 2)、RS(6, 3) 和 RS(10, 4) 的比例相同）进行编码并写入存储系统，以评估多故障情形的修复性能。与单节点故障类似，通过使两个节点或一个机架随机离线来对应模拟多节点故障及单机架故障。图 2.13 展示了单节点故障、双节点故障和单机架故障的修复性能。实验使用四个参数来分析修复性能，即 CRRT、CRRR、CRRW 和修复吞吐率。由于在发生多个故障时，NonPBD 的局部解码策略很难设计，因此不再与 NonPBD 进行性能比较。实验有以下观察结果：

(1) 与 HDFS 相比，rPDL 的 CRRT 值在单节点、双节点和单机架故障情形下分别降低了 64.16%、62.06% 和 30.59%。当两个节点失效时，产生的故障条带中将可能有 1 个或 2 个故障块。具有 2 个故障块的条带平均仅占到总故障条带的 20.12%。所以即使有两个节点故障，单故障修复仍然占修复过程的主要部分。



(a) CRRT　(b) CRRR

(c) CRRW　(d) 修复吞吐率

图 2.13　多节点故障的修复性能

(2) 对于 CRRR，rPDL 在单节点和单机架故障情形下仍然保持低 λ_r，因为 PDL 的均匀数据布局使得源数据块分布更加均匀。但是对于双节点故障，rPDL 的 λ_r 值几乎与 RDP 相同，因为 rPDL 使用局部解码，会从尽可能少的机架中读取源数据块，导致其 CRRR 很难在幸存的各个机架间达到均衡。

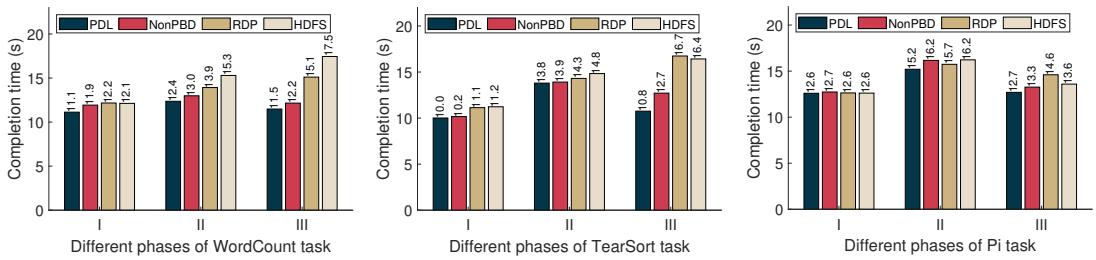
(3) rPDL 的 CRRW 在所有类型的故障中总是最低的。而 HDFS 的 λ_w 非常高，因为在单节点和双节点故障的情况下，HDFS 会就近选择替代节点。但在单机架故障的情况下，HDFS 更改为随机选择替代节点，显著降低了 λ_w 值。

(4) rPDL 的修复吞吐率在单节点、多节点和单机架故障情形下分别达到了 HDFS 的 6.60 倍、5.14 倍和 1.48 倍。rPDL 在单节点和双节点故障的高修复吞吐率主要得益于局部解码，当单机架故障时，负载均衡起着关键的优化作用。

2.6.4　实际应用的性能评估

现在我们使用 Hadoop [86] 发布的三个具有代表性的 MapReduce 应用程序 *WordCount*、*TeraSort* 和 *Pi* 来评估故障修复算法对前台应用程序的干扰。*WordCount* 计算给定文件中每个单词的出现次数，并导致大量跨机架网络流量和少量计算开销。*TeraSort* 对随机分布在分布式存储系统中的数据集进行排序，并导致大量的跨机架网络流量和计算开销。*Pi* 通过近似算法计算 π 并导致大量计算开销和少量网络流量。三个 MapReduce 应用程序将生成中间和临时数据，这些数据存储在分布式存储系统中，并在完成这些任务后被清理。我们为 *WordCount* 配置 5000000 个随机生成的不同长度的单词作为输入集，100 万条记录作为 *TeraSort*

的排序样本，并为 *Pi* 运行 20 个任务，每个任务运行 3000000 次。我们在分布式存储系统中分三个阶段运行 MapReduce 应用程序：(I) 仅运行应用程序；(II) 运行应用程序，Namenode 同时写入 300 个用 RS(3, 2)、RS(6, 3) 以及 RS(10, 4) 码的条带，这会导致分布式存储系统中产生大量的读写流量；(III) 在修复过程同时运行应用程序。应用程序的所有请求都是从 Namenode 发出的。



(a) WordCount　(b) TeraSort　(c) Pi

图 2.14　MapReduce 应用性能测试

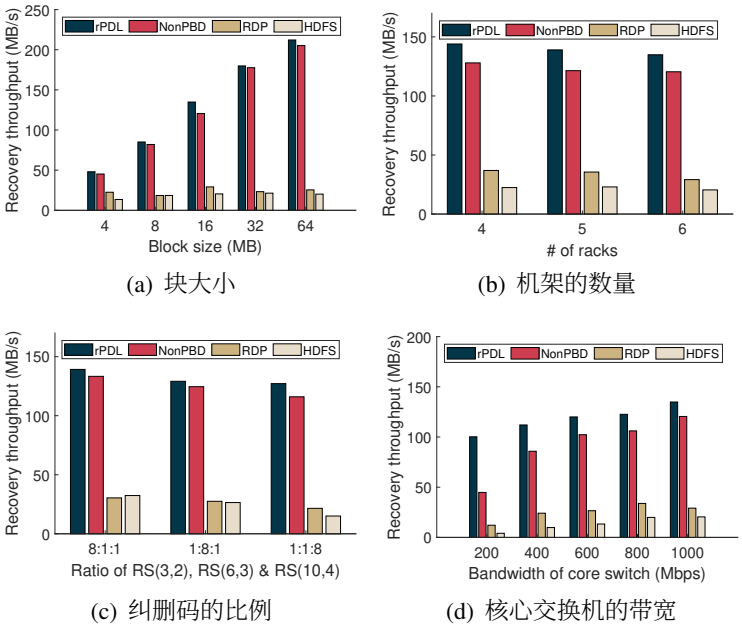
从图 2.14(a)、图 2.14(b) 和图 2.14(c) 中，我们发现应用程序的完成时间对数据布局很敏感。对于流量密集型应用程序 *WordCount* 和 *TeraSort*，PDL 显著地优于其他三个比较对象，尤其是在分布式存储系统执行修复时。当分布式存储系统在阶段（III）运行 *WordCount* 和 *TeraSort* 时，rPDL 的完成时间与 HDFS 相比分别减少了 34.29% 和 34.15%。即使对于流量很少的 CPU 密集型应用程序，*Pi*，rPDL 在阶段 (III) 的完成时间上也比 HDFS 减少了 6.62%。

2.6.5　灵敏度性能评估

在这组实验中，我们分别用 RS(3, 2)、RS(6, 3) 和 RS(10, 4) 编码了 300 个条带，以评估 rPDL 的性能关于块大小、分布式存储系统的拓扑、使用不同纠删码编码的条带比例以及核心交换机的可用带宽等系统关键参数的影响。我们使用三种分布式存储系统拓扑，DSS_1 具有 6 机架以及分别在每个机架中有 6, 5, 4, 4, 4, 4 节点，DSS_2 具有 5 机架以及分别在每个机架中有 7, 6, 5, 5, 4 个节点，DSS_3 分别在每个机架中有 4 个机架以及 7, 7, 7, 6 个节点。在下面的实验中如果没有额外指定，默认块大小为 16MB，拓扑结构为 DSS_1。

块大小: 图 2.15(a) 展示了不同块大小下 rPDL 的修复吞吐率是 HDFS 的 3.56 倍到 10.52 倍。随着块大小的增加，rPDL 和 NonPBD 的修复吞吐率显著增加，但 RDP 和 HDFS 的修复吞吐率较为稳定。这是因为 rPDL 和 NonPBD 使用局部解码，使得核心交换机的带宽未达到饱和。但 RDP 和 HDFS 不使用局部解码，即使对于较小的块大小，核心交换机的带宽仍然是瓶颈。

分布式存储系统拓扑: 图 2.15(b) 展示了不同机架数下 rPDL 的修复吞吐率是 HDFS 的 6.05 倍到 6.60 倍。随着机架数量的增加，rPDL、NonPBD、RDP 和 HDFS



(a) 块大小　(b) 机架的数量

(c) 纠删码的比例　(d) 核心交换机的带宽

图 2.15　单节点故障下系统关键参数的灵敏度修复性能

的修复吞吐率都会缓慢下降。这是因为随着 PBD 周期中条带数量的增加，CRRR 和 CRRW 的分布变得更加不均衡。但是，rPDL 可以更好地提高修复性能，因为它减少了更多的跨机架修复流量。

不同纠删码条带的比率: 图 2.15(c) 显示不同纠删码比例下 rPDL 的修复吞吐率为 HDFS 的 4.30 倍到 8.45 倍。当分布式存储系统中的条带大部分来自 RS(3, 2) 时，rPDL、NonPBD、RDP 和 HDFS 都实现了最高的修复吞吐率，因为这时候需要最少的块来参与修复。当大比例条带的占比变大时，rPDL 的修复吞吐率高于其他情况，因为它通过局部解码减少了更多的跨机架修复流量。

核心交换机带宽: 图 2.15(d) 展示了 rPDL 的修复吞吐率是 HDFS 的 6.16 倍到 24.56 倍。核心交换机用于修复的带宽是由交换机自带的终端所配置的。很少的核心交换机带宽会减慢所有修复方法的故障修复，但 rPDL 显示出相对稳定的性能。值得指出的是，在核心交换机带宽较小的情况下，rPDL 可以更有效地加速修复，因为它通过局部解码减少了跨机架流量以及达到了更均衡的修复负载。

2.6.6　大规模存储系统的模拟性能评估

为进一步探索 PDL 和 rPDL 在大规模存储系统中的性能，我们配置了一个模拟器来研究大规模系统下的 CRRT、CRRR 和 CRRW，并将结果展示在表 2.2。

模拟器配置: 在模拟器中，我们首先生成一个二维数组来模拟分布式存储系统的机架和节点。然后，我们生成一些虚拟的条带，这些条带由 RS(3,2)、RS(6,3) 和 RS(10,4) 码以相同的比例编码。在每组测试中，条带数为节点数的 20 倍，块大小为 128MB（这是 HDFS 默认块大小 [5]）。最后，我们根据四种方案将虚拟

表 2.2　模拟大规模分布式存储系统修复性能（配置栏表示机架数以及单个机架的节点数）

配置	CRRT (GB)				CRRR (λ_r)				CRRW (λ_w)			
	rPDL	NonPBD	RDP	HDFS	rPDL	NonPBD	RDP	HDFS	rPDL	NonPBD	RDP	HDFS
7,10	54	55	147	148	1.12	1.29	1.10	1.25	1.22	1.45	1.26	5.77
7,20	55	54	143	157	1.13	1.32	1.09	1.26	1.24	1.45	1.27	5.67
7,50	54	54	145	150	1.12	1.30	1.09	1.27	1.24	1.38	1.30	5.71
10,10	55	55	161	162	1.31	1.43	1.12	1.23	1.31	1.50	1.41	9.32
10,20	55	54	154	157	1.41	1.44	1.15	1.23	1.35	1.51	1.36	9.30
10,50	55	54	157	150	1.30	1.50	1.14	1.25	1.41	1.48	1.41	9.14
15,10	57	54	166	159	1.73	1.89	1.18	1.26	1.45	1.61	1.52	15.00
15,20	54	54	172	156	1.92	2.04	1.16	1.25	1.49	1.63	1.52	15.00
15,50	54	53	163	154	1.73	1.81	1.19	1.26	1.58	1.62	1.46	15.00
20,10	55	54	166	167	2.24	2.35	1.22	1.32	1.70	1.73	1.78	20.00
20,20	55	54	166	162	2.64	2.78	1.22	1.29	1.57	1.63	1.67	20.00
20,50	55	55	170	163	2.29	2.32	1.23	1.30	1.55	1.73	1.78	20.00

条带中的块映射到数组中（参见表 2.1）。我们随机选择一个节点来模拟单个节点故障。

rPDL 和 NonPBD 的 CRRT 最低，约为 HDFS 和 RDP 的 $\frac{1}{m}$（$\frac{1}{(2+3+4)/3}=\frac{1}{3}$）。主要原因如下：为了修复 (k,m) 码条带的块，rPDL 和 NonPBD 引起 CRRT = N_g-1（或者 CRRT = N_g-2），取决于局部解码而导致的满（或不满）条带（参见第 2.4 节）；而对于 HDFS 和 RDP，CRRT 几乎是 k。这里 $N_g-i=\lceil\frac{k+m}{m}\rceil-i=\lceil\frac{k}{m}\rceil-i+1\approx\frac{k}{m}$，$i=1,2$。因此，与 HDFS 和 RDP 相比，rPDL 和 NonPBD 将 CRRT 近似降低到 $\frac{k}{m}\div k=\frac{1}{m}$ 的比例。此外，由于 CRRT 主要依赖纠删码的参数，因此随着分布式存储系统规模的增加，CRRT 保持稳定。

由于基于 PDL 的均匀数据布局和选择替代/源节点的策略，rPDL 始终展示了低于 NonPBD 的 λ_r 和 λ_w。rPDL 的 λ_r 和 λ_w 随着分布式存储系统规模的增加而增加。原因是 rPDL 依赖于 PBD 表，它可以由一组带有参数 $(b_i,v,\kappa_i,r_i,\lambda_i)$ 的 BIBD 构成（v 和 $\sum b_i$ 表示数量分布式存储系统中的机架数和 PBD 表的周期大小）。随着分布式存储系统规模的增加，也就是 v（机架数量）增加，很难找到相应小参数 b_i 的 BIBD [81-83]，这会导致 PBD 的周期变大，从而导致 λ_r 和 λ_w 的性能很差。此外，RDP 在大规模分布式存储系统中显示出比 rPDL 和 NonPBD 更均衡的 CRRR/CRRW。原因是分组使得 CRRR 和 CRRW 聚集在一些机架中，在大型分布式存储系统中更加明显。因此 rPDL 在大规模分布式存储系统中的 CRRR/CRRW 表现不佳。

2.7　本章总结

分布式存储系统中故障修复速度受限于以下因素：(1) 在典型的树型结构的分布式存储系统的拓扑中，可用的跨机架带宽比机架内部带宽要稀缺得多，大量的跨机架通信严重减慢了修复过程。(2) 通常系统中块的随机分布，并且尽量将同一条带的块随机放置到最多的机架上，可以在分布式存储系统中实现高可靠性以及存储负载均衡，但通常会在一个或一组条带内引起严重的负载不均衡。由于有限的 I/O 和 CPU 资源，我们只能修复一组条带中的故障数据。因此，传统修复过程是负载不均衡的。(3) 为了满足用户不同的可靠性需求并适应异构应用程序的工作负载，许多分布式存储系统提供了多个纠删码，从而导致更加复杂的数据分布与修复算法，使故障修复过程更加困难。本章针对混合纠删码的分布式存储系统，提出了均匀的数据布局和高效的故障修复算法。由于数据布局的均匀分布，在修复过程中几乎达到均衡的读写 I/O，并通过局部解码算法，实现了高并行度的修复，并大大减少了跨机架流量。因此，与传统的随机数据放置相比，它显著加快了修复过程。

针对本章介绍的 PDL 数据布局以及 rPDL 修复算法，做如下总结：

(1) 基于数学上组合设计工具 PBD，PDL 首次提出了针对分布式存储系统的用于混合纠删码的数据布局。它可以在机架级别实现均匀的数据放置，并可以容忍单个机架故障，从而满足常见应用的可靠性要求。

(2) 基于均匀布局 PDL，rPDL 提出了一种负载均衡的修复算法，该算法均匀地选择替代节点并确定地选择可用块来修复故障块。rPDL 在修复过程中实现了均衡的读写 I/O 和高并行度。

(3) rPDL 通过设计在机架内的局部解码，大大减少了跨机架的流量，从而减轻了跨机架带宽的竞争并加快了修复过程。

第 3 章　基于纠删码存储系统的故障修复任务调度设计

3.1　引言

在基于纠删码的存储系统中，修复故障数据会导致磁盘 I/O、网络传输和解码等方面都有很高的开销。特别是，随着单个节点的容量和分布式存储系统中存储设备数量的不断增加，整体故障数据修复时间也在不断增加。例如分布式存储系统盘古 [50, 87]，它由超过一万个节点组成，每个节点存有多达 72TB 的数据。因此，需要为基于纠删码的分布式存储系统设计高效的故障数据修复方法，以减少不可用数据的停机时长，并满足各种前台应用程序对延迟、可用性和持久性的严格要求。

假定系统部署的是一个 (k, m) MDS 纠删码，从系统可靠性角度考虑，同一个条带中的 $k+m$ 个数据块/校验块分别存储在 $k+m$ 个不同的节点中，每个节点存一个块。在大多数实际部署的基于纠删码分布式存储系统中，数据块/校验块在节点之间随机分布。当系统出现故障时，对每一个丢失的块：首先从同一个条带中随机挑选 k 个幸存的块；然后从存有这 k 个块的节点中读取这些块；接着重构出丢失的块；最后写到一个随机选择的替代节点。

现有方法通常随机分布数据块/校验块，在故障修复过程中随机选择源节点和替代节点 [24, 88]。这种方法一方面实现简单；另一方面，分布式存储系统中有大量的数据块/校验块，从统计的角度可以达到节点之间数据分布和工作负载的均衡。但由于内存容量、网络带宽、CPU 计算能力等方面的限制，故障盘中数据的修复是分批次执行的。现有的修复方法将序列号连续的一些待修复的任务打包成一个批次。因为每个批次中待修复的数据量有限，随机分布数据块/校验块会导致每批次内各节点负载严重不均衡，而随机选择源节点和替代节点则会加剧这种不均衡性。此外，异构的存储系统环境，动态变化的前台工作负载和流量也会对此产生影响，最终这种不均衡会明显拖慢修复过程。

出于成本上的考虑，分布式存储系统中节点间的互联网络通常配置为 1Gbps 或 10Gbps 的以太网，这显著低于节点内的多块磁盘聚合的 I/O 带宽。系统还需要保留足够的网络带宽来服务前台在线应用程序，例如 MapReduce [89] 任务和数据查询任务等。所以在分布式存储系统中，数据修复工作仅能使用有限的网络带宽，例如在盘古系统中修复带宽的默认设置为 30MB/s [50]。尽管部署 Infiniband [90] 等高速网络可以加快网络传输过程，但不同节点上用于故障修复的工作量是不均衡的，这仍然会减慢整体修复。本章将介绍以下方法来均衡节点上的故障修复工作负载：(1) 不采用将连续序列号的多个条带打包到一个批次的方式，而是从大量待修复的任务队列中挑选一些，将这些条带打包到一个批次，结合对源节

点的确定性选择算法，达到从各个节点读取源数据的均衡。(2) 确定性地选择替代节点，使得各个节点间解码的工作量与写入已经修复的数据量达到均衡。基于上述两种方法，本章提出了一个均衡调度模块 SelectiveEC，用于在分布式存储系统中进行有效的故障修复。本方法可用于修复多节点故障，在异构的网络和磁盘 I/O 带宽等环境下同样适用。综上所述，本章有以下的贡献点：

- 我们分析了常用的分批修复机制，发现它可以加速故障修复，但在批处理修复中面临严重的负载不均衡，这也是我们工作的出发点。
- 我们提出了一个二分图模型和一个批处理算法来将修复任务打包成批处理，并选择源节点来均衡节点之间的上行修复流量。
- 我们使用另一个二分图模型和最大匹配算法来选择替代节点以均衡其下行修复流量和解码负载。
- 我们在 HDFS 3 中实现了 SelectiveEC，并在本地 18 节点集群和 AWS EC2 中评估了性能。在同构环境中，SelectiveEC 与 HDFS 相比，将修复吞吐率提高了 30.68%；与最先进的方法 CAR、ECPipe 和 PPR 相比，修复吞吐率提高超过 20%。此外，在异构网络环境下，与 HDFS 相比，SelectiveEC 可以实现 1.32 倍的修复吞吐率以及 1.23 倍的前台 MapReduce 任务的吞吐率。

本章其余部分组织如下：我们首先在第 3.2 节对分批故障修复数据的性能瓶颈问题进行了描述以及分析；然后在第 3.3 节提出了一个基于二分图的分批修复模型，从而来刻画修复任务与存储系统幸存节点的关系；在第 3.4 节介绍了负载均衡的故障修复调度模块 SelectiveEC 的设计；在第 3.5 节和第 3.6 节分别介绍了 SelectiveEC 的系统实现细节以及性能评估；最后，在第 3.7 节对本章总结。

3.2　分批故障修复数据的性能瓶颈问题描述

纠删码有很多的种类，其中 MDS 码具有最小的存储冗余，在当前的系统中应用最为广泛，因此本章介绍的算法是针对满足 MDS 限制的纠删码。尽管其存储效率高，但纠删码在故障修复过程中会引入大量 I/O 负载、网络流量和 CPU 开销，还会导致对故障数据块的降级读，干扰对幸存数据的正常访问。因此，在基于纠删码的分布式存储系统中提供快速故障修复方法，缩小不可用数据的时间窗口，并满足各种前台应用程序严格的延迟、可用性和持久性要求至关重要。

3.2.1　故障修复的网络瓶颈

针对满足 MDS 性质的纠删码，重构某个故障数据块的过程如下。首先，选择一个替代节点。基于可靠性的需求，替代节点不能存有与待修复的数据块在同一个条带的某个数据块或校验块。然后，从 $k+m-1$ 个源节点中读取与待修

复的数据块在同一条带中的 k 个块，通过解码修复丢失的块，最后将解码生成的块写入替代节点的持久化存储中。回顾第 1.2.3 节中关于单个故障数据块的重构时间的四阶段分解：(1) t_1 为源节点从本地磁盘读取 k 个块到网卡缓冲区的用时；(2) t_2 为将可用块从源节点传输到替代节点的用时；(3) t_3 为解码用时；(4) t_4 为替代节点上持久化已解码块的用时。设块大小为 B，源节点 s 的读磁盘带宽为 $B^s_{I/O}$，替代节点 r 的写磁盘带宽为 $B^r_{I/O}$，网络带宽为 B_w。修复时间可以估计为 $t = t_1 + t_2 + t_3 + t_4 = \max_s(\frac{B}{B^s_{I/O}}) + \frac{kB}{B_w} + t_{decoding} + \frac{B}{B^r_{I/O}}$。

为了解在实际系统中每个步骤的运行时间，在一个由 18 个节点组成的分布式存储系统上进行了实验分析，其中每个节点配有两个 Xeon(R) E5-2650 CPU、64GB 的 DRAM 和一个 500GB 的 SSD 盘，并通过 10Gbps 网络互连。所部署的系统为 HDFS 3，使用 RS 码作为编码方案，块大小设置为 HDFS 默认的 128MB，修复的网络带宽设置为盘古系统建议的配置 30MB/s [50]。表 3.1 给出了配置不同 RS 码的修复时间。这四个步骤中，网络传输时间 t_2 在总修复时间的占比最大，约为 92%，而 $t_1 + t_4$ 与纠删码参数几乎无关，比 t_2 小两个数量级。此外，当数据块的数量 k 变大时，t_2 和 t_3 都会增加，这是因为 k 变大时，修复一个数据块会导致更多的网络流量，以及更高的解码计算复杂度。而不论 k 如何取值，总修复时长中 t_2 的占比几乎是恒定的。这可以解释如下：如果 k 增加 α 倍，则网络传输时间与总修复时间的比例约为 $\frac{\alpha*t_2}{t_1+\alpha*t_2+\alpha*t_3+t_4} \approx \frac{\alpha*t_2}{\alpha*t_2+\alpha*t_3} = \frac{t_2}{t_2+t_3} \approx \frac{t_2}{t_1+t_2+t_3+t_4}$。因此，从源节点到替代节点的数据传输时间 t_2 占整个修复过程的绝大部分时间，修复过程面临的瓶颈主要是网络传输。

表 3.1　使用 RS 码和 30MB/s 网络带宽重构一个块的时间成本（其中第三列的百分比为网络传输时长占重构总时长的比例）

(k, m)	$t_1(ms)$	$t_2(ms)$	$t_3(ms)$	$t_4(ms)$
(3, 2)	30	12375(91.80%)	648	427
(6, 3)	51	25907(93.26%)	1367	454
(10, 4)	68	43431(92.56%)	2828	596

如果当网络带宽配置更为充裕时，如盘古系统中的 Pangu-mid 和 Pangu-fast 分别配置 90MB/s 和 150MB/s 的修复带宽 [50]，这时候 k 个块的网络传输时间 t_2 分别是 30MB/s 带宽的三分之一和五分之一左右。但是网络传输时间仍然比其他步骤的耗时高一到两个数量级，所以 t_2 仍然主导着整个修复过程。

3.2.2　修复批次内数据非均匀分布

在故障数据修复过程中，为了保证对前台请求的服务质量，分布式存储系统通常分配有限的修复带宽用于故障数据修复，并且分批次修复丢失的数据，在每个批次里，待修复的故障数据有限。例如，HDFS 的默认批次大小是幸存数据节点数的两倍。为了验证分批修复的必要性，在上述 HDFS 系统中分 15 批修复 512

个 128MB 的数据块（共 60GB），一个故障块对应一个修复任务，并与不分批的处理方式进行性能比较。图 3.1 描述了系统随时间推移完成的修复任务数，不分批修复的总用时比分批处理长 14.49%。这是因为不分批修复的过程中有过多的修复任务并发运行，导致严重的资源竞争，其中一些修复任务在超时后会重新启动，被视为新任务，从而多次执行冗余修复，导致系统引入了 12.30% 的额外修复任务。因此，分批修复是存储系统有效的设计策略。

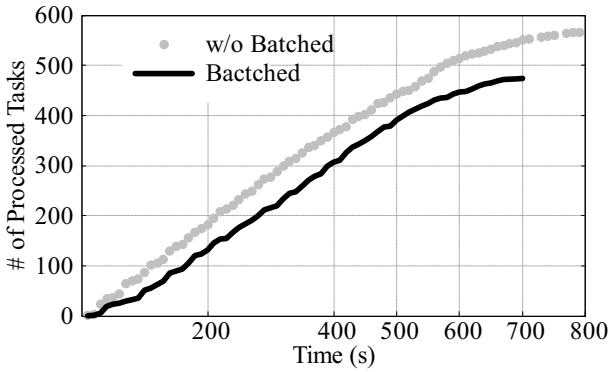


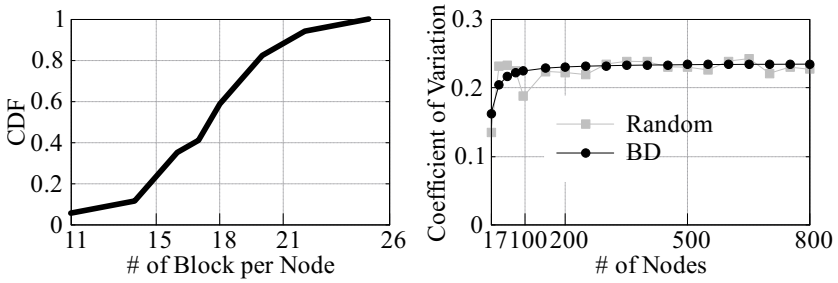
图 3.1　RS(6, 3) 码和 150MB/s 带宽下的分批修复与不分批修复完成修复任务的时间比较

然而，分批修复策略面临着严重的修复负载不均衡的问题，在一个批次的修复过程中，往往会出现部分节点的网络流量和解码计算负载过重，而其它节点的修复带宽没有得到充分利用，从而降低了修复速率。为了分析分批修复的负载不均衡程度，本节给出 HDFS 系统中每个修复批次在各个节点上负载情况的实验分析。实验中采用 HDFS 常用的数据放置方式，将数据块/校验块随机分布到各个物理节点，所有的条带均采用 RS(6, 3) 编码。默认情况下，HDFS 用于修复的批次大小是幸存数据节点数的两倍。在实验用的 18 节点集群中，除去 1 个主节点（NameNode），有 17 个数据节点作为数据存储节点，因此批次的默认大小为 34。对于一批修复任务，理论上将有 $(6+3)\times 34 = 306$ 个块随机分布在 17 个存储节点上，平均每个节点有 18 个块。然而，如图 3.2(a) 所示，实际上各节点保存的块数量并不均匀。大约 70% 的节点存有 15 ∼ 21 个块。存储块数最多的节点有 25 个块，而最少的节点仅有 12 个块，两者有 2.08 倍的存储数据量的差距。批次内不均匀的数据分布是导致修复负载严重不均衡问题的主要原因。

接下来，我们用变异系数（简记为 CV），即标准差与均值的比值，对上述数据布局不均匀现象进行理论分析。假设我们将 $2n$ 个条带中的 $2n\times(k+m)$ 个的块随机分布到 n 个节点。则一个块分配给某个节点的概率为 $p = (k+m)/n$。一个节点上存储的块数满足具有相同概率 p 的 $2n$ 个事件的二项式分布。因此，

$$CV = \frac{\sqrt{2np(1-p)}}{2np} = \sqrt{\frac{1-\frac{k+m}{n}}{2(k+m)}}$$

在大规模分布式存储系统中，系统节点数 n 的值往往大于几百，因此 $k+m \ll n$。



(a) Coefficient of variation　　(b) Data distribution

图 3.2　模拟 HDFS 系统中 RS(6, 3) 码的数据分布（(b) 中 Random 和 BD 分别表示实验结果以及二项分布的拟合结果）

所以有，

$$CV = \sqrt{\frac{1 - \frac{k+m}{n}}{2(k+m)}} \approx \sqrt{\frac{1}{2(k+m)}}$$

公式显示当系统规模足够大时，变异系数 CV 近似由 k 和 m 决定，与分布式存储系统规模 n 基本无关。图 3.2(b) 呈现了 RS(6, 3) 码一个批次内数据分布的 CV 与系统规模的关系图。从图中可以看出，我们模型估计的 CV 值与实验数据基本匹配。其次，CV 值随着集群规模的扩展而增加，当系统有不少于 100 个节点时，CV 趋于稳定。对于 RS(6, 3) 码，稳定后的 CV 恰好是按二项式分布估计的值 $\sqrt{1/18} \approx 0.2357$。

综上所述，我们的目标是从大量的待修复数据中，选出一些修复任务，将其打包在一个批次中，使得一个批次待修复任务对应条带中的数据块/校验块分布均衡，最终实现修复负载均衡。因为各个节点上都存有大量的数据块/校验块，使得我们有可能挑选出一批次的待修复数据，达到批次内负载均衡。

3.3　一个基于二分图的分批修复模型

本节介绍如何用二分图模型来刻画一批待修复数据的源节点和替代节点之间的映射关系，由此计算这一批修复任务引起的网络流量和修复负载的分布情况。为了便于理解，我们假设分布式存储系统的网络是同构的，即所有节点的下行和上行带宽都相同。我们进一步假设只有一个节点发生故障并且分布式存储系统部署了 (k, m) MDS 纠删码，因此每个条带中最多丢失一个块。我们称一个丢失块的修复是一个*修复任务*（用 T 表示）。我们将 n 个幸存节点的集合表示为 $\mathbb{N} = \{N_1, N_2, \cdots, N_n\}$。在本章的后面，我们将扩展此模型，使其适用于异构网络和多节点故障的情形。

3.3.1　替代节点图

给定一个批次中的 n 个修复任务集合 $\mathbb{T} = \{T_1, T_2, \cdots, T_n\}$，我们将替代节点的选择方式建模为一个二分图 $G_r = (\mathbb{T}, \Delta, \mathbb{N})$。在图 G_r 中，每个 $T_i \in \mathbb{T}$ 代表一个任务顶点，每个 $N_j \in \mathbb{N}$ 代表幸存顶点，而边 (T_i, N_j) 表示幸存节点 N_j 可以作为任务 T_i 的替代节点。我们称这样的图 G_r 为替代节点图。

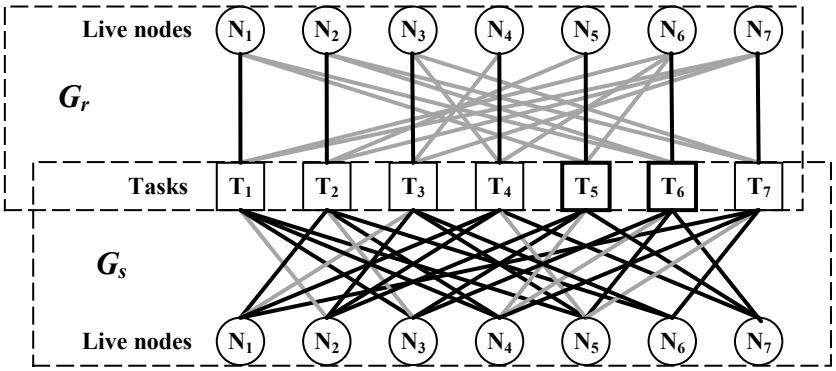


图 3.3　在部署 $(3,2)$ 纠删码并包含 7 个幸存节点的分布式存储系统中，应用替代节点图 G_r 和源节点图 G_s 调度 7 个修复任务的一个示例（灰边表示可能的选择，黑边表示均衡调度启用的实际选择）

若图 G_r 中存在完美匹配，则所有 n 个修复任务可以平均分配给 n 个幸存节点，每个节点执行一个修复任务。此时在相对应的这批修复过程中，所有幸存节点之间的下行网络流量和解码负载是均衡的。图 3.3 上半部说明了具有完美匹配的示例 G_r。每个任务 T_i 有 3 条边。例如，(T_1, N_1)、(T_1, N_6) 和 (T_1, N_7) 表示三个节点 N_1、N_6 和 N_7 中的任意一个都可以作为任务 T_1 的替代节点。在图 3.3 中，黑色的边对应于一个均衡调度的完美匹配，其中每个幸存节点都执行一个修复任务。

每个条带中有 $k+m$ 个数据块/校验块，同一个条带中的两个块不能存储在同一个节点上，因此系统中有 $k+m$ 个节点存储了某个条带中的块，其余节点没有存储该条带中的块。当一个节点发生故障时，每个被修复的数据块不能与同一个条带中的块存储于同一个节点，因此系统中有 $n-(k+m-1)$ 个节点可以作为一个被修复数据块的替代节点，每个任务顶点 T_i 的度数为 $n-(k+m-1)$。对于大规模分布式存储系统来说，$k+m \ll n$，所以 G_r 是一个密集图，在绝大多数情况下 G_r 中都存在完美匹配（参见表 3.2 中的结果）。值得一提的是，虽然绝大多数情况下图 G_r 中存在完美匹配，但仍会出现极少数没有完美匹配的情形。此时，我们可以使用最大匹配和一些优化方法来优化替代节点的选择。

3.3.2　源节点图

与替代节点图类似，定义另一个二分图 $G_s = (\mathbb{T}, \Delta, \mathbb{N})$，称为源节点图，对源节点的选择方案建立模型。在 G_s 中，每个 $T_i \in \mathbb{T}$ 代表一个任务顶点，而每个

$N_j \in \mathbb{N}$ 代表一个*源数据顶点*，而边 (T_i, N_j) 表示 N_j 是 T_i 的源节点，即 N_j 中存储了一个数据块，而该数据块与 T_i 对应的故障数据块在同一个条带中，可以参与修复对应的故障数据块。如果我们在 G_s 中找到一个 k-正则生成子图，那么每个任务可以读取 k 个幸存块进行修复，并且每个存储节点可以为 k 个任务提供 k 个块，故而每个节点的上行修复流量达到均衡。图 3.3 的下半部分给出了一个示例图 G_s。以 T_1 为例，它有 4 条边，即 (T_1, N_2)、(T_1, N_3)、(T_1, N_4) 和 (T_1, N_5)。这意味着 N_2、N_3、N_4 和 N_5 中的每一个幸存节点都可以为任务 T_1 提供数据块，用于故障数据修复。在图 3.3 中，黑边表示 G_s 的一个 3-正则生成子图，这意味着每个任务可以接收 3 个块进行故障数据修复，并且每个幸存节点贡献 3 个块，用于故障数据修复，达到上行修复流量的均衡。

当一个块发生故障时，存储系统中有 $k+m-1$ 个可用块来修复它。这表明在 G_s 中，每个任务顶点都连接到 $k+m-1$ 个源数据顶点。因此，源节点的选择等价于找到 G_s 的一个生成子图 H_s，使得在 H_s 中所有任务顶点都具有相同的度数 k，而源数据顶点的度数等于从对应源节点读取的块数。为了均衡幸存节点之间的上行流量，我们需要最小化各源数据顶点之间的度数差异。显然，如果所有源数据顶点的度数都是 k，则达到了*完全均衡*。然而，由于每个批次内的数据布局不均匀，很难在 G_s 中找到 k-正则子图（见表 3.2）。由于当前存储系统中的故障节点一般存储容量很大，例如盘古的 “胖节点” 容量达到 72TB [50]，所以形成的待修复队列通常是很长的。幸运的是，我们始终可以从其足够长的待处理修复任务队列中选择 n 个任务，使得源节点的上行流量达到（近似的）完全均衡。我们将在下一小节介绍一批修复任务的选择算法。

3.3.3　一批修复任务选择算法

对于一个批次中的 n 个修复任务，若我们在 G_r 中找到完美匹配，并且 G_s 中找到 k-正则生成子图，就可以为每个修复任务分配 1 个替代节点和 k 个源节点，使得该批次的修复流量与解码负载在所有幸存节点中达到均衡。本小节介绍如何选择 n 个修复任务，将其打包成一个批次，达到修复流量与解码负载的均衡。

由于在 G_s 中找到 k-正则生成子图比在 G_r 中找到完美匹配更困难，我们将按照以下思路来选择 n 个修复任务，将其打包到同一个批次。首先找到 n 个修复任务，使得 G_s 中存在 k-正则生成子图；然后由这 n 个任务构造 G_r，并且在 G_r 中为这 n 个修复任务确定一个完美匹配。需要说明的是，并不是永远都能够在 G_s 中找到 k-正则生成子图，并且在 G_r 中找到完美匹配。为了克服这个问题，我们设计了一个批处理算法，并使用最大匹配算法来最大程度地均衡修复负载。事实上，分布式存储系统故障节点的容量很大，其中有非常多的待修复数据，在实验中，在绝大多数的批次中，我们都可以在 G_r 中找到完美匹配，并且在 G_s 中

有 30% 的批次可以找到 k-正则生成子图（远大于表 3.2 中的概率）。余下的 70% 批次，我们也可以使它们的修复负载几乎达到均衡。

表 3.2　基于 RS$(3,2)$ 码的存储系统中，源节点图中存在 k-正则子图以及替代节点图中存在完美匹配的概率，表中数值是 100 次实验的平均值。

节点编号	7	10	13	16	19
替代节点	0.85	1	1	1	1
源节点	0.43	0.13	0.03	0	0

在修复过程中，理论上每批次的大小（即其中待修复任务数）没有限制。为了负载均衡，每个批次的大小应该是幸存节点数的整数倍。我们可以通过使用上述模型进行 l 次负载均衡，将每 $l \times n$ 个修复任务打包到一个批次。但是，修复性能受批次大小的影响，最优的批次大小会受到并发任务资源冲突、超时参数等多种因素的影响，与存储系统的配置密切相关。我们将在性能评估那一节，通过实验进一步讨论批次大小对修复性能的影响。另一方面，尽管上面介绍的模型假定分布式存储系统环境为同构的，但可以在 G_r 和 G_s 中设定权值，扩展 G_r 和 G_s，使它们可以应用于异构环境。

3.4　一种负载均衡的故障修复调度模块设计

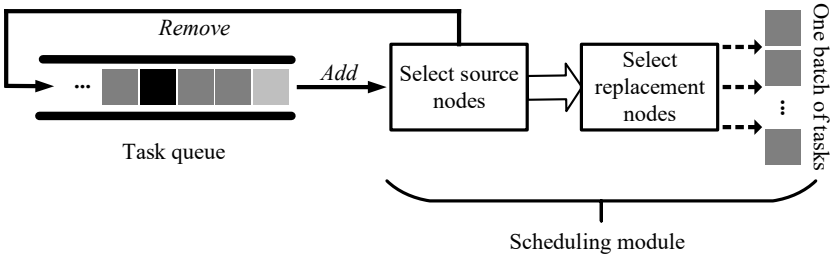


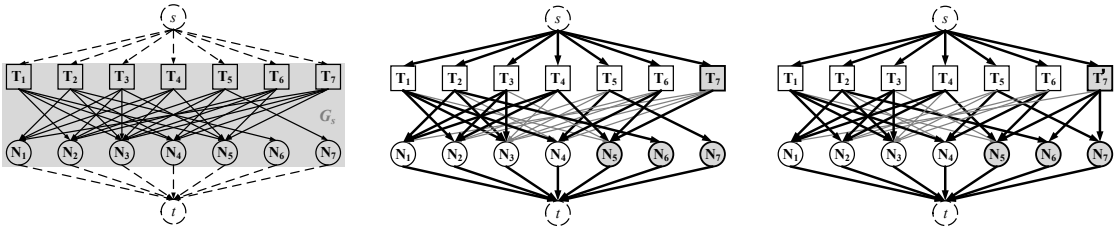
图 3.4　SelectiveEC 的整体架构

基于上一节介绍的模型, 图 3.4 概要描述了故障修复的调度方法 SelectiveEC。在发生故障时，分布式存储系统中有许多丢失的块需要修复，这些待修复的块存放于修复任务队列中。SelectiveEC 从队列中选择 n 个修复任务，将其打包到一个批次中，而不是通过传统队列先入先出（FIFO）的简单分批方式，以达成网络流量和修复负载的均衡。SelectiveEC 由以下三个步骤组成：

步骤 (1)：通过批处理算法找到 n 个修复任务的集合 $\mathbb{T}$，使得对应的 G_s 中存在一个 k-正则生成子图 H_s。如果找到，则上行修复流量完全均衡。否则，批处理算法找到一组任务，其中幸存节点之间的上行修复流量的差异最小。

步骤 (2)：基于 $\mathbb{T}$，构造 G_r，在 G_r 中找到最大匹配 $\mathbb{M}$。如果 $\mathbb{M}$ 是完美匹配，则下行修复流量和解码负载完全均衡；否则，为近似均衡。

步骤 (3)：将 $\mathbb{T}$ 中的任务打包成一个批次，然后根据步骤 (2) 中的匹配 $\mathbb{M}$ 和步骤 (1) 中的子图 H_s 分配替代节点和源节点，最后执行该批次的修复任务。



(a) Flow graph of G_s　(b) A maximum flow　(c) An updated maximum flow

图 3.5　基于 G_s 的分布式存储系统中修复任务的分批示例，其中有 7 个幸存节点，部署了 $(3,2)$ 纠删码（(a) G_s 的流图 F_{G_s}，每条虚线的边容量为 k，每条实线的边容量为 1。(b) 流图 F_{G_s} 的一个最大流，从任务顶点到源数据顶点的黑边上的流量都是 1，对应了为每个任务选择的源节点。最大流值为 $17 < nk = 3 \times 7 = 21$。任务 T_7 将被替换，它只与一个不饱和源数据顶点 N_5 连接。(c) 更新后的流图及最大流，最大流值为 19，替换进来的 T_7' 连接的不饱和源数据顶点比 T_7 多两个，从而拥有更大的流值）

以上三个步骤仅仅概述了 SelectiveEC 的基本思想，在后面的章节将介绍 SelectiveEC 的详细流程。

3.4.1　单节点故障修复调度

在本小节，我们将针对同构分布式存储系统中单节点故障的修复问题，介绍 SelectiveEC 的详细流程，在后面的两个小节，再讨论如何扩展到多节点故障与异构存储系统的情形。

给定一组 n 个修复任务的集合 $\mathbb{T}$，我们构建一个源节点图 G_s。基于 G_s，然后构造流图 F_{G_s} 如下：(1) 在 G_s 上添加一个顶点 s，作为 F_{G_s} 的 *source*，并且从 s 到 G_s 中每个任务顶点添加一条有向边，容量为 k；(2) 添加另一个顶点 t，作为 F_{G_s} 的 *sink*，并且从每个源数据顶点向 t 添加一条有向边，容量为 k；(3) 将 G_s 中所有的无向边转换为从任务顶点到源数据顶点的有向边，容量设置为 1。图 3.5(a) 给出了流图的构造示例。

对于 F_{G_s} 的一个流函数 f，如果某任务顶点流出的流量是 k，表示该顶点对应的修复任务能够读到 k 个可用块来完成修复任务，即可以修复对应的丢失块。在这种情况下，我们称这个任务顶点是*饱和的*。此外，流进每个源数据顶点的流量最多为 k，所以一个批次中的所有修复任务从一个源节点最多读 k 个块。我们的目标是找到一个流图 F_{G_s}，具有最大流 f，使得所有任务顶点都饱和，并且所有源数据顶点都具有相同的流进流量 k（我们也称此类源数据顶点是*饱和的*），使得从每个幸存节点都读取 k 个块用于故障数据的修复，在幸存节点间达到上行修复流量的均衡。我们期望 f 的流函数值是 nk，这样我们就可以构造一个 G_s 的 k-正则生成子图 H_s，其中，H_s 的边对应于 F_{G_s} 中流值为 1 的有向边。

接下来，介绍一种近似最优的一批修复任务的选择算法，将 n 个修复任务打包成一个批次，并为每个任务选择源节点与替代节点。

修复任务选择算法:

步骤 (1)：任选 n 个任务，组成 $\mathbb{T}$。比如说，可以从任务队列中顺序选择 n 个任务。

步骤 (2)：根据 $\mathbb{T}$ 构造 G_s 和 F_{G_s}。

步骤 (3)：找出 F_{G_s} 的最大流 f。

步骤 (4)：如果流 f 的流量为 nk，则 G_s 中存在 k-正则生成子图，转步骤 (5)。否则，在 G_s 中找出流出流量最小的任务顶点，设为 $T_i \in \mathbb{T}$。从剩余任务队列中选择另一个任务 $T \notin \mathbb{T}$，使得 T 的不饱和源数据顶点比 T_i 的多。用 T 替换 $\mathbb{T}$ 中的 T_i，即令 $\mathbb{T} \leftarrow (\mathbb{T} - \{T_i\}) \cup \{T\}$，转到步骤 (2)。如果找不到这样的 T，则转到步骤 (5)。

步骤 (5)：根据流 f 构造 H_s。停止。

图 3.5(b) 显示了图 3.5(a) 的最大流，其流量为 $17 < 3 \times 7 = 21$。选择流出流量最小的任务顶点 T_7(其流量为 1)，将作为备选任务从 $\mathbb{T}$ 中换出。T_7 仅仅与一个源数据顶点 N_5 相连，在为 T_7 找到源数据顶点之前，N_5 是不饱和的，即 T_7 只连接了一个不饱和源数据顶点 N_5。由算法的第 (4) 步，找到一个新任务 T_7'，它连接了三个不饱和源数据顶点，即 N_5、N_6 和 N_7。用 T_7' 代替 T_7，更新流图并找到一个新的最大流，其流量增加 2（见图 3.5(c)）。

下面的定理 3.1说明了修复任务选择算法的正确性。

定理 3.1　经过修复任务选择算法的每一轮迭代，最大流的值至少增加 1。因此修复任务选择算法将正确终止。

证明　在修复任务选择算法的每一轮迭代中，我们在第 (3) 步计算当前流图 F_{G_s} 的最大流 f（例如，用 Ford-Fulkerson 算法 [91]），其值为 $Val(f)$。如果 $Val(f) < nk$，则 F_{G_s} 中存在不饱和任务顶点。设流出任务顶点 T 的流量最小，显然 T 是不饱和任务顶点。设 T 与 $f_T(< k)$ 个不饱和源数据顶点相连。如果在步骤 (4) 找到一个任务 T'，用 T' 替换 T，算法将执行下一轮迭代，并将 F_{G_s} 更新为 $F_{G_s'}$。根据步骤 (4) 中的替换规则，假设 T' 与 l 个不饱和源数据顶点相连，则有 $l > f_T$（参见图 3.5(b) 和 (c)）。基于 $F_{G_s} - T$ 中的最大流，$F_{G_s'}$ 中至少存在 l 条可增广路径，因此 G_s' 中最大流 f' 的值满足 $Val(f') \geqslant Val(f) - f_T + l \geqslant Val(f) + 1 > Val(f)$。所以在每次迭代中，最大流的值至少增加 1。否则，算法将会终止。 ∎

下面介绍替代节点的选择。替代节点的选择决定了一个批次修复任务的执行过程中，系统中节点的下行修复流量和解码负载的均衡程度。针对上述修复任务选择算法生成的一批修复任务，我们构建一个替代节点图 G_r，用最大匹配算法（例如，匈牙利算法 [91]）在 G_r 中找到一个最大匹配，并据此选择每个任务的替代节点。如果最大匹配是 G_r 中的完美匹配，则修复的并行度最大，且下行修复流量和解码负载达到完全均衡。事实上，由于在大型或中等规模的分布式存

储系统中有很多节点，通常 $k+m \ll n$，二分图 G_r 中每个顶点的度数都很大。因此，在实际系统中，G_r 基本上都满足存在完美匹配的霍尔定理 [91-92]，这意味着通常都可以找到用于选择替代节点的完美匹配（见表 3.2）。以下定理 3.2 验证了这一点。

定理 3.2　设分布式存储系统中有 n 个幸存节点，采用 (k,m) 纠删码。在发生单节点故障的情况下，给定一组 n 个修复任务的集合 $\mathbb{T}$，G_r 中存在一个完美匹配的概率为 $P \geqslant 1-e^{-\frac{1}{2p}(p-\frac{1}{2})^2 n}$，其中 $p=\frac{1}{n-(k+m-1)}$。例如，当 $n \geqslant 100$，采用 RS(6,3) 码时，$P>99\%$。

证明　由霍尔定理可知，如果 G_r 中顶点的最小度数大于等于 $n/2$，则存在完美匹配 [91-92]。我们通过检查 G_r 中每个顶点的度数来给出证明。一方面，在发生单节点故障的情况下，每个任务顶点的度数为 $n-(k+m-1)$，当 $n \geqslant 2(k+m-1)$ 时，则大于 $n/2$。在大规模或中等规模的分布式存储系统中，$k+m \ll n$，所以 $n-(k+m-1) \geqslant n/2$ 自然成立。另一方面，对于每个幸存节点来说，该节点作为任务的替代节点的事件是独立的，概率为 $p=\frac{1}{n-(k+m-1)}$。所以每个替代节点顶点的度数，即对应的幸存节点上可以执行的修复任务的数量，遵循参数为 n 和 p 的二项分布。根据切尔诺夫界 [93]，满足 $P=Pr[X>n/2] \geqslant 1-e^{-\frac{1}{2p}n(p-\frac{1}{2})^2}$。随着 n 的增加，P 接近 1。例如，当 $n \geqslant 100$ 时，采用 RS(6,3) 码时，$P>99\%$。所以在大型或中等规模的分布式存储系统中，G_r 中通常都存在完美匹配。 ∎

SelectiveEC 的时间复杂度：SelectiveEC 的计算时间主要来自对源节点和替代节点的选择。在*修复任务选择算法*的每次迭代中，步骤 (1)-(3) 的主要运行时间是在 G_s 中找最大流。假定使用 Ford-Fulkerson 算法 [91]，其时间复杂度为 $\mathcal{O}(2n^2(k+m-1))$。步骤 (4) 是在 $\mathbb{T}$ 中找到一个具有最小流出流量的任务 T，使得 T 连接到的不饱和源数据顶点最少；步骤 (4) 还需要从长度为 L 的待处理任务队列中找到一个新任务来替换 $\mathbb{T}$，其时间复杂度是 $\mathcal{O}(n+n+(n+(k+m-1))L)$。由定理 3.1知，*修复任务选择算法*每迭代一轮，对应的最大流的值至少增加 1。而且，在算法的执行过程中，初始批次的修复任务对应的最大流的值至少是 $k(k+m-1)$；而算法结束时，一个批次修复任务的最大流的值最大为 kn，所以*修复任务选择算法*最多迭代运行 $kn-k(k+m-1)$ 轮。故 SelectiveEC 的时间复杂度为 $\mathcal{O}(2n^3k^2+kn^2L)$。

SelectiveEC 的时间复杂度随着待处理修复任务队列的长度 L 或集群规模 n 的增加而增加，从而影响到整个修复速度。以下是一些优化策略，从而减少 SelectiveEC 在大规模分布式存储系统中的调度开销。

大规模分布式存储系统：在实际部署中，大规模分布式存储系统通常被划分成区域（region）、分区（zone）或机架（rack），然后将数据存储到分布式存储系统的某个子集群中。因此，SelectiveEC 一般运行在节点数量有限的子集群中，

避免 n 过大，导致时间复杂度过高。

长任务队列：若某个“胖节点”上存有数百万个块 [50]，一旦该节点发生故障，待修复任务队列中会有大量的修复任务，增加算法的时间复杂度。因为 $L > n^2$，我们将任务分成多个组，在每个组内 L 变小，降低修复任务选择算法的时间复杂度。另外，分批调度算法的执行时间远比故障数据修复时间短，可以通过流水线方式将其隐藏在上一批次的修复过程中。

在本章的性能分析中，实验结果表明，在具有 18 个节点的真实 HDFS 系统中，修复任务队列长度约为 1600，SelectiveEC 仅使用 100 个批次进行故障修复，就可以达到很好的修复性能（参见图 3.12）。更进一步，通过模拟实验表明，即使在 700 节点的大规模分布式存储系统中，SelectiveEC 的调度开销也可以完美隐藏在调度和修复的流水线执行过程中（见图 3.16(b)）。

虽然 SelectiveEC 尽量均衡地分配了修复工作量，但在某些情况下，一个批次中仍存在不饱和顶点。为此，我们设计了启发式算法，进一步提高修复速度。

(1) 任务顶点不饱和：修复任务选择算法可能找不到最大流 f，使得所有任务顶点都饱和，尤其是在修复过程接近尾声，待修复任务不多，从中寻找替换任务的可选择性不大；或者系统规模很小的时候。若任务顶点不饱和，则读不到 k 块源数据，导致故障数据无法修复。在这种情况下，我们放宽从每个源数据顶点读取不超过 k 块的限制（此前从源顶点到 t 的有向边的容量都是 k）。对于修复任务选择算法尾期的不饱和任务，我们启动了一种启发式算法，该算法逐一检查不饱和任务顶点，并为其补充源节点。对于每个不饱和任务顶点 T，从 T 潜在的源数据顶点中选择负载最轻的顶点，将其更新为 T 选择的源数据顶点，直到 T 连接 k 个源数据顶点。

(2) 不存在完美匹配：在调度替代节点时也可能找不到完美的匹配。类似地，我们放宽每个幸存节点在一个批次的修复中最多修复一个任务的限制，对没有指定替代节点的每个任务，从其替代节点的候选中选择负载最轻的节点来执行。

启发式算法的时间复杂度来自两个方面: 扫描每个节点的流值, 时间为 $\mathcal{O}(n)$; 并根据源数据顶点/替代节点顶点的流入流量/流出流量的大小对其进行排序，选择不饱和任务顶点和没有安排替代节点的任务顶点，时间为 $\mathcal{O}(n \log n)$。与调度开销相比，启发式算法的时间复杂度可以忽略不计。

3.4.2　异构环境修复调度

在实际的分布式存储系统中，通常优先服务于前台的在线应用程序，所以网络带宽占用以及存储 I/O 等资源经常是动态变化的，从而造成了异构的修复环境。为了将 SelectiveEC 应用到异构环境中，我们通过将可用带宽的权重添加到流图中，对 SelectiveEC 进行扩展。以调度修复任务的源节点为例，设 B^i_{out} 为从

节点 N_i 实时收集的可用上行带宽，其平均值为

$$\overline{B}_{out} = \sum_i B^i_{out}/n$$

显然，从带宽更高的源节点读取更多的块，可以加快故障修复过程。我们将连接源顶点 N_i 到 t 的边的容量设置为 $\lceil k\frac{B^i_{out}}{B_{out}}\rceil$。此时，该加权流图中的最大流仍对应一批修复任务及其源节点的选择，而且可以有效地使用带宽，加快修复速度。替代节点的选择与此类似。

3.4.3　多节点故障修复调度

SelectiveEC 同时支持多节点故障修复。当多个节点故障时，待修复的条带要么只丢失 1 个块，要么丢失多个块。对于丢失了多个块的待修复条带，由一个主替代节点专门执行修复任务，修复完成后在主替代节点本地存储一个被修复的块，并将其他被修复的块迁移到辅助替代节点。此时，除了均衡修复下行流量及解码负载，我们还需要均衡迁移被修复块的工作负载。所以调度过程可以分为两个步骤。在步骤 (1) 中，SelectiveEC 执行类似于单节点故障修复的调度，以达到源节点和主替代节点的均衡选择。在步骤 (2) 中，SelectiveEC 通过寻找最大匹配或最大流来均衡地将被修复块从主替代节点分配到辅助替代节点上。

综上所述，均衡调度模块 SelectiveEC 用于在分布式存储系统中进行有效的故障修复，同时可以支持单节点以及多节点的故障修复，进一步支持异构的网络和磁盘 I/O 带宽等环境。通过对系统故障时修复任务的实时调度，SelectiveEC 一方面可以加快故障修复的速度，另一方面可以在故障修复时做到对前端应用更小的性能干扰。

3.5　系统实现

SelectiveEC 原型系统基于 Hadoop 3.1.2 纠删码模块实现，共有 3500 行 JAVA 代码[①]。HDFS 在 2014 年引入纠删码的模块用于保证系统可靠性，之后迭代更新了许多个优化的版本 [24, 94-95]。图 3.6 刻画了 SelectiveEC 的实现概要图。

系统原型实现： SelectiveEC 在节点发生故障的情况下，修复任务存入 NameNode 中的待处理任务队列。NameNode 上的 ECManager 负责调度修复任务，默认情况下首先将修复任务分批处理，每批 $2n$ 个修复任务（n 为集群幸存节点数），然后为每个修复任务选择一个替代节点，最后将修复任务分发到对应的替代节点所在的 DataNode 上。关于修复任务的源节点选择，HDFS 在 NameNode 中为修复任务维护一个候选源节点列表 “BlockIndices”，并通过 RPC 将其发送到替

[①]开源代码链接见：https://github.com/ADSL-EC/SelectiveEC。

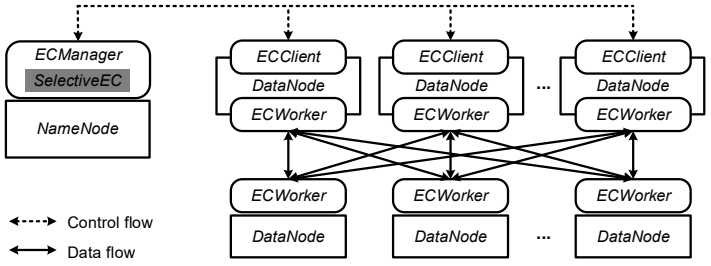


图 3.6　SelectiveEC 的原型系统实现概要图

代节点，其中前 k 个节点将参与修复。我们将 SelectiveEC 嵌入到 ECManager 中，将 n 个任务打包成一个批次，在各个节点间均衡修复流量和解码负载。ECClient（替代节点所属）负责从多个 DataNode（源节点）读取数据，其中 DataNodes（源节点）在修复 I/O 操作期间无感知修复任务（采用零拷贝策略）。ECWorker 守护线程用于监听来自 ECManager 的修复命令。它通过从关联的 DataNode 中读取数据、执行解码等来服务这些修复请求。如果需要在同一个条带中修复多个块，主的替代节点则会发送修复好的块到其他从的替代节点上，整个过程由 ECWorker 负责处理这些请求。

系统优化：为避免多批次之间严重地堆积修复任务，HDFS 默认设置连续批次发放的时间间隔为 3 秒（一次心跳的时间）。一批的修复任务完成时间取决于纠删码的参数、分布式存储系统的配置以及修复的数据量，所以配置以及需求差别很大。例如在我们的实验中，RS(3, 2) 码, RS(6, 3) 码和 RS(10, 4) 码的网络传输用时分别为 1.7 秒，3.2 秒和 5.8 秒左右。对于 RS(10, 4) 码来说，3 秒显然无法完成一个批次任务的修复，所以设置固定的批次间的时间间隔长度是不合理的。为了解决这个问题，我们把时间间隔设置为 $\frac{kB}{B_w}$（称为 $time_{slot}$），即传输 k 个源数据块到替代节点的网络用时。通过这种可调的批次间时间间隔，可以明显消除由于从 ECManager 发送一批任务太快或太慢而导致的性能下降。为了公平比较，我们在性能评估时将所有的故障修复方法都按照这种方式进行了优化。

额外的元数据开销：SelectiveEC 重用了 HDFS 中的主要元数据结构，包括 DataNode 和纠删码维护的条带组信息，几乎不会引入额外的 RPC 和元数据开销。

3.6　性能评估

3.6.1　实验设置

我们首先在本地的 18 个节点集群上部署 SelectiveEC，其中每个节点有 2 个 Intel(R) E5-2650 V4 CPU、64GB 内存、10Gbps 以太网网卡和 500GB SSD，运行 CentOS 7.8.2003 操作系统。此外，我们在 Amazon EC2 测试平台上也对其进行了评估，该测试平台由美国东部（俄亥俄州）地区的 50 个 m5.large 实例组成。每

个实例有两个 vCPU 和 8GB RAM 以及 10Gbps 的峰值网络带宽。

我们为 HDFS 3 配置了 1 个 NameNode 和 17 个 DataNodes。为了模拟节点故障，我们随机选择 DataNode，并人为地终止它们的进程。系统修复的默认配置是：30MB/s 的同构修复网络带宽，RS(6, 3) 码，16MB 块大小，待修复任务分成 100 批执行以及每批包含 16 个修复任务。我们为本地集群写入大约 286GB 数据（AWS 上为 2.45TB），单个故障节点会产生大约 30GB（AWS 为 84GB）的故障数据。我们还在实验中改变了网络带宽、纠删码参数和块大小。我们在实验图中展示了 5 次运行的平均实验结果。

表 3.3　调度方法的特性（SSN：调度源节点；SRN：调度替代节点）

调度方法	SSN	SRN	修复任务顺序
HDFS [85]	✗	✗	先入先出
G1 [54-55]	✓	✗	先入先出
G2 [56-57]	✓	✗	先入先出
SelectiveEC	✓	✓	有选择地

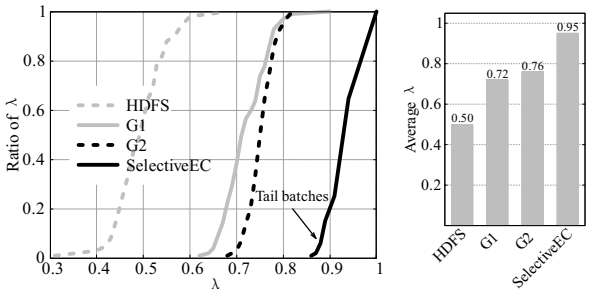
在大规模分布式存储系统中通常采用随机数据布局，从而达到存储负载均衡。所以实验中首先比较了 HDFS 默认系统配置，采取随机分布数据块和校验块以及随机选择源节点和替代节点参与修复。另外我们还比较了业界最先进的负载均衡调度方案：来自 CAR 的负载均衡方案 Greedy1 [54-55]，以及来自 ECPipe [56, 96] 和 PPR [57] 的 Greedy2。我们在表 3.3 总结了这几种调度方法的特性。Greedy1 从任务队列中依次初始化一批任务，逐一为每个任务选择源节点，以避免在有限的迭代中（比如 $n/2$ 次）出现负载过重的节点。Greedy2 通过将任务逐个地添加进来，选择当前 k 个负载最少的源节点，将任务打包成一个批次进行修复。他们通常仅对修复流量的负载均衡进行单方面优化，例如仅对源节点进行调度，或只是针对降级读等这样的特定场景进行优化。SelectiveEC 通过结合源节点调度和替代节点调度来实现修复流量的负载均衡，以实现更好的网络利用率，同时支持单节点和多节点故障修复，并且适用于同构和异构的网络环境。

3.6.2　单节点故障修复调度性能评估

1. 修复吞吐率

理想情况下，所有替代节点在 $time_{slot} = \frac{kB}{B_w}$ 时间内接收 k 个源数据块，然后它们可以同时批量执行丢失块的修复工作。令 # 是 $time_{slot}$ 内所有替代节点收到的块数。为了评估修复负载均衡的程度，我们定义了一个均衡因子 $\lambda = \frac{\#}{nk}$。当 $\lambda = 1$ 时，就达到了理想情况，完全均衡。λ 越大，上下行带宽负载均衡程度越高。

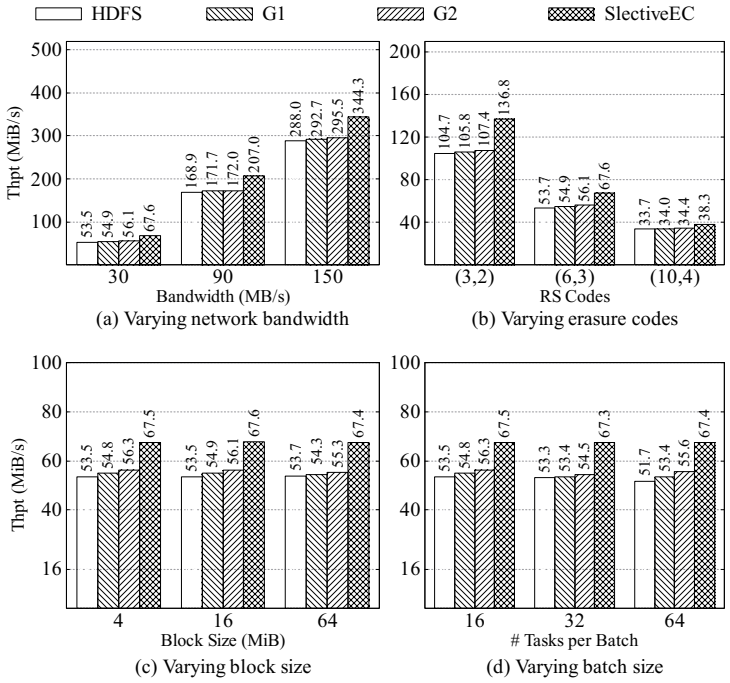
图 3.7(a) 和 3.7(b) 显示了 λ 在 100 个批次的 CDF 和平均值。SelectiveEC 的 λ 平均值为 0.95，而 HDFS 仅为 0.50，Greedy1 和 Greedy2 分别为 0.72 和 0.76，表明 SelectiveEC 实现了最佳负载均衡。



(a) CDF　　(b) Average

图 3.7　均衡因子 λ 的比较

我们发现 SelectiveEC 有少量一些批次的 $\lambda < 0.90$，这些批次只占总批次的 5%-10%，并且只出现在最后几批中，如图 3.7(a) 中黑色箭头所示。这是因为，在最后几批中，待选择修复任务队列中只剩下有限的修复任务来更新流图。但即便如此，SelectiveEC 的 $\lambda > 0.85$ 始终成立，表明 SelectiveEC 的 λ 随着队列长度的减小而下降非常缓慢。



(a) Varying network bandwidth　(b) Varying erasure codes

(c) Varying block size　(d) Varying batch size

图 3.8　不同参数下的修复吞吐率

在本地集群上的实验中，我们首先写入大约 500GB 的数据，然后随机宕机一个具有约 100 批修复任务的存储数据节点，最后执行单节点故障修复。我们通过改变可用网络带宽、纠删码参数和块大小来评估修复吞吐率。

修复带宽的影响：我们评估了不同修复带宽的性能，分别为 30MB/s、90MB/s 和 150MB/s（对应于盘古系统中的 Pangu-slow、Pangu-mid 和 Pangu-fast [50]）。在图 3.8(a) 中，SelectiveEC 的修复吞吐率高于三个比较对象。在网络带宽为 30MB/s 时，平均值比 HDFS、Greedy1 和 Greedy2 分别高出 26.31%、23.08% 和 20.35%。三个比较对象和 SelectiveEC 的修复吞吐率都随着可用修复带宽的增加而增加。

然而，当网络带宽为 90MB/s 和 150MB/s 时，SelectiveEC 相对于它们的优势会降低，因为高带宽时修复的网络传输用时减少，SelectiveEC 的优化策略对修复时间的相对贡献较小。更高的网络带宽意味着网络瓶颈将得到缓解。但是 SelectiveEC 不仅在网络传输上，而且在磁盘 I/O、CPU 和内存使用上都有更好的负载均衡。因此，即使有 150MB/s 的带宽，SelectiveEC 的修复吞吐率仍然分别比 HDFS、Greedy1 和 Greedy2 高出 19.54%、17.64% 和 16.52%。

纠删码的影响：我们接下来评估不同纠删码配置下的修复性能，即 RS(3, 2) 码、RS(6, 3) 码（Google 使用 [7]）和 RS(10, 4) 码（Facebook 使用 [26, 88]）。在图 3.8(b) 中，我们发现 SelectiveEC 与 HDFS、Greedy1 和 Greedy2 相比，SelectiveEC 实现了最高的修复吞吐率，对 RS(3, 2) 码，分别提高 30.68%、29.27% 和 27.38%。因为在只有 $n = 16$ 个幸存节点的小规模分布式存储系统中，用 RS(3, 2) 码比 RS(6, 3) 码和 RS(10, 4) 码更容易找到替代节点的完美匹配（参考定理 3.2）。

虽然 RS 码是最常用的，但也有一些流行的纠删码，例如 LRC 码 [14, 25] 和 MSR 码 [33-34]。因为用于修复的源节点的多样性和不确定性，SelectiveEC 现在不能直接应用于这些纠删码。例如，LRC 码并不能够使用任意 k 块源数据来修复丢失的块，但 SelectiveEC 有可能通过调整源节点的选择来支持 LRC。

块大小的影响：图 3.8(c) 展示了块大小设置为 4MB、16MB 和 64MB 时的修复吞吐率。所有比较对象和 SelectiveEC 的修复吞吐率在不同块大小下几乎没有变化，原因是我们在批次之间设置了一个可调整的时间间隔 ($\frac{kB}{B_w}$)。在当前批次的修复完成之前，便会预取下一批次任务的元数据，因此通过流水线，将元数据的管理过程隐藏在当前批次的修复中，不会影响修复时间。

批次大小的影响：图 3.8(d) 中展示了批次大小设置分别为 $n(n = 16)$、$2n$ 和 $4n$ 时的修复吞吐率。当批次大小为 $l \times n(l > 1)$ 时，SelectiveEC 的修复性能与批次大小为 n 时相似，其他三个算法的修复性能则略差一些。这是因为 SelectiveEC 高效的调度保证了一个批次中每 n 个任务都能使网络带宽饱和，但是过多的修复任务会产生大量的重构线程，从而占用更多的内存和 CPU 内核，这会造成激烈的竞争。因此，SelectiveEC 的批次大小被设置为 n，即分布式存储系统中幸存节点的数量。

2. 修复任务生命周期

为了更细粒度的理解修复过程的负载均衡状态，我们统计了每个修复任务的生命周期，也就是每个修复任务的开始以及结束时刻。我们根据执行顺序为每个任务分配一个 ID，最先启动的任务分配 ID 0。我们比较了 SelectiveEC 与 HDFS 的修复过程，每个任务的运行时间如图 3.9 所示（共 100 个批次的 1600 个修复任务）。

实验结果可以通过短线条的分布来观察，斜率表示单位时间内完成的任务

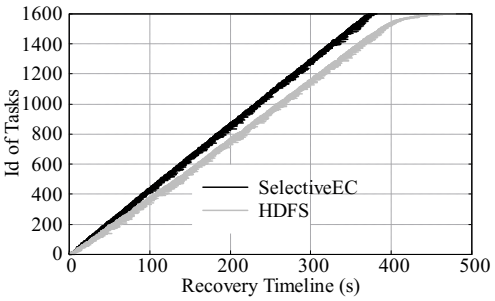


图 3.9　每个任务的运行时间

数。我们发现 SelectiveEC 的斜率更大，这意味着 SelectiveEC 比 HDFS 更快地修复了 1600 个任务，再次验证了我们的分析。具体来说，由于 SelectiveEC 更好的负载均衡，每批完成超过 90% 的任务，其 λ 几乎每批都超过 0.9。但是 HDFS 的 λ 约为 0.5，这意味着每批只能完成一半的修复任务，未完成的任务将推迟到下一批。虽然 HDFS 默认配置中允许一个 Datanode 并行运行 8 个修复线程，但多个线程相互干扰，导致它们都无法在理想的修复时间内完成，所以 HDFS 的修复速度比 SelectiveEC 慢。

此外，图 3.9 还展示了 HDFS 在修复时间上有一个长尾，而 SelectiveEC 几乎没有长尾。根本原因是 SelectiveEC 几乎每批都能完成所有的修复任务，所以推迟到下一批的任务较少。但对于 HDFS 来说，由于从 Namenode 无感知地分配任务，许多任务（远超过 8 个任务）可能被分配到同一个幸存节点上。该节点太多的任务之间造成了激烈竞争，从而导致越来越多的未完成任务被积累起来。所以在其他大多数的节点完成修复后，该节点的累积任务造成了长尾。

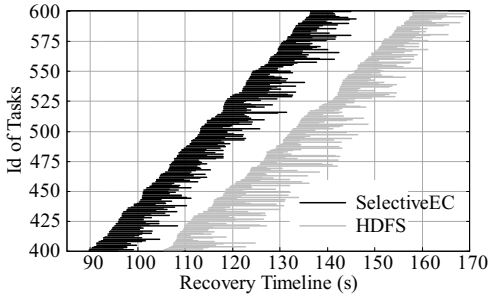


图 3.10　任务 ID 400 到 600 的运行时间.

为了更清楚的观察负载均衡对修复的影响，我们放大图 3.9 中的任务 ID 从 400 到 600 部分，并在图 3.10 展示。我们观察到与 HDFS 相比，SelectiveEC 的任务运行时间线更粗，这意味着更多的任务并行执行、毛刺更少，进一步说明了很少的任务没有及时完成。如前面分析，HDFS 中每批未完成的任务远比 SelectiveEC 多，未完成的任务将与下一批任务共享以及竞争资源，导致该节点的任务修复时间更长。SelectiveEC 几乎可以理想地完成每批次所有任务，从而提高任务的并行度，缓解数据修复过程中的资源竞争。但是 HDFS 中未完成的任务则逐批累积，导致资源的竞争越来越严重。

综上所述，SelectiveEC 从修复任务队列中逐批重组任务，使源节点间的数据分布均衡，进而动态调度源节点和替代节点。因此，它在网络、CPU、内存和磁盘 I/O 上实现了更好的负载均衡，从而可以更有效地利用系统有限资源。

3. 单一优化技术性能

为了更好地理解 SelectiveEC 的优势，我们实验分析了其各个优化环节对性能优势的贡献，实验结果参见图 3.11。这组实验中，修复带宽设置为 30MB/s。我们将 SelectiveEC 模块拆分为仅源节点调度（+ S）和仅替代节点调度（+ R）。与 HDFS 相比，+S、+R 和 SelectiveEC 的修复吞吐率分别增加了 7.15%、12.81% 和 26.31%。由于更均衡的上行和下行修复流量，+S、+R 都比 HDFS 的修复性能好。一方面，+R 显示出比 +S 更高的修复吞吐率，因为均衡的替代节点调度不仅使下行流量均衡，同时也使解码块写入本地磁盘的磁盘 I/O 和用于解码的 CPU/内存使用更加均衡。另一方面，SelectiveEC 提高的性能大于 +S 和 +R 的总和。原因是网络连接的传输速度受限于上下行带宽的最小值。所以 SelectiveEC 不仅受益于单独的均衡源节点和替代节点调度，而且受益于它们的组合，体现出 “1 + 1 > 2” 的特性。另外，图中的虚线是基于网络传输的理论计算出的修复吞吐率上限，由 $\frac{nB}{time_{slot}} = \frac{nBw}{k} = \frac{16\times30}{6} = 80$ 计算得出。然而，网络传输占用了大约 92% 的修复时间（见表 3.1），因此理想的修复吞吐率约为 $80 \times 0.92 = 73.6$MB/s。图 3.8(a) 显示，SelectiveEC 实现了 67.6MB/s 的吞吐率，即最佳吞吐率的 91.85%，说明 SelectiveEC 有很好的修复性能。

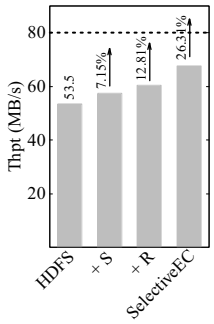


图 3.11　单一优化技术的修复性能

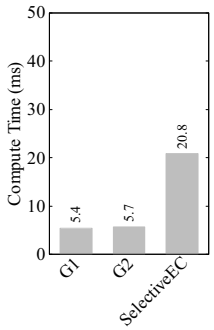


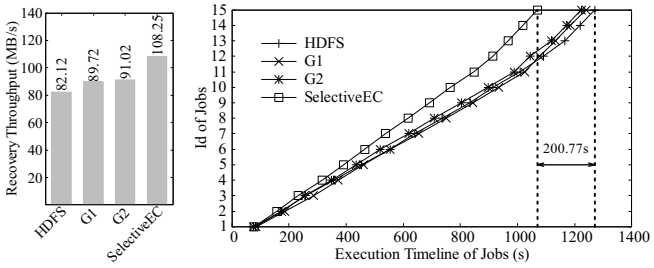
图 3.12　调度模块的计算开销

4. 修复调度开销

我们还实验评估了 ECManager 调度一个批次的用时。图 3.12 显示了 100 个批次的调度算法的平均运行时间，SelectiveEC、Greedy1 和 Greedy2 分别为 20.8 毫秒、5.4 毫秒和 5.7 毫秒。尽管 SelectiveEC 的调度时间明显长于其他两个算法，但与实现中的批次间间隔时间（3.2 秒）相比，可以忽略不计。因此，可以通过流水线方式隐藏在上一批的修复时间中。

5. 异构环境

为了评估调度模块在异构环境中的性能，我们在 HDFS 前台运行了一个具有代表性的 MapReduce 应用程序: TestDFSIO [97]。它是分布式 I/O 基准测试，旨在对存储 I/O 进行压力测试。TestDFSIO 是网络密集型的，因此它会在网络带宽不均衡的情况下性能显著下降。我们首先启动单个故障节点的修复任务，由 RS(6,3) 编码的 100GB 数据（大约 100 个批处理任务）；与此同时，在修复窗口中，我们运行 TestDFSIO 基准测试，它连续执行 15 个作业任务。每个作业从不同的客户端写入由 RS(3,2) 编码的 3GB 数据。故障修复和 TestDFSIO 都会竞争网络带宽。我们配置每个节点 1Gbps 的网络带宽，使用 Linux 命令 `ifstat` 采集网卡的实时占用信息，通过心跳反馈给 NameNode。因此 SelectiveEC 可以在异构环境中根据收集到的网络信息进行实时调度（详见第 3.4.2 节）。在图 3.13 中展示了故障修复性能和前台 TestDFSIO 作业性能。



(a) Recovery　　(b) Execution time of TestDFSIO jobs.

图 3.13　异构环境的修复性能与前台应用性能

故障修复性能：在图 3.13(a) 中，与 HDFS、G1 和 G2 相比，SelectiveEC 将修复吞吐率分别提高了 31.82%、20.65% 和 18.93%。主要原因是 SelectiveEC 根据动态的网络占用信息实现了动态地实时调度，保证了较高的修复性能。

前台 TestDFSIO 作业性能：在图 3.13(b) 中，SelectiveEC 下的作业执行时间总是比其他方法短。SelectiveEC 与 HDFS 相比，15 个作业的总执行时间减少了 200.77s，与 HDFS、G1 和 G2 相比，TestDFSIO 的平均吞吐率分别增加了 23.01%、15.30% 和 13.27%。主要原因是 SelectiveEC 可以实现实时动态地调度，避免在网络负载高的节点执行修复任务，从而减少了对 TestDFSIO 作业的干扰。

综上所述，SelectiveEC 在异构环境中也能高效地运行，提供了高修复吞吐率和对前台应用程序的低干扰。

3.6.3　多节点故障修复调度性能评估

我们运行实验来评估多节点故障的修复性能，在 18 个节点集群中部署 RS(6,3) 码，然后随机宕机 1-3 个节点。以双节点故障为例。根据我们的统计，当两个节点宕机时，有两个丢失块的条带数与丢失一个块的条带数比例为 45% : 55%。

如第 3.4.3 节描述，SelectiveEC 分别为两种类型的条带调度修复任务，以最大化调度步骤 (2) 中的带宽利用率。虽然 SelectiveEC 在步骤 (1) 中实现了负载均衡，但在步骤 (2) 中只有部分主替代节点需要将被修复块分发给辅助替代节点，导致将被修复块传输到辅助替代节点的带宽利用率不均衡。上述分析也适用于三节点故障。如图 3.14 所示，与 HDFS 相比，SelectiveEC 对于单节点、双节点和三节点故障的修复吞吐率分别提高了 26.31%、19.21% 和 18.86%。

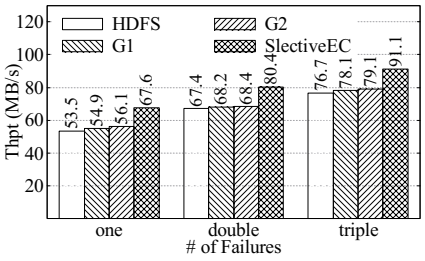


图 3.14　多节点故障修复性能

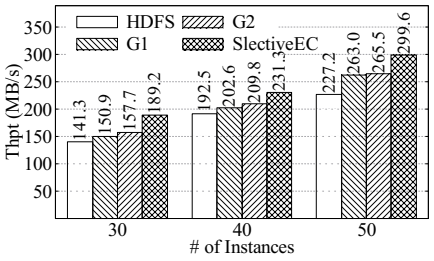


图 3.15　AWS 故障修复性能

3.6.4　Amazon EC2 中的修复调度性能评估

我们使用 n 个虚拟实例，评估 SelectiveEC 在 Amazon EC2 中的同构单节点故障修复。我们将 HDFS 3 配置为 1 个 NameNode 和 n-1 个 DataNode，其中 n = 30、40 和 50，并使用第 3.6.1 节的默认配置。图 3.15 展示了不同实例数量的修复性能。SelectiveEC 的修复性能在集群扩展时可以很好地扩展。与本地集群实验性能类似，SelectiveEC 相较于 HDFS 平均提高了 28.65% 的修复吞吐率。

3.6.5　大规模存储系统的模拟调度性能评估

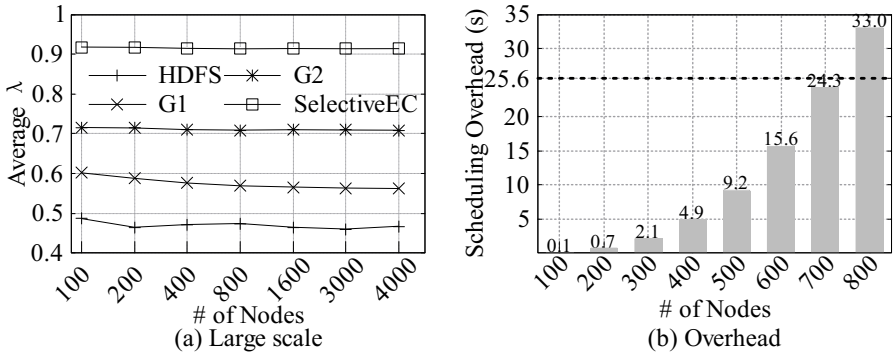


图 3.16　大规模集群下的修复性能

最后，我们通过模拟来估计 λ，以此证明 SelectiveEC 在大规模分布式存储系统中的优势。图 3.16(a) 比较了 SelectiveEC 和其他算法的 λ，其中节点数从 100 增加到 4000。我们发现 SelectiveEC 的 λ 始终大于 0.91，在某些情况下甚至达到 1，而 HDFS 的 λ 约为 0.5。与 HDFS 相比，SelectiveEC 的 λ 平均提高了 95.19%。此外，HDFS 的 λ 值很稳定，原因是 HDFS 中一批条带中数据块/校验块分布的均匀程度基本上由纠删码的参数 k, m 决定，与幸存节点的数量无关（见第 3.2.2

节）；在分布式存储系统中，由于源节点和替代节点的选择很难在少量的条带内实现负载均衡，因此在我们的实验中，HDFS 的 λ 保持在较低水平。

为了研究集群大小 n 对调度开销的影响，我们在图 3.16(b) 中展示了不同规模的分布式存储系统中调度算法的时间成本。批次之间的时间间隔是 $\frac{kB}{B_w} = \frac{6*128}{30} = 25.6$ 秒（采用谷歌默认的 RS(6,3) 码，HDFS 默认的 128MB 块大小，以及盘古默认的修复带宽 30MB/s），表示为图 3.16(b) 中的虚线。我们观察到，对于具有 $n \leqslant 700$ 个节点的分布式存储系统，调度时间始终保持在该虚线以下，因此可以通过流水线将调度开销隐藏于上一个批次的修复时间。此外，对于集群大小超过 700 的分布式存储系统，我们可以通过第 3.4.1 节讨论的方法来减少算法开销。根据我们的实验和讨论结果，SelectiveEC 的调度开销可以通过流水线方式完美隐藏在修复过程中。

3.7　本章总结

在基于纠删码的存储系统中，传统方法将数据块与校验块随机分布于各个节点中，而且在故障修复过程中随机选择源节点与替代节点，这对于大规模的存储系统来说，可以达到数据块分布均匀，负载均衡。然而，在故障修复的过程中，丢失数据块是分批次修复的。在一个批次内，数据块数量有限，随机分布造成数据分布与负载严重不均衡，拖慢了故障修复速度。本章提出了二分图模型，基于该模型，动态选择待修复的故障数据，将其打包到一个批次，然后在一个批次内，为每个待修复的故障数据，确定性地选择源节点与替代节点，在各个节点间达到了修复负载的均衡，加快了修复速度。实验结果表明，故障修复吞吐率明显优于已有方法。

针对本章介绍的 SelectiveEC 算法，做如下总结：

(1) 针对数据块/校验块随机分布的纠删码存储系统，SelectiveEC 首次提出数学模型，刻画故障数据修复过程中，一个批次内待修复数据的选择，以及源节点和替代节点的选择问题，从而达到了故障数据修复的负载均衡，加快故障数据修复速度。

(2) SelectiveEC 不仅适用于同构的分布式存储系统，而且适用于异构以及负载动态变化的分布式存储系统。这只要在相关的二分图模型中，加入边权函数与顶点权函数，就可以刻画系统的异构特性。另一方面，SelectiveEC 的设计思路也可以应用于多节点故障的修复问题。

第 4 章　基于分离内存系统架构的纠删码流程设计

4.1　引言

由于高速网络的发展, 例如远程内存直接访问（RDMA）技术 [12-13], 网络传输的带宽和时延都变得接近内存访问性能。例如，200 Gbps Mellanox ConnectX-6 IB RNIC（RDMA NIC）的吞吐率非常接近 DDR3-1600 DRAM 的 4 通道的本地内存访问 [9]。因此，设计具有微秒级访问时延并且可弹性扩展的大容量内存池成为可能，另外持久化内存 [10-11] 的商用也在存储成本上保证了可行性。为了实现这一点，分离内存架构应运而生，它通过解耦计算和内存来提高资源利用率。计算池包含一个内存非常有限的计算服务器集群，而内存池通过使用 DRAM 或持久化内存提供大的共享内存空间，另外内存池通常配置有低计算能力，仅仅服务于网络连接的请求。共享内存池提高了内存利用率，降低了内存成本，而且还很容易横向扩展系统来构建一个大容量的内存池。因此，分离内存系统吸引了学术界 [67-68, 77] 和工业界 [13, 79, 98-102] 的广泛关注。

分离内存系统通常配置多台内存服务器来提供大的统一内存空间，但是规模大也使得内存池容易出现故障。任何组件故障都可能停止在共享该组件的计算服务器上运行的用户服务。因此，数据容错 [7, 26] 应该是内存池中构建可靠分布式系统的基础。然而，尽管现有的工作花费了大量精力来解决分离内存系统的各种挑战，包括内存管理 [9, 64-69], 网络优化 [13, 70-74] 和架构设计 [61, 76] 等，但是高效的容错管理仍然是大家很少考虑的问题。

在传统基于磁盘的系统中广泛使用多副本策略来提供简单的容错，但它带来了高存储成本。在分离内存系统下，DRAM 和持久化内存都比传统磁盘更昂贵。由于写入冗余的数据副本，它还引入了额外的写入时延。纠删码提供了一种替代方案，它提供了相同级别的容错能力并且更小的存储开销。但是，当在分离内存系统中部署纠删码时，由于单边 RDMA 微秒级的低时延和编解码的高计算成本，我们面临着新的挑战和瓶颈。具体来说，由于内存服务器的计算能力低，数据应该在计算池中编解码，然后传输到内存池。因此，编解码时延应与单边 RDMA 等远程内存访问的时延相匹配。

然而, 根据我们的测试（参见第 4.2 节), 即使采用最快的纠删码库 Intel ISA-L [103]，(4, 2) 码的编码时间（52μs-819μs 用于对 16KB-1MB 的对象进行编码）超过单边 RDMA 时延（6μs-176μs 用于对 16KB-1MB 的对象网络传输）的 4 倍。所以直接将传统的编解码库部署在分离内存中会大大降低系统性能。然而，据我们所知，之前的工作没有仔细的分析分离内存系统中的编解码过程，以使其与低时延 RDMA 传输协作运行的很好。特别是，现有的加速编解码优化主要集中在减

少 I/O 开销或均衡工作负载以减少跨机架网络流量，因为它们主要针对基于磁盘的系统，如分布式存储系统 [22, 24]。对于分离内存系统中的纠删码，需要编解码的数据已经在内存中，编解码时延主要是数据从内存加载到 CPU 缓存和编解码函数栈的开销。因此，用于加速编解码过程的现有方法不适用于分离内存系统。

我们进一步分析编解码函数栈和 RDMA 传输过程。将纠删码部署在分离的内存中，我们发现有多种因素导致高写入/读取时延，它们在编解码和网络传输过程中提供了优化机会。首先，由于数据已经在计算服务器的内存中，通过精心设计的数据条带化，使其与 CPU 上 L1 cache 对齐，可以大大加快编解码速度。其次，通过重用缓存中第一个条带的辅助编解码数据，例如编解码矩阵和乘法表，重新设计编解码函数栈，也可以大大减少后续条带的编码时间。最后，我们设计了非阻塞流水线进行编解码和 RDMA 传输，并仔细调整编解码和网络传输对象的大小，使得流水线达到最大化的并行度。

基于上述分析，我们设计了一种新的纠删码方案 MicroEC，它优化了纠删码部署在分离内存系统的编解码和 RDMA 传输。具体来说，我们通过利用 L1 cache 访问局部性重新设计编解码函数栈以最小化编解码时延，并仔细协调编解码计算和 RDMA 传输以实现高效流水线。通过仔细实现这些技术和多项优化，MicroEC 大大减少了编解码时延并使其与单边 RDMA 相匹配，并且它还实现了比现有纠删码和副本方案更低的写入时延。综上所述，本章有以下的贡献点：

- 我们从 L1 cache 效率的角度，以及 RDMA 传输的工作流程，彻底分析了编解码函数栈。我们发现了在分离内存系统中部署纠删码时的关键性能瓶颈，并提供了三个关键系统发现来指导优化编解码和 RDMA 传输工作流程。
- 我们通过利用 L1 cache 对齐的条带化和重用辅助编解码数据来加速编解码计算，并且重新设计了编解码函数栈。我们还提出了有效的数据结构来支持新的设计。此外，我们协调使用非阻塞流水线进行编解码和 RDMA 传输，并仔细调整编解码和传输对象的大小使得流水线达到最大化的并行度。
- 我们在 Apache Crail [104] 之上实现了 MicroEC 的原型系统，并进行了各种实现优化。大量实验表明，与最先进的现有纠删码和三副本技术相比，MicroEC 分别实现了高达 2.08 倍和 1.74 倍的写入吞吐率。此外，与没有容错的系统相比，MicroEC 仅将写入吞吐率降低了 10%（如果包含 MicroEC 中的校验块写入）。因此，MicroEC 可以在具有高性能和低内存成本的分离内存系统中实现高效的纠删码。

本章其余部分组织如下：我们首先在第 4.2 节对分离内存系统架构下纠删码性能瓶颈问题进行了描述以及分析；在第 4.3 节介绍了基于分离内存系统架构的纠删码流程 MicroEC 的设计；在第 4.4 节和第 4.5 节分别介绍了 MicroEC 的系统实现细节以及性能评估；最后，在第 4.6 节对本章进行了总结。

4.2　分离内存系统架构下纠删码性能瓶颈问题分析

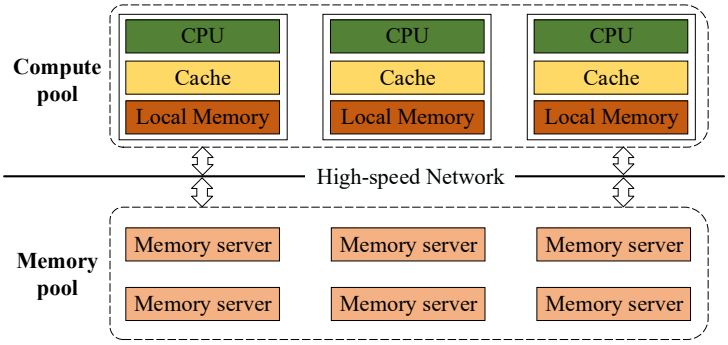


图 4.1　分离内存存储系统架构图

分离内存系统的容错：图 4.1 展示了典型的分离内存架构，计算资源和内存资源被分成*计算池*和*内存池*。由于内存池中计算能力有限，计算服务器负责计算任务和用户服务，同时由于本地内存有限，还必须访问内存服务器的数据。为了在内存池中不涉及 CPU 的情况下提供低访问延迟，计算服务器和内存服务器通过高速网络连接，通常使用远程访问技术 [12-13]，例如远程直接内存访问（RDMA）或 Gen-Z。本章重点关注广泛使用的单边 RDMA，包括 `WRITE` 和 `READ` 原语提供与本地内存访问相似的吞吐率和微秒（μs）级延迟 [9]。在使用纠删码的分离内存中，当我们写入一个对象时，我们通过计算服务器对其进行编码，然后通过 RDMA `WRITE` 将其写入内存服务器。并且在发生故障时需要修复时，我们通过 RDMA `READ` 从内存服务器读取相同条带的块到计算服务器，并通过解码修复故障的块。因此，为了在分离内存中高效部署纠删码，我们应该使编解码延迟与 RDMA `WRITE`/`READ` 的延迟相匹配。虽然最近有一些关于纠删码提供内存容错的工作 [105-109]，但是他们主要关注纠删码的元数据管理或弹性容错。正如我们接下来的实验所示，现有方法仍未提供与内存访问相匹配的低延迟性能。

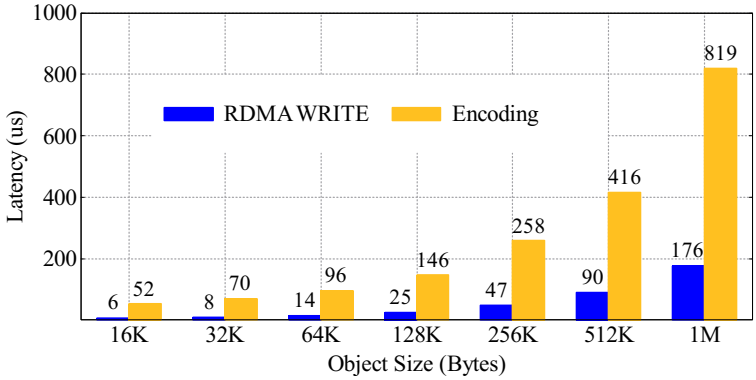


图 4.2　RDMA `WRITE` 和编码的时延

编解码计算与单边 RDMA 的速度：一般分离内存系统部署纠删码时需要为用户提供四种基本操作，即：写（*write*）、正常读（*read*）、降级读（*degraded read*）和故障修复（*recovery*），这些操作应该建立在单边的 RDMA `WRITE` 和

READ 之上。由于降级读和修复的工作流程与写工作流程相似，因此我们以写（*write*）为例分析性能瓶颈。我们在本地 18 节点的集群上测试了编码时延与端对端 RDMA WRITE 的时延，其中每个节点配置两个 Xeon(R) E5-2650 CPU 和 64GB DRAM，并与其他节点通过 56Gbps RDMA 网络互联。我们使用业界广泛使用 $(4,2)$ 码 [7, 35, 109] 为例，并采用业界最流行的纠删码编解码库 ISA-L [103]。RDMA WRITE 性能采用 ib_read_lat 命令测试端对端的时延。另外，需要编码的对象已经提前存储在计算服务器的内存中。图 4.2 展示了实验结果，我们发现编码时延比 RDMA WRITE 时延高的多。比如对于 16KB 和 32KB 的对象，编码时延分别为 RDMA WRITE 时延的 8.67 倍和 8.75 倍，而对于 1MB 的大对象，仍然达到 4.65 倍。因此，编码和 RDMA WRITE 之间存在巨大的时延差距，简单地将纠删码部署在分离内存系统中会导致低效的性能。虽然我们可以使用多线程来加速编解码，但它会消耗更多的 CPU 算力。另外，多线程对单个对象进行编解码也会造成高额的同步开销；多线程并行编解码不同对象，共享同一个网络进行传输也可以提高带宽利用率，但由于数据传输的竞争，也会严重延长了端到端的对象写入时延，我们在第 4.5.5 节进一步实验测试与分析了多线程的性能。所以，新的编解码计算瓶颈促使我们重新设计编解码函数栈，从而在分离内存系统中实现高效的纠删码部署。

纠删码部署在分离内存系统中的写流程拆解分析：我们首先探究了纠删码工作流程的编解码函数栈，并通过大量实验发现并提出了关于编解码计算和 RDMA 传输的三个关键的系统发现。要在客户端使用 (k,m) 码写一个对象，如图 4.3 所示，工作流程需要包括三个步骤：*split*、*encode* 和 RDMA WRITE。首先，在 *split* 步骤中，对象在逻辑上被拆分为 k 个数据块，这个过程开销很低，可以忽略不计。接下来，在 *encode* 步骤中，计算服务器基于 k 个数据块来执行编码计算得到 m 个校验块。最后，在 RDMA WRITE 步骤中，计算服务器将 $k+m$ 个数据块和校验块写入内存池端。我们进一步分析编码的函数栈，如图 4.3 所示，它由四个阶段组成：（I）*allocating buffers*：用于存储辅助系数和校验块，（II）*initializing the coding matrix*：用于初始化生成矩阵，（III）*generating MUL tables*：用于生成乘法表（一些编码算法使用位矩阵 [110-112] 而不是 MUL 表），和 (IV) *encoding data*：执行编码计算得到校验块。下文将介绍我们对加速纠删码编解码计算和 RDMA WRITE 的主要发现，并分析利用这些系统发现得到的机会以及所面临的挑战。

4.2.1　编解码函数栈不高效

我们关注编解码瓶颈并进行了大量的实验测试。我们使用相同的纠删码参数（相同的编码矩阵与乘法表等）对两个大小相同的对象单线程进行了连续编码，测试的对象从 16KB 到 1MB 不等。我们将两个对象的编码任务绑在同一个

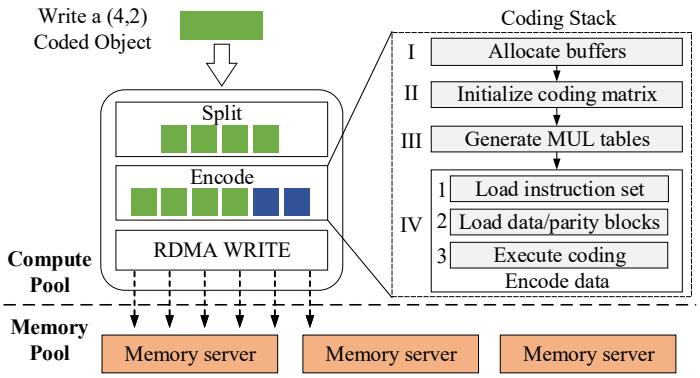


图 4.3　纠删码部署在分离内存系统中的写流程

CPU 核上来研究编码函数栈的效率。编码函数栈四个阶段（见图 4.3）的时延如表 4.1 所示，其中 “O1” 和 “O2” 分别表示第一个和第二个编码对象。

表 4.1　单线程连续编码两个对象的各阶段时延图 (μs)

-	第 I 阶段		第 II 阶段		第 III 阶段		第 IV 阶段		总共时间	
对象大小	O1	O2	O1	O2	O1	O2	O1	O2	O1	O2
16KB	20	20	7.5	0.4	2.1	0.4	23	4.5	52	25
32KB	22	22	6.3	0.4	1.9	0.3	40	18	70	41
64KB	21	20	6.8	0.4	2.1	0.4	66	45	96	66
128KB	20	20	9.8	0.4	2.2	0.3	113	83	146	105
256KB	20	23	7.6	0.4	2.2	0.4	229	201	258	225
512KB	25	20	6.9	0.5	2.1	0.4	382	381	416	402
1MB	24	25	7.2	0.6	2.0	0.4	785	737	819	762

我们首先发现在阶段 II-III 中对第二个对象进行编码的时延显著降低。比如编码第一个 16KB 对象的总时延为 52μs，是编码第二个对象的 2.08 倍。原因是编码线程在对第一个对象进行编码时：第 I 阶段为校验块和生成矩阵等参数分配内存空间；第 II 阶段初始化编码矩阵；在第 III 阶段生成 MUL 表。但是，在对第二个对象进行编码时，可以重用阶段 I-III 加载到缓存中的辅助数据，因此大大减少了其编码时间。表 4.2 中统计了编码函数栈的 L1 cache 未命中次数也进一步验证了这一结论。

表 4.2　编码过程的 L1 cache 未命中次数统计

第 II 阶段		第 III 阶段		第 IV 阶段	
O1	O2	O1	O2	O1	O2
186	10	106	0	985	700
180	0	111	8	1556	1342
186	2	128	1	2589	2401
190	9	132	2	4731	4503
158	0	166	0	9027	8831
161	0	146	5	17253	17215
186	0	110	0	34236	33840

我们接下来发现第二个对象在第 IV 阶段的编码计算时延也显著降低，尤其是对于 16KB 这样的小对象。第 IV 阶段进一步包括三个子阶段：(IV-1) *Load instruction set*：用于加载指令集，(IV-2) *Load data/parity blocks*：用于加载数据/校

验块，和 (IV-3) *Execute coding*：用于执行编码计算。因为对第一个对象编码后，相关的指令集仍在 L1 指令集缓存中，可以重复的用于对下一个对象的编码过程。

系统发现 1: 如果我们绕过阶段 I-III 并重用阶段 IV-1 的指令集，那么编码时间将大大减少，例如编码 16KB 对象的时间可以达到大约 4.5μs，实现了与 RDMA `WRITE` 相当的时延。

关键挑战 1：需要重新设计编解码计算函数栈，从而保证 L1 cache 中用于编解码的辅助数据可以被重用。

4.2.2　缓存不高效

通过观察表 4.1，阶段 IV 占据主要的编码时间，尤其是对于大对象。具体来说，用 16KB 对 O2 进行编码只需要 4.5μs，而对 32KB 的对象进行编码的时间增加到 18μs，增加到了 4 倍。此外对于大对象，阶段 IV 的时间几乎随对象大小呈线性增加，对于 1MB 的对象，它达到了 737μs。为了进一步理解上述结果，我们通过统计 `L1-dcache-stores` [①]展示了第 IV 阶段 L1 cache store 的数量。如表 4.3 所示，我们发现当对象大小从 16KB 增加到 32KB 时，L1 cache store 的数量从 270 增加到 1568（5.81 倍），并且当对象大小翻倍时，它会以 2.45-2.67 倍的速度持续增加。

表 4.3　编码第二个对象时关于阶段 IV 的时间开销以及 L1 cache stores 次数

对象大小	16KB	32KB	64KB	128KB
第 IV 阶段时延（μs）	**4.5**	**18**	45	83
L1 cache stores 的次数	**270**	**1568**	4181	10259

在第 IV 阶段为较大对象编码造成大量 L1 cache store 的次数，主要原因如下。在我们的测试平台中，L1 cache 大小为 32KB。而对于 16KB 的对象，所有校验块、辅助编码数据和数据对象都可以完全加载到 L1 cache 中。然而，L1 cache 不能为大对象保留所有这些数据。所以当我们对不小于 32KB 的对象进行编码时，校验块不断的被逐个加载进缓存的数据块更新，而 LRU 缓存替换策略 [114] 将会逐出上一次生成的校验块，从而造成大量的 L1 cache store。总之，由于频繁的 L1 cache store，对大对象进行编码会产生非常大的开销。因此，如表 4.1 所示，在第 IV 阶段编码一个 1MB 对象的时间可以长达 785μs，但是如果我们将 1MB 对象拆成 64 个 16KB 的子对象，那么第 IV 阶段的编码时间理论上仅为 $23 + 63 \times 4.5 = 306.5\mu s$。

系统发现 2：如果我们将大对象拆分为多个基于 L1 cache 对齐的最优大小子对象，例如我们的测试平台中的 16KB，那么我们可以大大提高缓存效率从而达到最大化编码速度。这促使我们设计缓存对齐的条带化方案来优化缓存效率。

关键挑战 2：需要设计有效的地址管理来支持对象的拆分，并且需要避免管

[①]这是 Linux 中 perf [113] 自带的硬件缓存事件。

理元数据以及小对象的读/写操作的高额开销。

4.2.3　流水线不高效

上述分析意味着应该将大对象拆分为小对象来优化编码速度。但是，拆分会引入许多子对象而带来小的网络传输包，这不可避免地会增加 RDMA 网络传输的趟数，从而增加传输时延。因此，在最大化编码速度和最小化 RDMA `WRITE` 时延的最佳对象大小之间存在着性能冲突。

简单的为 RDMA `WRITE` 打包尽可能多的小对象带来的效率是低下的。原因是随着对象大小的增加，RDMA `WRITE` 的时延减少趋势会增加。例如，如图 4.2 所示，当我们将对象大小从 16KB 翻倍到 32KB 时，RDMA `WRITE` 的时延从 $6\mu s$ 增加到 $8\mu s$，只有 1.33 倍。从 128KB 到 1MB，当对象大小翻倍时，时延增加了近 2 倍。此外，为 RDMA `WRITE` 打包一个太大的数据包会阻碍流水线效率，因为它会导致在启动 RDMA `WRITE` 之前等待编码数据的时间很长。例如，编码 1MB 对象的最佳时延预计为 $306.5\mu s$，而 1MB 对象的 RDMA `WRITE` 时延为 $176\mu s$（见图 4.2），所以客户端写入一个 1MB 的对象在没有流水线的解决方案下总共需要 $482.5\mu s$。然而，由于优化编码的时延已经与 RDMA `WRITE` 时延相当，如果我们仔细协调编码和传输操作，端到端的写入时延可以大大降低。

系统发现 3: 通过先优化编解码计算瓶颈，可以进一步的对编解码和网络传输进行流水线化。这促使我们设计高效的流水线来最小化纠删码写入的总时延。

关键挑战 3: 需要仔细协同优化编解码计算和 RDMA 传输，动态调整两个操作之间不匹配的对象大小，从而来实现最大的流水线并行度。

除了上面分析的写入过程之外，我们还需要在分离内存系统中部署纠删码时有效地支持正常读取、降级读取和故障修复。这也需要在解码和 RDMA `READ` 上实现高效的流水线，并且避免产生大量额外的元数据维护开销。简而言之，优化的纠删码设计和实现应该与通用的分离内存系统的架构相互兼容。

4.3　一种基于分离内存系统架构的纠删码流程设计

我们设计了一个纠删码方案 MicroEC, 它通过协同优化编解码计算和 RDMA 网络传输流程, 从而在分离内存中实现高效的纠删码部署。图 4.4 展示了 MicroEC 的整体架构，主要由*编解码流程*（*coding*）、*地址管理*（*address management*）和*协调器*（*coordinator*）三个模块组成。首先，针对编解码计算的瓶颈，*编解码流程*模块通过对解编码对象基于 L1 cache 对齐的条带化优化以及对编解码函数栈优化，从而有效的提升了编解码效率。然后，为了避免编解码条带化拆包所带来的高额元数据管理开销，*地址管理*模块通过设计高效的数据结构来逻辑上拆包，

从而高效地访存对象。最后，为了有效地保证编解码以及网络传输的友好，协调器模块协调编解码和 RDMA 传输操作，从而实现了高效流水线。我们将在下面的章节中详细的介绍这三个模块内容。

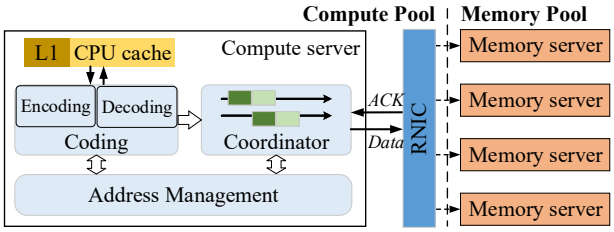


图 4.4　MicroEC 的整体架构

4.3.1　缓存友好的编解码流程

缓存友好的编解码过程：我们首先从 L1 cache 的角度介绍编码过程。假设我们使用编码矩阵为 $M = (m_{ij})_{k\times m}$ 的 (k, m) 码来编码对象。首先对象被分成 k 个数据块，比如 $B_0, B_1, ..., B_{k-1}$，然后编码成 m 个校验块 $P_0, P_1, ..., P_{m-1}$。我们以最先进的编码库 Intel ISA-L [103] 库为例来详细介绍编码过程。编码过程中，*辅助数据*（包括编码矩阵、乘法表和 SSE 向量化指令集）首先被加载到 L1 cache 中，接着数据块：$B_0, B_1, ..., B_{k-1}$ 被逐个地加载到 L1 cache 中进行编码。因为 L1 cache 空间有限，一旦 B_i $(0 \leqslant i \leqslant k-1)$ 被加载到 L1 cache 中，$m_{ij} \times B_i$ 的乘法就会被执行，并且所有的校验块 P_j's $(0 \leqslant j \leqslant m-1)$ 被更新为 $P_j \oplus (m_{ij} \times B_i)$。当同一对象的所有数据块都处理完后，编码过程就会终止，所有的校验块都会从 L1 cache 中被逐出到内存中。

最优的编码粒度：为了执行编码计算，*辅助数据*必须驻留在 L1 cache 中，剩余空间可用于保存数据块和校验块。如果剩余空间不足以容纳一个数据块和 m 个校验块，那么校验块只能在 L1 cache 中一一更新，这会导致校验块的频繁驱逐/加载。为避免频繁驱逐校验块，块大小 S_b 应满足以下不等式：[①]

$$(m+2) \times S_b + C < S_{L1}, \tag{4.1}$$

这里 S_{L1} 是 L1 cache 的大小, 通常是几十 KB 大小 [116], 在我们实验中是 32 KB; 这里 C 是*辅助数据*的大小, 通常是常量（不会变化很大）并且小于 L1 cache 的存储空间。例如，$(4, 2)$ 码的辅助数据大小约为 8 KB，而 $(10, 4)$ 码的辅助数据大小增加到 9KB 左右。辅助数据的常见大小是 8KB-10KB，因此对于 32KB L1 cache 仍有大约 22KB-24KB 的空间来保存数据块和校验块。因此，可以根据上述不等式计算最佳编码块大小，我们表示为 S_b^*。例如，对于 32KB 的 L1 cache，对于 $m = 2$ 到 4，S_b^* =4KB。

[①]Jerasure [115] 使用另外一种策略来计算校验块。首先它需要保持所有的数据块在缓存中，然后依次计算每一个校验块。通过简单的交换 m 和 k 的位置，不等式4.1仍然成立。

缓存友好的编码条带化 (Coding with cache aligned striping (CCAS))：编码的对象可能非常大，因此导致块的大小大于 S_b^*（即，对象大小大于 $k \times S_b^*$）。所以我们用 CCAS 重新设计编码，进一步将每个数据块划分成最优编码块大小为 S_b^* 的子块。

图 4.5 描述了 CCAS 的过程。具体来说，首先将一个对象 O 根据所使用的 (k, m) 码拆分成 k 个大小都为 S 的数据块 $B_0, B_1, ..., B_{k-1}$。目标是将 k 个数据块编码为 m 个校验块 $P_0, P_1, ..., P_{m-1}$，如果块大小 $S > S_b^*$，我们进一步拆分 B_i $(0 \leqslant i \leqslant k-1)$ 得到大小为 S_b^* 的子块 $B_{i0}, B_{i1}, ..., B_{i(n-1)}$。请注意，如果 S 不是 S_b^* 的倍数，我们可以简单地通过补零来处理这种情况。然后我们对子数据块 $B_{0j}, B_{1j}, ..., B_{(k-1)j}$ 进行编码以生成 m 个校验子块 $P_{0j}, P_{1j}, ..., P_{(m-1)j}$，从而形成一个条带 S_j。为了便于表示，我们将子块 $B_{0j}, B_{1j}, ..., B_{(k-1)j}$ 视为子对象 O_j 的 k 个数据块。一旦所有子对象 $O_0, O_1, ..., O_{n-1}$ 都被编码，我们将 $B_{i0}, B_{i1}, ..., B_{i(n-1)}$ 重新组合成 $B_i(0 \leqslant i \leqslant k-1)$，并将 $P_{i0}, P_{i1}, ..., P_{i(n-1)}$ 组合成 $P_i(0 \leqslant i \leqslant m-1)$，至此我们完成了从 $B_0, B_1, ..., B_{k-1}$ 编码得到 $P_0, P_1, ..., P_{m-1}$。

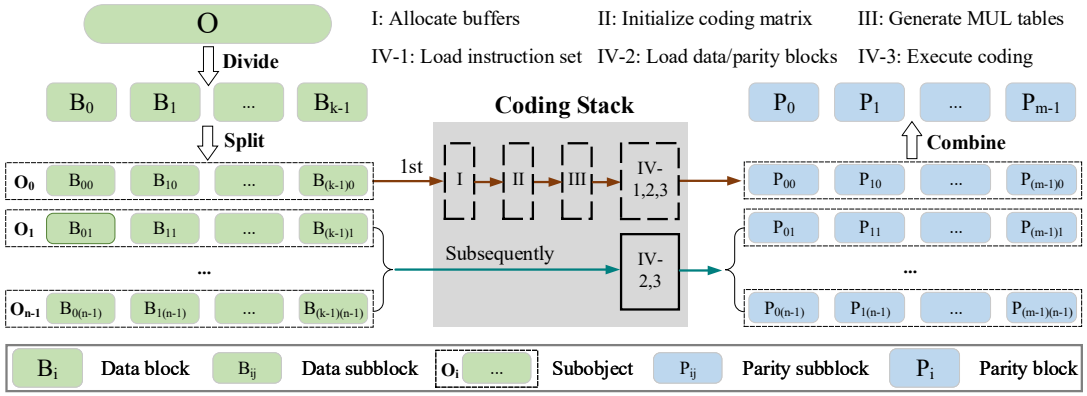


图 4.5　MicroEC 编码工作流程

上述编码设计有两个好处：1）子块的大小为 S_b^*，实现了对每个子对象的最快编码速度；2）后续子对象 $O_1, ..., O_{n-1}$ 的编码可以重用*辅助数据*并绕过阶段 I-III 和 IV-1（见图 4.5）。然而，要实现这两个好处，我们仍然面临以下挑战：(1) 如果我们在物理上将一个对象划分为子对象，并将每个子对象进一步拆分为子块进行编码，并在写入内存服务器时将子块重组为数据块和校验块，这样会导致高额的元数据管理开销和内存复制成本，最终减慢编码过程。我们通过设计数据结构来实现逻辑拆分，这对编码过程造成的延迟可以忽略不计（参见第 4.3.2 节）。(2) 传统纠删码方案中编码和 RDMA 传输是在同一个线程中执行的，因此网络传输经常阻塞下一个子对象的编码，这进一步影响了缓存的局部性，使得校验块缓冲区难以复用。因此，我们必须设置一个线程用于编码，另一个线程用于 RDMA 传输，并且还需要仔细协调两个线程以高效工作（参见第 4.3.3 节）。此外，我们还利用了校验块缓冲区共享，即我们仅在编码第一个子对象时为阶段 I

的校验子块分配缓冲区（*allocate buffers*），并将此缓冲区重用于后续子对象。

4.3.2　轻量级的地址管理

接着，为避免物理拆分对象带来的高额开销，我们设计了简单的数据结构来实现逻辑上的拆分。请注意，对于一个对象 O，它被划分成 k 个块，我们可以很容易地计算出划分块的大小 S。由于对象已经驻留在内存中，如果我们记录对象 O 在内存中的起始地址，记为 $\mathcal{A}_O$，那么我们可以很容易地计算出每个块和子块的起始地址，从而实现逻辑上的拆分。

如图 4.6 所示，对象 O 的数据结构记录了对象 ID、对象大小 S_O 及其在内存中的起始地址 $\mathcal{A}_O$。对于大小为 S $(0 \leqslant i \leqslant k-1)$ 的划分块 B_i，其数据结构由对象 ID、块 ID、块大小及其起始地址组成，起始地址计算为 $\mathcal{A}_O + i \times S$。由于数据块 B_i 被进一步分成 n 个子块 $B_{i0}, B_{i1}, ..., B_{i(n-1)}$，大小为 S_b^*，其中 $n = \lceil S/S_b^* \rceil$，$B_{ij}$ 的起始地址为 $\mathcal{A}_O + i \times S + j \times S_b^*$ 为 $0 \leqslant j \leqslant n-1$。子块的数据结构由对象 ID、块 ID、子块 ID 和子块大小组成。

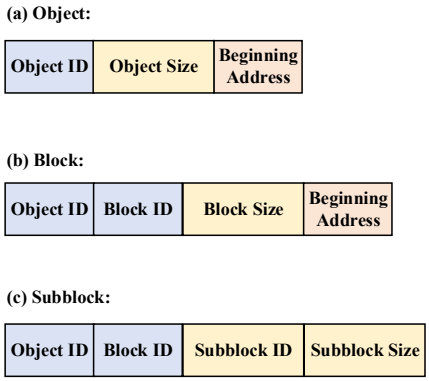


图 4.6　MicroEC 地址管理数据结构

当我们编码一个子对象 $O_j (0 \leqslant j \leqslant n-1)$ 时，我们可以根据它们的起始地址和子块大小读取 k 个子块 $B_{0j}, B_{1j}, ..., B_{(k-1)j}$，然后将它们编码成 m 个校验子块 $P_{0j}, P_{1j}, ..., P_{(m-1)j}$。根据维护的起始地址和子块大小，直接读取需要的部分数据，在逻辑上实现拆分，将校验块写入内存的过程也是类似的。我们首先在内存中为每个校验块 P_i 分配一个大小为 S 的缓冲区，通过使用由对象 ID、校验块 ID、校验块大小以及起始地址组成的数据结构。假设 P_i 的起始地址是 $\mathcal{A}_{P_i}$。当我们对子对象 O_j 进行编码后，我们可以从起始地址 $\mathcal{A}_{P_i} + j \times S_b^*$ 写入校验块 P_{ij}，长度为 S_b^*。类似地，校验子块的数据结构由对象 ID、校验块 ID 和校验子块 ID 组成。当所有校验子块 $P_{i0}, P_{i1}, ..., P_{i(n-1)}$ 都写入 P_i 的缓冲区时，则生成 P_i，无需物理上做子块合并。

支持缓存友好的编码条带化的额外开销只是上面介绍的数据结构。此外，子块的数据结构在完成每个对象的写入后被释放，因此开销可以忽略不计。

4.3.3　非阻塞的流水线

最后，我们需要对编码和 RDMA 传输进行有效的流水线化处理，以最大限度地减少写入延迟。我们也在第 4.5.4 节中的实验中详细表明，使用 CCAS 的编码速度可以与 RDMA 传输速度相当。

最优的包大小：为了达到最快的编码速度，CCAS 将一个对象划分成大小为 S_b^* 的子块，例如我们的测试平台中的 $(4,2)$ 码的 S_b^* 为 4KB。但是，S_b^* 并不是实现最快 RDMA 传输的最佳数据包大小。这是因为 S_b^* 由于 L1 cache 大小的通常很小，而用大数据包写入相同数量的数据会减少 RDMA 的传输趟数，从而减少传输延迟。从 RDMA 传输的角度来看，我们应该将更多的子块组合成一个相对较大的数据包以加快写入速度，但这可能会阻碍流水线效率，因为 RDMA WRITE 必须等到所有子块都被编码完成。因此，存在一个最优数据包大小 S_p^* 来最大化流水线效率，从而最小化端到端写入延迟。例如，在我们的 56Gbps RDMA 网络测试平台中，如果对象大小为 1MB，则异步 RDMA WRITE 的 S_p^*=16KB。我们还研究了网络数据包大小的敏感性实验性能（参见第 4.5 节中的图 4.12(b)）。

由于最优编码块大小为 S_b^*，而最优包大小为 S_p^*，我们需要在 RDMA 传输之前结合 S_p^*/S_b^* 个子块。我们指出，我们设计的数据结构也有效地支持了这种组合（参见第 4.3.2节）。例如，对于数据块 B_i，它的起始地址是 $\mathcal{A}_O + i \times S$，其中 $\mathcal{A}_O$ 是对象 O 的起始地址。从地址 $\mathcal{A}_O + i \times S$ 开始，我们每次读取一个长度为 S_p^* 的数据段，得到一个大小为 S_p^* 的包用于 RDMA 传输。即对于 B_i 的第 j 次 $(0 \leqslant j \leqslant \lceil S/S_p^* \rceil)$ RDMA 传输，我们从 B_i 中读取部分数据：起始地址为 $\mathcal{A}_O + i \times S + j \times S_p^*$ 以及结束地址为 $\mathcal{A}_O + i \times S + (j+1) \times S_p^* - 1$。校验块的写入过程也是类似的，在我们的数据结构支持下，为 RDMA WRITE 组合子块的开销可以忽略不计。

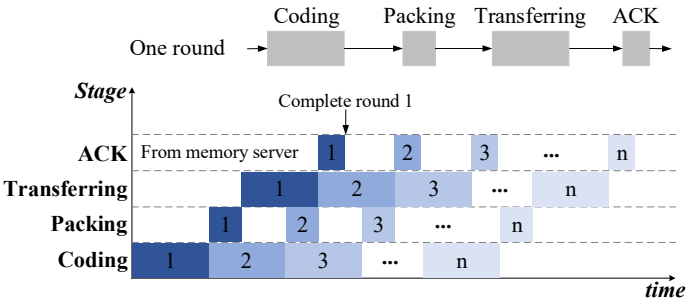


图 4.7　非阻塞的流水线设计概要图

异步线程：编码 S_b^* 的最优块大小和 RDMA 传输 S_p^* 的最优包大小可能不同，因此单线程的编码和网络传输无法实现高效流水线。更糟糕的是，在传统的流水线实现中使用单线程，例如 HDFS [24]，会在对每个子对象进行编码后清理相关缓冲区以便于内存管理，从而无法重用辅助数据。因此，我们采用两个异步线程，一个线程（编码线程）用于编码，另一个线程（打包线程）用于打

包编码子块以进行 RDMA 传输。为了协调这两个线程，我们定义了一个信号量 `Counter`，它由两个线程共享。一旦子对象被编码，编码线程将 `Counter` 加一。当 `Counter`= S_p^*/S_b^*，即 S_p^*/S_b^* 子对象编码完成时，打包线程将从每个数据（或校验）块中读取大小为 S_p^* 的数据，这些数据是已经刚编码好的子对象，并将其发送到网络缓冲区进行 RDMA 传输。与此同时，它将 `Counter` 重置为零。

非阻塞流水线：我们设计了一个非阻塞流水线来最小化客户端写入整个过程的延迟，它由四个阶段组成：*coding*、*packing*、*transferring* 和 *ACK*，如图 4.7 所示。具体来说，*coding* 表示异步编码多个子对象并实时更新相应的数据结构，*packing* 表示通过将多个子块逻辑组合成大数据包来辅助 RDMA `WRITE`，*transfer* 表示真正通过异步 RDMA `WRITE` 将数据写入内存服务器，最后 *ACK* 表示确认 RDMA `WRITE` 成功。一旦计算服务器收到来自所有内存服务器的 ACK，一轮数据写入成功完成。最后，我们仔细在实现上协调这四个阶段，使它们完美配合流水线流程，如图 4.7 所示。

4.4　系统实现

MicroEC 编解码库：基于英特尔 ISA-L [103] 基础库，我们使用大约 1.2K 行 C 代码从头开始实现编解码过程。基于第 4.3.2 节中提出的数据结构，我们可以从特定的内存区域读取一段并实现编解码。如图 4.8 所示，MicroEC 提供了两个简单的编解码接口 API：`encode()` 和 `decode()`。`encode()` 将数据块和编码参数（例如，k、m 和对象大小）作为其输入，并生成校验块作为其输出。对于 `decode()`，它需要额外的可用块 `indices` 信息来参与解码得到丢失的块。MicroEC 以共享库的形式提供其编解码功能，可以动态链接到客户端进程的地址空间并部署在分离内存系统中。

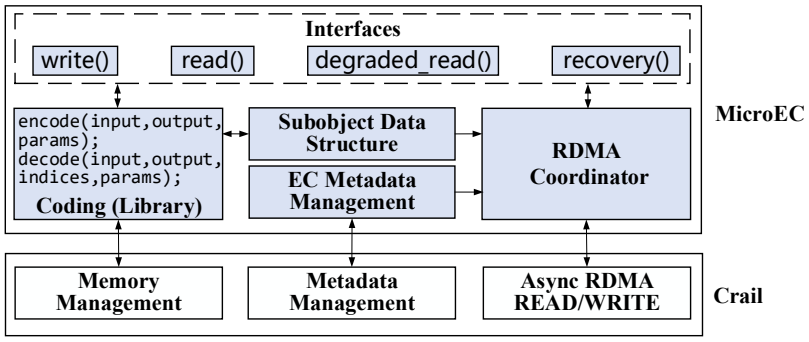


图 4.8　MircoEC 原型系统实现概要图（阴影框是 MircoEC 新引入的模块）

MicroEC 系统原型：我们通过在 Crail [104] 上修改以及添加大约 6K 行 JAVA 代码实现了 MicroEC，如图 4.8 所示。Crail 是一个内存存储系统，并提供统一的内存空间以及支持单边 RDMA 和内存池并发访问。MicroEC 基于 Crail 中的

metadata management 实现 *EC metadata management* 记录编解码信息,例如数据/校验块 ID 和编解码参数等。基于 Crail 中异步 RDMA `READ/WRITE`, 结合数据结构和元数据管理, 实现了 MicroEC 中的 coordinator。要写入格式为 `CrailBuffer` 的对象，客户端首先调用 `CrailStore.create` 在命名空间中创建对象元数据并获取 Crail `file`。然后，它以 `CrailBuffer` 作为输入来调用编解码 API, 并执行 MicroEC 的编解码过程。我们利用 `slice()` 从 `CrailBuffer` 中读取子块，并将它们放入 `file` 的文件流中，后者通过异步 RDMA `WRITE` 将子块写入内存池并返回 `future` 作为确认完成 RDMA `WRITE`。

MicroEC 接口： MicroEC 支持四种通用接口，`write`、`read`、`degraded read` 和 `recovery`，客户端程序可以通过以下方式使用它们。`CrailStore.create` 创建一个带有编解码策略的文件，`file.getMircoECIOStream` 创建一个隧道来发送/接收为 RDMA `READ/WRITE` 准备的编解码文件的数据包。

```
file=CrailStore.create(filename,policy);
s=file.getMircoECIOStream(filesize);
s.write/read/degraded_read/recovery(buf);
```

系统优化：我们在实现上做了以下两个优化：(1) 对于降级读/修复，当客户端开始为 RDMA `READ` 运行网络线程时，我们立即启动一个解码线程来执行编解码函数栈中的阶段 I 到 IV-1 函数栈。因此，解码第一个子对象的时间开销可以隐藏在第一轮网络传输中。(2) 一轮编码完成后，打包线程会在计算服务器的本地内存缓冲区中准备 $k+m$ 个块，等待写入内存池。一个简单的解决方案可能会执行多次 `memcpy()` 以从原始数据缓冲区复制数据以进行 RDMA 传输。为了消除 `memcpy()` 开销，我们利用 `ByteBuffer slice()` 来获取共享数据缓冲区中的数据并逻辑打包小数据包来进行 RDMA 传输。

4.5 性能评估

4.5.1 实验设置

实验平台：我们在 18 节点本地集群上进行了实验。每个节点配置两个 Intel(R) Xeon E5-2650 V4 CPU、64GB 内存和 56Gbps Mellanox ConnectX-3 IB RNIC，连接到 56Gbps Mellanox InfiniBand 交换机，每个节点运行 CentOS 7.8.2003。我们使用 6 个节点作为计算服务器，剩余的 12 个节点作为内存服务器来创建内存池，其总内存容量为 768 GB。内存服务器的 CPU 在初始化时只用于向 RNIC 注册内存，不参与编解码计算。计算池和内存池之间的数据传输使用单边 RDMA。

工作负载：我们同时考虑 microbenchmark 和真实应用工作负载。对于 mi-

crobenchmark，我们采用 Crail iobench [104, 117]，并使用它来评估单个操作的性能，例如写、读、降级读以及修复等操作。对于应用工作负载测试，我们考虑使用默认 Zipfian 分布的 YCSB [118] 和来自 IBM Docker 注册表 [109, 119] 的真实工作负载。

比较对象：我们将三个最先进的容错策略集成到 Crail 中，它们都使用单边 RDMA 进行公平比较。(1) **Replication (REP)：**副本是各种存储系统中使用的最广泛也是最简单的实现方法 [18-19]，我们实现了主从副本 [17]。业界广泛接受的标准是三副本，我们将其表示为 REP3。为了研究容错引入了多少性能损失，我们还考虑了无冗余的单副本（即 REP1）。(2) **MemoryEC：**调用 ISA-L 库进行编码，对整个对象进行编码后，将编码后的数据块和校验块一趟网络传输到内存服务器。它广泛用于分布式内存系统 [105-107, 120]，特别是 EC-Cache [106] 和 Cocytus [105] 也使用了这种方法。(3) **PipelineEC：**它是基于磁盘的分布式存储系统 [56, 121-122] 中使用的最先进的纠删码策略，也被 HDFS 等系统广泛采用 [5, 24]。PipelineEC 利用流水线设计来减少网络和磁盘 I/O 的高开销。

默认参数：我们实验默认使用 RS(4, 2) 码，因为它广泛用于许多系统 [7, 35, 109]，我们也研究了不同的纠删码参数对性能的影响。默认情况下，MicroEC 的编解码块大小和 RDMA 数据包大小分别为 4KB 和 16KB。由于 PipelineEC 也利用了流水线，我们使用 HDFS [5, 24] 中的默认设置，即编解码和网络数据包大小均为 64KB。我们还研究了各种参数对 MicroEC 的影响，例如编解码块大小、网络数据包大小和多个客户端。

对象大小：我们现在评估每个单独操作的性能，包括写、读、降级读和修复。我们对每个操作进行一千次测试并取平均结果。许多对象存储系统具有 MB 级对象大小，例如，Yahoo! 工作负载中的大多数文件都大于 20MB [123]；docker 镜像注册服务使用对象存储来存储大型容器镜像 [119, 124]，IBM 数据中心超过 20% 的对象大于 10MB；此外，也有很多研究关注大对象（>1MB）[106, 109]。因此，我们通过将对象大小从 16KB 变化到 64MB 来进行实验，并将默认对象大小设置为 1MB。由于延迟随着对象大小的不同而显著变化，为了更好地展示结果，我们将对象大小分为三组，即小对象、中等对象和大对象。

4.5.2　分离内存下的写性能评估

与存在的纠删码策略比较：图 4.9 通过比较 MicroEC, MemoryEC 和 PipelineEC 来展示写性能。我们有以下结论：首先，在不同的对象大小下，MicroEC 总是优于现有的纠删码策略：例如延迟降低高达 57.78%，吞吐率增加高达 136.96%。其次，MicroEC 对于大对象的提升更大，当对象大小不小于 256KB，这种改进就会非常显著。原因是当对象太小时，编码优化和流水线优化很少。此外，图 4.9(d)

表明，当对象大于 4MB 时，MicroEC 的吞吐率稳定在 3500MB/s 左右，大约是 PipelineEC 的 2 倍。最后，通过将 PipelineEC 与 MemoryEC 进行比较，我们发现简单地部署流水线只能在有限的情况下带来改进，例如对于中等大小的对象，因为编码和 RDMA 传输的延迟差距很大，并不能在相同粒度的包大小下能够很好地流水线化。

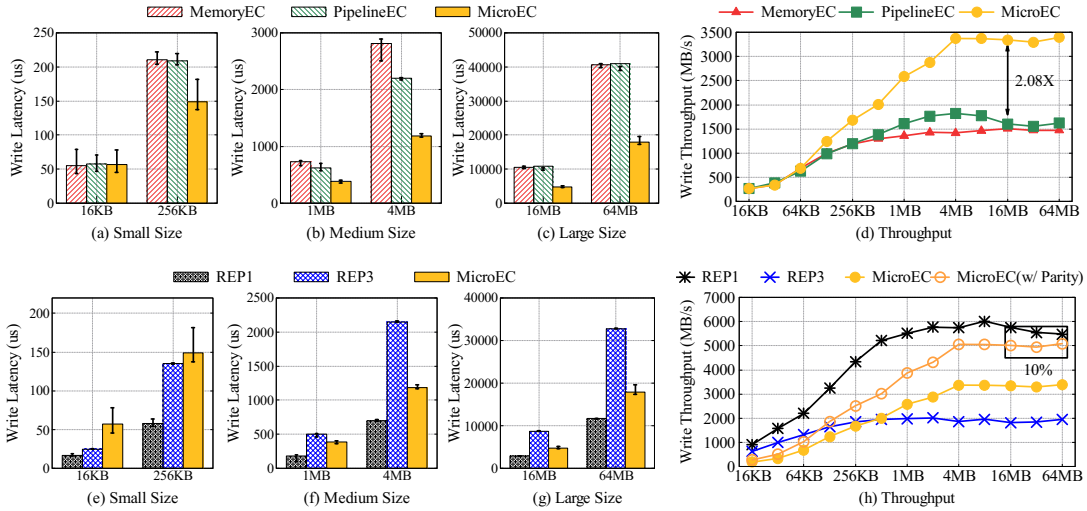


图 4.9　与纠删码以及副本比较写的时延以及吞吐率

与 REP3 比较：如图 4.9(e)-(h) 所示，与 REP3 相比，具有 (4, 2) 码的 MicroEC 提供了相同的可靠性，而内存成本仅为一半。此外，它还为中型和大型对象实现了更高的性能。例如，对于 64MB 的对象，它将写入延迟减少了 45.19%，并将写入吞吐率增加了 1.74 倍。REP3 的吞吐率很低，因为它写入额外的冗余副本会消耗网络带宽。对于小对象，MicroEC 会产生比三副本更高的延迟，这是因为 MicroEC 中编码优化的效果对于小对象来说会减弱，例如，对于 16KB 对象，不可避免的编码函数栈初始化和编码计算的开销主导了总写入延迟。

与 REP1 比较：由于 REP1 只写一个副本，可以看作是性能上限，MicroEC 增加了写入延迟，因为它需要额外的编码计算和校验块写入。但是，我们强调 MicroEC 已经充分优化，这可以通过 RDMA 带宽几乎占满来证明。具体来说，MicroEC 实现了 3392 MB/s 写入客户端数据的吞吐率，因此包括数据和奇偶校验块在内的总 RDMA 传输吞吐率为 $3392 \times 1.5 = 5088$ MB/s（在图中 4.9(h) 表示为“MicroEC w/ Parity”），总吞吐率达到 56Gbps RNIC 带宽限制的 80% 左右，仅比 REP1 低 10%。

4.5.3　分离内存下的读、降级读以及修复性能评估

我们现在评估读、降级读以及故障修复的性能。图 4.10 中展示了延迟结果，省略类似趋势的吞吐率结果。

读性能：图 4.10(a) 展示了正常读取的平均延迟。由于正常读取不涉及编解码，因此所有纠删码策略的性能几乎相同。与 REP3 相比，MicroEC 将大对象拆分为小块，它带来了异步 RDMA 的好处，但由于数据包较小，RDMA `WRITE` 的多对一连接产生了额外的开销。综合这两个因素，MicroEC 实现了与 REP3 几乎相同的性能。

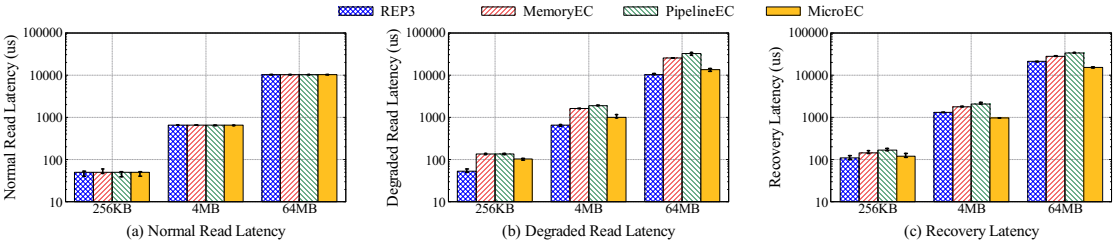


图 4.10　读、降级读以及修复性能

降级读性能：图 4.10(b) 展示了降级读延迟。MicroEC 始终优于 MemoryEC 和 PipelineEC：它将不同对象大小的平均延迟分别降低了 32.79% 和 41.46%。与 REP3 相比，所有纠删码策略都具有更高的延迟，因为它们需要读取幸存的块并执行解码。对于大对象，MicroEC 和 REP3 之间的延迟差距是可接受的，因为副本策略只需要读取一个可用块并且没有解码开销，对于 64MB 的对象，MicroEC 的降级读延迟为 13.27ms，REP3 为 10.31ms。

故障修复性能：图 4.10(c) 展示了修复不同大小的单个块的延迟。使用 (k, m) 码修复一个块，客户端首先通过单边 RDMA 从内存池中读取 k 个可用块，然后执行解码，最后将修复的块写回内存池。我们发现，与其他纠删码策略相比，MicroEC 实现了更低的延迟，例如，与 MemoryEC 和 PipelineEC 相比，它分别降低了高达 46.22% 和 55.13% 的延迟。此外，随着对象大小增加到 64MB，与 REP3 相比，MicroEC 还减少了 27.65% 的延迟。原因是 MicroEC 在所有阶段都利用了高效的非阻塞流水线，但 REP3 需要从内存服务器顺序读取整个大对象。

4.5.4　系统各阶段性能评估

我们这组实验来对 MicroEC 在系统各个阶段的性能优化进行拆解，从而更清楚的理解关键的优化环节。

编码和流水线优化的好处：我们以写一个 1MB 对象的过程为例来分析。我们通过比较 MicroEC 的三种设计配置来展示写入延迟：(1) MicroEC-CP，它去除了编码和流水线的优化，对象首先通过调用 ISA-L 库进行编码，然后通过单边 RDMA 传输到内存池；(2) MicroEC-P，只去除流水线优化，将编码块通过单边 RDMA 传输到内存池；(3) MicroEC，包括所有的设计和优化。图 4.11(a) 展示了编码优化将延迟降低了 39.18%（从 735μs 到 447μs），流水线优化进一步降低了延迟 12.98%（从 447μs 到 389μs）。最后，与没有优化的情况相比，MicroEC 总

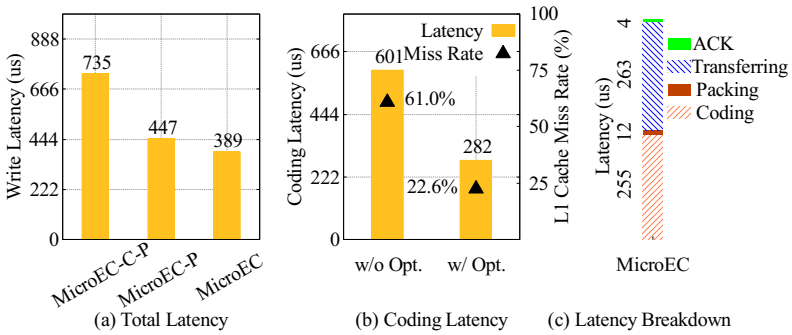


图 4.11　写 1MB 对象的性能拆解

共减少了近一半的延迟。

编码效率：我们进一步放大编码阶段以展示编码的时间成本和缓存未命中率。我们比较了在 MicroEC 中有和没有编码优化的两种方案，结果如图 4.11(b) 所示。我们发现通过利用 L1 cache 访问局部性来优化编码过程对于减少编码时间非常有效，例如，它将时间从 601μs 减少到 282μs。改进的一个主要因素是缓存未命中率降低了 62.95%。

流水线效率：现在我们研究流水线设计的效率。由于 MicroEC 的整个写入过程分为 *coding*、*packing*、*transferring* 和 *ACK* 四个阶段。因此我们忽略流水线效应来测量每个阶段的经过时间，结果如图 4.11(c) 所示。首先，我们发现编码阶段需要 255μs，传输阶段需要 263μs。也就是说，这两个阶段花费几乎相同的时间，这证明了流水线效率，因为编码和传输可以很好地流水线化，等待时间可以忽略不计。此外，我们发现打包阶段仅花费 12μs，这对于总时间来说可以忽略不计。这意味着通过维护 MicroEC 中的数据结构来进行地址管理会增加很小的开销。综上所述，使用 MicroEC，大部分时间开销是由编码和传输造成的，并且这两个操作可以有效地流水线。本组各阶段分解的性能分析也进一步证明了为什么 MicroEC 的性能甚至比 REP3 好得多。

4.5.5　灵敏度性能评估

我们这组实验对 MicroEC 关于不同参数配置下进行性能评估，从而进一步理解系统关键的灵敏度影响。

子块大小的影响：图 4.12(a) 展示了编码 1MB 对象时子块大小的影响。编码延迟在 4KB 子块大小处达到最小，主要原因有两方面：一方面，辅助数据，包括编码矩阵、乘法表和指令集等，占用大约 8KB，对于 4KB 的子块大小，我们在这里使用 (4,2) 码，因此可以在 L1 cache 中足够放下 $m+2=4$ 个子块（L1 cache 共 32KB）。注意，如果 L1 cache 不能保留所有 $m+2$ 个子块，则编码时间会迅速增加，例如，当子块大小从 4KB 增长到 16KB 时，编码时间会增加 119.15%。另一方面，使用小于 4KB 的子块大小会浪费 L1 cache 空间，并且需要多次编码函

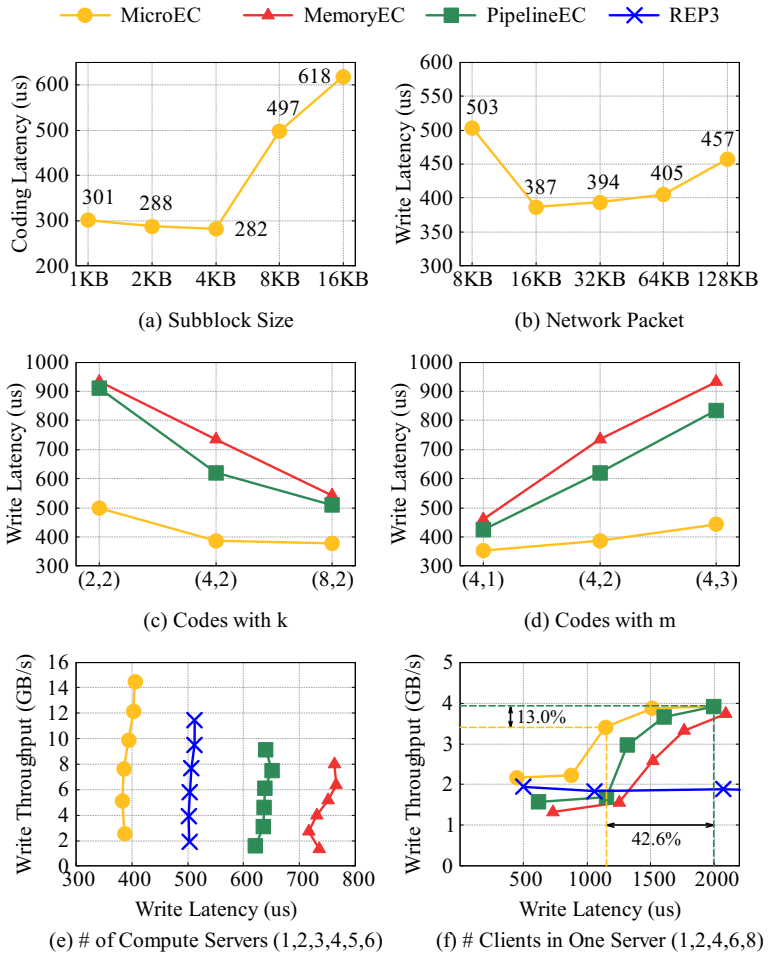


图 4.12　不同参数的灵敏度实验

数栈的调用才能将所需的 $m+2$ 个子块读入缓存。因此，在我们的测试平台中使用 4KB 子块大小实现了最低的编码延迟。

网络传输数据包大小的影响: 图 4.12(b) 展示了写入 1MB 对象的延迟。每次写入使用多趟 RDMA 传输。通过固定子对象大小为 16KB，并改变每次传输的数据包大小，我们发现写入延迟最小的时候为 16KB 的数据包大小。一方面大的数据包不适合流水线，因为网络会阻塞流水线流程；另一方面当数据包从 16KB 不断减小时，写入延迟会显著增加，因为较小的 RDMA `WRITE` 数据包带来多趟的网络传输，从而网络性能较差，在之前的工作 [125-126] 中也有类似的现象。通过比较图 4.12(a) 和图 4.12(b)，我们看到只有仔细协调编码和网络传输才能实现最佳性能，否则，编码时间或者传输时间会显著增加，从而阻碍流水线效率。

纠删码参数的影响: 图 4.12(c)-(d) 通过改变 k 和 m 展示了不同纠删码参数的影响。结果表明，增加 k 会降低 MicroEC 的优化比率，而增加 m 会扩大 MicroEC 的优化比率，因为小的 k 以及大的 m 都意味着需要编码产生更大比例的校验数据。因此，MicroEC 在更高级别的容错方面受益更多，这也证明了 MicroEC 的有效性，因为越来越多的应用程序需要比三副本更高的可靠性。

多客户端的影响：我们首先让每个计算服务器只运行一个客户端，因此每个客户端使用一个专用的 RNIC 进行数据传输。图 4.12(e) 通过将服务器数量从 1 变为 6 来显示吞吐率（y 轴）和延迟（x 轴）。我们看到，使用更多服务器可以持续增加吞吐率，而不会为所有设计牺牲太多延迟。这是因为客户端使用不同的 RNIC，写竞争并不严重。MicroEC 优于所有现有设计，并且随着计算服务器数量的增加，吞吐率的提高变得更大。

图 4.12(f) 展示了在单个服务器上运行多个客户端的结果。我们考虑 1、2、4、6 和 8 个客户端，它们共享同一个 RNIC。请注意，对于 1、2 和 4 个客户端，副本的吞吐率保持稳定，因为冗余副本的写入会消耗网络带宽。尽管现有的纠删码设计提高了多个客户端的吞吐率，但由于网络竞争，写入延迟也增加了。此外，与拥有 8 个客户端的 PipelineEC 相比，具有 4 个客户端的 MicroEC 实现了 87% 的吞吐率，而将延迟降低 42.6%。也就是说，MicroEC 可以用更少的客户端实现与 PipelineEC 相同的吞吐率，从而减轻竞争并实现更低的延迟。最后，当使用 8 个客户端时，由于网络带宽成为瓶颈，MicroEC 的改进效果减弱。

4.5.6　真实应用性能评估

YCSB 工作负载：为了验证 MicroEC 在实际应用中的效率，我们运行 YCSB benchmark [118]。我们展示了 YCSB A 下的性能结果，这是一个读写混合工作负载，包含 50% 的读取和 50% 的写入。我们将工作负载配置为客户端发送 30000 个请求并将对象大小设置为 1MB。我们在六个计算服务器上运行六个并发客户端，它们共享同一个内存池。

图 4.13(a) 展示了正常模式下请求延迟的 CDF 性能图。每条曲线可以分为两部分: 左下部分对应读请求, 右上部分对应写请求。我们看到 MicroEC 在不牺牲正常读取性能的情况下显著降低了写入延迟。特别是，与 PipelineEC 和 MemoryEC 相比, MicroEC 将平均写入延迟分别降低了 36.63% 和 45.64%。此外, 对于 97% 请求，MicroEC 的性能优于 REP3，而其余 3% 请求的延迟更长，因为纠删码需要比三副本建立更多的网络连接。图 4.13(b) 展示了将所有读请求设置为降级读请求的性能, 我们观察到类似的写入结果。对于降级读, MicroEC 显著优于 PipelineEC 和 MemoryEC：它将平均延迟分别降低了 45.10% 和 36.67%。但是，由于需要解码，MicroEC 的平均降级读延迟比 REP3 高 34.09%。

IBM Docker 注册表真实工作负载：对于 IBM Docker 注册表真实工作负载 [109, 119]，我们选取了负载最高的 Dallas 数据中心的数据集，我们重放了该数据集在在 6 小时内请求。对象大小从 16KB 到 428.11MB 不等，数据集大小为 29.25GB，其中 96% 的请求是读取，被视为降级读取，平均大小为 19.97MB。

图 4.14 展示了延迟结果。为了便于阅读，我们根据对象大小将请求分为多

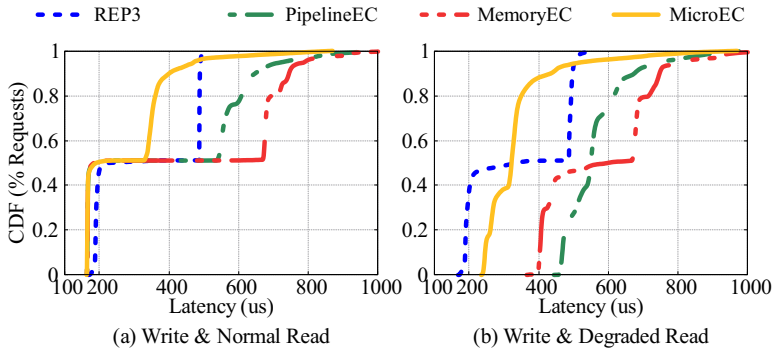


图 4.13　系统正常和降级模式下的 YCSB 性能

个组，并将纠删码的方案的平均延迟归一化为与 REP3 延迟的比例，因为 REP3 始终具有最低的降级读取延迟。对于每个纠删码策略，越接近蓝线（REP3 延迟），其性能越好。我们看到 MicroEC 总是优于 PipelineEC 和 MemoryEC。例如，对于 >10MB 对象组中的请求，与 PipelineEC 和 MemoryEC 相比，MicroEC 将平均延迟分别降低了 67.73% 和 63.80%。另外，MicroEC 实现了在大对象时与 REP3 接近的性能。

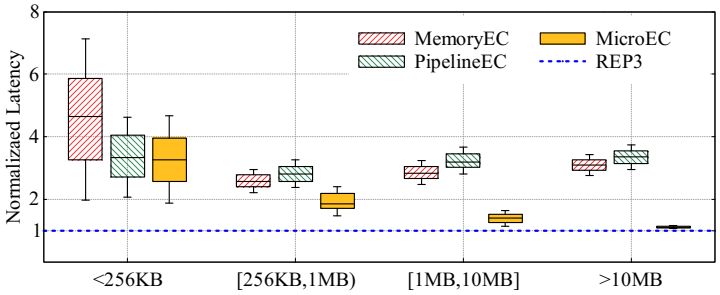


图 4.14　IBM Docker 注册表真实工作负载

4.6　本章总结

分离内存架构近些年在学术界和工业界都受到了广泛关注。由于共享内存池的规模很大，因此考虑分离内存系统的容错具有重要意义。纠删码提供了一种自然的选择，因为它能够以低内存成本提供高可靠性。当在分离内存系统中部署纠删码时，由于内存服务器的计算能力低，数据应该在计算池中执行编解码计算，然后传输到内存池。因此，编解码时延应与单边 RDMA 等远程内存访问的时延相匹配，同时也应保证客户端读写的低时延。然而，随着单边 RDMA 延迟可以达到微秒级，编解码计算成为分离内存系统部署纠删码的新瓶颈，不同于传统存储系统中稀缺的网络和磁盘 I/O 等资源。为了在分离内存系统中实现纠删码高效部署，我们先分析了编解码和 RDMA 传输的工作流程来提供指导设计的三个关键发现。然后，我们通过缓存优化重新设计了编解码函数栈，并利用高效的流水线来协同优化编解码计算以及 RDMA 传输。

针对本章介绍的 MicroEC 流程设计，做以下总结：

(1) 针对分离内存系统中部署纠删码，我们从缓存效率以及 RDMA 传输的工作流程的角度，分析了编解码函数栈。我们发现了在分离内存系统中部署纠删码时的关键性能瓶颈。我们提供了三个关键系统发现，用于在系统级别优化编解码和 RDMA 传输工作流程。我们提出一种新的纠删码方案 MicroEC，首次在分离内存系统中部署纠删码协同优化纠删码计算以及网络传输。

(2) MicroEC 重新设计了编解码函数栈，通过缓存对齐的条带化和重用辅助编解码数据来加速编解码。MicroEC 还提出了有效的数据结构来支持新的设计。

(3) MicroEC 通过设计非阻塞流水线进行编解码和 RDMA 传输，并仔细调整编解码以及传输网络包的大小实现最大化并行度。因此，MicroEC 可以在具有高性能和低内存成本的分离内存系统中实现了高效的纠删码部署。

(4) 我们实现了一个具有一般操作支持的系统原型，例如写/读/降级读/修复。实验表明，MicroEC 显著降低了编解码延迟，优于目前最先进的纠删码和三副本策略。

第 5 章　总结与展望

本章首先对全文进行了工作总结，介绍了本文在纠删码容错存储系统的编解码流程优化研究。然后指出本文研究工作的不足之处，以及关于纠删码容错存储系统未来可能的研究方向。

5.1　工作总结

本文主要关注于纠删码容错存储系统中在不同的应用场景和需求下，对纠删码存储策略进行编解码流程的设计与优化，以满足系统对关键指标的要求。具体包括基于纠删码存储系统的数据布局设计来优化故障修复性能、基于纠删码存储系统的故障修复任务调度设计来均衡修复负载以及基于分离内存系统架构的纠删码流程设计来提供高可靠/高性能存储系统。本文主要研究内容与创新点如下:

1）基于纠删码存储系统的数据布局设计:

纠删码存储策略可以同时提供高可靠性和低存储开销，从而在分布式存储系统中的部署变得越来越流行。然而，传统的随机数据放置在故障修复过程中会导致大量的跨机架流量和分批修复的负载不均衡，从而显著地降低了修复性能。此外，分布式存储系统中常常部署混合纠删码来满足多样性的用户需求，但是多纠删码策略进一步加剧了上述问题。我们提出了一种基于数学工具的均匀数据布局 PDL 来优化分布式存储系统中的故障修复性能。PDL 是基于成对均衡设计（一种具有均衡数学特性的组合设计工具）构造的，因此为混合纠删码提供了均匀的数据分布。基于数据布局设计 PDL，我们提出了负载均衡的故障修复方案 rPDL。该修复方案通过确定地选择替代节点并读取幸存块来修复丢失的块，从而有效地减少了跨机架流量，并提供了近似均衡的跨机架流量分布。

2）基于纠删码存储系统的故障修复任务调度设计:

在基于纠删码的分布式存储系统中，往往随机分布数据块/校验块，在故障修复过程中随机选择源节点和替代节点。在大规模的存储系统中，存有大量的数据，这种随机分布的方法可以达到数据存储的均匀分布与负载均衡。但是，由于内存容量、网络带宽、CPU 计算能力等方面的限制，故障盘中数据的修复过程是分批次执行的。现有的修复方法将序列号连续的一些待修复的任务打包成一个批次，且随机选择源节点和替代节点，造成在每个批次内故障修复的负载严重不均衡，影响故障修复速度。我们提出了 SelectiveEC 修复调度模块，一方面设计了二分图模型和批处理算法，将修复任务打包成批处理，并在一个批次确定性选

择源节点，均衡节点之间的上行修复流量；另一方面设计了另一个二分图模型和最大匹配算法，来选择替代节点，以均衡其下行修复流量和解码负载。

3）基于分离内存架构的纠删码流程设计：

分离内存架构近些年在学术界和工业界都受到了广泛关注。由于共享内存池的规模很大，因此考虑对分离内存系统的容错具有重要意义。纠删码提供了一种自然的选择，因为它能够以低内存成本提供高可靠性。然而，随着单边 RDMA 延迟可以达到微秒级，不同于传统存储系统中稀缺的网络和磁盘 I/O 等资源是性能瓶颈，编解码计算成为分离内存系统部署纠删码的新瓶颈。为了在分离内存系统中实现纠删码高效部署，我们先通过仔细分析了编解码和 RDMA 传输的工作流程来提供指导设计的三个关键系统发现。然后，我们设计了 MicroEC，它通过缓存优化重新设计了编解码函数栈，并利用高效的流水线来协同优化编解码计算和 RDMA 传输。

5.2　未来展望

本文尽管对纠删码容错存储系统进行了大量的研究，但我们认为还存在以下几方面的优化：

1）纠删码存储系统的数据布局设计：分布式存储系统的规模越来越大，动态扩展性也越来越重要，关于纠删码存储系统的数据布局设计还存在以下的问题：(1) 现有的均匀数据布局依赖于组合设计工具 PBD 和 BIBD 等，但是这些工具在大参数（v>100，即机架数大于 100）下不容易构造出来，另外大参数下需要更大的元组表，导致地址索引表的元数据开销也很大。所以如何在大规模的分布式存储场景下借助于更合适的组合工具来提供均匀的数据分布索引，另外如何避免索引表的高额元数据开销也是需要进一步优化的。(2) 由于分布式存储较为复杂的拓扑结构，现有的均匀组合设计工具只能保证数据在一次修复之后是完全均衡的，所以还需要进一步数据迁移来做到均匀的数据分布。如何设计故障修复算法来避免数据迁移，或者设计故障后迁移算法来减少迁移开销，从而保证二次故障的均匀数据布局，也是非常重要的。

2）纠删码存储系统的故障修复任务调度设计：随着分布式存储系统的容量越来越大，单个节点甚至可以达到几十 TB 的容量 [50, 87]。所以单点故障带来的修复时间也会越来越长，修复对前端应用的干扰也会更长。关于纠删码存储系统的故障修复任务调度设计存在以下的问题：(1) 现有的调度方法只能对网络带宽进行动态感知，但多样性的应用对存储系统的计算以及磁盘 I/O 也是敏感的。如何设计更加智能的前端任务感知的调度模块，从而达到在线修复最小化干扰前端应用。(2) 现有的调度算法只针对不存在修复优先级的修复任务，可以动态

的改变修复任务的顺序。但是随着多用户以及异构的存储系统，可能修复任务存在着修复的优先级。如何在这些场景下设计更智能的修复调度方法，也是很重要的问题。

3）分离内存架构的纠删码流程设计：考虑分离内存系统的容错是一个新的问题，也面临着许多挑战。关于分离内存系统的纠删码流程设计存在以下的问题：(1) 并发多客户端场景。计算服务器上可能有多个客户端，因为编解码计算需要保证每个编解码线程一个核来复用 L1 cache 中的辅助数据，如何避免为每个客户端分配一个专用的核，从而更有效的利用计算服务器的算力资源。(2) 小文件场景。因为目前编解码计算对于大于 16KB 的对象有更大的好处。对于小文件场景，如何利用缓存的局部性来加快编解码过程。(3) 校验块更新场景。目前主要针对具有大对象的应用程序，例如对象存储，因此校验块更新通常通过重写整个对象 [127] 来执行。但对于键值数据库等系统需要进行校验块及时更新，另外更新的 I/O 也是非常小的，如何设计分离内存系统下的校验块更新机制，也是非常重要的问题。

参 考 文 献

[1] REINSELD W, et al. IDC: Chinawill havethelargestdatacircleintheworldin2025, US44613919[R]. Framingham: IDC, 2019 (inChinese)(ReinselD, 武连峰, GantzJF, 等. IDC：2025 年中国将拥有全球最大的数据圈, 2019.

[2] 中国互联网络信息中心(CNNIC). 第 49 次《中国互联网络发展状况统计报告》[EB/OL]. 2022. https://www.cnnic.net.cn/hlwfzyj/hlwxzbg/hlwtjbg/202202/t20220225_71727.htm.

[3] GHEMAWAT S, GOBIOFF H, LEUNG S T. The Google file system[C]//Proceedings of the 19th ACM Symposium on Operating Systems Principles. 2003: 29-43.

[4] WEIL S A, BRANDT S A, MILLER E L, et al. Ceph: A scalable, high-performance distributed file system[C]//Proceedings of the 7th Symposium on Operating Systems Design and Implementation. 2006: 307-320.

[5] SHVACHKO K, KUANG H, RADIA S, et al. The Hadoop distributed file system[C]// Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies. 2010: 1-10.

[6] CALDER B, WANG J, OGUS A, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency[C]//Proceedings of the 23rd ACM Symposium on Operating Systems Principles. 2011: 143-157.

[7] FORD D, LABELLE F, POPOVICI F I, et al. Availability in globally distributed storage systems[C]//Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. USA, 2010: 61-74.

[8] CHOWDHURY M, KANDULA S, STOICA I. Leveraging endpoint flexibility in data-intensive clusters[C]//Proceedings of the ACM SIGCOMM Computer Communication Review: volume 43. 2013: 231-242.

[9] CAI Q, GUO W, ZHANG H, et al. Efficient distributed memory management with RDMA and caching[J]. Proceedings of the VLDB Endowment, 2018, 11(11): 1604-1617.

[10] Intel Corporation. PMem is a revolutionary memory technology[EB/OL]. 2021. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[11] Intel Corporation. Intel optane technology[EB/OL]. 2021. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[12] BIRRITTELLA M S, DEBBAGE M, HUGGAHALLI R, et al. Intel® omni-path architecture: Enabling scalable, high performance fabrics[C]//Proceedings of the 23rd Annual Symposium on High-Performance Interconnects. 2015.

[13] CONSORTIUM G Z. Gen-Z[EB/OL]. 2019. https://genzconsortium.org/.

[14] SATHIAMOORTHY M, ASTERIS M, PAPAILIOPOULOS D S, et al. XORing elephants: Novel erasure codes for big data[J]. Proceedings of the VLDB Endowment, 2013, 6(5): 325-336.

[15] Octave Klaba. Fire Has Destroyed OVH’s Strasbourg Data Center(SBG2)[EB/OL]. 2021. https://www.datacenterknowledge.com/uptime/fire-has-destroyed-ovh-s-strasbourg-data-center-sbg2.

[16] OFweek 云计算网. 欧洲最大云服务商遇火灾，阿里云、腾讯云、华为云有哪些宕机故事？[EB/OL]. 2021. https://cloud.ofweek.com/news/2021-03/ART-178804-8120-30489457.html.

[17] BRESSOUD T C, SCHNEIDER F B. Hypervisor-based fault tolerance[J]. ACM Transactions on Computer Systems, 1996, 14(1): 80-107.

[18] OUSTERHOUT J, GOPALAN A, GUPTA A, et al. The RAMCloud storage system[J]. ACM Transactions on Computer Systems, 2015, 33(3): 1-55.

[19] DORMANDO. Memcached[EB/OL]. 2018. https://memcached.org/.

[20] APACHE. HDFS[EB/OL]. 2019. https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[21] HDFS. Configuration of HDFS erasure coding[EB/OL]. 2019. https://hadoop.apache.org/docs/r3.1.1/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[22] CEPH. Ceph erasure coding[EB/OL]. 2018. http://docs.ceph.com/docs/mimic/rados/operations/erasure-code/.

[23] HUANG C, SIMITCI H, XU Y, et al. Erasure coding in windows azure storage[C]// Proceedings of the 2012 USENIX Conference on Annual Technical Conference. 2012: 15-26.

[24] APACHE. HDFS Erasure Coding[EB/OL]. 2020. https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[25] HUANG C, SIMITCI H, XU Y, et al. Erasure coding in windows azure storage[C]// Proceedings of the 2012 USENIX Annual Technical Conference. 2012: 15-26.

[26] RASHMI K, SHAH N B, GU D, et al. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster[C]// Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems. 2013.

[27] RASHMI K, SHAH N B, GU D, et al. A ”hitchhiker’s” guide to fast and efficient data reconstruction in erasure-coded data centers[C]//Proceedings of the 2014 ACM Conference on SIGCOMM. 2014: 331-342.

[28] XIANG L, XU Y, LUI J, et al. Optimal recovery of single disk failure in RDP code storage

systems[J]. ACM SIGMETRICS Performance Evaluation Review, 2010, 38: 119-130.

[29] XU S, LI R, LEE P P, et al. Single disk failure recovery for x-code-based parallel storage systems[J]. IEEE Transactions on Computers, 2014, 63: 995-1007.

[30] FIKES A. Storage architecture and challenges[J]. Talk at the Google Faculty Summit, 2010, 535.

[31] MACWILLIAMS F J, SLOANE N J A. The theory of error-correcting codes: volume 16[Z]. 1977.

[32] KHAN O, BURNS R C, PLANK J S, et al. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads[C]//Proceedings of the 10th USENIX Conference on File and Storage Technologies. 2012: 20.

[33] DIMAKIS A G, GODFREY B, WAINWRIGHT M J, et al. Network coding for distributed storage systems[C]//Proceedings of the 26th IEEE International Conference on Computer Communications. 2007: 2000-2008.

[34] DIMAKIS A G, GODFREY P B, WU Y, et al. Network coding for distributed storage systems [J]. IEEE Transactions on Information Theory, 2010, 56(9): 4539-4551.

[35] VAJHA M, RAMKUMAR V, PURANIK B, et al. Clay Codes: Moulding MDS codes to yield an MSR code[C]//Proceedings of the 16th USENIX Conference on File and Storage Technologies. 2018: 139-154.

[36] PAMIES-JUAREZ L, BLAGOJEVIC F, MATEESCU R, et al. Opening the Chrysalis: On the real repair performance of MSR codes[C]//Proceedings of the 14th USENIX Conference on File and Storage Technologies. 2016: 81-94.

[37] MARK HOLLAND G A G. Parity declustering for continuous operation in redundant disk arrays[C]//Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems. 1992: 23-35.

[38] ZHANG G, HUANG Z, MA X, et al. RAID+: Deterministic and balanced data distribution for large disk enclosures[C]//Proceedings of the 16th USENIX Conference on File and Storage Technologies. 2018: 279-294.

[39] MUNTZ R R, LUI J C S. Performance analysis of disk arrays under failure[C]//Proceedings of the 16th International Conference on Very Large Data Bases. 1990: 162-173.

[40] WAN J, WANG J, YANG Q, et al. S2-RAID: A new RAID architecture for fast data recovery [C]//Proceedings of the 26th Symposium on Mass Storage Systems and Technologies. IEEE, 2010: 1-9.

[41] KE H, GUNAWI H S, BONNIE D, et al. Extreme protection against data loss with single-overlap declustered parity[C]//Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2020: 343-354.

[42] ZHANG G, WANG Z, MA X, et al. Determining data distribution for large disk enclosures with 3-d data templates[J]. ACM Transactions on Storage, 2019, 15(4): 1-38.

[43] LI R, HU Y, LEE P P. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems[J]. IEEE Transactions on parallel and distributed systems, 2017, 28(9): 2500-2513.

[44] BRINKMANN A, SALZWEDEL K, SCHEIDELER C. Efficient, distributed data placement strategies for storage area networks[C]//Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures. 2000: 119-128.

[45] STOICA I, MORRIS R T, KARGER D R, et al. Chord: A scalable peer-to-peer lookup service for internet applications[C/OL]//CRUZ R L, VARGHESE G. Proceedings of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 27-31, 2001, San Diego, CA, USA. ACM, 2001: 149-160. https://doi.org/10.1145/383059.383071.

[46] KARGER D, LEHMAN E, LEIGHTON T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web[C]//Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. 1997: 654-663.

[47] GOEL A, SHAHABI C, YAO S Y D, et al. SCADDAR: An efficient randomized technique to reorganize continuous media blocks[C]//Proceedings of the 18th International Conference on Data Engineering. IEEE, 2002: 473-482.

[48] HONICKY R, MILLER E L. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution[C]//Proceedings of the 18th International Parallel and Distributed Processing Symposium. 2004: 1357-1366.

[49] WEIL S A, BRANDT S A, MILLER E L, et al. CRUSH: Controlled, scalable, decentralized placement of replicated data[C]//Proceedings of the 2006 ACM/IEEE Conference on Supercomputing. 2006: 31-31.

[50] WANG Z, ZHANG G, WANG Y, et al. Dayu: Fast and low-interference data recovery in very-large storage systems[C]//Proceedings of the 2019 USENIX Annual Technical Conference. 2019: 993-1008.

[51] OUSTERHOUT K, WENDELL P, ZAHARIA M, et al. Sparrow: distributed, low latency scheduling[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles. 2013: 69-84.

[52] LI S, LAN T, RA M, et al. S3: Joint scheduling and source selection for background traffic in erasure-coded storage[C]//Proceedings of the 37th IEEE International Conference on Distributed Computing Systems. 2017: 2025-2030.

[53] LI S, LAN T, RA M, et al. Joint scheduling and source selection for background traffic in

erasure-coded storage[J]. IEEE Transactions on Parallel and Distributed Systems, 2018, 29 (12): 2826-2837.

[54] SHEN Z, SHU J, LEE P P. Reconsidering single failure recovery in clustered file systems[C]// Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2016: 323-334.

[55] SHEN Z, LEE P P, SHU J, et al. Cross-rack-aware single failure recovery for clustered file systems[J]. IEEE Transactions on Dependable and Secure Computing, 2017, 17(2): 248-261.

[56] LI R, LI X, LEE P P, et al. Repair pipelining for erasure-coded storage[C]//Proceedings of the 2017 USENIX Annual Technical Conference. 2017: 567-579.

[57] MITRA S, PANTA R, RA M R, et al. Partial-parallel-repair (PPR) a distributed technique for repairing erasure coded storage[C]//Proceedings of the 11th European Conference on Computer Systems. 2016: 1-16.

[58] 赵博彦, 侯锐, 张乾龙, 等. 松耦合数据中心体系架构研究综述[J]. 高技术通讯, 2020, 30 (1): 1-12.

[59] INTEL. Intel Rack Scale Architecture[EB/OL]. 2021. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf.

[60] PUTNAM A, CAULFIELD A M, CHUNG E S, et al. A reconfigurable fabric for accelerating large-scale datacenter services[C]//Proceedings of the 2014 ACM/IEEE 41st International Symposium on Computer Architecture. IEEE, 2014: 13-24.

[61] LIM K, CHANG J, MUDGE T, et al. Disaggregated memory for expansion and sharing in blade servers[J]. ACM SIGARCH computer architecture news, 2009, 37(3): 267-278.

[62] GU J, LEE Y, ZHANG Y, et al. Efficient memory disaggregation with infiniswap[C]// Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation. 2017: 649-667.

[63] DONG J, HOU R, HUANG M, et al. Venice: Exploring server architectures for effective resource sharing[C]//Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture. IEEE, 2016: 507-518.

[64] AGUILERA M K, AMIT N, CALCIU I, et al. Remote regions: A simple abstraction for remote memory[C]//Proceedings of the 2018 USENIX Annual Technical Conference. 2018.

[65] RUAN Z, SCHWARZKOPF M, AGUILERA M K, et al. AIFM: High-performance, application-integrated far memory[C]//Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation. 2020.

[66] TSAI S Y, SHAN Y, ZHANG Y. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores[C]//Proceedings of the

2020 Annual Technical Conference. 2020.

[67] WANG C, MA H, LIU S, et al. Semeru: A memory-disaggregated managed runtime[C]// Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation. 2020.

[68] CALCIU I, IMRAN M T, PUDDU I, et al. Rethinking software runtimes for disaggregated memory[C]//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021.

[69] LEE S S, YU Y, TANG Y, et al. MIND: In-network memory management for disaggregated data centers[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021.

[70] CARBONARI A, BESCHASNIKH I. Tolerating faults in disaggregated datacenters[C]// Proceedings of the 16th ACM Workshop on Hot Topics in Networks. 2017.

[71] COSTA P, BALLANI H, RAZAVI K, et al. R2C2: A network stack for rack-scale computers [C]//Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. 2015.

[72] HAN S, EGI N, PANDA A, et al. Network support for resource disaggregation in next-generation datacenters[C]//Proceedings of the 12th ACM Workshop on Hot Topics in Networks. 2013.

[73] GAO P X, NARAYAN A, KARANDIKAR S, et al. Network requirements for resource disaggregation[C]//Proceedings of the 12th USENIX symposium on operating systems design and implementation. 2016.

[74] TSAI S Y, ZHANG Y. Lite kernel RDMA support for datacenter applications[C]//Proceedings of the 26th Symposium on Operating Systems Principles. 2017.

[75] SHAN Y, LIN W, KOSTA R, et al. Disaggregating and consolidating network functionalities with superNIC[Z]. 2021.

[76] LIM K, TURNER Y, SANTOS J R, et al. System-level implications of disaggregated memory [C]//Proceedings of the 2012 IEEE International Symposium on High-Performance Comp Architecture. 2012.

[77] SHAN Y, HUANG Y, CHEN Y, et al. LegoOS: A disseminated, distributed OS for hardware resource disaggregation[C]//Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation. 2018.

[78] GUO Z, SHAN Y, HUANG Y, et al. Clio: A hardware-software co-designed disaggregated memory system[C]//Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems. 2022.

[79] ZUO P, SUN J, YANG L, et al. One-sided RDMA-conscious extendible hashing for disaggre-

gated memory[C]//Proceedings of the 2021 USENIX Annual Technical Conference. 2021.

[80] LEE Y, AL MARUF H, CHOWDHURY M, et al. Hydra : Resilient and Highly Available Remote Memory[C/OL]//Proceedings of the 20th USENIX Conference on File and Storage Technologies. Santa Clara, CA: USENIX Association, 2022: 181-198. https://www.usenix.org/conference/fast22/presentation/lee.

[81] COLBOURN C J, DINITZ J H. Handbook of combinatorial designs[Z]. 2006: 72-75,75-79.

[82] STINSON D R. Combinatorial designs: constructions and analysis[Z]. 2007: 12-15.

[83] BETH T, JUNGNICKEL D, LENZ H. Design theory[Z]. 1999.

[84] SHEN Z, SHU J, LEE P P C. Reconsidering single failure recovery in clustered file systems [C]//Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. 2016: 323-334.

[85] APACHE. HDFS 3[EB/OL]. 2020. https://hadoop.apache.org/docs/r3.1.2/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[86] HADOOP. Apache Hadoop 3.1.1[EB/OL]. 2019. http://hadoop.apache.org/docs/r3.1.1/.

[87] REN Z, SHI W, WAN J, et al. Realistic and scalable benchmarking cloud file systems: Practices and lessons from AliCloud[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(11): 3272-3285.

[88] FACEBOOK. HDFS-Raid[EB/OL]. 2014. https://github.com/facebookarchive/hadoop-20.

[89] DEAN J, GHEMAWAT S. MapReduce: Simplified Data Processing on Large Clusters [C/OL]//BREWER E A, CHEN P. Proceedings of the 6th Symposium on Operating System Design and Implementation, San Francisco, California, USA, December 6-8, 2004. USENIX Association, 2004: 137-150. http://www.usenix.org/events/osdi04/tech/dean.html.

[90] Wikipedia. InfiniBand[EB/OL]. 2021. https://en.wikipedia.org/wiki/InfiniBand.

[91] BONDY J A, MURTY U S R. Graph theory with applications[Z]. 1976.

[92] HARRIS J M, HIRST J L, MOSSINGHOFF M J. Combinatorics and graph theory: volume 2 [Z]. 2008.

[93] WIKIPEDIA. Chernoff bound[EB/OL]. 2020. https://en.wikipedia.org/wiki/Chernoff_bound.

[94] APACHE. Erasure Coding Support inside HDFS[EB/OL]. 2015. https://issues.apache.org/jira/browse/HDFS-7285.

[95] APACHE. HDFS Erasure Coding Phase II – EC with contiguous layout[EB/OL]. 2015. https://issues.apache.org/jira/browse/HDFS-8030.

[96] LI X, YANG Z, LI J, et al. Repair Pipelining for Erasure-coded Storage: Algorithms and Evaluation[J]. ACM Transactions on Storage, 2021, 17(2): 1-29.

[97] GITHUB. TestDFSIO[EB/OL]. 2015. https://github.com/apache/hadoop/blob/master/hadoo

p-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-jobclient/src/test/java/org/apache/hadoop/fs/TestDFSIO.java.

[98] CORPORATION H P. The machine: A new kind of computer [EB/OL]. 2020. https://www.hpl.hp.com/research/systems-research/themachine/.

[99] CORPORATION I. Intel rack scale design: Just what is it? [EB/OL]. 2018. https://www.datacenterdynamics.com/en/opinions/intel-rack-scale-design-just-what-is-it/.

[100] CORPORATION F. Facebook’s disaggregated racks strategy provides an early glimpse into next gen cloud computing data center infrastructures[EB/OL]. 2015. https://dcig.com/2015/01/facebooks-disaggregated-racks-strategy-provides-early-glimpse-next-gen-cloud-computing.html.

[101] CORPORATION M. Rack-scale computing[EB/OL]. 2013. https://www.microsoft.com/en-us/research/project/rack-scale-computing/.

[102] CORPORATION D. In bid for major carriers and service providers, Dell EMC rack scale infrastructure offers hyperscale principles[EB/OL]. 2017. https://www.enterpriseai.news/2017/09/12/bid-major-carriers-service-providers-dell-emc-rack-scale-infrastructure-offers-hyperscale-principles/.

[103] CORPORATION I. Intel(R) intelligent storage acceleration library[EB/OL]. 2021. https://github.com/intel/isa-l.

[104] STUEDI P, TRIVEDI A, PFEFFERLE J, et al. Unification of temporary storage in the nodekernel architecture[C]//Proceedings of the 2019 USENIX Annual Technical Conference. 2019.

[105] ZHANG H, DONG M, CHEN H. Efficient and available in-memory KV-store with hybrid erasure coding and replication[C]//Proceedings of the 14th USENIX Conference on File and Storage Technologies. 2016.

[106] RASHMI K, CHOWDHURY M, KOSAIAN J, et al. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding[C]//Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation. 2016: 401-417.

[107] TARANOV K, ALONSO G, HOEFLER T. Fast and strongly-consistent per-item resilience in key-value stores[C]//Proceedings of the 13th EuroSys Conference. 2018.

[108] WU S, SHEN Z, LEE P P. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic Reed-Solomon codes[C]//Proceedings of the 2020 International Symposium on Reliable Distributed Systems. 2020.

[109] WANG A, ZHANG J, MA X, et al. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache[C]//Proceedings of the 18th USENIX Conference on File and Storage Technologies. 2020.

[110] PLANK J S, XU L. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network stor-

age applications[C]//Proceedings of the 5th IEEE International Symposium on Network Computing and Applications. 2006.

[111] ZHOU T, TIAN C. Fast erasure coding for data storage: A comprehensive study of the acceleration techniques[C]//Proceedings of the 17th USENIX Conference on File and Storage Technologies. 2019.

[112] BLOEMER J, KALFANE M, KARP R, et al. An XOR-based erasure-resilient coding scheme [J]. Technical Report TR-95-048, University of California at Berkeley, 1995.

[113] LINUX. Perf[EB/OL]. 2009. https://perf.wiki.kernel.org/.

[114] WIKIPEDIA. Cache replacement policies[EB/OL]. 2021. https://en.wikipedia.org/wiki/Cache_replacement_policies.

[115] PLANK J S, SIMMERMAN S, SCHUMAN C D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2[J]. University of Tennessee, Tech. Rep. CS-08-627, 2008, 23.

[116] WIKIPEDIA. CPU cache[EB/OL]. 2021. https://en.wikipedia.org/wiki/CPU_cache.

[117] FOUNDATION A S. Crail iobench[EB/OL]. 2018. https://incubator-crail.readthedocs.io/en/latest/iobench.html.

[118] COOPER B F, SILBERSTEIN A, TAM E, et al. Benchmarking cloud serving systems with YCSB[C]//Proceedings of the 1st ACM symposium on Cloud computing. 2010.

[119] ANWAR A, MOHAMED M, TARASOV V, et al. Improving docker registry design based on production workload analysis[C]//Proceedings of the 16th USENIX Conference on File and Storage Technologies. 2018.

[120] SHANKAR D, LU X, PANDA D K. High-performance and resilient key-value store with online erasure coding for big data workloads[C]//Proceedings of the 37th International Conference on Distributed Computing Systems. 2017.

[121] SHI H, LU X, SHANKAR D, et al. UMR-EC: A unified and multi-rail erasure coding library for high-performance distributed storage systems[C]//Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. 2019.

[122] SHI H, LU X. TriEC: Tripartite graph based erasure coding NIC offload[C]//Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis. 2019.

[123] YU Y, HUANG R, WANG W, et al. SP-cache: Load-balanced, redundancy-free cluster caching with selective partition[C]//Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis. 2018.

[124] LITTLEY M, ANWAR A, FAYYAZ H, et al. Bolt: Towards a scalable docker registry via hyperconvergence[C]//Proceedings of the 12th International Conference on Cloud Computing.

2019.

[125] KALIA A, KAMINSKY M, ANDERSEN D G. Design guidelines for high performance RDMA systems[C]//Proceedings of the 2016 USENIX Annual Technical Conference. 2016.

[126] MITCHELL C, GENG Y, LI J. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store[C]//Proceedings of the 2013 USENIX Annual Technical Conference. 2013.

[127] ARMBRUST M, DAS T, SUN L, et al. Delta lake: High-performance ACID table storage over cloud object stores[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 3411-3424.

[128] CEPH. Erasure codes in Ceph[EB/OL]. 2016. https://docs.ceph.com/en/latest/rados/operations/erasure-code/.

附录 A　第 2.4.3-D 节的理论结果证明

A.1　两故障关于 $CRRT_w$ 和 $CRRT_{w/o}$ 的分析

在本节中，我们将详细分析两故障时关于 $CRRT_w$ 和 $CRRT_{w/o}$ 的分析，从而来确定是否使用局部解码进行修复。考虑一个 (k,m) 码，假设 $k=tm+q, 0\leqslant q\leqslant m-1$，其中 $m\geqslant 2$ 且 $k>m$。我们按照 $q=0$ 和 $q>0$ 将讨论分为两种情况，其中后者进一步细分为三种情况，$m=2$、$t=1$ 和 $m\geqslant 3, t\geqslant 2$。

A.1.1　$k=tm$

每个条带有 $t+1$ 个满源机架，并且没有未满源机架。也就是说，当这样的条带故障时，它是一个满的条带，并且从任意非源机架中选择了两个替代节点，并且 $CRRT_{w/o}=k$（参见公式 2.3）。我们根据故障模式讨论 $CRRT_w$。

情况 1：同一机架 R_f 发生双块故障。其他 t 源机架正好包含 $k=tm$ 可用块，足以修复故障的块。R-w-PD 选择 k 个块，为每个故障的块聚合源机架中的 m 个块，并将局部解码的块发送到替代节点，t 个块发送到一个替代节点。因此，$CRRT_w=2t\leqslant tm=k=CRRT_{w/o}$，这里 $m\geqslant 2$。

情况 2：双块故障发生在不同的机架中，分别命名为 R_{f_1} 和 R_{f_2}。剩余的 $t-1$ 个源机架正好包含 $(t-1)m<k$ 个可用块，并且 R_{f_1} 和 R_{f_2} 都包含 $m-1$ 个可用块。因此，用于修复的 k 个块由 R_{f_1} 和 R_{f_2} 之一的 1 个块组成，比如 R_{f_1}，来自另一个机架的 $m-1$ 个块，比如 R_{f_2}，以及其他 $(t-1)m$ 块来自剩余的 $t-1$ 个源机架。注意到 R_{f_1} 只提供一个块，该块被跨机架发送到第一个替代节点，并由机架中的第一个替代节点发送到另一个替代节点，仅诱导一个 CRRT。对于其他至少提供 2 个块的源机架，每个机架为每个故障的块聚合其中的源数据块，即执行两次局部解码，并将每个聚合块传输到每个替代节点，从而产生 2 个 CRRT。所以当 $m\geqslant 3$，$CRRT_w=1+2t\leqslant k=CRRT_{w/o}$。当 $m=2$ 时，R_{f_2} 包含 $m-1=1$ 块，像 R_{f_1} 一样导致一个 CRRT，所以 $CRRT_w=2+2(t-1)=2t\leqslant k=CRRT_{w/o}$。

总之，$CRRT_w\leqslant CRRT_{w/o}$，当且仅当 $m=2$ 时等式成立。因此，对于 (k,m) 码的条带，当 $k=tm, m\geqslant 3$ 时，总是采用 R-w-PD，其他情况时采用 R-w/o-PD。

A.1.2　$k=tm+q, 1\leqslant q\leqslant m-1$

1. $m=2$

这时候，$k=2t+1$，对于每个条带，有 $t+2$ 个源机架存储其块，其中有 $t+1$ 个满的源机架，每个具有 2 个块，1 个仅包含 1 个块的未满源机架。通过公式 2.3，无论故障的条带是否已满，$CRRT_{w/o}=k$。我们根据故障模式讨论 $CRRT_w$。

情况 1：同一机架发生双块故障。剩余的 $t+1$ 源机架（t 个满源机架和 1 个未满源机架）正好包含 k 个可用块。由于存在一个未满的源机架，因此在其中选择一个替代节点，而在任意非源机架中选择另一个。为了修复每个故障的块，每个满的源机架聚合其中的 2 个可用块，并将局部解码的块跨机架发送到专用的替代节点，诱导 1 个 CRRT。共 t 个满源机架，所以导致 $2t$ 个 CRRT；未满源机架仅在机架之间传输一个块，因为其中有一个替代节点，从而产生 1 个 CRRT。因此，$CRRT_w = 2t+1 = k = CRRT_{w/o}$。

情况 2：双块故障发生在不同的满源机架中，分别命名为 R_{f_1} 和 R_{f_2}。所有可用的块都是修复所需要的。在未满源机架中选择一个替代节点，在任意非源机架中选择另一个替代节点。除了 R_{f_1} 和 R_{f_2} 之外，每个 $t-1$ 满源机架都将 2 解码块传输到两个替代节点，总共诱导 $2(t-1)$ 个 CRRT。虽然 R_{f_1}（或者 R_{f_2}）只包含一个可用块，但它会将块分别跨机架传输到两个替代节点，从而产生 2 个 CRRT。未满的源机架，其中有一个替代节点，跨机架只传输一个块，产生 1 个 CRRT。所以，$CRRT_w = 2(t-1)+2\times 2+1 = 2t+3 = k+2 > CRRT_{w/o}$。

情况 3：一个满源机架和一个未满源机架发生双块故障，分别命名为 R_{f_1} 和 R_{f_2}。由于此时的故障条带已满，并且修复需要所有的可用块。因此，两个替代节点是在一个随机的非源机架中选择的，比如 R_r。类似地，除了 R_{f_1} 之外的 t 个满源机架将 $2t$ 个局部解码的块全部发送到替代节点；R_{f_1} 中的唯一可用块被发送到 R_r 中的一个替代节点，诱导 1 个 CRRT，并发送该块到 R_r 中的另一个替代节点，不诱导 CRRT。所以，$CRRT_w = 2t+1 = k = CRRT_{w/o}$。

总而言之，$CRRT_w \geqslant CRRT_{w/o}$ 始终成立，因此对于 (k,m) 码的条带总是采用 R-w/o-PD，这里 $k = 2t+1$。

2. $t = 1$

这时候 $k = m+q$，其中 $m \geqslant 3$ 以及 $1 \leqslant q \leqslant m-1$，并且有 3 个组。由于我们只讨论每组条带包含 m 或 $m-1$ 个块。因此，根据条带中满组的数量，可以分为下面三种情况：

情况 1：两个满的组。每组中的块数分别为 m, m 和 $m-1$，也就是 $k = 2m-1$。通过类似的分析，我们可以得到 $CRRT_w < CRRT_{w/o}$，除了 $(5,3)$ 和 $(7,4)$ 码的条带在两个满的源机架中发生了两块故障。详细的分析内容如下：

子情况 1.1：双块故障都发生在一个满的源机架中。故障的条带未满，$CRRT_{w/o} = k+2-m = m+1$。所以，$CRRT_w = 3 < m+1 = CRRT_{w/o}$，这里 $m \geqslant 3$。

子情况 1.2：双块故障都发生在未满源机架中。由于 $m \geqslant 3$，故障的条带已满，$CRRT_{w/o} = k = 2m-1 \geqslant 5$。因此，$CRRT_w = 4 < CRRT_{w/o}$。

子情况 *1.3*：双块故障发送在两个满的源机架中。故障的条带未满，$CRRT_{w/o} = m+1$。因此，当 $m \geqslant 5$ 时，$CRRT_w = 5 < m+1 = CRRT_{w/o}$。当 $m \leqslant 4$，$CRRT_w \geqslant CRRT_{w/o}$。也就是说，对于 $(5,3)$ 或 $(7,4)$ 码的条带，如果两个满组中有两个故障的块，则首选 R-w/o-PD。

子情况 *1.4*：双块故障发送在一个满源机架和一个未满源机架中。由于 $m \geqslant 3$，故障的条带已满，$CRRT_{w/o} = k = 2m-1 \geqslant 5$。因此，$CRRT_w = 4 < CRRT_{w/o}$。

情况 *2*：一个满的组。每组中的块数分别为 $m, m-1$ 和 $m-1$。也就是说，$k = 2m-2$。通过与上述相同的分析，我们得到 $CRRT_w < CRRT_{w/o}$，这里 $m \geqslant 4$，以及 $(4,3)$ 码条带的两个故障都发生在满的源机架中。详细的分析内容如下：

子情况 *2.1*：双块故障都发生在满的源机架中。故障的条带未满，$CRRT_{w/o} = k+2-m = m$。所以，$CRRT_w = 2 < m = CRRT_{w/o}$，这里 $m \geqslant 3$。

子情况 *2.2*：双块故障都发生在一个未满源机架中。故障的条带未满，$CRRT_{w/o} = m$。因此，当 $m \geqslant 4$ 时，$CRRT_w = 3 < CRRT_{w/o}$。

子情况 *2.3*：双块故障发生在一个满源机架和一个未满源机架中。故障的条带未满，$CRRT_{w/o} = m$。因此，当 $m \geqslant 4$ 时，$CRRT_w = 3 < CRRT_{w/o}$。

子情况 *2.4*：双块故障发生在两个未满源机架中。故障的条带已满，$CRRT_{w/o} = k = 2m-2$。因此，当 $m \geqslant 4$ 时，$CRRT_w = 4 < CRRT_{w/o}$。

情况 *3*：没有满的组。每组中的块数分别为 $m-1, m-1$ 和 $m-1$。也就是说，$k = 2m-3$。注意 $k > m$，这意味着 $m \geqslant 4$。通过相同的分析，我们获得了 $CRRT_w < CRRT_{w/o}$，除了 $(5,4)$ 码条带的两个机架中发生了两块故障。详细的分析内容如下：

子情况 *3.1*：双块故障都发生在一个未满源机架中。由于 $m \geqslant 4$，故障的条带是未满的，$CRRT_{w/o} = k+2-m = m-1 \geqslant 3$。因此，$CRRT_w = 2 < CRRT_{w/o}$。

子情况 *3.2*：在两个未满源机架中发生双块故障。故障的条带未满，$CRRT_{w/o} = m-1$。因此，当 $m \geqslant 5$ 时，$CRRT_w = 3 < CRRT_{w/o}$。

综上所述，$CRRT_w < CRRT_{w/o}$ 并采用 R-w-PD，除了下面三个反例：(1) $(5,3)$ 或 $(7,4)$ 在两个满源机架中发生两块故障的条带；(2) 在一个未满源机架中发生至少一次故障的 $(4,3)$ 码条带；(3) $(5,4)$ 码的条带在两个机架中发生了两块故障。

3. $m \geqslant 3, t \geqslant 2$

这时候有 $t+2$ 个组，包括 $m-q \geqslant 1$ 个未满组。我们根据故障模式讨论 $CRRT_w$。

情况 *1*：双块故障都发生在同一个机架中，记作 R_f。无论 R_f 包含 m 还是 $m-1$ 个块，其他 $t+1$ 个机架至少包含 k 个可用块。所以我们可以只从除 R_f 外的 $t+1$ 个机架中选择 k 个块用于修复。每个 $t+1$ 个机架聚合机架内的源数据块并将聚合块发送到两个替代节点，这意味着如果替代节点不在这些机架中，则

$2(t+1)$ 个局部解码的块将跨机架传输机架。但是，如果某个替代节点 N 在 $t+1$ 个机架之中，例如 R_r，那么 R_r 会将其中的所有源数据块发送到机架内的替代节点 N，从而减少一个跨机架的聚合块传输。也就是说，只要在未满源机架中存在替代节点，就可以减少一次跨机架传输。因此，我们根据是否存在除 R_f 之外的未满源机架来讨论 $CRRT_w$。详细的分析内容如下：

子情况 *1.1*：没有未满的源机架。两个替代节点在一个非源机架中，所以 $CRRT_w = 2(t+1)$。故障的条带已满，$CRRT_{w/o} = k = tm + q$。所以 $CRRT_w < CRRT_{w/o}$。

子情况 *1.2*：一个未满源机架。在未满源机架中分配一个替代节点，减少 1 个 CRRT。所以 $CRRT_w = 2(t+1) - 1 = 2t + 1$，但是因为故障的条带未满，有 $CRRT_{w/o} = k + 2 - m$。我们有 $CRRT_w < CRRT_{w/o}$。例如图 A.1(a)。

子情况 *1.3*：至少两个未满源机架。在任意两个未满源机架中分配两个替代节点，减少 2 个 CRRT。所以 $CRRT_w = 2(t+1) - 2 = 2t$，但是因为故障的条带是未满的，有 $CRRT_{w/o} = k + 2 - m$。我们有 $CRRT_w < CRRT_{w/o}$。例如图 A.1(b)。

情况 *2*：双块故障发生在两个满的源机架中，分别命名为 R_{f_1} 和 R_{f_2}。因此，其他 t 源机架包含 $k - m$ 个可用块，R_{f_1}（或者 R_{f_2}）包含 $m - 1$ 个可用块。为了读取 k 个块进行修复，所有 $t+2$ 个源机架都提供了可用块。也就是说，$t+2$ 个源机架都聚合机架内源数据块并将局部解码的块发送到替代节点，从而产生 $2(t+2)$ 次聚合块。请注意，有 $m - q \geqslant 1$ 个未满源机架，因此至少一个替代节点在未满源机架中，并且故障的条带是未满的，所以 $CRRT_{w/o} = k + 2 - m$。我们有 $CRRT_w < CRRT_{w/o}$，除了 $(8,3)$ 码的条带。详细的分析内容如下：

子情况 *2.1*：一个替代节点在未满源机架中。只有一个未满源机架，即 $m-q = 1$，所以 $CRRT_w = 2(t+2) - 1 = 2t + 3 < CRRT_{w/o}$，这里 $m \geqslant 4$，以及当 $m = 3$ 时 $CRRT_w = CRRT_{w/o}$。

子情况 *2.2*：两个替代节点在两个未满源机架中。至少有两个未满源机架，即 $m - q \geqslant 2$，所以 $CRRT_w = 2(t+2) - 2 = 2t + 2 < CRRT_{w/o}$。例如图 A.1(c)。

情况 *3*：一个未满源机架和一个满/未满源机架发生双块故障，分别命名为 R_{f_1} 和 R_{f_2}。除了 R_{f_1} 之外的 $t+1$ 个源机架只包含 k 个可用块，这意味着只有 $t+1$ 个源机架聚合的块并将局部解码的块发送到替代节点，从而产生 $2(t+1)$ 次聚合。我们有类似的 $CRRT_w < CRRT_{w/o}$。详细的分析内容如下：

子情况 *3.1*：没有替代节点在未满源机架中。$CRRT_w = 2(t+1)$，但是因为故障的条带已满，有 $CRRT_{w/o} = k$。显然，$CRRT_w < CRRT_{w/o}$。例如图 A.1(d)。

子情况 *3.2*：s 个替代节点在 s 个未满源机架中，其中 $s = 1, 2$。这时候，$CRRT_w = 2(t+1) - s$，但是因为故障的条带未满，有 $CRRT_{w/o} = k + 2 - m$。显然，$CRRT_w < CRRT_{w/o}$。$s = 1$ 的例子见图 A.1(e)。

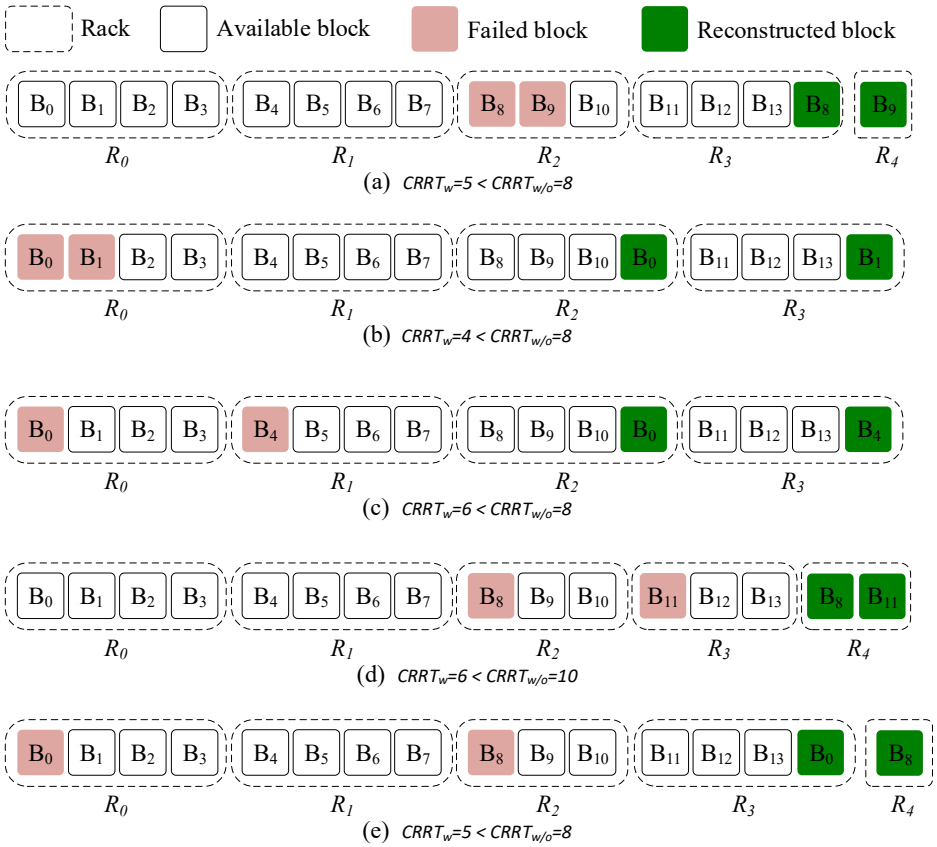


图 A.1　RS(10,4) 码的两块故障

综上来说，$CRRT_w < CRRT_{w/o}$，除了 $(8,3)$ 码的条带在两个满源机架中发生双块故障，因此除特例外均采用 R-w-PD。

基于上述的分析证明，除了少数特殊情况外，两故障修复优先选择 R-w-PD，并且适用于通常部署 RS$(6,3)$ 和 RS$(10,4)$ 码的工业界存储系统 [24-25, 88, 128]。

致　谢

光阴似箭，日月如梭。随着毕业论文的撰写完毕，科大五年硕博连读的时光进入收尾。回想这五年，很荣幸一路上遇到很多的老师以及同学们，在这里首先对您们在工作上以及生活上对我的帮助表示诚挚的感谢。

首先，我要感谢两位导师：许胤龙老师和吕敏老师。许老师和吕老师一路上对我科研工作以及生活无微不至的教导，让我受用终生。科研上，许老师教会了我严谨踏实、不怕困难。生活上，许老师教会了我谦逊礼貌、友善待人。读博阶段是我人生中最为宝贵的经历之一，我将牢记许老师的谆谆教诲。吕老师在科研工作上的认真仔细、生活上的佛系态度也让我感染。然后，我要感谢实验室几位年轻的老师：李永坤老师、李诚老师以及吴思老师，感谢您们一路上对我的指导以及帮助。很荣幸与其中几位老师都有过深度合作，与几位老师亦师亦友，感谢您们对先进系统实验室的发展尽心尽力，让同学们有着越来越大的舞台拼搏发挥。

然后，我要感谢读博期间一起奋斗的同学们。首先，感谢和我一起合作过的师兄以及师弟们，他们分别是：李志鹏、李启亮、汪威、杨振宇、谢灵江以及牛天洋等。感激以及怀恋与你们一起在科研道路上的奋斗以及互相鼓励，科研道路荆棘满满，我们都一一的解决了，team work 使得我们事半功倍。接着我要感谢实验室与我一届的小伙伴们，他们分别是：张强、邵新洋、苏景波、吴加禹、李佳伟以及光煦灿，17 年入学的我们七个葫芦娃，尽管我们现在工作生活状态不一，感激与你们的相遇以及陪伴，让我的研究生生涯丰富多彩。同时我要感谢室友，他们分别是：杨文、余辉煌以及徐景鑫，感激他们生活上的日夜相伴以及互相帮助。最后我要感谢先进数据系统实验室的其他师兄师姐师弟师妹们，读研时光里你们的帮助以及陪伴，使得我的研究生生涯精彩纷呈。他/她们分别是：张伟韬、郭帆、陈友旭、刘军明、刘彩银、陈吉强、田成锦、汪睿、白有辉、陈浩、周泉、许冠斌、王一多、姚路路、朱文喆、金泽文、林帅、李启亮、汪威、李嘉豪、林郅琦、火净泽、刘朕、余东波、游翎璟、左泽、王千里、阮超逸、王孟、董健、龚正、王霄阳、杨承儒、王海权、马凯、王时移、于笑颜、张家坤、毛浩宇、梁熙远、卢明祥、余伟强、邓龙、来逸瑞、杜清鹏、徐宇鸣、毕超、张钏楠、陈清源、龚平、柏志伟、张擎洋、朱嘉安、陈泺帆、李纯羽、朱先志、邢益鹏、曾源等同学。祝实验室同学科研顺利，万事如意！祝愿先进数据系统实验室（ADSL）越来越好，更上一层楼！

同时，我还要感谢在华为云存储 lab 实习时遇到的同学们。首先我要感谢实习的 mentor 左鹏飞，左博一直是我科研工作上的榜样，实习期间学习到了关于学术界以及工业界许多发现问题以及解决问题的新角度，感谢与你的长期合作。

另外，我还要感谢实习期间给予我生活上以及工作上极大帮助的其他同学，他们分别是：张双武、杨柳、冯雅植以及罗平等。

接着，我要感谢我的家人们。感谢父母对我二十多年的养育以及教导，让我逐渐长大成人。二十年的学业让爸妈饱受经济上以及生活上的艰难，我日益的成长也伴随着您们日益苍老的身躯，儿子以后一定会努力学习报答您们的恩情。同时，我要感谢我的两个姐姐，您们在我童年以及成年期间给了我莫大的帮助，让我无忧无虑的长大。此外，我还要感谢舅舅、舅妈、叔叔、婶婶等在我成长时的帮助，是您们的呵护，让我的童年健康快乐。最后，我要感谢我的女朋友陈婷婷，回想与你的大学相遇直到现在即将博士毕业，中间的片刻瞬间都是人生中最美的风景，感激与你一路陪伴，一起面对人生的重大抉择以及科研生活的种种困难，一起鼓励与进步，一切都没有那么困难了，祝福我们一起携手到老。

最后，感谢本文的审稿以及答辩专家，在百忙之中对我的毕业论文把好最后一道关。再次感激生命中遇到的老师、同学、朋友以及亲人们，祝福您们!

徐亮亮

2022 年 03 月

在读期间发表的学术论文与取得的研究成果

参与的科研项目

1. 国家自然科学基金重点项目 “新型分布式存储系统的高可靠性关键技术研究”, 编号：61832011。
2. 国家自然科学基金项目 “基于关联性的分布式元数据存取优化研究”, 编号: 61772486。
3. 国家重点研发计划项目 “数据科学的若干基础理论”, 编号: 2018YFB1003204。

已发表论文

1. **Liangliang Xu**, Min Lyu, Qiliang Li, Lingjiang Xie, Cheng Li and Yinlong Xu. “SelectiveEC: Towards Balanced Recovery Load on Erasure-coded Storage Systems”. IEEE Transactions on Parallel and Distributed Systems (**TPDS 2022**), 33(10), 2022.（中国计算机协会 CCF 推荐 A 类期刊）
2. **Liangliang Xu**, Min Lyu, Zhipeng Li, Cheng Li and Yinlong Xu. “A Data Layout and Fast Failure Recovery Scheme for Distributed Storage Systems with Mixed Erasure Codes”. Accepted for IEEE Transactions on Computers (**TC 2021**).（中国计算机协会 CCF 推荐 A 类期刊）
3. **Liangliang Xu**, Min Lv, Zhipeng Li, Cheng Li and Yinlong Xu. “PDL: A Data Layout towards Fast Failure Recovery for Erasure-coded Distributed Storage Systems”. IEEE International Conference on Computer Communications (**INFOCOM 2020**), Virtual Conference, 6-9 July, 2020. （中国计算机协会 CCF 推荐 A 类会议）
4. **Liangliang Xu**, Min Lyu, Zhipeng Li, Yongkun Li and Yinlong Xu. “Deterministic Data Distribution for Efficient Recovery in Erasure-Coded Distributed Storage Systems”. IEEE Transactions on Parallel and Distributed Systems (**TPDS 2020**), 31(10), 2020. （中国计算机协会 CCF 推荐 A 类期刊）
5. **Liangliang Xu**, Min Lyu, Qiliang Li, Lingjiang Xie and Yinlong Xu. “SelectiveEC: Selective Reconstruction in Erasure-coded Storage Systems”. The 12th USENIX Workshop on Hot Topics in Storage and File Systems (**HotStorage 2020**), Virtual Conference, 13 July, 2020.
6. Qiliang Li, Min Lyu, **Liangliang Xu**, Yinlong Xu and Wei Wang. “Fast Recon-

struction for Large Disk Enclosures Based on RAID2.0". The 50th International Conference on Parallel Processing (**ICPP 2021**), Virtual Conference, 9 - 12 August, 2021.（中国计算机协会 CCF 推荐 B 类会议）

7. Zhipeng Li, Min Lv, Yinlong Xu, Yongkun Li and **Liangliang Xu**. "D3: Deterministic Data Distribution for Efficient Data Reconstruction in Erasure-Coded Distributed Storage Systems". The 33rd IEEE International Parallel & Distributed Processing Symposium (**IPDPS 2019**), Rio de Janeiro, Brazil, 21 −25 May, 2019. （中国计算机协会 CCF 推荐 B 类会议）

已投稿论文

1. Qiliang Li, Min Lyu, **Liangliang Xu**, Yinlong Xu. "Fast Reconstruction for Large Disk Enclosures Based on RAID2.0". Submitted to IEEE Transactions on Computers (**TC 2022**).（中国计算机协会 CCF 推荐 A 类期刊）

待投稿论文

1. **Liangliang Xu**, Yongkun Li, Min Lyu, Qiliang Li, Wei Wang, Pengfei Zuo, Shuangwu Zhang and Yinlong Xu. "Enabling Efficient Erasure Coding in Disaggregated Memory Systems".

已公开专利

1. 吕敏，**徐亮亮**，李启亮，谢灵江，许胤龙，"一种基于纠删码存储系统的负载均衡修复调度方法"，专利号：202010313968.5，申请时间：2020.04.20，公开时间：2020.08.07。

获奖情况

- 中国科学技术大学优秀毕业生, 2022.
- 博士国家奖学金, 2021.
- 博士深交所奖学金, 2020.
- INFOCOM Student Conference Award, 2020.