分布式存储系统中的纠删码性能优化综述

蔡元浩[1)]

[1)](华中科技大学 计算机科学与技术学院，湖北省武汉市 435400)

摘 要 在高性能计算机集群中，使用纠删码替代多路复制能够节约存储成本，但纠删码会造成较高的计算和内存上的开销。为了降低使用纠删码造成的性能开销，本文分别介绍《利用程序优化技术加速基于 XOR 的纠删码》、《组合局部定位的大条带纠删码》、《LogECMem：将纠删码的内存键值存储与奇偶校验日志相结合》、《INEC：快速且连贯的网络内纠删码》和《NetEC：利用内网聚合加速纠删码重构》，这五篇关于提高纠删码性能方面的文章，以阐述当前纠删码的研究方向。

关键词 纠删码、基于 XOR 的纠删码、组合局部定位、LogECMem、INEC、NetEC

A Survey of Performance Optimization of Erasure Codes in Distributed Storage Systems

Cai Yuan-hao[1)]

[1)](School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan Hubei)

Abstract In high-performance computer clusters, using erasure codes instead of n-way replications can save storage costs, but erasure codes will cause high computing and memory overhead. In order to reduce the performance overhead caused by the use of erasure codes, this article introduces " Accelerating XOR-Based Erasure Coding using Program Optimization Techniques", " Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage","LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging", "INEC: Fast and Coherent In-Network Erasure Coding" and "NetEC: Accelerating Erasure Coding Reconstruction With In-Network Aggregation", these five articles is about improving the performance of erasure codes which illustrate the current research direction of erasure codes.

Key words erasure codes、XOR-Based Erasure Coding、Combined Locality、LogECMem、INEC、NetEC

1 引言

现代高性能计算机集群中，数据密集型应用对于高性能、可靠性存储系统的要求越来越高。现有的高性能计算存储系统大多数采用 n 路复制技术保证数据的可靠性和可用性，但是 n 路复制技术在保证可靠性的同时带来的缺陷是存储成本较高。因此，许多大型分布式存储系统正在向使用纠删码过渡。相比于 n 路复制，纠删码在提供高可用性的同时，降低了存储上的开销，但是另一方面，使用纠删码会造成较高的计算和内存上的开销。

为了降低纠删码在分布式存储系统中带来的性能开销，出现了许多的改进方案。本文主要对近三年内对纠删码性能提升方面的论文进行总结归纳，介绍几种不同方向上的纠删码改进方案。

1) 利用程序优化技术加速基于 XOR 的纠删码[1]：Yuya 等人基于观察提出了一种新颖的方法，基于 XOR 的 EC 为 XORing 字节数组虚拟生成领域特定语言的程序。并将此类程序形式化为编

译器构造的直线程序(SLP)，并使用各种程序优化技术对 SLP 进行优化。优化流程有三个方面：1）使用语法压缩算法减少 XOR 的数量;2)利用 Deforestation 减少内存访问，这是一种函数式程序优化方法;3)使用程序分析的(红-蓝)卵石博弈减少缓存失误。该项工作的贡献如下：

• 压缩。使用了压缩算法 RePair，该算法用于压缩语法压缩中的上下文无关语法(CFGs)。

• 融合。为了减少内存访问，在函数式程序优化中采用了一种称为 Deforestation 的技术。

• 调度。为了减少缓存遗漏，重新审视了众所周知但模糊的缓存优化准则，增加了数据访问的局部性。

2) 利用组合局部定位的大条带纠删码[2]：Yuchong Hu 等人提出了组合局部定位，这是第一个通过结合奇偶局部定位和拓扑局部定位系统地解决大条带修复问题的机制。该项工作的贡献如下：

• 第一个系统地解决大条带修复问题。提出的组合局部定位，减轻超低存储冗余下的跨机架修复带宽。研究了不同基于位置的方案的冗余和跨机架修复带宽之间的权衡。

• 设计了 ECWide，它实现了组合局部定位，以解决两种类型的修复：单块修复和全节点修复。还设计了一种高效的编码方案，允许大条带的奇偶校验块在多个节点上并行编码，以及一种内部机架奇偶校验更新方案，允许奇偶校验块在机架内进行本地更新，以减少跨机架传输。

• 实现了两个 ECWide 原型 ECWide-C 和 ECWide-H 来实现组合局部性。前者是为冷存储而设计的，而后者建立在基于 memcache 的内存键值存储上，用于热存储。

• 通过 Amazon EC2 实验将 ECWide-C 和 ECWide-H 与两种现有的基于位置的方案进行比较。结果表明，组合局部定位，单块修复时间分别比上述两种方案减少了 87.9%和 90.5%，而冗余度仅为 1.063 倍。

3) LogECMem[3]：Liangfeng Cheng 等人提出了一种新颖的基于奇偶校验日志的架构 HybridPL，它创建了就地更新(用于数据和 XOR 奇偶校验块)和基于日志的更新(用于剩余的奇偶校验块)的混合，从而平衡更新性能和内存成本，同时保持高效的单故障修复。并将 HybridPL 实现为内存中的键值存储，称为 LogECMem，并进一步设计了针对多种故障的高效修复方案。该项工作地贡献如下：

• 通过工作负载观察到，现有的用于纠删码的内存 KV 存储的更新方案要么会产生很多块传输，要么需要很高的内存开销，这促使开发一个基于奇偶校验日志的方案来平衡更新和内存的成本。

• 通过可靠性分析表明，改进单故障修复可以显著提高可靠性，因此提出了 HybridPL，它以低内存占用执行快速单故障修复的就地更新，同时利用平价日志记录和缓冲区日志记录进行快速更新。

• 在 HybridPL 上构建了一个内存 KV 存储 LogECMem，它实现了基本的请求(包括单故障修复)，基于更新的奇偶校验日志记录和缓冲区日志记录。还通过批处理合并奇偶校验更新来改进缓冲区日志记录。

• 在 LogECMem 上设计了高效的修复方案，用于多故障修复，以减少基于最先进的奇偶校验日志修复方案的磁盘 IOs，同时仍然保持高修复性能。

• 将 LogECMem 原型化，并通过 Amazon EC2 实验将其与就地更新和全条带更新进行比较。结果表明，LogECMem 在保持较高的基本 I/O 和修复性能的同时，将更新时间分别减少了 37.8%和 58.0%，内存开销分别减少了 22.2%和 49.0%。

4) INEC[4]：Haiyang Shi 等人提出了一套连贯的网络内 EC 原语，命名为 INEC。INEC 原语可以使不同类型的 EC 方案充分利用现代 SmartNIC 的 EC 卸载能力。该项工作的贡献如下：

•确定了为了完全支持五个最先进的 EC 方案，只需要三种类型的网络内 EC 原语

• 为了将这些 EC 原语和协议完全卸载到网络中，提出了 RDMA WAIT 的高效设计，以提供快速的 EC 和网络能力，并在 Mellanox OFED 驱动程序中实现了 INEC 原语。这是第一次提出了一套完整的连贯的网络内 EC 原语，在 RNIC 上使用 RDMA WAIT 来支持多种类型的 EC 协议。

•提出了一个 α - β 性能模型来分析五种最先进的 EC 方案的 INEC 原语的性能增益。

• 通过基准测试和与键值存储系统的协同设计，提出的 INEC 原语与五种最先进的 EC 方案从下到上进行了验证。基准测试中编码和解码的带宽分别提升了 5.87 倍和 2.94 倍。

5) NetEC[5]：NetEC 是 Yi Qiao 等人提出的一个网络内加速框架，它能够完全卸载 EC 到新一代可编程交换 ASICs。该项工作的贡献如下：

• 提出 NetEC，将 EC 卸载到交换 ASs，以解决 EC 的三个主要问题。

•设计了显式缓冲区大小通知(EBSN)来约束并发任务的缓冲区使用

• 设计了两种并行的 GF 卸载方法，以支持更大的数据包大小和更好的吞吐量

• 在硬件可编程交换机上实现 NetEC，并将其与 HDFS 集成

• 进行了评估并表明 NetEC 将重构率提高了 2.7~6.8 倍，显著降低了降级的读延迟，并完全消除了主机 CPU 开销。

2 利用程序优化技术加速基于 XOR 的纠删码

2.1 背景

以最为常见的纠删码——RS 编码为例。RS 编码在计算时最基本的原理是矩阵乘法，RS 空间效率的优势随着编码块的大小而显现出来。例如，在 RS(10, 4)上，一个系统的节点需要 N/10 字节的空间来处理一个 N 字节的数据。另一方面，矩阵乘法在有限域上的缺点是速度慢。及时使用最新的成果，n×n 矩阵相乘也需要~O（$n^{2.37287}$）复杂度。

RS 有两种主要的加速方法。

（1）将矩阵乘法和有限域乘法的复杂优化方法紧密地结合起来。英特尔提供了一个 EC 库，ISA-L（智能存储加速库），基于这种方法。ISA-L 对 $F_2{}^8$ 的矩阵乘法进行了特别的优化，并为每个平台提供不同的汇编代码，以最大限度地提高 SIMD 指令的性能。英特尔报告了 ISA-L 对 RS(10,4)的编码吞吐量约为 6.0GB/s。在本项工作的评估中，ISA-L 的得分约为 6.7 GB/s。

（2）基于 XOR 的 EC 将 $F_2{}^8$ 上的矩阵乘法转换为 F_2 上的矩阵乘法。hou 和 Tian 发表的研究，综合了几种执行阵列 XOR 的加速方法。这是基于基于 XOR 的 EC 的最先进的研究；然而，它对 RS(10, 4)编码的得分是 4.9 GB/s。

2.2 直线程序SLP

直线程序(SLP)是一种没有分支、循环和函数的程序。一个 SLP 是元组⟨V, C, s, g，⊗⟩，其中 V 是一组变量，C 是一组常数，s 是指令序列(即程序的主体)，g 是程序返回的变量序列，⊗是二进制操作符。

我们考虑一类 SLPs，即 XOR SLP，其二元运算符只满足关联性（（x⊕y）⊕z=x⊕（y⊕z））、交换性（x⊕y=y⊕x）和取消性（x⊕x⊕y=y）规律。我们用 $SLP_{\oplus}$来表示这个类。

2.3 使用RePair算法压缩SLP

在这项工作中，作者没有通过解决难以解决的优化问题来搜索最短的 SLPs，而是采用了语法压缩理论中的语法压缩算法 RePair 作为启发式算法。

对于一个 SLO P 和一对 term(x,y)，替换所有在 P 中出现的这一对 term，引入一个新的变量。把这个步骤称为 Pair(x,y)。

算法 1. RePair.

Loop

（1） 如果没有原始变量，则终止

（2） 否则，选择一对在原始变量的定义中中出现频率最高的 term，然后将这对 term 应用 Pair。如果有多对候选 term，则选择最小的一对。

图 1 展示了将 RePair 算法应用于一个 SLP P_0 的过程。

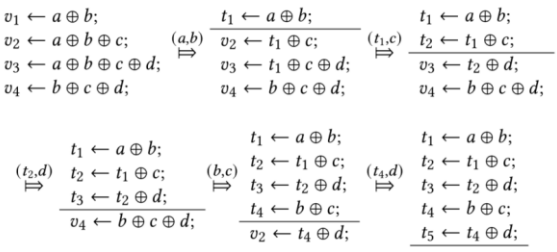


图 1 将 RePair 算法应用于 P_0

从图 1 的 RePair 过程中可以看到，RePair 算法将最初的八个 XOR 减少到五个，减少了 XOR 的数量。

2.4 XORRePair

通过容纳 XOR-cancellativity 来扩展 RePair。引入了一个辅助过程 REBUILD(v),它使用临时变量的值重写一个给定的原始变量 v。同时，海使用辅助符号(|w|)来表示变量 w 的值。

算法 2. REBUILD(v)

Initialize：让 rem:=（|v|）,S:= ∅。rem 表示一组要通过 XOR 现有的临时变量来消除的常数。

Loop

（1） 如果不能使 rem 缩短（没有临时变量 t 使得 |rem⊕(|t|)| < |rem|），则返回 rem ∪S 作为 v 的新的定义。

（2） 否则，选择一个临时变量 t 最小化|rem⊕(|t|)| 并且更新 rem:=rem⊕(|t|)，S:= S ∪ {t}。如果有多个可选的 t，则选择最小的一个。

图 2 展示了将 REBUILD(v)应用于 SLP 的过程。

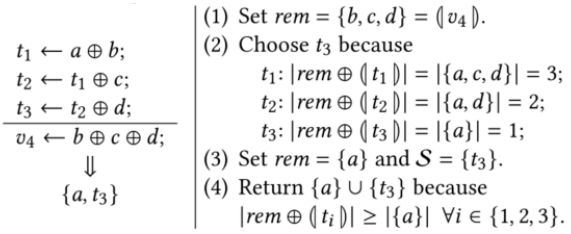


图 2 将 REBUILD(v)应用于 SLP

使用 REBUILD(v)增强 RePair，便得到了 XORRePair。

算法 3. XORRePair

Loop

（1） 与 RePair 的（1）相同

（2） 与 RePair 的（2）相同

（3） 对每一个原始变量 v，如果 REBUILD(v)严格小于当前的 v，则更新 v。

图 3 展示了将 XORRePair 应用于 P_0 的过程。

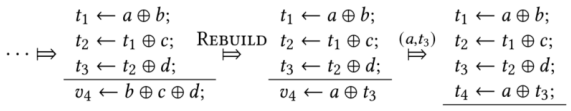


图 3 将 XORRePair 算法应用于 P_0

首先得到图 3 左边的形式，REBUILD(v_4)使 v_4 的缩短了，所以接下来将 v_4 更新为 $v_4 \leftarrow a \oplus t_3$。最后，对（a,$t_3$）生成了具有四个 XOR 最短的 $SLP_{\oplus}$。显然，XORRePair 以多项式时间运行。

2.5 减少内存访问

通过使用函数程序优化方法 Deforestation 来减少 SLP 的内存访问。利用 SLP 的简单性，可以很容易地利用 Deforestation 转换和优化 SLP，从而减少 SLP 地内存访问量。

在这项工作中提出了 XOR 融合，利用融合程序 XOR_4 取代((a xor b) xor c)xor d（图 4 所示），减少内存访问。

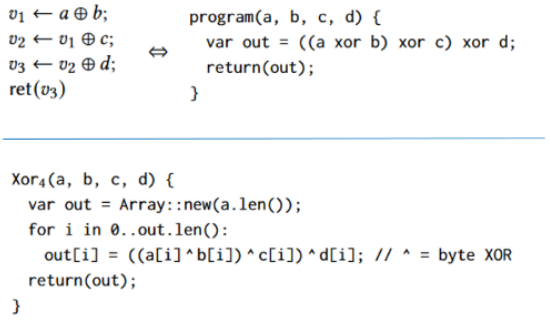


图 4 融合程序 XOR_4 取代((a xor b) xor c) xor d

XOR 融合将给定的 $SLP_{\oplus}$ 转化为 $SLP_{\oplus,}$ 来减少对内存的访问，如图 5 所示。由此产生一个疑问：是否可以多次展开变量？答案是否定的。在融合且没有压缩的情况下 SLP 可以快速运行，但是由于压缩引入了额外的变量，这可能会对缓存带来可怕的影响。

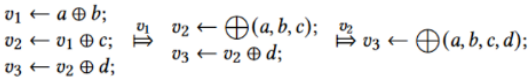


图 5XOR 融合减少 SLP 的内存访问量

2.6 通过卵石博弈优化SLP

使用经典的程序分析工具乱世博弈使一个给定的 SLP 对缓存友好。这里我们使用一个有向无环图（DAGs）来表示 SLPs 的价值依赖性，如图 6 所示。

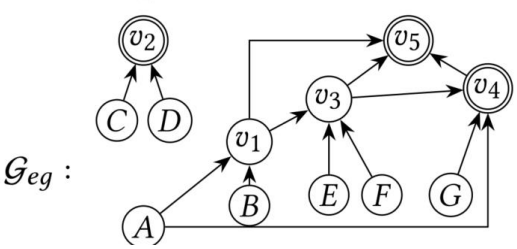


图 6 使用有向无环图表示 SLPs 的价值依赖性

每个叶节点(没有子节点)表示同名的常量。每个内部节点(有子节点)表示通过 XORing 所有子节点获得的值;因此，内部节点 v1 意味着 A⊕B, v3 意味着 A⊕B⊕E⊕F，以此类推。计算图(CGs)是带有双圈节点(目标节点)的 DAGs，目标节点表示程序返回的值。

使用卵石博弈不仅对我们的缓存优化问题的具体化有价值，而且对正确参考编译器构造和程序

分析的既定结果也有价值。

2.7 实验结果

涉及的实验在以下环境下进行：i7-7567U、DDR3-2133 16GB；Ryzen 2600、DDR4-2666 48GB。这些 CPU 的缓存规格是相同的：L1 缓存大小为 32KB/核，L1 缓存的关联性为 8，缓存行大小为 64 字节。

减少的运算符：评估了 SLP 压缩启发式，RePair 和 XorRePair。图 7 显示了(Xor)RePair 在 RS(10, 4)的 1002 个 SLPs 中的平均性能。第一个和第二个比率是通过启发式方法减少 XOR 的平均比率，比率越小，则压缩性能越好。尽管 RePair 很简单，而且最初是在语法压缩中开发的，但我们可以看到它的效果非常好。XorRePair 虽然利用了 XOR 的取消属性，但是差别很小。尽管 RePair 没有使用 XOR 的取消性，但它仍然有效地压缩了程序，这来自于 RePair 的鲁棒性。

Avg%	$\frac{\text{RePair}(P)}{P}$	$\frac{\text{XorRePair}(P)}{P}$	Corresp. Value from [104]
XOR Num. $\#_{\oplus}(\cdot)$	42.1%	40.8%	~65.0%

图 7 减少的运算符比率

减少的内存访问：XORRePair 和 XOR 融合减少的内存访问结果如图 8 所示。CO 表示 XORRePair，FU 表示 XOR 融合。第二个比率表示，对于未压缩的 SLP，XOR 融合平均减少了~65%的内存访问。我们可以以同样的方式看到其他列。因此，我们可以看出，XORRePair 和 XOR 融合独立工作得很好；此外，将它们结合起来，平均减少了 76%的内存访问量。

Avg%	$\frac{\text{Co}(P)}{P}$	$\frac{\text{Fu}(P)}{P}$	$\frac{\text{Fu}(\text{Co}(P))}{\text{Co}(P)}$	$\frac{\text{Fu}(\text{Co}(P))}{P}$
$\#_M(\cdot)$	40.8%	35.1%	59.2%	24.1%

图 8 减少的内存访问比率

减少的变量和所需缓存大小：考虑 XOR 融合和的 DFS 启发式调度是如何平均影响缓存效率 NVAR 和 CCap 的两个指标的。结果如图 9 所示。第一个比率说明 XORRePair 明显降低了缓存效率。对比第三和第四个比率，我们可以说调度启发式肯定会提高缓存效率。将第一和第四个比率相乘，结果为~199%，因此可以说调度启发式可以在一定程度上抑制 XORRepair 的副作用。

Avg%	$\frac{\text{Co}(P)}{P}$	$\frac{\text{Fu}(P)}{P}$	$\frac{\text{Fu}(\text{Co}(P))}{\text{Co}(P)}$	$\frac{\text{Dfs}(\text{Fu}(\text{Co}(P)))}{\text{Co}(P)}$
NVar	1552%	100%	38.9%	24.5%
CCap	498%	98.7%	51.2%	40.0%

图 9 减少的变量和所需缓存大小比率

吞吐量比较：将编码 SLP 定义为 Penc，解码 SLP 定义为 Pdec。将 Penc 和 Pdec 的完全优化版本于 ISA-L v2.30.0 进行比较。比较了 RS(d,4)在英特尔上的编码吞吐量，其中我们使用 B = 1K 作为块大小。对比结果如图 10 所示。结果表明，该工作的 EC 库在编码方面超过了 ISA-L。

intel 1K	Ours		ISA-L v2.30	
(GB/sec)	Enc	Dec	Enc	Dec
RS(8, 4)	8.86	6.78	7.18	7.04
RS(9, 4)	8.83	6.71	6.91	6.58
RS(10, 4)	8.92	6.67	6.79	4.88

图 10 RS(d,4)于 ISA-L 吞吐量比较

2.8 结论

提出了一个精简的方法来实现一个高效的基于 XOR 的 EC 库。我们结合了四个概念：来自程序优化的直线程序（SLPs）、语法压缩算法 RePair、函数式程序优化技术 deforestation 和程序分析的卵石博弈。

将 RePair 扩展到 XORRePair，以适应 XOR 的取消性属性。我们用卵石博弈来正式确定在 SLP 上的抽象 LRU 缓存的优化问题。

通过这些方法的正交组合，实现了一个实验库，其性能超过了英特尔的高性能库 ISA-L。

3 利用组合局部定位的大条带纠删码

3.1 背景

当前最先进的存储系统往往使用中等大小条带的纠删码--为了能够进一步降低冗余度，该工作中提出了使用大条带纠删码的方式。但是，大条带的使用带来了三个性能挑战。

修复：纠删码会导致修复损失，对于大条带来说，这种损失甚至更严重。对于(n,k) RS 代码，修复单个丢失块的常规方法是从其他非故障节点检索 k 个可用块，这意味着带宽和 I/O 成本放大了 k 倍。

编码：随着 k 的增加，纠删码的(每条)编码开销变得更加突出(解码的参数也一样)。在(n,k) RS 码

中，每个奇偶校验块是 k 个数据块的线性组合，因此计算开销随着 k 的增加而线性增加。最重要的是，随着 k 的增加，编码过程将更难以将大条带的输入数据适合 CPU 缓存，导致编码性能显著下降。图 11 显示了使用 Intel ISA-L 编码 APIs 在三个 Intel CPU 系列上对 k 的编码吞吐量。在这里，修复了一个 64MiB 和 n- k = 4 的块大小。从 k = 4 到 k = 16，编码吞吐量仍然很高，但从 k = 32 开始，随着 k 的进一步增加，编码吞吐量急剧下降;例如，从 k = 4 到 k = 128，吞吐量下降了 43-70%。

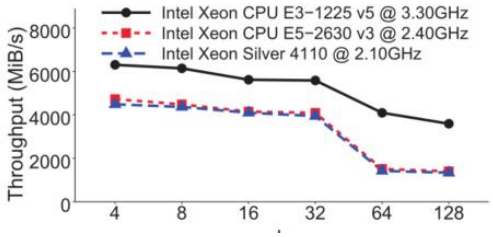


图 11 使用 ISA-L 编码 APIs 在三个 Intel CPU 上对 k 的编码吞吐量

更新：(每个条带)纠删码的更新开销非常大:如果同一条带的任何数据块已经更新，那么所有 n- k 个奇偶校验块都需要更新。

3.2 奇偶局部定位

奇偶局部定位添加本地奇偶校验块以减少用于修复丢失块的幸存块的数量(以及修复带宽和 I/O)。给定三个可配置参数 n,k 和 r(其中 r < k < n)，一个(n,k,r) Azure-LRC 将每个 r 个数据块的本地组(除了最后一组，它可能有少于 r 个数据块)编码成一个本地奇偶校验块，因此一个丢失块的修复现在只访问 r 个幸存块(r < k)。

图 2(a)显示了(32,20,2)Azure-LRC。它有 20 个数据块(用 D_1,D_2，…，D_{20} 表示)。它有 10 个本地奇偶校验块，其中第 l 个本地奇偶校验块 P_l[i- j]是数据块 D_i,D_{i+1}，…D_j 的线性组合。它还有 2 个全局校验块 Q_1[1-20]和 Q_2[1-20]，每个都是所有 20 个数据块的线性组合。所有上述 32 个块都放置在 32 个节点中，以容忍任何 3 个节点故障。因此，(32,20,2)AzureLRC 的单块修复带宽为两个块(例如，修复 D_1 需要访问 D_2 和 P_1[1-2])，同时产生 1.6 倍的冗余。相比之下，(23,20)RS 代码也有 20 个数据块，并且可以容忍任何三个节点故障。其单块修复带宽为 20 块，但冗余度仅为 1.15 倍。奇偶局部定位降低了修复带宽，但导致了高冗余。

3.3 拓扑局部定位

现有的纠删码存储系统(包括 Azure-LRC)将条带的每个块放在位于不同机架的不同节点中。这为相同数量的节点故障和机架故障提供了容忍度，但修复会占用大量跨机架带宽，这通常比机架内部带宽更受限制。

最近的研究利用拓扑局部定位来降低跨机架修复带宽，方法是将机架内的修复操作本地化，但代价是降低了机架级的容错能力。它们将条带的块存储在机架内的多个节点中，并将修复操作分为机架内和跨机架修复子操作。跨机架修复带宽可证明是最小化的，受最小冗余。一些类似的研究关注于通过簇内修复子操作最小化跨簇修复带宽。我们将拓扑局部定位方案定义为(n,k,z) TL，其中(n,k)个 RS 编码的块放置在 z 个机架(或集群)中。

图 12(b)显示了(23,20,8)TL，它将 20 个数据块和 3 个 RS 编码的奇偶校验块放置在位于 8 个机架的 23 个节点中，以容忍任何 3 个节点故障和 1 个机架故障。(23,20,8) TL 的最小冗余为 1.15 倍，但传输 7 个跨机架块来修复丢失的块。例如，修复 D_1 需要从其他机架中检索 Q_1[1-20]和 6 个为 Q_1[1-20]线性部分的块，通过抵消线性部分、D_2 和 D_3，从 Q_1[1-20]求解 D_1。单块修复带宽高于(32,20,2)Azure-LRC(即两个块)。拓扑局部定位实现了最小的冗余，但会产生很高的跨机架修复带宽。

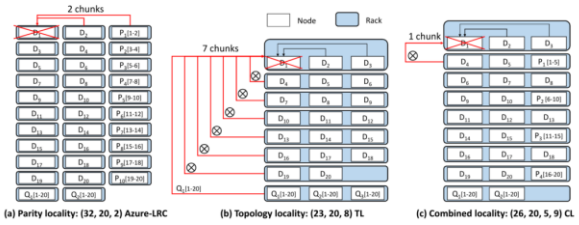


图 12 三种局部定位方案例子，每一个存储 20 个数据块并且可以容忍 3 个丢失块

3.4 组合局部定位

本项工作提出了组合局部定位，利用奇偶局部定位和拓扑局部定位的组合来减少跨机架修复带宽，以限制大条带纠删码造成的冗余。

将组合的局部机制定义为(n,k,r,z) CL，它将跨 z 个机架的(n,k,r) Azure-LRC 和(n,k,z) TL 组合在一起。组合局部机制的主要目标是确定参数(n,k,r,z)，以最小化跨机架修复带宽。组合局部定位的符号意义如图 13 所示。

Notation	Description
n	total number of chunks of a stripe
k	number of data chunks of a stripe
r	number of retrieved chunks to repair a lost chunk
z	number of racks to store a stripe
c	number of chunks of a stripe in a rack
f	number of tolerable node failures of a stripe
γ	maximum allowed redundancy

图 13 组合局部定位中的符号意义

我们专注于优化两种类型的修复操作:单块修复和全节点修复。这两种修复操作都假设每个丢失的条带恰好有一个丢失的块。对于具有多个丢失块的丢失条条带，我们采用传统的修复方法，检索 k 个可用块，以重建 RS 代码中的所有丢失块。

为了实现组合局部定位的目标，我们从图 12 中观察到，组合局部定位通过下载 r- 1 个数据块加上一个本地奇偶校验块(即修复带宽为 r 个块)来修复一个数据块。由于组合局部岗位将 r 个块中的一部分放置在相同的机架中，因此可以对每个机架中的块进行局部修复，从而减少跨机架的修复带宽。直观地说，如果 c 增加(即条带的更多块可以驻留在一个机架中)，本地修复可以包括更多的块，从而进一步降低跨机架修复带宽。因此，我们的目标是找到最大的可能 c。由此，可以得到(n,k,r,z) CL 的构造是为了保证 c = f。然而，有不同结构的(n,k,r) LRC 提供不同级别的容错 f。因此，我们的想法是选择合适的 LRC 结构，具有最高的容错，其结果是选择 Azure-LRC。

3.5 (n,k,r,z)CL的构造

提供(n,k,r,z) CL 的构造如下。在这里，我们专注于一个有 k 个数据块的条带，在最大允许冗余度 γ 的前提下，有固定数量的可容忍节点故障 f。该结构包括两个步骤。(i)找到(n,k,r)Azure-LRC 的参数，(ii)将所有 n 个数据块放在 z 个机架上进行本地修复操作。

3.6 ECWide

我们设计了一种实现组合局部性的大条带纠删码存储系统 ECWide。ECWide 解决了在大条带纠删码中实现高效修复、编码和更新的挑战，其目标如下:

•最小跨机架修复带宽:ECWide 通过组合局部定位最小化跨机架修复带宽。

•高效编码:ECWide 应用多节点编码，支持大条纹的高效编码。

•高效的奇偶校验更新:ECWide 应用内部机架奇偶校验更新，允许全局和本地奇偶校验块主要在本地机架内更新。

在三个主要模块中实现了 ECWide:一个修复模块，根据组合局部定位执行修复操作，一个编码模块，执行多节点编码，以及一个更新模块，执行机架内奇偶校验更新。分别为冷热存储系统实现了 ECWide 的两个原型，即 ECWide-C 和 ECWide- H，如图 14 所示。

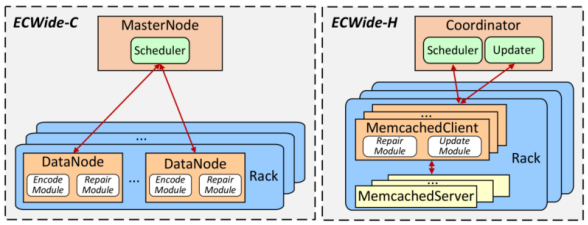


图 14 ECWide 的结构

ECWide-C 主要用 Java 实现，大约有 1500 个 SLoC，而编码模块是用 C++实现的，大约有 300 个 SLoC，基于 Intel ISA-L。它有一个 MasterNode，用于存储元数据并使用 Scheduler 守护进程组织修复和编码操作，还有多个 Datanode，用于存储数据并执行修复和多节点编码操作。注意 ECWide-C 不考虑更新模块，假设冷存储中很少有更新。

对于修复操作，Scheduler 触发每个 DataNode 的修复模块，该模块充当本地修复器。这样的 DataNode(充当本地修复者)将部分修复的结果发送到充当请求者的 DataNode，后者最终重建丢失的块。对于编码操作，Scheduler 选择一个机架并触发机架中每个相关 DataNode 的编码模块。涉及的 DataNode 执行多节点编码，作为目标节点的 DataNode 生成所有全局校验块。

ECWide-H 构建在 Memcached 内存键值存储(v1.4)和 libMemcached (v1.0.18)的基础上，用于热存储。它是用 C 语言实现的，大约有 3000 个 SLoC。它遵循客户机-服务器体系结构。它包含存储键值项的 MemcachedServers，以及执行修复和奇偶校验更新操作的 MemcachedClients。它还包括用于管理元数据的 Coordinator。Coordinator 包括协调修复和校验更新操作的 Scheduler 守护进程和分析更新频率状态的 Updater 守护进程。ECWide-H 不像 ECWide-C 那样包含编码模块，因为在纠删码的内存键值存储中的块大小通常很小(例如 4 KiB)，并且单节点 CPU 缓存足够大，可以预取大条带的所有

数据块以获得高编码性能。

对于修复操作，ECWide-H 执行的方式与 ECWide-C 相同，只是它使用 MemcachedClients 作为本地修复器。对于全局校验块的更新，Scheduler 定位全局校验块所在的机架，并触发 MemcachedClients 的更新模块来执行机架内部的校验更新。对于本地校验块的更新，Updater 首先触发交换，其中涉及的两个 MemcachedClients 交换相应的块。稍后可以执行本地校验块的机架内校验更新。

3.7 评估

本章节给出了 ECWide-C 组合局部定位(CL)的实验结果，并与 AzureLRC (LRC)和拓扑局部定位(TL)进行了比较，它们代表了目前最先进的基于局部定位的方案。

使用 ECWide-C 评估 LRC、TL 和 CL 的修复性能，结果如图 15 所示。图 15 显示了不同 f 值下 LRC、TL 和 CL 的平均全节点修复率;我们还比较了使用和不使用最近最少选择(LRS)方法的 CL。结果显示，在不同的 k、f、网络带宽下，CL 的平均修复时间都远远小于 TL 和 LRC。

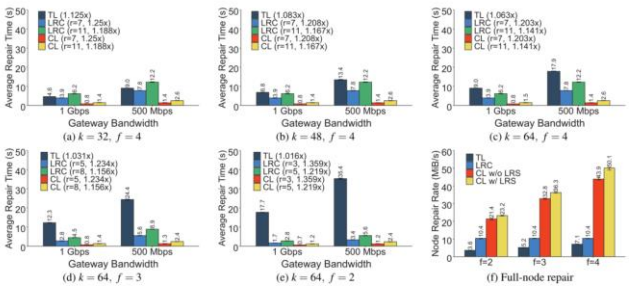


图 15 ECWide-C 修复性能对比结果

为了评估 ECWide-C 的编码时间，我们固定 k=64，f=4，让 r 在 11 到 19 之间，图 16 显示了单节点编码和多节点编码的结果。多节点编码的编码时间明显低于单节点编码。例如，当 r = 11 时，与单节点编码相比，多节点编码减少了 84%的编码时间。

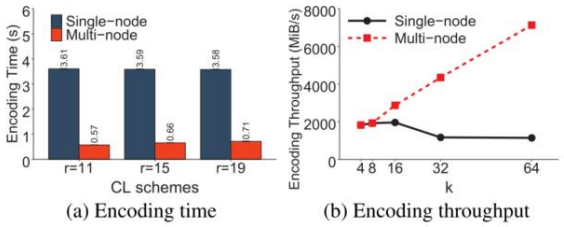


图 16 单节点编码和多节点编码下的编码时间和编码吞吐量

3.8 总结

大条纹是纠删码分布式存储的一个新概念，可以实现极大的存储节省。提出了组合局部定位，这是一种新型的修复机制，它结合了奇偶局部定位和拓扑局部定位，以有效解决大条带的修复问题。设计了 ECWide，一个实现组合局部定位的原型系统。我们进一步设计了多节点编码和机架内奇偶性更新，以分别提高编码和更新性能。为冷、热存储系统实现了 ECWide，在 Amazon EC2 的实验证明了 ECWide 在修复、编码和更新方面的效率。

4 LogECMem ：将纠删码的内存键值存储与奇偶校验日志相结合

4.1 背景

纠删码存储中的校验更新会引起大量的网络传输，因为它们需要重新计算校验块以保持一致性。重新检查现有的奇偶校验更新方案，将这些方案分为四类:直接重建、就地更新、全条带更新和奇偶校验日志记录。

直接重建：一个直接的方法是首先读取所有没有参与更新的数据块，然后用读取的数据块和新的数据块来重建新的奇偶性数据块，这显然要花费大量的数据块传输。

就地更新：就地更新读取旧的奇偶校验块，通过*Delta* 来计算 *Parity delta*，通过*Parity delta* 生成新的奇偶校验块，并替换掉旧的奇偶校验 块。但是，就地更新会产生额外的奇偶校验块的读取，这会影响更新的性能。

全条带更新：为了消除校验块的读取，全条带更新直接将新的数据块编码为新的条带，并将旧的数据块标记为无效，以便通过垃圾收集(GC)直接释放。已应用于实际系统，如 QFS、BCStore、Giza 等。但是，全条带更新可能会导致存储过期数据块

的高存储开销，并且需要在 GC 期间对剩余的活动的未更改数据块重新计算奇偶校验

奇偶校验日志记录：奇偶校验日志(PL)通过在日志设备中记录奇偶校验增量来改进就地更新，从而减少更新期间旧奇偶校验块的读取开销，同时保持比全条带更新更低的存储开销。

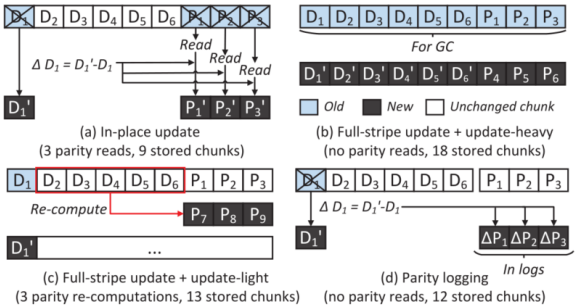


图 17 不同更新方案展示

在图 17 中展示了全条带更新的两个观察结果:对于更新量大的工作负载，内存开销很高;对于更新量小的工作负载，剩余活动数据块的重新计算需要大量的块传输。

4.2 HybridPL 架构

基于上一节的观察结果，我们的主要目标是解决如何将内存 KV 存储与奇偶校验日志连接起来以实现高修复和更新性能的挑战。提出的 HybridPL 主要实现以下目标:

1）低内存开销:HybridPL 通过就地更新数据块来减轻内存开销。

2）高效更新:HybridPL 对非 XOR 奇偶校验块执行奇偶日志记录，并利用日志节点中的奇偶校验增量缓冲日志记录来加速更新。

3）高效的单故障修复:HybridPL 在 DRAM 节点中保持 XOR 奇偶校验块，以确保降级的读取效率。

Hybrid PL 通过对数据和 XOR 奇偶校验快的就地更新实现目标 1 和目标 3：就地更新不会产生额外的内存开销，而全条带更新则会，因为后者在内存中保留了多个旧版本的数据块，特别是对于更新量大的工作负载。因此，与全条带更新相比，HybridPL 可以通过部署数据块的就地更新来减少内存开销；修复单节点故障的节点的数据传输率可靠性有很大的影响，也就是说，我们只能在 DRAM 中保留 XOR 奇偶校验块以及数据块来获得高性能的单故障修复。具体来说，当一个请求的数据块由于单次故障不能被直接读取时，HybridPL 会检索剩余的 k-1 个数据和同一条带的 XOR 奇偶校验块来解码 DRAM 节点内的请求数据块，这就保证了单次故障修复的高性能。

HybridPL 通过非 XOR 奇偶校验块的奇偶校验记录实现目标 2：HybridPL 通过奇偶校验日志更新奇偶校验块(除了 XOR)，这样 HybridPL 只需要将奇偶校验增量存储到日志设备中就可以更新奇偶校验块，显然，HybridPL 既不读取旧的校验块以更新校验块，也不检索活动的未更改数据块以重新计算校验块，如全条带更新；使用奇偶校验日志，日志节点的低传输速率将成为与 DRAM 节点相比降低写(更新)性能的瓶颈。为了解决这个问题，我们引入缓冲区日志记录方法，它为 DRMA 和磁盘分发副本。RAMCloud 将数据存储在主服务器的 DRMA 中，副本存储在备份服务器的磁盘上，如图 18(a)所示。在写操作时，主服务器将数据写入 DRMA，并将数据副本转发给所有备份服务器。一旦副本被写入备份服务器的 DRAM, RAMCloud 就认为写操作已经完成，在备份服务器中，这些 DRMA 中的磁盘副本可以异步地刷新到磁盘。缓冲区日志记录也可以应用于 HybridPL，如图 18(b)所示。HybridPL 可以执行快速写(更新)操作，因为它们可以在奇偶校验增量存储到日志节点的 DRMA 时立即完成，并且 DRMA 中的奇偶校验增量将异步批量刷新到磁盘。

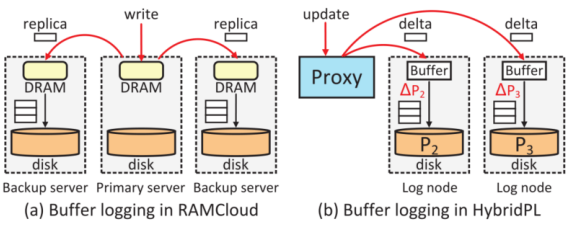


图 18 RAMClound 和 HybridPL 中的缓冲区日志记录

4.3 LogECMem设计

LogECMem 支持四种基本请求:写、读、降级读和删除，其中每个对象的键和值都是任意字符串。

更新操作将现有对象的旧值更改为新值，其中我们还需要更新包含该对象的条带的奇偶校验块。

要更新一个对象，代理首先从对象索引中获取该对象的 Stripe ID、序列号、偏移量和长度。然后，代理通过读操作检索对象的旧值和条带的 XOR 校

验块。代理首先从旧数据块和新数据块中计算增量，使用增量和相应的系数(由序列号决定)计算 XOR 校验块的校验增量，然后通过将旧校验块与校验增量合并来计算新的 XOR 校验块。代理还用相应的系数(由序列号决定)将增量发送到每个日志节点，每个日志节点以类似的方式计算其奇偶增量。最后，LogECMem 将新对象和 XOR 奇偶校验块写入 DRAM 节点，并将带有偏移和长度的奇偶校验增量写入日志节点。

基于 HybridPL，LogECMem 将数据和 XOR 校验块作为对象存储在内存 KV 存储中，而将其他校验块和校验增量作为文件存储在日志节点中。在这里，LogECMem 为每个奇偶校验块及其对应的奇偶校验增量创建一个日志文件，其文件名通过其 Stripe ID 生成。LogECMem 通过合并多个奇偶校验增量(称为基于合并的缓冲区日志记录)来改进 HybridPL 的缓冲区日志记录方法，将它们减少到一个块中。通过这种方式，LogECMem 可以减少日志节点更新期间的磁盘 IOs。

4.4 评估

比较了 5 路(或 4 路)复制、IPMem、FSMem 和 LogECMem 与 PLM 的更新延迟。考虑不同的(k，r)编码和读取/更新比率。图 19 显示 LogECMem 优于 IPMem，因为前者在更新期间减少了从 r 到 1 的奇偶校验读取数量，因为它只需要将 XOR 奇偶校验块读回来进行原地更新。此外，我们看到 LogECMem 在轻度更新场景下，在(6,3)，(10,4)，(12,4)和(15,3)编码中分别比 FSMem 高出 58.0%，42.4%，37.8%和 37.3%。原因是在轻度更新场景下，FSMem 中大多数更新的条带可能只有一个新的数据块，从而导致沉重的重新计算开销。在重度更新场景下，LogECMem 是一个很好的选择，但会产生严重的内存开销。

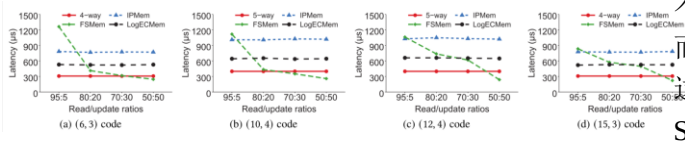


图 19 不同(k,r)编码下更新延迟和读/更新率对比

在内存开销方面，图 20 显示，与 5 路复制相比，LogECMem 减少了高达 79.3%的内存开销（在（12，4）编码中），因为前者为 12 个数据块存储 4 个奇偶校验块，而后者为每个对象存储 4 个副本；与 IPMem 相比，高达 22.2%（在（6，3）编码中），因为它只在 DRAM 节点中维护 XOR 奇偶校验块，而将剩余奇偶校验块和校验增量存储在磁盘中。此外，与 FSMem 相比，LogECMem 减少了高达 49.0%的内存开销（在（6，3）编码中时），因为 FSMem 另外在 DRAM 中保留了多个版本的旧数据和奇偶校验块。我们还看到，当更新率增加时，LogECMem 的性能优于 IPMem 和 FSMem。

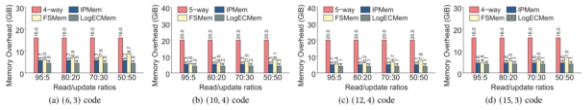


图 20 不同(k,r)编码的内存开销和读/更新比率对比

4.5 总结

提出了 HybridPL，一种基于奇偶校验日志的新型架构，对 DRAM 中的数据和 XOR 奇偶校验块采取就地更新，对日志中的剩余奇偶校验块采取基于日志的更新，这样就可以很好地平衡更新和单次故障修复的内存成本和性能。我们将 HybridPL 作为 Memcached 上的 LogECMem 的原型来提高多故障修复性能。基于云的实验证明了 LogECMem 在基本 I/O、更新和修复方面的效率，而且内存开销很低。

5 INEC：快速且连贯的网络内纠删码

5.1 背景

当前已经提出了许多商品智能网卡，支持网卡上的 EC。然而，目前这一代智能网卡的 API 以不连贯的方式展示了 EC 和网络功能，没有意识到 EC 计算与网络密切相关的事实。为了充分利用 SmartNIC 的能力，减少 CPU 的参与以进一步缩短延迟，一种理想的设计方法是连贯的 EC 计算和网络，或连贯的网络内 EC。连贯的网络内 EC 不是为不连贯地执行 EC 和网络操作而暴露单独的 API，而是提供联合 API 来连贯地执行 EC 计算和网络。这些连贯的 API 可以将一个任务集卸载到 SmartNIC 上，SmartNIC 将负责等待接收完成、执行 EC 计算、等待 EC 完成以及发送结果。如图 21 所示，连贯的网络内 EC 中，每个任务的完成都会自动激活后续任务的执行，无需 CPU 参与。

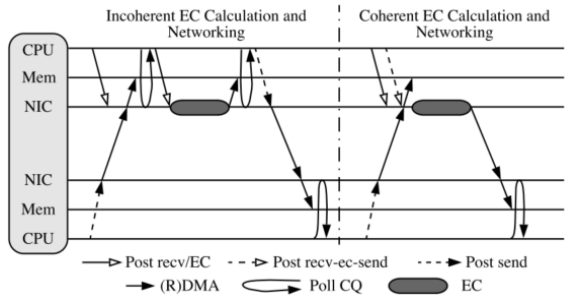


图 21 连贯网络内 EC 与不连贯网络内 EC 对比

5.2 RDMA WAIT

RDMA W AIT 是一个网络级的操作，它可以在两个通信通道上实现工作请求（WR）的触发。如图 22 所示，WAIT WR 在同一个发送队列（SQ）中保持对后续预发 WR（即 Op）的执行，直到完成队列（CQ）中达到预先指定的完成数量。请注意，触发和激活操作是由 RNIC 进行的，没有 CPU 参与。

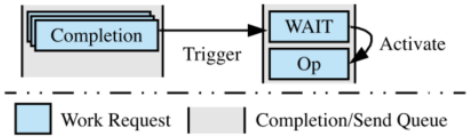


图 22 RDMA WAIT 过程

5.3 INEC原语设计

经过探索，抽象出三类连贯的网络内 EC 原语，它们与网络功能（如 RDMA SEND 和 RECV）一起，足以表达五种最先进的 EC 方案。

ec/xor-send：在几乎所有最先进的 EC 方案中，都有一些节点负责执行 EC/XOR 计算，并将输出发送到集群中的其他节点。将这一过程抽象为 ec/xor-send，这使得上层应用可以将这一过程卸载到 SmartNIC，并从连贯的网络内 EC 中获益。

recv-ec/xor-send：许多 EC 方案将编码和（或）解码计算分解成子问题，并将它们安排在参与计算的不同节点上，以获得更好的并行性、重叠性和对可用资源的充分使用。因此，这些 EC 方案具有复杂而明确的结构。这些结构的内部节点有着相同的过程模式。这些内部节点等待来自上游的结果，对本地块和收到的结果进行计算，并将输出发送到下游。这个过程被抽象为 recv-ec/xor-send。

recv-ec/xor：另一个常见的过程模式为，EC 方案的第一层接收来自远程对等体的数据块，并对收到的数据块进行 EC 计算以重建数据块。我们使用 recv-ec/xor 原语来抽象这个过程。

5.4 INEC原语在RNIC上的实现和优化

如前所述，RDMA WAIT 是 RNIC 上的一个事件驱动机制，支持工作请求（WR）的触发。在本节中，我们将解释关于用 RDMA WAIT 实现 INEC 基元的细节。

正如图 23(a)所阐明的，发布 ec/xor-send 实际上是向计算器的发送队列（SQ）发布一个 EC/XOR 工作请求（WR），以及向每个 SQ 发布一个 WAIT WR，然后是 SEND WR，用于发送块。向同一个 SQ 发送多个 WR 是通过一次发布的，这样可以减少 PCIe 交易和 CPU 的使用。在完成发布的 EC/XOR WR 后，SQ 前面的 WAIT WR 被触发，因此随后的 SEND WR 被激活并由 RNIC 执行，而不涉及主机 CPU。

图 23(b)说明了 recv-ec/xor-send 的细节。与 ec/xor-send 不同，recv-ec/xor-send 也会向计算器的 SQ 发布一些 WAIT WRs，WAIT WRs 的数量等于调用者指定的 CQ 的数量。这些 WAIT WRs 最后将由接收完成逐一触发，最后一个 WAIT WR 将激活 EC/XOR 计算。EC/XOR WR 的完成将触发所有 SQ 前面的 WAIT WRs，因此 SEND WRs 被激活，结果被发送出去。在我们的实现中，WAIT WRs 和 EC/XOR WR 到计算器的 SQ 是由同一个 PCIe 事务发布的，这进一步减少了 CPU 的参与和延迟。

对于 recv-ec/xor 原语，如图 23(c)所示，几个 WAIT WR 和一个 EC/XOR WR 被一个帖子发布到计算器的 SQ 中，这些 WAIT WR 最后将由被等待的 CQ 的接收完成来触发。最后，EC/XOR WR 被它前面的最后一个 WAIT WR 激活。

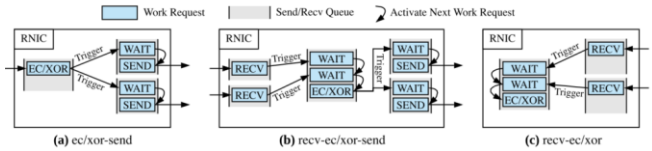


图 23 INEC 原语在 RNIC 上的实现

5.5 评估

在本节评估了用所提出的 INEC 原语实现的五个最先进的 EC 方案，以说明 INEC 所带来的性能优势。我们选择了一个自定义的基准评估 INEC 原语的性能。本节实验的 baseline 是指相同的实现，但采用的是不连贯的网络内 EC 方法。

自定义的基准包括一个延迟基准测试和一个带宽基准测试。延迟基准由两部分组成：编码基准和解码基准。带宽基准测试的设计类似于传统的基于窗口的消息传递基准测试，用于网络带宽测量。

图 24 展示了 RS、LRC、TriEC 三种 EC 方案在不同块大小的情况下使用 INEC 原语的编码延迟性能对比。24(a)和 24(b)表明，ec-send 的整合大大减少了 EC 方案 RS 和 LRC 在中小块大小上的编码延迟，但对大块大小没有多大帮助。因为 RS 和 LRC 并不会减少 DMA 流量，因此编码延迟与 baseline 相当。24(c)说明 INEC 原语为 TriEC 提供了小、中、甚至大块大小的显著性能优势，因为 TriEC 使用的 recv-ec-send 原语减少了不必要的 DMA 流量。

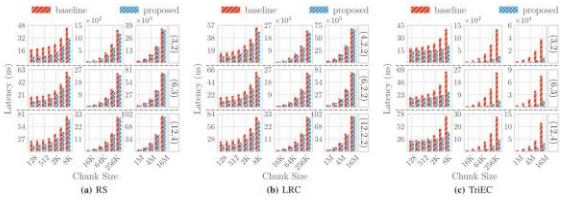


图 24 RS、LRC、TriEC 使用 INEC 原语的编码延迟结果

图 25 表明，INEC 能够将 RS(6,3)的带宽提高 2.71 倍，LRC(6,2,2)的带宽提高 2.63 倍，TriEC(6,3)的带宽提高 5.87 倍。使用 INEC 原语的 TriEC 带宽性能最好。

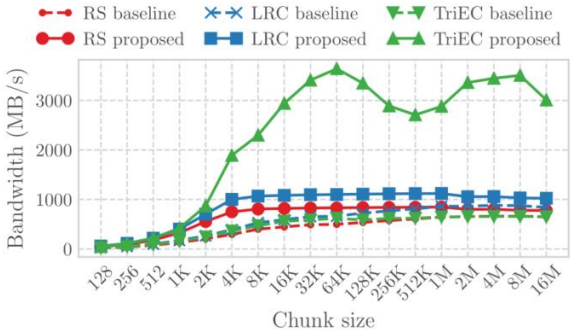


图 25 RS、LRC、TriEC 使用 INEC 原语的编码带宽结果

图 26(a)(b)表明，除了小块大小的情况外，recv-ec 的解码性能明显更好。在小块大小的情况下，CPU 使用率保持相对较低，CPU 的参与不是不连贯基准测试的性能瓶颈；而 recv-ec 引入的额外网络内状态降低了其延迟性能。26(d)的结果表明，所有评估的数据块大小，使用 INEC 可以加快 ECPipe 的速度。

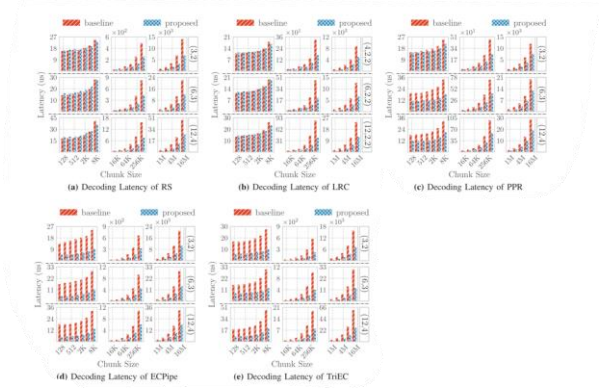


图 26 五种 EC 方案使用 INEC 原语的解码延迟结果

图 27 表明，使用 INEC 的五种 EC 方案在解码带宽上的性能都优于 baseline，其中使用 INEC 的 LRC 获得了最好的解码带宽性能。

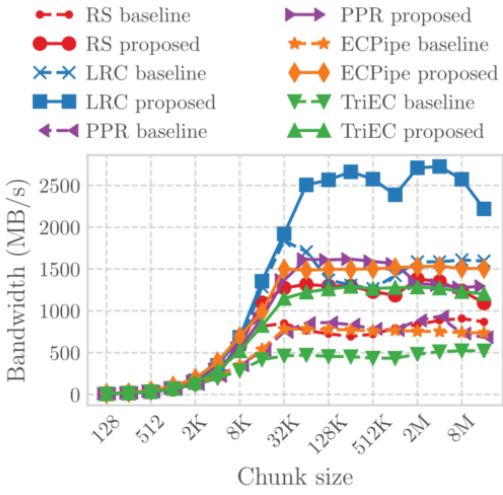


图 27 五种 EC 方案使用 INEC 原语的解码带宽结果

5.6 结论

提出了一套连贯的网络内原语——INEC，使最先进的 EC 方案能够充分利用现代 SmartNIC 上的 EC 卸载能力，INEC 的性能增益在五个最先进的 EC 方案中得到了验证，使用 INEC 的 TriEC 在编码带宽方面明显优于其他 EC 方案，使用 INEC 的 LRC 获得了最佳的解码带宽性能。

6 NetEC：加速纠删码重构与网络内聚合

6.1 背景

将 EC 卸载到可编程交换 ASIC 中可以解决重

构时间长、降级读取延迟高和大量资源使用的问题。首先，在可编程交换机上，数据流到达不同的接口，聚合并转发到另一个接口。因此，没有共享带宽。其次，与数据复制类似，重构数据只有一个数据流通过出接口发送，避免了 incast。最后，网络内计算完全消除了 RS 解码的额外 CPU 占用。存储节点接收已重构的数据，可直接写入硬盘。

因此提出了 NetEC，一个网络内加速框架，将 EC 完全卸载到可编程交换 ASIC 上。

6.2 NetEC组件

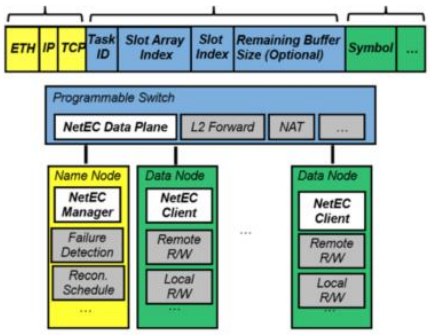
NetEC 由以下三个系统组件和 NetEC 数据包格式组成（图 28）。

NetEC 数据平面。NetEC 数据平面是负责 RS 解码的交换 ASIC 程序。它对符号进行 GF 乘法运算，并对同一分组条带的符号进行聚合。它还实现了明确的缓冲区大小通知，通过明确通知 NetEC 客户的剩余缓冲区大小来防止缓冲区溢出。一对多的 TCP 代理被部署在我们的 HDFS 实现中。

NetEC 管理器。NetEC 管理器管理着 NetEC 重建过程的生命周期。在我们的 HDFS 实现中，NetEC 管理器与 HDFS Namenode 共存，可以访问所有的 RS 解码矩阵。当一个区块的重建被安排好后，NetEC 管理器就会接管，并用它自己的逻辑来管理重建工作。无论数据是用 NetEC 还是本地 EC 进行重建，对文件系统的其他模块来说都是透明的。NetEC 管理器还跟踪正在进行的任务和它们使用的交换槽阵列。

NetEC 客户端。NetEC 客户端负责数据传输。NetEC 客户端使用滑动窗口进行速率控制。窗口的大小随着 NetEC 数据平面所发布的剩余缓冲区大小而更新。在我们的 HDFS 实现中，NetEC 客户端与 Datanode 共存，并使用传统的 TCP，由一对多的 TCP 代理启用。

NetEC 数据包格式。NetEC 数据包的格式如图 28 所示。一个 NetEC 数据包包括一个 4 位任务 ID 字段、一个 8 位槽位数组索引字段、一个 16 位槽位索引字段和一个 16 位剩余缓冲区大小字段。一个重建任务使用一个槽位数组，这是一块连续的开关 SRAM。剩余的缓冲区大小字段由 EBSN 机制使用。



(b) NetEC components.

图 28 NetEC 组件

6.3 NetEC实现

在商用硬件上实现了 NetEC 的原型。数据平面组件使用 P4 (457 LoC)实现，并使用 Barefoot Capilano 软件套件进行编译。数据平面的控制器程序用 python (269 LoC)编写。在服务器端，使用 Java (2587 LoC)实现了一组 HDFS-EC 策略。还使用 DPDK (Data Plane Development Kit)在 C (652 LOC)中实现 NetEC 简单版。

6.4 结论

该项工作提出了 NetEC，一个网络内加速框架，完全卸载 EC 到可编程交换 ASIC。

提出了显式缓冲区大小通知(EBSN)来约束解码缓冲区的使用，并设计了一个交换机上的一对多 TCP 代理来集成 EBSN 和 TCP。经过评估表明 NetEC 是有效的，能够支持~GB/s 的重构速率和数十个并发任务。

7 总结

上述五篇文章从软件和硬件方面提出了纠删码性能提升的方法。软件方面的基本思想是优化纠删码计算程序以减少 CPU 的开销和内存的访问，而硬件方面的基本思想则是将 EC 卸载到网络设备中。

将 EC 卸载到网络设备上虽然可以减少纠删码性能上的开销并提高吞吐量，但是受限于网络设备的性能，这样的提高相比与软件方面的提升并不是非常显著。因此，在未来的发展方向可以考虑硬件与软件相结合的方法，以此对纠删码性能进行进一步的提高。

参 考 文 献

[1] Yuya Uezato. 2021. Accelerating XOR-based erasure coding using program optimization techniques. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21). Association for Computing Machinery, New York, NY, USA, Article 87, 1–14. https://doi.org/10.1145/3458817.3476204

[2] Hu, Yuchong, Liangfeng Cheng, Qiaori Yao, Patrick PC Lee, Weichun Wang, and Wei Chen. “Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage.” In 19th USENIX Conference on File and Storage Technologies (FAST’21), pp. 233-248. 2021.

[3] Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, Jia Xu, Qiaori Yao, Dan Feng, Weichun Wang, and Wei Chen. 2021. LogECMem: coupling erasure-coded in-memory key-value stores with parity logging. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21). Association for Computing Machinery, New York, NY, USA, Article 89, 1–15. https://doi.org/10.1145/3458817.3480852

[4] H. Shi and X. Lu, "INEC: Fast and Coherent In-Network Erasure Coding," SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 2020, pp. 1-17, doi: 10.1109/SC41405.2020.00070.

[5] Y. Qiao et al., "NetEC: Accelerating Erasure Coding Reconstruction With In-Network Aggregation," in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 10, pp. 2571-2583, 1 Oct. 2022, doi: 10.1109/TPDS.2022.3145836.