计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.20210577
59(3):597-616,2022

# 近数据计算下键值存储中 Compaction 并行优化方法

孙 辉[1] 娄本冬[1] 黄建忠[2] 赵雨虹[3] 符 松[4]
[1](安徽大学计算机科学与技术学院 合肥 230601)
[2](华中科技大学武汉光电国家研究中心 武汉 430074)
[3](中国科学院信息工程研究所 北京 100093)
[4](北德克萨斯大学计算机科学与工程系 美国德克萨斯州登顿 76203)
(sunhui@ahu.edu.cn)

# Near-Data Processing-Based Parallel Compaction Optimization for Key-Value Stores

Sun Hui[1], Lou Bendong[1], Huang Jianzhong[2], Zhao Yuhong[3], and Fu Song[4]
[1](*School of Computer Science and Technology, Anhui University, Hefei* 230601)
[2](*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan* 430074)
[3](*Institute of Information Engineering, Chinese Academy of Sciences, Beijing* 100093)
[4](*Department of Computer Science and Engineering, University of North Texas, Denton, TX, USA* 76203)

**Abstract** Large-scale unstructured data management brings unprecedented challenges to existing relational databases. The log-structured merge tree (LSM-tree) based key-value store has been widely used and plays an essential role in data-intensive applications. The LSM-tree can convert random-write operations into sequential ones, thereby improving write performance. However, the LSM-tree key-value storage system also has some problems. First, the key-value storage system uses compaction operations to update data to balance system performance, but it impacts system performance and causes serious write amplification. Second, the traditional computing-centric data transmission also limits the overall system performance in compaction. This paper applied the data-centric near-data processing (NDP) model in the storage system. We propose a collaborative parallel compaction optimization for LSM-tree key-value stores named CoPro. The two parallel (i.e., data and pipeline parallelism) are fully utilized to improve compaction performance. When the compaction is triggered, the host-side CoPro determines the partitioning ratio of the compaction tasks according to the offloading strategy and divides tasks according to the ratio. Then, compaction subtasks are offloaded to the host and device sides, respectively, through the semantic management module. We design a decision component in the host-side and device-side CoPro, which is remarked as CoPro+. CoPro+ can dynamically adjust the parallelism according to changes in the resource of system and the value of key-value pairs in workloads. Extensive experimental results validate the benefits of CoPro compared with two popular NDP-based key-value stores.

**收稿日期**:2021-06-08;**修回日期**:2021-09-24
**基金项目**:安徽高校协同创新项目(GXXT-2019-007);计算机体系结构国家重点实验室(中国科学院计算技术研究所)开放课题(CARCH201915);国家自然科学基金项目(62072001,61702004,61572209)
This work was supported by the University Synergy Innovation Program of Anhui Province (GXXT-2019-007), the State Key Laboratory of Computer Architecture (ICT, CAS) (CARCH201915), and the National Natural Science Foundation of China (62072001, 61702004, 61572209).
**通信作者**:赵雨虹(zhaoyuhong@iie.ac.cn)

**Key words** log-structured merge tree (LSM-tree); key-value store; near-data processing (NDP); task offloading; data-pipeline parallelism

**摘 要** 大规模非结构化数据的爆炸式增长给传统关系型数据库带来了极大的挑战.基于日志结构合并树(log-structured merge tree, LSM-tree)的键值存储系统已被广泛应用,并起到重要的作用,原因在于基于 LSM-tree 的键值存储能够将随机写转化为顺序写,从而提升性能.然而,LSM-tree 键值存储也存在一些性能问题.一方面,键值存储利用 compaction 操作更新数据,保持系统平衡,但造成严重的写放大问题.另一方面,以传统计算为中心的架构下,compaction 操作带来大量的数据传输,影响了系统性能.以数据为中心的近数据计算模型(near-data processing, NDP)为基础,利用该模型下主机端与近数据计算使能设备端的并行资源,提出基于系统并行与流水线并行的 compaction 优化方法(collaborative parallel compaction optimization for LSM-tree key-value stores, CoPro).当处理 compaction 操作时,CoPro 主机端与 NDP 设备端协同执行 compaction 卸载任务.此外,进一步提出基于决策组件的 CoPro+,根据系统资源变化以及负载键值对中值大小的变化来动态调整并行度,使 NDP 架构中计算资源的使用更加高效.在搭建的硬件平台上验证了 CoPro 的有效性.

**关键词** 日志归并树;键值存储;近数据计算;任务卸载;数据-流水线并行

**中图法分类号** TP391

国际数据公司在报告“Data Age 2025”[1]中预测,从 2018—2025 年,全球数据总量将从 33 ZB 跃升到 175 ZB,其中非结构化数据将占数据总量的 80%以上.传统关系型数据库已无法满足高效组织与管理大规模非结构化数据的需求以及数据库高并发和易扩展性的愿望[2-3],关系型数据库旨在管理结构化数据或与预定义数据类型对齐的数据,从而使其易于分类和搜索,面对海量非结构化数据,更改关系型数据库中的结构可能会非常昂贵且耗时,并且通常会导致停机或服务中断.另一方面,关系型数据库难以进行横向扩展[4],这与关系模型旨在确保一致性有关,要在多台服务器上水平扩展关系型数据库,则很难确保其一致性.

针对传统关系型数据库已不能胜任管理海量非结构化数据的问题[3],人们开始着眼基于日志归并树(log structured merge tree, LSM-tree)的键值存储系统来管理海量非结构化数据,如 Apache 的 HBase[5] 和 Cassandra[6]、Google 的 BigTable[7] 和 LevelDB[8] 以及 Facebook 的 RocksDB[9] 等.LSM-tree 键值存储通过对写入操作进行顺序化实现性能提升,具有高写入性、易扩展等优势.LSM-tree 包括内存组件和硬盘组件 2 部分,其将随机写缓存在内存组件(MemTable)中,待内存组件写满时再转储到硬盘组件上,将随机写转化为批量顺序写.随着数据的持续写入,硬盘组件数量越来越多,读取数据时需要查询的文件数量增多,同时旧的组件中会有很多过期的数据,所以需要整理和合并大量文件,将上层数据转储到下层:一方面是为了减少文件的数量以提高读性能;另一方面则是为了删除过期数据,释放存储空间,这一过程被称为 compaction. compaction 是一个后台线程,会持续将达到预定阈值的上层文件转移到下层.在随机写密集型负载下,compaction 任务会频繁发生,引起数据压缩/解压、拷贝和字符串比较等操作,消耗大量计算资源,同时读写操作也会占用大量硬盘 I/O,引起系统写放大,降低系统性能.

随着存储设备内计算能力的提升,基于近数据计算(near-data processing, NDP)的计算模型重新受到了广泛关注[10],数据处理模式从以传统计算为中心的方式转变为以数据为中心.近数据计算模型的核心是让“计算接近数据”,改变存储设备获取数据后再传输到主机端处理的方式,而是直接在存储设备内计算,主机端接收计算结果[10-11].该模型减少了数据移动开销,如前文提到的 compaction 操作,有效减少主机与存储设备之间的 I/O 负担并提高系统性能.面对基于 LSM-tree 的键值存储中存在 compaction 的问题,在近数据计算模型下,实现 compaction 的多级并行优化方法(CoPro),降低 compaction 所带来的性能损失,提升键值存储系统吞吐量.CoPro 具有 4 个优点:1)利用系统并行与流

水线提高 compaction 性能;2)为了平衡资源消耗与性能提升,在 CoPro 内部实现了决策组件,用于从线性方式和流水线方式中动态选择 compaction 的执行方式,即动态改变系统整体的并行度;3)在具有不同大小值的负载下进行 compaction 优化,让 CoPro 在更趋于真实的复杂负载下依旧可以发挥较好的效果;4)具有良好的可扩展性,该方案不仅适用于 compaction 任务的静态卸载,同样适用于 compaction 任务的动态卸载.使用 db_bench 与 YCSB-C 对 CoPro 进行了大量的实验验证 CoPro 的优势,CoPro 分别将主机端和设备端的 compaction 带宽提高了 2.34 倍和 1.64 倍,与此同时,两端的吞吐量增加了约 2 倍.主机端和设备端的 CPU 使用率分别增加了 74.0%和 54.0%.

## 1　背景介绍与研究动机

### 1.1　研究背景

LSM-tree 作为一种可以延迟更新且批量写入硬盘的日志型数据结构,利用顺序写来提高写性能,但其分层的设计会牺牲小部分读性能换来提高写性能.

LSM-tree 包含 2 个或 2 个以上的存储数据结构.在有 2 个以上部件的 LSM-tree 中,通常由大小连续增加的 $C_0,C_1,\cdots,C_{n-1},C_n$ 组件组成,如图 1 所示.$C_0$ 在内存,而 $C_1\sim C_n$ 通常在硬盘内,但内存会缓存 $C_1,C_2,\cdots,C_{n-1},C_n$ 中常被读取的数据.

1) 日志文件.当有新的数据需要被写入时,首先向存放在硬盘的写前日志文件中写入此次操作的日志.当系统发生故障时,可以通过写前日志来恢复数据.当内存组件中的数据转储到硬盘后,对应的写前日志也就失效,可以删除来释放存储空间.

2) 内存组件($C_0$).根据新数据的索引将该数据添加到 $C_0$ 中.因为 $C_0$ 设置在内存中,所以这一过程与硬盘 $I/O$ 无关.但内存的数据存储成本要高于硬盘,所以需要为 $C_0$ 的大小设定阈值.每当 $C_0$ 的大小即将超出阈值时,$C_0$ 就会把部分数据滚动合并到硬盘组件的 $C_1$ 中,以此缓解内存与硬盘之间存储成本的差异.

3) 硬盘组件($C_1,C_2,\cdots,C_{n-1},C_n$).随着数据的持续添加,当 $C_{i-1}$ 的大小达到其阈值后,滚动合并就会在相邻的 2 个部件 $C_{i-1}$ 和 $C_i$ 之间发生,数据会从小部件 $C_{i-1}$ 转储到大部件 $C_i$ 中.即数据一开始会插入到 $C_0$ 中,之后随着不停地滚动合并,最终数据会写入到 $C_n$ 中.

本文研究内容主要优化键值存储中的 compaction.接下来用一个典型的键值存储数据库——LevelDB 为例,介绍 compaction 的具体流程.LevelDB 有 2 种 compaction 操作:小 compaction 和大 compaction,如图 1 所示:

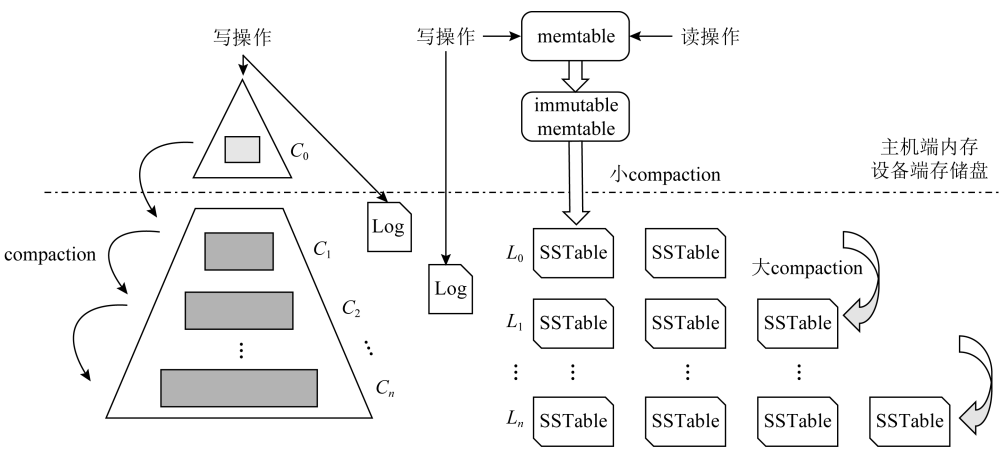



Fig. 1　LSM-tree and compaction of LevelDB

图 1　LSM-tree 和 LevelDB compaction 流程

小 compaction 将内存中不可修改的内存表(immutable memtable)持久化为 SSTable. 小 compaction 主要负责:1)构造 SSTable;2)决定新的 SSTable 文件写入哪一层.当内存中出现 immutable memtable 时即触发小 compaction,将 immutable memtable 构造为 SSTable,根据该 SSTable 文件与下面 level 层文件的重叠情况决定该表的具体写入层数.因后台只有一个线程负责 compaction 任务,所以当小 compaction 触发时,则暂停大 compaction,待小 compaction 操作处理完后再继续执行.

大 compaction 指的是 SSTable 之间的 compaction 操作.大 compaction 又有 3 种情况:1)手动 compaction,由人工触发的 compaction 任务,通过外部接口调用 compaction 任务具体的 level 与键范围,符合条件的 SSTable 参与 compaction 操作.这种 compaction 操作在 LevelDB 内部是不会自动触发调用的.2)容量 compaction,如在 $L_i$ 层,当超出该层的大小阈值时,则触发 compaction,通过轮询 $L_i$ 中 SSTable 的元信息(FileMetaData)选取 SSTable 参与 compaction 任务.然后,比较 $L_{i+1}$ 中的所有元信息,找出与 $L_i$ 选中的 SSTable 键范围重叠的文件加入到 compaction 任务中.对所有选中参与 compaction 任务的 SSTable 进行归并排序,生成新的 SSTable 写入 $L_{i+1}$,同时删除参与过 compaction 任务的旧 SSTable.容量 compaction 是 LevelDB 的核心 compaction 过程.3)查询 compaction,当某一 SSTable 的无效查找次数达到了阈值,则对该表进行 compaction 操作,同时在该表的下一层查找与其键范围重叠的文件加入 compaction 任务,执行归并排序操作.生成的新文件写入下一层并删除旧文件.该过程中会造成写入硬盘的数据量大于用户需求数据量,这一现象被称为写放大[8].

当前,近数据计算 NDP 的研究方向存在 2 种类型:1)内存计算(processing in-memory, PIM),其将处理器或加速器与主存储器并置[12];2)存储计算(in-storage computing, ISC),其将处理器或加速器与持久性存储(如 HDD 或 SSD)并置[13].近数据计算是计算机体系结构研究的一个非常活跃的领域[13-14],涉及有存内计算原型的研究[15-17]以及专用体系结构的研究[18-19]等.Balasubramonian 等人[10]提出了对近数据计算研究热情重新兴起的原因,例如基础技术的成熟以及当前架构无法满足的用户需求等.三星[11]在其 SSD 设备内部控制器集成了用户可编程的通用 ARM 处理器内核[20],采用特定的编程模型或框架,可将代码卸载到 SSD 上执行.

图 2(a)描述了传统的计算模型.CPU 与内存在结构的最顶层,且为任务的唯一处理位置.主机端响应应用任务请求,从存储盘中读取数据并执行相应任务.在该模型中存储盘仅用于存储数据.在图 2(b)中,是一个基于近数据计算模型的存储盘:该存储盘具有处理任务的能力,可接收从主机端卸载来的数据密集型任务(如数据检索).存储盘将数据读取到设备端内存中,交由 ARM 处理器执行任务,任务处理结束后将任务结果反馈给主机端即可.该模型可显著降低主机端接口的 I/O 压力,提高系统性能.近数据计算的基本思想是使计算能力接近数据存储的位置,从而减少数据在存储器层次结构中的移动.

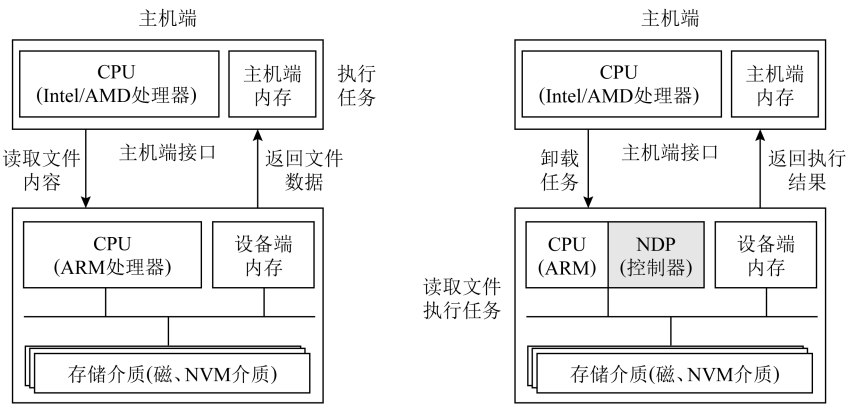



Fig. 2 Traditional computing model and near-data processing model

图 2 传统计算模型与基于近数据计算模型

### 1.2 研究动机

针对 compaction 操作会影响到键值存储系统性能的问题,在主机端并行优化方案中,PCP[21]使用流水线技术提高 compaction 性能和系统吞吐量,但带来较大的写放大.这是因为想要保证较高的 compaction 性能,PCP 需选取更多的 SSTable 参与 compaction,这增加了额外的数据写入,从而增加了写放大.这对于基于闪存的存储设备而言,会缩短其使用寿命.现有面向 LSM-tree 键值存储的近数据计算优化方案主要关注于主机端与 NDP 设备端之间的系统级并行性,或 NDP 设备本身所具有的并行性,较少将近数据计算架构中两者的并行性都充分

利用起来,如 Co-KV[22].基于上述问题,在近数据计算模型下,本文提出了基于 compaction 数据级-流水级的多级并行优化方法(CoPro).

## 2　CoPro 系统概览

CoPro 系统的架构如图 3 所示,主要分为 2 个部分,即主机端子系统和设备端子系统.主机端子系统负责接收键值存储应用的 API 指令(如 put,get,delete 等),同时负责与设备端子系统之间进行信息交互,协同处理键值存储应用的读写与 compaction 任务.设备端子系统负责接收从主机端子系统传来的指令,解析后执行相应的操作并向主机端返回结果.compaction 任务被分割成 2 个子任务后由主机端子系统和设备端子系统协同处理,实现数据并行.CoPro 在主机端与设备端内部实现对 compaction 子任务的流水线处理,增加了流水线并行.在 CoPro 中实现同时采用数据并行与流水线并行加速 compaction 任务.下面详细介绍主机端与设备端模块与处理流程.

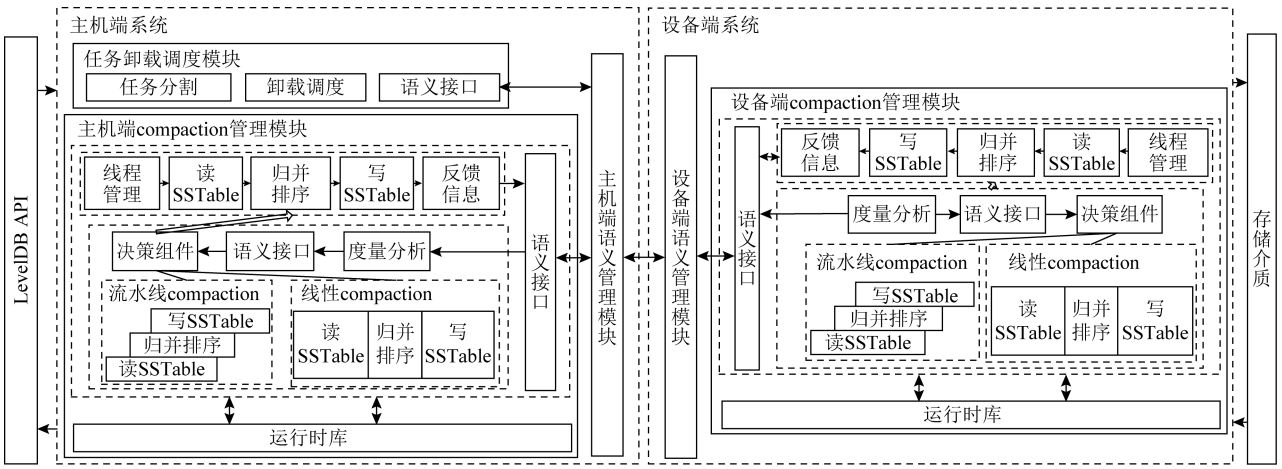



Fig. 3　Architecture of CoPro

图 3　CoPro 系统架构图

### 2.1　主机端子系统

主机端子系统主要接收处理 2 部分的指令信息:1)主机端键值存储操作指令信息;2)设备端执行结果的反馈信息.与设备端子系统之间的信息交互,需要主机端的语义管理模块支持.主机端子系统连接起了键值存储应用与设备端子系统,实现 compaction 任务的主机端数据并行,以及 compaction 流水线并行.

任务卸载调度模块.当大 compaction 触发时,由任务卸载调度模块依据调度策略对此次 compaction 任务进行分割,然后通过主机端语义管理模块将此次分割好的任务卸载到主机端和设备端,该模块是数据并行中对 compaction 数据进行分割的模块.至于任务卸载的调度策略,可以搭载静态调度策略,或动态调度策略.

主机端 compaction 管理模块.主机端 compaction 子任务的执行模块,负责处理 compaction 任务数据并行中主机端部分.该模块包括线程管理、读操作、归并排序、写操作、结果反馈等功能.该模块包含 2 种 compaction 子任务执行方式:一是传统的线性 compaction 处理方式,读操作、归并排序、写操作这 3 个阶段以顺序方式执行;二是流水线 compaction 处理方式,将读操作、归并排序、写操作这 3 个阶段按流水线方式组织,compaction 数据在这 3 个阶段间依次传递.通过线程管理组织线程去执行这 3 个阶段,实现了对 compaction 子任务的流水线处理,同时保留了传统线性 compaction 方式.

考虑到 NDP 设备的计算和内存资源可能有受限的情况,添加了决策组件用于决定采用何种 compaction 子任务执行方式,以适应不同的资源使用需求.度量分析器负责收集决策组件所需的判断依据(如 CPU 使用率、内存使用率等),然后按照语义协议封装信息,传递给决策组件,决策组件对这些信息进行处理,最后决定使用哪种 compaction 执行方式.决策依据可根据实际需求来采集不同的决策信息,使用相应的决策算法.

compaction 子任务完成后,将反馈信息(如 metadata 等)由主机端语义管理模块交给任务卸载

调度模块,其会整合主机端与设备端的 compaction 子任务反馈结果.运行时库为主机端 compaction 管理模块提供必要的 compaction 操作支持.

主机端语义管理模块.主机端语义管理模块保障了主机端子系统内部模块之间、主机端与设备端之间的通信.信息交互按照如下格式封装消息:[数据包头,数据包长度,类型,标志,序列化数据,校验码].其中数据包头表明该消息所代表的操作类型,如任务卸载、打开文件等,类型表明了该消息是由主机端(0x00)还是设备端(0x01)发出,如图 4 所示.此外,决策功能实现所需的信息通信,也由主机通过语义管理模块将所需度量信息发送给度量分析器,由其进行数据收集.消息按如下格式封装:[大小,标志,长度,度量信息,校验码],详见 3.2 节.

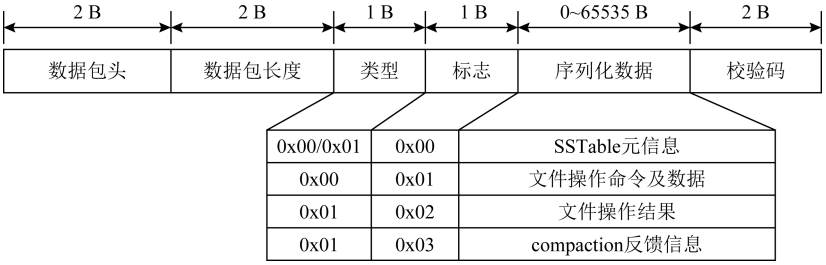



Fig. 4 Semantics transfer protocol of CoPro

图 4 CoPro 传输协议语义

### 2.2 设备端子系统

设备端子系统是键值存储应用读写请求的实际执行者.所有的读写操作都是由设备端子系统来执行,设备端子系统与存储介质进行直接的数据交互.同样,设备端实现 compaction 任务的设备端数据并行,以及 compaction 流水线并行.

设备端 compaction 管理模块.设备端 compaction 子任务的执行模块.因为其对 compaction 子任务的处理过程与主机端相同,同样是将读操作、归并排序、写操作这 3 个阶段按流水线方式组织以实现流水线并行,同时保留线性方式,故不再赘述.需要指出的是:设备端 compaction 管理模块的决策组件与主机端并无依赖关系,也就是说,可以在主机端与设备端同时执行不同的决策方案.同样地,运行时库为设备端 compaction 管理模块提供必要的 compaction 操作支持.

设备端语义管理模块.设备端语义管理模块与主机端语义管理模块功能类似,其接收来自主机端子系统的指令并解析(类型为 0x00),之后交由设备端子系统执行并将结果按既定语义封装反馈给主机端子系统(类型为 0x01).如接收设备端 compaction 子任务信息,将其解析后交给设备端 compaction 管理模块执行,任务完成后将反馈信息封装发送给主机端子系统.

设备端需要收集的度量信息也是设备端通过语义管理模块将消息发送给其内部的度量分析器,由其完成度量信息的收集.

从系统整体看,任务卸载调度模块对 compaction 任务进行分割,主机端与设备端子系统协同处理 compaction 任务,实现对 compaction 任务的数据并行;从系统内部看,主机端与设备端子系统用流水线方式处理 compaction 子任务,实现对 compaction 任务的流水线并行.

## 3 2 级 CoPro 并行动态调度策略

Co-KV 系统利用 NDP 模型,采用传统的线性 compaction 流程.Co-KV 中,在除了第 0 层以外的其他层中,SSTable 的键范围不存在重叠的情况,compaction 子任务之间也没有数据依赖关系.基于这个特点,主机端和设备端进行 compaction 子任务时可以将其并行执行.本文利用流水线并行的 compaction 优化方式,与 Co-KV 原有的数据并行方式结合,提高 compaction 任务的执行效率,提升主机端与设备端的资源利用率,优化系统性能.改进的同时并不会产生额外的写放大.

### 3.1 2 级 compaction 并行

对 SSTable 进行 compaction,主要包括读 SSTable、将 SSTable 中的键值对归并排序、写 SSTable.在现有近数据计算下的键值存储系统中,这 3 个操作是按顺序线性执行的.在主机端,首先对 compaction 任务用调度策略进行分割,主机端对 compaction 子任务依次执行读、归并排序、写操作,设备端同样对分配给自己的 compaction 子任务执行上述操作.待

2 个子任务完成时,均将结果反馈给主机端进行整合操作,各阶段执行过程如图 5(a)所示.主机端与设备端子系统协同处理同一个 compaction 任务,但是各自处理该任务的不同数据,实现对 compaction 任务的数据并行.

现在还是以读 SSTable—归并排序—写 SSTable 的顺序来执行 compaction 子任务,不同的是,利用子任务数据间的无依赖关系,可以将这 3 个不同的操作并行执行,以流水线的方式执行.具体的方法为:创建 2 个新线程用于处理键值对的合并排序和写 SSTable 操作.首先是读取 SSTable 内容,该操作依然由原本的 compaction 处理线程来执行,在 CoPro 中,每当读取了一定量的键值对后,就将其交给归并排序处理线程进行归并排序,同时 compaction 线程继续读取后面的内容;归并排序处理线程对键值对归并排序,保留有效的数据,删除过期数据,完成后再将有序的键值对交给写线程,之后继续处理 compaction 线程后续传来的键值对;写线程中设置有一块缓冲区,缓存从归并排序处理线程传来的键值对,当数据量达到 SSTable 的大小时(默认为 2MB),写线程会生成新的 SSTable 与 metadata,并将新的 SSTable 写入硬盘中.这样就实现了流水线的并行.

这样,在一次 compaction 任务中,刚开始依旧是执行任务分割,将任务卸载到主机端和设备端分别处理.在 compaction 子任务的处理过程中,将读、归并排序、写操作按照流水线的方式组织,形成了 3 段流水线.子任务结束后,主机端与设备端分别将各自的执行结果反馈并由主机端进行整合.各阶段过程如图 5(b)所示.

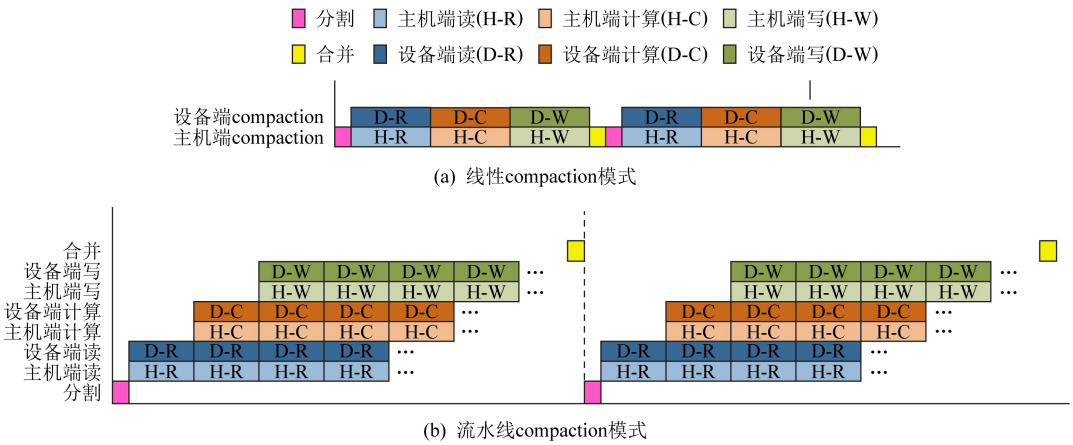



Fig. 5　Sequential and pipelined compaction modes

图 5　线性与流水线 compaction 模式

在 compaction 任务执行过程中,主机端与设备端子系统并行地处理各自的 compaction 子任务,实现 compaction 任务的数据级并行;在主机端与设备端 compaction 子任务中,compaction 的读、归并排序与写操作也并行执行,实现 compaction 任务的流水线并行.

### 3.2　CoPro 决策组件

通过 3.1 节中的方法,引入流水线 compaction 处理方式,进一步提升了系统处理 compaction 的性能,但是流水线方式是利用多线程实现的,在提升性能的同时也会增长资源消耗.NDP 设备存在异构性,其计算能力也有差异,从能耗角度考虑,设备端的计算能力可能会受到限制.此外,设备端的资源不可能全部用于服务 compaction 子任务,还需要保证其他任务的正常执行,这对如何利用好计算资源提出了挑战,尤其是否要使用多线程来开启流水线处理模块加速 compaction,以及何时需要开启与关闭多线程功能,判断的依据是什么.需要尽可能发挥资源受限设备的计算能力,减少对其他应用的影响.因此本文提出基于 CPU 使用率、内存使用率等参数的决策组件来根据实际需求,决策是否开启 compaction 流水线处理方式.

由图 6 所示,首先通过度量分析器收集系统运行过程中需要关心的资源情况,然后将消息按照语义格式封装.消息头部由消息大小构成,消息大小为本次消息的总大小.消息体由标志、长度、度量信息和校验码组成,标志记录其后的度量信息是什么类型的度量信息,而长度则记录着这个度量信息所占用的长度,方便度量信息的读取.最后对所有信息计算校验码,将校验码封装进消息中.封装好后交由语

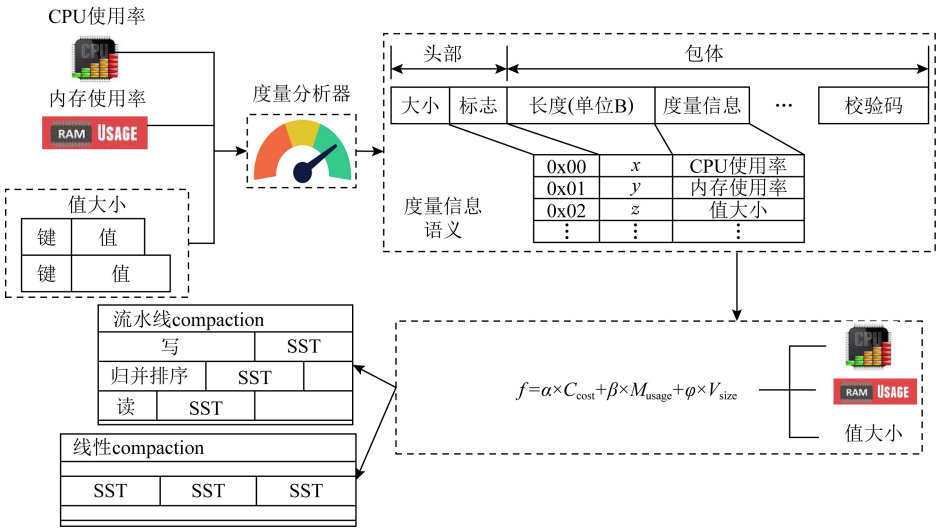



Fig. 6 Decision-making process in CoPro

图 6 CoPro 决策流程图

义接口传给决策组件,决策组件收到信息后,通过计算公式决定本次 compaction 子任务的处理方式.

$$f=\alpha\times C_{cost}+\beta\times M_{usage}+\varphi\times V_{size}, \quad (1)$$

式(1)用于决策数值 $f$ 的计算,其中 $C_{cost}$ 表示 CPU 使用率,$M_{usage}$ 表示内存使用率,$V_{size}$ 表示键值对中值大小,$\alpha$,$\beta$,$\varphi$ 则为各参数的权重.这样,可以根据实际环境设置适合的决策公式,如系统对 CPU 资源比较敏感,可以相应地增加 $C_{cost}$ 的权重.计算结果 $f$ 与预设的阈值 $f_{threshold}$ 进行比较,根据结果选择相应的 compaction 处理策略.

决策组件的加入,可以缓解性能与资源消耗之间的矛盾,尽可能地在性能提升与资源消耗之间找到平衡.例如当 CoPro 的 CPU 使用率超出了既定的阈值,决策组件就可以通过关闭流水线来降低 CoPro 的 CPU 使用率,待检测到系统的 CPU 资源充足时,此时决策组件就可以开启流水线来加速 compaction 过程.

在实际的执行过程中,compaction 子任务是采取传统线性流程还是流水线流程由决策组件决定,在后续的实验中,为了测试数据与流水线并行和决策组件的效果,并且为了方便区分,本文中的 CoPro 均指代主机端与设备端都默认采用流水线处理方式,即决策组件不生效,CoPro+均指代主机端与设备端的 compaction 处理方式由决策组件决定,即 compaction 处理方式是动态变化的.

### 3.3 CoPro 决策阈值($V_{size}$)

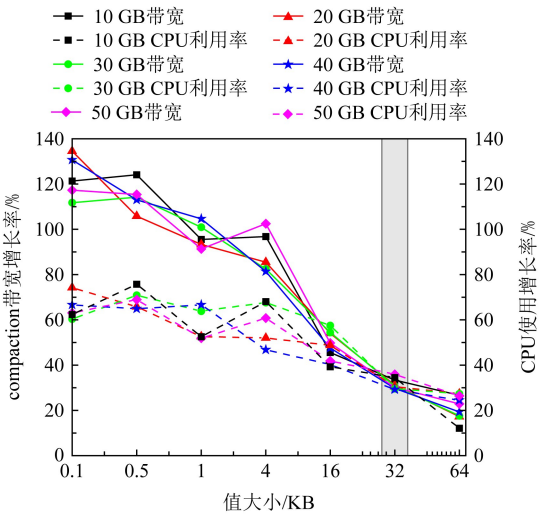
3.2 节提到的 $C_{cost}$ 和 $M_{usage}$ 比较好确定阈值选取,因为流水线 compaction 的启用与否直接会反映到 CPU 和内存使用率上,这样根据系统的实际需求可以很容易地设置其具体值.而对于 $V_{size}$ 的确定则有些困难,因为不清楚流水线 compaction 与负载键值对中值大小的关系.最为常见的是大小划分,即对不同大小的值采取决策,本文称为值大小的阈值.由于流水线是作用于 compaction 整体过程而不是单一的键值对,所以确定了值大小阈值后,还需要确定不同大小值在整个 compaction 过程中所占的比例,达到多少比例就需要触发决策,在本文中称为比例阈值.下面通过实验,从 compaction 带宽与 CPU 使用率分别代表性能与资源的角度来确定这 2 个阈值,其中 compaction 带宽为单位时间内 compaction 处理的数据量.

1) $V_{size}$——键值对中值大小阈值的确定

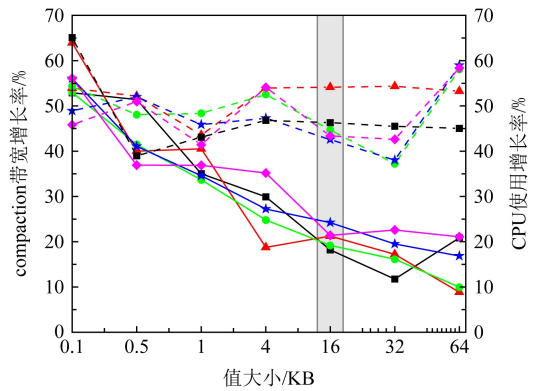
需要分别确定主机端与设备端的值大小阈值,原因在于主机端与设备端内均有决策组件,需要分别确定两端各自的阈值.

其次,在 NDP 架构下,主机端和设备端的计算、存储等资源不同,主机端和设备端执行数据级并行的 compaction 子任务时,执行特征也有所不同,两端环境的不同造成了需要分别确定阈值.本节使用 compaction 带宽增长率与 CPU 使用率的增长率比值作为参考依据,该比值为 1 时,表示 compaction 带宽增长率与 CPU 使用率的增长率相同,即 CPU 使用率 100%转化为 compaction 带宽,也就是 CPU 资源的消耗全部用于 compaction 性能的提升.下面利用负载 db_bench 进行实验,研究值大小对流水线 compaction 的影响,实验结果如图 7 所示.

如图 7(a)所示,在主机端,值大小为 16 KB,32 KB,64 KB 时的比值分别为 1.10,0.97,0.88.可以看到当




(a) 主机端值大小阈值




(b) 设备端值大小阈值

Fig. 7　Host-side and device-side threshold of record size

图 7　主机端与设备端 record 大小阈值

值大小达到 32 KB 时 compaction 带宽增长率低于 CPU 增长率.主机端,键值对中值为 64 KB 时的 compaction 带宽增长幅度已经开始低于 CPU 使用率的增长幅度,说明此时更多的 CPU 消耗并不能带来等量的性能提升.与 32 KB 时接近 1 的比值对比,表明从 32 KB 开始,继续增加值的大小,相同的 CPU 使用率增长已不能获取等量的 compaction 带宽提升,所以主机端选择 32 KB 作为值大小的阈值.因此本文定义,在主机端超过 32 KB 的值被认为是大键值对.在设备端该比值普遍低于 1,值大小为 4 KB,16 KB,32 KB,64 KB 时,比值分别为 0.53,0.45,0.40,0.28,如图 7(b)所示.从 16 KB 开始比值低于 0.5,意味着 1 倍的 compaction 带宽增长需要 2 倍以上的 CPU 消耗,所以设备端选择 16 KB 作为键值对中值大小的阈值,即对于设备端来说值超过 16 KB 的键值对认为是大键值对.至此,主机端与设备端值大小的阈值已确定.

2) $V_{size}$——比例阈值的确定

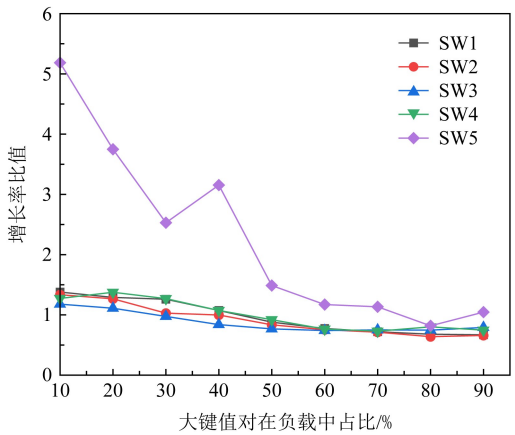
使用 compaction 带宽增长率与 CPU 使用率增长率的比值作为参考依据,负载如表 1 所示:

**Table 1　Workloads for the Proportion Threshold**

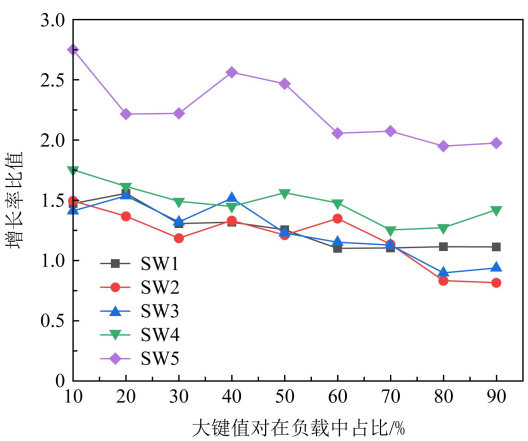
**表 1　比例阈值实验负载特征**

| 负载类型 | 负载描述 |
|---|---|
| SW1 | load 600000 键值对在 YCSB-C 负载 |
| SW2 | load 800000 键值对在 YCSB-C 负载 |
| SW3 | load 1000000 键值对在 YCSB-C 负载 |
| SW4 | run 800000 操作在负载 A(50%读,50%更新)在 YCSB-C 负载 |
| SW5 | run 800000 操作在负载(20%读,80%更新)在 YCSB-C 负载 |

如图 8 所示,在主机端,除了 SW5,其余负载比值基本重合于 60% 和 70%,在 70% 后,100% 写负载继续下降,读写混合负载数值上升.SW5 总体也




(a) 主机端增长率比值




(b) 设备端增长率比值

注:$x$ 轴为大键值对占负载中所有键值对的比例,10%即代表负载中大键值对在所有键值对中占10%,主机端大键值对为值大小超过 32 KB的键值对,设备端大键值对为值大小超过16KB的键值对.

Fig. 8　The host-side and device-side growth rate ratio

图 8　主机端与设备端增长率比值

是随着大键值对比例增大,比值呈下降趋势.80%是所有负载的平均最低点,往左右横跨一个区间,70%～90%是实验中平均比值最低的区间,所以主机端选择70%这一比例作为决策触发的比例阈值.在设备端,所有负载在70%前保持整体下降趋势,在70%后部分继续下降,比值低于1,部分负载缓慢的上升,70%～90%同样是设备端所有负载平均比值最低的区间,所以设备端同样选择70%作为决策触发的比例阈值.在本节中,当大键值对在负载中的比例达到70%时,使用流水线compaction将难以获得理想的性能提升,还会占用大量的计算资源,形成资源浪费,因此,将70%设置为比例阈值.

## 4 实验与结果

CoPro结合了数据并行和流水并行的优势,利用2种并行结合的方式进一步提高系统的总体性能.在本节进行大量实验来评估CoPro的compaction性能.在4.3节、4.4节与4.5节中,CoPro采用静态卸载方案(Co-KV),主机端与设备端各处理一半的compaction任务数据.在Co-KV的实验中已经证实Co-KV性能较PCP更优,写放大更小,所以本节仅与Co-KV进行实验对比,探究在不同类型负载下写放大、吞吐量、compaction带宽、带宽利用率和CPU使用率的差异.即使用Co-KV作为本节实验的评估基准.

### 4.1 实验环境

基于近数据计算架构的键值存储系统CoPro需要可执行近数据计算的存储设备作为支撑,需要增加专用硬件.由于缺乏真实基于近数据计算的SSD,所以搭建了一个模拟近数据计算环境的测试平台,用来测试Co-KV与CoPro的性能.主机端与设备端通过千兆以太网接口交互数据.

测试程序由主机端和设备端组成,主机端子系统运行在双核Pentium® Dual-Core, E6700 CPU,4 GB内存的计算机上,操作系统是Ubuntu 16.04.利用主机端的计算资源,构建基于近数据架构的新型键值存储软件.设备端需要额外增加专用硬件,本工作采用的是一种基于ARM的开发板Firefly作为计算单元.设备端运行在ARM开发板上,配置4GB运行内存,开发板连接256GB Intel 545S SATA SSD作为存储介质,开发板和SATA盘模拟为近数据计算盘,操作系统是轻量级嵌入式Ubuntu系统,设计CoPro设备端子系统的键值存储软件栈.

### 4.2 测试负载

负载的详细配置如表2和表3所示.CoPro主要在compaction流程上进行优化,而随机写又是引起compaction的主要原因之一,所以本节的测试负载主要集中于随机写方面.Co-KV和CoPro使用相同的默认配置,即4 MB MemTable,2 MB SSTable,4 KB data block.键值对中的值为16 B.本节使用DB_Bench(LevelDB默认的测试工具)和YCSB-C(YCSB的C++版本)作为测试工具,测试Co-KV和CoPro的写放大、吞吐量和CPU使用率等参数.使用YCSB-C作为测试真实环境下密集型随机写负载的测试工具,配置不同的数据量和记录大小,这些都是对compaction有较大影响的参数.请求分布使用zipfian和uniform分布.

**Table 2 Workload Characteristics in DB_Bench**

**表2 DB_Bench实验负载参数**

| 负载类型 | 随机负载 | 顺序负载 | 值大小/KB | 数据量/GB |
|---|---|---|---|---|
| db_bench_1 | 选择本项 | | 0.1,0.5,1,4,16,64 | 1,2,3,4,5 |
| db_bench_2 | | 选择本项 | | |

**Table 3 Workload Characteristics in YCSB-C**

**表3 YCSB-C实验负载参数**

| 写操作比例/% | 数据量大小/GB | | record大小/KB | record分布类型 |
|---|---|---|---|---|
| | load | run | | |
| 100 | 10 | 10 | 0.5,1,4,16,64 | zipfian,uniform |
| | 20 | 5,10,15,20,25 | 1 | |

### 4.3 基于DB_Bench负载的测试

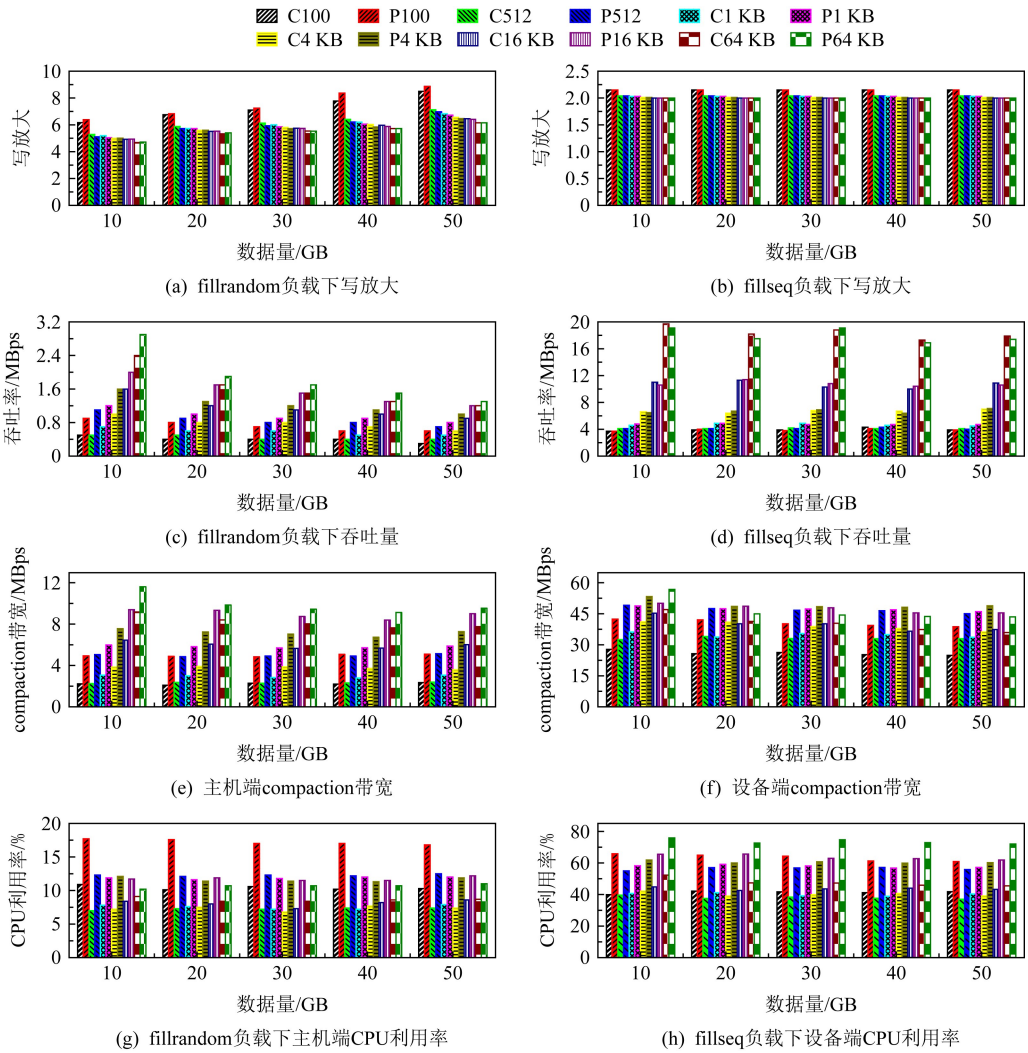
在本节,使用DB_Bench测试Co-KV与CoPro在不同数据量、不同大小值的负载下写放大、吞吐量、compaction带宽和CPU使用率,并分析实验数据,如图9所示.

写放大.不管是Co-KV还是CoPro,写放大主

要来自于主机端的写前日志与主机端的 compaction 子任务操作.由图 9(a)(b)可知,在随机写负载下,当值的大小不变时,随着数据量的增加,两者写放大也均在增大(例如在数据量 10 GB 和值为 100 B 时,Co-KV 写放大为 6.16,CoPro 写放大为 6.38;在数据量 50 GB 和值为 100 B 时,Co-KV 写放大为 8.49,CoPro 写放大为 8.86);而当数据量不变时,随着键值对中值的增大,两者写放大均在减小,例如在数据量 10 GB 和值 64 KB 时,Co-KV 写放大为 4.68,CoPro 写放大为 4.72.这是由于数据量的增加带来的 compaction 次数增加,造成了写放大的增大;而数据量一定的情况下,值的增大导致总的键值对减少,compaction 次数减少,写放大减小.

可以看出,在随机写的情况下,CoPro 与 Co-KV 的写放大差别不大,变化范围在±5%以内.这是因为 CoPro 改变 Co-KV 的 compaction 的流程,将其中的读、归并排序、写操作按流水线方式组织,但是在 SSTable 选取时不倾向于选取更多的文件参与,所以在加快 compaction 速度的同时并不会带来额外的写放大.在顺序写负载下,CoPro 与 Co-KV 的写放大基本相同,这是因为顺序写不涉及大 compaction,也就不涉及本文所做的优化部分,CoPro 与 Co-KV 处理方式相同,所以写放大一致.

吞吐量.吞吐量是评价数据库性能优劣的重要指标.由图 9(c)(d)可以看出,在随机写情况下,不论是何种负载,CoPro 的吞吐量都是要大于 Co-KV 的.这是因为 CoPro 不仅数据并行 compaction 任务,还流水线并行 compaction 子任务,加快了 compaction 子任务的处理效率,最直观的体现就是在吞吐量上.在实验最好情况下(数据量 20 GB 和值为 100 B),




注: C100代表Co-KV且实验value为100 B, P100代表CoPro且实验value为100 B.

Fig. 9　Results of CoPro and Co-KV under fillrandom- and fillseq-based DB_bench with various data volume

图 9　CoPro 与 Co-KV 在 DB_Bench 顺序与随机写负载下的实验结果

Co-KV 为 0.4 MBps,CoPro 为 0.8 MBps,吞吐量提升了 95%.但是随着值的增大,吞吐量的提升趋势逐渐下降.在实验最差情况下(数据量 20 GB 和值 64 KB),Co-KV 为 1.7 MBps,CoPro 为 1.9 MBps,吞吐量提升仅为 7%.这是因为,当数据量一定,值增大时,总的键值对减少,导致一次 compaction 任务中的键值对数量下降,流水线处理方式的优势被大大降低,所以总体的性能提升效果不明显,与 3.3 节的结果一致.通过不同负载的实验数据可以看出,CoPro 在处理较小值的负载时能最大程度上发挥流水线处理方式的优势,而在值增大后,优势逐渐减小,总体性能的提升逐渐降低.

由于顺序写不涉及 compaction 中的归并排序操作,所以在顺序写情况下 CoPro 与 Co-KV 吞吐量大致相同,没有太大的变化.

compaction 带宽.本文对主机端与设备端的 compaction 子任务处理流程进行优化,使用 compaction 带宽作为 compaction 操作性能评测指标.图 9(e)(f)展示的是随机写情况下,主机端和设备端的 compaction 带宽变化,由于顺序写不涉及 compaction 操作,所以不记录 compaction 带宽,本实验仅针对随机负载情况.当值大小不变时,随着数据量的增长,主机端和设备端 compaction 带宽总体呈现下降的趋势,这是由于数据量增长导致 SSTable 变多,compaction 操作的数据量和耗时同时增加,从实验结果来看,耗时的增长幅度要大于数据量的增长幅度,导致了 compaction 带宽下降.当数据量一定,值增大时,主机端的 compaction 带宽逐渐上升,这是因为键值对的减少缩短了 compaction 的处理时间.在设备端,情况更为复杂一些.Co-KV 的 compaction 带宽随着值增大而增大,CoPro 却没有太过明显的变化规律.从吞吐量的实验结果分析中可以看出随着值增大 CoPro 的性能提升效果逐渐降低,不管在主机端还是设备端,值增大时 CoPro 与 Co-KV 的 compaction 带宽也在逐渐接近.

在实验数据的最好情况下(数据量 20 GB 和值为 100 B),Co-KV 主机端 compaction 带宽为 2.09 MBps,设备端带宽为 25.68 MBps;CoPro 主机端和设备端 compaction 带宽分别为 4.90 MBps,42.10 MBps,分别提升 134%和 64%.在实验数据的最坏情况下(数据量 20 GB 和值 64 KB),Co-KV 主机端 compaction 带宽为 8.42 MBps,设备端带宽为 41.36 MBps;CoPro 主机端和设备端 compaction 带宽分别为 9.87 MBps,45.03 MBps,仅分别提升 17%和 9%.

CPU 使用率.使用流水线方式处理 compaction,性能提升的同时不可避免的也增加了资源的消耗,所以测试了主机端与设备端 CPU 的使用情况来统计 CoPro 对计算资源的消耗.

由图 9(g)(h)可知,在随机写负载下,不论主机端还是设备端,CoPro 要比 Co-KV 的 CPU 使用率高,最高是在数据量 20 GB 和值为 100 B 时,Co-KV 主机端 CPU 使用率为 10.1%,设备端 CPU 使用率为 42.1%;CoPro 主机端 CPU 使用率为 17.6%,设备端使用率为 64.8%.CPU 使用率分别提高了 74%和 54%.这是因为 CoPro 在 compaction 子任务的处理过程中使用流水线方式,而为了实现流水线方式,新增了 2 个线程分别处理归并排序和写操作,增加了 CPU 资源的消耗.顺序写情况下不涉及大 compaction 归并排序操作,新增线程不参与处理过程,处于睡眠状态,所以两者 CPU 使用率差别不大.例如在数据量 20 GB 和值为 100 B 情况下,Co-KV 主机端 CPU 使用率为 25.9%,设备端 CPU 使用率为 32.4%;CoPro 主机端 CPU 使用率为 26.5%,设备端使用率为 33.1%.

### 4.4 基于 YCSB-C 负载的测试

在 YCSB-C 下,首先测试了在不同负载情况下 record 大小对性能的影响,结果如图 10 所示.

由图 10(a)所示,在 zipfian 与 uniform 负载下 CoPro 的写放大与 Co-KV 差别不大,因为 CoPro 对 compaction 的处理流程做了优化,并不会带来额外的写放大.例如在 record size 为 512 B 时,在 zipfian 负载下,Co-KV 写放大为 4.21,CoPro 为 4.33;在 uniform 负载下,Co-KV 写放大为 6.44,CoPro 为 6.38.

随着 record 增大吞吐量下降,是因为 record 增大的同时增加了写入和 compaction 处理的时间,降低了系统的吞吐量,见图 10(b).在 record 为 1 KB 时,Zipfian 负载下 Co-KV 为 830.66 ops/s,CoPro 为 1 533.63 ops/s,uniform 负载下 Co-KV 为 441.43 ops/s,CoPro 为 817.66 ops/s,吞吐量均提升了 85%.与在 db_bench 下的测试结果相同,随着 record 增大,CoPro 的性能提升效果也逐渐下降.在 record 为 64 KB 时 CoPro 吞吐量仅仅提升 11%.

由图 10(c)(d)可知,不论在主机端还是设备端,CoPro 的 compaction 吞吐量均要高于 Co-KV,record 为 1 KB 时,zipfian 负载下 Co-KV 与 CoPro 主机端 compaction 带宽分别为 2.51 MBps 和 5.85 MBps,设备端 compaction 带宽分别为 41.53 MBps 和 60.14 MBps,带宽分别提升 133%和 45%.uniform

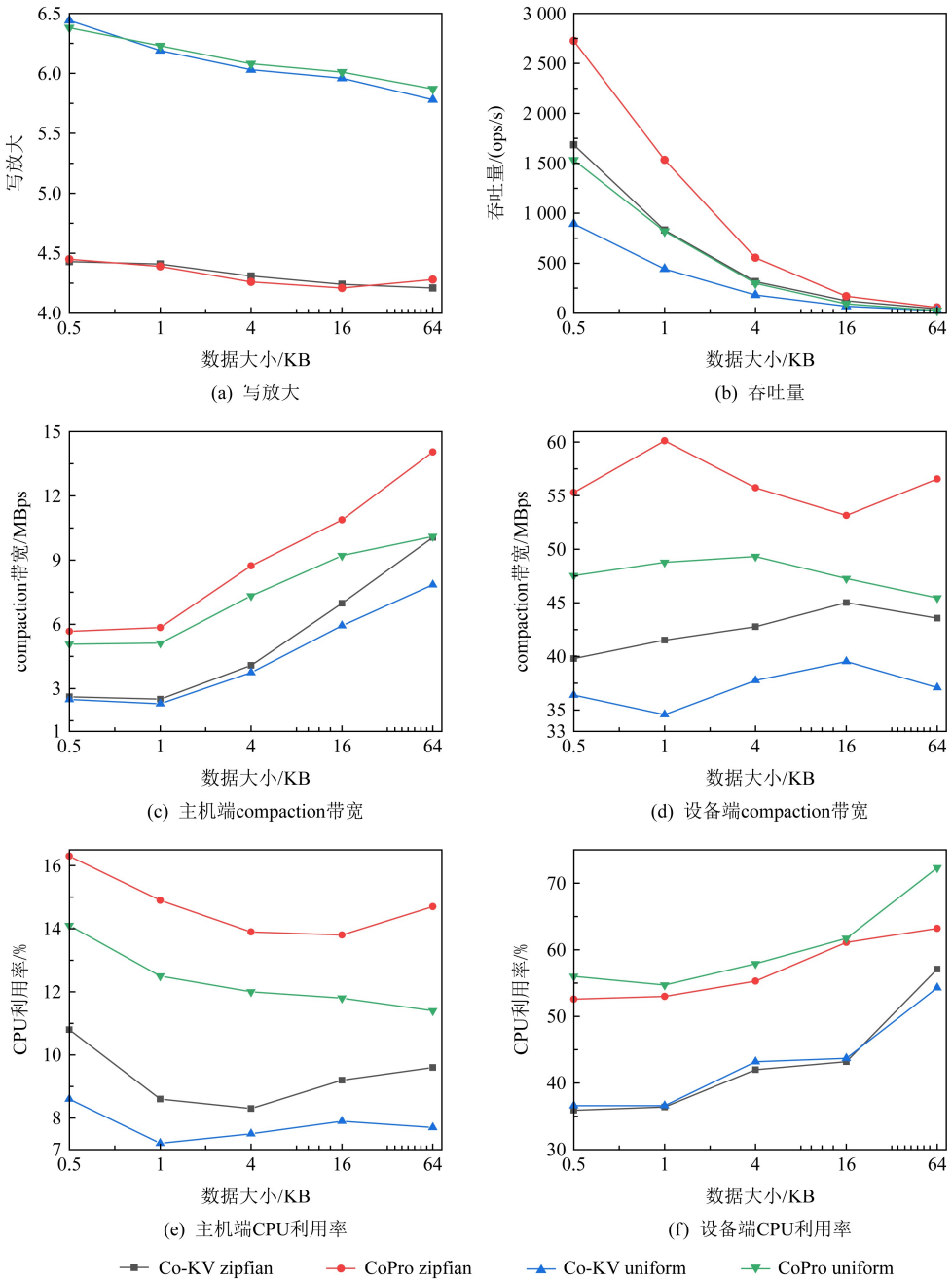



Fig. 10　Results of CoPro and Co-KV under zipfian- and uniform-based in YCSB-C with various record size

图 10　YCSB-C zipfian 和 uniform 不同 record 大小负载下 CoPro 和 Co-KV 的实验结果

负载下 Co-KV 与 CoPro 主机端 compaction 带宽分别为 2.29 MBps 和 5.12 MBps,设备端 compaction 带宽分别为 34.56 MBps 和 48.80 MBps,带宽分别提升 124%和 41%.这是因为 CoPro 增加线程加速了 compaction 过程,相同时间内 compaction 的数据量更多.

在 zipfian 负载下,record 为 1 KB 时 CoPro CPU 使用率增长最高,此时 Co-KV 主机端 CPU 使用率为 8.6%,设备端 CPU 使用率为 36.4%,CoPro 主机端 CPU 使用率为 14.9%,设备端 CPU 使用率为 53%,分别增长 73%和 47%;在 uniform 负载下,同样是 record 为 1 KB 时 CoPro CPU 使用率增长最高,此时 Co-KV 主机端 CPU 使用率为 7.2%,设备端 CPU 使用率为 36.6%,CoPro 主机端 CPU 使用率为 12.5%,设备端 CPU 使用率为 54.7%,分别增长 74%和 49%.具体结果参见图 10(e)(f).

此外,我们在实验中还测试了数据量对各项性能指标的影响,如图 11 所示:

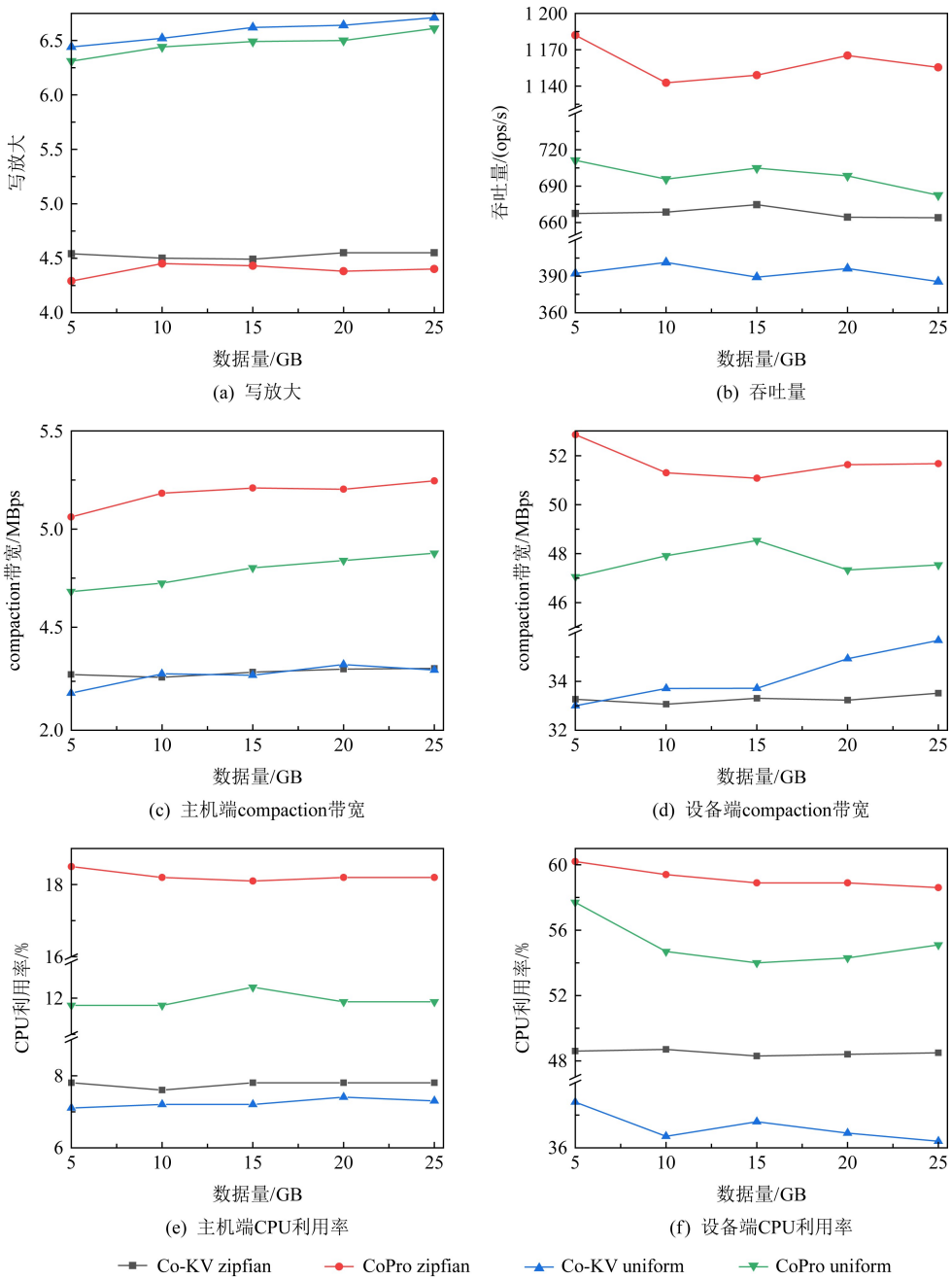



Fig. 11 Results of CoPro and Co-KV under YCSB-C with various data volume

图 11 CoPro 与 Co-KV 在 YCSB-C 不同数据量实验结果

随着数据量的增加写放大呈缓慢上升的趋势，当数据量达到 25 GB 时，zipfian 负载下 Co-KV 与 CoPro 写放大分别为 4.6 和 4.4，uniform 负载下 Co-KV 与 CoPro 写放大分别为 6.7 和 6.6.由图 11(a)可知，Co-KV 与 CoPro 写放大差别不大.

尽管数据量变化的同时吞吐量的变化趋势不明显，但总体上来说还是随着数据量的增加吞吐量呈下降趋势，如图 11(b)所示.当数据量为 5 GB 时，zipfian 负载下 Co-KV 与 CoPro 分别为 667.40 ops/s 和 1 181.95 ops/s. 在 uniform 负载下，Co-KV 与 CoPro 分别为 392.22 ops/s 和 711.12 ops/s.当数据量为 25 GB 时，zipfian 负载下 Co-KV 与 CoPro 分别为 663.82 ops/s 和 1 155.49 ops/s.在 uniform 负载下，Co-KV 与 CoPro 分别为 385.57 ops/s 和 682.48 ops/s.

如图 11(c)(d)所示，compaction 带宽的变化趋势随数据量变化相对来说不明显，但可以看到，不论是主机端还是设备端，CoPro 的 compaction 带宽都是要大于 Co-KV 的，在 zipfian 负载下，主机端与设

备端 compaction 带宽分别平均提升 126%和 55%;在 uniform 负载下,分别平均提升 110%和 39%.

CPU 使用率基本不受数据量变化的影响,因为 compaction 次数会随着数据量的增加而增加,但系统总的运行时间也在增加,一段时间内 CPU 的使用量还是相同的,如图 11(e)(f)所示.

### 4.5　扩展实验

1) CoPro+实验

本节研究 CoPro+(具有决策组件的 CoPro)在不同工作负载下的性能.由于在 3.3 节 $V_{size}$ 的确定实验中,发现流水线 compaction 对值大小敏感,为了简单测试决策组件的效果,本节决策组件在决策公式中仅设置 $V_{size}$,即阈值与比例阈值,其中值大小的阈值,主机端为 32 KB,设备端为 16 KB,比例阈值为 70%(原因见 3.3 节),$C_{cost}$ 与 $M_{usage}$ 参数暂未启用.即 CoPro+的决策组件计算公式完全以 $V_{size}$ 为导向,此时的 $\varphi=1,\alpha=\beta=0,f_{threshold}=1\times V_{size}=0.7$.通过这种配置可以更直观地展现负载键值对中值的大小变化过程中决策组件的作用.表 4 显示了此次实验中配置的 4 种工作负载.负载 $W1$,$W3$,$W4$ 分别对应工作负载早期、中期和后期出现大键值对(大键值对的定义见 3.3 节).

**Table 4　Workloads for Scalability Evaluation**

**表 4　扩展实验的 YCSB-C 负载特征**

| 负载类型 | 负载描述 |
|---|---|
| 负载 $W1$ | 800000 操作次数 (100%更新),在负载运行的早期设置大键值对 |
| 负载 $W2$ | 800000 操作次数 (20%读,80%更新),在负载运行的中期设置大键值对 |
| 负载 $W3$ | 800000 操作次数(100%更新),在负载运行的中期设置大键值对 |
| 负载 $W4$ | 操作次数(100%更新),在负载运行的后期设置大键值对 |

如图 12 所示,其中 decision number 为系统运行时的检测点,每个检测点决策组件会进行一次决策.决策组件初始状态默认使用流水线 compaction 方式.图 12 中阴影部分表示运行过程中大键值对占比超过 70%.当占比小于 70%时,决策组件开启流水线 compaction 模式,否则,将使用线性 compaction 模

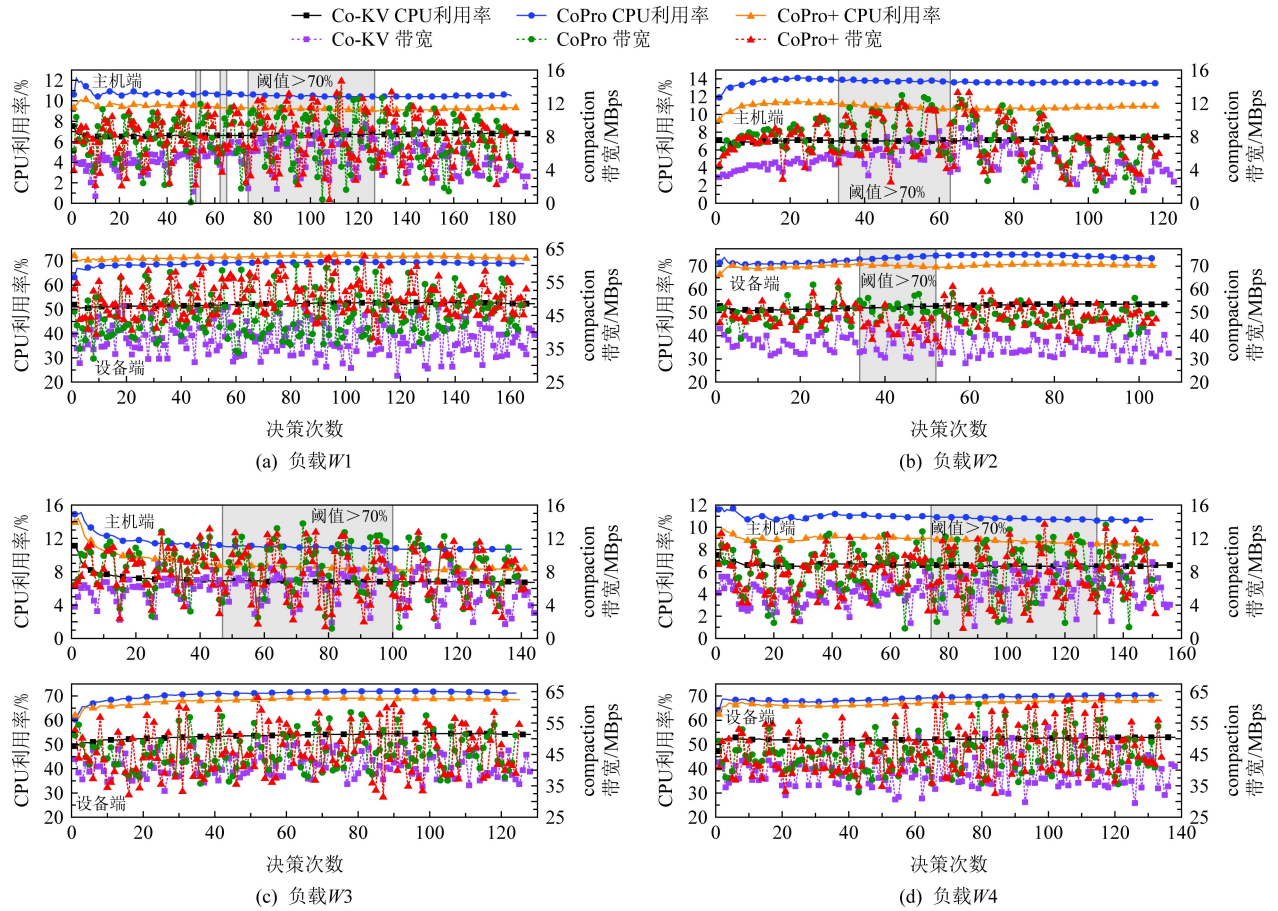



Fig. 12　Results of Co-KV, CoPro, and CoPro+ under YCSB-C at run stage

图 12　Co-KV, CoPro, CoPro+在 YCSB-C 不同负载 run 阶段的实验结果

式.当发生动态决策时,主机端和设备端中,CoPro+的CPU使用率介于CoPro与Co-KV之间,运行过程中达到阈值时,CoPro+关闭了流水线处理方式,使得CPU使用率下降.compaction带宽CoPro+更接近于CoPro,因无法保持一段时间内检测点的数值均高于阈值,短时间内反复开启流水线方式,这使得CoPro+对compaction带宽的影响不是很直观.

根据表5的实验数据可知,在大多数情况下,CoPro+比CoPro用更低的CPU使用率带来了更高compaction带宽.这主要归功于CoPro+在负载中大键值对占比较大时关闭了流水线,避免了更多的资源消耗.表5中的CPU与compaction带宽增长率是相对于Co-KV的增长率,符号$MBp_{CPU}$表示compaction带宽除以CPU使用率的值,其表示每1%CPU使用率所带来的compaction带宽,CoPro+不论在主机端还是设备端均高于CoPro.与Co-KV相比,主机端基本持平甚至略优(负载$W1$:Co-KV为0.81,CoPro+为0.82),而设备端则要低于Co-KV(负载$W1$下Co-KV为0.73,CoPro+为0.70),由4.3节和4.4节的实验可知,当record大小增大时设备端的处理耗时相比主机端要增加更多,所以CoPro+在设备端的优化效果要次于主机端.另外从负载$W3$实验结果可知,在写多读少的混合负载下,CoPro+依然可以发挥作用.

**Table 5 Compaction Performance and CPU Growth Rate for Co-KV, CoPro and CoPro+ Under YCSB-C**

**表5 Co-KV, CoPro, CoPro+在YCSB-C不同负载下的compaction性能与CPU资源增长率**

| 负载类型 | 操作 | $MBp_{CPU}$(CK)/% | $MBp_{CPU}$(CP)/% | $G_C$(CP)/% | $G_B$(CP)/% | $MBp_{CPU}$(CP+)/% | $G_C^+$(CP+)/% | $G_B^+$(CP+)/% |
|---|---|---|---|---|---|---|---|---|
| 负载$W1$ | run(主机端) | 0.81 | 0.73 | 58.6 | 42.3 | 0.82 | 39.5 | 41.2 |
| | run(设备端) | 0.73 | 0.65 | 32.2 | 17.9 | 0.70 | 37.3 | 31.8 |
| 负载$W2$ | run(主机端) | 0.71 | 0.57 | 91.6 | 54.0 | 0.67 | 52.2 | 45.1 |
| | run(设备端) | 0.69 | 0.66 | 39.4 | 34.5 | 0.68 | 33.3 | 32.5 |
| 负载$W3$ | run(主机端) | 0.85 | 0.75 | 59.6 | 40.4 | 0.87 | 27.4 | 31.1 |
| | run(设备端) | 0.77 | 0.64 | 32.1 | 10.4 | 0.67 | 27.2 | 10.9 |
| 负载$W4$ | run(主机端) | 0.87 | 0.75 | 65.1 | 41.8 | 0.86 | 34.5 | 32.9 |
| | run(设备端) | 0.76 | 0.65 | 32.2 | 12.9 | 0.68 | 28.4 | 15.9 |

注:CK,CP,CP+分别表示Co-KV,CoPro,CoPro+.$MBp_{CPU}$表示compaction带宽与CPU利用率的比值,说明单位CPU消耗下compaction带宽的提升.$G_C$表示CoPro相比Co-KV的CPU使用率增长率.$G_C^+$表示CoPro+相比Co-KV的CPU使用率增长率.$G_B$表示CoPro相比Co-KV的compaction带宽增长率.$G_B^+$表示CoPro+相比Co-KV的compaction带宽增长率.

CoPro+的作用体现在负载中record大小动态变化时,其可以降低因流水线compaction而导致的在负载中大键值对占比较大时不必要的CPU资源消耗.换句话说,CoPro+能够以小于CoPro的CPU使用率换来接近于CoPro的compaction带宽增长.但还是需要根据实际的情况选择,要想获得最好的性能提升效果当然是使用CoPro,但想要节省CPU资源并不降低太多性能的话,就使用CoPro+.

2) CoPro在读写混合负载下的实验

如图13所示,我们还分析了不同读写比例下Co-KV和CoPro的吞吐量、尾延迟、平均响应时间及主机端和设备端CPU使用率.我们选择了YCSB作为负载程序,并且设定了2种负载参数YCSB-A(读写混合):load 10 GB,run 10 GB,50%read,50%write.YCSB-B(读密集)load 10 GB,run 10 GB,90%read,10%write.

在YCSB-A负载下,CoPro吞吐量和平均延迟都比Co-KV优化了约23.3%;在CPU使用率上我们分别统计了主机端CPU使用率和设备端CPU使用率.发现CoPro的主机端使用率超过50%,相比Co-KV,提高了2.78倍,说明主机端compaction加入流水线技术后对CPU使用率影响很大,因此我们在CoPro增加了可以对流水线进行控制的机制.而CoPro设备端CPU使用率与CoKV相近.我们测量了CoPro和Co-KV的尾延迟P99,实验结果表明,CoPro的P99比Co-KV优化了10.6%.P99的优化主要来自于CoPro对写操作的优化,提高了99%的响应时间.

在YCSB-B负载下,CoPro的吞吐量和平均延迟都比Co-KV下降了约3.2%;同样我们也发现CoPro的主机端CPU使用率超过50%,相比Co-KV的10%提高了6.1倍,说明主机端compaction

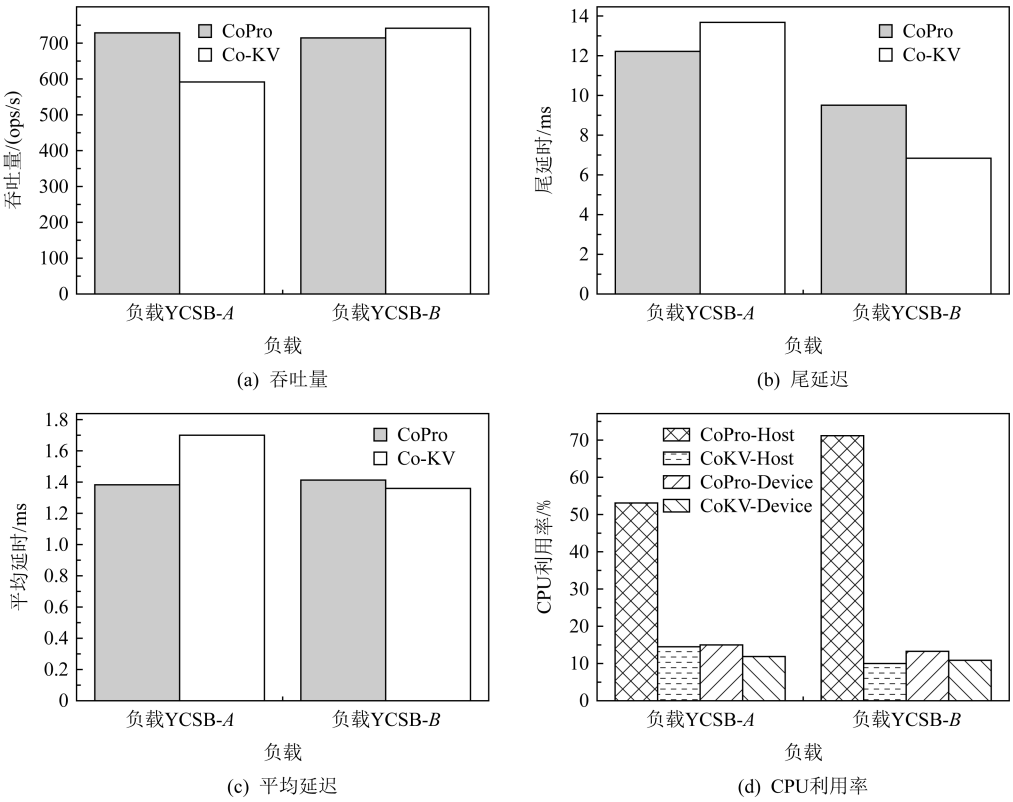



Fig. 13　Performance of CoPro under mixed-read-write workloads

图 13　读写混合负载下的 CoPro 性能

加入流水线技术后密集的读操作对 CPU 使用率影响较大,而 CoPro 设备端 CPU 使用率与 Co-KV 相近.对于长尾延迟,实验结果表明,CoPro 的尾延迟 P99 比 Co-KV 下降了 19%.前面提到 CoPro 在长尾延迟的优化可能来自于写操作的优化,但是在读密集操作下,这种优化无法体现,另外可以看出读密集负载下 CoPro 的性能有所下降,主要在于流水线 compaction 操作消耗了主机 CPU 资源,由此验证了主机 CPU 资源被 compaction 占用较多时,会影响读性能.

## 5　相关工作

### 5.1　基于 LSM-tree 的键值存储相关工作

基于 LSM-tree 的键值存储系统吞吐量高、扩展性强,适用于时延敏感的互联网服务,被广泛应用于大规模数据密集型互联网应用中,并逐渐替代传统关系型数据库,迅速成为研究热点.

PCP[21]从 compaction 处理流程入手,提出了一种流水线 compaction 方法,以更好地利用 CPU 和 I/O 并行性来提高 compaction 性能.为了更好地利用 CPU 和 I/O 并行性,文献[21]将读、归并排序、写阶段的执行流水线化,以此实现并行 compaction 操作来改善 compaction 的性能,但是为了提高流水线的效果,PCP 倾向于让更多的 SSTable 参与 compaction,增加了写放大.WiscKey[23]提出键值分离的方案,将键值分开存储,可以大幅降低 LSM-tree 的大小与 compaction 后的读写放大.HashKV[24]基于键值分离的基础上,根据键将值散列到多个分区中,并独立地对每个分区进行垃圾回收.GearDB[25]利用叠瓦盘垃圾回收的特点,利用 compaction 的消除数据操作来减少垃圾回收操作.SEALDB[26]通过避免随机写入和 SMR 驱动器上相应的写入放大来专门为 SMR 驱动器优化.NoveLSM[27]是 NVM 上 LSM-tree 的实现,添加了一个基于 NVM 的内存组件,利用 I/O 并行性来同时搜索多层以减少查找延迟.SLM-DB[28]在新型字节型存储器件 Persistent Memory 上利用 PM 的特性使用 B+tree 作为索引与使用 PM 进行缓冲写入,加速读取时的数据检索并省略了写前日志的写开销.

上述 LSM-tree 键值存储系统的相关工作均有效地提升了存储系统性能与吞吐量.但是,以上方案均没有考虑利用存储设备所具有的计算资源.随着近数据计算模型研究的深入,学者开始利用具有一定计算能力的存储设备来处理部分数据密集型应用任务,以提升系统性能.

### 5.2 近数据计算相关工作

1) 近数据计算基本架构

随着技术发展,存储设备内部的计算资源日益增长,其内部带宽与外部带宽日渐悬殊,于是学者们开始研究如何把数据密集型应用的部分任务卸载到存储设备内部进行处理来减少数据移动消耗.Kang 等人[29]在真实的固态硬盘上验证近数据计算模型,并将其在 SSD 固件中实现,通过有效利用 SSD 内部并行性而提高主机端处理性能.中国科学院计算技术研究所提出了 Cognitive SSD[30],这是一种基于深度学习的非结构化数据检索的节能引擎,以实现近数据深度学习和图形搜索.不仅是闪存访问加速器,以 FPGA 为代表的多功能可编程硬件加速器也逐渐流行起来[31].INSIDER[32]引入了基于 FPGA 的可重配置驱动控制器作为存储计算单元,极大提高了使用的灵活性.

2) 近数据计算键值系统

Co-KV 首次利用 ARM 开发板将 LevelDB 的部分 compaction 任务通过以太网接口发送到设备端执行,主机端处理剩余的 compaction 任务,实现了 compaction 任务的静态卸载.验证了近数据计算模型可以有效解决 LSM-tree 写放大的问题,并且提高了写操作的吞吐量,减轻了主机端计算任务的压力.在 Co-KV 的基础上,Sun 等人又提出动态卸载方案 TStore[33]与 DStore[34].分别将 NDP 设备的执行时间和计算资源作为卸载条件,以此为依据来动态调整 compaction 任务的分割比例,从而进一步提高了 compaction 的执行速度,同时有效合理的分割任务,使主机端和设备端的计算资源都能够充分利用,使系统性能进一步提高.nKV[35]是一个利用本机计算存储与近数据计算的键值存储系统,将读操作、搜索操作和复杂的图形处理算法卸载到存储设备处理.nKV 消除了兼容层并利用了 NDP 上的计算资源,所以有较好的性能表现.

上述研究方案很好地展现出近数据计算模型的可行性和性能优势,将数据密集型应用的部分计算和 I/O 任务卸载到基于近数据计算模型的存储设备上进行处理,可以有效地利用 NDP 存储设备内部的高带宽,提升任务整体的处理效率.然而上述研究方案大多是直接将应用的某个或多个任务全部卸载到 NDP 存储设备上,没有考虑到主机端计算资源也可参与任务处理,可能会造成主机端资源的空闲与浪费.或者是简单利用主机端与 NDP 设备端之间的系统并行性,没有充分发挥 NDP 设备端的并行资源.

## 6 结 语

本文提出了近数据计算架构下 compaction 的流水并行优化方法,CoPro.与 Co-KV 相比,CoPro 分别将主机端和设备端的 compaction 带宽提高了 2.34 倍和 1.64 倍.与此同时,主机端和设备端的吞吐量均增加了约 2 倍,CPU 使用率分别增加了 74.0% 和 54.0%.在扩展实验中,与 CoPro 相比,在不明显降低性能的同时 Co-Pro+将 CPU 使用率的增长率降低了 16.0%.CoPro 实现了 compaction 操作的数据并行与流水并行共存,使用流水方式执行 compaction 的同时减小了写放大.同一 compaction 任务被主机端与设备端并行执行,而在主机端和设备端内部 compaction 子任务按流水的方式组织并行执行.在主机端子系统中,CoPro 重新设计了任务卸载调度模块,使其可以同时支持 compaction 任务的静态卸载方案,增强了可扩展性.增加了度量收集器用于收集主机端子系统运行时的度量信息,为决策组件提供数据以实现对系统整体并行度的动态调整.在设备端子系统中同样增加了度量收集器以及决策组件,用于收集设备端子系统运行时的度量信息.主机端与设备端的决策组件相互独立,可以根据不同的度量信息确定不同的决策,例如可以缓解因值增大造成的性能提升下降与资源消耗上升的问题,使 CoPro 具有一定程度上的环境感知功能.通过真实硬件模拟近数据计算环境进行验证实验和扩展实验,实验结果显示 CoPro 在不带来额外写放大的前提下进一步提升了 compaction 性能.

**作者贡献声明**:孙辉、娄本冬负责工作思路、系统搭建、实现及测试;黄建忠、赵雨虹、符松主要负责思路及实验讨论.

## 参 考 文 献

[1] David R, John G, John R. Data Age 2025: The digitization of the world from edge to core [EB/OL]. International Data Corporation 2018 [2021-06-05]. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf

[2] Han Jing, Song Meina, Song Junde. A novel solution of distributed memory NoSQL database for cloud computing [C] //Proc of the 10th IEEE/ACIS Int Conf on Computer and Information Science. Piscataway, NJ: IEEE, 2011: 351-355

[3] Xie Huacheng, Chen Xiangdong. Cloud storage-oriented unstructured data storage [J]. Journal of Computer Applications, 2012, 32(6):1924-1928

[4] Cattell R. Scalable SQL and NoSQL data stores [J]. ACMSIGMOD Record, 2011, 39(4): 12-27

[5] Apache. HBase [OL]. Apache software foundation, 1999 [2021-06-05]. http://hbase.apache.org/

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

[7] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data [J]. ACM Transactions on Computer Systems, 2008, 26(2): 1-26

[8] Sanjay G, Jeff D. LevelDB, a light-weight, single-purpose library for persistence with bindings to many platforms [OL]. Google Inc, 2012 [2021-06-05]. https://github.com/google/leveldb

[9] Facebook Database Engineering Team. RocksDB, a persistent key-value store [EB/OL]. [2021-06-05]. https://rocksdb.org/

[10] Balasubramonian R, Chang Jichuan, Manning T, et al. Near-data processing: Insights from a micro-46 workshop [J]. IEEE Micro, 2014, 34(4): 36-42

[11] Gu B, Yoon A S, Bae D H, et al. Biscuit: A framework for near-data processing of big data workloads [J]. ACM SIGARCH Computer Architecture News, 2016, 44(3): 153-165

[12] Patterson D, Anderson T, Cardwell N, et al. A case for intelligent RAM [J]. IEEE Micro, 1997, 17(2): 34-44

[13] Acharya A, Uysal M, Saltz J. Active disks: Programming model, algorithms and evaluation [J]. ACM SIGOPS Operating Systems Review, 1998, 32(5): 81-91

[14] Draper J, Chame J, Hall M, et al. The architecture of the DIVA processing-in-memory chip [C] //Proc of the 16th Int Conf on Supercomputing. New York: ACM, 2002: 14-25

[15] Computer Systems Laboratory at Sungkyunkwan University. The OpenSSD project [OL]. 2016 [2021-06-05]. http://www.openssd-project.org/

[16] Nair R, Antao S F, Bertolli C, et al. Active memory cube: A processing-in-memory architecture for exascale systems [J]. IBM Journal of Research and Development, 2015, 59(2/3): 17:1-17:14

[17] Seshadri S, Gahagan M, Bhaskaran S, et al. Willow: A user-programmable SSD [C] //Proc of the 11th USENIX Symp on Operating Systems Design and Implementation (OSDI 14). Berkeley, CA: USENIX Association, 2014: 67-80

[18] Brewer T M. Instruction set innovations for the Convey HC-1 computer [J]. IEEE Micro, 2010, 30(2): 70-79

[19] Minutoli M, Kuntz S K, Tumeo A, et al. Implementing radix sort on EMU1 [C/OL] //Proc of the 3rd Workshop on Near-Data Processing (WoNDP). 2015 (2015-12-05)[2021-06-01]. https://www.cs.utah.edu/~rajeev/m inutoli15.pdf

[20] Tseng Hung-Wei, Zhao Qianchen, Zhou Yuxiao, et al. Morpheus: Creating application objects efficiently for heterogeneous computing [J]. ACM SIGARCH Computer Architecture News, 2016, 44(3): 53-65

[21] Zhang Zigang, Yue Yinliang, He Bingsheng, et al. Pipelined compaction for the LSM-tree [C] //Proc of the 28th Int Parallel and Distributed Processing Symp. Piscataway, NJ: IEEE, 2014: 777-786

[22] Sun Hui, Liu Wei, Huang Jianzhong, et al. Collaborative compaction optimization system using near-data processing for LSM-tree-based key-value stores [J]. Journal of Parallel and Distributed Computing, 2019, 131: 29-43

[23] Lu Lanyue, Pillai T S, Gopalakrishnan H, et al. Wisckey: Separating keys from values in SSD-conscious storage [J]. ACM Transactions on Storage, 2017, 13(1): 1-28

[24] Chan H H W, Li Yongkun, Liang C J M, et al. HashKV: Enabling efficient updates in KV storage via hashing [C] //Proc of the 29th USENIX Annual Technical Conf (USENIX ATC 18). Berkeley, CA: USENIX Association, 2018: 1007-1019

[25] Yao Ting, Wan Jiguang, Huang Ping, et al. GearDB: A GC-free key-value store on HM-SMR drives with gear compaction [C] //Proc of the 17th USENIX Conf on File and Storage Technologies (FAST 19). Berkeley, CA: USENIX Association, 2019: 159-171

[26] Yao Ting, Tan Zhihu, Wan Jiguang, et al. SEALDB: An efficient LSM-tree based KV store on SMR drives with sets and dynamic bands [J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 30(11): 2595-2607

[27] Kannan S, Bhat N, Gavrilovska A, et al. Redesigning LSMs for nonvolatile memory with NoveLSM [C] //Proc of the 29th USENIX Annual Technical Conf (USENIX ATC 18). Berkeley, CA: USENIX Association, 2018: 993-1005

[28] Kaiyrakhmet O, Lee S, Nam B, et al. SLM-DB: Single-level key-value store with persistent memory [C] //Proc of the 17th USENIX Conf on File and Storage Technologies (FAST 19). Berkeley, CA: USENIX Association, 2019: 191-205

[29] Kang Y, Kee Y, Miller E L, et al. Enabling cost-effective data processing with smart SSD [C] //Proc of the 29th IEEE Symp on Mass Storage Systems and Technologies (MSST). Piscataway, NJ:IEEE, 2013: 1-12

[30] Liang Shengwen, Wang Ying, Lu Youyou, et al. Cognitive SSD: A deep learning engine for in-storage data retrieval [C] //Proc of the 30th USENIX Annual Technical Conf (USENIX ATC 19). Berkeley, CA: USENIX Association, 2019: 395-410

[31] Firestone D, Putnam A, Mundkur S, et al. Azure accelerated networking: Smartnics in the public cloud [C] // Proc of the 15th USENIX Symp on Networked Systems Design and Implementation (NSDI 18). Berkeley, CA: USENIX Association, 2018: 51-66

[32] Ruan Zhenyuan, He Tong, Cong J. INSIDER: Designing in-storage computing system for emerging high-performance drive [C] //Proc of the 30th USENIX Annual Technical Conf (USENIX ATC 19). Berkeley, CA: USENIX Association, 2019: 379-394

[33] Sun Hui, Liu Wei, Huang Jianzhong, et al. Near-data processing-enabled and time-aware compaction optimization for LSM-tree-based key-value stores [C/OL] //Proc of the 48th Int Conf on Parallel Processing. 2019 (2019-08-05) [2021-06-01]. https://dl.acm.org/doi/pdf/10.114 5/3337821.3337855

[34] Sun Hui, Liu Wei, Qiao Zhi, et al. DStore: A holistic key-value store exploring near-data processing and on-demand scheduling for compaction optimization [J]. IEEE Access, 2018, 6: 61233-61253

[35] Vinçon T, Bernhardt A, Petrov I, et al. nKV in action: Accelerating KV-stores on native computational storage with near-data processing [J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2981-2984

**Sun Hui**, born in 1983. PhD, associate professor. His main research interests include computer system, edge computing, performance evaluation, non-volatile memory-based storage systems, file systems, and I/O architectures.

**孙 辉**,1983 年生.博士,副教授.主要研究方向为计算机系统、边缘计算、性能评估、非易失性存储系统、文件系统和 I/O 架构.

**Lou Bendong**, born in 1995. Mater. His main research interests include computer system, key-value storage system.

**娄本冬**,1995 年生.硕士.主要研究方向为计算机系统与键值存储系统.

**Huang Jianzhong**, born in 1975. PhD, associate professor. His main research interests include computer architecture and dependable storage systems.

**黄建忠**,1975 年生.博士,副教授.主要研究方向为计算机体系结构和可靠的存储系统.

**Zhao Yuhong**, born in 1983. PhD, associate professor. His main research interests include computer architecture, social network, and big data.

**赵雨虹**,1983 年生.博士,助理研究员.主要研究方向为计算机系统结构、社交网络和大数据.

**Fu Song**, born in 1977. PhD, professor. His main research interests include cyberinfrastructures, parallel, distributed and IoT-edge-cloud systems, including architecture, performance, dependability, cybersecurity, and machine learning.

**符 松**,1977 年生.博士,教授.主要研究方向为网络基础设施、并行、分布式和物联网边缘云系统,包括架构、性能、可靠性、网络安全和机器学习.