高技术通讯 2024 年 第 34 卷 第 11 期:1142-1152

doi:10.3772/j.issn.1002-0470.2024.11.002

面向 EOV 联盟链的高效隔离性校验方法①

薛 晗② 包云岗③

(中国科学院计算技术研究所 北京 100190)

(中国科学院大学 北京 100049)

摘 要 Hyperledger Fabric 等采用交易预执行(EOV)模型的区块链系统,通常采用多版本并发控制(MVCC)方法校验交易,其基本思想是在提交前校验当前交易的读集合是否已经被跨块和块内的交易修改过,以保证交易隔离性和正确性,然而 MVCC 常因为并行化低和频繁底层数据库访问成为整个交易周期的性能瓶颈。为解决此问题,本文提出了 2 种新 MVCC 算法:C-MVCC 和 P-MVCC。C-MVCC 以内存作为缓存,减少 MVCC 校验时对底层数据库系统的访问,并且针对跨片校验阶段引入并行化机制;P-MVCC 则是在 C-MVCC 的基础上进一步拆解交易间的依赖关系,同时实现了跨块和块内校验的并行化,从而消除分片场景下过高的跨片校验时延累积。实验结果表明,C-MVCC 在一般场景下相较于当前的算法可以带来 7.5 倍的校验效率提升,而 P-MVCC 在分片场景中可以带来 9.0 倍的校验效率提升。

关键词 区块链;多版本并发控制(MVCC);缓存;并行化

区块链具有历史数据可追溯、不可篡改和安全透明的特点,受到众多学界和工业界的广泛关注[1-2]。区块链技术发展到现在,已经从最初的比特币[3]衍生出众多变种,如支持图灵完备智能合约的以太坊[4]、面向企业联盟的 Hyperledger Fabric[5]等。从应用场景和节点接入模式来划分这些区块链系统可以分成公有链、联盟链和私有链[6]。公有链允许任意身份的节点加入到系统中,比特币和以太坊是典型的代表;联盟链则要求组成联盟的各个实体的成员节点可以加入到链系统中,Fabric 在其中最有代表性和影响力;私有链则是指只有实体内部成员才允许加入。

从交易执行的顺序模型角度来划分,区块链大致可以分为交易后执行(order-execute,OX)与交易预执行(execute-order-validate,EOV)2 类。排序(order)表示区块链中的共识服务在打包交易时为交易赋予先后顺序。以比特币和以太坊为代表的公有链系统通常采用 OX 的执行模型,即先排序后按序执行。Hyperledger Fabric 为代表的联盟链有很多则采用 EOV 执行模型,先乐观地并行执行交易,排序后再进行校验。在国内区块链市场中,Fabric 技术底座在 2021 年已占据 48% 的份额[7],因此 EOV 执行模型拥有相当重要的影响力。

采用 EOV 执行模型的联盟链,基于版本快照并行预执行交易,所以其交易执行效率相比 OX 要高很多,但是基于快照执行需要在提交交易前引入多版本并发控制(multi-version concurrency control,MVCC)机制对交易的隔离性和正确性展开校验,而这往往会成为 EOV 架构联盟链的重要性能瓶颈。MVCC 基本思想是提交前校验当前交易的读集是否已经被跨块和块内的交易修改过,以保证当前读集的内容依然有效。然而当前 MVCC 校验方案存在

① 国家重点研发计划(2019YFB1802600),国家自然科学基金联合基金(U20A20179)和国家自然科学基金(62172007)资助项目。
② 男,1994 年生,博士生;研究方向:计算机系统结构;E-mail:xuehan19b@ict.ac.cn。
③ 通信作者,E-mail:baoyg@ict.ac.cn。
(收稿日期:2023-05-31)

性能问题,以 Fabric 为例。首先,串行地执行单笔校验,很难充分利用宿主机的计算资源;其次,对比读集的版本号时完全借助底层数据存储系统如 LevelDB,这无疑增加了中央处理器(central processing unit,CPU)和底层输入/输出(input/output,IO)的开销。并且随着存储数据规模的增大,底层数据库对数据的检索、查询性能也会出现衰减。

针对以上 EOV 联盟链在进行 MVCC 校验时存在的性能问题,本文遵循增加缓存和并行化 2 条主线对其进行优化,主要贡献包括以下几个方面。

(1)首先分析了以 Fabric 为代表的采用 EOV 模型的联盟链在 MVCC 校验时存在的性能问题,指出 MVCC 算法潜在的优化方向。

(2)提出 C-MVCC(cached MVCC)校验方法,以内存作为缓存,减少 MVCC 校验时对底层数据库系统的访问,避免了大量的底层 IO 开销,并且对于跨块校验进行充分地并行化,大幅度提升校验效率。本文通过详细实验,对比 Fabric 的 MVCC 算法,一般场景下 C-MVCC 的校验效率可以提升 7.5 倍。

(3)在 C-MVCC 的基础上,通过分析块内交易的依赖关系,进一步对块内校验并行化,提出 P-MVCC(parallel MVCC)校验方法,将跨块校验和块内校验过程都彻底并行化,以应对像分片等对校验并行度要求更高的场景。实验结果表明 P-MVCC 在分片场景下的校验效率可以提升 9.0 倍。

1　研究背景

1.1　EOV 联盟链简介

本节以 Fabric 为代表介绍 EOV 联盟链。Fabric 中有 3 种角色:客户端是交易提议的发起者;排序节点,也被称为共识节点,负责排序交易、打包交易和分发区块;Peer 节点,也被称为账本节点,负责预执行交易(也称为背书)、校验区块和维护账本数据。

图 1 是 Fabric 的交易流程示意图。交易执行流程主要分成以下 4 个阶段。

阶段 1:预执行。该步骤也被称为“背书”(endorsement)。客户端或 SDK(Software Development Kit)发起交易提议(proposal)到不同组织的背书节点(步骤 1),背书节点根据最新的账本状态运行智能合约,生成交易的读写集,并且附加上自身签名(步骤 2),最后将结果返回给客户端(步骤 3)。客户端收到来自不同组织的背书结果,检查其有效性。预执行其实只是一种模拟执行,不会对状态数据库产生任何修改。

阶段 2:排序。客户端收到来自不同背书节点的背书反馈后,检测这些反馈中读写集是否一致,如果一致则将这些读写集打包成交易然后发送给排序节点(步骤 4)。每个排序节点同时接收多个这样的交易,排序节点间采用拜占庭容错/故障容错等共识协议来对接收到的交易进行排序并将其打包成区块(步骤 5),并且将区块广播到各组织的账本节点(步骤 6)。

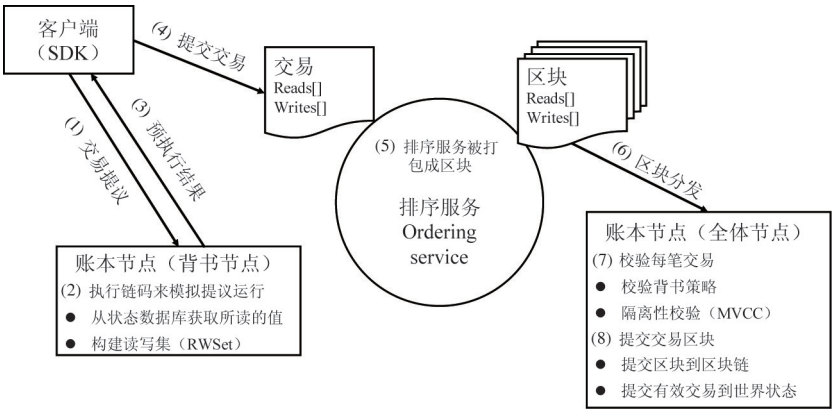

图 1　EOV 联盟链的工作流

阶段 3:校验。账本节点在接收到新区块后会对其进行背书策略校验和可串行化校验(或者叫隔离性校验)。背书策略校验用来判断交易是否被背

书策略所指定的组织签名。可串行化校验通常用多版本并发控制完成,具体做法是在提交前对交易读集是否被修改进行判定,如果没有被修改,那么交易则被认为是有效的,否则认为交易已经失效,这些失效交易最终不会对状态数据库产生影响(步骤 7)。

阶段 4:提交。校验结束后,账本节点对区块进行提交:首先把区块存储到文件系统中;然后提交校验有效的交易到状态数据库,更新世界状态;最后存储所有有效交易更新的元数据到历史数据库中(步骤 8)。

1.2 交易的隔离性与 MVCC

EOV 联盟链的隔离性校验使用 MVCC 方法来保证。其具体做法是:状态数据库中记录每个关键字的条目是一个三元组 $<key, ver, val>$,ver 表示最后一次更新该 key 的版本号,由区块号和块内交易号的二元组构成。每笔交易在预执行读取 key 时,连同版本号一同读出,在最后校验时,判断交易中所有的读集版本号是否依然是最新的(没有被修改过)。通过以上方式,EOV 架构的联盟链的隔离性达到了“快照隔离”(snapshot isolation,SI)级别。

本节以一个例子来描述 MVCC 校验的过程,表 1所示是某个账本节点收到的 2 个区块的交易校验流程。节点根据区块号和交易在块内顺序所构成的全局顺序依次对交易展开校验。节点首先对 T_1 校验,发现它所读的 k_1 的版本依然是最新的,这笔交易被标记为有效,因此它对 k_2 的写操作最后会生效,k_2 的版本号会更新为 v_2。随后节点校验 T_2 为失败,因为它的读集中 k_2 已经被 T_1 修改。T_3 和 T_4 因为没有涉及到读操作,因此直接被判定为生效。B_1 校验完成后,节点提交区块并且更新修改到世界状态数据库。

然后节点开始校验区块 B_2。判断出 T_5 的读集中 k_3 已经在 B_1 中被 T_4 修改了,因此判定 T_5 为失败交易,而 T_6 交易虽然同样读取了 k_3,因为它读取的关键字的版本号依然是最新的,因此最终判定为成功。以上例子中不难发现,判断交易的读集是否为最新时只需判断 2 种情况:之前区块内交易及当前区块内交易是否更新过读集中关键字的内容和版本号。这 2 种情况的校验分别被称为跨块校验和块内校验。上述例子中,T_2 是因为块内校验而被判定失败,T_5 则是因跨块校验被判定为失败。

表 1 MVCC 校验示例

区块	交易	读集	写集	结果	更新操作
B_1	T_1	(k_1, v_1)	k_2	√	$(k_2, v_1 \rightarrow v_2)$
	T_2	(k_2, v_1)	k_1	×	-
	T_3	-	k_1	√	$(k_1, v_1 \rightarrow v_2)$
	T_4	-	k_3	√	$(k_3, \varnothing \rightarrow v_1)$
B_1 提交					
B_2	T_5	$(k_3, \varnothing)$	k_4	×	-
	T_6	(k_3, v_1)	k_4	√	$(k_4, v_1 \rightarrow v_2)$
B_2 提交					

2 研究动机

本文聚焦于 EOV 联盟链的 MVCC 流程校验效率提升,本节以 Fabric 为例分析当前 MVCC 算法存在的性能问题。

Fabric 的 MVCC 校验流程如算法 1 所示。首先,它收到区块后,针对每笔交易循环校验(第 3 行),在校验交易的过程中,依次校验每一个读集中的 key 是否有效(第 9 ~ 13 行)。这里的 key 校验涉及 2 部分,首先是检测当前区块的更新(updates)中是否包含了该 key,如果包含则意味着已被块内顺序更靠前的其他交易修改,因此失效(第 10 ~ 11 行),这一过程被称为块内校验,在内存中进行;然后是对比该 key 的读取版本和状态数据库作对比,如果不一致,则说明已被之前区块中的交易所修改(第 12 ~ 13 行),这一过程为跨块校验,该阶段需要访问底层数据库系统(LevelDB、CoughDB)。

算法 1 Fabric 现有 MVCC 方案

```
Input: 待校验区块 blk,状态数据库 db
Output: 区块内交易所有更新 updates
1 function VALIDATEBLOCK(blk)
2     updates ← newUpdates( )
3     for tx in blk. txs do
4         if VALIDATETX(tx, updates) then
5             updates applies tx. writes
6     return updates
7
8 function VALIDATETX (tx, updates)
```

```
9    for key in tx.reads do
10       if key in updates then
11           return FALSE
12       if key.ver ! = VER(db, key) then
13           return FALSE
14   return TURE
```

分析目前 Fabric 的 MVCC 校验方案,如图 2 所示,可以发现这个过程是完全串行进行的。这里串行体现在 2 方面,首先是交易间完全串行,其次是交易内部的读集间校验也是完全串行的。这种串行的校验方式,并不能完全释放多核架构的处理能力,此外,对于跨块校验步骤,由于其需要频繁访问底层存储系统,串行执行更是成倍地增加了校验延迟。

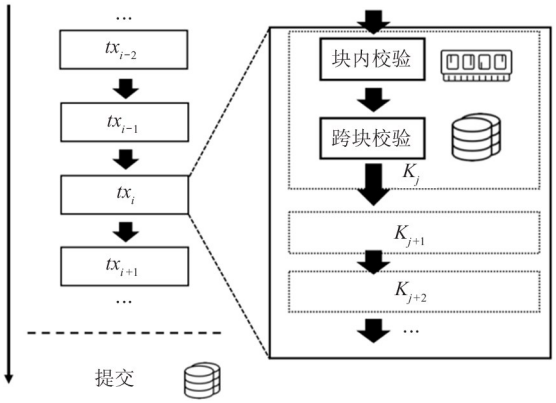

图 2　Fabric 的 MVCC 算法示意图

3　EOV 联盟链 MVCC 校验方法优化

针对以上观察,本文提出 C-MVCC 和 P-MVCC 这 2 种校验方法来提升 MVCC 流程的校验效率。

3.1　C-MVCC 算法

C-MVCC 相较于上文中介绍的 Fabric 的 MVCC,将交易的跨块校验和块内校验过程分离,主要提升在 2 点:(1)为跨块校验增加内存缓存,减少跨块校验时的底层数据库访问;(2)跨块校验实现完全并行化。

3.1.1　算法设计

图 3 和算法 2 分别是 C-MVCC 的示意图和 C-MVCC 算法流程。算法的输入是待校验的区块、状态数据库以及可设定大小的内存缓存,其中缓存记录了当前校验区块前 *maxLen* 个区块所有成功的写操作。C-MVCC 首先对区块内所有交易并行展开跨块校验(第 3 ~5 行),具体到每笔交易操作是:循环校验读集中的全部 *key*,判断其是否在缓存区间内,然后对其版本进行对比。在此有 2 种情况,第 1 种是如果在区间内,那么只需判断该 *key* 的版本号与缓存中的最新版本是否一致,如果不一致,则表示该 *key* 在交易背书后已被修改,因此直接判定该交易为无效(第 15 ~ 18 行);相反,如果版本一致,则意味着该 *key* 校验成功。第 2 种情况是,如果该 *key* 不在缓存区间内,那么说明缓存不能覆盖该交易从背书到当前校验区块的所有写操作,因此直接对比该 *key* 的版本是否与状态数据库中的版本相同,如果不相同则该 *key* 校验为失效(第 20 ~22 行)。交易读集中所有 *key* 校验完成后,表示该交易的跨块交易通过。所有交易的跨块校验结束后,C-MVCC 进行块内校验(第 6 ~9 行)。以上过程在交易间并行化进行,C-MVCC 并没有在交易内部对每个读集的读集并行化,因为通常交易的数量已经足够多,并行度已经非常高,再在内部并行化并不能带来明显的效果提升。

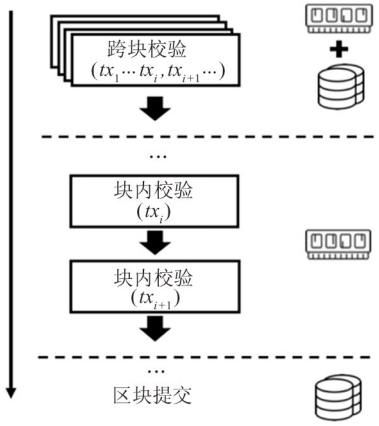

图 3　C-MVCC 示意图

块内校验由于在内存中进行,没有太复杂的 IO 操作,而且本身计算逻辑也比较简单,因此通常不会成为性能瓶颈。所以 C-MVCC 块内校验直接串行循环判断各交易的读 *key* 是否已存在当前区块的修改操作(*updates*)中,如果存在则判定该交易为无效交易(第 25 ~28 行)。

算法 2　C-MVCC 算法流程

Input：待校验区块 *blk*，状态数据库 *db*，区块写集缓存 *cache*
Output：区块内交易所有更新 *updates*

```
1 function VALIDATEBLOCK(blk, cache)
2   updates← newUpdates( )
3   for tx in blk. txs do // in parallel
4       VLDCROSSBLK(tx, cache)
5   Sync all threads
6   for tx in blk. txs do
7       if tx is valid then
8           if VLDINNERBLK(tx, updates) then
9               updates applies tx. writes
10  cache applies updates
11  return updates
12
13 function VLDCROSSBLK(tx, cache)
14  for key in tx. reads do
15      if key in cache then
16          if key. ver ! = VER(cache, key) then
17              mark tx as invalid
18              return
19      else
20          if key. ver ! = VER(db, key) then
21              mark tx as invalid
22              return
23
24 function VLDINNERBLK(tx, updates)
25  for key in tx. reads do
26      if key in updates then
27          return FALSE
28  return TURE
```

3.1.2　缓存设计

本小节介绍如何设计高效缓存。缓存的数据结构如图 4 所示，包含已缓存的关键字和最新版本（keys）、缓存区间内每个区块中的有效写操作（blocks）、当前缓存的区块数量（len）以及最大可缓存区间（maxLen）。

当新创建链时，无需初始化，只有在节点重新启动时，需要从链上获取最新 *maxLen* 个区块，将这些区块的写操作更新到缓存中，这个更新操作和下面介绍的正常运行时的更新相同。运行时更新是指新区块校验完成后，C-MVCC 对缓存的更新。具体做法是：当前缓存的区块数量小于*maxLen*，则直接将

```
1. type Cache struct {
2.        keys   map[string]*KeyWithVersion
3.        blocks *list
4.        len    uint32
5.        maxLen uint32
6. }
```

图 4　缓存的数据结构

该区块的写操作放入 *blocks* 中并且更新 *keys* 中对应的 *key* 的最新版本号，然后 *len* 增加 1；如果当前缓存已经存满（*len* = *maxLen*），需要从 *blocks* 中剔除最旧的区块，并且从 *keys* 中删除一些项，这些项的 *key* 存在于该块中并且版本号与块中的写操作的版本相同。遵循以上的运行机制，缓存中始终保存了最新 *maxLen* 个区块的所有写操作和这些写操作中 *key* 的最新版本以便于校验时能被快速获取。

3.2　P-MVCC 算法

为进一步提升 MVCC 校验的并行度，使其能在某些对并行度要求更高的场景下（例如分片场景）拥有更好的校验效率，本文在 C-MVCC 的基础上提出 P-MVCC。

3.2.1　设计动机

分片（sharding）[8-10] 作为解决区块链可扩展性问题最有前景的路线之一，一直以来受到业界和学界的关注。EOV 架构的区块链在采用分片架构时，针对跨片交易 MVCC 校验时需要多分片之间协调，此时交易间的依赖关系和校验顺序将很大程度上影响 MVCC 的执行效率。

本节以实例来介绍分片给 MVCC 校验提出的新要求。在 Fabric 的 MVCC 校验流程中，交易将严格按照顺序依次校验，即使在 C-MVCC 中，块内校验也需要顺序校验。这样的校验流程潜在地为交易附加了依赖关系，即每笔交易都要等待前面的交易校验结束后才会开始本交易的校验。未分片场景下，这种附加的依赖关系并不会给校验的效率造成过多的影响，因为块内校验在内存中进行，并且操作并不复杂。然而在分片场景下，这种附加的依赖关系会大幅影响校验效率。假如某区块中包含了 $\{tx_1, tx_2, tx_3, tx_4, tx_5, tx_6\}$ 这 6 笔交易，其中 tx_2 和 tx_5 为跨分片交易。由于交易按顺序校验，tx_6 需要

等前面所有交易校验结束后才能开始校验。tx_2 和 tx_5 是跨分片交易,需要和其他分片同步校验结果,因此它们需要消耗更长的时延,而它们两者的校验延迟最后累加到该区块的整体校验延迟上,因此当排到 tx_6 校验时可能已经过了很长时间。从用户的角度,可以感知到交易校验的延迟变得更长,从系统角度也会发现这种附加的依赖关系会显著影响区块的整体校验效率。

3.2.2　P-MVCC 设计理念

MVCC 算法本身就是用来校验交易间的冲突,进而判断交易的有效性,这种冲突其实就是交易间读写集的重叠。这种重叠包括对同一个 *key* 的“读后写(read-after-write,RAW)”、“写后读(write-after-read,WRA)”、“读后读(read-after-read,RAR)”和“写后写(write-after-write,WAW)”。如果交易的读写集之间没有重叠,那么就不会存在这种冲突。结合上文分析,可以针对交易的依赖关系类型做简单分类:交易间仅因排序先后导致的依赖关系,可以称之为附加依赖,对相同 *key* 的操作覆盖所形成的依赖关系,称之为偏序依赖。

以表 1 中区块 B_1 内的交易为例分析依赖关系在 MVCC 中扮演的角色。T_1、T_2、T_3 这 3 笔交易之间存在偏序依赖,T_4 和前面 3 笔交易仅存在附加依赖。具体地,T_1 和 T_2 之间因 k_1 存在读后写依赖,因 k_2 存在写后读依赖;T_1 和 T_3 之间因 k_1 存在读后写依赖;T_2 和 T_3 之间因 k_1 存在写后写依赖。根据这些依赖关系,校验中应该保持 T_1、T_2、T_3 的校验顺序,因为这三者之间都相互存在偏序依赖,交换任意两者之间的顺序都会导致偏序依赖关系的改变,从而可能导致校验结果的错误(事实上,除了读后读依赖,其他 3 种形式的潜在依赖都会因交换顺序导致结果错误)。这里的正确与错误表示校验结果和按照原始共识服务排序的顺序校验的结果是否相同。如 tx_1 和 tx_2 之间交换之后,tx_2 将判断为生效而 tx_1 此时会被判定为无效,这显然与正确的校验结果是不相符的。从另一个角度来分析,对 tx_4 而言,无论在其他 3 笔交易中任意交易之前或者之后执行都不影响整体校验的正确性。基于以上分析可以得出,MVCC 无论采用何种顺序校验,只要保证交易间的偏序依赖不被破坏,那么最后的校验结果就是正确的。即两交易间如果仅存在附加依赖,那么它们可以乱序校验而不会影响最终校验结果的正确性。

P-MVCC 的设计理念是拆解交易间的依赖关系,尽可能提高交易间的并行化程度,在保证偏序依赖不被破坏的前提下,彻底解除交易间的附加依赖。

3.2.3　P-MVCC 算法设计

图 5 和算法 3 分别是 P-MVCC 的示意图和 P-MVCC 算法描述。P-MVCC 主体流程也是将跨块校验和块内校验拆分进行。其跨块校验和 C-MVCC 采用同样的方式,完全并行地校验每笔交易,针对交易中的读集依次对比缓存或数据库中最新版本号,并以此判断读集是否已经失效(第 4 ~ 6 行)。

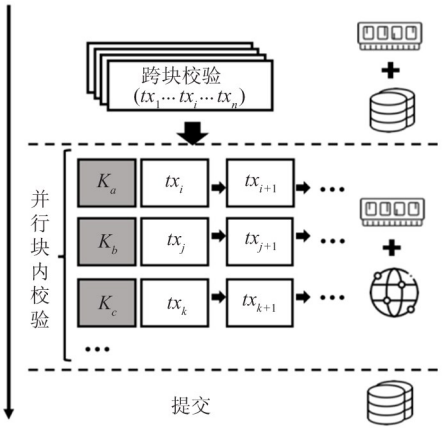

图 5　P-MVCC 示意图

算法 3　P- MVCC 算法流程

Input: 待校验区块 *blk*,状态数据库 *db*, 区块写集缓存 *cache*
Output: 区块内交易所有更新 *updates*

```
1 function VALIDATEBLOCK(blk)
2   updates ← newUpdates( )
3   depQ ← ANALYZEDEP(blk)
4   for tx in blk. txs do // in parallel
5       VLDCROSSBLK(tx, cache)
6   Sync all threads
7   PVLDINNERBLK(updates, depQ)
8   cache applies updates
9   return updates
10
11 function ANALYZEDEP(blk)
12   Initialize depQ
13   for tx in blk. txs do
```

```
14        for op in tx.reads and tx.writes do
15            depQ[op.key]←{tx, op}
16    return depQ
17
18 function PVLDINNERBLK(updates, depQ)
19    for k in depQ do//in parallel
20        for depQ[k] is not empty do
21            tx, op ← depQ[k].pop()
22            if tx is marked as invalid then
23                Continue
24            if op is READ then
25                Check if k is in updates, if in, mark tx as
                  invalid
26            if op is WRITE then
27                Wait for the verification of all reads in tx
28                updates applies op if tx is validated
29    Sync all threads
30    return
```

所有跨块校验线程执行结束后,P-MVCC 开始执行块内校验。为了剥离交易间的附加依赖,块内校验不再以交易为单位,而是以交易中所有读写集的 *key* 为单位并行校验。在块内校验之前,P-MVCC 会先对交易进行依赖分析(第 3 行),具体做法是:创建一个以交易中所涉及关键字为 *key*,以交易指针和具体读写操作的二元组列表为 *value* 的 *Map*(*depQ*),如图 4 所示;依次提取每笔交易中读写集所涉及的关键字,然后将每个关键字所对应{*tx*,*op*}二元组放到 *depQ* 所对应的列表中(第 13 ~ 16 行)。不难发现,*depQ* 中记录了交易间的所有偏序依赖关系,而彻底丢弃了交易间的附加依赖。基于 *depQ*,P-MVCC 展开块内校验。针对块内交易所涉及的每个关键字,分别起一个线程并行校验(第 19 行)。线程内部对操作队列按顺序分析,针对当前操作,如果该操作对应的交易已经被其他交易标记为无效,那么直接跳过当前操作;如果是读操作,则判断该 *key* 是否在 *updates* 中,即判断之前交易是否已对它修改过,如果修改过则标记对应该交易无效;如果是写操作,则等待该操作对应交易的所有读操作校验完成,如果该交易有效,则将该写操作添加到 *updates* 中,以记录该 *key* 的修改(第 26 ~ 28 行)。待所有线程校验完成,返回校验结果并且更新 *cache*,P-MVCC 执行结束。

4 实验评测

4.1 实验设置

4.1.1 实验平台

本文采用华为区块链作为实验平台。华为链由华为公司自主设计实现,是完全自研的国产联盟链系统,采用和 Hyperledger Fabric 相同的通用 EOV 架构,但是相比 Fabric,华为链在区块校验、背书策略校验、交易瘦身和去重等多环节进行优化。在实验中,本文将上文中提到的 3 种 MVCC 算法实现到华为链中,以 Fabric 的 MVCC 算法作为性能基准,表示为 V-MVCC (vanilla MVCC),使其与 C-MVCC 和 P-MVCC 的性能进行对比。目前本文提出的 2 种算法已正式应用于华为区块链平台,分别面向一般和分片场景。

4.1.2 工作负载和评测指标

本文使用业界流行的 SmallBank[11] 作为工作负载展开性能评测,SmallBank 有多种类型的银行业务交易。在实验开始时,SmallBank 创建一定数量的账户,每个账户会有账户余额和其他附加的数据,对应的 *value* 的大小为 2 kB。本文实验中使用相同的余额初始化每个账户,然后生成随机交易,在不同的账户之间转账,资产转移交易在 2 个账户间发生。在评测中,账户数量被设定为 10 万,转账交易的账户挑选符合随机分布。关于性能指标,本文关注区块 MVCC 校验的时延,因为它能直接反映出 MVCC 算法的执行效率。

4.1.3 参数设置

本节实验的软硬件参数如表 2 所示。每个节点独享一台物理机器,机器使用 Intel 8 核 Xeon(R) CPU,运行 Ubuntu 18.04 操作系统,拥有 16 GB 的物理内存。实验规模是 2 个组织,每个组织包含 2 个节点,一共 4 个节点。关于软件参数,由于 MVCC 校验几乎不受共识算法影响,因此实验中为了简化流程设置共识算法为 Solo;最后,设置 C-MVCC 和 P-MVCC 的缓存大小都为 100 个区块,这个缓存大小实际所需物理内存较小,可以大幅减少访问底层存

储引擎,实验中使用 RocksDB 作为底层存储引擎。

表 2　软硬件参数设置

参数	规格
OS	Ubuntu 18.04
CPU	Intel(R) Xeon(R) Gold 6161 CPU@2.20 GHz, 8 cores per CPU
内存	16 GB
共识算法	Solo
缓存大小	100 block

4.2　一般场景评测

本文首先对一般场景下的性能表现展开评测。EOV 执行模型下,区块整体的 MVCC 校验结束后,统一对区块内交易提交落盘,因此实验中以一个区块校验完成的时延来进行对比。在该场景下,设置区块内分别包含 100 ~ 1 600 不同的交易数量。现实场景中,一定程度地提升区块大小可以提升区块共识的效率,而过高的区块 batch 大小并不会给系统整体带来明显性能提升[12-13],因此本实验中只测试到 1 600 个交易每区块。

图 6 所示是一般场景下各校验算法性能表现对比。首先整体来看,3 种算法的校验延迟随区块内交易数量的增加是一起增长的,但是 3 种算法的增长幅度却差异很大:V-MVCC 时延和区块内的交易数量成正比,P-MVCC 次之,C-MVCC 增长最为缓慢。这种情况的原因主要是并行化和缓存的引入。P-MVCC 和 C-MVCC 在跨块校验时可以大幅减少底层 IO的操作次数,并且,由于跨块校验的并行化,

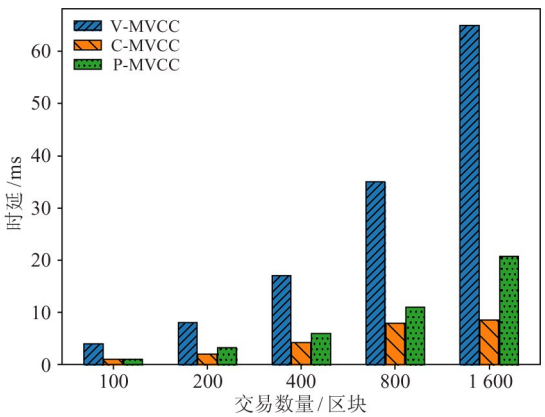

图 6　一般场景下校验效率表现对比

即使需要底层 IO 也不需要串行等待,因此两者相比 V-MVCC 校验时延增长要缓慢。其次可以观察到 C-MVCC 的校验时延在块内交易由 800 增加到 1 600 时仅增加了约 2 ms,其背后的原因是,跨块校验已并行化,这一部分几乎没有差异,增长的时延仅来自于块内校验;然而由于块内校验是在内存中进行,因此该部分运行效率非常高,所以只会带来极少的时延。

可以观察到本文提出的 C-MVCC 和 P-MVCC 的时延差异随区块大小逐渐增大,这是由于 P-MVCC 在进行块内校验前需要串行分析交易间的依赖,这一部分开销相对于块内校验的并行化所带来的收益要大得多,这也证明了在一般的场景下,C-MVCC 的性能要比 P-MVCC 更好(P-MVCC 本身提出是为了提升分片等场景的本地校验效率,而非一般场景)。最后将本文提出的 2 个算法和 V-MVCC 的效率作对比,在1 600个交易每区块的情况下,C-MVCC 和 P-MVCC 分别是 V-MVCC 的 7.4 和 3.2 倍,这证明了本文提出的 2 个算法在一般场景下的有效性。

4.3　分片场景评测

本小节展开分片场景下的测试,该场景主要分析 P-MVCC 有效性。分片交易比例在该场景下是对算法校验时延影响的最主要因素,因此本文测试不同分片交易比例的情况下的性能表现,设置每个区块中有 500 个交易。需要强调的是,本节中的分片场景是模拟的,跨片交易需在本地模拟随机等待 1 ~5 ms,进而模拟与其他分片同步校验的网络延迟,这种模拟方法与真实的分片场景相差无几,可以反映真实分片下的效果。

图 7 是分片场景下的测试结果。可以发现 P-MVCC 相较于 V-MVCC 和 C-MVCC 的优势是巨大的,而 V-MVCC 和 C-MVCC 之间的时延差异一直相对稳定。造成这种现象的原因是跨分片交易的块内校验已经成为了整个校验步骤的瓶颈。由于跨分片交易需要等待一些时间来获取其他分片的校验结果,而 V-MVCC 和 C-MVCC 则是在该步骤串行等待之前所有交易的校验结束,因此导致整体的校验性能很差。反观 P-MVCC,由于其对交易间依赖分析后拆解成多个短队列并行校验,此时跨片交易造成

的累积时延大幅降低。

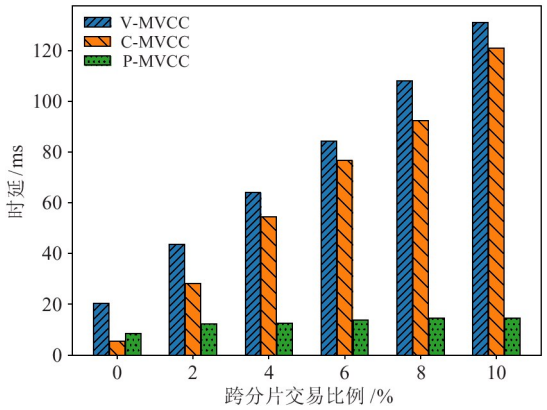

图 7　分片场景下校验效率表现对比

P-MVCC 的校验时延随跨片比例的增加并没有明显变化,这是由于 P-MVCC 多队列并行校验,最终的时延取决于最长的队列,而这个最长队列的长度受跨片比例的影响并不大(因为 *key* 随机分布),所以时延也不会明显增加。最后在 10% 跨片比例的情况对比 P-MVCC 和 V-MVCC 的效率差异,可以发现 P-MVCC 的校验效率提升了约 9 倍,这证明了 P-MVCC 在分片场景下的有效性。

5　相关工作

5.1　EOV 联盟链性能分析

文献[14]对 Fabric 进行了全面的性能评测和分析,发现了 Fabric v1.0 中的瓶颈,并且从系统、网络设置和上层应用程序设计角度给出指南以期望用户获得更高吞吐量。文献[15]分析了不同工作负载下 Fabric v1.4 的延迟和吞吐量以评估其性能。通过改变某些基准参数,例如链码参数、事件负载大小和链码调用次数等对 Fabric 评测。文献[16]在敌手设置下评估了 Hyperledger Fabric 的性能。通过模拟不同的敌手行为并测量区块链的性能同时建议如何在这种恶意行为下提高系统性能。文献[17]采用分层建模方法,对 Fabric v1.4 中的交易处理性能进行了建模。基于背书和区块超时约束提出了一种方法,可用于估计系统的性能,该模型经过广泛的仿真和数值分析验证。

5.2　EOV 联盟链性能提升

文献[12]利用数据库领域中的思想,通过分析交易间的依赖性,构建交易的依赖图,通过删减依赖图中特殊挑选的交易,减少整体被丢弃的交易数量。文献[13]扩展了文献[12]的工作,在排序之前中止不可序列化的交易,并通过重新排序来完全避免块内的交易中止。以上这两者的共同目标都是降低丢弃率,从而整体上提升系统的吞吐。文献[18]提出了 FastFabric,它包括各种优化,例如用内存哈希表替换状态数据库;将区块存储在单独的服务器中等手段使得吞吐达到 20 000 TPS(transaction per second)。文献[19]在 XOX Fabric 中扩展了 FastFabric,引入了交易的排序后执行,以减少交易的丢弃率,提升系统的整体吞吐。文献[20]给 Fabric 的账本节点增加了流水线校验提升隔离性校验的并行度,并且引入了稀疏节点的概念,其思想是账本节点只记录和服务部分合约,类似于分片但是又不完全一致,因为它只有一个共识服务。这些工作基本都可以跟本文工作结合来进一步提升 EOV 联盟链的系统性能。

6　结　论

本文聚焦于 EOV 架构的联盟链交易 MVCC 校验的效率提升。Hyperledger Fabric 作为当前最具代表性和影响力的联盟链系统,本文以其为例分析了当前 EOV 架构下联盟链 MVCC 目前存在的问题,主要是并行化程度低以及过于频繁的底层数据库访问操作导致的潜在性能下降。针对以上问题,本文提出了 C-MVCC 算法,为 MVCC 校验过程增加了内存缓存,减少了与底层存储系统的交互,并且将块内校验和跨块校验剥离,实现了跨块校验的完全并行化。进一步地,为更好地适配分片等对于块内校验时延敏感的高并发场景,本文提出了 P-MVCC,进一步提升了块内校验的并行度。在 P-MVCC 中,分析交易间的附加依赖和偏序依赖,在保证校验正确性的前提下,提升交易间校验并行度,并且完全剥离了交易校验过程中的附加依赖,从而能更好地适应分片场景,释放分片的性能提升潜力。

最后,本文在华为链中实现了 C-MVCC 和 P-MVCC,将两者与 Fabric 中的 MVCC 算法进行对比,实验证明了两者对 MVCC 校验效率的显著提升效果。

参考文献

[1] 佟兴,张召,金澈清,等. 面向端边云协同架构的区块链技术综述[J]. 计算机学报, 2021,44(12):2345-2366.

[2] 蔡晓晴,邓尧,张亮,等. 区块链原理及其核心技术[J]. 计算机学报, 2021,44(1):84-131.

[3] NAKAMOTO S. Bitcoin: a peer-to-peer electronic cash system[EB/OL]. [2023-04-09]. https://bitcoin.org/bitcoin.pdf.

[4] BUTERIN V. Ethereum: a next-generation smart contract and decentralized application platform[EB/OL]. [2023-04-09]. https://github.com/ethereum/wiki/wiki/White-Paper.

[5] ANDROULAKI E, BARGER A, BORTNIKOV V, et al. Hyperledger Fabric: a distributed operatingsystem for permissioned blockchains[C]//Proceedings of the 13th EuroSys Conference. New York, USA:ACM, 2018:1-15.

[6] 袁勇,王飞跃. 区块链技术发展现状与展望[J]. 自动化学报, 2016,42(4):481-494.

[7] 中国信通院. 区块链白皮书(2022 年)[EB/OL]. [2023-05-26]. http://www.caict.ac.cn/kxyj/qwfb/bps/202212/P020230105572446062995.pdf.

[8] AMIRI M J, AGRAWAL D, EL-ABBADI A. SharPer: sharding permissioned blockchains over network clusters[C]//International Conference on Management of Data. New York, USA: ACM, 2021:76-88.

[9] DANG H, DINHTT A, LOGHIN D, et al. Towards scaling blockchain systems via sharding[EB/OL]. (2019-03-12) [2023-05-26]. https://arxiv.org/pdf/1804.00399.pdf.

[10] HELLINGS J, SADOGHI M. Byshard: sharding in a byzantine environment[J]. Proceedings of the VLDB Endowment, 2021,14(11):2230-2243.

[11] Hyperledger foundation. Hyperledger caliper ben-chmarks[EB/OL]. [2023-04-09]. https://github.com/hyperledger/caliper-benchmarks/tree/master/benchmarks/scena- rio/smallbank.

[12] SHARMA A, SCHUHKNECHT F M, AGRAWAL D, et al. Blurring the lines between blockchains and database systems: the case of Hyperledger Fabric[C]//Proceedings of the 2019 International Conference on Management of Data. New York, USA: ACM, 2019:105-122.

[13] RUAN P, LOGHIN D, TA Q T, et al. A transactional perspective on execute-order-validate blockchains[C]//Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. New York, USA: ACM, 2020:543-557.

[14] THAKKAR P, NATHAN S, VISWANATHAN B. Performance benchmarking and optimizing Hyperledger Fabric blockchain platform[C]//2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). Milwaukee, USA: IEEE, 2018:264-276.

[15] BALIGA A, SOLANKI N, VEREKAR S, et al. Performance characterrization of Hyperledger Fabric[C]//2018 Crypto Valley Conference on Blockchain Technology (CVCBT). Zug, Switzerland: IEEE, 2018:65-74.

[16] WANG S. Performance evaluation of Hyperledger Fabric with malicious behavior[C]//Blockchain ICBC 2019 2nd International Conference. San Diego, USA: Springer, 2019:211-219.

[17] JIANG L, CHANG X, LIU Y, et al. Performance analysis of Hyperledger Fabric platform: a hierarchical model approach[J]. Peer-to-Peer Networking and Applications, 2020,13:1014-1025.

[18] GORENFLO C, LEE S, GOLAB L, et al. FastFabric: scaling Hyperledger Fabric to 20 000 transactions per second[C]//2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). Seoul, Korea: IEEE, 2019:455-463.

[19] GORENFLO C, GOLAB L, KESHAV S. XOX Fabric: a hybrid approach to transaction execution[EB/OL]. (2020-03-09) [2023-05-26]. https://arxiv.org/pdf/1906.11229.pdf.

[20] THAKKAR P, NATARAJAN S. Scaling blockchains using pipelined execution and sparse peers[C]//Proceedings of the ACM Symposium on Cloud Computing. New York, USA: ACM, 2021:489-502.

High-efficiency isolation verification methods for EOV consortium blockchain

XUE Han, BAO Yungang
(Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190)
(University of Chinese Academy of Sciences, Beijing 100049)

Abstract

Blockchain systems such as Hyperledger Fabric that adopts the execute-order-validate (EOV) transaction model usually use the multi-version concurrency control (MVCC) method to validate transactions. The basic idea is to check whether the read set of the current transaction has been modified by cross-block and intra-block transactions before committing, to ensure transaction isolation and correctness. However, MVCC often becomes the performance bottleneck of the entire transaction due to low parallelization and frequent access to the underlying database system. To solve this problem, this paper proposes two new MVCC algorithms: C-MVCC and P-MVCC. C-MVCC uses memory as a cache to reduce access to the underlying database system during MVCC validation, and introduces a parallelization mechanism for the cross-shard validation stage. P-MVCC further decomposes the dependency relationship between transactions based on C-MVCC, and realizes parallelization of cross-block and intra-block validation, thus eliminating the accumulation of high cross-shard validation latency in sharding scenarios. Experimental results show that C-MVCC can bring 7.5 times improvement in validation efficiency compared with the current algorithm in general scenarios, while P-MVCC can bring 9.0 times improvement in validation efficiency in sharding scenarios.

Key words: blockchain, multi-version concurrency control(MVCC), cache, parallelization