计算机科学
COMPUTER SCIENCE

许可链下的事务并行执行模型

董昊, 赵恒泰, 王子尧, 袁野, 张奥千

引用本文

董昊, 赵恒泰, 王子尧, 袁野, 张奥千. 许可链下的事务并行执行模型[J]. 计算机科学, 2024, 51(1): 124-132.

DONG Hao, ZHAO Hengtai, WANG Ziyao, YUAN Ye, ZHANG Aoqian. Parallel Transaction Execution Models Under Permissioned Blockchains [J]. Computer Science, 2024, 51(1): 124-132.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.230800201

许可链下的事务并行执行模型

董　昊[1]　赵恒泰[2]　王子尧[2]　袁　野[1]　张奥千[1]
1 北京理工大学计算机学院　北京 100081
2 东北大学计算机科学与工程学院　沈阳 110169
(donghaomail@foxmail.com)

摘　要　现有的许可链系统大多采取串行的事务执行方式,无法利用多核处理器的性能优势。在共识算法性能较高的许可链中,这种串行的事务执行方法将会成为性能瓶颈。为降低排序-执行-验证架构的许可链中事务执行的时间开销,文中提出了两种事务并发模型。首先,提出了基于地址表的并行执行模型,通过静态分析的方法将事务的读写集映射到地址表中,并利用地址表构建调度图实现无数据冲突的事务并行执行;其次,针对静态分析方法不适用于读写需求复杂的应用场景,提出了基于多版本时间戳排序的并行执行模型,领导者节点使用多版本时间戳排序算法并行地预执行事务并将调度图以事务依赖三元组的形式存储入区块,所有验证节点通过事务依赖三元组进行调度,在保证一致性的前提下实现事务的并行执行;最后,在Tendermint中实现了所设计的两种事务并发模型,并进行了事务执行阶段性能测试和多节点性能测试。实验结果表明,相比串行执行,所提模型在单节点8线程时的事务执行时间分别减少了68.6%和28.5%,4节点8线程时区块链吞吐量分别提升了约43.4%和19.5%。

关键词:区块链;实用拜占庭容错;事务并发;多版本时间戳排序;Tendermint
中图分类号　TP311

Parallel Transaction Execution Models Under Permissioned Blockchains

DONG Hao[1],ZHAO Hengtai[2],WANG Ziyao[2],YUAN Ye[1] and ZHANG Aoqian[1]
1 School of Computer Science,Beijing Institute of Technology,Beijing 100081,China
2 School of Computer Science and Engineering,Northeastern University,Shenyang 110169,China

Abstract　Most existing permissioned blockchain systems adopt serial transaction execution methods,which cannot take advantage of the high performance of multi-core processors. This serial method will be a performance bottleneck in permissioned block-chains with high performance consensus algorithms. To reduce execution time of transactions in permissioned block-chains with order-execute-validate architecture,two transaction concurrency models are proposed. First,an address table-based parallel execution model is proposed that maps the read and write sets of transactions to the address table through static analysis and constructs a scheduling graph using the address table to achieve parallel execution of transactions without data conflicts. Second,a parallel execution model based on a multi-version timestamp ordering algorithm is proposed,in which the leader node uses a multi-version timestamp ordering algorithm to pre-execute transactions in parallel and stores the scheduling graph into the block in the form of transaction dependency triplets. All validation nodes schedule via transaction dependency triplets to achieve parallel execution of transactions under the premise of consistency. Finally,the two parallel transaction execution models designed in this paper are implemented in Tendermint,and a performance experiment during the transaction execution phase and a performance experiment with multiple nodes are conducted. Experimental results show that the above models reduce the transaction execution time by 68.6% and 28.5% with a single node and 8 threads,and increase the blockchain throughput by about 43.4% and 19.5% with 4 peer nodes and 8 threads per node,respectively.

Keywords　Blockchain,Practical Byzantine fault tolerance,Transaction concurrency,Multi-version timestamp ordering,Tendermint

1　引言

区块链是一种分布式账本,由互不信任的节点网络共同维护。通过哈希链技术和共识协议,可以保证区块链中的数据是不可变且无法篡改的[1]。根据准入机制,区块链可以分为公有链和许可链。以比特币[1]和以太坊[2]为代表的公有链

到稿日期:2023-08-31　返修日期:2023-10-25
基金项目:国家重点研发计划(2022YFB2702100);国防基础科研计划(JCKY2021211B017)
This work was supported by the National Key R&D Program of China(2022YFB2702100) and Defense Industrial Technology Development Program(JCKY2021211B017).
通信作者:张奥千(aoqian.zhang@bit.edu.cn)

没有身份审核机制,任何节点都可以自由加入和退出。为了防止节点作恶,公有链常使用效率较低的工作量证明(Proof of Work,PoW)或权益证明(Proof of Stake,PoS)等共识算法。区别于公有链,许可链通过牺牲一定的去中心化性来引入身份认证和权限控制功能。由于数量有限的节点间相互存在一定的信任基础,Hyperledge Fabric[3],Tendermint[4]等许可链使用更高效的实用拜占庭容错算法[5](Practical Byzantine Fault Tolerance,PBFT)来维护数据的一致性。

目前,许多许可链系统采用排序-执行-验证(Order-Execute-Validate,OEV)的执行架构[3]。该架构中,从一批事务被广播至网络再到最终某个区块被提交,称为一轮共识。在Tendermint等许可链中,每轮的事务排序由领导者节点负责,它由对等节点选举产生。除领导者节点之外,参与该轮共识的其他节点被称为验证节点,通过共识投票等方式共同维护一致性的数据副本。如图1所示,每轮共识开始于一批事务从客户端提交至领导者节点。在排序阶段,领导者节点将这些事务排序成事务集并原子广播至各验证节点;在执行阶段,各验证节点在本地执行由领导者节点排序的事务集,然后互相广播事务执行的结果;在验证阶段,所有对等节点通过网络中广播的执行结果确定本轮所共识的区块,并验证其正确性。最后,所有对等节点通过网络中广播的验证阶段的验证结果来决定是否提交区块。综上所述,一轮共识主要由共识算法和事务执行组成。

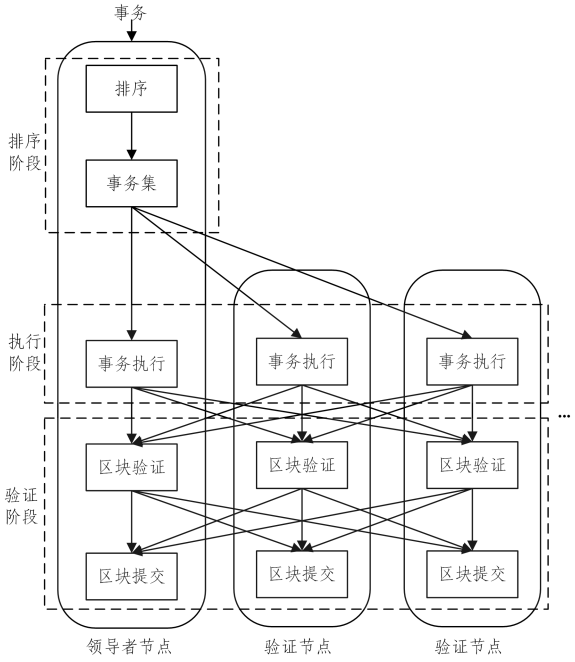

图1　许可链的排序-执行-验证执行架构

Fig. 1　Order-Execute-Validate architecture of permissioned blockchains

近年来,许多致力于提升许可链共识算法性能的工作都取得了巨大进展,例如,Tendermint处理250字节的事务时,共识算法的吞吐量(不执行事务)能达到10 000事务每秒(transactions per second,tps)以上[6]。单领导者节点的区块链结构容易受到单节点网络带宽的限制,一些工作[7-9]设计了由多个领导者节点并发提出区块的共识算法,这些领导者节点共同参与构建了有向无环图(Directed Acyclic Graph,DAG)形式的区块链结构。

然而,为了确保一致性,上述许可链在事务执行阶段依然采用串行的执行模型。受限于网络环境、安全性需求等问题,这种串行执行模型无法充分利用多核处理器的高性能优势,很难进一步提升共识算法的效率,成为了许可链共识的性能瓶颈。为了解决这一问题,本文针对OEV架构的许可链设计了两种并行执行模型。

本文的主要贡献如下:

1)提出了一种基于地址表的并行执行模型,该模型通过静态分析工具提前获取事务的读写集并将其映射到地址上,形成地址表。任何节点根据地址表都可以构建出与串行执行等价的并发调度,从而实现节点内的事务并行执行。

2)为了执行具有复杂读写需求的事务,提出了一种基于多版本时间戳排序(Multi-Version Timestamp Ordering,MVTSO)的并行执行模型。在该模型中,领导者节点尝试性地执行区块中的事务并将冲突事务重排序,然后通过事务依赖三元组在区块中存储事务的依赖关系,验证节点根据事务依赖三元组在保证一致性的前提下并行执行事务。

3)基于Tendermint实现了上述两种模型。性能测试结果显示,相较于串行执行,基于地址表的并行执行模型和基于多版本时间戳排序的并行执行模型在单节点测试中使用8线程时,执行10000事务的时间分别减少了68.6%和28.5%;4节点共识时,上述模型使用8线程时吞吐量分别提升了43.4%和19.5%。

2　相关工作

现有工作使用静态分析或动态分析的方法,在以太坊等公有链上实现了事务并行执行,这些方法提升了单节点内的事务执行速度。

静态分析方法通过由开发人员指定共享数据对象或者静态编译的方式获取事务的读写集。企业级区块链底层平台FISCO BCOS[1)]要求开发者在编写事务时罗列出所需访问的共享数据对象,并根据共享数据对象判断事务间的冲突关系。ES-ETH[2)]等静态分析工具提前通过编译的方式获取事务的读写集,从而构建事务间的依赖关系。Yu等[10]根据事务的依赖关系,将不存在直接或间接依赖关系的事务划分到不同的资源互斥分组,从而实现了资源互斥分组间的事务并行执行。Bartoletti等[11]给出了区块中事务的符号化定义,将智能合约事务的执行符号转化为状态转换函数,从而证明了静态分析方法的理论可行性。Bartoletti等使用petri-net进行调度,在以太坊虚拟机(Ethereum Virtual Machine,EVM)中实现了他们的静态分析模型。上述工作中,事务间的依赖关系是通过直接比较事务读写集生成的,这种做法具有$O(n^2)$的时间复杂度,随着区块中事务的数量增加,算法性能将显著下降。

动态分析方法中,领导者节点和验证节点使用相同的并发控制算法并行地执行事务。由于并发控制算法具有随机

1) https://fisco-bcos-documentation.readthedocs.io/en/latest/docs/introduction.htm.

2) https://github.com/DiegoMarcia/ES-ETH

性，领导者节点必须预执行所有事务，并将事务间的依赖关系存储在区块中，以确保验证节点间并行执行结果的一致性。Dickerson 等[12]通过抽象锁和回退日志进行并发调度，利用抽象锁的计数功能构建 happen-before 图，并将其编码为 fork-join 程序记录进区块。Pang 等[13]通过在区块中记录写链来传递事务间的依赖关系与执行顺序。Zhang 等[14]将事务集划分为不同的事务子集，将跨事务子集的读写数据通过一致性写集记录在区块中。Anjana 等[15]使用软件事务内存(Software Transactional Memory，STM)的方法并发地执行事务，将依赖事务与无依赖事务分别存储，降低了依赖关系的存储开销。在单版本存储中，Anjana 等的执行模型可以将依赖关系的存储开销降低至区块大小的约 17.71%。在 STM 的基础上，Anjana 等[16]开发了支持增删改查的高效的对象语义框架，在这个框架下，矿工使用乐观哈希表记录基于对象的软件事务内存(Object-based Software Transactional Memory，OSTM)，以实现并发调度。为了提升效率，Anjana 等提出使用多版本的 OSTM。此外，Anjana 等[16]还提出了一种分散的智能多线程验证器，使验证线程能相对独立地处理事务。实验结果证明，这种去中心化的多线程并行验证方法是有效的。

上述工作大多针对公有链进行设计与实现，仅测试了单节点在事务执行阶段的加速比，没有讨论多节点共识的过程中事务并行执行对吞吐量的具体影响。以太坊等公有链中，为了防止分叉，出块速度会设置较慢，事务并行执行并不能带来显著的性能提升。本文以 OEV 架构的许可链为基础，设计了许可链下的事务并行执行模型，并着重探讨了事务并行执行对许可链共识性能的影响。

3 许可链下的事务并行执行模型

为了提升许可链事务执行阶段的性能，本章提出了基于地址表的并行执行模型(见 3.2 节)。基于地址表的并行执行模型要求客户端在生产事务时预先指定事务对共享资源的访问情况，或者事务所访问的共享资源可以通过一定的规则编译得到，这种做法限制了事务的表达性。为了适应具有复杂读写需求的应用场景，本章提出了基于多版本时间戳排序的并行执行模型(见 3.3 节)。

基于地址表的并行执行模型使用静态分析的方法分析事务的读写集，从而确定事务间的冲突关系，并根据事务在事务集中的顺序确定性地生成事务集总的并发调度顺序。领导者节点可以在广播区块后和验证节点的同时通过上述并行执行流程执行事务集，因此该模型具有较高的并发性。

基于多版本时间戳排序的并行执行模型使用了动态分析的方法实现冲突可串行化的并发调度。为了确保一致性，领导者节点必须预执行事务并在区块中存储调度信息。虽然该模型并发性低于基于地址表的并行执行模型，但是它无需通过静态分析的方法提前获取事务的读写集并分析依赖关系，因此可以用来执行包含嵌套智能合约、指针运算等具有复杂读写操作的事务。

3.1 定义和概念

本节给出了文中涉及的一些定义及概念。

定义 1(地址) 地址是可以唯一确定一个数据单元的字符串，且任何数据单元只能有一个地址。

定义 2(依赖关系) 假设事务 T_1 在事务集中的序号小于 T_2 的序号，若 T_1 的读集和 T_2 的写集存在公共地址，则称 T_1 与 T_2 存在读写依赖；若 T_1 的写集和 T_2 的读集存在公共地址，则称 T_1 与 T_2 存在写读依赖；若 T_1 的写集和 T_2 的写集存在公共地址，则称 T_1 与 T_2 存在写写依赖。

定义 3(调度图) 调度图(Scheduling Graph，SG)是一个有向无环图，其节点表示事务集中的事务，任意两个存在依赖关系的事务在调度图上存在一条有向边，有向边的方向从序号小的事务指向序号大的事务。

定义 4(地址表) 地址表是一个二维数组，数组的每一行对应一个地址。地址表的每一行存储若干个读或写标签，这些标签根据其事务在事务集中的序号从小到大排列。

3.2 基于地址表的并行执行模型

许可链高效的共识算法使得生成包含数千甚至数万事务的区块成为可能，这些区块的事务集中，事务间潜在的依赖关系数随事务数量的增加呈幂律增长，传统的静态分析方法常通过两两事务比较读写集的方式来判断事务间的依赖关系，但这种方式不适用于事务吞吐量较高的许可链。受 Xiao 等工作的启发[17]，本文将事务的读写集映射至地址，提出了名为地址表的数据结构，并在许可链下提出了基于地址表的并行执行模型。由于基于地址表的并行执行模型在执行事务前提前获取事务间的依赖关系，并以此为依据进行并发调度，因此它是一种静态分析模型。

如图 2 所示，基于地址表的并行执行模型通过以下流程完成一轮共识。

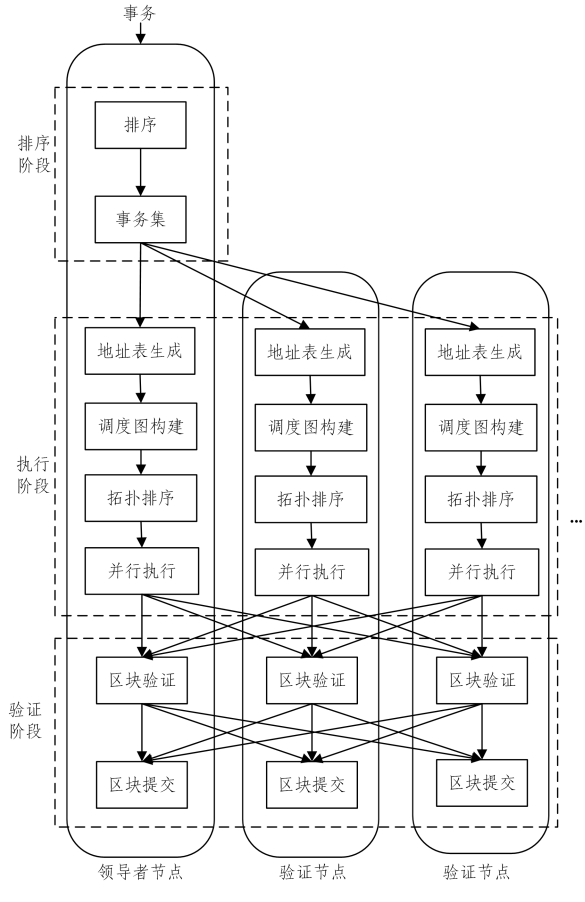

图 2 基于地址表的并行执行模型

Fig. 2 Parallel execution model based on address table

排序阶段，领导者节点从客户端接收一批事务并排序为

事务集,然后将事务集广播至各验证节点。执行阶段,领导者节点与验证节点通过以下流程确定性地生成事务并行执行顺序:1)静态分析事务的读写集,将事务的读写集映射至地址表上,然后基于地址表构建调度图(见 3.2.1 节);2)在调度图上使用拓扑排序算法生成一个与串行执行等价的并发调度,然后并行地执行事务(见 3.2.2 节);3)对等节点间相互广播执行结果;4)模型通过验证阶段就本轮应提交的区块达成共识。

3.2.1　基于地址表的调度图构建

事务执行阶段,存在依赖关系的事务不可以同时执行,否则可能产生冲突不可串行化的调度。调度图是记录事务依赖关系的一种方式,本节通过算法 1 实现了地址表的高效构建。算法 1 使用静态分析工具在执行前获取事务的读写集并将其映射为地址表上的标签。根据定义,地址表上的标签根据事务的序号从小到大排列,因此根据地址表生成的调度图中,有向边总是从序号小的事务指向序号大的事务。

算法 1　基于地址表的调度图构建算法

输入:事务集 R

输出:调度图 SG

1. 初始化地址表 L
2. 并行地使用静态分析工具分析 R 中事务的读写集,生成相应的读或写标签,若某事务在一个地址上同时具有读和写标签,则仅生成该地址上的写标签
3. 遍历 R 中的每一个事务 T
4. 　将 T 的读、写标签加入 L 的相应地址上
5. 遍历 L 中的每一行
6. 初始化前序读事务集 RL 和前序写事务集 WL
7. 　遍历行中的每一个标签 l
8. 　　根据 WL,RL 在 SG 中生成有向边
9. 　　根据 l 更新 WL 和 RL 的内容

算法 1 使用读事务集 *RL* 和写事务集 *WL* 记录标签状态,避免了冗余的回溯。算法第 8 行根据 *WL* 和 *RL* 判断标签 *l* 对应的事务的依赖关系,并为每个依赖关系在调度图中添加相应的有向边。算法第 9 行根据 *l* 的内容更新 *WL* 和 *RL* 的内容,具体来说:1)如果 *l* 是写标签,则清空 *RL* 和 *WL*(因为 *l* 之后的事务读写标签都可以通过 *l* 与这些事务产生间接的依赖关系),然后将 *l* 加入 *WL* 中;2)如果 *l* 是读标签,则将 *l* 加入 *RL* 中。

算法 1 将事务读写集映射到地址表上,避免了额外的两两事务间的比较。地址表中,只有在同一个地址上具有读写标签的事务间才会进行依赖关系的判断。由于事务的平均读写集大小是与区块中事务的数量无关的常数,因此算法 1 减少了判断读写依赖所需的计算次数。此外,算法 1 剪枝了具有间接依赖关系的事务间的直接依赖关系判断,进一步维护了调度图的稀疏性。由算法 1 可知,任意两个具有依赖关系的事务在调度图上都会存在至少一条通路。因此,算法 1 生成的调度图能正确记录事务集中的依赖关系。

3.2.2　基于拓扑排序的调度方法

调度图中,有向边两端的事务存在依赖关系,不可以同时执行。并行执行过程中,有向边出端的事务必须等待入端的事务提交后才能开始执行。算法 1 生成的调度图是个有向无环图,因此可以使用拓扑排序的方法为调度图上的事务确定一个并发调度顺序。事务执行阶段,通过算法 1 生成调度图后,可以使用算法 2 在调度图上通过拓扑排序并行地执行事务。

算法 2　基于拓扑排序的调度算法

输入:调度图 SG

输出:无

1. 并行执行 SG 中入度为 0 的事务
2. 循环执行以下步骤,直到 SG 被清空
3. 　等待某个事务 T 提交
4. 　删除 SG 中 T 对应的节点及其关联边
5. 　并行执行 SG 中因步骤 4 入度变为 0 的事务

算法 2 省略了事务的具体执行流程。调度图保证了同时执行的事务间不存在依赖关系,因此事务的执行不需要任何额外的并发控制,即可实现并发调度的冲突可串行化。

3.3　基于多版本时间戳排序的并行执行模型

基于地址表的并行执行模型具有较高的并发性,然而静态分析工具对事务模型的设计者有较高的要求[18],因此其不适用于有复杂读写需求的应用场景。对于无法给出静态分析工具的应用程序,受到多版本并发控制[19]的启发,本文在 OEV 架构的许可链下提出了一种基于多版本时间戳排序算法的并行执行模型,如图 3 所示。该模型使用动态分析的方法分析事务的依赖关系。

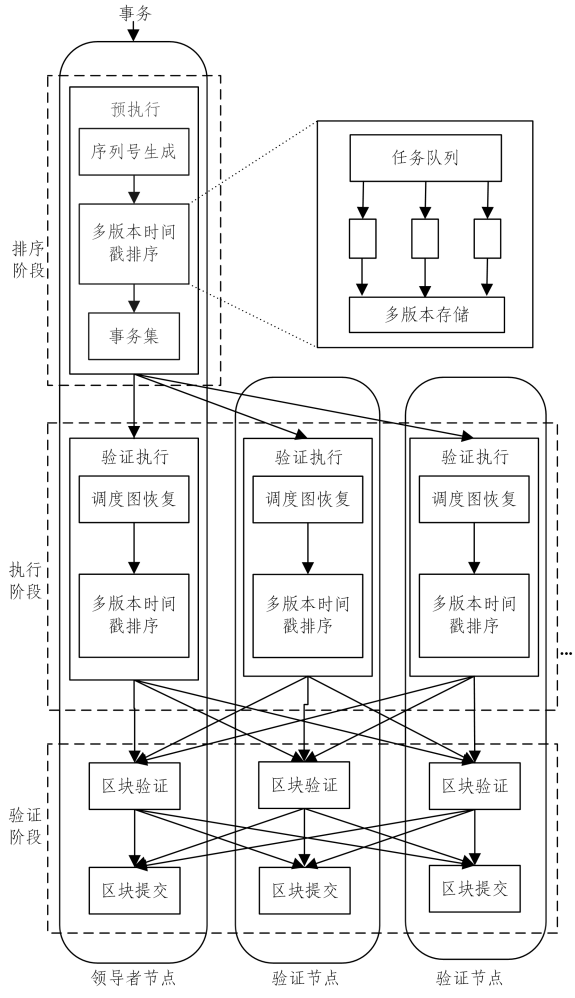

图 3　基于多版本时间戳排序的并行执行模型

Fig. 3　Parallel execution model based on MVTSO

不同于在执行前提前获取事务读写集的静态分析，动态分析首先通过传统数据库中的并发控制算法试探性地执行事务，生成一个冲突可串行化的并发调度，并根据执行结果生成事务的读写集。由于并发算法的执行结果具有随机性，领导者节点需要在区块中传输模拟执行过程中产生的调度图，验证节点利用调度图进行并发调度，以确保能够得到与主节点相同的执行结果。

在基于多版本时间戳排序的并行执行模型中，所有节点内部都设置一个工作线程池，工作线程池通过任务队列与调度线程进行通信，使用多版本时间戳排序算法在多版本存储上对从任务队列中获取的事务进行并发的事务执行(见 3.3.1 节)。排序阶段，领导者节点首先为事务集分配序列号，然后通过任务队列向工作线程池分配任务，间接地使用多版本时间戳排序并行地预执行事务。事务的排序结果以及事务间的依赖关系以事务依赖三元组的形式广播至各验证节点。执行阶段，验证节点根据领导者节点广播的事务依赖三元组恢复领导者节点的调度图，根据调度图进行并发调度，通过在任务队列中发布任务的方式，并行地执行并验证事务。执行结束后，节点将执行结果广播至网络中。验证阶段中，所有对等节点就本轮应提交的区块达成共识，最后提交区块。

3.3.1 基于多版本时间戳排序的并发读写

本小节介绍工作线程使用多版本时间戳排序算法在多版本存储上进行并发读写的过程。本文使用的多版本存储是一种键值存储模型，每个主键维护多个数据版本，这些数据版本通过版本号(时间戳)进行区分。此外，每个主键都具有一个最大读时间戳(Read Timestamp，RTS)，用于标识在该数据单元上进行过读取的事务的最大序列号。工作线程每次从任务队列中取出一个任务，任务的内容包含一个事务 T 及其序列号 tid_T，然后通过以下 3 个步骤完成事务的执行。

1)读写集生成。工作线程在多版本存储中进行事务执行，生成事务的读写集。事务进行读取时，工作线程首先根据 tid_T 判断是否更新对应主键的 RTS，然后读取满足以下条件的数据版本：(1)数据版本的版本号不大于 tid_T；(2)数据版本的版本号是所有满足条件(1)的数据版本中的最大值，并将该数据的版本号存入事务的读集中。事务进行写入时，采用延迟写策略，数据被暂存于线程本地的写集中。

2)冲突消解。冲突消解阶段，工作线程将事务的写集写入多版本存储中，然后检测事务是否会导致不可串行化的数据冲突，并根据结果选择中止事务或提交事务。冲突消解的步骤如下：(1)在多版本存储中为写集中的每一个键值对在多版本存储中创建一个版本号为 tid_T 的数据版本。若某次写入时，主键的 RTS 大于 tid_T，则认为事务 T 执行了“过时的写”，可能导致其他事务的读取结果变得不正确，因此将事务 T 的状态设置为“已中止”，回滚事务 T 的所有写入，并向调度线程反馈这一情况。(2)阻塞等待事务 T 的读集中所有依赖事务提交。如果 T 的某个依赖事务因冲突中止，则事务 T 的读集中将存在脏数据，因此工作线程将事务 T 的状态设置为“已中止”，回滚事务 T 的所有写入，并向调度线程反馈这一情况；如果 T 的所有依赖事务的执行状态均被设置为“已提交”，则进入排序确认阶段。

3)排序确认。排序确认阶段，工作线程将事务 T 的状态设置为“已提交”，其写入将变得不可更改。事务 T 提交后，工作线程将 T 的序列号 tid_T 以及 T 的读集中的最大版本号 D_T(如果读集为空则 $D_T=-1$)返回至调度线程。

事务写入时，不会覆盖原始数据或者其他事务的写入，而是创建一个新的版本，因此事务间不存在写写依赖；事务读取时，不会读取版本号高于自身序列号的数据版本，因此事务间不存在读写依赖。因此，多版本存储中，事务的调度图中仅记录由 D_T 表示的写读依赖。

3.3.2 两阶段并发调度

基于多版本时间戳排序的并行执行模型中，事务的执行分为预执行阶段和验证阶段。领导者节点在排序阶段预执行事务，生成了一个冲突可串行化的调度顺序，并将事务根据调度顺序进行排序，这个过程称为预执行阶段。执行阶段，验证节点根据排序结果执行事务集，这个过程称为验证执行阶段。为了确保验证执行阶段的执行结果与预执行阶段的执行结果一致，区块中需要存储预执行阶段生成的调度图。Shi 等指出[18]，区块中的调度图的存储开销是无法忽视的。调度图占用过多区块存储资源，将会在区块广播过程中带来额外的通信开销，从而降低共识算法的效率，因此对调度图的存储优化非常重要。鉴于此，本节提出的算法通过一种低存储开销的数据结构——事务依赖三元组来存储调度图。

1)预执行阶段。领导者节点进入预执行阶段，通过算法 3 预执行事务集，并为因冲突中止的事务分配新的序列号。

领导者节点以三元组$\langle T, tid_T, D_T\rangle$的形式记录事务及其依赖关系，其中 tid_T 是事务被预执行算法分配的序列号，D_T 是事务读集中版本号最大的数据版本的版本号。本文将这个三元组称为事务依赖三元组。与传统的调度图存储方法不同，事务依赖三元组只记录事务读集中最大的数据版本号，用 D_T 表示，以此来表示事务对序列号小于 D_T 的事务依赖关系。验证节点在验证执行阶段根据事务依赖三元组重构的调度图中，事务 T 对序列号小于 D_T 的所有事务均存在依赖边。

算法 3 预执行算法

输入：事务集 R

输出：事务依赖三元组集 U

1. 为 R 中的每一个事务 T 赋予递增的序列号 tid_T
2. 将 R 中事务按照序列号顺序加入任务队列
3. 读取某个工作线程执行事务 T 的反馈结果
4. 如果事务 T 因错误中止，则从区块中删除事务
5. 如果事务 T 第一次因冲突中止，则赋予事务 T 新的序列号并将其加入任务队列尾部
6. 如果事务 T 成功提交并返回最大依赖版本号 D_T，则将事务依赖三元组$\langle T, tid_T, D_T\rangle$加入 U 中
7. 如果任务队列被清空，则转到步骤 8，否则转到步骤 3
8. 为所有连续两次因冲突中止的事务分配新的序列号，并串行地执行这些事务

如果使用 4 字节有符号数表示序列号 tid_T 和最大读依赖 D_T，则事务依赖三元组为记录每个事务的依赖信息而引入的额外存储开销只有 8 字节。以太坊的智能合约事务平均大小

为 200 字节[17],以以太坊的智能合约事务为参照,事务依赖三元组只需引入约 4%的额外存储开销。

2)验证执行阶段。验证节点在验证执行阶段使用算法 4 执行区块中的事务,并验证事务依赖关系的正确性。验证阶段读取区块中的事务依赖三元组以获取事务并重构调度图。在验证执行阶段构建的调度图中,任何事务 T 对序列号小于其最大读依赖 D_T 的事务均存在一条依赖边,因此预执行阶段的调度图实际上是验证执行阶段所恢复的调度图的子图。算法 4 采用了拓扑排序的思想,但是没有显式地构建调度图。

算法 4　验证执行算法

输入:事务依赖三元组集 U

输出:验证结果

1. 将 U 中所有 D_T 为−1 的事务加入任务队列
2. 如果 U 中存在未被执行的事务,且所有工作线程均空闲,返回"Error"
3. 读取某工作线程执行事务 T 的反馈结果
4. 如果事务因冲突中止,返回"Error"
5. 如果反馈结果中,该事务的最大读依赖 D_T' 不等于 U 中该事务的最大读依赖 D_T,则返回"Error"
6. 如果 U 中,所有序列号小于 tid_T 的事务均提交,则将所有最大依赖 $D_{T'}=tid_T$ 的事务 T′加入任务队列
7. 如果 U 中所有事务已提交,则返回"Pass",否则转到步骤 2

为了防止拜占庭的领导者节点在区块中添加错误的调度图,验证节点如果验证调度算法返回值为"Pass",表明执行过程中没有出现错误;如果返回值为"Error",则说明领导者节点提供了错误的调度图,诚实的验证节点应当广播这个执行结果,以反对这个区块进入区块链。

事务依赖三元组虽然减少了调度图传递开销,但是这种粗粒度的调度图传输方法使验证节点在执行事务时损失了一定的并发度。然而,相对于存储完整调度图带来的通信开销的增长,这种并发度的损失是值得的。

4　实验分析

本章针对前文所提出的基于地址表的并行执行模型和基于多版本时间戳排序的并行执行模型在 Tendermint 上进行了实现,然后进行了事务执行阶段性能测试和多节点性能测试。

Tendermint 是一个区块链协议,采用 OEV 的执行架构。Tendermint 的共识算法是在 PBFT 算法基础上的改进,在仅共识大小为 250 字节的无操作事务时,吞吐量最高能达到 10000 tps 以上。此外,Tendermint 是一个具有可插拔性的共识协议,可以方便地移植到任何其他区块链系统中。

4.1　编码实现

本文的模型使用 Go1.19.2 在 Tendermint 上实现,数据库底层采用 LevelDB。

4.2　测试负载

本文使用一种基于账户的转账交易作为测试负载。每笔转账交易包含任意数量的转出账户和转入账户,以及转出账户对交易的数字签名。

实验设置了 10000 个账户,为每个账户分配一个私钥和一个公钥,使用账户的公钥作为账户的地址。为了模拟具有一定事务冲突的真实应用场景,实验采用具有 2 个转出账户与 2 个转入账户的转账交易,同时将 5%的账户设置为热门账户,每次数据读写都有 95%的概率访问这些账户。上述事务的平均大小约为 580 字节。

4.3　实验环境

实验采用的分布式环境由 4 台相同的服务器组成,服务器配置为:CPU Intel Xeon Silver 4110,核心数目 32 核心,主频 2.1 GHz;内存 256 GB,主频 2 666 MHz;操作系统 Ubuntu18.04。

4.4　实验结果与分析

为了简便起见,下文中将串行执行模型记为 SE,将基于地址表的并行执行模型记为 SACE,将基于多版本时间戳排序的并行执行模型记为 MVCE。

本文设置了两组实验:1)事务执行阶段性能测试,该实验在单节点中进行,用于比较 SE,SACE,MVCE 在线程数和事务集大小变化时,对同一事务集的处理速度的变化情况;2)多节点性能测试,该实验在 4 个服务器中进行,通过负载测试工具 tm-load-test 添加工作负载,分别测试了 4,8,12,16 个对等节点下的系统吞吐量、4 个对等节点下的确认延迟随负载大小的变化情况,以及 4 个对等节点下线程数对吞吐量的影响。

4.4.1　事务执行阶段性能测试

下面介绍事务执行阶段的性能测试结果。该组实验在单节点上进行,以事务集大小、工作线程数作为变量,测试 SE,SACE,MVCE 执行相同事务集的执行时间。每次实验随机生成一个事务集,分别提交至 3 个模型。在 SE 和 SACE 中,领导者节点无须预执行事务集,而是在广播区块后与验证者节点同时执行区块,因此执行阶段的时间开销就是一次事务集执行的用时。MVCE 中,领导者节点必须预执行事务集,并将依赖关系以事务依赖三元组的形式存储入区块,此后才可以将区块广播至各验证节点执行,因此 MVCE 的执行阶段时间开销是主节点预执行一次事务集的用时与验证节点验证执行一次事务集的时间之和。

1)线程数对事务处理速度的影响

实验 1 使用固定的大小为 10000 的事务集,探究了不同工作线程数对 SACE 和 MVCE 事务执行性能的影响,实验结果如图 4 所示。

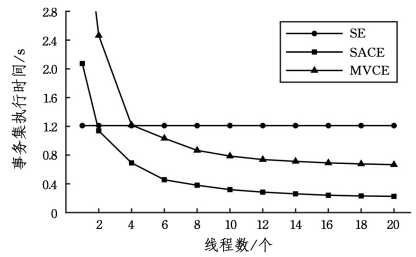

图 4　线程数对事务处理速度的影响(10000 事务)

Fig. 4　Impact of the number of threads on transaction processing speed(10000 transactions)

SE 采用单线程串行执行方式,执行速度不受线程数的影响。事务集大小为 10000 时,平均用时为 1.21 s。10 线程以下时,线程数的增加能显著提升 SACE 和 MVCE 的执行速度。线程数为 1 时,SACE 和 MVCE 的执行速度低于 SE。

线程数为 2 时，SACE 的执行速度与 SE 相当，并行执行带来的性能提升完全抵消了调度图的构建开销与调度开销；MVCE 在 4 线程时达到与 SE 相同的执行速度，由于 MVCE 中，领导者节点的预执行阶段与验证节点的验证执行阶段必须分别在不同时间阶段完成，因此每个节点使用 4 线程进行并发时，刚好抵消了冲突消解阶段、提交阶段的额外调度用时，以及验证节点等待预执行阶段执行结果的时间开销。线程数为 6 时，SACE 的执行阶段用时约为 0.46 s，MVCE 的执行阶段用时则为 1.03 s；线程数为 8 时，SACE 和 MVCE 执行相同事务集的时间分别缩短至 0.38 s 和0.865 s。线程数超过 10 时，SACE 与 MVCE 执行阶段用时的下降速度逐渐放缓。20 线程时，SACE 仅用 0.22 s 即可执行上述事务集，而 MVCE 的执行阶段也仅需 0.66 s。实验结果显示，增加线程数可以使 SACE 和 MVCE 的执行阶段的性能得到一定程度的提升。

实验显示，MVCE 的执行性能低于 SACE。一方面，MVCE 在排序阶段必须经历一次预执行，所有验证节点必须等待预执行的结果，造成了额外的时间开销；另一方面，事务依赖三元组虽然降低了调度图存储开销，但这种粗粒度的调度方法使验证节点损失了并发度。

2)事务集大小对事务处理速度的影响

实验 1 显示，工作线程数为 8 时，SACE 和 MVCE 的事务执行速度均能得到较大的提升，工作线程数超过 10 时，增加线程数带来的性能收益逐渐减小。因此，实验 2 为 SACE 和 MVCE 设置了 8 个工作线程，通过改变事务集的大小，探究了 SE，SACE 和 MVCE 处理不同规模的事务集的时间开销，实验结果如图 5 所示。

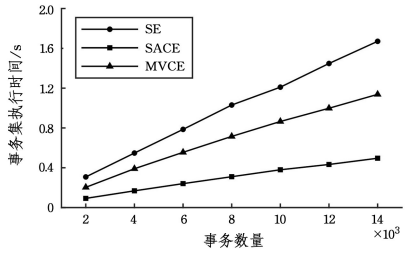

图 5 事务数量对事务处理速度的影响(8 线程)

Fig. 5 Impact of the number of transactions on transaction processing speed(8 threads)

SE，SACE 和 MVCE 执行事务集的时间开销均与事务数量呈正比。随着事务集规模的增大，SE 处理事务集所需时间的增长速度显著高于 SACE 和 MVCE。8 线程时，SACE 的执行速度能达到 SE 的 3 倍以上，而 MVCE 的执行速度仅为 SE 的约 1.5 倍。当批处理 10000 个事务时，SE 的执行时间为1.21 s，而 SACE 和 MVCE 的执行时间分别为 0.38 s 和 0.865 s，处理时间相比 SE 分别减少了 68.6%和 28.5%。

实验结果显示，事务集大小对 SACE 和 MVCE 的执行速度没有显著的影响，SACE 和 MVCE 能从多线程中获得比较稳定的加速比，因此其适用于高事务吞吐量的环境中。

3)调度图构建算法的时间开销

基于地址表的调度图构建算法是影响 SACE 执行阶段性能的重要因素。实验 3 测试了不同事务集大小下，使用 8 工作线程的 SACE 构建调度图的时间开销，实验结果如图 6 所示。基于地址表的调度图构建算法时间复杂度与事务集大小呈正比。事务集包含 2000 个事务时，SACE 使用 22.17 ms 进行读写分析并构建调度图，占事务执行时间的 24.33%。当事务集大小增加至 10000 时，SACE 使用 91.83 ms 进行读写分析并构建调度图。构建地址表与事务并行执行使用相同数量的工作线程，调度图构建与事务并行执行可以从多线程中获得相近的加速比。当事务集规模增大时，调度图构建算法的用时不超过 SACE 的事务执行阶段总用时的 25%。

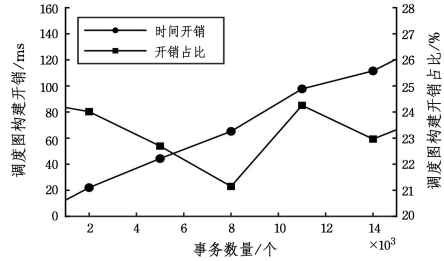

图 6 事务集大小对调度图构建算法的影响(8 线程)

Fig. 6 Impact of the number of threads on the time used to construct scheduling graph(8 threads)

实验结果表明，基于地址表的并行执行模型构建调度图时具有线性的时间复杂度，在区块中事务数量增加时仍能保持良好的执行性能，因此其适用于高事务吞吐量的环境中。

4)MVCE 的性能瓶颈

实验 1 显示，MVCE 的执行效率低于 SACE，当线程数大于 10 时，增加线程数为 MVCE 执行阶段带来的性能提升逐渐减少。实验 4 探究了 MVCE 的性能瓶颈。

实验 4 首先记录了 MVCE 在并行执行大小为 10000 的事务集时，预执行阶段事务冲突中止率随线程数的变化情况，实验结果如图 7 所示。6 线程时，冲突中止率仅为2.83%。当线程数增加时，并行执行的事务数量变多，发生数据冲突的概率也随之增大。20 线程时，冲突中止率达到11.87%。

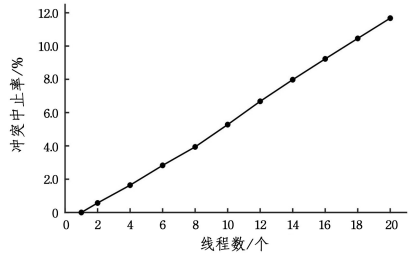

图 7 线程数对预执行阶段冲突中止率的影响

Fig. 7 Impact of the number of threads on the abort rate of pre-execution phase

此外，实验 4 记录了不同线程数下，MVCE 的预执行阶段与验证执行阶段执行 10000 个事务的时间开销，实验结果如图 8 所示。线程数小于 4 时，验证执行阶段与预执行阶段的执行时间基本相同。线程数大于 6 时，预执行阶段的执行时间逐渐少于验证执行阶段。线程数为 8 时，预执行阶段用时 0.39 s，而验证执行阶段用时则达到 0.47 s。线程数大于 10 时，验证执行阶段的执行效率几乎不再因为线程数的增加而

提升。上述实验结果表明,事务依赖三元组使验证节点重构的调度图损失了一定的并发性,当线程数超过一定数量时,验证执行阶段将无法充分利用线程资源,增加线程数很难继续带来性能提升。但是,事务依赖三元组大大减少了区块中调度图的存储开销,因此这种牺牲并发性的做法是值得的。

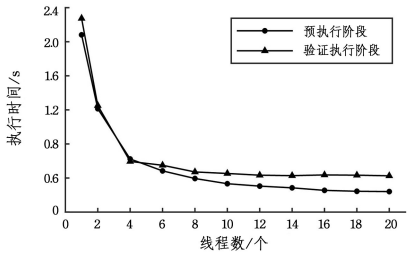

图 8　线程数对事务处理速度的影响(10000 事务)

Fig. 8　Impact of the number of threads on transaction processing speed(10000 transactions)

由图 7 可知,事务的冲突中止率随并行线程数量的增加呈线性增长趋势;图 8 显示,当线程数大于 15 时,预执行阶段的执行速度几乎不再有明显的提升,对冲突事务进行重排序以及重新执行的时间开销与线程数增加带来的性能提升相互抵消,并行执行对线程的有效利用率下降。

4.4.2　多节点性能测试

本节分别在 4 个服务器上部署 4,8,12,16 个节点,每个区块至多包含 10000 个事务,平均区块大小约为 5.5 MB。每次实验持续 60 s,取 5 次实验的平均值作为本文的实验结果。本节使用吞吐量(Throughput)和确认延迟(Latency)作为衡量系统性能的指标。

吞吐量指单位时间内提交的事务数量,单位为 tps,反映了区块链系统的负载能力。本文通过式(1)计算吞吐量。

$$throughput=\frac{blockSize}{commitTime-lastCommitTime} \tag{1}$$

其中,*blockSize* 代表区块的大小,*lastCommitSize* 是上一个区块的提交时间,*commitTime* 是本区块的提交时间。

确认延迟是从发出到被确认的时间间隔,单位为秒(Second,s),反映了数据处理的实时性。本文通过式(2)计算确认延迟。

$$latency=commitTime-broadcastTime \tag{2}$$

其中,*commitTime* 是事务所在区块的提交时间,而 *broadcastTime* 是客户端第一次将事务广播至领导者节点的时间。

1)多节点吞吐量测试

本实验将 SE,SACE,MVCE 分别部署于 4,8,12,16 个对等节点上,每个节点施加 30000 tps 的工作负载,以测试系统的极限吞吐量。实验中,SACE 和 MVCE 均部署 8 个工作线程进行事务的并行执行,实验结果如表 1 所列。

表 1　多节点吞吐量测试

Table 1　Testing on throughput of multiple nodes

methods	4	8	12	16
SE	3469	3356	3140	2819
SACE	4973	4735	4318	3731
MVCE	4147	3955	3618	3160

只有 4 个对等节点参与共识时,SE 的吞吐量仅为 456 tps。SACE 具有最高的并发性,领导者节点和验证节点能够同时执行区块,吞吐量最高,达到 4954 tps,相比 SE 提升了 43.4%。MVCE 中,领导者节点的预执行阶段和验证节点的验证执行阶段具有明确的先后顺序,因此整体并发性低于 SACE。在上述实验环境下,MVCE 的吞吐量为 4146 tps,相比 SE 提升了 19.5%。

4 节点时,共识算法平均用时为 1.67 s;当节点数增加至 16 时,共识算法的通信复杂度增加,平均用时达到 2.28 s,共识算法性能相比 4 节点时下降了 26.8%。采用 SE 进行 16 节点共识时,吞吐量相比 4 节点时仅下降了 17.8%,而 SACE 和 MVCE 的吞吐量则分别下降了 25.0% 和 22.6%。相比 SE,SACE 和 MVCE 显著地降低了事务执行阶段的时间开销。一轮共识中,共识算法的时间开销的占比更高,因此系统性能受共识算法性能的影响更大。故相比直接串行地执行事务,在使用 SACE 或 MVCE 的同时进一步优化共识算法,能取得更高的吞吐量提升。

仅共识 250 字节的无操作事务时,Tendermint 的吞吐量可以达到 10000 tps 以上。多节点吞吐量测试中,事务的执行需要进行 4 次读写以及 1 次数字签名验证,串行执行时,连续执行 10000 次上述事务用时为 1.21 s,事务的执行阶段成为影响吞吐量的重要因素之一。多节点吞吐量测试的实验结果表明,在执行阶段引入并行执行,能显著提升系统的吞吐量。

2)4 节点确认延迟测试

本实验将 SE,SACE,MVCE 分别部署于 4 个节点中,测试了不同负载下,系统对事务的确认延迟。实验中,SACE 和 MVCE 均使用 8 工作线程。实验结果如图 9 所示。

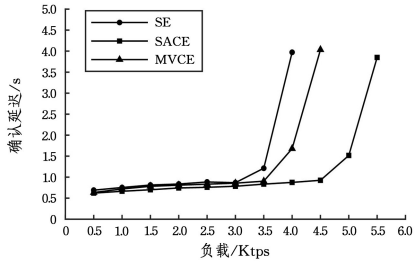

图 9　确认延迟测试(4 节点)

Fig. 9　Testing on latency(4 nodes)

负载为 500 tps 时,SE,SACE,MVCE 的确认延迟分别为 0.69 s,0.62 s,0.63 s。随着负载的增大,每个区块中的事务数量逐渐增加,共识算法和事务执行的时间开销也因此增大,出块速度的降低导致了延迟的增加。工作负载为 3000 tps 时,SE,SACE,MVCE 的延迟分别为 0.87 s,0.85 s,0.78 s。工作负载超过 3500 tps 时,SE 的吞吐量已经低于负载,一部分事务无法及时进入区块,从而造成确认延迟的迅速增加。MVCE 在负载超过 4000 tps 时开始出现上述现象,而 SE 在负载超过 5000 tps 时才会出现上述现象。实验结果显示,在高负载环境下,SACE,MVCE 比 SE 具有更高的实时性。

3)线程数对吞吐量的影响

本实验将 SE,SACE,MVCE 分别部署于 4 节点中,通过改变 SACE 和 MVCE 中的工作线程数来测试线程数的增加对并行执行模型的吞吐量的影响。实验结果如图 10 所示。

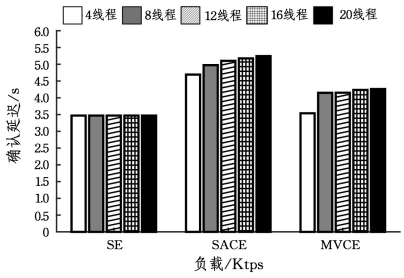

图 10 线程数对吞吐量的影响(4 节点)

Fig. 10 Impact of the number of threads on throughput(4 nodes)

SE 串行的事务执行方式不受线程数影响,吞吐量平均为 3 469 tps。线程数增加时,SACE 的吞吐量持续增加,4 线程时,SACE 的吞吐量达到 4 692 tps,当线程数增加至 20 时,吞吐量则能达到 5 243 tps。4 线程时,MVCE 的吞吐量仅为 3 538 tps,当线程数增加至 8 时,吞吐量增加至 4 147 tps。但是,线程数继续增加时,MVCE 的吞吐量增长非常缓慢。

实验显示,由于 SACE 具有高并发性,因此能更好地应用多核处理器的高性能优势。MVCE 的吞吐量虽然低于 SACE,但如果分配超过 8 个工作线程,使用 MVCE 的许可链相较于使用 SE 的许可链吞吐量也有一定的提升。因此,在执行一些具有复杂读写需求而不宜使用 SACE 的事务时,使用 MVCE 也可以在一定程度上提升系统的吞吐量。

结束语 在共识算法性能较高的许可链中,串行的事务执行方式可能成为性能瓶颈。为了降低事务执行阶段的时间开销,提升许可链吞吐量,本文针对 OEV 架构的许可链提出了基于地址表的并行执行模型与基于多版本时间戳排序的并行执行模型。基于地址表的并行执行模型使用静态分析的方法获取事务的读写集,地址表使调度图的构建开销降低为线性的时间复杂度。基于多版本时间戳排序的并行执行模型使用动态分析的方法获取事务的读写集,解决了静态分析方法无法获取具有复杂读写行为的事务读写集的问题。实验表明,上述事务并行执行算法能有效降低执行阶段的时间开销,单节点 8 线程时,基于地址表的并行执行模型和基于多版本时间戳排序的并行执行模型分别将 10000 个事务的执行时间减少了 68.6%和 28.5%;4 节点共识时,上述模型使用 8 线程分别将吞吐量提升了 43.4%和 19.5%。

参 考 文 献

[1] NAKAMOTO S. Bitcoin: A peer-to-peer electronic cash system [EB/OL]. (2018-10-31) https://bitcoin.org/bitcoin.pdf.

[2] WOOD D D. Ethereum: a secure decentralized generalized transaction ledger [J]. EthereumProject Yellow Paper, 2014, 151: 1-32.

[3] ANDROULAKI E, MANEIVCH Y, MURALIDHARAN S, et al. Hy-perledger fabric: a distributed operating system for permissioned blockchains [C] // The Thirteen Eurosys Conference. Porto, Portugal, 2018: 1-15.

[4] BUCHMAN E, KWON J, MILOSEVIC Z. The latest gossip on bft consensus [J]. arXiv: 1807.04938, 2019.

[5] CASTRO M, LISKOV B. Practical byzantine fault tolerance [C] // The Third Symposium on Operating Systems Design and Implementation. New York, USA, 1999: 173-186.

[6] BUCHMAN E. Tendermint: byzantine fault tolerance in the age of blockchains [D]. Guelph: University of Guelph, 2016.

[7] GUPTA H, JANAKIRAM D. CDAG: a serialized blockDAG for permissioned blockchain [J]. arXiv: 1910.08547, 2019.

[8] FU X, WANG H M, SHI P C, et al. Jointgraph: a DAG-based efficient consensus algorithm for consortium blockchains [J]. Software: Practice and Experience, 2021, 51(10): 1987-1999.

[9] ZHANG Z, LI Q, GAN J, et al. Design and implementation of a new blockchain model based on DAG [J]. Computer Applications and Software, 2021, 38(10): 114-124.

[10] YU W, LUO K, DING Y, et al. A parallel smart contract model [C] // 2018 International Conference on Machine Learning and Machine Intelligence. New York, USA, 2018: 72-77.

[11] BARTOLETTI M, GALLETTA L, MURGIA M. A true concurrent model of smart contracts executions [C] // 22nd IFIP WG 6.1 International Conference on Coordination Models and Languages. Valletta, Malta, 2020: 243-260.

[12] DICKERSON T, GAZZILLO P, HERLIHY M, et al. Adding concurrency to smart contracts [C] // The ACM Symposium on Principles of Distributed Computing. Washington DC, USA, 2017: 303-312.

[13] PANG S F, QI X D, ZHANG Z, et al. Concur-rency Protocol Aiming at High Performance of Execution and Replay for Smart Contracts [J]. arXiv: 1905.07169, 2019.

[14] ZHANG A. Towards concurrency control on smart contract in blockchain platforms [D]. Tianjin: Tianjin University, 2018.

[15] ANJANA P S, KUMARI S, PERI S, et al. OptSmart: A space efficient optimistic concurrent execution of smart contracts [J]. arXiv: 2102.04875, 2021.

[16] ANJANA P S, ATTIYA H, KUMARI S, et al. Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory [C] // Networked Systems: 8th International Conference. 2020: 77-93.

[17] XIAO J, ZHANG S, ZHANG Z, et al. Nezha: exploiting concurrency for transaction processing in dag-based blockchains [C] // 2022 IEEE 42nd International Conference on Distributed Computing Systems. 2022: 269-279.

[18] SHI J F, WU H, GAO H R, et al. Overview on parallel execution models of smart contract transactions in blockchains [J]. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 4084-4106.

[19] REED D P. Naming and synchronization in a decentralized computer system [R]. USA: Massachusetts Institute of Technology, 1978.

DONG Hao, born in 2000, postgraduate, is a member of CCF (No. P7867M). His main research interests include blockchain and so on.

ZHANG Aoqian, born in 1990, Ph.D, associate researcher, is a member of CCF (No. H7286M). His main research interests include database, data governance, blockchain and knowledge graph.

(责任编辑:杨雪敏)