计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330294
60(11): 2469−2488, 2023

面向分片许可链的无协调者跨片交易处理

阙琦峰[1,2] 陈之豪[1,2] 张 召[1,2] 杨艳琴[1,3] 周傲英[1,2]

[1](区块链数据管理教育部工程研究中心(华东师范大学) 上海 200062)

[2](华东师范大学数据科学与工程学院 上海 200062)

[3](华东师范大学软件工程学院 上海 200062)

(51205903035@stu.ecnu.edu.cn)

A Coordinator-Free Cross-Shard Transaction Execution for Sharded Permissioned Blockchains

Que Qifeng[1,2], Chen Zhihao[1,2], Zhang Zhao[1,2], Yang Yanqin[1,3], and Zhou Aoying[1,2]

[1](*Engineering Research Center of Blockchain Data Management* (*East China Normal University*), *Ministry of Education*, *Shanghai* 200062)

[2](*School of Data Science and Engineering*, *East China Normal University*, *Shanghai* 200062)

[3](*Software Engineering Institute*, *East China Normal University*, *Shanghai* 200062)

Abstract Recently, as blockchain technology continues to gain traction in various industries, there is an increasing need to improve the performance of permissioned blockchains in order to accommodate a wide range of applications. Sharding techniques have been proposed to optimize blockchain performance by dividing the network into committees, allowing for parallel transaction execution within each committee. However, the existence of expensive cross-shard transactions hinders the progress of sharded blockchain. Some work attempts to use the two-phase commit(2PC) protocol to process cross-shard transactions. However, these approaches suffer from substantial limitations in terms of performance and scalability, failing to meet the demands of modern industries for large-scale systems. Furthermore, these transactions demonstrate inadequate performance under high conflict scenarios, imposing additional constraints on the overall system performance. In this paper, we propose an approach for executing cross-shard transactions in sharded permissioned blockchains. The approach introduces determinism to the execution of cross-shard transactions, eliminating the need for additional coordination overhead while improving the efficiency of the system. To further improve system throughput, we utilize a transaction reordering mechanism to optimize the execution under conflicts. Experimental results show that our approach offers 1.6 times to 2.5 times higher throughput compared with the 2PC method, and 2.9 times to 25 times higher throughput compared with the non-optimized system in conflict scenarios.

Key words permissioned blockchains; sharding technology; cross-shard transaction; coordinator-free; conflict-resilient

摘 要 区块链作为一种防篡改、去中心化的分布式系统引起了学术界和工业界的极大关注.然而,传统区块链系统的吞吐量较低,且难以扩展到支持大规模系统,这使得其在商业中的应用范围受到限制.为了

收稿日期: 2023-04-06; **修回日期**: 2023-07-06

基金项目: 国家重点研发计划项目(2021YFB2700100); 国家自然科学基金项目(61972152); 上海市优秀学术/技术带头人计划资助(23XD1401100)

This work was supported by the National Key Research and Development Program of China (2021YFB2700100), the National Natural Science Foundation of China (61972152) and the Program of Shanghai Academic/Technology Research Leader(23XD1401100).

通信作者: 杨艳琴(yqyang@admin.ecnu.edu.cn)

解决这些问题,人们尝试利用分片技术把区块链网络分成多个可单独执行交易的子网,各个子网能够并行执行交易,其性能则可以随子网数成比例提升.然而,昂贵的跨片交易执行成本阻碍了分片区块链系统性能的进一步提升.传统的基于两阶段提交的跨片交易执行方法无论在性能上还是扩展性上都无法满足现代产业对大规模系统的需求;同时,这些方法在高冲突负载下的表现不佳,导致跨片交易的延迟急剧增加,严重影响到系统的正常运行.为了解决此问题,提出了一个针对分片许可链的跨片交易执行方法.该方法将确定性引入跨片交易执行,避免了额外的协调开销,同时提高了系统执行跨片交易的效率.此外,该执行方法也配备了抗冲突的交易重排序方法,除了提高跨片执行方法在高冲突的场景下交易处理性能之外,还能优化跨片交易执行中状态传输的效率.实验结果证明,该方法的吞吐量比基于两阶段提交协议的方法提高 1.6~2.5 倍;在冲突场景下,相较于优化前系统吞吐量则提高 2.9~25 倍.

关键词 许可链;分片技术;跨片交易;无协调者;抗冲突

中图法分类号 TP311.13

作为一个具有严格准入机制的去中心化的分布式账本,许可链经常被应用于企业间数据共享、管理、多方协作以及用户隐私保护等场景.然而,与分布式数据库相比,区块链系统的吞吐率较低,扩展性也有限,很难满足现代产业对高频交易的需求.为提高区块链系统的吞吐率和扩展性,一些将分片技术与区块链相结合的方案被提出.已有的分片区块链方案,如 Elastico[1] 和 OmniLedger[2],通常将整个网络划分为小的委员会(分片).在理想情况下,每个交易仅访问单个分片,分片可并行处理交易,以实现系统执行性能成倍提高.但由于跨片交易的存在,实现这种理想情况很困难.跨片交易需要操作多个分片上的数据,这给系统设计带来了挑战.一些工作,例如 RapidChain[3] 和 Monoxide[4],通过在 UTXO 模型下提出了一种最终原子性技术来解决这个问题.后续的工作,如 AHL[5],Chainspace[6] 和 Byshard[7],则通过账户模型的基于协调者的两阶段提交(two-phase commit, 2PC)协议来处理这些交易.

目前,基于 2PC 协议的跨片交易执行需要涉及所有相关分片之间的多轮信息交互,并以阻塞的方式处理跨片交易.特别是在实际基于分片的系统中,超过 96% 的交易是跨片交易,跨片交易的执行可能会因其性能不佳而导致区块链吞吐量下降.本文进行了实验,展示了基于协调者的跨片交易执行的效率,当跨片交易率为 30% 时吞吐量减少了近 67%,当跨片交易率为 90% 时吞吐量减少了 80%.此外,尽管区块链能够高效地处理低冲突负载,但在高冲突负载下性能会降低,这会进一步放大跨片交易的影响.因此,对于以性能为主要目标的许可链,优化跨片交易的执行是一个紧迫的需求,以支持广泛的应用.

基于以上分析,多轮交互、阻塞和冲突被认为是制约跨片交易执行性能的关键因素.在本文中,我们决定从 2 个方面对这一过程进行优化:首先,通过降低跨片交易执行期间的通信成本来提高效率;其次,通过容忍任意程度的冲突交易工作量来获得性能优势,从而为整个系统的吞吐量做出贡献.综上所述,本文的主要贡献总结为 3 个方面:

1) 针对分片许可链场景,提出了一种无协调者的跨分片交易执行方法,通过基于交易序列号的单向通信进行处理,来提高跨片交易的执行效率;设计低通信开销的状态传输策略,保证片间传输状态数据的正确性.

2) 针对高冲突负载场景,提出了一种抗冲突的跨片交易执行优化方案,结合交易重排序技术和跨片交易执行技术,以提高系统在高冲突负载下交易执行的效率,并优化跨片交易执行中的状态传输,以缩短跨片交易响应时间.

3) 实现了一个集成上述技术的原型系统,并在不同的工作负载下进行实验,实验结果表明该原型系统胜过其他协议的系统.

1 相关工作

1.1 确定性执行策略

传统的数据库领域中有许多关于确定性执行的研究. BOMH[8] 将并发控制与事务执行解耦,通过类似于 MVCC 方式对交易并发控制.在并发控制层维护一个多版本的链表,根据事务集的读写关系,直接确定事务执行时可见的快照版本.在执行层,事务寻找执行所依赖的快照版本,并将执行后的结果填入最新快照版本中.若事务执行时发现了所依赖的快照版本尚未生成,则系统尝试递归处理该事务,直到

发现所需的快照版本. 基于 BOMH, PWV[9] 对事务的读可见进行了进一步的优化. PWV 将一个事务分成多个子交易, 并寻找这些子交易的提交节点, 保证提交节点之前不会出现逻辑上的交易中止, 这使得某些事务的写集可以在其提交之前被其他事务看到, 而无需等待被提交.

Calvin[10] 则通过锁机制来并发处理交易, 主要分为排序层、调度层和存储层. 排序层主要负责给事务分配序列号; 调度层根据分配的序列号对事务进行冲突检测并上锁, 尽可能保证高效的事务并发效率; 最后将执行后的数据写入存储层. Calvin 在执行事务前需要检测其读写集, 这给事务处理带来了一些局限性. 而 Aria[11] 提出了一种不需要预处理阶段的确定性执行策略. 在执行阶段, Aria 基于相同快照并发执行一批交易, 并把写集保存在本地. 在提交阶段, 它根据交易序列号以确定的顺序提交交易. 此外, 该工作还提出了一个优化方案, 旨在降低交易的中止率, 获取更高的系统吞吐.

总之, 在确定性数据库中, 每个节点都会以确定的方式执行同一组有序的事务, 并将数据库从相同的初始状态转换为相同的最终状态. 每个事务在传递给节点之前都已被分配了一个序列值, 基于这个序列值, 不同的节点无需相互协调就能保持执行结果的一致性. 为了实现节点间的确定性的并发处理, 主流的并发控制协议主要采用了有序锁和交易依赖图这 2 种方法来保证确定性.

更重要的是, 区块链系统要求交易的执行是确定的, 因此数据库的确定性执行为区块链的执行优化提供了参考. Dickerson 等人[12] 将智能合约执行与并发控制相结合, 矿工节点通过软件事务内存(software transactional memory, STM)并行执行智能合约. 该方法与数据库常用两阶段锁原理类似, 智能合约访问一个状态数据时, 需要先获得该状态对应的一个抽象锁, 并填写好回退日志方便交易中止时回退, 并最终将执行顺序以 happen-before 图的方式广播给验证者以验证结果. 在验证阶段, 所有验证节点根据矿工传来的可串行化顺序和 happen-before 图来获得主节点中每一个交易获得锁的顺序, 以此作为验证交易的顺序. 最终, 验证节点可以确定地并发执行该任务, 并检测最终状态是否与矿工节点一致. 而在 Anjana[13] 的工作中, 矿工节点采用并发度更高的 OCC 协议来处理交易, 并根据执行结果生成一个交易拓扑图. 验证节点根据收到的交易和拓扑顺序来验证交易, 保证验证的结果和矿工执行结果相同.

MVTO[14] 方案借鉴了 BOMH 的思想, 将一个事务切分成多个子交易, 利用矿工节点生成的交易写集来构建记录各个版本数据信息的链式数据结构, 即版本链. 通过版本链, 系统能够并发地、确定性地重新验证交易, 从而提高了区块链系统的验证效率. PEEP[15] 提出了一个针对许可区块链系统的并行执行引擎, 该引擎允许交易在逻辑上并发执行, 并且在底层存储层面也能并发更新存储在状态树的叶子节点的状态数据块.

Jin 等人[16] 提出了一个高效的智能合约执行框架, 即 2PX. 针对主节点执行阶段, 2PX 提出了 Batch OCC 协议, 将数据库中常用的乐观并发控制协议应用于智能合约处理的主阶段中, 赋予了矿工节点并发执行交易的能力. 主节点会在交易的读阶段结束后生成该轮交易的冲突图. 待执行完区块内所有交易之后, 主节点会生成一个反映交易依赖关系的拓扑图, 然后将其与新区块一起广播给验证节点. 由于传输的交易依赖图包含主节点执行交易时读入或写入的具体数值, 主节点根据图切分算法将依赖图切分成多个子图以提高验证节点的验证效率. 同时, 该图切分策略将传输代价更大的边连接的交易节点放在同一个子图中, 极大程度地减少网络传输的开销, 进一步提高了系统性能.

1.2 区块链分片技术

区块链分片技术通过将节点划分为多个分片, 来提高区块链系统的吞吐量和处理速度. 最早的基于公有链设计的分片系统 Elastico[1] 将整个网络分为多个分片, 要求所有节点计算 PoW 的验证字段, 并且根据提交的结果决定该节点将被分配到的分片. 这些分片之间相互独立, 可以并行地进行交易处理, 有效地提高了整个系统的吞吐量. 由于每个分片的规模较小, 分片内部则运行经典的 BFT 共识协议. 因此, Elastico 系统的吞吐量可以随着节点的增加而提升, 近乎完成了线性扩展; 此外, 其灵活的分片划分规则避免了跨分片通信, 降低了通信复杂度. 但同时, 频繁的分片划分仍然会造成较大的系统开销, 尤其是在复杂且庞大的业务场景下, 其可能会成为主要的性能瓶颈. 分片区块链仍然需要更高效的方法来保证数据的一致性和处理跨片交易.

2018 年, OmniLedger[2] 提出了一种跨新型的跨片交易原子提交协议 Atomix, 保证每个交易被完全提交或最终取消. Atomix 授权客户端利用锁机制来协调分片之间处理跨片交易, 这使得分片之间无需直接通信, 保持了简单的分片工作逻辑. 以 UTXO 账户

系统为例,第一阶段,客户端首先会生成一笔跨片交易,再将该交易发送给与其输入相关的分片.每个相关分片检测相关输入是否合法,若检测通过,则记录输出状态,向客户端返回处理结果.当客户端收到了所有分片的消息,该协议进入第二阶段,即将提交跨片交易请求传送至相关分片.若任意一个相关分片取消了该笔交易,客户端也会向相关分片返回撤销交易的请求,以此来保证交易的一致性.但是,由于该协议由客户端驱动,在处理跨片交易的过程中,客户端被要求不能离线;同时,该协议的性能受限于客户端本身的性能,容易陷入单点瓶颈问题.

因此,秉承"轻客户端"的设计原则,AHL[5]则通过结合两阶段锁(two-phase locking, 2PL)和两阶段提交协议(two phase commit, 2PC)来处理跨片交易.与Elastico类似,AHL通过对比节点生成的随机数将整个网络划分为多个分片,分片内部采用将TEE与PBFT算法相结合的共识协议,所有协议也都由该协议得出.在处理跨片交易时,选择一个分片作为2PC协调者,称其为参考分片.第一阶段,参考分片向所有相关分片发送准备请求,并收集所有分片的返回结果.第二阶段,参考分片会根据上个阶段投票结果来决定交易的提交或者中止,最后同样地把处理结果返回给各个分片.

分片架构Chainspace[6]通过分片之间新的分布式原子提交协议S-BAC,来处理跨片交易.该方法根据交易模拟执行的结果标记与其他交易的冲突,将冲突信息发送到相关分片,然后每个分片通过冲突信息解决交易之间的冲突关系,最后将交易执行并提交至本地中.从协议实现的过程来看,S-BAC协议结合了BFT协议与2PC协议的特点,能够在拜占庭环境下保证片间交易的一致.但AHL与Chainspace都是针对单一系统设计的分片方法,它们无法灵活地适应复杂的业务场景与多变的需求.

于是,Byshard[7]提出了一种弹性的分片系统框架,以支持更加灵活的数据管理.该框架基于2PC和2PL技术,设计了3种跨片协调方法和4种执行方法,提供各种隔离级别的服务,能够适用于在拜占庭环境下的分布式系统.Byshard通过结合其所设计的跨片协调方法和执行方法,得到了18种实用的协议,以满足不同场景下对吞吐量、隔离级别、延迟和中止率等性能特征的需求.此外,该框架支持AHL和Chainspace等现有工作的实现.

SharPer[17]是一个用于许可区块链的分片架构,不同于上述基于单一协调者的工作,它通过片间共识支持在CTF和BTF情况下的跨片交易确定性并发执行.该工作解决了跨片交易可能出现的交易冲突、死锁、主节点故障的问题.主节点收到客户端传来的交易后,将主节点在该笔交易设置一个序列号,广播给所有的相关节点.分片从节点接收主节点传来的消息后,验证序号和交易内容,无需与该节点所在的分片共识,各个节点直接将决策结果返回主节点.当主节点收到并验证来自每个分片的$f+1$个"接受"消息后,向所有涉及交易的分片广播"提交"消息.最后,主节点向客户端返回结果,相关的分片收到消息后将本地交易提交.

跨分片共识协议被广泛应用来解决跨片交易执行的问题,但同时也引入了大量的片间交互,延长了跨片交易的提交时间.因此,一些研究人员开始探索更高效的分片技术.其中,基于UTXO账户模型的RapidChain[3]提出了一种新的跨片交易处理机制.该机制首先将一笔多输入、多输出的跨片交易分解成多笔单进、单出的子交易.由于这些子交易只有一个输入和输出,分片在处理相关子交易时只需验证本地输入是否有效,省去了分片之间互相验证的开销.该方法通过将交易重新划分,有效地减少网络开销,提高跨片交易的处理效率.

Monoxide[4]是一种针对公有链场景的分片技术,它将节点按照网络分布来划分多个子网络,并将每个用户钱包地址的前k位映射到相应的子网络.为了解决跨片交易问题,Monoxide将一笔跨片交易拆分为许多内部交易和中继交易,其中内部交易直接在片内执行,而中继交易以异步的方式在其他分片执行.具体地,当处理一笔跨片转账交易时,转账账户所在的子网络会在本分片共识并执行该交易,然后该子网会生成一笔中继交易并将其传送至收款账户所在子网共识并执行,这有效地保证了交易的最终原子性.这种方式避免了分片间互相验证的开销,使得跨片交易更加高效.

BrokerChain[18]引入了"市商账户"来协调处理跨片交易,以达到降低跨片交易延迟的目的."市商账户"会以相同的账户地址被划分至多个不同的分片.在处理跨片交易时,BrokerChain会将该交易划分为2个与市商账户相关的片内交易.由于一个市商账户在不同分片上都维护相同的地址,转账和收款账户仅需要与本分片内的相同市商账户交互即可完成跨片交易的执行.该方法能够有效地降低执行跨片交易的开销,也缓解了分片负载不均衡的问题.

但是方法Monoxide和BrokerChain需要执行来

自跨片交易划分的子交易，这会产生额外的执行成本. 后来，Pyramid[19] 设计了一个层级化的分片系统，该分片系统允许一些复合分片维护多个分片存储的数据. 如此，一些跨片交易可以在复合分片中转换为片内交易，并且可以在一轮交互中提交. 总的来说，Pyramid 通过牺牲一定的存储来极大程度地提高了系统性能.

2 无协调者的跨片交易执行

传统的分片区块链的跨片交易处理方法会引入多轮片间通信来保证交易的原子性和一致性. 以基于两阶段提交共识协议的分片区块链系统为例，首先，系统会选出一个分片担任共识协议中的协调者角色. 在协调者分片的帮助下，系统可以通过 2 轮通信来保证跨片交易在所有分片成功提交，但这会导致大量的网络开销和极高的交易延迟. 其次，系统不仅需要花费额外的资源去维护协调者分片，而且面临协调者带来的单点瓶颈问题. 因此，为了解决上述问题，本节基于以太坊执行范式，提出并详细介绍了一种无需协调者的高效跨片交易执行方法.

2.1 基本定义

本节介绍了本文执行方法依托的分片系统架构，并对跨片交易问题进行了形式化的定义与描述. 表 1 列出了本文出现的重要符号.

Table 1 Symbol Table

表 1 符号表

符号	符号描述	符号	符号描述
T_i	标号为 i 的交易	ES	系统分片
$R(T_i)$	交易 T_i 的读集	$W(T_i)$	交易 T_i 的写集
TS	传输分片	US	更新分片
N_t	TS 中节点集合	$Writer(T_i)$	T_i 的 US 集合
n_t	TS 中节点数量	N_u	US 中节点集合
f_t	TS 可容忍的拜占庭节点数量	n_u	US 中节点数量
		f_u	US 可容忍的拜占庭节点数量

2.1.1 系统架构

系统架构如图 1 所示，其由 2 类分片构成，即由多个共识节点组成的共识分片与多个执行节点组成的执行分片构成. 共识分片负责接收客户端发送的交易，共识打包并产生区块，对交易进行排序；当共识分片生成共识区块后，执行分片从共识分片同步区块，并按交易在区块中的顺序执行区块，触发本分片内部的状态改变. 同时，每个分片节点数量满足 $3f+1$ 的条件，其中 f 表示容忍的拜占庭节点数量.

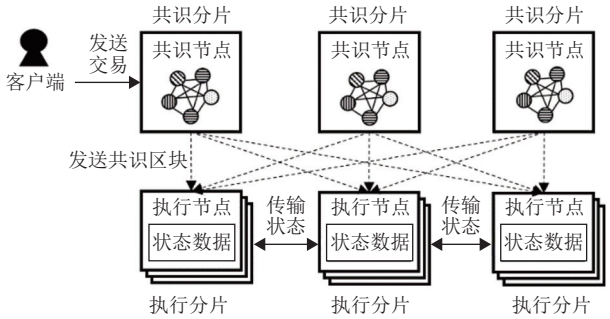


Fig. 1 System architecture

图 1 系统架构

当存在多个执行分片时，需要对所有的状态数据进行分片，以实现将负载划分到多个不同的执行分片中. 本系统需要将状态数据均匀地划分到不同的执行分片中，保证同一个状态的所有数据都被划分到相同的分片.

2.1.2 跨片交易执行

在分片区块链系统中，交易可以根据其特点分为 2 类：片内交易和跨片交易. 片内交易不需要与其他分片进行交互，仅涉及到本地数据的访问和修改. 而跨片交易需要修改或访问不同的多个数据，相比片内交易更加复杂，并且其原子性与一致性难以保证. 由于分片之间维护的状态数据不同，分片处理跨片交易的方式有明显差异；具体来说，一笔跨片交易 T_i 与某一分片 ES_j 的关系主要分为 3 类：

1) T_i 与 $W(T_i)$ 中的状态数据均不由分片 ES_j 维护；

2) $R(T_i)$ 中有状态数据由分片 ES_j 维护，$W(T_i)$ 中无状态数据由分片 ES_j 维护；

3) $W(T_i)$ 中有状态数据由分片 ES_j 维护，$R(T_i)$ 中是否有状态由分片 ES_j 维护均可；

对于 1)，分片 ES_j 在处理跨片交易 T_i 时，该笔交易与该分片所维护的状态数据无关，因此无需处理该笔跨片交易，直接将其忽略即可. 对于 2)，执行交易 T_i 需要读取分片 ES_j 中的状态数据，但不会修改该分片的状态数据；由于分片 ES_j 负责传输跨片交易读取的状态数据至其他分片，分片 ES_j 也被称为交易 T_i 的一个传输分片. 对于 3)，交易 T_i 不仅需要从分片 ES_j 读取状态数据，同时也要更新该分片的状态数据. 因此，分片 ES_j 是跨片交易 T_i 的一个更新分片.

由于数据分布在不同的分片，跨片交易 T_i 可能存在多个传送分片和多个更新分片；一个分片可以既是 T_i 的更新分片又是传输分片.

2.2 基于排序锁的跨片交易执行方法

本节描述了跨片交易执行方法处理跨片交易的细节，包括排序锁的工作原理、跨片交易执行流程以及拜占庭状态传输方案，从而使整个系统具有高性能、可扩展性和安全性.

2.2.1 排序锁的工作原理

本文的跨片执行方法是基于排序锁机制实现的，该机制通过利用基于锁定的并发控制来保证交易的原子性和可串行性，并通过以特定的交易锁定顺序来确保交易的隔离性. 在区块链系统中，排序锁机制可以根据交易在区块内的序列号顺序地锁定交易. 排序锁机制有 2 个重要的组件，分别是锁管理器和线程调度器. 前者负责按照有序的锁方案向交易授予锁；后者协调交易的执行，如工作线程分配和管理.

具体来说，锁管理器首先会根据先前预定的交易顺序依次给交易加锁，然后根据交易的读写集授予该交易当前可用的锁. 每个交易对锁的请求消息被保存在相应的锁的请求队列中. 如果一笔交易获得了其执行所需访问的所有数据的锁，它就能被送入线程调度器的缓存池中准备开始执行. 当一笔交易执行结束时，锁管理器会回收所有的锁，并将它们授权给下一笔交易. 线程调度器设置了多个工作线程并发地执行交易，调度器将把缓冲区中的每个交易分配给一个空闲的工作线程来执行. 在执行过程中，工作线程可能会因为某些原因中止一笔交易，但是这种行为被保证在所有节点中都是相同的. 具体实现如图 2 所示.

在本方法中，每个节点都会按照共识阶段确定的顺序给交易加锁，以保证每个节点执行的顺序是一致且唯一的. 即使拜占庭节点以不一致的顺序锁定交易，系统也会通过容错方法恢复正常，具体实现将在 2.3 节中阐述.

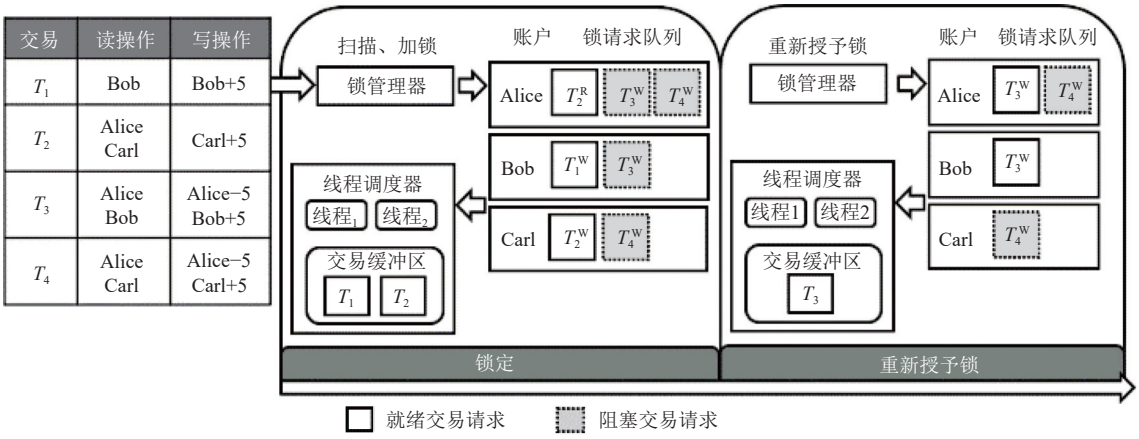


Fig. 2 An example of order lock mechanism

图 2 排序锁机制实例

2.2.2 跨片交易的执行流程

本文提出的跨片交易执行方法处理跨片交易 T 的流程有 4 个：

1)将每个跨片交易 T_i 的传输分片和更新分片读取 T_i 执行时读取的本地数据，发送至除自身以外的所有更新分片中；

2)若 T_i 没有读取到所有状态数据，则 T_i 会被更新分片挂起，并等待传输分片发送远端状态数据；

3)当更新分片收集 T_i 所需的所有远端状态数据之后，它会唤醒并执行 T_i，最后将执行结果写入本地数据中；

4)每个更新分片只写入本地维护的数据，忽略 T_i 对其他更新分片的更新操作.

举例来展示该跨片交易执行方法的实现细节. 如图 3 所示，分片 1 与 T_1、T_2 和 T_3 这三笔交易的关系可以分别对应本节中所描述的 3 种情况. 图 3 中分片 1 维护 Alice 的账户数据，分片 2 则维护 Bob 与 Carl 的账户数据. 首先，T_1 的执行逻辑是将 Bob 的账户余额增加 5，这笔交易只与维护 Bob 账户数据的分片 2 有关，所以分片 1 无需执行 T_1，自然地将该交易忽略，这符合 2.1.2 节中所述的第 1 类关系中描述的场景. 对于 T_2，它需要判断 Alice 账户是否有余额，以此来确定是否在 Carl 账户余额上增加 5，此时分片 1 就会成为 T_2 的一个传输分片，向 Carl 账户所在的分片传输 Alice 账户数据，这符合 2.1.2 节中所述的第 2 类关系中描述的场景. 最后在处理 T_3 时，Alice 需要向 Bob 转账 5. 由于该笔交易需要同时修改 Alice 和 Bob 的账户余额，这 2 个分片分别将 Alice 与 Bob 的

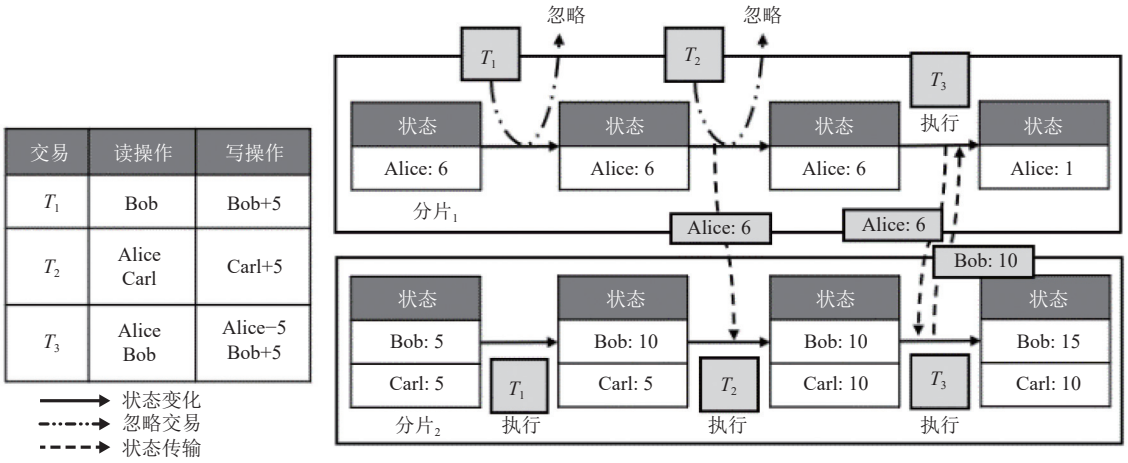


Fig. 3 The process of system transactions execution

图 3 系统处理交易的流程

账户余额发送给对方，然后双方都执行该笔交易，并根据执行的结果修改各自的账户余额，这符合 2.1.2 节中所述的第 3 类关系中描述的场景.

不难发现，该跨片执行方法要求传输分片状态数据传输至更新分片，这产生了一定的通信开销. 然而，对于执行复杂逻辑的智能合约，状态传输是不可避免的，即便在较为成熟的基于两阶段提交的跨片执行方法中，协调者也需要将执行需要的数据传输给参与者. 与其他只支持简单转账交易的分片系统不同，基于状态传输的执行方法可以适应更广泛的应用场景，支持更复杂的逻辑判断.

此外，本文采用的状态数据传输是通过异步实现的，不会阻塞其他交易的执行. 因此，即便是个别分片不能及时处理其对应的跨片交易，其他分片也会将该交易挂起并处理其他交易，不会严重影响系统处理交易的能力.

2.2.3 片间状态传输

在处理跨片交易的过程中，传输分片需要将本地的最新状态传输至更新分片上. 与传统分布式数据库的数据传输有所不同，区块链中的拜占庭节点不仅会发送错误的信息干扰系统正常运行，而且会合谋欺骗正常节点. 具体表现为，一方面，拜占庭节点传输错误的状态数据会影响其他节点的正常执行，浪费系统的计算资源；另一方面，由于传输分片和更新分片之间的恶意节点的数量是不对称的，规模更大的分片可以篡改或隐藏真实的数据，达到劫持其他小规模分片执行结果的效果. 综上所述，在区块链系统中，基于节点对节点的数据传输方式是不安全的.

此外，尽管使用拜占庭广播的方式传输状态数据可以有效地保证系统安全性，但在大规模网络中，数据广播将占用大量带宽资源和计算资源，导致网络拥塞和性能下降. 这与本研究设计的低通信开销的跨片执行方法的出发点相悖.

本研究基于双射传输[20]设计一种拜占庭数据传输方案，具体实现如图 4 所示. 该方案支持跨片状态传输，并能够以最小的消息传输代价完成传输，同时避免了广播所带来的巨大成本. 我们将对最小状态数据传输问题进行形式化定义.

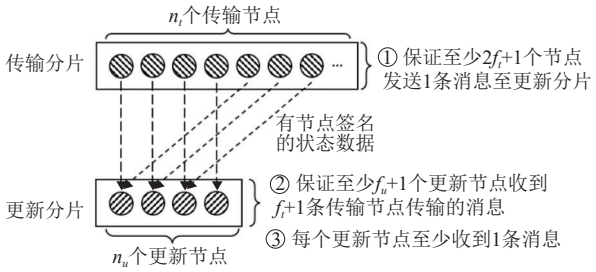


Fig. 4 Inter-shard state transfer

图 4 片间状态传输

定义 1. 给定一些传输分片中的节点 N_t 和一些更新分片中的节点 N_u，其中 N_t 中的节点将状态数据传输给 N_u 中的不同节点. 给定传输节点数量 $|N_t| = n_t$，N_t 能够容忍 f_t 个拜占庭节点；给定更新节点数量 $|N_u| = n_u$，N_u 能够容忍 f_u 个拜占庭节点. 找到传输分片需要发送的状态数据（简称消息）的最小数量 m.

根据传输节点数量 n_t 和更新节点数量 n_u 的大小关系不同，需要传输消息的最小数量 m 的决定性条件也不同. 当 $n_t \geq n_u$ 时，必须确保至少有 $2f_t+1$ 个不同传输节点传输消息；而当 $n_t < n_u$ 时，必须确保至少有 f_u+1 个节点接收到 f_t+1 个不同传输节点的消息. 综上所述，只有满足特定场景所需的条件，才能够保证传输分片发送的消息成功被更新分片所接收. 下面是

本文针对这 2 种情况提出的 2 个定理.

定理 1. 给定一些传输分片的节点 N_t 和更新分片的节点 N_u. 当 $n_t \geqslant n_u$ 时, 让 $q=\lfloor(2f_t+1)/(n_u-f_u)\rfloor$ 和 $r=(2f_t+1)\bmod(n_u-f_u)$. 消息传输的最小数量 m 定义为

$$m = qn_u + f_u \operatorname{sgn} r + r. \tag{1}$$

定理 2. 给定一些传输分片的节点 N_t 和更新分片的节点 N_u. 当 $n_t<n_u$ 时, 让 $q=\lfloor(f_u+1)/(n_t-2f_t)\rfloor$ 和 $r=(f_u+1)\bmod(n_t-2f_t)$. 消息传输的最小数量 m 定义为

$$m = qn_t + 2f_t \operatorname{sgn} r + r. \tag{2}$$

变量 q 是一个诚实的节点应该接收或发送的最小消息数量. 而变量 r 是需要接收或发送 $q+1$ 条消息的诚实节点数量. 以下是这 2 条定理的证明.

证明 1. 通过反证法证明 m 为最小消息传输数量. 假设仅需要传输 $m'=m-1$ 条消息, 就可以保证消息成功被更新分片接收. 当 $n_t \geqslant n_u$ 时, 将 N_u 中的节点分为 P_1 和 P_2 两部分, 将收到消息数量最多的前 f_u 个节点划入 P_1 中, 其余的 n_u-f_u 个节点划入 P_2 中. 其中, 需要证明 P_1 收到的消息数量满足 $m_{P_1} \geqslant qf_u+f_u \operatorname{sgn} r$, 于是利用反证法, 让 $m_{P_1}=qf_u+f_u \operatorname{sgn} r-v$, $v\geqslant 1$; 则剩余的节点群 P_2 只能收到 $m_{P_2}=m'-m_{P_1}=q(n_u-f_u)+r+v-1$ 条消息.

讨论 2 种情况: 当 $r=0$ 时, 可以得到 $m_{P_1}=qf_u-v<qf_u$ 和 $m_{P_2}=q(n_u-f_u)+v-1 \geqslant q(n_u-f_u)$. 这意味着在 P_1 至少存在一个最多收到 $q-1$ 条消息的节点, 在 P_2 中至少存在一个收到 q 条消息的节点. 当 $r>0$ 时, 有 $m_{P_1}=qf_u+f_u-v<(q+1)f_u$ 和 $m_{P_2}=q(n_u-f_u)+r+v-1>q(n_u-f_u)$, 这意味着在 P_1 中至少存在一个最多收到 q 条消息的节点, 在 P_2 中至少存在一个收到了 $q+1$ 条消息的节点. 由此可知, 在 P_2 中至少存在一个节点, 使其收到的消息数量比 P_1 中某个节点更多, 这与 P_1 划分规则相悖. 因此, 证明 $m_{P_1} \geqslant qf_u+f_u \operatorname{sgn} r$ 和 $m_{P_2} \leqslant q(n_u-f_u)+r-1$ 成立.

由于更新节点中诚实节点至少要收到 $2f_t+1$ 条消息才能保证更新分片不会被传输分片欺骗, 所以有 $q(n_u-f_u)+r=2f_t+1$ 成立. 最糟糕的情况是 P_1 的节点全为拜占庭节点, 在这种情况下只有 P_2 的消息被有效接收. 但是, 结合等式 $q(n_u-f_u)+r=2f_t+1$, P_2 所收到的消息为 $m_{P_2} \leqslant q(n_u-f_u)+r-1=2f_t$ 条, 无法收到来自诚实节点的 f_t+1 条消息. 因此, m'的假设是不成立的. 证毕.

证明 2. 同理, 当 $n_t<n_u$ 时, 将 N_t 中的节点分为 P_1 和 P_2 两部分, 将发送消息最多的前 $2f_t$ 个节点划入 P_1 中, 其余的 n_t-2f_t 个节点划入 P_2 中. P_1 的发送信息为 $m_{P_1}=2qf_t+2f_t \operatorname{sgn} r$, P_2 的发送信息为 $m_{P_2}=q(n_t-2f_t)$. 证明此过程与证明 1 相同, $m_{P_1} \geqslant 2qf_t+2f_t \operatorname{sgn} r$ 和 $m_{P_2} \leqslant q(n_t-2f_t)+r-1$ 成立.

只有保证在更新分片中至少有 f_u+1 个节点收到 f_t+1 个传输节点的信息, 更新分片才不会被恶意节点劫持, 所以有 $q(n_t-2f_t)+r=f_u+1$ 成立. 结合此等式知道, $m_{P_2} \leqslant f_u$ 成立. 这意味着, 当所有接收 P_2 传输的消息的更新节点都是拜占庭节点时, 在 P_1 中剩余的 f_t 个诚实节点传输的消息可能无法被更新分片有效地确认. 证毕.

由于传输节点不能同时向同一个更新节点多次传输消息, 在传输消息时, 传输节点需要在消息上签名, 注明消息的来源. 更新节点在接收消息时会对该消息进行验签, 以防止受到拜占庭环境中可能存在的消息重放攻击. 此外, 该方案仅确保整个分片收到 f_t+1 条消息, 这意味着单个节点仍有可能面临收到伪造消息的风险, 本文在 2.3 节中设计了一种区块重构方法, 被攻击的节点可以在区块重构时获得正确的状态数据, 因此不需要在状态传输的过程中采用即时验证策略来确认正确的消息.

2.2.4 跨片交易执行的实现

算法 1 展示了本文提出的跨片交易执行方法在处理交易时的细节. 代码第 1 行从交易队列中获取一笔交易来执行. 代码第 4 行通过判断交易的读集 $R(T)$ 和写集 $W(T)$ 是否与本分片有关, 从而决定是否忽略该笔交易. 其余代码详细描述了传输分片的工作细节, 首先, 扫描交易的读集和写集, 来判断需要传输至更新分片的本地状态数据; 然后, 再将这些状态数据发送到相应的更新分片. 此外, 当节点检测到该跨分片交易需要修改本地数据时, 节点不仅需要从本地数据 S 读取状态数据, 还需要从其他分片获取远端数据 RD(代码第 19 行). 如果一笔跨片交易成功读取所需的全部状态数据, 系统开始执行该笔交易, 并把它送入线程协调器的缓存中等待工作线程的处理(代码 21 行); 否则将该交易挂起并处理其他交易, 直到获取到相应的远端数据.

算法 1. 跨片交易执行算法 *TransactionExecution*.

输入: 交易执行队列 *Txs*, 本地维护的状态数据 *S*, 其他分片发送到本地的状态数据 *RD*.

① when *BeginExecutingTransaction*
② $T \leftarrow Dequeue(Txs)$; /*从交易队列中取出一笔交易执行*/
③ end when
④ if $R(T) \cap S=\varnothing$ & $W(T) \cap S=\varnothing$
⑤ $Discard(T)$;
⑥ end if

⑦ if $R(T) \cap S \neq \varnothing$
⑧ *Init*(*data_set*)；/*初始化需要传输至更新分片的数据集*/
⑨ for each $data \in R(T)$
⑩ if $record \in S$
⑪ $data\_set \leftarrow data\_set \cup record$;
⑫ end if
⑬ for each $ES \in Writer(T)$
⑭ $ES \leftarrow Send(data\_set)$；/*传输本地状态至更新分片*/
⑮ end for
⑯ end for
⑰ end if
⑱ if $W(T) \cap S \neq \varnothing$
⑲ $T \leftarrow Read\_data(S,RD)$；/*从本地存储和传输分片读取数据*/
⑳ if $R(T) \subseteq S \cup RD$
㉑ *Executing*(*T*)；
㉒ else
㉓ *Pending*(*T*)；/*将未完成的交易挂起*/
㉔ end if
㉕ end if

2.3 存储与数据恢复

在经过共识协议产生区块后，每个分片只需处理与其相关的交易，忽略其他分片中的交易. 然而，这种情况下传统的存储方法存在 3 个问题：1）共识协议生成的区块包含所有的交易信息，不能直接被分片用作最终的存储区块结构. 2）在每个节点独立执行交易的情况下，当处理跨片交易时，节点获取远端数据的正确性难以保证，可能影响系统数据的一致性. 3）分片之间也缺乏同步手段，难以确定系统处理交易的总进度. 综上所述，传统的区块存储方案无法满足分片区块链场景下数据管理的需求，因此，需要设计一种区块重构与数据恢复机制，以解决这 3 个问题.

2.3.1 区块重构

每个执行分片需要重新生成新的区块结构，特别是计算和存储包含交易信息与状态数据的 MPT 树. 每当分片中的节点执行一定数量的交易（例如，每执行 1 000 个交易），就将这些交易重新打包成一个新的区块，然后广播该区块的块号、状态树的根哈希 、交易树的根哈希以及从每个传输分片中得到的所有状态数据. 当每个节点收到新区块的 $2f$+1 条消息后，节点会验证消息的有效性并将它们传输的根哈希与本地计算出的根哈希进行对比.

通过比较 2 个状态树和交易树的根哈希，可以验证它们是否相同，从而验证不同节点之间区块链状态与块内交易信息的有效性和完整性是否一致.

当确定 f+1 条根哈希信息与本地生成的根哈希信息相同，节点确认该区块，并将区块存储在本地. 由于上述的重构区块的确认仅需要 1 轮网络通信，分片内不需要执行一个完整的 PBFT 共识来验证重构区块的正确性，大大减少了网络开销. 当共识协议的原始区块内的所有交易被分片获取后，即可删除该共识区块，节省系统存储开销. 该机制的工作流程如图 5 所示.

2.3.2 数据恢复

如果任何节点发现其生成的重构区块的根哈希与确认的区块不同，这意味着该节点很可能遭受到了拜占庭传输节点的恶意攻击. 举例来说，拜占庭节点可能会向更新节点发送错误的读集，导致不同节点在执行相同的跨片交易时视图不一致. 为了确保系统一致性，执行节点需要重新执行特定的交易集，以达到正确和一致的结果. 此外，重新执行的顺序应该遵循共识协议确认的顺序.

在区块重构阶段，需要在节点之间广播生成的重构区块以及一些相关信息，其中包括处理跨片交易时收到的状态数据集合. 遭受到攻击的节点能够在该阶段收到正确的状态数据集合，这些数据的正确性是被传输分片中 f_t+1 个节点签名保证的. 如果某个节点发现有节点故意提供错误的状态数据，它可以从其他节点发送来的区块确认信息中获取足够多的正确状态数据来进行正确的验证和执行.

尽管交易重复执行会给系统带来额外的计算开销，但本文认为该开销是可以接受的. 首先，在许可链中，由于节点是经过认证和授权的，节点之间的信任程度较高，产生拜占庭节点的概率较低. 其次，如果投票结果表明某个节点故意提供了错误的状态数据，那么该节点可能会面临一定的惩罚，比如将其标记为不诚实节点并拒绝它所传输的状态数据，进一步优化网络的安全性. 最后，在真实场景中，交易的依赖性通常不会太复杂，只有少部分交易会访问同一个状态数据，因此交易重做的代价较低. 综上所述，数据恢复的开销与风险是可以接受的.

2.4 安全性分析

根据区块链的特性，系统中各个分片通过共识协议达成一致，以确定需要处理的交易集. 然而，在交易执行过程中，可能会发生异常中止，这会破坏交

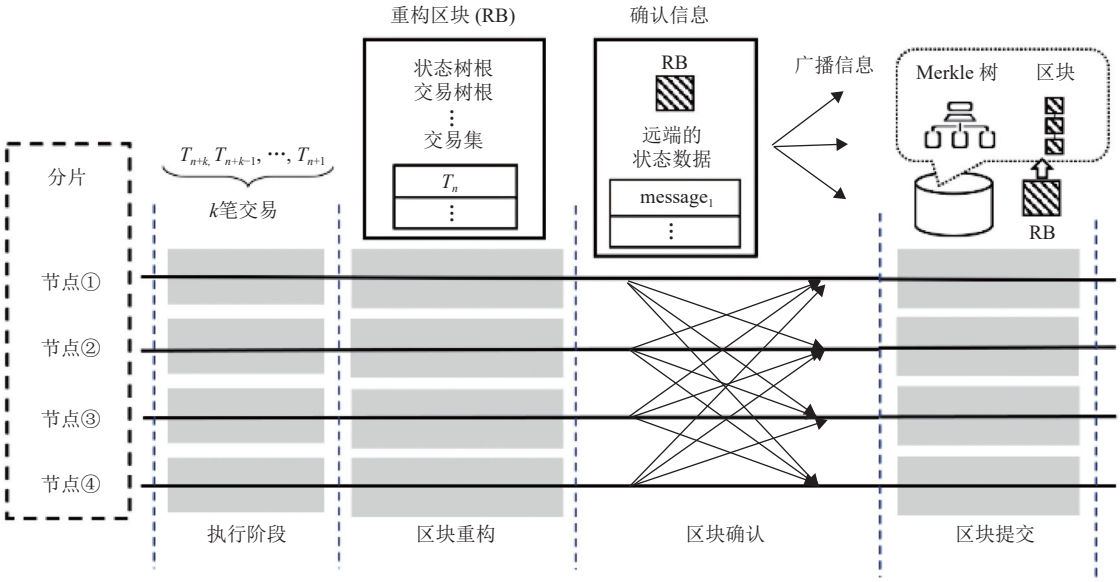


Fig. 5 Workflow of block storage and reconfiguration

图 5 区块存储与重构的工作流程

易的一致性. 除了拜占庭节点的恶意中止，本节将总结、分析和处理交易的其它异常中止.

交易异常中止可分为 2 种：1)由崩溃故障引起的中止；2)不符合执行条件引起的中止. 在情况 1)下，由于共识区块依旧保留着异常中止的交易信息与序列号，崩溃的节点只需要在恢复时根据交易序列号重新获取在共识区块内未提交的交易信息，并重新执行. 而在情况 2)中，如图 6 所示，共识区块内包含 2 笔跨片交易. 2 个分片不仅都需要更新状态，而且需要相互传输状态数据. 如果其中一个更新分片中的跨片交易(如 T_4)被中止，那么所有更新分片中的这个交易都将被中止. 这是因为 2 个分片看到的状态数据是相同的，操作逻辑也是相同的，因此最终执行结果必然相同.

此外，如果在处理跨分片交易时存在恶意节点故意保持沉默，这可能导致一些更新节点无法收到执行交易所需的状态数据. 为应对这种情况，更新节点可以主动向同一分片中的其他节点请求状态数据. 因此，本节提出的方法可以在所有情况下保证跨分片交易的一致性.

3 抗冲突的跨片交易执行优化方案

第 2 节提出的无协调者的跨片交易执行方法显著地提高了跨片交易处理的效率，令分片区块链系

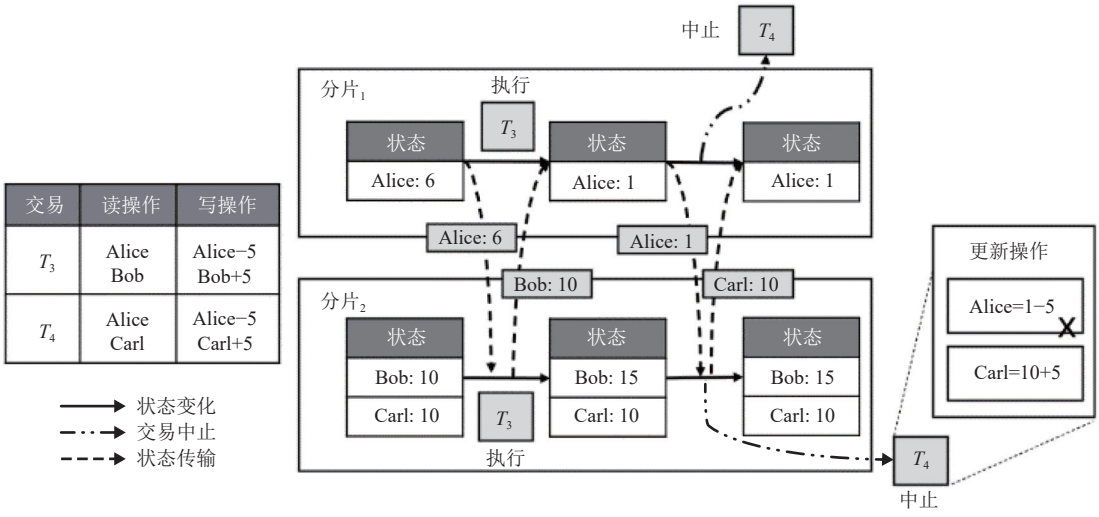


Fig. 6 The transactions abort

图 6 交易异常中止

统在面对大量跨片交易时依然保持良好的性能. 尽管区块链能够容忍低冲突的工作负载, 但在面对高冲突的场景时, 大量的冲突交易(包括跨片交易)会因网络拥堵或计算资源不足而被阻塞, 进而影响系统吞吐量, 延长跨片交易的响应时间. 此外, 如果跨片交易长时间未能被成功提交, 相关分片将中止该交易, 这需要额外引入多轮通信协议来保证交易一致性, 造成资源浪费和高昂的通信代价.

现今, 一些工作利用交易重排序技术来提高系统在高冲突负载下的性能. 然而, 这些方法仍然存在一些局限, 无法直接应用于排序锁机制. 首先, 这些重排序算法主要是针对 EOV 范式系统中高交易中止率的问题提出的, 目的是通过改变交易提交顺序, 找到最小中止交易集合, 从而提升系统的交易处理性能. 其次, 由于这类重排序操作发生在排序阶段之前, 所以它们无需考虑交易排序的限制, 这与需要全局序的排序锁机制相矛盾. 最后, 重排序算法是针对单机节点实现的优化算法, 尚未考虑跨片交易执行时状态传输对系统交易处理性能的影响, 以及跨片交易执行开销要远大于片内交易的情况.

因此, 本节对第 2 节提出的跨片交易执行方法进行优化, 通过将交易重排序策略与跨片执行方法相结合, 提高执行方法的抗冲突能力. 与传统重排序方案不同, 本节提出的策略主要通过提高执行并发度来提升系统性能, 它不仅可以通过重排序方案获得多个支持高并发执行的交易子集, 而且也针对跨片交易执行场景进一步优化, 提高了跨片交易传输的效率, 显著降低了跨片交易的延迟.

3.1 基于排序锁机制的交易重排序

本文提出的跨片执行方法是通过排序锁机制对块内交易进行并发控制. 排序锁机制虽然赋予了交易一定程度的并行处理能力, 但在高冲突负载下, 跨片交易的执行效率仍然会被严重影响. 因此, 本文研究对排序锁机制进行了优化, 设计了一个交易重排序方案, 以提高系统在处理高冲突负载时的表现. 同时, 优化后的并发处理方法具有通用性, 跨片交易和片内交易执行都可以从该方案中受益. 本文将结合一个具体的例子来介绍重排序方案的工作原理和执行细节, 交易用例以及交易之间的冲突关系参见表 2 和图 7.

本节提出的交易重排序策略主要是通过交易子集划分算法来解决低交易并发度和单一锁管理器串行加锁的问题. 交易子集划分算法基于读集和写集之间的冲突关系, 动态将需要执行的一批交易划分

Table 2 Transaction Table

表 2 交易集表

交易序号	交易读集	交易写集	交易类型
T_1	$R(a)$	$W(\ )$	片内交易
T_2	$R(a, c, d)$	$W(a, c)$	跨片交易
T_3	$R(b, d, f)$	$W(d, f)$	片内交易
T_4	$R(c, b)$	$W(b)$	片内交易
T_5	$R(b, e)$	$W(e)$	跨片交易
T_6	$R(f)$	$W(f)$	片内交易

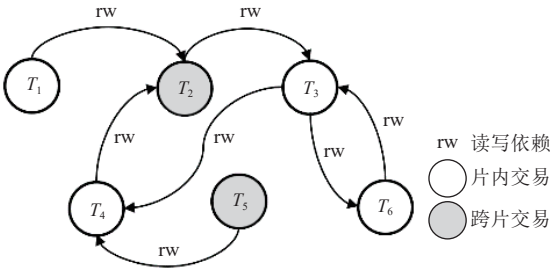


Fig. 7 Conflict graph

图 7 冲突图

为多个子集内部不存在冲突的交易子集. 由于任何一个交易子集中的交易互不冲突, 它们可以并行地执行, 从而最大限度地提高交易的并发执行性能. 此外, 该算法还实现了对交易子集内部所有交易的并行无冲突加锁, 消除了排序锁机制的单点性能瓶颈. 算法 2 详细描述了如何将一批交易通过交易重排序机制划分为多个交易子集.

算法 2. 交易子集划分算法 *TransactionsPartition*.

输入: 带有交易序号的交易队列 *Btxs*;

输出: 互不冲突的交易子集集合 *TSS*.

① $TSS \leftarrow \varnothing$; /*初始化交易子集集合*/

② when *Btxs* 不为空

③ $T \leftarrow Dequeue(BTxs)$;

④ $n \leftarrow |TSS|$;

⑤ for 下标从小到大遍历每一个 $S_i \in TSS$

⑥ if $HasConflict(T, S_i)$

⑦ continue;

⑧ else

⑨ $S_i \leftarrow S_i \cup T$;

⑩ end if

⑪ end for

⑫ if T 没有被任何一个 S_i 合并

⑬ 初始化一个新的交易子集 $S_{n+1} \leftarrow S_{n+1} \cup T$;

⑭ $TSS \leftarrow TSS \cup S_{n+1}$;

⑮ end if

⑯ end when

⑰ return *TSS*.

代码第 1 行初始化了输出的交易子集的集合. 代码 2~16 行将交易划分为内部互不冲突的交易子集, 其中子集标号较小的具有更高的执行优先级. 具体而言, 代码第 3 行从交易队列中取出一笔交易进行处理, 第 4 行确定交易子集的数量. 代码 5~11 行判断所处理的交易是否与当前交易子集内的交易存在冲突. 若不存在冲突, 则将该交易并入该子集中; 否则继续检索下一个子集. 如果遍历所有子集后发现该笔交易与所有子集都存在冲突情况, 则初始化一个只包含该笔交易的子集, 并将其标号赋值为 n+1.

图 8 展示了应用交易子集划分算法来划分交易的一个例子, T_1~T_6 是同一个区块内的 6 笔交易. 根据交易序号从小到大遍历所有的交易读写集, 并把它们分为多个集合内无冲突的交易子集. 对于第 1 笔交易 T_1, 根据此策略, 初始化过后的 *TSS* 没有一个交易子集, 于是将生成第 1 个交易子集 S_1 并将 T_1 并入 S_1; 然后, 对于第 2 笔交易 T_2, 它的读写集与 T_1 存在冲突, 因此无法将其放入子集 S_1, 而是将其放入新生成的第 2 个交易子集 S_2; 对于第 3 笔交易 T_3, 该策略根据子集下标从小到大寻找与 T_3 不冲突的集合, 可知 S_1 中的交易均和 T_3 无冲突, 于是将 T_3 放入交易子集 S_1 中. 同理, T_4 将被放入新生成的交易子集 S_3 中, T_5 并入 S_1 中, T_6 并入 S_2 中. 最终, 本算法可以获得 S_1={T_1, T_3, T_5}, S_2={T_2, T_6}和 S_3={T_4}这 3 个交易子集.

尽管同一个交易子集中的交易可以并行加锁, 但不同子集之间的加锁操作必须按照一定的顺序进行. 具体而言, 标号较小的交易子集需要优先于标号较大的交易子集进行加锁操作. 结合图 8 用例, 对于 S_1 中的 3 笔交易, 多个线程可以并行地对它们进行加锁操作, 因为它们之间不存在任何依赖关系; 而对于 S_2 中的交易, 由于它们依赖于 S_1 中的交易, 必须等待 S_1 中的所有交易均被锁定后, 才能进行加锁操作. 最后, 这批交易的执行结果等价于串行执行结果 T_1→T_3→T_5→T_2→T_6→T_4.

算法 3. 冲突检测函数 *HasConflict* 算法.

输入: 一笔交易 T, 一个交易子集 S;

输出: 一个布尔值 *Bool*.

① for each T 的写集 $W \in W(T)$

② if 满足 $W \in W(S)$ 或 $W \in R(S)$

③ return true; /*$R(S)$和 $W(S)$为集合 S 中交易读集与写集总和*/

④ end if

⑤ end for

⑥ for each T 的读集 $R \in R(T)$

⑦ if 满足 $R \in W(S)$

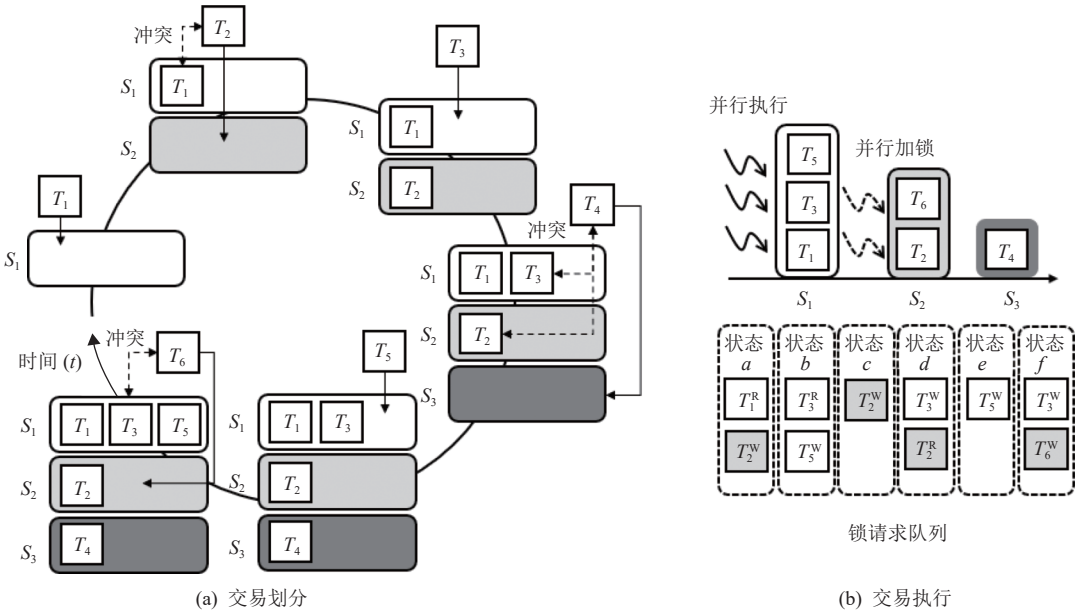
⑧ return true;

⑨ end if

⑩ end for

⑪ return false.

函数 *HasConflict* 用于检测一笔交易是否与一个交易子集存在冲突, 输入包括一笔交易 T 和一个交易子集 S, 输出为代表两者冲突结果的布尔值 *Bool*. 该函数首先将交易 T 写集 $W(T)$ 与交易子集 S 的 $R(S)$



(a) 交易划分 (b) 交易执行

Fig. 8 Transaction partition with reorder scheme and transactions execution

图 8 重排序方案的交易划分与交易执行

和 $W(S)$进行比较，判断该笔交易是否与集合 S 中的任何一个交易存在“读写”冲突或“写写”冲突. 再者，由于“读读”操作之间不存在冲突，在处理该交易的读集时，该函数仅会判断交易 T 的读集 $R(T)$与交易子集 S 的写集 $W(S)$的冲突情况，即判断该交易是否可能读到集合 S 中交易所修改的数据. 根据这 2 轮检测的结果，该算法会决定是否将该交易并入交易子集 S 中. 具体来说，接图 8 所示的例子，当 T_4 尝试加入交易子集 $S_1=\{T_1,T_3\}$时，T_4 的写集中的 b 与 S_1 中 T_3 的读集产生冲突，因而无法并入 S_1.

3.2 针对跨片交易的重排序优化策略

3.1 节提出的交易重排序策略有效地提升了交易的执行并发度，从而增强了跨片交易执行方法的抗冲突能力. 但是，这种策略忽视了交易重排序对跨片交易执行的影响. 在执行跨片交易时，每个更新分片需要等待传输分片传输的状态数据，而影响状态数据传输效率的原因有 2 个方面：一方面，由于每个分片执行的交易集不同，处理同一笔跨片交易的时间点也会有差异. 如果贸然地通过重排序策略将某笔跨片交易的执行顺序排在与之冲突的片内交易之后，那么与该笔跨片交易相关的状态数据必须等到与之冲突的片内交易全部执行完毕后才能传输，这严重影响了其他更新分片的交易处理性能. 甚至，跨片交易可能因为长期得不到状态数据而需要重新执行，这无疑会极大地浪费计算资源.另一方面，为了保证数据的一致性，传统的排序锁机制仅在跨片交易执行阶段以交易为单位进行状态数据交换，以确保一笔跨片交易在相同的视图下执行. 如此细粒度的状态传输会导致分片之间频繁地进行信息交互，从而造成繁重的网络开销. 同时，该机制也无法有效地利用带宽优势来处理跨片交易. 正如图 8 中的例子所示，交易子集 S_2 中的跨片交易 T_2 所需的状态数据 a 需要等待 S_1 中的片内交易 T_1 执行完毕. 同时，执行 T_2 与 T_5 这 2 笔跨片交易也伴随着等量次数的片间数据交换.

针对这 2 个方面的挑战，本节将进一步优化交易重排序策略，消除跨片交易对片内交易的依赖，提高状态数据的传输效率. 首先，在锁管理器遍历交易时，该优化方法会对跨片交易和片内交易区别处理. 对于跨片交易，只需要确保它不与交易子集中的其它交易发生冲突即可将其并入该子集，处理方法与算法 2 如出一辙. 而对于片内交易，该算法不仅需要检测与当前遍历的交易子集的冲突，还需要检测比当前交易子集优先级更高的子集中的跨片交易的冲突，以消除与跨片交易的“读写”和“写写”依赖. 最后，在交易子集划分完成之后，本策略将部分跨片交易所需的且不会被片内交易修改的状态数据批量传输至更新节点. 通过这种方式，传输分片可以更高效地将状态数据传输到每个更新分片，降低分片之间状态传输的频率，大幅度缩短跨片交易的响应时间.

算法 4 详细描述了如何将一批片内交易放入已存在的无冲突交易子集中，同时保证每一笔跨片交易都不会产生对片内交易的依赖. 然后，输出一个包含片内交易与跨片交易的交易子集集合. 算法 4 与算法 2 主要有 2 处不同：1)片内交易 T 先从下标更大的交易子集开始遍历. 如果算法 4 从下标较小的交易子集开始遍历，检测到交易 T 与某个交易子集 S_j 没有发生冲突，算法 4 无法确定 T 与 $S_k(j<k)$中的跨片交易是否存在冲突，这也就意味着无法保证跨片交易不依赖于片内交易的执行结果. 而从下标较大的子集开始遍历，当 T 第一次与某个交易子集 S_i 产生冲突，表明 S_i 是与 T 有冲突的子集中下标最大的子集，所以算法 4 可以将该交易 T 并入最靠近该子集的不冲突交易子集 $S_n(n>i)$中. 2)算法 4 在判断交易 T 与交易子集是否冲突时，需要检测子集中与之产生冲突的交易类型. 如果 T 与跨片交易产生冲突，则算法 4 的处理方式与算法 2 相同，直接中止遍历并将交易 T 并入对应的交易子集中. 当 T 与片内交易产生冲突时，算法 4 会继续向前遍历，直到遍历完所有子集或遇到与其冲突的交易才会停止，这保证了算法 4 产生的交易子集数量尽可能少，最大程度提升执行的并发度.

算法 4. 优化交易子集划分算法 *TransactionsPatitionByType*.

输入：交易子集集合 *TSS*，带有交易序号的交易队列 *Itxs*；

输出：插入片内交易的交易子集集合 *TSS*.

① $S_{Ctxs}\leftarrow\bigcup_{S\in TSS}S$; /*只包含跨片交易的交易集合*/
② when *Itxs* 不为空
③ $T\leftarrow Dequeue(ITxs)$;
④ $n\leftarrow|TSS|+1$;
⑤ for each 下标从大到小遍历每一个 $S_i\in TSS$
⑥ if $HasConflict(T,S_i)$
⑦ if $HasConflict(T,S_i-S_{Ctxs})$
⑧ continue; /*T 只与片内交易产生冲突，继续向前遍历*/
⑨ else
⑩ break; /*T 与跨片交易产生冲突，中止遍历*/

```
⑪          end if
⑫       end if
⑬       else
⑭          n←i;
⑮    end for
⑯    if S_n 不存在
⑰       S_n←S_n∪T; /*初始化一个新交易子集 S_n*/
⑱       TSS←TSS∪S_n;
⑲    else
⑳       S_n←S_n∪T; /*将 T 并入最后一个不冲突
          的交易子集*/
㉑    end if
㉒ end when
㉓ return TSS.
```

代码 5~10 行展示了片内交易寻找适合并入的交易子集的过程. 每次检测到该子集与交易 T 无冲突时，算法 4 都会更新并入的子集下标 n. 由于采用自后向前遍历方法，所选择的并入子集的优先级随着遍历的进行而逐渐提高. 当一轮交易子集的遍历完成后，代码 16~20 行将交易 T 并入先前选出的最适合的子集 S_n 中. 如果下标 n 代表的交易子集不存在，则会新生成一个包含交易 T 的交易子集.

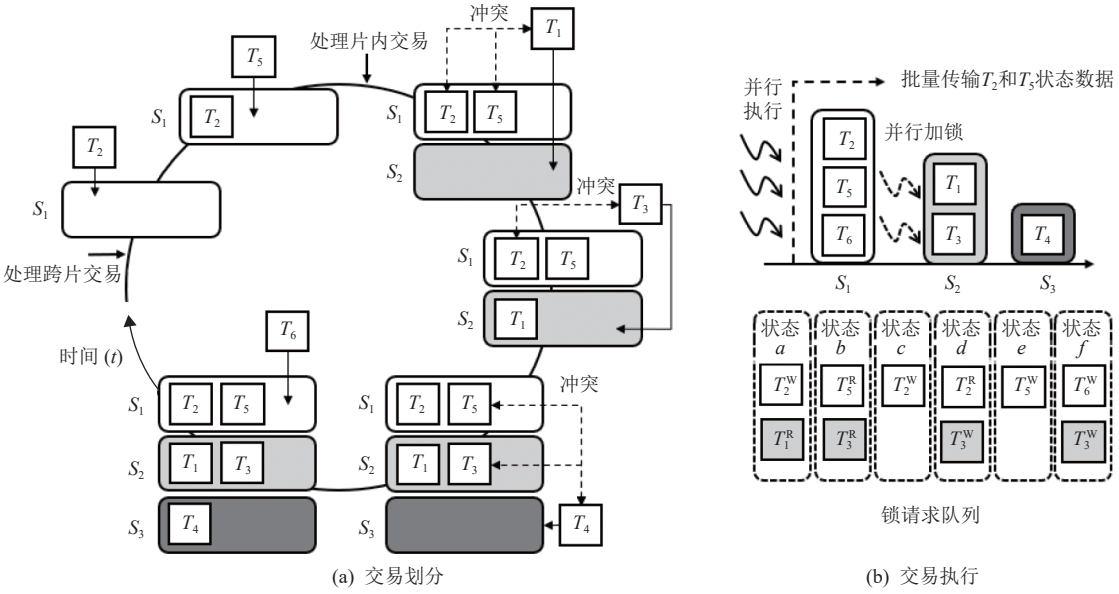
最后，算法 4 获得了一个包含片内交易与跨片交易的交易子集集合 TSS. 由于不存在跨片交易对片内交易的数据依赖，在执行任意一个交易子集前，传输分片可以将子集中跨片交易所需的状态数据打包发送至对应的更新分片，不会发生由于更新分片读到旧数据产生的执行结果不一致的情况.

图 9 展示了应用算法 4 优化后的交易子集算法来划分交易的一个例子. 首先，使用算法 2 处理跨片交易，并将它们划分到交易子集 S_1 中，因为 T_2 和 T_5 之间不存在冲突. 接下来，使用算法 4 处理片内交易. 交易 T_1 与跨片交易 T_2 存在冲突，因此被合并到新的交易子集 S_2 中. 类似地，交易 T_3 与 T_2 存在冲突，但与 T_1 无冲突，因此也被合并到 S_2 中. 对于交易 T_4，根据算法 4，当遍历到 S_2 时，它与片内交易 T_3 冲突. 继续遍历，直到检测到与跨片交易 T_2 冲突，记录其并入的子集下标为 3，然后将其合并到新的交易子集 S_3 中. 交易 T_6 与 S_2 中的片内交易 T_3 存在冲突，但与任何跨片交易都没有冲突，因此被合并到 S_1 中. 综上所述，算法 4 可以生成 3 个交易子集 S_1={T_2,T_5,T_6}，S_2={T_1,T_3}和 S_3={T_4}. 简而言之，通过优化的重排方案，不仅跨片交易以并发执行，而且整个分片区块链系统在高传输效率运行.

算法 5. 引入交易重排序策略后的交易执行过程.

输入：一批交易 B.

```
① when BeginTransactions
②    ExecutionQueue← ∅;
③    RD← ∅;
④    初始化本地数据 S;
⑤ end when
⑥ when ReceiveRemoteData
⑦    RD←RD∪Remote_Data;
⑧ end when
```



(a) 交易划分 (b) 交易执行

Fig. 9 Optimized transactions partition and transactions execution

图 9 优化后的交易划分与交易执行

⑨ when *LockPhase*

⑩ *Itxs,Ctxs*←*B*; /*片内交易和跨片交易划分至2个有序队列*/

⑪ *TSS*←*TransactionsPartition*(*Ctxs*);

⑫ *TSS*←*TransactionsPartitionByType*(*Itxs*, *TSS*);

⑬ for each 每个交易子集 $S_i \in TSS$

⑭ 并行加锁*ExecutionQueue*←*LockParallel*(S_i);

⑮ end for

⑯ end when

⑰ when *ExecutionPhase*

⑱ ES_j←*SendStateData*(S_i); /*将状态数据集打包传输*/

⑲ *TransactionExecution*(*ExecutionQueue,S,RD*); /*并行执行交易*/

⑳ end when

算法5描述了引入交易重排序策略后交易的执行过程. 交易开始执行时, 会初始化一个执行队列*ExecutionQueue*和远程状态数据集*RD*. 当有远程数据传输至本地时, 会将其暂存至*RD*中. 在锁定阶段, 算法5首先将一批交易*B*按交易类型划分至不同的交易队列中, 队列中的交易顺序严格按照交易序列号从低到高排列. 然后, 通过算法2与算法4将多个交易子集进行划分. 最后, 利用多个锁线程并行地给一个交易子集中的所有交易加锁. 顺利完成加锁操作的交易会进入执行阶段, 算法并行地从*ExecutionQueue*中取出交易并执行. 由于跨片交易在同一个交易子集中不存在冲突, 传输分片可以一次性将多个状态数据提前传输至更新分片, 无需等待交易执行过程中传输.

4 实验与结果

基于提出的无协调者跨片交易执行方法, 本节实现了原型系统CoCoS, 并对其性能进行测试. 主要测试了该方法在不同跨片交易占比下的系统执行交易的吞吐量、交易延迟等方面的性能, 并与基于2PC协议的分片系统(如AHL)进行对比.

4.1 实验环境

1) 硬件环境. 所有的实验都在一个集群上进行, 该集群有20台机器, 配备了英特尔至强2.5 GHz CPU, 每个CPU包含了32个核心, 还配备了32 GB内存和320 GB磁盘空间以及100 MB网络带宽. 本实验使用Ubuntu 16系统, 并运行g++的7.3版本. 本实验使用go-Ethereum作为库, 并将一个开源的PBFT实现集成到本文的系统中. 每个分片被分配多个工作线程来处理交易. 实验的结果取10次测试结果的平均值, 以提高实验的准确性.

2) 软件环境. 本文采用具有C++版本的Ethereum作为系统库来执行智能合约. 为了实现交易的并发执行, 本实验创建了一个虚拟机实例池, 每个实例池管理1个以太坊虚拟机(Ethereum virtual machine, EVM). 实验设置了1个共识分片和4个执行分片. 每一个分片都由4个节点组成, 并且每个节点都被分配在不同的机器上. 共识分片负责向每个执行分片传输共识协议产生的共识区块.

3) 智能合约. 本文通过SmallBank智能合约来测试跨片交易执行方法的性能. SmallBank智能合约经常被用作评估OLTP数据库系统的性能基准, 也被广泛用于区块链系统的性能测试. 本文所有实验使用的数据集包含1 000万客户和相应的1 000万账户. 此外, 本文为SmallBank设计了一个额外的*SendPayment*交易来实现转账功能.

4.2 测试负载

在介绍实验中使用的工作负载之前, 需要先定义跨片率(*cross-shard rate*)与冲突率(*conflict rate*)的概念. 跨片率主要指的是跨片交易数量与总交易数量之比. 本实验实现的负载生成器考虑了交易数、账户数和跨片率, 可以为每组实验生成相应的负载. 实验测试了在4个不同跨片率下系统执行跨片交易的性能, 这4个跨片率分别为0, 0.3, 0.6和0.9. 跨片率也反映了分片之间的通信频率. 所有跨片和分片内的交易被均匀地分布在不同的分片上, 以达到分片的负载均衡. 而冲突率表示冲突交易的数量与总交易的数量之比. 实验使用冲突率来描述交易集的负载情况, 并选择0.3, 0.6和0.9的冲突率代表3种不同程度的冲突. 此外, 这些冲突交易被均匀地分布在每个分片中, 以确保每个分片的性能保持相似. 另外, 需要注意的是, 一个区块最多只能包含1 000笔交易.

4.3 系统跨片处理性能

本节介绍跨片交易执行方法的实验结果, 包括吞吐量、延迟和扩展性. 我们从单节点吞吐量和每个块的延迟评估方法的总体性能, 无需共识. 本文部署了4个执行分片. 实验细节如下.

4.3.1 系统吞吐量

图10展示了每个节点的跨片率与吞吐量之间的关系. 本文方法和基于2PC协议的跨片执行方法进行了比较. 由于2PC协议涉及分片之间多轮的通信与交易阻塞, 它在处理跨片交易时表现不佳. 基于

2PC 协议的跨片执行方法在高跨片率下最高只能达到 6 000 TPS, 而在高跨片率情况下仅能达到 1 200 TPS. 其性能低下的主要原因在于处理跨片交易时需要进行多轮片间交互, 导致跨片交易确认时间延长, 并严重阻塞其他交易的执行. 同时, 随着负载跨片率的升高, 庞大的通信开销使得系统处理性能急剧恶化, 该现象与本文的预期结果一致.

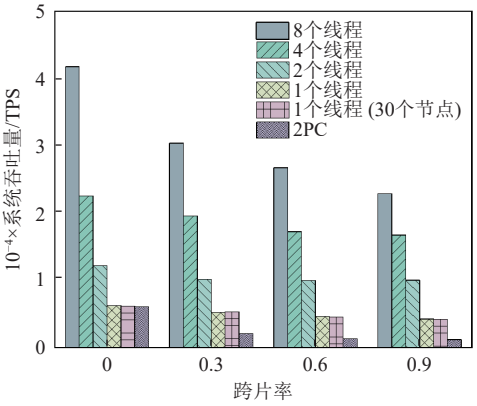


Fig. 10 System throughput

图 10 系统吞吐量

本节还测试了不同数量的工作线程下无协调者的跨片交易执行方法的表现. 由图 10 可知, 该方法可以利用多线程的优势来处理交易. 当利用 4 个工作线程处理无跨片交易的交易集时, 每个节点的吞吐量可以达到 22 000 TPS; 相较于基于 2PC 协议的方法, 在高跨片率下系统的性能仅下降了约 30%. 并且, 随着工作线程数量的增加, 系统的吞吐量也会明显地提升. 然而, 由于状态数据传输所需的时间以及单一锁管理器可能存在的瓶颈, 特别是在高跨片率负载时, 增加线程数并不能显著提高系统的执行性能. 这是因为系统性能的瓶颈从线程数量转移到了状态传输和锁管理器上. 此外, 随着节点数量增加至 30 个, 系统吞吐量并无明显地降低.

4.3.2 交易延迟

图 11 展示了每个节点的跨片率与交易延迟之间的关系. 在区块链中, 交易以块为单位进行提交, 并且只有区块内所有交易都执行完毕, 对应区块才能提交. 在区块内的交易设置为 1 000 笔的情况下, 基于 2PC 的执行方法的交易延迟随着跨片率的上升而急剧增加, 在高跨片率下延迟达到 837 ms. 这是因为在高跨片负载下大量的块内交易堆积, 执行速度缓慢, 而区块提交的时间也会因此延长.

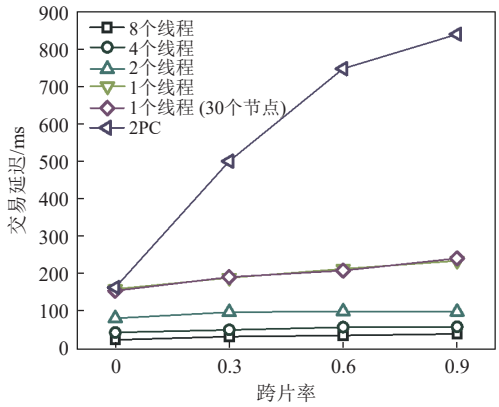


Fig. 11 Transaction latency

图 11 交易延迟

反观本文提出的跨片交易执行方法, 无论在任何跨片负载下, 甚至系统扩展至 30 个节点, 交易延迟都能够保持在较低水平, 且趋势几乎为一条水平线. 同时, 与 2PC 协议不同, 本文方法还可以更灵活地利用多线程的优势来加速跨片交易的执行, 无需考虑协调者的数量与开销.

4.3.3 系统扩展性

图 12 显示了 CoCoS 系统在不同分片数量的执行分片下的系统吞吐量. 实验使用的交易集包含 1 000 万名客户和相应的 1 000 万账户, 交易的读写集均匀分布在 1 000 万数量集的账户里. 同时, 本实验给每个分片配备了 2 个工作线程来执行交易. 本测试将 CoCoS 系统分片数量从 1 扩展至 12, 并通过不同分片数量下的吞吐量表现来衡量系统的可扩展性. 实验表明, 随着分片数量的增加, 系统的吞吐量呈线性增长趋势. 当扩展到 12 个分片时, CoCoS 系统的吞吐量达到了近 75 000 TPS.

随着分片数量的不断增加, 跨片交易比例也不断上升, 从而导致分片单点性能下降, 交易延迟略微

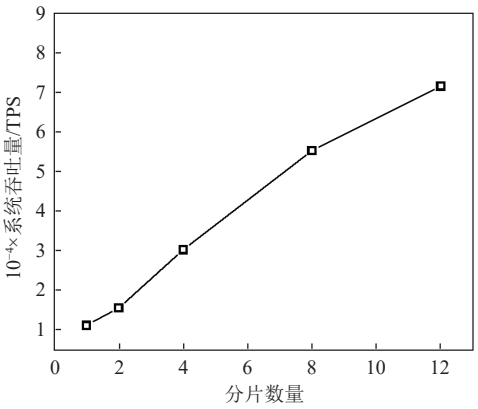


Fig. 12 System scalability

图 12 系统可扩展性

增加. 然而, 整个系统的性能随着分片数量的增加而线性提升, 这也符合对系统扩展性的定义. 综合本文的所有实验, 可以证明本节设计的 CoCoS 系统在系统吞吐量、交易延迟和可扩展性方面远优于采用两阶段提交协议的系统.

4.3.4 系统容错恢复

图 13 展示了 CoCoS 系统在面对各种类型攻击时的容错能力. 本文通过模拟传输节点的宕机或作恶行为, 使得执行节点无法收到状态数据或接收到错误的状态数据.

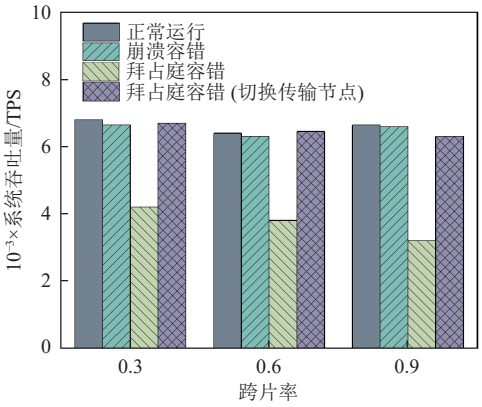


Fig. 13 System fault tolerance

图 13 系统容错

图 13 表明, 在单一工作线程的条件下, 传输节点的崩溃并不会严重影响交易执行. 这是因为本文提出的执行方法是非阻塞的, 当一笔跨片交易无法满足执行条件时, 系统会将其挂起, 并继续处理其他交易, 直到所需的状态数据到达. 当节点收到拜占庭节点的错误数据时, 会产生额外的开销用于重做交易. 随着跨片率的提高, 节点受到攻击的频率也会增加, 导致需要重做的交易数量增加, 从而降低系统的吞吐量. 此外, 当节点检测到恶意的传输节点时, 可以选择拒绝该节点的消息, 并从其他诚实的传输节点获取数据; 由于不再从恶意传输节点获得状态数据, 系统的吞吐也不会因重做交易而降低.

4.4 交易重排序方法性能

本节将分别介绍排序锁机制的原方案、引入交易重排序方案和针对跨片交易优化后的方案的实验结果. 节点的配置与 4.3 节的实验相同. 此外, 本节设置了 2 个不同的交易集, 分别是跨片冲突和片内冲突, 以展示交易重排序方案在不同性质负载冲突下的效率.

4.4.1 原始的跨片交易执行方法

图 14 和图 15 分别展示了传统排序锁机制在面对不同冲突率和不同冲突类型时的表现.

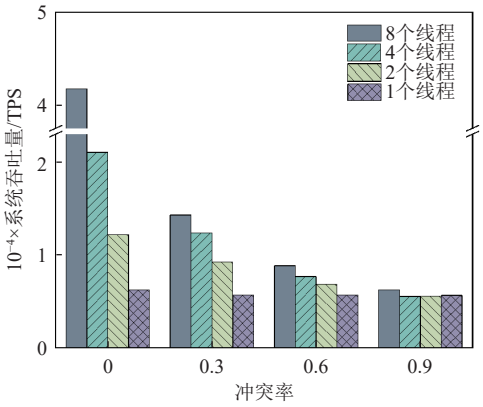


Fig. 14 Throughput of ordered locking mechanism under intra-conflict

图 14 采用排序锁机制的片内冲突下的吞吐量

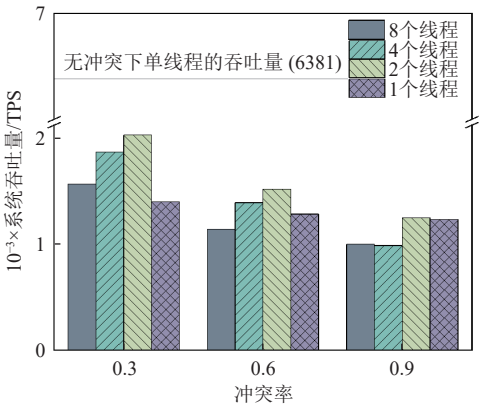


Fig. 15 Throughput of ordered locking mechanism under cross-conflict

图 15 采用排序锁机制的跨片冲突下的吞吐量

通过实验可以看出, 基于排序锁的并发控制方法不能高效地处理冲突交易. 当交易不存在冲突时, 交易之间不会同时访问一个状态数据, 因此锁线程可以无阻塞地向这些交易授予锁, 所有交易可以完全地并行执行. 在这种情况下, 基于排序锁的并发控制方法的性能和乐观并发控制方法的性能相当, 在 8 个工作线程的条件下单分片的吞吐量可以达到 45 000 TPS.

但随着冲突率的增加, 基于排序锁机制的并发控制方法的性能急剧降低. 由图 14 可知, 在冲突率为 0.3 的情况下, 吞吐量下降到原来的 30% 左右. 产生这种现象的原因在于, 根据基于锁的并发控制的特点, 在高冲突负载下, 多个交易会同时请求对同一状态数据进行大量的加解锁操作, 线程在等待锁和释放锁的过程中来回颠簸, 从而影响了系统整体的性能. 冲突严重时, 系统近乎串行地执行交易. 此时,

即便拥有多个工作线程，系统也难以利用它们高效地执行交易，反而多线程频繁的上下文切换会消耗大量的CPU资源，从而降低系统的吞吐量.

特别是在基于排序锁机制的分片区块链系统中，跨片交易冲突对系统性能的影响是灾难性的. 由图15可知，在负载冲突率为0.3的情况下，分片的吞吐量最高仅为2 031 TPS，仅为同条件下片内交易冲突的22%；而在面对高冲突率负载时，系统性能更加糟糕；在8个工作线程的条件下，系统性能仅为同条件下片内冲突的15%. 随着线程数量的增加，系统的处理性能会进一步下降. 具体而言，高冲突场景下跨片交易的状态传输延迟和锁机制本身的缺陷是系统性能恶化的主要原因. 在跨片交易冲突率达到0.9的情况下，将线程数量从4增加到8时几乎没有对系统吞吐量产生提升，此时系统的性能瓶颈在于激烈的锁资源竞争. 因此，采用更为高效的策略来应对高冲突的场景是必要的.

4.4.2 引入交易重排序的跨片交易执行方法

图16展示了排序锁机制在引入交易重排序策略后，系统在处理片内交易冲突时的性能，其中吞吐比为引入交易重排序方案的吞吐量和基于排序锁方案的吞吐量的比值. 在高冲突的交易负载下，系统可以利用8个工作线程来达到41 000 TPS的吞吐量. 与执行无冲突的交易集相比，采用交易重排序方法的系统在处理高冲突交易集的表现依旧出色，其吞吐量降低了不到10%. 不难发现，交易重排序方案通过将交易的顺序进行调换来增加交易执行的并发度，缩短了交易的阻塞时间. 此外. 根据交易子集划分算法，

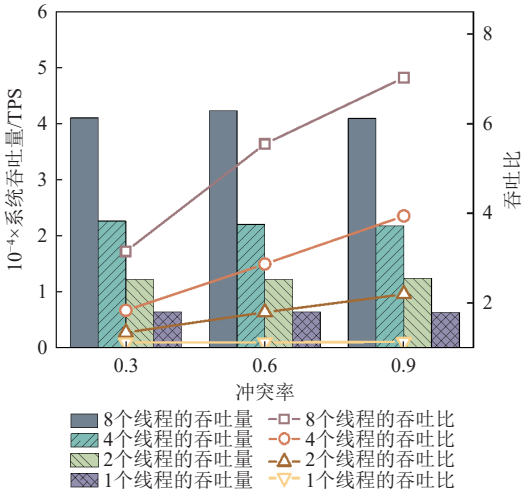


Fig. 16 Throughput of scheme with reorder mechanism under intra-conflict

图16 引入交易重排序机制的片内冲突下的吞吐量

锁线程还可以并行给同一个交易子集内部的交易上锁，突破了单点瓶颈问题. 如此一来，尽可能多的交易能够通过交易重排序机制迅速获得锁资源并执行. 总而言之，该并发执行方法有效地利用多核处理器执行交易，显著增加了系统的吞吐量，整体性能明显优于原排序锁机制.

而图17展示了排序锁算法在引入交易重排序后，系统在处理跨片交易冲突时的性能，其中吞吐比为引入交易重排序方案的吞吐量和基于排序锁方案的吞吐量的比值. 实验数据表明，引入重排序方案的系统在处理跨片交易冲突时表现依然优秀. 在采用4个工作线程的条件下，系统的吞吐量基本维持在20 000 TPS左右，并且不会因负载的变化而大幅度下降. 与片内交易不同，在处理跨片交易中，交易执行并发度越高，越多的跨片交易可以有效地利用带宽将状态数据更加迅速地发送到更新分片上，提高跨片交易的执行效率. 但是在8个工作线程的情况下，处理跨片冲突的性能略低于处理片内冲突的性能. 这是因为此时整个系统的性能瓶颈已从线程数量转移到了执行跨片交易时的状态传输上，系统已经无法通过增加工作线程数量来提高吞吐量，唯有通过对状态数据传输进行优化，系统的性能才能再进一步提高.

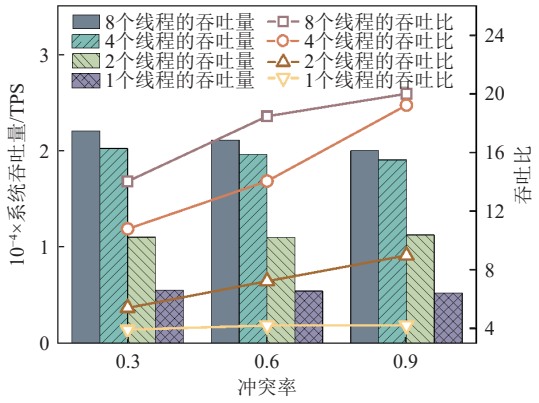


Fig. 17 Throughput of scheme with reorder mechanism under cross-conflict

图17 引入交易重排序机制的跨片冲突下的吞吐量

4.4.3 针对跨片交易的重排序优化策略

图18展示了针对跨片交易优化后的排序锁机制的性能，其中吞吐比为针对跨片交易优化方案的吞吐量和引入交易重排序方案的吞吐量的比值. 该优化策略在8个工作线程下的表现比单纯引入重排序的策略高出6 000 TPS. 由于传输节点提前将某些状态传输打包传输到更新节点，更新分片在处理部分

跨片交易时无需等待数据的传输就可以及时从本地获取，一定程度降低了跨片交易的延迟.

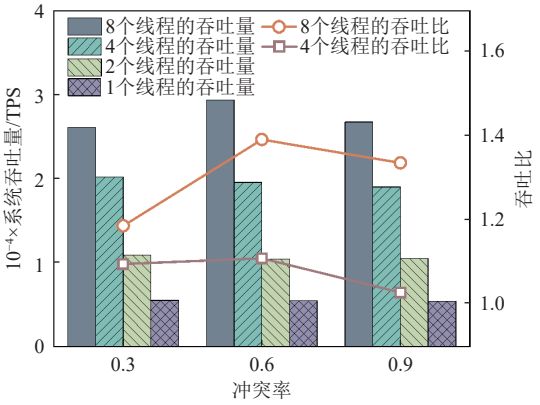


Fig. 18 Throughput of optimized scheme for state transfer

图 18 针对状态传输优化后的吞吐量

5 总 结

本文提出了一种用于分片许可区块链的跨片交易执行方法，用来提高分片系统的性能和降低跨片交易的延迟. 在本文提出的方法中，跨片交易可以通过基于确定性执行的单向通信执行，并且无需协调者的帮助. 此外，本文设计了一种抗冲突的跨片交易执行优化方案，通过交易重排序策略实现支持高并发的交易子集划分，从而提高系统在高冲突负载下的吞吐量和状态传输的效率. 实验结果支持了本文工作的实用性与有效性.

作者贡献声明：阙琦峰提出了研究思路，设计与完成实验；陈之豪负责论文修改和实验设计；张召提出指导意见并修改论文；杨艳琴与周傲英负责审阅论文并提出指导意见.

参 考 文 献

[1] Luu L, Narayanan V, Zheng Chaodong, et al. A secure sharding protocol for open blockchains[C] //Proc of the 2016 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2016: 17−30

[2] Kokoris-Kogias E, Jovanovic P, Gasser L, et al. OmniLedger: A secure, scale-out, decentralized ledger via sharding[C] //Proc of 2018 IEEE Symp on Security and Privacy (SP). Piscataway, NJ: IEEE, 2018: 583−598

[3] Zamani M, Movahedi M, Raykova M. RapidChain: Scaling blockchain via full sharding[C] //Proc of the 2018 ACM SIGSAC Conf on Computer and Communications Security. New York: ACM, 2018: 931−948

[4] Wang Jiaping, Wang Hao. Monoxide: Scale out blockchains with asynchronous consensus zones[C] //Proc of Symp on Networked Systems Design and Implementation (BSDL). Berkeley, CA: USENIX Association, 2019: 95−112

[5] Dang H, Dinh T T A, Loghin D, et al. Towards scaling blockchain systems via sharding[C] //Proc of the 2019 Int Conf on Management of Data. New York: ACM, 2019: 123−140

[6] Al-Bassam M, Sonnino A, Bano S, et al. Chainspace: A sharded smart contracts platform[C/OL] //Proc of the 2018 Network and Distributed System Security Symp. Reston, VA : the Internet Society, 2018[2022-01-12]. http://dx.doi.org/10.14722/ndss.2018.23241

[7] Hellings J, Sadoghi M. Byshard: Sharding in a Byzantine environment[J]. Proceedings of the VLDB Endowment, 2021, 14(11): 2230−2243

[8] Faleiro J M, Abadi D J. Rethinking serializable multiversion concurrency control[J]. Proceedings of the VLDB Endowment, 2015, 8(11): 1190−1201

[9] Faleiro J M, Abadi D J, Hellerstein J M. High performance transactions via early write visibility[J]. Proceedings of the VLDB Endowment, 2017, 10(5): 613−624

[10] Thomson A, Diamond T, Weng S C, et al. Calvin: Fast distributed transactions for partitioned database systems[C] //Proc of the 2012 ACM SIGMOD Int Conf on Management of Data. New York: ACM, 2012: 1−12

[11] Lu Yi, Yu Xiangyao, Cao Lei, et al. Aria: A fast and practical deterministic OLTP database[J]. Proceedings of the VLDB Endowment, 2020, 13(12): 2047−2060

[12] Dickerson T, Gazzillo P, Herlihy M, et al. Adding concurrency to smart contracts[C] //Proc of the ACM Symp on Principles of Distributed Computing. New York: ACM, 2017: 303−312

[13] Anjana P S, Kumari S, Peri S, et al. An efficient framework for optimistic concurrent execution of smart contracts[C] //Proc of 2019 27th Euromicro Int Conf on Parallel, Distributed and Network-Based Processing (PDP). Piscataway, NJ: IEEE, 2019: 83−92

[14] Zhang An, Zhang Kunlong. Enabling concurrency on smart contracts using multiversion ordering[C] //Proc of Asia-Pacific Web (APWeb) and Web-Age Information Management (WAIM) Joint Int Conf on Web and Big Data. Berlin: Springer, 2018: 425−439

[15] Chen Zhihao, Qi Xiaodong, Du Xiaofan, et al. PEEP: A parallel execution engine for permissioned blockchain systems[C] //Proc of Int Conf on Database Systems for Advanced Applications. Berlin: Springer, 2021: 341−357

[16] Jin Cheqing, Pang Shuaifeng, Qi Xiaodong, et al. A high performance concurrency protocol for smart contracts of permissioned blockchain[J]. IEEE Transactions on Knowledge and Data Engineering, 2021, 34(11): 5070−5083

[17] Amiri M J, Agrawal D, El Abbadi A. SharPer: Sharding permissioned blockchains over network clusters[C] //Proc of the 2021 Int Conf on Management of Data. New York: ACM, 2021: 76−88

[18] Huang Huawei, Peng Xiaowen, Zhan Jiazhou, et al. BrokerChain: A cross-shard blockChain protocol for account/balance-based state sharding[C] //Proc of IEEE Conf on Computer Communications (IEEE INFOCOM 2022). Piscataway, NJ: IEEE, 2022: 1968−1977

[19] Hong Zicong, Guo Song, Li Peng, et al. Pyramid: A layered sharding blockchain system[C] //Proc of IEEE Conf on Computer Communications (IEEE INFOCOM 2021). Piscataway, NJ: IEEE, 2021: 1−10

[20] Hellings J, Sadoghi M. The fault-tolerant cluster-sending problem[C] //Proc of Int Symp on Foundations of Information and Knowledge Systems. Berlin: Springer, 2022: 168−186

Que Qifeng, born in 1998. Master candidate. His main research interest includes execution of smart contracts in blockchain.

阙琦峰, 1998 年生. 硕士研究生. 主要研究方向为区块链中智能合约的执行.

Chen Zhihao, born in 1996. PhD candidate. His main research interest includes performance and scalability optimization of blockchain systems.

陈之豪, 1996 年生. 博士研究生. 主要研究方向为区块链系统性能与扩展性优化.

Zhang Zhao, born in 1977. PhD, professor, PhD supervisor. Member of CCF. Her main research interests include distributed database and blockchain data management.

张　召, 1977 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为分布式数据库、区块链数据管理.

Yang Yanqin, born in 1977. PhD, associate professor. Member of CCF. Her main research interests include compile optimization, embedded systems, and blockchain technology.

杨艳琴, 1977 年生. 博士, 副教授. CCF 会员. 主要研究方向为编译优化、嵌入式系统、区块链技术.

Zhou Aoying, born in 1965. PhD, professor, PhD supervisor. Member of CCF. His main research interests include database systems and blockchain data management.

周傲英, 1965 年生. 博士, 教授, 博士生导师. CCF 会员. 主要研究方向为数据库系统、区块链数据管理.