计 算 机 研 究 与 发 展
Journal of Computer Research and Development

DOI: 10.7544/issn1000-1239.202220644
61(8): 2110−2126, 2024

一种面向联盟链 Hyperledger Fabric 的并发冲突事务优化方法

吴海博[1,2] 刘 辉[1,2] 孙 毅[3,4] 李 俊[1,2]

[1](中国科学院计算机网络信息中心 北京 100190)

[2](中国科学院大学 北京 100049)

[3](山东省区块链金融重点实验室(山东财经大学) 济南 250014)

[4](中国科学院计算技术研究所 北京 100190)

(wuhaibo@cstnet.cn)

A Concurrent Conflict Transaction Optimization Method for Consortium Blockchain Hyperledger Fabric

Wu Haibo[1,2], Liu Hui[1,2], Sun Yi[3,4], and Li Jun[1,2]

[1](*Computer Network Information Center*, *Chinese Academy of Sciences*, *Beijing* 100190)

[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)

[3](*Shandong Key Laboratory of Blockchain Finance* (*Shandong University of Finance and Economics*), *Jinan* 250014)

[4](*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)

Abstract With the prevalence of blockchain technology, Hyperledger Fabric (Fabric for short), as a well-known open source blockchain platform, has received wide attention. However, Fabric still suffers from conflicts between concurrent transactions. Conflicts will cause a large number of invalid transactions entering the chain, resulting in a decrease in throughput and hindering its development. For this problem, existing intra-block-conflict-oriented schemes lack efficient conflict detection and avoidance methods, and ignore the adverse impact of inter-block conflicts on throughput. We propose an optimization scheme for Fabric, Fabric-HT (Fabric with high throughput), from both intra-block and inter-block aspects to effectively reduce concurrency inter-transaction conflicts and improve system throughput. For intra-block transaction conflicts, we present a transaction scheduling mechanism, in which an efficient data structure (the dependency chain) is defined to identify and abort transactions with "dangerous structures" in advance, reasonably schedule transactions and eliminate conflicts; for inter-block transaction conflicts, the conflict transaction detection is moved to the sorting node to complete, and an early conflict transaction avoidance mechanism following "push-match" pattern is established. A large number of experiments are carried out in multiple scenarios, and the results show that Fabric-HT overperforms existing schemes in terms of throughput, transaction abort rate, average transaction execution time, and invalid transaction space occupancy. The results show that the throughput of Fabric-HT can reach up to 9.51x that of Fabric and 1.18x of the latest optimized scheme FabricSharp; compared with FabricSharp, the space utilization is increased by 14%. In addition, Fabric-HT also shows good robustness and anti-attack ability in solving concurrent transaction conflict.

收稿日期: 2022-07-24; **修回日期**: 2023-10-20

基金项目: 国家重点研发计划项目(2021YFE0111500); 中国科学院国际伙伴计划(241711KYSB20200023); 开放科学基础设施治理前沿研究(CNIC20220101)

This work was supported by the National Key Research and Development Program of China (2021YFE0111500), the International Partnership Program of the Chinese Academy of Sciences (241711KYSB20200023), and the Frontier Research on Open Science Infrastructure Governance (CNIC20220101).

通信作者: 李俊(lijun@cnic.cn)

Key words concurrency conflict; blockchain; Hyperledger Fabric; transaction scheduling; conflict detection

摘 要 随着区块链技术应用的普及，联盟链 Hyperledger Fabric（简称 Fabric）已成为知名区块链开源平台，并得到广泛关注. 然而 Fabric 仍受困于并发事务间冲突问题，冲突发生时会引发大量无效交易上链，导致吞吐量下降，阻碍其发展. 对于该问题，现有面向块内冲突的方案缺乏高效的冲突检测和避免方法，同时现有研究往往忽略区块间冲突对吞吐量的不利影响. 提出了一种 Fabric 的优化方案 Fabric-HT（fabric with high throughput），从区块内和区块间 2 方面入手，有效降低事务间并发冲突和提高系统吞吐量. 针对区块内事务冲突，提出了一种事务调度机制，根据块内冲突事务集定义了一种高效数据结构——依赖关系链，识别具有“危险结构”的事务并提前中止，合理调度事务和消除冲突；针对区块间事务冲突，将冲突事务检测提前至排序节点完成，建立以“推送-匹配”为核心的冲突事务早期避免机制. 在多场景下开展大量实验，结果表明 Fabric-HT 在吞吐量、事务中止率、事务平均执行时间、无效事务空间占用率等方面均优于对比方案. Fabric-HT 吞吐量最高可达 Fabric 的 9.51 倍，是最新优化方案 FabricSharp 的 1.18 倍；空间利用率上相比 FabricSharp 提升了 14%. 此外，Fabric-HT 也表现出较好的鲁棒性和抗攻击能力.

关键词 并发冲突；区块链；Hyperledger Fabric；事务调度；冲突检测

中图法分类号 TP391

区块链起源于比特币，经过 10 余年的发展，凭借其去中心化、不可篡改、透明性、可追溯等特性得到了迅猛发展. 整体而言，区块链是以数据库为底层数据存储依托，借助 P2P 对等网络进行信息通信，依靠密码学实现权益保障和隐私保护，通过分布式共识机制达成一致性，从而构建价值交换体系[1]. 区块链早期主要聚集数字货币领域，随着技术的不断完备和发展，目前正应用于溯源、跟踪、取证等多种场景.

区块链按照读写权限的开放程度，可分为公有链、私有链和联盟链. 公有链允许任何节点参与链的读写和维护，具有较强去中心化特征，尤以比特币[2]和以太坊[3]等为代表；私有链由单个区块链服务客户使用，仅有授权的节点才能接入，并按照规制参与和读写数据；联盟链是仅允许联盟内部节点参与链的读写与维护的区块链，具有较弱的去中心化特征，相比于公有链，联盟链具有处理速度快、松散共识、隐私性好等优点，目前在商品溯源、版权存证等领域已有广泛应用.

联盟链发展至今，Hyperledger Fabric[4-5]（以下简称 Fabric）一直是其中应用最为广泛的代表方案，它的模块化和通用设计适应于多种企业级应用，得到了学术界和产业界的广泛关注.

Fabric 采用“执行—排序—验证”（execute-order-validate, EOV）[6]三段式架构，将智能合约与共识信任分离，提高了合约信任的灵活性以及系统的可扩展性，但同时也加剧了并发事务冲突问题[7-9]. 区块中的冲突事务在验证阶段被检测并标记为无效，但限于区块达成共识后不可更改，冲突事务仍执行上链操作. 如此以来，既降低了系统吞吐量又增加了延迟，同时被标记为无效的事务占用了系统的存储空间，耗费计算资源. 因此如何有效减少 Fabric 中的并发冲突[10-14]事务，成为目前国内外研究的焦点问题.

针对该问题，一些现有工作进行了初步探索. Xu 等人[11]设计了一种包含锁机制和账本存储的 LMLS 方法来改善“写-写”并发冲突，引入 Redis 数据库来实现锁定机制，然而该方法难以解决“双花”问题，且锁定机制的引入在并发环境下事务执行效率低下. Nasirifard 等人[12]提出将非冲突复制数据结构（CRDT）集成到 Fabric 中用于解决并发冲突问题，将冲突事务的事务值在提交到账本之前通过 CRDT 技术自动合并. 然而由于合并 CRDT 会引入额外时间开销，从而会导致更高的提交延迟. Sharma 等人[13]提出了 Fabric++，其通过事务冲突循环检测以及事务提前中止解决高并发场景下块内冲突问题. 然而对事务冲突图执行循环检测的方式对计算性能提出了更高的要求，无法适应于工业生产环境[15-16]. 同时 Fabric++在块间冲突问题的解决上无法处理多客户端发起的并发冲突事务[17]. FabricSharp[18]从细粒度出发提出了事务重排序方案，以依赖图的形式检测循环冲突，然而该方案未能考虑块间冲突场景且依赖图的检测算法，时间复杂度较高，节点规模较大时耗时过长.

综上所述，以 Fabric 和 FabricCRDT 为代表的传统方案常采用悲观并发控制的方式来解决冲突，存在高并发场景下事务处理时间过长、效率较低等问题；以 Fabric++和 FabricSharp 为代表的优化方案，存在块内冲突检测复杂度过高、块间冲突有待优化等

问题. 为此，本文提出了 Fabric-HT 优化方法，在区块内提出高效的冲突检测和避免算法；并在块间引入冲突检测方案，将块间冲突检测提前至排序阶段完成，从块内、块间 2 个维度进一步提升 Fabric 性能.

首先，我们基于已有研究工作，对冲突事务的形成原因进行了充分的理论分析和实验佐证. 根据形成原因的不同分为 2 种情形：区块内冲突与区块间冲突，并分别针对这 2 类冲突提出了优化方法. 在保证不改变原有架构的基础上实现提高系统吞吐量和并发性能的目的. 具体而言，本文的主要贡献有 3 点：

1)针对区块内冲突问题，设计了一种高效事务调度机制. 面向块内冲突事务提出了依赖关系链和并发事务危险结构条件，中止少量事务消除冲突并生成无冲突事务的执行序列.

2)针对区块间冲突问题，系统性提出相关解决方案，建立以“推送-匹配”为核心的冲突事务避免机制. 该机制可在线性时间内完成冲突事务的匹配性检测，保证链上空间的合理利用.

3)为验证 Fabric-HT 方案的有效性，在多种工作负载场景下开展大量实验. 结果表明所提方案与其他代表性方案相比，在改善吞吐量、降低无效事务、节约存储空间等方面具有明显优势.

1 相关工作

作为联盟链的典型代表，Fabric 近年来得到广泛关注，存在许多致力于其性能优化的工作. Thakkar 等人[16] 提出了 Fabric 的性能基准，研究各种配置参数对事务吞吐量和延迟的影响，并指出当下存在的 3 个性能瓶颈：背书策略验证、块中事务的顺序策略验证、状态验证和提交. Gorenflo 等人[17] 提出了 Fast-Fabric，使 Fabric 每秒能够处理 2 万个事务. 同时关注了共识机制之外的性能瓶颈：将事务 ID 从数据块中分离，降低 Kafka 共识的吞吐量；利用基于内存的哈希表取代世界存储状态，加快数据访问；使用线程并行处理验证过程. Xu 等人[19] 提出了一种计算不同网络结构(如区块大小、块间隔等)下事务延迟的理论模型，确定 Fabric 存在的一些性能瓶颈，并提出可行性建议. Sousa 等人[20] 为 Fabric 设计并实现了 BFT-SMART，使排序服务阶段可以每秒处理多达 10 000 个事务且在 0.5 s 内不可撤销的区块链中写入 1 个事务. Nakaike 等人[21] 关注账本访问和数据存储所引发的性能瓶颈，提出关闭 LevelDB 数据压缩机制、执行联合访问查询、调整数据库大小等方法. Raman 等人[22] 研究了在使用区块链存储大数据集时，使用有损压缩来降低 Fabric 背书节点和记账节点之间共享状态的通信成本，然而有损压缩在区块链中的适用场景极其有限. Dinh 等人[23] 使用衡量私有区块链性能的工具 BlockBench 来研究区块链的性能，并比较 Fabric、以太坊和 Parity 的性能优劣. 虽然上述研究针对 Fabric 结构、事务验证、共识机制等方面进行了优化，但很多优化结果仍不理想.

并发冲突问题作为分布式系统中的研究热点，在区块链同样有着深远影响. 现有很多研究针对 Fabric 中的并发冲突提出了一些解决方案. Xu 等人[11] 设计了一种包含锁机制和账本存储的 LMLS 方法来改善“写-写”并发冲突，引入 Redis 数据库来实现锁定机制. 然而该方法无法解决区块链中著名的双花问题. Nasirifard 等人[12] 将近年来提出的非冲突复制数据类型(CRDT)集成到 Fabric 中用于解决并发冲突问题，对于冲突事务的事务值在提交到账本之前，通过 CRDT 技术自动合并，成功合并所有冲突事务. 然而由于合并 CRDT 会引入额外时间开销，从而会导致更高的提交延迟. Meir 等人[24] 提出了一种新的无锁方法用于提供事务隔离. 该方法利用 Savepoint 作为事务之间的边界，使用它来检测模拟的事务是否违反并发提交事务的边界，当检测到事务违例时中止事务模拟. 虽然该方法能够有效检测冲突事务，但对冲突事务的处理并不友好. Zhang 等人[25] 能够有效解决同一客户端发起的多笔交易事务之间引发的冲突，通过将冲突事务放入缓存队列中来避免无效交易的产生. 然而该方法并未考虑高并发场景下多客户端间交易引发的冲突问题，存在一定局限性. Sharma 等人[13] 提出了 Fabric++，通过引入事务重排序机制以及事务提前中止可有效解决高并发场景下块内冲突问题，然而对事务冲突图执行循环检测的方式对计算性能提出了更高的要求无法适应于工业生产环境[15-16]，易造成更多有效事务的流失. Ruan 等人[18] 从细粒度出发提出了事务重排序方案，以依赖图的形式检测循环冲突，然而方案未能考量块间冲突场景且依赖图的检测算法受节点规模的增大表现出疲软性.

与上述现有优化方案相比，Fabric-HT 根据冲突成因的不同，分别从区块内和区块间 2 个角度提出不同的优化方案，保证以提前中止最小数量的冲突事务实现非冲突事务成块的目的，且不改变原有功能的架构设计，具有较好兼容性.

2 背景知识

Fabric 是目前最具代表性的联盟链系统之一，已形成了强大可持续社区和繁荣生态系统. Fabric 目前已经广泛应用于金融、保险、医疗、供应链等行业[26-27]. 此处简要介绍 Fabric 节点类型以及 EOV 架构下的事务执行流程.

2.1 节点类型

Fabric 网络主要由 3 类节点组成：

1）客户端. 客户端是用户向 Fabric 网络发送请求的接口，是事务的发起者，通常通过软件开发包与平台通信.

2）Peer 节点. Fabric 主要包含背书节点、主节点、记账节点 3 类 Peer 节点. 背书节点负责模拟执行交易提案并签名背书；主节点负责与排序服务通信并在节点间传播区块数据；记账节点负责验证交易合法性并提交账本，维护交易记录.

3）排序节点. 排序节点负责为网络中所有合法交易进行全局排序，并将一批排序后的交易组合生成区块结构. 所有客户端提交的事务都需要进入排序服务阶段生成区块.

2.2 事务执行流程

按照 Fabric 的 EOV 架构的设计思想，Fabric 将整个事务流程分为执行、排序、验证 3 个阶段. 下面介绍各阶段处理过程.

1）执行阶段. 此阶段又名背书阶段，客户端向背书节点提交交易提案并请求背书，背书节点的选择由背书策略决定. 背书节点负责检查提案消息并模拟执行. 检查通过后，背书节点将启动链码容器模拟执行交易提案，并将模拟结果暂时保存在交易模拟容器中，等待排序达成共识和交易验证，而不是直接更新到账本中，交易模拟结果采用状态数据读写集记录交易造成的状态变更结果. 背书节点将提案响应消息及模拟结果返回给客户端.

2）排序阶段. 一旦客户端接收到满足背书策略的响应消息后，会创建一个事务. 该事务包含有效载荷、元数据等内容，并将其发送给排序节点，受信任的排序服务将所有收到的事务按照一致性协议（共识机制）建立全局顺序，并按照出块规则将事务切割并打包成一定数量的区块. 默认情况下，事务基本按照到达服务的时间戳排序.

3）验证阶段. 首先组织中的记账节点负责验证排序后的交易数据，检查交易结构格式的正确性，调用校验系统链码（validation system chain code, VSCC）验证交易背书签名是否满足指定的背书策略；其次执行多版本并发控制（multi-version concurrency control, MVCC）检查交易数据读写集的版本冲突等，将冲突事务标记为无效. 验证完成后保存所有的区块数据到区块数据文件中，将最新的有效交易数据更新到状态数据库；最后将区块数据中经过背书的有效交易数据保存到历史数据库.

图 1 展示了 3 个事务 T_1, T_2, T_3 从发起到上链的 3 阶段流程. 事务由客户端发起，在节点 A 和节点 B 上背书，客户端收集足够数量的背书响应后交由排序服务，在 Kafka 共识后生成区块，最后进入验证阶段，节点 A 和节点 B 分别验证区块，并将有效事务提交账本执行上链.

2.3 事务读写集

Fabric 采用基于 KV 键值对的状态数据模型 KV-Ledger 作为账本. 账本支持读取状态数据、写状态数据和删除状态数据. 背书节点收到客户端发送的待处理事务提案后，首先模拟事务提案执行并签名背书. 提案模拟结果以读写集的形式返回给客户端. 一个事务的读写集由 2 部分组成：

1）读集合〈键，版本〉. 读集合中包括一个键列表和对应键的版本号.

2）写集合〈键，是否删除，值〉. 写集合中包含最终要提交给账本的键值对以及是否删除标记.

在验证阶段，记账节点会基于本地状态数据库对区块中所包含的交易执行 MVCC 检查，用于验证交易结果中读集合是否有效. 具体方法为读集合中的键和版本与当前状态中的键和版本进行比较. 如果版本不匹配，事务将被标记为无效. 最终将有效交易添加到更新账本操作中. 尽管存在标记的无效交易，但 Fabric 不会将其剔除掉，依然将其和有效交易共同上链.

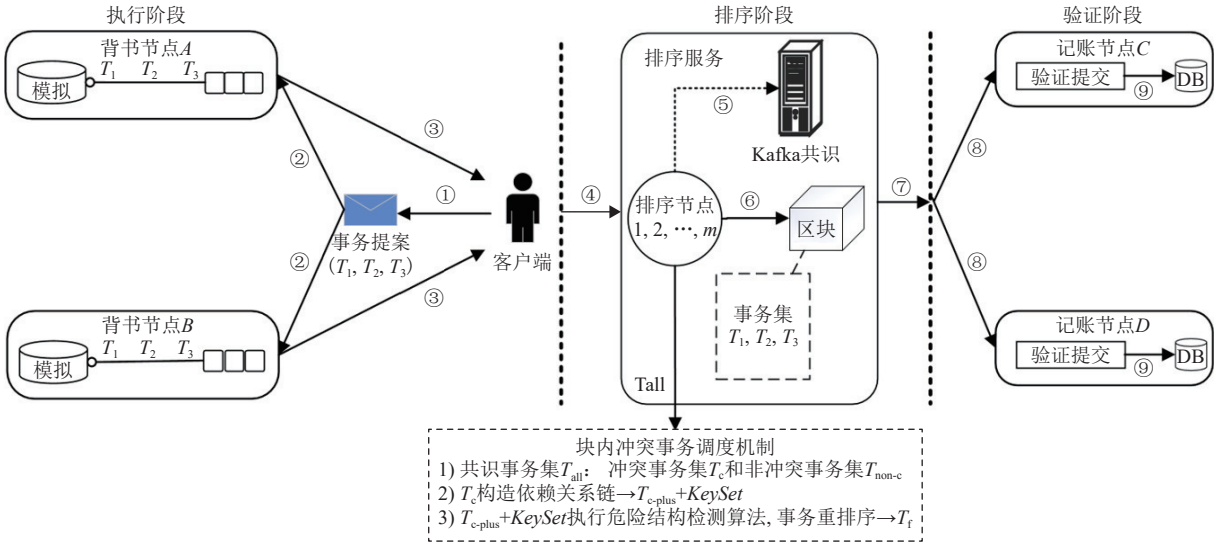
2.4 事务间依赖关系

事务间依赖关系反映了不同事务之间的关联性. 并发事务间可能造成冲突的依赖关系有 3 种[18,28-34]：

1）$T_i \xrightarrow{\text{w:r}} T_j$（写读依赖）. T_i写入数据 x_i，并且 T_j在之后读取 x_i. 则事务 T_i和 T_j构成写读依赖关系. T_i应在 T_j之前序列化.

2）$T_i \xrightarrow{\text{r:w}} T_j$（读写反依赖）. T_i首先读取数据 x_i，而 T_j在之后更新该数据为 x_{i+1}. 则事务 T_i和 T_j构成读写反依赖关系. T_i应在 T_j之前序列化.

3）$T_i \xrightarrow{\text{w:w}} T_j$（写写依赖）. T_i写入数据 x_i，并且 T_j在之



①客户端形成事务提案；②将事务提案提交给多个背书节点；③多个背书节点返回背书结果；④客户端在接受到满足背书策略的响应消息后，将形成一个事务并提交给排序节点；⑤排序节点事务提交给排序服务，排序服务将所有收到的事务按照一致性协议（共识机制）建立全局顺序；⑥排序节点按照出块规则切割并打包成一定数量的区块；⑦排序节点将区块分发给多个组织的记账节点；⑧记账节点负责验证排序后的交易数据；⑨记账节点将区块数据中经过背书的有效交易数据保存到历史数据库.

Fig. 1 Three-phase transaction flow

图 1 3 阶段事务流

后更新该数据为x_{i+1}. 事务T_i和T_j构成写写依赖关系.

以上 3 种依赖关系通过事务重排序, 可以有效解决冲突问题. 当并发事务间的依赖关系较为复杂时, 易构成循环依赖, 此类问题无法通过事务重排序解决.

4) $T_i \overset{r:w}{\rightarrow} T_j \overset{w:r}{\rightarrow} T_i$(循环依赖). T_i读取数据x_i, 而T_j在此之后更新该数据为x_{i+1}. 与此同时T_j写入数据y_i, 并且T_i在之后读取y_i, T_i和T_j就构成了循环依赖.

3 问题剖析

本节主要介绍影响 Fabric 系统吞吐量的 2 种主要冲突类型: 区块内冲突和区块间冲突. 同时对冲突原因进行理论分析.

3.1 区块内冲突

区块内冲突是指同一区块内的不同事务读写集合之间的冲突, 发生于构造区块时将对同一键操作的事务划分到同一区块中. 为了方便阐述冲突的形成原因, 此处举例说明. 假设在区块链的状态数据库中已存储 2 个数据K_1和K_2, 它们的初始版本和键值都为V_0和*ValueA*, 图 2 展示了在并发场景下同一区块X内有T_1, T_2, T_3共 3 个事务依序执行的过程, 每个事务有自己单独的读写集列表, 首先T_1事务把键K_1和K_2的值分别做出了更新*ValueA*→*ValueB*, 使它们的版本都更新到了V_1, 并通过了事务检测. 其次继续验证T_2事务, 执行 MVCC 检测时发现K_1的版本$V_0 \neq V_1$, 冲突检测不通过, T_2事务被标记为无效事务. 同理验证T_3事务时发现K_2的版本$V_0 \neq V_1$, 因此也被标记为无效事务. 类似这种在同一区块内先读后写的情况下, 造成键版本更新使靠后事务的读集合无法通过 MVCC 冲突检测的状况, 我们称之为区块内冲突.

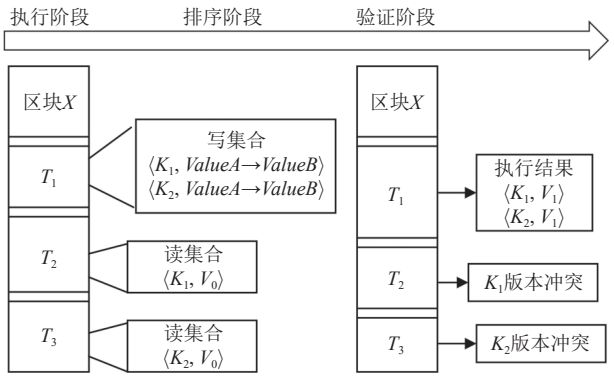


Fig. 2 Example of intra-block conflicts

图 2 区块内冲突举例

区块内冲突的根本原因在于并发环境下同一区块内 2 个事务对相同键的先写后读. 写事务的提交更新了当前数据库中该元组的版本, 造成靠后事务的读集合版本较低且验证不通过. 此类冲突问题在数据库系统中表现为读写反依赖.

3.2 区块间冲突

与区块内冲突不同, 区块间冲突是发生在不同区块之间. 这是由于 EOV 架构的天然特性, 使得事务

的模拟执行与验证提交之间存在较大延迟，导致靠后的区块中事务出现脏读，最终引发读写冲突. 如图 3 所示，有 2 个事务 T_1 和 T_2. 首先 T_1 和 T_2 依次背书，获得的 K_1 的版本都为 V_0，当 T_1 背书之后进入排序，在构造区块以及验证阶段，更新了 K_1 的值，并将 K_1 的版本 V_0 更新为 V_1，然而此时 T_2 模拟结果的读集合 K_1 仍旧处于 V_0 版本，直到提交账本之前并不会检测到这个过时的读取. 在后续验证阶段 MVCC 检测时 T_2 无法通过，被标记为无效.

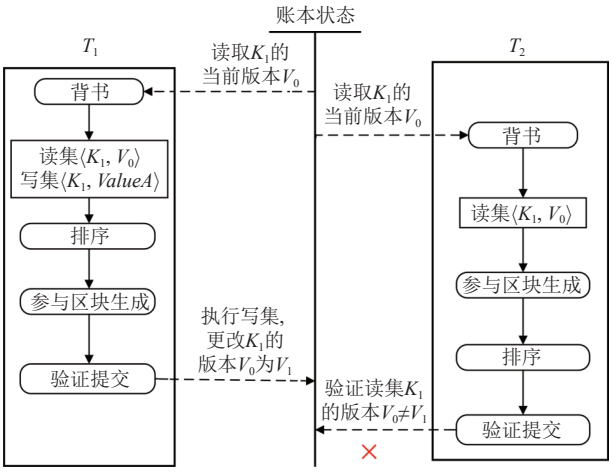


Fig. 3 Cause diagram of inter-block conflicts

图 3 区块间冲突成因图

4 Fabric-HT 方案设计

基于对上述 2 类事务间冲突原因的分析和判断，分别提出针对块内和块间的 2 种优化方案，这 2 种方案可在排序节点上同时运行.

针对块内冲突，提出了一种块内冲突事务调度机制. 首先提出一种 2 维链式结构“依赖关系链”；其次分析了构成冲突问题的 3 种事务间依赖关系，并提出了中止事务的危险结构条件；最后借助“依赖关系链”对事务队列依次执行危险结构检测和中止.

针对区块间冲突，由于块间事务存在一定延迟，无法在同一时段内解决此类问题. 为此，方案设计在排序节点建立缓存区，通过“推送”机制将区块链上的数据推送到缓存区，在排序阶段提前监测区块间事务间可能发生的冲突. 对于监测出的冲突事务，由客户端重新发起和上链.

4.1 块内冲突事务调度机制

调度机制通过合理中止事务，将剩余非冲突事务重排序以构造非冲突事务序列. 首先，提出“事务块”“读写块”“依赖关系链”等数据结构，将包含读写集的事务序列转换为 2 维链式结构. 其次，分析并发环境下造成冲突的 3 种依赖关系，提出了无法通过循环依赖检测的危险结构条件. 最后，基于危险结构条件对块内事务执行串行检测和调度，实现块内无冲突事务上链.

4.1.1 数据结构定义

首先给出“事务块”“读写块”“依赖关系链”等相关结构.

“事务块”的示例如图 4 所示，事务块中包含的具体信息有：*Tx_N* 表示了当前事务唯一的 *ID*. *TimeStamp*

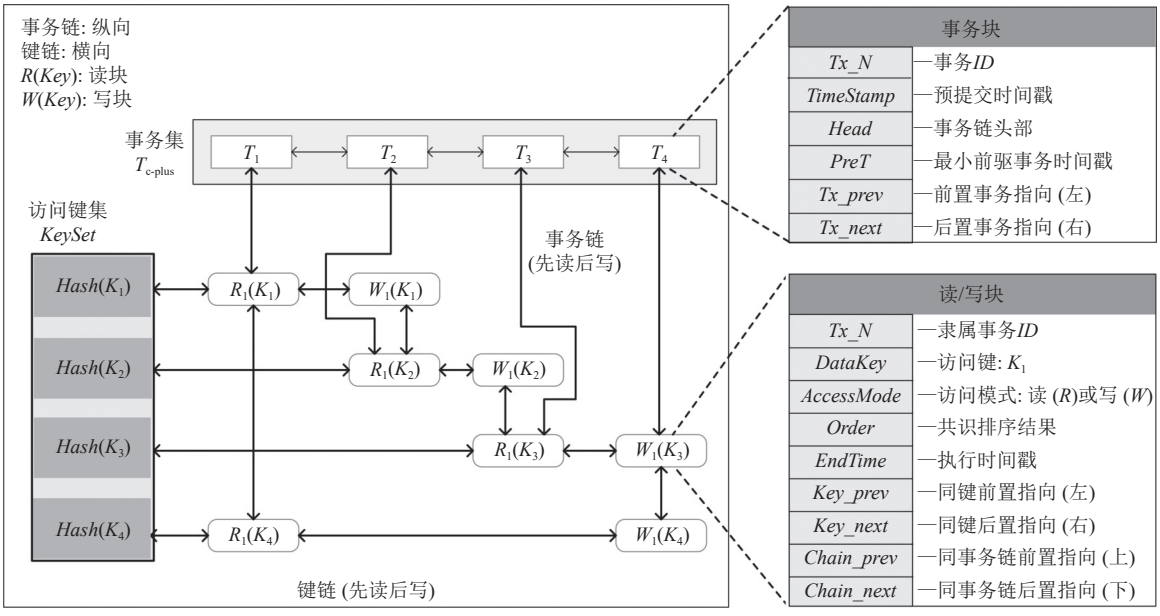


Fig. 4 Dependency relationship chain structure

图 4 依赖关系链结构

表示当前事务的预提交时间戳，该时间戳模拟了实际的事务提交时间. *Head* 作为事务链的头部，指向事务所包含的读写块. *Tx_prev* 指针和 *Tx_next* 指针分别指向当前事务的前置事务和后置事务(事务前置与后置关系由共识确定)，该事务信息用于算法 3 中依赖关系链剪枝时使用.

“读/写块”由事务的读/写集衍生而来，对读/写块内容的分析，可反映各事务间的依赖关系. 其中 *Tx_N* 表示当前读/写块隶属的事务. *DataKey* 表示当前读/写块访问的键，该信息的设定是为了缩短后期检测算法的计算时间. *AccessMode* 用于区分当前读/写块的访问模式(读操作或写操作). *Order* 为当前事务所处事务集中的顺序. *EndTime* 为当前事务的读/写集在执行阶段(或背书阶段)的完成时间，用于反映并发事务间不同读/写集的先后关系. 键链中各节点由前向指针 *Key_prev* 和后向指针 *Key_next* 链接而成. 事务链中各节点由前向指针 *Chain_prev* 和后向指针 *Chain_next* 链接而成.

如图 4 所示，左侧的“访问键集”由哈希值组成，该值由各事务访问值对应的键的哈希值构成. 顶端的“事务集”由共识后的事务集构成. 我们定义了一种 2 维链式结构，由事务链(纵向)和键链(横向)2 种链构成. 事务链由每个事务按照“先读后写”的顺序指向该事务的各个读/写块，用于辅助算法 3 计算各个事务的时间戳；键链由不同事务访问的数据键计算出的映射值，按照“先读后写”的顺序指向对该键访问的各个读/写块. 键链通过访问读/写块来构造各个事务间的依赖关系，辅助计算单个读写操作的重要时间戳.

4.1.2 构造依赖关系链

下面介绍依赖关系链的 2 个算法，其中算法 2 是算法 1 的组成部分.

算法 1. 依赖关系链构造算法.

输入：事务集 T_{all}；

输出：访问键集 *KeySet*〈Map of K_i, *Head*〉，事务集 $T_{c\text{-}plus}$.

① *ConstructChain*(T_c);
② T_c, $T_{non\text{-}c}$ =*NCTDissociate*(T_{all});
/*非冲突事务抽离*/
③ *CurrentTxLink* = null;
④ for T_i in T_c
/*初始化全量事务集 $T_{i\text{-}plus}$，生成事务块*/
⑤ $T_{i\text{-}plus}$. *Tx_prev* = *CurrentTxLink*;
⑥ *CurrentTxLink*. *Tx_next* = $T_{i\text{-}plus}$;
⑦ *CurrentTxLink* = $T_{i\text{-}plus}$;
⑧ *link* = $T_{i\text{-}plus}$. *Head*;
⑨ end for
⑩ for R_i in T_i. *ReadSet*
/*遍历读集，增加读块*/
⑪ *RWblock*=*ConstructRWBlock* (*Tx_N*= T_i. *Tx_N*, *DateKey*=R_i. *key*, *AccessMode*='*r*', *order*=T_c. *index*, *EndTime* =R_i. *time*);
/*键链和事务链块插入算法*/
⑫ *AddBlock* (*RWblock*, *link*, *KeySet*);
⑬ end for
⑭ for R_i in T_i. *WriteSet*
/*遍历写集，增加写块*/
⑮ *RWblock* = *ConstructRWBlock*(*Tx_N*= T_i. *Tx_N*, *DateKey*=R_i. *key*, *AccessMode*='*w*',*order*=T_c. *index*,*EndTime*=R_i. *time*);
/*键链和事务链块插入算法*/
⑯ *AddBlock*(*RWblock*, *link*, *KeySet*);
⑰ end for
⑱ return *KeySet*, $T_{c\text{-}plus}$.

算法 1 给出了依赖关系链的构造过程，遍历冲突事务集得到依赖关系链中的键链和事务链. 主要包含 3 个步骤：

1)事务预处理. 算法 1 中行②作用是将共识后生成的总事务集中的非冲突事务抽离. 非冲突事务采取前置操作，冲突事务作为输入，经过算法 1 行①~②后消除冲突. 非冲突事务的抽离，可以减少依赖关系链的空间占用，同时降低检测算法的计算开销.

2)生成事务块. 算法 1 中行③~⑨表示遍历冲突集 T_c，生成初始化全量事务集 $T_{c\text{-}plus}$. $T_{c\text{-}plus}$ 包含各个事务生成的事务块，其通过指针 *Tx_prev* 和指针 *Tx_next* 链接. 以当前时间赋值 *TimeStamp* 模拟提交阶段的事务时间戳.

3)键链和事务链插入块. 遍历冲突事务的读/写集(*ReadSet*,*WriteSet*)初始化读/写块. 如算法 1 中行⑩~⑯所示，按照“读集生成的读块在前，写集生成的写块在后”的规则，依次将读/写块加入键链和事务链. 行⑱返回访问键集和事务集作为算法 3 的输入.

算法 2. 基于键链和事务链的块插入算法.

输入：读/写块 *RWblock*，事务链指向 *link*，访问键集 *KeySet*；

输出：访问键集 *KeySet*〈Map of K_i, *Head*〉，事务集 $T_{c\text{-}plus}$.

① *AddBlock* (*RWblock*, *link*, *KeySet*);

② if (Map of *RWblock. DataKey*)
not in *KeySet*
/*将该键添加至访问键集 *KeySet*/
③ end if
④ if (*RWblock. AccessMode* == ‘r’)
⑤ 根据当前键链指向赋值读/写块 *Key_next* 和 *Key_prev*;
⑥ 键链插入块，遵循“读集生成的读块在前”原则;
⑦ else
⑧ 获取访问键指向 *keylink*;
⑨ 遍历找到 *keylink* 尾部节点;
⑩ *RWblock. Key_next* = null;
⑪ *RWblock. Key_prev* = *Keylink*;
⑫ *Keylink. next* = *RWblock*;
⑬ *link. next* = *RWblock*;
/*事务链插入块*/
⑭ *RWblock. Chain_prev=link*;
⑮ *link* = *link. next*;
⑯ *RWblock. Chain_next* = null;
⑰ end if
⑱ return *KeySet*, $T_{\text{c-plus}}$.

算法 2 对应算法 1 中函数 *AddBlock*，主要涉及读/写块加入键链和事务链的过程. 为优化此计算过程，在算法 2 中提出了“读集生成的读块在前，写集生成的写块在后”的块插入规则. 在读集生成块的基础上，访问同一键链上写集生成块时面向同一搜索方向提高了计算效率.

为了便于理解后续调度机制，我们给出一个示例. 该示例包含 5 个事务以及每个事务的读集合与写集合. 如表 1、表 2 所示. 示例集合$T_{\text{all}}=\{T_1,T_2,T_3,T_4,T_5\}$，表中值为 1 表示该事务包含对此键的读操作或者写操作.

总事务集T_{all}划分为冲突事务集$T_{\text{c}}=\{T_1,T_2,T_3,T_4\}$与非冲突事务集$T_{\text{non-c}}=\{T_5\}$.$T_{\text{non-c}}$事务集前置且不参

Table 1 Read-Set of Inter-Block Conflicting Transactions

表 1 块内冲突事务的读集合

事务	K_1	K_2	K_3	K_4	K_5
T_1	1	0	0	1	0
T_2	0	1	0	0	0
T_3	0	0	1	0	0
T_4	0	0	0	0	0
T_5	0	0	0	0	1

Table 2 Write-Set of Inter-Block Conflicting Transactions

表 2 块内冲突事务的写集合

事务	K_1	K_2	K_3	K_4	K_5
T_1	0	0	0	0	0
T_2	1	0	0	0	0
T_3	0	1	0	0	0
T_4	0	0	1	1	0
T_5	0	0	0	0	0

与后续调度机制. $T_{\text{non-c}}$事务集内事务顺序仍遵循 Kafka 共识协议生成的逻辑顺序. 根据依赖关系链构造算法，T_{c}事务集输入后得到图 4 所示的依赖关系链结构.

需要说明的是，依赖关系链的相关算法在排序节点中运行. 以排序节点共识后的事务集为原型数据构造依赖关系链，不仅兼容了原有的共识算法，而且也为后续冲突事务危险结构检测提供数据支撑.

4.1.3 危险结构检测和处理算法

受串行安全网[35]思想启发，我们提出危险结构检测算法，检测冲突集T_{c}中构成了依赖循环需要中止的事务，在中止部分事务后经事务调度，可对块内非冲突事务合理排序.

在介绍检查算法前，先引出事务间依赖关系、2 个定义和 1 个定理作为预备知识.

定义 1. 前驱事务与后驱事务. 假设事务集 T_{c} 中存在依赖关系，形如 $T_i \xrightarrow{x:x} T_j \xrightarrow{x:x} T_k$（$x$ 取 w 或 r）. 称 T_i 是 T_k 的前驱事务，表示为：$T_i = pre\times(T_k)$；T_k 是 T_i 的后驱事务，表示为：$T_k = suc\times(T_i)$. 且在该示例中，T_i 是 T_j 的直接前驱事务，表示为：$T_i = pre\times(T_j)$. 当事务集构成循环依赖时，形如 $T_i \xrightarrow{x:x} T_j \xrightarrow{x:x} T_i$，称 T_j 既是 T_i 的前驱事务又是 T_i 的后驱事务.

定义 2. 危险结构. 危险结构是指根据事务的依赖关系判断该事务是否存在成环风险的一种结构.

我们定义了 3 个时间戳，用于检测当前事务是否构成危险结构. 时间戳的定义结合了依赖关系链中读/写块间的关联性.

1）事务 T 的模拟提交时间戳 $c(T)$. 由于算法检测在排序阶段执行，无法获取实际的事务提交时间戳. 因此 Fabric-HT 在依赖关系链的事务块中定义了 *TimeStamp* 模拟提交时间戳，该时间戳作用于区分各个事务的提交顺序以及中止事务的判断，而不作为实际提交时间. *TimeStamp* 在事务块初始化时赋值.

2）事务 T 的已提交最小前驱事务 T_{P} 的模拟提交时间戳 $\pi(T)$. 已提交最小前驱意味着事务 T_{P} 需满足式（1）~（3）：

$$T_{\mathrm{P}} = pre\times(T); \tag{1}$$

$$T_{\mathrm{P}}.TimeStamp < T.TimeStamp; \tag{2}$$

$$T_{\mathrm{P}}.TimeStamp = \min(c(T), pre\times(T).TimeStamp). \tag{3}$$

3)事务 T 的直接后驱事务 T_{S} 的模拟提交时间戳 $\rho(T)$. 需要注意的是，与 $c(T)$ 和 $\pi(T)$ 不同，事务 T 可能存在多个直接后驱事务从而产生多个 $\rho(T)$. 因此在实际的计算中 $\rho(T)$ 优先取小于 $c(T)$ 的最大值(见算法 3). 而已提交直接后驱事务 T_S 需满足式(4)~(6)：

$$T_{\mathrm{S}} = suc(T); \tag{4}$$

$$T_{\mathrm{S}}.TimeStamp < T.TimeStamp; \tag{5}$$

$$T_{\mathrm{S}}.TimeStamp = \max(suc\times(T).TimeStamp). \tag{6}$$

定理 1. 在事务间依赖关系中，当事务 T 及其已提交最小前驱事务 T_{P} 确定时，即 $c(P)=\pi(T)<c(T)$. 如果存在事务 T 的直接后驱事务 T_S 使事务 T 满足 $\pi(T)\leqslant\rho(T)<c(T)$ 时，则事务 T 的依赖关系构成危险结构. 其中 $\pi(T)=c(P)$，$\rho(T)=c(S)$. 即若事务集中存在循环依赖关系，则必然存在危险结构.

证明. 假设环内所有事务 T_i 都不满足条件 $\pi(T_i)\leqslant\rho(T_i)<c(T_i)$，即危险结构不存在. 那么环内事务 T_i 必然满足以下 2 种条件之一(需要注意的是 $c(T)\neq\rho(T)$，因为不存在同时提交的 2 个事务).

1) $\rho(T_i)<\pi(T_i)<c(T_i)$，即存在事务 T_i 的直接后驱事务 T_j 在事务 T_i 之前提交，且提交时间戳小于事务 T_i 的已提交最小前驱事务 T_k. 然而根据 $\pi(T)$ 定义得到：$T_k.TimeStamp=\min(c(T_i), pre\times(T_i).TimeStamp)$，且当事务间构成循环时，根据定义 3，满足 $T_j=pre\times(T_i)$，即 $T_j.TimeStamp\geqslant\min(pre\times(T_i).TimeStamp)$. 所以 $\rho(T_i)\geqslant c(T_i)$，与前置条件相悖.

2) $\pi(T_i)<c(T_i)<\rho(T_i)$，即存在事务 T_i 的直接后驱事务 T_j 在事务 T_i 之后提交，即 $c(T_i)<c\left(T_j\right)$. 假设被检测事务 T_i 为当前循环内最晚提交事务，则不满足条件 $c(T_i)<c\left(T_j\right)$，与前置条件相悖.

因此若事务集 T_{c} 中存在循环依赖关系，则必然存在事务 T_i 满足危险结构检测. 证毕.

算法 3. 危险结构检测和处理算法.

输入：访问键集 *KeySet*〈Map of K_i, *Head*〉，事务集 $T_{\text{c-plus}}$；

输出：非冲突事务集 T_{f}.

① *SerialDetectionAlgorithm*(*KeySet*, $T_{\text{c-plus}}$)；

② *Init*(*G*)；

/*初始化无环依赖图，初始化边和节点*/

③ for T_i in $T_{\text{c-plus}}$

④ $c(T_i)=T_i.TimeStamp$；

$\rho(T_i)=-\infty$；

$\pi(T_i)=+\infty$；

$link=T_i.Head$；

/*初始化参数*/

⑤ 纵向遍历单个事务链 *link*；

⑥ if (*link.AccessMode* == 'r')

⑦ 向后遍历键链寻找写集依赖；

⑧ else if (写读依赖)

更新 $\pi(T_i)$；

在 *G* 中添加依赖边；

⑨ else if (读写反依赖)

更新 $\rho(T_i)$，在 *G* 中添加依赖边；

⑩ else if (写写依赖)

按照 First Commiter Wins 规则；

⑪ else

⑫ 向前遍历键链寻找读集依赖；

⑬ if (写读依赖)

更新 $\rho(T_i)$；

在 *G* 中添加依赖边；

⑭ else if (读写反依赖)

更新 $\pi(T_i)$；

在 *G* 中添加依赖边；

⑮ else if (写写依赖)

按照 First Commiter Wins 规则；

⑯ end if

⑰ end if

⑱ end for

⑲ if ($\pi(T_i)>\rho(T_i)$) /*危险结构测试*/

中止 T_i；/*中止事务*/

删除在 *G* 中的 T_i；

Pruning($T_{\text{c-plus}}$)；/*剪枝优化*/

end if

⑳ T_{f} = *Topical* (*G*)；/*事务重排序*/

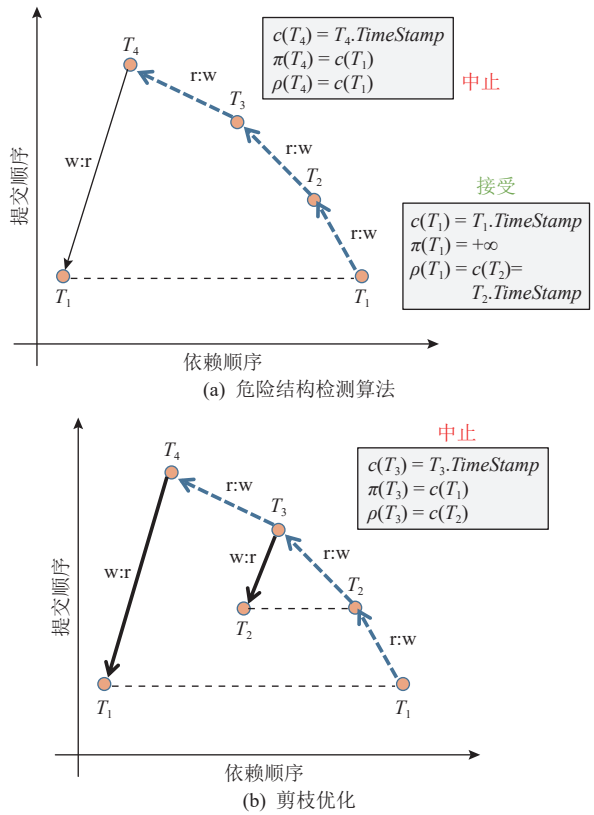
㉑ return T_{f}.

如算法 3 所示，我们在依赖关系链的基础上提出了 EOV 架构下的危险结构检测算法. 从事务集中初始事务出发，依次判断当前事务是否满足危险结构条件. 若满足，则中止该事务且删除无环依赖图中该事务的节点和依赖边. 同时为了优化计算规模以及降低事务中止率，对依赖关系链进行剪枝，删除该事务在键链和事务链上的节点.

在算法 3 中：1)当检测当前读/写块为读集生成

的块时，依据键链后指向 *Key_next* 寻找写集生成的块. 根据当前读/写集的模拟时间戳判断 2 个事务的依赖关系，若为写读依赖，则更新$\pi(T_i)$；若为读写反依赖，则更新$\rho(T_i)$. 如算法 3 中行⑦~⑫所示. 2）当检测当前读/写块为写集生成的块时，依据键链前指向 *Key_prev* 寻找读集生成的块. 根据当前读/写集的模拟时间戳判断 2 个事务的依赖关系，若为读写反依赖，则更新$\pi(T_i)$；若为写读依赖，则更新$\rho(T_i)$，如算法 3 中行⑬~⑭所示. 最终算法执行结果集返回无环依赖图 G 用于事务可串行化处理.

为了便于理解定理 1 在算法 3 中的应用，给出如下示例说明. 以表 1~2 为例，冲突事务集$T_c=\{T_1,T_2,T_3,T_4\}$根据危险结构检测算法依次执行. 根据依赖关系链中事务读/写块可得到图 5(a)中的事务间依赖关系. 首先计算事务T_1，$c(T_1)=T_1.TimeStamp$. 因找不到事务T_1的已提交最小前驱事务 P，$\pi(T_1)$初始值不变. 事务T_1的直接后驱事务为T_2，因此$\rho(T_1)=c(T_2)$. 综上$\rho(T_1)<\pi(T_1)$不满足危险结构条件. 同理分别计算事务T_2,T_3,T_4. 当计算事务T_4时，$c(T_4)=T_4.TimeStamp$. 事务T_1的已提交最小前驱事务为T_1，因此$\pi(T_4)=c(T_1)$. 事务T_1的直接后驱事务为T_1，因此$\rho(T_4)=c(T_1)$. 综上$\pi(T_4)=\rho(T_4)<c(T_4)$满足危险结构条件，事务$T_4$被中止，其他事务不再构成依赖循环关系.



(a) 危险结构检测算法

(b) 剪枝优化

Fig. 5 Example of dangerous structure detection algorithm

图 5 危险结构检测算法示例

1）剪枝优化. 在危险结构检测算法中，当事务因满足危险结构检测而被中止时，该事务的依赖关系将不具备参考性. 因为事务中止，该事务与其他事务的冲突问题也将中止. 以图 5(b)为例，当事务T_3被中止时，事务T_4已不构成循环依赖关系. 假设在事务T_4检测前保留T_3信息，则$\pi(T_4)=c(T_1)=\rho(T_4)<c(T_4)$满足危险结构条件，造成错误中止事务. 因此在事务检测过程中的剪枝优化一方面可以避免错误中止事务，另一方面降低了事务计算规模. 剪枝优化算法在依赖关系链上执行，当该事务满足危险结构条件时，事务链依据 *Tx_prev* 和 *Tx_next* 前后指向完成剪枝操作，键链依据 *Key_prev* 和 *Key_next* 前后指向完成剪枝操作.

2）事务重排序. 区块内事务调度机制执行危险结构检测算法对构成循环依赖的事务提前中止，剩余事务构成有向无环依赖图. 有向无环依赖图的构造确保了后续可以通过事务重排序来解决事务间冲突问题. Fabric-HT 引入拓扑排序[36]来建立冲突事务间的非冲突序列.

4.1.4 算法复杂度分析

区块内事务调度机制由 2 部分组成：构造依赖关系链和执行危险结构检测算法. 假设区块内有 n 个事务，每个事务有 m 个读写集（一般性假设）. 则这 2 部分的空间复杂度均为 $O(n\times m)$. 在时间复杂度方面，以事务为基本单位时，构造依赖关系链和危险结构检测算法的时间复杂度都为 $O(n)$；以读/写块为基本单位时，构造依赖关系链的时间复杂度为 $O(m)$，而危险结构检测算法的时间复杂度为 $O(n\times m^2)$. 随着冲突事务的增多，该算法的耗时不会出现显著性变化. 这也取决于区块链系统本身的特殊逻辑，在一定时间内只允许固定数量的事务或固定大小的区块执行上链操作. 因此参与调度机制的事务始终在一个可控的范围之内.

4.2 块间冲突事务避免机制

如第 3 节所述，由于事务的模拟执行与验证提交之间存在较大延迟引发了区块间冲突，此类冲突问题无法通过事务重排序解决，因此我们提出了以“推送-匹配”为核心的区块间事务避免机制. 在排序节点中引入缓存区，可以在区块生成前对事务提前进行冲突检测，并根据冲突检测结果决定该事务是否加入区块. 整个冲突事务避免机制分 3 个阶段：上链数据“推送”缓存区、冲突事务匹配检测和冲突事务重新提交.

4.2.1 方案设计

在排序节点设置缓存区，缓存区的作用为检测当前排序节点中是否包含冲突事务，对冲突事务执行提前事务中止，严格限制冲突事务参与后续区块的生成. 缓存区的内容与相连主节点内的状态数据库内容保持同步. 为了减少空间消耗，缓存区内仅保留键和版本用作冲突检测，缓存区采用 key-value 数据库.

4.2.1.1 上链数据推送缓存区

当区块验证通过在主节点数据库中更新时，主节点将推送最新数据版本至缓存区用于块间事务冲突检测. 缓存区将被动更新数据.

自旋检测算法为在联盟链的去中心化和不可信的环境中单一主节点的数据推送存在可信问题时，通过排序服务节点对推送数据进行核验. 算法具体流程为：当排序服务节点接收到单一主节点推送的上链数据后，将进入自旋状态，持续向其他主节点发起上链数据推送请求；在请求阈值内，若排序服务节点接受到其他主节点的请求数据，则进行多节点数据一致性匹配，若匹配通过，执行后续冲突事务检测流程；若不通过或阈值内未能收到其他主节点请求，则跳过此次冲突事务避免机制.

4.2.1.2 冲突事务匹配检测

假设当前待检测队列中存在某个事务的读集合〈K_1,*VersionA*〉和缓存区内数据〈K_1,*VersionB*〉，如果 *VersionA*<*VersionB*，则该事务为无效事务. 在排序节点接收到客户端发送的事务后，该事务加入检测队列中等候冲突检测，无法通过冲突检测的事务被标记为无效事务，等候重新提交.

区块间冲突的事务可能来自不同客户端. 考虑到多客户端引发并发冲突的可能. 我们将缓存区部署在排序服务而不是客户端. 整个冲突检测流程分 4 步完成，对应图 6 所示.

1)排序节点接收客户端发送的事务提案.

2)对收到的事务执行冲突事务检测，缓存区更新当前缓存内容，检测当前事务是否存在过时读集. 检测方式按照无效事务检测规则.

3)无效事务执行提前中止.

4)未发生冲突事务参与区块的生成，并被排序节点发送给组织内负责接收的主节点.

4.2.1.3 冲突事务重新提交

为了不破坏事务的原子性，对无法通过冲突检测的无效事务执行事务提前中止，并由排序节点通知客户端重新发起. 重新发起的事务语义不变，重新进入执行—排序—验证流程. 区块间冲突避免机制可以很好地检测靠后区块内的冲突事务，实施提前事务中止并通知客户端重新发起.

4.2.2 举例说明

为了便于理解区块间事务避免机制，此处给出一个示例说明. 假设当前区块链系统中有 2 个邻接区块 10001 和 10002，所含冲突事务信息如表 3 所示.

由于事务模拟阶段与验证阶段间的延迟，区块 10002 中事务 T_2 在区块 10001 中事务 T_1 提交之前模拟运行，所以 K_1 的读集合为旧版本 *Version*1. 区块 10001 和 10002 按照生成顺序依次进入区块间冲突检测流程. 前置区块的上链结果更新状态数据库且影响后置区块的检测结果，具体执行步骤为：

1)T_1 进入排序服务阶段.

2)缓存区刷新，请求最新键版本数据：K_1 最新版

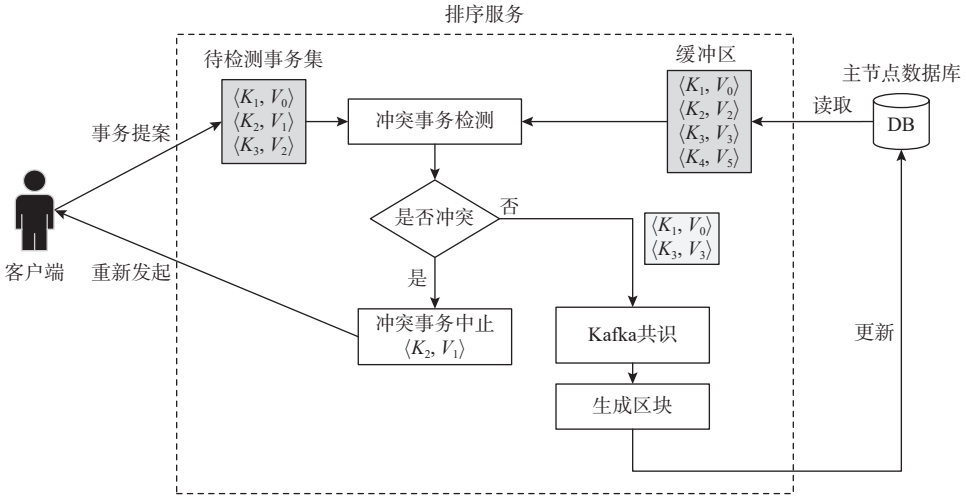


Fig. 6 Inter-block conflict transaction processing flow

图 6 区块间冲突事务处理流程

Table 3 Example of Inter-Block Conflict Transaction

表 3 区块间冲突事务示例

区块	所含事务	读/写集合	执行结果
10001	T_1	〈K_1, *ValueA*〉 〈K_1, *Version*1〉	〈K_1, *Version*2〉
10002	T_2	〈K_1, *Version*1〉	冲突

本为 *Version*1.

3) T_1 通过事务检测机制，K_1 版本更新至 *Version*2，生成区块 10001.

4) 区块 10001 经组织记账节点验证通过，执行上链操作.

5) T_2 进入排序服务阶段.

6) 缓存区刷新，请求最新键版本数据：K_1 最新版本为 *Version*2.

7) T_2 符合无效事务定义，无法通过事务检测，执行提前事务中止.

8) 通知客户端重新发起 T_2.

4.2.3 算法时间复杂度

整个冲突事务避免机制的运行时间复杂度为 $O(n)$，空间度为 $O(n)$，n 为客户端提交事务的数量.

5 实验及分析

为验证 Fabric-HT 方案的有效性，开展实验并与基准 Fabric、最新的 Fabric++[13] 和 FabricSharp[18] 等方案进行对比，围绕关键指标，在不同场景下开展实验评估工作.

5.1 实验设置及环境

本文实验场景包含 3 个组织，每个组织包含 3 个对等节点，节点通过 Docker 容器部署. 采用 Kafka 共识机制，并使用 LevelDB 作为状态数据库. 实验环境共由 11 台服务器组成，每台服务器含 4 核 CPU（Intel Xeon 2.2 GHz），12 GB RAM，均安装有 Fabric V1.4.8，并运行在 Ubuntu 16.04 LTS 环境下. Fabric-HT 其他相关实验参数如表 4 所示.

Table 4 Experimental Parameters Settings

表 4 实验参数设置

实验参数	取值
通道数量	1
每个通道的客户端数量	4
每秒每个客户端触发的事务提案数	512
事务提案完成的最长持续时间/s	90
形成一个区块的最长时间/s	1
每个区块的最大容量/MB	2

实验在 Smallbank 工作负载和自定义工作负载情况下进行研究. 这 2 种负载的测试在 Caliper 和 BlockBench[23] 测试框架下展开.

在 Smallbank 工作负载下，1 个交易可从 1 万个账号中读写 4 个账号，调整热门账号（表示并发环境下，该账号同时存在读操作和写操作）的比例以确保每次操作都有一定概率访问到热门账号. 此工作负载是开源平台 OLTP 的代表性工作负载之一，广泛应用于 Fabric 系统的性能测试.

自定义工作负载侧重于调整读写比例观察各方案的优势场景.在自定义工作负载下，如表 5 所示，设置 10 000 个账号用于读写操作，事务包含的读写操作规模介于 4 和 8 之间. 选择读取热门账号的概率 *HR*=10%, 20%, 40%; 写入访问账号概率 *HW*=5%, 10%; 热门账号比例 *HSS*=1%, 2%, 3% 时工作负载探究各方案的性能表现.

Table 5 Workload Settings

表 5 工作负载设置

实验参数	取值
账号数量	10 000
每个事务的读写操作量	4，8
选择读热门账号的概率 *HR*/%	10，20，40
选择写热门账号的概率 *HW*/%	5，10
热门账号比例 *HSS*/%	1，2，3

为便于阐述和实验对比，本文所提优化方案为：

1) Fabric-HT-Ⅰ. 基于区块内冲突的事务调度机制在 Fabric 系统中独立运行.

2) Fabric-HT-Ⅱ. 基于区块间冲突的事务避免机制在 Fabric 系统中独立运行.

3) Fabric-HT-Ⅰ+Ⅱ. 上述 2 种优化方案在 Fabric 系统中并行运行.

其中 Fabric-HT-Ⅰ和 Fabric-HT-Ⅱ方案均属独立优化方案，在排序节点可单独运行或同时运行.

5.2 评价指标

为了对比不同工作负载下区块链系统的性能，我们引入了 4 个评价指标：

1) 每秒平均成功事务数（average number of successful transactions per second, *TPS*）. *TPS* 用于衡量区块链系统的吞吐量，表示一段时间内组织内记账节点每秒平均上链的事务数量[37]，是衡量区块链系统性能的重要指标. 具体计算方法为

$$TPS = \frac{1}{L}\sum_{i=1}^{n}\sum_{j=1}^{m_n} T_{ij}, \tag{7}$$

其中 L 表示衡量周期, 由于每秒上链成功事务数的高度集中性, 即同一区块内所有事务同时上链, 所以统计每秒上链事务数不具有代表性, 选取一段时间内平均成功上链事务数可以有效衡量该指标. n 表示 L 时间内上链的区块数, m_n 表示第 n 个区块内的事务数, T_{ij} 衡量第 i 个区块中第 j 个事务是否成功, 如成功, T_{ij}=1, 不成功, T_{ij}=0.

2)平均事务执行时间(average transaction execution time, *TET*). *TET* 表示事务从客户端发起, 经过背书、排序, 直至最后上链所经历的平均延迟, 反映区块链系统处理事务的效率. 具体计算方法为

$$TET = \frac{1}{n}\sum_{i=1}^{n} T_i, \tag{8}$$

其中, n 为事务的数量, T_i 为第 i 个事务从发起到上链所经历的延迟.

3)事务中止率(transaction abort rate, *TAR*). *TAR* 表示一段时间内被中止的冲突事务数与全部事务数的比值, 反映了区块链系统对冲突事务的处理能力. 事务中止率越低, 系统处理冲突的能力越高. 具体计算方法为

$$TAR = \frac{C}{\sum_{i=1}^{n} Total_i + C}, \tag{9}$$

其中, C 为某时间内被中止的事务总数, n 为某时间内生成的区块数, $Total_i$ 为第 i 个区块内事务总数.

4)无效事务空间占用率(invalid transaction space occupancy rate, *ITS*). *ITS* 表示在区块链系统中所有无效事务占用的存储空间与所有区块占用的存储空间的比值. 本指标反映了各方案节约存储空间的能力. *ITS* 值越低, 表明方案在节约空间方面的能力越高. 具体计算方法为

$$ITS = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m_n} Ts_i^j}{\sum_{i=1}^{n} Block_i}, \tag{10}$$

其中, Ts_i^j 表示第 i 个区块中第 j 个事务的大小, $Block_i$ 表示第 i 个区块的大小, n 表示测试区块的个数, m_n 表示第 n 个区块中无效事务的个数.

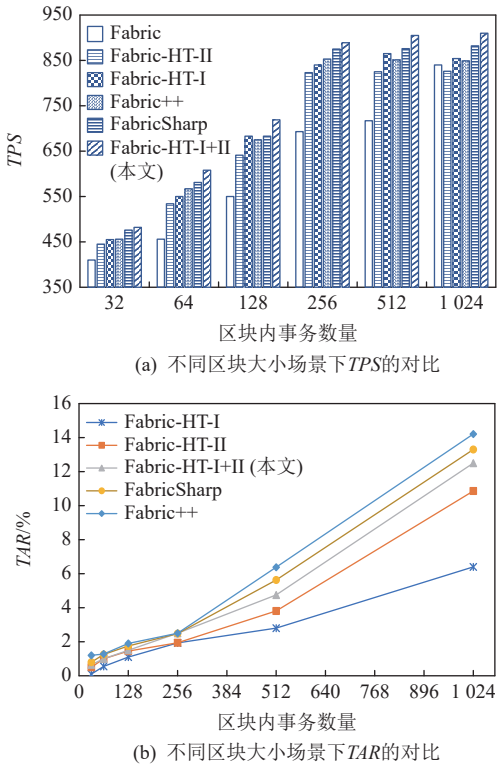
5.3 实验结果分析

围绕 *TPS*、*TET*、*TAR*、*ITS* 等指标, 研究各对比方案在区块大小、冲突事务占比、通道数量和客户端数量、区块数量等不同参数取值下的性能对比情况. 4 个指标和 4 种变化参数设计总共 16 种情况. 由于部分实验结果接近, 为使内容简洁我们挑选并列出代表性结果.

5.3.1 区块大小的影响

为评估区块大小对系统性能的影响, 以 2 的整数倍为步长将区块容纳事务数从 32 提升至 2 048[23], 并观察成功上链事务数以及无效事务的比例. 使用 Smallbank 工作负载进行测试, 并初始化 100 000 个用户, 在满足写事务比例为 95%、冲突事务比例为 20% 的条件下编写事务用例.

图 7(a)显示了运行期间的 *TPS*. 当区块内事务数量从 32 增至 1 024 时, 各方案的 *TPS* 都呈现明显增长. 首先, Fabric-HT-I+II 区块大小提升最为明显. 实验观察到在事务数量达到 1 024 时, *TPS* 不再出现显著增长, 区块达到饱和状态. 其次, Fabric-HT-I, Fabric++, FabricSharp 的优化效果也较为明显, 单独的区块内的事务优化方案 Fabric-HT-I 的处理能力较弱, 与 Fabric++ 接近. 再次, Fabric-HT-II 略优于 Fabric, 最高 *TPS* 可达 825. 最后, Fabric 由于未考虑冲突事务处理, 在区块大小为 1 024 笔事务时 *TPS* 仅为 738, 为最低.



(a) 不同区块大小场景下*TPS*的对比

(b) 不同区块大小场景下*TAR*的对比

Fig. 7 Impact of block sizes on *TPS* and *TAR*

图 7 区块大小对 *TPS* 和 *TAR* 的影响

图 7(b)展示了在区块内事务数量递增的情况下, *TAR* 的变化情况. 如图所示, 当区块内并发事务指数增长时, 各优化方案 *TAR* 呈增长趋势. 相比于 Fabric++

和 FabricSharp, Fabric-HT-I+II 具有更好的处理性能，*TAR* 较 Fabric++最高可降低 20%, 较 FabricSharp 最高可降低 14%. 此外，由于 Fabric 不涉及冲突事务处理机制，因此此处未列出 *TAR*.

5.3.2 冲突事务比重（*RW*）的影响

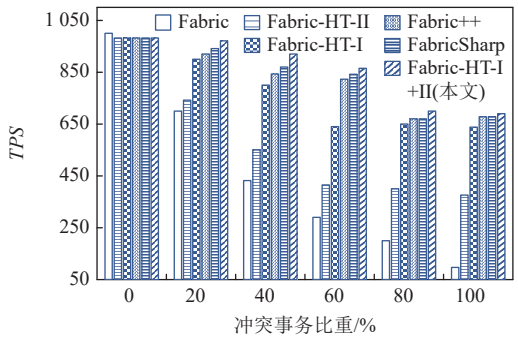
在冲突事务比例变化时，考察各对比方案成功事务吞吐量和平均延迟情况. 实验用例设置读事务比例为 50%，其他参数设置如表 6 所示. 测试事务用例满足 Zipf 分布. 工作负载采用 Smallbank，其中 Zipf 分布的倾斜参数从 0 增长至 2.0，以步长 0.4 设置不同程度的事务冲突，小倾斜参数意味着低事务冲突比重.

图 8 展示在各个冲突事务比重下，*TPS* 以及 *TET*

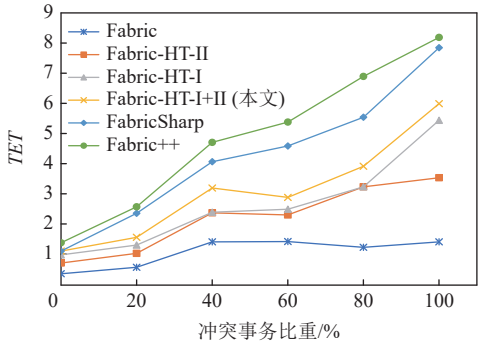
Table 6 Zipf Distribution Skew Parameter Corresponds to the Proportion of Conflict Transaction

表 6 Zipf 分布倾斜参数对应冲突事务比重

Zipf 分布倾斜参数 α	冲突事务比重/%
0	0
0.4	20
0.8	40
1.2	60
1.6	80
2.0	100



(a) 不同冲突事务比重场景下*TPS*的对比



(b) 不同冲突事务比重场景下*TET*的对比

Fig. 8 Impact of proportion of conflict transactions on *TPS* and *TET*

图 8 冲突事务比重对 *TPS* 和 *TET* 的影响

的变化趋势. 如图 7(a)所示，各方案 *TPS* 均随冲突事务比重的增加而减少. 其中 Fabric 的 *TPS* 下降尤为明显. 其次，Fabric-HT-Ⅱ虽然相对于 Fabric 有所提升，但较其他优化方案仍有差距. Fabric-HT-Ⅰ+Ⅱ在高并发冲突环境下仍能保持较高 *TPS*. 当冲突事务比重在 0%~100% 间变化时，Fabric-HT-Ⅰ+Ⅱ将 Fabric 的 *TPS* 提高了 1.23~9.51 倍，将 Fabric++和 FabricSharp 的 *TPS* 分别平均提高了 1.33 倍和 1.18 倍.

图 8(b)展示了冲突事务比重变化时事务的 *TET*. 与 Fabric 相比，Fabric++, FabricSharp, Fabric-HT-Ⅰ, Fabric-HT-Ⅱ, Fabric-HT-Ⅰ+Ⅱ都表现出了更高的延迟，这是由于它们需要额外的处理时间来解决冲突事务，且随着冲突事务比例的增加，*TET* 也在显著提高. 而 Fabric 本身没有相关处理机制，所以 *TET* 并未在图中给出.

实验观察到，在冲突事务比重由 40% 提升到 60% 时，*TET* 降低，表现为更优的效果，更多的冲突事务被中止在排序服务节点，不再参与后续区块的生成以及上链过程，因此 *TET* 出现抖动，但该抖动不具备通用参考性. 实验仍旧以对比相同事务集下不同优化方案的优劣作为重要指标.

值得注意的是，在冲突事务比重达到 40% 后，Fabric-HT-Ⅰ+Ⅱ比 Fabric++和 FabricSharp 具备更为明显的低延迟表现，更适应于处理高并发环境下冲突事务密集度较高的场景. Fabric-HT-Ⅰ+Ⅱ能够获得该收益的原因取决于其可以大规模处理并发事务，对区块内事务规模不作限制，提高了事务处理效率.

研究高冲突场景下不同读写操作比重下的系统优化情况. 如图 9 所示，展示了事务读写操作比重

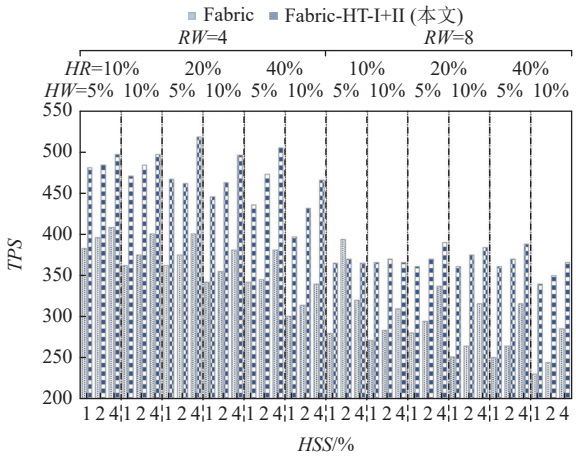


Fig. 9 Comparison of Fabric and Fabric-HT-I+II under 36 different configurations for *TPS*

图 9 36 种不同配置 Fabric 和 Fabric-HT-I+II 的 *TPS* 对比

(*HR*, *HW*)以及冲突事务比重(*RW*)改变时, Fabric 和 Fabric-HT-Ⅰ+Ⅱ的测试结果. Fabric-HT-Ⅰ+Ⅱ显著提高了系统 *TPS*, 尤其在 *RW*=8, *HR*=40%, *HW*=10%, *HSS*=1% 的配置下, *TPS* 高达 Fabric 的 2.4 倍.

5.3.3 通道数量和客户端数量的影响

此前实验采用单个通道和 4 个客户端提交事务. 为研究 Fabric-HT 的鲁棒性[18], 我们调整通道数量和客户端数量, 观察它们对 *TPS* 的影响.

首先, 实验调整通道数量观察对 *TPS* 的影响. 如图 10 所示, 通道数量从 1 增加到 8. 为节省计算资源, 每个通道使用 2 个客户端发起交易提案. 当通道数量在 1~4 之间增长时, 各方案 *TPS* 均在增长. Fabric-HT-Ⅰ, Fabric-HT-Ⅱ, Fabric-HT-Ⅰ+Ⅱ在多链环境下表现出良好的可扩展性和健壮性. 当通道数量达到 8 后, 由于通道间竞争资源, 导致失败事务的数量增加, 进而引起 *TPS* 下降. 然而, Fabric-HT-Ⅰ+Ⅱ仍具有最好性能.

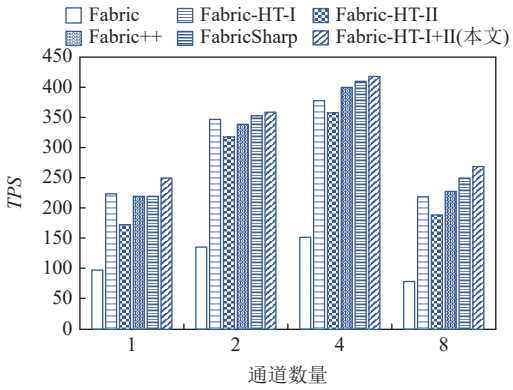


Fig. 10 Impact of channel numbers on *TPS*

图 10 通道数量对 *TPS* 的影响

实验同时研究单通道中客户端数量的变化, 如客户端数量从 1 增加到 8 对 *TPS* 的影响. 图 11 展示了当单通道客户端数量处于 1~4 时, Fabric-HT-Ⅰ+Ⅱ的 *TPS* 随客户端数量增加而提高. 当单通道客户端数量增长到 8 时, *TPS* 明显下降, 这是由于客户端数量增加引起资源竞争加剧, 造成无效事务增加所致. 此时 Fabric-HT-Ⅰ+Ⅱ仍具有最好的性能.

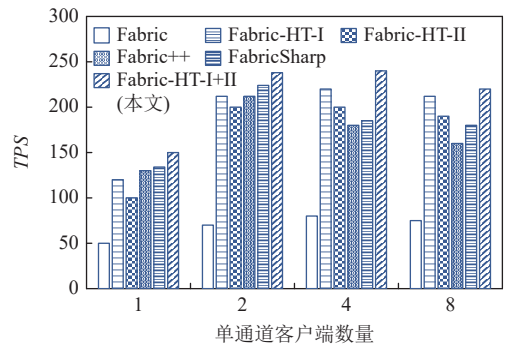


Fig. 11 Impact of client numbers on *TPS* in single channel

图 11 单通道客户端数量对 *TPS* 的影响

5.3.4 区块数量的影响

随着区块数量的不断增加, 对区块链节点而言链上数据所需存储空间也在增大.

图 12 展示了随着区块链系统中区块数量的增加 *ITS* 的变化. 从图中可以看出, 区块数量从 10 增长到 35 时, Fabric 无效事务造成的空间损耗较为突出, 其余方案均有改进, 且 Fabric-HT-Ⅰ+Ⅱ表现出最优的性能, 能有效节省存储空间. 这源于我们在区块内冲突中通过事务重排序降低无效事务的数量, 以及通过缓存区冲突检测实现对区块间无效事务的提前中止.

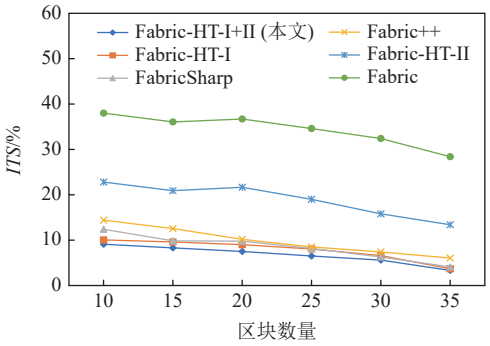


Fig. 12 The impact of block numbers on *ITS*

图 12 区块数量对 *ITS* 的影响

6 讨 论

Fabric-HT 除通过有效解决并发冲突提高 *TPS* 外, 同时具有 3 点优势:

1)节省存储空间. 并发冲突事务的解决, 极大地提高了上链事务的成功率, 且有效事务比例的提高减少了链上存储资源的浪费. 当基准区块链系统 Fabric 链上的无效事务占比维持在 40% 以下时, Fabric-HT 链上无效事务占比维持在 15% 以下, 极大地节省了存储空间.

2)有效防范攻击. 当系统内某时刻处于不同位置的恶意节点针对某一个账户或节点发动 DDoS 攻击[28-29]时, 服务器中充斥着大量同类信息, 消耗网络带宽或系统资源, 可能导致系统瘫痪[30]. 冲突事务检测机制可以很好地防范此类攻击, 在排序服务阶段对事务语义执行提前检查, 对检测不通过的事务执行提前中止, 使其无法进入后续流程影响上链操作. Fabric-HT 将此类问题在构建区块前解决, 一定程度

上提高了区块链系统的安全性.

3)检测速度快. 现有方案往往将冲突图分解为多个循环, 贪婪地中止环内的冲突事务. 该方法虽然可以解决大多数场景下的冲突问题但检测冲突事务时间较长, 严重限制了并发事务规模. 本文的危险结构检测算法确定了以时间戳为基础、以依赖关系链为阶梯的危险结构检测方式, 在以事务为单位的线性时间内判断冲突事务是否应该中止.

7 结　　论

针对 Fabric 并发冲突影响性能的问题, 本文提出一种综合考虑块内冲突和块间冲突的区块链性能优化方案 Fabric-HT. 针对区块内事务冲突, 提出了一种事务调度算法, 根据区块内冲突事务集构造依赖关系链, 并以依赖关系链为基础提出了危险结构检测算法. 针对区块间冲突问题, 由于区块间事务的延迟性, 通过在排序服务部分构造缓存区, 建立冲突事务检测机制. 实验结果表明, Fabric-HT 在吞吐量、事务中止率、事务平均执行时间、无效事务空间占用率等方面, 均优于对比方案. 此外, Fabric-HT 也表现出较好的鲁棒性和抗攻击能力.

作者贡献声明: 吴海博和刘辉完成相关文献资料调研和分析以及论文撰写; 孙毅参与相关方案的交流和指导; 李俊指导论文撰写.

参　考　文　献

[1] Cai Xiaoqing, Deng Yao, Zhang Liang, et al. The principle and core technology of blockchain[J]. Chinese Journal of Computers, 2021, 44(1): 84−131 (in Chinese)
(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)

[2] Urquhart A. The inefficiency of bitcoin[J]. Economics Letters, 2016, 148: 80−82

[3] Wood G. Ethereum: A secure decentralised generalised transaction ledger[J/OL]. 2014[2023-09-09]. http://explore-ip.com/2017_Comparison-of-Ethereum-Hyperledger-Corda.pdf

[4] The Linux Foundation. Hyperledger Fabric[EB/OL]. 2018[2023-09-09].https://github.com/hyperledger/Fabric

[5] Cachin C. Architecture of the hyperledger blockchain Fabric[C/OL]// Proc of Workshop on Distributed Cryptocurrencies and Consensus Ledgers (DCCL). 2016[2023-09-09].https://www.zurich.ibm.com/dccl/

[6] Androulaki E, Barger A, Bortnikov V, et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains[C/OL]// Proc of the 13th EuroSys Conf (EuroSys). New York: ACM, 2018[2023-09-09].https://dl.acm.org/doi/10.1145/3190508.3190538

[7] Brandenburger M, Cachin C, Kapitza R, et al. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric[J]. arXiv preprint, arXiv: 1805.08541, 2018

[8] Jiang Lili, Chang Xiaolin, Liu Yuhang, et al. Performance analysis of Hyperledger Fabric platform: A hierarchical model approach[J]. Peer-to-Peer Networking and Applications, 2020, 13(3): 1014−1025

[9] Valenta M, Sandner P. Comparison of Ethereum, Hyperledger Fabric and Corda[EB/OL]. Frankfurt School Blockchain Center, 2017[2023-09-09]. http://explore-ip.com/2017_Comparison-of-Ethereum-Hyperledger-Corda.pdf

[10] Nasir Q, Qasse I A, Talib M A, et al. Performance analysis of hyperledger fabric platforms[J]. Security and Communication Networks, 2018, 2018: 1−14

[11] Xu Xiaoqiong, Sun Gang, Luo Long, et al. Latency performance modeling and analysis for Hyperledger Fabric blockchain network[J]. Information Processing & Management, 2021, 58(1): 102436−102437

[12] Nasirifard P, Mayer R, Jacobsen H A. FabricCRDT: A conflict-free replicated datatypes approach to permissioned blockchains[C]//Proc of the 20th Int Middleware Conf (Middleware). New York: ACM, 2019: 110−122

[13] Sharma A, Schuhknecht F M, Agrawal D, et al. Blurring the lines between blockchains and database systems: The case of Hyperledger Fabric[C]//Proc of the 37th Int Conf on Management of Data (SIGMOD). New York: ACM, 2019: 105−122

[14] Xia Qing, Dou Wensheng, Guo Kaiwen, et al. Survey on blockchain consensus protocol[J]. Journal of Software, 2021, 32(2): 277−299 (in Chinese)
(夏清, 窦文生, 郭凯文, 等. 区块链共识协议综述[J]. 软件学报, 2021, 32(2): 277−299)

[15] Lomet D, Fekete A, Wang Rui, et al. Multi-version concurrency via timestamp range conflict management[C]//Proc of the 28th Int Conf on Data Engineering (ICDE). Piscataway, NJ: IEEE, 2012: 714−725

[16] Thakkar P, Nathan S, Viswanathan B. Performance benchmarking and optimizing Hyperledger Fabric blockchain platform[C]//Proc of the 26th Int Symp on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). Piscataway, NJ: IEEE, 2018: 264−276

[17] Gorenflo C, Lee S, Golab L, et al. FastFabric: Scaling Hyperledger Fabric to 20000 transactions per second[J]. International Journal of Network Management, 2020, 30(5): 2099−2100

[18] Ruan P, Loghin D, Ta Q T, et al. A transactional perspective on execute-order-validate blockchains[C]//Proc of the 38th ACM Int Conf on Management of Data (SIGMOD). New York: ACM, 2020: 543−557

[19] Xu Lu, Chen Wei, Li Zhixu, et al. Solutions for concurrency conflict problem on Hyperledger Fabric[J]. World Wide Web, 2021, 24(1): 463−482

[20] Sousa J, Bessani A, Vukolic M. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform[C]//Proc of the 48th Annual IEEE/IFIP Int Conf on Dependable Systems and

Networks (DSN). Piscataway, NJ: IEEE, 2018: 51−58

[21] Nakaike T, Zhang Qi, Ueda Y, et al. Hyperledger Fabric performance characterization and optimization using goLevelDB benchmark [C/OL]// Proc of the 2nd Int Conf on Blockchain and Cryptocurrency (ICBC). Piscataway, NJ: IEEE, 2020[2023-09-09].https://ieeexplore.ieee.org/document/9169454

[22] Raman R K, Vaculin R, Hind M, et al. Trusted multi-party computation and verifiable simulations: A scalable blockchain approach[J]. arXiv preprint , arXiv: 1809.08438, 2018

[23] Dinh T T A, Wang Ji, Chen Gang, et al. BlockBench: A framework for analyzing private blockchains[C]//Proc of the 43rd ACM Int Conf on Management of Data (SIGMOD). New York: ACM, 2017: 1085−1100

[24] Meir H, Barger A, Manevich Y, et al. Lockless transaction isolation in hyperledger fabric[C]//Proc of the 2nd Int Conf on Blockchain (Blockchain). Piscataway, NJ: IEEE, 2019: 59−66

[25] Zhang Shenbin, Zhou Ence, Pi Bingfeng, et al. A solution for the risk of non-deterministic transactions in Hyperledger Fabric[C]//Proc of the 1st Int Conf on Blockchain and Cryptocurrency (ICBC). Piscataway, NJ: IEEE, 2019: 253−261

[26] Reed D P. Naming and synchronization in a decentralized computer system[D]. Cambrideg, Massachusetts: MIT Press, 1978

[27] 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

[28] Nguyen T S L, Jourjon G, Potop-Butucaru M, et al. Impact of network delays on Hyperledger Fabric[C]//Proc of the 38th Conf on Computer Communications Workshops (INFOCOM WKSHPS). Piscataway, NJ: IEEE, 2019: 222−227

[29] Schaefer C, Edman C. Transparent logging with Hyperledger Fabric[C]// Proc of the 1st Int Conf on Blockchain and Cryptocurrency (ICBC). Piscataway, NJ: IEEE, 2019: 65−69

[30] Dinh T T A, Liu Rui, Zhang Meihui, et al. Untangling blockchain: A data processing view of blockchain systems[J]. IEEE Transactions on Knowledge and Data Engineering, 2018, 30(7): 1366−1385

[31] Wang Rui, Ye Kejiang, Meng Tianhui, et al. Performance evaluation on blockchain systems: A case study on Ethereum, Fabric, Sawtooth and Fisco-bcos[C]// Proc of the 17th Int Conf on Services Computing. Berlin: Springer, 2020: 120−134

[32] Zhang Zhiwei, Wang Guoren, Xu Jianliang, et al. Survey on data management in blockchain systems[J]. Journal of Software, 2020, 31(9): 2903−2925 (in Chinese)
(张志威, 王国仁, 徐建良, 等. 区块链的数据管理技术综述[J]. 软件学报, 2020, 31(9): 2903−2925)

[33] Liu Hanqing, Ruan Na. A survey on attacking strategies in blockchain[J]. Chinese Journal of Computers, 2021, 44(4): 786−805 (in Chinese)
(刘汉卿, 阮娜. 区块链中攻击方式的研究[J]. 计算机学报, 2021, 44(4): 786−805)

[34] Zhong Botao, Wu Haitao, Ding Lieyun, et al. Hyperledger Fabric-based consortium blockchain for construction quality information management[J]. Frontiers of Engineering Management, 2020, 7(4): 512−52

[35] Tarjan R. Depth-first search and linear graph algorithms[J]. SIAM Journal on Computing, 1972, 1(2): 146−160

[36] Kahn A B. Topological sorting of large networks[J]. Communications of the ACM, 1962, 5(11): 558−562

[37] Sharma A, Schuhknecht F M, Agrawal D, et al. How to databasify a blockchain: The case of Hyperledger Fabric[J]. arXiv preprint, arXiv: 1810.13177, 2018

Wu Haibo, born in 1981. PhD, associate professor. Senior member of CCF. His current research interests include blockchain technology, future Internet architecture, data-driven network, and NLP.
吴海博, 1981 年生. 博士, 副教授. CCF 高级会员. 主要研究方向为区块链技术、未来网络架构、数据驱动网络、然语言处理.

Liu Hui, born in 1996. Master. His main research interests include blockchain technology, NDN, and distributed database.
刘　辉, 1996 年生. 硕士. 主要研究方向为区块链技术、内容分发网络、分布式数据库.

Sun Yi, born in 1979. PhD, professor. His main research interests include blockchain technology, distributed applications, and Internet video distribution.
孙　毅, 1979 年生. 博士, 研究员. 主要研究方向为区块链技术、分布式应用、互联网视频分发.

Li Jun, born in 1968. PhD, professor. His research interests include Internet security, Internet architecture, artificial intelligence, and big data application.
李　俊, 1968 年生. 博士, 教授. 主要研究方向为网络安全、网络架构、人工智能、大数据应用.