分类号：TP311　　单位代码：10422

密　级：　　学　　号：


山东大学
SHANDONG UNIVERSITY

# 硕士学位论文

Thesis for Master Degree

（专业学位）

论文题目：区块链计算密集型合约的并行策略研究

**Research on Parallel Strategies for Blockchain Computational Intensive Contracts**

作　者　姓　名　　

培　养　单　位　山东大学

专　业　名　称　软件工程

指　导　教　师　　

合　作　导　师　　

2022 年 3 月 14 日

## 原 创 性 声 明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的科研成果。对本文的研究作出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律责任由本人承担。

论文作者签名：_____ 日 期：_____

## 关于学位论文使用授权的声明

本人同意学校保留或向国家有关部门或机构送交论文的印刷件和电子版，允许论文被查阅和借阅；本人授权山东大学可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或其他复制手段保存论文和汇编本学位论文。

(保密论文在解密后应遵守此规定)

论文作者签名：_____导师签名：_____日 期：_____

# 摘　要

智能合约作为部署在区块链上的一种代码脚本，能够在被事务调用时自动为用户完成预定的业务或计算。但伴随着区块链的应用场景变得多样而复杂，智能合约正变得越来越复杂，智能合约中所包含的计算量也出现了大幅上升。据统计，以太坊上已经出现了计算耗时超过 20 秒的智能合约。因此，计算密集型合约事务被用来指代此类验证过程需要消耗大量计算资源的智能合约的调用事务。如何实现区块链对计算密集型合约事务的快速验证，成为区块链突破性能瓶颈，实现多场景落地的重要研究。

在众多已有研究中，使用并行验证的策略，缩短事务验证用时，已被证明是提高区块链吞吐性能的有效手段。然而，这些方案大多针对简单事务提出，并没有考虑计算密集型合约事务的特殊需求，没有产生对并行技术和计算密集型合约事务的系统讨论，难以指导两者的结合。另外，区块链对计算密集型合约事务的支持能力有限。一方面，智能合约多核适配能力差，在被多核并行执行时存在重放不确定性问题。另一方面，计算密集型合约事务尚缺乏一种具备良好链上链下数据一致性的验证者困境解决方案。为此，本文在以下方面做出了努力。

首先，针对现有并行方案无法良好的应用于计算密集型合约事务这一问题，文章提出了面向计算密集型合约事务的两种并行分类。文章使用典型的计算密集型合约案例，分析了计算密集型合约事务的特殊需求，从计算密集型合约事务的角度将现有并行技术进行了分类讨论，以此建立了两者之间的联系，并探讨了方案集成与组合的可能性，分析了计算密集型合约事务在并行过程中存在的两处缺陷。

其次，针对智能合约多核并行执行时存在的重放不确定性问题，文章提出了支持重放确定性的并行编程合约。包括提供了一种能够允许多个计算核心同时验证一个合约事务的智能合约模型，提高计算密集型合约事务的验证效率。同时，该模型能够解决并行任务中存在的资源无序竞争问题，保证并行结果的重放确定性。

最后，针对计算密集型合约事务存在的验证者困境，文章提出了基于链上挑战协议的链下计算方案。该方案能够允许计算密集型事务不必在共识过程中被重复计算，而事务结果仍然能够获得正确共识，计算过程也具备了分组并行的特性。该模型解决了之前同类模型存在的链上链下数据短期不一致的问题，在大幅提升区块链吞吐的同时，能够有效减少无效交易。链上挑战协议中同样包含了相应的激励模型以及资源超前交付策略，进一步保障了链上挑战协议的可靠性及可用性。

**关键词：**区块链；智能合约；链上链下协同；并行执行

# ABSTRACT

Smart contracts, as a kind of code script deployed on the blockchain, can automatically complete a predefined business or computation for the user when called by a transaction. However, as blockchain application scenarios become diverse, smart contracts are becoming increasingly complex, and the amount of computation contained in smart contracts has seen a significant increase. Therefore, computationally intensive contract transactions are used to refer to such invocation transactions of smart contracts where the validation process consumes a lot of computational resources. How to achieve fast verification of computation-intensive contract transactions by blockchain has become an important research for blockchain to break through the performance bottleneck and achieve multi-scenario implementation.

Among the existing studies, the strategy of using parallel execution to shorten the transaction verification time has been shown to be an effective means to improve the throughput performance of blockchains. However, most of these schemes are proposed for simple transactions and do not consider the special needs of computationally intensive contract transactions, and do not generate a systematic discussion between parallel techniques and computationally intensive contract transactions to guide the combination of the two. In addition, blockchain has limited ability to support computationally intensive contract transactions. On the one hand, smart contracts have poor multicore adaptability and suffer from replay indeterminacy when being executed in parallel. On the other hand, computationally intensive contract transactions lack a verifier dilemma solution with good on-chain and off-chain data consistency. To this end, this thesis makes efforts in the following aspects.

First, for the lack of theoretical guidance on the application of existing parallel schemes to computationally intensive contract transactions, the thesis proposes two parallel classifications for computationally intensive contract transactions. Using a typical computation-intensive contract case, the thesis analyzes the special requirements of computation-intensive contract transactions, broadly classifies the existing parallel techniques into two categories, and discusses the solutions in both categories of parallelism in several dimensions, explores the possibility of integration and combination of existing solutions from the perspective of computation-intensive contract transactions, and analyzes the two shortcomings of computational intensive contract transactions in the parallelization process.

Second, to address the problem of replay uncertainty when smart contracts are executed in parallel by multiple cores, the thesis proposes parallel programming contracts that support replay determination. This includes providing a smart contract model that can allow multiple

computing cores to verify a contract transaction simultaneously, improving the verification efficiency of computationally intensive contract transactions. At the same time, the model can solve the problem of disorderly competition for resources that exists in parallel tasks and guarantee the replay determination of parallel results.

Finally, to address the verifier's dilemma of computationally intensive contract transactions, the thesis proposes an off-chain computation scheme based on an on-chain challenge protocol. The scheme can allow computation-intensive transactions to not have to be repeatedly computed in the consensus process, while the transaction results can still get correct consensus, and the computation process also has the grouping parallelism feature. This model solves the problem of short-term inconsistency of on-chain and off-chain data that existed in previous similar models, and can effectively reduce invalid transactions while significantly improving blockchain throughput. The on-chain challenge protocol also contains the corresponding incentive model and resource over-delivery strategy, which further guarantees the reliability and availability of the on-chain challenge protocol.

**Key words:** Blockchain; Smart Contracts; On-Chain and Off-Chain Collaboration; Parallel Execution

# 目　录

# CONTENTS

# 1 绪论

## 1.1 研究背景及意义

2008 年比特币问世以来，区块链从最初的加密货币开始，直到现在被认为是可能带来多个行业的颠覆性进步的技术，涉及领域包括金融、物联网、医疗保健、能源、物流等[1]。至此，区块链作为新一代基础设施的前景开始被挖掘。

与传统的中心化解决方案相比，区块链具有不可篡改、去中心化、强透明性等显著优势。然而，另一方面，区块链去中心化的优势对其吞吐性能形成了制约，使其难以支持真实场景的性能需求。以金融为例，Mastercard[2]的存取款业务，平均吞吐量在 40000TPS（transaction per second）左右，VISA[3]的信用卡业务，平均吞吐量则在 65000TPS 以上。而当前的主流区块链（仅开源）项目，以太坊吞吐量只能维持在 15TPS 左右。Fabric[4]和 Fisco[5]，其吞吐量最高都只能够达到 20000 左右。

此外，区块链上越来越复杂的事务，更加剧了区块链技术在性能方面的危机。伴随着区块链技术在越来越多的领域中被应用，区块链也被尝试接入更多更复杂的系统，承担更多样、流程更复杂的计算任务。其中表现最明显的是人工智能技术与区块链技术的结合。据 OpenAI[6]调查，自 2012 年至今，人工智能算力消耗每年增长不少于十倍，对摩尔定律形成了直接的挑战。此类复杂的业务与计算，在区块链中通常借助智能合约实现。于是，区块链合约的调用事务的算力消耗也逐年递增[7]。这一类验证过程需要消耗大量计算资源的智能合约的调用事务，被称为计算密集型合约事务[8]。寻找对计算密集合约事务的更高效的处理方法，是区块链实现性能突破，进行多场景落地必不可少的一步。

在区块链性能突破方面，已经有了许多有效策略。例如，链下计算[9]、跨链、分片、DAG 等技术。其中的绝大部分技术对区块链吞吐的提升都通过并行实现[10]。但此类技术通常面向普通交易而被设计，例如转账，其中只包含极简单的计算内容。因此，对计算密集型合约事务进行分析，从事务并行的观点挖掘现有技术在区块链性能提升方面的本质，并面向计算密集型合约事务的特殊需求进行改善，是极有意义的。另外，这些技术常常是封闭且互不关联的。在多个维度上探讨其作用本质，有助于指导进行方案间的集成，形成一个联合的立体架构，带来性能的进一步突破。本文将诸多技术的特性、作用原理以及与计算密集型合约事务的关系进行了总结与整理，为技术间的兼容与选择提供了参考，为计算密集型合约事务并行技术的缺陷改进提供了建议。

另外，计算密集型合约事务验证耗费的算力或者时间更多，区块链在面对此类事务时无法保证较高的吞吐[11]。研究支持多核适配的智能合约模型，能有效地发挥并行计算

的优势，显著提高计算密集型合约事务的处理效率。区块链智能合约在多核适配方面的困境之一，在于并行编程形式实现的智能合约，在被执行时会频繁发生资源竞争，那么该合约的调用事务在其他成员上无法保证重放结果是相同的。有关资源竞争处理的方法以及重放确定性保证的方法，已经有所出现，但该类技术如何在区块链智能合约这一特殊环境下，提供资源管理支持仍然需要进一步研究。因此，研究支持并行编程、适配多核计算的智能合约模型，研究智能合约并行编程的确定性重放方法，是解决该类问题的重要途径。

最后，计算密集型合约事务会为区块链系统带来过高的计算压力，导致区块链吞吐性能下降。链下计算是解决此类问题的一种有效手段。一方面是因为链下计算改变了区块链对事务的执行方式，共识成员不必再付出巨大的算力代价，网络的算力压力被缓解，共识过程变得更短更快。另一方面，事务在链下的计算可以通过成员分工分组实现并行，不同分组之间同时进行多个计算密集型合约事务的执行，可以使时间消耗大大下降。因此，使用链下计算，将计算密集型事务的计算任务从共识过程中剥离，交由部分高性能设备并行完成，同时保证事务在不被重放的前提下正确共识，能够有效提高区块链系统的效率以及稳定性。但随之而来的计算结果可靠性难题以及链上链下数据一致性问题，需要进一步的解决。

## 1.2 问题与挑战

区块链中的计算密集型事务，通常使用智能合约来实现。然而，智能合约无法支持并行编程，这导致计算密集型合约事务无法在多核心上通过并行计算加快验证过程。这主要是因为智能合约环境下的并行编程与传统并行编程并不相同。合约的最终目的是实现状态机复制，所以智能合约的并行模型并不满足离散事务（Discrete Transaction）的计算特征[12]，必须保证子计算任务的可序列化以及序列化结果的唯一性[13]。这是因为区块链事务必须保证在任意时刻，任意参与成员都能够执行该合约事务，并且获得唯一结果，这是区块链实现分布式账本间信任的前提条件，通常称其为重放确定性。与之相反的，普通并行编程实践中，在并行计算的过程中，如果线程间存在资源竞争，那么资源读写结果受访问顺序影响严重，线程间计算进度受偶然性影响较大[14]，因此访问序列不受控，那么最终计算结果通常是不唯一的，这是并行编程的重放不确定性问题。

因此，对合约并行编程模型的研究，在于在编程语言的上层，如何实现计算任务的调度。因此，如何在区块链合约中应用并行编程成为一个难题。有研究尝试通过制定合约代码编写标准，来严格控制线程并行顺序，实现并行编程的结果确定性，但结果并不理想。因为该种方式是一种对编程人员的软性标准，并不能严格地保证事务计算结果能

够在任何一个参与成员上再现。一旦因为编程漏洞导致再现失败，可能会对区块链的安全性带来巨大威胁。

传统数据库系统的资源管理策略能够对该类问题提供启发。STM 类技术可以在不提供任何声明、不进行静态分析的情况下，提供细粒度的、准确的事务读写集[15]。基于读写集与资源依赖图的计算任务序列化方法以及系列化结果唯一性保证机制，曾被使用在 Parblockchain[16]中用于在合约事务层面进行事务并行与排序，具有很好的参考价值。通过建立并行计算任务之间的确定性序列，可以保证合约事务并行编程的确定性重放，构建具备支持并行编程、严格具备确定性重放保证的智能合约模型。

相比之下，链下计算方案，则是通过链下计算结果的链上共识，实现计算密集型事务的并行与快速处理[17]。在合约事务发布以后，不再由全部共识成员承担重新计算并验证合约事务结果的任务，而是由部分高性能设备完成，这是因为高性能设备更不容易因计算密集型合约事务导致堵塞。另外，计算设备之间，能够以并行的模式处理事务，这能够有效提升区块链吞吐量。最后，因为链下的计算设备组仅仅是区块链全部参与成员的很小一部分，因此，计算密集型事务的算力消耗被放大的程度会大大降低。

链下计算的重要方案之一，是基于交互式验证协议的链下计算协议[18]。因为区块链成员间不存在信任假设，为了保证所有参与成员，包括普通区块链成员节点与高性能计算中心，能够以理想的模式参与协议实施，该协议设计了相应的激励策略，这是链下计算方案不得不解决的一个重要问题[19]。该协议中，链上合约事务的计算奖励（比如 gas）不再分配给记账人，而是由计算者认领。计算者将合约事务的结果公开在区块链网络中，由所有人进行挑战，挑战者则需要支付一定量的押金，然后计算者和挑战者共同出示计算的证明，由正确的一方获得全部奖金。

该方案使用交互式验证协议解决了计算结果可信性的问题。但同样引入了存在新问题，比如过多的成员交互使协议容易遭受“脱机攻击（offline attacks）”。而且，链下计算结果短时间内无法在链上生效，这段时间可能会导致链上链下数据的短期不一致，从而导致许多无效交易的产生，阻碍系统吞吐的提升。简单的改变事务上链时间，会使链下计算协议在不安全与资源长期阻塞之间难以寻找到良好的平衡。对此类问题，本文设计了一种基于链上挑战的链下计算协议，该协议同样包括一系列激励策略以及利用链上未确认资源推进新事务的超前交付策略，能够帮助实现高效的合约事务链下计算以及防止合约事务结果阻塞带来的资源释放困难等问题。

## 1.3 主要贡献

本文主要解决了计算密集型合约事务在区块链上通过并行提高吞吐性能的相关问题。详细贡献被记录如下：

（1）本文提出了面向计算密集型合约事务的两种并行分类方法。文章通过分析计算密集型合约事务的需求，将现有的区块链并行策略进行了分析与比较，总结了现有策略在促进计算密集型合约事务并行处理方面的两种作用原理，并以此在多种维度上讨论了并行方案间集成与叠加的可能性，并进一步提出了计算密集型合约事务高效并行现存的两个重要不足。

（2）本文面向计算密集型合约事务，提出了支持重放确定性的并行编程智能合约模型，以实现计算密集型合约的多核适配。包括一种支持在区块链上运行并行编程的智能合约模型，该模型能够有效发挥多核设备的计算性能，缩短计算密集型合约事务的验证用时，提高其吞吐量；以及一种严格的重放确定性保障方法。该方法能够解决计算密集型合约事务并行验证中，子计算任务间的资源竞争问题，保证事务重放结果不会受资源竞争中偶然性因素的影响。

（3）本文提出了一种基于链上挑战的链下计算协议，该协议能够使不同计算密集型合约事务在不同的计算设备组中被并行地执行，而共识过程中不需要再重复执行事务中的计算。该协议解决了交互式验证协议中，因为链上链下数据短期不一致导致的无效交易频发的问题。另外，协议中还包括一种改进的激励策略，该策略能够激励链上挑战协议的良好运行；以及一种链上数据能够在未确认状态下被使用的超前交付策略，该策略能够使链上资源不会因长时间等待释放而发生堵塞。

## 1.4 文章组织结构

本文共包含六章节内容，各章节组织结构如下：

第二章简述了文章研究所涉及到的基础知识，以及文章所使用的一些基本术语与定义。包括智能合约的基本原理，以及计算密集型合约事务的基本定义。第二章同样描述了文章所使用到的一些基本研究与结论。包括读写集、重放确定性、验证者困境、交互式验证协议以及链下计算等。

第三章首先描述了一种典型的计算密集型合约事务示例，并以此对计算密集型合约事务在事务并行方面的特殊性质与需求进行了讨论，提出了计算密集型合约事务并行的两种分类方法。在 3.4 节中，文章将现存区块链并行方案在多种维度上进行了讨论，并探讨了不同维度与分类的并行方法间的集成思路。最后，该章针对两种分类中存在的并行问题，提出了改进的可能性方案。

第四章首先描述了一种支持在区块链上运行并行编程的智能合约模型，包括该模型的基本组成、任务调度过程以及执行过程中的资源共享控制方法。第四章在该合约模型的基础上提出了重放确定性保障方法，该方法中包括基于计算任务读写集的多版本资源控制方法，以及计算任务的重序列化方法。重序列化方法中具体描述了逻辑序建立方法、

资源依赖图以及重序列化图等内容。4.4 节中提供了对重放确定性的证明。最后，该章通过实验证明了本文提出的智能合约模型具备更高的并行效率，能够有效适配多核架构。实验同样证明了该合约模型在纠正重放结果时，能够有较好的性能表现。

第五章提出了一种链下计算协议用以提高计算密集型合约事务的并行执行效率。章节中首先阐明了该协议的参与角色与基本流程，以及一种无响应挑战以解决协议成员消极响应的问题。第五章同样提供了该协议的激励策略来保证协议的良性运行，以及超前交付策略保证资源的快速利用。另外，该章对该协议的并行效率进行了数学分析，并提供了该协议的安全性情况分析。最后，该章通过实验证明了该协议能够有效通过事务并行提高区块链对计算密集型合约事务的处理性能。实验同样证明了该协议相比其他协议能有效抑制交易失效的问题，并且能够提供更快的资源使用效率。

第六章总结了文章的内容，对文章的得失做了思考，并提出了对未来工作的展望。

# 2 研究基础

## 2.1 智能合约

智能合约，最初被使用来指代能够自动执行的一般法律合同[20]（automation of legal contracts in general）。后来随着区块链技术在更多领域应用的潜力被挖掘出来，现在智能合约通常指运行在区块链上的，由代码脚本表示的承诺。智能合约能够被事务（或称交易）调用，使得区块链有能力处理复杂的、自定义的事务逻辑[21]，是区块链由简单账本向综合服务平台转变的关键技术。

通常来说，每一份被部署的智能合约，都有独立的存储空间。以以太坊为例，智能合约的被部署后，会被分配一个独立的地址，合约可以通过这份地址被检索。合约代码会被存储在这个地址中，合约中使用到的数据及变量，以及合约的计算结果（除对其他账户的转账外）都会被存储在该地址中。一次合约调用，对本合约内存储的数据内容、其他账户的加密货币余额、其他合约中存储的数据内容，都有可能发生改写。我们将这些数据统称为数据资源，以下简称为资源。两个合约调用若同时改写了某一项资源，则被称为资源竞争，两事务将只有其一会生效。本文章不讨论资源的具体类型、形式与存储方式，只关注事务可能存在的资源竞争，以及资源的生效与失效问题。

智能合约所使用的编程语言是多样的。以太坊使用了 Solidity[22]作为合约编程语言，该语言是一种图灵完整的，专门运行在以太坊虚拟机上的编程语言。Monoxide[23]则为自己特殊的底层架构设计了专用的合约编程语言 PREDA，该语言强调以接力执行的方式为分布式架构上的并行计算提供编程支持。除此之外，许多区块链项目强调使用已有的编程语言来编写智能合约。比如 Fabric 的链码[24]（chaincode）支持以 JAVA、Go 等主流高级编程语言提供智能合约。基于 Cosmos SDK[25]开发的公链平台 Juno 支持使用 Rust、GO 等作为其智能合约编程语言，等等。支持使用此类成熟的高级编程语言编写智能合约，能够利用其丰厚的经验积累以及庞大的从业人员群体，快速促进智能合约成熟，是加快区块链推广的重要思路。

## 2.2 计算密集型合约事务

区块链系统最早被应用于加密货币支付，最初大家将区块中记录的一次加密货币转移记录称作一笔交易（transactions）。事务（transactions）最常被使用来称呼数据库系统中对数据项发生读取或可能发生改写的一个程序执行单元[26]。因此，在对区块链系统与

分布式数据库的某些相似特性进行讨论时，通常不对这两个概念做区分。值得注意的是，区块链系统中通常不对仅包含对数据项的读操作的事务进行讨论。

从事务的角度来看，区块并不是本地数据的全部，而是事务打包的一种结构。每个新区块的产出，都会使所有参与成员节点执行区块中的事务，从而更新本地数据。因此，区块链更贴近一种依赖有序地重复执行事务来进行状态机复制。

传统区块链事务，大多是转账交易，其执行过程需要的计算资源消耗通常是被忽略的。但随着区块链对智能合约的支持，执行区块链事务消耗的算力正逐渐增加。通常，此类包含大量计算内容的合约被称为计算密集型合约[27]（Computationally Intensive Contracts, CIC）。文章中将对该类合约产生了调用的事务称为计算密集型合约事务，记作 CICT。该类事务的典型特征就是其执行过程需要消耗大量的计算资源，因此普遍耗时较长。

为了限制合约事务的算力消耗，许多区块链系统都对单笔事务的算力消耗上限进行了约束，比如以太坊的 gaslimit[28]。但是，这会极大的限制区块链智能合约的应用场景，尤其是在机器学习、大数据计算的需求下，如果无法支持计算密集型合约事务的计算，区块链能产生的社会意义将大打折扣。

但区块链对 CICT 的支持并不容易实现。一方面，CICT 的验证必须要完整地对其内计算进行一遍执行，这会消耗过长的时间，可能会阻碍状态机的更新，使区块链性能受限。另一方面，区块链的参与成员并不存在对 CICT 的验证动机，这就是 CICT 的验证者困境（见 2.5 节）。本文中，我们认为对 CICT 验证效率产生最大影响的是计算耗时，传输、存储等方面可能会为 CICT 验证效率带来的影响，本文不做讨论。

## 2.3 并行与多核适配

### 2.3.1 读写集

对任意两个事务，如果他们访问了相同的数据，并且其中一个进行了写操作，那么他们间就会发生冲突。这种情况下，后一个事务必须等待前一个事务完成，其他的执行顺序都是不被允许的[16]。因此，事务冲突的检测是必要的，事务冲突检测的前提则是要记录下每一个事务执行时所使用的数据项的版本，这是多版本并行控制（Multi-Version Concurrency Control, MVCC）的重要基础。

读写集作为版本化数据的传递与校验的重要工具，能够提供相应的功能支持。读写集将每一个事务执行时所使用到的数据项版本化，记录为读集，然后将执行后发生了改写的数据项版本化，记录为写集。至此，读写集很好地将事务的一次执行进行了记录，并且其中包含事务发生的最小环境基础以及精确结果。

### 2.3.2 重放确定性保障

随着多核设备的广泛应用，也随着计算需求变得更加庞大和复杂，多线程编程成为提高程序执行效率的重要手段。但多线程程序的执行具有不确定性[29]，该种不确定性主要指多线程程序在不同的两次执行中，存在着完全不同的偶然性因素，这可能会导致同一程序的两次执行有不同的结果。

对传统程序来说，偶然性因素主要因为存储器竞争而发生。不同线程中的计算会对共享存储器中的数据进行同步访问，因此，如何保证不同的两次执行中，线程对存储器的竞争呈现相同结果，是能否保证重放确定性的关键。

更详细地说，存储器的竞争主要有两种表现形式，分别是同步和数据竞争。同步指并行程序编程过程中，数据在有意控制下进行更新与共享，通常易于检测，也容易在编程过程中进行修正。数据竞争则源于程序进行数据相关操作时，线程间的互斥与同步关系存在偶然性，此类错误很难进行显式的声明与静态的检测，但却会在程序调试时直接影响结果的正确性。

在重放确定性方面的研究，有两种主要的方案。一种是通过直接记录程序执行时的存储器竞争顺序，来指导重放，实现确定性。Instant Replay[30]是其中的代表，通过记录指令对存储器的访问顺序来保证重放确定性，但其明显缺陷便是存储与时间开销明显变大。PRES[31]是一种典型的改良方案，该方案进一步压缩了额外的时间开销，但重放确定性上存在了部分妥协。为了进一步压缩时间开销但不影响其重放确定性，另一类方案尝试添加硬件支持。代表性方案包括 FDR[32]、RTR[33]等，该类方案使用专用的硬件来记录并控制程序指令的逻辑序，对降低额外时间开销有极大的效果。

但区块链智能合约的重放确定性有其特殊的需求。一方面，区块链的吞吐性能有限，无法接受较大的时间开销，因此，区块链合约事务必须使用一些更乐观的重放确定性保证方法。另一方面，区块链作为一种基础设施，无法对运行的硬件设施做特殊约束。最后，合约事务的运行模式与常规程序不同，合约会先被部署在区块链上，在将来的时刻被事务调用，事务会传入新的参数，这可能会使指令对数据的访问顺序随时发生变化。

## 2.4 并行与链下计算

区块链相关研究中，对“链下（off-chain）”行为的概念和定义繁杂而模糊。与本文要做出较大区分的是 NFT.storage[34]中使用的链下存储这一概念。链下存储强调数据存储设备不应该属于区块链成员。与本文概念相近的是闪电网络[35]中的链下支付概念，强调行为的执行过程不会被共识并记录，而仅将结果记录在区块链上。此处重新对本文中使用的“链下”概念做出明确定义：

任何行为，若不会进入区块链的共识过程，那称该行为是链下的，与发起成员的类型、所处的网络、行为的内容没有关系。

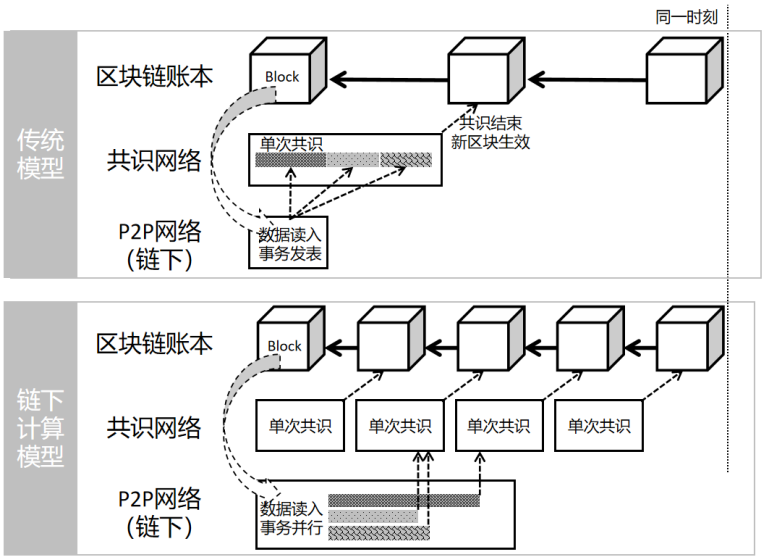



图 2.1 链下计算示意

Fig. 2.1 Off-Chain Computing

区块链成员设备间依赖一个 P2P 网络相互连接。区块链的必要信息交互会在该网络中进行，比如维持连接、广播交易、请求数据。而共识层是一个逻辑分层，一个区块一旦被打包并发布，就进入了共识过程。共识过程中，共识成员会验证区块的合法性，并逐个执行区块中事务的计算任务。对 CICT 来说，这一过程的时长主要受共识算法以及事务执行时长的影响。共识过程结束后，被共识的数据正式在区块链帐本中被承认，成为受信数据。因此，共识过程的长短通常决定了区块链的吞吐性能。链下计算，则指合约事务中的计算会在共识过程外，由部分成员执行，共识过程中只需要验证结果的合法性即可。这能有效减少网络共识的计算压力，缩短共识过程，提高区块链吞吐性能。另外，链下的不同成员群体之间，可以同时验证不同的事务，这种并行的特性能大幅提高区块链对 CICT 的吞吐。

### 2.4.1 验证者困境

在比特币网络中，新区块的打包者，能够获得称作 coinbase 的比特币奖励[36]。以太坊在打包奖励以外，新增了 gas 奖励，该奖励会发放给完成了合约事务执行的交易打包者，以激励参与成员去主动完成合约事务中的计算，并将其打包进区块链。但对除打包者之外的普通参与成员者来说，其并不具备完整执行合约内计算以验证事务的动机，尤其是 CICT。假设任意非打包者愿意为事务验证承担的算力开销是ε，该意愿来自于任意一个区块链参与者对维持区块链正常运行的预期收益，称该类非打包的验证者为ε共识

机[37]（ε-consensus computer）。如果任意一个事务的验证开销高于ε，那么验证者都更倾向直接跳过对该事务的验证。这会为区块链带来严重的安全隐患。

打破区块链验证者困境的充要条件是使得非打包参与成员的计算代价要远低于打包者的计算代价[38]，直到低于任意非打包参与成员的验证收益ε。当前最可靠的方案是借助链下计算完成 CICT 的计算任务，区块链参与者只承担极低的结果可靠性验证开销。一种重要的可靠性保障方法是借用可信硬件完成计算，那么对计算任务可靠性的验证就可以使用对计算任务承担人身份的验证代替，验证代价下降为一次非对称加密的验证难度。但是该类方法存在对可信硬件技术的极度依赖，目前的可信硬件仍不能被证明完全无法被篡改[39]，更不成熟，无法大量投入使用。可验证计算是另一种计算结果可靠性保障方法[40]，最初用于解决任务分包以及任务委托计算中产生的计算结果的可靠性验证。客户端由于自身能力或资源的限制，需要将复杂函数 F(x)在点 a 上的计算任务委托给一个计算能力强的服务器。由于服务器不是完全可信的，所以需要服务器返回一个可验证其结果正确性的证明，该证明验算过程比实际计算 F(a)值本身简易的多。然而，截至目前，可验证计算能够提供的计算种类仍远远不能满足需求。

### 2.4.2 可验证计算与交互式验证协议

可验证计算（Verifiable computing, VC）是一种确保能够在不可信方执行可信计算的技术[41]，该技术是实现 CICT 结果在共识成员上，不重放地完成正确性验证的重要手段。可验证计算由三部分算法组成[42]：

（1）密钥生成算法，假设计算任务为$task_i$，秘密参数为$\lambda$，能够通过密钥生成算法获得评估密钥$EK_i$（evaluation key）和公共验证密钥$VK_i$（public verification key）。记作：

$$KeyGen(\lambda, task_i) \Rightarrow (EK_i, VK_i) \tag{2.1}$$

（2）计算算法，对于某个需求输入$u$，结合评估密钥$EK_i$，该算法能够计算获得该输入对应的结果$y$，以及该结果的证明$\pi_y$，记作：

$$Compute(EK_i, u) \Rightarrow (y, \pi_y) \tag{2.2}$$

（3）验证算法，通过公共验证密钥$VK_i$与结果证明$\pi_y$，任何人能够验证对某输入$u$的结果$y$的正确性，记作：

$$Verify(VK_i, u, y, \pi_y) \Rightarrow \{0,1\} \tag{2.3}$$

可验证计算，通常需要满足三个基本条件：

（1）正确性。任意设备，如果其运算过程是诚实的，那么其运算结果必然能够通过$VK_i$与$\pi_y$的正确性验证。

（2）可靠性。任意设备，如果其运算过程是错误的，那么其运算结果必然无法通过$VK_i$与$\pi_y$的正确性验证。

（3）高效性。验证计算结果的设备对数据进行格式转换的时间，加上通过验证函数的时间，必须小于在本地直接执行原计算任务的时间。

基本条件中，（1）、（2）通常是必须条件，在特殊需求下，条件（3）可能不被要求。可验证计算的解决方案正逐步走向成熟。其实现思路基本分为两种，基于可信硬件，以及基于密码学。可信硬件主要依赖安全的非遗失性存储器(secure non-volatile memory)生效。可信硬件可以提供隔离的执行环境，任何恶意行为都无法获取或者改写处理器的内部状态。可信硬件方案能够保证数据和计算过程的隐私性和完整性[43]，为建立安全的分布式系统提供了基础。虽然该种方案效率极高，但当前实用性尚无法保障[39]，除设备昂贵外，技术亦不够成熟，频繁有安全漏洞被公布。

而基于密码学的可验证计算，常常因效率太低难以实际使用。近期获得应用更广泛的，是 zk-SNARKs 及其一系列算法[44]，该类算法已经在 zk-cash 区块链系统中成功落地，被用于资产的隐私保护。该类算法目前能够支持的计算行为尚十分匮乏，且通常只能支持简单计算行为，为复杂多变的智能合约提供完善的可验证计算短期内仍难以实现。

除此之外，TrueBit 中提供了一种极具实用性的可验证计算方案，交互式验证协议（interactive verification game，又译作交互式验证游戏或交互式经济加密协议）。交互式验证协议有两个以上设备参与，执行计算任务的设备称为计算者，验证计算结果是否可信的设备称为验证者。

该协议将一个计算任务在逻辑上的总时长$t$（假设构成计算任务的一项基本运算所用时常为 1，计算任务在逻辑上的总时长即与该任务的基本运算个数相等）平均分为$c$（超参，为常数整数）份，即可以找到$c$个时刻，分别为：

$$\frac{t}{c}, \frac{2t}{c}, \frac{3t}{c}, \ldots, \frac{ct}{c} \tag{2.4}$$

将这些时刻，执行计算任务的虚拟机的状态进行记录，记作$VMMS$，并将记录值构造为 Merkle 树，如下图：

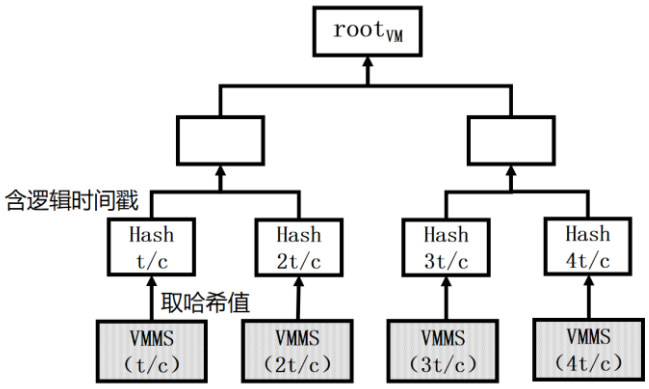



图 2.2 VMMS 的 Merkle 树结构示意

Fig. 2.2 Merkle Tree Structure of VMMS

本文中将该 Merkle 树简称为计算过程树，记作$tree_{VM}$。计算者会将该树的根$root_{VM}$与合约事务计算结果共同签名，然后发布。其他计算者会比较本地计算本地结果与收到

的结果是否相同。相同者，该事务会被接受，不同者则会向全网请求获得完整计算过程树（若拒绝提供，则认定为结果无效），本地会以该过程树的取值时刻为准，将本地虚拟机在这些时刻的状态，同样构造为计算过程树，记录树根为$root'_{VM}$。将$root'_{VM}$与$root_{VM}$向下比对，直到寻找到最小时刻的异常哈希值（如果$root_{VM}$相同而结果不同，说明结果被篡改，则挑战最后时刻的$VMMS$）。该计算者会将异常哈希对应的$VMMS$签名，然后向全网广播，这一过程称为“挑战”。原计算者受到挑战后，必须将受质疑的$VMMS$与其前置$VMMS$一同公布，这一过程称为“响应”。验证者需要执行运算以判断两个计算者，哪一方诚实地进行了两个$VMMS$间的运算，若初始发布者诚实，则结果继续接受下一轮挑战，直到满足等待时间，结果即被验证为真。若挑战者诚实，则结果被验证为假。

通过设置合适的挑战代价、激励以及生效边界，该方案能够有效保证计算结果的可靠性，并且实现验证者不多于$\hbar_c/c$倍的合约事务算力消耗（$\hbar_c$为发生挑战的轮次数，显然$\hbar_c < c$）。与常规可验证计算算法不同的是，该类算法要求有不只一个计算者，通过计算者间的博弈实现结果的可靠性。显然，这一类方法的安全性会受到验证者间的拜占庭错误影响。另外，验证者间的数量、网络状况等也是安全性的重要影响因素。该类算法尚未被发现有明显安全漏洞，其突破与应用已经有案例可循，文章假设交互式验证协议的基础论述与证明是正确的。

### 2.4.3 “挑战-响应”型链下计算协议

在本文中，将使用了交互式验证协议的链下计算协议统称为“挑战-响应”型链下计算协议（Challenge-Response Game Based Off-Chain Computing Protocols）。在不引入可信硬件的前提下，“挑战-响应”型链下计算协议是最成熟的链下计算技术之一[45]。

区块链参与成员大致被分为计算成员与共识成员两类。计算成员被分为不同的组，并行地执行不同合约事务中的计算任务，完成交互式验证协议中计算者的责任。而共识成员只负责验证合约事务结果的合法性，承担交互式验证协议中验证者的职责。

用户将调用合约的事务向 P2P 网络进行传播。有意愿完成合约事务中计算任务的计算成员，会执行合约代码，获得事务结果。计算成员按照交互式验证协议对合约事务的计算结果以及计算过程的$root_{VM}$进行签名，并向区块链网络公开。其他对该合约事务感兴趣的计算成员，会对比己方计算过程树的根$root'_{VM}$与收到的计算过程树根$root_{VM}$之间的一致性，以及计算结果的一致性。若不一致将会向全网广播挑战信息。受到挑战的计算者，必须响应受到的挑战，否则其它参与方将不会承认该结果的有效性。响应结束后，原结果的真实性得以确定，若真实，则接受下一轮挑战。不断接受挑战直到达到生效条件，该交易就会被打包进入区块链。

容易发现，该类协议有两个缺陷，很可能为区块链带来隐患。其一是合约事务的验证具有时效性，这可能直接会威胁区块链可追溯性这一基本性质。合约事务结果被发布

到区块链上，之所以能够被参与者接受，是因为参与者本地都保有了交互式验证的过程记录，即合约事务的有效性是建立在交互式验证过程的合法性上的。

对任何一个参与节点，如果其丢失了交互式验证的记录，那么该节点将无法再对该笔交易进行验证。这不仅对新成员节点不友好，旧有成员节点也不得不承担巨大存储压力。该缺陷可以通过在本地重放丢失交互式验证记录的合约事务来解决，或使用 Arbitrum 提议的基于多方质询的方式在链上推进合约验证。然而，这些方式都会为区块链重新带来计算压力。

另一个问题是链上链下数据一致性的弱化，这可能带来区块链吞吐量瓶颈，并弱化区块链的数据可用性。例如图 5.1 中，合约事务读取了合约状态，直到两区块后才能将其更新到新状态。在这一过程中，早已经在链下承认了最新合约事务的成员会读取到新状态，质疑合约事务的成员则会读取旧状态，即区块链参与者各节点间并不能获取到一致的合约状态。

链上链下数据一致性的弱化会导致同样涉及到该合约状态的其他后续交易无法进入共识阶段，有大量错误交易会被发布，区块链吞吐将受到巨大影响。更大的困扰是，事务一旦发布，发布者就会失去对事务的控制。后续交易一旦被阻塞在共识阶段外，事务的发布者将会有很长一段时间无法确定其资产最终呈现何种状态。这对区块链数据的可用性及数据安全是巨大隐患。

## 2.5 本章小结

本章中对文章的研究前提与面临的问题做出了详细描述，阐述了智能合约、计算密集型合约事务 CICT、区块链 CICT 的多核适配与重放确定性问题、CICT 的验证者困境与常见的链下计算协议等基本概念，下文论述将会基于本章中的技术背景以及基本假设开展。

# 3 面向 CICT 的并行方案分析

## 3.1 引言

传统区块链使用串行的方式进行合约事务的执行，以此实现严格的状态机复制。随着对区块链性能要求的上涨，串行事务执行成为了制约区块链性能提升的重要瓶颈。尤其是在许多复杂场景下，合约事务中包含的计算量通常会严重影响到区块链系统的吞吐。因此，使用并行的方案来处理此类计算密集型合约事务（CICT），是提升区块链性能的重要方案。

然而，CICT 对区块链的并行方式有更高、更独特的要求。首先，在区块链成员设备计算性能不变的情况下，CICT 因为其庞大的计算消耗，每个单笔交易通常都需要一个较长的执行时间。因此，仅仅实现事务间的并行，并不能提供足够的性能提升。CICT 需要更细粒度的并行执行方案来缩减单笔交易的执行时间。其次，区块链共识成员并不具备足够的动力去执行 CICT 中的计算任务，因为其开销远远大于收益，共识成员更倾向跳过 CICT 的验证，这是 CICT 的验证者困境。因此，在共识层面，CICT 的并行存在将执行（包括计算任务）与验证（仅含合法性验证）剥离的需求。

尽管一些学者对区块链事务的并行执行做出过系统的讨论[10]，但其关注点仍然在于普通交易的并行。其中许多能够满足 CICT 的需求，而且通常作用于不同的维度，有望通过方案集成带来更高的性能提升。然而，鲜有研究会对 CICT 的需求进行分析，也缺乏现有并行方案对 CICT 的作用效果与原理的讨论，也就难以使现有方案在 CICT 处理方面发挥作用，区块链性能的突破也就变得更加困难。

本章以多元线性回归问题作为一种 CICT 的典型用例，讨论了 CICT 在并行执行方面的特征与需求。根据两类不同的需求瓶颈，本章提出了针对 CICT 并行策略的两种分类，系统地讨论了诸多区块链领域技术与 CICT 并行执行的关系。本章总结了一些常见的并行执行案例，并将其进行了不同维度的划分，不同维度的并行技术更倾向被集成、组合。通过对 CICT 案例以及常见技术的总结，文章提出了 CICT 并行的两方面缺失，对进一步的工作形成了指导。

## 3.2 CICT 实例

我们以一种最常见的机器学习算法——多元线性回归（MLR）为例，简单分析 CICT 在面临执行时的困境，以及如何使用事务并行的方法提高 CICT 的执行效率。另外，在

本文后续提出的事务并行方案的评估和实验中，多元线性回归问题都将作为典型 CICT 测试用例使用。

此处首先给出多元线性回归问题的简单描述。多元线性回归尝试寻找一个与样本集较为拟合的连续目标函数，并利用该目标函数，进行对新输入样本评价的预测。该章节将会使用到的符号定义如下：

表 3.1 多元线性回归问题相关符号定义表

Tab. 3.1 Definition of Symbols Associated with Multiple Linear Regression

| 符号 | 含义 |
|---|---|
| $x_j^{(i)}$ | 样本 $i$ 的第 $j$ 个属性值 |
| $m$ | 样本总数 |
| $n$ | 样本属性值维度 |
| $\omega_i$ | 第 $i$ 个目标参数 |
| $X^{(i)}$ | 样本 $i$ 的属性向量 |
| $\mathbb{X}$ | 样本属性矩阵 |
| $W$ | 参数向量 |
| $y^{(i)}$ | 样本 $i$ 对应的真实值 |

假设该多元线性回归的目标函数如公式（3.1）：

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \ldots + \theta_n x_n \tag{3.1}$$

假设样本矩阵、样本属性向量与参数矩阵如下：

$$\mathbb{X} = \begin{pmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \ldots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \ldots & x_n^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} & \ldots & x_n^{(3)} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ 1 & x_1^{(m)} & x_2^{(m)} & \ldots & x_n^{(m)} \end{pmatrix} \tag{3.2}$$

$$X^{(i)} = \begin{pmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & \ldots & x_n^{(i)} \end{pmatrix} \tag{3.3}$$

$$W = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \ldots \\ \omega_n \end{pmatrix} \tag{3.4}$$

使用计算残差平方和的方式，我们可以获得其损失函数：

$$J(\omega) = \frac{1}{m}\sum_{i=1}^{m}\left(f\left(x^{(i)}\right) - y^{(i)}\right)^2 \tag{3.5}$$

$$= \frac{1}{m}\sum_{i=1}^{m}(X^{(i)}W - y^{(i)})^2 \tag{3.6}$$

学习率为$\alpha$，使用梯度下降的方法对 $W$ 中的任意参数进行求解，满足以下公式：

$$\omega_j = \omega_j - \alpha \frac{2}{m} \sum_{i=1}^{m} (x_j^{(i)} \omega_j - y^{(i)}) x_j^{(i)} \quad \text{……（3.7）}$$

值得关注的是，我们并不关注线性回归的预测准确度如何，而更关心其智能合约实现在区块链平台上会表现出怎样的吞吐性能。因此，我们将该算法中的有关算力分配与规划的部分单独列出，忽略其他算力消耗较小的步骤，将该过程简化为以下流程：

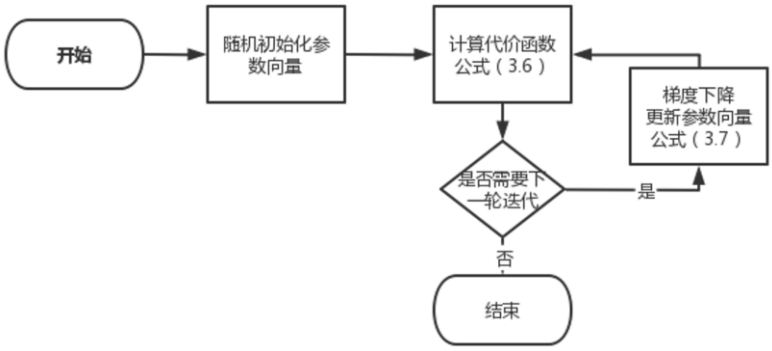



图 3.1 多元线性回归算法流程图

Fig. 3.1 Multiple Linear Regression Algorithm

该算法是机器学习中的一个基础算法，也是一个典型的计算密集型算法。在计算代价函数与梯度下降时，需要进行大量的矩阵乘法运算，通常耗时较长。

## 3.3 CICT 并行执行效率瓶颈分析

### 3.3.1 多核架构适配困难

对 3.2 节中的计算实例分析可知，多元线性回归算法中，有两个过程存在大量的算力消耗，可能会导致该合约执行过程耗时增加。将其描述如下：

（1）每轮迭代中，公式（3.6）要求必然发生不少于$m$次$n$维向量的乘法运算；

（2）每轮迭代中，公式（3.7）要求对$n$个参数，分别包含不少于$m$次$n$维向量的乘法运算。

其中，矩阵$\mathbb{X}$的列数$n$通常由可能对该模型的预测结果有较大影响的样本属性值数量决定，受模型的复杂程度影响，通常不少于几十列。而其行数$m$由样本数量决定，通常远高于矩阵列数。因此，在多元线性回归的一轮迭代中，过程（1）的时间复杂度为$O(mn)$，而过程（2）的时间复杂度为$O(mn^2)$。一轮迭代的时间消耗主要受过程（2）影响，过程（1）影响较小，其他可以忽略不计。

然而，以上两个计算过程，主要涉及的求和运算，可以通过并行编程使其适配多核架构，缩短计算时间。只要使区块链的智能合约能够提供对该并行计算的支持，就能够

通过减少每一轮迭代的执行时间，减少事务的总执行时间。我们给出一种并行执行多元线性回归的智能合约编写方式的流程描述：

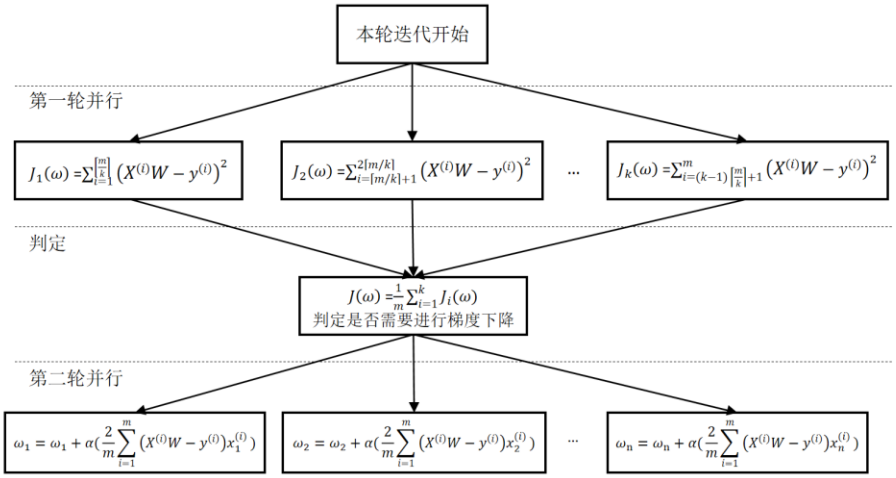



图 3.2 一轮迭代中的并行计算方法示例

Fig. 3.2 Parallel Computing Methods in One Round of Iterations

使用该种方式进行多元线性回归智能合约的编写，在第一轮并行中，将有关公式（3.6）中对$J(\omega)$的计算任务，拆分成了$k$个能够相互并行的子任务，通过并行计算这些子任务，能够有效缩短$J(\omega)$的计算用时。在判定步骤中，需要通过合适的控制手段，对第一轮并行中的计算结果进行同步。在第二轮并行中，将有关公式（3.7）中对向量$WW$的计算任务，拆分成了对其中$n$维属性的并行计算，有效缩减计算用时。

因为以上编写方式，在并行的多线程之间不存在资源竞争的问题，我们将其称为无竞争编写方式。并不是所有 CICT 都能完成无竞争编写方式，为了探讨线程间的资源竞争关系，我们给出了一种多元线性回归的有竞争编写方式：

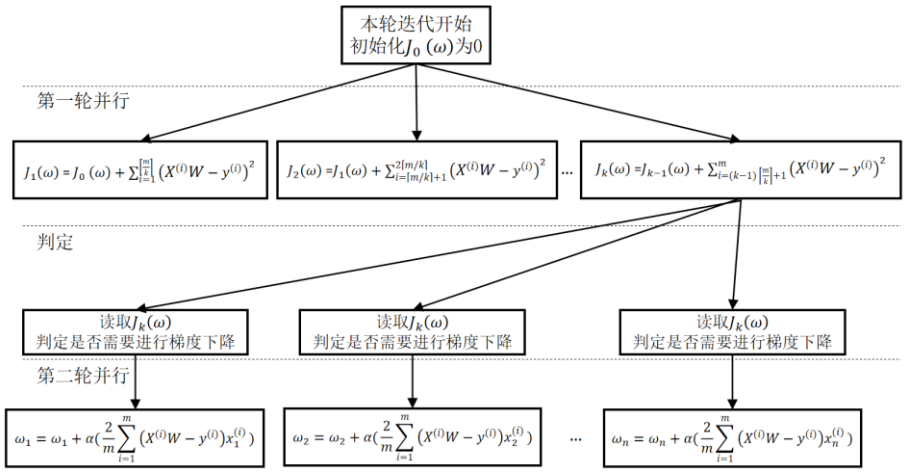



图 3.3 一轮迭代的有竞争合约编写方式

Fig. 3.3 A Contract Programming Approach with Competition

容易发现，在第一轮并行过程中，$J_i(\omega)$必须要读取到$J_{i-1}(\omega)$的值，$J_{i+1}(\omega)$则必须要读到$J_i(\omega)$的值。这只能在合约编写过程中，在代码层面进行保证。这种软性的编程标准并不能提供严格的读写控制。另外，进程间执行时的诸多偶然事件，比如线程堵塞，也可能改变不同线程对资源的访问顺序，这会导致合约结果在不同区块链成员节点上的重放不确定性，为区块链共识带来很大的负担。这些问题会在第四章中被进一步讨论。

仅从合约事务执行效率的角度来说，并行的执行方式，能够有效发挥多核计算设备的优势，实现更高的事务吞吐。

### 3.3.2 计算消耗放大严重

CICT 会对区块链性能产生巨大影响的一个原因，在于 CICT 的算力消耗放大对区块链来说是难以接受的负担。区块链中的任意参与方，必须通过重放相关的交易，才能够获得或执行所需要的数据。这意味着，每一个区块链的参与者，在每一轮区块的产出过程中，都不得不完成其所有前置区块中涉及的交易。假设区块链所有参与者数量为 $p$，那么 CICT 在每一个参与者上的重放，会导致该事务的算力消耗被放大 $p$ 倍。这是对算力和能源的极大浪费。更直观地看，该事务的执行会延误后续工作，比如交易发布、区块打包等。

要抑制 CICT 的算力消耗放大，意味着对一笔 CICT 的验证，需要由更少的节点完成其计算内容的执行。我们以链下计算为例，描述其基本原理。假设需要执行全部计算任务的参与节点为计算成员，而全部需要对事务结果进行验证的参与节点，即所有共识节点，为共识成员，链下计算可以被描述如下：

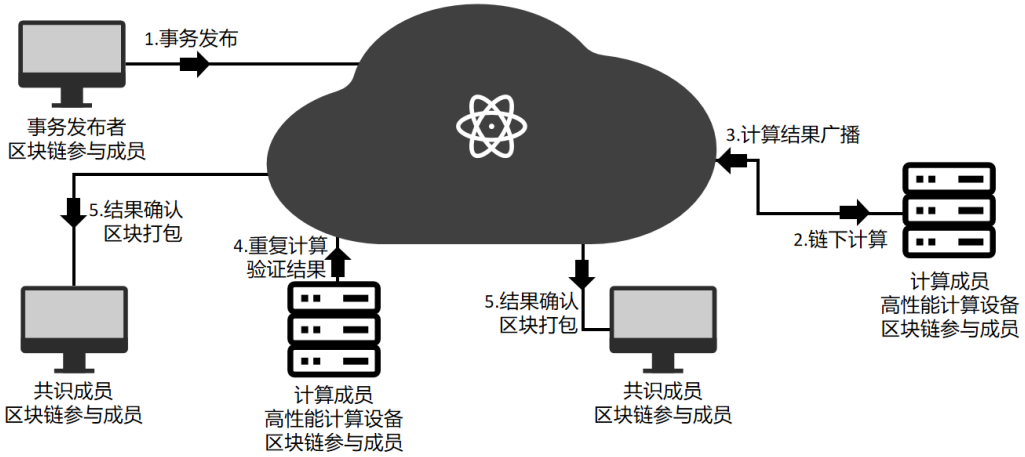



图 3.4 链下计算流程示意

Fig. 3.4 Off-Chain Computing Process

尽管链下计算协议种类众多，具体的工作流程、计算方法、验证规则都有不同，但其工作流程具有极大的相似性。事务的发布者在将该事务发布后，计算工作，将会交给具有极强计算能力的部分成员完成，成员间也能够互不干扰地同时进行多个事务的执行。

假设计算成员数量为$p'$，显然，一个 CICT 的算力消耗，只会被放大$p'$倍，具体数值主要受区块链网络安全性要求影响，更多的计算者意味着区块链网络更高的安全性，但很显然，$p' << p$，算力消耗放大倍数将会明显下降。针对共识成员而言，其不再必须承担 CICT 的计算任务，对设备要求的压力会有效减缓，也不会妨碍节点的其他必要任务，避免了区块链阻塞。

减小事务算力消耗的放大，实际上是事务并行方案的一类重要组成。包括分片、私有合约、XO 验证模型，都是通过减小事务算力消耗的放大，降低成员计算压力，从而实现事务的并行。对 CICT 来说，因为本身算力消耗的基数较大，所以减小算力放大倍数显得尤为重要。

## 3.4 现有 CICT 并行执行方案分析

事务并行能够减少 CICT 的验证时间[46]，并提高区块链系统的吞吐量。通过以上分析发现，文章将现有并行方案大致分为两类：

（1）通过使 CICT 的计算适应多核计算架构，实现事务的并行执行。

（2）抑制 CICT 算力消耗在区块链网络中的放大，使不同成员能够独立执行不同事务，实现事务的并行。

文章中将主要通过第一种方式提高区块链性能的并行方案统称为多核适配类并行策略。该方案主要侧重提高事务验证模式对多核设备计算模式的适配，挖掘设备计算效率，以提高单一节点内事务验证的效率。与普通区块链事务相比，CICT 对该类并行方案的需求会明显更大。

文章中将主要通过第二种方式提高区块链性能的并行方案统称为计算抑制类并行策略。该类方案主要标志通常是进行区块链参与成员间的分工、分组来细化账本与计算任务的分布，使区块链事务不必在每一个成员上重放，相应的算力消耗也只发生在部分成员身上，算力消耗的放大倍数大大下降。

另外，为了进一步探讨并行方案间的关系，文章按并行方案作用维度的不同，进行了方案的进一步分类。如图 3.6 所示，大致分为五个维度。对应图中 1 至 5，分别为（1）多链间事务并行；（2）分片间事务并行；（3）分片内事务并行；（4）节点内事务间并行；（5）单笔事务并行执行。不同维度的方案更容易进行组合与叠加，通常会带来优于单一方案的效率提升。同一维度的并行方案间常常是互斥的，难以同时采用多种方案，然而，这一规律并不绝对。

节点内并行、单笔事务并行执行通常是多核适配类并行，多链间并行、分片间并行、分片内并行，大多是计算抑制类并行。然而，许多分片技术同样采用了多核适配类并行的设计思路。并行维度与并行分类之间并不具备严格对应关系，具体方案分类仍然需要

具体分析。而单笔事务的并行执行则尚少有成熟方案，因为除 CICT 外，其对简单转账事务鲜有意义。下文会进一步讨论其与 CICT 的关系。

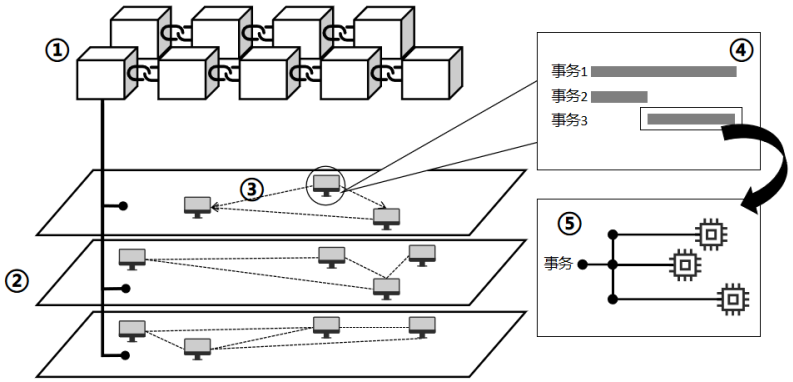



图 3.5 区块链多维并行架构示意

Fig. 3.5 Blockchain Multi-Dimensional Parallel Architecture

### 3.4.1 多核适配类方案

（1）节点内的事务间并行

节点内的事务并行方案无一例外地属于多核适配类方案。比较有代表性的有静态分析、动态分析方法，动态分析方法中又包括锁方法、多版本并发控制（MVCC）以及软件事务内存（STM）等解决方案。

节点内并行的关键是寻找到一种与顺序执行会产生同样结果的并行执行序列，通过执行新的并行序列，避免事务冲突，实现并行。其中，静态分析策略是通过对智能合约源代码及其辅助性资料进行分析的策略，通过建立交易有关资源集，在执行前就寻找到一种事务的等价序列，然后按照本地 CPU 数量，寻找较优的执行序列，避免发生资源冲突或者事务矛盾。这是一种对资源密集型合约事务的常规并行方案，但可行性极低。仅仅通过合约事务执行前分析，是难以建立准确的资源集的。动态分析方案正是对这一缺陷的补充。其中最具代表性的是锁方法。锁方法是传统事务系统处理事务并行中数据资源竞争的方法，是一种悲观的并发控制方法。通过对事务进行大规模并行，记录所有可能产生读写锁的位置。任意两个事务，如果访问了同一个内存，那么他们就被认为是可能产生冲突的，会在下一阶段被重新串行执行。这是对合约事务最朴素的动态并行办法。多版本并发控制则进一步实现了不加锁的非阻塞并发读。因为该方案能够保留过程中的全部数据版本，所以可以找到事务间读取的数据版本的变更过程，大量减少事务重放的数量。软件事务内存则将一组访问了相同内存的事务包装成了一个原子性事务，避免了读写冲突，也避免了其他方案复杂度过高，实际应用过程鲁棒性差等问题。但实际上，在组装原子性事务的过程中，也不可避免地处理事务冲突和重放的问题。该方案极

适合事务数据资源竞争不激烈的情况。面对较为激烈的资源竞争，STM 必然会存在大规模的事务重放，事务处理效率的提升未必可观。

综合来说，现在的节点内并行方案，技术较为全面，这得益于传统中心化数据库在事务管理方面的积累[47]。节点内并行，与其他维度的并行，通常也能够进行组合或者叠加，进一步提升区块链系统效率。

（2）单笔事务的并行执行

区块链领域对单笔事务的并行执行的研究几乎还是空白。这主要是因为区块链系统前期的业务类型大多以简单事务为主，比如转账。对于此类计算资源消耗并不高的事务来说，进行单笔事务的并行执行，反而会白白导致过高的计算任务调度开销。但对 CICT 来说，其处理时间远远长于简单交易，这可能会导致整个区块链系统产生阻塞，面临系统性能瓶颈。这种情况下，对合约事务内计算任务进行并行编程，充分发挥设备的多核特性，缩短 CICT 的验证时长，是在区块链上提供 CICT 支持的关键。

但区块链系统对智能合约并行编程的支持并非不可预见，许多区块链系统都在推广使用常见高级语言来进行智能合约编程，比如 GO 语言，或者 JAVA。这些语言通常是支持并行编程的。GO 语言因为其能够在语言层面实现并行脱颖而出，包括 goroutine 以及 channel 等设计，本身就是为了并行计算而诞生。JAVA 作为用户群体最庞大的语言之一，其并行编程组件也越来越成熟。在将来事务变得更加复杂，算力消耗更高时，这些使用高级语言进行智能合约编程的区块链，可能会最早进入智能合约在多核适配下的并行执行时代。

然而，区块链环境与其他并行编程环境存在巨大区别。区块链系统要求其事务在任何一个参与成员节点上，在任何时刻的重放，都必然会得到完全相同的结果，这是区块链系统进行账本同步的本质要求。而并行编程中因为存在资源竞争等偶发情况，可能会存在不同的竞争顺序，从而造成不同的最终结果。因此，普通并行编程并不能严格地保证其重放时结果的确定性。

另外，同样有部分研究已经开始尝试允许合约进行嵌套调用，并在嵌套时允许并行调用，以实现单笔事务的并行执行[15]。可惜的是，这会为区块链带来巨大的潜在危险。比如 Dao 攻击，就因为跨合约调用的安全漏洞，造成了超过 5000 万美元的损失[48]。Fisco 等项目，已经明确可并行的合约必须满足无调用外部合约，且不调用其他函数接口等要求。目前来看，单笔事务的并行执行，暂时要避免通过合约嵌套结构实现。

因此，本文提出了支持重放确定性的智能合约框架，解决了并行结果不确定性的问题，以支持只能合约的单笔事物并行执行。相关方案可见于第四章。

（3）其他并行方案

分片方案中的 Meepo[49]中提到的影子分片（shadow shards）就是一种多核适配类并行的分片方案。通过将完整账本分割为多个分片，实现存储以及计算的切分，却在每个成员节点中保存所有的分片，来维持账本的一致和跨账本事务的高效。

另外还有区块链领域的异类，DAG 技术。其主张直接抛弃区块链的链状账本结构，采用有向无环图结构组织账本[14]。比较有代表性的项目有 byteball、IOTA[50]。该类方案的典型思路，是将任何一笔交易打包为区块，新区块被传播给其他参与方之后，其他参与方将其在账本上记录，若选择验证，则可以在其后进行延长，若不验证或验证不通过，该区块仍会被记录在 DAG 中，但不会再被延长，从而失去活力。任何区块的后方都可以添加延长，任何一个区块都可以被添加任何数量的延长，新区块需要同时延长许多个已被验证的区块。严格来说，DAG 不属于计算抑制类并行方案，因为 DAG 性能的提升是通过降低共识安全性实现的。如果要保障足够的安全性，每一笔交易同样需要被所有节点验证通过，算力消耗放大并不会减弱。但 DAG 的一大优势是，可以在吞吐高峰通过同时产出多个分叉，满足吞吐需求，然后在将来的空闲时间逐步处理这些分叉，该过程当然不可避免地存在安全隐患。DAG 本身也不要求必须实现严格的多核适配，但他的结构确实更适配多核设备的计算方式，通过编程手段的控制，DAG 可以轻松实现事务的并行执行。但 DAG 并不是解决 CICT 的好方案。因为普通参与者，缺少在 CICT 上进行验证、延长的动机，CICT 有更大的可能成为失活的事务。

### 3.4.2 计算抑制类方案

CICT 会对区块链性能产生巨大影响的一个原因，在于 CICT 的算力消耗放大对区块链来说是难以接受的负担。

（1）多链间事务并行

多链并行是进行事务并行，提高事务吞吐的重要方案[51]。区块链间的信息孤岛一旦打破，区块链间便可以做到信息共享，那么就可以为不同场景、不同联盟定制不同区块链。区块链吞吐不足的问题就可以通过横向扩展区块链数量来解决。

多链技术的使用和研究尚显贫乏[52]。在多链协作方面，受到更多关注的是跨链技术，其实现方案大致有三个思路：公证人机制，通过选举一个或者多个公证人对链上的数据进行共识，然后注入另一条区块链；侧链/中继，通过将一条区块链的区块头锚定到另一条链，实现数据的跨链验证；哈希锁定，通过秘密数的传递与交易规则，实现两条链上秘密数的同时揭秘，保障两条链间多个子交易构成的跨链交易，能够实现原子性。

使用了多链或跨链技术的项目不在少数，比如 Interledger、BTCRelay、Cosmos、RootStock 等，其最终目的是打通所有区块链间的数据共享。所以通常来说，多链维度的并行方案容易与其他维度的方案产生性能的叠加，对 CICT 亦是如此。

（2）分片间事务并行

分片间并行是指区块链传统意义上的分片（sharding）技术，也是区块链并行实现方案最为多样的一个维度。常规意义上的分片，指区块链为了提高性能，将一个完整账本切分为几个部分，交由不同的成员节点进行维护，从而降低单个成员节点的负载[53]。因此，账本如何划分、成员如何分配、如何处理跨分片交易等诸多方面，都有不同的方案，组合成了多样的分片策略[54]。包括 Fabric 中使用的通道（channel）策略就是分片方案的一种。通道为对某一类业务更为感兴趣的成员节点创建了独立的交流群组。该部分成员只需要维护本通道内感兴趣的事务的发表、验证、共识、提交等，而不必关注全网其他通道中的事务。Monoxide[23]是一种比较创新的分片方案，使用异步共识组（Asynchronous Consensus Zones）进行账本的分割，使用连弩挖矿将单个共识组（可以类比于分片）上的挖矿算力放大到其他共识组，从而抑制节点作恶。实际上这仍属于计算抑制类并行，因为放大的是挖矿算力，事务并不会跨共识组执行，执行的计算消耗也就不会被放大到全网。

对 CICT 来说，分片间维度的并行策略也容易与其他维度的并行策略进行组合。

（3）分片内事务并行

Fabric 是采用了分片内并行方案的代表。Fabric 使用的事务验证模型是 XO（Execute-Order）模型[16]，即先执行，再排序，最后提交。事务会由客户端发送给一定数量的背书节点，背书节点需要执行事务，并将事务结果签名，即提供背书，然后返回给客户端。客户端收集到足够的背书后，会将该事务发送给排序节点。排序节点会将事务重新排序、打包，然后将区块发布。发布后的区块通过共识，被其他所有参与成员同步，此时区块上的事务被发布成功。这一过程中，为事务提供计算服务的只有背书节点，事务算力消耗的放大倍数由全部参与节点，缩小到背书节点数量以内，这是计算抑制类并行的典型特征。该方案首先被 Fabric 发表后，被应用到了诸多许可链中。面对无许可环境，该种背书方案不再奏效。在处理计算密集性事务时，通常仍需要考虑背书节点可能不具备承担高算力消耗的动机，这需要另外的激励策略改进[37]。

常常被许可链使用的分片内并行方案，还有私有合约。在该种方案设计中，合约不再作为完全公开的数据资产，而是完全私有资产，只有被许可授权的参与成员才能够获取合约内容。通常，合约事务被验证时，也就只有被授权的合约参与方能够进行合约的验证，合约事务验证的算力消耗放大倍数，自然也等同于少数的几个被授权节点的数量。该种方案能够实现对 CICT 验证的激励，因为所有被授权方都是合约的相关利益方，他们乐于去验证合约。但该方案的应用场景受限严重。

相比之下，链下计算方案目前是一种较优的计算抑制类 CICT 并行方案。以交互式验证协议为例，CICT 在发布后，会由专业的高性能计算设备组完成计算[55]，普通参与成员只需要验证计算结果的合法性即可，付出的计算开销[56]极少，同时区块链的激励机制不会因此破坏。详见第五章。

## 3.5 现有 CICT 并行执行方案局限性

在多核适配类并行方案中，节点内的事务并行策略大多来自传统数据库事务系统，已经渐趋成熟。但如何实现单笔事物的并行执行，对减少 CICT 执行时间来说仍是极有必要的。此类研究仍显匮乏，并且并行合约编程的重放不确定性仍未解决。虽然现在已有部分重放确定性保证方案可供参考，但此类研究从未尝试在区块链这一特殊环境下进行讨论。尝试合约嵌套的并行方案，也都遭遇了合约调用方面的安全困境。因此，本文提出了 CICT 的支持重放确定性的并行策略，以解决单笔事务并行执行方面存在的重放不确定性问题，

在计算抑制类并行方案中，跨链维度、分片间维度的并行策略，大多同样能够对 CICT 有效，并且容易与其他维度的并行方法叠加。而 CICT 的分片内并行策略，通常存在着激励困难的问题，验证者趋向逃避验证 CICT，以获取更大收益。这是因为 CICT 庞大的验证开销，加重了验证者困境。比较有效的解决方案，是使用链下计算策略，降低事务再次验证的开销，实现链下的事务并行执行，降低算力消耗放大倍数，实现更大的吞吐。但链下计算策略所依赖的交互式验证协议同样存在链上链下数据一致性下降等诸多问题。因此，本文提出了 CICT 的链上挑战型链下计算协议，以在抑制计算消耗放大的同时，解决分片内并行策略中存在的激励困难、链上链下数据不一致等问题。

## 3.6 本章小结

本章中对 CICT 的实例以及需求进行了举例与分析，提出了面向 CICT 的两种并行策略的分类，并对现有区块链事务并行方案做出了讨论，进一步在多个维度对 CICT 所面临的困境进行了分析。提出 CICT 并行的重要目标是解决事务内并行的重放不确定性问题，以及改进链下计算协议，解决验证者困境。另外，本章给出了本文中用来进行实验测试使用的标准 CICT 案例，即多元线性回归问题。并且分析了多元线性回归问题在单笔事物并行执行、分片内并行两个维度的处境与问题。

# 4 基于重放确定性合约的 CICT 多核适配类并行策略

## 4.1 引言

随着区块链技术的成熟，越来越多的行业正在尝试使用该技术实现信任传递。作为复杂行为的载体，智能合约正面临着越来越复杂的计算需求。据调查，以太坊上一些复杂的智能合约事务验证时间已经超过了 20s[13]。特别地，当区块链技术逐渐与人工智能领域发生交叉时，该类问题将会尤为突出。例如，使用深度神经网络（RNN）技术解决去中心化金融（DeFi）中存在的货币异常交易问题[57]，或者是使用生成式对抗网络（GAN）解决非同质化代币（NFT）铸币问题，都意味着区块链将无法避免的遭遇处理此类计算密集型合约事务（CICT）的需求[58]。

然而，当前区块链并不能适应对 CICT 的处理。其原因在于对 CICT 过长的验证时间，会大大限制区块链的吞吐，导致区块链性能瓶颈的出现。因为执行单笔 CICT 事务花费的时间过长，区块链的共识将极为缓慢，并且该种缓慢无法通过事务间的并行被进一步改善。因此，研究面向 CICT 的单笔事务并行执行策略，在区块链平台提供智能合约的并行编程支持，有助于发挥计算设备的多核特性，有效降低事务验证时间。

CICT 的验证，接近于完成一次较大型计算任务。本文通过设计支持重放确定性的多核适配智能合约模型（Multicore Replay Deterministic Smart Contracts, MRDSC），实现将 CICT 拆分为多条代码段，通过代码段之间的并行计算缩短事务执行时间。针对该种方案可能带来的资源争用以及事务重放不确定性问题，文章在 MRDSC 虚拟机的读写层拦截代码段执行结果，并通过资源读写集与有向无环图的方法，对代码段进行重序列化，寻找发生资源冲突的代码段并进行重放，实现并行合约事务的重放确定性。并且克服了静态分析类方法在冲突检测方面粒度低、效率慢的问题[59]。借鉴了传统动态重放类技术，在其基本思想上，本文实现了细粒度的单笔事务并行执行。另外，文章使用了缓冲池等方法，解决了动态重放类技术在面对大规模资源争用的情况下的过量回滚与重放问题[60]。

## 4.2 多核适配的智能合约模型

智能合约并行编程的确定性与事务系统的确定性并行有很大相似性。其中不得不提的是软件事务内存（Software Transactional Memory, STM）[61]技术。STM 是一类常见的事务确定性并行方案，其执行流程如图 4.1。STM 类技术首先会将事务进行大规模并行，然后在重序列化阶段对并行过程中可能发生的读写、写写冲突，进行检测与纠正，实现事务的重序列化。最后，通过重放存在冲突的事务，在不使用锁的情况下，保证一次包

含并行计算的完整执行能够在任何一个区块链参与节点上获得相同的结果，维护区块链共识的稳定性与效率。


图 4.1 STM 类技术的事务并行执行流程示意

Fig. 4.1 Concurrent Transaction Execution Flow for STM-like Technologies

简单来说，这一理念与事务系统中，尤其是数据库系统中，已经尝试过的“投机性提交”是相似的。投机性提交不使用锁或其他技术进行事务顺序控制，仅在提交失败时进行事务的冲突处理[62]。其会为每一个事务使用一个单独的线程进行运算，并在最后尝试提交。如果提交成功，则事务的结果生效；否则，将尝试解决计算中可能存在的读写冲突并重放。不同的是，STM 类技术不允许事务单独向持久层提交，而是将所有结果进行拦截，在冲突纠正、重放结束后，将所有事务进行统一提交。另外，STM 类技术在事务重放时不止要求重放成功，而是严格的要求了事务应该发生的顺序，事务必须在正确的环境下被重放。

因此，相比于投机性提交，STM 类技术有更明确的使用前提：

（1）每一个事务是原子的。其内容只能存在全部生效与全部失效两种状态，不存在中间状态。

（2）存在唯一的输入序列。事务原本应该以一个确定的已知顺序执行。

然而，不管是投机性提交，还是 STM 类技术，通常发生在整个事务系统内，在事务层面进行调度的并行技术[63]。为了实现更低层面的并行，即单笔事务的并行执行，文章提出了 MRDSC 模型。MRDSC 合约事务的一次完整执行分为三个阶段，分别为预执行阶段，重序列化阶段，重放阶段。该过程如图所示：

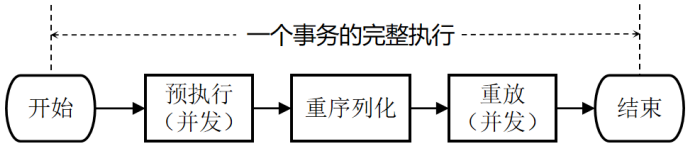



图 4.2 MRDSC 事务处理流程示意

Fig. 4.2 MRDSC Transaction Processing Flow

MRDSC 定义了计算任务的新概念，作为任务调度层的最小单元。简单来说，MRDSC 会将一个 CICT 分为多个更小的计算任务，并在预执行阶段使用多个线程来处理这些计算任务。在重序列化阶段，为了在区块链合约并行编程中实现确定性保证，借鉴了投机性提交与 STM 技术的一些特点。MRDSC 会尝试统一提交每一个任务的运算结果。如果

提交成功，则每一个小计算任务的结果组成最终结果；否则，将会为其中提交失败的任务重新寻找正确的执行环境，然后重放，最后达成事务的最终结果。

因此，文章进一步提出了 MRDSC 模型的作用前提：

（1）每个计算任务是原子的。其内容只存在全部生效与全部失效两种状态，不存在中间状态。

（2）MRDSC 合约事务是原子的。其包含的计算任务只存在全部提交与全部失效两种状态，不存在中间状态。

（3）计算任务存在一个唯一的输入序列，规定了任务生效的基本顺序。

在以上前提的约束下，我们在以下内容中给出了 MRDSC 模型的一种设计。我们使用“计算任务”取代“事务”，作为计算的最小单元。文章设计了“持久层拦截层”用以拦截计算任务在持久层上的行为，并通过“重序列化层”进行有序重放，并进一步保证事务总体的原子性。另外，文章在 4.2.3 小节中阐述了计算任务初始序列，规定了计算任务间的基本逻辑序。文章还使用了读写集等新型数据结构，并设计了缓冲池，以解决 STM 技术中因频繁资源征用导致的大规模重放问题。通过添加读写集、缓冲池与通过基于有向无环图的重序列化方案，我们大规模减少了 STM 中存在的重放问题，进一步提高了并行效率。

### 4.2.1 模型组成

MRDSC 模型，将合约代码以代码块的形式进行组织，一个代码块由不少于一句合约代码组成，所有代码块组成完整的智能合约。每一个代码块可以被视为一个计算任务。代码块被放入新的结构——zone 中，zone 指计算空间，是一种辅助合约进行计算任务模块化的结构。zone 之间可以进行多层嵌套，每一个 zone 可以由不少于一个的子 zone 组成，子 zone 之间有严格的顺序。根据功能与解析方法不同，zone 总共可以分为三类。

s-zone：串行计算空间（Serial Code Zone），其内容会被串行执行。内部可以包含并且只能包含代码或者下一级 zone 中的一种。工作线程会逐行解析并执行 s-zone 中的内容。s-zone 中的代码对所属工作线程具有独占性。

p-zone：并行计算空间（Parallel Code Zone），其内容会被并行执行。内部可以包含并且只能包含业务代码或者下一级 zone 中的一种。工作线程在处理一个新的 p-zone 时，会创建一个新的子线程，工作线程不会因此阻塞，而是直接解析并处理下一个 zone，子线程完成对会逐行解析并执行 p-zone 中的内容。

a-zone：等待同步计算空间（Awaiting Zone），内部不能包含任何代码或其他 zone。a-zone 用来控制并行代码块间严格先后关系。该代码块标识需要等待已有所有计算任务结束后，才会开启新的计算任务。a-zone 所在线程会在遇到 a-zone 时进入休眠，直到所在线程的子线程全部结束。

下文中给出一个简易的合约代码实例：

| 智能合约示例 | |
|---|---|
| 1: | **FUNCTION CONSTRUCTOR:** |
| 2: | ... //构造器代码内容 |
| 3: | **FUNCTION ITERATIONFORMLR ( paramMatrix float[m][n] ):** |
| 4: | S-ZONE: |
| 5: | P-ZONE: ... //计算 $J_1(\omega)=\frac{1}{m}\sum_{i=1}^{\lfloor \frac{m}{k} \rfloor}\left(f\left(x^{(i)}\right)-y^{(i)}\right)^2$ |
| 6: | P-ZONE: ... //计算 $J_2(\omega)=\frac{1}{m}\sum_{i=\lfloor \frac{m}{k} \rfloor+1}^{m}\left(f\left(x^{(i)}\right)-y^{(i)}\right)^2$ |
| 7: | A-ZONE: |
| 8: | ... //读取$J(\omega)$，计算$J(\omega)=J_1(\omega)+J_2(\omega)$，判定是否继续 |
| 9: | S-ZONE: |
| 10： | P-ZONE: ... //计算并更新$\omega_1$ |
| 11： | P-ZONE: ... //计算并更新$\omega_2$ |

图 4.3 MRDSC 代码结构示意

Fig. 4.3 MRDSC Contract Structure

上图给出了多元线性回归的 MRDSC 的示例，例中省去了计算过程的大部分代码，主要保留了合约结构。构造器 CONSTRUCTOR 会在合约被部署时调用，常被使用于初始化并记录合约相关数据或参数，该部分数据与合约代码会被记录在合约专用的独立存储空间中，等待使用。智能合约方法 ITERATIONFORMLR 描述了多元线性回归的一次迭代过程，会在被事务调用时执行，通过传入的样本矩阵，该方法会更新合约中记录的$J(\omega)$、$\omega_1$、$\omega_2$等数据。方法 ITERATIONFORMLR 在被调用时会创建一个主进程，该进程在执行行进到第四行时会遭遇一个串行代码块，从而开始处理该代码块中的内容。在遭遇第五行 p-zone 后，会创建一个子进程，该子进程会被加入当前进程的子进程池中。子进程中的内容不会对父进程造成影响，父进程会继续进行第六行的内容，直到遭遇 a-zone。a-zone 会使得主线程进入休眠，直到其子进程池中被清空，这一过程在下一节任务调度中会被详细介绍。在第十行与第十一行，工作进程同样会创建新的子进程，不同的是，工作进程没有再设置等待同步的代码块标记。工作进程在第十一行完成，但在等待子进程结束后才会结束进程。

MRDSC 与传统并行编程在编写上沿用了相似的习惯，但是在更底层的执行与持久化方面有极大的区别。负责处理 MRDSC 的虚拟机系统，可以按照功能分为以下几个模块：

解析层：一个新的合约事务到达时，是一组待解析的交易字符串。除交易合法性验证的必要数据，比如付款账号、签名等，参与合约调用的有合约地址、方法名、参数等。解析层会解析交易请求，调用合约代码，解析参数，然后传递至任务调度层进行执行。

任务调度层：MRDSC 会严格控制内部计算任务的顺序关系。该顺序关系会借助 zone 结构进行表达，zone 结构的解析会在该层进行，线程池的管理也在该层进行。

计算层：任务调度层对代码块的执行顺序做出决策。需要执行的代码块会交由该层进行。

持久层拦截层：任何代码块的执行结果，都会被该层拦截，不会直接在数据库中进行持久化。这是因为此时的计算结果仍不能严格的保证没有因为资源争用而导致错误的并行结果。

缓冲池：被持久层拦截层所拦截的计算结果，会暂时存入缓冲池中。缓冲池的根本目的并不是存储拦截结果，因为拦截结果可以暂时在拦截层中存放。根本上，缓冲池是为了解决大规模资源争用时，可能出现的大规模计算重放问题。详见下一节。

持久层：持久化数据库。合约事务的运算结果需要被保存到数据库中，以实现智能合约状态的持久化。新一轮合约事务的计算，需要在持久化数据库中读入上一轮计算的结果。

重序列化层：所有代码段的计算结果都会在此处进行检查，保证严格的顺序正确。这是并行结果确定性的核心保障模块。

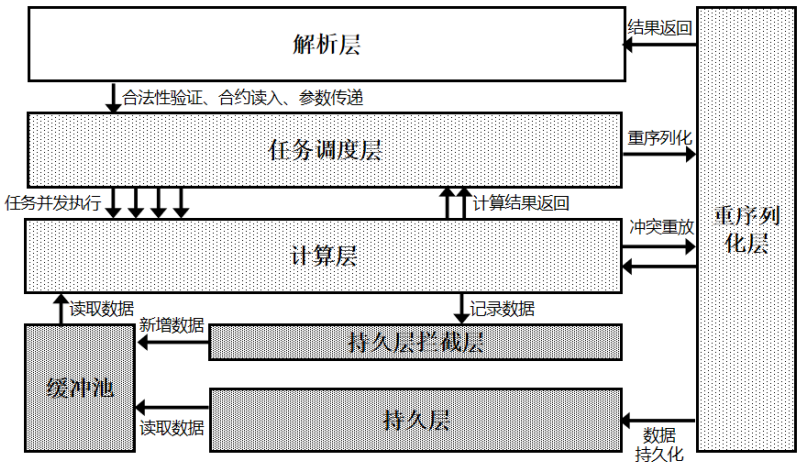



图 4.4 MRDSC 虚拟机架构示意

Fig. 4.4 MRDSC Virtual Machine Architecture

值得注意的是，在该架构设计中，持久层不会与计算层有任何直接数据交互。持久层中保管的合约数据，会被读取进入缓冲池，计算层需要使用的所有数据都在缓冲池中获得。计算层产生的新数据，不会在数据库中被持久化，而是被持久层拦截层拦截，存放进入缓冲池。新数据并不会替换掉缓冲池中的旧数据，而是与旧数据共存。持久层在重序列化层中保留了唯一的写入口。重序列化层的工作顺序为“重序列化-冲突重放-数据持久化-结果返回”，只有经过重序列化后的结果才会在持久层中被持久化。

### 4.2.2 计算任务调度

下图给出了一个 MRDSC 按照计算任务与计算空间的包含关系展开的示例图：

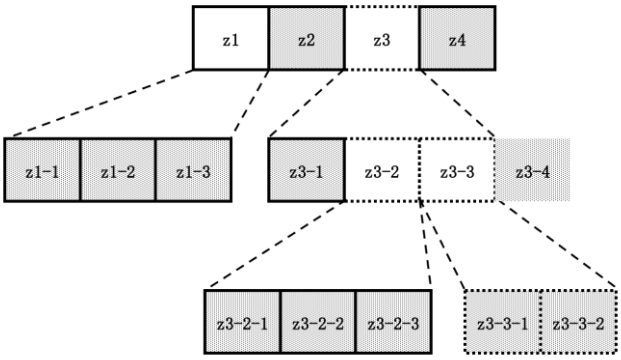



图 4.5 计算任务与计算空间结构图示例

Fig. 4.5 Structure for Computational Tasks and Computational Space

在上图所示的案例中，每一个方形图案表示一个计算空间 zone，方形图案间的虚线表示了父子 zone 间的包含关系。s-zone 使用实线边框标识，p-zone 使用虚线边框标识，a-zone 使用无边框标识。zone 内容有仅包含子 zone 或仅包含代码块两种，其中使用无图案填充标识仅包含子 zone 的 zone，使用阴影标识仅包含代码的 zone。容易发现，只需要将整个完整的合约方法视为一个 s-zone，最上层的计算空间，比如图中的 z1、z2、z3、z4，便是该 s-zone 的子 zone。将父 zone 与子 zone 进行连线，计算任务与计算空间的嵌套结构可以转换为一个树状结构，将其称作等价树结构。这是一个重要结论，使 MRDSC 能够借用树的一些特性来解决部分逻辑序与任务调度的问题。

显然，计算任务只会被包含在仅含代码的 zone（即阴影 zone）中，该类 zone 可能位于等价树的不同深度上，但一定不包含子 zone，将其称作叶子计算空间。可知，叶子计算空间与计算任务能够构成一一对应的关系。叶子计算空间也就成为了任务调度层的调度对象，即计算的最小单元。非叶子计算空间则不会指代任何计算任务。

a-zone 是一个例外，a-zone 用来提示任务调度层等待旧任务对资源的更新完成，以在新任务中使用新资源值，但其本身不会对任何资源发生改写，因此可以将其视作一种进行定时休眠的计算任务。

在 MRDSC 被执行时，计算任务会按照如上图所示的 zone 间的包含关系被串行或并行执行。这是计算任务的调度过程，该过程采用了以栈为基础的任务调度策略。文章给出了几种任务调度的详细过程，如下：

$$\frac{\{task=Read(tar),zone_S^{tar}\rightarrow S(zone^{tar-1})\}}{\left\{\begin{array}{c}(S(zone^{tar-1}),task)\\ =[\![HandleSzoneLeaf]\!]\Rightarrow(S(zone_S^{tar}),Excute(task))\end{array}\right\}} \tag{4.1}$$

公式（4.1）描述了处理串行计算空间叶子空间时的任务调度过程。公式分为上下两部分，使用分割线隔开，下部分包括调度前状态，调度过程操作代码名$[\![HandleSzoneLeaf]\!]$，以及调度后状态。上部分记录了初始化过程，外来输入，以及操作过程等。公式中，使用$S(zone^{tar-1})$表示当前工作栈，$zone^{tar-1}$为栈顶元素，使用$tar$表示当前待处理的计算空间在等价树上的深度，$tar-1$为深度更低一级的父空间。操作

过程中，新计算任务（$task$）被读入，待处理串行计算空间（$zone_s^{tar}$）会放入当前工作栈（→表示传出，←表示传出）。在调度后状态中，新的栈顶元素为$zone_s^{tar}$，新读入任务将被执行。按照同样的方式，我们给出剩下的几种调度操作。

$$\frac{\{zone_s^{tar+1}=Read(tar)\ ,zone_s^{tar}\rightarrow S(zone^{tar-1})\}}{\left\{\begin{matrix}(S(zone_s^{tar}),zone_s^{tar+1})\\=[\![HandleSzone]\!]\Rightarrow(zone_s^{targ+1}\rightarrow S(zone_s^{tar}),Read(tar+1))\end{matrix}\right\}} \tag{4.2}$$

$$\frac{\{'zone^{tar}=Read(tar-1)\ ,zone_s^{tar}\leftarrow S(zone^{tar-1})\}}{\left\{\begin{matrix}(S(zone^{tar-1}),'zone_s^{tar})\\=[\![TurnBackSzone]\!]\Rightarrow('zone_s^{tar}\rightarrow S(zone^{tar-1}),Read(tar))\end{matrix}\right\}} \tag{4.3}$$

$$\frac{\{task=Read(tar),T'(null),task\rightarrow T',Pool\}}{\left\{\begin{matrix}(S(zone^{tar-1}),Pool,task)\\=[\![HandlePzoneLeaf]\!]\Rightarrow(S(zone^{tar-1}),T'\rightarrow Pool,'zone^{tar}=Read(tar-1))\end{matrix}\right\}} \tag{4.4}$$

$$\frac{\{zone_p^{tar}=Read(tar-1),zone_p^{tar}\rightarrow S'(null),S'(zone_p^{tar})\rightarrow T'(null)\}}{\left\{\begin{matrix}(S(zone^{tar-1}),Pool,zone_p^{tar})\\=[\![HandlePzone]\!]\Rightarrow(S(zone^{tar-1}),T'(S')\rightarrow Pool(\ ),Read(tar-1))\end{matrix}\right\}} \tag{4.5}$$

$$\frac{\{Read(tar-1)==null,Pool(null)\}}{\{(S(zone_p^{tar}),Pool''(T))=[\![TurnBackPzone]\!]\Rightarrow(zone_p^{tar}\leftarrow S(null),T\leftarrow Pool'')\}} \tag{4.6}$$

$$\frac{\{Sleep(T)\ \}}{\{(zone_A^{tar}\rightarrow S(zone^{tar-1}))=[\![HandleAzone]\!]\Rightarrow(S(zone_A^{tar}))\}} \tag{4.7}$$

$$\frac{\{Pool(null),'zone^{tar}=Read(tar-1)\}}{\{(S(zone_a^{tar}))=[\![TurnbackAzone]\!]\Rightarrow(zone_a^{tar}\leftarrow S(zone^{tar-1}))\}} \tag{4.8}$$

公式（4.1）至（4.8）系列公式描述了在 s-zone、p-zone、a-zone 处理过程中的任务调度过程，包括栈、线程、子线程、线程池之间的关系。公式（4.2）中，处理新的串行化空间时，新的串行空间会入栈，并继续读入该栈中第一个计算空间（该空间深度为$tar+1$）。公式（4.3）中，在串行空间处理结束并向上返回时，当前空间会出栈，并在上层栈内空间（$Read(tar-1)$）中读入新的下级空间（$'zone^{tar}$），该新空间入栈并被处理（$Read(tar)$）。公式（4.4）中，在处理并行计算任务时，该任务会被放入子线程（$T'$），并被记录在当前线程（$T$）的子线程池（$Pool$）中，当前线程则会继续在上层栈中读入新的下级空间。公式（4.5）中，在处理并行空间时，当前并行空间（$zone_p^{tar}$）会被存入新的空栈（$S'(null)$）中，空栈将作为新线程的工作栈，一同记录在当前线程的子线程池中。当前线程则会在当前空间的上层（tar-1）继续读入新空间。公式（4.6）中，并行空间的返回要求当前工作线程的读入内容为空，且子线程池为空，这时当前工作栈会被清空，当前工作线程关闭，且被移除出父线程的子线程池（$Pool''$）。

a-zone 是一类特殊的空间，该空间内不含子空间或计算任务，而是等待当前线程的子线程池清空。其操作过程如公式（4.7）、公式（4.8）描述。当前工作线程会在目标空间入栈后进入休眠，直到子线程池被清空后，等待空间出栈，工作线程向上层栈内空间中读入新的下级空间。值得注意的是，a-zone 只会对当前线程及其子线程间关系进行调控，并不能影响父线程。下图描述了图 4.5 中所有计算任务的执行时关系。

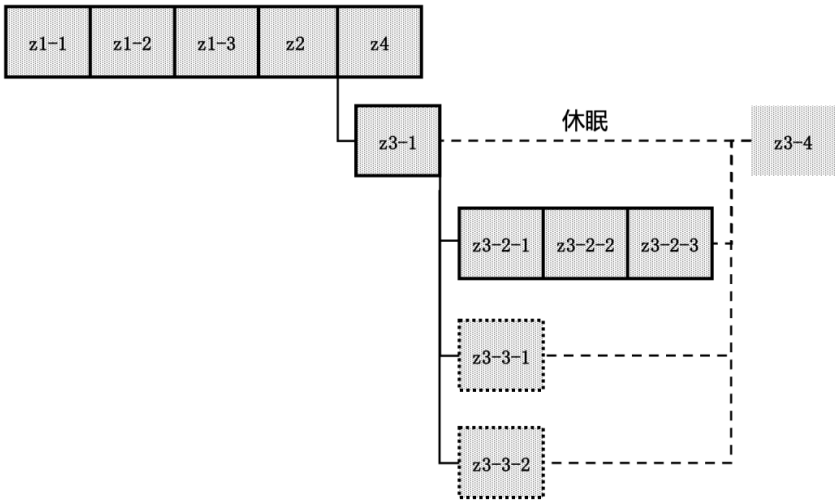



图 4.6 MRDSC 计算任务间调度关系示意

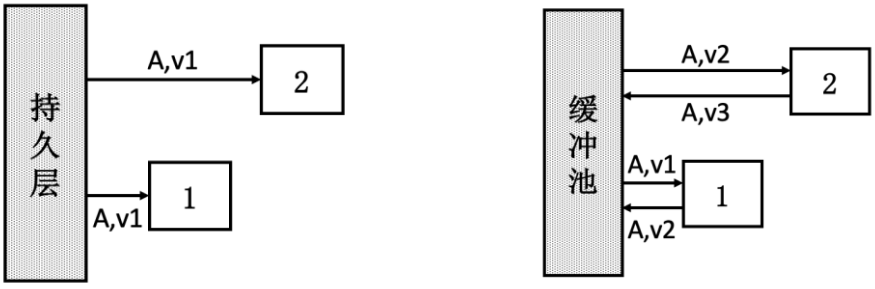
Fig. 4.6 MRDSC Scheduling Relationships for Computational Tasks

如图 4.6。通过本线程休眠，z3-2 能够获得 z3-2-3、z3-3-1、z3-3-2 的计算结果。然而，该方案仍不能带来确定性重放的保证。该方案下的确定性重放依然要依赖于代码编写过程的严谨。4.3 节中描述了合约事务结果确定性重放问题的解决。

### 4.2.3 任务间资源的乐观共享

所有的链上数据，包括账户、余额、合约、合约中的某个数据，统称为数据资源，以区别区块链成员的计算资源与存储资源。若不特殊说明，文章中资源为数据资源的简称。

任何类 STM 技术，在面对密集资源竞争时，都不可避免地遭遇大规模重放问题。这是因为线程间相对独立的数据，难以通过妥当的方式进行同步。我们通过缓冲池缓解这一问题。




（a）持久层交互　　（b）缓冲池交互

图 4.7 缓冲池原理示意

Fig. 4.7 Example of Buffer Pool

相比于 STM 中为每一个任务创建一条新线程，我们的合约采用多层嵌套结构，通常是几个任务顺序使用同一线程，不同线程任务之间也通常是相对有序的（参照图 4.6，z3-1 与 z3-3-1 相对有序）。这种相对有序实际上会使任务的实际执行时间排序与任务的原始序列有很高的相似度。在这种情况下，利用缓冲池可以避免后执行任务无法读取到其他更早时刻的任务执行结果，从而导致的资源冲突与任务重放，如图 4.7（a）所示，$A, v1$表示名为$A$的资源，获取值$v1$，尽管不同线程的计算任务 1 与任务 2 之间相对有序，计算任务 2 虽然更晚执行，但仍然无法读取任务 2 的结果，则任务 2 最终必然被重放。图 4.7（b）中缓冲池缓解了这一问题。

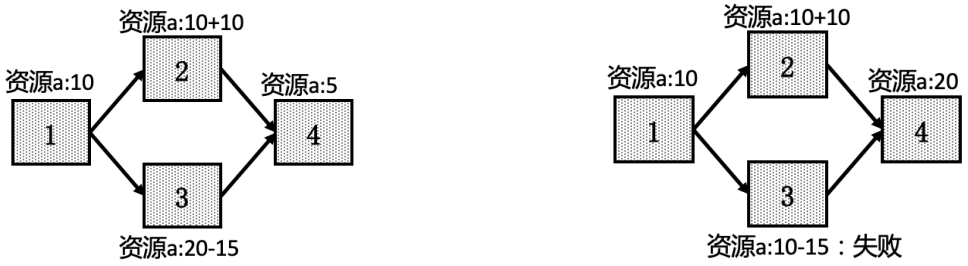
当然，缓冲池无法绝对性的解决资源冲突与任务重放。相关问题会在 4.3 节中被讨论。

## 4.3 基于重序列化的重放确定性保障方法

此处给出智能合约重放确定性的定义：

对于任意智能合约，在其执行前区块链上资源的集合确定时，若其执行后的资源集也是确定的，则其满足智能合约的重放确定性。

智能合约并行编程的重放确定性，通常依赖高质量的编程代码，不能提供严格的确定性重放保证。编程过程中，对资源竞争控制的失误，会严重影响区块链安全性，以下图为例：




（a）并行结果 1　　（b）并行结果 2

图 4.8 并行编程重放不确定性示例

Fig. 4.8 Example of Replay Uncertainty for Concurrent Programming

图（a）中，任务 2 先于任务 3 竞争获得资源 a，全部计算任务结束后，资源 a 值为 5。图（b）中，因为任务 3 先于任务 2 竞争获得资源 a，最终结果资源 a 值为 20。在时间上的资源竞争的偶然性会带来最终计算结果的不同。确定性重放保证，需要消除资源竞争中的该种偶然性。

另外，因为 a-zone 并不存在资源读写，故本节不对 a-zone 做特殊讨论。事实上，a-zone 不会对并行的重放确定性有影响。

### 4.3.1 计算任务的版本控制方法

为了能够记录计算任务在预执行阶段的执行环境与结果，需要一个良好的数据版本化管理方案。读写集能够满足此类需求。此处，我们重新定义了计算任务的读写集：

计算任务读写集是某一计算任务执行过程所读入的全部资源集与产生了改写行为的资源集的组合。将该计算任务记作$t_i$，则该读写集可以记作$RW(t_i)$。

一定程度上，计算任务读写集是对某一计算任务的开始及结束状态的描述。读写集以某任务向其映射的方式保管公式（4.9），构成上实际是读集$R(t_i)$与写集$W(t_i)$的二元组公式（4.10）。读集记录了计算任务开始时所进行过读行为的所有资源的取值，写集则记录了计算任务结束后将要进行改写行为的资源的目标值。特殊的是，对某一资源，其并非是一个单纯值，而是资源名$r$，数值$V(r)$与时间戳$TS(t_i)$的复合值。我们给出其形式化描述：

$$t_i \longmapsto RW(t_i) \tag{4.9}$$

$$RW(t_i) = (R(t_i), W(t_i)) \tag{4.10}$$

$$R(t_i) = \{(r, V_i^{read}(r), TS(t_i)) | r \in (rs_{read}(t_i) \cup rs_{write}(t_i))\} \tag{4.11}$$

$$W(t_i) = \{(r, V_i^{write}(r), TS(t_i)) | r \in rs_{write}(t_i)\} \tag{4.12}$$

公式（4.11）中，$rs_{read}(t_i)$表示计算任务$t_i$中发生读行为的资源名的集合，$V_i^{read}(r)$某资源$r$在计算任务$t_i$中的读数值。不难发现，任意计算任务中，写集所涉及到的资源的集合，必是读集中涉及到资源的集合的子集，这是因为任何数值，都一定会基于原有数值。在 4.3.2 节中，读写集将作为计算任务重序列化的重要基础。

另外，在 MRDSC 合约中每一个代码段的开端，需要对该代码段所有可能使用的资源进行统一读入，在代码段的末尾需要对可能产生改写的资源进行统一提交。在任务代码编译时，需要满足以上两个格式检查。这是因为计算任务中存在的判断与选择类型计算，可能导致某些未运行代码中的资源不被记录在读写集中，从而在重放时出现新名称的资源，破坏重放过程中资源的传递关系。总之，任何一个计算任务，在任何时刻的执行结果，其读写集所包含的资源名不会发生变化。

读写集能够提供对资源的良好的版本化记录。该种版本化资源同样会在缓冲池中被使用，进行乐观的数据共享。缓冲池的最重要性能，是快速检索某一资源最新取值的能力。因此，计算任务结果在缓冲池中的存放与检索方法对缓冲池性能表现有很重要的影响。因此，本文提出了反向读写集，该种方案有更优于读写集的检索速度。相比于读写集建立“任务-资源读写结果”之间的映射，反向读写集是建立“资源-任务历史”的映射。我们给出反向读写集的形式化表达：

$$r \mapsto RW^{reverse}(r) \tag{4.13}$$

$$RW^{reverse}(r) = (R^{reverse}(r), W^{reverse}(r)) \tag{4.14}$$

$$R^{reverse}(r) = \{(t_i, V_i^{read}(r), TS(t_i)) | r \in (rs_{read}(t_i) \cup rs_{write}(t_i)\} \tag{4.15}$$

$$RW^{reverse}(r) = \{(t_i, V_i^{write}(r), TS(t_i)) | r \in rs_{write}(t_i)\} \tag{4.16}$$

在公式（4.15）所表示的反向读集中，任何对资源$r$产生过读或写行为的任务，都会被记录，反向写集与其相似。实际上，缓冲池真正频繁使用的只有反向写集，因为只有写操作才有可能造成值变化，最终形成资源冲突。通过形式化表达上的对比，可以发现，对资源改写历史进行正向检索的反向读写集，更符合缓冲池的工作模式。

### 4.3.2 计算任务的重序列化方法

要想保证智能合约重放确定性，首先要保证合约的计算任务有确定的逻辑序[64]，逻辑序由 Lamport 提出，用来表示分布式系统中，事件在缺乏全局时钟情况下的顺序关系。以同样逻辑序发生的事件，最终会使系统到达一个确定的状态。只要找到逻辑序的等价执行序列，即通过执行该序列，其结果与执行初始序列完全相同的序列，就可以使并行的任务最终达成重放确定性。

在 4.2.2 节中，已经获得推断，叶子计算空间与计算任务满足一一对应的关系。因此，通过对计算任务与计算空间的等价树结构进行先序遍历，就容易获得一个确定的计算任务的序列。该序列满足我们合约编写过程中对计算任务执行的期望顺序，将该序列视作 MRDSC 的逻辑序，称为初始序列。

接下来对 MRDSC 的任务预执行顺序与初始序列做出比较。对于存在资源竞争的任意两个计算任务，如果任务的预执行顺序与这两个任务的在初始序列中的顺序不同，那么就需要进行重序列化并重放。

在预执行阶段，使用有向无环图（Direct Acyclic Graph, DAG）分析资源依赖情况，有助于找到并表示计算任务的实际执行顺序。我们给出图 4.6 中部分计算任务的执行关系与资源读写关系图：

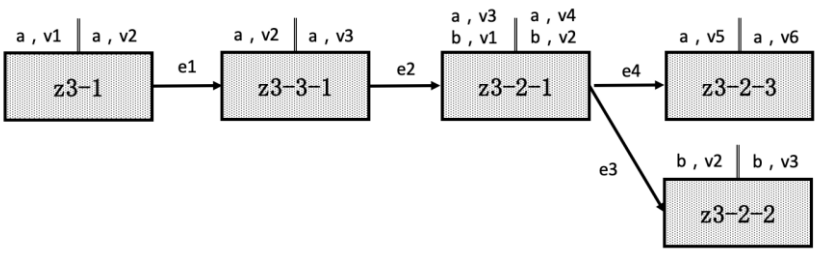



图 4.9 部分任务执行与资源读写关系图

Fig. 4.9 Partial Task Execution and Resource Read/Write Relationship

如图 4.9，合约事务以图示顺序完成计算任务的执行，相关资源根据该流程形成完整的改写链条，任务框上方双竖线左侧表示该任务的读集，双竖线右侧表示该任务的写集。两条任务间的有向边表示后方任务中的读集必须包含前方任务写集中的至少一个资源，称为“依赖”。若两个事务依赖了同一个前置事务中的同一个资源，则称两个事务间存在资源竞争。图 4.9 又被称为资源依赖图。

接下来寻找需要重序列化的任务。比如，任务 z3-3-1 在 z3-2-1 前改写了资源$a$的取值，这与初始序列并不相同。我们在图 4.10 中给出图 4.9 的重序列化后的结果图，称为重序列化图。重序列化图的使命是，在只考虑资源名引用连贯，不考虑资源版本连贯的情况下，使任意两个任务的先后关系严格与初始序列一致。该图能够直观展示哪些任务的顺序关系是错位的，以及哪些任务是需要重新在新序列下重放的。

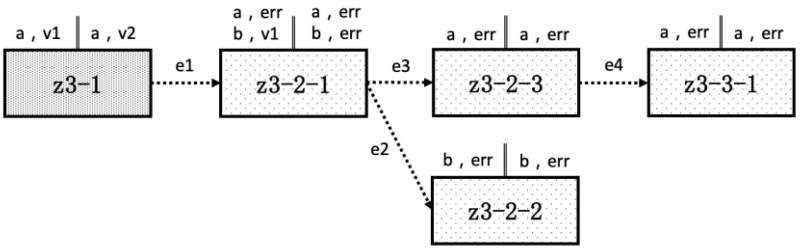



图 4.10 期望任务顺序与计算错误比较图

Fig. 4.10 Comparison of Expected Task Order and Computational Errors

边$e1$表示了期望的资源$a$的依赖关系，其左侧任务写值 v2 与右侧任务读值 v3 并不能匹配，表示未符合期望。该处错误会导致任务 z3-2-1 中对 b 的计算结果同样变得不可靠，从而导致所有对该任务存在依赖或间接依赖关系的后续任务必须按照该有向无环图中的序列重放（浅色阴影标记）。

这一过程可以被简称为计算任务重序列化过程，此处正式给出任务重序列化图的基本构造规则：

（1）任一计算任务的读集中的任一资源都会对应该任务的一条入边，该入边的起点任务的写集应该包含该资源的同名资源。

（2）有向边只能从初始序列中更靠前的任务指向更靠后的任务。

（3）任一任务，其入边的起点，在初始序列中应该与其尽可能的靠近，直到无法找到替代。

（4）任一计算任务的读集中的任一资源，若无法在初始序列更靠前的任务的写集中找到同名资源，则对应的入边可以省略。

（5）起点、终点相同的入边只记录一次。

建立资源依赖图，并逐一对比任务顺序，逐步建立重序列化图，是繁琐的。根据以上规则，在获得了任务预执行的读写集之后，可以不必构造资源依赖图，而直接建立重

序列化图，再通过对比重序列化图中的资源版本冲突进行任务重放。以下我们给出该快速任务重序列化算法：

**算法 4.1 快速重序列化算法**

```
1: procedure ReSerializeDAG(InitialTaskSequence ITS, ReadSet R, WriteSet W)
2:     EdgeSet es
3:     VertexSet vs
4:     for task in ITS then
5:         vs.newVertex(task)
6:         for (rkey,rvalue) in R(task) then
7:             pretask=ITS.getPreviousTask(task)
8:             while pretask= ITS.getPreviousTask(pretask) then
9:                 if W(pretask).containsKey(rkey) then
10:                    es.newEdge(pretask,task)
11:                    if rvalue!= W(pretask)[rkey] then
12:                        task.markError
13:                    break
14:    return (vs,es)
```

以下给出任务的不可靠结果传播状况的检测算法。该算法会将所有结果不可靠的任务进行标记，并找到需要重放的任务序列中的起始任务。

**算法 4.2 不可靠任务检测算法**

```
1: procedure ErrorTaskMark(VertexSet vs, EdgeSet es)
2:     TaskSet result
3:     for task in vs then
4:         if task.errorFlag then
5:             re.add(task)
6:             ErrorPropagate(task,es)
7:     for task in result then
8:         for pretask in es.previousTaskSet(task) then
```

```
9:          if preTask.errorFlag then
10:             result.remove(task)
11:             break
12:     return result
13: procedure ErrorPropagate(Task t, EdgeSet es)
14:     t.markError()
15:     for nextTask in es.nextVertexSet(t) then
16:         ErrorPropagate(nextTask,es)
```

## 4.4 重放确定性证明

本节给出 MRDSC 结果确定性重放的简单证明。证明过程主要包括三个部分：

（1）任意 MRDSC 合约，一定拥有计算任务的唯一的初始序列。

证明：将一个合约方法视为一个根 zone，因为子 zone 在父 zone 中是严格有序的，故 zone 间的包含关系满足有序树的基本特征。有序树叶子节点的先序遍历结果是唯一的，可以推断叶子 zone 的先序遍历结果是唯一的，即计算任务的初始序列是唯一的。

（2）任意 MRDSC 合约，若计算任务具备唯一初始序列，则其初步执行结果的重序列化图唯一。

证明：为了探究图是否唯一，首先要明确图相等的概念。若能够保证任意两次初步执行结果构造的重序列化图之间相等，则重序列化图是唯一的。

假设重序列化图能够被写为$Graph(Vset, Eset)$，其中，顶点集$Vset = \{(i, rs_{read}(t_i), rs_{write}(t_i))|t_i \in \text{MRDSC}\}$，表示顶点集中包含 MRDSC 中任意一个计算任务$t_i$的唯一标识符$i$，任务中发生读行为的资源名集$rs_{read}(t_i)$，以及写行为资源名集$rs_{read}(t_i)$。有向边集$Eset = \{(j_r, i^r)|\text{r} \in rs_{read}(t_i), t_i \in \text{MRDSC}\}$，表示由顶点$j$指向顶点$i$的边，其右上标标识了其读集中的某个资源，右下标标识了其写集中的某个资源。若两个重序列化图$GraphA(VsetA, EsetA)$、$GraphB(VsetB, EsetB)$，满足其顶点集相等（$VsetA \subseteq VsetB$，$VsetB \subseteq VsetA$）同时边集相等（E$setA \subseteq EsetB$、$EsetB \subseteq EsetA$），则可以称$GraphA = GraphB$。

容易发现，因为顶点集$\{(i, rs_{read}(t_i), rs_{write}(t_i))|t_i \in \text{MRDSC}\}$只与相关资源的资源名有关，而与任一次初步执行时的资源值无关。又因为文章定义了任何一个计算任务，在任何时刻的执行结果，其读写集所包含的资源名不会发生变化。故任意两次初步执行产生的顶点集是相同的。接下来证明这两个重序列化图的边集相同。

因为两个重序列化图的顶点集相同，那么对两图中的相同顶点$t_i$，两者读资源名集中同样名为$r$的资源，由 4.3.2 节中的约束 1 可知，两者各对应了一条入边。因为初始序列唯一，则约束 3 能够保证这两条边会取到相同的起点。约束 2、4、5 实际保证了该边能满足功能需求，因为两个重序列化图会同样遵守这些约束，所以不会影响边的起点的选择。至此可以证明，两个重序列化图中相同顶点的相同资源，其对应相同的边也会相同，将该边记作$(j_r, i^r)$。

又因为两个重序列化图的顶点集相同，则两图中的相同顶点，其读资源名集中的资源完全相同。又已证明相同顶点的相同资源的入边$(j_r, i^r)$是相同的，则两个重序列化图中相同顶点的读资源名集中所有资源对应的入边的集合，即相同顶点$\big(i, R(t_i), W(t_i)\big)$的入边的集合$\{(j_r, i^r)|\mathrm{r} \in R(t_i)\}$是相同的。

又因为两个重序列化图的顶点集相同，且相同顶点的入边的集合$\{(j_r, i^r)|\mathrm{r} \in R(t_i)\}$相同，则两图全部顶点的入边的并集，即两图的边集$\{(j_r, i^r)|\mathrm{r} \in R(t_i), t_i \in \mathrm{MRDSC}\}$是相同的。

至此，目标得证。

（3）任意 MRDSC 合约，若其重序列化图唯一，则其满足结果确定性。

证明：对名为$r$的资源，假设其在重序列化图中的最后一次改写发生在任务$t_i$中。在重序列化图唯一的前提下，所有指向$t_i$或间接指向$t_i$的任务是确定的，其中使用了同名资源的任务之间，在重序列化图中必有严格的先后顺序，满足顺序编程的特征。因此，能保证$r$值在完成计算后结果是确定的。该结论对任意$r$皆成立。因此，该合约执行后的资源集也是确定的，故该合约满足结果确定性。

## 4.5 实验评估

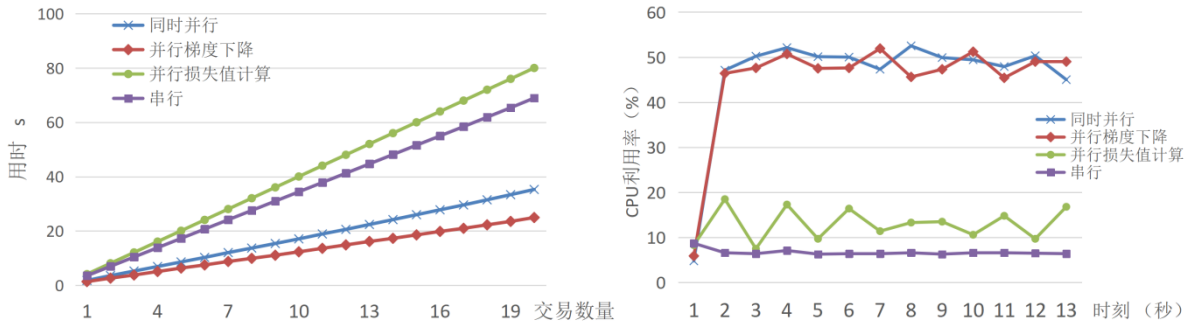
### 4.5.1 实验环境

实验设备的基本硬件配置，为 Intel(R) Core(TM) i7-10700 CPU @2.90GHz 2.90 GHz，16G 内存。操作系统使用了 64 位 Windows 操作系统。语言环境为 python 语言环境。为了探索 MRDSC 的并行处理性能以及资源冲突处理效率，我们以多元线性回归的 MRDSC 为例，测试了其在在不同情况下的性能表现。

多元线性回归及其单笔事务并行执行方法，可见于 3.2 节。本节中的线性回归实验中，样本数量在 5000 左右，参数数量，即特征数量为 100。该合约的并行过程可以分为两阶段。第一阶段是损失值的并行计算，第二阶段是向量参数梯度下降的并行计算。从计算时间复杂度上来说，第二阶段是第一阶段的 n（特征数量）倍。从并行程度上来说，第一阶段将所有样本分为了五组，而第二阶段，对 100 个参数分别进行了并行，其并行程度远高于第一阶段。在以上条件下，文章完成了以下实验并给出了相应说明与分析。

### 4.5.2 并行性能测试

合约并行能够有效减少合约计算的时间消耗，最重要的原因是并行计算更能够适配计算机的多核计算架构，有效挖掘 CPU 的处理性能。因此，该部分实验通过测量多元线性回归的串行合约、仅损失值进行并行计算的合约、仅向量参数梯度下降进行并行计算的合约以及损失值与梯度下降过程都进行了并行计算的合约，这四种编写方式的 MRDSC 的处理时间消耗，以及 CPU 利用率，以推测 MRDSC 在线程数量、计算资源密集程度、多核计算适配程度方面的表现。在使用四处理器的情况下，实验数据被记录如下：




（a） 计算用时测试　　（b）CPU 利用率测试

图 4.11 MRDSC 性能实验图

Fig. 4.11 MRDSC Performance Experimental

由图 4.11（a）中数据可以发现，MRDSC 更能够适配多核计算架构，带来显著的事务验证速度的上升。在并行梯度下降和同时在两阶段采用并行的方案中，事务的计算时间都非常明显的低于串行验证。这一结论在图（b）中可以得到印证。

另一方面，并不是并行程度越高，就可以节省更多事务计算时间。损失值并行计算时间消耗多于串行计算、两个阶段同时使用并行的时间消耗多于仅在参数梯度下降时使用并行计算，通过这两个对比可以发现，在损失值上的并行计算显然会导致计算时间消耗的增长。通过对比图（b），并行损失值计算 CPU 使用率会高于串行计算，可以发现，时间消耗的增长并不是因为无法适配多核架构，而是计算量发生了上升。这是因为在计算密集度较小的情况下，过高的并行程度，会带来额外的任务调度开销，反而会拖累事务验证速度。

### 4.5.3 冲突处理测试

MRDSC 在实现确定性重放的过程中，不得不进行资源冲突的处理。为了测量 MRDSC 的冲突处理能力与效率，我们改变了多线线性回归合约的编写方式。

采用 3.2 节中的有竞争编写方式。在第一轮并行中，每一个独立并行任务，都对另一个任务的计算结果存在依赖。在第二轮并行中，每个并行任务，都存在对第一轮并行

的结果（该种编写方案中，即第一轮并行的最后一个计算任务）存在依赖。在任务的并行过程中，如果实际资源依赖情况与期望依赖顺序不同，MRDSC 会识别这些错误，然后以期望的资源依赖顺序进行重放。这能够保证并行编程的确定性重放，但是会带来额外的处理开销。

我们首先尝试仅仅使用缓冲池，而不使用 a-zone 来解决资源读取版本错位的问题，并对这部分开销做了测量，实验结果如图：

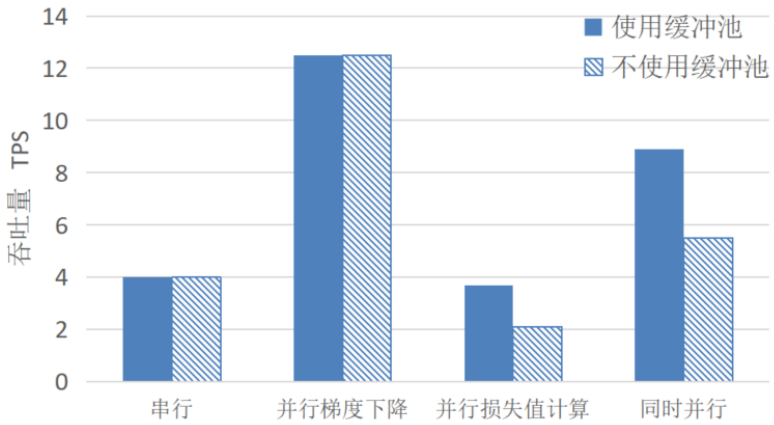



图 4.12 是否使用缓冲池对吞吐量的影响

Fig. 4.12 Impact on Throughput of Using Buffer Pools or Not

显然，在使用串行编写方案时，区块链系统吞吐量极低，但并不会存在资源依赖冲突的问题，所以缓冲池不会带来吞吐性能的提升。通过对比并行梯度下降方案与串行方案、并行损失值方案，能够发现对吞吐量具备最大影响的，是第二轮参数的并行梯度下降过程，并行损失值计算反而会带来吞吐量一定程度的下降。据此推断，并行梯度下降方案总体吞吐极高，但缓冲池没有带来吞吐提升，这是因为损失值计算使用了串行方案，因此不会在并行梯度下降阶段存在资源读取落后的问题。并行损失值计算，若缺少缓冲池，则会导致梯度下降阶段出现大规模的计算任务重放。这一结论同样能够在两阶段同时采用并行方案的合约测试结果中得到印证。实验证明，缓冲池缺失能够带来吞吐性能的提升。

除此之外，我们在有缓冲池的情况下，同时使用 a-zone 在第二轮并行前等待第一轮并行结果，测量了使用 a-zone 能够带来的吞吐提升，如图 4.13。采用同时并行的合约编写方式，施加稳定的多元线性回归事务负载，MRDSC 能够表现出稳定的事务处理效率。通过使用 a-zone，能够帮助计算任务建立对最新资源的依赖，从而避免大量重放。实验表明，使用 a-zone 虽然不可避免地带来后续计算任务挂起的等待用时，但时间消耗的总体表现，却明显好于不使用 a-zone。

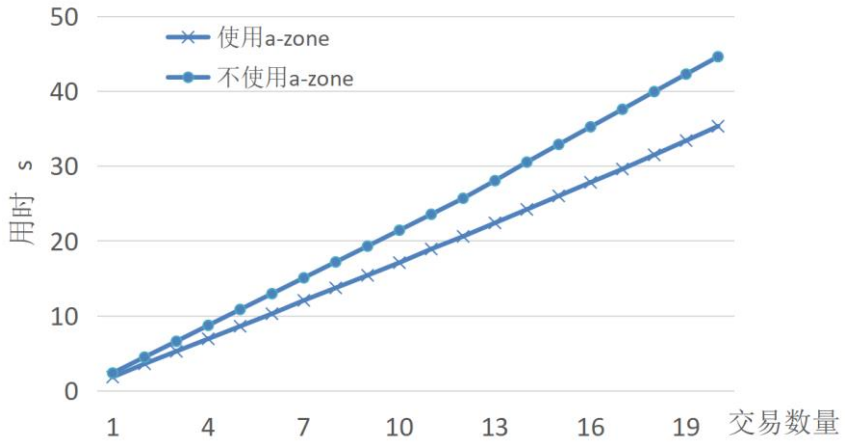



图 4.13 a-zone 对合约事务计算耗时的影响

Fig. 4.13 Impact on The Computation Time of a-zone

## 4.6 本章小结

在本章节中，我们提出了具备确定性重放保证的多核适配智能合约 MRDSC，以解决 CICT 执行时间过长的问题，实现对单笔 CICT 的并行计算策略。包括多核适配的支持并行编程的智能合约模型，解决了该模型的任务调度以及资源的乐观共享问题。另外，本章为 MRDSC 提供了重放确定性保障方法。该方法通过读写集与反向读写集进行数据的版本化管理，使用新的逻辑序、资源依赖图、任务重序列化图来寻找等价序列，解决区块链智能合约并行编程中存在的不确定性问题。本章同样给出了重放确定性的证明。最后通过实验，证明了 MRDSC 能够有效通过并行执行 CICT，进一步挖掘多核设备的计算性能，提高区块链系统吞吐。实验同样证明了 MRDSC 的资源冲突处理策略有较高的效率与有效性。

# 5 基于链下计算的 CICT 计算抑制类并行策略

## 5.1 引言

合约事务的链下计算一直是区块链领域的一个重要课题，主要原因在于链下计算能有效缓解区块链算力压力，且有助于解决传统区块链在面临计算密集型合约事务（CICT）时，可能出现的激励层失效问题。但较为完善的链下计算方案始终未被提出。

区块链算力压力过大，主要是因为 CICT 会随区块的打包，被传播给区块链所有参与成员，参与成员则必须要重新计算并验证该事务，区块链才能达成共识[65]。该过程中，该合约的算力消耗，会以参与计算的成员数目为倍数放大。这一庞大的算力消耗会致使区块链系统面临因算力不足带来的性能瓶颈，尤其是对计算性能较差的成员节点来说，过长的事务计算时间，会直接导致该成员无法继续执行其他正常任务，事务结果将长时间无法达成共识，直接威胁区块链的共识安全及数据可用性[66]。

激励层失效问题，在于区块链系统会将合约事务的验证奖励（比如 gas）发放给区块打包者，以激励区块链成员验证并发布合约事务[64]。然而，区块中合约事务验证时，并不直接激励成员执行合约中的计算。随着合约事务验证代价，即算力消耗的逐渐上升，放弃合约事务的验证将成为区块链成员的更优方案[37]，这一困境又被称为验证者困境。

而采用链下计算方案，参与合约事务验证的计算者将被局限在少部分算力资源丰富的节点范围内，这一数量将远远小于区块链全部参与成员的数量。因此，链下计算能够有效限制合约事务的算力放大问题，减缓区块链系统算力压力，促进区块链突破性能瓶颈。对非计算者的参与成员来说，其不再需要重新执行合约计算，也就不再面临激励层失效的问题。新的激励层，则需要补充针对计算者的相关激励策略。

综上，链下计算是解决区块链系统在面对 CICT 时，可能出现的吞吐量下降、激励层失效的重要方案。随着可新硬件与可验证计算技术的成熟，已经出现了许多链下计算方案，比如基于交互式验证协议的链下计算策略，但该方案始终存在链上链下数据一致性较弱，稳定性较差等问题。

本文中，我们提出了链上挑战协议，以加强链上链下数据的一致性，解决交互式验证协议可能造成的失效交易过多的问题，进一步突破区块链性能瓶颈。文章同样提出了基于链上挑战协议的完整链下计算方案，包括激励策略、安全性分析以及一种超前交付策略，该策略能有效提升链上数据的使用效率。

## 5.2 链上挑战型的链下计算协议

为了实现链下计算，文章将所有区块链成员分为了两类，如图 5.1。共识组中成员被称为共识成员，其主要工作是完成区块链中的常规共识任务，比如新区块共识、事务结果共识、最长链共识等，是区块链信任传递的基础。计算组中成员被称为计算成员，计算成员通常使用计算性能更强的设备，甚至是专业的大型计算设备。计算成员同样是区块链的参与成员，保存了完整的区块链账本，可以发表新的交易。在链下计算协议中，计算成员主要负责完成合约事务中的计算，并提供计算结果的证明，共识成员只负责验证该证明，以确定合约事务结果的合法性。

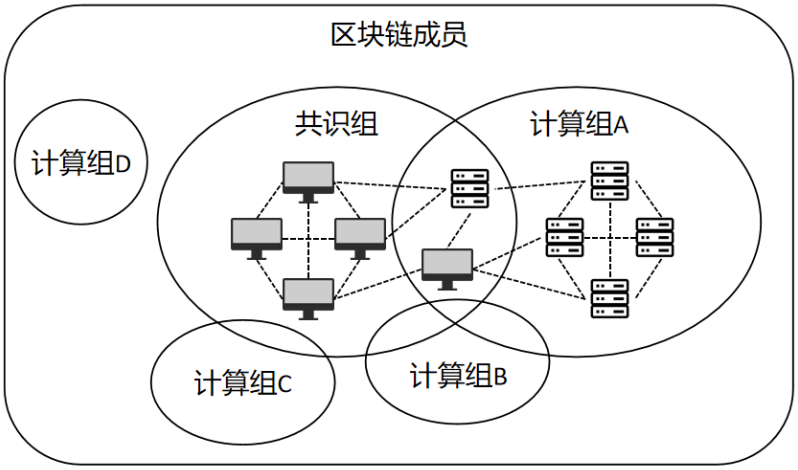



图 5.1 区块链成员分类

Fig. 5.1 Classification of Blockchain Members

容易发现，共识组与计算组并不是严格互斥关系。同一个设备可能既需要完成共识组的任务，又需要承担计算组的任务，则该设备同时具备两种角色。同样，计算组之间也不是互斥的。计算组并不进行严格的成员管理，若某些计算成员同时决定执行某个 CICT，则它们自动被划入了同一个计算组。一个计算成员则可能因为同时执行了多个 CICT 所以同时处于多个计算组。计算组之间互不干涉对方的计算任务，这正是链下计算能够实现 CICT 事务并行的基础。另外，为了保障 CICT 一定会有对应的计算组承担其计算任务，激励层策略是必要的。

### 5.2.1 协议的基本流程

为了解决“挑战-响应”型链下计算协议存在的缺陷，我们提出了链上挑战型链下计算协议，以下简称为链上挑战协议。该协议使用了交互式验证协议（详见 2.4.2 节）进行事务结果可靠性保证。链上挑战协议工作流程如下图：

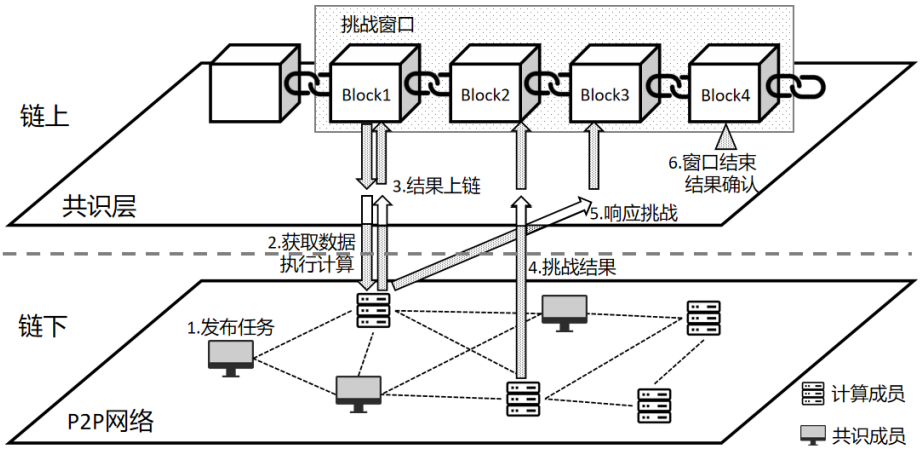



图 5.2 链上挑战协议工作流程示意

Fig. 5.2 Workflow of On-Chain Challenge Protocol

步骤 1:用户发布合约事务，并向区块链网络进行传播。

步骤 2:有意愿完成合约事务中计算任务的计算成员，自动组成计算组，读取合约状态，并执行合约代码，获得事务结果。

步骤 3:计算成员需要将合约事务、事务结果以及按照交互式验证协议生成的$root_{VM}$进行签名，并打包发布在区块链上。计算过程树则在 P2P 网络中进行广播。

步骤 4:在收到新区块后，其他同组计算成员，会对比区块中的事务结果与本地计算结果。若不一致，则按照交互式验证协议，将挑战信息发布在区块链上。

步骤 5:受到挑战的计算成员，将响应信息以新事务的形式发布在区块链上。所有参与者在原结果与挑战中的新结果中保留诚实者，成为合约事务的最新结果。

步骤 6:在挑战窗口期中，合约事务结果随时接受挑战。窗口期结束，合约事务的最新结果生效。

过程中的重要数据结构展示如下：

$$< transaction_i^{contract}, txhash_i, txInfo_i, RW(tx_i), root_{VM}, sig > \tag{5.1}$$

$$< challenge_j^{onchain}, txhash_j, txInfo_j, txhash_i, RW(tx_j), root'_{VM}, hash(VMMS_{chal}), hash(VMMS_{chal}^{pre}), sig > \tag{5.2}$$

$$< respond_k^{onchain}, txhash_k, txInfo_k, txhash_j hash(VMMS_{resp}), hash(VMMS_{resp}^{pre}), sig > \tag{5.3}$$

$txhash$，$txInfo$指区块链的基本属性，包括地址，签名等等，此处不做讨论。$transaction_i^{contract}$是结果发布事务，$challenge_j^{onchain}$会在链上发起对事务结果的挑战，$respond_k^{onchain}$是对挑战的响应。对比交互式验证协议与“挑战-响应”型链下计算协议，链上挑战协议有几个明显特性。

（1）挑战消息中包含的不是$VMMS$，而是其哈希，这是因为要适应区块链存储稀缺的特性。在挑战交易在链上发表后，仍然需要向 P2P 网络发布$VMMS$的内容。与此相似的还有响应信息中的$VMMS$，以及$root_{VM}$。

（2）链上挑战协议中，只要结果正确，并不关心树根是否正确，过程树根只用于辅助计算过程错误查验。只要结果正确，错误的过程树根并不会带来任何影响，同样，任何参与者都没有动机故意篡改事务结果与过程树根的联系，因为这除了增加自身响应的困难以外，没有任何收益。

（3）链上挑战协议中，挑战过程会公布一个新的事务结果，挑战成功时，会使用新结果代替旧结果。

（4）根据对链下的定义，链上挑战协议的步骤 3、4、5、6 都必须经过共识层，故属于链上行为。但是这些事物中只包含合法性验证，计算消耗几乎忽略不计。CICT 的计算过程并不在共识层中执行，故该协议明显满足链下计算特征。

对合约事务结果的表示，采用的是读写集的方法，可参照文章 2.3.1 节与 4.3.1 节。另外，在特性（1）中，$VMMS$、响应信息，以及$root_{VM}$都必须伴随着向链下网络的信息发布。如果计算成员在挑战与响应过程中，消极参加链下信息的公布，可能会阻碍链上挑战协议的运行，导致错误结果通过窗口期。为应对此类问题，文章提出了无响应挑战的概念。

无响应挑战以新交易的形式被存储在链上。无响应挑战单独计时，期间用时不被记入挑战窗口期。正在接受无响应挑战的交易暂不生效，也不会被标记无效。详细作用规则见 5.2.2 节。无响应挑战会在三种情况下被发出：

（1）在交易结果在链上发表后，挑战者需要获取该结果对应的过程树。若向网络发出的对过程树的请求无响应，则发布对过程树的无响应挑战。对应的挑战与响应格式如下：

$$Marker \in \{VMTREE, VMMS\} \tag{5.4}$$

$$< challenge_j^{noresponds}, txhash_j, txInfo_j, txhash_i, Marker(VMTREE), root_{VM}, sig > \tag{5.5}$$

$$< respond_k^{noresponds}, txhash_k, txInfo_k, txhash_j, tree_{VM}, sig > \tag{5.6}$$

（2）挑战者需要获取$VMMS$时无法收到响应，则发布对$VMMS$的无响应挑战。对应的挑战与响应格式如下：

$$< challenge_j^{noresponds}, txhash_j, txInfo_j, txhash_i, Marker(VMMS), hash(VMMS_{resp}), sig > \tag{5.7}$$

$$< respond_k^{noresponds}, txhash_k, txInfo_k, txhash_j, VMMS_{resp}, sig > \tag{5.8}$$

（3）对$VMMS^{pre}$的请求与挑战过程同$VMMS$。

（4）无响应挑战有其独立的挑战窗口期，称为无响应窗口。窗口期内收到正确响应，则挑战结束。若窗口期结束时，无响应挑战仍未结束，则挑战成功。被挑战交易标记为无效。

（5）对同一数据，比如$root_{VM}$或者$hash(VMMS_{resp})$，只能挑战一次。这是为了防止恶意挑战阻碍链上挑战协议运行。

通过无响应挑战，可以解决计算成员消极公布计算细节，阻碍协议运行的问题。发起无响应挑战的经济成本、存储成本都显然较高。但是无响应挑战只是协议不能良好运作时的补救与应急措施，并不是链上挑战协议的常规步骤。在诚实节点占绝大多数的情况下，无响应挑战并不会为区块链系统带来明显负担。

### 5.2.2 协议的资源使用策略

（1）资源的有限状态转换

文章中使用资源指代数据资源，包括账户、余额、合约、合约中存储的某个数据项等链上数据，以区别区块链成员的计算资源与存储资源等。

公式（5.9）给出了资源的表示方法，该方法沿用了 4.3.1 中读写集中的表示形式，资源由资源名$r$、取值$V$、时间戳$TS$构成。$V_i^{write}(r)$表示在事务$i$中向资源名为$r$的资源的写值，新数值的写入表示新资源的诞生。

$$W(tx_i) = \{(r, V_i^{write}(r), TS(tx_i)) | r \in rs_{write}(tx_i)\} \tag{5.9}$$

在链上挑战协议的运行中，链上资源的状态并不一直是已确认的，还有正处于挑战窗口等待确认，以及正在被挑战的资源。保证资源状态的有序转换是保证资源可靠的重要前提。实际上，链上挑战协议中任何资源的状态转换，满足有限状态机的特性[67]。资源值的状态转换总会在已确认$\mathcal{S}_c$，待确认$\mathcal{S}_w$，已废弃$\mathcal{S}_a$，被挑战$\mathcal{S}_{chal}$，被无响应挑战$\mathcal{S}_{nrchal}$间进行。如图 5.3。任何一个资源值都会以$\mathcal{S}_w$的状态被发布，在事件的驱动下，在$\mathcal{S}_w$、$\mathcal{S}_{chal}$、$\mathcal{S}_{nrchal}$三者中转换，最终转换成为$\mathcal{S}_c$或$\mathcal{S}_a$这两种稳定状态中的一种。

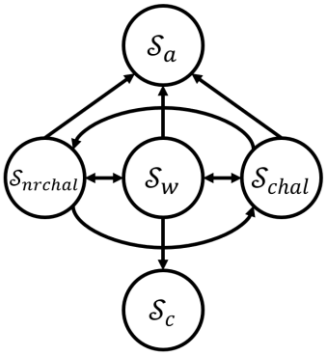



图 5.3 资源的有限状态转换示意

Fig. 5.3 Finite Resource State Transition

下文中我们给出了所有可能促使状态转换发生的事件。每一个事件都会以一个事务的形式被发起。状态转换能且仅能遵循公式（5.10）至（5.15）进行转化。

$$(r, V_{i-1}^{write}(r) * \mathcal{S}_c) \xrightarrow{PUBLISH(tx_i)} (r, V_{i-1}^{write}(r) * \mathcal{S}_c, V_i^{write}(r) * \mathcal{S}_w) \tag{5.10}$$

公式（5.10）描述了一个新资源值伴随着合约事务结果被打包而诞生的过程，$i$为当前事务，使用$i-1$、$i-2$来简单描述事务间先后关系，而不指代明确的间隔数量关系。长箭头上方标记了事件名称与本事件的关联事务。$V(r) * \mathcal{S}_c$表示资源名为$r$，取值为$V$的资源，处于$\mathcal{S}_c$状态。仿照此格式，我们给出剩下的全部事件。

$$\left(r, V_{i-1}^{write}(r) * \mathcal{S}_w, V_i^{write}(r) * null\right) \xrightarrow{CHALLENGE(tx_{i-1}, tx_i)} (r, V_{i-1}^{write}(r) * \mathcal{S}_{chal}, V_i^{write}(r) * \mathcal{S}_w) \tag{5.11}$$

$$(r, V_{i-1}^{write}(r) * \mathcal{S}_{w/chal}) \xrightarrow{NRCHALLENGE(tx_{i-1}, tx_i)} (r, V_{i-1}^{write}(r) * \mathcal{S}_{nrchal}^{w/chal}) \tag{5.12}$$

$$(r, V_{i-2}^{write}(r) * \mathcal{S}_{chal}, V_{i-1}^{write}(r) * \mathcal{S}_w) \xrightarrow{RESPOND(tx_{i-2}, tx_{i-1}, tx_i)} (r, V_{i-2}^{write}(r) * \mathcal{S}_w, V_{i-1}^{write}(r) * \mathcal{S}_a) \tag{5.13}$$

$$(r, V_{i-2}^{write}(r) * \mathcal{S}_{chal}, V_{i-1}^{write}(r) * \mathcal{S}_w) \xrightarrow{FAILRESPOND(tx_{i-2}, tx_{i-1}, tx_i)} (r, V_{i-2}^{write}(r) * \mathcal{S}_a, V_{i-1}^{write}(r) * \mathcal{S}_w) \tag{5.14}$$

$$(r, V_{i-2}^{write}(r) * \mathcal{S}_{nrchal}^{w/chal}) \xrightarrow{NRRESPOND(tx_{i-2}, tx_{i-1}, tx_i)} (r, V_{i-2}^{write}(r) * \mathcal{S}_{w/chal}) \tag{5.15}$$

公式（5.11）中，挑战只能发生在待确认状态的资源上，使其进入被挑战状态，但会有新的结果被发布为$\mathcal{S}_w$状态。公式（5.12）中，无响应挑战会发生在待确认或者被挑战状态的资源上，使其进入被无响应挑战状态，并包含对上一个状态的记忆，记作$\mathcal{S}_{nrchal}^{w/chal}$。公式（5.13）中，响应会使被挑战状态的资源返回待确认状态。公式（5.14）表示在响应挑战后，如果确实被证明出错，则原值会被废弃。公式（5.15）中，无响应挑战的响应会使资源返回原状态。除此之外，还有三种挑战窗口触发的事件，在此处给出：

$$(r, V_{i-t_{win}}^{write}(r) * \mathcal{S}_w) \xrightarrow{WINSHUT(tx_i)} (r, V_{i-t_{win}}^{write}(r) * \mathcal{S}_c) \tag{5.16}$$

$$(r, V_{i-t_{win}}^{write}(r) * \mathcal{S}_{chal}) \xrightarrow{WINSHUT(tx_i)} (r, V_{i-t_{win}}^{write}(r) * \mathcal{S}_a) \tag{5.17}$$

$$(r, V_{i-t_{nrwin}-\theta t_{win}}^{write}(r) * \mathcal{S}_{nrchal}) \xrightarrow{NRWINSHUT(tx_i)} (r, V_{i-t_{nrwin}-\theta t_{win}}^{write}(r) * \mathcal{S}_a) \tag{5.18}$$

公式（5.16）中，使用$t_{win}$简单描述交易相对于挑战窗口的位置关系。在挑战窗口关闭时，待确认状态资源会被确认，公式（5.17）中，被挑战状态资源则会被废弃。公式（5.18）中，无响应挑战窗口关闭时，资源会被废弃。使用一个小于 1 的变量$\theta$简单描述发生在挑战窗口内的任意时间的无响应挑战。

（2）资源的超前交付策略

根据资源状态转化，资源向已确认状态的转化只会发生在挑战窗口关闭时。显然，从交易结果发布，至结果被确认，必然经过一个延迟时期$t_{delay}$，本文中将其称为资源到达延迟。其取值范围容易推断：

$$t_{win} \le t_{delay} \le t_{win} + (2k_c + 1)t_{nrwin} \tag{5.19}$$

容易发现，资源到达延迟一定高于挑战窗口长度，同时受无响应挑战次数影响。尽管激励措施会保证无响应挑战不会频繁发生，但该延迟仍会消耗一个较长的时间，可能从几个区块到几十个区块不等。从传统的数据管理角度来看，该资源在这段时间需要被锁定，以防止资源竞争与事务冲突。这使得相关资源较长时间无法被使用，文章中称其为资源阻塞。

为了解决这一问题，我们允许一笔资源在未被确认的状态下被使用，该策略称为超前交付策略。这无可避免的会造成资源竞争问题，因此，超前交付策略的本质是资源间关系影响下的资源状态控制策略。我们以读写集的形式发表合约事务结果，通过事务间的资源依赖 DAG 进行资源的竞争管理。（读写集内容详见 4.3.1 节，资源依赖图详见 4.3.2 节。）

如图 5.4 所示，每个方形代表一笔已发布交易，其上方双竖线左侧为读集，是资源名和资源值的二元组构成的集合。右侧为写集。五个事务在区块链上的发布顺序为$1 \prec 2 \prec 3 \prec 4 \prec 5$。两条任务间的有向边表示后方任务中的计算必须读取前方任务中至少一个资源的值，称为依赖。若两个事务依赖了同一个前置事务中的同一个资源，则称两个事务间存在资源竞争。显然，因为事务 2 生效，存在对资源 a 竞争的事务 3 则必须废弃。对事务 3 存在依赖的事务同样需要废弃，如事务 6。因为事务 4 还未被确认，因此对其存在依赖的事务仍需要进行等待，比如事务 5。

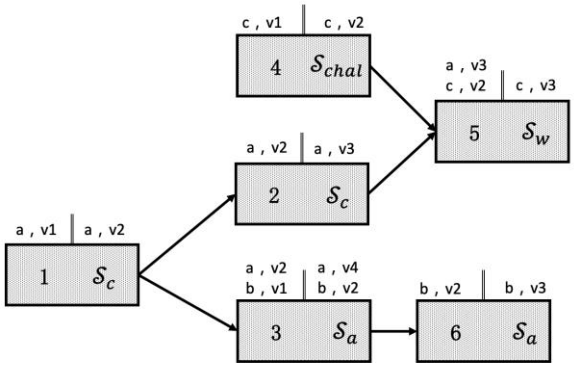



图 5.4 未确认资源的使用示意

Fig. 5.4 Unconfirmed Resource Usage

为了管理资源竞争下的资源状态转换，我们提出了以下三条基本约束：

（1）事务向$\mathcal{S}_c$状态进行转换的必要条件是其依赖的资源尚未被另一个$\mathcal{S}_c$状态的事务改写。

（2）事务向$\mathcal{S}_c$状态进行转换的必要条件是其依赖的交易必须全部处于$\mathcal{S}_c$状态。

（3）事务向$\mathcal{S}_c$状态进行转换的必要条件是与其存在资源竞争且在区块链上更靠前的事务全部处于$\mathcal{S}_a$状态。

以上为超前交付策略的三个基本约束，在其基础上，另外添加一条基本推论。

（4）因为处于$\mathcal{S}_a$的事务不可能向$\mathcal{S}_c$进行转化，故所有对已废弃交易具有依赖的交易都会被废弃。

约束（1）是区块链资源不会双花（Double Spend）的基本保证。约束（2）保证了交易的确认不会基于无效的资源发生，新事务选择计算依赖时会尽量选择正确把握更大的事务作为依赖，以避免整个分支被遗弃。约束（3）是约束（1）的补充约束，加强保证了存在资源竞争的待确认事务的潜在双花威胁。另外，约束（3）保证了存在资源竞争的事务，更早发布的事务更具优势，而非更早通过窗口期的事务。这是一个重要约束，能够鼓励区块链事务的资源依赖尽可能的发生在 DAG 的最长分支上，而不执意添加竞争交易。该约束能够有效减少被废弃的交易，并遏制对竞争交易的攻击。基本推论（4）是提高 DAG 管理效率的重要推论。可见事务结果发布者，不会倾向于在不可靠事务后进行后续计算，任何事物一旦转入$\mathcal{S}_a$状态，区块链即可放弃对该事务及后续事务的管理，甚至直接删除。该方案能有效对 DAG 进行剪枝。

### 5.2.3 协议的激励策略

在交互式验证协议中，激励策略是必不可少的组成。计算成员会获得事务发布者支付的手续费，同时必须为自己的结果支付押金。其他计算成员试图挑战原结果，以获得事务计算手续费以及被挑战者的押金。挑战者同样需要支付押金，以保障不会恶意阻碍合约运行。虽然该过程不够完善，但能够满足交互式验证协议的基本需求。本文对该需求做出以下总结：

（1）计算成员有动力承担合约中的计算、响应其他成员的数据请求、响应可能发生的挑战。

（2）挑战者有动力挑战错误结果

（3）共识成员有动力验证合约事务结果。

链上挑战协议同样需要满足以上需求，在其基础上，有另外的需求：

（4）计算成员能够承担进行挑战或响应时，在区块链上发布交易的成本。

（5）计算成员能够承担在公布交易结果、进行挑战等行为时，由于押金丢失流动性所产生的代价。

（6）任何人发起无响应挑战时，必须不具备正向收益。否则会干扰链上挑战协议的正常运行。

根据对激励策略的基本需求，可以得到链上挑战协议激励策略的基本约束。下文中给出了约束分析，而非固定的激励值。

表 5.1 激励分析相关符号定义表

Tab. 5.1 Definition of Symbols about Incentive Analysis

| 符号 | 含义 |
|---|---|
| $P_p, P_c, P_v$ | 结果发布者/挑战者/共识成员的收益 |
| $\mathcal{E}$ | 区块链参与者对维持区块链正常运行的预期收益 |
| $\mathcal{C}$ | 发布合约任务的成员对计算成员承诺的报酬 |
| $\mathcal{T}_p, \mathcal{T}_c, \mathcal{T}_r$ | 区块链上发布计算结果/挑战/响应的成本 |
| $\mathcal{T}_{nrc}, \mathcal{T}_{nrr}$ | 区块链上发布无响应挑战/响应的成本 |
| $\mathcal{I}_p(i)$ | 发布结果押金在$i$时长损失流动性的成本 |
| $E(i)$ | $i$的期望值 |
| $t_{win}, t_{nrwin}$ | 挑战/无响应挑战窗口长度 |
| $\mathcal{H}$ | 一次合约计算的算力成本 |
| $k_c, k_{nrc}$ | 事务生效前发生挑战/无响应挑战的次数 |
| $o$ | 可忽略不计的其他成本，比如链下通信等 |
| $\mathcal{D}_p, \mathcal{D}_c$ | 发布结果/挑战时押金 |
| $t_c$ | 挑战时剩余窗口长度 |
| $c$ | 虚拟机状态被均分的个数 |

$$P_p = \mathcal{E} + \mathcal{C} > \mathcal{H} + \mathcal{T}_p + \mathcal{I}_p(t_{win}) + o \tag{5.20}$$

$$P_p \geq \mathcal{H} + \mathcal{T}_p + \mathcal{I}_p(t_{win} + E(k_{nrc})E(t_{nrwin})) + E(k_c)(\mathcal{T}_r - \mathcal{D}_c) + E(k_{nrc})\mathcal{T}_{nrr} + o \tag{5.21}$$

$$P_c = \mathcal{E} + \mathcal{C} + \mathcal{D}_p > \mathcal{H} + \mathcal{T}_c + \mathcal{I}_c(t_c) + o \tag{5.22}$$

$$\mathcal{E} < \mathcal{H} + \mathcal{T}_c + \mathcal{I}_c(t_c) + o \tag{5.23}$$

$$P_c \geq \mathcal{H} + \mathcal{T}_c + \mathcal{I}_c(t_c + E(k_{nrc})E(t_{nrwin})) + E(k_c)(\mathcal{T}_r - \mathcal{D}_c) + E(k_{nrc})\mathcal{T}_{nrr} + o \tag{5.24}$$

$$P_v = \mathcal{E} > \frac{k_c}{c}\mathcal{H} + o \tag{5.25}$$

$$P_{nrc} = -\mathcal{T}_{nrc} - o < 0 \tag{5.26}$$

公式（5.20）中，限制了合约事务发布方，必须向结果发布方承诺一笔报酬，该报酬数目的覆盖范围应该能满足公式（5.20）与（5.21）的约束。公式（5.21）为合约事务结果发布者（通常是被挑战的计算成员）收益与理想开销的关系，收益应当能够覆盖预

期开销。$\mathcal{D}_c$约束了挑战者失败挑战中的押金，将会转移给被挑战者，减小被挑战的计算成员的开销。在该激励下，预期开销与本身的计算结果可靠性产生很大关系，可靠的计算结果，会减少收到的挑战，带来更高的收益。（5.22）中，约束了结果发布者若被挑战成功，事务发布者承诺的报酬，与结果发布者的押金，将成为挑战者的收益。（5.23）约束了挑战者若发起无效挑战，并不会获得正收益，这激励了挑战者的积极行为。对比公式（5.20）与（5.22），似乎挑战者收益高于结果发布者，不鼓励计算者积极发布事务结果。实际上，这两种可以获得正收益的角色，对比其他无法产生收益的成员，每一种角色都需要竞争得到。（5.24）约束了，成功的挑战者，将获得包括所有其他失败挑战者的押金。（5.25）表现了普通验证者计算压力的缩小，即控制成员可靠性，减少挑战轮次，可以有效缓解挑战者困境。（5.26）约束了无响应挑战者只会获得负收益，因为成功的无响应挑战会将合约事务失效，而不是被新结果替代。这种情况下，只会由有把握完成正式挑战的挑战者，会发布无响应挑战，靠其收回成本。这避免了大量伪造的无响应挑战进行投机，阻碍链上挑战协议的运行。

## 5.3 链上挑战协议的并行情况分析

因为链上挑战协议中，每一个 CICT 都对应了独立的计算组，各计算组之间互不干扰，因此 CICT 可以被良好地并行处理。假设任意 CICT 的计算消耗为$CC$，该事务的计算组成员数量为$gNum$，全部共识成员数量为$cNum$，全部执行节点数量为$eNum$。传统区块链模型中，所有成员都必须完整执行事务的计算内容。则该 CICT 想要获得共识必须消耗的网络算力$NCC$服从公式（5.27）。在链上挑战协议中，该网络算力消耗$NCC_{off-chain}$服从公式（5.28）。使用符号o标记过程中产生的其他计算开销，比如非对称加密验证等，因为计算密集型合约的计算消耗基数$CC$较大，该开销对$NCC$的影响几乎可以忽视。

$$NCC = (eNum + cNum) * CC + o \tag{5.27}$$

$$NCC_{off-chain} = (gNum + \frac{\hbar_c}{c} * cNum) * CC + o \tag{5.28}$$

两种方案的放大系数的大小容易进行比较。因为该事务的计算组成员是全部计算成员的子集，故$gNum$数值上恒不大于$eNum$。$\hbar_c$为一次链上挑战协议中发生的挑战次数，该数值必小于整个事务执行过程的等份切分数量$c$。得到$eNum + cNum \geq gNum + \frac{\hbar_c}{c} * cNum$，故$NCC \geq NCC_{off-chain}$，计算消耗放大情况能够被有效抑制。并且容易发现，该放大情况受 CICT 一次执行过程被等分的数量$c$的增加，以及计算组成员数量$p_G$的减少而减弱。事实上，等分数量$c$的调节幅度有限，因为$\frac{\hbar_c}{c}$的波动范围只能在 0 到 1 之间。而

且过大的取值会带来过多的中间数据，增大区块链对中间数据的存储压力，同样可能造成过多的额外计算。$c$可以在实践中逐步调优。而$gNum$的波动范围更大，带来的影响也更大。因此，在恰当的取值范围内，$gNum$的下降带来对算力消耗放大倍数会有多大的抑制效果，该效果在整个系统的 CICT 吞吐上会有多大程度的表现，是本节更关注的问题。

忽略网络性能、存储性能等方面可能会对区块链系统带来的吞吐瓶颈，记 CICT 在链下的执行吞吐为$TPS_{off-chain}$，平均一个 CICT 的算力消耗为$ACC$，平均一个计算成员单位时间能够提供的算力为$ACP$。链下事务执行吞吐可以表示为：

$$TPS_{off-chain} = \frac{ACP}{ACC} * \frac{eNum}{gNum} \tag{5.29}$$

CICT 的链下并行吞吐在公式（5.29）中有比较直观的展现。该吞吐会随$\frac{ACP}{ACC}$以及$\frac{eNum}{gNum}$的上升而上升，具体措施包括使用计算性能更强的设备、优化合约的计算开销、吸引更多的计算成员、减小交易对应计算组的成员数量等。但需要注意的是，过小的$gNum$可能会带来安全性危机，这个问题会在 5.4 节中被讨论。根据激励策略，在计算成员较为空闲时，为了获取更多收益，成员倾向于尽可能多的执行事务，$gNum$上升，CICT 的并行程度及吞吐会下降。但在网络负载较大、计算成员待执行事务较多的情况下，任何一个计算成员不可能执行全部事务，故倾向于只执行预期收益更高的事务，此时$gNum$下降，CICT 并行程度上升，吞吐上升。但$gNum$的下降过程中，不会存在计算分组过度聚集或过度稀疏的极端状况。因为对计算成员而言，某事务的计算组成员数量越小，对发布结果的竞争就越有优势，且通过纠正结果获得奖励的概率就越高，预期收益则越高。并行效率与安全能够有所保障。

## 5.4 链上挑战协议的安全性分析

在传统区块链中，共识策略能够保证事务的安全。但对基于链上挑战协议的链下计算策略来说，共识层的功能被拆分重组。因此，链下计算协议的安全性必须被重新讨论。本节对同时使用链上挑战协议与实用拜占庭容错共识算法（PBFT）方案的安全性进行了讨论，将该方案记作 occ-PBFT。本节进一步讨论了 occ-PBFT 与传统 PBFT 共识的安全性比较。本节讨论基于 PBFT 的一个基本性质，即当作恶节点数为$f$时，PBFT 共识能保证共识成员数在超过$3f+1$时的安全性。这是相关研究中的一个常见结论。

假设对任意一个事务，PBFT 过程有参与成员总数$Num$，其中共识成员数量为$cNum$，不参与共识的普通成员数量为$nNum$，数量关系满足$Num = cNum + nNum$。对 PBFT 方案来说，当作恶成员在共识委员会中数量超过$f$时，交易安全不再受到保障。容易计算，

保持容错的情况下$f$取值最大为$(cNum-1)/3$，记作$f_{max}=(cNum-1)/3$。假设作恶成员数量为$b$，则该假设下 PBFT 方案不安全概率为：

$$\begin{cases} 0\ ,(0\le b\le \lfloor f_{max}\rfloor) \\ \sum_{K=\lfloor f_{max}\rfloor+1}^{b}\frac{C_{cNum}^{K}C_{nNum}^{b-K}}{C_{Num}^{b}}\ ,(\lfloor f_{max}\rfloor<b\le nNum+\lfloor f_{max}\rfloor+1, b<cNum) \\ \sum_{K=\lfloor f_{max}\rfloor+1}^{cNum}\frac{C_{cNum}^{K}C_{nNum}^{b-K}}{C_{Num}^{b}}\ ,(\lfloor f_{max}\rfloor<b\le nNum+\lfloor f_{max}\rfloor+1, b>cNum) \\ 1\ ,(nNum+\lceil f_{max}\rceil<b\le Num) \end{cases} \tag{5.30}$$

相应地，假设对任意 CICT，occ-PBFT 方案涉及参与成员总数$Num'$，其中共识成员数量为$cNum'$。该事务的执行成员数量$gNum'$。然而，因为共识组与执行组不互斥，数量关系满足$Num'\le cNum'+gNum'$。当作恶成员在共识组中数量超过$f$时，交易安全不再受到保障。容易计算，保持容错的情况下$f$取值最大为$(cNum'-1)/3$，记作$f_{max}=(cNum'-1)/3$。假设作恶成员数量为$b$。对 occ-PBFT 方案来说，交易在以下两种情况会被认为不安全：

（1）执行成员全部是作恶成员。即执行组作恶成员数量和为$gNum'$。

（2）共识组有超过$(cNum'-1)/3$个作恶成员，且执行组有不少于 1 个且少于$gNum'$个作恶成员。

为了探索 occ-PBFT 方案的安全性，文章改变$Num'$、$cNum'$、$gNum'$以及$b$的数量关系，在每组取值下重复了十万次试验，以使每组实验的不安全概率的测量值向期望值更好地收敛。统计图中同样绘出了公式（5.30）所表示的 PBFT 不安全概率曲线。当 occ-PBFT 方案的不安全概率曲线并不明显高于 PBFT 的不安全性曲线时，认为 occ-PBFT 方案是安全的。结果被记录如下。

图 5.5 展示了具有相同共识成员数量与计算成员数量（或普通成员数量）时，两个方案在安全性上的承受能力。occ-PBFT 方案与纯 PBFT 方案有相同的共识成员数量 20，全部成员数量如图例所示，成员数量差值，PBFT 方案由普通成员补齐，occ-PBFT 方案由执行成员补齐。改变网络中不诚实成员数量，两方案的不安全概率变化如图所示。可以发现，当有相同的环境参数时，两方案的不安全概率几乎重合。所以，使用链上挑战协议，并不会使共识算法安全性下降。

图 5.6 展示了两方案安全性受共识成员数量或计算成员数量（或普通成员数量）的影响情况。在图 5.6（a）中，occ-PBFT 方案与纯 PBFT 方案有相同的共识成员数量 40，不诚实成员数量如图例所示，改变普通成员（或执行成员）数量，两方案不安全概率被记录如图。可以发现，同样环境参数时，occ-PBFT 方案的安全性与纯 PBFT 方案是几乎重合的，两者安全性会随成员数量的上升由明显提升。不同的是，在执行成员数量极少的情况下（曲线左端），两方案曲线并未重合，说明链上挑战协议，在执行成员极少时会为共识协议带来安全漏洞，该安全问题能够随执行成员数量增加而迅速解决。另外，执行成员过少的问题容易通过激励层策略改善。

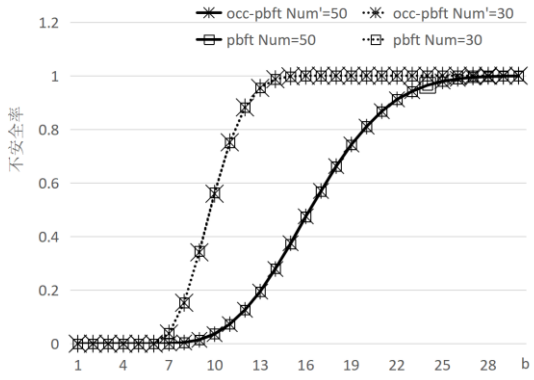
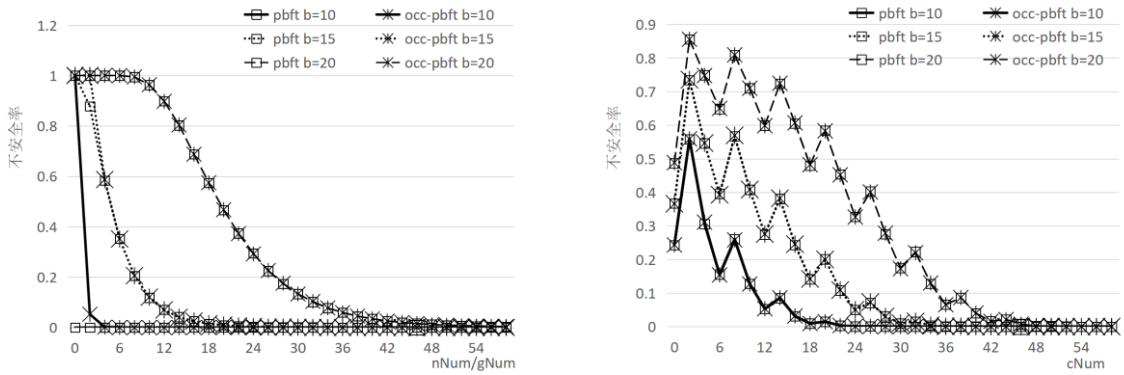



图 5.5 链上挑战协议安全性比较

Fig. 5.5 Security of On-Chain Challenge Protocols



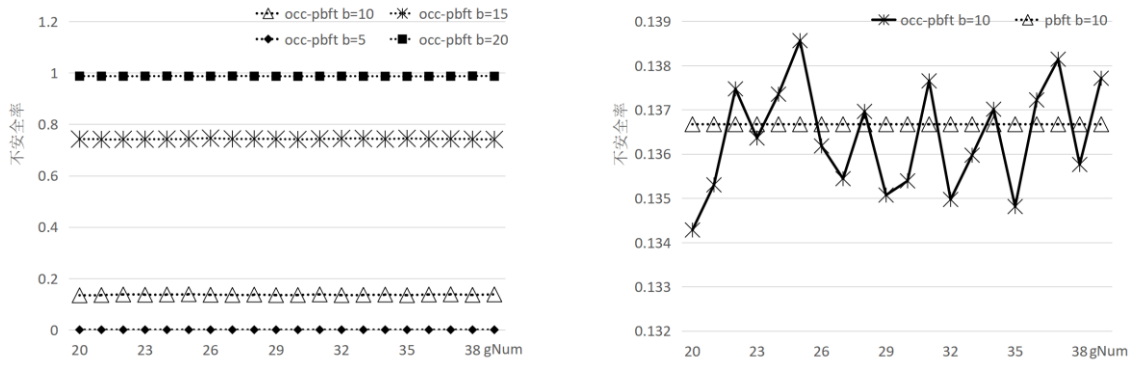

（a） (b)

图 5.6 成员数量对链上挑战协议的安全性影响

Fig. 5.6 Security Impact of Number of Members

在图 5.6（b）中，occ-PBFT 方案与纯 PBFT 方案有相同的计算成员（或执行成员）数量 40，不诚实成员数量如图例所示，改变共识成员数量，两方案不安全概率被记录如图。容易发现，两方案的安全性随共识成员数量的增加会明显提高。两方案曲线基本重合，说明任何共识成员数量下，链上挑战协议都不会使共识的安全性下降。

图 5.7 展示了链上挑战协议中，同一设备同时承担共识成员与计算成员两种角色带来的安全性影响。图 5.7（a）中保持共识成员数量 20，总成员数 40 不变，通过改变同时具备两种角色的设备的数量，测量其在不同恶意成员下的安全性表现。可以发现，在合理的成员数量下，双角色设备的出现不会带来明显的安全性改变。图（b）在更细的精度下展示了 occ-pbft 测量值与 pbft 理论安全曲线的拟合程度。在十万次实验量级上，两者的差距精度不高于 0.01。




（a） (b)

图 5.7 双角色成员的安全性影响

Fig. 5.7 Security Impact of Multi-Role Members

## 5.5 实验评估

### 5.5.1 实验环境

为了测试链上挑战协议的有效性，我们为两种角色的参与节点，配置了不同的计算性能。计算成员有四位，每个成员能够使用 8 个计算核心，共识成员有 4 位，只使用 2 核。除此之外，两者在存储能力上并无差别，使用了 16G 内存，并未进行数据持久化，排除了数据库与硬盘性能差异。我们同样选择了 3.2 节中的多元线性回归作为一种标准的 CICT，采用 4.5.2 节中仅并行梯度下降的合约编写方式，使智能合约更适配多核计算设备。在此前提下，我们测定了链下计算协议的并行效率、因资源冲突导致的事务失效问题情况以及结果到达延迟情况。相关实验结果与分析如下。

### 5.5.2 并行效率测试

为了测试链下计算策略在并行方面对 CICT 效率带来的的提升，我们将同样数量的合约事务负载施加在了两种区块链系统上。第一组实验不允许使用链上挑战协议；第二组实验使用链上挑战协议，但最多只允许产生两个并行的计算组；第三组实验使用链上挑战协议，最多允许产生八个并行的计算组。每个区块会打包当前收到的全部合法、已被验证的事务，因此区块链性能不会受区块大小影响。我们记录了三次实验在面对相同事务负载时的性能表现。能够发现，只要交易量不处于极少的水平，采用链上挑战协议能够有效减少区块链系统对 CICT 的处理用时。但是随着计算组数量的上升，验证用时并不总能获得对应比例的缩减，这可能受计算成员总数、共识成员性能、共识算法等诸多因素影响。

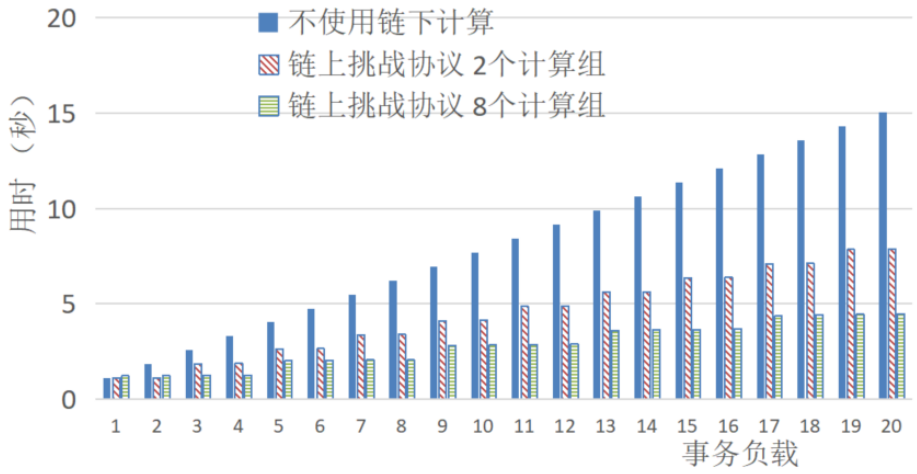



图 5.8 链上挑战协议并行效率对比

Fig. 5.8 Comparison of The Parallel Efficiency of On-Chain Challenge Protocols

### 5.5.3 事务失效情况测试

2.4.2 与 2.4.3 节中，基于交互式验证协议的“挑战-响应”型链下计算协议，会造成链上与链下数据短时间内的不一致。这段时间的不一致则会导致大量无效事务的出现。无效事务，会严重局限区块链吞吐量的提升。另外，区块链并不存在无效交易的反馈系统，因为交互式验证协议需要一个较长的等待时间，如果无效交易大量存在，将使事务发起人无法判断事务状态，从而失去对区块链链上资产的准确掌控。

因此，我们对“挑战-响应”型链下计算协议中可能会导致发生的无效交易数量做出了测试，并测试了链上挑战协议在该方面的表现。有三个重要指标可能会对无效交易数量产生重要影响，我们在此处做出说明。

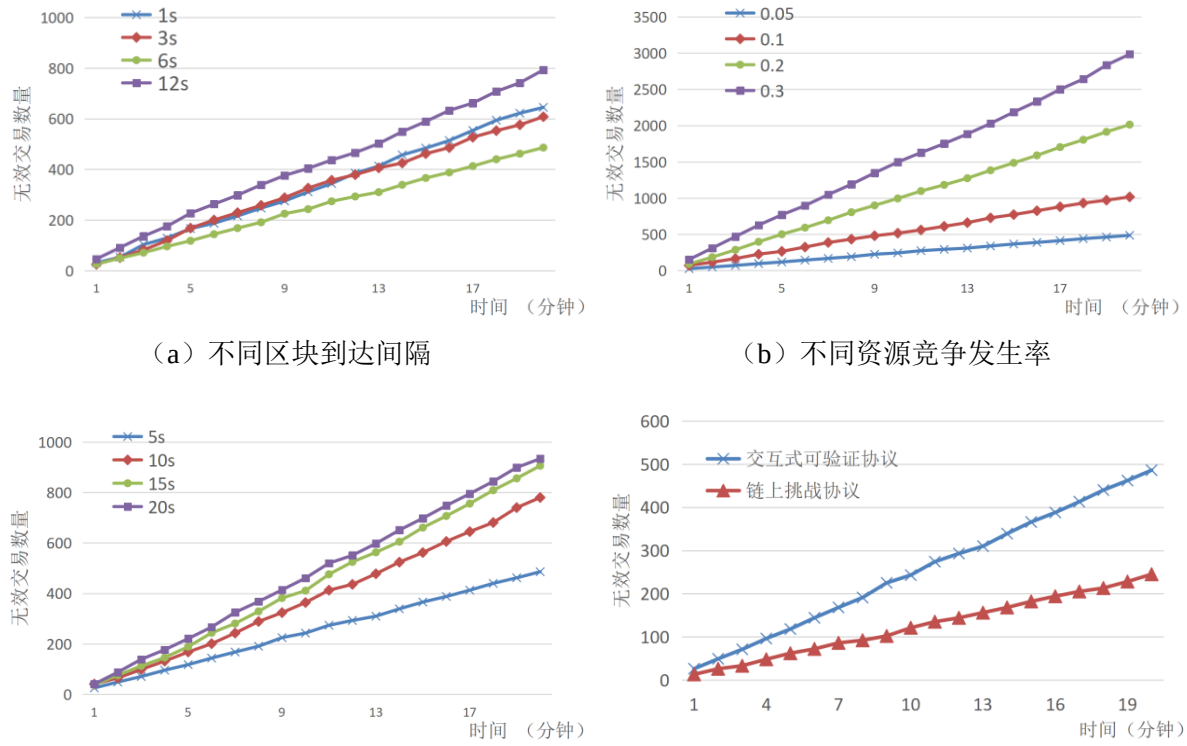
（1）区块到达间隔。区块与区块间的间隔时间可能会对无效数量有影响。

（2）资源竞争发生率。平均每百笔事务中，存在资源竞争的事务的数量。因为资源竞争是导致交易失效的根本原因，资源竞争发生率对无效交易竞争率可能有重要影响。

（3）挑战等待时间。交互式验证协议中，事务结果需要等到特定的挑战等待时间才能够生效。这一等待时间对链上链下数据不一致的持续时间有重要影响，因此猜测其对无效事务数量有重要影响。

实验结果如图 5.9 所示。图（a）中资源竞争发生率为 0.05，挑战等待时间为 5s，区块到达间隔如图例所示。显然，区块到达间隔对无效交易数量有影响，但并不存在简单的正比例或反比例关系，在 20 分钟后，无效交易数量差值仅在 310 笔左右。

图（b）中区块到达间隔为 6 秒，挑战等待时间为 5s，资源竞争发生率如图例所示，很明显，一段时间内无效交易数量与资源竞争发生率基本成正比例关系，而且受资源竞争发生率影响极大，对照组中，20 分钟最大差值达到 23500 左右。与其他实验中无效交易数量比较，容易发现资源竞争发生率是影响无效交易数量的最大因素。




（a）不同区块到达间隔　　（b）不同资源竞争发生率

（c）不同挑战等待时间　　（d）不同链下计算协议

图 5.9 不同环境下事务失效情况测试

Fig. 5.9 Transaction Failure with Different Environment Variables

图（c）中，资源竞争发生率为 0.05，区块到达间隔为 6 秒，挑战等待时间如图例。容易发现，随着挑战等待时长增加，无效交易数量也会增加，20 分钟对照组间无效交易数量差值在 450 左右。更短的挑战等待时间有利于减少无效交易，但是显然，这会为区块链安全带来隐患。从影响程度上来说，在资源竞争发生率已经确定的情况下，选择挑战等待时长时，可以不将无效交易数量作为最主要考量因素，而更多的考虑数据安全、网络情况等。

图（d）选择了对照组中较优的几个参数，包括资源竞争发生率 0.05、挑战等待时间 5s、区块到达间隔 6 秒。在该环境下，我们将交互式验证协议与链上挑战协议做了对比。结果表明，链上挑战协议能有效地减少链下计算过程中无效交易的数量。

### 5.5.4 资源阻塞测试

因为链上挑战协议中，资源无法在到达区块链后立即生效，后续事务将无法使用该资源，故会发生资源堵塞情况。本实验测量了链上挑战协议在未使用超前交付策略以及使用了超前交付策略两种情况下，比较了资源生效速度。实验设置竞争发生率 0.05、挑战等待时间 5s，改变出块间隔，测量了两个对照组中资源生效用时。

结果表明，不管区块到达间隔取值为多少，采用了超前交付策略，都能够有效释放资源，避免区块链阻塞。

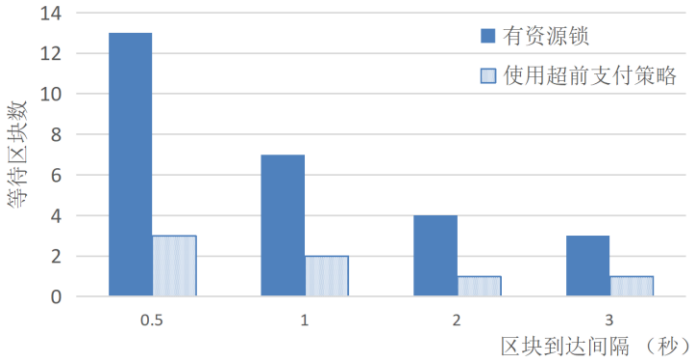



图 5.10 链上挑战协议资源阻塞情况

Fig. 5.10 Resource Logging Situation of On-Chain Challenge Protocol

## 5.6 本章小结

为了解决 CICT 算力消耗放大，区块链吞吐受限的问题，文章提出了一种链下计算协议——链上挑战协议。该协议使用了交互式验证协议作为可验证计算的基本策略，但是解决了交互式验证协议应用于区块链时存在的链上链下数据短期不一致的问题。链上挑战协议同样包含了对应的激励策略，能够保证链上挑战协议的良性运行；以及包含一种能够提前使用未确认资源的超前交付策略，能够提高资源的使用效率。另外，本章中提供了链上挑战协议的并行效率分析，以及安全性分析。本章通过实验，证明链下计算相比于传统方案，能通过事务并行大幅提高 CICT 的处理效率。实验同样探讨了交互式验证协议中无效交易的影响因素，并证明链上计算协议能够更有效地减少无效事务。最后，实验证明了文章提出的基于资源依赖图的超前交付策略能有效促进资源快速释放。

# 结　论

CICT 处理的低效，为区块链的性能瓶颈以及场景推广带来了很大的困难。通过使用并行策略，能够有效地缓解这一问题，实现区块链在处理 CICT 方面的更高吞吐。此处对文章内容做出三点总结。

（1）事务并行的两种分类，多核适配类并行与计算抑制类并行，能够将当前面向普通事务的区块链并行策略与 CICT 之间建立良好的关联，并且该种分类方法有一定概括性。通过该种分类方法能有效指导对 CICT 并行策略进行系统讨论，发现并分析事务在大规模并行中的缺陷。在将来的工作中，可以尝试使用该种分类方案指导进行 CICT 并行策略集成。

（2）支持重放确定性的多核适配智能合约，能够通过支持在区块链系统中进行并行编程，有效提升 CICT 的处理效率。实验证明，当采用 python 作为智能合约编程语言，处理多元线性回归这一典型 CICT 用例时，区块链事务处理效率能够提升超过 40%。本文主要关注和解决了合约并行编程中结果确定性的问题，后续研究可以进一步关注更细粒度的冲突检测方法，这是进一步提高并行效率的重要手段。另外，后续研究也可以与更多的编程语言尝试更深度的适配，效果可能会进一步提升。

（3）相比于直接在链上进行 CICT 的验证，链下计算能够有效实现事务并行执行，提升区块链系统的吞吐。而采用链上挑战协议，可以比直接使用交互式验证协议进一步减少超过一半的无效交易。链上挑战协议造成的资源阻塞，可以通过本文中的超前交付策略有效缓解。在 CICT 处理方面，链下计算的效果提升是客观的，该类方案同样可以在普通事务上进行尝试，但推测提升效果有限。

# 参考文献

[1]Zonyin Shae, Jeffrey Tsai. Transform blockchain into distributed parallel computing architecture for precision medicine[C]. 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), 2018: 1290-1299.

[2]Mastercard[EB/OL]. https://www.mastercard.co.in/en-in.html.

[3]VISA[EB/OL]. https://www.visa.com.hk/.

[4]Elli Androulaki, Artem Barger, Vita Bortnikov, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains[C]. Proceedings of the thirteenth EuroSys conference, 2018: 1-15.

[5]FISCO[EB/OL]. https://fisco-bcos-documentation.readthedocs.io/.

[6]OpenAI[EB/OL]. https://openai.com/.

[7]Yin Yang. Training Massive Deep Neural Networks in a Smart Contract: A New Hope[J]. arXiv preprint arXiv:2106.14763, 2021.

[8]Sourav Das, Vinay Joseph Ribeiro, Abhijeet Anand. YODA: Enabling computationally intensive contracts on blockchains with Byzantine and Selfish nodes[J]. arXiv preprint arXiv:1811.03265, 2018.

[9]Christian Liu, Peter Bodorik, Dawn Jutla. A Tool for Moving Blockchain Computations Off-Chain[C]. Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure, 2021: 103-109.

[10]施建锋, 吴恒, 高赫然, et al. 区块链智能合约并行执行模型综述[J]. 软件学报, 2021: 0-0.

[11]Zryan Najat Rashid, Subhi Rm Zebari, Karzan Hussein Sharif, et al. Distributed cloud computing and distributed parallel computing: A review[C]. 2018 International Conference on Advanced Science and Engineering (ICOASE), 2018: 167-172.

[12]Weiwei Chen, Xu Han, Che-Wei Chang, et al. Out-of-order parallel discrete event simulation for transaction level models[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2014, 33(12): 1859-1872.

[13]Wei Yu, Kan Luo, Yi Ding, et al. A parallel smart contract model[C]. Proceedings of the 2018 International Conference on Machine Learning and Machine Intelligence, 2018: 72-77.

[14]Huma Pervez, Muhammad Muneeb, Muhammad Usama Irfan, et al. A comparative analysis of DAG-based blockchain architectures[C]. 2018 12th International conference on open source systems and technologies (ICOSST), 2018: 27-34.

[15]Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, et al. Adding concurrency to smart contracts[J]. Distributed Computing, 2020, 33(3): 209-225.

[16]Mohammad Javad Amiri, Divyakant Agrawal, Amr El Abbadi. Parblockchain: Leveraging transaction parallelism in permissioned blockchain systems[C]. 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019: 1337-1347.

[17]Karl Wüst, Sinisa Matetic, Silvan Egli, et al. ACE: asynchronous and concurrent execution of complex smart contracts[C]. Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020: 587-600.

[18]Jason Teutsch, Christian Reitwießner. A scalable verification solution for blockchains[J]. arXiv preprint arXiv:1908.04756, 2019.

[19]Mingming Wang, Qianhong Wu. Lever: Breaking the shackles of scalable on-chain validation[J]. Cryptology ePrint Archive, 2019.

[20]Weiqin Zou, David Lo, Pavneet Singh Kochhar, et al. Smart contract development: Challenges and opportunities[J]. IEEE Transactions on Software Engineering, 2019, 47(10): 2084-2106.

[21]Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, et al. Chainspace: A sharded smart contracts platform[J]. arXiv preprint arXiv:1708.03778, 2017.

[22]Solidity[EB/OL]. https://solidity.readthedocs.io.

[23]Jiaping Wang, Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones[C]. 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), 2019: 95-112.

[24]Christian Cachin. Architecture of the hyperledger blockchain fabric[C]. Workshop on distributed cryptocurrencies and consensus ledgers, 2016: 1-4.

[25]Ilham A Qasse, Manar Abu Talib, Qassim Nasir. Inter blockchain communication: A survey[C]. Proceedings of the ArabWIC 6th Annual International Conference Research Track, 2019: 1-6.

[26]C Mohan, Bruce Lindsay, Ron Obermarck. Transaction management in the R* distributed database management system[J]. ACM Transactions on Database Systems (TODS), 1986, 11(4): 378-396.

[27]Emrah Sariboz, Kartick Kolachala, Gaurav Panwar, et al. Off-chain Execution and Verification of Computationally Intensive Smart Contracts[C]. 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC), 2021: 1-3.

[28]Mohamed Seifelnasr, Hisham S Galal, Amr M Youssef. Scalable open-vote network on ethereum[C]. International Conference on Financial Cryptography and Data Security, 2020: 436-450.

[29]王昭淼. 面向多线程程序的确定性重演研究[D]. 大连理工大学, 2016.

[30]Thomas J. Leblanc, John M. Mellor-Crummey. Debugging parallel programs with instant replay[J]. IEEE Transactions on Computers, 1987, 36(4): 471-482.

[31]Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, et al. Pres: probabilistic replay with execution sketching on multiprocessors[C]. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009: 177-192.

[32]Min Xu, Rastislav Bodik, Mark D Hill. A" flight data recorder" for enabling full-system multiprocessor deterministic replay[C]. Proceedings of the 30th annual international symposium on Computer architecture, 2003: 122-135.

[33]Min Xu, Mark D Hill, Rastislav Bodik. A regulated transitive reduction (RTR) for longer memory race recording[J]. ACM SIGARCH Computer Architecture News, 2006, 34(5): 49-60.

[34]NFT.storage[EB/OL]. https://nft.storage/.

[35]Joseph Poon, Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[36]Rainer Böhme, Nicolas Christin, Benjamin Edelman, et al. Bitcoin: Economics, technology, and governance[J]. Journal of economic Perspectives, 2015, 29(2): 213-38.

[37]Loi Luu, Jason Teutsch, Raghav Kulkarni, et al. Demystifying incentives in the consensus computer[C]. Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, 2015: 706-719.

[38]Zibin Zheng, Shaoan Xie, Hongning Dai, et al. An overview of blockchain technology: Architecture, consensus, and future trends[C]. 2017 IEEE international congress on big data (BigData congress), 2017: 557-564.

[39]Guoxing Chen, Sanchuan Chen, Yuan Xiao, et al. Sgxpectre attacks: Leaking enclave secrets via speculative execution[J]. arXiv preprint arXiv:1802.09085, 2018.

[40]K Rustan M Leino, Nadia Polikarpova. Verified calculations[C]. Working Conference on Verified Software: Theories, Tools, and Experiments, 2013: 170-190.

[41]Sanjay Jain, Prateek Saxena, Frank Stephan, et al. How to verify computation with a rational network[J]. arXiv preprint arXiv:1606.05917, 2016.

[42]薛锐, 吴迎, 刘牧华, et al. 可验证计算研究进展[J]. 中国科学: 信息科学, 2015, 45(11): 1370-1388.

[43]Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, et al. Software grand exposure:{SGX} cache attacks are practical[C]. 11th USENIX Workshop on Offensive Technologies (WOOT 17), 2017.

[44]Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, et al. Succinct {Non-Interactive} Zero Knowledge for a von Neumann Architecture[C]. 23rd USENIX Security Symposium (USENIX Security 14), 2014: 781-796.

[45]Harry Kalodner, Steven Goldfeder, Xiaoqi Chen, et al. Arbitrum: Scalable, private smart contracts[C]. 27th USENIX Security Symposium (USENIX Security 18), 2018: 1353-1370.

[46]Shihab Shahriar Hazari, Qusay H Mahmoud. A parallel proof of work to improve transaction speed and scalability in blockchain systems[C]. 2019 IEEE 9th annual computing and communication workshop and conference (CCWC), 2019: 0916-0921.

[47]Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman. Concurrency control and recovery in database systems[M]. 370. Addison-wesley Reading, 1987.

[48]Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, et al. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack[J]. Journal of Cases on Information Technology (JCIT), 2019, 21(1): 19-32.

[49]Peilin Zheng, Quanqing Xu, Zibin Zheng, et al. Meepo: Sharded consortium blockchain[C]. 2021 IEEE 37th International Conference on Data Engineering (ICDE), 2021: 1847-1852.

[50]Wellington Fernandes Silvano, Roderval Marcelino. Iota Tangle: A cryptocurrency to communicate Internet-of-Things data[J]. Future generation computer systems, 2020, 112: 307-319.

[51]李芳, 李卓然, 赵赫. 区块链跨链技术进展研究[J]. Journal of Software, 2019, 30(6): 1649-1660.

[52]Gwan-Hwan Hwang, Po-Han Chen, Chun-Hao Lu, et al. InfiniteChain: A multi-chain architecture with distributed auditing of sidechains for public blockchains[C]. International Conference on Blockchain, 2018: 47-60.

[53]Yizhong Liu, Jianwei Liu, Marcos Antonio Vaz Salles, et al. Building Blocks of Sharding Blockchain Systems: Concepts, Approaches, and Open Problems[J]. arXiv preprint arXiv:2102.13364, 2021.

[54]Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, et al. Omniledger: A secure, scale-out, decentralized ledger via sharding[C]. 2018 IEEE Symposium on Security and Privacy (SP), 2018: 583-598.

[55]Jacob Eberhardt. Scalable and privacy-preserving off-chain computations[J], 2021.

[56]Cheng Xu, Ce Zhang, Jianliang Xu, et al. SlimChain: scaling blockchain transactions through off-chain storage and parallel processing[J]. Proceedings of the VLDB Endowment, 2021, 14(11): 2314-2326.

[57]Kenneth L Judd. Computationally intensive analyses in economics[J]. Handbook of computational economics, 2006, 2: 881-893.

[58]Paul J Darwen. Computationally intensive and noisy tasks: co-evolutionary learning and temporal difference learning on backgammon[C]. Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512), 2000: 872-879.

[59]Giovanni Agosta, Francesco Bruschi, Donatella Sciuto. Static analysis of transaction-level models[C]. Proceedings of the 40th annual Design Automation Conference, 2003: 448-453.

[60]Nir Shavit, Dan Touitou. Software transactional memory[J]. Distributed Computing, 1997, 10(2): 99-116.

[61]Parwat Singh Anjana, Sweta Kumari, Sathya Peri, et al. An efficient framework for optimistic concurrent execution of smart contracts[C]. 2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2019: 83-92.

[62]Parwat Singh Anjana, Hagit Attiya, Sweta Kumari, et al. Efficient concurrent execution of smart contracts in blockchains using object-based transactional memory[C]. International Conference on Networked Systems, 2020: 77-93.

[63]Maurice Herlihy, Victor Luchangco, Mark Moir, et al. Software transactional memory for dynamic-sized data structures[C]. Proceedings of the twenty-second annual symposium on Principles of distributed computing, 2003: 92-101.

[64]Sukrit Kalra, Seep Goel, Mohan Dhawan, et al. Zeus: analyzing safety of smart contracts[C]. Ndss, 2018: 1-12.

[65]Arthur Gervais, Ghassan O Karame, Karl Wüst, et al. On the security and performance of proof of work blockchains[C]. Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016: 3-16.

[66]Zibin Zheng, Shaoan Xie, Hong-Ning Dai, et al. Blockchain challenges and opportunities: A survey[J]. International Journal of Web and Grid Services, 2018, 14(4): 352-375.

[67]Pavel Gladyshev, Ahmed Patel. Finite state machine approach to digital event reconstruction[J]. Digital Investigation, 2004, 1(2): 130-149.

# 致　　谢

硕士阶段的学习即将结束，值此时刻，向所有关爱过我的长辈、同学、朋友、亲人，表示由衷的感谢！

感谢***老师、***老师在我求学阶段对我的帮助与鼓励。老师在生活中的和蔼友善，工作中的敬业勤奋，是我终生学习的榜样。也感谢实验室的其他老师对我生活中的照顾，从实验室环境的维护到活动的筹备，耗费了老师不少的心血，我与同学们都铭感在心。也感谢对我大论文、小论文提供过意见的素未谋面的老师们，感谢您辛苦阅读我的工作，提供宝贵的意见，这是我们学生进步的不竭动力。

感谢对我有过帮助、照顾的师兄师姐，师弟师妹，以及同级的小伙伴们。我在枯燥的研究生活中，获得了你们带来的帮助与启发，也获得了快乐与鼓励。我们珍贵的友谊是我一生的财富。

感谢我的父母，对我生活中的所有决定，都提供无私的支持，你们是我最大的后盾，永远给我力量和勇气。也感谢我的妻子，感谢你对我学习工作的支持与理解。我时常觉得对你的陪伴太少，而你给我的鼓励和理解却多的多。

感谢所有对我有过善意的人们，望你们生活幸福，万事顺遂。

最后，祝愿疫情早日结束。

祝愿世界人民团结友爱，不受战争伤害。

祝愿祖国繁荣昌盛。

# 附　　录

# 攻读学位期间发表的学术论文目录

（1）NFT Content Data Placement Strategy in P2P Storage Network for Permissioned Blockchain, International Conference on Parallel and Distributed Systems (ICPADS CCF-C), 第一作者

（2）StateSnap: A State-Aware P2P Storage Network for Blockchain NFT Content Data, International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP CCF-C), 第二作者

（3）A Secure and Efficient Atomic Cross-Chain Commitment, Information Sciences (CCF-B), 第二作者, Under Review

（4）一种区块链的跨链交互方法, 发明专利, CN202110438931.X

（5）区块链大文件副本选址方法、系统、设备及存储介质，发明专利，CN202011525557.9