

深圳大学

本科毕业论文（设计）

题目：对提高以太坊实际处理能力的设想

姓名：肖炆

专业：通信工程

学院：信息工程

学号：2014130099

指导教师：张胜利

职称：教授

2018 年 04 月 20 日

深圳大学本科毕业论文（设计）诚信声明

本人郑重声明：所呈交的毕业论文（设计），题目《 **对提高以太坊实际处理能力的设想** 》是本人在指导教师的指导下，独立进行研究工作所取得的成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式注明。除此之外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。本人完全意识到本声明的法律结果。

毕业论文（设计）作者签名：

日期： 年 月 日

摘要(关键词)	1
1. 引言	1
2. 问题与挑战	1
2.1 信任问题。	1
2.2 合约成本问题	1
2.3 公开透明问题	2
2.4 无间断运作问题	2
2.5 其他问题	2
3. 技术背景	2
3.1 比特币与区块链	2
3.2 区块链的处理能力	2
3.3 提高区块链实际处理能力的已有方案	3
4. 区块链原理及现状	3
4.1 比特币工作简要概述	3
4.1.1 比特币工作流程	3
4.1.2 比特币处理能力	4
4.2 以太坊简单介绍	4
4.2.1 以太坊改进	5
4.3 目前困境和现有解决手段	6
4.3.1 闪电网络	6
4.3.2 Plasma	7
4.3.3 分片	7
4.3.4 链上扩容	8
4.3.5 共识	8
5. 以太坊处理能力	9
5.1 以太坊处理能力的表现形式	9
5.2 具体影响要素	9
6. 基于 evm 的改进	9
6.1 evm 原理	9

6.2 evm 层次提高处理能力的原因.....	10
6.3 256bit.....	10
6.3.1 使用 256bit 字长的原因.....	10
6.3.2 256bit 字长的缺点.....	10
6.3.3 evm 的 64bit 模式.....	11
6.3.4 evm 在 64bit 下的问题.....	13
6.4 内存复用.....	13
6.4.1 evm 内存布局.....	13
6.4.2 内存回收问题.....	13
6.4.3 提供内存回收的字节码.....	13
6.4.4 evm 内存复用的隐患.....	14
6.5 确定性、调度表与并行.....	14
6.5.1 以太坊的确定性.....	14
6.5.2 确定性与串行执行.....	15
6.5.4 evm 的调度表.....	15
6.5.5 调度表的缺陷.....	21
6.6 其他.....	21
7. 基于合约代码的改进.....	21
7.1 智能合约原理.....	21
7.2 合约代码层次提高处理能力的原因.....	21
7.3 solidity.....	22
7.4 标准库.....	22
7.4.1 标准库的必要性.....	22
7.4.2 实现标准库的方法.....	22
7.4.3 标准库目前的困境.....	23
7.5 assembly.....	23
7.5.1 solidity 的缺陷.....	23
7.5.2 assembly 的使用.....	23
7.5.3 直接使用 assembly 的隐患.....	24

7.6 其他.....	24
8. 结语.....	24
参考文献.....	25
致谢.....	26
Abstract (Key words).....	27

对提高以太坊实际处理能力的设想

信息工程学院通信工程 肖炀

学号: 2014130099

【摘要】区块链技术为去中心化的价值转移、不可篡改的分布式账本提供了可能性,而以太坊在此基础上构建了一个去中心化、分布式的合约平台,用以自动、安全、方便地部署与执行合约。但作为一个全球化的分布式结算系统,以太坊对事务的处理能力比较低下,难以满足现在日益增长的事务。本文先是介绍了区块链和以太坊的基本概念和工作原理,再简要描述了目前社区对提高区块链处理能力的改进方案,最后提出几个用以提高以太坊实际处理能力的设想。

【关键词】以太坊;处理能力;evm;solidity

1. 引言

当今世界经济发展迅猛,全球化互联网的进一步深入世界数字化经济运作的每一个层次,如电子商务平台、网络社交平台、数字结账平台、金融产品交易平台等。以电子商务为例,2017年上半年的中国短短的6个月内,网络零售总额就已经达到了30229亿人民币^[1],这超过了包含购物节双11、黑色星期五在内的2014年全年总额,不断地刷新着零售总额记录。

这意味着世界运作对技术实现本身提出了新的挑战和要求,虽然这些新问题与挑战在传统运作尚未得到有效的解决,但是借助区块链的力量这些问题造成的影响将会有效地缓解。

2. 问题与挑战

2.1 信任问题。

传统信任组成上有着来自客户信任的中心化机构的潜在威胁,客户信任的中心化机构可能会恶意危害客户,也可能自身无意之下犯错从而导致客户受到损失。

例如商品交易平台,用户的交易数据全部存放到平台服务器上,显然该交易平台的管理人有着对交易数据的控制和修改能力,而平台管理人存在为了自身利益或其他非法目的而篡改了客户数据或相关账目的可能,若出现这种恶意行为客户难以发现和维权。

2.2 合约成本问题

传统合约的签订与执行都需要多方参与,导致步骤复杂,时间和金钱成本都很大。

例如某商业合作需要签订合约或订单,需要多方法务人员参与和实施,还需要公证机构和仲裁机构是合约具备法律效力,而合约的执行也需要人员操作,倘若出现违约也要通过司法手段维权,这些流程不可避免地产生非常大的时间和金钱成本,对客户造成不必要的损失。

2.3 公开透明问题

传统数据一般保存在中心化的机构中，难以有说服力地披露真实的数据。

例如某公示系统，由于账目和其他数据都在中心化的机构控制下，即使该机构没有恶意行为也没有出现无意错误，公众也可能会对其公示系统所展示的内容的真实程度持怀疑态度。

2.4 无间断运作问题

传统结算系统因为更新维护等需求，不可避免地需要暂停对外提供服务。

例如银行结算系统，会对不同级别的业务做不同长度的周期性结算，而在这些周期性结算时可能会暂停对外提供服务，这对于全球化的业务或其他要求零停机的业务而言是不利的。

2.5 其他问题

除此之外，还存在地理障碍成本、隐私保护等等挑战，而这些问题在区块链技术的发展下已经得到了初步的改善，并且随着区块链技术的继续改进，相信会有更多的问题都能得到充分解决。

3. 技术背景

3.1 比特币与区块链

在 2008 年有一个自称“中本聪”的 p2pfoundation 论坛成员在 metzdowd.com 的密码学公开邮件列表上发表了论文“A peer-to-peer electronic cash system”^[2]，该论文简要阐述了一种技术，利用密码学、大规模共识、p2p 网络、token 激励等技术，让参与组成网络中的大部分成员共同维护一个不可篡改的、公开的、去中心化的分布式账本，并命名为“比特币”(bitcoin)，之后社区抽象出 bitcoin 的技术轮廓并命名该类型为“区块链”(blockchain)。

Bitcoin 作为第一代区块链技术的实现，对后世的区块链技术的发展有着非常深远的意义。Bitcoin 主要用途是不可篡改的 token 转移，而之后的区块链技术逐渐发展，已经被运用在不同的领域上，如无地理限制的支付、信用凭证、产品溯源、自动合约(智能合约)、无停机运作、分布式冗余存储等方面上。

3.2 区块链的处理能力

区块链的实现如 bitcoin、ethereum 等虽然是全球化的分布式系统，但是 tps(transactions per second)却非常的低下，bitcoin 的 tps 一般在 1~2，而以太坊(ethereum)的 tps 一般在 5~15，对于一个受众如此之多，应用前景广阔的应用而言，这种程度的事务处理能力显然是远远不够的。

3.3 提高区块链实际处理能力的已有方案

为了提高区块链的实际处理能力，开发者团队和技术社区提出了非常多的改进方案，方向可以大致分为两种：

1. 链上，主要思想是在区块链本身的链上增加 tps，如增大区块链在单个区块上的 tx(transaction) 的容量，或者增加共识速度，或使用 pos、dpos、pbft 来达成共识，或使用 dag 而不是单条链等，这样做可以直接增强区块链的处理能力，但是会导致全节点的负担变大。

2. 链下，主要思想是把部分应该存在在区块链上的 tx 移至区块链外，如闪电网络、侧链、sharding、plasma 等，牺牲部分的安全来换取区块链更强的处理能力，让用户灵活的在安全与成本之间权衡利弊，从而间接地提高以太坊的实际处理能力。

虽然以上方案都有着坚实的理论基础，但是很多都还在推论或开发中，离实际应用落地还需要很多研究测试，而且如果希望使用以上的方案很可能需要硬分叉从而导致社区分裂，又或者部署成本大而难以实现。本文提出几个提高以太坊实际处理能力的想法，如在 evm、合约代码层次上提高以太坊处理能力，是代价相对较小的改进想法。

4. 区块链原理及现状

区块链本质上是一种分布式数据库，该数据库以账本的形式出现，被 p2p 网络里的全节点共同维护，通过 token 激励来促使整个 p2p 网络为某种目的工作。

4.1 比特币工作简要概述

这里以 bitcoin^[2] 为例简要概述区块链运行机制，bitcoin 是最早的开源的区块链的实现，对区块链后来的发展起到重要的指引和启发作用。

4.1.1 比特币工作流程

1. bitcoin 以区块为单位链状延伸，每个块头都带有上一个块头数据的哈希，从一个固定创世区块(genesis block)出发，一直延伸到最新的区块。

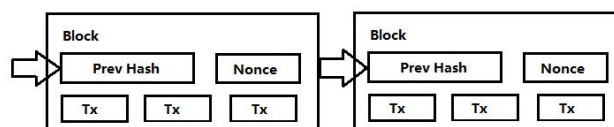


图 1 bitcoin 区块连接图

2. 每个区块都有区块头和区块数据，区块头包含了该区块的元数据，如版本号、上一个区块的 sha256 哈希值、merkle 根、时间戳、难度目标、nonce，而区块数据则放置了该区块的所有 tx 的内容。

3. 每个地址由 27~34 个数字或字母组成，地址由公钥经过多重哈希运算和签名得出，而拥有对应的私钥则会拥有对该地址上所有 token 的实际控制权。

4. 私钥可以对一个锁定在相对应的地址上的 token 解锁，并签名一个新的 tx 转移 token 到一个新的锁定脚本的地址上，从而实现 token 安全的、无法篡改的价值转移。

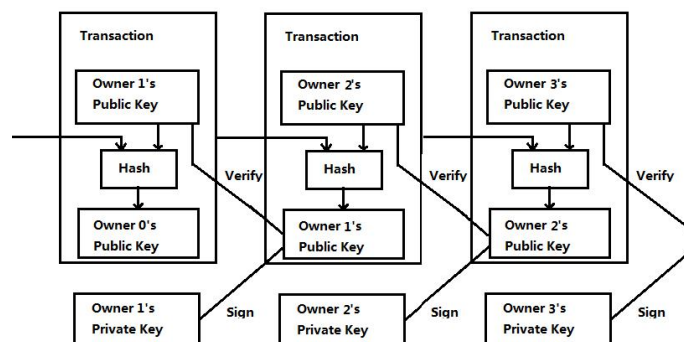


图 2 transaction (简称 tx) 工作示意图

5. 在这个 p2p 网络中每个全节点(拥有该区块链所有数据或能够直接获取这些信息的节点)都有着创建一个新块的权利，为了拿到创造区块的 token 奖励而尝试创造新块的全节点一般称之为“矿工”。

6. 矿工收集 p2p 网络上广播的 tx, 验证并记录在区块的数据部分, 再通过不断修改 nonce 值以达到难度目标的要求以创建一个 pow 证明, 成功创造出 pow 证明的矿工则广播该块以使全网达成这个最新的块的共识。

7. 难度目标会随着之前的出块时间动态调整, 在 bitcoin 上是每 2016 个块做一次调整, 从而使出块的时间稳定在一个值左右。

4.1.2 比特币处理能力

对 bitcoin 而言, 出一个块的时间一般在 10 分钟左右, 习惯上如果一个 tx 被某个块包含, 而这个块已经延伸出 5 个新块时, 认为这个 tx 已经被确认了 6 次(包括了包含该 tx 的块)且被最终确认了, 也就是说一次“确认”的交易数学期望上需要长达一个小时的等待时间, 这意味着仅仅凭借 bitcoin 在链上的结算, 远远不能满足日常交易结算的要求。

4.2 以太坊简单介绍

以太坊(ethereum)是程序员 Vitalik Buterin 从 bitcoin 中受到启发于 2014 年正式开发发展起来的第二代区块链技术。VB 早期参与 bitcoin 社区时发现 bitcoin 迫切需要一种有效的、安全的新机制来构建区块链上的应用, 为了构建这个去中心化的应用平台, VB 于 2014 年开始募集支持并发表了论文“A next-generation smart contract and decentralized application platform”^[3], 该论文描述了一种新的区块链技术, 可以让开发者在这条链上简便地开发去中心化的应用同时保护网络中的节点不受恶意代码的攻击。

4.2.1 以太坊改进

以太坊和 bitcoin 是区块链的不同实现, 相对于 bitcoin 而言, 以太坊继承了第一代区块链技术 bitcoin 优势的同时, 实现原理更复杂也更先进。相比于 bitcoin, 以太坊有以下改进^[4]:

1. 以太坊 tx 不需要凑足多个 utxo(unused transaction output)来构成期望的输出(output), 与之相对应的是以太坊每个地址都保留了该地址所拥有的以太币(ether)的数量, 保留了传统结算系统的随意转账的灵活性。

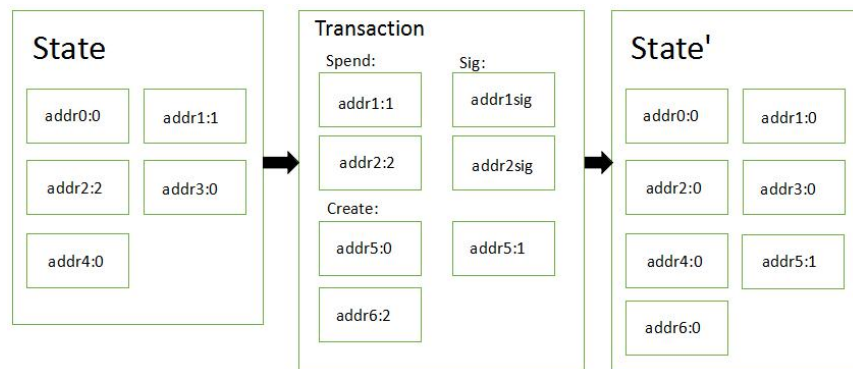


图3 以太坊 tx 与 state 关系图

2. 以太坊减少了共识时间, 出块平均时间大约为 15s, tps 大约为 5~15, 相对于 bitcoin 有着更强的事务处理能力。习惯上以太坊上一个 tx 经过 12 个确认后可以认为已经被最终确认, 这意味着在以太坊上一个最终确认 tx 大约需要 180s (3min), 3 分钟的最终确认延迟相对于比特币已经有非常大的提升。

3. 以太坊支持智能合约功能, 智能合约允许开发者通过使用图灵完备的、易用的高级语言(如 solidity)编写合约代码, 合约代码会被部署到链上而被每一个全节点保存, 对网络中的任意一个成员而言合约代码都是公开透明的, 因为共识的约束, 合约代码本身将无法被越权恶意篡改, 且消耗少量 gas 调用合约函数时合约将被强制执行, 从而实现合约的公开强制执行。

4. 以太坊中每个地址都对应了一个账户, 地址由 160bits 组成, 每个账户都有 nonce、balance、storageRoot、codeHash。账户分为为外部地址和合约地址, 外部地址可以理解为普通的用户地址, 只需拥有私钥则拥有外部地址的控制权, 合约地址则拥有合约代码, 对合约的操作和约束只能由代码来决定。

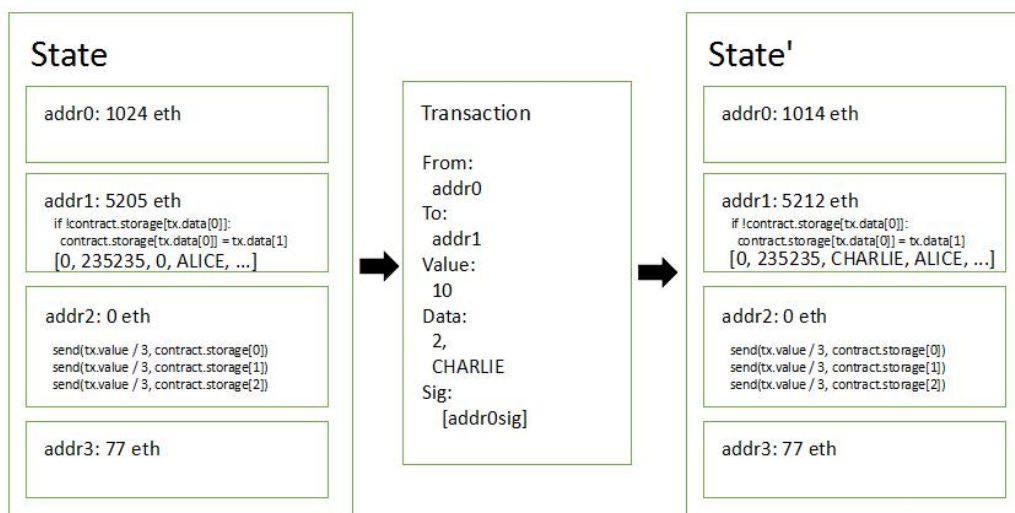


图 4 合约地址的 tx 示意图

5. 以太坊具有 Asic (application specific integrated circuit) 抵抗能力, 早期区块链想要得到一个 pow 证明, 只需大量运算设备和电力, 这导致了矿场通过购买大量 asic 矿机从而控制整个网络, 这让网络存在“51%”攻击导致双花的风险。而以太坊中想要得到一个 pow 证明, 不仅需要运算设备和电力, 还需要与之匹配的内存读取速度 (如为每个运算设备附加内存和内存读写总线), 这大大减轻了传统 asic 的优势, 从而在一定程度上维护了区块链的去中心化特征。

除此之外, 以太坊还有许多改进, 这些改进融合了不同区块链实现的思想和技术, 同时又把实现逻辑抽象化从而保留了继续自我改进的机会。

4.3 目前困境和现有解决手段

目前大部分区块链 tps 比较低, 最终确认时间长, 其实际处理能力与日常生活零售的结算速度要求还有很大距离, 上面已经简要论述了提高其实际处理能力的两个方向, 这里详细介绍几个社区提出的、可行有效的提高处理能力的方法。

4.3.1 闪电网络

闪电网络 (lightning networks) 概念出自 2016 年 Joseph Poon 的论文 “The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments”^[5]。闪电网络是通过巧妙的锁定脚本或合约设计一种把存在于链上的交易移动至链下的技术。闪电网络最早是一种用于 bitcoin 上的改进技术, 后来以太坊社区参考其核心思想, 结合以太坊的实际情况对其优化改进为雷电网络 (raiden networks)。

为了实现闪电网络, 有两个核心问题需要解决:

1. 如何安全地把交易存放在链外
2. 如何让交易存在中转

闪电网络在没有改变已有共识的前提下提出一种巧妙的手段解决了这两个问题, 这得益于 RSMC 和 HTLC。

1. RSMC

RSMC (Recoverable Sequence Maturity Contract)，首先要解决 tx 如何发生在链下却具备及时返回链上的能力。A 与 B 试图在链下交易，A 与 B 各自解锁一个（或多个）utxo 并 output 到一个特殊的锁定脚本上，此时该新的 utxo 为一个资金池，A 若要转一个单位 token 给 B，则资金池 A 减去 1 而 B 加上 1。

要解锁该 utxo 需要 A 和 B 签名，A 和 B 每次（包括第一次开启资金池）交易时，A 做出签名并 output 属于 A 自己的资金、属于 B 的带延时体现的资金，且属于 B 的资金押上一个承诺（hash 值），若能提供一个 secret 经过 hash 得到这个承诺则可以提现 B 的资金，最后把这个半签名 tx 链下给 B。与之相对的，B 也做对称行为。每次出现新的链下交易 AB 都相互公开 secret 以作废之前的链下交易，若想退出则把对方的半签名 tx 做自己的签名并广播到链上以结算。

2. HTLC

HTLC (Hashed Timelock Contract)，之后要解决链下中转支付的问题。假设 A 与 B，B 与 C 已经开启 RSMC，那么 A 与 C 可以间接线下交易。A 为了线下转账给 C 1 个 token，A 先选择一个 secret 并生成其哈希值 commitment，公开 commitment，B 与 C 在 RSMC 中达成协议（以下省略 RSMC 流程，假设 B 与 C 直接交易），若 C 能在时限内拿到一个 secret，它的 hash 值等于一个承诺（commitment），则 C 获得一个 token，否则该 token 归还给 B，同理 A 与 B 签订同一个协议（特殊的锁定脚本）。A 在观察这两个协定是否已经签订，若已经签订，则告诉 C 真正的 secret，C 从 B 得到 1 个 token，此时 secret 公开，B 也可以从 A 得到 1 个 token，完成中转转账。

4.3.2 Plasma

与闪电网络一样，Plasma^[6]主要思想也是通过把大量不那么重要的 tx 移至链下，通过部署主链上 plasma 合约，构建新的 plasma 链（可以使用新的 token 激励），定期提交子链区块头哈希，同时允许子链随时广播一个对父链提现的证明以提现（有可被提供 fraud proof 的公告期），从而提高整个网络的 tps。

对于 plasma 而言，有以下注意事项：

1. 子链的安全性本质上依赖与 root 链，虽然 plasma 理论上可以分很多层以大大增加实际 tps，在过深的 plasma 链上却有着更大的欺诈和区块扣押风险。

2. 虽然子链退出时的公告期可以提供 fraud proof 在博弈上会让尝试欺诈的人承受巨大风险，但依然存在可能性，恰好在公告期没有人观察是否存在 fraud，这会导致该欺诈被正式提交到父链。

3. 和闪电网络一样，退出子链到父链需要一个时间，这会给退出增加不便。

4.3.3 分片

传统区块链网络中的全节点会保留所有区块数据，显然因为共识的存在，这个区块链网络的处理能力会被网络中的单个全节点所约束。为了解决这个问题，可以在安全与性能之间做权衡利弊，舍弃适当的安全来提高性能。

分片 (sharding)^[7] 主要思想是把全节点分成多个组, 每个组内的节点只会保留区块链数据中自己应该保存的部分, 如需访问自己所没有的数据则作为轻节点 (light client) 向相应的组里的节点请求数据, 同时作为其他组的轻服务的服务提供者。在每个单独的分片里有自己的账户, 交易等数据, 不同分片有最低限度的通信能力。为了保持以太坊的抽象拓展性, 让主链只作为结算层工作, 分片系统部署到主链合约 VMC (validator manager contract) 上。

4.3.4 链上扩容

为了提高区块链处理能力, 在链上直接扩容也是一种显而易见的方向。虽然在链上扩容一般都会直接影响共识的内容, 但是却能直接提高区块链网络的实际处理能力。在链上扩容具体来说可以两个方向:

1. 增大单个区块的容量

每个区块都有其容纳数据的上限, 在不同的区块链中会有不同的体现, 如在比特币中体现为区块的 tx 的容量 1MB, 在以太坊中体现为区块的 gaslimit。由于对比特币来说 1MB 的区块 tx 容量是硬编码指定的, 倘若修改会导致硬分叉, 如 bitcoin cash、segwit2 等, 对以太坊而言 gaslimit 不是硬编码, 而是一个可以比例微调的动态参数, 矿工可以相互达成共识来增减 gaslimit 以适应目前的需求。

2. 减少出块的时间

区块链实现上有很多共识实现的方法, 如 pow、poa、pbft 等, 这些共识虽然原理各不相同, 但都需要协定一个出块时间, 来打包一个周期内出现的 tx。因为区块链是分布式系统, 可能会出现单点故障、传输故障甚至恶意行为, 必须能够容忍一定程度的故障且自我恢复, 所以才会有共识方法来约束出块。以 pow 为例, 若两个块出现时间接近, 则会是链短暂分叉, 如果出块的期望时间足够长, 则其中一个分叉被抛弃且很难重新达到最高的高度, 若时间太短则会导致暂时被抛弃的分叉存在追上主链高度的更大可能, 这会导致“双花”等问题。

4.3.5 共识

区块链作为分布式系统, 自然需要维持整个系统的一致性, 在对于区块链这种冗余程度极高的系统而言, 一致性便体现在大部分单个节点所达成的共识。早期区块链都使用 pow (proof of work) 工作证明共识, pow 证明具体表现为矿工节点公开一个证明, 生成该证明需要付出一定的代价而验证该证明代价极小, 只有拥有 pow 证明的块才是合法块。pow 共识机制使得区块链网络避免受到女巫攻击 (sybil attack, 如果 p2p 网络中广播数据没有一定代价, 恶意节点可能伪造身份来广播大量的无用信息来发起 dos 攻击), 但是却浪费了许多资源如电力、asic 设备等, 间接导致区块链的处理能力低下。为了减少 pow 共识带来的影响, 区块链可以使用不同的共识机制, 如 pos, pbft 等, 减少了资源浪费, 可以侧面提高区块链的实际处理能力。

5. 以太坊处理能力

5.1 以太坊处理能力的表现形式

以太坊的处理能力体现在多种形式多个层次上，除了上文所说的方案，在以太坊工作的整个流程上都存在影响其实际处理能力的要素，不可简单地把控制以太坊处理能力的缘由归结到链上扩容、链下转移这些正在探讨是否合理的改进手段。

5.2 具体影响要素

对于区块链而言，实际处理能力体现在很多方面，这里参照以太坊举例：

1. 主链上的出块速度
2. 主链上每个块的容量
3. 链上合约代码的质量
4. 执行合约代码的效率
5. 以太坊网络中数据块传输速度
6. P2P 网络发现速度
7. 链下交易量容量和比例
8. 以太坊网络中节点处理能力下限
9. 以太坊网络的共识种类
10. 以太坊网络中的无效交易和粉尘交易数量

.....

以上列出部分的可以影响以太坊处理能力的因素可以看到，以太坊的实际处理能力会在很多层次上被影响，不可只考虑链上的速度来体现以太坊的实际处理能力，从最高层的用户级合约代码，到最底层的数据传输，都会对以太坊的实际处理能力的产生促进或抑制的影响，减少 gas 消耗、提高合约代码的质量也会间接提高以太坊的处理能力。一般都会期望一个结算系统有着更强的处理能力，下面提出一些在不同层次上（evm、合约代码）进一步提高以太坊实际处理能力的设想。

6. 基于 evm 的改进

6.1 evm 原理

Evm (ethereum virtual machine) 是以太坊实现智能合约的核心部分，其参考标准文档为以 Gavin Wood 主导编写的论文“ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER”^[4]。

Evm 与普通虚拟机功能一样，只需供给其正确的字节码，虚拟机就可以按照设定好的规则执行字节码。不同的是，evm 是一个面向安全的虚拟机，这是因为字节码将会有不可预知的内容且会被以太坊网络的大部分全节点所执行，evm 在执行字节码时必须保护矿工的安全。

Evm 是一个基于栈工作的虚拟机，栈里单个元素是 32bytes 长的字节，称作 256bits 全字，绝大部分操作的字节码都是以单个栈元素为最小操作数据单元(push 可以对字节操作)。evm 在执行字节码时是隔离状态，即无法对其他网络连接，具体来说就是无法发出一个 tcp 连接请求，但是可以访问以太坊区块内容，因为区块内容也在隔离环境内^[9]。

6.2 evm 层次提高处理能力的原因

Evm 在以太坊网络中是执行合约的核心部件，同时也是除提供 pow 证明外最消耗时间和空间资源的部分，这意味着如果使用其他共识方法（如 pos），evm 的性能则会成为以太坊实际处理能力的最重要的影响因子，因此若能有效提高 evm 的性能，将会对以太坊未来的发展与运用有着极大帮助。

6.3 256bit

6.3.1 使用 256bit 字长的原因

Evm 是一个工作在 256bit 的虚拟机，这意味着 evm 在不同的数据存放区域（storage, memory, stack）都是以 256bit 存放数据，以太坊开发团体为使用 256 作为字长做出以下解释：

1. 密码学运算如 keccak, sha256, ripemd160 等在低于 256bit 字长的虚拟机中，无法简单地表示成少量的 opcode。
2. 以太坊地址为 160bit，使用 256 字长虚拟机可以直接比较地址。
3. 256bit 可以提供极大的寻址空间和更大的数值表示。

6.3.2 256bit 字长的缺点

实际上，考虑到 gas 的消耗费用，在 evm 中执行密码学运算不值得推荐，而在 64bit 字长环境中做 160bit 地址比较操作只比 256bit 字长环境下的比较操作所产生的 opcode 的长度多 1 倍，反而因为虚拟机工作在 256bit 但是物理机工作在 64bit 或 32bit 而导致在执行一些常见操作时出现很多额外机器码，也消耗了更多时间，同时 64bit 的寻址空间在可以预见的未来已经可以彻底满足寻址需求，没必要拓展到 256bit 字长来增大内存寻址空间大小。

由此可以发现，256bit 带来的好处十分有限，却降低了 evm 在物理机上执行代码的性能，如果能让 evm 工作在 64bit 下，就会让实际的机器码变少，从而提高 evm 的性能。

6.3.3 evm 的 64bit 模式

以 evm 的其中一个实现 pyethereum^[8] 为例，修改其实现使其以 64bit 字长工作。因为 python 自身的特殊性和简便性，仅仅在 evm 上修改成为 64bit 工作模式并不困难，考虑到 evm 大部分操作是在 stack 上且对修改 memory 和 storage 可能会导致共识不一致的后果，故只让 evm 的运算和 stack 工作在 64bit。

实现 64bit 步骤：

1. pyethereum/ethereum/vm.py 里保留 256bit 的截取值 TT256、TT256M1、TT255，增加 64bit 的截取值为

```
1. TT64 = 2 ** 64
2. TT64M1 = 2 ** 64 - 1
3. TT63 = 2 ** 63
```

同时把期望使用 64bit 的运算的 TT256、TT256M1、TT255 分别改为 TT64、TT64M1、TT63。

2. 修改 pyethereum/ethereum/opcodes.py 里的 push 字节码生成，把原为最高 32bytes 一次性 push 的字节码改为最高 8bytes（开区间则上限为 9）。（在兼容 256bit 的时候无需此修改）

```
1. for i in range(1, 9):
2.     opcodes[0x5f + i] = ['PUSH' + str(i), 0, 1, 3]
```

3. 修改“BYTE”字节码实现（在兼容 256bit 的时候无需此修改）

```
1. elif op == 'BYTE':
2.     s0, s1 = stk.pop(), stk.pop()
3.     if s0 >= 8:
4.         stk.append(0)
5.     else:
6.         stk.append((s1 // 256 ** (7 - s0)) % 256)
```

4. 字节码位于 0x20~0x4f 大部分为区块链自身的 256bit 数据，而 python 的 list 支持不同类型、长度的数据进栈，故这里不做 64bit 处理以保留兼容。显然这会导致这些数据不能直接做 64bit 的运算，然而 python 支持任意长度 int 型算术运算，可以在运算前进行长度判断来使用截取值，如 add 操作：

```
1. elif op == 'ADD':
2.     s0, s1 = stk.pop(), stk.pop()
3.     if s0 < TT63 and s1 < TT63:
4.         stk.append((s0 + s1) & TT64M1)
5.     else:
6.         stk.append((s0 + s1) & TT256M1)
```

5. mload、mstore、sload、sstore 这些操作会让数据在 storage、memory 和 stack 之间转移，为了能够容纳 256bit 的区块数据应该保留原有的操作，这里可以使用空置的字节码来做 64bit 的操作（需要在 opcodes.py 里添加描述），以 mload64 为例：

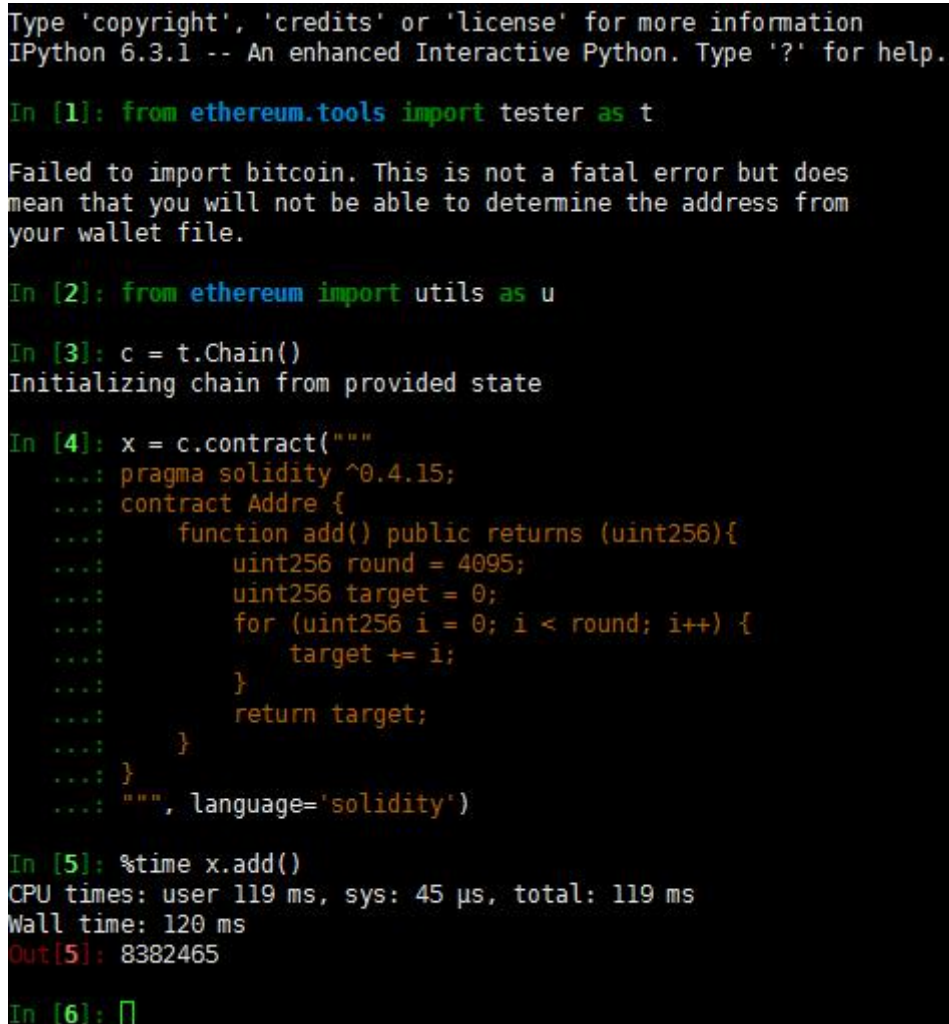

```

1. elif op == 'MLOAD64':
2.     s0 = stk.pop()
3.     if not mem_extend(mem, compustate, op, s0, 8):
4.         return vm_exception('OOG EXTENDING MEMORY')
5.     stk.append(utils.bytes_to_int(mem[s0: s0 + 8]))

```

至此 evm 的 64bit 同时兼容 256bit 的简单修改完成，能如此简单地在 evm 上做出这种修改，是因为 python 里对一个整数 int 没有字长的约束，超出单个 int 现有内存空间将会自动扩容，且所有基本数学运算支持任意字长 int 安全运算，所以只需要一个截取值即可约束为 64bit。显然这种修改依赖 python 的特性，在其他语言实现中，如 geth (golang 实现)、parity (rust 实现)^[9]，并不能如此修改，要根据其他语言的特性修改。

现在实际部署到 pyethereum 来测试该修改是否能正常工作，使用虚拟机下的 Intel (R) Xeon (R) CPU E5-2682 v4 @ 2.50GHz、1GRAM 下的 docker 打包 pyethereum 与 solc 集合镜像（因为不需要直接的网络组件，故不需要 pydevp2p 和 pyethapp），使用 tester 组件直接测试 evm。



```

Type 'copyright', 'credits' or 'license' for more information
IPython 6.3.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from ethereum.tools import tester as t

Failed to import bitcoin. This is not a fatal error but does
mean that you will not be able to determine the address from
your wallet file.

In [2]: from ethereum import utils as u

In [3]: c = t.Chain()
Initializing chain from provided state

In [4]: x = c.contract("""
...: pragma solidity ^0.4.15;
...: contract Addre {
...:     function add() public returns (uint256){
...:         uint256 round = 4095;
...:         uint256 target = 0;
...:         for (uint256 i = 0; i < round; i++) {
...:             target += i;
...:         }
...:         return target;
...:     }
...: }
...: """, language='solidity')

In [5]: %time x.add()
CPU times: user 119 ms, sys: 45 µs, total: 119 ms
Wall time: 120 ms
Out[5]: 8382465

In [6]: []

```

图5 evm 在 64bit 下执行 ADD 操作码

值得注意的是，即使以太坊标准文档^[4]明确指出 stack 是以 256bit 单元存放数据，但

是这并不是以太坊共识的强制要求，只有运算结果是确定性的，即可保证其共识。这会导致有一些以太坊实现并不遵循这个要求（如 pyethereum、remix 的 JavaScriptVM），而是利用其语言本身的特性微改实现，如 pyethereum 使用 python 实现，int 能自动增大存储空间以适应更大的数字，且实现了相关的所有算术运算，所以 python 并不遵循 stack 中 256bit 元素的限制，但这会适用于其他以太坊实现。

6.3.4 evm 在 64bit 下的问题

显然即使 evm 做出以上修改，不会改变以太坊对 256bit 的补全，如在 tx 中 input 的 256 补全、默认 int 的 256bit 等，同时 value 不能接受直接的截断，否则会出现溢出问题。虽然 64bit 的运算可以让实际机器码减少从而提高运算速度，但是却要和 256bit 同时存在，造成了实现上的隐患。

6.4 内存复用

6.4.1 evm 内存布局

Evm 作为虚拟机自然需要管理自己的内存，而 evm 中有 3 种区域可以用来存放数据：storage, memory, stack。Storage 中的数据将会保存在链上，memory 的数据是虚拟机运行时的临时数据存放区域，stack 是字节码能够直接操作的数据区域。其中 stack 最多只能放置 1024 个 slot（32bytes 为一个 slot），这限制了 stack 的容量，而 memory 则没有这个限制，允许用户有代价地临时存放大量数据。

Memory 在 evm 工作周期里存在保留用内存，0x00~0x3f 段 2 个 slot 用于密码学运算的临时空间，0x40~0x5f 段 1 个 slot 用于记录目前已分配内存，0x60~0x7f 段 1 个 slot 填充为 0 以初始化在 memory 里的动态数组内容^[10]。

6.4.2 内存回收问题

Memory 区会在每次 message call 清理，也就是说不同的 message call 之间的 memory 无法直接交互（当前 message call 结束后 memory 不会清零，导致重新需要 memory 的 message call 可能可以访问前 message call 的 memory 数据），但是在同一个 message call 中 memory 里的内存不会被回收，这意味着如果单个 message call 中出现大量 memory 使用 evm 将无法回收这些空间，这种空间的浪费行为可能导致较大的 message call 失败，从而间接降低以太坊的处理能力。

6.4.3 提供内存回收的字节码

可以考虑提供一个新的操作码（字节码）来回收已经不再需要的内存空间。以 pyethereum^[8]为例，添加操作码 MDEL，与 mload 和 mstore 一样只对全字 32bytes 进行操作，考虑到 evm 只会记录已分配的最多 slot 个数而无法提供类似 ptmalloc、tcmalloc 等高级功能（同样也是出于面向安全的考虑），故只允许回收最高位置的内存以保留原有功能的完整。

在 pyethereum/ethereum/vm.py 中小于 0x60 段 opcode 占用 0x5c 操作码实现内存回收

```

1. elif op == 'MDEL':
2.     oldsize = mem[0x40: 0x60]
3.     newsize = oldsize - 0x20
4.     if newsize <= 0x60 or oldsize != len(mem):
5.         return vm_exception('ERROR DELETING MEMORY')
6.     mem[0x40: 0x60] = newsize
7.     del mem[newsize: oldsize]

```

同时在 opcodes.py 里 opcodes 列表里添加 0x5c 操作码的描述（gas 为临时取值）

```
1. 0x5c: ['MDEL', 0, 0, 1]
```

考虑到尽量应该鼓励节省空间资源，只在执行该 opcode 时收取少量的 gas。

6.4.4 evm 内存复用的隐患

以太坊官方文档有提及虽然暂时 evm 不提供内存回收功能，但对一个完整的虚拟机而言，能够管理回收内存并复用是一个很常见的功能，以太坊官方已经意识到这个问题并可能在以后真正安全地实现这个问题。显然复用内存的确会存在安全的隐患，如清零覆盖被绕过、内存计数和管理变复杂，这些潜在的问题可能会被恶意人员利用来攻击以太坊网络。

6.5 确定性、调度表与并行

6.5.1 以太坊的确定性

以太坊作为区块链的一种实现，显然要实现其确定性（deterministic），即在链上的数据和证明都是确定的，证明出现后的任意时间和环境下都能够在单个节点上进行验证，正是因为存在这种确定性，赋予了网络中任意节点的验证权力，这个巨大的分布式账本才具备了信任和可行性。如果在区块链中证明验证的任意一个环节失去了确定性，那么该证明将无法被单个节点验证从而导致无法达成共识。

具体来说至少有以下行为会导致不确定性：

1. 对外部直接发起一个请求（如 http）。
2. 字节码的执行结果受到运行时环境直接影响。
3. 执行字节码的设计上本就不会被确定地执行（如 Prolog）。
4. tx 直接并行处理。

而在以太坊中，确定性体现在很多方面，如：

1. 字节码执行的确定性。
2. 单个区块里 tx 顺序、结果的确定性。
3. 区块顺序的确定性。
4. 证明的确定性。

6.5.2 确定性与串行执行

确定性在目前看来是以太坊或其他区块链实现的不可缺少的特性，然而确定性本身也会为区块链的实现带来缺点。

为了保证单个区块上 tx 的确定性，以太坊在处理 tx 表现为串行处理。若以太坊直接并行处理则无法确定 tx 之间的影响，也无法确定世界状态（world state）的真实结果，因为并行会出现竞争条件，无法在不同的条件下产生确定的先后顺序，这违背了区块链确定性的原则，但对于全球化的分布式系统却只能串行工作，又会大大减低该系统的处理能力。

6.5.3 调度表与并行

目前以太坊为了维护确定性，tx 的执行只能串行执行，与之相对的，eos（另一种区块链上合约平台的实现）^[1]却有着另一种处理方法。

在 eos 设计里，虽然直接并行执行 tx 不能保证确定性，但是显然单个 tx 对世界状态（world state）的影响是有限的，如果可以预先知道该 tx 会影响到了哪些地址上的状态，完全可以由出块者打包处理 tx 时生成一张调度表，表里描述了该区块哪些 tx 是毫不相关的，哪些 tx 是相互影响且在调度表里指定先后顺序，从而实现基于地址的并行（相互影响的 tx 依旧不能并行）。对于其他节点而言，该块上的 tx 验证执行可以实现有限的并行执行，因为调度表是确认的，验证时相互影响的 tx 会串行执行，无关则并行执行，所以验证该块时最后的结果也是确认的。

6.5.4 evm 的调度表

Eos 的这种处理手段以太坊也可以考虑实现，其关键在于如何在 tx 执行前低成本地得知一个 tx 会对哪些链上的世界状态产生影响。

对于外部账户的相互转账，世界状态的改变显然只出现在转账的双方地址上；对于合约账户参与的 tx 中，每一个接受 message 的账户（如被调用合约函数的合约地址）和最早发起 tx 的外部账户会出现状态的改变。

考虑一个支持账户级别并行的以太坊，在正式处理 tx 前，逐个静态分析 tx，

1. 外部账户相互转账影响的地址直接从 tx 中 from, to 段取出并记录。

2. 合约账户参与的 tx 中发起者和接受者地址先记录下来，再分析 message call 其字节码是否有 CALL, CALLCODE, DELEGATECALL (0xf1, 0xf2, 0xf4)，如果有则提取其中接受者地址，递归执行上述步骤。

至此得到张临时表，记录了每个 tx 所影响的地址，使用并查集算法合并归类得到关系表，存在关系的表由矿工节点自行指定执行顺序，而相互无关 tx 可以由 evm 并行执行。

在矿工出块时带上调度表，evm 就可以通过并行执行 tx 大大提高无关事务的速度，从而有效提高了以太坊的实际处理能力。

以下提供一个基于 pyethereum 的调度表并行方案，并假设 tx 本身附加了一个记录了需要读写数据的地址的列表或已经静态分析得出了相关联的地址列表。

```
1. # schedule.py
2.
3. import multiprocessing as mp
4. import time
5. # from ethereum.transactions import Transaction
6.
7. MAX_THREADS = 2
8.
9. class TxWrapper(object):
10.     def __init__(self, tx, relevant):
11.         """
12.         Initiate a TxWrapper.
13.         Args:
14.             tx: a Transaction instance
15.             relevant: a list of relevant address of this tx
16.         """
17.         self.tx = tx
18.         self.relevant = relevant
19.
20.     def setRelevant(self, isSpecified=True):
21.         """
22.         Findout the relevant address of this tx by static
23.         analyse if relevant addresses are not specified.
24.         Args:
25.             isSpecified: if the relevant address is specified
26.         Returns:
27.             A list of relevant address
28.         """
29.         if isSpecified:
30.             return self.relevant
31.         # TODO: static analyse for data to findout relevant adresss.
32.         return self.relevant
33.
34. class DisjointSet(object):
35.     def __init__(self, data=None, disjointSet=dict()):
36.         """
37.         Initiate a DisjointSet.
38.         Args:
39.             data: a nesting list of relevant element
40.             disjointSet: implement of disjointSet, default to dict
41.         """
42.         self.data = data
43.         self.disjointSet = disjointSet
44.
```

```
45.     def setDisjointSet(self, disjointSet):
46.         self.disjointSet = disjointSet
47.
48.     def setData(self, data):
49.         self.data = data
50.
51.     def getSet(self):
52.         return self.disjointSet
53.
54.     def generateSet(self):
55.         """
56.         Entry
57.         """
58.         self.initSet()
59.         return self.disjoint()
60.
61.     def initSet(self):
62.         for relevants in self.data:
63.             tmpRoot = relevants[-1]
64.             for i in relevants:
65.                 self.disjointSet[i] = tmpRoot
66.         return self.disjointSet
67.
68.     def disjoint(self):
69.         for relevants in self.data:
70.             for i in range(len(relevants)):
71.                 if i + 1 < len(relevants):
72.                     self.join(relevants[i], relevants[i + 1])
73.         return self.disjointSet
74.
75.     def find(self, i):
76.         root = i
77.         target = i
78.         d = self.disjointSet
79.         while d[root] != root:
80.             root = d[root]
81.         while target != root:
82.             tmpRoot = d[target]
83.             d[target] = root
84.             target = tmpRoot
85.         return root
86.
87.     def join(self, i, j):
88.         d = self.disjointSet
```

```

89.         iRoot = self.find(i)
90.         jRoot = self.find(j)
91.         if iRoot != jRoot:
92.             d[iRoot] = jRoot
93.
94.
95. class TxScheduleManage(object):
96.     def __init__(self, txlist, schedule=None):
97.         """
98.         Initiate a TxScheduleManage.
99.         Args:
100.             txlist: a list of TxWrapper
101.             schedule: a nesting list of tx schedule
102.         """
103.         self.txlist = txlist
104.         self.schedule = schedule
105.         self.maxThreads = MAX_THREADS
106.         self.createDisjointSet()
107.
108.     def createDisjointSet(self):
109.         self.disjointSetGenerator = DisjointSet()
110.         d = self.disjointSetGenerator
111.         data = list()
112.         for i in self.txlist:
113.             data.append(i.relevant)
114.         d.setData(data)
115.         d.setDisjointSet(dict())
116.         self.disjointSet = d.generateSet()
117.         return self.disjointSet
118.
119.     def createSchedule(self):
120.         if self.schedule:
121.             return self.schedule
122.         self.schedule = list()
123.         addressMapToList = dict()
124.         for t in self.txlist:
125.             tmpRoot = self.disjointSet[t.relevant[0]]
126.             if tmpRoot not in addressMapToList:
127.                 addressMapToList[tmpRoot] = [t.tx]
128.             else:
129.                 addressMapToList[tmpRoot].append(t.tx)
130.         for key, value in addressMapToList.items():
131.             self.schedule.append(value)
132.         return self.schedule

```

```
133.
134.     def getSchedulerLP(self):
135.         # TODO: RLP encode schedule
136.         pass
137.
138.     def dispatch(self):
139.         if not self.schedule:
140.             # print("need to be scheduled")
141.             pass
142.         pool = mp.Pool(self.maxThreads)
143.         pool.map(self.executeSerial, self.schedule)
144.         #pool.close()
145.         #pool.join()
146.
147.
148.     def executeSerial(self, txlist):
149.         for tx in txlist:
150.             self.execute(tx)
151.
152.     def execute(self, tx):
153.         # TODO: execute a tx in evm
154.         time.sleep(3)
155.         print(tx, " is done.")
156.
157.
158. def main():
159.     pass
160.
161. def test_disjointSet():
162.     d = DisjointSet()
163.     data = [[4,7,3],[8],[2,9],[5,4],[1,6],[10,9]]
164.     d.setData(data)
165.     d.setDisjointSet(dict())
166.     print(d.generateSet())
167.
168. def test_txDisjoint():
169.     txlist = list()
170.     txlist.append(TxWrapper(None, [4,7,3]))
171.     txlist.append(TxWrapper(None, [8]))
172.     txlist.append(TxWrapper(None, [2,9]))
173.     txlist.append(TxWrapper(None, [5,4]))
174.     txlist.append(TxWrapper(None, [1,6]))
175.     txlist.append(TxWrapper(None, [10,9]))
176.     t = TxScheduleManage(txlist)
```



```
177.     print(t.createDisjointSet())
178.
179. def test_txSchedule():
180.     txlist = list()
181.     txlist.append(TxWrapper('tx1', [4,7,3]))
182.     txlist.append(TxWrapper('tx2', [8]))
183.     txlist.append(TxWrapper('tx3', [2,9]))
184.     txlist.append(TxWrapper('tx4', [5,4]))
185.     txlist.append(TxWrapper('tx5', [1,6]))
186.     txlist.append(TxWrapper('tx6', [10,9]))
187.     t = TxScheduleManage(txlist)
188.     print(t.createDisjointSet())
189.     print(t.createSchedule())
190.
191. def test_txDispatch():
192.     txlist = list()
193.     txlist.append(TxWrapper('tx1', [4,7,3]))
194.     txlist.append(TxWrapper('tx2', [8]))
195.     txlist.append(TxWrapper('tx3', [2,9]))
196.     txlist.append(TxWrapper('tx4', [5,4]))
197.     txlist.append(TxWrapper('tx5', [1,6]))
198.     txlist.append(TxWrapper('tx6', [10,9]))
199.     t = TxScheduleManage(txlist)
200.     print(t.createDisjointSet())
201.     print(t.createSchedule())
202.     t.dispatch()
203.     print("all done!")
204.
205.
206. if __name__ == '__main__':
207.     #main()
208.     #test_disjointSet()
209.     test_txDispatch()
210.
```

以上代码附加了一些单元测试用例，可以稍作改动并使用 pytest 分离并测试，以下是 dispatch 单元测试 mock 的结果。

```
PS D:\Project\_20180419_evmparallel_py> python .\schedule.py
{1: 6, 2: 9, 3: 3, 4: 3, 5: 3, 6: 6, 7: 3, 8: 8, 9: 9, 10: 9}
[['tx2'], ['tx3', 'tx6'], ['tx1', 'tx4'], ['tx5']]
tx2 is done.
tx3 is done.
tx1 is done.
tx6 is done.
tx4 is done.
tx5 is done.
all done!
```

图 6 调度表 dispatch 单元测试结果

6.5.5 调度表的缺陷

虽然调度表能为以太坊提供有限的并行能力，也会造成其他不利影响，如：

1. 并行约束在地址级别，对于单个地址的 tx 依旧是串行执行，这意味着热门合约、大机构合作、大型冷钱包等大规模的迫切需要处理速度的商业应用并没有因此得到改进。
2. 并查集的运算与 opcode 的静态分析需要由矿工自行负责并为之付出成本，而并查集的时间复杂度和空间复杂度为 $O(N)$, N 为单个区块 tx 数量，同时 opcode 的静态分析问题规模受到 tx 数量和合约字节码长度影响。
3. Evm 并行执行时可能需要并发修改 MPT（如 state root, receipt root），考虑到调度表的影响隔离，可以简单地加一个互斥锁来保持同步。

6.6 其他

除此之外，evm 还有很多其他可能可行的性能提高方案，如选择不同的 evm 实现，操作码的值域等等，这需要以太坊社区共同探讨它们的安全、可行性等因素来研究这些改进方向是否有价值。

7. 基于合约代码的改进

7.1 智能合约原理

以太坊是一种支持运行智能合约（smart contract）的区块链合约平台，合约由去中心化的以太坊网络中的全节点共同部署与实施以保证合约的正确、安全、不可抵赖地执行。合约内容由代码指定合约的逻辑，并由 evm 执行，在执行合约时提出执行的人需要支付少量的 gas 费用以激励以太坊网络正常工作。

7.2 合约代码层次提高处理能力的原因

智能合约是以太坊上最常见也是最强大的功能，除了外部账户的单对单基本转账，其他任何转账或涉及逻辑的事务（如多重签名，延时提现，机构托管等），都会用上智能合约功

能。由此可见智能合约在以太坊上的出现频率极高，而智能合约的代码内容则直接能决定该合约的执行效率，进而影响以太坊的实际处理能力。

7.3 solidity

以太坊中智能合约由 evm 字节码组成，显然直接使用基于栈工作的字节码来编写复杂的合约并不理智，一般会使用语义清晰，逻辑分明，易学易用的高级语言来编写智能合约，再用编译器把高级语言编译成为字节码。官方推荐使用高级语言为 solidity^[10]，一种仿 js 风格的面向合约的高级语言，使用静态类型，支持继承、库调用和用户自定义结构数据等特性，可以使用其方便、安全地构建合约内容。

7.4 标准库

7.4.1 标准库的必要性

对于一个完善的高级语言，语言本身保证底层工作的稳定和安全，而一些高级的、常用的、依靠该语言实现的功能（如字符串处理，算术运算，时间处理等）则打包进库，经过严格的实践和测试后以标准接口打包为标准库，以提供给相应的开发者使用。因为标准库经过严格测试，其安全性和性能都比较高，使用标准库开发应用不仅仅能提高开发者的工作效率、降低开发成本，同时也能提高应用本身的安全性及性能。

对于 solidity 来说也一样，solidity 以其易用性作为特点，而官方却没有提供合适的标准库。此外，以太坊是一种分布式合约平台，dapp（decentralized applications）程序的编写与传统程序的编写有很大差异，如果没有标准库，开发者可能会开发出安全性差、效率低下的应用，进而降低了以太坊的处理能力。

7.4.2 实现标准库的方法

虽然官方并没有直接提供标准库，却留下了提供标准库的方法。具体来说至少有以下方法实现标准库的调用：

1. 直接在编写的合约里引用标准库的内容或 library 所有函数声明为 internal，并把标准库的部分代码编译进字节码。显然这样做会导致很多合约代码中有高度重复，合约的创建者需要支付更多的 gas，全节点也需要更大的空间来容纳这些重复的不必要的内容。

以下举例使用 library 所有函数声明为 internal：

```
1. library LibMath {
2.     function add(uint256 self, uint256 i) internal pure returns (uint256) {
3.         return self + i;
4.     }
5.     function sub(uint256 self, uint256 i) internal pure returns (uint256) {
6.         assert(self > i);
7.         return self - i;
8.     }
9. }
```

Import 该 library 不需要 link，故不会出现占位符。

2. 使用 library 关键字部署库，然后在合约中调用库功能函数，这会使合约硬编码绑定库地址（library 也是一种特殊的合约，具有地址），灵活性有限，改进升级困难。若在 solc 编译时没有指定链接的地址，则会创建一个新的库合约。

```
30405180910390f35b6000805473__browser/testme.sol:LibMath_____63771602f79091846040518363ffffffff
x0 DUF1 REVERT JUMPDEST POP PUSH2 0x87 PUSH1 0x4 DUF1 CALLDATASIZE SUB DUF2 ADD SWAP1 DUF1 DUF1 CALLD.
```

图 7 等待绑定地址而拥有占位符的字节码示例

3. 合约中 storage 区域放置一个 address 类型状态变量，用于保存库地址，使用时直接外部调用，灵活性强，但是不能直接使用上下文（如当前合约的 storage 等）。

7.4.3 标准库目前的困境

目前官方没有提供权威的标准库，但社区提供了一些测试用库，这些库的安全性和可行性并没有严格经过验证，需要进一步的观察其价值，而且库地址可能存在欺诈行为，即恶意人员提供地址前几位一样的恶意库，从而可能危害到安全意识不高的开发人员和链上合约。

7.5 assembly

7.5.1 solidity 的缺陷

Solidity 目前发布最高版本为 0.4.21，仍未属于正式发布的稳定版，许多规范标准和功能拓展尚未实现，这导致了一些在传统高级语言中很简单就可以实现的功能在 solidity 中难以实现和一些其他问题，如

1. solidity 无法直接取出 calldata，calldata 原始内容。

2. 动态长度类型 bytes 与固定长度类型 bytes32、uint256 之间不能在 solidity 中直接转换，即使 bytes 已经分配了 256bit 内存空间，solidity 中只能逐个字节填充，浪费巨量的 gas。

这些问题都能在 solidity 中使用 assembly 改进或解决，从而节省 gas，提高代码执行效率。

7.5.2 assembly 的使用

既然 solidity 作为高级语言，实际工作是由其编译后的字节码所执行的，那么和其他高级语言一样，用户有权直接在 solidity 中内嵌字节码来直接编写逻辑。

在 solidity 中，可以使用 assembly 代码块内嵌字节码，为了增加可读性，assembly 里的大部分字节码可以以函数形式编写，从而避免了直接编写字节码的困难。

对于上面提及的问题，assembly 内嵌的汇编码可以有效解决。

1. 由于可以直接使用操作码（如 calldataload，calldatacopy 等），tx 中的原始数据可以直接访问。

2. 对于已经分配好空间的动态长度数组（如 `bytes`, `string`），可以通过定位其数据段直接赋值，如

```
1. bytes32 b32 = 0x12345678;
2. bytes memory b = new bytes(32);
3. assembly {
4.     mstore(add(b, 32), b32)
5.     re := add(b, 32)
6. }
```

需要加上 32 是因为动态长度数组在 `memory` 里第一个 `slot` 是长度。

同理从访问动态数组读取其数据并存放到相应的固定长度类型或普通类型：

```
1. assembly {
2.     b32 := mload(add(b, 32))
3. }
```

此外，动态长度类型可以直接在 `solidity` 中直接互转，固定长度类型同理（长度不一样可能会截断或补零）。

通过利用 `assembly` 可以绕过 `solidity` 的一些限制，从而不需要耗费大量 `gas` 来逐字节填充目标，节省了 `gas` 的同时又能提高合约代码的效率。

7.5.3 直接使用 `assembly` 的隐患

`Solidity` 这种高级语言设计初衷不仅仅是为了方便开发者快速开发应用，与其他高级语言和虚拟机一样，安全才是最重要的考虑因素，如果开发者随意使用字节码或机器码编写逻辑，很有可能出现内存溢出、索引越界、数据混乱等后果严重且难以发现的问题，为了能在安全与性能之间做出较好的权衡，应该只在相对应的条件下使用广泛认可的 `assembly` 代码块，以防为了少许的性能而大大降低安全性。

7.6 其他

`Solidity` 依旧在早期开发阶段，还有非常多缺陷和尚未完成的功能，为了改进 `solidity` 合约代码的质量，也可以从形式验证等方面入手，从而提高以太坊的安全性及处理能力。

8. 结语

以太坊与区块链都是新生的技术与概念，其使用价值和工作机制仍需时间和社区的考验，并从中不断地自我改进。虽然上文提出了几个提高以太坊实际处理能力的设想，但是鉴于作者水平有限，这些设想的安全考虑、可行性考虑可能会有所欠缺，而随着以太坊的继续改进发展，作者本人也会持续关注并研究其改进的思路，为以太坊的未来献上自己的绵薄之力。

【参考文献】

- [1] 中国社会科学院财经战略研究院. 中国电子商务半年报(2017) [R]. 中国:中国社会科学院, 2017: 9-10.
- [2] Nakamoto S. Bitcoin: A peer-to-peer electronic cash system[J]. Consulted, 2008, 1(1):1-8.
- [3] Buterin V. A next-generation smart contract and decentralized application platform[J]. white paper, 2014, 1(1):1-22.
- [4] Wood G. Ethereum: A secure decentralised generalised transaction ledger[J]. Ethereum Project Yellow Paper, 2014, 151(1): 1-32.
- [5] Poon J, Dryja T. The bitcoin lightning network: Scalable off-chain instant payments[J]. draft version 0.5, 2016, 9(1): 1-25.
- [6] Poon J, Buterin V. Plasma: Scalable Autonomous Smart Contracts[J]. White paper, 2017, 3(1): 1-32. .
- [7] Buterin V. Sharding spec[R]. Switzerland:ethresearch, 2018:35.
- [8] Buterin V. Python core library of the Ethereum project[R]. Switzerland:Ethereum Foundation, 2014:23.
- [9] Fabian V. Ethereum Virtual Machine (EVM) Awesome List[R]. Switzerland:Ethereum Foundation, 2014:53.
- [10] Wood G. solidity document[R]. Switzerland:Ethereum Foundation, 2015:17.
- [11] Daniel L. eos white paper[R]. George:BlockOne, 2016:2

致谢

本文能顺利完成,首先最要感谢的是我的导师张胜利教授,正是以为张胜利教授的引导,我才得以开始系统地学习区块链技术及原理,参与进区块链这个新时代的进步中,这不仅仅是一次毕业设计更是一次自我提升;我也要感谢深大区块链实验室的师兄师姐们在我学习上遇到困难时给予帮助;同时我也要感谢各位同学的鼎力帮忙和家人的支持理解。正是因为有了这些支持帮助,我才能不停地提升自己、磨练自己,从而实现自己的价值。

The idea of improving the actual processing ability of the Ethereum

【Abstract】

Block chain technology provides a possibility for decentralized and tamper-resistant distributed ledger, and the Ethereum builds a decentralized and distributed contract platform to deploy and execute contracts automatically, safely and conveniently. However, as a global distributed settlement system, Ethereum processing capacity for transactions is relatively low, which is hard to meet the increasing business nowadays. This paper first introduces the basic concepts and working principles of the blockchain and the Ethereum, and then briefly describes the improvement scheme of the community to improve the capacity of the block chain, and finally puts forward some ideas for improving the actual processing capacity of the Ethereum.

【Key words】 ethereum;processing capacity;evm;solidity

指导教师：张胜利教授