计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.191200159

基于改进区块链的智能制造安全模型

王卫红　陈震宇
浙江工业大学计算机学院　杭州 310023

摘　要　针对传统区块链智能制造安全模型存在的区块构建和数据查询速度慢、插入查询操作的时间复杂度高等难题，提出了基于改进区块链的智能制造安全模型。首先为了克服传统区块链耗电量大和吞吐量低的弊端，引入新型 Merkle Patricia 树(MPT)扩展区块链结构，以提供节点状态的快速查询；然后针对 MPT 不支持并发操作和高负载状态下性能较差的问题，设计无锁并发缓存 Merkle Patricia 树，支持无锁的并发数据操作，可以提升在多核系统下的效率；最后采用具体仿真实验分析了所提模型的性能。结果表明，改进区块链的智能制造安全模型可以有效降低插入查询操作的时间复杂度，大幅提升区块构建和数据查询的速度，相较于传统模型，获得了更优的整体性能。

关键词：智能制造；工业物联网；区块链；Merkle Patricia 树；并发数据结构
中图法分类号　TP309

Intelligent Manufacturing Security Model Based on Improved Blockchain

WANG Wei-hong and CHEN Zhen-yu
College of Computer Science and Technology, Zhejiang University of Technology, Hangzhou 310023, China

Abstract　In view of the traditional block chain intelligent manufacturing security model's slow speed of block construction and data query, and high time complexity of inserting query operation, an intelligent manufacturing security model based on improved block chain is proposed. Firstly, the disadvantages of traditional block chain are solved, such as large power consumption and low throughput. A new Merkle Patricia tree (MPT) is introduced to expand the block chain structure to provide fast query of node state. Aiming at the problem that MPT does not support concurrent operation and poor performance under high load state, Merkle is designed as a lockless concurrent cache Patricia tree, which supports concurrent data operation without lock, and can improve the efficiency in multi-core system. Finally, the performance of the proposed model is analyzed by specific simulation experiments. The results show that the improved intelligent manufacturing security model of block chain can effectively reduce the time complexity of insertion query operation, greatly improve the speed of block construction and data query, and compared with the traditional model, it can get better overall performance.

Keywords　Smart manufacturing, Industrial Internet of Things, Blockchain, Merkle Patricia tree, Concurrent data structures

1　引言

工业 4.0 使集中式制造业向分布式制造业转变，其核心是将物联网(Internet of Things，IoT)的标准和技术集成到工业流程中，即工业物联网(Industrial Internet of Things，IIoT)[1-2]。随着智能工厂链接设备的数目不断增长，由于网络攻击而导致的隐私泄露会带来经济和安全问题，并给设备制造商和智能工厂带来风险，因此 IIoT 架构的安全性、隐私性与容错性亟待解决[3-4]。

通常采用区块链技术解决传统中心实体模式所带来的安全问题[4-5]。由于 IIoT 设备众多以及设备有限的计算力和功率限制，基于低性能 IIoT 设备的区块链系统面临着安全性能、吞吐量限制、交易延迟和存储资源等方面的挑战。传统区块链存在问题有：1)交易公开化，完全透明，并由于共识算法的局限性，交易延迟较大，不能满足工业互联网的即时性[6-8]；2)去中心化设计，所有节点都可随时加入或脱离区块链网络，IIoT 系统通常每秒需要管理大量交易，而由于去中心化导致的系统冗余，比特币和以太坊平均每秒只能确认 3～4 笔和数十笔交易，难以满足 IIoT 的高交易吞吐量要求。性能与可伸缩性是 IIoT 系统和区块链集成过程中面临的主要问题[9-11]。文献[12]基于区块链和智能合约技术，提出了名为 BPIIoT 的端对端 IIoT 平台，通过智能合约实现区块链上交易，利用链外网络进行存储和数据处理。分布式的区块链网络减少了存储和计算开销，降低了物联网的运营成本，但是 BPIIoT 还是

到稿日期：2019-12-26　返修日期：2020-03-24　本文已加入开放科学计划(OSID)，请扫描上方二维码获取补充信息。
基金项目：国家自然科学基金项目(61340058)；浙江省自然科学基金重点项目(LZ14F020001)
This work was supported by the National Natural Science Foundation of China(61340058) and Key Natural Science Foundation of Zhejiang pvovince, China(LZ14F020001).
通信作者：王卫红(wwh@zjut.edu.com)

基于传统公有区块链，完全分布式的开放网络不适合工业互联网的场景。文献[13]提出了一种基于私有链的高私密性的IIoT区块链安全架构，将体系分为内外两层，但私有链只适合单智能工厂环境，不适合多行业多智能工厂交互的场景。文献[14]提出了一种针对应用于IIoT系统区块链的性能优化框架，通过深度学习来动态调整生产节点、共识算法、区块大小和区块间隔，但该文只是基于配置的优化，缺乏对区块链内部算法的改进。文献[15]提出了一种基于安全多方计算协议的区块链物联网系统，服务器可以在不解密数据的情况下对加密数据进行计算，减少了资料泄露的风险。

Patricia树是一种字典树，与传统的n叉树相比，树高很低，降低了查询的时间复杂度，并且维护索引结构不需要复杂的平衡措施，适合并发实现。Merkel树是摘要算法，旨在特定的组合方法下对消息的各个部分进行迭代哈希，以确保数据各个部分的准确性。由于哈希操作的简单性，并行操作易于实现。但是关于Merkle Patricia树的并发研究较少。文献[16]提出了一种非阻塞的Patricia字典树数据结构，并提供了高效的并行插入、删除、替换的算法。文献[17]对如何优化Merkle树在并行条件下的效率进行了研究，如最优的树拓扑结构以及处理器数量的优化[18]。文献[19]把Merkle Patricia树应用于学历证书管理领域，不仅支持交易的快速查询，而且还支持账户信息的历史查询。但Merkle Patricia树插入性能有限，每次插入都要从底遍历计算哈希树根，不支持并发插入，可伸缩性不足，影响了区块构建的速度和查询速度。

针对以上问题，文中提出了基于改进区块链的智能制造安全模型，采用MPT来扩展常规区块链，通过少量的哈希计算，可以从智能制造系统中快速定位到制造设备或制造产品。该模型可用于智能制造管理或产品溯源，查询者可以通过最新的区块中的MPT获取系统的最新数据，降低了查询的时间复杂度和存储要求。同时利用基于CAS(Compare-And-Swap)原子操作的无锁的并发缓存Merkle Patricia (CMPT)树对区块链进行改进，相比传统的MPT，插入和查询的效率以及系统的整体性能都得到了提升。

2 基于区块链的智能制造安全模型

2.1 概述

目前，大部分智能制造架构还是基于传统云中心模式[20]，然而基于云中心模式的工业物联网架构存在可靠性较差的问题。由于通信延迟和云中心潜在的故障，整个系统将处于危险之中，一旦中心节点发生故障，整个系统的可用性将得不到保证。因此，本文提出了基于区块链模式的分布式弱中心化系统，能有效地避免传统云中心模式的上述问题。

如图1所示，拟议的基于区块链的智能制造安全模型包括5层。第一层是物理资源层，物理资源包括整个制造生命周期中涉及的所有制造资源；第二层是网络结构层，是总线和传感器网络的合集，可以获取各种设备的信息，管理设备情况，下达制造订单；第三层是管理节点层，以高性能服务器节点为核心，管理节点层将收集到的数据、订单信息加密打包成区块，并且通过共识算法达成结果的一致性，另外也要调度各个工厂的生产设备，分发制造订单，集成和操作智能制造设备；第四层是存储层，负责存储加密后的区块数据，在弱中心化的多中心的结构下，每个节点都存储全量数据，通过共识算法达成同步，当一个节点失效后，不影响整个系统的可用性；最后一层是应用终端层，为用户提供不同的服务。

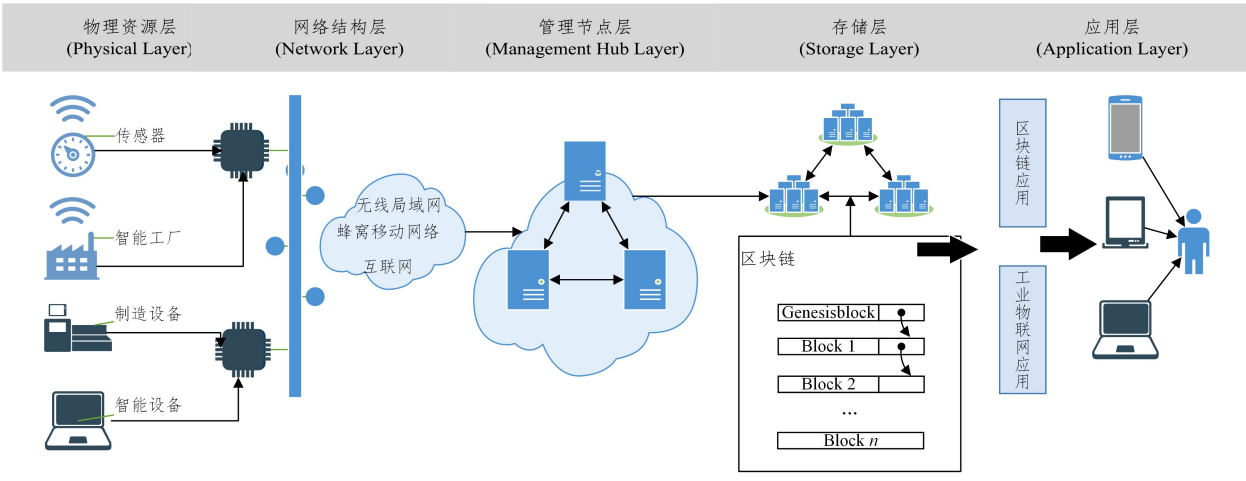

图1 基于改进区块链的智能工厂IIoT架构

Fig. 1 Improved blockchain-based IIoT architecture for a smart factory

在拟议的区块链系统中，每个用户都需要身份认证才能加入系统。用户所使用的设备被称为交易节点或者通用节点，可以在网络之中创建区块，可以用公钥加密区块信息，用私钥进行区块信息的签名，保证信息的可靠性和安全性。在区块链系统中，除了交易节点外，还有负责记录区块的节点，在拟议的区块链系统中为管理中心节点。与比特币中不受信任的系统不同，管理中心节点是由可信的计算节点构成，本文更加关注资源的利用效率，因此相比传统区块链系统采用的高复杂度的一致性算法，本文采用PBFT算法或raft算法等轻量级一致性算法，提升了整体系统的可伸缩性和实时性，减少了计算资源的浪费。结合工业环境的实际情况，在智能工厂中的每个车间或每个工厂配备一个或多个专门的数据中心和多个分布式节点可以减少单个中心节点的压力，并避免由于单个中心节点崩溃而导致的系统故障。在系统中，物理资

源节点通过链接管理中心接受下发的订单并生产制造，其可以向管理中心节点发送验证请求，从区块链中验证下发订单的正确性。

2.2 改进区块链的存储结构

数据和交易通过管理节点打包进入区块，为了适应工业互联网的应用环境与场景，本文将交易结构进行了扩展，基本结构如图 2 所示。

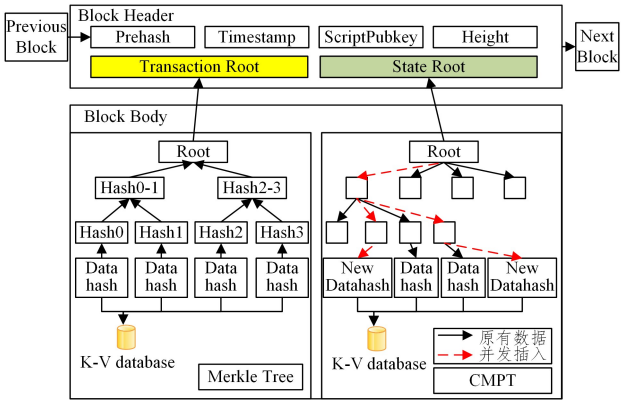

图 2 改进的区块链结构

Fig. 2 Improved blockchain structure

区块由区块头和区块体组成，区块头包括前一个区块的哈希值（Prehash）、该区块的哈希值、创建者公钥（Script-Pubkey）、区块高度（Height）、时间戳（Timestamp）、区块体的交易 Merkle 根（Transaction Root）和区块全局状态 Merkle Patricia 树的 Merkle 根（State Root）。所有交易信息存储在 *k-v* 数据库中，区块体存储交易产生的哈希摘要按时间顺序存储在 Merkle 树中，区块链中仅存储哈希摘要，因此节约了内存空间，并且可以通过卸载叶子节点来压缩区块。

交易的格式如表 1 所列，账户或设备的每次操作都会通过网络发送到管理节点中。通过管理节点对交易打包，并且更新全局状态树，可以记录每个设备的最新的运行状态。管理节点通过将打包进区块的交易进行两两哈希运算来构建 Merkle 树，并将 Merkle Root 更新至区块头，从而实现交易数据的不可篡改。

表 1 区块中的交易示例

Table 1 Transactions in the block body

From Address	To address	Message	Signature
01f1	01ffb	Message1	Sign1
01ffb	01ffc	Message2	Sign2
01f1	01ffd	Message3	Sign3

在本文提出的方案中，每个设备或账户都有独一无二的基于哈希摘要函数生成的公钥作为节点 id。管理中心节点维护以节点 id 为 key 值的 CMPT 状态，value 是节点状态信息的哈希值。当区块构建完成时，管理中心节点将验证通过的区块体中的交易并发插入 CMPT，更新整个区块链系统的最新状态，插入完成后通过并行运算得到并更新 CMPT 的根。CMPT 的更新过程如图 3 所示。当前交易树中记录了实体“01f1”“01ffb”“01ffc”“01ffd”相关的交易，在构建新区块的过程中，CMPT 对参与交易的实体的状态进行处理，并将最新的状态更新到 CMPT 中。灰色框代表打包过程中进行了修改的区块，通过只计算修改了的区块的哈希可提升区块构建的速度。从 CMPT 中读取实体节点的最新状态时，首先尝试通过 CMPT 缓存快速找到节点状态的哈希值，否则从区块头的 MerkleRoot 开始遍历，在 O(logn)时间内找到节点状态的哈希值，然后根据该哈希值到 k-v 数据库中查询对应的信息，实现了交易和节点状态的快速查询。同时，索引是经由哈希运算保存在 CMPT 中，保证了数据的不可篡改。关于 CMPT 更多的优化会在第 4 节中阐释。

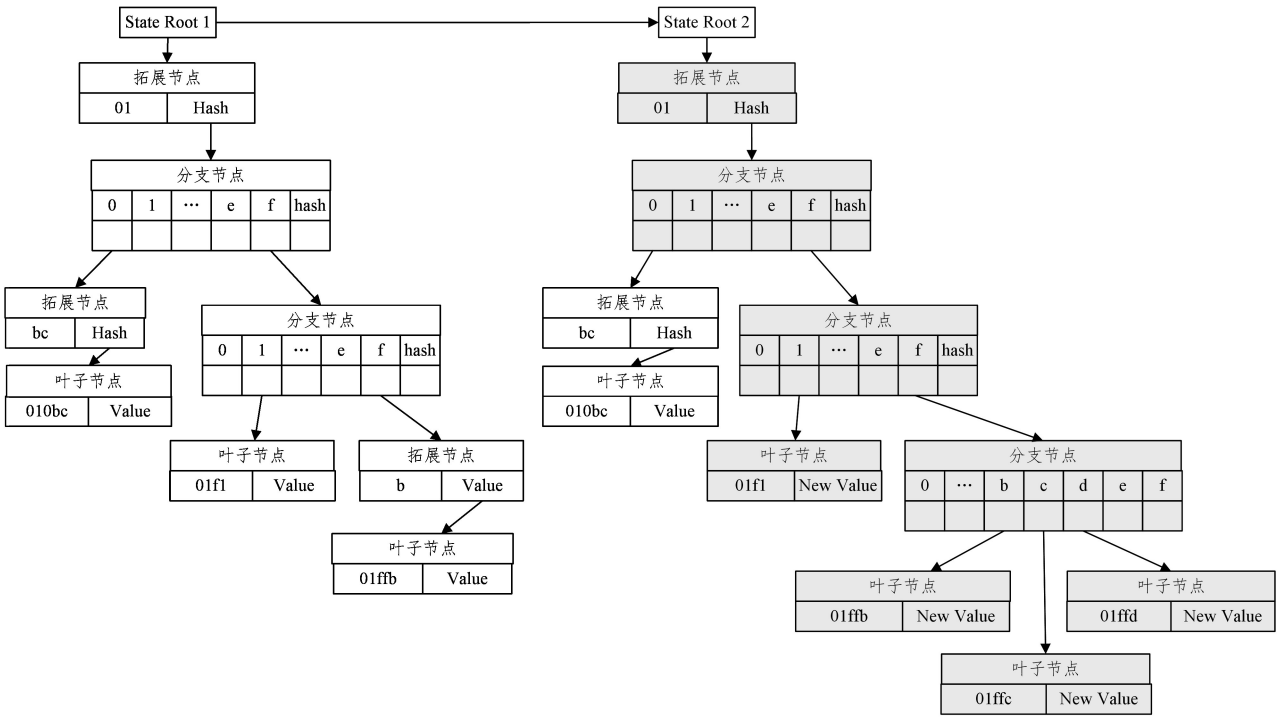

图 3 CMPT 的插入示例

Fig. 3 Insert operation example of CMPT

3 无锁的并发缓存 Merkle Patricia 树

3.1 概述

本文提出的基于区块链的智能制造安全模型不仅解决了数据安全和系统信任的问题，同时也尝试解决了传统区块链数据结构的性能瓶颈。本节提出一种称为 CMPT 的树形结构，支持无锁并发的插入查找，还介绍了如何使用 CMPT 扩展常规区块链。算法使用简单的伪代码表示。

定义的 CMPT 存储了键值对(k,v)的集合，主要有以下算法。

insert(k,v)：给定键值对(k,v)，如果 CMPT 里不包含 key 值为 k 的键值对，则添加一个键值对到树中，否则替换现有的键值对。

find(k)：给定 key 值 k，如果 CMPT 里包含 key 值为 k 的键值对，则返回相应键值对(k,v)的值(v)，否则返回空值 null。

commit$(Trie)$：给定树 *trie*，将所有节点的信息写入内存数据库，并且返回该 *trie* 计算所得的根哈希。

3.2 CMPT 的数据结构

CMPT 基本节点的数据结构如图 4 所示。CMPT 由 4 种节点结构组成：LeafNode 是叶子节点，也是 CMPT 中数据部分的节点，value 存储了序列化后的数据的哈希值，相对应的数据存储在 k-v 数据库中；HashNode 是一个简单节点，包含一个数组存储节点的 Hash 值；BranchNode 是拥有多个子节点的节点，CMPT 是 16 叉 Trie，因此 BranchNode 最多包含 16 个子节点的指针，第十七个节点存储一个 HashNode，以存储子节点计算得到的哈希值；ExtendNode 是拓展节点，通过合并只有一个子节点的 BranchNode 节点来压缩树的深度。每个节点在 CMPT 中都有 4 的倍数的缓存级别，根节点的缓存级别是 0，其子节点的缓存级别是 4。以上节点构成的 CMPT 确保了如果 CMPT 包含一个哈希值为 h 的 LeafNode，则可以通过一系列 BranchNode 和 ExtendNode 的组合从根节点开始遍历得到此节点。

$$a_1 \xrightarrow{b1[p1]|e1} a_2 \xrightarrow{b2[p2]|e2} \cdots \xrightarrow{bn[pn]|en} an \quad (1)$$

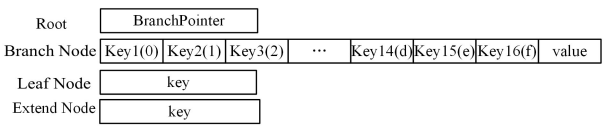

图 4 CMPT 的基本数据类型

Fig. 4 Basic data types of CMPT

3.3 数据插入算法

给定一个(k,v)键值对，从根节点开始查找，找到与给定 key 值 k 有最长公共前缀的 BranchNode，如果找到对应节点，则根据以下情况在该节点插入新值。

情况 1 如果相对应的最长公共前缀的 BranchNode 的槽位为空，则新建一个 LeafNode 或 ExtendNode 放入相应的 BranchNode，并把键值对(k,v)放入 LeafNode，或 ExtendNode 指向的 LeafNode。

情况 2 如果发生冲突，则在 ExtendNode 最长公共前缀的那一级创建一个新的 BranchNode，并把两个不同的 ExtendNode 或者 LeafNode 插入。

情况 3 在 LeafNode 节点发生冲突，如果 key 值 k 相同，则替换原有的键值对(k,v)。

算法 1 数据插入算法

```
输入:(k,v)
输出:null
1. insert(k,v)
2.    if(!insert(k,v,0,root,null))
3.       insert(k,v)
4. insert(k,v,level,cur,prev)
5.    pos=k[level/8] >>>(level%8)&15
6.    if(cur∈BranchNode)
7.       old=READ(cur.BranchKey[pos])
8.       else if(cur∈ExtendNode)
9.          old=READ(cur.Branchpointer)
10.         if(old==null)
11.      if(keylevel(key)==level))
12.         en=new LeafNode(k,v)
13.         else en=new ExtendNode(Subkey(k,level),LeafNode(k,
            v))
14.         if(CAS(cur.BranchKey[pos],old,en) ‖ CAS(cur.Branch-
            pointer,old,en)) return true
15.         else return insert(k,v,level,cur,prev)
16.      else if(old∈BranchNode)
17.         return insert(k,v,h,level+4,old,cur)
18.      else if(old∈ExtendNode)
19.         matchlen=prelen(subkey(k,level),old.hashnode.value)
20.         if(matchlen==old.hashnode.value)
21.            insert(k,v,keylevel(matchlen)+level,old,cur)
22.         else
23.            bn=extendbreanch(key,old)
24.             if(CAS(cur.BranchKey[pos],old,bn) ‖ CAS(cur.
                Branchpointer,old,bn))
25.                return insert(k,v,keylevel(matchlen)+level,bn,cur)
26.            else return insert(k,v,level,cur,prev)
27.   else
28.      en=new LeafNode(k,v)
29.         if(CAS(cur.BranchKey[pos],old,en) ‖ CAS(cur.Branch-
            pointer,old,en))
30.         return true
31.      else return insert(k,v,level,cur,prev)
32.   return false
```

算法 1 第 5－9 行以原子读的方式从相应父节点读取哈希值对应的子节点；第 10－15 行是当子节点为空，则按情况 1 在相应的子节点以 CAS 的方式插入数据；第 16－17 行和第 20－21 行代表有对应节点与当前 key 值有公共前缀，在下一层递归插入；第 23－26 行是当子节点为 ExtendNode 且当前 key 与 ExtendNode 有部分公共前缀，按情况 2 扩展节点并在节点的下一级插入；第 28－31 行是当前节点为叶子节点，按情况 3 替换子节点的(k,v)值，并检查插入是否失败，必要时从根目录重新开始插入。

3.4 数据查找算法

给定一个键值 k，数据查找算法提供了查询与 k 相关联的值 v 的查询方法，如果键值 k 不属于 CMPT 的一部分，则查找返回空值。查找算法基于式(1)，输入键值 k，从根节点开始查找，若遍历到的节点为 ExtendNode 且节点的树高为 l，则判断 ExtendNode 的哈希值数组与键值 k 从 l 位开始的子数组的最大公共前缀是否等于其自身；若遍历到的节点为 BranchNode，则将哈希的$[l,l+4)$位截出作为索引在节点数组中找到下一个节点。重复以上过程直到找到对应的 LeafNode 或者空节点。

算法 2 数据查找算法

```
输入：k
输出：v
1. find (k)
2.     find(k,0,root)
3. find(k,level,cur)
4.     pos=k[level/8] >>>(level%8)&15
5.       if(cur∈BranchNode)
6.           old=READ(cur. BranchKey[pos])
7.           else if(cur∈ExtendNode)
8.               old=READ(cur. Branchpointer)
9.       if (old==null)
10.          return null
11.      else if(old∈BranchNode)
12.          return find(k,level+4,old)
13.      else if(old∈ExtendNode)
14.          matchlen=prelen(subkey(k,level),old. hashnode. value)
15.      find(k,keylevel(matchlen)+level,old)
16.   else if(old∈LeafNode)
17.      return old. value()
18. return null
```

算法 2 第 5－9 行以原子读的方式从相应父节点读取哈希值对应的子节点；第 9－18 行是不同节点的递归查找，找到相对应的叶子节点则返回对应叶子节点的 value 值，否则返回空值。

3.5 基于缓存的数据操作优化

3.3 节和 3.4 节介绍了基本的数据查找和数据插入算法，但是这些算法的时间复杂度为 $O(\log n)$，随着数据量的增多，数据的查找和插入速度也会随之变慢。为了提升查找和插入算法的性能，用额外的缓存数据结构扩展了 CMPT，缓存数组缓存了大部分 key 所在节点的引用，能够将查找和插入操作的时间复杂度优化到 O(1)。

缓存数据结构如图 5 所示，缓存以数组形式存储当前缓存级别的索引节点的引用，数组的第一个槽位存放一个特殊的对象 CacheNode，其指向上一个级别的缓存数组。如图 5 所示，每个缓存级别都有一个缓存数组，由当前级别一直指向缓存级别为 0 的缓存数组，而每层的 misses 数组统计缓存未命中的情况。改进的查找算法首先调用 *fastfind* 方法从当前缓存数组中读取公共前缀对应位置的数据，若没有构建缓存数组，则调用普通 *find* 方法，否则从最高级别的缓存数组尝试读取对应公共前缀位置的值，然后根据读取的结果调用普通 *find* 方法进行查找。成功的快速查找不会更新缓存。为了维护缓存数组，每次普通查找都会有两个操作。首先，如果查找到的节点的缓存级别等于当前缓存数组的缓存级别，则将该节点加入缓存数组；如果当前节点的缓存级别大于当前缓存数组的缓存级别，则在当前缓存数组的 CacheNode 中调用 *cachemiss* 方法，记录一次缓存失效，如果当前缓存级别的缓存数组的失效次数达到一定阈值，则会尝试调整并提升缓存级别。

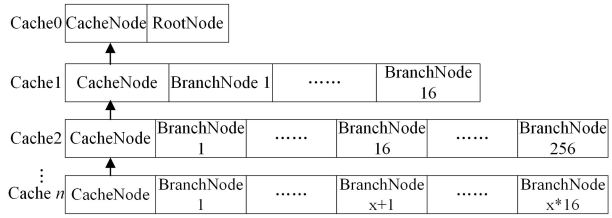

图 5 缓存的数据类型

Fig. 5 Cache data types

算法 3 基于缓存的改进查找算法

```
输入：k
输出：v
1. find(k,level,cur,cache,cachelevel)
2.     if(level==cacheLevel)
3.         recordcache (cache,cur,level)
4.         …
5.     else if (old∈ExtendNode)
6.         if(level<cachelevel||level>cachelevel+4)
7.             cachemiss()
8.         if(level+4==cachelevel)
9.             recordcache(cache,old,level+4)
10  …
11.            return find(k,level+4,old,cache)
12. fastfind (k)
13.    cacheold=READ(cache)
14.        if(cacheold=null)
15.            return find(k,0,root,null,-1)
16.        toplevel=countbits(cache. length-1)
17.        while(cache !=null)
18.            pos=1+k&(cache. length-2)
19.            cachenode=READ(cache[pos])
20.            level=countbits (cache. length-1)
21.            if(cachenode∈ExtendNode)
22.                return find(k,keylevel(cachenode. value)+level,old)
23.            else if (cachenode∈BranchNode)
24.                return find(k,level,cachenode)
25.            cache=cache[0]. parent
26.            return find(k,0,root,null,toplevel)
```

算法 4 缓存维护算法

```
输入：k,node,cachelevel
输出：null
1. Recordcache(cache:Array[],node:any,cachelevel:int)
2.    if (cache==null)
3.        if( cachelevel >=12)
4.            cache=createcache(8,null)
5.            CAS(cacheHead,null,cache)
6.            Recordcache(cache,node,cachelevel)
```

```
7.      else
8.          length=cache.length
9.          cachelevel=countbits( length-1 )
10.         if(cachelevel==cachelevel)
11.             val pos=1+k& (cache.length-2)
12.             WRITE(cache[pos],node)
```

为了实现缓存的维护，对 3.3 节和 3.4 节的数据操作算法进行了改进，算法 3 第 5－9 行展示了当改进的普通查找方法遇到和当前缓存的缓存等级相同的节点时，会调用 *Recordcache* 方法更新缓存；而当遇到偏离当前缓存级别的节点时，则调用 *cachemiss* 记录一次缓存失效。算法 4 展示了缓存数组的更新，CMPT 在低树高时不需要缓存数组，当读取到的当前节点的缓存级别达到 12 时，则会从缓存级别 8 开始创建缓存数组；而如果当前节点和缓存数组的级别相同，那么算法会将该节点写入缓存数组以维护缓存。

3.6 Merkle 根的并行哈希计算

传统的 Merkle 树是平衡二叉树，因此允许在树的同一级别上并行处理数据的多个子树，可以通过把树快速拆分成相同大小的数个部分，然后通过多核 CPU 并行处理来完成整个 Merkle 根的哈希运算过程。但是 CMPT 是前缀树，并非平衡树，在不知道整体结构和数据量的情况下无法把 CMPT 均匀拆分成数个子树，为此本文改进了并发哈希操作。如果在低数据量的情况下，CMPT 并没有构建缓存数组，且并行计算对运行效率的改善有限，则 Merkle 根的哈希运算采用串行方式。如图 6 所示，若缓存存在，哈希操作分为两个阶段，首先在当前缓存级别的缓存数组上执行细粒度的均匀分配，在当前缓存级别的树高均匀地分配节点并行计算，当整个缓存数组计算完毕后再将剩余的满 n 叉树分成 n 个子树并交给 n 个处理器并行计算，最终再汇总计算得到 CMPT 的哈希根。

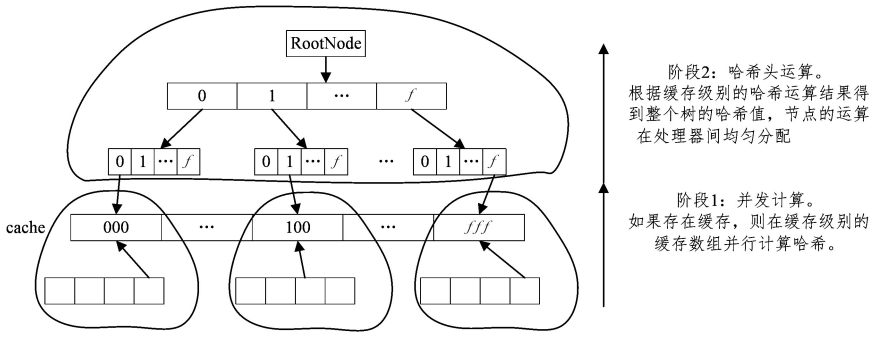

图 6 Merkle 根计算示例

Fig. 6 Example of Merkle root computation

3.7 无锁算法的正确性分析

由多个线程共享的 MPT，若没有并发控制，当两个线程同时操作相同的节点时，则会导致多线程冲突，从而造成数据丢失或脏读问题。本文将从安全性和活性方面分析并发操作的正确性。

3.7.1 安全性分析

安全性意味着对并发数据结构的操作不会出错。为了证明算法的正确性，确定了线性化点，如算法 1 第 14，24，19 行的 3 个 CAS 指令是线性化点，除去这 3 个指令，算法的其他部分不会对这个树进行修改。而 CAS 指令是原子操作，多个线程产生竞争修改时，CAS 能够确保只有其中一个线程操作成功，不会产生修改丢失等问题，因此 CMPT 的操作算法是安全的。

3.7.2 活性分析

CMPT 的操作算法是无锁的，且能够证明对于树任何的状态更改，都会在有限步骤内完成。以数据插入算法为例，首先，递归调用永远不会降低缓存级别，且只有 CAS 操作失败才会在当前缓存级别重新调用插入操作。由式(1)可知，每个叶子节点可通过根节点经过有限个节点的路径访问得到，因此两个 CAS 操作之间只存在有限个步骤。而由于 CAS 操作的特性，如果一个 CAS 操作 C0 失败，则在 C0 读取预期值的时间 $t0$ 到 CAS 操作失败的时间 $t1$ 之间，存在同一位置执行成功的 CAS 操作 C1。由上可得，树的两个状态之间的步骤是有限的。因此 CMPT 的操作算法是无锁的。

4 仿真实验分析

4.1 实验环境

实验在一台操作系统为 Ubuntu16.04 的电脑上进行，主要硬件配置如下：CPU 为 Intel(R)_Core(TM)_i7-4790K，4.0GHz×4core×2，RAM 为 32GB；并安装了 Oracle JDK 8 的环境，堆内存为 6G。在以太坊的区块链实现中，需要通过将交易进行打包以生成 MerkleRoot，并且通过更新 MPT 的 MerkleRoot 来实现最新状态的更新。在本文的模型中，工厂管理中心节点通过收集数据并构建 Merkle 根从而实现数据的区块链化，并且通过快速更新 CMPT 的节点值和 Merkle 根获得当前区块链各个工业设备的最新状态的索引，然后将最新状态的哈希摘要打包进入区块链。

本文的研究重点是区块链中数据与节点状态的索引方法，不涉及网络及分布式共识，因此以单机进行测试，构建代码实现了区块链的打包过程，并设计实验以测试索引的性能。

4.2 评价指标

该实验的比较指标如下。

(1)CMPT 的基准测试包括插入和查找操作，在传统场景下与其他传统并发数据结构的性能相比，操作所用的时间越短，代表其性能越好。

(2)在区块链系统中性能的主要衡量标准是每秒的交易数(Transaction Per Second)。TPS 是区块链的每秒区块的增长数乘以每个区块的大小，再除以交易的平均大小，TPS 越

高，每秒处理的交易就越多，整体系统的性能也越好。

(3)伴随着数据量的上升，固定时间内查询成功的次数。数目越高，索引的性能越好，设备与管理中心节点通信的效率也就越高。

本文用 java 语言在 JDK8 下实现了 CMPT。实现分为两部分，首先是与 JDK8 中的 ConcurrentHashMap 和 ConcurrentSkipListMap 两种数据结构实现在传统场景下进行基准测试，设计实验通过比较这两种高效的并发数据结构的性能来测试 CMPT 在多线程环境下的性能。本组实验的比较对象为：1)Merkle Patricia Trie；2) ConcurrentHashMap；3)ConcurrentSkipListMap；4) Elixir 以太坊标准库中的 MPT 实现[21]。设计多组实验来测试 CMPT 的性能，每组测试记录 5 次，并将平均值定义为算法的运行时间。

4.3 实验结果与分析

4.3.1 CMPT 的基准测试

本文在单线程和多线程环境下将 CMPT 的插入和查找性能与 JDK 中 ConcurrentHashMap 和 ConcurrentSkipListMap 的插入和查找性能进行对比，结果如图 7 所示。可以看出，CHM 的性能优于 CMPT，CSkipList 的性能最差，因为大规模数据下会有大量的缓存未命中，CMPT 劣于 CHM 的主要原因是读取缓存数组导致的额外指针访问。在多线程环境下，实验结果如图 8 所示。在低数据量下，CMPT 与 CHM 的性能相近；而在高数据量情况下，CMPT 的性能略逊于 CHM，这是因为实验过程中 CMPT 产生了多次慢速查询和缓存升级。

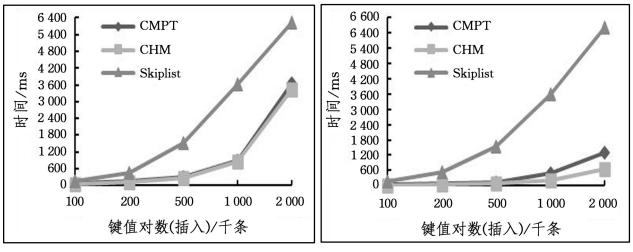

图 7 CMPT 单线程查询与插入性能对比

Fig. 7 Perfomance comparison of CMPT single-threaded lookup and insert

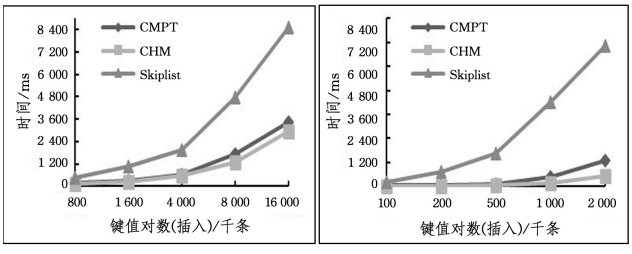

图 8 CMPT 多线程查询与插入性能对比

Fig. 8 Perfomance comparison of CMPT muti-threaded lookup and insert

4.3.2 CMPT 的区块构造性能测试

本文将 4 线程环境下的 CMPT，单线程下的 CMPT 和 MPT 进行对比，分别设置每个区块中交易数量为 64，128，256，512，1 024，2 048，4 096，8 192，构建 10 个区块得到平均执行时间和 TPS，实验结果如图 9 所示。从图 9 可以看出，在低数据量下，CMPT 的性能与 MPT 相近，因为低树高下没有构建缓存，插入和哈希运算都是慢速操作；但达到高数据量时，CMPT 和多线程下的 CMPT 的性能较原版 MPT 有较大提升，尤其是 MPT 在高数据量下每次操作都要从叶子节点遍历求哈希树根，每次插入操作都要做 $\log(n)$ 次哈希运算，算法开销较大。

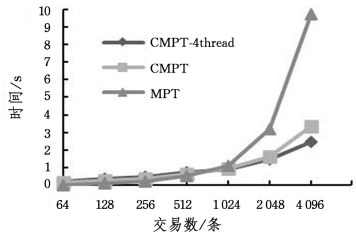

图 9 CMPT 和 MPT 的构建性能对比

Fig. 9 Construction performance comparison of CMPT and MPT

4.3.3 CMPT 的查询性能测试

在本文模型中，系统维护了 CMPT，以各个设备的哈希值为键值进行查询，以获取各个设备对应的最新状态，并设计了实验测试在不同的数据量构成的树下每 10 000 次查询所需的时间，结果如图 10 所示。由图 10 可知，由于缓存数据结构的辅助，在不同数据量下，CMPT 每 10 000 次查询消耗的时间和 MPT 接近，伴随着树高的上升略有增加，而且多线程并不影响查找操作的效率；而伴随着数据量的上升，标准 MPT 的查询时间会不断延长，在大规模数据量的场景下，CMPT 提供了优异的查询性能。

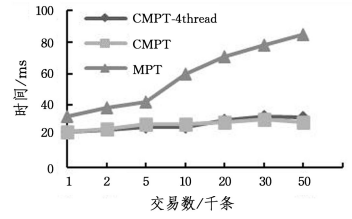

图 10 CMPT 和 MPT 的查询性能对比

Fig. 10 Query performance comparison of CMPT and MPT

4.4 实验分析总结

综合上述实验，本文实现的 CMPT 在低数据量时和以太坊的标准 MPT 性能相仿，但是在智能制造的工业场景下，大规模、大量数据的情况下，CMPT 保持了可伸缩性和高性能，支持线程安全的并行数据操作，加快了区块验证和构建的速度，提升了交易吞吐量。此外，在使用基于账户的模型时，交易需要依赖历史记录，CMPT 提供了历史记录的高效查询，能够有效提升整个区块链系统在大规模数据和应用节点下的可用性。

结束语 区块链具有安全性、私密性、不可篡改性和分布式部署的特性。本文基于区块链为智能工厂提出了一种分布式的智能制造安全模型，介绍并分析了模型的层次结构，解决了传统 IIoT 系统的单点故障问题和伸缩性问题，提出了基于 MPT 的改进区块链结构，优化了区块链的存储模式，加快了数据的查询速度。针对大数据量下 MPT 的性能瓶颈，提出了无锁并发缓存 Merkle Patricia 树(CMPT)，提供了无锁并

发的数据操作。实验证明,本文提出的 CMPT 有效提升了智能制造工业高数据量场景下的区块链系统的性能。

参 考 文 献

[1] LIAO Y, LOURES E D F R, DESCHAMPS F. Industrial Internet of Things: A Systematic Literature Review and Insights[J]. IEEE Internet of Things Journal, 2018, 5: 6-16.

[2] DRATH R, HORCH A. Industrie 4.0: Hit or Hype? [J]. IEEE Industrial Electronics Magazine, 2014, 8(2): 56-58.

[3] SADEGHI A R, WACHSAMNN C, WAIDNER M. Security and privacy challenges in industrial Internet of Things[C]// 52nd ACM/EDAC/IEEE Design Automation Conference. ACM/EDAC/IEEE, 2015: 1-6.

[4] FERNÁNDEZ-CARAMÉS T M, FRAGA-LAMAS P. A Review on the Application of Blockchain to the Next Generation of Cybersecure Industry 4.0 Smart Factories[J]. IEEE Access, 2019, 7: 45201-45218.

[5] SAMANIEGO M, DETERS R. Internet of Smart Things-IoST: Using Blockchain and CLIPS to Make Things Autonomous [C]// 2017 IEEE International Conference on Cognitive Computing (ICCC). IEEE, 2017.

[6] NAKAMOTO S. Bitcoin: a peer-to-peer electronic cash system [EB/OL]. https://bitcoin.org/bitcoin.

[7] Bitcion charts and graphs—blockchain[EB/OL]. https://www.blockchain.com/zh-cn/charts.

[8] Ethereum project[EB/OL]. https://www.ethereum.org/.

[9] LO S K, LIU Y, CHIA S Y, et al. Analysis of Blockchain Solutions for IoT: A Systematic Literature Review[J]. IEEE Access, 2019, 7: 58822-58835.

[10] BONNEAU J. EthIKS: Using Ethereum to audit a CONIKS key transparency log[C]// International Conference on Financial Cryptography and Data Security. Berlin, Heidelberg: Springer, 2016: 95-105.

[11] ZHANG R, XUE R, LIU L. Security and Privacy on Blockchain [J]. ACM Computing Surveys(CSUR), 2019, 52(3): 52-86.

[12] BAI L, HU M, LIU M, et al. BPIIoT: A Light-Weighted Blockchain-Based Platform for Industrial IoT[J]. IEEE Access, 2019, 7: 58381-58393.

[13] WAN J, LI J, IMRAN M, et al. A blockchain-based solution for enhancing security and privacy in smart factory[J]. IEEE Transactions on Industrial Informatics, 2019, 6: 3652-3660.

[14] LIU M, YU R, TENG Y, et al. Performance optimization for blockchain-enabled industrial internet of things (iiot) systems: A deep reinforcement learning approach[J]. IEEE Transactions on Industrial Informatics, 2019, 6: 3559-3570.

[15] ZHOU L, WANG L, SUN Y, et al. Beekeeper: A blockchain-based iot system with secure storage and homomorphic computation[J]. IEEE Access, 2018, 6: 43472-43488.

[16] SHAFIEI N. Non-blocking Patricia tries with replace operations [J]. Distributed Computing, 2019, 32(5): 423-442.

[17] ATIGHEHCHI K, ROLLAND R. Optimization of tree modes for parallel hash functions: A case study[J]. IEEE Transactions on Computers, 2017, 66(9): 1585-1598.

[18] YUE D, LI R, ZHANG Y, et al. Blockchain Based Data Integrity Verification in P2P Cloud Storage[C]// 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS). IEEE, 2018: 561-568.

[19] XU Y, ZHAO S, KONG L, et al. ECBC: A high performance educational certificate blockchain with efficient query[C]// International Colloquium on Theoretical Aspects of Computing. Cham: Springer, 2017: 288-304.

[20] HEGAZY T, HEFEEDA M. Industrial automation as a cloud service[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 26(10): 2750-2763.

[21] Elixir implementation of modified Merkle Patricia tree [EB/OL]. https://github.com/exthereum/merkle_patricia_tree.

[22] ZHU Q, LOKE S W, TRUJILLO-RASUA R, et al. Applications of Distributed Ledger Technologies to the Internet of Things: A Survey[J]. ACM Computing Surveys (CSUR), 2019, 52(6): 120.

WANG Wei-hong, born in 1969, postgraduate, professor, master's supervisor. His main research interests include cloud computing, big data analytics, and information security.