國立臺灣大學電機資訊學院資訊工程學系
碩士論文
Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University
Master Thesis

Structural Assessment on Stateless Blockchain Improvements with LRU-oriented Cache Replacement Policy

無狀態區塊鏈及運用 LRU 快取策略的評估

蔡存哲
TSUN-CHE TSAI

指導教授：廖世偉博士
Advisor: Shih-Wei Liao, Ph.D.

中華民國 109 年 6 月
June, 2020

國立臺灣大學碩士學位論文
口試委員會審定書

Structural Assessment on Stateless Blockchain Improvements with LRU-oriented Cache Replacement Policy
無狀態區塊鏈及運用 LRU 快取策略的評估

本論文係蔡存哲君 (R05922086) 在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 109 年 6 月 11 日承下列考試委員審查通過及口試及格，特此證明

口試委員：

_____

_____ _____

_____ _____

_____ _____

_____ _____

所　長：_____

誌謝

感謝指導教授廖世偉教授提出最小堆積的想法，也感謝教授仔細閱讀論文並修正諸多錯誤。

感謝陪伴我的家人、朋友。

感謝在 514 和我一起討論論文的建中、贊鈞，區塊鏈組的德倚、彥智、鼎元、伯駒，以及其他實驗室的同學子賢、爾晨、宗興。

感謝跟我討論複合名詞文法問題的朱瑜同學。

感謝實驗室助理 Kelly 在庶務上的幫忙。

摘要

基於區塊鏈的加密貨幣網路中，每個節點都維護一個完整的賬本，經年累月，賬本的儲存容量越來越高，維護一個節點的成本也會越來越高。無狀態區塊鏈是一種針對以上問題的改進方案，它的每一筆交易中附上了合法性證明，使得節點不需要進入資料儲存裝置查詢過往賬本資訊，只要驗證交易內附的證明即可，從而使節點所需儲存的資訊量大幅減少，然而，要付出額外的網路流量與驗證時間。

在本論文中，我們將在基於賬戶的區塊鏈上討論以下兩點：第一，一筆交易的合法性很可能由最近出現過的其他交易的合法性推知，只要快取賬戶的訊息，就可以省略部分交易的證明，以此來減少網路流量與驗證時間。第二，快取替換有多種策略，針對 LRU 策略，我們設計了基於不可變紅黑樹、不可變值無關順序樹、不可變堆積的三種資料結構，透過共享資料，它們的時空間複雜度都控制在合理的範圍。我們也進行實驗比較了快取命中與快取失效的代價，模擬現今以太坊的區塊性質，當快取數十個區塊大小的賬戶資訊量時，快取命中所需的查詢時間仍數倍低於梅克爾 Patricia 證明驗證的時間。

關鍵字：區塊鏈、無狀態區塊鏈、快取、資料結構

Abstract

Cryptocurrency based on blockchain is a peer to peer trading system. In this decentralized system, each node maintains a complete ledger which records the entire transaction history of all accounts. Years after years, the requirement to a single node will get higher and higher. Dealing with the issue, stateless blockchain is a promising approach. To be more specific, in this system, each transaction is attached with proof of validity. With this approach, there is no need for a node to access the data storage device to search past information of the ledger. Instead, a node only verifies the proof attached to each transaction. Therefore, it will significantly reduce the space of data storage for each node. But it will cause additional network traffic and verification time.

In this paper, we will discuss the following two points on account-based blockchain. First, it is highly possible to infer the validity of a transaction from another valid transaction appearing recently. If we cache account information, we can eliminate some proof in transactions and then reduce network traffic and verification time. Second, there are so many cache replacement policies. For LRU policies, we designed three composite data structures based on red-black tree, value-independent order tree, and heap. By sharing data, these data structures have feasible space complexity and time complexity. In our experiments, we compared the cost of a cache hit with the penalty of a cache miss. Simulating Ethereum's block property nowadays, if cache size is able to contain tens of blocks' information, the cache searching time

will be significantly lower than the time of verifying Merkle Patricia proofs.

Keywords: blockchain, stateless blockchain, cache, data structure

Contents

口試委員會審定書 **iii**

誌謝 **v**

摘要 **vii**

Abstract **ix**

1 緒論 **1**

2 背景與相關研究 **3**

2.1 區塊鏈 3

2.1.1 區塊驗證 4

2.1.2 狀態儲存的問題 4

2.2 無狀態區塊鏈 5

2.2.1 vector commitment 6

2.3 快取證明 6

2.3.1 持久化資料結構 7

3 設計 **11**

3.1 淺狀態區塊鏈 11

3.2 快取設計 12

3.2.1 分叉處理 12

3.2.2 持久化快取 13

3.3 快取策略 14

3.3.1 最近 k 塊 14
3.3.2 LRU 15
3.4 持久化 LRU 演算法 16
3.4.1 雜湊 + 紅黑樹 17
3.4.2 雜湊 + 值無關順序樹（value-independent order tree） 19
3.4.3 雜湊 + 最小堆積 24

4 實驗設計與結果 **31**
4.1 實驗環境 31
4.2 資料結構 31
4.3 速度 32
4.3.1 工作量 32
4.3.2 調整快取大小 33
4.3.3 調整快取命中率 34
4.3.4 調整 put 比例 34
4.4 失效代價（miss penalty） 35
4.5 總結 36

5 未來工作 **37**
5.1 快取拆分 37

6 總結 **39**

參考文獻 **41**

List of Figures

2.1 一般區塊鏈、無狀態區塊鏈交易比較 6
2.2 路徑複製 7

3.1 不一致的快取 12
3.2 區塊鏈分叉 13
3.3 分叉頻繁切換 13
3.4 區塊 7a 為粉紅色區塊中的所有交易資訊 14
3.5 LRU 資料結構 16
3.6 雜湊 + 紅黑樹 18
3.7 不可變紅黑樹 19
3.8 以網連接節點 20
3.9 滿二元樹連接節點 20
3.10 雜湊 + 值無關順序樹 21
3.11 對值無關順序樹進行一連串操作 22
3.12 持久化堆積 shiftdown 28

4.1 調整快取大小 33
4.2 調整命中率 34
4.3 調整 put 比例 35

List of Tables

Chapter 1

緒論

2008 年，中本聰發佈了比特幣的白皮書 [14]，這是一個以區塊鏈為根本的新型加密貨幣系統，這個系統中的每個節點都會保留一個完整的賬本，如此確實使得該系統的去中心化程度達到相當高的程度，但不可磨滅的賬本資訊也導致每個節點所需要的儲存空間逐年增加。

在十多年後的現在，人們需要上百 GB 的資料儲存空間，才能夠運行最為知名的加密貨幣系統比特幣、以太坊 [18] 的一個節點，這使得部分由私人維護的節點逐漸不堪負荷。若退而求其次，僅儲存驗證新區塊所需的資訊，亦即比特幣中的 UTXO 、以太坊中的世界狀態，也需要數 GB 到數十 GB 的儲存空間，不止是在空間上對資料儲存裝置有所要求，速度上也是，傳統硬碟的隨機存取太慢，驗證以太坊區塊的速度已經跟不上共識生成區塊的速度，如此迫使以太坊節點的維護者必須購買較為昂貴的固態硬碟，進一步提高了節點的運行成本。

為此，加密貨幣的社群開始嘗試提出一些解決方案，無狀態區塊鏈（stateless blockchain）就是其一。無狀態區塊鏈僅要求每個節點儲存區塊標頭，但每一筆交易必須附帶合法性證明，如此大幅降低了空間需求，並且無需在資料儲存裝置中查詢過去狀態就能夠驗證交易合法性。

本研究嘗試進一步優化無狀態區塊鏈，我們讓無狀態區塊鏈的節點有共識的維護快取，若一筆交易的付款人曾在最近的其他交易中出現過，就有機會從快取中推知此交易仍然是合法的，此時便可以在交易中省略合法性證明，進而降低了交易的大小，也減少廣播時所需要的網路流量。

但是，如何高效的維護快取成了新的問題。每一筆區塊接上區塊鏈時，都會有

一個對應的快取，而當快取相對於一個區塊足夠大時，臨近區塊上的快取資訊會有很高的重疊，若選用一種能夠共享臨近區塊資料的資料結構，就能大幅降低所需儲存空間。該資料結構的設計與快取的策略有緊密的關係，我們分別為簡單的最近 k 塊策略、較複雜的 LRU 兩種快取策略，討論了能夠減少儲存空間負擔的資料結構。對於 LRU 策略，在討論基於持久化紅黑樹以及基於持久化堆積的資料結構之外，我們還設計了值無關順序樹這一種資料結構，並且進行實驗得知，使用這持久化資料結構快取數十個區塊大小的賬戶資訊量時，快取命中所需的查詢時間仍數倍低於快取代價（梅克爾 Patricia 證明驗證的時間）。

論文的其餘章節依序如下：第二章說明背景以及相關工作，第三章開始說明淺狀態區塊鏈（shallow-state blockchain）[1]的設計，包含快取的策略以及快取使用的資料結構，第四章為實驗的設計與結果，第五章為未來工作，第六章為總結。

[1]一般的區塊鏈節點儲存所有歷史狀態；無狀態區塊鏈僅儲存區塊標頭；本論文所設計的淺狀態區塊鏈則選擇性地儲存最近存取過的賬戶狀態，若說一般區塊鏈儲存的是整個大海，那淺狀態區塊鏈儲存的就是淺水區的資訊，並且隨潮汐不斷更迭。英文也就順勢翻譯成 shallow-state blockchain 。

Chapter 2

背景與相關研究

2.1 區塊鏈

在區塊鏈網路中的每一個全節點 (full node) 都會維護一份完整的賬本，賬本記錄了過往所有區塊中的交易，由此我們能夠知曉交易發生的時間點、交易的付款人收款人、交易的金額，付款人得以向收款人證明自己確實已經付款。

區塊鏈網路中，賬本以區塊為單位更新，當一個礦工節點蒐集交易並挖掘出一個區塊後，會將它廣播到網路上，收到區塊的節點必須驗證後決定是否接收，若接收，賬本的內容就得到更新。

根據收付款的方式，區塊鏈貨幣可分為類似銀行賬戶式的賬戶系統，以及類似零錢式的 UTXO (unspent transaction output) 系統。

賬戶系統中，每個用戶擁有一個地址，地址有它相對應的私鑰，經由私鑰簽章，用戶就能將他持有的資產轉移到其他地址。

UTXO 系統中存在著大量的 UTXO ，每個 UTXO 上有一個腳本，當一個使用者能夠提供一個輸入（通常是使用他的私鑰簽名）使得腳本執行結果為真，他就得以花費這筆 UTXO ，花費 UTXO 後會產生其他的 UTXO ，只要新生成的 UTXO 只有收款人才能花費，資產的轉移就完成了。在 UTXO 系統中，使用者擁有的是一組它能夠花費的 UTXO ，而非一個賬戶上的一個代表存款的數字。

在本論文中，只關注基於賬戶的區塊鏈系統，忽略 UTXO 的區塊鏈系統。

2.1.1 區塊驗證

在支援智慧合約的賬戶系統中，對於區塊中的一般交易，節點必須知道付款人是否有足夠的餘額；若涉及智慧合約，節點需要獲取合約程式碼以及合約當前的狀態，才能夠執行合約。

我們可以用一個鍵值對來表示賬戶系統在驗證交易時所需要儲存的資訊：

$$state = f : address \rightarrow account$$

給一個地址，我們能得到對應賬戶的資訊，可能是餘額，也可能是智慧合約的狀態。

若要儘速完成這些工作，節點必須快速地檢索交易，因此，一個一個區塊地尋找、計算出賬戶資訊是不切實際的，通常節點會內置一個資料庫，並利用資料庫的索引來高速完成查詢工作。

2.1.2 狀態儲存的問題

區塊鏈的狀態儲存方式造成了三個衝擊：

1. 佔用儲存空間大
2. 驗證交易時需要資料儲存裝置的隨機存取
3. 節點初始同步時間太久

佔用儲存空間大

永遠增長的區塊已經對節點維護者造成負擔，比特幣與以太坊佔用的空間都已經超過 200 GB ，如果在 AWS 上租用機器來運行以太坊節點，每個月需花費 50 - 70 美金，這個成本如果降不下來，慈善節點的數量勢必會不斷下滑。根據歷史資料，2018 年 2 月時尚有逾兩萬個節點，到 2020 年 5 月節點數量已經下降到七千。[1]

另外一個問題是，以太坊中存在許多智慧合約已經不再被使用，例如 EOS 的首次貨幣發行 (ICO) ，在發行結束後，該合約已無作用，但所有的全節點卻仍得

[1] 資料來自 Wayback Machine 上 https://ethernodes.org/ 的快照記錄

繼續儲存它的歷史數據。這類僅有短暫效力的合約這無疑造成了大量的空間浪費。

需要資料儲存裝置隨機存取

交易中的賬戶地址沒有規則，因此當節點驗證交易，去獲取賬戶狀態時，必須進行隨機存取，一個區塊中有幾筆交易，就必須進行幾次隨機存取。傳統硬碟的存取依賴讀寫頭移動跟碟片旋轉，隨機操作的效率低下，以致於使用傳統硬碟的節點甚至無法跟上以太坊網路 15 TPS (transaction per second) 的交易速度。不得不使用造價較為昂貴的固態硬碟又進一步加劇了節點維護者的經濟負擔。

節點初始同步時間太久

當以太坊節點初始化時，必須從網路上下載過去的區塊，並且計算出當前的世界狀態才能開始驗證、接收新區塊，即使不在節點中執行交易以計算世界狀態，而是從網路上下載最新的世界狀態，都需要超過一週的時間才能同步到最新的區塊。

不同以太坊實作不斷提出新的同步模式，試圖加速同步速度，[16] 的論文中也提出了 Turbo Sync 、Checkpoint Sync 等等新的同步模式，試圖透過避免展開狀態樹（state trie）來加速同步。

2.2 無狀態區塊鏈

為了解決上述問題，區塊鏈社群提出了狀態租賃（state rent）、狀態修剪（state pruning）、無狀態區塊鏈（stateless blockchain）等等改進方案。

以下介紹無狀態區塊鏈的概念。

無狀態區塊鏈的核心想法在於，節點能否在只儲存區塊標頭的情況下，仍然能夠驗證交易的正確性？若能僅儲存區塊標頭，也就代表節點初始化同步時僅需下載區塊標頭，在 2018 年時，耗時不到一小時就能讓節點同步完畢 [16]。

可以的，如果我們在交易中附上一些額外資訊，並且以區塊標頭來驗證這些額外資訊無誤，就能夠利用這些額外資訊來驗證交易的正確性了。

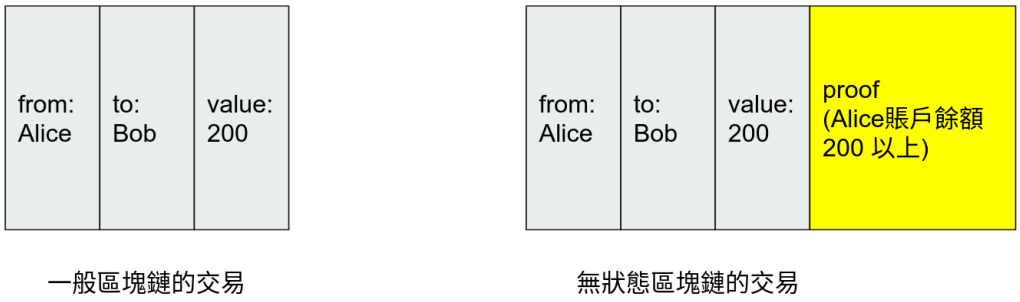

圖 2.1: 一般區塊鏈、無狀態區塊鏈交易比較

在賬戶系統中，則可以讓所有賬戶狀態構成一顆梅克爾 Patricia 樹（Patricia Merkle tree）或是稀疏梅克爾樹（sparse Merkle tree）[7]，並且在區塊標頭裡放入根（也就是整個狀態的摘要），在交易中附上付款人的餘額，以及付款人狀態的梅克爾路徑（Merkle path），如此，就可以確認付款人的餘額，再去看它是否足夠支應本次交易。

圖 2.1 是一般區塊鏈交易與無狀態區塊鏈交易的對比，同樣是一筆 Bob 給 Alice 200 元的交易，在無狀態區塊鏈上，必須多附上一筆證明（黃色部分），以供無狀態節點驗證交易的合法性。

2.2.1 vector commitment

不僅僅上述的梅克爾樹變形能夠生成、更新摘要和證明，可以做到這些功能的結構被稱為 vector commitment[4]。

近年來，vector commitment 領域也持續出現不同針對無狀態區塊鏈的代數構造，它們相較於梅克爾樹，時空間複雜度更小，或是擁有其他優點：有的能夠將多個證明打包以降低空間 [3]，有的則有同態加法的特性，使得製作交易證明時，付款人不用知道收款人的當前餘額 [5]。

2.3 快取證明

Utreexo [9] 應用梅克爾樹構成的森林設計了一種新的 accumulator [2]，基於此種 accumulator ，Utreexo 節點不需要真正的儲存比特幣所有的 UTXO ，就能夠驗證區塊。

該篇論文也提及到，分析比特幣的歷史記錄，約有 40% 的 UTXO 會在 20 個區塊內被消耗，將近 80% 的 UTXO 會在 1000 個區塊內被消耗，因此使用少量的記憶體空間來快取最近出現的 UTXO ，就能夠省略傳輸許多證明。

然而，Utreexo 僅僅討論了在初始同步節點資料時的情形，並未考慮到同步到最長鏈之後可能發生的分叉問題。此外，由於 UTXO 的產生與消耗都各只有一次，使得具有統計性質的快取策略難以應用，而在賬戶系統中，賬戶可以多次出現，因此可以應用的快取策略更廣泛。

本論文將會討論在賬戶系統中該如何應對分叉問題，以及 LRU 快取策略在賬戶系統中的使用。

2.3.1 持久化資料結構

為了在分叉情況下高效存取快取，我們將會使用持久化資料結構，故先行在此介紹基本概念與相關技巧。

如果一種資料結構在修改之後，能夠保留之前的版本，就可以被稱為持久化資料結構 [8]（Persistent Data Structure），與之相對的是暫時性資料結構（Ephemeral Data Structure）。

路徑複製

對於樹這種資料結構，可以利用路徑複製的技巧來將它變得持久化，當樹中的一個節點被修改，就連帶修改指向它的父親節點，如此一直遞迴往上修改到根節點，就可以得到一棵新版本的樹。

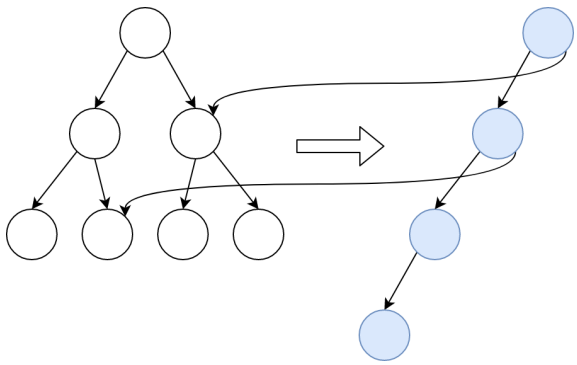

圖 2.2: 路徑複製

圖 2.2 是一個路徑複製的例子，可以從圖中看到，只有被插入葉子的節點所在的分支被複製了一趟（已被塗為淺藍色）。

在樹上使用路徑複製這種技巧時，資料結構也是不可變（immutable）的，亦即舊的樹依然是完整的、沒有受到任何改動，新樹只是用指針指向了舊樹的部分資料而已。

在這種情況下，若要刪除舊樹，僅需從舊樹的根開始向下遞迴，遇到引用計數[2] 大於 1 的節點，將其引用計數減 1 後則停止遞迴，遇到引用計數等於 1 的節點則刪除之並繼續向下遞迴。

用虛擬碼描述這個過程：

假設一個節點的結構是

Struct *Node* **contains**

> Node *left;
>
> Node *right;
>
> int reference_count;

end

刪除的虛擬碼如下

```
void remove(Node *node) {
  if (node == null) { return; }
  if (node->reference_count == 1) {
    remove(node->left);
    remove(node->right);
    delete(node);
  } else {
    node->reference_count--;
  }
}
```

用高階程式語言撰寫刪除舊樹的算法實際上是很容易的，C++ 支援引用計數指標（shared_ptr），只要釋放舊樹的根，解構子就會自動遞迴完成整棵舊樹的刪除；在支援垃圾回收（Garbage Collection）的語言如 Java ，只要丟棄舊樹的根，運行

[2]一個節點被多少個指標指到，該節點的引用計數就是多少，節點剛創造時引用計數為 1 ，只有有指標指向該節點，該節點的引用計數加 1，若指向該節點的指標被刪除或改指它處，引用計數減 1 。

時（runtime）的垃圾回收器就會清除整棵舊樹。

Chapter 3

設計

無狀態區塊鏈為了縮減資料儲存空間，必須在交易附上證明，導致所需網路流量增大；為了省去資料儲存裝置隨機存取所耗用的時間，必須花費 CPU 計算能力來驗證證明。

無狀態區塊鏈相較於一般區塊鏈做出了一些取捨 (trade-off)，而淺狀態區塊鏈的目的是讓這種取捨不再是全有或全無（全部都附上證明或全部不附上證明），使得狀態儲存的程度變得可調節。

3.1 淺狀態區塊鏈

無狀態區塊鏈中的一個區塊，如果多份交易的付款人、收款人都相同，交易的證明也會是完全相同的，這樣重複的資訊顯然可以省略。

擴展這個想法，如果我們快取最近出現過的交易中的賬戶資訊，則下一次收到同樣賬戶的交易時，也不需要去驗證證明。

再更進一步，讓整個網路上的節點都遵循同一套規則來記錄快取，使得所有節點在區塊廣播的時刻，對於區塊中的哪個交易附證明、哪個交易不用附證明有共識，那節點就能夠剝離掉不必要的證明，進而省下網路流量。

淺狀態相對於無狀態，犧牲了一些記憶體空間，但是只要存取快取的速度快過驗證證明的速度，淺狀態區塊鏈就有望在效能上勝過無狀態區塊鏈。

如果節點在驗證同一個區塊時，使用的快取不一致，將會導致某些節點承認該區塊，某些節點不承認，從而導致分叉。

譬如，如果節點快取住它高度最高的 k 個區塊中的交易資訊，當網路延遲，不同節點中的鏈分叉情形不同時，快取就會不一致，見下圖：

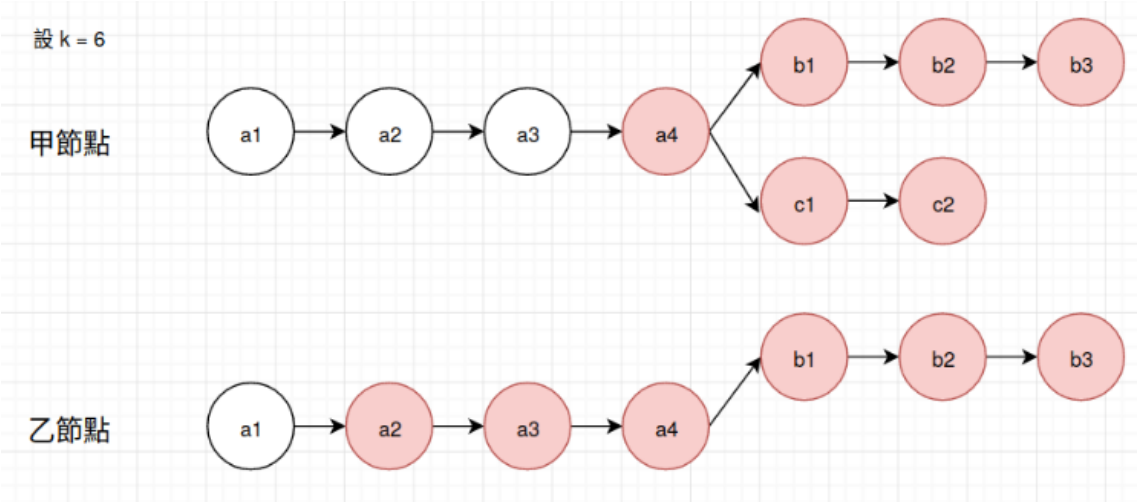

圖 3.1: 不一致的快取

粉紅色表示在快取，白色區塊表示不在快取。此時若有一個不附證明的交易，付款方在 a2 區塊出現過，則乙節點會接收交易，甲節點則不會接收。

3.2 快取設計

一個簡單的設計準則可以避免前述的錯誤：每一個區塊都有自己的快取，快取的內容僅僅由該區塊所在的鏈的資料所決定。如此，我們把樹狀結構縮減為一個串列（list），而不同節點上同個區塊所在的串列必定是相同的，只要每個節點都用同樣的確定性演算法（deterministic algorithm）從這個串列計算出快取，就能夠保證每個節點驗證同一個區塊時的快取一致。

3.2.1 分叉處理

觀察圖 3.2 中的鏈：

此時，區塊 4b 嘗試接上區塊 3 ，因此它必須基於區塊 3 的快取來進行驗證。也就是說，如果我們在接上區塊 4a 時，將區塊 3 的快取直接修改而稱為區塊 4a 的快取，那當我們要街區塊 4b 時，就無從知悉區塊 3 的快取了，使用某些快取策略時，我們可以透過回退 (roll back) 來取回區塊 3 的快取，但當使用某些快取策略時，遺失的快取無法輕易找回。

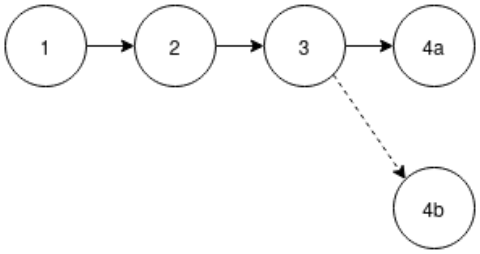

圖 3.2: 區塊鏈分叉

即使使用可以回退的快取策略，當分支切換越頻繁，回退的次數也會越頻繁，回退的效能就可能就會成為瓶頸。

例如在圖 3.3 中，如果同時維持 a, b 兩條分支，則每次接收到非當前分叉的區塊時，都要進行回退，並且隨著分叉的差異越大，回退的長度也越長，若從 7a 要走到 7b ，就必須先退回 3 ，再走回 7b ，若之前曾經從 6a 走到過 6b ，則過程中的大部分運算都是相同的，這顯然是一種浪費。

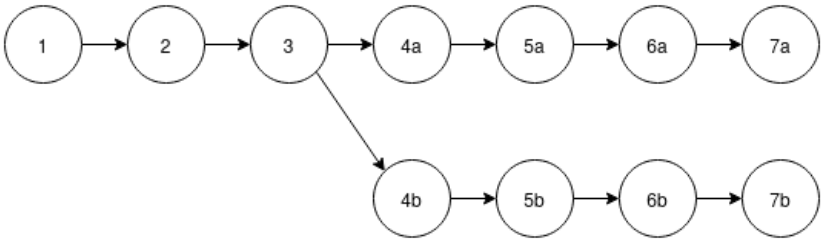

圖 3.3: 分叉頻繁切換

3.2.2 持久化快取

我們採用另外一種思路：在每個區塊上都保留它自己的快取，當新區塊要接上鏈的時候，就可以直接取用它前一個區塊的快取，無需重新計算。換句話說，我們採用全持久化資料結構 (fully persistent data structure) [8] 來儲存快取，每接受一個區塊，就生成一個快取的版本。

在這個方案中，我們必須定時刪除太舊 (例如，距離最長鏈超過 20 個區塊) 區塊的快取，以將整條鏈的快取大小限制在一定範圍，否則任由快取無限增長，將導致記憶體用罄，以致於必須存取資料儲存裝置，快取就變得沒有意義了。

這個方案帶來了一個立即問題，如果我們每次都複製前一個區塊的快取，那快取的所佔用的空間將會正比於未刪除的快取的數量。然而，相鄰區塊中的快取有很高的相似性，若能選用適當的資料結構來讓相鄰區塊共享快取，將能夠有效提高空間使用率。

3.3 快取策略

不同的快取策略在應對不同工作量 (workload) 時的命中率 (hit rate) 各不相同，以下討論實作簡單的「最近 k 塊」策略，以及實作較為複雜，但經驗上命中率較高的 LRU 策略。

3.3.1 最近 k 塊

在「最近 k 塊」策略中，一個區塊的快取即為由該區塊開始，由高往低取 k 個區塊，這 k 個區塊中出現過的交易中的資訊。

以下為 k = 6 的示意圖

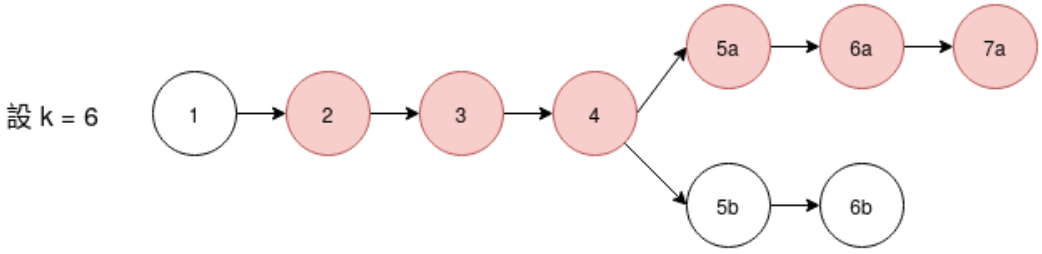

圖 3.4: 區塊 7a 為粉紅色區塊中的所有交易資訊

可以用一個鍵值對 (key value pairs)（其底層可為雜湊表 (hash table)、平衡搜尋樹 (balenced search tree)、跳躍鏈接串列 [13](skiplist)..... 等等）來表示快取，若考慮每次重算的情境，例如在圖 3.4 中，要在 7a 區塊後接上一個 8a 區塊，則我們加入 8a 區塊中的交易資訊，並丟掉只在 6a 中出現但沒有在其餘區塊中出現的交易資訊。

「最近 k 塊」的快取可以回退，跟前進時的演算法一樣，只是換了個方向。

當考慮全持久化時，我們可以選用可持久化的鍵值對資料結構，例如雜湊[1][15][12]、搜尋樹都有相對應高效成熟的持久化實作，若一個快取有 n 個鍵，則持久化雜湊／搜尋樹插入／刪除一筆鍵值時，時間、空間複雜度皆為 $O(log(n))$。

3.3.2 LRU

LRU 是 least recently used 的縮寫，這種策略中，快取大小是固定的，若快取已滿，在插入新資料之前必須先丟棄一筆資料，LRU 會去挑選快取中所有資料中最久沒被用到的那一筆來丟棄。

LRU 之所以比 FIFO 的命中率更高，是因為以太坊的歷史數據顯示，(1) 一個賬戶花錢之後，很可能又接著花錢。(2) 一個賬戶收到錢之後，很可能會馬上將錢花掉。LRU 由於會更新資料的使用時間，得以一直快取住頻繁出現的資料，FIFO 則只看資料進入快取的時間，只要快取失效會發生，快取需要被抽換，那即使某些資料頻繁出現，遲早還是會被丟掉。

下表是 FIFO 跟 LRU 策略在不同快取大小時，快取以太坊前一百萬個區塊中出現的賬戶所得到的命中率，由此可以看出 LRU 在各種情況下都優於 FIFO 。因此我們優先研究 LRU 策略。

快取賬戶數量	10	100	1000	10000
FIFO 命中率	0.436	0.648	0.803	0.975
LRU 命中率	0.476	0.667	0.815	0.980

抽象來看，LRU 是一種支援兩個介面的資料結構，

- get(key)
- put(key, value)

get(key) 時，若 LRU 存在該鍵，則返回對應值，並且將 key 的使用時間調整到最新。

put(key, value) 時，若 LRU 空間未滿，直接插入一筆鍵值對，這筆新鍵值對的的使用時間為最新；若 LRU 空間已滿，就要找出當前快取中使用時間最舊的丟掉，再插入新鍵值對，此新鍵值對的使用時間亦為最新。

對應到淺狀態區塊鏈的情境中，每當一個區塊要接上，我們要計算新區塊快取時，會把一系列賬戶資訊的讀取跟修改操作轉變成 get 跟 put ，鍵是賬戶地址，值是賬戶狀態，然後在 LRU 底層的資料結構上進行相應操作。

當在淺狀態區塊鏈中使用 LRU 策略時，是無法高效回退的。當插入一筆鍵值對時，要丟棄的資料可能在好幾個區塊之外，然而被修改的 LRU 無從得知這筆資料要去哪個區塊尋回。

3.4 持久化 LRU 演算法

在討論持久化 LRU 演算法之前，我們先觀察如何在軟體上高效實作 LRU 快取，調查 github 上多個高使用量的 LRU 函式庫，內部資料結構都是雜湊表與雙向鏈表 (doubly linked list) 的組合：

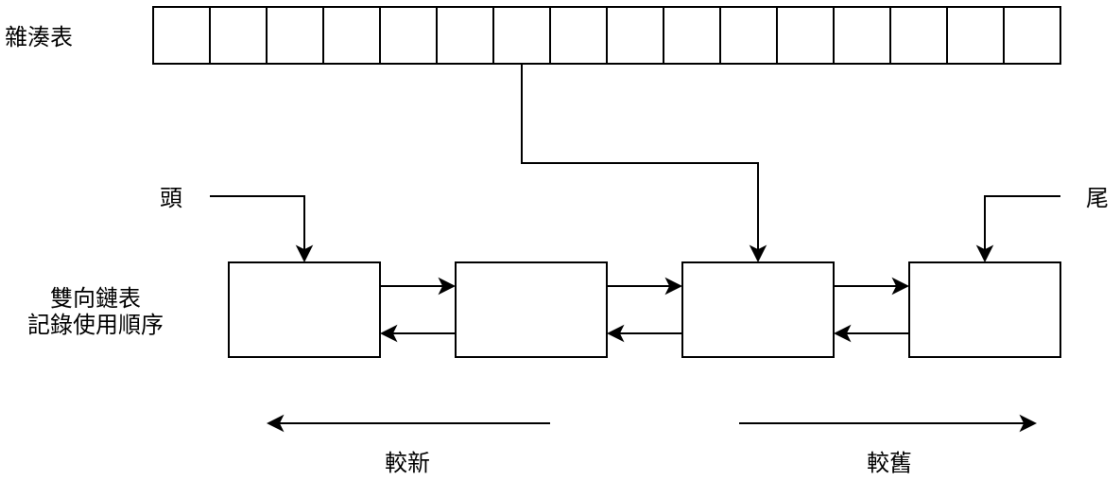

圖 3.5: LRU 資料結構

執行 get 時，透過雜湊得到指向節點的指標，獲取資料，並且將指標指向的鏈表節點移動至鏈表頭部（最左側）。執行 put 時，若快取命中，更新節點的值，並將節點移動至鏈表頭部，若快取未滿，從頭部加入快取值，並將其指標放入雜湊表，若快取已滿，拔出雙向鏈表的尾部（最右側）節點，並且在雜湊表中移除該舊鍵，然後在頭部加入快取值，放指標到雜湊表。

觀察到 LRU 需要記錄的資訊有二：

1. 由鍵找到值（鍵值對）

2. 各個鏈的順序資訊

在前述的雜湊表 + 雙向鏈表的實現方案中，雜湊表負責 (1)，雙向鏈表則負責 (2)，注意到，雙向鏈表極為自然的記錄了順序關係，它甚至不需要記錄確切的使用時間。

一個簡單的想法是，我們直接把雜湊跟雙向鏈表的持久化替代品組合起來，就得到了一個持久化 LRU ，然而，雖然持久化雜湊有很成熟的替代品，持久化雙向鏈表卻沒有。

[8] 提出了多種演算法能夠使得任何基於節點的資料結構半／全持久化（partial/fully persistent），根據該論文的 node-splitting 演算法，甚至能夠在 $O(1)$ 時間複雜度內完成雙向鏈表的插入、修改。

然而，該論文並沒有提出刪除持久化資料結構的老舊版本的方法，這使得這些演算法難以應用到淺狀態區塊鏈的快取上，因為無法刪除過去版本將導致快取所需的空間始終無法釋放，最終耗盡記憶體容量。

持久化雜湊可以由使用路徑複製的不可變樹來實現 [1]，根據 2.3.1 節的討論，它可以輕易地刪除老舊版本，並沒有上述的問題，我們現在需要的是其他可高效持久化、又可刪除老舊版本的資料結構來取代雙向鏈表。

進一步抽象雙向鏈表做的事情有：

- 更新一個節點的使用時間到最新
- 刪除使用時間最舊的節點
- 插入新節點，新節點的使用時間為最新

以下，先討論了如何用紅黑樹 [11]（平衡搜尋樹）來完成上述任務，再介紹我們設計的值無關順序樹（value-independent order tree）資料結構，最後，討論了基於堆積 [6]（heap）的設計。這三者都以使用路徑複製的不可變樹實作，因此可以很容易地刪除老舊版本。

3.4.1 雜湊 + 紅黑樹

我們嘗試使用紅黑樹來記錄順序資訊。首先，為每一筆賬戶資訊設置一個獨一無二的時間序，這個時間序可以很容易得到，例如說設置成 $block\_height *$

$max\_tx\_in\_one\_block + tx\_number$。

然後，以時間序做為鍵，賬戶資訊為值，建造一棵紅黑樹。雜湊表則用賬戶地址為鍵，對應的時間序為值。

get 時，先在雜湊表中由賬戶地址得到時間序，再到紅黑樹中由時間序得到賬戶資訊。例如，在圖 3.6 中，我們會先從雜湊表得到賬戶的時間序為 243 ，再到紅黑樹中查詢 243 。

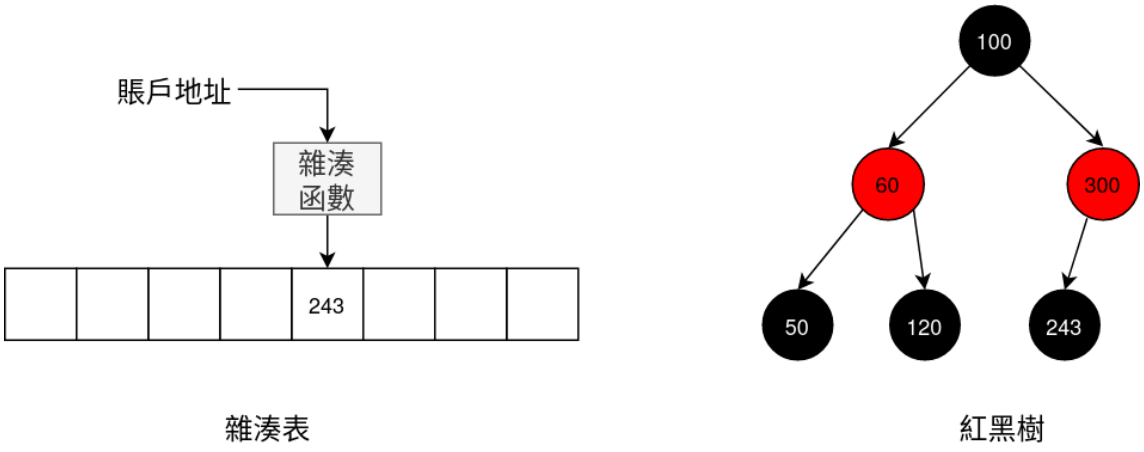

圖 3.6: 雜湊 + 紅黑樹

查詢完成後，需更新使用時間。具體操作為修改雜湊表中地址對應的時間序，並移除紅黑樹中的原節點，加入新時間序做為鍵。

put 類似於 get ，但需要修改賬戶資訊。

圖 3.7 表示不可變紅黑樹的共享結構。該圖中，快取的大小設為 8 。狀態 1 時，只有 7 筆資料，狀態 2 對狀態 1 插入 14 ，資料變成 8 筆，狀態 3 再對狀態 2 插入 15 ，由於快取已滿，必須先刪除時間序最小的資料，也就是最左下角的紅色 7 。

時空間複雜度分析

由上述分析可知，get 跟 put 會執行常數次的雜湊查詢、修改，以及常數次的紅黑樹刪除跟插入，而不可變樹實作的持久化雜湊跟持久化紅黑樹的這些操作都只耗用 $O(\log n)$ 的時空間複雜度，因此 LRU get, put 的時空間複雜度也是 $O(\log n)$ 。

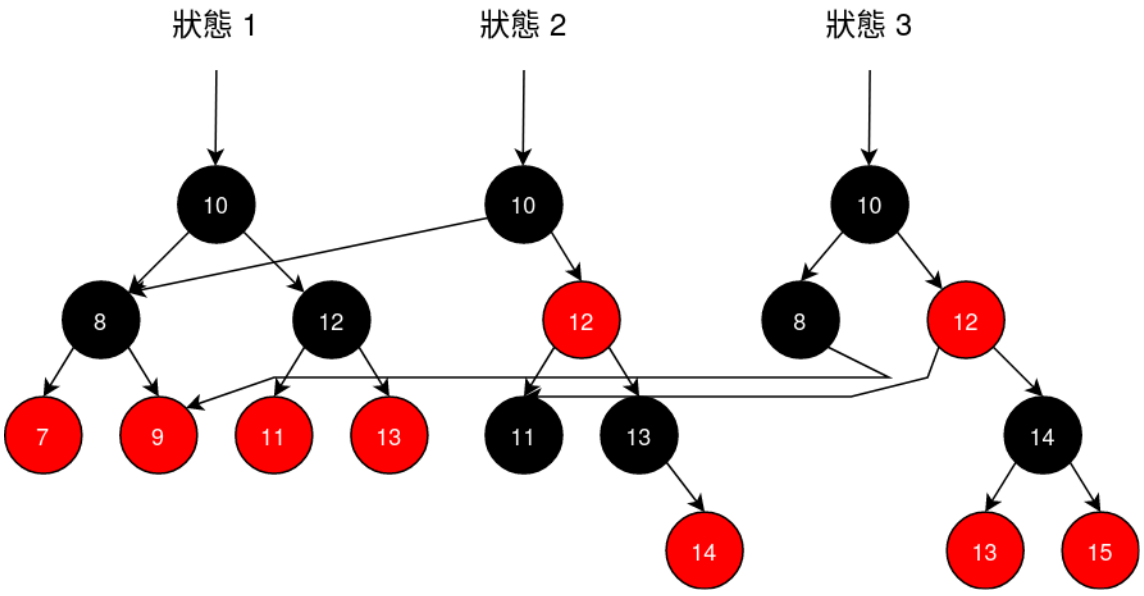

圖 3.7: 不可變紅黑樹

為什麼雜湊表不存紅黑樹節點地址

當不可變紅黑樹插入／刪除時，會導致 $O(\log n)$ 個節點發生路徑複製，如果雜湊表中是儲存紅黑樹的指標，在雜湊表上指向原 $O(\log n)$ 個節點的指標也必須跟著更新，一個持久化雜湊的更新操作耗時 $O(\log n)$ ，相乘後時間複雜度提高到 $O((\log n)^2)$，比前述使用序號的做法來得差，因此不考慮。

3.4.2 雜湊 + 值無關順序樹（value-independent order tree）

雙向鏈接串列以節點之間的指向關係記錄順序關係，紅黑樹卻必須額外記錄時間序，此外，原本透過雜湊就能一次查詢到賬戶資訊，紅黑樹方案卻得地址 -> 時間序 -> 賬戶資訊兩段式地查詢。

雙向鏈表無法以路徑複製來變換為不可變資料結構的原因在於，兩個相鄰節點總是互指，一旦以路徑複製的方式修改節點，就得複製整個鏈表。於是我們思考，不要用互指的方式來連接節點，就可以順利路徑複製。

想像在這些節點的背後編織一張網，然後將它們粘在一起（圖 3.8）。

這個網狀結構最簡單形式就是一棵滿二元樹 (full binary tree)（圖 3.9）。

我們將利用滿二元樹來儲存順序資訊的資料結構稱為值無關順序樹，以下開始一一介紹值無關順序樹的各種操作。

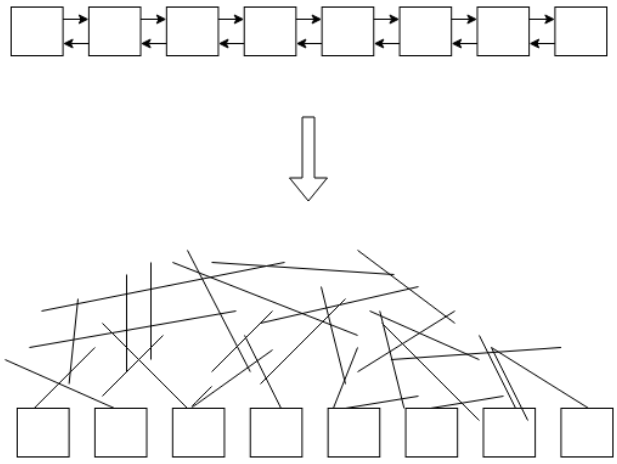

圖 3.8: 以網連接節點

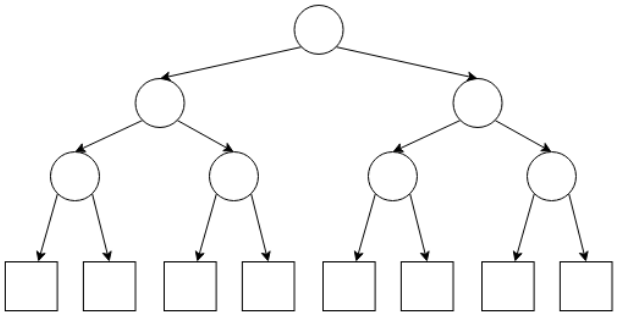

圖 3.9: 滿二元樹連接節點

若快取的容量為 n ，值無關順序樹的高度將會設置為 $1 + \lceil \log_2 n \rceil$，也就是說，葉子的數量至少是快取容量的兩倍。圖 3.10 就是一棵容量為 4 的值無關順序樹，它有 8 個葉子節點。

應用於 LRU 快取時，雜湊表所儲存的值會是一個指向值無關順序樹葉子節點的指標，只要一次雜湊表查詢就能取得資料。（見圖 3.10）

值無關順序樹將所有資料都存放在葉子節點，每份資料按照使用時間的新舊來排列，本文往後都按照資料越新，葉子的位置越右側的慣例來解釋跟畫圖。

值無關順序樹的葉子有些有存放資料，有些沒有，我們將有存放資料的葉子稱為「有用葉子（used leaf）」，沒有存放資料的葉子稱為「未用葉子（unused leaf）」。

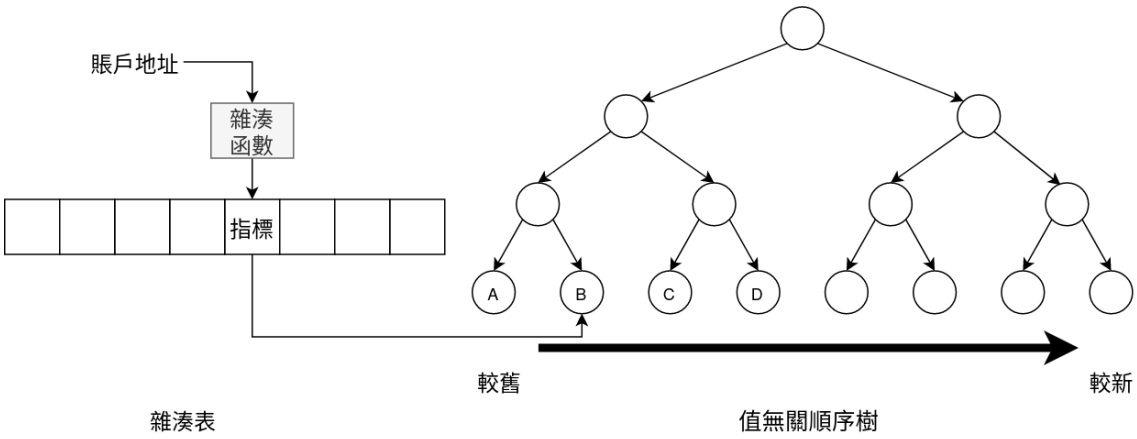

圖 3.10: 雜湊 + 值無關順序樹

一個葉子節點的資料可以用以下結構表示：

Struct *Node<Key, Value>* **contains**

> Node *left;
>
> Node *right;
>
> Key key;
>
> Value value;
>
> int index;

end

非葉子節點由於不存資料，可以簡化成：

Struct *Node* **contains**

> Node *left;
>
> Node *right;

end

get(key) 時，先從雜湊表中查詢有用葉子的指標，將被查詢到的有用葉子改為未用的，並在當前最右有用葉子再往右一個的未用葉子中寫入原葉子的資料，最後更新雜湊表 key 對應的的指標為新葉子的指標。

put(key, value) 時，先從順序樹根部向下找出最左側的有用葉子，將之改為未用的，並利用最左側有用葉子中的 key 欄位，去刪除雜湊表中對應的鍵值對，在當前最右有用葉子再往右一個的未用葉子中寫入資料，再將（key , 新葉子的指標）此一鍵值對寫入雜湊表。

無論是 get 還是 put ，每次都會往右多佔用一個葉子，如果當前最右的有用葉

子已經在整棵樹的最右側了，就必須執行一次全複製，把所有有用的葉子節點按照原本的順序緊密的排列在新值無關順序樹的左側。

圖 3.11 演示了一連串的值無關順序樹操作，其中狀態 7 到狀態 8 的時候發生了全複製。圖中淺藍色節點表示它是未用的，注意到我們將所有葉子都未用的子樹的所有節點也都塗成淺藍了。

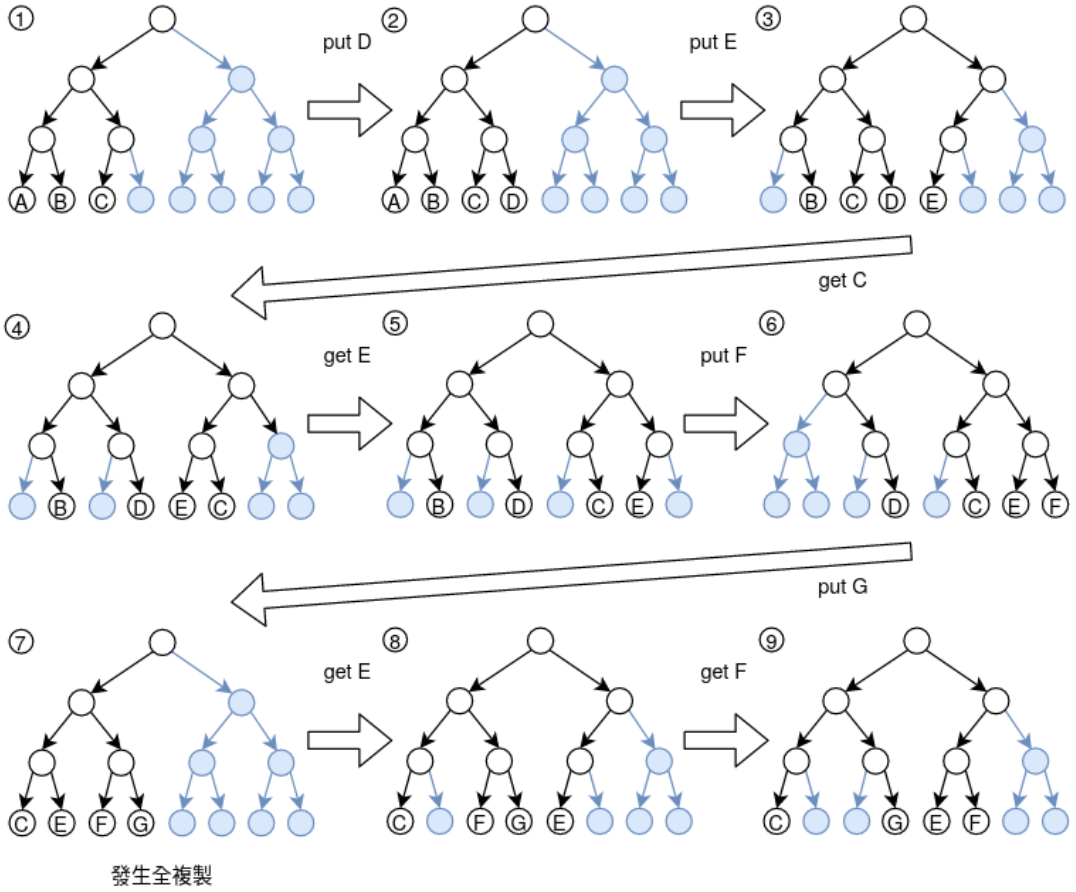

圖 3.11: 對值無關順序樹進行一連串操作

在實作中，不需要真的為淺藍色節點分配記憶體空間，指向純淺藍色子樹的指標會是一個空指標 (null pointer)。

操作細節

我們再接著探討 get, put 的操作細節。

要如何找出最左側的有用葉子？從根部開始往下，若左側子節點非空，往左走，若左側子節點為空，往右走，走到高度為零時，就走到最左側有用葉子了。

從雜湊表中得到有用葉子的指標時，要如何知道該葉子在值無關順序樹中的位置，以刪除它？在葉子節點中，會儲存一個序號（也就是虛擬碼中的 $index$）表示自己是從左到右的第幾個葉子，刪除葉子時，會從根一路往下，判斷 $index\&(1 << height)$ 是否為 0 來決定往左還是右。

時空間複雜度分析

以下分析中，值無關順序樹容量皆為 n 。

若沒有發生全複製，get, put 都從根到葉子兩次，值無關順序樹的高度是 $O(\log n)$，時空間複雜度都是 $O(\log n)$。

如果發生全複製，就得遍歷一次值無關順序樹，將所有有用葉子集結建一棵新樹，注意到滿二元樹的總節點數量是葉子數量 * 2 - 1，不會超過 $n * 8$ ，因此全複製的時空間複雜度是 $O(n)$ 。

$O(n)$ 的複雜度似乎有點高，但是每一次的 get/put 操作都只會讓有用葉子往右走一格，一棵樹至少要 n 次操作，才會使有用葉子走到整棵樹的最右側。值無關順序樹是持久化資料結構，會同時維護多個版本，若每次在所有版本中選擇一個版本修改，那平均要修改 n 次以上，才會觸發一次全複製，在這種情況下，把全複製的成本攤銷 [17]（amortized）到每一次操作，時空間複雜度是 $O(1)$ ，get, put 的時空間複雜度依然是 $O(\log n)$ 。

順序樹的高度不一定非得是 $1 + \lceil \log_2 n \rceil$ ，可以如替罪羊樹 [10]（scapegoat tree）一樣用一個常數去調控。例如將高度設成 $1 + \lceil \log_\alpha n \rceil$ ，只要 α 是一個大於 1 的常數，使得順序樹的葉子數量至少是快取容量的 $alpha$ 倍，仍然能分析出上一段的時空間複雜度。

最壞的情況是，每次都選擇將要發生全複製的版本來進行修改，如此每個 get, put 操作的時空間複雜度都會是 $O(n)$ ，但在 PoW 共識的系統中，必須有足夠的算力才能製造出一個區塊，惡意打擊特定一個節點所耗費的代價巨大，而收效甚微。此外，不一定要等有用葉子走到整棵樹的最右側才進行全複製，可以將全複製的時間點隨機化，使得惡意的對手難以確定一個節點全複製的時刻。

嚴格來說，一個葉子的佔用的空間是 $O(\log n)$ ，因為葉子必須記錄自己的序號，而序號在二進位的長度與值無關順序樹的高度相同。但這不影響前述的空間

複雜度分析，即使考慮這點，時空間複雜度依然是 $O(\log n)$ 。況且，實作中不可能會創建容量超過 CPU 定址空間大小的值無關順序樹，序號必定能用 64bit 的整數存下。

3.4.3 雜湊 + 最小堆積

我們也可以巧妙地用雜湊搭配最小堆積（min heap）來實作持久化 LRU 。

在這種構造中，雜湊表的鍵為賬戶地址，值為 (時間戳, 賬戶資訊) 。堆積用單向指標來實作，堆積中的每一個節點會儲存賬戶地址以及時間戳。

以下是堆積中一個節點的結構，在我們所維護的最小堆積中，父節點的時間戳都小於自己的時間戳。

```
struct Node<Key> {
    Node *left;             // 左子節點
    Node *right;            // 右子節點
    Key key;                // key 在此應用中即為賬戶地址

    // 時間戳，可用整數作為 TimeStamp
    // 例如之前可能有 using TimeStamp = int;
    TimeStamp timestamp;
};
```

時間戳擁有「只增不減」的特性，在 LRU 應用中的最小堆積的節點的時間戳只會變大，因此節點只會下移（shiftdown），不會發生上移（shiftup），也因此並不需要記錄父節點的指標。

一般的堆積在加入新元素的時候，也有可能發生上移，但因為時間戳「只增不減」，新元素的時間戳一定是最大的，加入堆積尾部後，無需上移，整個堆積依然符合最小堆積的規則。

下移的虛擬碼如下：

```
void shiftdown(Node *node) {
    // 若沒有子節點，結束
    if (node->left == NULL && node->right == NULL) {
        return;
    }
```

```
    // 若有子節點，找出時間戳較小的
    Node *small_child = node->left;
    if (node->right != NULL &&
        node->right->timestamp < small_child->timestamp) {
        small_child = node->right;
    }

    // 若子節點時間戳小於父節點，交換父子節點
    if (small_child->timestamp < node->timestamp) {
        swap(node->key, small_child->key);
        swap(node->timestamp, small_child->timestamp);
        // 向下遞迴
        shiftdown(small_child);
    }
}
```

雜湊表則可表示成

```
struct Info<Value> {
    TimeStamp timestamp;
    Value value;            // Value 為賬戶狀態
}
map<Key, Info> table;
```

維護條件

我們考慮快取已滿時的狀態，假設當前的堆積符合最小堆的規則，亦即，除了根節點以外的所有節點都滿足

條件一、父節點的時間戳都小於自己的時間戳

除此之外，我們還要維護另一個條件

條件二、堆積的根的時間戳與雜湊表中的時間戳相同

解釋一下條件二：雜湊表中我們可以由地址去查詢到時間戳；在堆積中的每個節點，地址也都對應到一個時間戳。然而在整個 LRU 進行操作的過程中，除了堆積的根的時間戳，雜湊表中的時間戳跟堆積中的時間戳並不總是同步的，這並不

要緊，只要保證堆積的根的時間戳與雜湊表中的時間戳同步，我們就能夠順利完成 LRU 的操作。

LRU get

若快取沒有命中，不做任何事；若快取命中，更新雜湊表中的時間序。

寫為虛擬碼如下：

```
if (table.has(address)) {    // 如果雜湊表中含有所求地址
    Info info = table.get(address);
    info.timestamp = current_timestamp++;
    table.set(address, info);
    maintain_root();
}
```

在以上過程中，我們只更新了雜湊表中的時間戳，因此雜湊表中的時間戳在 get 之後就會與堆積中的時間戳不同，當 get 的地址非根時無所謂，但若恰巧 get 到了堆積的根對應的地址，我們就得執行一個額外操作來維護前一節所提到的條件二。

這個額外操作的虛擬碼如下：

```
void maintain_root() {
    while (heap.root.timestamp != table[heap.root.key].timestamp) {
        heap.root.timestamp = table[heap.root.key].timestamp;
        shiftdown(heap.root);
    }
}
```

maintain_root 會更新堆積的根的時間戳，並將根下移，若下移後堆積的新根的時間戳依然與雜湊表中的時間戳不同，重複此過程。

LRU put

若快取沒有命中，我們得丟棄最舊賬戶，並加入新賬戶。具體操作中可將時間戳最小的節點，亦即堆積的根（由於條件二，堆積的根總是最舊的）改為欲置入的新賬戶，並執行 maintain_root 來維護堆積規則。

若快取命中，更新雜湊表中的時間序跟賬戶狀態，並注意是否更動到堆積的根。

虛擬碼如下

```
void lru_put(Key address, Value value) {
    if (table.has(address)) {
        // 快取命中
        Info info = table.get(address);
        info.timestamp = current_timestamp++;
        info.value = value;
        table.set(address, info);
        maintain_root();
    } else {
        // 快取失效，丟棄最舊賬戶並塞入新賬戶資訊
        table.remove(heap.root.key);
        table.set(address, { current_timestamp, value });
        heap.root.key = address;
        heap.root.timestamp = current_timestamp++;
        shiftdown(heap.root);
        maintain_root();
    }
}
```

如何填滿快取

無論快取是否已經填滿，get 操作的行為都相同，但 put 時若快取失效，當快取未滿時，必須往堆積中加入新節點。因為時間戳只增不減，將新節點塞入堆積的尾部後，新堆積仍會符合最小堆積的規則。

至於要如何找到堆積的尾部，我們可以維護一個 count 變數來記錄當前的堆積節點數量，再透過讀取 count 的二進位表示式來從堆積的根走到尾部。

虛擬碼如下：

```
Node *get_tail(int count) {
    // high_bit 取得一個數字的二進位表示式中最左側的 1 的位置
    // 例如 high_bit(0b1011) = 4, high_bit(0b0010) = 2
    int h = high_bit(count) - 1; // h 為 count 代表的節點的深度
```

```
    Node *ret = heap.root;
    while (h > 0) {
        if (count & (1 << h)) {
            ret = ret->left;
        } else {
            ret = ret->right;
        }
        h--;
    }
    return ret;
}
```

持久化

我們已知雜湊可輕易持久化。而僅支援下移（shiftdown）而無需支援上移（shiftup）的堆積亦可以路徑複製來完成持久化。

堆積的 shiftdown 由根遞迴向下執行，至多執行至葉子，時間複雜度 $O(\log n)$，採用路徑複製來持久化時，由於修改到的節點恰巧組成一條從根到葉子的路徑，直接複製該路徑即可，耗用空間複雜度 $O(\log n)$ 。

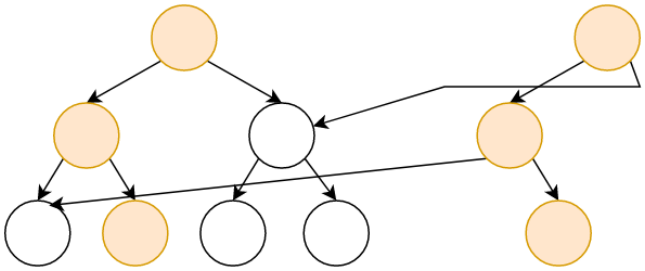

圖 3.12: 持久化堆積 shiftdown

時空間複雜度分析

假設快取大小為 n 。我們分析幾個重要函式的時間複雜度。

由上節討論中，我們知道 shiftdown 的時空間複雜度是 $O(\log n)$。

maintain_root 在最壞情況下需要將所有堆積中的時間戳都更新，此時會執行 n 次 shiftdown ，時空間複雜度為 $O(n \log n)$ 。

但我們觀察到，maintain_root 執行的 shiftdown 次數不超過堆積中時間戳與雜湊表中時間戳相異的數量，而每一次 LRU 的 get, put 頂多只會使堆積中時間戳與雜湊表中時間戳相異的數量增加 1 ，總的來說，執行 shiftdown 的次數會少於等於執行 LRU get, put 的次數，因此我們若將 maintain_root 的成本攤銷到每一次的 get, put 操作，一次 get, put 將分攤不到一次 shiftdown。

攤銷掉 maintain_root 的開銷之後，每一次 get, put 操作增加了 $O(\log n)$ 的時空間開銷，再加上 get, put 需要查找／插入雜湊表所消耗的 $O(\log n)$ 的時空間開銷，get, put 的時空間複雜度仍為 $O(\log n)$ 。

由以上分析，以最小堆積為基礎的實作在時空間複雜度上與先前我們所討論的其他資料結構相當，雖然這個時空間複雜度跟值無關順序樹一樣是攤銷得來的，最壞情況下一個操作可能較耗時，但如同順序樹分析複雜度的章節提到的一樣，依此惡意打擊的成本巨大，故在實際應用中仍可考慮此實作。

Chapter 4

實驗設計與結果

在本章中，我們將在不同工作量下評估以不同資料結構分別實作的持久化 LRU 性能。

4.1 實驗環境

實驗均在以下環境中進行

- CPU：AMD Ryzen 7 2700X (8 核心 16 執行緒、3.6 GHz、32K L1d 快取、64k L1i 快取、512K L2 快取、8M L3 快取)
- 記憶體：DDR4 64 GB
- 作業系統：Linux manjaro 4.19.133-1-MANJARO x86_64
- 編譯器：GCC 10.1.0

4.2 資料結構

我們以 C++ 17 撰寫了四種資料結構：

除了 3.4 節提出的三種資料結構之外，還實作了雜湊 + 雙向鏈表的資料結構，這是最簡易的持久化 LRU 實作，它的各項操作與我們在 3.4 節介紹的暫時性 LRU 資料結構相同，但當它需要創建新版本時，會複製一份舊版本再繼續操作，我們以此來對照具有共享結構特性的其他三個持久化資料結構。

- 雜湊 + 雙向鏈表（3.4 節）
- 持久化雜湊 + 持久化紅黑樹（3.4.1 節）
- 持久化雜湊 + 持久化值無關順序樹（3.4.2 節）
- 持久化雜湊 + 持久化最小堆積（3.4.3 節）

由於要模擬的是區塊鏈，所以所有的工作量都以區塊為單位，而一個區塊會包含多個 get/put 。為此，以上四種持久化資料結構的實作都支援了 transient [15]，也就是說，資料結構不需要對每一個 get/put 都生成一個新版本，而是暫時生成一個版本，在其上進行多個操作，再將它固定為不可變的，由於減少了大量的中間狀態，此舉能夠大幅提升效能。

函式庫使用方面，雜湊直接使用了 C++ 標準函式庫的 unordered_map ，持久化雜湊則採用了著名的 C++ 不可變資料結構函式庫 immer ，其餘的雙向鏈表、持久化紅黑樹、持久化值無關順序樹、持久化最小堆積則自行撰寫。

往後，我們會以雙向鏈表、紅黑樹、值無關順序樹、最小堆積來簡稱這四個複合資料結構。

4.3 速度

4.3.1 工作量

我們採用一個簡單的模型來模擬區塊生成，網路中有 n 個節點，每個節點都會針對自己認定的最高區塊（亦即，它所認同的最長鏈的頂端）挖礦，初始時，每個節點的最高區塊都是創世區塊 (genesis block)。其後，有若干回合，每個回合中，每個節點有一定概率會挖出新區塊。若它沒有挖到，則會投奔其他節點挖出的同最高區塊的後繼區塊，若該最高區塊沒有任何後繼區塊，則會隨機投奔其他最長鏈。若沒有任何節點挖到礦，就將同樣的過程再來一次。

在所有的工作量中，都是假定網路中有 10 個節點、每個節點在一個回合內挖到礦的機率是十分之一，並且在挖出 1000 個區塊之後停止。在此設定下，同個高度大約會有 1.5 個區塊。

從 etherscan 可以得知，以太坊現今的 gas limit 大約是一千萬，而一筆交易的 gas 費用為 21000 ，一個區塊大約可以包含 500 左右的交易。因此工作量的一個區塊的 get/put 指令數量設定為 500 。

此外，還有一項參數是 get/put 的佔比，對於一般的以太坊交易而言，每次的付款都會導致賬戶的狀態改變，然而智慧合約的情形就不一定，因此 get, put 兩種指令都有可能出現，以下若無額外說明，put 都占 50%（get 也占 50%）。

在開始任何一項工作前，我們都會先將整份快取裝滿，若非如此，當快取大小設定太大時，需要很長一段時間才能將它裝滿，快取未滿時的狀態並非我們關注的，因為區塊鏈動輒幾百萬個區塊，極大部分時間，快取都是裝滿的。

4.3.2 調整快取大小

第一項實驗將快取命中率固定在 100% ，調節快取的大小（鍵值對的數量）。

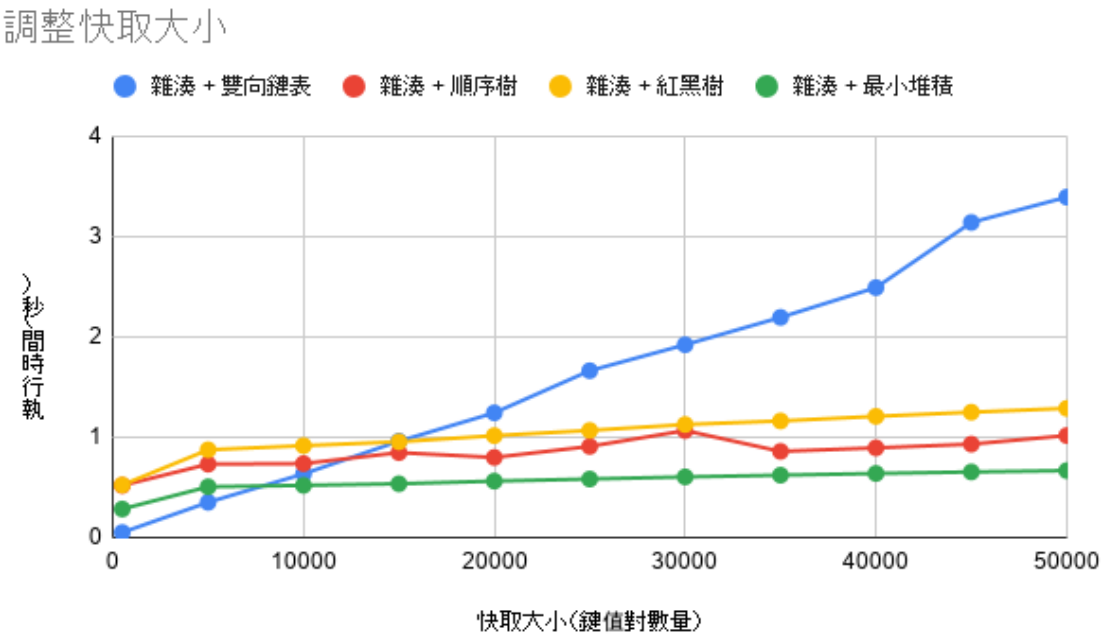

圖 4.1: 調整快取大小

所畫出的折線圖 4.1，可以看出，雙向鏈表所耗費的時間基本上正比於快取的大小，顯見雙向鏈表的主要消耗在於複製，快取有多大，每次區塊更新，所要複製的量就有多大。

紅黑樹、值無關順序樹、最小堆積的成長曲線就十分平緩（$O(\log n)$），注意到

值無關順序樹的折線中，相較紅黑樹、最小堆積的穩定增長，是有所震盪的，這跟值無關順序樹的葉子數量是 $1 + \lceil \log_2 n \rceil$ 有關，它的葉子數量更加離散，當快取大小達到 2^k 次方時，葉子數量往上翻倍，此時就會影響效能，導致震盪。

4.3.3 調整快取命中率

將快取大小定為 30000 ，調整命中率。

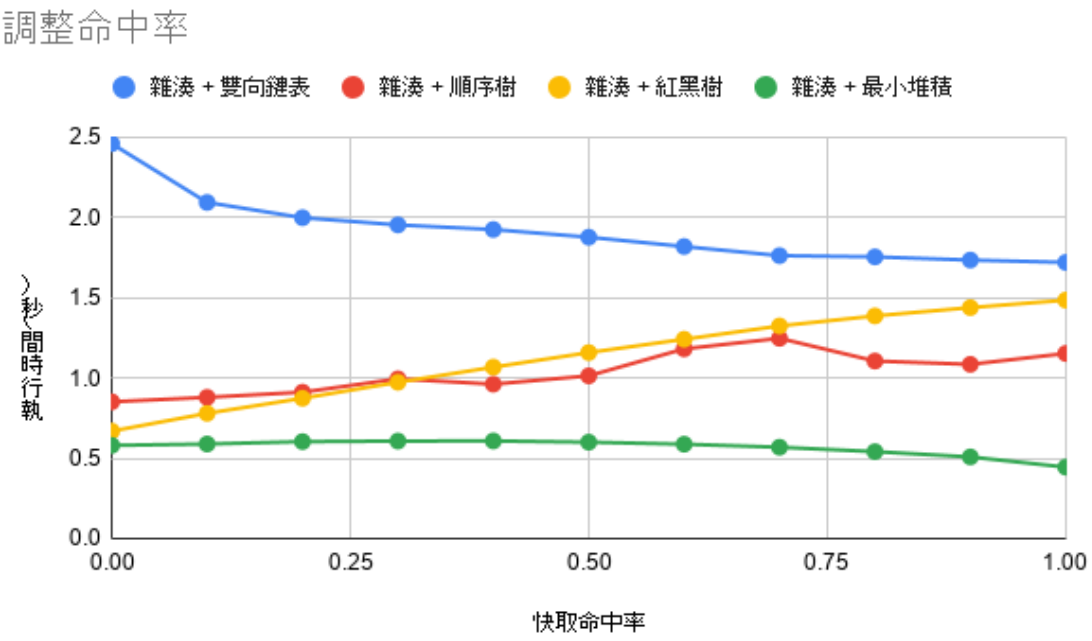

圖 4.2: 調整命中率

圖 4.2 中可見，命中率對雙向鏈表並沒有什麼影響，因為它的主要消耗來自於複製，快取命中之後要做的事情只是搬動幾個指標，不花什麼時間。紅黑樹、值無關順序樹的耗時都會隨命中率提高而有所上升，因為它們主要的耗時來自於路徑複製，命中越多，需要複製、創建的分支也就越多。而最小堆積在命中率提高時，耗時變少，因為快取命中時不需要從雜湊中剔除鍵值對，也不一定會觸發 maintain_root 操作。

4.3.4 調整 put 比例

將快取大小定為 30000 ，命中率設為 1.0 ，調整 put 佔兩種指令的比例。從圖 4.3 可見基本不影響效能。

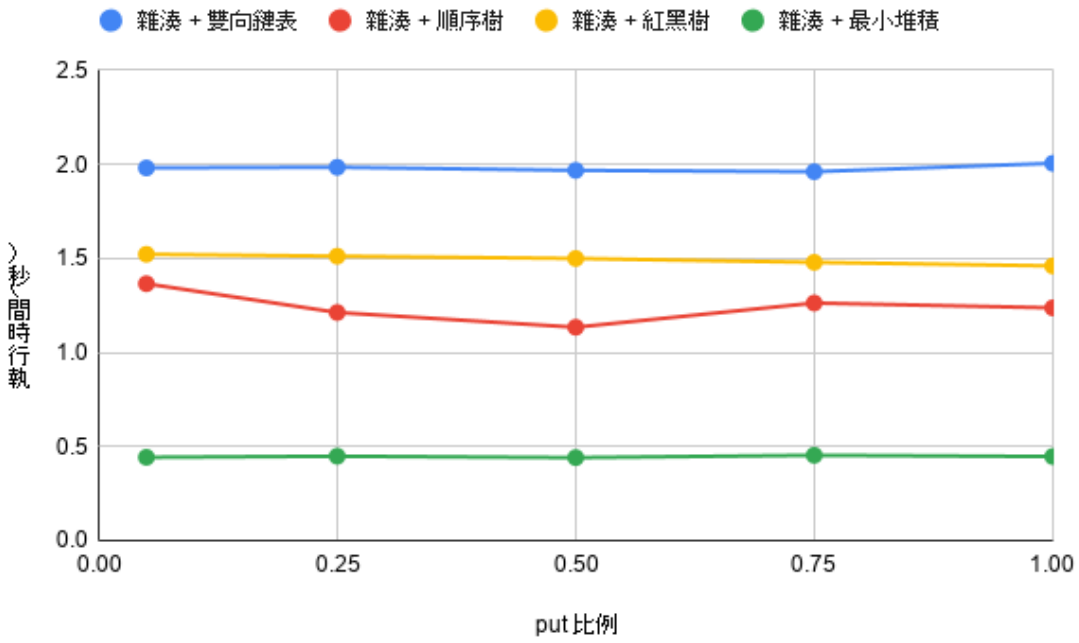

圖 4.3: 調整 put 比例

4.4 失效代價（miss penalty）

如果驗證一筆交易時，它相關賬戶的資訊不在節點的快取中（快取失效），這筆交易就得附上證明以供節點驗證。因此快取失效的代價有二：(1) 接收證明所需要的網路 I/O ，這與證明所佔的空間有直接關係。(2) 驗證證明所消耗的 CPU 運算時間。

而快取命中所耗費的時間，我們可以觀察圖 4.3 ，最小堆積實作在快取大小 30000 、put 比例 100%、命中率 1.0 時，0.447 秒內執行了 1000 個區塊，而一個區塊中有 500 條指令，亦即一個區塊的平均執行時間是 0.447 毫秒，當快取更小的時候，這個數字還會更低。

我們挑選最常見的梅克爾 Patricia 樹的驗證成本來與最小堆積實作比較，函式庫選用了兩大以太坊實作之一的 Parity Ethereum 的 trie-db 模組。

影響梅克爾 Patricia 證明的因素有二：(1) 葉子的數量 s ，證明的大小就約為 $O(\log s)$ ，要驗證一個證明的時間複雜度也是 $O(\log s)$ 。(2) 批量處理時，一批的數量。梅克爾證明就是梅克爾樹的一個分支，當一個節點被多個分支共享時，不必重複計算，因此一次驗證 k 個證明的時間會短於 k 乘以驗證一個證明的時間。

由 etherchain.org 的資料我們可以得知，現今的以太坊賬戶數量已經超過八千

萬，所以我們先在梅克爾 Patricia 樹中隨機插入八千萬筆資料（也就是說葉子的數量為八千萬），並且根據此前一個區塊中 500 個交易的假設，批量驗證 500 個證明，量測得到所費時間 5.536 毫秒，明顯要比在最小堆積實作中取用快取來得耗時，即使與實驗中稍慢的紅黑樹跟值無關順序樹實作比較，也是大大不如。

而一個區塊的證明大小約為 900 KB ，1000 個區塊就是將近 900 MB ，若使用當今一般的家用網路，下載這些額外證明所需的時間又比驗證證明更加耗時。

4.5 總結

實驗驗證了值無關順序樹、紅黑樹、最小堆積較低的時空間複雜度，在快取大小增大到一定程度後，表現上顯著領先總是複製自身的雙向鏈表。

同時也實驗、量測了梅克爾 Patricia 樹的證明大小以及驗證速度，瞭解到通常情況下直接讀取快取要比下載、驗證證明要來得快速。根據這些數據，能夠推論若以梅克爾 Patricia 樹做為 vector commitment 時，淺狀態區塊鏈若採取適當的快取大小，能夠在 LRU 策略下帶來速度提升。

Chapter 5

未來工作

5.1 快取拆分

若要讓淺狀態區塊鏈支援智慧合約，單以一個賬戶為快取單位會導致一些問題，因為每個智慧合約的狀態所佔用的空間可能相差巨大，如果快取中的每一個狀態都非常大，那整個快取就會佔用很多空間，反之若快取中的狀態都很小，整個快取佔用空間就很小，當快取佔用的空間不穩定時，一條鏈上節點的維護者們就很難決定該如何設置快取大小。例如，若一條鏈預計要給物聯網裝置使用，這些裝置的記憶體通常不超過 1GB ，但快取佔用空間不穩定，即使選擇要快取的賬戶數量很少，依然可能導致記憶體耗盡。

我們可以借鑑作業系統的分頁設計，將智慧合約的狀態切割成等體積的多個小塊，每次快取只會存取到用到的小塊，如此就能夠使快取佔用空間變得穩定。

Chapter 6

總結

本論文描述了淺狀態區塊鏈的設計，並探討了多種快取策略的優缺點。特別對於 LRU 快取策略，我們設計了持久化雜湊加持久化紅黑樹的組合、持久化雜湊與值無關順序樹的組合、持久化雜湊與持久化堆積的組合，這三種時空間複雜度相當的複合資料結構。

最後進行實驗，瞭解到在實驗環境中採用這幾種資料結構來實作 LRU 快取策略，獲取快取時的速度皆顯著高過真實以太坊節點（parity）中直接驗證梅克爾證明的速度，佐證了淺狀態區塊鏈有機會帶來效能提升。

參考文獻

[1] P. Bagwell. Ideal hash trees. Technical report, 2001.

[2] J. Benaloh and M. De Mare. One-way accumulators: A decentralized alternative to digital signatures. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 274–285. Springer, 1993.

[3] D. Boneh, B. Bünz, and B. Fisch. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*, pages 561–586. Springer, 2019.

[4] D. Catalano and D. Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[5] A. Chepurnoy, C. Papamanthou, and Y. Zhang. Edrax: A cryptocurrency with stateless transaction validation. *IACR Cryptology ePrint Archive*, 2018:968, 2018.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[7] R. Dahlberg, T. Pulls, and R. Peeters. Efficient sparse merkle trees. In *Nordic Conference on Secure IT Systems*, pages 199–215. Springer, 2016.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, 1986.

[9] T. Dryja. Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set. Technical report, IACR Cryptology ePrint Archive, 2019.

[10] I. Galperin and R. L. Rivest. Scapegoat trees. In *SODA*, volume 93, pages 165–174, 1993.

[11] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21. IEEE, 1978.

[12] R. Hinze and R. Paterson. Finger trees: a simple general-purpose data structure. *Journal of functional programming*, 16(2):197–217, 2006.

[13] S. Lists. A probabilistic alternative to balanced trees william pugh. *Communications of the ACM*, 1990.

[14] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[15] J. P. B. Puente. Persistence for the masses: Rrb-vectors in a systems language. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–28, 2017.

[16] X. Qian. *Improved authenticated data structures for blockchain synchronization*. PhD thesis, 2018.

[17] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic Discrete Methods*, 6(2):306–318, 1985.

[18] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.