华中科技大学

大数据存储系统与管理报告

姓　　名：　　庄景豪

学　　院：　　计算机科学与技术学院

专　　业：　　计算机科学与技术

班　　级：　　CS2109

学　　号：　　U202114055

指导教师：　　华宇

分数	
教师签名	

2024 年 04 月 23 日

目录

1 背景 3
1.1 题目介绍 3
1.2 本文贡献 3
2 理论分析和算法设计 4
2.1 原始的 Cuckoo hashing 算法 4
2.2 two-array-Cuckoo hashing 5
2.3 多路相连优化 5
2.4 基于并查集的 Cuckoo 循环提前预知 5
2.4.1 问题抽象 5
2.4.2 算法设计 7
2.4.3 复杂度分析 8
3 代码实现 9
3.1 文件介绍 9
3.2 结构介绍 9
3.3 核心代码解析 10
4 性能测试 15
4.1 测试方法 15
4.2 性能分析 15
4.2.1 Test1 结果分析 15
4.2.2 Test2 结果分析 16
5 总结 17

1 背景

1.1 题目介绍

Cuckoo hashing 是一种 hash 表的实现方式，hash 表是一种高效的数据结构，它能够通过 hash 函数将键值映射到存储桶中的特定位置，通常是将一个大的值域映射到一个较小的值域当中，从而实现高效的数据检索。

在最优的情况下 hash 表的插入，查询和删除都能够达到平均 O(1)的复杂度。然而 hash 表面临着 hash 冲突的问题，即不同的 key 经过 hash 函数之后被映射到相同的 hash 值，虽然当映射空间足够大时，出现 hash 碰撞的概率很小，但是根据生日悖论，当我们插入的数量增加到足够多时，碰撞的概率将会大大增加，例如每 23 个人当中出现相同生日的概率居然超过了 50%。Hash 碰撞可能会导致插入和查询的复杂度退化，甚至插入或查询失败。

为了最小化 hash 碰撞带来的影响，前人们提出了许多常用的方法，例如：链地址法，开放地址法，再 hash，完美 hash 等。而 Cuckoo hash 就是其中的一种，这个名字源于某些杜鹃的行为，每一个 key 都可以通过两个 hash 函数映射到两个不同的位置上，当 hash 碰撞发生时，新插入的值会尝试挤掉先前在这个位置上的值，而先前位置上的值会尝试插入到其备用位置（另一个 hash 值得到的位置）上去，并且依次类推直到所有 key 都能找到一个不碰撞的位置。

然而 Cuckoo hashing 也面临着一些问题，有些时候上述的策略并不能找到一个没有冲突的解决方案，而是会陷入循环当中，并且如果查找链的深度过深即使找到解决方案也会使得插入的时间复杂度很糟糕，本报告旨在探寻如何优化这一问题。

1.2 本文贡献

1. 对自己实现的 2array 优化的 cuckoo hashing 和 multi-slot 优化的 cuckoo hashing 进行了性能测试。
2. 创新地提出了一种基于并查集优化的 Cuckoo-driven Way 循环检测和避免，并且分析了其正确性、可行性和时间复杂度。

2 理论分析和算法设计

2.1 原始的 Cuckoo hashing 算法

最原始的 Cuckoo hashing 算法中两个相互独立的 hash 得到的 index 都对应同一个存储桶中的索引。

Cuckoo hashing 的查找逻辑即是通过这两个相互独立的 hash 获得两个索引，然后分别访问这两个索引中存储的键值，如果其中有一个存储的键和目前我们查询的键相同，那么说明这个位置就是我们要查询的位置。

删除逻辑和查找逻辑相似，在查找到我们要删除的位置后，只需要将当前位置的键值设置为非法值，或者标记为被释放即可。

插入逻辑比较复杂，同样的，我们先通过这两个相互独立的 hash 函数获得两个相互的独立的索引。如果这两个索引的位置都是空余的，那么我们随机选择一个位置插入键值即可；如果这两个位置只有一个位置是空余的，那么我们直接将键值插入到唯一空余的索引当中去。如果这两个位置都不空余，那么我们就要考虑采取 Cuckoo driven way 策略来尝试调整出来一个空余的位置。

具体地：我们先随机选择着两个位置中的一个，我们假设选择的索引为 index，那么我们去查询 index 这个位置先前保存的键为 key’，因为我们直到每一个键都对应两个位置，由于这里 key’已经被保存到了 index，那么我们就可以利用这两个 hash 函数来获得 key’对应的两个索引，其中一个是 index，而另一个我们令为 index’，假如 index’对应的位置为空，那么我们可以直接将原先 index 位置存储的键值移动到 index’，然后将新插入的键值存储到 index。假设 index’不为空，那么我们只能递归地使用相同地策略查询 index’存储的 key’’对应的另一个 index’’是否空余。为了方便讨论我们将这一递归策略命名为 seize_place。

不难发现由于每一个键 key 只对应了两个位置，并且其中一个位置已经被占用，因此其实 seize_place 查询始终只有一条分支，复杂度是 O(k)的，除非进入了死循环。

对于死循环的判断有以下两种方法：

一种简单的想法是限制上述 seize_place 查询的最大深度为 max_depth，当超过上述深度之后依然没有查询到空余位置时，我们便认为无法查询到这样的位置，

从而返回。另一个简单的想法是每一次执行 seize_place 递归查找时都标记自己经过的索引，如果遇到了访问过的索引那么就找到了可行解。这两种方法各有各自的有点，第一种方法的缺点是有时会忽略掉潜在的可行方案，第二种方法的缺点是虽然不会漏掉可行方案，但是 insert 的复杂度可能会很高，并且即使使用时间戳优化访问标记不用每次清空。其仍然需要额外的空间来存储访问标记。

2.2 two-array-Cuckoo hashing

这是一种普遍的对于原始 Cuckoo hashing 的优化，将我们先前提到的两个 hash 得到的两个索引值分别作为两个不同的存储桶的索引。

可以通过一个简单的例子来证明上述简单的策略是有效的：在极端情况下，键 key 通过两个 hash 映射出来的值是相同的，假如我们只使用一个存储桶，这两个 index 就成为了一个自环，那么 Cuckoo hashing 就退化成为了最简单的 hash。

我在 Cuckoo hash 文件夹中的 Cuckoo2Array.hpp 实现了一个 two-array-Cuckoo hashing 的类，并且在 test1.cpp 中对其进行了测试，测试结果见第 4 节。

2.3 多路相连优化

这是一种简单，但是极为有效的优化，通过增加存储桶每个槽位能够存储的键值对个数，来降低 hash 冲突的频率。

我在 Cuckoo hash 文件夹中的 multiSlotCuckoo.hpp 实现了一个这样优化的 2-array Cuckoo hashing 的类，并且再 test2.cpp 中对其进行了测试，测试结果见第 4 节。

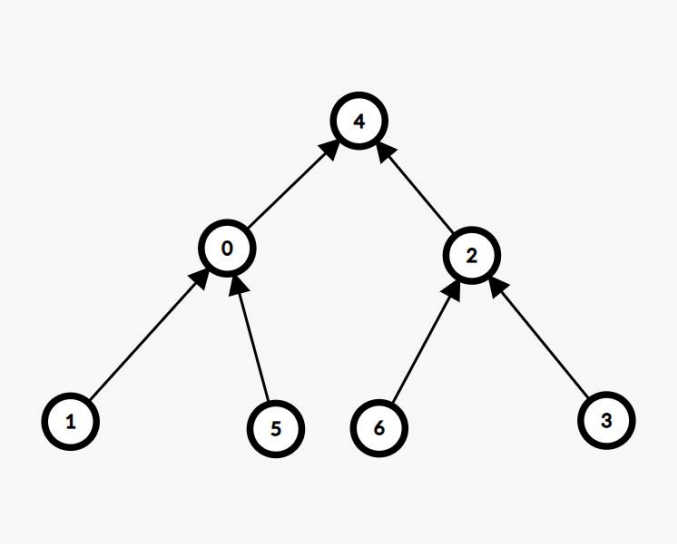
2.4 基于并查集的 Cuckoo 循环提前预知

2.4.1 问题抽象

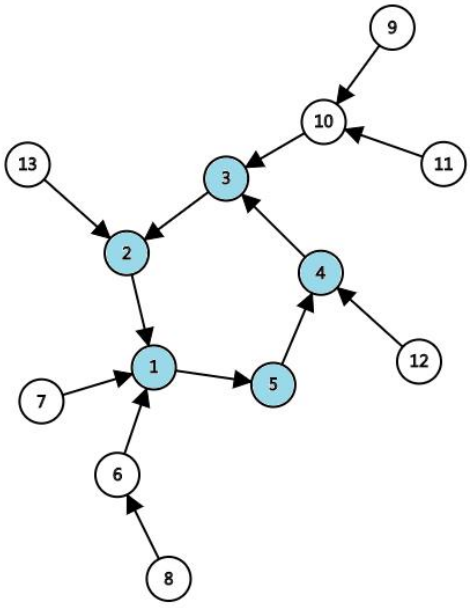
考虑将 2-array-Cuckoo hashing 中的 seize_place 递归寻找可行方案的这一过程抽象为一个图论问题：对于存储桶中所有非空余的位置 index，其存储的值为 key，value。则 h1=hash1(key)和 h2=hash2(key)中必然有一个值等于当前的索引 index（毕竟它已经被存储进来了，而这对键值只可能被存储在 h1 和 h2 当中），

那么假设当我们的 seize_place 查找到这个节点时（我们将存储桶中一个 index 对应的位置称为一个节点），实际上他只会有一条出边（指 seize_place 过程中这一节点接下来只会进入到确定的唯一节点），并且不难证明一个节点可能有多条入边（指 seize_place 过程中可能从多个节点进入到这一节点）。

有了上述的性质，我们能够知道假如将 seize_place 的所有转移边构建成为一个有向图，删掉所有指向空余位置的有向边（毕竟如果 seize_place 找到有向边就相当于找到可行方案），那么这个有向图一定是一个内向基环树森林（即由多个树形结构和内向基环树组成）。考虑树形结构和内向基环树的性质，当我们的 seize_place 的起始节点位于一颗树形结构上时，我们最终一定会来到树根（因为树根的入度不为零而出度为零，出度为零只能是因为其指向了一个空余节点，而这条边被我们删掉了），因为一定能到达树根，所以一定不会进入循环，一定能找到插入方案。如下图所示，所有点最终都会到达节点 4.



假设我们 seize_place 的起始节点位于一颗内向基环树上，如下图所示，无论从哪个节点出发，我们最终都会进入到蓝色节点构成的环当中，在这种情况下，seize_place 最终会进入死循环从而无法找到解。



这样一来，只要我们能够在 seize_place 最开始就知道我们的起始节点究竟是位于一颗树上还是位于一颗有向基环树上，就能立即判断出是否会进入死循环。

2.4.2 算法设计

在将问题抽象转化为一个图论问题之后，我们不难想到能够使用并查集来动态维护这一个图。

还记得我们将所有连向空余节点的边都删除了当作无边吗？这也意味着一开始，当我们没有插入任何数据时，这张图是一个空图。像传统并查集算法一样，我们让所有空节点的父亲为自己（即出边指向自己）。每当一个空节点插入一对键值（key，value），变为非空节点时，我们进行以下操作：计算出另一个 hash 函数对 key 的映射，然后连接一条有向边到这个映射索引对应的节点，注意：由于有两个存储桶，所以我们令第一个存储桶索引 i 的节点编号为 i，第二个存储桶索引 i 的节点编号为 i+capacity（其中 capacity 对应存储桶的容量）。

在执行 seize_place 前，首先使用 getFa 函数，查看当前并查集的树根，如果树根对应的存储桶位置为空，那么一定能找到可行方案，否则我们便可以预知一定会进入死循环不用执行 seize_place 了。此外由于并查集还可以维护节点的深度，因此，在一开始我们还可以选择深度较浅的那个存储桶开始 seize_place，进一步优化 insert 的时间。

2.4.3 复杂度分析

常见的并查集实现方法均包含了路径压缩与启发式合并这两个优化来保证并查集的时间复杂度为 O(m \alpha(m,n))（其中 alpha 函数代表反阿克曼函数，是阿克曼函数的反函数，阿克曼函数增长极其迅速，在 m=n=4 时数量级就达到了 2^2^10^19729，因此反阿克曼函数增长极为缓慢，可以在计算机科学中认为其为不大于 4 的常数），下面我将讨论这两种优化用在此处的可行性与复杂度。

启发式合并：指在合并并查集时，将较小的一颗子树合并到较大的一颗子树当中去。然而，由于我们的边都是有向边，因此无法实现启发式合并。

路径压缩：指在每次并查集查询时遍历过的点的父节点都直接指向其根节点，由于我们的算法只需要知道根节点是否空余，就可以预知是否会进入死循环，不难证明，路径压缩适用于这里的并查集。

那么只使用路径压缩优化的并查集时间复杂度是多少呢，在 Tarjan 的论文中，证明了不使用启发式合并、只使用路径压缩的最坏时间复杂度是 O (m \log n)。在姚期智的论文中，证明了不使用启发式合并、只使用路径压缩，在平均情况下，时间复杂度依然是 O (m\alpha(m,n))。也就是说每一次 insert 的复杂度从 O(max_depth)变为了均摊 O(\alpha(m,n)).

然而我们还需要考虑从 hash 表中删除一个元素的时间复杂度，如果采取了路径压缩，我们是没法在低于线性的时间复杂度内恢复其的影响，然而我们可以考虑容忍一部分时间内，并查集的维护的内容是过时的信息，由于容忍错误的并查集信息并不致命，因为并查集只会影响到我们对 seize_place 是否进入循环的判断。所以我们可以每次删除只将桶的位置清空，而并不更新并查集信息，积累到一定数量的删除操作之后，再统一一起重构并查集，这样能到达较优秀的均摊时间复杂度 O(k)删除，其中 k 为常数。

3 代码实现

3.1 文件介绍

在 Cuckoo Hash 文件夹当中，cuckoo2array.hpp 实现了 2-array-cuckoo hashing，multiSlotCuckoo.hpp 实现了 multi-slot Cuckoo hashing。而 test1.cpp 和 test2.cpp 分别是对其的性能测试。

3.2 结构介绍

由于两者的内容比较接近，这里我只介绍 multi-slot-Cuckoo hashing。

以下是 multi_slot_cuckoo 的模板类结构。

```
template<typename K, typename V>
class multi_slot_cuckoo {
    private:
        int32_t capacity;
        int32_t max_depth;
        int32_t slot_num;
        uint32_t Size;
        uint32_t seed;
        uint32_t seize_times = 0, total_depth = 0; // for test
        std::default_random_engine generator; // 随机数生成器
        std::uniform_int_distribution<int32_t> distribution01; // 分布对象
        std::vector<std::vector<std::pair<K, V>>> hashTable1;
        std::vector<uint8_t> bitset1;
        std::vector<std::vector<std::pair<K, V>>> hashTable2;
        std::vector<uint8_t> bitset2;
        int32_t my_hash1(const K &key) const;
        int32_t my_hash2(const K &key) const;
        int32_t find_slot(uint8_t status) const;
        int32_t seize_place(int8_t tableIndex, int32_t index, int32_t
depth);
    public:
        multi_slot_cuckoo(int32_t capacity, int32_t max_depth = 3, int32_t
slot_num = 4, uint32_t seedValue = 0);
        bool insert(const K &key, const V &data);
        void remove(const K &key);
        bool find(const K &key) const;
        V& operator[](const K &key);
        inline uint32_t size() const;
```

```
        inline uint32_t get_seed() const;
        void printInfo() const;
};
```

3.3 核心代码解析

这里是我在类中使用的两个 hash 函数，最开始的时候我使用的 hash1 和现在的 hash1 是相同的，但是 hash2 采用的是 hash1 乘以 key 然后再取模，然而我在实践中发现，这样得到的 hash2 大概率和 hash1 是相关的，即当两个 key 的 hash1 碰撞时，他们的 hash2 也大概率碰撞了，这会极大的增加碰撞概率以及降低空间利用率，因此我将 hash2 改为了如下的形式，避免了于 hash1 之间相互不独立。

```
        int32_t my_hash1(const K &key) const {
            return std::hash<K>{}(key) % capacity;
        }

        int32_t my_hash2(const K &key) const {
            int32_t h = std::hash<K>{}(key);
            return (h ^ (0x9e3779b9 + (1ll * h << 6) + (h >> 2))) % capacity; // 为了
避免 hash1 和 hash2 不独立
        }
```

下面的函数就是我们最关键的函数 seize_place，在 multi-slot 的的该过程中中，由于有多个 slot，因此备用位置也有多个 slot，因此当该节点为满的情况时，每次我们需要查询 slot_num 个递归分支，这是指数复杂度的。所以我们将 max_depth 递归深度限制在了 depth=3，牺牲了一部分可行的插入来避免过高的时间复杂度。

```
/*
@param tableIndex 目前所在的表
@param index 目前需要移到备用位置的对象的 index
@param depth 已经找了几层了
@return 在第几层找到的（移动了几个原有的）
*/
int32_t seize_place(int8_t tableIndex, int32_t index, int32_t depth) {

    if (depth > max_depth) {
        return 0;
    }
    const uint8_t full = (1 << slot_num) - 1;
    auto my_hash = (tableIndex ^ 1) ? &multi_slot_cuckoo::my_hash2 :
&multi_slot_cuckoo::my_hash1;
    std::vector<uint8_t>& bitset = tableIndex ? bitset2 : bitset1;
```

```
    std::vector<uint8_t>& nextBitset = (tableIndex ^ 1) ? bitset2 : bitset1;
    std::vector<std::vector<std::pair<K, V>>>& hashTable = tableIndex ? hashTable2 :
hashTable1;
    std::vector<std::vector<std::pair<K, V>>>& nextHashTable = (tableIndex ^ 1) ?
hashTable2 : hashTable1;

    for(int32_t i = 0;i < slot_num;++ i) {
        K& now_key = hashTable[index][i].first;
        // printf("%d %d %d %d %d %d\n", tableIndex, index, depth, now_key,
my_hash1(now_key), my_hash2(now_key));
        int32_t nextIndex = (this->*my_hash)(now_key);
        int32_t ret;
        if (nextBitset[nextIndex] != full) {
            // 这个的下一个有空
            ret = depth;
        }
        else if (!(ret = seize_place(tableIndex ^ 1, nextIndex, depth + 1))) {
            // 这个的下一个没找到
            continue;
        }
        int32_t free_slot = find_slot(nextBitset[nextIndex]);
        nextBitset[nextIndex] |= (1 << free_slot);
        nextHashTable[nextIndex][free_slot] = hashTable[index][i];
        bitset[index] -= (1 << i);
        return ret;
    }

    return 0;
}
```

这一部分是我实现的插入部分的逻辑，首先查看两个 hash 映射到的位置是否还有空闲位置，如果两个存储桶的位置都有空闲，那么我们随机选择一个，如果两个存储桶只有一个位置有空闲，那么我们选择有空闲的那个插入。

如果两个存储桶都没有空闲位置，那么我们先随机选一个桶，尝试进行 seize_place 是否能够找到可行插入方案，如果不行，我们再尝试使用另一个，如果其中有一个可行，那么正常插入，否则报错插入失败。（因为我们要在第四节测试占有率，因此这里注释掉了插入失败时扩容）

```
/*
@param key 需要插入的数据的索引
@param data 需要存储的数据
@return 是否插入成功 1 成功，0 失败
*/
```

```
bool insert(const K &key, const V &data) {
    int32_t index1 = my_hash1(key);
    int32_t index2 = my_hash2(key);
    const uint8_t full = (1 << slot_num) - 1;
    bool free1 = (bitset1[index1] != full);
    bool free2 = (bitset2[index2] != full);
    bool random_num = distribution01(generator);
    if (free1 && !free2) {
        random_num = 0;
    }
    if (free2 && !free1) {
        random_num = 1;
    }
    int32_t index = random_num ? index2 : index1;
    std::vector<uint8_t>& bitset = random_num ? bitset2 : bitset1;
    std::vector<std::vector<std::pair<K, V>>>& hashTable = random_num ? hashTable2 :
hashTable1;

    if (!free1 && !free2) {
        int32_t depth = 0;
        seize_times ++;
        if (!(depth = seize_place(random_num, index, 1))) {
            total_depth += max_depth; // 失败肯定搜满了

            random_num ^= 1; // 随机的插入遇到死循环了，试试另一边
            int32_t index = random_num ? index2 : index1;
            std::vector<uint8_t>& bitset = random_num ? bitset2 : bitset1;
            std::vector<std::vector<std::pair<K, V>>>& hashTable = random_num ?
hashTable2 : hashTable1;
            if (!(depth = seize_place(random_num, index, 1))) {
                total_depth += max_depth;
                // 都不行，报错
                // expand_capacity();
                #ifdef __CUCKOO2_DEBUG__
                    printf("Failed to seize place.\n");
                #endif
                return false;
            }
            #ifdef __CUCKOO2_DEBUG__
                printf("Seized place for %d cycles.\n", depth);
            #endif
            total_depth += depth;
            int32_t slot_index = find_slot(bitset[index]);
            bitset[index] |= (1 << slot_index);
```

```
            hashTable[index][slot_index] = std::pair<K, V>(key, data);
            Size ++;
            return true; // 因为上面改了引用所以要单独处理
        }
        total_depth += depth;
        #ifdef __CUCKOO2_DEBUG__
            printf("Seized place for %d cycles.\n", depth);
        #endif
    }
    int32_t slot_index = find_slot(bitset[index]);
    bitset[index] |= (1 << slot_index);
    hashTable[index][slot_index] = std::pair<K, V>(key, data);
    Size ++;
    return true;
}
```

最后我们再介绍以下 remove 函数，其逻辑和 find 以及重载的[]运算符相似，因此我们这里只介绍这一个，其它都大同小异。

首先根据 key 获得其在第一个存储桶中对应的位置，如果这个位置中某个 slot 槽位中的 key 和当前 key 对应的上，并且有效位为 1，那么说明找到了目标数据，将其有效位设置为 0，同时 hash 表元素-1 返回即可。如果存储桶 1 中没有找到，再到第二个存储桶中去找。

```
/*
@param key 需要移除的数据的索引
*/
void remove(const K &key) {
    int32_t index1 = my_hash1(key);
    for(int32_t i = 0;i < slot_num;++ i) {
        if (bitset1[index1] & (1 << i) && hashTable1[index1][i].first == key) {
            bitset1[index1] - (1 << i);
            Size --;
            return;
        }
    }
    int32_t index2 = my_hash2(key);
    for(int32_t i = 0;i < slot_num;++ i) {
        if (bitset2[index2] & (1 << i) && hashTable2[index2][i].first == key) {
            bitset2[index2] - (1 << i);
            Size --;
            return;
        }
    }
```

```
}
```

4 性能测试

4.1 测试方法

进入到\Cuckoo hash 文件夹当中，在终端中输入：

```
1. g++ .\test1.cpp -o test1 -D__CUCKOO2_DEBUG__
2. .\test1.exe
```

来进入对于 two-array-Cuckoo hashing 的测试，如果想要关闭调试信息，只需要去掉-D__CUCKOO2_DEBUG__即可。

同理在终端中输入：

```
1. g++ .\test2.cpp -o test2 -D__CUCKOO2_DEBUG__
2. .\test2.exe
```

来进入对于 multi-slot-cuckoo hashing 的测试，如果想要关闭调试信息，只需要去掉-D__CUCKOO2_DEBUG__即可。

4.2 性能分析

4.2.1 Test1 结果分析

我们保持总容量相同为 200w 的前提下，改变 max_depth 的大小，尝试插入容量个数个（200w）的随机元素，比较不同情况下最终的容量占用率，总的插入时间以及平均每次 seize_place 的搜索深度。

Max_depth	占用率	总耗时(ms)	平均搜索层数
0	76.16160%	263356	0.000000
1	81.96735%	348460	1.798929
2	83.32195%	398750	3.206369
3	83.69295%	472640	4.533883
4	83.79190%	534594	5.831605
8	83.83220%	737671	10.892449
16	83.83260%	940760	20.801481

可以发现，当 Max_depth=1 时，相较于 Max_depth=0 时，占用率有显著的提升，耗时增加也较少，之后 Max_depth 增加的较为缓慢，并且在 Max_depth=3

时，占用率就基本接近了最大值，之后再提升 Max_depth 的大小带来的占用率提升并不明显，反而会带来大量的性能消耗，因此在这一测试当中，在百万数量级下 Max_depth 最佳设置值为 2~3。

4.2.2 Test2 结果分析

我们保持总的容量相同（为 20w），并且 Max_depth=3 的前提下，改变每个索引对应的槽位个数，尝试插入容量个数个（20w）随机元素，比较不同情况下的最终的容量占用率，总的插入时间以及平均每次 seize_place 搜索的深度。结果如表所示：

Slot 个数	占用率	总耗时(ms)	碰撞次数	平均搜索层数
1	83.6805%	23077	57366	4.532859
2	92.984%	50513	43667	3.625323
4	98.157%	55507	31453	2.899119
8	99.756%	47680	21992	2.621499

可以发现，slot 为 1 的耗时最短，但是占用率最低并且远低于多槽位的 3 种情况，slot 为 2 时，耗时增加到了 50000ms，而占用率提高到了 93%，slot=4 和 slot=8 的情况下占用率都接近 100%，然而由于 slot=8 的碰撞次数和平均搜索层数都小于 slot=4,因而消耗的时间 slot=8 的情况也显著的小于 slot=4 的情况，甚至低于 slot=2 的情况。因此在此数量级下，最优的 slot 应该设置为 8，如果对占用率要求不高也可以设置为 1。

5 总结

通过学习 Cuckoo hashing 相关方面的知识概念，以及自己对该问题的思考和实践，我对 hash 的各种算法都有了更深刻的认识，这实际上也是我第一次自己实现 hash 表的各种功能，并且实现了两个测试程序，对实现的 2-array-cuckoo hash table 和 multi-slot cuckoo hash table 进行了性能测试，分析了不同参数下其各种性能指标的变化，得出了在测试数量级下较优的参数设置。

我也通过对问题的抽象化，独立设计出了一种正确，可行的算法来解决 Cuckoo dirven way 中死循环的预知问题。虽然这个算法还存在许多不足，例如插入虽然能够以接近常数的代价 O(\alpha(m,n))最差 O(\log(m))(其中 alpha 为反阿克曼函数其值在计算机领域中可以认为<4)，但是删除复杂度却因此退化变成了均摊 O(k)。希望之后能继续改进。