Computer Organization Final Project Report

Part 2 – FM Index

0710764 黃聖偉 0710796 吳俊樺

1. Profile FM-index with gem5, and identify its bottleneck.

	system.cpu.workload.profile_totaltime search 總執行時間	profile_totMemAccLat_wo_overlap Memory time	Compute time	Compute time ÷ memory time
Number of ticks Original	26128958000	10603997854	15524960146	1.464
Number of ticks Modified	24788293000	10210842992	14577450008	1.428

	Final tick	system.cpu.dcache.overall_miss_rate::total
Original	27353855500	0.560658
Modified	26026856500	0.558154

- system.cpu.workload.profile_totaltime 是 total function time。
- profile_totMemAccLat_wo_overlap 是 memory access time。
- compute time = total function time - memory access time。
- 由上表可得，模擬的結果不論是原始的程式或是優化後的程式，
  其 compute time 皆大於 memory access time，compute time/memory time 約等於 1.4，
  可得知 FM index 中的 bottleneck 為 compute time。

2. the difference between suffix array and FM-index

- Suffix array:

  實作方法先將 Text 中的 starting positions 根據 lexicographical order 排列後，再透過字串比較大小來更改 L、mid、R 的值，並且逐步決定好 L 與 R 的值之後，L 與 R 之間的所有 Text 即為符合目標子字串的 starting positions。

  此方法所需要的 space 為不同 starting position 的字母至$(end bit)的總和，假如長度為 n 的 Text，其空間需要 O(n^2)，雖然其空間需求較大，不過演算法的實作較為單純，也較容易聯想出來。

- FM-index:

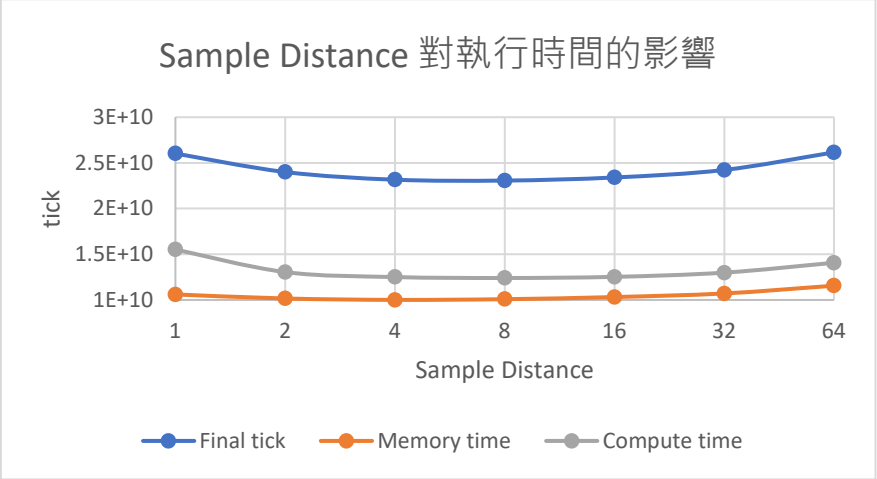
  實作方法與 suffix array 不同的地方是將 suffix 進行 rotation，並且只保留 First column 以及 Last column。其查找的方式為從目標子字串的最後一個字母開始尋找，並且從其對應的 Last column 中判斷是否有其前一個字母，並且找到其對應 Occurrence(rank)的 prefix 字母，持續 iteration 至找到目標子字串的 starting position。

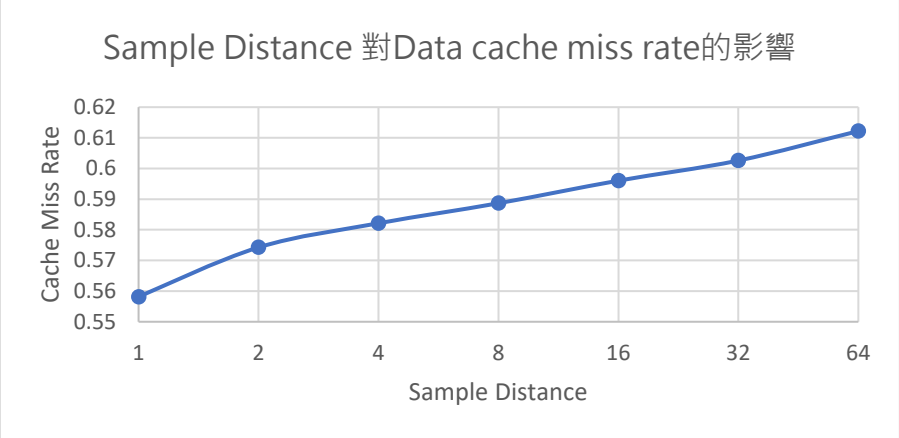
  此實作上還需要有一個陣列來存放各字母的 occurrence，以查找不同 position 所出現的字母，不過此陣列可以透過不同的 sampling distance 來節省空間上的使用。

  此方法所需要的 space 相較於 Suffix array，因為不需要儲存中間 Text 的資料，只剩下 First column、Last column 及 occurrence array，即 O(n)，大量省下空間的利用，不過其實作過程較為複雜，需要克服不同的障礙：找尋前一個子母速度過慢、occurrence 占用空間、需要只透過 F、L 來找出目標子字串等。不過這些問題在 FM-index 的教學影片中，都可以透過優化演算法、減少使用空間等方式來讓問題可以適當地被解決。

3. Sample Distance 對程式執行的影響

Sample Distance	Final tick	Data cache miss rate	Memory time	Compute time	Compute time ÷ memory time
1	26026856500	0.558154	10603997854	15524960146	1.464
2	23998886000	0.574259	10159460601	13049120399	1.284
4	23158932500	0.582123	10007856862	12504630138	1.249
8	23063768500	0.588664	10083252177	12404062823	1.230
16	23403084500	0.596026	10318034473	12528420527	1.214
32	24226602500	0.602589	10710899240	12989271760	1.213
64	26149700500	0.612192	11559181002	14067730998	1.217





- final tick 跟 Compute time 在 sample distance = 8 時降到最低，之後又上升回去。
- Memory time 在 sample distance = 4 時降到最低，之後又上升回去。
- Data cache miss rate 則是隨 sample distance 持續上升，我們認為原因是增加 sample distance 造成程式的 locality 降低，使 data cache miss rate 增加。
- Compute time / memory time ratio 持續緩慢下降。
- 增加 Sample distance 的初期可以使計算量減少，讓 compute 跟 memory time 都降低，程式效能提升。但是如果 sample distance 太大，如這次實驗到 8 的時候，會使程式太容易略過想搜尋的值而需要重新找，加上 miss rate 的增加可能使 miss penalty 變多到超過增加 sample distance 的好處，因此造成 compute 跟 memory time 都增加，而使程式效能降低。

4. 如何改善程式

由於 FM 演算法看起來沒有什麼空間，因此我只在程式細節上做修改。

Original	Modified

```
uint32_t encode_to_int(std::string string_to_encode)
{
    int encoded_value;
    switch (string_to_encode[0])
    {
    case 'A':
        encoded_value = 0;
        break;
    case 'C':
        encoded_value = 1;
        break;
    case 'G':
        encoded_value = 2;
        break;
    case 'T':
        encoded_value = 3;
        break;
    }

    return encoded_value;
}
```

```
uint32_t encode_to_int(const char &string_to_encode)
{
    int encoded_value;
    switch (string_to_encode)
    {
    case 'A':
        encoded_value = 0;
        break;
    case 'C':
        encoded_value = 1;
        break;
    case 'G':
        encoded_value = 2;
        break;
    case 'T':
        encoded_value = 3;
        break;
    }

    return encoded_value;
}
```

encode_to_int 只需要字串第一個字元的資料，因此我將它的參數改成字元，也將傳值改成傳 reference 來加速。

```
void build_tables(const std::string &reference_string, std::vector<uint32_t> &F_offsets, std::vector<char> &L_column, std::ve
{
    // rotate and sort

    std::vector<std::string> rotate_and_sort_strings;

    std::string current_rotation = reference_string;
    for (int i = 0; i < reference_string.length(); ++i)


    {
        current_rotation.push_back(current_rotation[0]);
        current_rotation.erase(current_rotation.begin());
        rotate_and_sort_strings.push_back(current_rotation);
    }
    std::sort(rotate_and_sort_strings.begin(), rotate_and_sort_strings.end());



    //create tables

    std::vector<uint32_t> running_occ(4, 0);


    for (int i = 0; i < rotate_and_sort_strings.size(); ++i)
    {
        // create F_offsets
        std::string F_string = rotate_and_sort_strings[i].substr(0, 1);
        uint32_t encoded_F = encode_to_int(F_string);
        if (F_offsets[encoded_F] == reference_string.size())
        {
            F_offsets[encoded_F] = i;
        }

        // create L_column
        L_column.push_back(rotate_and_sort_strings[i][rotate_and_sort_strings[i].size() - 1]);


        // create occ_table
        uint32_t encoded_L = encode_to_int(rotate_and_sort_strings[i].substr(rotate_and_sort_strings[i].size() - 1));
        ++running_occ[encoded_L];
        if (i % OCC_SAMPLING_DIST == 0)
        {
            occ_table.push_back(running_occ);
        }
    }

}
```

```
void build_tables(const std::string &reference_string, std::vector<uint32_t> &F_offsets, std::vector<char> &L_column, std:
{
    // rotate and sort

    std::vector<std::string> rotate_and_sort_strings;

    std::string current_rotation = reference_string;
    int reference_string_length = reference_string.length();

    for (int i = 0; i < reference_string_length; ++i)
    {
        current_rotation.push_back(current_rotation[0]);
        current_rotation.erase(current_rotation.begin());
        rotate_and_sort_strings.push_back(current_rotation);
    }
    std::sort(rotate_and_sort_strings.begin(), rotate_and_sort_strings.end());


    //create tables

    std::vector<uint32_t> running_occ(4, 0);

    for (int i = 0; i < rotate_and_sort_strings.size(); ++i)
    {
        // create F_offsets
        uint32_t encoded_F = encode_to_int(rotate_and_sort_strings[i][0]);
        if (F_offsets[encoded_F] == reference_string_length)


        {
            F_offsets[encoded_F] = i;
        }

        // create L_column
        char c = rotate_and_sort_strings[i][reference_string_length - 1];
        L_column.push_back(c);

        // create occ_table
        uint32_t encoded_L = encode_to_int(c);
        ++running_occ[encoded_L];
        if (i % OCC_SAMPLING_DIST == 0)
        {
            occ_table.push_back(running_occ);
        }
    }

}
```

- reference_string.length() 需要使用多次，因此我們把它存成變數 reference_string_length 來減少 reference_string 的 access 次數。
- 把 str.substr(n, 1)換成速度較快的 str[n]。
- rotate_and_sort_strings[i][reference_string_length - 1] 需要使用多次，因此我們把它存成變數 c 來加速，也能減少記憶體 access 次數來降低 miss rate。

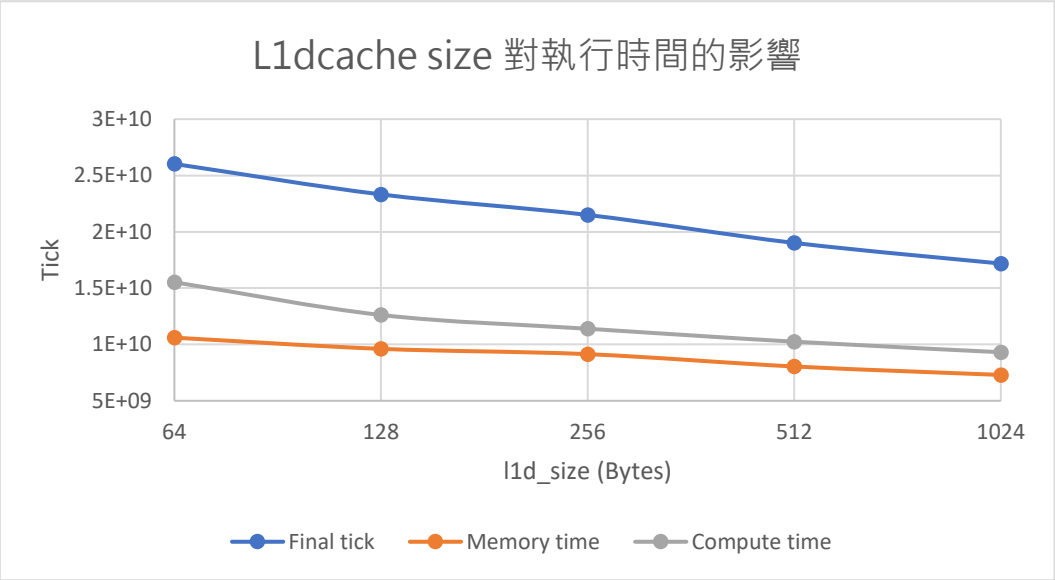
結果看起來 cache miss rate 從 0.560658 降至 0.558154，Final tick 由 27353855500 降至 26026856500。程式速度有變快，cache miss rate 也有降低，有成功改善程式。

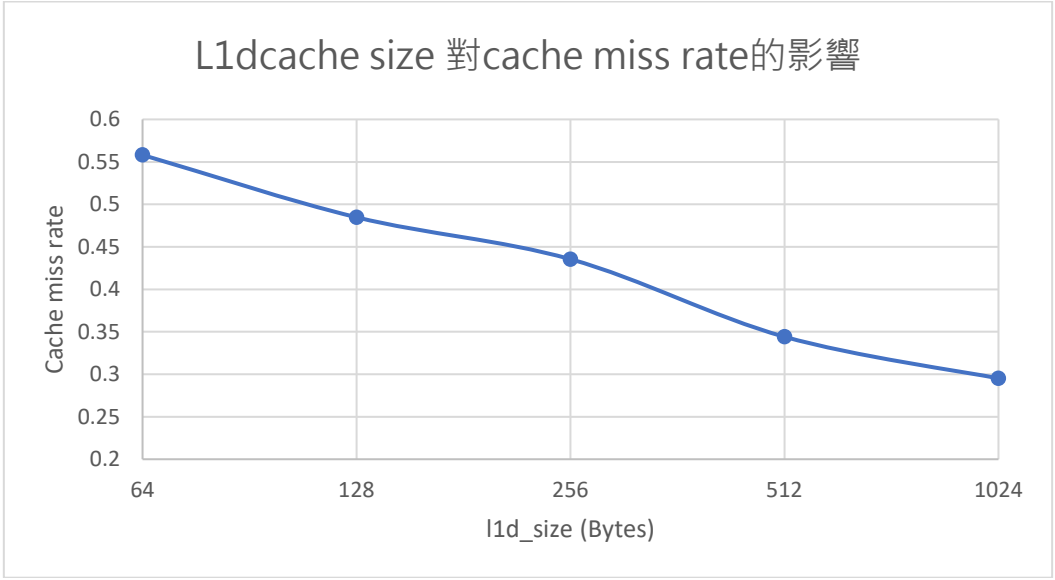
Cache 對程式執行的影響

L1i_size: 16 kB

L1d_assoc: 1

l1d_size	Final tick	Data cache miss rate	Memory time	Compute time
64B	26026856500	0.558154	10603997854	15524960146
128B	23321459500	0.484803	9610791532	12618404468
256B	21498697500	0.435464	9138790526	11396220474
512B	19015184500	0.344160	8044790089	10247362911
1024B	17185040500	0.295246	7295670613	9310620387



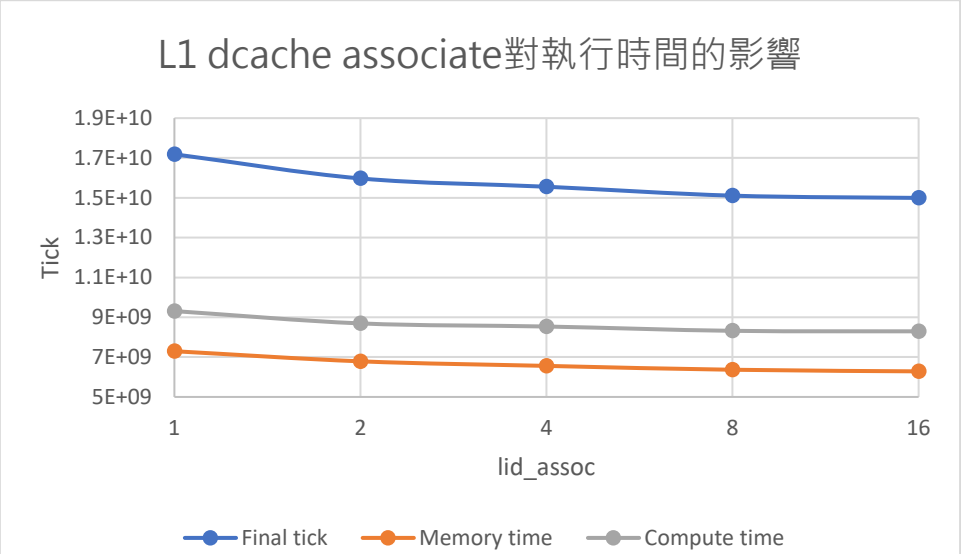


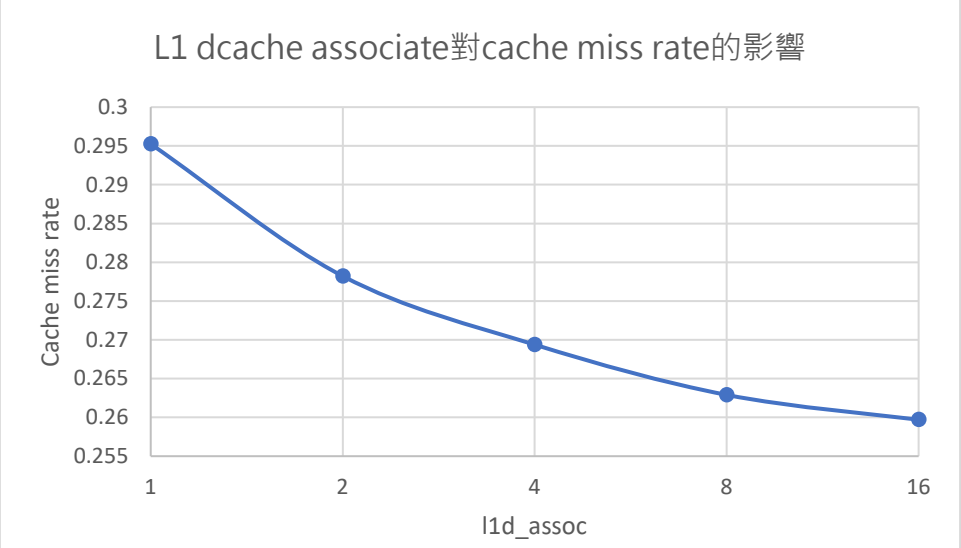
- 維持相同 instruction cache size 跟 Associativity。
- 當 data cache size 越大時，越多資料能被存在 cache 裡，因此 cache miss rate 會降低。
- 隨著 cache miss rate 的降低，memory time 也會降低，compute time 也跟著降低，final tick 變小。
- 在價格合理的狀況下，data cache 越大效能越好。

L1d_size: 1024 B

L1i_size: 16 kB

l1d_assoc	Final tick	Data cache miss rate	Memory time	Compute time
1	17185040500	0.295246	7295670613	9310620387
2	15971355500	0.278218	6787583297	8692307703
4	15555045500	0.269381	6559218648	8535677352
8	15106818500	0.262901	6365001560	8319845440
16	14994238500	0.259705	6284620990	8294753010



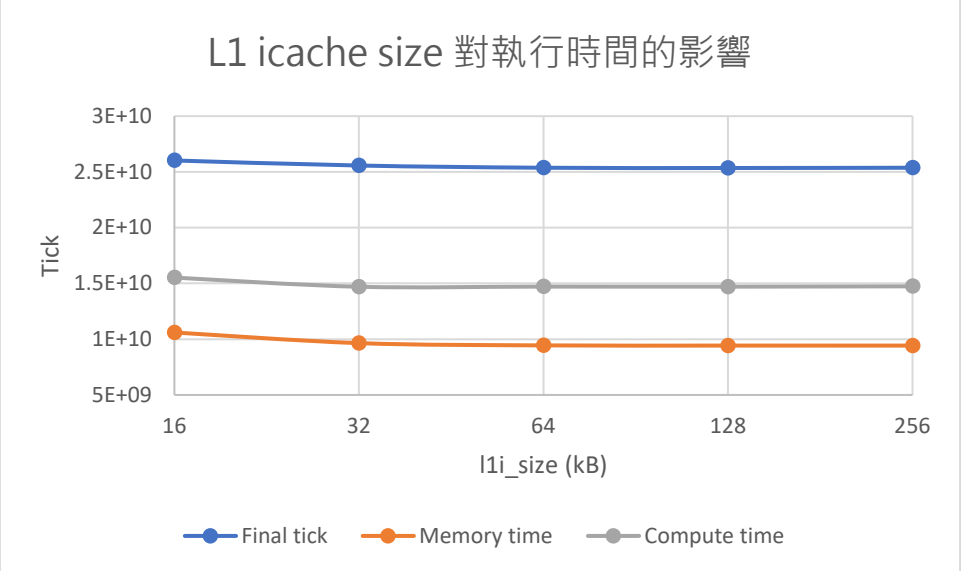


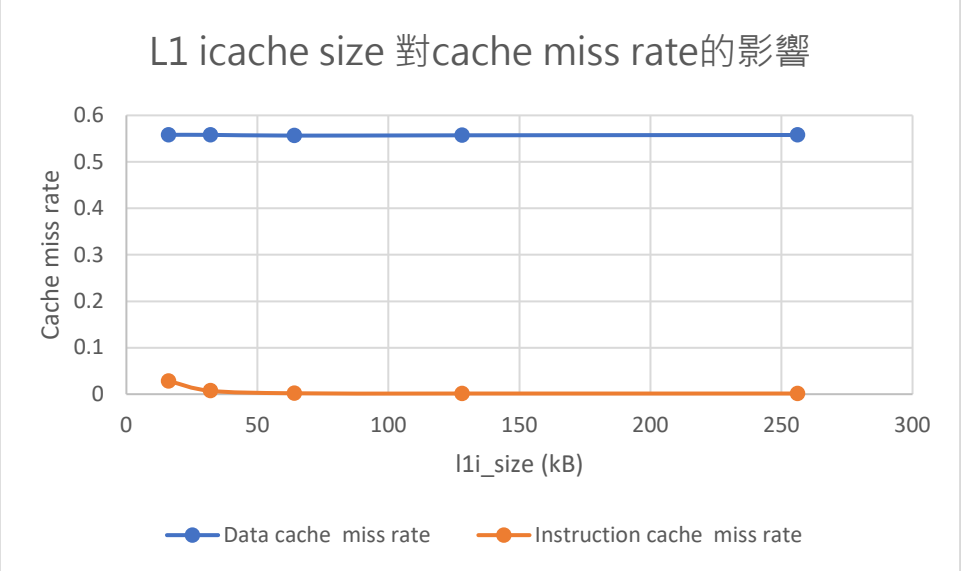
- 維持相同 instruction cache size 跟 data cache size。
- Higher Associativity -> lower miss rate -> lower miss penalty，cache miss rate 會降低。
- 隨著 cache miss rate 的降低，memory time 也會降低，compute time 也跟著降低，final tick 變小。
- Associativity 越高，系統也會越複雜，miss penalty 會提高，因此在 miss penalty 合理的狀況下，也就是 miss penalty 不會造成 AMAT 提升的狀況，Associativity 越大效能越好。

L1d_size: 64 B

L1d_assoc: 1

l1i_size	Final tick	Data cache miss rate	Instruction cache miss rate	Memory time	Compute time
16kB	26026856500	0.558154	0.028462	10603997854	15524960146
32kB	25572255500	0.558048	0.007372	9647562543	14703173457
64kB	25374358500	0.556461	0.001955	9445700437	14716826563
128kB	25351518500	0.557115	0.001427	9429028495	14711410505
256kB	25374716500	0.557775	0.001431	9426643474	14742001526





- 維持相同 data cache size 跟 Associativity
- 當 instruction cache size 越大時，Instruction cache miss rate 有持續下降，維持很低。
- 當 instruction cache size 越大時，Data cache miss rate 維持一致，有小幅波動。
- Memory 跟 compute time 一開始從 16kB 到 32kB 時稍微下降，但之後繼續增加 size 時就沒什麼變化，final tick 甚至還增加。
- 我們推測是一開始 16kB 時整個程式還沒辦法都存到 instruction cache 裡，但到了 32kB 以後整個程式都可以存到 instruction cache 裡了，因此對效能提升就沒有太大的影響。