**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。

1. **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 |
| 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。 | |
| * 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。