軟體分析與最佳化 Workload Analysis Stage 3

組別：2
成員： 612410017 林靖紳、612410066 蔡宏遠

分析的 Workload: Dedup

- 目的: 通過檢測和消除數據中重複的區塊，實現數據壓縮。
- 大致流程:
  - 輸入：將數據分成固定大小的區塊(blocks)，作為基本的處理單元
  - 檢測是否有重複區塊： 使用 hash 比較不同區塊之間的內容，以識別重複區塊
  - 去重處理： 一旦檢測到重複區塊，Dedup僅保留一個副本，然後在需要比對時引用它，從而顯著減少記憶體需求。
  - 輸出： Dedup處理後的數據會被壓縮
    - 被壓縮的區塊是：沒有重複資料的區塊。以減小記憶體需求，提高數據傳輸效率

Review Stage 2 Discussion

- 找到的 hotspot:
  - 使用 gcc -O0 編譯發現 hotspot 為 rabinseg、deflate_slow

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
[Outside any known module]	[Unknown]	6.112s	24.0%
rabinseg	dedup	4.075s	16.0%
deflate_slow	dedup	3.715s	14.6%
pqdownheap	dedup	3.149s	12.3%
sha1_block_data_order	dedup	2.617s	10.3%
[Others]	N/A*	5.850s	22.9%

*N/A is applied to non-summable metrics.

- 初步利用編譯選項做優化：
  - 利用 gcc -O3 編譯發現 hotspot 只剩下 deflate_slow
  - 優化了 rabinseg 函式的執行時間比例，並在 stage 2 對這個部份做了討論

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function	Module	CPU Time	% of CPU Time
deflate_slow	dedup	3.523s	20.9%
pqdownheap	dedup	3.305s	19.6%
[Outside any known module]	[Unknown]	2.660s	15.8%
sha1_block_data_order	dedup	1.878s	11.2%
rabinseg	dedup	1.440s	8.5%
[Others]	N/A*	4.032s	23.9%

*N/A is applied to non-summable metrics.

討論 如何優化 deflate_slow

- deflate_slow 目的：用於實現 dedup 最後輸出時進行壓縮處理
- **認為的熱點：**
  - **迴圈尋找不定次數**
  - **複雜的 if else 判斷**
  - **hash table**
- 下面將分別對以上三點進行優化

優化 for 迴圈

使用 unroll / nounroll

- 函式 deflate_slow()

```
local block_state deflate_slow(s, flush)
    deflate_state *s;
    int flush;
{
    IPos hash_head = NIL;     /* head of hash chain */
    int bflush;               /* set if current block must be flushed

    /* Process the input block. */
    for (;;) {
        /* Make sure that we always have enough lookahead, except …
        if (s->lookahead < MIN_LOOKAHEAD) {
            fill_window(s);
            if (s->lookahead < MIN_LOOKAHEAD && flush == Z_NO_FLUSH)
                return need_more;
```

- 在實驗前，我們先進行了一些討論，認為：
    - 因為 deflate_slow 中的 for 迴圈次數並不固定，因此做 loop unroll/roll 可能不會得到好的結果
    - 但仍然嘗試使用看看

```
#pragma omp parallel for
for (;;) {
    /* Make sure that we always have enough lookahead, except…
    #pragma omp unroll
    if (s->lookahead < MIN_LOOKAHEAD) {
        fill_window(s);
        if (s->lookahead < MIN_LOOKAHEAD && flush == Z_NO_FLUSH) {
            return need_more;
        }
        if (s->lookahead == 0) break; /* flush the current block */
    }
```

- 首先使用 `#pragma omp parallel for` 做嘗試，可以看到效果並不太好，執行時間變慢了許多

```
real    0m5.248s
user    0m14.443s
sys     0m3.814s
```

- 然後我們再加上 `#pragma omp nounroll` 去做嘗試，實驗結果跟原本的相同。

```
real    0m4.925s
user    0m14.634s
sys     0m3.183s
```

- 最後使用 `#pragma omp unroll`

```
real    0m5.246s
user    0m14.451s
sys     0m3.749s
```

- 討論實驗結果
    - 實驗結果確實如我們先前的討論
    - 由於 deflate_slow 中的 for loop 每次迭代的模式並不固定，例如迴圈次數可能會有差異
    - 在 deflate_slow 中，由於迴圈次數和模式的變化，循環展開可能導致冗余的程式碼或者無法提供實際的性能改善。
    - 因此在 deflate_slow 中不適合使用 pragma loop unrolling去加速

SIMD

討論是否使用 SIMD

- 使用 omp parallel 效率不佳
  - 如上一段 unroll 的部份所說，由於迴圈的特性，每一輪迴圈進行的行為不盡相同的情況下，使用平行執行的方法並不會讓程式加速
  - 相反，可能還會拖累程式的效能，因此使用 SIMD 指令增加平行度在這裡是不可行的

優化複雜計算

- 討論認為：
  - 可能不需要用到 hash table 這種相對較為複雜的資料結構
  - 可以針對 Run-Length Encoding，簡單地比較相鄰的字節，而不需要複雜的匹配條件。

- Pseudo code

```
deflate_rle:
  while not end_of_input:
    fill_window(s)
    while s.lookahead >= MAX_MATCH:
      run = find_consecutive_repeats(s)
      if run >= MIN_MATCH:
        process_consecutive_repeats(s, run)
      else:
        process_single_literal(s)
      if flushing_needed:
        flush_block(s)

  flush_block(s, Z_FINISH)
  return Z_FINISH ? finish_done : block_done

```

- 壓縮率比較：（兩者相同）
  - 原本：

```
PARSEC Benchmark Suite Version 3.0-beta-20150206
Total input size:                  671.58 MB
Total output size:                 637.28 MB
Effective compression factor:        1.05x

Mean data chunk size:                1.88 KB (stddev: 2023.50 KB)
Amount of duplicate chunks:         54.49%
Data size after deduplication:     658.95 MB (compression factor: 1.02x)
Data size after compression:       630.26 MB (compression factor: 1.05x)
```

- 更改之後

```
Total input size:                    671.58 MB
Total output size:                   637.28 MB
Effective compression factor:          1.05x

Mean data chunk size:                  1.88 KB (stddev: 2023.50 KB)
Amount of duplicate chunks:           54.49%
Data size after deduplication:       658.95 MB (compression factor: 1.02x)
Data size after compression:         630.26 MB (compression factor: 1.05x)
Output overhead:                       1.10%
```

- 執行時間比較：（**更改完之後較原本的執行時間短！**）
  - 原本：

```
real    0m4.919s
user    0m14.588s
sys     0m3.225s
```

  - 更改之後：

```
real    0m4.651s
user    0m14.927s
sys     0m2.958s
```

結論

在對 deflate_slow 演算法進行優化的過程中，我們發現這段程式碼並不適合使用 loop unroll 和 SIMD 這樣的優化手段。相反，通過更換演算法，我們成功地實現了約 **6%** 的執行時間改善。
這次改進的主要原因可以歸結為以下兩點：

- 省略 Hash Table 使用：
  - 我們認識到在某些情況下，Hash Table 可能會引入額外的複雜度，尤其是當資料的模式變化不大的時候
  - 因此，省略 Hash Table 不僅能夠降低記憶體和運算的開銷，還使得程式碼更加簡潔和高效
- Run-Length Encoding 簡化：
  - 透過簡化 Run-Length Encoding 的比較過程，僅比較相鄰字節而不需要複雜的匹配條件，我們成功地減少了計算量，進而提高了效能
  - 這種簡化的方法特別適用於一些數據模式相對簡單的情況，為改進整體執行效能提供了實際的幫助

綜合來看，我們的改進著眼於**減少複雜性、簡化比較過程以及優化資料結構**，最終取得了令人滿意的執行時間改善。