軟體分析與最佳化 Workload Analysis Final

組別：2
成員： 612410017 林靖紳、612410066 蔡宏遠

介紹 workload program

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
- Parsec 對每個基準程序提供了六個輸入測試
  - test（測試）： 用於驗證程序是否可執行的最小輸入。 執行幾乎瞬間完成
  - simdev（模擬開發）： 非常小的輸入，使代碼執行時間與該程序的典型輸入可比。用於微架構模擬器的開發。
  - simsmall（模擬小）： 用於使用微架構模擬器進行性能測量的小型輸入。
  - simmedium（模擬中）： 用於使用微架構模擬器進行性能測量的中等大小輸入。
  - simlarge（模擬大）： 用於使用微架構模擬器進行性能測量的大型輸入。
  - native（本機）： 非常大的輸入，適用於在實際機器上進行大規模實驗。
- 以下實驗皆採用 native 作為輸入測試

Execution environments

- CPU information

```
ashen@Stephanie-Lin:~$ lscpu
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            11th Gen Intel(R) Core(TM) i5-11500 @ 2.70GHz
    CPU family:          6
    Model:               167
    Thread(s) per core:  2
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            1
    CPU max MHz:         4600.0000
    CPU min MHz:         800.0000
    BogoMIPS:            5424.00
```

- Memory

```
ashen@Stephanie-Lin:~$ free -h
               total        used        free      shared  buff/cache   available
Mem:            31Gi       4.2Gi        12Gi       1.8Gi        14Gi        24Gi
Swap:          2.0Gi          0B       2.0Gi
```

- OS version

```
ashen@Stephanie-Lin:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 22.04.2 LTS
Release:        22.04
Codename:       jammy
```

- GCC version

```
ashen@Stephanie-Lin:~$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

編驛與執行

編譯

Dedup build with gcc -O3

```
# Arguments to use
export CFLAGS=" -DUNIX -O3 -funroll-loops -fprefetch-loop-arrays ${PORTABILITY_FLAGS} -pg"
export CXXFLAGS="-DUNIX -O3 -funroll-loops -fprefetch-loop-arrays -fpermissive -fno-exceptions ${PORTABILITY_FLAGS} -pg"
export CPPFLAGS=""
export CXXCPPFLAGS=""
export LDFLAGS="-L${CC_HOME}/lib64 -L${CC_HOME}/lib"
export LIBS=""
export EXTRA_LIBS=""
export PARMACS_MACRO_FILE="pthreads"
```

```
ashen@Stephanie-Lin:~/parsec-benchmark-master$ source env.sh
ashen@Stephanie-Lin:~/parsec-benchmark-master$ parsecmgmt -a build -p parsec.dedup -c gcc
[PARSEC] Packages to build:  parsec.dedup

[PARSEC] [========== Building package parsec.dedup [1] ==========]
[PARSEC] [---------- Analyzing package parsec.dedup ----------]
[PARSEC] Package parsec.dedup already exists, proceeding.
[PARSEC]
[PARSEC] BIBLIOGRAPHY
[PARSEC]
[PARSEC] [1] Bienia. Benchmarking Modern Multiprocessors. Ph.D. Thesis, 2011.
[PARSEC]
[PARSEC] Done.
```

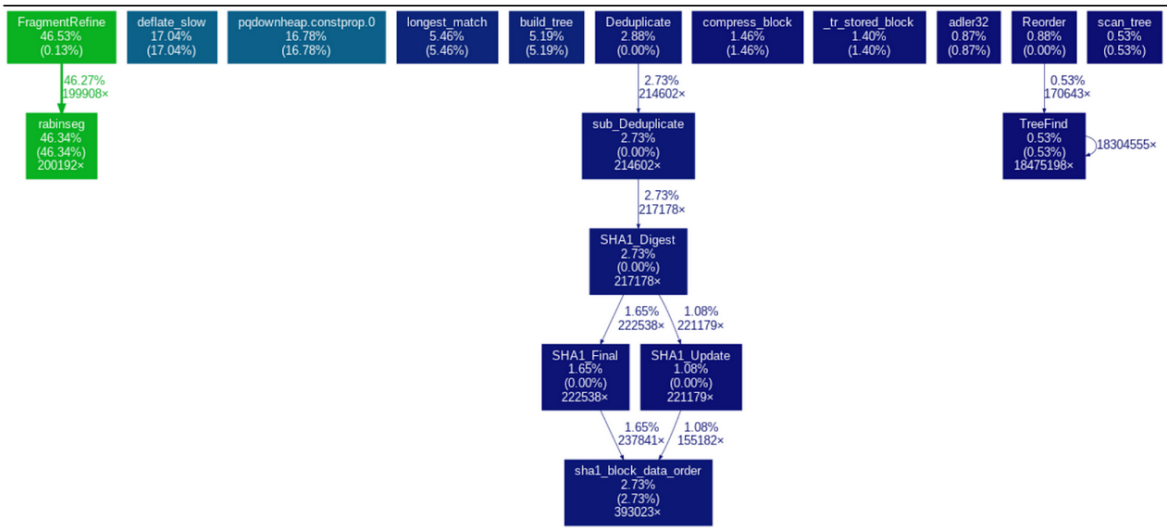
執行

```
ashen@Stephanie-Lin:~/parsec-benchmark-master$ parsecmgmt -a run -p parsec.dedup -c gcc -i native -n 4
[PARSEC] Benchmarks to run:  parsec.dedup

[PARSEC] [========== Running benchmark parsec.dedup [1] ==========]
[PARSEC] deleting old run directory.
[PARSEC] Setting up run directory.
[PARSEC] Unpacking benchmark input 'native'.
FC-6-x86_64-disc1.iso
[PARSEC] Running '  /home/ashen/parsec-benchmark-master/pkgs/kernels/dedup/inst/amd64-linux.gcc/bin/dedup
[PARSEC] [---------- Beginning of output ----------]
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)
Output overhead:                     1.10%
[PARSEC] [----------    End of output    ----------]
[PARSEC]
[PARSEC] BIBLIOGRAPHY
[PARSEC]
[PARSEC] [1] Bienia. Benchmarking Modern Multiprocessors. Ph.D. Thesis, 2011.
[PARSEC]
[PARSEC] Done.
```

效能分析

gprof 分析

- 可以看到 hotspot 是 FragmentRefine 的 function，這個 function 主要是在進行 Rabin Finger printing 演算法
- rabinseg 演算法會根據數據像是字串之類的，根據一些規則去 hash 出一個絕大部分是獨特的指紋



Vtune 分析

- 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.

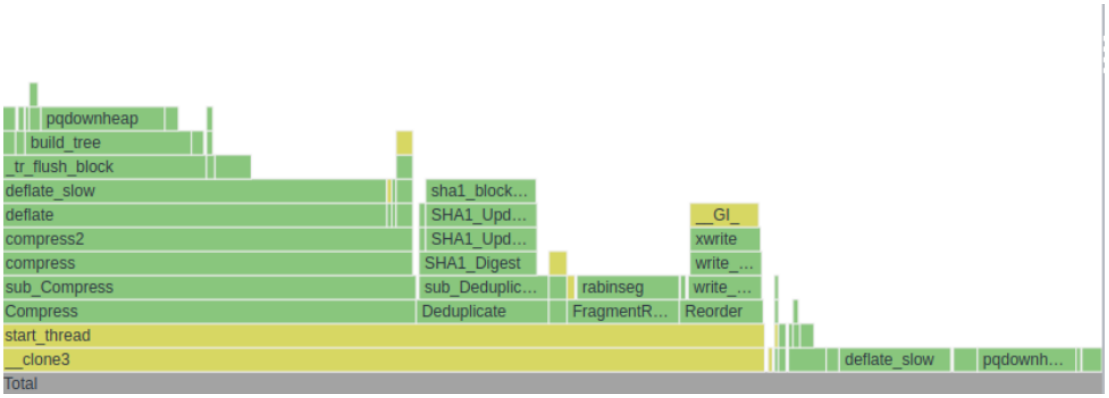
Function	Module	CPU Time	% of CPU Time
[Outside any known module]	[Unknown]	6.112s	24.0%
rabinseg	dedup	4.075s	16.0%
deflate_slow	dedup	3.715s	14.6%
pqdownheap	dedup	3.149s	12.3%
sha1_block_data_order	dedup	2.617s	10.3%
[Others]	N/A*	5.850s	22.9%

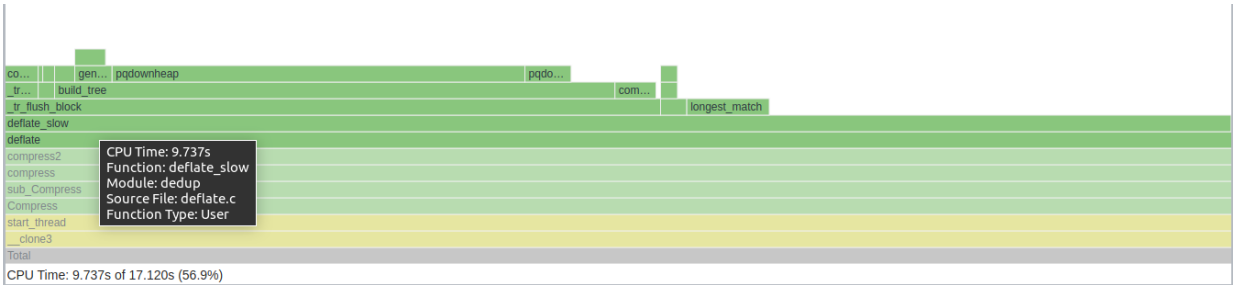
**N/A is applied to non-summable metrics.*

Flame Graph

- 觀察整個程式的執行流程
- 函數呼叫的頻率與深度

    - 深度較深的函數呼叫可能是潛在的性能瓶頸，像是 deflate_slow 函式
- 耗時的操作
    - 矩形區塊的寬度反映了相應函數執行的時間





嘗試修改、優化的方式

初步利用編譯選項做優化：

使用 gcc -O3

- 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

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的討論：

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 */
        }
```

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

優化複雜計算和判斷

- 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

```
for (;;) {
    /* Make sure that we always have enough lookahead, except
     * at the end of the input file. We need MAX_MATCH bytes
     * for the longest encodable run.
     */
    if (s->lookahead < MAX_MATCH) {…
    }

    /* See how many times the previous byte repeats */
    run = 0;
    if (s->strstart > 0) {      /* if there is a previous byte, that is */…
    }

    /* Emit match if have run of MIN_MATCH or longer, else emit literal */
    if (run >= MIN_MATCH) {…
    } else {
        /* No match, output a literal byte */
        Tracevv((stderr,"%c", s->window[s->strstart]));
        _tr_tally_lit (s, s->window[s->strstart], bflush);
        s->lookahead--;
        s->strstart++;
    }
    if (bflush) FLUSH_BLOCK(s, 0);
}
```

評估與驗證結果

gcc -O3 的優化

- 執行時間
  - -O3

```
real    0m4.940s
user    0m14.492s
sys     0m3.518s
```

  - -O0

```
real    0m5.923s
user    0m19.574s
sys     0m5.635s
```

- 成功提昇了大約 17 % 的執行時間！(原因討論位於 [結果與討論])

- **以下實驗基於 gcc -O3 進行比較**

loop unroll

- 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` 也並無改善

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

替換演算法

- 執行時間比較，成功提昇了約 6 % 的執行時間 (原因討論位於 [結果與討論])
  - 原本

```
real    0m4.940s
user    0m14.492s
sys     0m3.518s
```

  - 替換後

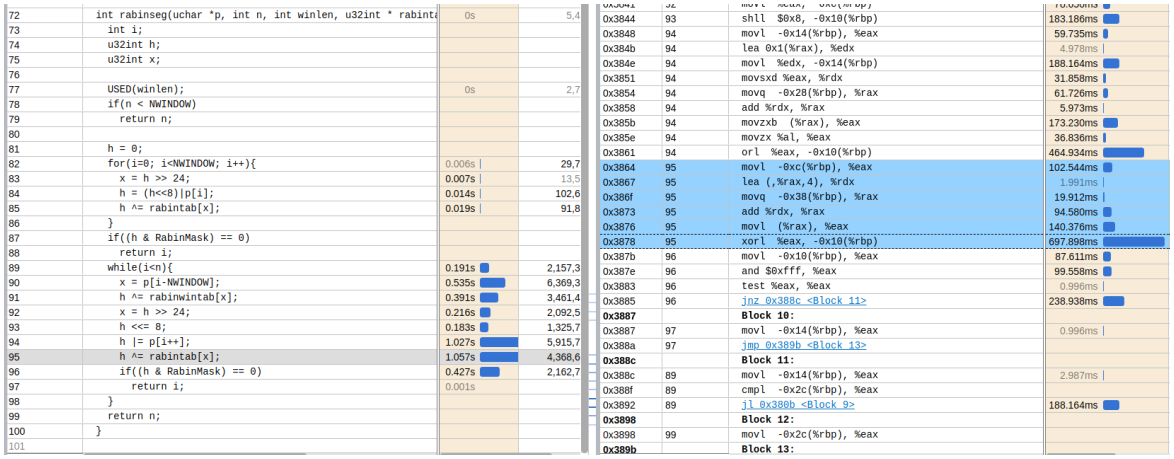
```
real    0m4.651s
user    0m14.927s
sys     0m2.958s
```

討論

討論 -O3 如何優化 rabinseg 函式

反組譯

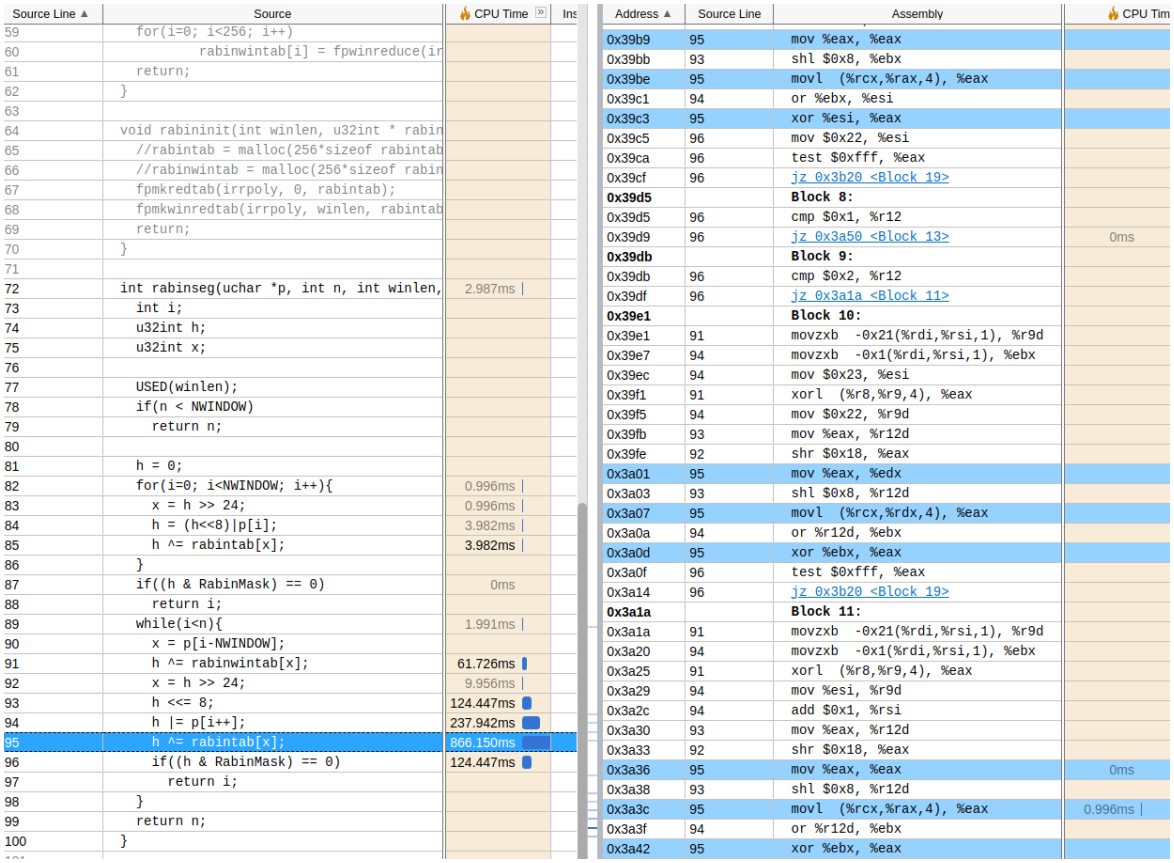
- -O0



```
72  int rabinseg(uchar *p, int n, int winlen, u32int * rabinta   0s      5.4
73    int i;
74    u32int h;
75    u32int x;
76
77    USED(winlen);                                               0s      2.7
78    if(n < NWINDOW)
79      return n;
80
81    h = 0;
82    for(i=0; i<NWINDOW; i++){                                   0.006s  29,7
83      x = h >> 24;                                              0.007s  13.5
84      h = (h<<8)|p[i];                                          0.014s  102,6
85      h ^= rabintab[x];                                         0.019s  91,8
86    }
87    if((h & RabinMask) == 0)
88      return i;
89    while(i<n){                                                 0.191s  2,157,3
90      x = p[i-NWINDOW];                                         0.535s  6,369,3
91      h ^= rabinwintab[x];                                      0.391s  3,461,4
92      x = h >> 24;                                              0.216s  2,092,5
93      h <<= 8;                                                  0.183s  1,325,7
94      h |= p[i++];                                              1.027s  5,915,7
95      h ^= rabintab[x];                                         1.057s  4,368,6
96      if((h & RabinMask) == 0)                                  0.427s  2,162,7
97        return i;                                               0.001s
98    }
99    return n;
100 }
101
```

```
0x3844  93  shll  $0x8, -0x10(%rbp)                 183.186ms
0x3848  94  movl  -0x14(%rbp), %eax                  59.735ms
0x384b  94  lea 0x1(%rax), %edx                      4.978ms
0x384e  94  movl  %edx, -0x14(%rbp)                  188.164ms
0x3851  94  movsxd %eax, %rdx                        31.858ms
0x3854  94  movq  -0x28(%rbp), %rax                  61.726ms
0x3858  94  add %rdx, %rax                           5.973ms
0x385b  94  movzxb  (%rax), %eax                     173.230ms
0x385e  94  movzx %al, %eax                          36.836ms
0x3861  94  orl  %eax, -0x10(%rbp)                   464.934ms
0x3864  95  movl  -0xc(%rbp), %eax                   102.544ms
0x3867  95  lea (,%rax,4), %rdx                      1.991ms
0x386f  95  movq  -0x38(%rbp), %rax                  19.912ms
0x3873  95  add %rdx, %rax                           94.580ms
0x3876  95  movl  (%rax), %eax                       140.376ms
0x3878  95  xorl  %eax, -0x10(%rbp)                  697.898ms
0x387b  96  movl  -0x10(%rbp), %eax                  87.611ms
0x387e  96  and $0xfff, %eax                         99.558ms
0x3883  96  test %eax, %eax                          0.996ms
0x3885  96  jnz 0x388c <Block 11>                    238.938ms
0x3887      Block 10:
0x3887  97  movl  -0x14(%rbp), %eax                  0.996ms
0x388a  97  jmp 0x389b <Block 13>
0x388c      Block 11:
0x388c  89  movl  -0x14(%rbp), %eax                  2.987ms
0x388f  89  cmpl  -0x2c(%rbp), %eax
0x3892  89  jl 0x380b <Block 9>                      188.164ms
0x3898      Block 12:
0x3898  99  movl  -0x2c(%rbp), %eax
0x389b      Block 13:
```

- -O3



```
Source Line | Source                                                  | CPU Time
59          |   for(i=0; i<256; i++)
60          |     rabinwintab[i] = fpwinreduce(ir
61          |   return;
62          | }
63          |
64          | void rabininit(int winlen, u32int * rabin
65          |   //rabintab = malloc(256*sizeof rabintab
66          |   //rabinwintab = malloc(256*sizeof rabin
67          |   fpmkredtab(irrpoly, 0, rabintab);
68          |   fpmkwinredtab(irrpoly, winlen, rabintab
69          |   return;
70          | }
71          |
72          | int rabinseg(uchar *p, int n, int winlen,   2.987ms
73          |   int i;
74          |   u32int h;
75          |   u32int x;
76          |
77          |   USED(winlen);
78          |   if(n < NWINDOW)
79          |     return n;
80          |
81          |   h = 0;
82          |   for(i=0; i<NWINDOW; i++){                 0.996ms
83          |     x = h >> 24;                            0.996ms
84          |     h = (h<<8)|p[i];                        3.982ms
85          |     h ^= rabintab[x];                       3.982ms
86          |   }
87          |   if((h & RabinMask) == 0)                  0ms
88          |     return i;
89          |   while(i<n){                               1.991ms
90          |     x = p[i-NWINDOW];
91          |     h ^= rabinwintab[x];                    61.726ms
92          |     x = h >> 24;                            9.956ms
93          |     h <<= 8;                                124.447ms
94          |     h |= p[i++];                            237.942ms
95          |     h ^= rabintab[x];                       866.150ms
96          |     if((h & RabinMask) == 0)                124.447ms
97          |       return i;
98          |   }
99          |   return n;
100         | }
```

```
Address | Source Line | Assembly                                  | CPU Tim
0x39b9  | 95          | mov %eax, %eax
0x39bb  | 93          | shl $0x8, %ebx
0x39be  | 95          | movl  (%rcx,%rax,4), %eax
0x39c1  | 94          | or %ebx, %esi
0x39c3  | 95          | xor %esi, %eax
0x39c5  | 96          | mov $0x22, %esi
0x39ca  | 96          | test $0xfff, %eax
0x39cf  | 96          | jz 0x3b20 <Block 19>
0x39d5  |             | Block 8:
0x39d5  | 96          | cmp $0x1, %r12
0x39d9  | 96          | jz 0x3a50 <Block 13>                      0ms
0x39db  |             | Block 9:
0x39db  | 96          | cmp $0x2, %r12
0x39df  | 96          | jz 0x3a1a <Block 11>
0x39e1  |             | Block 10:
0x39e1  | 91          | movzxb  -0x21(%rdi,%rsi,1), %r9d
0x39e7  | 94          | movzxb  -0x1(%rdi,%rsi,1), %ebx
0x39ec  | 94          | mov $0x23, %esi
0x39f1  | 91          | xorl  (%r8,%r9,4), %eax
0x39f5  | 94          | mov $0x22, %r9d
0x39fb  | 93          | mov %eax, %r12d
0x39fe  | 92          | shr $0x18, %eax
0x3a01  | 95          | mov %eax, %edx
0x3a03  | 93          | shl $0x8, %r12d
0x3a07  | 95          | movl  (%rcx,%rdx,4), %eax
0x3a0a  | 94          | or %r12d, %ebx
0x3a0d  | 95          | xor %ebx, %eax
0x3a0f  | 96          | test $0xfff, %eax
0x3a14  | 96          | jz 0x3b20 <Block 19>
0x3a1a  |             | Block 11:
0x3a1a  | 91          | movzxb  -0x21(%rdi,%rsi,1), %r9d
0x3a20  | 94          | movzxb  -0x1(%rdi,%rsi,1), %ebx
0x3a25  | 91          | xorl  (%r8,%r9,4), %eax
0x3a29  | 94          | mov %esi, %r9d
0x3a2c  | 94          | add $0x1, %rsi
0x3a30  | 93          | mov %eax, %r12d
0x3a33  | 92          | shr $0x18, %eax
0x3a36  | 95          | mov %eax, %eax                            0ms
0x3a38  | 93          | shl $0x8, %r12d
0x3a3c  | 95          | movl  (%rcx,%rax,4), %eax                 0.996ms
0x3a3f  | 94          | or %r12d, %ebx
0x3a42  | 95          | xor %ebx, %eax
```

分析

1. 可以看到組語的部份 -O3 將 -O0 一共五行的程式碼分為三部份去做亂序執行
2. 分析原因應該是因為亂序執行允許 CPU 在執行指令時根據可用的執行單元和資源來選擇最優的執行順序，提高程式並行性，從而在同一時鐘周期內執行更多的

指令 （Instruction per cycle）

3. 這個結果可以從上面第三部份的 gcc -O3 vtune 分析圖中看到其 CPI 確實小於 gcc -O0 (Cycle per instruction)

討論使用 loop unroll 無法優化 deflate_slow 的原因

- 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

討論替換演算法可以優化的原因

- 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

結論

在本學期的 workload 分析中，我們深入研究了優化程式碼的方法。首先，比較了使用不同編譯優化級別（-O0 和 -O3）的 rabinseg 函式組語，顯示出 -O3 的亂序執行可以提升並行性和效能。其次，探討了在某些情況下避免使用 loop unroll，尤其針對循環展開可能導致冗余程式碼的情境。最後，討論了替換演算法的優化，通過省略 Hash Table 使用和簡化 Run-Length Encoding 成功減少了記憶體和運算開銷，特別適用於簡單數據模式的情況。
總的來說，本篇報告剖析了不同優化方法的效果與原因，強調了根據程式碼特性選擇適當的優化策略的重要性。