軟體分析與最佳化 Workload Analysis Stage 2

組別：2
成員： 612410017 林靖紳、612410066 蔡宏遠

Review Stage 1 Results

- 根據 Stage 1 的分析結果，我們打算以 gcc 作為之後主要分析的編譯器
  - GCC
    - -O3

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

    - -O0

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

  - ICC
    - -O3

```
real    0m5.249s
user    0m13.975s
sys     0m3.782s
```

    - -O0

```
real    0m5.344s
user    0m14.020s
sys     0m4.003s
```

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.
```

- ICC version

```
ashen@Stephanie-Lin:~/Documents/Software_Analysis-git/HW2$ icc --version
icc: remark #10441: The Intel(R) C++ Compiler Classic (ICC) is deprecated and will be removed from
product release in the second half of 2023. The Intel(R) oneAPI DPC++/C++ Compiler (ICX) is the rec
ommended compiler moving forward. Please transition to use this compiler. Use '-diag-disable=10441'
 to disable this message.
icc (ICC) 2021.10.0 20230609
Copyright (C) 1985-2023 Intel Corporation.  All rights reserved.
```

編譯參數:

```
export CFLAGS="-DUNIX -O0 -funroll-loops -fprefetch-loop-arrays ${
export CXXFLAGS="-DUNIX -O0 -funroll-loops -fprefetch-loop-arrays
export CPPFLAGS=""
export CXXCPPFLAGS=""
export LDFLAGS="-L${CC_HOME}/lib64 -L${CC_HOME}/lib"
export LIBS=""
export EXTRA_LIBS=""
export PARMACS_MACRO_FILE="pthreads"
```

使用 vtune 初步分析

gcc -O0

Elapsed Time: 5.041s

CPU Time:	**25.519s**	
Effective Time:	25.519s	
Spin Time:	0s	
Overhead Time:	0s	
Instructions Retired:	170,988,300,000	
Microarchitecture Usage:	**51.3%**	**of Pipeline Slots**
CPI Rate:	0.632	
Total Thread Count:	16	
Paused Time:	0s	

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.*

1. 首先我們使用 gcc -O0 編譯發現 hotspot 為 rabinseg 函式，緊接著是第二高的 hotspot 是 deflate_slow 函式
2. 在這樣的情況下，需要考量優化的函式有兩個，要直接用腦袋想兩個辦法有些麻煩，因此我們先嘗試使用 compiler 的編譯選項做簡單且直接的優化

gcc -O3

Elapsed Time: 4.080s

CPU Time:	**16.838s**	
Effective Time:	16.838s	
Spin Time:	0s	
Overhead Time:	0s	
Instructions Retired:	125,096,400,000	
Microarchitecture Usage:	**52.6%**	of Pipeline Slots
CPI Rate:	0.564	
Total Thread Count:	16	
Paused Time:	0s	

Top Hotspots

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

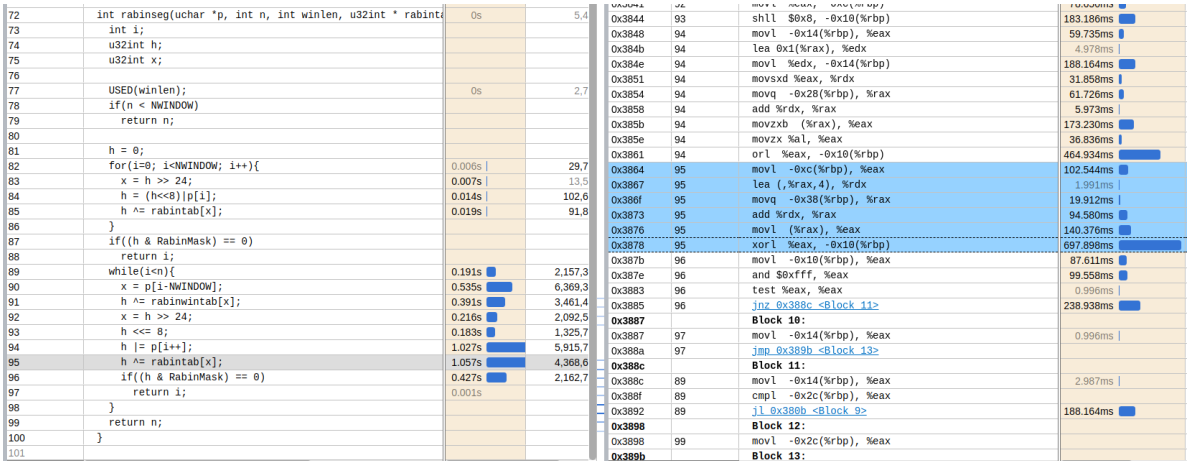
Function	Module	CPU Time	% of CPU Time
deflate_slow	dedup	3.523s	20.9%
pqdownheap	dedup	3.305s	19.6%
[Outside any known module]	[Unknown]	2.660s	15.8%
sha1_block_data_order	dedup	1.878s	11.2%
rabinseg	dedup	1.440s	8.5%
[Others]	N/A*	4.032s	23.9%

N/A is applied to non-summable metrics.

1. 可以看到 gcc -O3 編譯發現 hotspot 只剩下 deflate_slow ， 並優化了 rabinseg 函式的執行時間比例
2. **因此後續主要會針對在 -O3 編譯選項下的 deflate_slow 函式進行優化**

討論 -O3 如何優化 rabinseg 函式

- **-O0**



- **-O3**

Source Line ▲	Source	CPU Time	Ins
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	

Source Line ▲	Source	CPU Time	Ins
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
0x3a6a	93	mov %eax, %r12d	
0x3a6d	92	shr $0x18, %eax	3.982ms
0x3a70	93	shl $0x8, %r12d	41.814ms
0x3a74	94	or %r12d, %ebx	6.969ms
0x3a77	95	xorl (%rcx,%rax,4), %ebx	218.031ms
0x3a7a	95	mov %ebx, %eax	1.991ms
0x3a7c	96	lea 0x1(%rsi), %rbx	
0x3a80	96	test $0xfff, %eax	18.916ms
0x3a85	96	jz 0x3b20 <Block 19>	
0x3a8b		**Block 15:**	
0x3a8b	91	movzxb -0x21(%rdi,%rbx,1), %r9d	23.894ms
0x3a91	94	movzxb -0x1(%rdi,%rbx,1), %edx	1.991ms
0x3a96	91	xorl (%r8,%r9,4), %eax	1.991ms
0x3a9a	94	mov %ebx, %r9d	12.942ms
0x3a9d	94	lea 0x2(%rsi), %rbx	27.876ms
0x3aa1	93	mov %eax, %r12d	1.991ms
0x3aa4	92	shr $0x18, %eax	0ms
0x3aa7	93	shl $0x8, %r12d	23.894ms
0x3aab	94	or %r12d, %edx	41.814ms
0x3aae	95	xorl (%rcx,%rax,4), %edx	174.226ms
0x3ab1	95	mov %edx, %eax	1.991ms
0x3ab3	96	test $0xfff, %eax	28.872ms
0x3ab8	96	jz 0x3b20 <Block 19>	0.996ms
0x3aba		**Block 16:**	
0x3aba	91	movzxb -0x1f(%rdi,%rsi,1), %r9d	0ms
0x3ac0	94	movzxb 0x1(%rdi,%rsi,1), %edx	23.894ms
0x3ac5	91	xorl (%r8,%r9,4), %eax	0.996ms
0x3ac9	94	mov %ebx, %r9d	16.925ms
0x3acc	94	lea 0x3(%rsi), %rbx	0.996ms
0x3ad0	93	mov %eax, %r12d	23.894ms
0x3ad3	92	shr $0x18, %eax	2.987ms
0x3ad6	93	shl $0x8, %r12d	21.903ms
0x3ada	94	or %r12d, %edx	0ms
0x3add	95	xorl (%rcx,%rax,4), %edx	238.938ms
0x3ae0	95	mov %edx, %eax	0.996ms
0x3ae2	96	test $0xfff, %eax	30.863ms
0x3ae7	96	jz 0x3b20 <Block 19>	
0x3ae9		**Block 17:**	
0x3ae9	91	movzxb -0x1e(%rdi,%rsi,1), %r9d	
0x3aef	91	xorl (%r8,%r9,4), %eax	16.925ms
0x3af3	94	mov %ebx, %r9d	
0x3af6	91	mov %eax, %edx	17.920ms

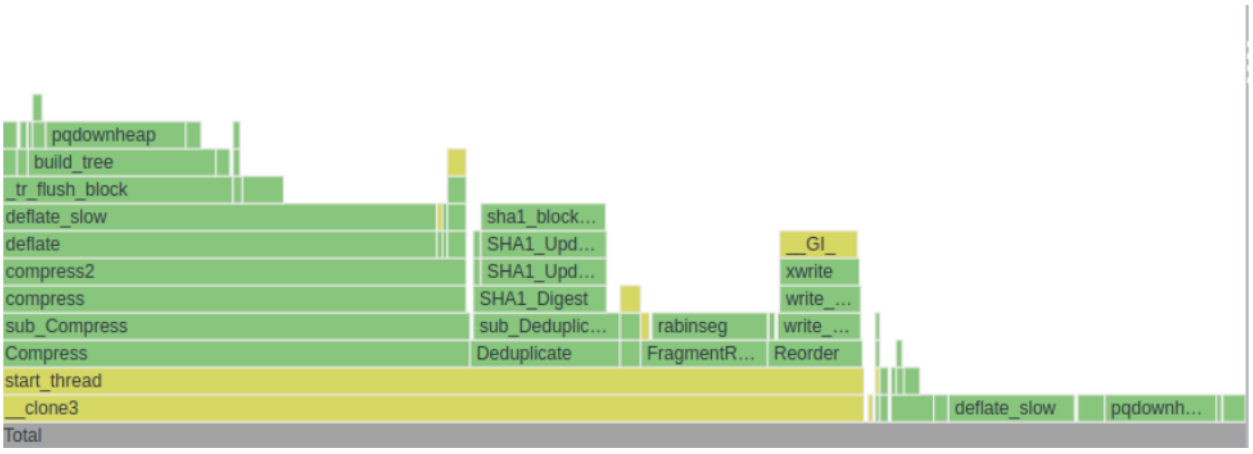
Source Line ▲	Source	CPU Time	Ins
59	for(i=0; i<256; i++)		
60	rabinwintab[i] = fpwinreduce(ir		
61	return;		
62	}		
63			
64	void rabininit(int winlen, u32int * rabin		
65	//rabintab = malloc(256*sizeof rabintab		
66	//rabinwintab = malloc(256*sizeof rabin		
67	fpmkredtab(irrpoly, 0, rabintab);		
68	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 Time
0x3ac0	94	movzxb 0x1(%rdi,%rsi,1), %edx	23.894ms
0x3ac5	91	xorl (%r8,%r9,4), %eax	0.996ms
0x3ac9	94	mov %ebx, %r9d	16.925ms
0x3acc	94	lea 0x3(%rsi), %rbx	0.996ms
0x3ad0	93	mov %eax, %r12d	23.894ms
0x3ad3	92	shr $0x18, %eax	2.987ms
0x3ad6	93	shl $0x8, %r12d	21.903ms
0x3ada	94	or %r12d, %edx	0ms
0x3add	95	xorl (%rcx,%rax,4), %edx	238.938ms
0x3ae0	95	mov %edx, %eax	0.996ms
0x3ae2	96	test $0xfff, %eax	30.863ms
0x3ae7	96	jz 0x3b20 <Block 19>	
0x3ae9		**Block 17:**	
0x3ae9	91	movzxb -0x1e(%rdi,%rsi,1), %r9d	
0x3aef	91	xorl (%r8,%r9,4), %eax	16.925ms
0x3af3	94	mov %ebx, %r9d	
0x3af6	91	mov %eax, %edx	17.920ms
0x3af8	93	mov %eax, %r12d	0ms
0x3afb	94	movzxb 0x2(%rdi,%rsi,1), %eax	28.872ms
0x3b00	94	add $0x4, %rsi	0.996ms
0x3b04	93	shl $0x8, %r12d	10.951ms
0x3b08	92	shr $0x18, %edx	2.987ms
0x3b0b	94	or %r12d, %eax	24.889ms
0x3b0e	95	xorl (%rcx,%rdx,4), %eax	228.982ms
0x3b11	96	test $0xfff, %eax	
0x3b16	96	jnz 0x3a50 <Block 13>	44.801ms
0x3b1c		**Block 18:**	
0x3b1c	96	nopl %eax, (%rax)	
0x3b20		**Block 19:**	
0x3b20	94	mov %r9d, %r10d	
0x3b23		**Block 20:**	
0x3b23	100	popq %rbx	
0x3b24	100	mov %r10d, %eax	
0x3b27	100	popq %r12	
0x3b29	100	popq %rbp	
0x3b2a	100	retq	
0x3b2b		**Block 21:**	
0x3b2b	100	mov $0x20, %r10d	
0x3b31	100	popq %rbx	
0x3b32	100	popq %r12	
0x3b34	100	mov %r10d, %eax	
0x3b37	100	popq %rbp	

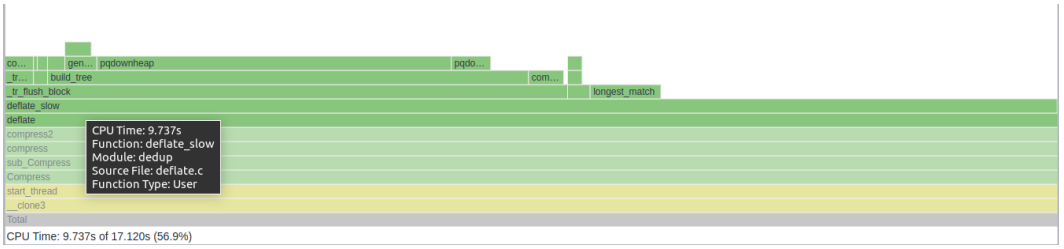
- **分析**

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)

Flame Graph



- 從 flame graph 中可以看到以下幾點：
  - 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



Sampling

CPI Rate

可以從 CPI 看到 -O3 程式的平行化程度有明顯的提昇

- CPI（-O0）: 0.632
- CPI（-O3）: 0.564

Memory

可以看到下圖使用 -O3 與 -O0 比較在 load/store 還有 LLC miss 的次數都大幅度的降低

- gcc -O0

Elapsed Time: 4.792s

CPU Time:	20.030s	
Memory Bound:	**7.1%**	**of Pipeline Slots**
L1 Bound:	14.1%	of Clockticks
L2 Bound:	2.0%	of Clockticks
L3 Bound:	1.9%	of Clockticks
DRAM Bound:	**1.0%**	**of Clockticks**
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
Store Bound:	1.4%	of Clockticks
Loads:	53,766,070,468	
Stores:	24,477,828,827	
LLC Miss Count:	394,603	
Total Thread Count:	16	
Paused Time:	0s	

- gcc -O3

Elapsed Time: 3.866s

CPU Time:	11.712s	
Memory Bound:	**8.3%**	**of Pipeline Slots**
L1 Bound:	16.0%	of Clockticks
L2 Bound:	0.0%	of Clockticks
L3 Bound:	0.9%	of Clockticks
DRAM Bound:	**1.4%**	**of Clockticks**
DRAM Bandwidth Bound:	0.0%	of Elapsed Time
Store Bound:	2.3%	of Clockticks
Loads:	20,190,359,746	
Stores:	8,824,850,780	
LLC Miss Count:	329,261	
Total Thread Count:	16	
Paused Time:	0s	

是否有使用 compiler directive 的機會

在 deflate_slow 函式中使用率最高的 hotspot 就是 INSERT_STRING，這個函式主要是將 str 插入到 hash dictionary 中，並記錄具有相同 hash key 的最近字串的資訊

Source Line	Source	CPU Time: Total	CPU
1553	*/		
1554	local block_state deflate_slow(s, flush)		
1555	deflate_state *s;		
1556	int flush;		
1557	{		
1558	IPos hash_head = NIL; /* head of hash chain */		
1559	int bflush; /* set if current block		
1560			
1561	/* Process the input block. */		
1562	for (;;) {		
1563	/* Make sure that we always have enough looka		
1564	* at the end of the input file. We need MAX_		
1565	* for the next match, plus MIN_MATCH bytes t		
1566	* string following the next match.		
1567	*/		
1568	if (s->lookahead < MIN_LOOKAHEAD) {		
1569	fill_window(s);	0.1%	
1570	if (s->lookahead < MIN_LOOKAHEAD && flush		
1571	return need_more;		
1572	}		
1573	if (s->lookahead == 0) break; /* flush the		
1574	}		
1575			
1576	/* Insert the string window[strstart .. strsta		
1577	* dictionary, and set hash_head to the head		
1578	*/		
1579	if (s->lookahead >= MIN_MATCH) {		
1580	INSERT_STRING(s, s->strstart, hash_head);	11.6%	
1581	}		
1582			
1583	/* Find the longest match, discarding those <=		
1584	*/		
1585	s->prev_length = s->match_length, s->prev_mat	0.4%	
1586	s->match_length = MIN_MATCH-1;		

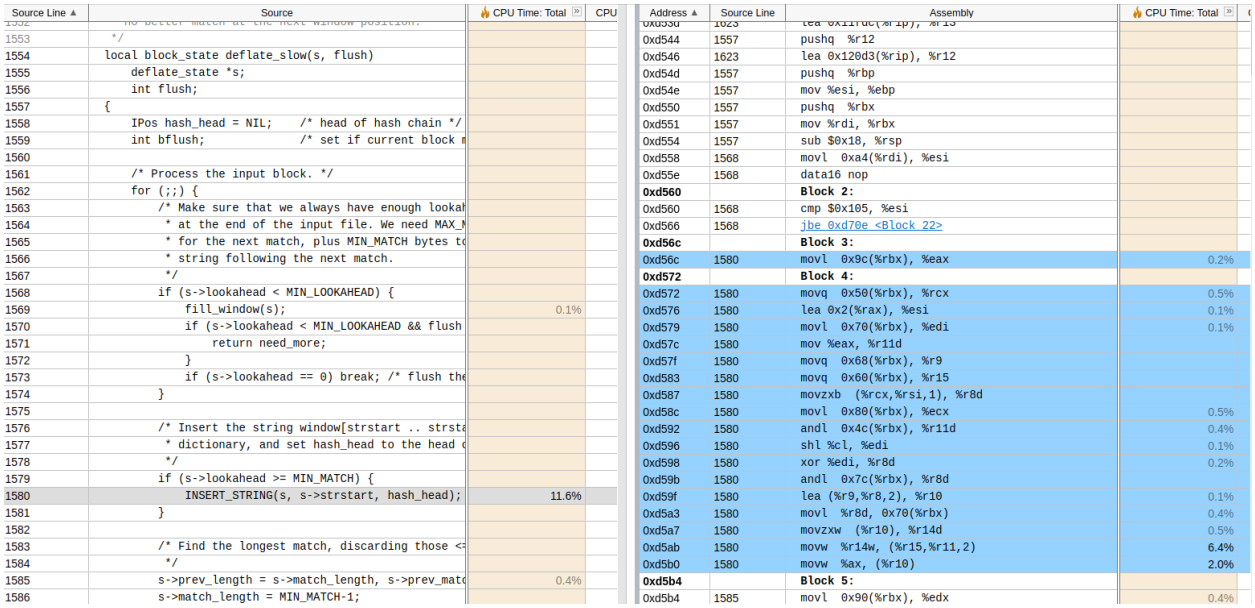
Address	Source Line	Assembly	CPU Time: Total
0xd544	1557	pushq %r12	
0xd546	1623	lea 0x120d3(%rip), %r12	
0xd54d	1557	pushq %rbp	
0xd54e	1557	mov %esi, %ebp	
0xd550	1557	pushq %rbx	
0xd551	1557	mov %rdi, %rbx	
0xd554	1557	sub $0x18, %rsp	
0xd558	1568	movl 0xa4(%rdi), %esi	
0xd55e	1568	data16 nop	
0xd560		**Block 2:**	
0xd560	1568	cmp $0x105, %esi	
0xd566	1568	jbe 0xd70e <Block 22>	
0xd56c		**Block 3:**	
0xd56c	1580	movl 0x9c(%rbx), %eax	0.2%
0xd572		**Block 4:**	
0xd572	1580	movq 0x50(%rbx), %rcx	0.5%
0xd576	1580	lea 0x2(%rax), %esi	0.1%
0xd579	1580	movl 0x70(%rbx), %edi	0.1%
0xd57c	1580	mov %eax, %r11d	
0xd57f	1580	movq 0x68(%rbx), %r9	
0xd583	1580	movq 0x60(%rbx), %r15	
0xd587	1580	movzxb (%rcx,%rsi,1), %r8d	
0xd58c	1580	movl 0x80(%rbx), %ecx	0.5%
0xd592	1580	andl 0x4c(%rbx), %r11d	0.4%
0xd596	1580	shl %cl, %edi	0.1%
0xd598	1580	xor %edi, %r8d	0.2%
0xd59b	1580	andl 0x7c(%rbx), %r8d	
0xd59f	1580	lea (%r9,%r8,2), %r10	0.1%
0xd5a3	1580	movl %r8d, 0x70(%rbx)	0.4%
0xd5a7	1580	movzxw (%r10), %r14d	0.5%
0xd5ab	1580	movw %r14w, (%r15,%r11,2)	6.4%
0xd5b0	1580	movw %ax, (%r10)	2.0%
0xd5b4		**Block 5:**	
0xd5b4	1585	movl 0x90(%rbx), %edx	0.4%

且在 INSERT_STRING 函式的部份已使用 compiler directive

```
/* ===========================================================================
 * Insert string str in the dictionary and set match_head to the previous head
 * of the hash chain (the most recent string with same hash key). Return
 * the previous length of the hash chain.
 * If this file is compiled with -DFASTEST, the compression level is forced
 * to 1, and no hash chains are maintained.
 * IN  assertion: all calls to to INSERT_STRING are made with consecutive
 *    input characters and the first MIN_MATCH bytes of str are valid
 *    (except for the last MIN_MATCH-1 bytes of the input file).
 */
#ifdef FASTEST
#define INSERT_STRING(s, str, match_head) \
   (UPDATE_HASH(s, s->ins_h, s->window[(str) + (MIN_MATCH-1)]), \
    match_head = s->head[s->ins_h], \
    s->head[s->ins_h] = (Pos)(str))
#else
#define INSERT_STRING(s, str, match_head) \
   (UPDATE_HASH(s, s->ins_h, s->window[(str) + (MIN_MATCH-1)]), \
    match_head = s->prev[(str) & s->w_mask] = s->head[s->ins_h], \
    s->head[s->ins_h] = (Pos)(str))
#endif
```

是否有使用 SIMD 的機會

同樣是 INSERT_STRING 作為 hotspot，反組譯後在 vtune 上並沒有看到使用 SIMD 的痕跡



Source Line	Source	CPU Time: Total	CPU
1553	*/		
1554	local block_state deflate_slow(s, flush)		
1555	deflate_state *s;		
1556	int flush;		
1557	{		
1558	IPos hash_head = NIL; /* head of hash chain */		
1559	int bflush; /* set if current block m		
1560			
1561	/* Process the input block. */		
1562	for (;;) {		
1563	/* Make sure that we always have enough lookah		
1564	* at the end of the input file. We need MAX_M		
1565	* for the next match, plus MIN_MATCH bytes to		
1566	* string following the next match.		
1567	*/		
1568	if (s->lookahead < MIN_LOOKAHEAD) {		
1569	fill_window(s);	0.1%	
1570	if (s->lookahead < MIN_LOOKAHEAD && flush		
1571	return need_more;		
1572	}		
1573	if (s->lookahead == 0) break; /* flush the		
1574	}		
1575			
1576	/* Insert the string window[strstart .. strsta		
1577	* dictionary, and set hash_head to the head o		
1578	*/		
1579	if (s->lookahead >= MIN_MATCH) {		
1580	INSERT_STRING(s, s->strstart, hash_head);	11.6%	
1581	}		
1582			
1583	/* Find the longest match, discarding those <=		
1584	*/		
1585	s->prev_length = s->match_length, s->prev_matc	0.4%	
1586	s->match_length = MIN_MATCH-1;		

Address	Source Line	Assembly	CPU Time: Total
0xd544	1557	pushq %r12	
0xd546	1623	lea 0x120d3(%rip), %r12	
0xd54d	1557	pushq %rbp	
0xd54e	1557	mov %esi, %ebp	
0xd550	1557	pushq %rbx	
0xd551	1557	mov %rdi, %rbx	
0xd554	1557	sub $0x18, %rsp	
0xd558	1568	movl 0xa4(%rdi), %esi	
0xd55e	1568	data16 nop	
0xd560		**Block 2:**	
0xd560	1568	cmp $0x105, %esi	
0xd566	1568	jbe 0xd70e <Block 22>	
0xd56c		**Block 3:**	
0xd56c	1580	movl 0x9c(%rbx), %eax	0.2%
0xd572		**Block 4:**	
0xd572	1580	movq 0x50(%rbx), %rcx	0.5%
0xd576	1580	lea 0x2(%rax), %esi	0.1%
0xd579	1580	movl 0x70(%rbx), %edi	0.1%
0xd57c	1580	mov %eax, %r11d	
0xd57f	1580	movq 0x68(%rbx), %r9	
0xd583	1580	movq 0x60(%rbx), %r15	
0xd587	1580	movzxb (%rcx,%rsi,1), %r8d	
0xd58c	1580	movl 0x80(%rbx), %ecx	0.5%
0xd592	1580	andl 0x4c(%rbx), %r11d	0.4%
0xd596	1580	shl %cl, %edi	0.1%
0xd598	1580	xor %edi, %r8d	0.2%
0xd59b	1580	andl 0x7c(%rbx), %r8d	
0xd59f	1580	lea (%r9,%r8,2), %r10	0.1%
0xd5a3	1580	movl %r8d, 0x70(%rbx)	0.4%
0xd5a7	1580	movzxw (%r10), %r14d	0.5%
0xd5ab	1580	movw %r14w, (%r15,%r11,2)	6.4%
0xd5b0	1580	movw %ax, (%r10)	2.0%
0xd5b4		**Block 5:**	
0xd5b4	1585	movl 0x90(%rbx), %edx	0.4%

但我認為使用 SIMD 的機會是有的，可以將需要插入 hash dictionary 的 string 和 string 準備插入的相關資訊 buffer 起來，然後在批次記錄這些具有相同 hash key 的字串，因此就可以使用 SIMD 指令做插入，大幅度提高 INSERT_STRING 的並行處理效率

討論

- 結論：以上分析出 gcc -O3 中 deflate_slow 函式是整個執行過程中的熱點

- 觀察 deflate_slow 函式的 source code

```
1554 local block_state deflate_slow(s, flush)
1555     deflate_state *s;
1556     int flush;
1557 {
1558     IPos hash_head = NIL;     /* head of hash chain */
1559     int bflush;               /* set if current block must be flushed */
1560 
1561     /* Process the input block. */
1562     for (;;) {
1563         /* Make sure that we always have enough lookahead, except
1564          * at the end of the input file. We need MAX_MATCH bytes
1565          * for the next match, plus MIN_MATCH bytes to insert the
1566          * string following the next match.
1567          */
1568         if (s->lookahead < MIN_LOOKAHEAD) {
1569             fill_window(s);
1570             if (s->lookahead < MIN_LOOKAHEAD && flush == Z_NO_FLUSH) {
1571                 return need_more;
1572             }
1573             if (s->lookahead == 0) break; /* flush the current block */
1574         }
1575 
1576         /* Insert the string window[strstart .. strstart+2] in the
1577          * dictionary, and set hash_head to the head of the hash chain:
1578          */
1579         if (s->lookahead >= MIN_MATCH) {
1580             INSERT_STRING(s, s->strstart, hash_head);
1581         }
```

- 目的：用於實現「slow」壓縮策略

- 大致可以分成以下幾個步驟

  - 透過迴圈處理輸入的 blocks
  - 確保始終有足夠的 lookahead 進行處理，需要時填充window
  - 插入當前字串到字典中
  - 尋找最長匹配，捨棄比前一個匹配短的匹配
  - 根據壓縮策略處理匹配和文字，更新 hash table
  - 在需要時刷新 block

認為可以優化的地方

- 如果是對於包含較多重複資料（字串）的數據，我們認為或許不需要用到 hash table 相對較為複雜的資料結構

- 只需要針對 Run-Length Encoding，簡單地比較相鄰的字節，而不需要複雜的匹配條件。

- 利用相同字節的重複次數即為匹配的長度，不需要進行額外的計算。

- 不用像 deflate_slow 一樣嘗試找到最長的匹配，並僅在確定在下一個窗口位置沒有更好匹配時，才使用當前的匹配

- 影響
    - 壓縮率：
        - 以上的改動可能會影響資料的壓縮效率
        - 以上方法應該比較適用於像是對於像素圖像中的單色區域，或者包含大量重複字符的文本
    - 複雜度：
        - 上述方法的實現應該較為簡單易懂，deflate_slow 有太多的條件判斷分支

- 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

```