How C programs work

(寧夏 Day 1)

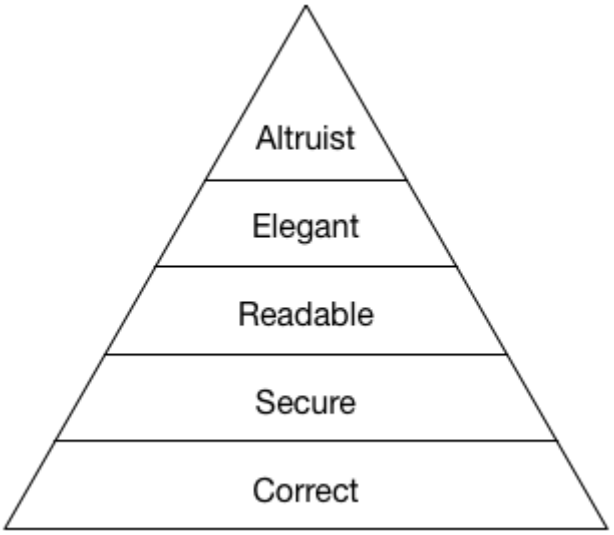
有所變，有所不變

- 1980 年代的電腦廣告: IMSAI 8080

source: twitter

- USD $5995 (而且是 35 年前的物價！) 只能買到 8-bit 微處理器，搭配 64 KB 主記憶體和 10 MB 硬碟。有趣的是，當時軟體開發的模式部分還延續至今
  - 硬體突飛猛進，軟體的問題依舊還在

Maslow's pyramid of code review



- 21 世紀的軟體開發均已規模化，絕非「有就好」，而是持續演化和重構，code review 是免不了的訓練
- uber 工程師 Charles-Axel Dein 認為好的程式碼應該要:
  - [Correct] : 做到預期的行為了嗎？能夠處理各式邊際狀況嗎？即便其他人修改程式碼後，主體的行為仍符合預期嗎？
  - [Secure] : 面對各式輸入條件或攻擊，程式仍可正確運作嗎？
  - [Readable] : 程式碼易於理解和維護嗎？
  - [Elegant] : 程式碼夠「美」嗎？可以簡潔又清晰地解決問題嗎？
  - [Altruist] : 除了滿足現有的狀況，軟體在日後能夠重用嗎？甚至能夠抽離一部分元件，給其他專案使用嗎？
- 「需求」層次: 正確 → 安全 → 可讀 → 優雅 → 利他
- @徐元豪 : 「工程師都會寫程式，但是 (往往) 欠了 code review 的能力。code review 不是用來戰別人，而是訓練自己如何看得懂別人程式、了解別人的想法，常看人家的程式，無形中是在訓練自己。」
- @Chaoint ：「沒關係。」總是講給自己聽的。
- 「人們不敘述自己的過往，而是為自己的過往作證。」—— Frantz Fanon
- 「如果你把游泳池當作浴缸泡著，再泡幾年還是不會游泳」 -- jserv

延伸閱讀: Code Reading

- 大量篇幅回顧 C 語言概念，以及在真實世界中的程式如何展現，細節！

什麼叫做簡潔？

COUNTEREXAMPLE TO EULER'S CONJECTURE ON SUMS OF LIKE POWERS

BY L. J. LANDER AND T. R. PARKIN

Communicated by J. D. Swift, June 27, 1966

A direct search on the CDC 6600 yielded

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5$$

as the smallest instance in which four fifth powers sum to a fifth power. This is a counterexample to a conjecture by Euler [1] that at least n nth powers are required to sum to an nth power, $n > 2$.

REFERENCE

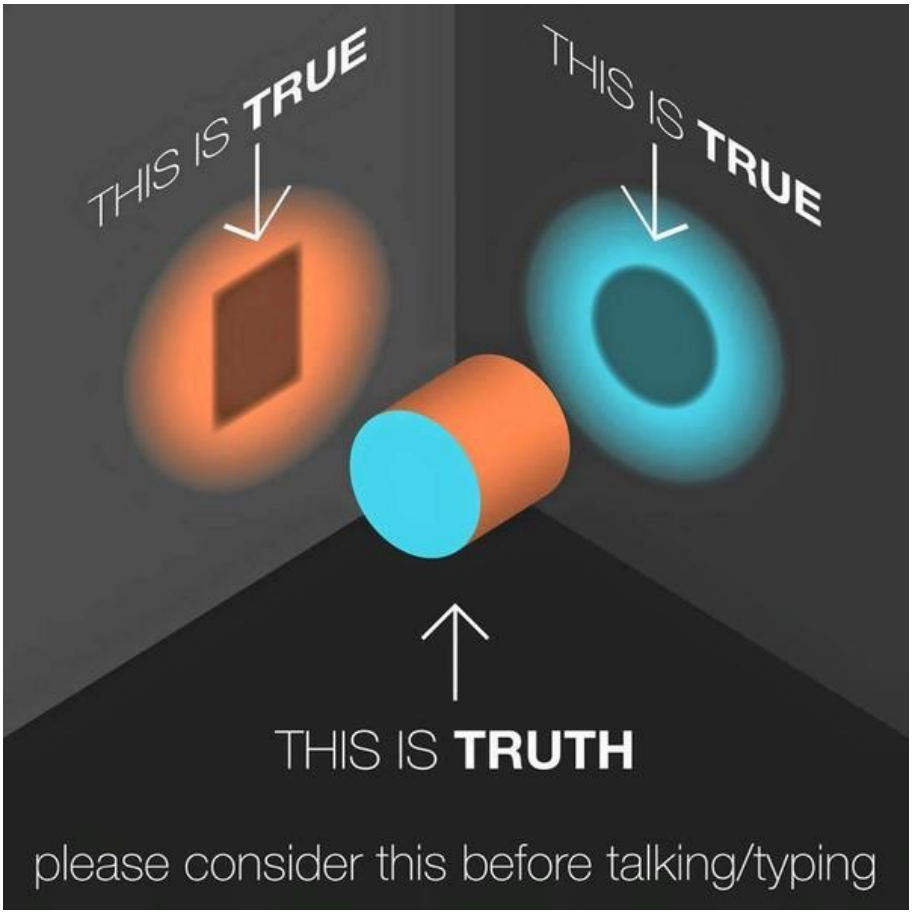
1. L. E. Dickson, *History of the theory of numbers*, Vol. 2, Chelsea, New York, 1952, p. 648.

尤拉猜想是 Euler 在 1769 年對費馬定理的延伸： n 個正整數的 k 次方的總和，若是另一個正整數的 k 次方，則 n 不可能小於 k。n = 2 就是費馬最後定理，不過這個猜想在 1966 年被號稱「最短論文」給否定。

延伸閱讀:

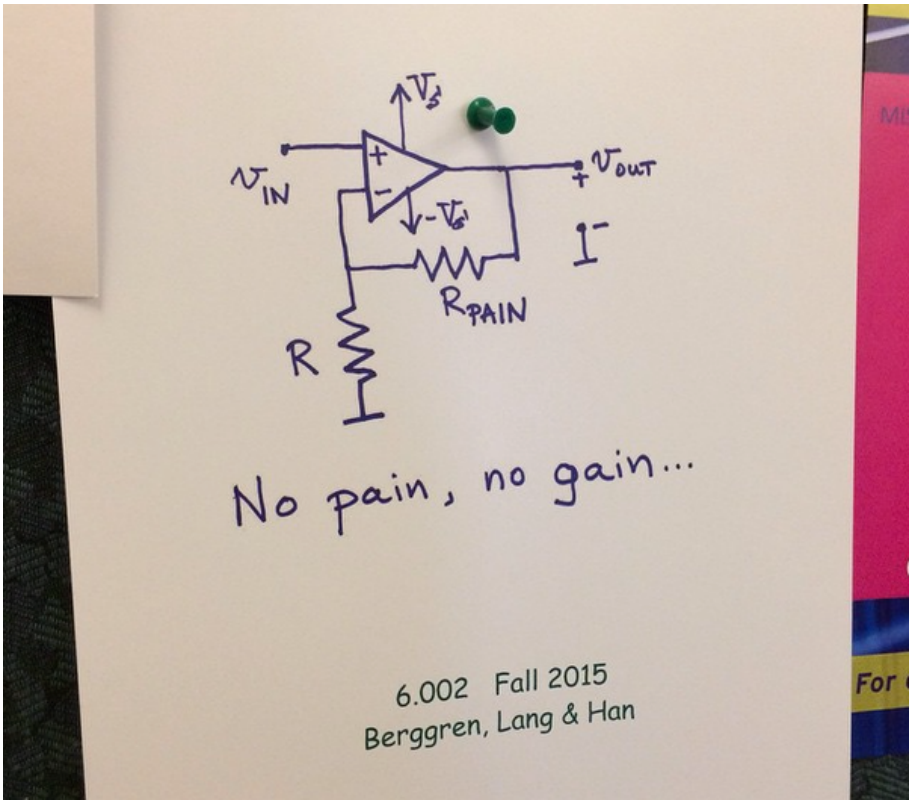
- 美麗的錯誤--猜想的真與假
- 邏輯思維：費馬大定理 (YouTube)

「事實」很容易被遮蔽，所以我們要 Benchmark / Profiling



source: twitter

《進擊的鼓手》之後



"No pain, no gain"
source: CSAIL at MIT twitter

- 延伸閱讀: Operational Amplifier

理解系統程式

- 從實際案例著手: 從無到有打造類似 Facebook 社群網站

計算機結構的改變

[source]

- 早年計算能力相對低的年代，常常有用查表法代替計算，有空間換取時間的做法，來增進效能。... 那個時候 個人電腦的 CPU 可以在一個時脈週期中讀取一筆資料，但是要做乘法計算則需要幾十個時脈週期，所以用查表的比較快。除法和超越函數更是如此，而現在還有一些低階的處理器，還在用這些技巧。
- 後來當 CPU 時脈提高，但記憶體存取相對變慢的時候，我們必須反過來減少記憶體存取的次數，所以高階處理器 cache 越來越大，做 data prefetch 來提早取得資料、使用 multi-threaded architecture 來容忍資料遲到的狀況、使用壓縮的

方式傳送資料，甚至還會用 speculation 的方式來猜測資料是在哪裡和是什麼。

- 在多處理機和多核心電腦上，存取資料的問題更嚴重，除了時間延遲和頻寬之外，還要考慮到尖峰時刻塞車的問題，所以有時候簡單的工作，就可能就不分工了，要不就由一個 CPU 代表去做，做完把結果給大家，要不就大家都做同樣的事情。前者多半在有共享記憶體的多核心處理器上看到，後者多半在分散式的系統看到。
- 到了異質計算的年代，CPU 和 GPU 的分工，更需要好好地做效能分析。因為傳統 GPU 和 CPU 不共享記憶體，透過較慢的 PCIe Bus 交換資料，所以有些工作 CPU 自己做比較快。另一方面，當 GPU 有超過 2000 個核心的時候，用重複的計算 (redundant computation)取代資料交換，也是常見的事。
  - http://www.hsafoundation.com/
- 更進一步談巨量資料，為了節省資料的取得時間，我們往往費盡心思。我們花時間將資料和計算擺在同一個地方，做所謂的 data computation co-location，將重複出現的資料利用 data deduplication 技術節省儲存空間和取得時間，用一堆目錄 (indexing) 讓資料可以快速被找到。
- 當計算機結構有所不同時，優化的策略可能會隨之而變，不能食古不化。但原理雖然簡單，系統和實作技巧卻越來越複雜，軟硬體優化的機會越來越多，可惜能夠真正連通理論和實務的人則越來越少。
- 以上這些技術，講起來很容易，但在實作上，必須先搞清楚運算和資料的相對位置、距離、時間先後、相依性、數量等等，才知道該如何取捨。但很多人根本不會用效能分析工具，就在那邊瞎子摸象，隨便亂講，這時候要解決問題，就需要瞎貓遇到死耗子的運氣。

- i586 和 i686 看起來指令相似，但本質不同！
  - 從 i686 (Pentium Pro) 開始，底層已經是 **RISC 架構**
- 因為現在計算機結構改變很大
  - 即便把程式用組合語言重寫，效能也不見得比 Compiler 產生的還好
    - 效能的問題在存取資料本身
    - 組合語言會快，是因為你分析過程式要怎樣寫才可以比較快
      - 直接照著程式碼的邏輯改寫組合語言不見得比較好

- hyperloglog
  - 使用 1.5k 表達 10 億筆資料
    - 正規表示法？

    - Python implementation of Hyperloglog, redis, fuzzy hashing for malware detection

(從測驗來刺激思考: Quiz-0)
「你所不知道的 C 語言」系列講座
A circuit-like notation for lambda calculus

- 1928年，Alonzo Church 發明了lambda演算（當時他 25 歲）。Lambda 演算被設計為一個通用的計算模型，並不是為了解決某個特定的問題而誕生的。1929 年，Church 成為普林斯頓大學教授。1932 年，Church 在 Annals of Mathematics 發表了一篇論文，糾正邏輯領域裡幾個常見的問題，他的論述中用了lambda演算。1935年，Church 發表論文，使用 lambda 演算證明基本數論中存在不可解決的問題。1936 年 4 月，Church發 表了一篇兩頁紙的 “note”，指出自己 1935 年那篇論文可以推論得出，著名的Hilbert“可判定性問題”是不可解決的

Programming Small

- 在小處下功夫，不放棄整體改善的機會
    - 快速計算和表達圓周率
        - 解說
    - 字串反轉
- C 語言講究效率
    - 為什麼 C 語言沒有內建 swap 機制?
        - 很難作出通用且有效率的 swap 方式
        - Swapping in C, C++, and Java
- 最佳化來自對系統的認知
    - 假設我們有兩個****有號整數***:

```
#include <stdint.h>

int32_t a, b;
```

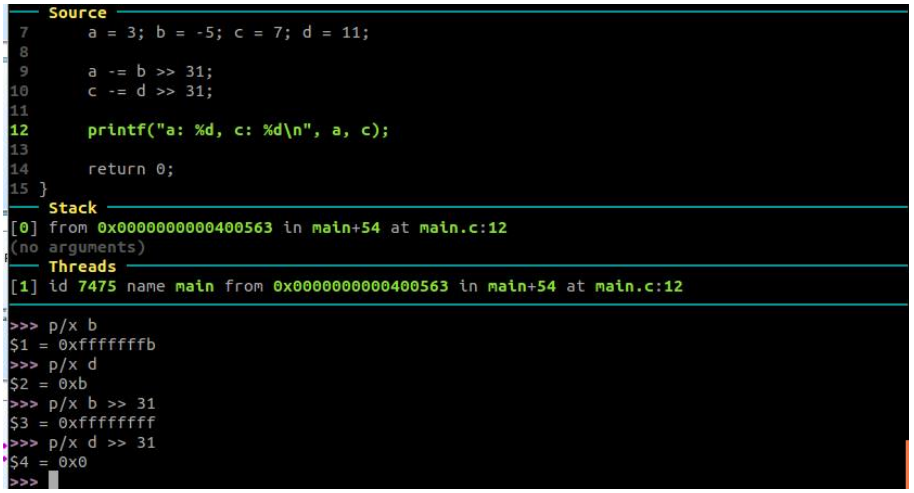
    - 然後原本涉及到分支的陳述:

```
if (b < 0) a++;
```

    - 可更換為沒有分支的版本: # superscalar https://en.wikipedia.org/wiki/Superscalar_processor

```
a -= b >> 31;
```

    - 為什麼呢？一旦 b 右移 31 個 bit，在右移 `>>` 時在前面補上 sign bit ，所以移完會是 0xffff 也就是-1，故結果會是 `a -= -1`，即 `a++`，於是，原本為負數的值就變成 1 ，而原本正數的值就變成 0，又因為原本的行為是有條件的 a++，如此就是數值操作，沒有任何分支指令。



案例分析: Phone Book

目的：分析電話簿搜尋程式，探討 cache miss 對於整體效能有顯著影響

題目說明

思考以下程式碼可能存在的問題，並著手改善效能。

Hint: cache miss

```
#define MAX_LAST_NAME_SIZE 16
typedef struct __PHONE_BOOK_ENTRY {65
    char LastName[MAX_LAST_NAME_SIZE];
    char FirstName[16];
    char email[16];
    char phone[10];
    char cell[10];
    char addr1[16];
    char addr2[16];
    char city[16];
    char state[2];
    char zip[5];
    struct __PHONE_BOOK_ENTRY *pNext;
} PhoneBook;

PhoneBook *FindName(char Last[], PhoneBook *pHead) {
    while (pHead != NULL) {
        if (stricmp(Last, pHead->LastName) == 0)
            return pHead;
        pHead = pHead->pNext;
    }
    return NULL;
}
```

- L1 Cache : 32KB = 32*1024，PhoneBook size = 136 bytes，32*1024 / (136*8) = 30.12 (只能存 30 筆左右)
- 是否可最佳化成 32*1024 / (32*8) = 128 筆？ 或使用 collision 較少的 hash function？

未優化版本

程式碼: phonebook

測試檔：

GitHub上有提供兩種字典檔當輸入 data set，以下測試使用**第二種**。

1. 男生 + 女生英文名字的攻擊字典檔。約 16,750 個。
2. 英文單字表，約 350,000 個 (sorted)

`$ perf stat -r 10 -e cache-misses,cache-references,L1-dcache-load-misses,L1-dcache-store-misses,L1-dcache-prefetch-misses,L1-icache-load-misses ./main_origin`

```
size of entry : 136 bytes
  uninvolved  is found!
      zyxel  is found!
  whiteshank  is found!
  odontomous  is found!
 pungoteague  is found!
  reweighted  is found!
xiphisternal  is found!
   yakattalo  is found!
execution time of append() : 0.053549
execution time of findName() : 0.050462

Performance counter stats for './main_origin' (10 runs):
3,519,048   cache-misses       #   96.963 % of all cache refs    ( +-  1.61% ) [65.55%]
3,629,269   cache-references                                     ( +-  1.29% ) [67.73%]
4,185,434   L1-dcache-load-misses                               ( +-  1.10% ) [69.53%]
1,005,501   L1-dcache-store-misses                              ( +-  0.77% ) [70.04%]
3,088,038   L1-dcache-prefetch-misses                           ( +-  1.61% ) [66.24%]
  139,497     L1-icache-load-misses                             ( +-  6.17% ) [63.29%]
```

```
0.107159363 seconds time elapsed                                          ( +-  0.92%
)
```

`$ perf record -F 12500 -e cache-misses ./main_origin && perf report`

```
 62.36%  main_origin  libc-2.19.so       [.] __strcasecmp_l_avx
 16.31%  main_origin  [kernel.kallsyms]  [k] clear_page_c_e
 12.87%  main_origin  main_origin        [.] findName
  4.54%  main_origin  [kernel.kallsyms]  [k] mem_cgroup_try_charge
  0.60%  main_origin  libc-2.19.so       [.] __strcasecmp_avx
  0.57%  main_origin  [kernel.kallsyms]  [k] copy_user_enhanced_fas
t_string
```

優化版本

程式碼：embedded-summer2015/phonebook (Github)

> 我的筆電 level 1 cache 有 32 kbit，struct 的大小有 136 bytes，32 * 1024 / (136*8) = 30.12，若把整個 __PHONE_BOOK_ENTRY 的 struct 都丟進 findName() 尋找，即使未考慮電腦中其他正在運行的程式，我的 level 1 cache 最多也只能放 30 個entry，一共有 35 萬個單字，必定會發生頻繁的 cache miss。

查看電腦 cache 大小 `$ lscpu | grep cache`

```
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              3072K
```

第一種優化方式：使用體積較小的 struct

根據題目要求，我們只需要知道「有沒有找到 last name」，對於 email、phone number、address、zip 等等資訊是可以在搜尋過程中忽略不看。另外我又希望能不改變原本 phonebook entry 的結構，所以我另
外設計一個 struct 只儲存 last name，並用一個指向 phonebook entry 的指標叫 *detail 來儲存詳細資訊，新的 struct 大小只有 32 bytes，這樣搜尋的過程中，cache 就可以塞進 (32 * 1024) / (32*8) = 128 個單字，增加 cache hit 的機率。

在實作 appendOptimal() 的過程中，*detail 並沒有沒指向一塊空間，我想專心讓 appendOptimal() 產生含有 lastName 的節點就好。為什麼呢？因為若在 append 過程中 malloc() 空間給 *detail，會增加很多 cache miss，嚴重影響到效能表現，經過實

測，總體效能甚至比原始版本還差一點。目前想法是將 *detail 的 assign （當有需要時）交給另外一個 function 處理，畢竟我們一開始只有 last name 的資訊。

```
typedef struct __LAST_NAME_ENTRY{
    char lastName[MAX_LAST_NAME_SIZE];
    entry *detail;
    struct __LAST_NAME_ENTRY *pNext;
} lastNameEntry;
```

`$ perf stat -r 10 -e cache-misses,cache-references,L1-dcache-load-misses,L1-dcache-store-misses,L1-dcache-prefetch-misses,L1-icache-load-misses ./main_optimal`

```
size of entry : 32 bytes
  uninvolved  is found!
      zyxel  is found!
  whiteshank  is found!
  odontomous  is found!
 pungoteague  is found!
  reweighted  is found!
 xiphisternal  is found!
   yakattalo  is found!
execution time of appendOptimal() : 0.044819
execution time of findNameOptimal() : 0.023803

Performance counter stats for './main_optimal' (10 runs):
486,127    cache-misses      #   60.472 % of all cache refs    ( +-  0.74% ) [65.67%]
803,882    cache-references                                    ( +-  0.37% ) [65.65%]
2,531,398  L1-dcache-load-misses                               ( +-  1.51% ) [66.70%]
329,753    L1-dcache-store-misses                              ( +-  1.70% ) [68.36%]
1,642,925  L1-dcache-prefetch-misses                          ( +-  2.18% ) [69.27%]
100,133    L1-icache-load-misses                               ( +-  6.45% ) [67.15%]
      0.071458929 seconds time elapsed                         ( +-  0.84% )
```

`$ perf record -F 12500 -e cache-misses ./main_optimal && perf report`

```
  36.16%  main_optimal  [kernel.kallsyms]  [k] clear_page_c_e
```

```
  29.01%  main_optimal  libc-2.19.so       [.] __strcasecmp_l_avx
  10.71%  main_optimal  [kernel.kallsyms]  [k] mem_cgroup_try_char
ge
   7.20%  main_optimal  main_optimal       [.] findNameOptimal
   4.04%  main_optimal  [kernel.kallsyms]  [k] copy_user_enhanced_f
ast_string
   2.07%  main_optimal  libc-2.19.so       [.] __strcasecmp_avx
   1.17%  main_optimal  [kernel.kallsyms]  [k] __rmqueue
```

第二種優化方式：使用 hash function

第一種方式已經見到不錯的效能成長了！findName() 的 CPU time 從 `0.050462` 變成 `0.023803`，進步一倍以上。另外 cache miss 的次數從 `3,519,048` 降到 `486,127`！

不過 35 萬個單字量畢竟還是太大，如果利用字串當作key，對 hash table 作搜尋顯然比每次從頭開始查找快多了。所以接下來的問題就是，如何選擇 hash function？如何減少碰撞發生呢？我選了兩種 hash function。不過以下先用 hash2() 和上面的結果作比較。另外 table size 我使用 42737，挑選的原因是因為有約35萬個英文單字，為了減少碰撞機會，挑了一了蠻大的「質數」。

`$ perf stat -r 10 -e cache-misses,cache-references,L1-dcache-load-misses,L1-dcache-store-misses,L1-dcache-prefetch-misses,L1-icache-load-misses ./main_hash`

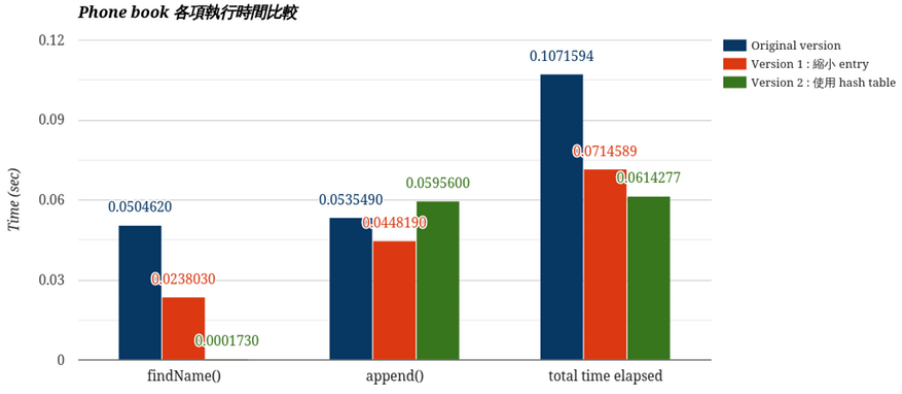
```
hash table size (prime number) : 42737
size of entry : 32 bytes
  uninvolved is found!
      zyxel is found!
  whiteshank is found!
  odontomous is found!
 pungoteague is found!
  reweighted is found!
xiphisternal is found!
   yakattalo is found!
execution time of appendHash() : 0.059560
execution time of findNameHash() : 0.000173

Performance counter stats for './main_hash' (10 runs):

    367,138      cache-misses      #   60.950 % of all cache refs     ( +-
 1.78% ) [61.51%]
    602,362      cache-references                                     ( +-  1.62% ) [6
```

```
6.17%]
887,342      L1-dcache-load-misses                     ( +-  1.20% )
[71.61%]
694,774      L1-dcache-store-misses                    ( +-  0.92% )
[73.62%]
 77,931      L1-dcache-prefetch-misses                 ( +-  2.12%
) [68.81%]
110,773      L1-icache-load-misses                     ( +-  8.71% )
[62.00%]
   0.061427658 seconds time elapsed                    ( +-  1.46%
)
```

`$ perf record -F 12500 -e cache-misses ./main_hash && perf report`

```
 48.01%  main_hash  [kernel.kallsyms]  [k] clear_page_c_e
 13.66%  main_hash  libc-2.19.so       [.] _IO_getline_info
 12.53%  main_hash  [kernel.kallsyms]  [k] mem_cgroup_try_charge
  6.35%  main_hash  [kernel.kallsyms]  [k] copy_user_enhanced_fast
_string
  1.66%  main_hash  libc-2.19.so       [.] __memcpy_sse2
  1.50%  main_hash  [kernel.kallsyms]  [k] __rmqueue
  1.36%  main_hash  libc-2.19.so       [.] memchr
  1.33%  main_hash  [kernel.kallsyms]  [k] alloc_pages_vma
  1.20%  main_hash  main_hash          [.] appendHash
  0.97%  main_hash  [kernel.kallsyms]  [k] __alloc_pages_nodemask
```

執行時間分析



- findName()：本次測試需要查找 **8** 個 last name，兩種優化版本顯然都比原始的還要好，hash 版本不用每次從 35 萬個單字的頭開始找起省掉許多時間，而且再配上縮小過後的 entry，所以 L1 cache 可以塞進更多的 last name，兩種優化方式一加乘，查找時間比原始快了近 300 倍！

- append()：先來比較 原始版本 和 第一個版本，在 append() 的算法上這兩者幾乎一模一樣唯一的差別就是 malloc() 的記憶體空間不一樣大。

```
// entry *append(char lastName[], entry *e)
e->pNext = (entry *) malloc(sizeof(entry));  // 136 bytes

// lastNameEntry *appendOptimal(char lastName[], lastNameEntry *ln
e)
lne->pNext = (lastNameEntry *) malloc(sizeof(lastNameEntry)); // 32
 bytes
```

寫了一個獨立程式來產生 350,000 個 entry，測試這兩者的 page-faults，cache-misses，結果如下。兩者時間相差約 0.0105 sec，和上面 0.00873 sec 差距不大。要動態分配一個大的記憶體空間，它可能產生的效能損失是可觀的。所以如果已知記憶體需求，盡量一開始就分配大塊一點的空間，就如同開學考那題 malloc 2D array 一樣。

	malloc_entry()	malloc_last_name_entry()
malloc size	136 bytes	32 bytes
page-faults	12353 times	4150 times
cache-misses	855167 times	289665 times
instructions	103354196 times	83878860 times
time elapsed	0.031205	0.020662

最後，同樣都是分配 32 bytes 大小的 entry，hash 版本的 append() 時間，比起第一種版本還多了 0.01474 秒，這是可預期的， 因為為了將 last name 放到對應 bucket，還需要多一步 hash 運算。

cache miss 情形

	Original version	Version 1 : 縮小 entry	Version 2 : 使用 hash table		
cache-misses	3,519,048	486,127	367,138		第一名
L1-dcache-load-misses	4,185,434	2,531,398	887,342		
L1-dcache-store-misses	1,005,501	329,753	694,774		第二名
L1-dcache-prefetch-misses	3,088,038	1,642,925	77,931		
cache-references	3,629,269	803,882	602,362		第三名
cache-misses of all cache refs	96.963 %	60.472 %	60.95 %		

perf 能偵測的 cache miss 種類非常多，基本上分為 load-miss、store-miss、prefetch-miss 三種，而再根據 cache 種類又分 Level 1 cache 和 Last Level cache，instruction cache 和 data cache，TLB cache。

我們最關心的是 `L1-dcache-load-misses`，也就是要尋找的 data 不在 Level 1 cache，要往更下層的 memory (我的電腦有 L2、L3)。經過縮小 entry後，version 1 的 `L1-dcache-load-misses` 減少了 0.4 倍，而 hash 版本則因為查找的數量從 242.6 萬次降到不到

20 次，如下表。進而讓 cache miss 再更減少了近 0.8 倍！不過當然這樣的成果要歸功於一開始辛苦建立 Hash table。

	Original version	Hash, bucket = 42737
uninvolved	323494	1
zyxel	349891	1
whiteshank	341124	1
odontomous	214467	7
pungoteague	247445	1
reweighted	258854	6
xiphisternal	345128	1
yakattalo	345913	1
total	2426316	19

不同 Table size 對效能的影響

Hash function 裡常會對字串作很多次的加法或乘法，以這邊 hash2() 來說除了加法還有乘32，如果我取的不是質數，剛好是某些數字的倍數，譬如 2 或 3 等等，mod 之後就會很容易往某個幾個 bucket 集中，若很不巧我找的字剛好在這幾的 bucket 裡，平均來講效能就不好了，list會很長，所以 table size 我會選用質數。

另外一件事情是， Table size 使用 42737，選用這麼大的數字的考量在 data set 有 35 萬個，size 越大的話，每個 bucket 的 linked list 才不會太長，findName() 起來才會快。

就前面結果我們已經知道 hash 版本速度很快，cache miss 也很低。剩下能主導效能就是查找次數了，以下列出幾個 size 的結果。而當 Table size 大於 7919 時，其查找的次數差異已經非常小了。

# of prime number	10	100	1000	2000	5000	8000	10000
prime number	29	541	7919	17389	48611	81799	104729
uninvolved	901	61	2	18	1	2	1
zyxel	2	1	1	1	1	1	1
whiteshank	278	21	2	1	1	1	1
odontomous	4793	296	27	6	3	1	1
pungoteague	3667	184	14	6	3	2	2
reweighted	3257	166	15	10	7	6	5
xiphisternal	163	12	2	2	1	1	1
yakattalo	145	9	1	1	1	1	1
Total	13206	750	64	45	18	15	13

不同的 hash function 對效能的影響

hash2() 是有名的 djb2，兩者差別是 `hashVal << 5`，也就是 hashVal * 32的意思。ASCII 字元的數值最大為 127 的整數，不管是人名還是英文單字當作輸入，平均鍵值長度大約 6～10，所以 hashVal 最多也只有 1270 種可能，**不是一個均勻分佈，而在效能表現上可能就有很大的差異**。

```
hashIndex hash1(char *key, hashTable *ht)
{
    unsigned int hashVal = 0;
    while(*key != '\0'){
        hashVal+= *key++;
    }
    return hashVal % ht->tableSize;
}

hashIndex hash2(char *key, hashTable *ht)
{
    unsigned int hashVal = 0;
    while(*key != '\0'){
        hashVal = (hashVal << 5) + *key++;
    }

    return hashVal % ht->tableSize;
}
```

`$ ./main_hash ` with hash1()

```
hash table size (prime number) : 42737
  uninvolved is found! n = 60
      zyxel is found! n = 1
  whiteshank is found! n = 12
  odontomous is found! n = 143
 pungoteague is found! n = 192
  reweighted is found! n = 133
 xiphisternal is found! n = 3
   yakattalo is found! n = 8
```

==> 共 552 次

`$ ./main_hash ` with hash2()

```
hash table size (prime number) : 42737
size of entry : 32 bytes
  uninvolved is found! n = 1
      zyxel is found! n = 1
  whiteshank is found! n = 1
  odontomous is found! n = 7
 pungoteague is found! n = 1
  reweighted is found! n = 6
 xiphisternal is found! n = 1
   yakattalo is found! n = 1
```

==> 共 19 次

Jakub Jelinek 針對不同的 hash function 所作的分析:

The number of collisions in the 537403 symbols is:

name	2sym collision #	3sym collision #	more than 3sym collision #
sysv	1749	5	
libiberty	42		
dcache	37		
djb2	29		
sdbm	39		
djb3	31		
rot	337	39	61
sax	34		
fnv	40		
oat	30		

==> 應用於 GNU Hash ELF Sections，加速動態函式庫的載入時間
==> 延伸閱讀: Optimizing Linker Load Times
==> 完整的測試程式 phonebook

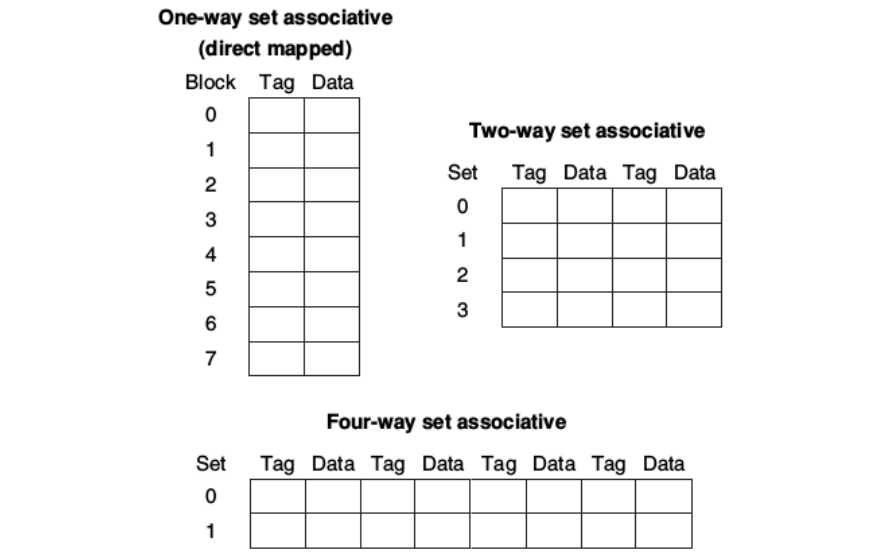
(16:25-17:40)

Cache 背景知識

- 原理：cache利用**Temporal Locality**和**Spatial Locality**設計記憶體架構的兩大原則

CHIA-CHI H *https://www.cs.umd.edu/class/fall2001/cmsc411/proj01/cache/matrix.html*

- set associative：set associative的方式，也就是把cache分成多個set，CPU必須檢查指定set內的每個block是否有可用的cache。最極端的情況下就是Fully Associative，也就是CPU要檢查cache內所有的block。



- 實作多個four-way set的方式

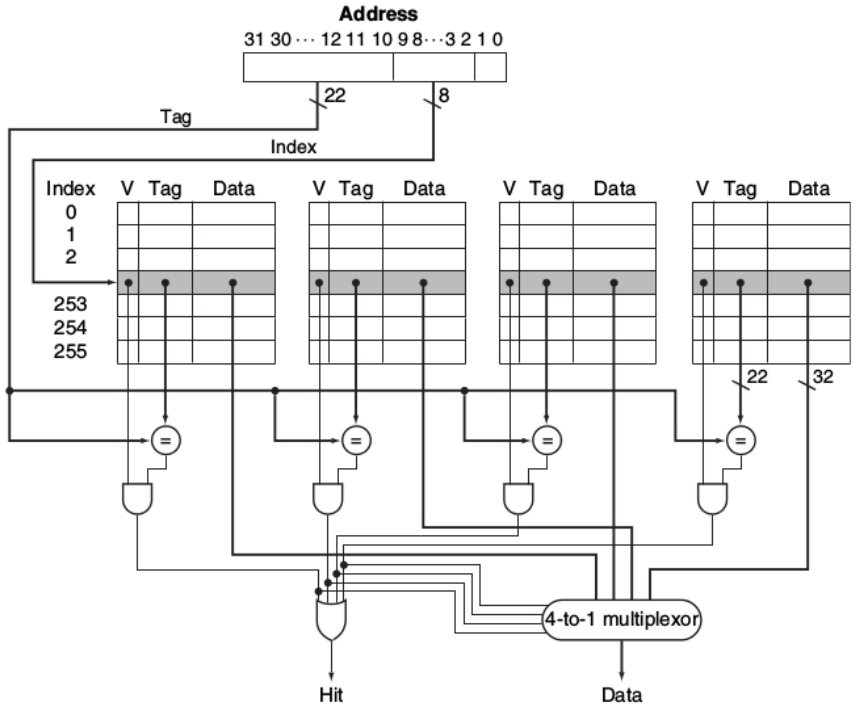


FIGURE 5.18 The implementation of a four-way set-associative cache requires four comparators and a 4-to-1 multiplexor. The comparators determine which element of the selected set (if any) matches the tag. The output of the comparators is used to select the data from one of the four blocks of the indexed set, using a multiplexor with a decoded select signal. In some implementations, the Output enable signals on the data portions of the cache RAMs can be used to select the entry in the set that drives the output. The Output enable signal comes from the comparators, causing the element that matches to drive the data outputs. This organization eliminates the need for the multiplexor.

更多內容如下：

- 資料來源：http://enginechang.logdown.com/tags/cache
- http://www.cs.iit.edu/~virgil/cs470/Book/chapter9.pdf
- http://www.mouseos.com/arch/cache.html

Cache Miss的計算

假設硬體規格如下:

```
Handle 0x000A, DMI type 7, 19 bytes
Cache Information
        Socket Designation: L1 Cache
        Configuration: Enabled, Not Socketed, Level 1
        Operational Mode: Write Through
        Location: Internal
        Installed Size: 32 kB
        Maximum Size: 32 kB
        Supported SRAM Types:
                Unknown
        Installed SRAM Type: Unknown
        Speed: Unknown
        Error Correction Type: Parity
        System Type: Data
        Associativity: 8-way Set-associative
```

☐ 條件：

- cache總大小是32KByte
- 一個cache的block為64Byte
- 一個entry的大小為(136byte)+memory control block(8byte) = 144byte
- 8 way set associative

☐ 計算：

- 有多少block?
  (我考慮main.c findName() & append 這2個function 因為 linked-list 跟搜尋存取花最多時間)
  350000筆資料*每筆144byte=5040000byte
  5040000byte / 64byte * 2次function=1574000block
  讀一個block是一次 reference

```
Performance counter stats for './phonebook_orig' (5 runs):

       1,969,004      cache-misses              #   93.204 % of all cache refs
       2,098,272      cache-references
     263,055,766      instructions              #    1.34  insns per cycle
     200,199,947      cycles

     0.069918325 seconds time elapsed                                          ( +-  3.72% )
```

  可以看到1574000量級和2098272相同
- cache有幾個set？
  cache 32KByte/block 64Byte = 128個block
  128 / 8way set = 16個
- 一個cache line可以存多少筆entry
  144byte / 64byte = 2.25　所以是3個block
  已知是8way set所以可以存2個
- cache miss
  1574000(總block)/ 3(一個entry 需要3 block) = 524667組 entry
  524667 / 16(index數) = 32790(tag數 ，填到相同index的個

數)
由上知一個cache line最多可以放2組entry，所以有兩次機會
32790/2=16395(可能被對到次數)
16395*16(index數)*3(entry佔的block數) = 786960(cache hit 次數)
1574000-78960 = 1495040(cache miss)
1495040/1574000 = 94%(miss rate)

JIM H *修改 Makefile，在 CFLAGS 加上 `-DNDEBUG`，降低 assert() 帶來的影響，重新編譯、執行，再用 perf 分析效能*

Hash Function

```
unsigned int hash(hash_table *hashtable , char *str)
  {
    unsigned int hash_value = 0;
    while(*str)
      hash_value = (hash_value << 5) - hash_value + (*str++);
    return (hash_value % SIZE);
  }
```

(2^n)-1 = X<<n - X

BKDR hash function

- 特色：計算過程中有seed參與，seed為31 131 1313....
- 出處： “The C Programming Language"Brian W. Kernighan, Dennis M. Ritchie的 Table Lookup章節

```
unsigned int BKDRHash(char *str)
{
  unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
  unsigned int hash = 0;

  while (*str)
  {
    hash = hash * seed + (*str++);
  }

  return hash;
}
```

- 為甚麼要乘上一個係數(seed)？
  - 若只單純使用各字母的ASCII相加得到hash值，碰撞率很高

        - 例如： a(97)+d(100) = b(98)+c(99)
    - 所以可以再乘上一個係數，讓hash值差距更大
- 為甚麼係數(seed)要取31 131 1313 13131.... ?
    - 把係數分為三種探討： 偶數(2的次方)、 偶數(非2的次方)、奇數
        - **偶數(2的次方)**->取seed=32
            - 兩字串：abhijklmn abchijklmn(後者多一個c)分別帶入上面程式碼，結果為：

              abhijklmn = 3637984782

              abchijklmn = 3637984782
            - 兩字串的hash值一模一樣，且把其中一個字串加長為abcdehijklmn，hash值仍沒有改變，為甚麼？
                - 計算時若overflow變會拋棄最高位，在hijklmn的前面加上多長的字串，其值仍為3637984782
        - 偶數(非二的次方)
            - 可以推論其在使用上結果應與上例相同，當結果overflow時，就會捨棄最高位，兩字串的hash值會相同
        - 奇數->取seed=31
            - 31=2^5-1，即使字串長到會overflow，但後面的-1仍會影響到整體的hash值

量化分析
=> 2016q1 Homework 1

Computer Systems: A Programmer's Perspective

- CMU 官方網站
- 推薦閱讀: (和本課程高度相關)
    - Machine-Level Representation of Programs
    - Processor Architecture
    - Optimizing Program Performance
    - The Memory Hierarchy
    - Linking
    - Exceptional Control Flow
    - Virtual Memory
    - System-Level I/O
    - Concurrent Programming
- Modern Microprocessors A 90 Minute Guide! (必讀)

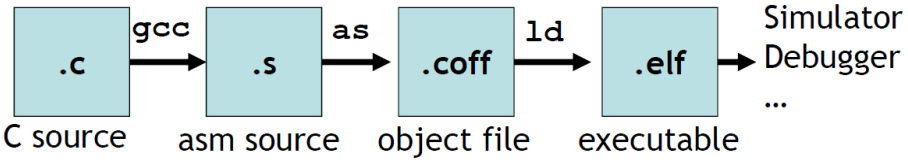
GNU Toolchain

- gcc : GNU compiler collection
- as : GNU assembler
- ld : GNU linker
- gdb : GNU debugger

介紹完編譯工具後，來講點大概編譯的流程還有格式

Compile flow

先來看這張圖：



.c 和 .s 我想大家比較常見，所以就解釋一下 .coff 和 .elf 是什麼:

- COFF (common object file format) : 是種用於執行檔、目的碼、共享函式庫 (shared library) 的檔案格式
- ELF (extended linker format) : 現在最常用的文件格式，是一種用於執行檔、目的碼、共享函式庫和核心的標準檔案格式，用來取代COFF

GAS program format (AT&T)

```
.file “test.s”
.text
.global main
.type main, %function
main:
   MOV R0, #100
   ADD R0, R0, R0
   SWI #11
.end
```

由一個簡短的 code 來介紹，在程式的section 會看到 `.` ，是定義一些 control information，如 *.file* , *.global* 等

- %function : 是一種 type 的表示方式 .type 後面可以放 function 或者是 object 來定義之
- SWI #11 : 透過 software interrupt 去中斷現有程式的執行，通常會存取作業系統核心的服務 (system call)
- .end : 表示是 the end of the program

注意，在後來的 ARM 中，一律以 "SVC" (Supervisor Call) 取代 "SWI"，但指令編碼完全一致

==> ARM Instruction Set Quick Reference Card

以下簡介 ELF 中個別 section 的意義： (注意: ELF section 的命名都由 `.` 開頭)

- .bss : 表示未初始化的 data，依照定義，系統會賦予這些未初始化的值 0
- .data : 表示有初始過的 data
- .dynamic : 表示 dynamic linking information
- .text : 表示 "text" 或者是 executable instructions

寫程式的要點

- 程式是拿來實現一些演算法 (想法) 進而去解決問題的
- 可以透過 Flow of control 去實現我們的程式
  - Sequence
  - Decision: if-t
  - hen-else, switch
  - Iteration: repeat-until, do-while, for
- 將問題拆成多個更小而且方便管理的單元 (每一個單元或稱 function ，盡量要互相獨立)
- Think: In C, for / while ?

Procedures

來複習一下名詞

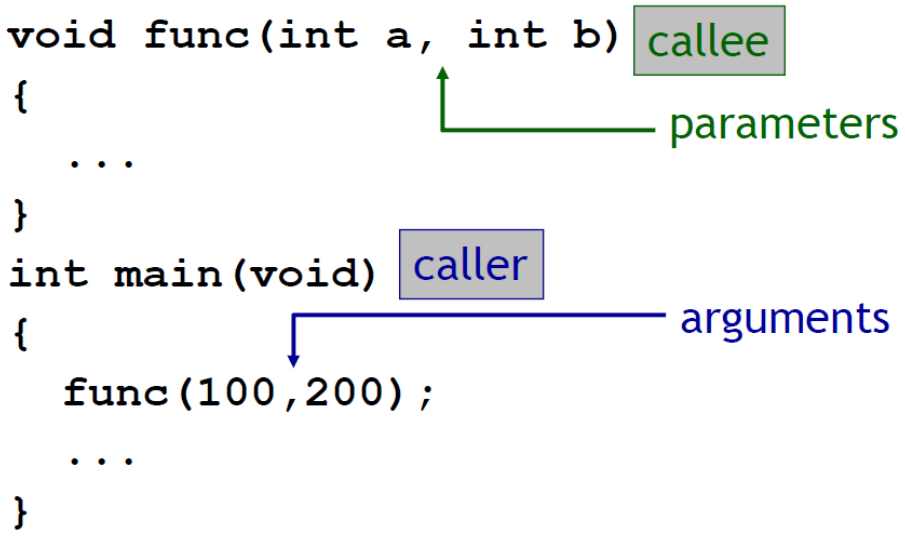
- Arguments: expressions passed into a function
- Parameters: values received by the function
- Caller & callee [呼叫者(通常就是在其他函式呼叫的function) 與 被呼叫者(被呼叫的 function)]

==> "argument" 和 "parameter" 在中文翻譯一般寫「參數」或「引數」，常常混淆

==> "argument" 的重點是「傳遞給函式的形式」，所以在 C 語言程式寫 `int main(int argc, char *argv[])` 時，我們稱 argc 是 argument count，而 argv 是 argument vector

==> "parameter" 的重點是「接受到的數值」，比方說 C++ 有 parameterized type，就是說某個型態可以當作另外一個型態的「參數」，換個角度說，「型態」變成像是數值一樣的參數了。

==> https://en.wikipedia.org/wiki/Parameter_(computer_programming)



在撰寫程式常常會使用呼叫(call)，在上圖中高階語言直接將參數傳入即可，那麼在組語的時候是如何實作的呢?是透過暫存器? Stack？ memory ? In what order ?我們必須要有個 protocol 來規範

CHIA-CHI H *ABI*

https://en.wikipedia.org/wiki/Application_binary_interface

ARM Procedure Call Standard (AAPCS)

- ARM Ltd. 定義一套規則給 procedure entry 和 exit
  - Object codes generated by different compilers can be linked together
  - Procedures can be called between high-level languages and assembly
- AAPCS 是 Procedure Call Standard for the ARM® Architecture 的簡稱，從 10 年前開始，全面切換到 EABI (embedded ABI)
  - 過去的 ARM ABI 稱為 oabi (old ABI)，閱讀簡體中文書籍時，要格外小心，因為資訊過時
- APCS 定義了
  - Use of registers
  - Use of stack
  - stack-based 資料結構型式
  - argument passing 的機制
    - first four word arguments 傳到 R0 到 R3
    - 剩餘的 parameters 會被 push 到 stack (參數依照反過來的排序丟入堆疊中)
    - 少於 4 個 parameters 的 procedure 會比較有效率

Register	APCS name	APCS role
0	a1	Argument 1 / integer result / scratch register
1	a2	Argument 2 / scratch register
2	a3	Argument 3 / scratch register
3	a4	Argument 4 / scratch register
4	v1	Register variable 1
5	v2	Register variable 2
6	v3	Register variable 3
7	v4	Register variable 4
8	v5	Register variable 5
9	sb/v6	Static base / register variable 6
10	sl/v7	Stack limit / register variable 7
11	fp	Frame pointer
12	ip	Scratch reg. / new sb in inter-link-unit calls
13	sp	Lower end of current stack frame
14	lr	Link address / scratch register
15	pc	Program counter

* 被用來傳遞前 4 個 parameters
* Caller-saved if nessary

* Register variables, must return unchanged
* Callee-saved

* 特殊功用的 registers
* 如果存取正確，可用來當作 temporary variables

- Return value
  - one word value 存在 R0
  - 2 ~ 4 words 的 value 存在 (R0-R1, R0-R2, R0-R3)

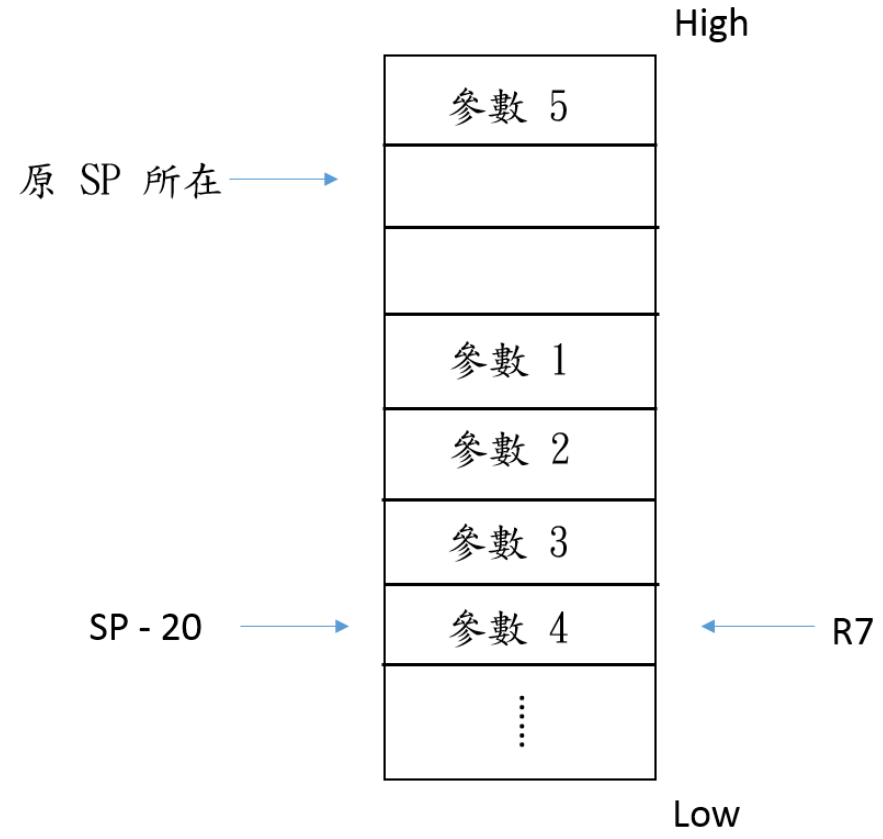
以下測試一個參數數量為 4 和 5的程式:

```
int add4(int a, int b, int c , int d){
   return a + b + c + d;
}

int add5(int a, int b, int c , int d, int e){
   return a + b + c + d + e;
}
```

程式編譯後，用 objdump 組譯會得到類似以下:

```
000103c0 <add4>:
   103c0:       b480            push    {r7}
   103c2:       b085            sub     sp, #20
   103c4:       af00            add     r7, sp, #0
   103c6:       60f8            str     r0, [r7, #12]
   103c8:       60b9            str     r1, [r7, #8]
   103ca:       607a            str     r2, [r7, #4]
   103cc:       603b            str     r3, [r7, #0]
   103ce:       68fa            ldr     r2, [r7, #12]
   103d0:       68bb            ldr     r3, [r7, #8]
   103d2:       441a            add     r2, r3
   103d4:       687b            ldr     r3, [r7, #4]
   103d6:       441a            add     r2, r3
   103d8:       683b            ldr     r3, [r7, #0]
   103da:       4413            add     r3, r2
   103dc:       4618            mov     r0, r3
   103de:       3714            adds    r7, #20
   103e0:       46bd            mov     sp, r7
   103e2:       f85d 7b04       ldr.w   r7, [sp], #4
   103e6:       4770            bx      lr

000103e8 <add5>:
   103e8:       b480            push    {r7}
   103ea:       b085            sub     sp, #20
   103ec:       af00            add     r7, sp, #0
   103ee:       60f8            str     r0, [r7, #12]
   103f0:       60b9            str     r1, [r7, #8]
   103f2:       607a            str     r2, [r7, #4]
   103f4:       603b            str     r3, [r7, #0]
   103f6:       68fa            ldr     r2, [r7, #12]
   103f8:       68bb            ldr     r3, [r7, #8]
   103fa:       441a            add     r2, r3
   103fc:       687b            ldr     r3, [r7, #4]
   103fe:       441a            add     r2, r3
   10400:       683b            ldr     r3, [r7, #0]
   10402:       441a            add     r2, r3
   10404:       69bb            ldr     r3, [r7, #24]
   10406:       4413            add     r3, r2
   10408:       4618            mov     r0, r3
   1040a:       3714            adds    r7, #20
   1040c:       46bd            mov     sp, r7
   1040e:       f85d 7b04       ldr.w   r7, [sp], #4
   10412:       4770            bx      lr
```

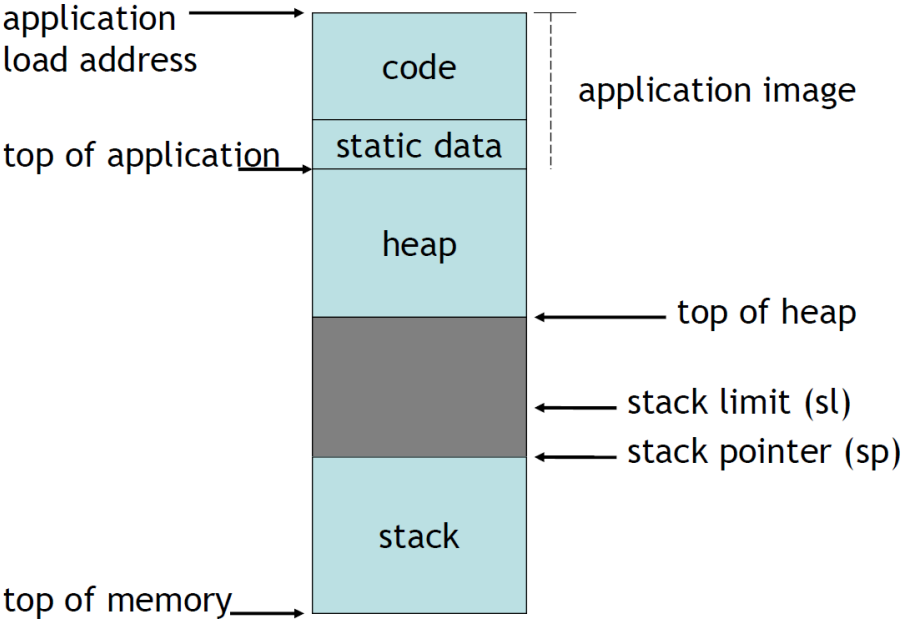
紅框標注的是比左邊多出的程式碼，從這裡可以看到參數 1-4 是存在 R0-R3，而第 5個參數存在原本 sp + 4 的位置，隨著程式碼進行 R0-R3 存在 stack 中，圖下為 stack 恢復前的樣子:



因此若寫到需要輸入5個或以上的參數時，就必須存取外部記憶體，這也導致效能的損失。
==> xorg xserver 的最佳化案例

Standard ARM C program address space

下圖為 ARM C program 標準配置記憶體空間的概念圖:



Accessing operands

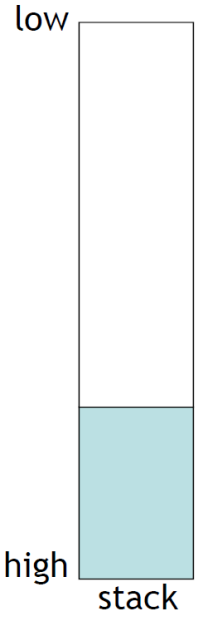
通常 procedure 存取 operands 透過以下幾種方式:

- An argument passed on a register : 直接使用暫存器

- An argument passed on the stack : 使用 stack pointer (R13) 的相對定址 (immediate offset)
- A constant : PC-relative addressing
- A local variable : 分配在 stack 上，透過 stack pointer 相對定址方式存取
- A global variable : 分配在 static area (就是樓上圖片的 static data)，透過 static base (R9) 相對定址存取

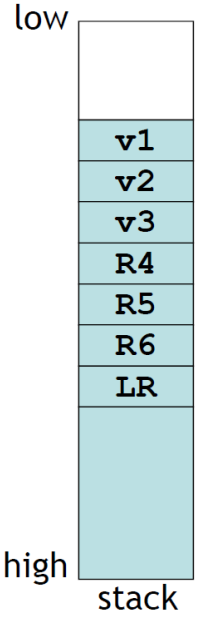
用幾張圖來表現出存取 operands 時，stack 的變化:
圖下為 passed on a register:

```
main:
        LDR    R0, #0
        ...
        BL     func
        ...
```



圖下為存取 local variables:

```
func:   STMFD SP!, {R4-R6, LR}
        SUB   SP, SP, #0xC
        ...
        STR   R0, [SP, #0] @ v1=a1

        ...
        ADD   SP, SP, #0xC
        LDMFD SP!, {R4-R6, PC}
```



Target Triple

在使用 Cross Compiler 時，gcc 前面總有一串術語，例如:

- arm-none-linux-gnueabi-gcc

這樣 `arm-none-linux-gnueabi-` 稱為 target triple，通常規則如下：
<target>[<endian>][-<vender>]-<os>[-<extra-info>]

- vendor 部份只是常見是可以塞 vendor 而已, 但沒有一定塞 vendor 的資訊
- extra-info 部份大部份是在拿來描述用的 ABI 或著是 library 相關資訊

先以常見 x86 的平台為例子：
gcc 下 -v 可以看到以下：

    Target: x86_64-redhat-linux

<target>-<vendor>-<os> 的形式

Android Toolchain:

    Target: arm-linux-androideabi

<target>-<os>-<extra-info>
androideabi : 雖然 Android 本身是 Linux 但其 ABI 細節跟一般 linux 不太一樣

Linaro ELF toolchain:

    Target: arm-none-eabi

<target>-<vender>-<extra-info>
vender = none
extra-info = eabi

Linaro Linux toolchain:

    Target: arm-linux-gnueabihf

<target><endian>-<os>-<extra-info>
extra-info:
eabi: EABI
hf : Hard float, 預設有 FPU

- soft (GPR), softfp (GPR -> float register), hardfp

Linaro big-endian Linux toolchain:

    Target: armeb-linux-gnueabihf

<target><endian>-<vender>-<extra-info>
endian = be = big-endian

Buildroot 2015-02

    Target: arm-buildroot-linux-uclibcgnueabi

<target>-<vender>-<os>-<extra-info>
extra-info:

uclibc 用 uclibc (通常預設是 glibc, uclibc 有些細節與 glibc 不同)
gnu : 無意義
eabi : 採用 EABI

NDS32 Toolchain:

Target: nds32le-elf

<target><endian>-<os>
Andes 家的，應該淺顯易懂不用解釋

由以上眾多 pattern 大概可以歸納出一些事情：
vender 欄位變化很大，os 欄位可不填。若不填的話，通常代表 Non-OS (Bare-metal)

source: http://kitoslab.blogspot.tw/2015/08/target-triple.html

編譯器原理

為何我們要理解編譯器？這是電腦科學最早的思想體系

- Compiler的多元應用
- 從理解動態編譯器 (如 Just-in-Time) 的運作，可一路探索作業系統核心, ABI,效能分析的原理
- 編譯器和最佳化原理篇
  - 以 GNU Toolchain 為探討對象，簡述編譯器如何運作，以及如何實現最佳化
  - C 語言程式如何轉換為機械碼，以及最佳化的空間和限制
  - Compiler Development

GCC 會透過 builtins 來達到最佳化，在 GCC flag -funit-at-a-time 指定時 (-O3 implies)，原本的：

```
sprintf(buf,"%s",str);
```

在 front-end 會被改寫為：

```
strcpy(buf,str);
```

printf 本身就是一個小型的 interpreter，所以 GCC 的作法就是簡化複雜度，這對一般情況下是 正面的，不過涉及 function rewriting 的議題，對系統程式或設計一個作業系統核心來說，就是一個潛在的問題。

[Interpreter, Compiler, JIT from scratch]

- 碎形程式 in C (Brainf*ck 的版本作為 benchmark)

[Virtual Machine Constructions for Dummies]
[A brief history of just-in-time]
[Making an Embedded DBMS JIT-friendly]

值得注意的是，program loader 也很關鍵

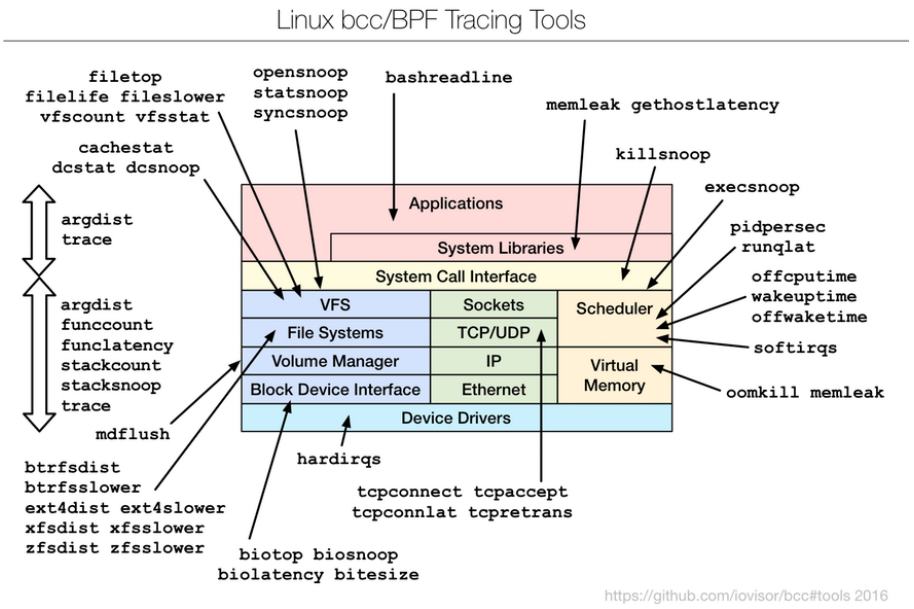
- [ ] PokémonGo Hacking without Jailbreak

作業系統的核心裡面也有編譯器

- A JIT for packet filters: Linux 核心已收錄 Berkeley Packet Filter (BPF) JITC
  - 形式來說來，bpf 就是 in-kernel virtual machine
  - BPF - the forgotten bytecode
- tcpdump 不但可分析封包的流向，連封包的內容也可以進行「監聽」
  - 鳥哥 Linux 私房菜: 第五章、 Linux 常用網路指令
  - Berkeley Packet Filter (BPF) 會透過 bytecode 和 JIT compiler 來執行

```
$ sudo tcpdump -p -ni eth0 -d "ip and udp"
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 5
(002) ldb      [23]
(003) jeq      #0x11            jt 4    jf 5
(004) ret      #65535
(005) ret      #0
```

- Introduction to Linux kernel tracing and eBPF integration
- BPF Compiler Collection (BCC)
  - BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. It makes use of eBPF (Extended Berkeley Packet Filters), a new feature that was first added to Linux 3.15.



實做案例

- jit-construct : Brainf*ck
- rubi : Ruby-like JIT compiler
- 將 Rubi 實做切換到 DynASM，並且設計效能評估機制，從而改善
  - 原本 x86 code generator 換成 DynASM 語法
  - The Unofficial DynASM Documentation
  - 「作業區」(C)
- amacc: Small C Compiler generating ELF executable for ARM architecture

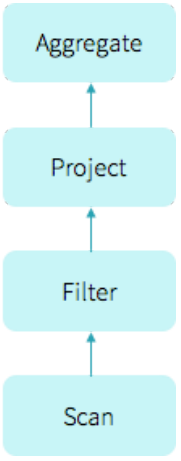
Can we make Apache Spark 10x faster?

好比在金庸《射雕英雄傳》，馬鈺教郭靖修煉內功的方式，無外乎就是一些呼吸、走路、睡覺的法子。

- Apache Spark as a Compiler: Joining a Billion Rows per Second on a Laptop

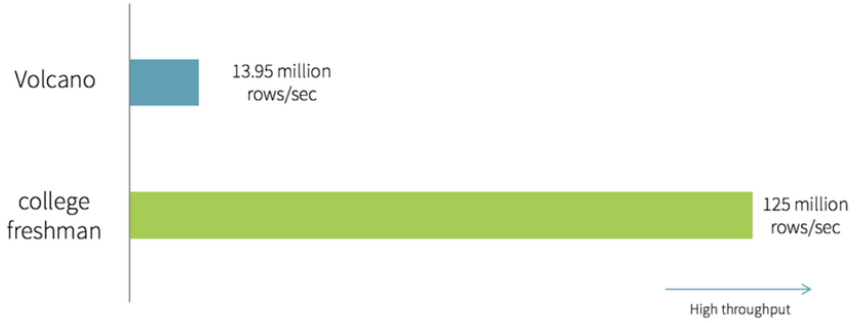
[舊有的資料處理模式]

```
select count(*) from store_sales
where ss_item_sk = 1000
```




```
class Filter(child: Operator, predicate: (Row => Boolean))
  extends Operator {
  def next(): Row = {
    var current = child.next()
    while (current == null || predicate(current)) {
      current = child.next()
    }
    return current
  }
}
```

[以簡馭繁]

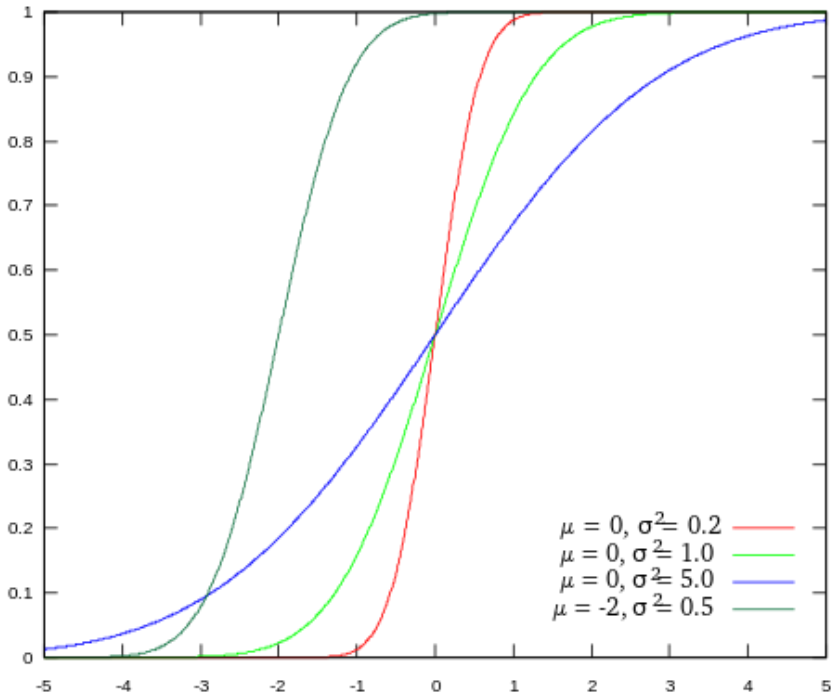



```
var count = 0
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1
  }
}
```

[進一步改善]

- No virtual function dispatches
- Intermediate data in memory vs CPU registers
- Loop unrolling and SIMD

- 生日悖論



編譯器技術對於現行運算需求的重要性

- [ ] AutoFDO, Google
- GCC Function Instrumentation
- 該機制出現於 GCC 2.x，由 Cygnus (現為 RedHat) 所提出，在 GCC 中對應的選項為："finstrument-functions"
- 測試 GCC Function Instrumentation 的小程式： (hello.c)

```
#include <stdio.h>
#define DUMP(func, call) \
        printf("%s: func = %p, called by = %p\n", __FUNCTION__, func,
call)
void __attribute__((__no_instrument_function__))
__cyg_profile_func_enter(void *this_func, void *call_site)
{ DUMP(this_func, call_site); }
void __attribute__((__no_instrument_function__))
__cyg_profile_func_exit(void *this_func, void *call_site)
{ DUMP(this_func, call_site); }

main() { puts("Hello World!"); return 0; }
```

- 編譯與執行：

```
$ gcc -finstrument-functions hello.c -o hello
$ ./hello
__cyg_profile_func_enter: func = 0x8048468, called by = 0xb7e36eb
c
Hello World!
__cyg_profile_func_exit: func = 0x8048468, called by = 0xb7e36ebc
```

- 看到 "__attribute__" 就知道一定是 GNU extension，而前述的 man page 也提到 -finstrument-functions 會在每次進入與退出函式前呼叫 "__cyg_profile_func_enter" 與 "__cyg_profile_func_exit" 這兩個 hook function。等等，「進入」與「退出」是又何解？
- C Programming Language 最經典之處在於，雖然沒有定義語言實做的方式，但實際上 function call 皆以 stack frame 的形式存在。
- 如果我們不透過 GCC 內建函式 "__builtin_return_address" 取得 caller 與 callee 相關的動態位址，那麼仍可透過 -finstrument-functions，讓 GCC 合成相關的處理指令，讓我們得以追蹤。而看到 __cyg 開頭的函式，就知道是來自 Cygnus 的貢獻，在 gcc 2.x 內部設計可瞥見不少。
- 當我們試著移除 "__attribute__((__no_instrument_function__))" 那兩行來看看： (wrong.c)
- 編譯與執行：

  $ **gcc -g -finstrument-functions wrong.c -o wrong**

  $ **./wrong**

  Segmentation Fault
- 發生什麼事情呢？請出 gdb 協助：

  $ **gdb -q ./wrong**

  (gdb) **run**

  Starting program: /home/jserv/HelloWorld/helloworld/instrument/wron
  g

  Program received signal SIGSEGV, Segmentation fault.

  0x0804841d in __cyg_profile_func_enter (this_func=0x8048414, call
  _site=0x804842d) at wrong.c:7

  7 {

  (gdb) **bt**

  #0 0x0804841d in __cyg_profile_func_enter (this_func=0x8048414,
  call_site=0x804842d) at wrong.c:7

  #1 0x0804842d in __cyg_profile_func_enter (this_func=0x8048414,
  call_site=0x804842d) at wrong.c:7

  #2 0x0804842d in __cyg_profile_func_enter (this_func=0x8048414,
  call_site=0x804842d) at wrong.c:7

  ...

  #30 0x0804842d in __cyg_profile_func_enter (this_func=0x8048414,
  call_site=0x804842d) at wrong.c:7

  #31 0x0804842d in __cyg_profile_func_enter (this_func=0x8048414,
  call_site=0x804842d) at wrong.c:7

#32 0x0804842d in __cyg_profile_func_enter (this_func=0x8048414, call_site=0x804842d) at wrong.c:7

- 既然 "__cyg_profile_func_enter" 是 function hook，則本身不得也被施以 "function instrument"，否則就無止盡遞迴了，不過我們也可以發現一件有趣的事情：

  $ **nm wrong | grep 8048414**

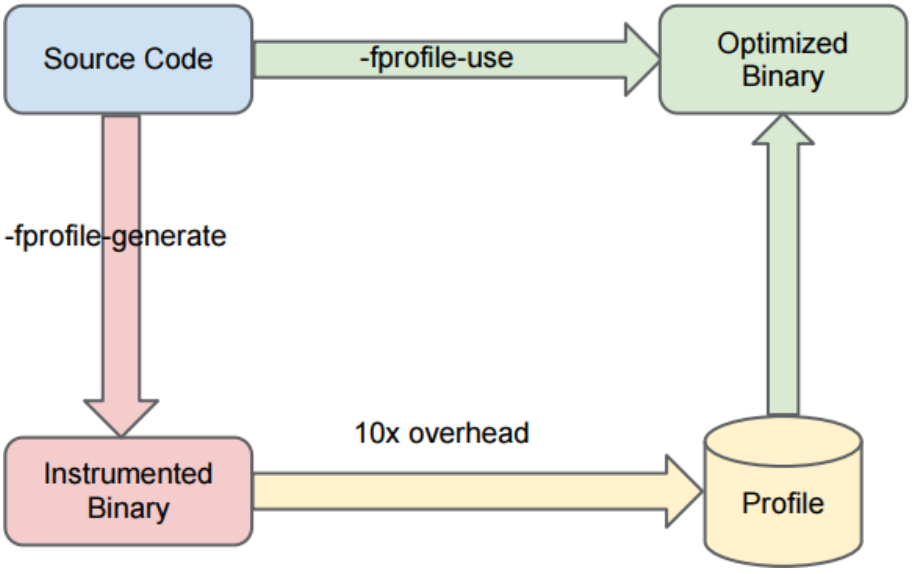
  08048414 T __cyg_profile_func_enter

- 如我們所見，"__cyg_profile_func_enter" 的位址被不斷代入 __cyg_profile_func_enter function arg 中。GNU binutils 裡面有個小工具 addr2line，我們可以該工具取得虛擬位址對應的行號或符號：

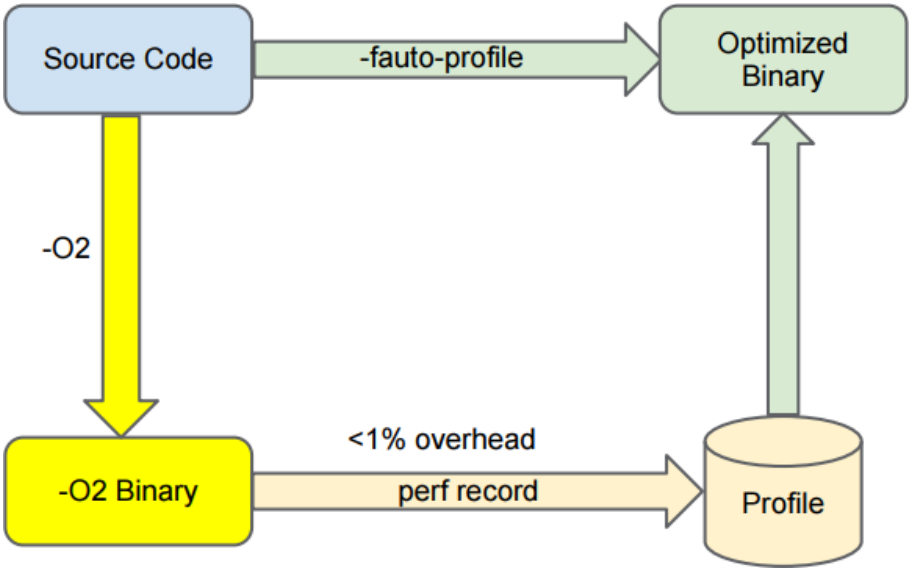
  $ **addr2line -e wrong 0x8048414**

  /home/jserv/HelloWorld/helloworld/instrument/wrong.c:7

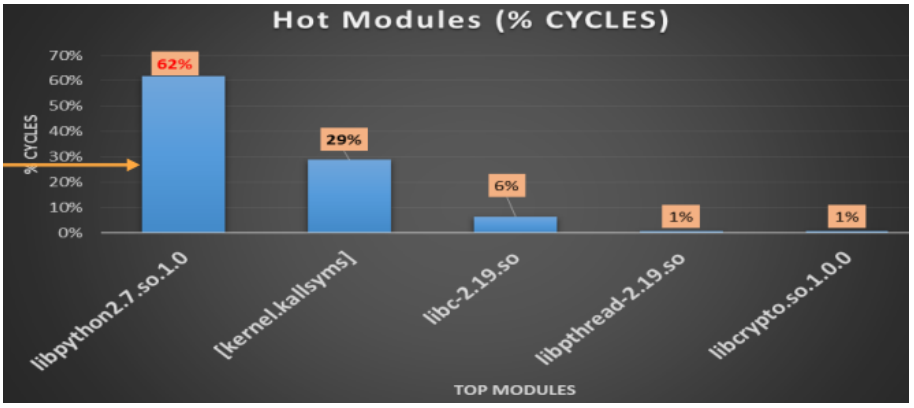
- 就 Linux 應用程式開發來說，我們可透過這個機制作以下應用：
  - 撰寫特製的 profiler
  - 取得執行時期的 Call Graph
  - 置入自製的 signal handler，實做 backtrace 功能
  - 模擬 Reflection 機制
    - Java reflection 參考

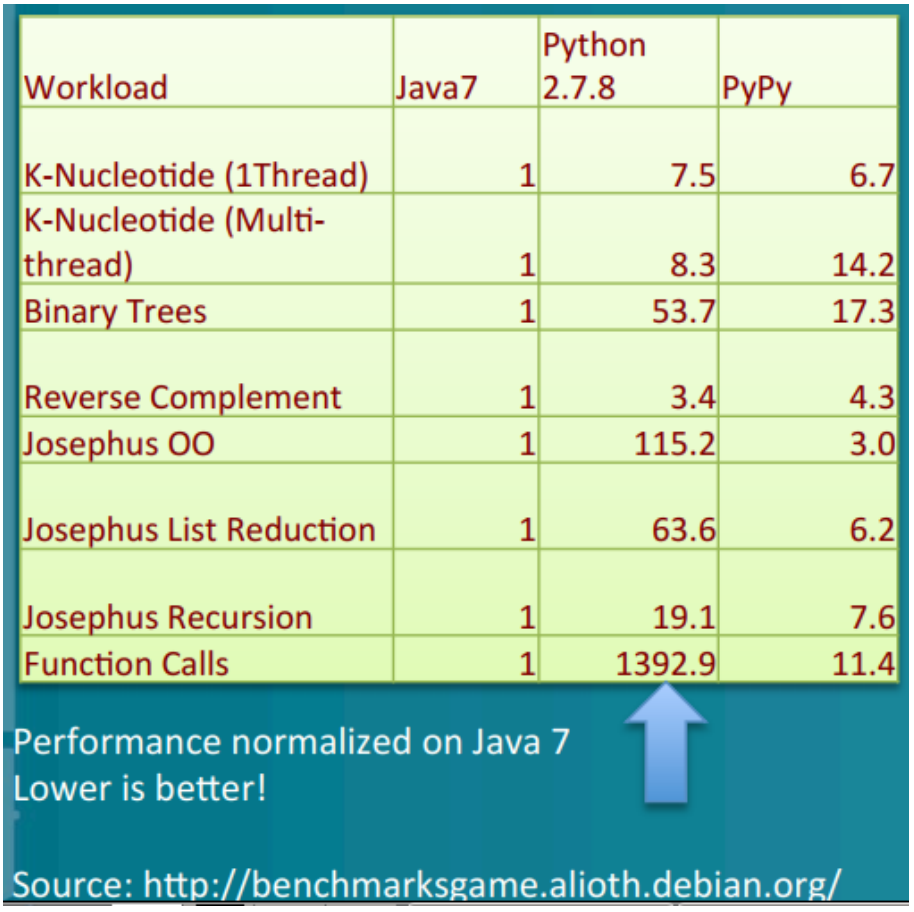


但 profiling 的成本在許多環境，像是雲端運算來說，實在太沈重，所以 Google 提出透過 perf 工具來取得 profiling 資料的 AutoFDO:



- [x] *Accelerating the Core of the Cloud*, *Intel*
  - Core software optimization: a strategy
  - Case Study: OpenStack





- PLT vs. GOT
  - http://refspecs.linuxfoundation.org/ELF/zSeries/lzsabi0_zSeries/x2251.html

Skymizer 針對日前釋出的 GCC 6.1，翻譯部份 GCC Release note:

- 上半段: http://blog.skymizer.com/gcc_6_1/
- 下半段: http://blog.skymizer.com/gcc_6_1_2/

"so you all hate on C++. all right. all right. tell me. does your favorite language have a concept of friendship, then? eh? thought so." from twitter

"C++ Is my favorite garbage collected language because it generates so little garbage."
- Bjarne Stroustrup (origin)

延伸題目: 偵測一個 word
以下的 DETECT 巨集用途為何？

```
#if LONG_MAX == 2147483647L
#define DETECT(X) (((X) - 0x01010101) & ~(X) & 0x80808080)
#else
#if LONG_MAX == 9223372036854775807L
#define DETECT(X) (((X) - 0x0101010101010101) & ~(X) & 0x80808
```

```
08080808080)
#else
#error long int is not a 32bit or 64bit type.
#endif
#endif
```

在測試這個程式時，要注意到由於 **LONG_MAX** 定義在 **limits.h 裡面，因此要記得**將他 include 進去。

這個巨集的用途是在偵測是否為 0 或者說是否為 NULL char **'\0'**，也因此，我們可以在 strlen 的實作中看到這一段，所以為什麼這一段程式碼可以用來偵測 NULL char ?

我們先來看看假設要你實作strlen的情況下你會怎麼作，以下實作一個簡單的版本

```
unsigned int strlen(const char *s)
{
	char *p = s;
	while (*p != '\0') p++;
	return (p - s);
}
```

這樣的版本有什麼問題？雖然看起來精簡，但是因為他一次只檢查 1byte，所以一旦字串很長，他就會處理很久。另外一個問題是，假設是在 32-bit 的 CPU 上，一次是處理 4-byte (32-bit) 大小的資訊，不覺得這樣很浪費嗎?

為了可以思考這樣的程式，我們由已知的計算方式來逆推原作者可能的思考流程，首先先將計算再簡化一點點，將他從 **(((X) - 0x01010101) & ~(X) & 0x80808080)** 變成

((X) - 0x01) & ~(X) & 0x80

還是看不懂.....以前唸書唸過笛摩根定理，將他套用上去，看這樣我們會不會突然迸出好點子，於是這個式子就變成了

~(~(X - 0x01) | X) & 0x80

再稍微調整一下順序

~(X | ~(X - 0x01)) & 0x80

所以我們就可以進行分析

- X | ~(X - 0x01) => 取得最低位元是否為 0 ，並將其他位元設為 1
  - X = 0000 0011 => 1111 1111
  - X = 0000 0010 => 1111 1110
- 想想 0x80 是什麼? 0x80 是 1000 0000 ，也就是 1-byte

的最高位元

上面這兩組組合起來，我們可以得到以下結果

- X = 0 => 1000 0000 => 0x80
- X = 1 => 0000 0000 => 0
- X = 2 => 0000 0000 => 0
- .....
- X = 255 => 0000 0000 => 0

於是我們知道了什麼？原來這樣的運算，如果一個byte是0，那經由這個運算得到的結果會是 0x80，反之為 0。

再將這個想法擴展到 32-bit，是不是可以想到說在 32bit 的情況下，0會得到 0x80808080 這樣的答案?我們只要判斷這個數值是不是存在，就可以找到 '\0' 在哪了！

- 參考資料
  - Hacker's Delight
  - http://www.hackersdelight.org/corres.txt
  - FreeBSD 的 strlen(3)
  - Bug 60538 - [SH] improve support for cmp/str insn
  - CMSC 311 Answers-Draft

應用:

- https://github.com/eblot/newlib/blob/master/newlib/libc/string/strlen.c
- https://github.com/eblot/newlib/blob/master/newlib/libc/string/strcpy.c

隨堂測驗

☐ Quiz-0 : Give you the prototype of the function:
int p(int i, int N);
use it to print the numbers from i up to N and then down to i again, one number per line
Don't use those keywords: do, while, for, switch, case, break, continue, goto, if.
Don't use ':?'
You can only use ONE statement!
==>

☐ Quiz-1 : 只能使用位元運算子和遞迴，在C程式中實做兩個整數的加法
void add(int a, int b) { ... }
=>

- [ ] Quiz-2 : 給定一個 singly-linked list 結構:

```
typedef struct _List {
    struct _List *next;
    int val;
} List;
```

實做得以交換存在 list 中兩個節點的函式: (考慮 HEAD 與 TAIL)

```
int swap_list(List **head, List *a, List *b) {
...
}
```

==>

https://github.com/shengwen1997/sys2016_homework

- Swap & Bubble sort & Test code