

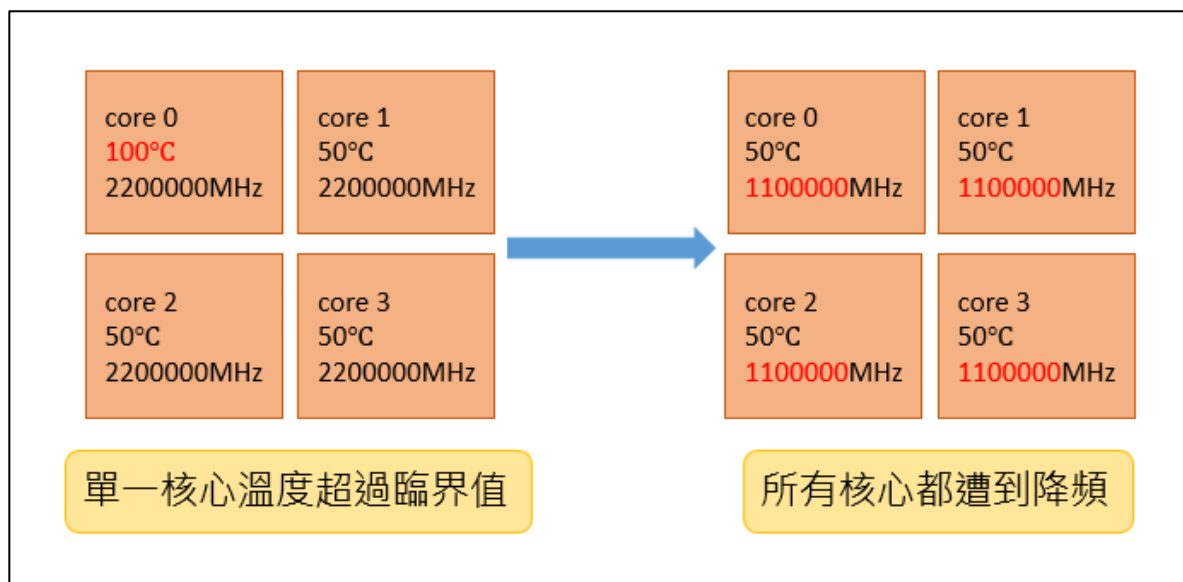
一、摘要

由於處理器(central processing unit, cpu)溫度過高時會影響到系統的效能，因此，如何有效控管核心的溫度以達成較佳的系統效能一直是個熱門的研究議題。在眾多研究當中，Performance-Aware Thermal Management via Task scheduling這篇論文提出一排演算法(ThreshHot scheduler)且聲稱其在單核心的環境下能有效控管核心溫度及提升系統效能。近年來處理器走向多核心的設計，所以我的研究除了驗證該篇論文提出的排程策略(scheduling policy)在單核心的環境下確實有效，還希望知道同樣的排程策略(scheduling policy)是否在多核心的環境下同樣有效？為了達成我的目標，我設計了一個工具程式，它不但可以實現ThreshHot scheduler的排程策略(scheduling policy)，還能夠將Linux作業系統核心(kernel)內部的排程策略(scheduling policy)加以實踐。接著，我分別在單核心及多核心的環境下設計了幾組實驗，然後再利用上述的工具程式來測試不同的排程策略在不同的系統環境下的行為表現並討論之。

二、研究動機

處理器(cpu)在進行運算時，溫度會升高。在這樣的前提下，多核心的系統會產生一個問題：其中一個核心溫度上升的速率最快，所以也最快到達系統規定的溫度臨界值(threshold)而遭到被系統降頻的處理(一旦核心溫度到達系統規定的溫度臨界值(threshold)，該核心就會遭到降頻的處理)。但問題是，通常在多核心的環境下數個核心會共用一個供電系統，所以在同一個供電系統中其他溫度尚未達到溫度臨界值(threshold)的核心一樣會遭到降頻的處理，結果大大的降低了整個系統的執行效能，因為降頻造成能執行的程式指令大大的減少。

舉例來說(示意圖如下面圖(一))，一個多核心系統有四個核心(core)且這四個核心(core)共用同一個供電系統，然後目前每的核心(core)的頻率(frequency)皆為2200000。因為其中核心零號(core 0)正在執行大量的運算，所以它的溫度迅速升高到100°C。假設此多核心系統規定的溫度臨界值(threshold)是99°C，則我們可以知道核心零號(core 0)會遭到降頻(從2200000變成1100000)的處理。不幸的是，由於前一段所敘述的性質，其他三個溫度較低的核心(core)一樣會遭到被系統降頻(從2200000變成1100000)的處理。



圖(一)

解決上述的問題有兩種方法，一是只讓溫度超標的那一個核心遭到降頻的處罰，然後其他溫度沒有超標的核心維持原有的頻率執行程式指令；另外一種方法則是確保各個核心的溫度升高速率相當，如此當核心溫度超標的時候，系統降頻就不會無故懲罰到其他溫度沒有超標的核心。

作業系統多半會實現多種不同的多核心排程器，而每一種排程器有其設計的目的與效果，我想要深入研究、實作並評估各種類型的排程器在多核心處理器上的效果，而其中的效能、耗能及溫度的變化更是我最重要的研究部分。最後希望能提出一個優化Linux核心(kernel)的排程(scheduler)來達成整體效能最佳的目標。

三、文獻探討

為了研究Linux作業系統的耗電與溫度管理，我研讀了許多與Linux作業系統核心排程(kernel scheduling)及與溫度、耗能和效能管理相關的文獻。在讀過的所有文獻當中，Performance-Aware Thermal Management via Task scheduling這篇論文的内容和我的研究動機、目的十分類似，因而引起我極大的興趣。

Performance-Aware Thermal Management via Task scheduling這篇論文指出：在單核心系統的環境下，達到溫度臨界值(threshold)的核(core)會遭到降頻的處理，因此大幅度的降低了整個系統的執行效能，因為降頻造成能執行的程式指令大大的減少。為了解決上述的問題，Performance-Aware Thermal Management via Task scheduling這篇論文提出了一個新的排程演算法(ThreshHot scheduler)，而該排程演算法的目標就是讓核心(core)的溫度不會常常超過系統規定的溫度臨界值(threshold)而遭到降頻的處理。

簡單來說，這個排程演算法(ThreshHot scheduler)的策略(policy)是：總是優先選擇不超過溫度臨界值(threshold)(一旦溫度達到溫度臨界值(threshold)，系統就會遭到降頻的處理)的熱工作(hot jobs)來執行，然後才是選擇冷工作(cold jobs)來執行；但假如每一個熱工作(hot job)皆會超過系統所規定的溫度臨界值(threshold)，則該排程(scheduler)就

會選擇讓最熱的工作(hottest job)最優先被處理器(cpu)執行。(這邊說的熱工作(hot jobs)指的是計算密集型工作(computation intensive jobs)，而冷工作(cold jobs)指的是記憶體使用密集型工作(memory access intensive jobs))。

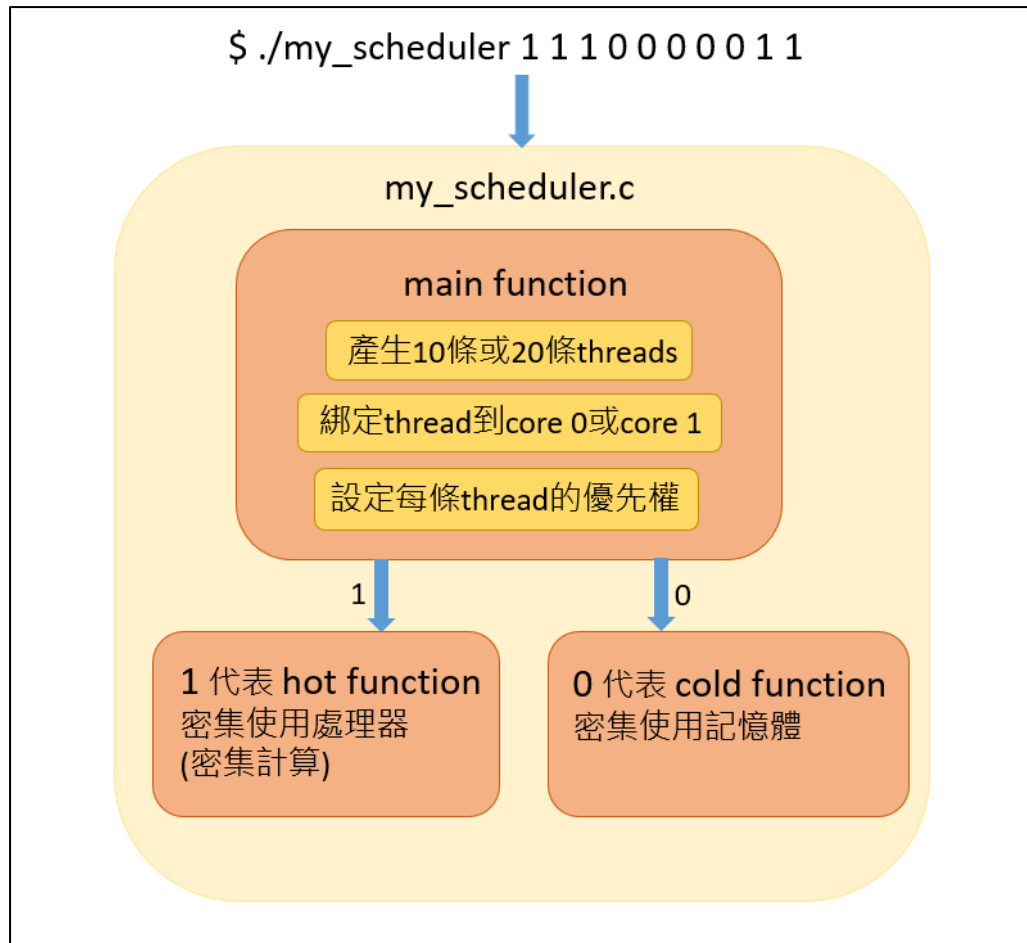
Performance-Aware Thermal Management via Task scheduling這篇論文的結論是，藉由他們提出的排程演算法(ThreshHot scheduler)，單核心系統的核心(core)溫度超過溫度臨界值(threshold)的次數有減少並且系統整體效能亦有提升。但是，因為近年來處理器走向多核心的設計，所以我更想知道的是：是否同樣策略(policy)的排程演算法(ThreshHot scheduler)在多核心系統的環境下依然能達成同樣好的成果？

因此，我設計了一些實驗，希望能證實Performance-Aware Thermal Management via Task scheduling這篇論文所提出的排程演算法(ThreshHot scheduler)其排程策略(policy)(優先讓熱工作(hot job)執行)在多核心系統的環境下，相對於Linux作業系統傳統的核心排程(kernel scheduling)(不刻意去管控溫度(thermal management))，的確能產生較低的系統溫度及較佳的系統效能；但若是兩者的結果其實差不多，那我則是證實了這個排程演算法在多核心系統的環境下是不可行的。

四、研究方法

由於 Linux 核心(kernel)的原始碼(source code)十分龐大且各種不同的功能彼此環環相扣，所以直接修改 Linux 核心(kernel)的原始碼(source code)容易造成無法預期的錯誤，有時甚至會無法開機。所以我設計來進行實驗的工具程式並不會直接動到 Linux 核心(kernel)的原始碼(source code)，而是透過一些由軟硬體所提供的函數庫，也就是 API(Application Programming Interface)來操控系統的行為。而我的實驗方法是盡量讓系統只執行一個進程(process)，然後對由這個進程(process)所產生的多個線程(threads)進行排程。接下來的內容會詳細說明工具程式的功能及實驗環境。

工具程式的功能：



圖(二)

工具程式的軟體架構圖

1. 執行不同順序組合的熱工作(hot job)和冷工作(cold job)

我以指令列(command line)上的 argv 參數來傳遞我想要測試的冷工作(cold job)和熱工作(hot job)的順序。舉例來說，在指令列(command line)輸入「./test 1 1 1 0 0 0 0 1 1」(0 代表要執行冷工作(cold job)，1 代表要執行熱工作(hot job))，則 ./test 對應 argv[0]，第一個 1 對應 argv[1]，第一個 0 對應 argv[4]，其他皆可以此類推。接著，程式會產生 10 條線程(thread)，第一條線程(thread)對應到 argv[1]，所以會去做熱工作(hot job)；相對的，第四條線程(thread)則是對應到 argv[4]，所以是去做冷工作(cold job)。另外，我會將第一條到第五條線程(thread)綁定到編號為 0 的核心(cpu0)，而第六條到第十條線程(thread)則是綁定到編號為 1 的核心(cpu1)。

所以，我們會期望在編號為 0 的核心上，其執行的情況應該是：熱→熱→熱→冷→冷，然後重複數次；而在編號為 1 的核心上則應該要是這樣的情形：冷→冷→冷→熱→熱，然後重複數次。

2. 將線程(thread)綁定到多核心系統下的某一個核心(core)

使用 CPU_ZERO()、CPU_SET()及 pthread_setaffinity_np()這三個函數(function)來做處理器親和性(affinity)的設定，也就是說，利用這三個函數(function)將某個線程(thread)綁定到某個特定的核心(cpu)上。參考程式碼如下面的程式範例(一)：

```
cpu_set_t cpuset;
int cpu = 0;                //the CPU we want to use
CPU_ZERO(&cpuset);          //clear the cpuset
CPU_SET( cpu , &cpuset);    //set CPU 0 on cpuset
If (pthread_setaffinity_np(pthread_self(), sizeof(cpuset), &cpuset)
< 0) { fprintf(stderr, "set thread affinity failed\n");}
//set pthread on CPU 0
```

程式範例(一)

3. 設定每一個線程(thread)個別的優先權(priority)

一般來說，Linux 核心(kernel)的排程(scheduler)其排程策略(scheduling policy)主要可以分為 SCHED_RR、SCHED_FIFO 及 SCHED_OTHER 這三種。其中，SCHED_RR 是時間片(Time Slice)輪轉(Round Robin)排程演算法，而 SCHED_FIFO 則是先來先執行(First in, first out.)排程演算法，然後這兩種策略(policy)都是針對實時任務(Real-Time Task)。相對的，SCHED_OTHER 則是針對一般目的的任務，所以也叫 SCHED_NORMAL，這種排程策略(scheduling policy)會根據任務(task)的 Nice 值來計算該任務(task)的執行時間片(Time-Slice)的大小。當使用者把進程(process)或線程(thread)的 Nice 值遞減時(最低為-20)，就表示該進程(process)或線程(thread)的優先權提高，所以在排程(scheduling)時就會取得比較長的時間片(Time-Slice)。

```
pid_t tid = syscall(SYS_gettid);
//取得該線程的 ID
struct sched_param param;
param.sched_priority = priority;
//設定線程的優先權，變數 priority 用以儲存預設定的線程優先權
if (sched_setscheduler(tid, SCHED_RR, &param) == -1)
{
    perror("sched_setscheduler() failed");
    exit(1);
}
//利用函數 sched_setscheduler()設定某一線程要使用的排程方法及其優先權，SCHED_RR 亦可以 SCHED_FIFO 替代。
```

程式範例(二)

(1) SCHED_RR 和 SCHED_FIFO

如果使用者希望使用 SCHED_RR 或 SCHED_FIFO 這兩種排程策略(scheduling policy)對某一個線程(thread)進行排程，則使用者可以在線程函數(thread function)中加入如程式範例(二)中的程式碼。

(2) SCHED_OTHER

如果沒有在線程函數(thread function)中特別加上前面提及的幾行程式碼，基本上 Linux 核心(kernel)就會以 SCHED_OTHER 這個排程策略(scheduling policy)對該線程(thread)進行排程。但是，使用者可以在線程函數(thread function)中利用 setpriority()這個函數(function)來改變該線程(thread)的 Nice 值，如下：

```
pid_t tid = syscall(SYS_gettid);
//取得該線程的 ID
setpriority(PRIO_PROCESS, tid, -10);
//設定該線程的 Nice 值，數值可以設定在-20 - 19 之間，
//數值越小則該線程的優先權越高，反之。
```

程式範例(三)

4. 執行熱工作(hot job)和冷工作(cold job)

(1)熱工作(hot job)

熱工作(hot job)由 hot function 專門執行。熱工作(hot job)基本上就是指計算密集型工作(computation intensive jobs)，所以程式碼基本上就是要迫使核心(cpu)處於不斷地進行計算的狀態。我的方法是，在一個執行非常多次的迴圈中，讓十個變數做各種計算並且重複十次，如下：

```
for(i=0;i<100000000;i++) //反覆執行的 for 迴圈
{
    h1 = h1 * 2 + 1;
    h2 = h2 * 3 + 2;
    h3 = h3 * 4 + 3;
    h4 = h4 * 5 + 4;
    h5 = h5 * 6 + 5;
    h6 = h6 * 7 + 6;
    h7 = h7 * 8 + 7;
    h8 = h8 * 9 + 8;
    h9 = h9 * 10 + 9;
    h10 = h10 * 11 + 10;
    //再重複上面的程式碼十遍
}
```

程式範例(四)

(2)冷工作(cold job)

```
int *c[1000000];
int **ptr;
int i,j,k;
//以下為初始化指標陣列，
//使得每一個指標皆指向距離自己  $2^n$  個陣列元素的位置
for(i=0;i<1000000;i++)
{
    j=(i+2n)%1000000;
    c[i] = (int*)&c[j];
}
struct timeval start_1, end_1;
gettimeofday(&start_1, 0); //紀錄開始時間
ptr = &c[0]; //使 ptr 指向指標陣列中的第一個元素
for(k=0;k<1000000;k++) //反覆執行的 for 迴圈
{
    //下面這行程式碼會讓 ptr 移動到
    //離現在位置  $2^n$  個陣列元素的位置
    ptr = (int**)*ptr; //重複此步驟 200 次
```

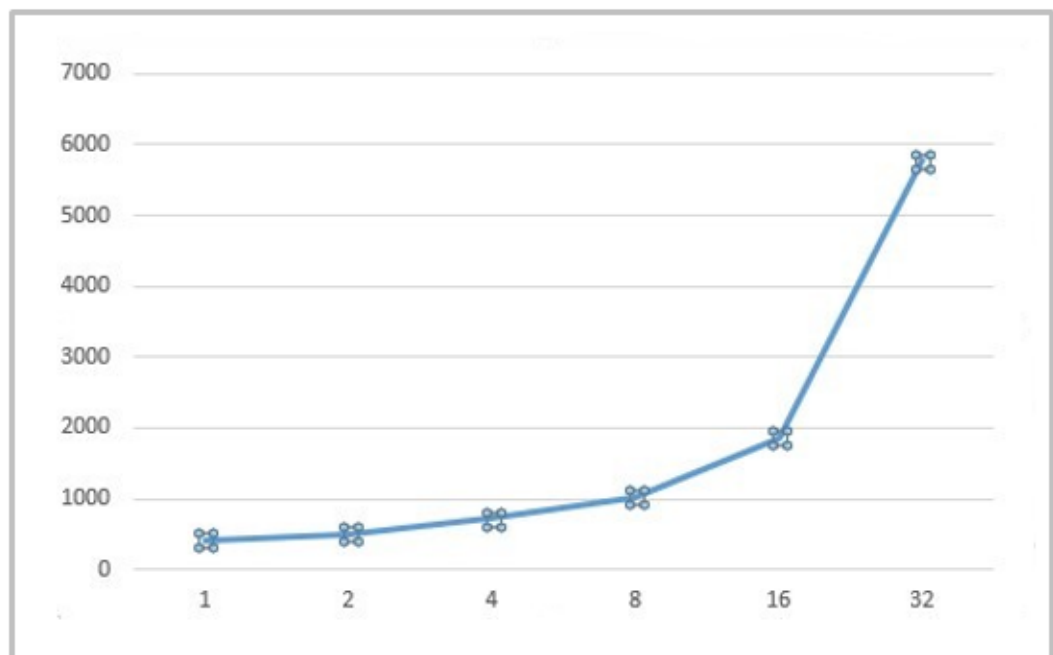
程式範例(五)

冷工作(cold job)由 cold function 專門執行。冷工作(cold job)則是指記憶體使用密集型工作(memory access intensive jobs)，顧名思義，就是要讓程式不斷的去記憶體(memory)讀或寫資料以閒置核心(cpu)，然後盡量不要使用核心(cpu)去做任何的計算。基本上，核心(cpu)會先把記憶體(memory)的資料讀取到暫存器(cache)以加快計算的速度。若總是讀或寫記憶體(memory)位置接近的資料，程式執行就不會花很多時間在記憶體的使用(memory access)，因為記憶體(memory)的資料早就都被讀取到暫存器(cache)裡了。沒有花時間在記憶體使用(memory access)其背後所代表的意義就是沒有閒置核心(cpu)，所以系統還是會因為計算頻繁而溫度升高。

因此，我必須使我的冷工作(cold job)每次要使用下一個記憶體位置時，都是讀或寫一個距離現在的記憶體位置(memory address)超過暫存器大小(size of cache line)的另一個記憶體位置(memory address)。為了確認系統的暫存器大小(size of cache line)，我設計了如程式範例(五)的一個利用指標(pointer)來使用記憶體(memory access)的程式進行測試。

重複執行程式範例(五)的程式碼，而每次重新執行前只改變 n 的大小， n 從

0 開始，每次加 1。每次執行程式都會計算執行的時間，當發先執行時間突然拉長時便可知道此時 ptr 的移動距離已經超過暫存器(cache)的大小。實驗結果如下圖：



圖(三)

由上圖可以發現，32($n=5$ 的時候， $2^5=32$)是程式執行時間突然急遽上升的時候，由此可知，此時 ptr 的移動距離已經超過暫存器的大小(size of cache line)了。

實驗環境：

1. 量測程式的執行時間

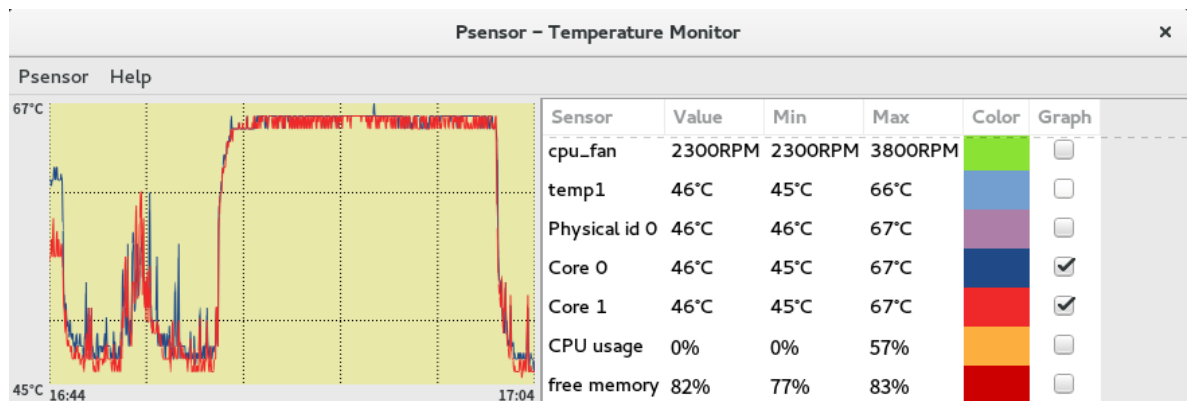
我使用 gettimeofday() 這個函數(function)來量測我的程式的執行時間。用法如下：

```
struct timeval start, end;
gettimeofday(&start, 0); //取得程式執行開始時間
(欲量測執行時間的程式碼區段)
gettimeofday(&end, 0); //取得程式執行結束時間
int sec = end.tv_sec - start.tv_sec;
int usec = end.tv_usec - start.tv_usec;
float interval = (sec*1000+(usec/1000.0));
//變數 interval 就是程式碼區段的執行時間
```

程式範例(六)

2. 監控執行期間的溫度變化

我利用 psensor 來監測程式執行期間的溫度變化，而 psensor 其實是一個將 lm_sensors 擷取到的系統及核心(core)溫度進行圖形化的應用程式。如圖(四)。



圖(四)

3. 設定處理器(cpu)的頻率(frequency)為固定值

只有當處理器(cpu)的頻率(frequency)為固定值的時候，實驗測出來的溫度變化才具有參考價值，這是因為處理器(cpu)的頻率(frequency)大小會影響系統溫度的高低。

(1)確認目前使用的 Linux 核心(kernel)的驅動程式(driver)不是使用 Intel P-State Driver。

(2)在指令列(command line)上輸入如下兩行指令：

➤ `cpupower frequency-set -g <governor>`

將 `<governor>` 設定為 `userspace`。

➤ `cpupower frequency-set -f <frequency>`

將 `<frequency>` 設定成欲設定的頻率，而我是設定成系統的 `max priority` (2200000)。

為了證實頻率(frequency)設定完以後，在程式執行的時候設定好的頻率(frequency)不會再有任何的變動，我做了一個小實驗：

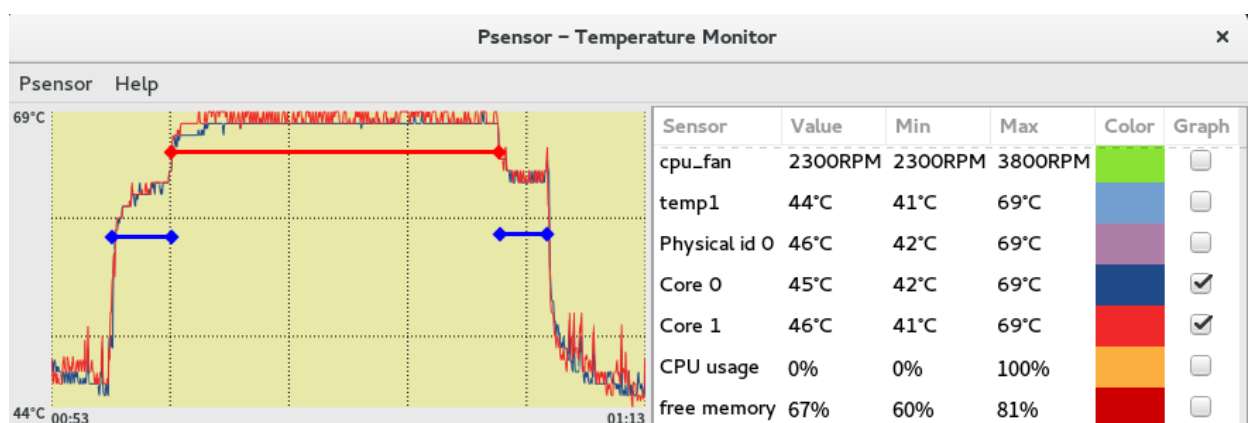
首先我寫了一個 python 檔案，檔案內容如範例程式(七)。

```
# 將這支程式綁定到某一核心(cpu)，例如此程式就是被綁定在 cpu2
import os
os.system("taskset -p -c 2 %d" % os.getpid())

log = open('freq.log', 'w')
while True:
    # 打開記錄核心(core)當前頻率(frequency)的檔案
    f = open('/sys/devices/system/cpu/cpufreq/policy0/scaling_cur_freq')
    # 將現在的核心頻率(frequency)存到變數 freq
    freq = f.read()
    f.close()
    # 將變數 freq 的值寫入名為 freq.log 的檔案
    log.write(freq)
log.close()
```

範例程式(七)

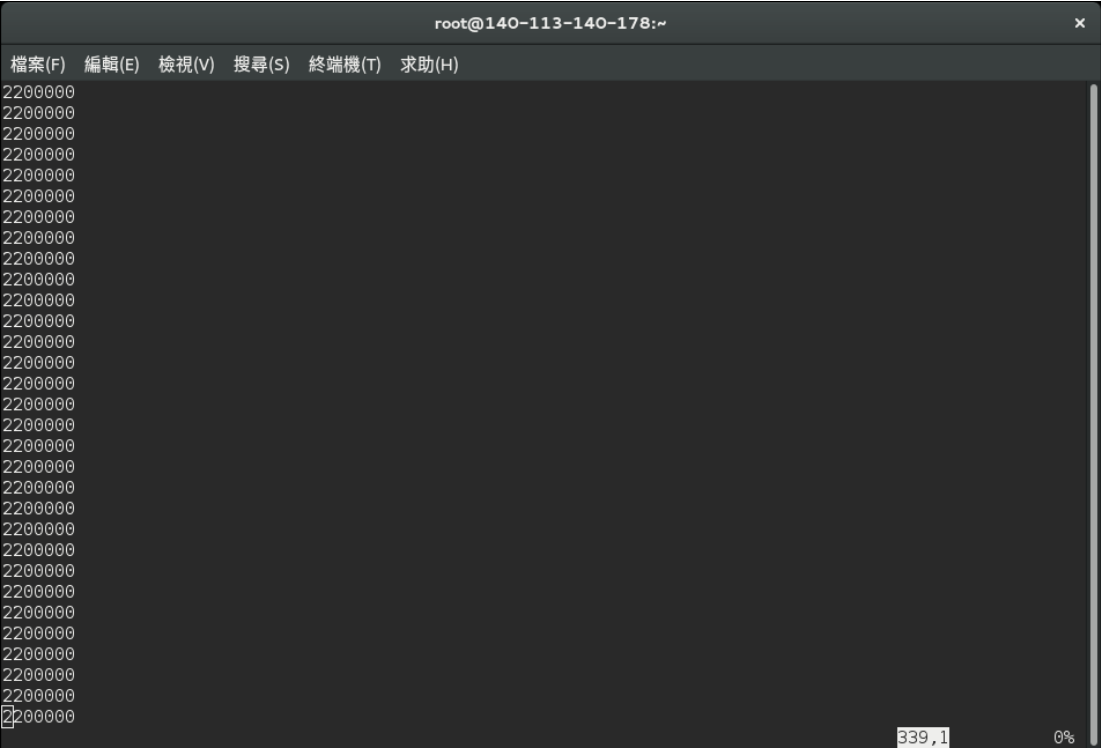
由於範例程式(七)本身會致使核心(core)溫度上升，所以我同時執行兩個同樣寫有範例程式(七)的 python 檔案，只是一個綁定在 cpu2，另一個則是綁定在 cpu3，目的是平衡兩個核心(core 0 對應 cpu0 和 cpu2，core 1 對應 cpu1 和 cpu2)的溫度。接下來，我開始執行我所設計的工具程式，讓 cpu0 和 cpu1 分別執行十條做熱工作(hot job)的線程(thread)並且一邊監控核心(core)的溫度變化直到上述的三支程式都執行完畢。



圖(五)

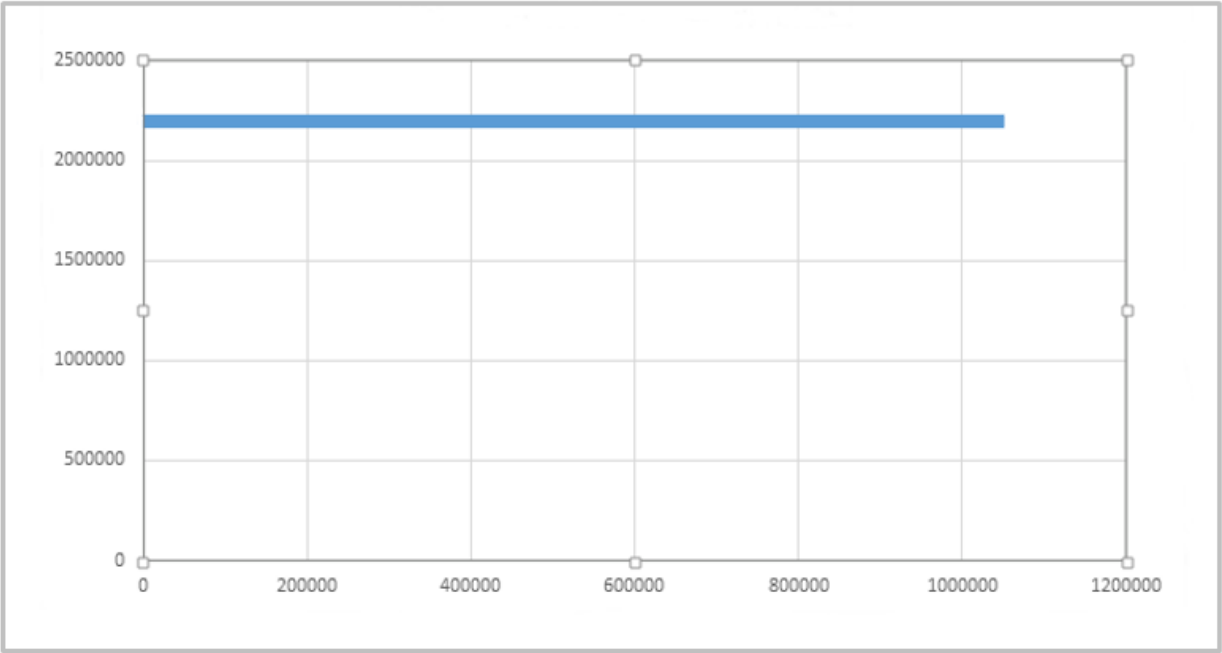
實驗過程的核心(core)溫度變化如圖(五)。圖中的兩段藍色區間是 cpu2 和 cpu3 正在執行 python 檔案，可以發現光是執行範例程式(七)就可以讓核心溫度上升到攝氏 63、64 度；所以當我的工具程式也加入被核心(core)執行的行列時，核心溫度上升到一個更高的位置(大約攝氏 68、69 度)，如圖中的紅色區間。

程式執行完畢後，我打開 freq.log 檔案，可以清楚看見程式執行期間的每個時刻，核心(core)的頻率(frequency)都是固定在 2200000。如下面圖(六)。



圖(六)

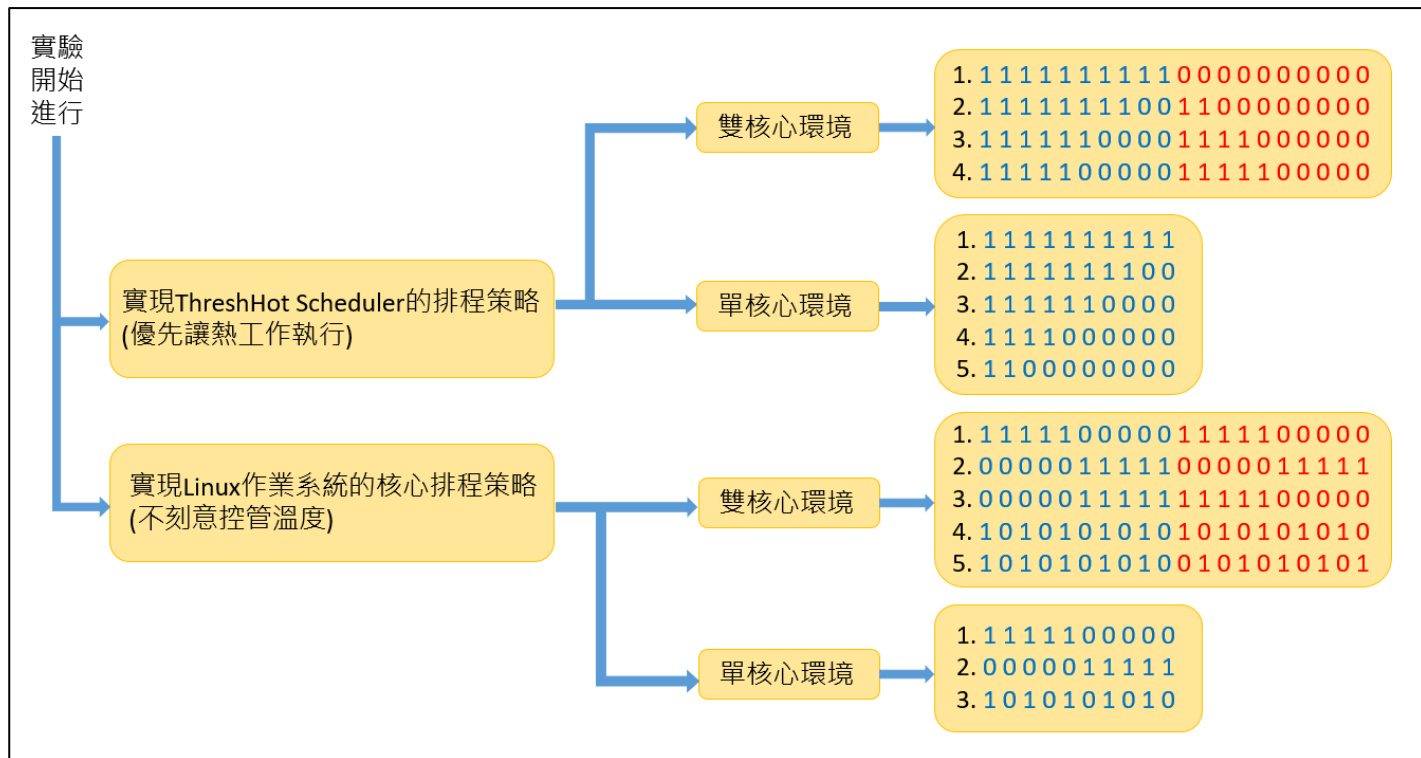
為了更進一步確認 freq.log 檔案中的所有數值真的都是 2200000，我將 freq.log 檔案中的所有數據畫成了圖(七)並成功的確認核心(core)頻率(frequency)確實在我的工具程式執行期間一直維持在固定的數值(2200000)。



圖(七)

核心(core)的頻率(frequency)變化圖

五、結果與討論



圖(八)

實驗進行流程圖

1. 實現 ThreshHot scheduler 的排程策略(scheduling policy) (優先讓熱工作(hot job)執行)：

最一開始進行實驗，我採用了 SCHED_OTHER 排程策略(scheduling policy)，並且給予熱工作(hot job)較低的 Nice 值，也就是給予熱工作(hot job)較高的優先權(priority)。如此一來，就可以像 Performance-Aware Thermal Management via Task scheduling 這篇論文所說的讓熱工作(hot job)優先被核心(cpu)所執行。接下來，我會就雙核心(以其代表多核心)與單核心的環境進行個別討論。

雙核心環境：

起先我設計了如下的四組冷工作(cold job)與熱工作(hot job)的順序組合來測試(前十個工作綁定在 cpu0，後十個工作綁定在 cpu1)：

1. 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
2. 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
3. 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0
4. 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1

但是由於熱工作(hot job)有較高的優先權(priority)，所以實際執行時的結果如下(前十個工作綁定在 cpu0，後十個工作綁定在 cpu1)：

1. 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
2. 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0

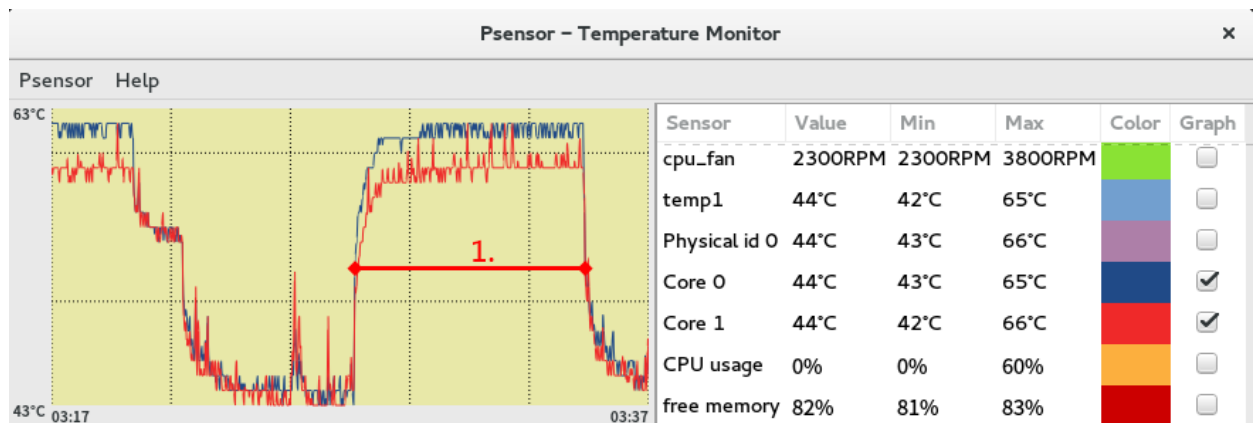
3. 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0
4. 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0

由上面可看出，都是熱工作(hot job)被優先執行，然後重複數次。那麼我們可以發現，其實不管冷工作(cold job)和熱工作(hot job)的順序組合是如何，實際執行時的結果都不出以下這六種：

1. 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
2. 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0 0
3. 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0
4. 1 1 1 1 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0
5. 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0
6. 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
7. 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0
8. 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 0 0 0
9. 1 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0
10. 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0
11. 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1

而且我們還可以發現，1和11、2和10、3和9、4和8、5和7其實都只能算是同一種情形，因為他們唯一的差異是綁定的核心(cpu)剛好相反。以下分別針對上面比較具有代表性的 1、3、5 及 6 這四種情形作進一步的說明(psensor 中，core 0 對應編號為 0 和 2 的核心(cpu0 & cpu2)，core 1 則是對應編號為 1 和 3 的核心(cpu1 & cpu3))：

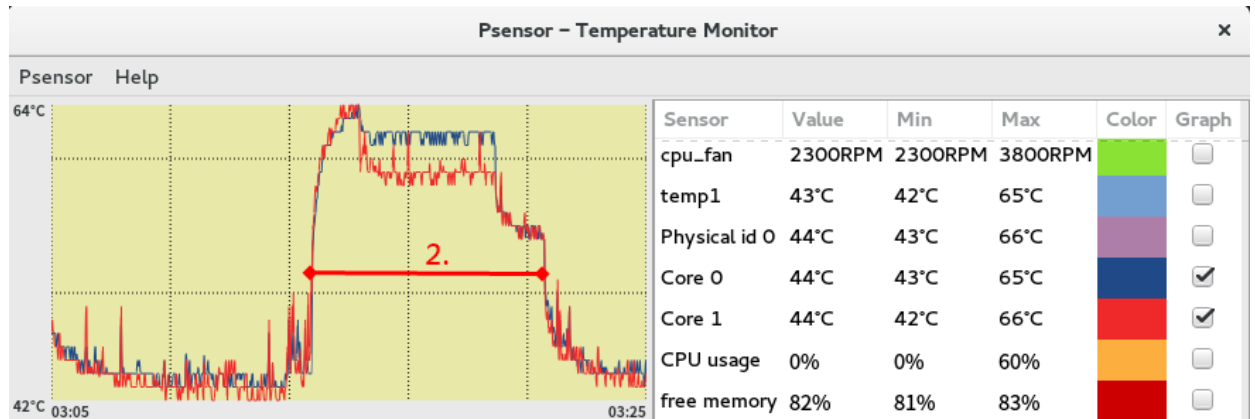
1. 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0



圖(九)

上圖區間「1.」是程式執行區間，可以看的出來，因為 cpu0 都是執行熱工作(hot job)，而 cpu1 都是執行冷工作(cold job)，所以 cpu0 的溫度高於 cpu1 的溫度。

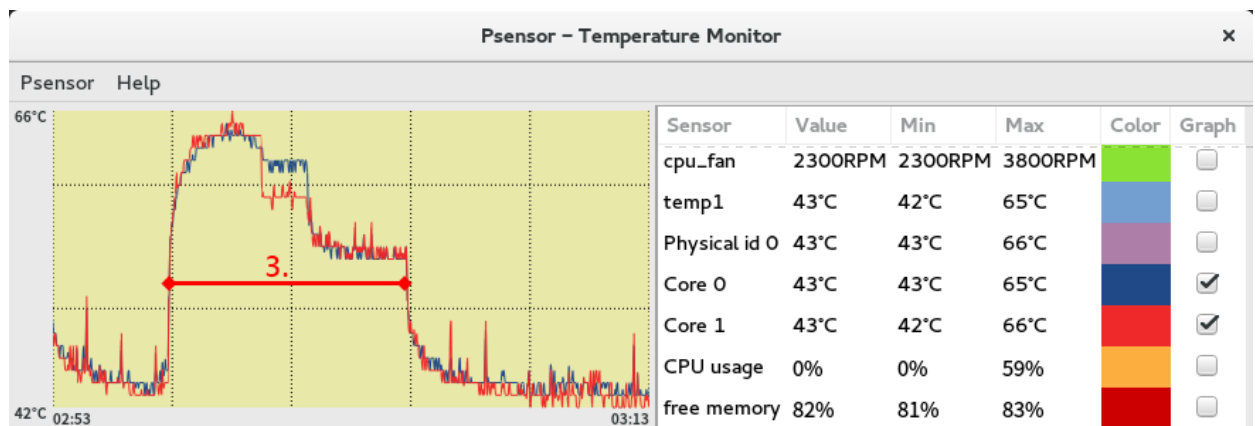
2. 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0 0 0 0 0



圖(十)

由上圖可以發現，區間「2.」其實可以細分為三區段：

- (1) 第一個區段是兩個核心(cpu0 和 cpu1)溫度都很高，因為他們都正在執行熱工作(hot job)。
 - (2) 第二個區段是 cpu1 已經執行完熱工作(hot job)並改成主要執行冷工作(cold job)，而 cpu0 則還是在執行熱工作(hot job)。所以由圖(十)可見，cpu0 繼續維持高溫，而 cpu1 的溫度則已下降。
 - (3) 第三個區段是 cpu0 也執行完熱工作(hot job)，所以這時候的情況變成 cpu0 和 cpu1 都是在執行冷工作(cold job)，因此兩者的溫度一起呈現低溫的狀態。
3. 1 1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0 0 0

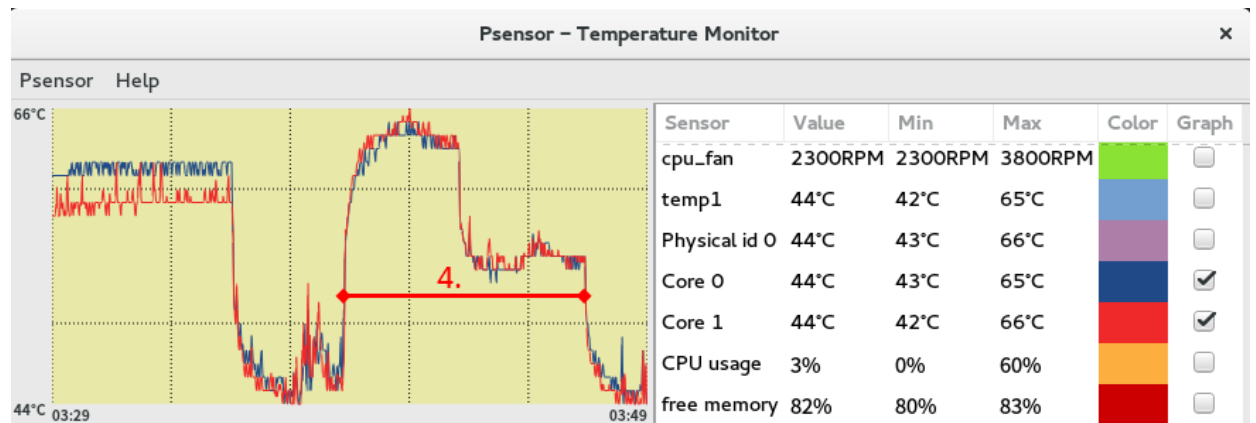


圖(十一)

在圖(十一)中，我們同樣可以把區間「3.」細分為三區段：

- (1) 第一個區段是 cpu0 和 cpu1 都在執行熱工作，所以都是處於高溫的狀態。
- (2) 二個區段是 cpu1 已經執行完熱工作(hot job)並改成主要執行冷工作(cold job)，而 cpu0 則還是在執行熱工作(hot job)。所以由圖(十一)可見，cpu0 繼續維持高溫，而 cpu1 的溫度則是逐漸下降。比較特別的是，因為兩個核心(core)的熱工作(hot job)和冷工作(cold job)的數量差異較小，所以中間的這個區間，圖(十一)相對於圖(十)來的短。
- (3) 第三個區段是 cpu0 也執行完熱工作(hot job)，所以這時候的情況變成 cpu0 和 cpu1 都是在執行冷工作(cold job)，因此兩者的溫度一起下降。

4. 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0



圖(十二)

在圖(十二)中，我們可以把區間「4.」細分為兩區段：

- (1)第一個區段是 cpu0 和 cpu1 都在執行熱工作，所以都是處於高溫的狀態。
- (2)第二個區段是由於兩個核心(cpu0 和 cpu1)執行的熱工作(hot job)數量相同，所以兩者同時執行完畢熱工作(hot job)，然後改成一起執行冷工作，造成兩個核心(cpu0 和 cpu1)的溫度同時急遽下降，並共同維持在低溫的狀態一段時間。

討論：

綜合前面四張圖進行比較，可以發現在圖(十)、圖(十一)還有圖(十二)中，核心(cpu)的溫度變化的確是隨著程式的執行逐漸下降，如此即符合 Performance-Aware Thermal Management via Task scheduling 這篇論文的期望，就是讓核心(cpu)的溫度逐漸降低，那麼隨著程式的執行，核心(cpu)被系統降頻的機會也會逐漸降低。

單核心環境：

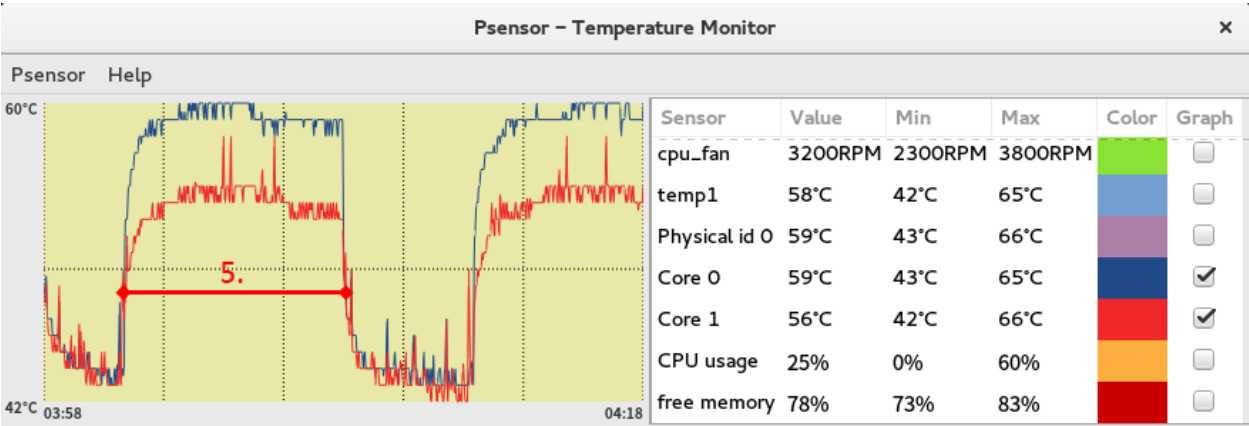
由前一段在雙核心環境下的實驗可以知道，因為熱工作(hot job)擁有較高的優先權(priority)，所以實際在執行程式的時候都會是熱工作(hot job)被核心(core)優先執行。於是我可以直接推論，在單核心的環境下，實際執行程式的結果應不出以下的 11 種順序組合。

1. 1 1 1 1 1 1 1 1 1 1
2. 1 1 1 1 1 1 1 1 1 0
3. 1 1 1 1 1 1 1 1 0 0
4. 1 1 1 1 1 1 1 0 0 0
5. 1 1 1 1 1 1 0 0 0 0
6. 1 1 1 1 1 0 0 0 0 0
7. 1 1 1 1 0 0 0 0 0 0
8. 1 1 1 0 0 0 0 0 0 0
9. 1 1 0 0 0 0 0 0 0 0
10. 1 0 0 0 0 0 0 0 0 0
11. 0 0 0 0 0 0 0 0 0 0

和在雙核心的環境下做實驗一樣，我取了其中幾個比較具有代表性的順序組合來

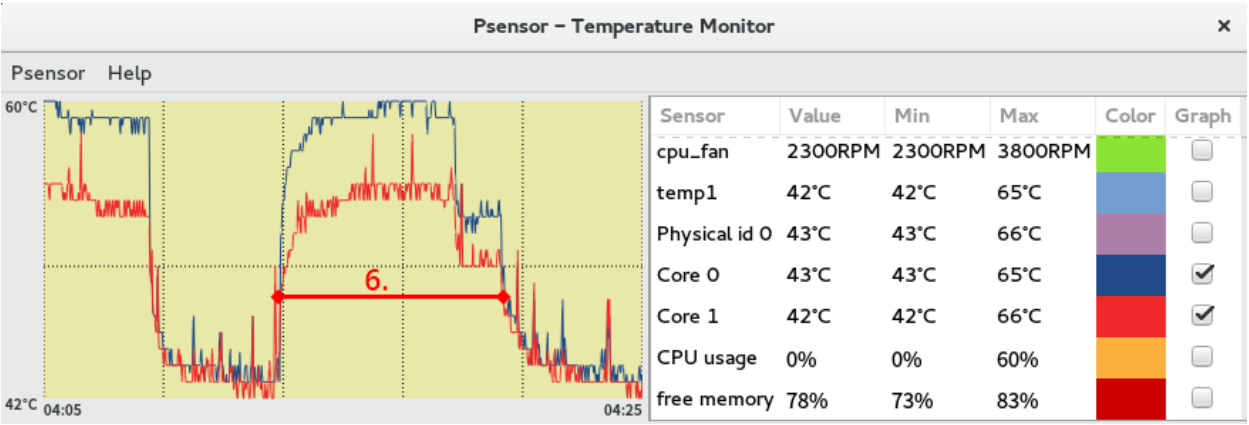
做實驗，分別是第 1、3、5、7 及 9 組，以下圖(十三)到圖(十七)為實驗過程中核心(core)的溫度變化。

1. 1 1 1 1 1 1 1 1 1 1



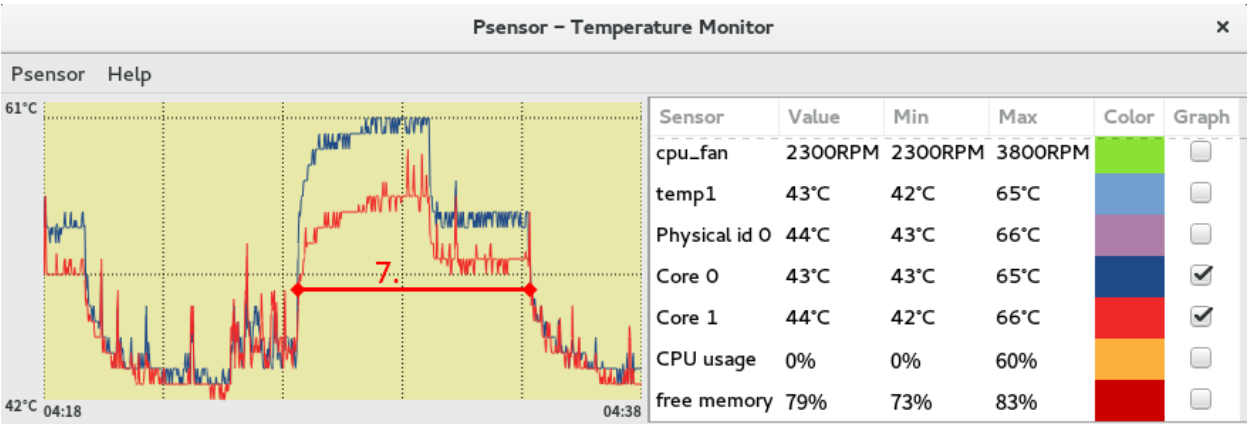
圖(十三)

2. 1 1 1 1 1 1 1 1 0 0



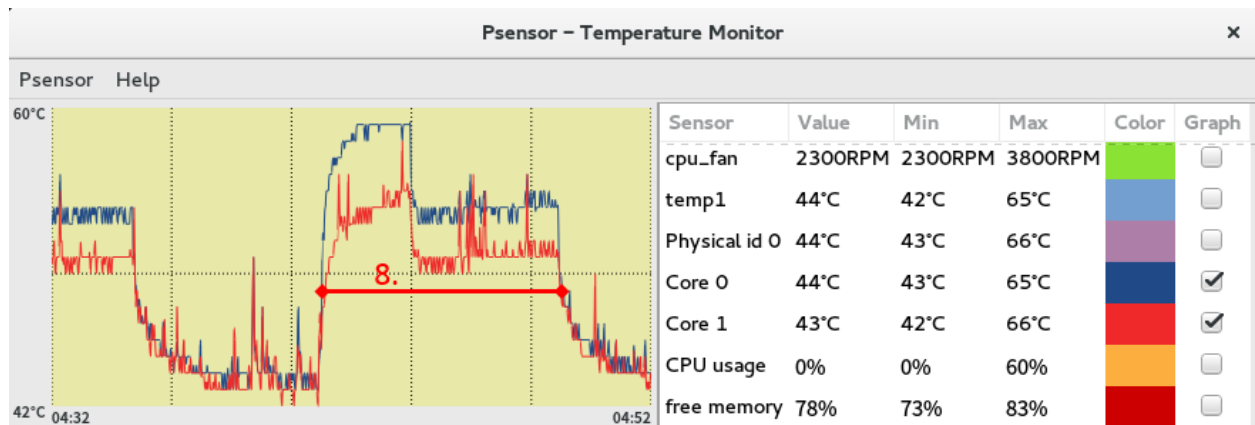
圖(十四)

3. 1 1 1 1 1 1 0 0 0 0



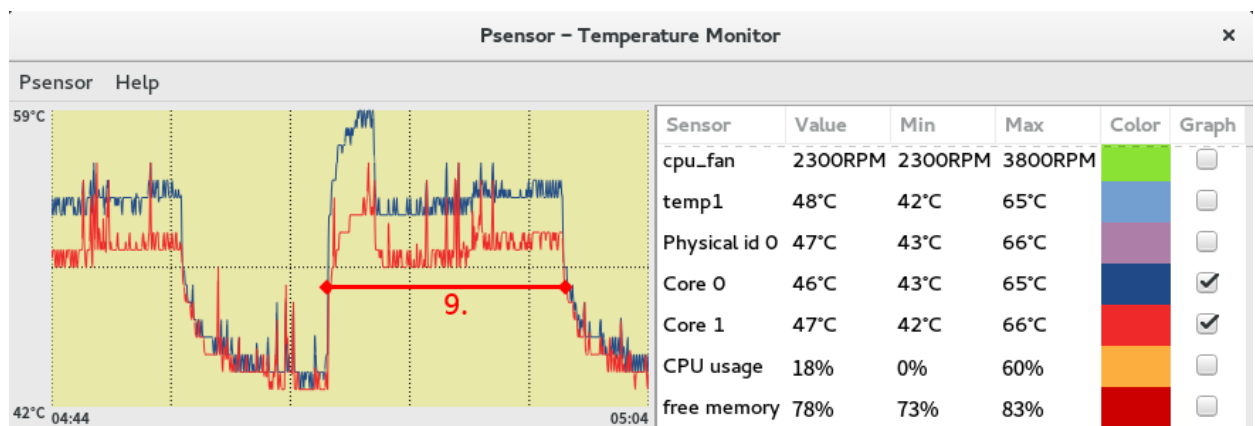
圖(十五)

4. 1 1 1 1 0 0 0 0 0 0



圖(十六)

5. 1 1 0 0 0 0 0 0 0 0



圖(十七)

討論：

從前面圖(十三)至圖(十七)的實驗結果來看，我主要可以整理出以下的三個重點：

1. 第 1 組實驗(圖(十三))因為 cpu0 要執行的工作全部都是熱工作(hot job)的關係，所以在實驗過程中 cpu0 的溫度明顯高於 cpu1，而 cpu1 應是受到發熱的 cpu0 影響才會溫度也跟著升高。
2. 第 2 組到第 10 組實驗(圖(十四)到圖(十七))的共同點是：因為 cpu0 都是首先執行優先權(priority)較高的熱工作(hot job)，所以它在實驗一開始會比較高溫；等到每一組所有的熱工作(hot job)都執行完畢以後，cpu0 就會換成主要執行冷工作(cold job)，於是它的溫度變會下降許多。
3. 接續以上第 2 點討論，我證實了在單核心的環境下，核心(cpu)的溫度變化確實是隨著程式的執行逐漸下降，如此就跟 Performance-Aware Thermal Management via Task scheduling 這篇論文成果相仿，也就是隨著程式的執行讓核心(cpu)的溫度逐漸降低，那麼核心(cpu)被系統降頻的機會也會逐漸降低。

2. 實現 Linux 作業系統傳統的核心排程(kernel scheduling) (不刻意去管控溫度(thermal management))

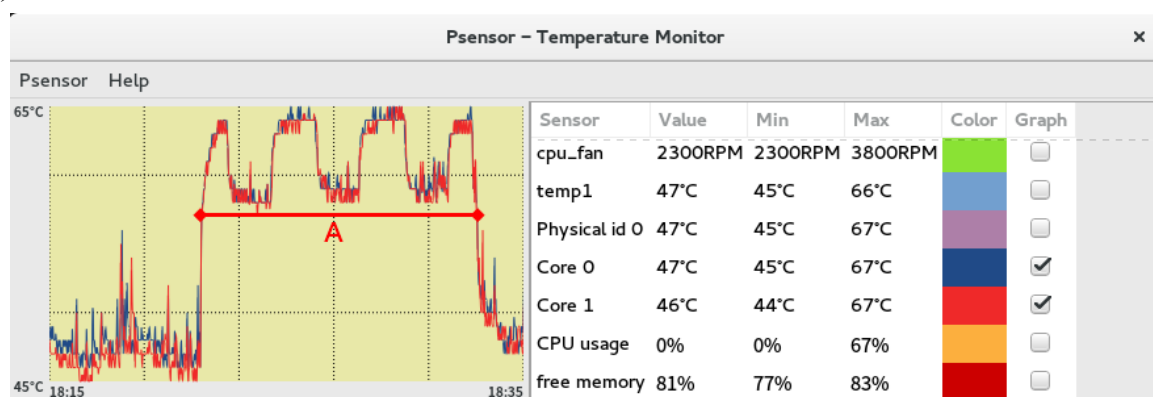
相對於 ThreshHot scheduler，Linux 作業系統傳統的核心排程(kernel scheduling)在進行排程的時候部會刻意去控管溫度。所以它的排程(scheduling)結果可能會是優先執行熱工作(hot job)、優先執行冷工作(cold job)、熱工作(hot job)與冷工作(cold job)交互執行或是更多種各式各樣的順序組合。

為了實現上述的各種順序組合，我希望可以讓每條線程(thread)都按照我在指令列(command line)上給定的順序被處理器(cpu)執行，且每隔一段時間就要做不同線程(thread)之間的切換(context switch)。因此，SCHD_RR 會是我的最佳選擇，因為對於被設定成 SCHD_RR 的線程(thread)來說，基本上其排程行為會和 SCHD_FIFO 一樣，也就是先來先執行(First in, first out.)，但不同的是，該線程(thread)將被 Linux 核心(kernel)分配一個時間片(time slice)去限制時間，時間一但用盡，它就會被暫停以讓出處理器(cpu)。

實驗環境一樣分成單核心及雙核心(以其代表多核心)，而在這兩種實驗環境之下，我也各自設計了幾組冷工作(cold job)與熱工作(hot job)的順序組合來進行測試。然後為了使每條線程(thread)按照我在指令列(command line)上給定的順序執行，我讓每條線程(thread)被產生的時間都彼此相隔 1000 微秒，如此一來，每條線程(thread)就會按照順序被產生並且被處理器(cpu)所執行。

雙核心環境：

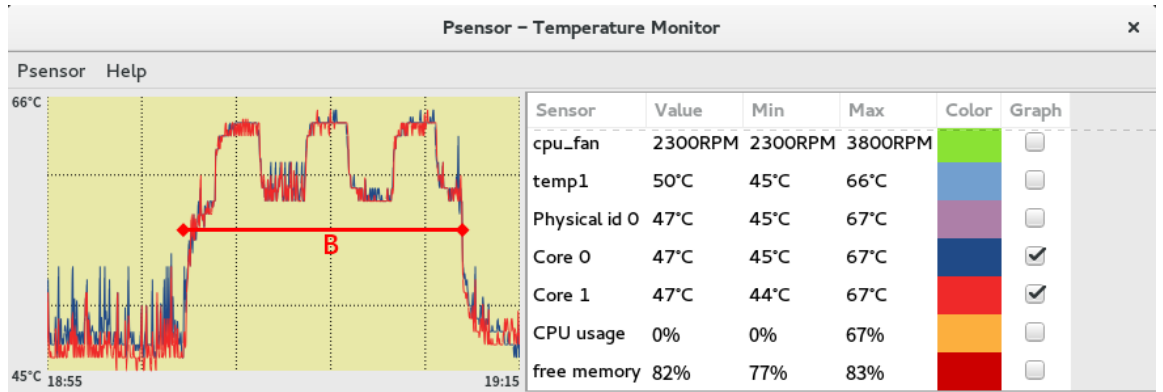
(1) 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0



圖(十八)

使 cpu0 與 cpu1 皆先執行熱工作(hot job)再執行冷工作(cold job)，並且重複這樣的執行順序數次，其結果如圖(十八)。由圖(十八)可以發現，cpu0 與 cpu1 的溫度變化幾乎重合，而且它們的溫度變化就如同單核心第(1)組，一樣是先熱再冷，然後重複同樣的溫度變化數次。另外，由於兩個核心(core)同時執行工作(job)，所以雙核心第(1)組的溫度相對於單核心第(1)組的溫度來的更高。

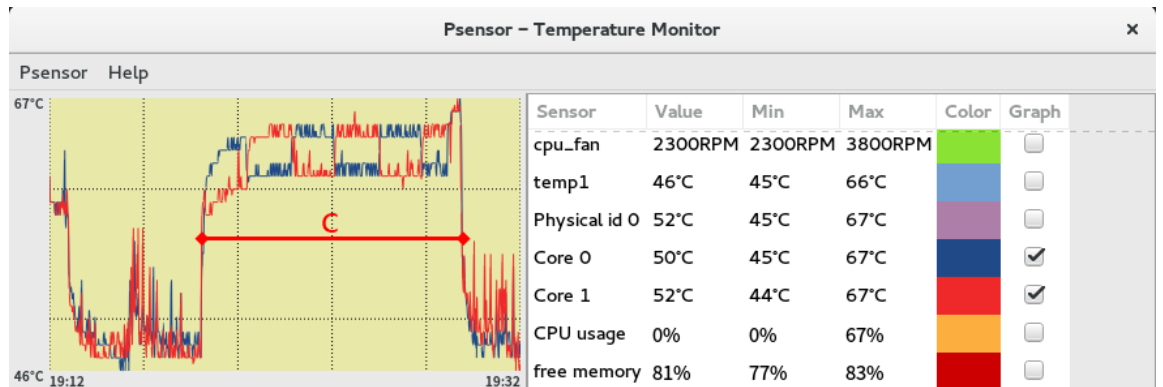
(2) 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 1 1 1 1 1



圖(十九)

與雙核心第(1)組相反，雙核心第(2)組是使 cpu0 與 cpu1 皆先執行熱工作(hot job)再執行冷工作(cold job)，並且重複這樣的執行順序數次。執行結果的溫度變化(圖(十九))並不令人意外，cpu0 與 cpu1 的溫度變化如同雙核心第(1)組，幾乎完全重合；但是它們的溫度變化則與雙核心第(1)組完全相反，也就是先冷在熱，然後重複同樣的溫度變化數次。

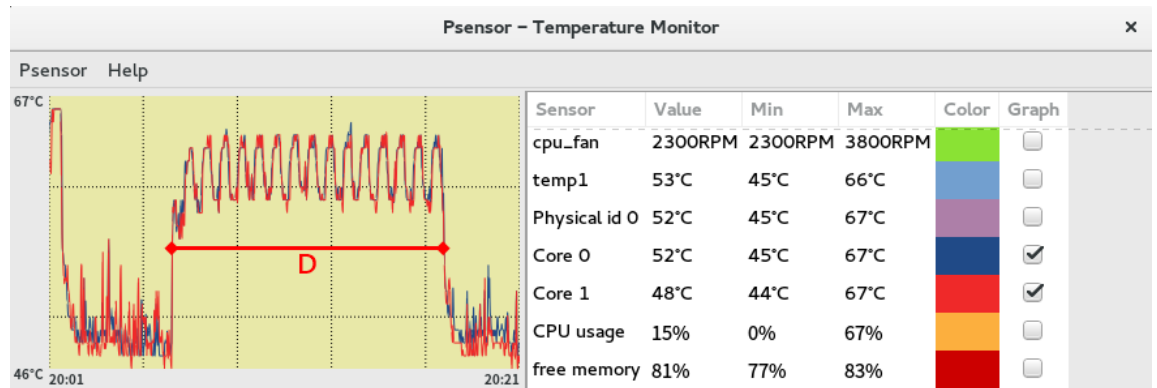
(3) 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0



圖(二十)

在雙核心第(3)組中，cpu0 先執行冷工作(cold job)再執行熱工作(hot job)，而 cpu1 則是先執行熱工作(hot job)再執行冷工作(cold job)。由於兩個核心(core)對於工作(job)的執行順序是彼此互補，所以兩個核心(core)在執行期間的溫度變化也是呈現彼此互補的樣貌(如圖(二十))：當 cpu0 為高溫的時候，cpu1 就是低溫的狀態；相反的，當 cpu0 呈現低溫時，cpu1 就變得高溫。

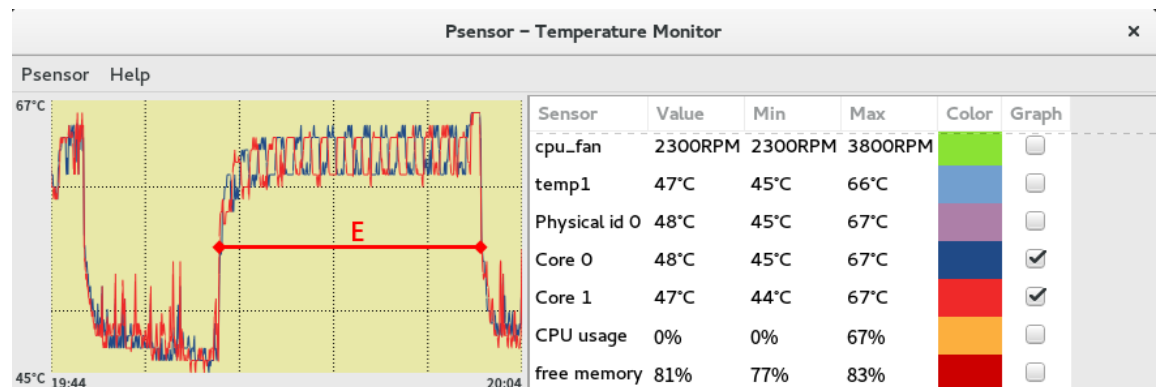
(4) 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0



圖(二十一)

雙核心第(4)組是讓熱工作(hot job)與冷工作(cold job)交互執行，並且讓兩個核心(core)對於熱工作(hot job)與冷工作(cold job)的執行順序完全相同。兩個核心(core)的執行順序完全相同的意思是：cpu0 與 cpu1 對於熱工作(hot job)與冷工作(cold job)的執行順序是一致的，也就是說，若 cpu0 的執行順序是「熱→冷→熱→冷→熱」，那麼 cpu1 的執行順序就會是「熱→冷→熱→冷→熱」。雙核心第(4)組執行期間的溫度變化如圖(二十一)所示，可看出 cpu0 與 cpu1 的溫度變化彼此重合，並呈現熱與冷交互變化的模式。

(5) 1 0 1 0 1 0 1 0 1 0 0 1 0 1 0 1 0 1 0 1



圖(二十二)

雙核心第(5)組跟雙核心第(4)組一樣，都是讓熱工作(hot job)與冷工作(cold job)交互執行，但是它會讓兩個核心(core)對於熱工作(hot job)與冷工作(cold job)的執行順序互相相反。因此，如果 cpu0 的執行順序是「熱→冷→熱→冷→熱」，那麼 cpu1 的執行順序就會是「冷→熱→冷→熱→冷」。執行結果的溫度變化如圖(二十二)，可以發現 cpu0 與 cpu1 皆是呈現熱與冷交互變化的模式，但他們的溫度變化卻是互補，意即若 cpu0 為高溫，則 cpu1 就是低溫，反之。

討論：

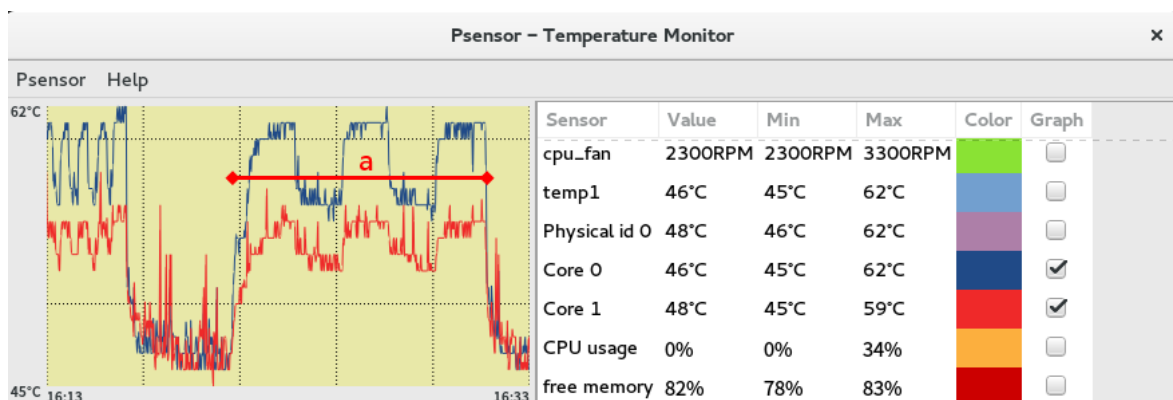
在雙核心的環境下，如果 cpu0 和 cpu1 對於熱工作(hot job)和冷工作(cold job)的執行順序是一致的，則兩個核心(core)的溫度變化會幾乎重合；相反的，若 cpu0 和 cpu1 對於熱工作(hot job)和冷工作(cold job)的執行順序是互相相反的，那麼兩個核心(core)

的溫度變化就會呈現與彼此互補的樣貌。

其中，值得注意的是，雙核心第(1)組是先做熱工作(hot job)再做冷工作(cold job)，但是它執行結束的時候卻是呈現高溫的狀態；而雙核心第(2)組則是先做冷工作(cold job)再做熱工作(hot job)，其執行結束的時候則是呈現低溫的狀態。對於前述雙核心第(1)組和第(2)組的溫度變化，我認為比較合理的解釋是：因為系統總是先執行一陣子的熱(冷)工作(hot/cold job)，再去執行一陣子的冷(熱)工作(cold/hot job)，然後又回頭來執行一陣子的熱(冷)工作(hot/cold job)，並以這樣的規則持續到執行完所有的冷和熱工作(cold job and hot job)。所以，如果程式執行到了最後正好是在執行熱工作(hot job)，那麼它執行結束的時候便會是高溫的狀態，反之。

單核心環境：

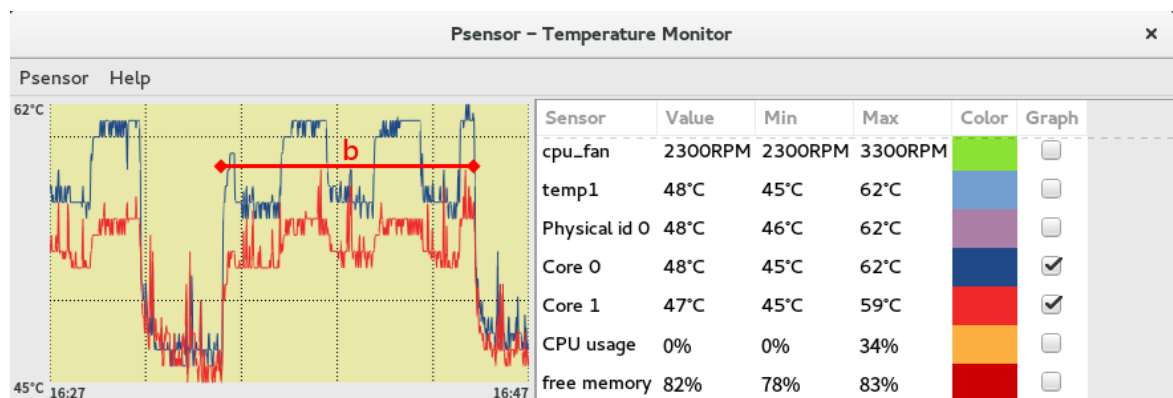
(1) 1 1 1 1 1 0 0 0 0 0



圖(二十三)

上圖區間「a」是程式執行區間，可以看得出來，cpu0 的溫度變化是先熱再冷，然後重複這樣的溫度變化數次。如此的溫度變化有符合預期，因為 cpu0 也是先執行熱工作(hot job)再執行冷工作(cold job)，並且重複這樣的執行順序數次。

(2) 0 0 0 0 0 1 1 1 1 1

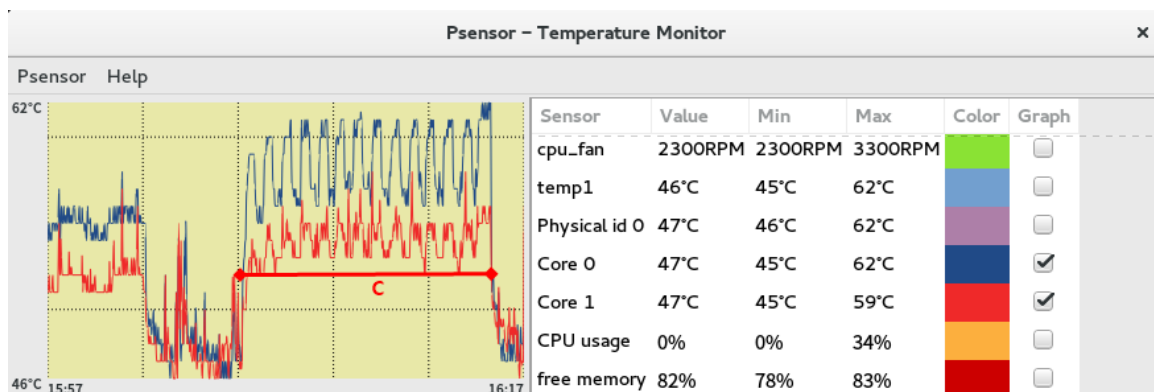


圖(二十四)

在圖(二十四)中，由於熱工作(hot job)與冷工作(cold job)的執行順序與單核心第(1)組的執行順序完全相反，所以區間「b」的溫度變化基本上呈現與圖(二

十三)中區間「a」相反的溫度變化，也就是 cpu0 的溫度變化是先冷再熱，然後重複這樣的溫度變化數次。

(3) 1 0 1 0 1 0 1 0 1 0



圖(二十五)

單核心第(3)組是讓熱工作(hot job)與冷工作(cold job)交互執行，如同預期的，cpu0 的溫度變化呈現熱與冷交互變化的模式。

討論：

在單核心的執行環境下，由優先執行熱工作(hot job)、優先執行冷工作(cold job)或是熱工作(hot job)與冷工作(cold job)交互執行所得到的溫度變化圖(圖(二十三)、圖(二十四)及圖(二十五))來看，這三種不同的線程(thread)執行順序在高溫時的溫度其實都差不多，且沒有一個的溫度變化是呈現逐漸下降的趨勢，此外在圖(二十四)和圖(二十五)中，當程式執行到了尾聲核心(core)的溫度甚至還往上提升。

3. 總結討論

本研究利用我所設計的工具程式成功的實踐了 **ThreshHot scheduler** (由 Performance-Aware Thermal Management via Task scheduling 這篇論文所提出的排程演算法)的排程策略(scheduling policy)(優先讓熱工作(hot job)執行)以及 Linux 作業系統傳統的核心排程(kernel scheduling)(不刻意去管控溫度(thermal management))。然後使上述的兩種排程策略(scheduling policy)皆在雙核心和單核心的環境下，進行實驗(執行各種熱工作(hot job)和冷工作(cold job)的順序組合)並觀察它們的行為表現。因此實驗進行至此，總共有以下四種組合：

1. ThreshHot scheduler 在雙核心的環境下執行冷和熱工作(cold job and hot job)
2. ThreshHot scheduler 在單核心的環境下執行冷和熱工作(cold job and hot job)
3. Kernel scheduler 在雙核心的環境下執行冷和熱工作(cold job and hot job)
4. Kernel scheduler 在單核心的環境下執行冷和熱工作(cold job and hot job)

其中第 1 組和第 2 組，都有成功達成 ThreshHot scheduler 所期望的目標，也就是讓核心(core)的溫度逐漸下降，這樣一來核心(core)溫度不會常常超過系統規定的溫度臨界值(threshold)而遭到降頻的處理。因此，透過這兩組實驗的進行，我驗

證了 ThreshHot scheduler 除了在單核心的情況下有效，在雙核心的環境下亦是有效的。

而第 3 組和第 4 組的實驗結果則是說明了 Linux 作業系統傳統的核心排程 (kernel scheduling) 確實在控管核心 (core) 溫度這一方面的效果相對不好，由實驗過程中的核心 (core) 溫度變化圖我們可以知道，當程式執行到快要結束的時候，核心 (core) 溫度不一定是處於一個比較低溫的狀態，甚至核心 (core) 溫度還可能變得更高。基本上，核心 (core) 的溫度越高就越有可能超過系統規定的溫度臨界值 (threshold)，而核心 (core) 被系統降頻的機率當然也是越高。

六、參考文獻

- [1] Zhou, Xiuyi, et al. "Performance-aware thermal management via task scheduling." ARM Transactions on Architecture and Code Optimization (TACO) 7.1 (2010): 5.
- [2] Linux Kernel 排程機制介紹(2011/12/2)
參考網址 <http://loda.hala01.com/2011/12/linux-kernel-排程機制介紹/>
- [3] 第七章进程调度 - Linux 内核之旅
參考網址 <http://www.kerneltravel.net/books/ch07.pdf>
- [4] BROOKS, D. AND MARTONOSI, M. 2001. Dynamic thermal management for high-performance microprocessors. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. IEEE, Los Alamitos, CA, 171–180.
- [5] How to create a new Linux kernel scheduler
參考網址 <http://stackoverflow.com/questions/3086864/how-to-create-a-new-linux-kernel-scheduler>
- [6] Real-time programming with Linux
參考網址 <http://stackoverflow.com/questions/10502508/real-time-programming-with-linux>
- [7] COSKUN, A., ROSING, T., WHISNANT, K., AND GROSS, K. 2008. Static and dynamic temperature-aware scheduling for multiprocessor socs. *IEEE Trans. VLSI Syst.* 16, 9, 1127–1140.
- [8] DONALD, J. AND MARTONOSI, M. 2006. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd International Symposium on Computer Architecture*. ACM, New York, 78–88.
- [9] Program / Process / Thread 的差別
參考網址 <http://finalfrank.pixnet.net/blog/post/27781751-program---process---thread-的差別>
- [10] 一片好文解釋 Mutex，Semaphore，spinlock 的差異 - 轉
參考網址 <http://www.programgo.com/article/41974049459/>
- [11] LI, Y., BROOKS, D., HU, Z., AND SKADRON, K. 2005. Performance, energy, and

thermal considerations for smt and cmp architectures. In *Proceedings of the 11th International Symposium on High Performance Computer Architecture*. IEEE, Los Alamitos, CA, 71–82.

[12] 應佳山(2012)，多核心處理器之群組式動態溫度管理機制。

[13] YUAN, W. AND NAHRSTEDT, K. 2003. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *Proceedings of the 19th Symposium on Operating Systems Principles*. ACM, New York, 149–163.