Linux 核心 IP 層的延遲分析

黃霆鈞 張舜理 林盈達
國立交通大學資訊科學系
October 21, 2009

摘要

找出 Linux 在網路封包處理的瓶頸將可改善各種以 Linux 為平台的設備與服務之效能。本文提供一種方法以分析 Linux 在 IP 層(IP Layer)的函式執行流程，並且量測每個函式所耗費的時間，依此可統計 IP 層總耗費時間，並找出此一層級的瓶頸所在。IP 層的瓶頸所在可分送出端與接收端來討論，以送出端而言，函式 ip_finish_output2()佔了約 50%的執行時間，而接收端的部份 ip_route_input()、ip_local_deliver_finish()、ip_rcv()、ip_rcv_finish()這四個函式分別佔據 15%～25%的時間。雖然改進上述函式可以提高執行效率，但由於 IP 層的整體執行時間(最大值為 7 μs)相較於其它層級(連結層至少 10 μs)仍顯得較小，所以就整體系統來說應該要改進其它層級函式較佳。

關鍵字：Linux，IP 層，KFT，rdtscll，延遲分析

一. 簡介

Linux 的網路架構分為五層：「應用層」、「傳輸層」、「網路層」、「連結層」、「實體層」，其中後三層不論是一般 PC、Server 以至於 Router、Switch 等都會實作到，所以設法增進這三層的處理速度將對上述設備都會帶來效益。基於這個原因，我們採取量測 Linux 核心執行過程的做法，來找出處理速度的瓶頸；但是 Linux 的核心程式歷經許多版本的演變，從核心 2.2、2.4 到如今的 2.6.x，每一次的版本演進都加入了許多新模組，以致核心越來越龐大，尤以網路實作方面改變甚多；從 2.4 版核心做出的實驗結果並無法完整套用到 2.6 版核心上，使得重製以往舊版的實驗困難重重。筆者參考前人做過的 Linux 網路核心的實驗，都遇到實驗工具無法順利在 2.6 版核心上運作[1]，以及量測的精確度太低[2]的窘境。

以量測精確度太低為例，解決的辦法是將前人用的 KernProf 改為「插入程式碼 rdtscll()」的方式量測。捨棄 KernProf 的原因有二，其一是因為 KernProf 的量測採用定時偵測 CPU 正在執行的函式，而執行時間較短的函式如：ip_local_out()(約 500 ns)就很有可能被忽略，反之插入 rdtscll()可保證執行到的程式碼一定會有結果；第二個原因是 KernProf 的時間精確度只到 millisecond，而 rdtscll()所量測到的精確度可達到 nanosecond。基於這兩個原因，故選擇 rdtscll()函式是較為妥當的。

第二節介紹的實驗將會以 Kernel Function Tracer(KFT) [3]來追蹤網路層的函

式流程，再以插入程式碼的方式將時間量測函式 rdtscll()放入每個欲追蹤的核心程式碼中。

第三節依據前述實驗做出數據，統計繪成長條圖，並解釋圖形所代表的意義，以及標出封包處理的瓶頸。在此，我們猜測不同協定繪出的圖形應該要相似(因為協定在傳輸層才有定義不同的處理方法)，執行時間也應該要相近。

二. 實驗環境與使用工具簡介

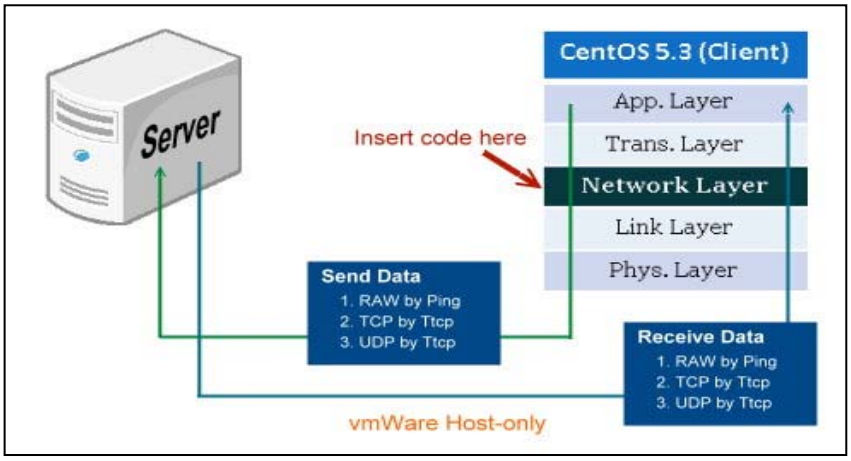

圖 1 實驗架構圖

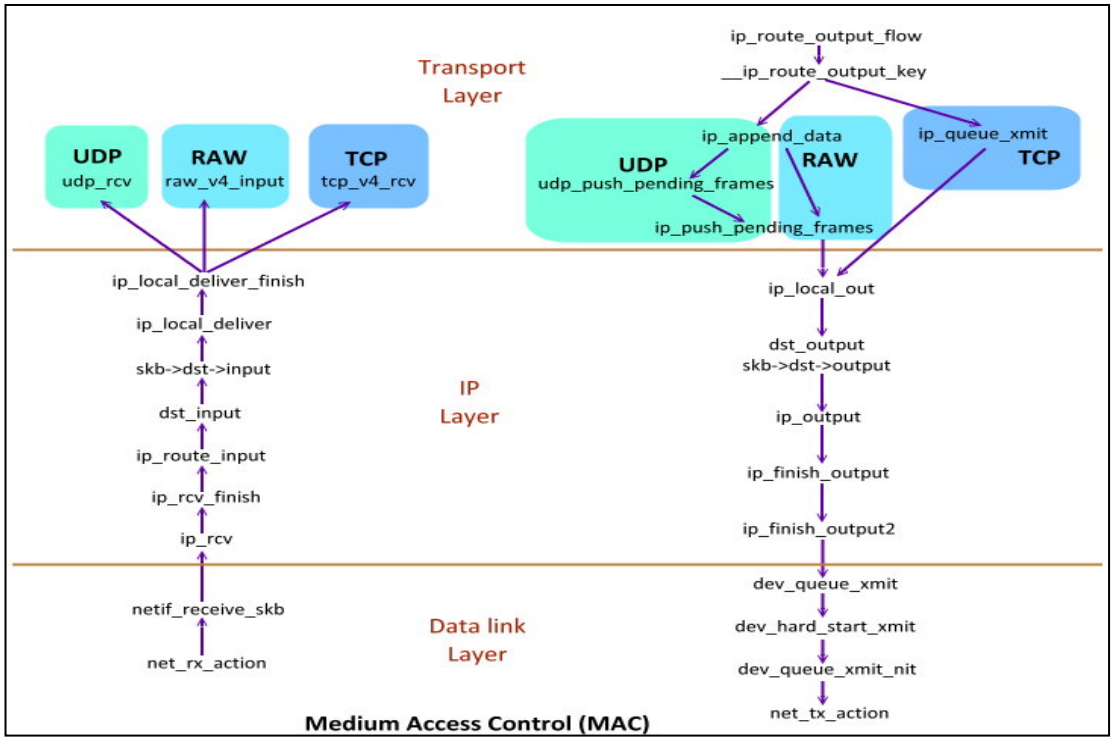

圖 2 由 KFT 產生的數據整理出的 Call Graph

1. 實驗環境

實驗的環境如圖 1 所示，在 VMware 上架設兩台機器，分為 Client 與 Server，

Client 端是主要進行量測的主機。在此筆者選用 CentOS 5.3 做為作業系統，並將 kernel 的版本更新至 2.6.26，Server 端亦是同樣的配置。關於如何建立新版本的 kernel，請參照[4]。

在環境架設好之後，進一步要產生網路流量進行量測。為了避免受其它無關的封包影響，筆者將 VMware 的網路設定為 Host-only，並依不同的量測目標設定不同的量測方式：欲量測 RAW 協定的送出與接收，在 Client 端使用 Ping 來產生封包；而量測 TCP 與 UDP 的送出端則在 Client 端用 Ttcp[5]這個軟體來產生封包；量測 TCP 與 UDP 的接收端是在 Server 端用 Ttcp 產生封包。

2. 使用工具

欲量測 Linux 核心函式所耗費時間，我們分成三個步驟來進行；第一個步驟先使用 Kernel Function Tracer (KFT)來畫出函式間的呼叫順序，稱為 Call Graph。在 Linux 下使用 KFT 必須先 patch 進核心中，再重新編譯過才能運行，而製作 Call Graph 的方式可參考 KFT 網頁。製作出的 Call Graph 如圖 2。

第二個步驟是插入程式碼到 Call Graph 內的所有核心函式中，此處會用到 printk()與 rdtscll()這兩個系統提供的函式。插入的方法以 ip_output()[6]為例：

```
long long begin_cycle, end_cycle;                    // 加入宣告
int ip_local_out(struct sk_buff *skb)
{
    int err;
    rdtscll(begin_cycle);                            // 開始量測
    err = __ip_local_out(skb);
    if( likely(err==1) )
        err = dst_output(skb);
    rdtscll(end_cycle);                              // 結束量測
    return err;
}

void printCycle()
{
    // 印出執行間經過的 Clock cycle 數
    printk(KERN_DEBUG "Execute cycles: %lld", end_cycle-begin_cycle);
}
```

灰底部份為插入的程式碼，ip_local_out()是 Linux 核心函式，printCycle()是筆者撰寫的函式，在 ip_local_out()執行完成後印出資訊。請注意，此時由 printCycle()印出的數值並非是執行時間，而是執行所經 clock cycle 數，故要再計算才能得出時間，公式為：Exe. time = (end_cycle-begin_cycle) / CPU frequency。

最後一個步驟就是產生封包來進行量測。針對不同的協定，量測方法如表一

表一 協定與封包產生方式對照表

協定名稱	產生封包工具與參數	說明
RAW	ping -c 20 addr. (※)	產生 20 個 ping 封包
TCP	ttcp -t -s -n 5 -l 10 addr.	產生 5 個 TCP 封包，每個 10 Bytes
UDP	ttcp -u -t -s -n 5 -l 10 addr.	產生 5 個 UDP 封包，每個 10 Bytes
※ addr. 為目的機器的 IP 位址		

三. 結果與討論

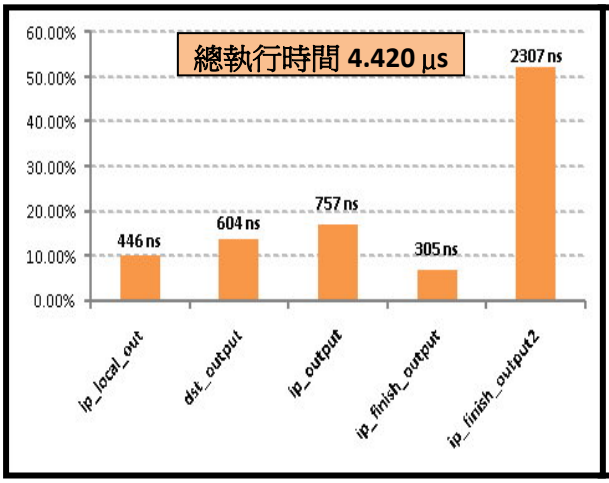

圖 3(a) RAW 送出端各函式所佔百分比

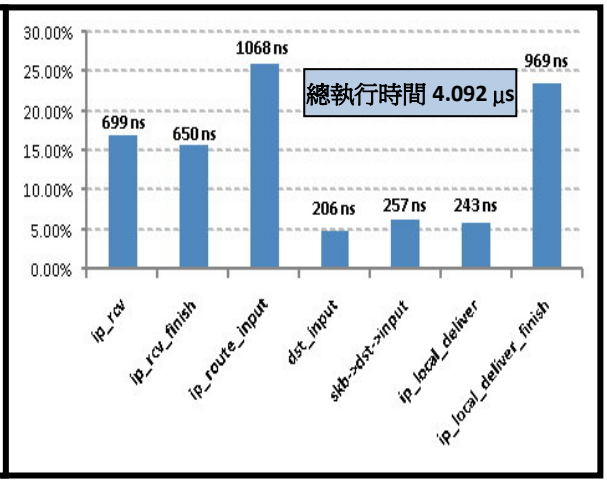

圖 3(b) RAW 接收端各函式所佔百分比

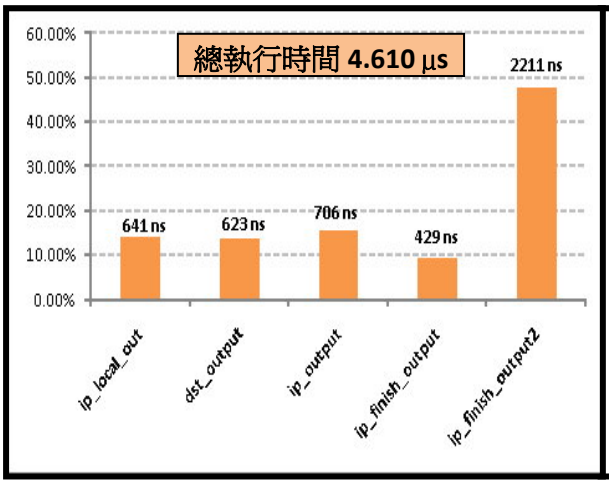

圖 3(c) TCP 送出端各函式所佔百分比

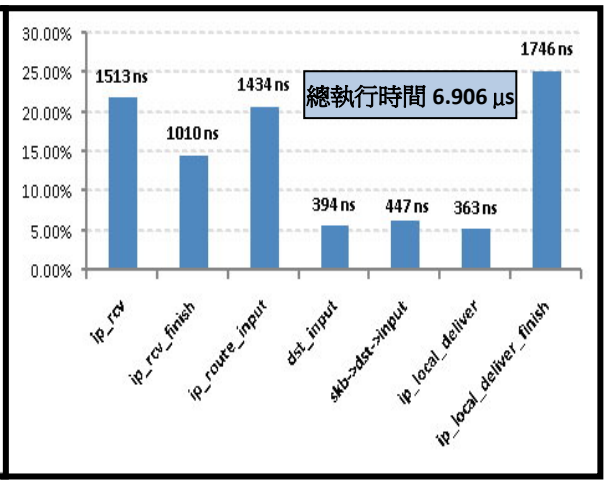

圖 3(d) TCP 接收端各函式所佔百分比

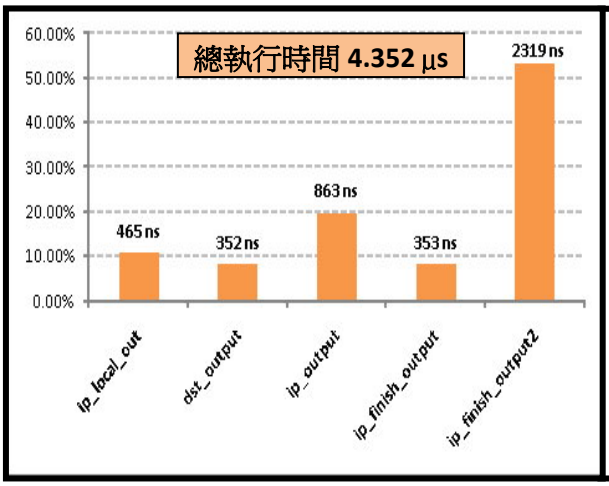

圖 3(e)　UDP 送出端各函式所佔百分比

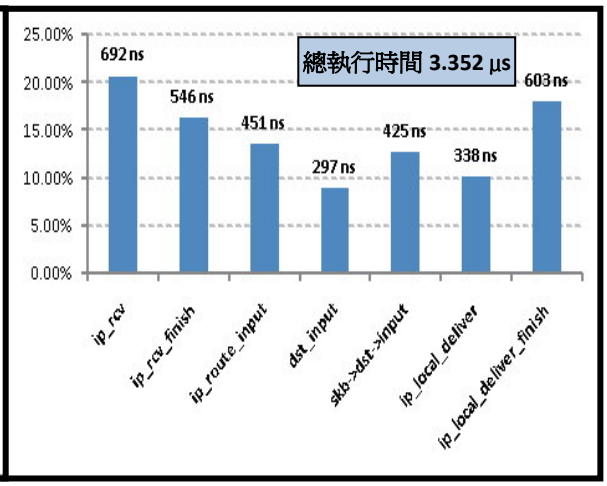

圖 3(f)　UDP 接收端各函式所佔百分比

實驗數據共分成六張圖來呈現，分別是送出端三個協定與接收端三個協定。每張圖在上方顯示量測總執行時間，下方標示量測目標，以長條圖的方式表示各函式佔總執行時間的比例，並標示該函式執行時間。以圖 3(a)為例，目的為量測 RAW 協定送出端的時間，其總執行時間為 4.420 µs，由長條圖可看出最耗時的函式為 ip_finish_output2()，執行時間為 2.3 µs。由 3(a)的例子我們可推論出 3(b)、3(c)、3(d)、3(e)、3(f)的總執行時間分別為 4.092 µs、4.610 µs、6.906 µs、4.352 µs、3.352 µs，且其中最耗時的函式為 ip_route_input(1.07 µs)、ip_finish_output2(2.2 µs)、ip_local_deliver_finish(1.75 µs)、ip_finish_output2(2.3 µs)、ip_rcv(0.7 µs)。

根據圖 3(a)～圖 3(f)六張圖，歸納出以下觀察重點：

1. 送出端在這三個協定運作之下，總執行時間(即延遲時間)並無太大差異，且皆是由函式 ip_finish_output2()佔最多時間。
2. 接收端在 RAW 協定、TCP 協定與 UDP 協定下有各自最費時的函式，跟實驗前所猜測「IP 層接收端所費時間並不因協定而異」有所差別。查閱相關主題的書籍[7]後，找到了最重要的差異點：ip_rcv() 與 ip_local_deliver_finish()，前者會做 check sum 的動作，依據封包大小而有時間上的差異；後者會先註冊該協定，讓協定開始接收此種封包，而處理的時間相差逾 1 µs。

四. 結論

關於網路的瓶頸不在 IP 層，可以從總執行時間(延遲時間)來做個說明：延遲時間最大者為 TCP 在接收一個封包時(圖 3(d))，花費近 7 µs，但相比鄰近的連結層，其函式 dev_hard_start_xmit()動輒耗費 10 µs，對整體效能的影響較小。所以，開發者若需加快網路速度應往其它層級去尋求，而不用花費全力改進 IP 層。

參考資料

[1] 蔡品再、林盈達; 「追蹤 Linux TCP/IP 核心– 使用遠端除錯」2000, http://speed.cis.nctu.edu.tw/~ydlin/miscpub/remote_debug.pdf .

[2] “Kernprof (Kernel Profiling),” http://oss.sgi.com/projects/kernprof/ .

[3] elinux. “Kernel Function Trace,” 17 September 2009, http://elinux.org/Kernel_Function_Trace .

[4] vbird, “Linux 核心編譯與管理,” 18 September 2009, http://linux.vbird.org/linux_basic/0540kernel.php .

[5] Wikipedia. “Ttcp,” 17 September 2009, http://en.wikipedia.org/wiki/Ttcp .

[6] “The Linux Cross Referencer,” 22 September 2009, http://lxr.linux.no/ .

[7] C. Benvenuti, “Understanding Linux Network Internals, ” December 2006