100GbE スイッチのテスト

千代浩司
KEK/IPNS

2023-06-10

100GbE スイッチをテストする機会があったのでメモとしてまとめておく。今回のテストの目的は 100GbE スイッチから 100Gbps でデータがでるかどうかである。たとえば 10GbE を 10 本まとめて 100Gbps で PC にデータを送るテストも考えられるが今回はデータソースの準備の問題があるので単純に 100Gbps をスイッチにいれて 100Gbps でデータがでるかどうかを確認した。TCP セッションを 2 個使うことで PC1 台をデータソースにして、他の PC1 台で読み、100Gbps でデータ転送ができることを確認した。

各種調整はデータフローごとに調整事項は変わります。ここでは 100 GbE スイッチのテストのために 100 Gb/s 程度の TCP フローをつくり出す必要があったのでいろいろ調節しているが、実験データの読み出しには必要ない、あるいは悪影響がある調整も含まれる可能性があるのでご注意ください。

1　最大スループットの計算

今回の測定は TCP ペイロード転送レートを測定している。

イーサネット上で TCP/IP を使う場合にはイーサネットフレーム間ギャップ、イーサネットのヘッダ、トレーラ、IP、TCP のヘッダがあり、さらに TCP のオプションを設定する場合にはユーザーデータ長が減ることになる。

TCP Timestamp が有効の場合 (通常 PC 間の通信では有効になっている)、ユーザーデータは最大 1448 バイトになり最大スループットは 94.1 Gbps になる。TCP Timestamp オプションを使わない場合は最大スループットは 94.9 Gbps である。

2　使用機材、使用 OS

使用した機材と主なスペックを表にまとめておく。全ての機材は e16 実験グループから拝借した。

100 GbE スイッチとして fs.com の S8550-6Q2C を使用した。このスイッチには 40 GbE ポートが 6 個、100 GbE ポートが 2 個ついているが使用したのは 100 GbE ポートのみである。

型番	ASUS ESC4000A-E10
CPU	AMD EPYC 4773 (24 コア)
メモリ	256 GB
NIC	Mellanox MT2892 [ConnectX-6]

表 1: 読み出し PC ハードウェア

型番	DELL PowerEdge T430
CPU	Intel Xeon E5-2630 (10 Core) 2 個
メモリ	128 GB
NIC	Mellanox MT2892 [ConnectX-6]

表 2: データ生成用 PC ハードウェア

使用 OS は AlmaLinux 9 である。またネットワークインターフェイスのドライバは AlmaLinux 9 付属のものを使用した。

3 ハードウェアセットアップ

スイッチと 2 台の PC との接続にはファイバーではなくダイレクトアクセスケーブルを使用した。

100 GbE ポートにケーブルを接続するだけではリンクアップしなかった。スイッチにはメンテナンス用 RJ-45 ポートがあり、LAN ケーブルを接続すると Web ブラウザ経由で各種設定ができるようになる。

ケーブル接続後、100 GbE ポートを 100 GbE として使うと設定、保存したあとスイッチをリブートすると無事 100 Gb/s でリンクアップした。

またフローコントロールの有効化も Web UI で行うことができた。Web UI の他にも ssh あるいは telnet でセットアップ可能であったかもしれないが試してはいない[*1]。

4 PCIe 帯域の確認

100Gb/s でデータが流れることになるので PCIe 帯域がまにあうかどうか確認しておく。

確認には sudo lspc -vv で当該デバイスの LnkSta の項をみればよい。PCIe3 の場合は x1 は 8GT/s = 7876923076 Bps = 7.88 Gbps である。

5 CPU コア番号の把握と L3 共有状態の確認

複数コアがある CPU では L3 キャッシュが一部の CPU コア間でのみ共有していることがある。複数 CPU コアを使って処理する場合、L3 キャッシュを共有しているコアにジョブを投入すると処理速度的に有利になる場合があるので確認しておく。

特別なコマンドを使わなくても、コア番号を N として`cat /sys/devices/system/cpu/cpuN/`

[*1] fs.com の 40GbE スイッチがあるのだが、最近試してみたところ AlmaLinux 9 の ssh クライアントではアクセスできなかった。AlmaLinux 9 から SHA-1 が使えなくなったことが影響しているようだ。かわりに telnet クライアントでアクセスすることはできている。

```
Machine (251GB total)
 Package L#0
  NUMANode L#0 (P#0 251GB)
   L3 L#0 (32MB)
     L2 L#0 (512KB) + L1d L#0 (32KB) + L1i L#0 (32KB) + Core L#0 + PU L#0 (P#0)
     L2 L#1 (512KB) + L1d L#1 (32KB) + L1i L#1 (32KB) + Core L#1 + PU L#1 (P#1)
     L2 L#2 (512KB) + L1d L#2 (32KB) + L1i L#2 (32KB) + Core L#2 + PU L#2 (P#2)
     L2 L#3 (512KB) + L1d L#3 (32KB) + L1i L#3 (32KB) + Core L#3 + PU L#3 (P#3)
     L2 L#4 (512KB) + L1d L#4 (32KB) + L1i L#4 (32KB) + Core L#4 + PU L#4 (P#4)
     L2 L#5 (512KB) + L1d L#5 (32KB) + L1i L#5 (32KB) + Core L#5 + PU L#5 (P#5)
   L3 L#1 (32MB)
     L2 L#6 (512KB) + L1d L#6 (32KB) + L1i L#6 (32KB) + Core L#6 + PU L#6 (P#6)
     L2 L#7 (512KB) + L1d L#7 (32KB) + L1i L#7 (32KB) + Core L#7 + PU L#7 (P#7)
     L2 L#8 (512KB) + L1d L#8 (32KB) + L1i L#8 (32KB) + Core L#8 + PU L#8 (P#8)
     L2 L#9 (512KB) + L1d L#9 (32KB) + L1i L#9 (32KB) + Core L#9 + PU L#9 (P#9)
     L2 L#10 (512KB) + L1d L#10 (32KB) + L1i L#10 (32KB) + Core L#10 + PU L#10 (P#10)
     L2 L#11 (512KB) + L1d L#11 (32KB) + L1i L#11 (32KB) + Core L#11 + PU L#11 (P#11)
   L3 L#2 (32MB)
     L2 L#12 (512KB) + L1d L#12 (32KB) + L1i L#12 (32KB) + Core L#12 + PU L#12 (P#12)
     L2 L#13 (512KB) + L1d L#13 (32KB) + L1i L#13 (32KB) + Core L#13 + PU L#13 (P#13)
     L2 L#14 (512KB) + L1d L#14 (32KB) + L1i L#14 (32KB) + Core L#14 + PU L#14 (P#14)
     L2 L#15 (512KB) + L1d L#15 (32KB) + L1i L#15 (32KB) + Core L#15 + PU L#15 (P#15)
     L2 L#16 (512KB) + L1d L#16 (32KB) + L1i L#16 (32KB) + Core L#16 + PU L#16 (P#16)
     L2 L#17 (512KB) + L1d L#17 (32KB) + L1i L#17 (32KB) + Core L#17 + PU L#17 (P#17)
   L3 L#3 (32MB)
     L2 L#18 (512KB) + L1d L#18 (32KB) + L1i L#18 (32KB) + Core L#18 + PU L#18 (P#18)
     L2 L#19 (512KB) + L1d L#19 (32KB) + L1i L#19 (32KB) + Core L#19 + PU L#19 (P#19)
     L2 L#20 (512KB) + L1d L#20 (32KB) + L1i L#20 (32KB) + Core L#20 + PU L#20 (P#20)
     L2 L#21 (512KB) + L1d L#21 (32KB) + L1i L#21 (32KB) + Core L#21 + PU L#21 (P#21)
     L2 L#22 (512KB) + L1d L#22 (32KB) + L1i L#22 (32KB) + Core L#22 + PU L#22 (P#22)
     L2 L#23 (512KB) + L1d L#23 (32KB) + L1i L#23 (32KB) + Core L#23 + PU L#23 (P#23)
```

図 1: 今回読みだし側で使用した PC の hwloc-ls 出力。6 個の CPU コア (たとえば CPU# 0 から 5) が L3 キャッシュを共有しているなどがわかる。

`cache/index3/shared_cpu_list` すると L3 キャッシュを共有している CPU コアのリストがとれる。

この他の方法として、hwloc パッケージをインストールしておくと hwloc-ls コマンドで CPU L3 キャッシュ共有情報の他に CPU コア番号の付け方が出力される。hwloc-ls の出力中、最後の`(P#N)` の N が`taskset` コマンド、あるいは`sched_setaffinity()` システムコールで指定する CPU コア番号となっている。今回読み出しに使用した PC での実行例を図 1 に示す（Simaltaneous Multi-Threading (Intel 用語では HyperThread) を無効化して起動している)。データ用、コード用の 32kB の L1 キャッシュ、512kB の L2 キャッシュは各 CPU コア毎にあり、L3 キャッシュは 6 個の CPU コアで共有していることがわかる。

さらに hwloc-gui パッケージをインストールしておくと `lstopo` コマンドで、これらの情報を png あるいは pdf ファイルにグラフィカルに取得することができるので便利である。出力例を図 2 に示す。各線の下に小さく書かれている数字は PCIe の帯域幅で、単位は Gbps ではなく GB/s（ギガバイト毎秒）である。

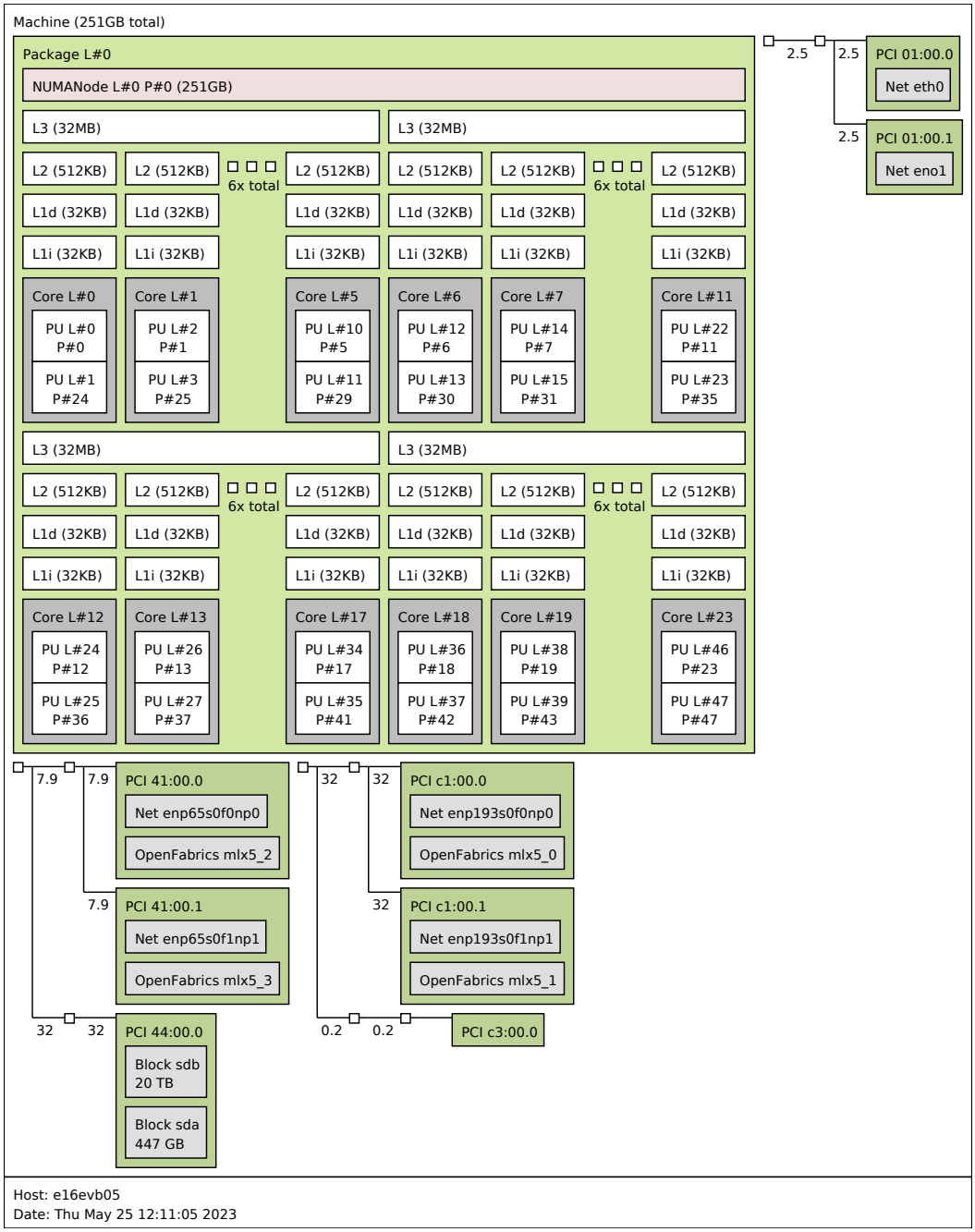


図 2: lstopo コマンドの出力例。6 個の CPU コアが L3 キャッシュを共有しているなどが視覚的にわかる。小さな数字は PCIe の帯域幅（単位: GB/s)。

6 受信キューの構造

6.1 IRQ 番号と対応 CPU コアの設定

サーバー用 1 GbE イーサネットカード (Intel の製品だと Linux のドライバ igb を使うもの) あるいは 10 GbE 以上の速度のイーサネットカードには複数の受信キューをあつかう機能 (Receive

```
           CPU0       CPU1       CPU2 .... CPU23
 109:        113          0        124          0 .... mlx5_comp0@pci:0000:c1:00.0
 110:          0     250581          0          0      mlx5_comp1@pci:0000:c1:00.0
 111:          0          0    6116633          0      mlx5_comp2@pci:0000:c1:00.0
 112:          0          0          0          0      mlx5_comp3@pci:0000:c1:00.0
 113:     459412          0          0          0      mlx5_comp4@pci:0000:c1:00.0
 114:          0   10522154          0          0      mlx5_comp5@pci:0000:c1:00.0
 115:          0          0          0          0      mlx5_comp6@pci:0000:c1:00.0
 116:          0          0          0          0      mlx5_comp7@pci:0000:c1:00.0
 117:          0          0          0          0      mlx5_comp8@pci:0000:c1:00.0
 118:          0          0          0          0      mlx5_comp9@pci:0000:c1:00.0
 119:          0          0          0          0      mlx5_comp10@pci:0000:c1:00.0
 120:          0          0          0          0      mlx5_comp11@pci:0000:c1:00.0
 121:          0          0          1          0      mlx5_comp12@pci:0000:c1:00.0
 122:          0          0          0          0      mlx5_comp13@pci:0000:c1:00.0
 123:     166981          0          0          0      mlx5_comp14@pci:0000:c1:00.0
 124:          0          0          0          0      mlx5_comp15@pci:0000:c1:00.0
 125:          0          0          0          0      mlx5_comp16@pci:0000:c1:00.0
 126:          0          0          0          0      mlx5_comp17@pci:0000:c1:00.0
 127:          0          0          0          0      mlx5_comp18@pci:0000:c1:00.0
 128:          0          0          0          0      mlx5_comp19@pci:0000:c1:00.0
 129:          0          0          0          0      mlx5_comp20@pci:0000:c1:00.0
 130:          0          0          0          0      mlx5_comp21@pci:0000:c1:00.0
 131:          0          0          0     331080      mlx5_comp22@pci:0000:c1:00.0
 132:          0          0          0          3      mlx5_comp23@pci:0000:c1:00.0
 133:          0          0          0          0      mlx5_async24@pci:0000:c1:00.0
```

図 3: /proc/interrupts ファイル。CPU0 から CPU23 まで表示されるが長くなるので途中を省略している。CPU 数が多いとだいぶ横に長い (疑似) ファイルとなっている。`less -S /proc/interrupts` とすると端末上で横にはみでた分は表示されなくなる。はみでた部分をみるには矢印キーを使う。各 CPU コアごとに処理したハードウェア割り込みの回数が書かれている。この数値は単調増加するので、たとえば 1 秒間に起こったハードウェア割り込みを知りたい場合は一度読んで数値を覚えておき、1 秒後に再び読み、前に読んだ数値を引く必要がある。

Side Scaling、 RSS) がついているのがふつうである。

各受信キューには irq 番号が割り当てられる。どの番号が割り当てられたかは`/proc/interrupts` をみればわかる。

今回テストに使用した読み出し PC の`/proc/interrupts` の受信キューに関連する部分を図 3 に示す。この例の場合`mlx5_comp0@pci:0000:c1:00.0` が最初のキューで`mlx5_comp23@pci:0000:c1:00.0` が最後のキューで全部で 24 個のキューがあることがわかる。10 GbE 以上の NIC の場合は NIC にそなわっているキュー数は十分存在していて、CPU の数と使用できるキューの数は同じになることが多い。

`/proc/interrupts` の一番左の数字が irq 番号である。上の例ではキュー`mlx5_comp0@pci:0000:c1:00.0` には irq 番号 109 が割り当てられている。

各 irq には担当 CPU コアが割り当てられている。irq 番号が N の担当 CPU コア番号は proc ファイルの`/proc/irq/N/smp_affinity_list` を読めばわかる。担当 CPU コア番号を変更するにはこのファイルに echo で番号を書けばよい:

```
echo 0 > /proc/irq/109/smp_affinity_list
```

デフォルトでは、各 CPU コア間で割り込み処理を公平に分担させる目的でirqbalance デーモンが起動しており、デフォルトでは 10 秒に 1 回、CPU コア割り当てを変更している (結果的に変更されない場合もある)。手動で irq 担当番号を割り振る場合には、割り振り後、irqbalance デーモンが割り振りを変更することがないようにするために irqbalance デーモンを停止しておく必要がある。

```
systemctl stop irqbalance
```

6.2 パケット到着後の処理

パケットが NIC に到着すると、次のような順序で処理される。

1. NIC はどのキューにいれるか決定し、パケットデータを RX descriptor に書かれた PC 側メモリ領域のキューに DMA 転送する。DMA 転送されるのでこの時点では CPU キャッシュにははいっていない。
2. ある条件が成立すると NIC コントローラはキュー担当 CPU に対して割り込みをかけ (ハードウェア割り込み)、パケット到着を知らせる。
3. OS が割り込み処理を行う。割り込みはすばやく処理する必要がある前段 (ハードウェア割り込み処理) と、あとから行う後段 (ソフトウェア割り込み、softirq) にわかれており、後段を softirq という。前段を処理した CPU コアが後段の処理も行う。おもに後段 (softirq) の部分が CPU 負荷として mpstat などのコマンドで観測される。
4. 最終的に `read()` システムコールを呼んだアプリケーションバッファに格納される。

ハードウェア割り込み処理回数は`/proc/interrupts` をみればわかる。Softirq 処理回数は`/proc/softirqs` でわかる。`/proc/softirqs` の例を図 4 に示す。`NET_RX` が受信に関する softirq を示している。

最初の「どのキューにいれるか」の決定には NIC はパケットの中身をみて IPv4 か、IPv4 なら TCP か UDP か、TCP あるいは UDP ならポート番号は何番かなどをみてハッシュをとり、入れるキューを決める。これで同一 TCP セッションのパケットは同一キューに入るようになり処理速度的に有利になる。

	CPU0	CPU1	CPU2	CPU3	...	CPU6	CPU7
HI:	3	0	3	1		0	0
TIMER:	14581455	9656626	9085047	7103381		3692584	35760650
NET_TX:	101326	4106	482	732		52899	112853
NET_RX:	22345881	69520609	53588093	29710975		21081429	89112590
BLOCK:	2892	26772	1173126	376		1624	260
IRQ_POLL:	0	0	0	0		0	0
TASKLET:	95029	47048	1015	4759		47873	132481
SCHED:	13283051	9375221	9935950	8801665		5950711	35897882
HRTIMER:	50	527	370	264		68	80
RCU:	25349708	17010220	18712137	16316964		10301030	139401504

図 4: /proc/softirqs の例 (今回テストに使用した PC とは別の PC で CPU 数が 8 個のものを示している。

7 一般設定

これまで、経験上、必要になった各種の設定を行う。

7.1 Simultaneous Multi-Threading の無効化

インテルでは HyperThreading Technology といっているものである。各種ベンチマークの結果をみると「Simultanaous Multi-Threading (SMT) を無効化して測定した」というのが散見されたのでここでも SMT は無効化した状態で測定を行った。通常は SMT は有効化されていることが多い。次のいずれかを行うことで無効化できる。

- BIOS、UEFI Firmware で無効化する。
- カーネルコマンドラインに nosmt を追加して起動する。grub.conf の設定で書き換えてもよいし、grub 画面で e を押してその場限りで追加することも可能である。今回は借用物品だったので設定ファイルの変更を避ける方法として、再起動するたびに grub 画面で nosmt を追加した。
- `echo off > /sys/devices/system/cpu/smt/control` を実行する。
- `echo 0 > /sys/devices/system/cpu/cpuN/online` を必要な数だけ繰り返すことで動的に CPU をオフラインにすることができる。N に smt に対応する CPU コア番号を入れる。

7.2 tuned-adm によるチューニング

AlmaLinux 9 では tuned-adm コマンドによりできあいのチューニングをすることができるようになっている。今回は network-throughput (意味は Optimize for streaming network throughput, generally only necessary on older CPUs or 40G+ networks となっている) にセットした。コマンドとしては次のコマンドを実行する。

```
tuned-adm profile network-throghtput
(確認コマンド)
tuned-adm active
```

network-throughput にセットすると次節で述べるソケットレシーブバッファについて、自動調節ありの場合の最大値が 16MB に書き換えられてしまうので、ソケットレシーブバッファの大きさを手動で設定する場合は上記コマンドのあとに行う必要がある。

```
# /proc/sys/net/ipv4/tcp_window_scaling:  1
# /proc/sys/net/ipv4/tcp_timestamps:      1
# /proc/sys/net/ipv4/tcp_moderate_rcvbuf: 1
# 以上の値はデフォルト値。
echo $((1024*1024*1024)) > /proc/sys/net/core/wmem_max
echo $((1024*1024*1024)) > /proc/sys/net/core/rmem_max
echo 4096   131072  $((1024*1024*1024)) > /proc/sys/net/ipv4/tcp_rmem
echo 4096   16384   $((1024*1024*1024)) > /proc/sys/net/ipv4/tcp_wmem
```

図 5: ソケットレシーブバッファサイズの設定。ソケットレシーブバッファの大きさが自動調節される場合の最大値のみを 1 GB に変更している。

7.3 ソケットレシーブバッファ

ソケットレシーブバッファの大きさは図 5 のように最大値のみを変更し、1 GB に設定した (デフォルト値を変更してしまうと動作が変になったことがある)。今回のテストではすべての読み出しで自動調節機能を使って読み出しを行い、getsockopt() で SO_RCVBUF をプログラムで設定するということは行っていない。

/proc/sys/net/core/{w,r}mem_max は setsockopt() でソケットレシーブバッファの大きさを指定するときに必要となる。自動調節機能のみを使う場合は変更する必要はない。

7.4 イーサネットリングバッファ

以下 $NIC にはイーサネットデバイス名 (以前は eth0 だった。昨今は eno0 とか enp193s0f0 だったりする) を指定する。

```
ethtool -g $NIC
```

でイーサネットカードのリングバッファサイズの現在値と最大値 (最大値はハードウェアに依存する) が取得できる。

最大値が 8192 だった場合は以下のように設定する:

```
ethtool -G $NIC rx 8192 tx 8192
```

受信リングバッファ、送信リングバッファが複数ある場合はこれで全てのリングバッファの値が上記の値に設定される。

7.5 イーサネットフローコントロール

イーサネットにはフローコントロールの機能があり、自身のバッファがあふれそうになるとリンクパートナー (PC にイーサネットカードをさしてネットワークスイッチに接続する場合はイーサ

ネットカードのリンクパートナーはネットワークスイッチになる) に対してポーズフレームを送出し、少しの時間、送信を停止するように依頼することができる。ポーズフレーム中には送信を停止する時間の情報が含まれる。ポーズフレームを受信した側が実際に送信を停止するかどうかはこの機能が有効化されているかどうかによる。イーサネットカード側の設定の現在値は次のコマンドで取得できる。

```
ethtool -a $NIC
```

設定するには

```
ethtool -A $NIC tx on rx on
```

とする。

前記のとおりネットワークスイッチに接続している場合はネットワークスイッチ側もフローコントロールを有効にしておく必要がある。今回使用したネットワークスイッチにはメンテナンス用 RJ45 がついていて、LAN ケーブルを接続して Web ブラウザを使って設定することができた。

7.6 割り込みまでの時間の調整

今回は設定しなかったが、ハードウェア割り込み回数を調整する機能がある。設定は

```
ethtool -C $nic rx-usecs NNN
```

として設定する。設定する値 NNN の単位はマイクロ秒で、パケットを受信してからどのくらいの時間ハードウェア割り込みを遅延させるかを設定する。

ドライバによっては NNN が小さい値 (0,1,2,4 等) の場合には、この値はマイクロ秒を意味するのではなく、別の意味を持たせているものもある (Intel NIC 用 e1000e ドライバ、igb ドライバ等)。

7.7 初期状態での計測結果

ここまでセットした状態でのテスト結果を図 6 に示す。最大スループットは 55 Gbps 付近まで到達しているいっぽう 38 Gbps まで下がることもあり、転送レートが安定していない。よくみると、55 Gbps, 50 Gbps, 38 Gbps のように転送レートについていくつかの状態があるようにみえる。

読み出しにかかる CPU 負荷は、おもに

- 読み出しプロセス（ユーザープロセス）
- 受信キューを処理するタスク（カーネルタスク。softirq）

である。

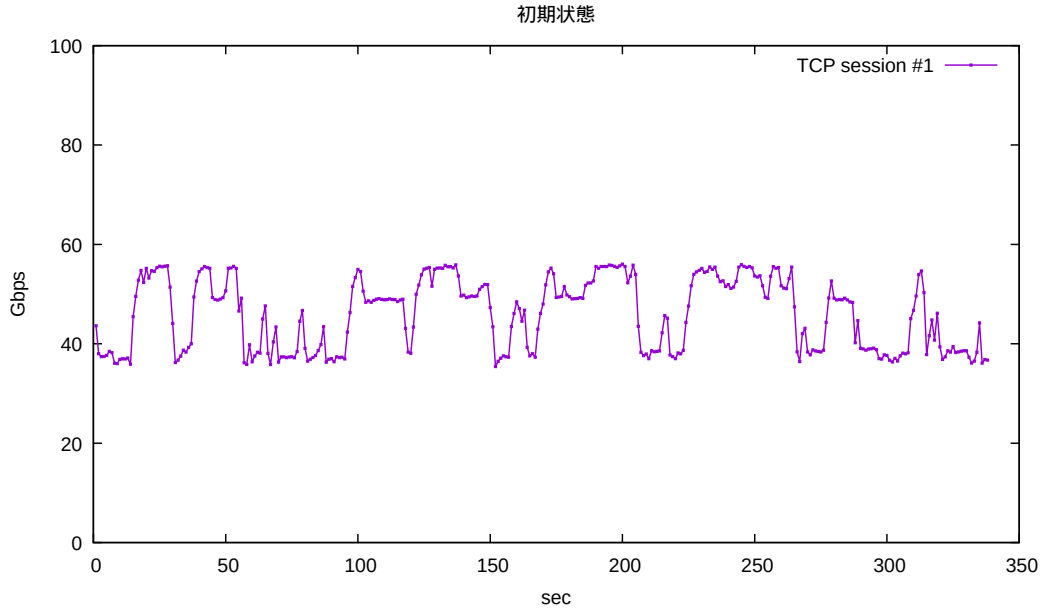


図 6: ここまで設定したときのスループット。

```
15:20:13 CPU   %usr %nice   %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
15:20:14 all   0.12  0.00   4.12    0.00    0.04    4.21    0.00    0.00    0.00   91.50
15:20:14   0   0.00  0.00   0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
15:20:14   1   0.00  0.00   0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
15:20:14   2   0.00  0.00   0.00    0.00    0.00  100.00    0.00    0.00    0.00    0.00
15:20:14   3   0.00  0.00   1.00    0.00    0.00    0.00    0.00    0.00    0.00   99.00
15:20:14   4   0.00  0.00   0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
15:20:14   5   0.00  0.00  97.03    0.00    0.99    0.99    0.00    0.00    0.00    0.99
15:20:14   6   0.00  0.00   0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
15:20:14   7   0.00  0.00   0.00    0.00    0.00    0.00    0.00    0.00    0.00  100.00
```

図 7: mpstat 出力例。全ての CPU コアごとに出力がでる。ここでは最初の 8 個の CPU コアのものだけを示している。%usr はユーザープロセスの非カーネルコードの時間割合、%sys はユーザープロセスのカーネルコードの時間割合、%soft は softirq の時間割合を示している。これが 1 秒に 1 回出力される。この例では CPU コア 5 がユーザープロセスを処理していて、softirq は CPU コア 2 で処理されている。

ユーザープロセスが走行している CPU コアは、スケジューラーにより CPU コアが変更されうる。softirq のほうは irqbalance デーモンが負荷をすべての CPU で平均化するために担当 CPU をときどき変えている (デフォルトでは irqbalance は 10 秒に一度起動して調整をはかる)。読み出しに必要なタスクが走行している CPU コアが移動したことによりタスク間で同一の L3 キャッシュが使えなくなることがあるというのが転送レートが安定しない原因ではないかと考え、各負荷の CPU コア固定化を試みた。

CPU コアごとの負荷状態は`mpstat -P ALL 1` というコマンドで 1 秒おきに取得することができる。このコマンドの出力例を図 7 に示す。

8 受信キュー関連の設定

受信キューを処理する CPU コアとユーザープロセスが走る CPU コアとが L3 キャッシュを共有すると処理速度があがると思われるのでその設定を行う。

8.1 受信キューを処理する CPU コアの固定

まず受信キューの担当 CPU を固定する。

1. irqbalance をとめる
2. `echo 0 > /proc/irq/109/smp_affinity_list` などしてキューを処理する CPU コアを固定する

今回は全部で 24 個ある受信キューに対し、受信キュー 0 に CPU コア 0、受信キュー 1 に CPU コア 1、···、受信キュー 23 に CPU コア 23 を割り当てた。

8.2 ユーザープロセスが走る CPU コアの固定

ユーザープロセスが走る CPU コアの指定には`taskset` コマンド、あるいは`sched_setaffinity()` システムコールが使える。今回は読み出しプログラム中で`sched_setaffinity()` システムコールで固定化した。

このままではネットワークからやってくるパケットがどの受信キューに入れられるか不明なので、パケットを特定のキューに流し込む設定を行う必要がある。これについては次節で述べる。

8.3 特定ネットワークトラフィックを特定キューに流し込む設定

NIC によってはある条件にマッチするパケットを特定のキューに流し込む設定ができるものがある。10 GbE 上の NIC は大抵できるようで今回使用した NIC もできるようになっている。コマンドとしては `ethtool -U` を使う。

```
ethtool -U $nic flow-type tcp4 src-port 1234 action 2
```

とすると tcp4 でソースポートが 1234 の TCP パケットをキュー番号 2 (キュー番号は 0 から始まる) に流し込むことができる。ただし NIC によってはこれを実行するまえに

```
ethtool -K $nic ntuple on
```

を実行する必要がある (今回使用した NIC では必要なかったがたとえば Linux で ixgbe ドライバを使う Intel の 10 GbE が該当する)。

設定の確認は `ethtool -u $nic` で行うことができ、上のように設定すると次のように出力される。

```
Filter: 1023
    Rule Type: TCP over IPv4
    Src IP addr: 0.0.0.0 mask: 255.255.255.255
    Dest IP addr: 0.0.0.0 mask: 255.255.255.255
    TOS: 0x0 mask: 0xff
    Src port: 1234 mask: 0x0
    Dest port: 0 mask: 0xffff
    Action: Direct to queue 2
```

設定を消すには Filter 番号 (上の例だと 1023 番) を指定して delete コマンドを使う。

```
ethtool -U $nic delete 1023
```

NIC によってはパケットを指定する条件に制限があったり、複数個の条件を書くときに条件の種類がそろっている必要があったりする。たとえば Intel ixgbe ドライバのドキュメント (Linux カーネルソースコード`Documentation/networking/device_drivers/ethernet/intel/ixgbe.rst`) には以下の記載がある。

> For each flow-type, the programmed filters must all have the same matching input set.
>
> For example, issuing the following two commands is acceptable::
> ```
> ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
> ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.5 src-port 55 action 10
> ```
> Issuing the next two commands, however, is not acceptable, since the first specifies src-ip and the second specifies dst-ip::
> ```
> ethtool -U enp130s0 flow-type ip4 src-ip 192.168.0.1 src-port 5300 action 7
> ethtool -U enp130s0 flow-type ip4 dst-ip 192.168.0.5 src-port 55 action 10
> ```
> The second command will fail with an error. You may program multiple filters with the same fields, using different values, but, on one device, you may not program two TCP4 filters with different matching fields.

今回使用した NVIDIA Mellanox の NIC にこのような制限があるかどうかは不明である (ソースポートの指定ができればよかったので調べていない)。

以上でパケットを特定のキューに流し込むことができるようになったので、パケット処理を行う CPU コアが固定され、CPU コア間で L3 キャッシュを必ず共有することができるようになる。

送信も受信同様に複数のキューがあるが特定のキューにデータを流し込む機構は用意されていない。

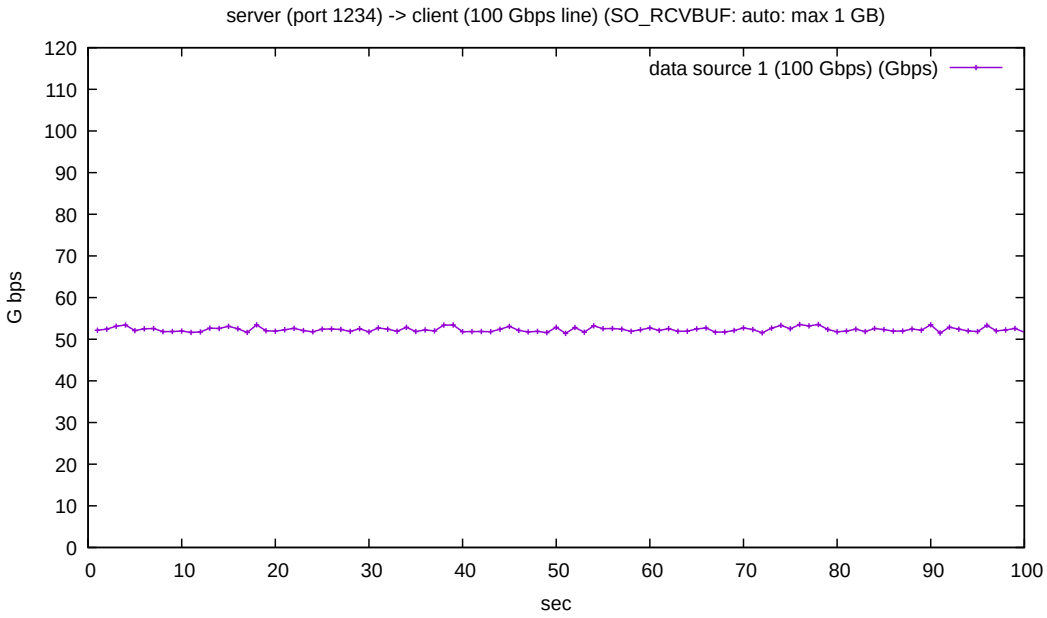
9　テストフローの作成

9.1　L3 を共有する場合

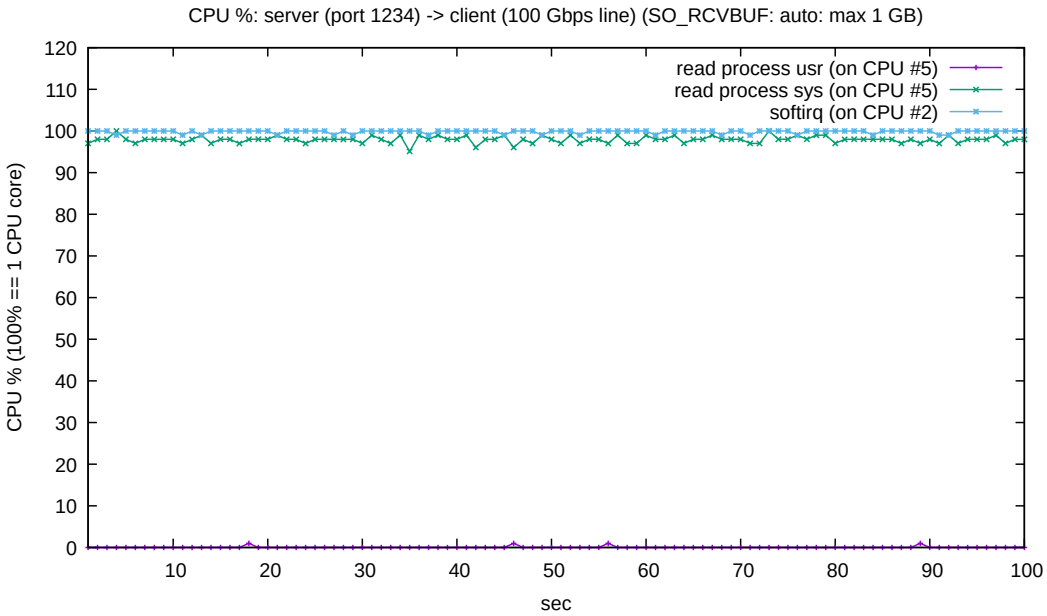
今回のテストではまず安定した受信レートを確立するために受信キューと担当 CPU コア番号の組を設定した。具体的には単純にキュー 0 番と CPU コア 0 番、キュー 1 番と CPU コア 1 番、･･･、キュー番号 23 番と CPU 番号 23 番、とした。

さらに 8.3 節で例として書いた `ethtool -U` コマンドを実行した。これでキュー番号 2 番にやってきたパケットを処理する CPU コアは CPU コア 2 となり、読み出しプロセスを CPU コア 0〜5 のどれかでで動作させると、キューを処理する CPU コアと読み出しプロセスの CPU コアは L3 キャッシュを共有することになる。今回は CPU コア 5 番を使うことにした。

このように設定した場合のデータ転送速度、CPU 消費量を図 8 に示す。データ転送速度の平均は 52.3 Gbps だった。図 6 の CPU コア固定化前の最大速度 55 Gbps にだいたい符合する。同じにはなっていないのでまだなにかあるのかもしれない (未解決)。



(a) 読み出しプロセスが動作する CPU コアとキュー処理 CPU コアを固定し、それぞれの CPU コアが L3 キャッシュを共有するようにセットしたときのスループット。平均 52.3 Gbps。



(b) 読み出しプロセスの sys% と usrCPU 消費量。1 コア全部使っている場合を 100% としてプロットしている。

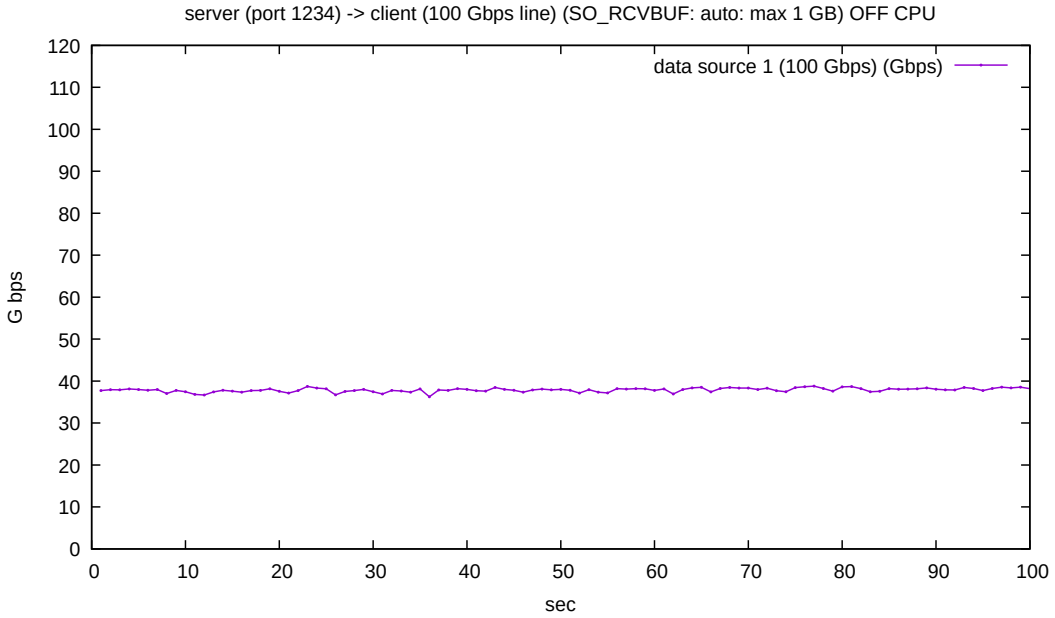
図 8: L3 キャッシュを共有するようにセットした場合。

9.2 L3 を共有しない場合

次に L3 キャッシュの影響をみるため受信キューを処理する CPU コアと読み出しプロセスが走行する CPU コアが L3 キャッシュを共有しない状態にセットしてテストした。それぞれ CPU コア 17、CPU コア 2 で動作するようにセットした。結果を図 9 に示す。

L3 は共有しないがそれぞれの処理を担当する CPU は動かないようになっているので (比較的) 安定した転送レートがえられた。平均は 37.9 Gbps であった。図 6 の CPU コア固定化前の最低速度 38 Gbps に符合する。

L3 キャッシュを共有する場合に比べて 14.4 Gbps の差があった。



(a) 読み出しプロセスが動作する CPU コアとキュー処理 CPU コアを固定し、それぞれの CPU コアが L3 キャッシュを共有しないようにセットしたときのスループット。平均 37.9 Gbps。

(b) 読み出しプロセスの sys% と usrCPU 消費量。1 コア全部使っている場合を 100% としてプロットしている。

図 9: L3 キャッシュを共有しないようにセットした場合。

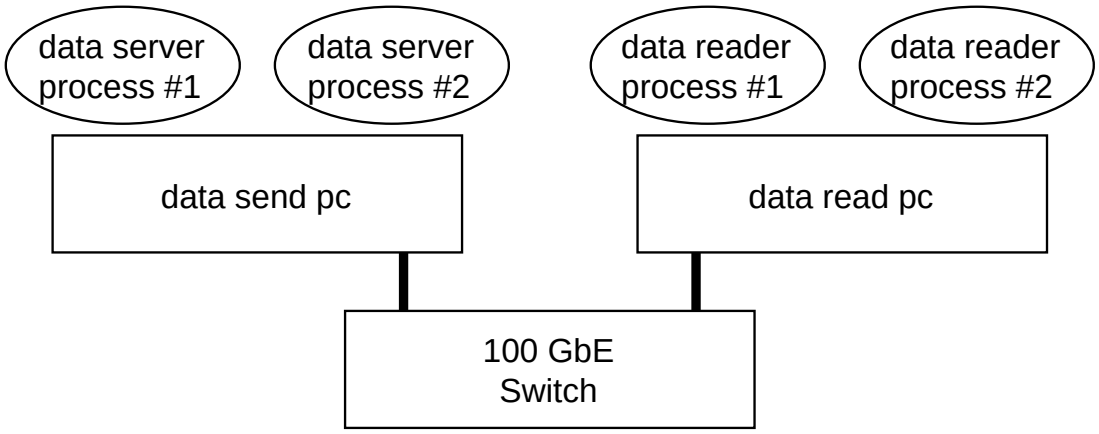


図 10: ふたつの TCP セッション。

9.3 100 Gbps テスト用 TCP フローの作成

1 TCP セッションで 50 Gbps 以上の転送レートがえられたので TCP セッションをもう 1 本はると 100 Gbps 以上となり 100 Gbps スイッチのテストができると考えた。プロセス構成図を図 10 に示す。

受信キュー 1 番と読みだしプロセスその 1 が最初の L3 キャッシュを共有し、受信キュー 2 番と読みだしプロセスその 2 が 2 番目の L3 キャッシュを共有するようにセットして転送レートを測定した。結果を図 11 に示す。2 TCP セッションの合計平均は 93.9 Gbps であった。

今回使用した TCP セッションは、TCP Timestamps オプションを使うようになっている。TCP Timestamps オプションを使う場合ユーザーデータの大きさは 1448 バイトとなるので 100 Gbps の場合は最大 94.1 Gbps となる。

これでスイッチにほぼ最大レートでデータをいれることができ、ほぼ最大レートで出力されることが確認できた。

以上でスイッチが 100Gbps の扱いができるかどうかのテストは終了である。

10 TCP セッション 2 本に参加する CPU で同一 L3 キャッシュを使うテスト

前節の 2 個の TCP セッションでは別々の L3 キャッシュを使ったが、読み出しに参加する計 4 個の CPU 負荷が全て同じ L3 キャッシュを共有する場合も計測してみた。L3 キャッシュが枯渇するかどうか、それにより転送レートが変化するかどうかテストするためである。結果を図 12 に示す。2 個の TCP の合計は 94.0 Gbps であり、L3 キャッシュが不足するということはないという結果だった。

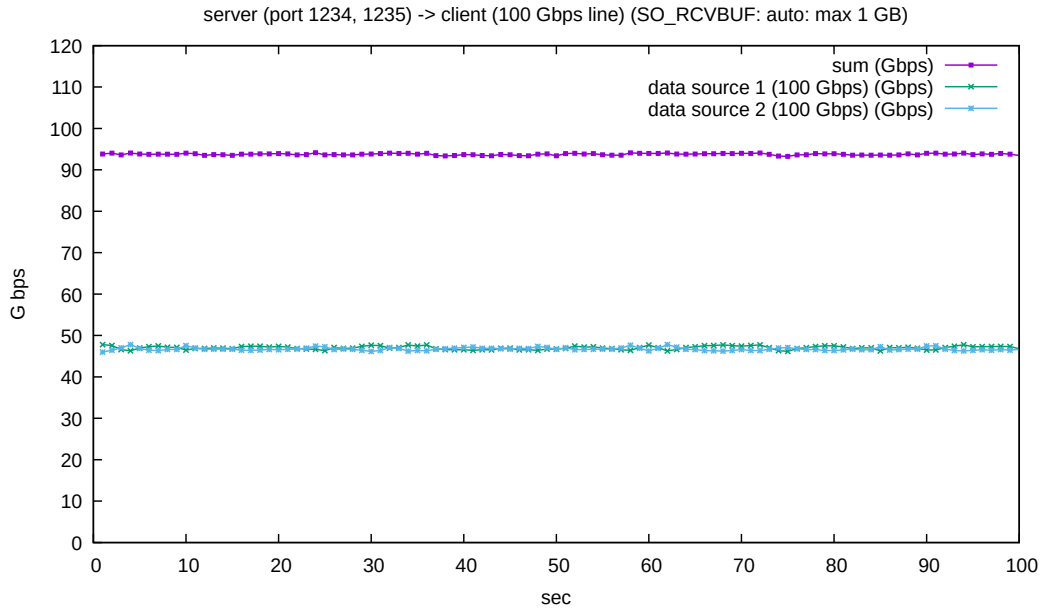


図 11: 2 TCP セッションを利用した場合の結果。2 本ともほぼ同じ転送レートだった。合計の平均は 93.9 Gbps でほぼ最大値が出ていることが確認できた。

11 TCP セッション 3 本に参加する CPU で同一 L3 キャッシュを使うテスト

TCP セッションを 3 本はり、読み出しに参加する負荷を全て同一の L3 キャッシュを共有するというテストも行ってみた。結果を 13 に示す。合計の平均は 94.1 Gbps であった。

帯域を 3 等分して転送している時間帯と、ひとつの TCP セッションがよりはやく転送している時間帯ができている。特定の TCP セッションがはやくなるのではなくて、たとえば 10 秒あたりではやくなっている TCP セッションと 20 秒以降あたりではやくなっている TCP セッションは別のセッションである。L3 キャッシュが不足しているのか、そもそも 3 つの TCP セッションを使うとこのような現象が発生するのかは時間がなくて解決できなかった。

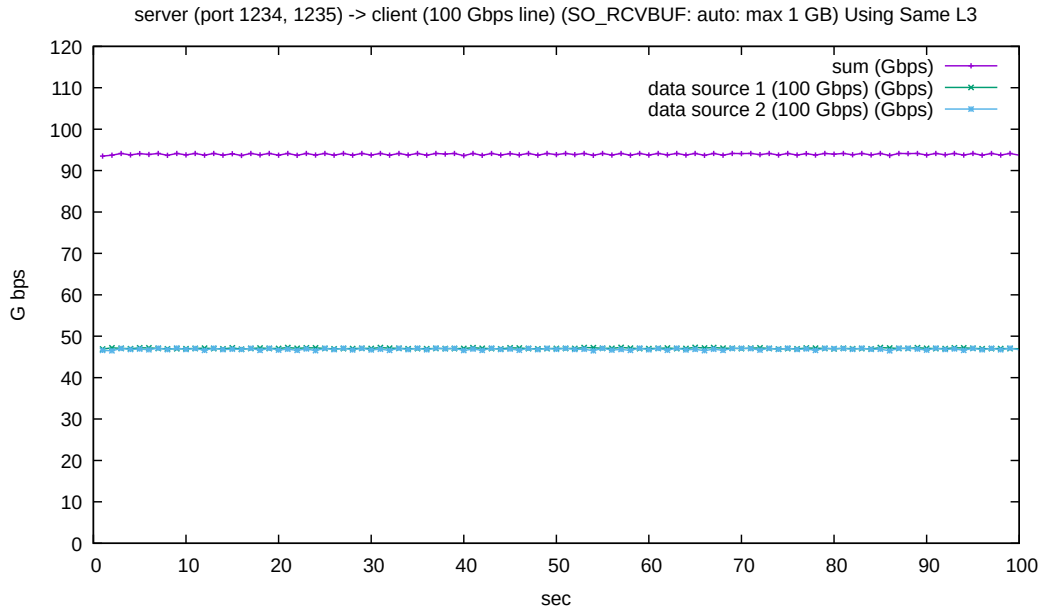


図 12: 2 個の TCP セッションを使い、関連する CPU コアを同じひとつの L3 キャッシュにいれた場合。合計の平均は 94.0 Gbps

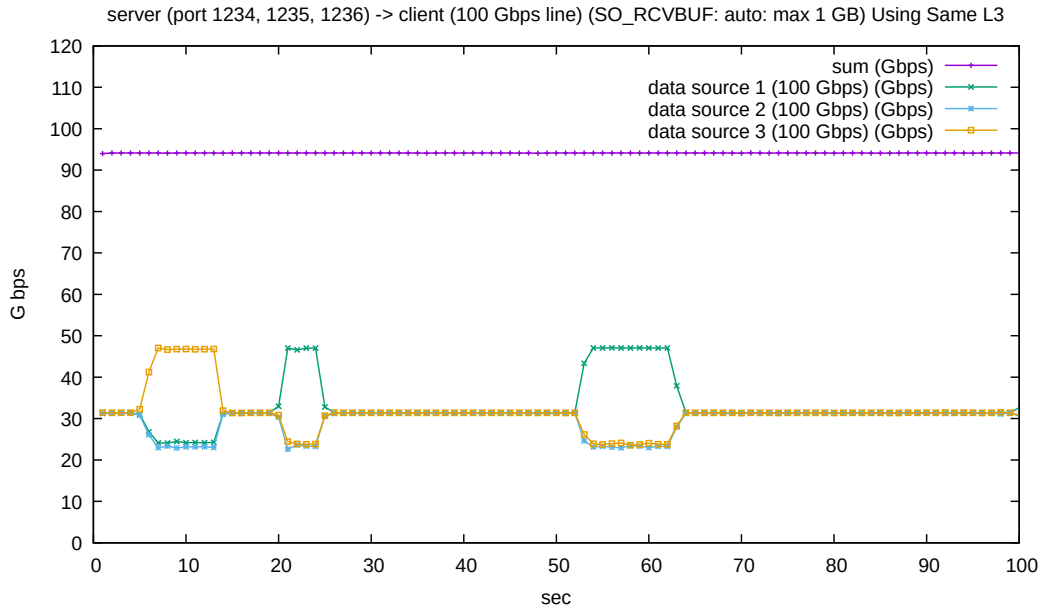


図 13: 3 個の TCP セッションを使いすべて同じ L3 キャッシュを共有するようにした場合。合計の平均は 94.1 Gbps。

12 TCP Timestamps の影響

10 Gbps の SiTCP (XG-SiTCP) では 1 GbE 版にくらべて TCP Window scale オプションがサポートされた。これなしでは PC 側ではラインレートでの受信ができないためである。

PC 間での TCP 通信は TCP Timestamps オプションが有効になっていることが多い。XG-SiTCP も 1GbE SiTCP もまだ TCP Timestamps は実装されていない。

ここで高速 TCP 通信での TCP Timestamps の有無の影響をみるために TCP Timestamps オプションを無効化し 1 個の TCP セッションで転送速度を計測してみた。TCP Timestamps オプションの無効化は次のコマンドを使った。

```
echo 0 > /proc/sys/net/ipv4/tcp_timestamps
```

2 回計測し、その結果を図 14 に示す。それぞれで条件の違いはない。ソケットレシーブバッファの大きさは自動調整する設定で、またソケットレシーブバッファの初期値は 128kB、最大値は 1 GB の設定で読み出しを行った結果である。図には比較のために TCP Timestamps オプションありで計測した図 8 の結果もプロットしている。また転送レート以外にもソケットレシーブバッファの大きさもプロットした。

各図で一番上のライン (縦軸は右) は TCP Timestamps を有効にした場合の読み取り時のソケットレシーブバッファの大きさである。初期値は前述のように 128kB であるがあっというまに大きくなっている。TCP Timestamps オプションありでは 44.5MB まで大きくなっている。TCP Timestamps オプションなしでも初期値からすぐに大きくなるのはかわらないが 4.1MB までしか大きくなっていない。またときどき転送レートの落下が測定された。

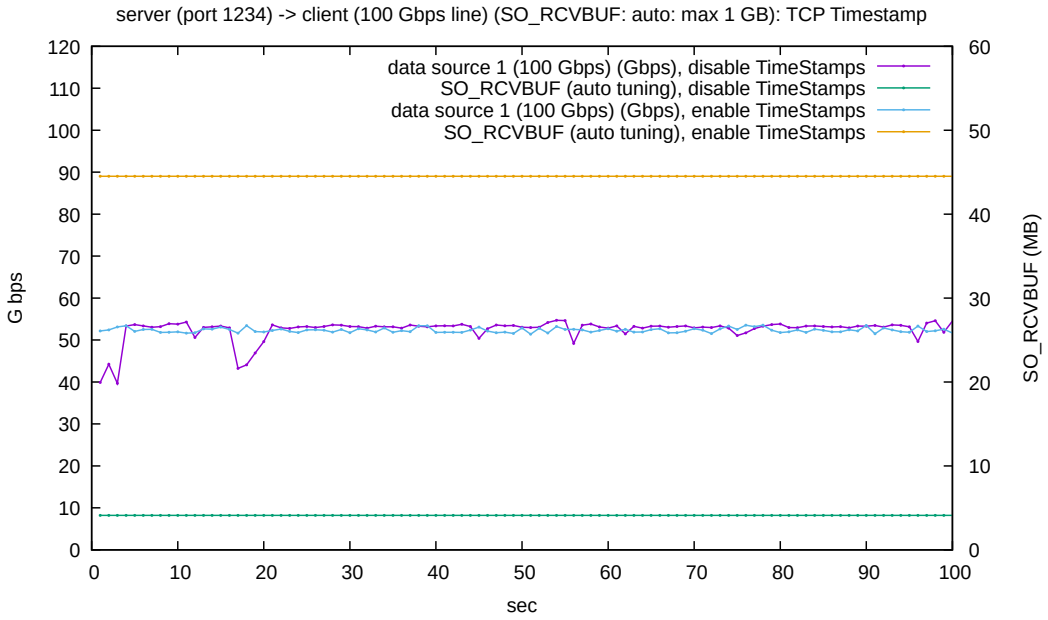
転送レートの落下は同じ条件で測定した 2 回目のほう (図 14b) のほうが顕著である。

TCP Timestamps オプションなしの場合の転送レート低下の原因はいまのところ不明である。

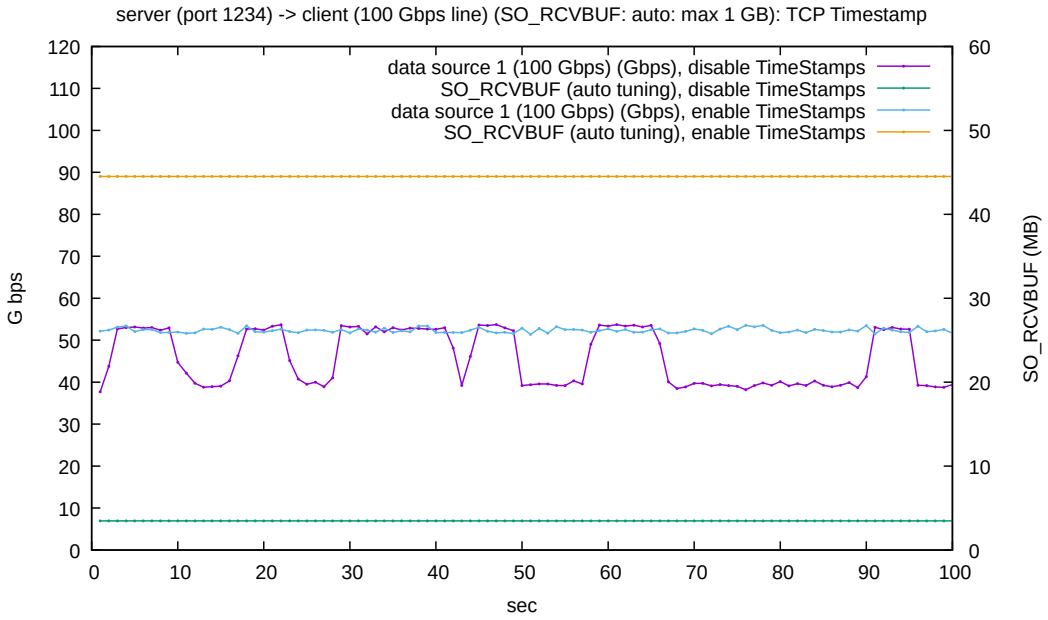
ソケットレシーブバッファが大きくならないことが転送レート低下の原因なら、Timestamps オプションなしでも `setsockopt()` で手動でレシーブバッファの大きさを指定すればいいのではないかと考え、測定してみた。ソケットレシーブバッファを手動で 128MB にセットしている[*2]結果を 15 に示す。3 回測定し、1 回目、2 回目は 52 Gbps 程度出たが、3 回目にときどき微妙に落ちる現象が計測された。原因は不明である。

Timestamps ありなしについては今後機会があれば計測などしてみたい。

[*2] Linux では setsockopt() でソケットレシーブバッファの大きさを設定すると、指定した値の 2 倍の値が設定されるので、setsockopt() には 64MB を指定した。



(a) ラン 1



(b) ラン 2

図 14: TCP タイムスタンプを無効にした場合の転送レート。比較のためにタイムスタンプを有効にした場合のデータもプロットしている。

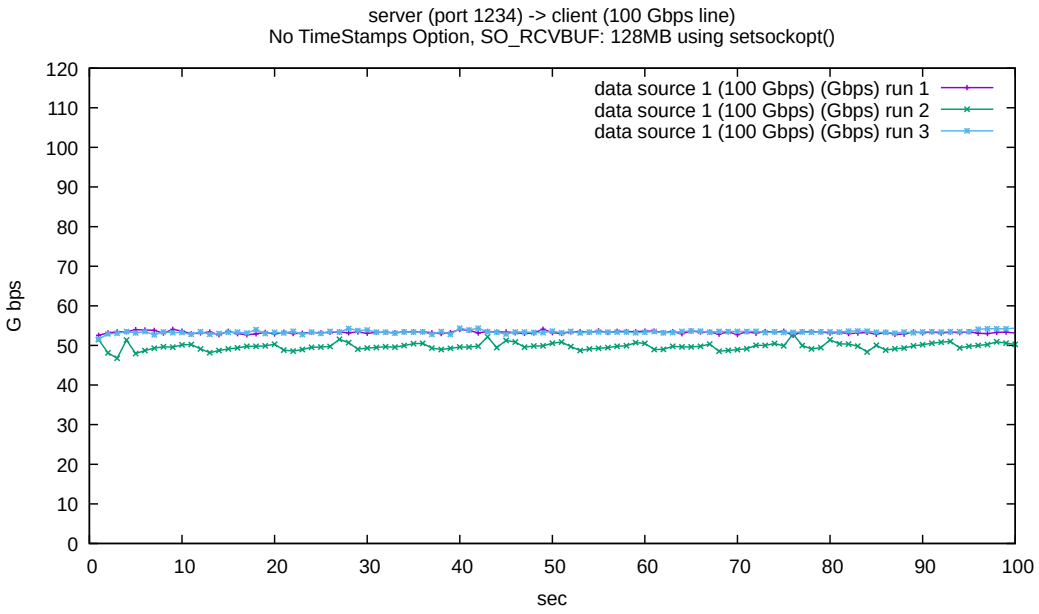


図 15: TimeStamps オプションなし。ソケットレシーブバッファを手動でセット。

付録 A　パケットドロップがないかどうかの確認

NIC で受信するところでパケットを落としていないかどうかの確認は `ethtool -S $nic` で NIC の統計情報をみて確認する。

落としている場合にはフローコントロールの機能を使う、あるいは `ethtool -G` を使ってリングバッファ長を長くする。

softirq の部分で落としてないかどうかは`/proc/net/softnet_stat` をみて確認する。このファイルは CPU コアひとつにつき 1 行のデータとなっている。

最初のコラムは処理したフレーム数を表している。ここでいうフレームとは Generic Received Offload (GRO) あるいは Large Received Offload (LRO) 以降のフレームでこれらが有効になっているとフレーム長は 1500 バイトより大きくみえる (たとえば tcpdump でみると length が長く見える)。

2 番目のコラムは `netdev_max_backlog` にくらべてパケット数が多くてキューに入り切らなかった回数を示している。これが増えているようなら `sysctl` コマンドで`net.core.netdev_max_backlog` の値を増やす (デフォルト 1000)。

3 番目のコラムは softirq 中の `net_rx_action` で poll で制限時間をこえたか budget 数をこえたかで処理を終了したがまだ処理するものがのこっていた回数を示す。これが増えているようなら `sysctl` コマンドで`net.core.netdev_budget` を増やす (デフォルト 600)。

Linux はかなりよくチューニングされているらしく`/proc/net/softnet_stat` の第 2、第 3 コラムの値が増えているという事態に遭遇したことはない。