Speeding up Networking

Van Jacobson
van@packetdesign.com

Bob Felderman
feldy@precisionio.com

Precision I/O

Linux.conf.au 2006
Dunedin, NZ

This talk is not about ‘fixing’ the Linux networking stack

The Linux networking stack isn’t broken.

- The people who take care of the stack know what they’re doing & do good work.
- Based on all the measurements I’m aware of, Linux has the fastest & most complete stack of any OS.

This talk is about fixing an architectural problem created a long time ago in a place far, far away. . .

In the beginning . . .

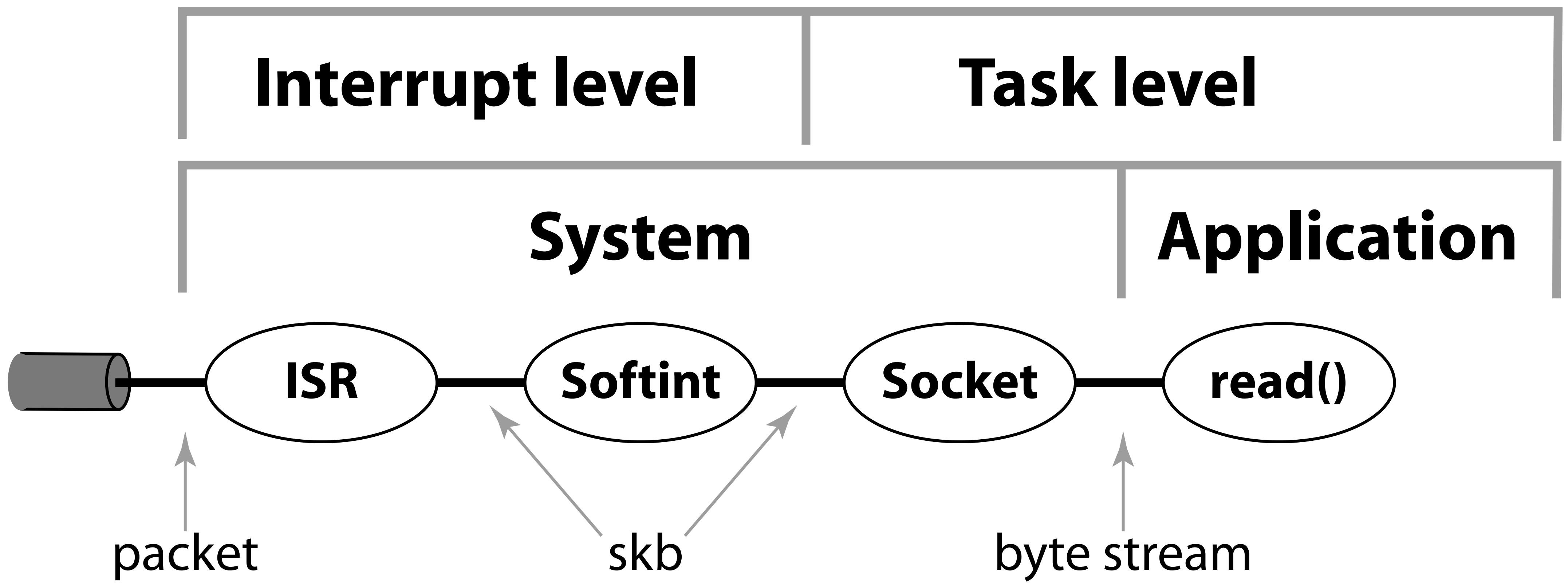
ARPA created MULTICS

- First OS networking stack (MIT, 1970)
- Ran on a multi-user 'super-computer' (GE-640 @ 0.4 MIPS)
- Rarely fewer than100 users; took ~2 minutes to page in a user.
- Since ARPAnet performance depended <u>only</u> on how fast host could empty its 6 IMP buffers, had to put stack in kernel.

The Multics stack begat many other stacks . . .

- First TCP/IP stack done on Multics (1980)
- People from that project went to BBN to do first TCP/IP stack for Berkeley Unix (1983).
- Berkeley CSRG used BBN stack as functional spec for 4.1c BSD stack (1985).
- CSRG wins long battle with University of California lawyers & makes stack source available under 'BSD copyright' (1987).

Multics architecture, as elaborated by Berkeley, became 'Standard Model'



“The way we’ve always done it” is not necessarily the same as “the right way to do it”

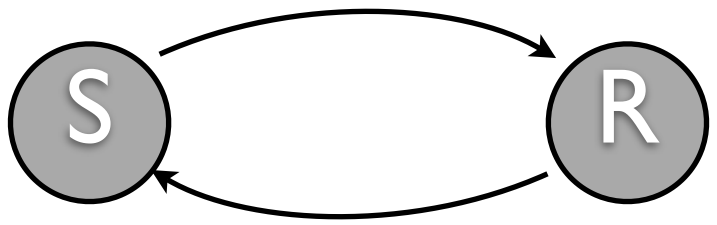
There are a lot of problems associated
with this style of implementation . . .

Protocol Complication

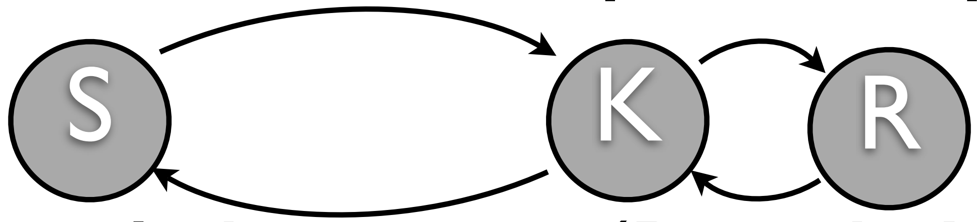
- Since data is received by destination kernel, "window" was added to distinguish between "data has arrived" & "data was consumed".
- This addition more than triples the size of the protocol (window probes, persist states) and is responsible for at least half the interoperability issues (Silly Window Syndrome, FIN wars, etc.)

Internet Stability

- You can view a network connection as a servo-loop:



- A kernel-based protocol implementation converts this to two coupled loops:



- A very general theorem (Routh-Hurwitz) says that the two coupled loops will always be less stable then one.
- The kernel loop also hides the receiving app dynamics from the sender which screws up the RTT estimate & causes spurious retransmissions.

Compromises

Even for a simple stream abstraction like TCP, there’s no such thing as a “one size fits all” protocol implementation.

- The packetization and send strategies are completely different for bulk data vs. transactions vs. event streams.
- The ack strategies are completely different for streaming vs. request response.

Some of this can be handled with sockopts but some app / kernel implementation mismatch is inevitable.

Performance
(the topic for the rest of this talk)

- Kernel-based implementations often have extra data copies (packet to skb to user).
- Kernel-based implementations often have extra boundary crossings (hardware interrupt to software interrupt to context switch to syscall return).
- Kernel-based implementations often have lock contention and hotspots.

Why should we care?

- Networking gear has gotten fast enough (10Gb/s) and cheap enough ($10 for an 8 port Gb switch) that it's changing from a communications technology to a backplane technology.
- The huge mismatch between processor clock rate & memory latency has forced chip makers to put multiple cores on a die.

Why multiple cores?

- Vanilla 2GHz P4 issues 2-4 instr / clock
  $\Rightarrow$ 4-8 instr / ns.
- Internal structure of DRAM chip makes cache line fetch take 50-100ns (FSB speed doesn't matter).
- If you did 400 instructions of computing on every cache line, system would be 50% efficient with one core & 100% with two.
- Typical number is more like 20 instr / line or 2.5% efficient with one core (20 cores for 100%).

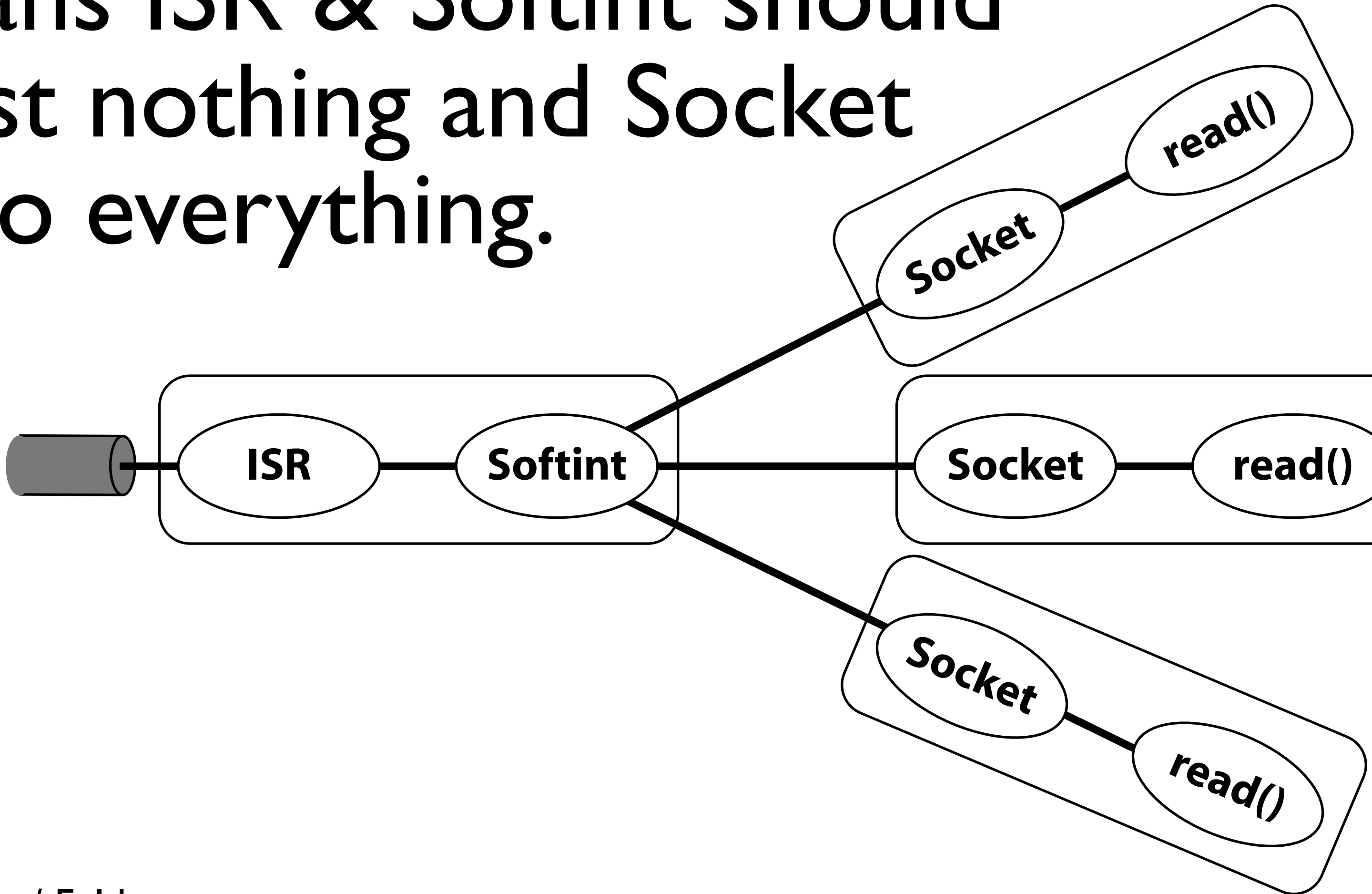
Good system performance comes from having lots of cores working independently

- This is the canonical Internet problem.
- The solution is called the "end-to-end principle". It says you should push all work to the edge & do the absolute minimum inside the net.

The end of the wire isn’t the end of the net

On a uni-processor it doesn’t matter but on a multi-processor the protocol work should be done on the processor that’s going to consume the data.

This means ISR & Softint should do almost nothing and Socket should do everything.

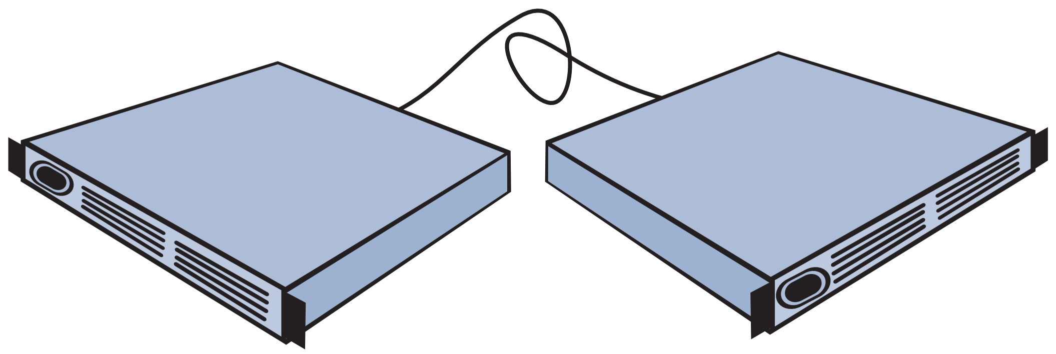


How good is the stack at spreading out the work?

Let’s look at some DaTa

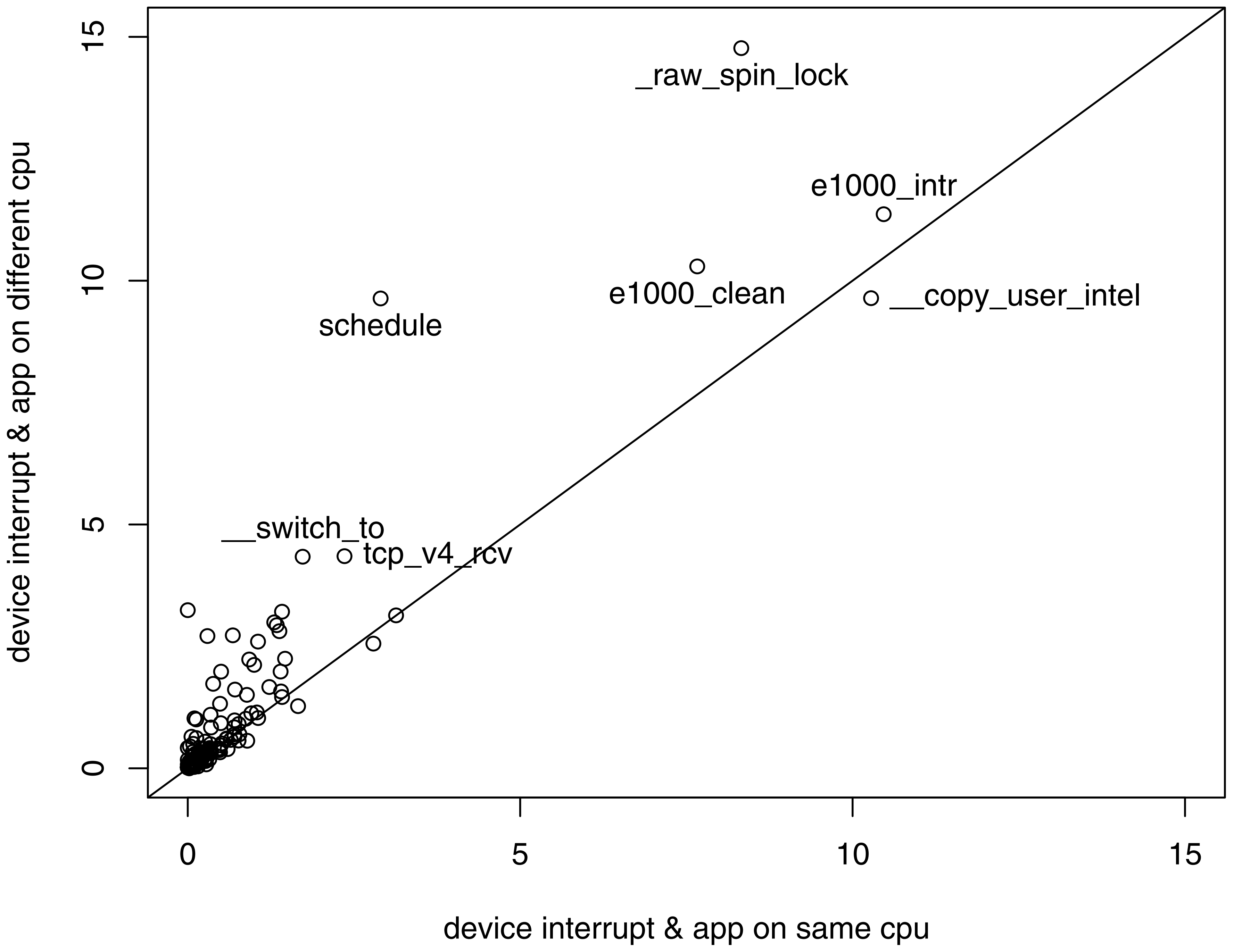
Test setup

- Two Dell Poweredge 1750s (2.4GHz P4 Xeon, dual processor, hyperthreading off) hooked up back-to-back via Intel e1000 gig ether cards.



- Running stock 2.6.15 plus current Sourceforge e1000 driver (6.3.9).
- Measurements done with oprofile 0.9.1. Each test was 5 5-minute runs. Showing median of 5.
- booted with idle=poll_idle. Irqbalance off.

Digression: comparing two profiles



Uni vs. dual processor

- 1cpu: run netserver (netperf) with cpu affinity set to same cpu as e1000 interrupts.
- 2cpu: run netserver with cpu affinity set to different cpu from e1000 interrupts.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1

Uni vs. dual processor

- 1cpu: run netserver (netperf) with cpu affinity set to same cpu as e1000 interrupts.
- 2cpu: run netserver with cpu affinity set to different cpu from e1000 interrupts.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1

This is just Amdahl's law in action

When adding additional processors:

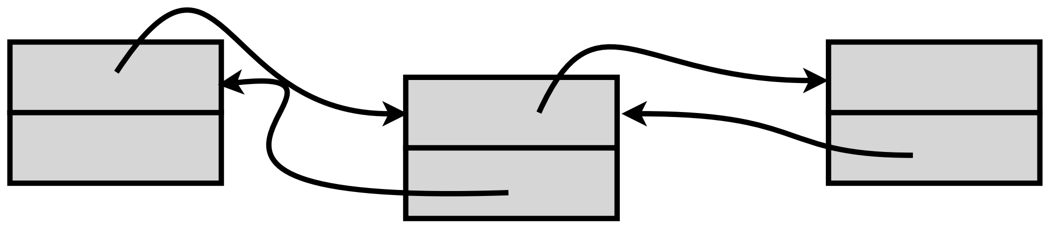
- Benefit (cycles to do work) grows at most linearly.
- Cost (contention, competition, serialization, etc.) grows quadratically.
- System capacity goes as $C(n) = an - bn^2$
  For big enough *n*, the quadratic always wins.
- The key to good scaling is to minimize b.

Locking destroys performance two ways

- The lock has multiple writers so each has to do a (fabulously expensive) RFO cache cycle.
- The lock requires an atomic update which is implemented by freezing the cache.

To go fast you want to have a single writer per line and no locks.

- Networking involves a lot of queues. They’re often implented as doubly linked lists:



- This is the poster child for cache thrashing. Every user has to write every line and every change has to be made multiple places.
- Since most network components have a producer / consumer relationship, a lock free fifo can work a lot better.

net_channel - a cache aware, cache friendly queue

```
typedef struct {
        uint16_t        tail;           /* next element to add */
        uint8_t         wakecnt;        /* do wakeup if != consumer wakecnt */
        uint8_t         pad;
} net_channel_producer_t;

typedef struct {
        uint16_t        head;           /* next element to remove */
        uint8_t         wakecnt;        /* increment to request wakeup */
        uint8_t         wake_type;      /* how to wakeup consumer */
        void*           wake_arg;       /* opaque argument to wakeup routine */
} net_channel_consumer_t;


struct {
        net_channel_producer_t p CACHE_ALIGN;   /* producer's header */
        uint32_t q[NET_CHANNEL_Q_ENTRIES];
        net_channel_consumer_t c;               /* consumer's header */
} net_channel_t ;
```

net_channel (cont.)

```
#define NET_CHANNEL_ENTRIES 512 /* approx number of entries in channel q */

#define NET_CHANNEL_Q_ENTRIES \
            ((ROUND_UP(NET_CHANNEL_ENTRIES*sizeof(uint32_t),CACHE_LINE_SIZE) \
              - sizeof(net_channel_producer_t) - sizeof(net_channel_consumer_t)) \
             / sizeof(uint32_t))

#define CACHE_ALIGN __attribute__((aligned(CACHE_LINE_SIZE)))


static inline void net_channel_queue(net_channel_t *chan, uint32_t item) {
        uint16_t tail = chan->p.tail;
        uint16_t nxt = (tail + 1) % NET_CHANNEL_Q_ENTRIES;
        if (nxt != chan->c.head) {
                chan->q[tail] = item;
                STORE_BARRIER;
                chan->p.tail = nxt;
                if (chan->p.wakecnt != chan->c.wakecnt) {
                        ++chan->p.wakecnt;
                        net_chan_wakeup(chan);
                }
        }
}
```

“Channelize” driver

- Remove e1000 driver hard_start_xmit & napi_poll routines. No softint code left in driver & no skb’s (driver deals only in packets).
- Send packets to generic_napi_poll via a net channel.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1

“Channelize” driver

- Remove e1000 driver hard_start_xmit & napi_poll routines. No softint code left in driver & no skb’s (driver deals only in packets).
- Send packets to generic_napi_poll via a net channel.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1

“Channelize” socket

- socket “registers” transport signature with driver on “accept()”. Gets back a channel.
- driver drops all packets with matching signature into socket’s channel & wakes app if sleeping in socket code. Socket code processes packet(s) on wakeup.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1
sock	28	6	0	16	1	3	1

“Channelize” socket

- socket “registers” transport signature with driver on “accept()”. Gets back a channel.
- driver drops all packets with matching signature into socket’s channel & wakes app if sleeping in socket code. Socket code processes packet(s) on wakeup.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1
sock	28	6	0	16	1	3	1

“Channelize” socket

- socket “registers” transport signature with driver on “accept()”. Gets back a channel.
- driver drops all packets with matching signature into socket’s channel & wakes app if sleeping in socket code. Socket code processes packet(s) on wakeup.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1
sock	28	6	0	16	1	3	1

"Channelize" App

- App "registers" transport signature. Gets back an (mmaped) channel & buffer pool.
- driver drops matching packets into channel & wakes app if sleeping. TCP stack in library processes packet(s) on wakeup.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1
sock	28	6	0	16	1	3	1
App	14	6	0	0	0	2	5

“Channelize” App

- App “registers” transport signature. Gets back an (mmaped) channel & buffer pool.
- driver drops matching packets into channel & wakes app if sleeping. TCP stack in library processes packet(s) on wakeup.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1
sock	28	6	0	16	1	3	1
App	14	6	0	0	0	2	5

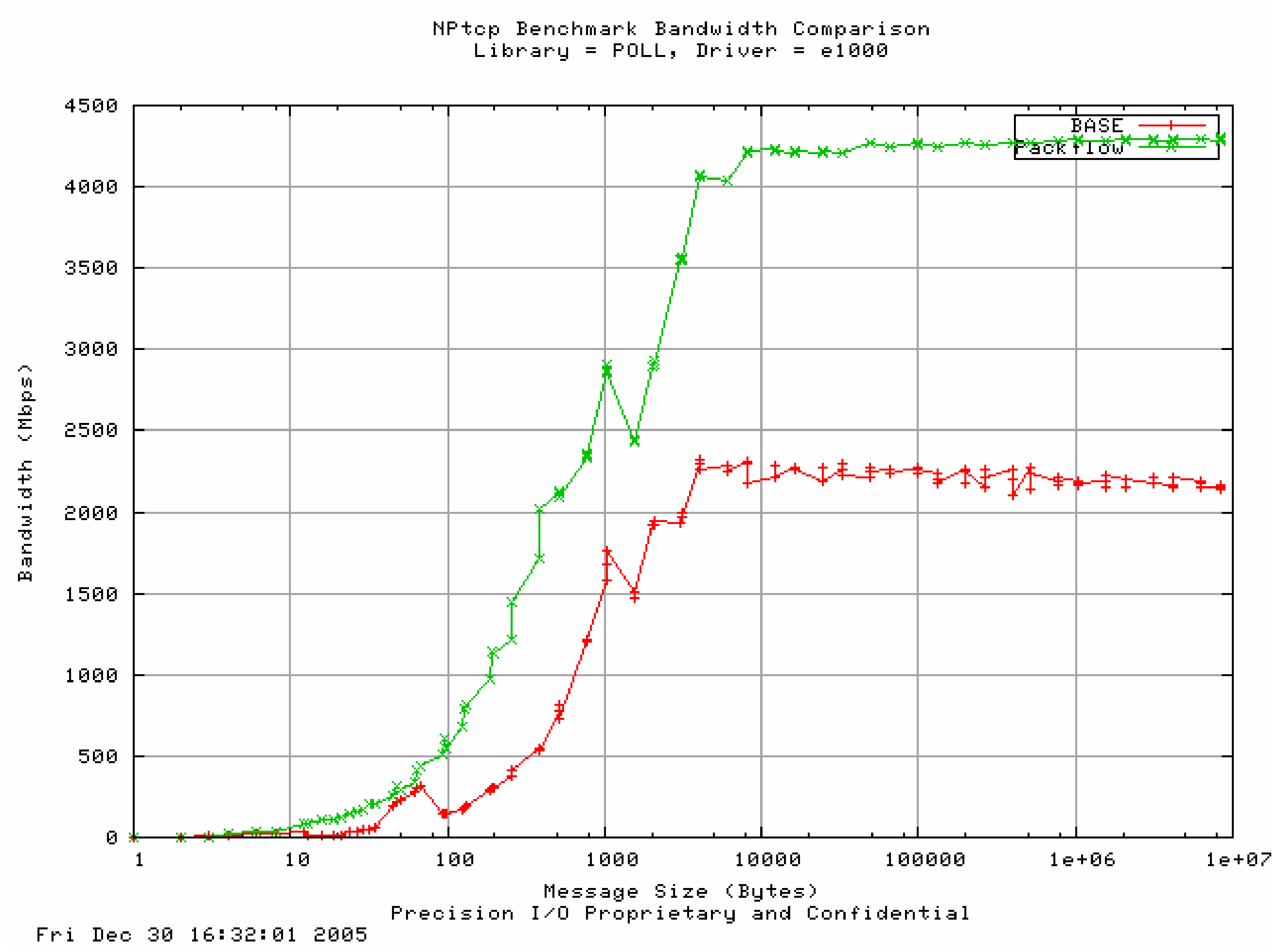
“Channelize” App

- App “registers” transport signature. Gets back an (mmaped) channel & buffer pool.
- driver drops matching packets into channel & wakes app if sleeping. TCP stack in library processes packet(s) on wakeup.

(%)	Busy	Intr	Softint	Socket	Locks	Sched	App
1cpu	50	7	11	16	8	5	1
2cpu	77	9	13	24	14	12	1
drvr	58	6	12	16	9	9	1
sock	28	6	0	16	1	3	1
App	14	6	0	0	0	2	5

10Gb/s ixgb netpipe tests

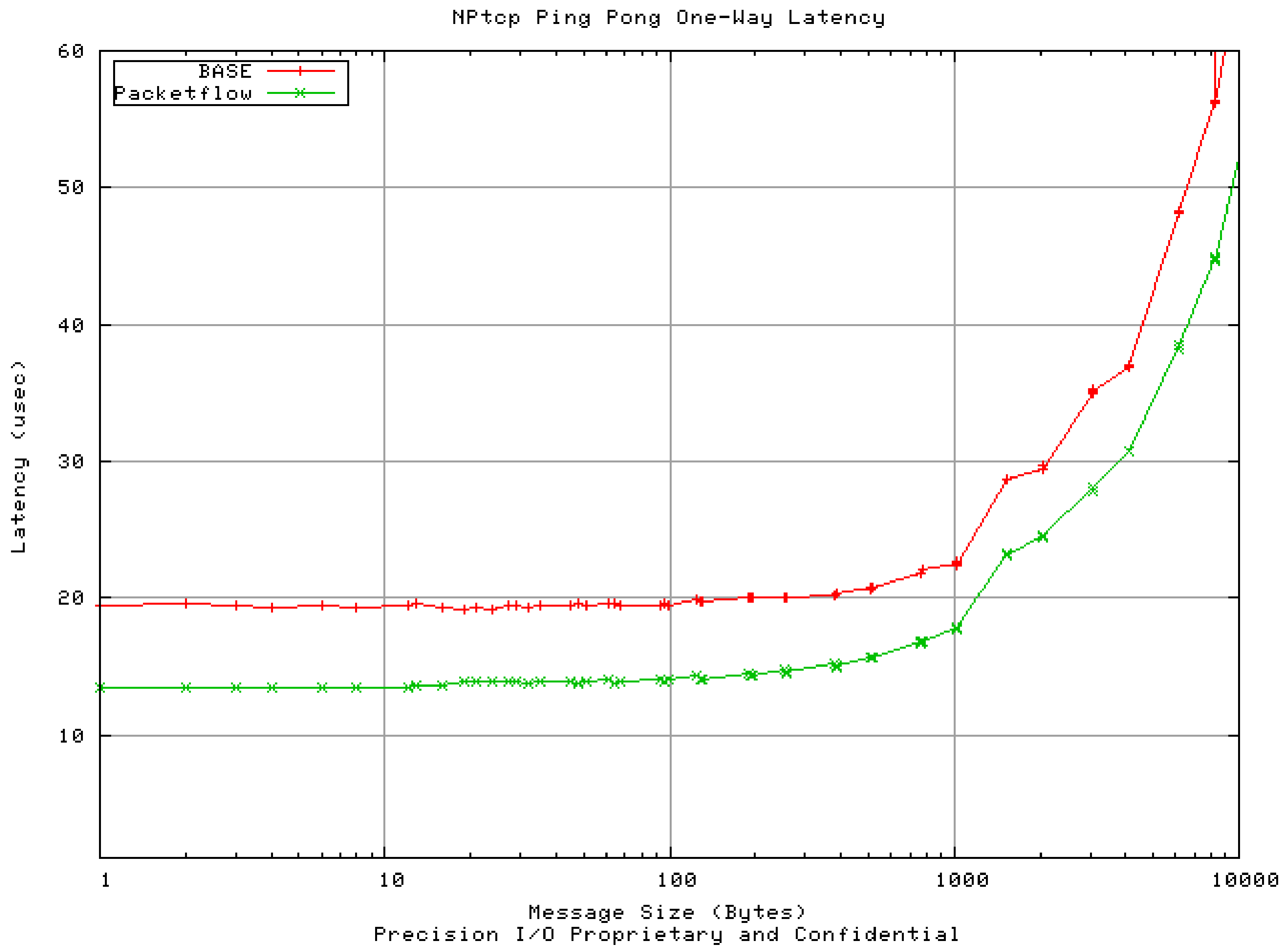
NPtcp streaming test between two nodes.



(4.3Gb/s throughput limit due to DDR333 memory; cpus were loafing)

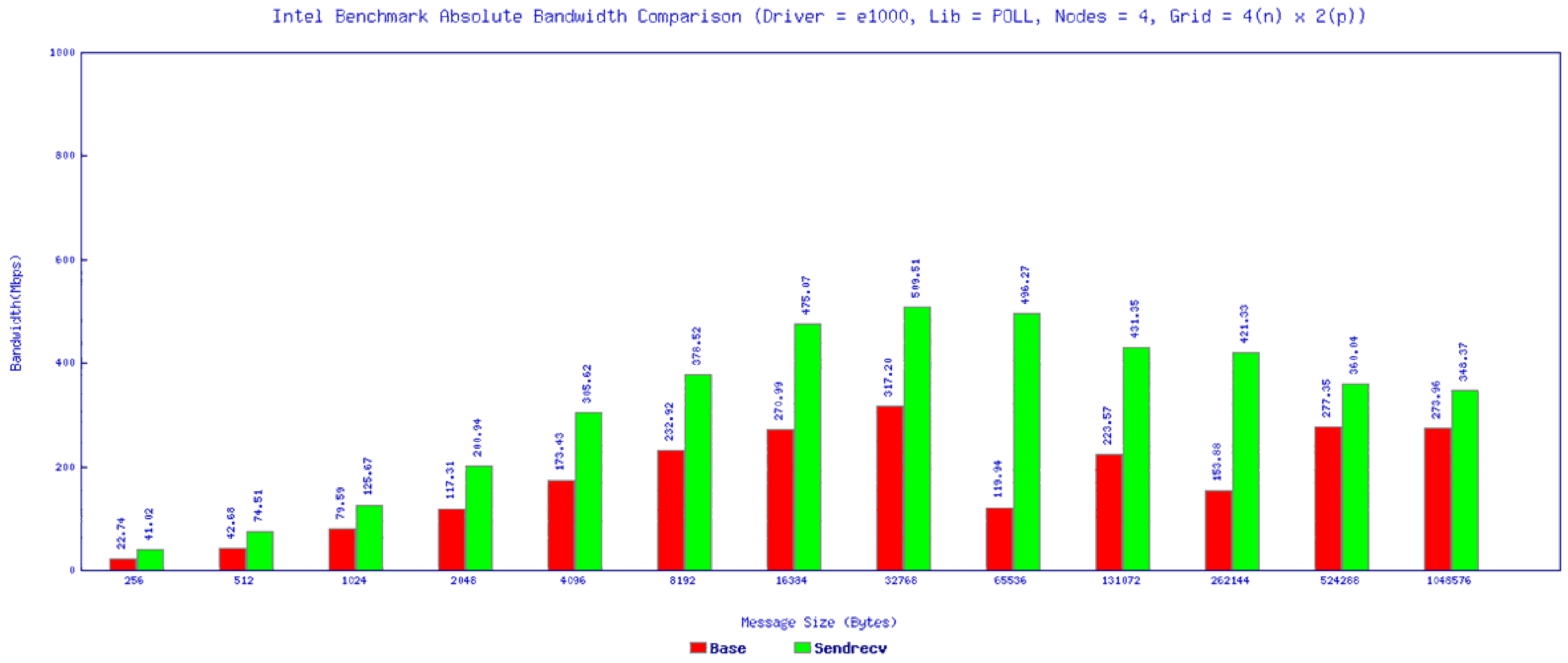
more 10Gb/s

NPtcp ping-pong test between two nodes (one-way latency measured).



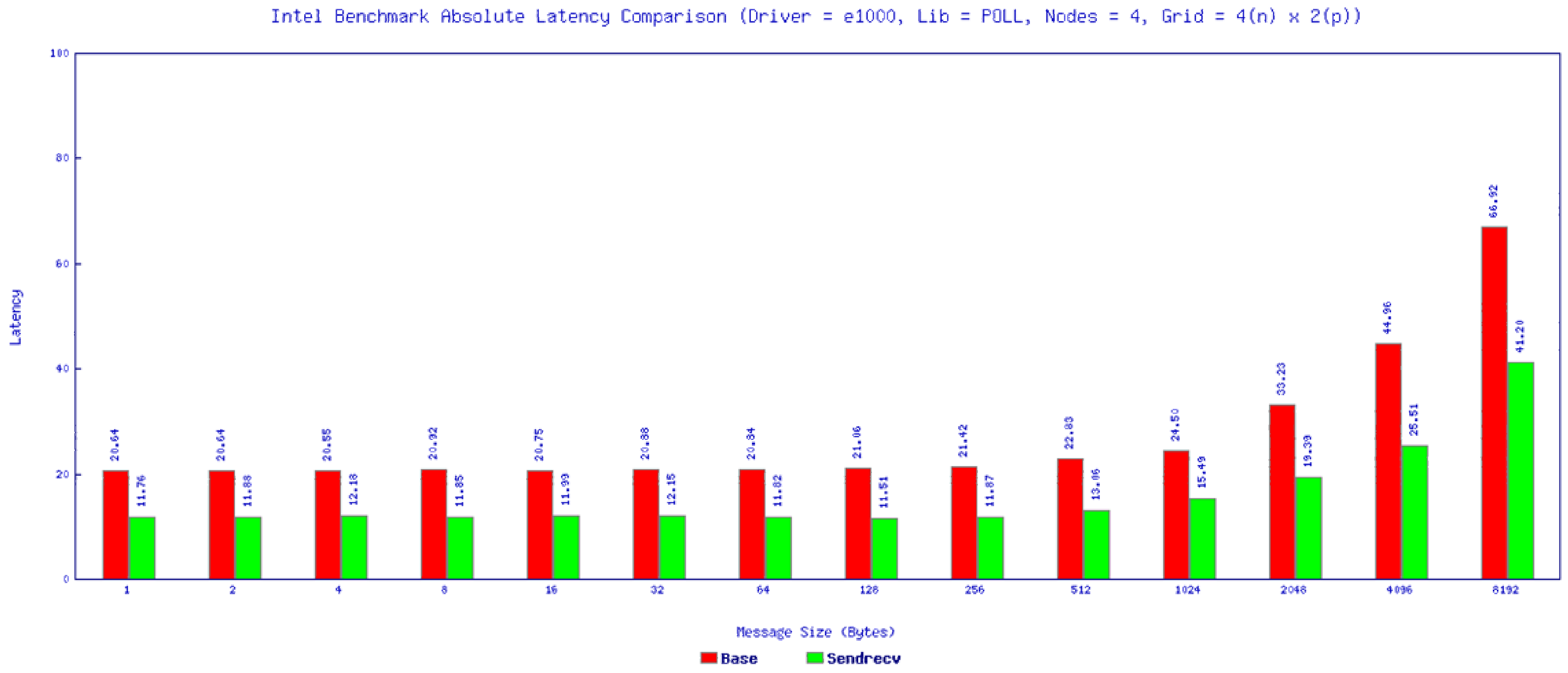
more 10Gb/s

LAM MPI: Intel MPI Benchmark (IMB) using 4 boxes (8 processes)
SendRecv bandwidth (bigger is better)



more 10Gb/s

LAM MPI: Intel MPI Benchmark (IMB) using 4 boxes (8 processes)
SendRecv Latency (smaller is better)



Conclusion

- With some relatively trivial changes, it’s possible to finish the good work started by NAPI & get rid of almost all the interrupt / softint processing.
- As a result, everything gets a lot faster.
- Get linear scalability on multi-cpu / multi-core systems.

Conclusion (cont.)

- Drivers get simpler (hard_start_xmit & napi_poll become generic; drivers only service hardware interrupts).
- Anything can send or receive packets, without locks, very cheaply.
- Easy, incremental transition strategy.