Design Changes to the Kernel Network Architecture for 4.4BSD

Van Jacobson

van@ee.lbl.gov
Lawrence Berkeley Laboratory
Berkeley, CA 94720

4.4BSD Class
Berkeley, CA
May 5 & 7, 1992

What's Different?

4.4 prototype is almost the same as net-2 with a few minor changes:

- mbufs are gone.
- sockbufs are gone.
- netipl is gone.
- sosend / soreceive are gone.
- inpcb's are gone.
- pr_usrreq is gone.
- ip_output, and most of the protocol layering, are gone.
- . . .

Why make changes?

We did a bunch of measurements. Interrupts and memory-to-memory copies are very expensive compared to protocol processing.

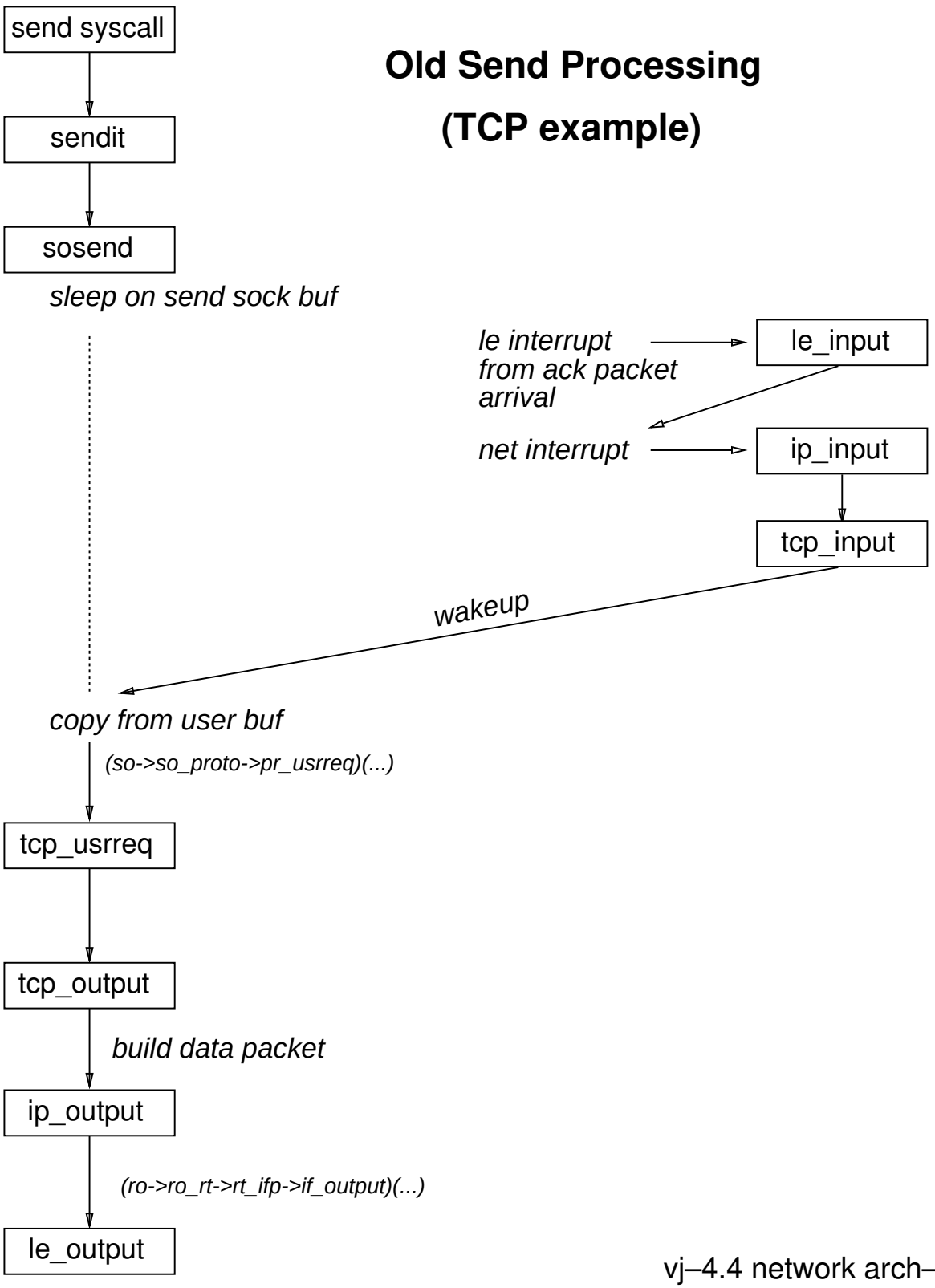
Original code took two interrupts per packet sent or received. New code takes at most one per received packet and zero per sent packet.

Old code's strict layering model caused bad buffering decisions which generated lots of extra memory traffic. New code touches any piece of data exactly once.

Old code's layering model lost perfomance due to lack of parallelism. New code dumps layering and maximizes parallelism.

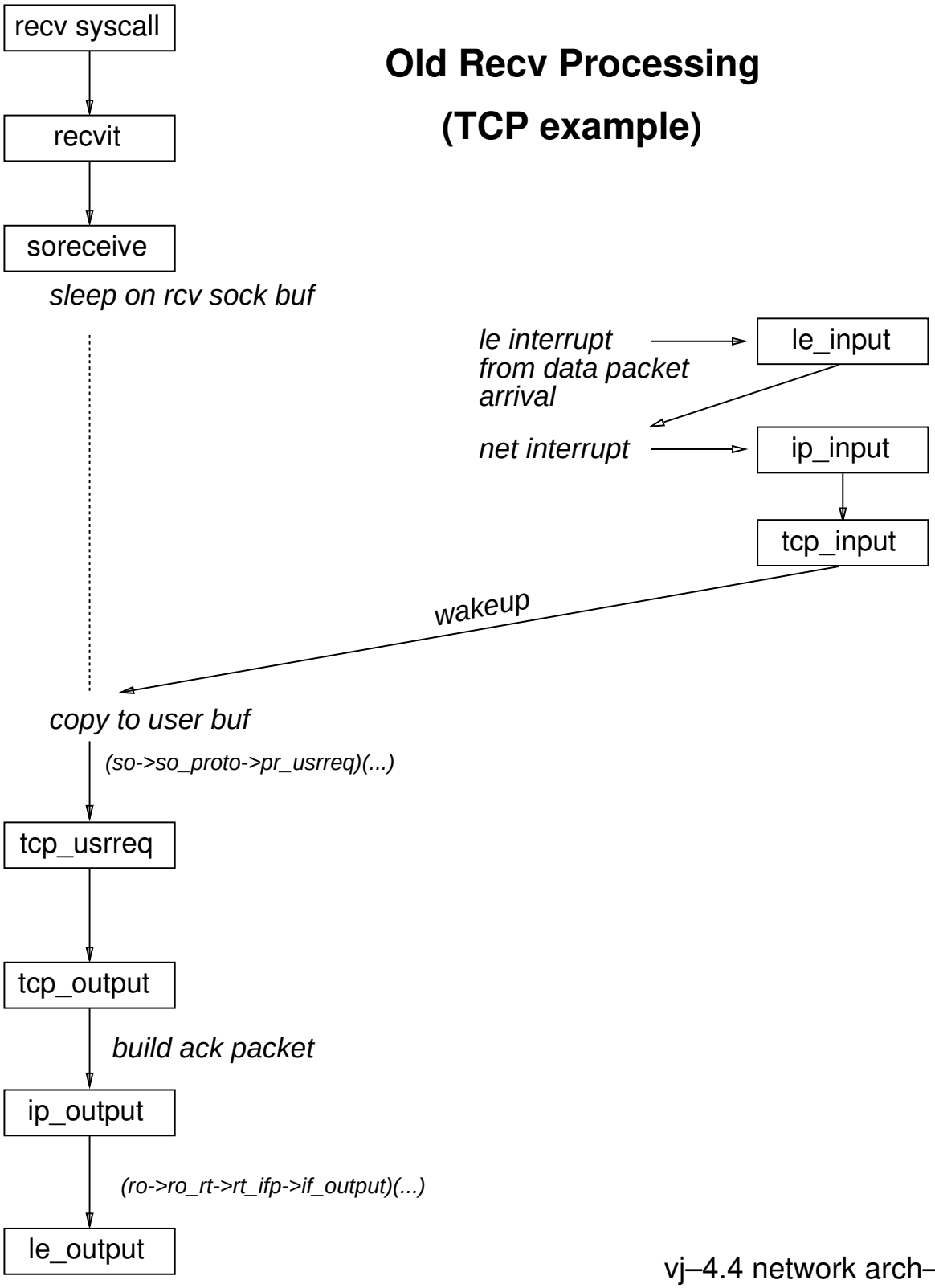
Old Send Processing

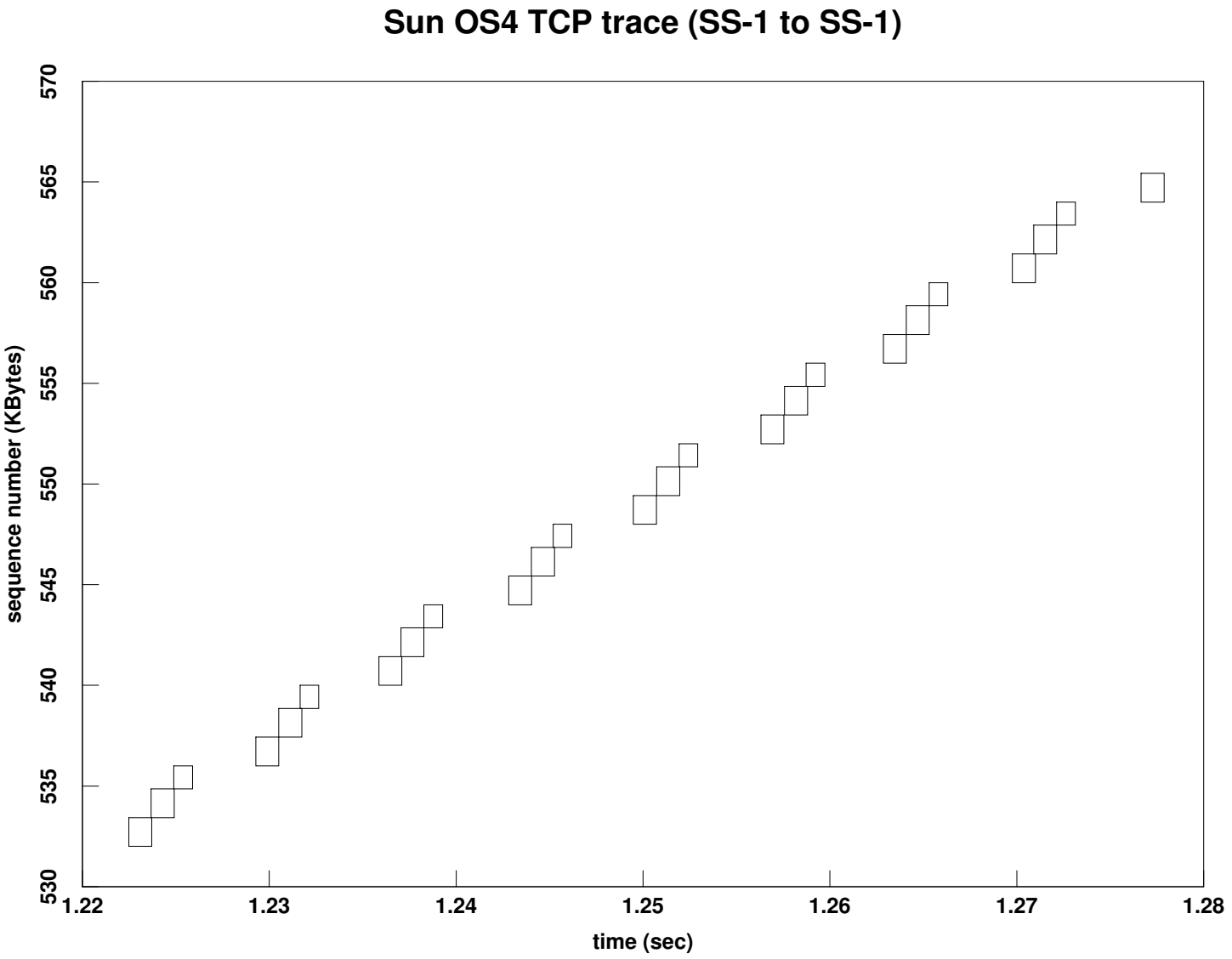
(TCP example)

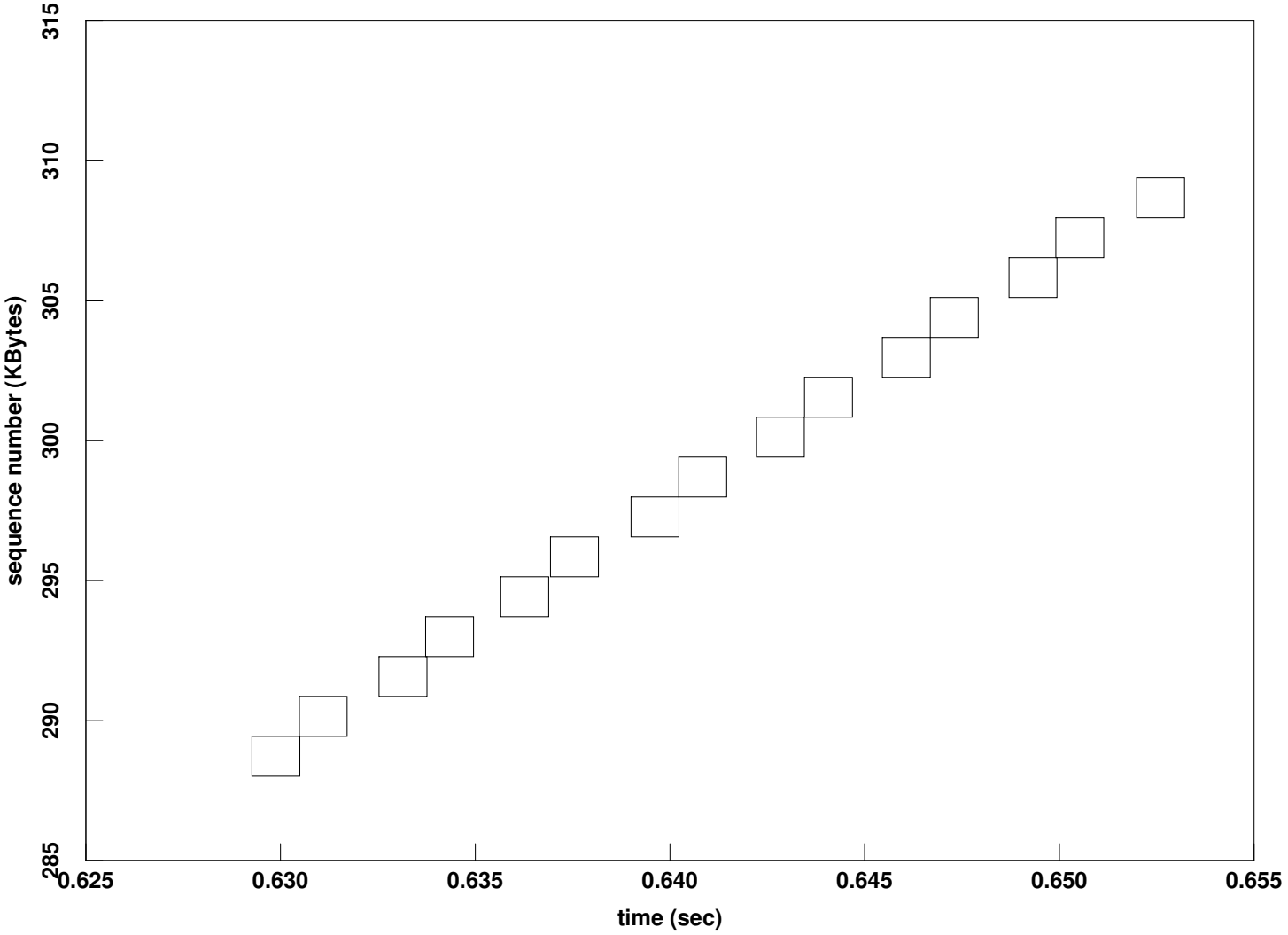


Old Recv Processing

(TCP example)




Sun OS4 TCP trace (SS-1 to SS-1)
sequence number (KBytes)
530
535
540
545
550
555
560
565
570
1.22
1.23
1.24
1.25
1.26
1.27
1.28
time (sec)


Sun OS4 + new sosend TCP trace (SS-1 to SS-1)
sequence number (KBytes)
285
290
295
300
305
310
315
0.625
0.630
0.635
0.640
0.645
0.650
0.655
time (sec)

Layering?

Dave Clark at MIT (and many others) have been saying for the past decade that layered models are a great way to design protocols but a lousy way to implement them. All our measurements support this theory. So . . .

Application level send and receive go to protocol specific routines, not generic routines, that know what packets look like and understand protocol's flow control and reliability policy.

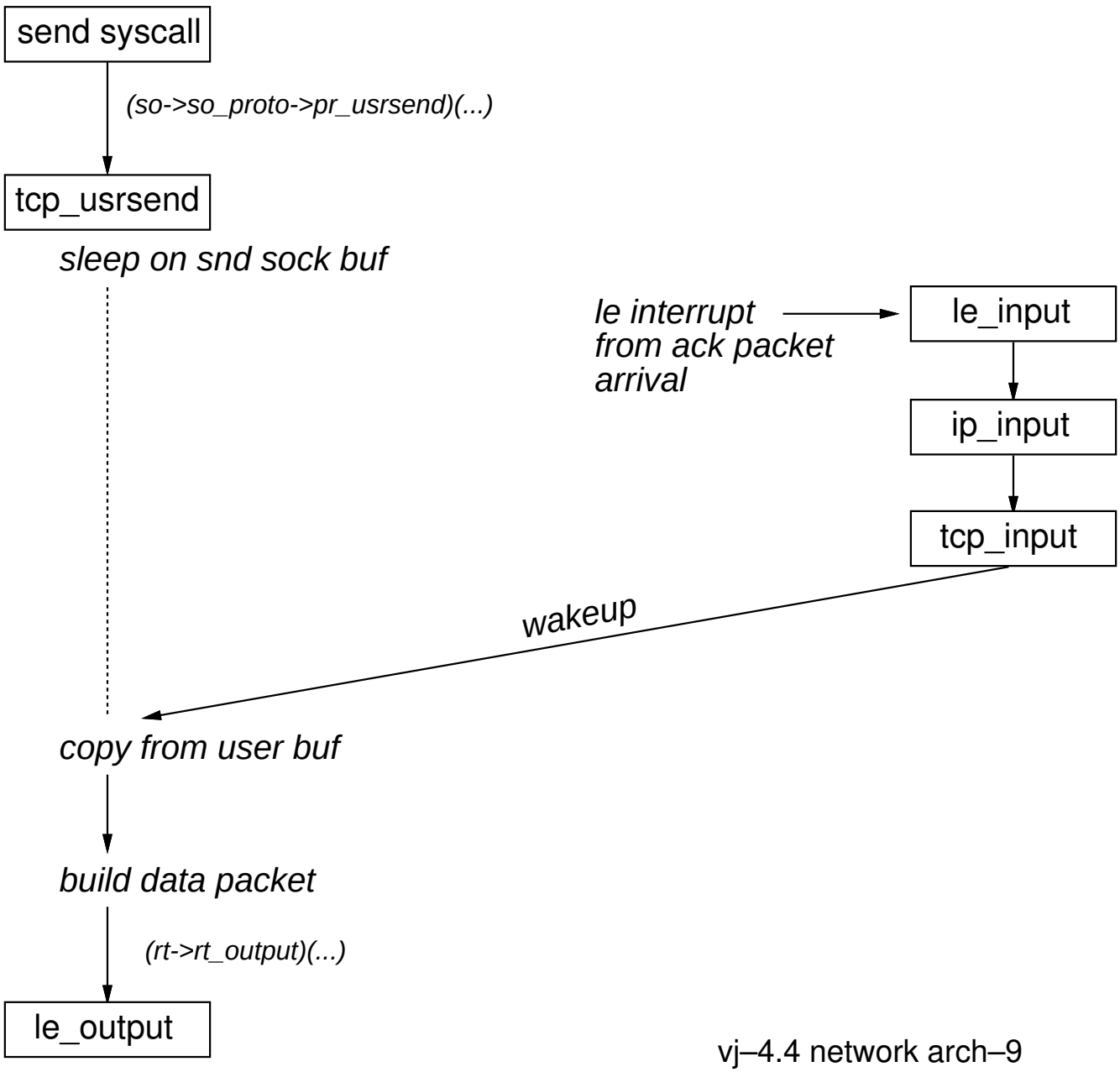
These routines call output driver, not generic allocator, to get packet buffers.

They build and ship one packet at a time to maximize parallelism.

They cache everything that might be useful later and can do most of their packet building with a single bcopy.

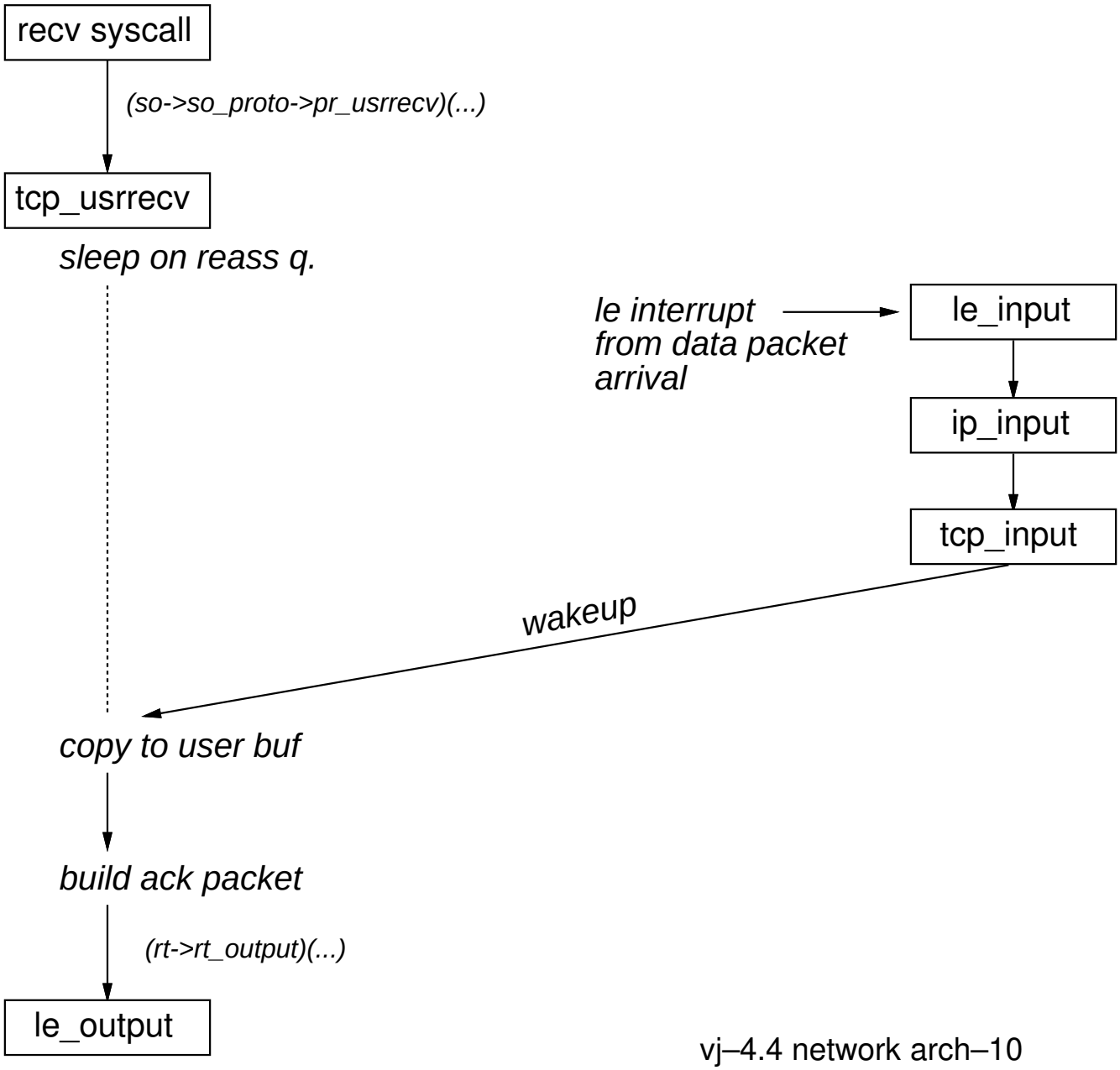
New Send Processing

(TCP example - fast sender)

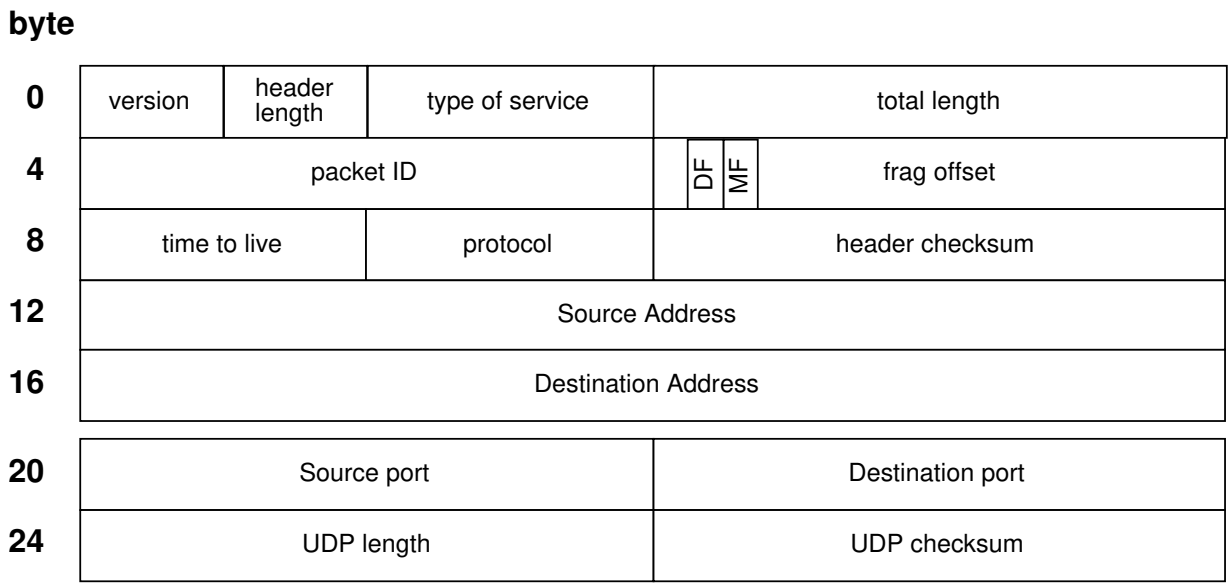


New Recv Processing

(TCP example - fast receiver)



UDP/IP Datagram Format



pr_usrsend example

```
/* Copyright (C) 1992 by Van Jacobson
 * All rights reserved.
 *
 * handle a user "send" or "write"
 * on a "connected" udp socket.
 */
int udp_usrsendc(
        struct socket *so,
        struct mbuf *nam,
        struct uio *uio,
        int flags,
        struct mbuf *rights)
{
        struct udpcb *up = sotoudpcb(so);
        struct route *rt = up->up_rt;
        struct ifnet *ifp = rt->rt_ifp;
        u_int len = uio->uio_resid, hlen = up->up_hdrlen;
        u_int totlen = len + hlen;
        struct pbuf *p;
        struct iphdr *ih;
        struct udphdr *uh;
        int cksum;
```

pr_usrsend example (cont.)

```
        if (totlen > ifp->if_mtu) {
                /* slow path: fragment & send */
                ...
                return (0);
        }
        p = (ifp->if_pget)(ifp, totlen);
        ih = ptod(p, struct iphdr *);
        bcopy(up->up_hdr, (caddr_t)ih, hlen);
        ih->ip_id = ++ip_id;
        ih->ip_len = htons(totlen);
        ih->ip_cksum = htons(~oc_add(ih->ip_cksum,
                                     id_id + totlen));
        cksum = in_uiomovein((caddr_t)ih + hlen, len, uio);
        if (cksum < 0) {
                (p->p_free)(p);
                return (-cksum);
        }
        uh = (struct udphdr *)
                ((caddr_t)ih + hlen - sizeof(*uh));
        len += sizeof(*ih) + sizeof(*uh);
        uh->udp_len = len;
        uh->udp_sum = htons(~oc_add(uh->udp_sum,
                                    len + cksum));
        return (rt->rt_output)(rt, p);
}
```

Copy and Checksum Timings

(all times in ns/byte)

Machine	*bcopy*	*bcopy & cksum*	*cksum cost*
Sun 3/60	133	206	58%
HP9000/370	(68) 122	(97) 144	(43) 18%
Sparcstation-1	(94) 164	(108) 177	(15) 8%
Sparcstation-2	(42) 109	(49) 109	(15) 0%
HP9000/720	(20) 54	(20) 54	(0) 0%

(numbers in paren are for all data in cache)

Buffering model changes

Buffers for outgoing packets allocated by calling interface: *(ifp->if_pget)(ifp, len)*.

Guarantees that packets will be in ‘good’ memory for interface, aligned on ‘good’ boundaries for architecture, contiguous, and that space is left for link level headers.

Interface guarantees same constraints for incoming packets. (in particular, that packet is contiguous).

Contiguous input packets $\Longrightarrow$ all protocol headers are available to any layer (via new
(ifp->if_reparse)(p, level) /
(pr->pr_reparse)(p, level) calls).

Buffers always freed via *(p->p_free)(p)* call.

pbuf's

```
struct pbuf {
        caddr_t  p_data;
        int      p_len;
        struct   pbuf *p_next;
        void     (*p_free)(struct pbuf *);
        struct   ifnet *p_ifp;
        caddr_t  p_packet;
        int      p_pktlen;
        int      p_flags;
};
#define  P_BCAST  0x01
#define  P_MCAST  0x02
```

Driver changes — output

Driver participation in buffer allocation policy drops a lot of code from most driver output routines.

Link level headers cached in route's (no arp table) $\Longrightarrow$ packet setup is just a bcopy.

'Transmit done' interrupts avoided where hardware permits (will recycle output buffers on future pget request).

Driver changes — output (cont.)

```
witless_output(struct pbuf *p, struct rtentry *rt)
{
        u_int outbuf;
        int s;
        struct witless_device *wd =
                        rt->rt_ifp->if_dev_addr;

        outbuf = (u_int)p->p_data -
                        sizeof(struct ether_header);
        *(struct ether_header *)outbuf =
                        *(struct ether_header *)rt->rt_ifcache;
        wd->wd_outbuf = outbuf +
                                p->p_len +
                                sizeof(struct ether_header);
        s = splimp();
        p->p_next = ifp->if_freelist;
        ifp->if_freelist = p;
        splx(s);
        return (0);
}
```

Driver changes — input

Driver calls through protocol stack in device interrupt routine (no more queuing to higher level protocols via netipl).

Higher level protocols either finish processing packet (e.g., ip forwarding) or find a process waiting for input and unblock that process (e.g., tcp_input, udp_input).

Calling rather than queuing means that driver gets a chance to check for new packets after old one processed $\Longrightarrow$ will take only one interrupt for burst of packets.

Driver changes — input (cont.)

```
witless_interrupt(struct ifnet *ifp)
{
        u_int inbuf;
        struct witless_device *wd = ifp->if_dev_addr;

        while (inbuf = wd->wd_inbuf) {
                caddr_t pp = (caddr_t)(inbuf &~ 0x7ff);
                u_int len = inbuf & 0x7ff;
                struct pbuf *p = ifp->if_freelist;
                ifp->if_freelist = p->p_next;
                p->p_next = 0;
                p->p_packet = (caddr_t)pp;
                p->p_pktlen = len;
                p->p_data = pp + sizeof(struct ether_header);
                p->p_len = len - sizeof(struct ether_header);
                switch ((struct ether_header *)pp->eh_type) {
                case ETHERPROTO_IP:
                        ip_input(p, ifp);
                        break;
                case ETHERPROTO_ARP:
                        arp_input(p, ifp);
                        break;
                  ...
                }
        }
}
```

IP input and packet forwarding

This hash table gives the route to the next hop for packets we forward. It also tells up whether the packet is destined for this host or should be forwarded: If the *rt* pointer is zero, it means the destination address is this host (i.e., the address of one of our interfaces or a multicast or broadcast address).

For policy or TOS routing, the cache key needs to be larger (i.e., *ip_src* and/or *ip_tos*). Other than that, no changes are required.

Note that the other kernel routing code knows that this cache exists and will take appropriate steps to keep it consistent.

```
#define HASH_BITS 11
#define HASH_SIZE (1 << HASH_BITS)
#define HASH(dst) (((dst) ^ ((dst) >> HASH_BITS))
                   & (HASH_SIZE-1))

struct ipfwd_cache {
        u_long   dst;      /* ip address of final destination */
        struct rtentry *rt; /* route to get there (forwarding) */
} ipfwd_cache[HASH_SIZE];
```

IP input and packet forwarding (cont.)

IP input handling: Check packet for consistency. Process any IP options. Pass to higher layer if for us, otherwise try to forward.

```
ip_input(p, ifp)
        register struct pbuf *p;        /* buffer containing IP pack
        register struct ifnet *ifp;     /* interface packet arrived
{
        register struct ip *ip = ptod(p, struct ip *);
        register u_int len = p->p_len;
        register u_int dst;
        register struct ipfwd_cache *cache;
        register struct rtentry *rt;
        register int i;
        extern int ip_fill_fwd_cache();
```

IP input and packet forwarding (cont.)

```
/* make sure there's at least an IP header of data */
if (len < sizeof (struct ip)) {
        ipstat.ips_tooshort++;
        goto bad;
}
/* make sure length we received agrees with IP length */
if (len != ntohs(ip->ip_len)) {
        /*
         * length doesn't agree: if didn't rcv enough, discard.
         * Otherwise, trim excess.
         */
        if (len < ntohs(ip->ip_len)) {
                ipstat.ips_toosmall++;
                goto bad;
        }
        len = ntohs(ip->ip_len);
        p->p_len = len;
}
```

IP input and packet forwarding (cont.)

Check IP version number & header length. Min length header with correct version is special-cased (we know we won't have to deal with source route or other IP options).

```
if (*(char *)ip != ((IPVERSION << 4) | sizeof(struct ip)/4)) {
        /* discard if IP version isn't 4 */
        if (ip->ip_v != IPVERSION) {
                ipstat.ips_badver++;
                goto bad;
        }
        /* discard if hdr len > received length or < min */
        i = ip->ip_hl << 2;
        if (i > len || i < sizeof(struct ip)) {
                ipstat.ips_badhlen++;
                goto bad;
        }
        if ((dst = ip_do_options(p, ip)) == 0)
                /* some problem with options */
                goto bad;

} else
        dst = ip->ip_dst.s_addr;
```

IP input and packet forwarding (cont.)

```
/* find (or create) cache entry for this destination */
cache = &ipfwd_cache[HASH(dst)];
if (cache->dst != dst) {
        if (ip_fill_fwd_cache(p, dst, cache))
                /* destination unreachable */
                goto bad;

        /* if going out same interface, redirect originator. */
        if (ifp == rt->rt_ifp)
                ip_send_redirect(p, ip, rt);
}
```

IP input and packet forwarding (cont.)

```
if (rt = cache->rt) {
        /* packet not for us - forward it */
        /* update time-to-live and checksum */
        i = ip->ip_ttl;
        if (--i <= 0) {
                ip_send_time_exceeded(p);
                return;
        }
        ip->ip_ttl = i;
        i = (int)ip->ip_sum + 256;
        ip->ip_sum = i + (i >> 16);
        ifp = rt->rt_ifp;
        if (ifp->if_mtu < len) {
                (fragment and send packet)
        } else if (p = (ifp->if_pmove)(ifp, p))
                (rt->rt_output)(rt, p);
} else {
        i = ip->ip_hl << 2;
        p->p_data += i;
        p->p_len -= i;
        (ip_protosw[ip->ip_p])(p, ip);
}
return;
```

Measurements?

The current prototype networking code on a SPARC says:

- The total cost for IP to forward a packet is 37 instructions and 13 memory references (11 loads and 2 stores).
- The total cost to process the protocol in a TCP datagram is <60 instructions and 22 memory references.
- For modest datagram sizes (≥ 1 KB), TCP/IP runs at the speed of main memory.
- Poor performance is almost always the fault of baroque, complex, poorly integrated, network hardware.

Where does the time go?

Say we're receiving back-to-back 4KB FDDI packets (one packet every 330μs).

On a Sparcstation-2 (40MHz clock or 25ns/instr.):

	instr	μs
TCP + IP + ARP	100	3
Interrupt entry/exit	600	25
DMA 4KB into memory	–	164
copy 4KB to application	1000	446