Speeding up Protocols for Small Messages

Trevor Blackwell
Harvard University

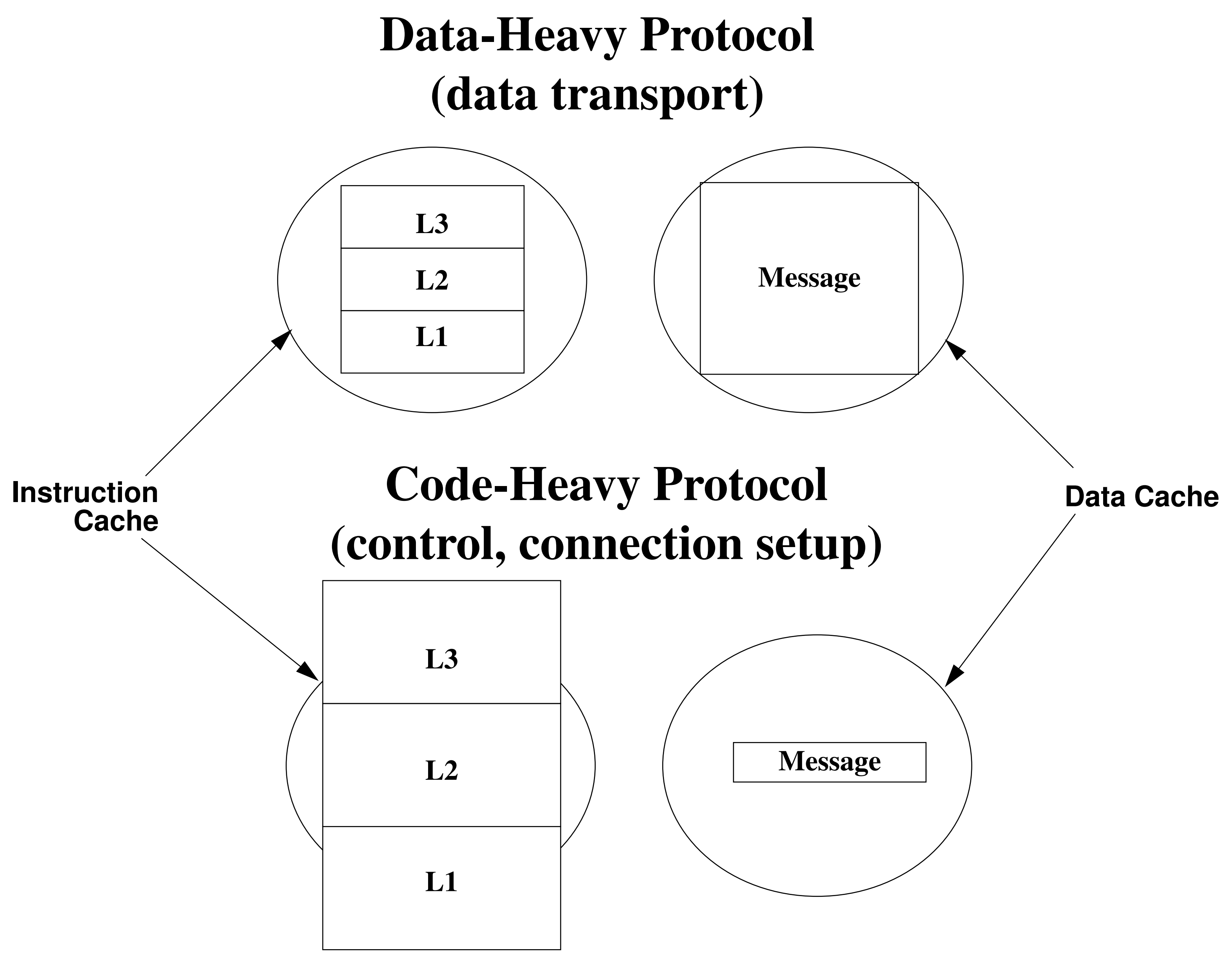
Outline: Memory Performance of Control Protocols

- Memory system performance
- Code-heavy vs. Data-heavy protocols
- Measurements of TCP/IP
- Locality-Driven Layer Processing
- Implementation
- Synthetic Benchmark, Results
- Relation to BSD Network Implementation
- Conclusions

Background: Memory System Performance

- **Hardware:**
  - **Instructions get cheaper and cheaper**
  - **Fetching data or code from off-chip doesn't**
- **Kernel hasn't gotten faster the way SPEC performance has because:**
  - **Moves large amounts of data**
  - **Large working set of instructions**
- **Overall software architecture affects memory system performance**

Code-Heavy vs Data-Heavy



Measuring TCP/IP

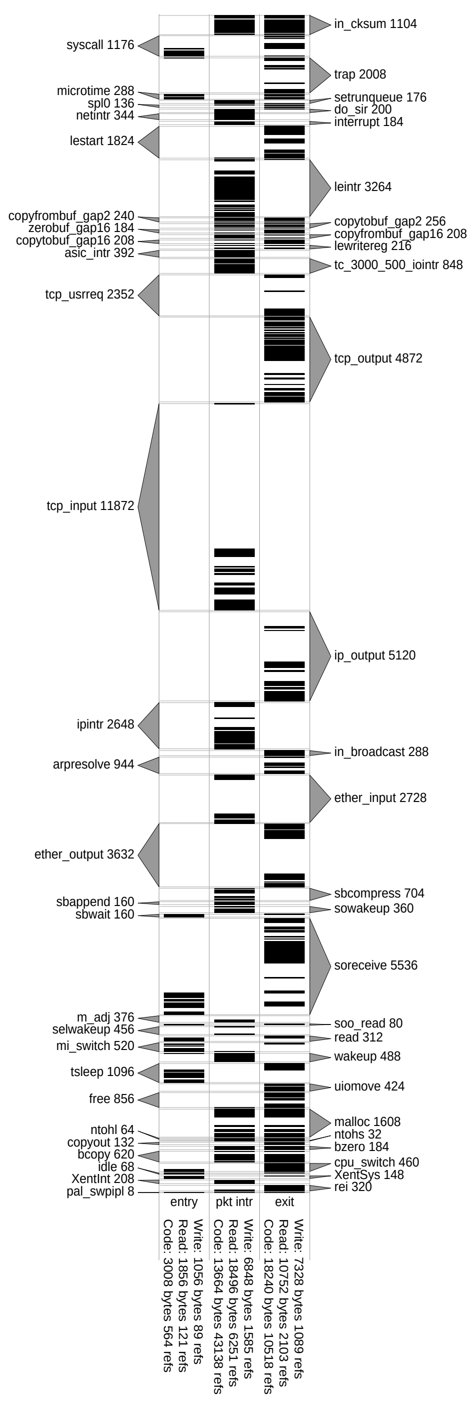
- **Lance Controller + Ethernet + TCP/IP + Sockets**
  - **NetBSD 1.0 (based on 4.4 Lite)**
  - **Digital Unix 3.0**
- **Machine: DEC 3000/400 (Alpha)**
  - **133 MHz, 8 KB primary cache**
- **Record trace of all memory references**
  - **NetBSD: simulate instructions**
  - **Digital Unix: instrument kernel using Atom**
  - **Misses PAL code on both systems**

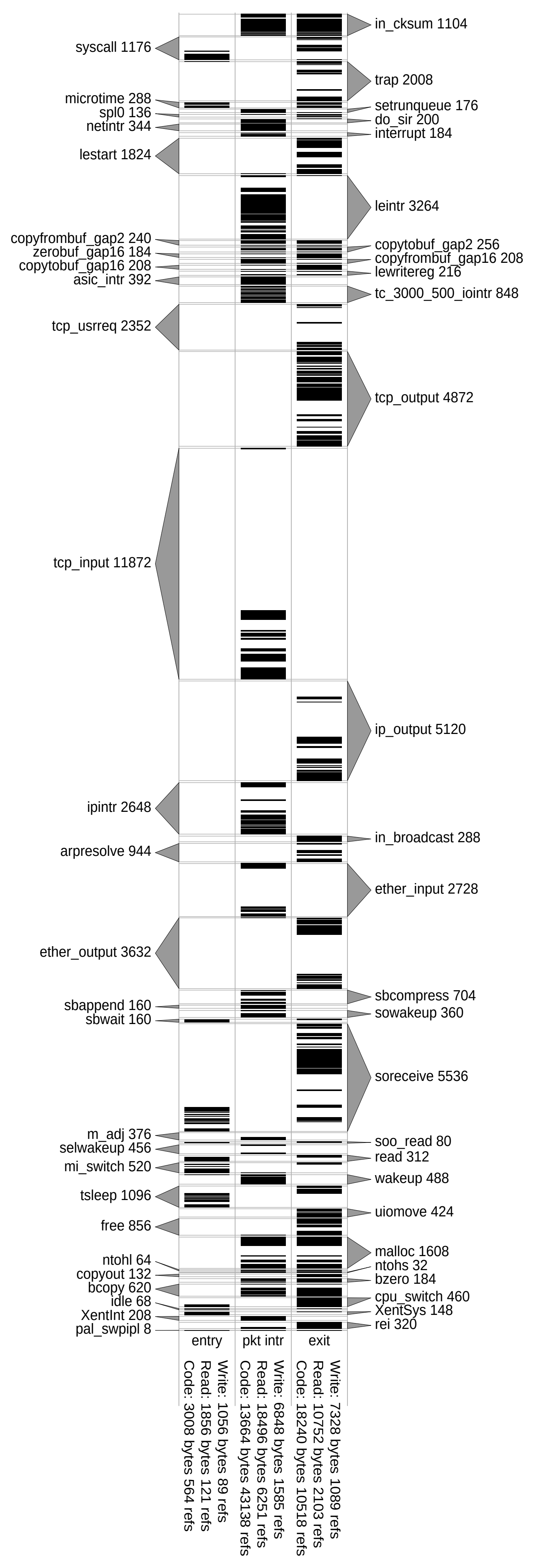
A Common Case of TCP/IP Receive

- Process makes read system call, goes to sleep
- Packet arrives on Ethernet
- Copy into main memory
- Ethernet, IP processing
- TCP processing:
  - find PCB (cached)
  - Verify checksum
  - Notice that it’s an in-sequence data packet
  - Update sequence numbers, timers, etc.
- Contents appended to socket buffer
- Process woken up
- Dequeue from socket buffer, copy into process memory
- Send TCP ACK
- IP, Ethernet processing for outbound ACK
- Copy onto Ethernet board, initiate transmit
- Return from system call

Subsystem		Working Set Size: Code	Read-only Data	Mutable Data
Protocol Layers	Device	4480	864	672
	Ethernet	2784	480	128
	IP	3168	448	160
	TCP	5536	544	448
	Socket low	608	32	160
	Socket high	1184	256	64
Overhead	Kernel entry/exit	2208	1280	640
	Process control	5472	544	736
Common	Buffer mgmt	1632	192	512
	Copy, checksum	3232	448	128
Total		**30592**	**5088**	**3648**

- **30 KB code doesn't fit in I-cache**





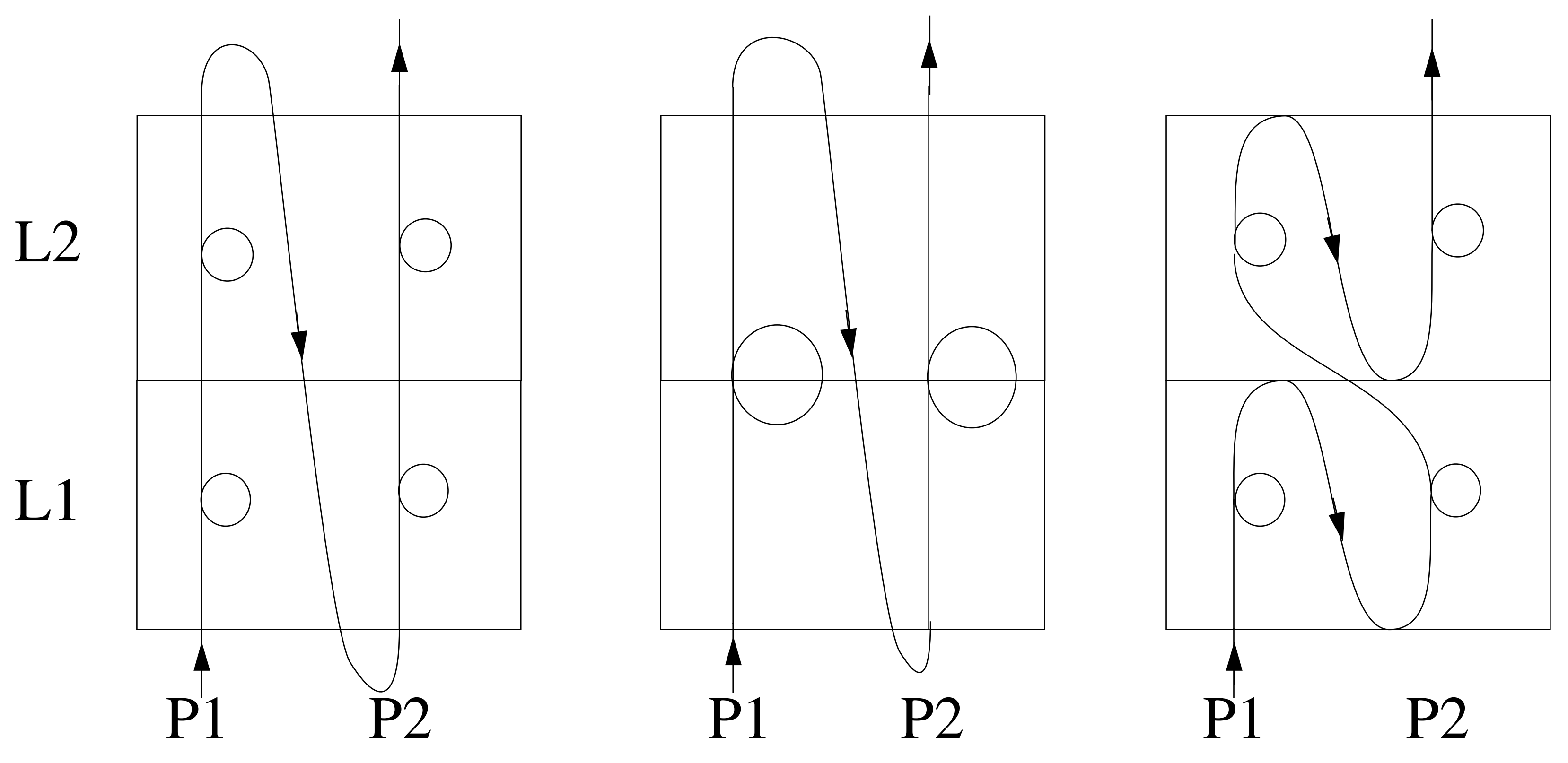
syscall 1176
microtime 288
spl0 136
netintr 344
lestart 1824
copyfrombuf_gap2 240
zerobuf_gap16 184
copytobuf_gap16 208
asic_intr 392
tcp_usrreq 2352
tcp_input 11872
ipintr 2648
arpresolve 944
ether_output 3632
sbappend 160
sbwait 160
m_adj 376
selwakeup 456
mi_switch 520
tsleep 1096
free 856
ntohl 64
copyout 132
bcopy 620
idle 68
XentInt 208
pal_swpipl 8
in_cksum 1104
trap 2008
setrunqueue 176
do_sir 200
interrupt 184
leintr 3264
copytobuf_gap2 256
copyfrombuf_gap16 208
lewritereg 216
tc_3000_500_iointr 848
tcp_output 4872
ip_output 5120
in_broadcast 288
ether_input 2728
sbcompress 704
sowakeup 360
soreceive 5536
soo_read 80
read 312
wakeup 488
uiomove 424
malloc 1608
ntohs 32
bzero 184
cpu_switch 460
XentSys 148
rei 320
entry
pkt intr
exit
Write: 1056 bytes 89 refs
Read: 1856 bytes 121 refs
Code: 3008 bytes 564 refs
Write: 6848 bytes 1585 refs
Read: 18496 bytes 6251 refs
Code: 13664 bytes 43138 refs
Write: 7328 bytes 1089 refs
Read: 10752 bytes 2103 refs
Code: 18240 bytes 10518 refs

syscall 1176
microtime 288
spl0 136
netintr 344
lestart 1824
copyfrombuf_gap2 240
zerobuf_gap16 184
copytobuf_gap16 208
asic_intr 392
tcp_usrreq 2352
tcp_input 11872
ipintr 2648
arpresolve 944
ether_output 3632
sbappend 160
sbwait 160
m_adj 376
selwakeup 456
mi_switch 520
tsleep 1096
free 856
ntohl 64
copyout 132
bcopy 620
idle 68
XentInt 208
pal_swpipl 8
in_cksum 1104
trap 2008
setrunqueue 176
do_sir 200
interrupt 184
leintr 3264
copytobuf_gap2 256
copyfrombuf_gap16 208
lewritereg 216
tc_3000_500_iointr 848
tcp_output 4872
ip_output 5120
in_broadcast 288
ether_input 2728
sbcompress 704
sowakeup 360
soreceive 5536
soo_read 80
read 312
wakeup 488
uiomove 424
malloc 1608
ntohs 32
bzero 184
cpu_switch 460
XentSys 148
rei 320
entry
pkt intr
exit
Write: 1056 bytes 89 refs
Read: 1856 bytes 121 refs
Code: 3008 bytes 564 refs
Write: 6848 bytes 1585 refs
Read: 18496 bytes 6251 refs
Code: 13664 bytes 43138 refs
Write: 7328 bytes 1089 refs
Read: 10752 bytes 2103 refs
Code: 18240 bytes 10518 refs

TCP/IP + Sockets is Code-Heavy

- **For each 552-byte packet received:**
  - **Move about 2.2 KB of packet contents**
  - **Touch 25-35KB of code or read-only data**
- **With 8 KB caches, 90% of off-CPU traffic isn't payload**

Layer Processing Loop Structure



Conventional **ILP** **Blocked**

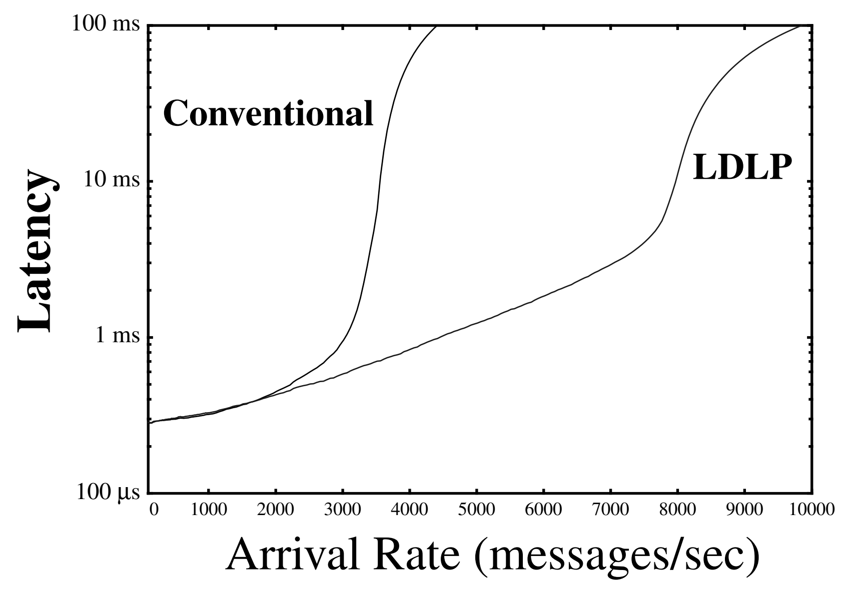
Implementing LDLP

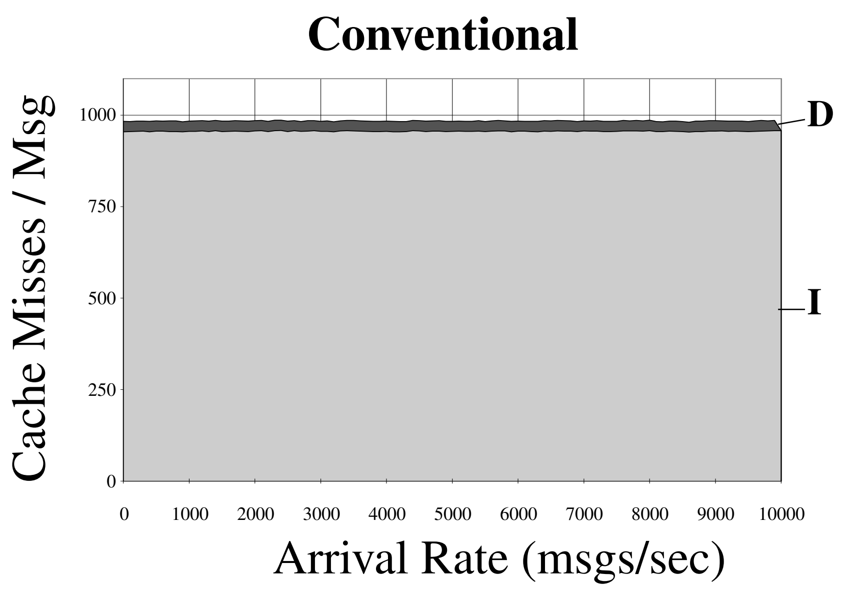
- Goal: Process packets in batches
- But don’t increase latency by waiting for complete batch
- Algorithm: On receive interrupt:
  - Disable further interrupts
  - Take all available packets off controller
  - Do layer 1 processing on all
  - Do layer 2 processing on all
  - ...
  - Reenable receive interrupt
- Number of packets in batch increases with load
- No major changes to layer code: just to interfaces

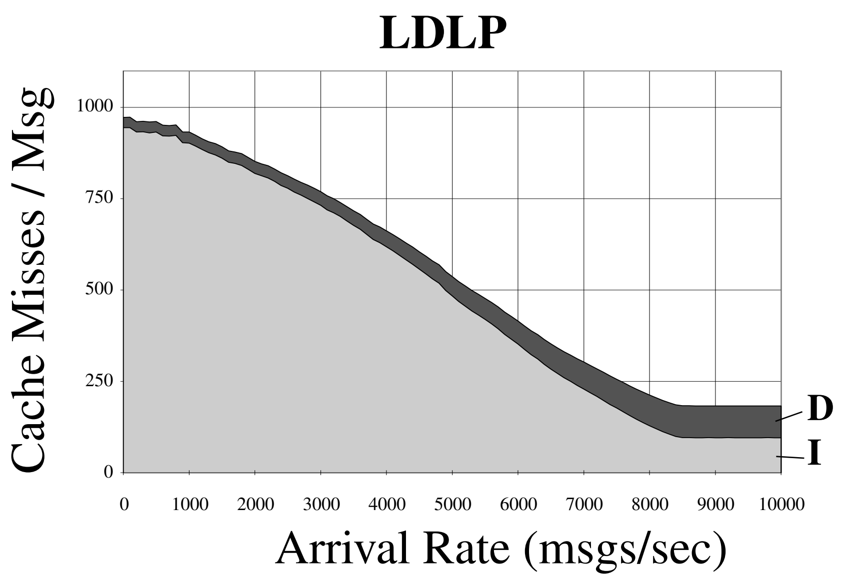
Synthetic Benchmark

- **5 simulated layers, each:**
  - **6 KB active code (1500 static instructions) (30 KB total)**
  - **256 bytes active static data (1280 bytes total)**
  - **1652 cycles of instruction processing (8260 total)**
  - **Touches each byte of packet**
- **Packets arriving:**
  - **552-byte packets (common TCP size)**
  - **Poisson arrival rates**
- **Simulate Timing**
  - **Simplified Alpha 3000/400 cache**
    - **32 byte cache lines**
    - **10 cycle miss penalty (assume hit in L2 cache)**

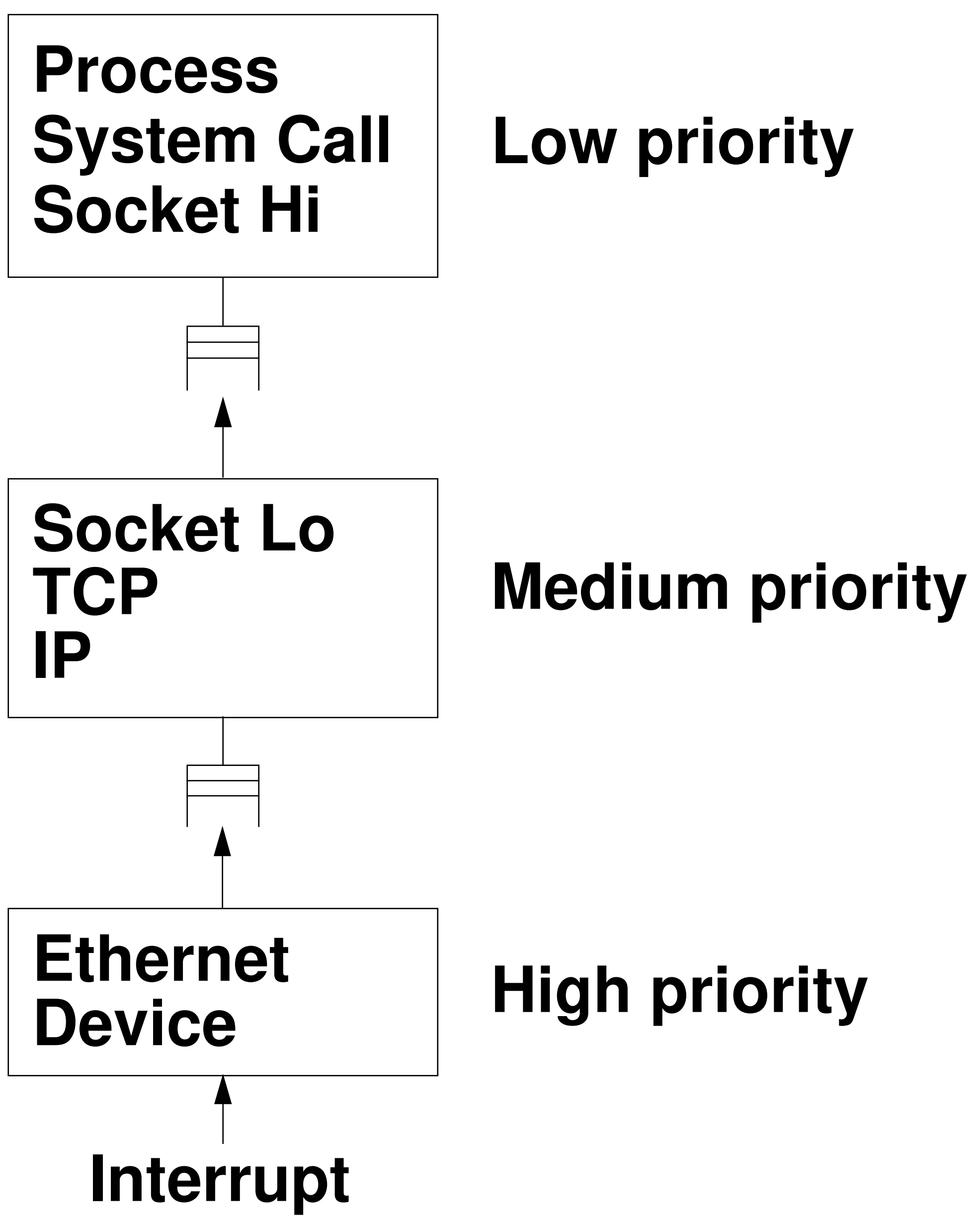
Simulation Results







Relation to BSD Network Implementation



Other Notes on Locality

- **CISC Beats RISC**

	Pentium vs Alpha Code Memory Footprint
TCP/IP Protocol Code	55% smaller
Data copy routines	86% smaller

- **Elaborate buffer management routines save data copying, but add code bloat**
- **Data loop optimizations (such as unrolling) should be compared with a cold instruction cache**

Conclusions

- **Many protocols spend more time fetching code from memory than fetching data**
  - **10:1 Code:Data ratio for receive-side TCP/IP over WAN**
  - **TCP/IP is code-heavy**
- **CISC beats RISC**
  - **Compact code means less time waiting for memory**
- **Locality can be improved by scheduling layer processing differently**
  - **Not nearly as invasive as ILP**
  - **Good time to also add overload control**

Summary: Techniques in Protocol Performance

Technique	Applicability: Data Transfer	Applicability: Control
Fast paths, optimizing common cases	X	X
Copy and Checksum loop implementation (CPU-specific)	X	
Avoiding payload copies	X	
Integrated layer processing	X	
Close coupling of layers	X	X
Optimizing PCB lookups (caching, hashing)	X	X
Overload management	x	X
Efficient decoding of complex message formats		X
Scheduling Layer Processing for Locality	x	X