

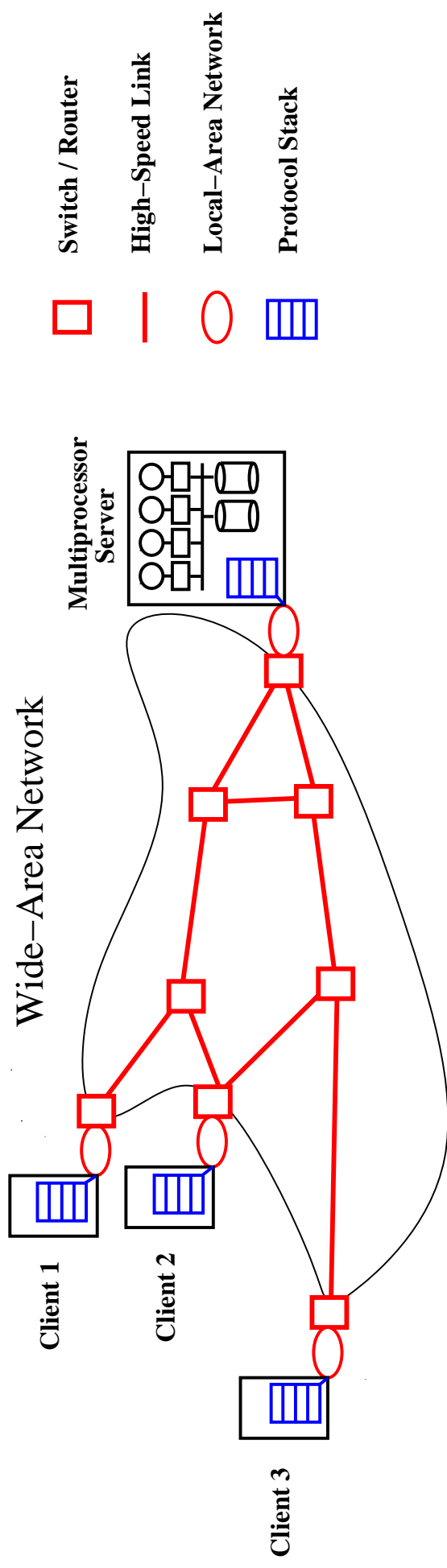
Cache Behavior of Network Protocols

Erich Nahum, David Yates,
Jim Kurose, and Don Towsley



Department of Computer Science
University of Massachusetts at Amherst

Motivation



Networked Information Servers

- World-Wide Web, Video, Image, File servers
- Large demands on servers

⇒ *Provide appropriate networking support for high-performance servers*

Network Protocol Stacks

HTTP	NFS
TCP	UDP
IP	
Network Interface	

Applications determine *functionality*

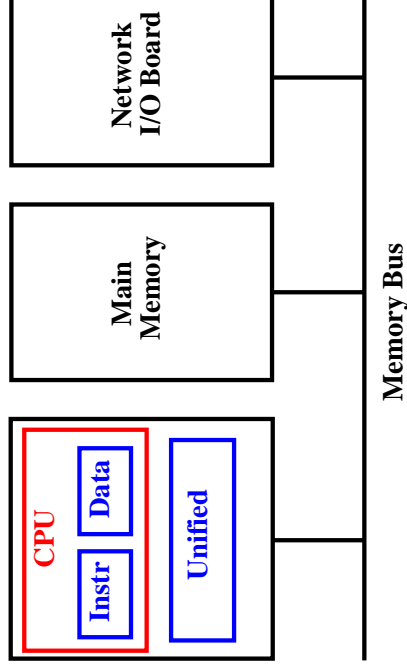
Functionality divided into conceptual *layers*

Layer functionality provided by a *protocol*

Protocols have different complexities:

- TCP: reliable, connection-oriented
- UDP: unreliable, connectionless

Impact of Memory on Protocol Performance



Increasing gap between CPU and memory speeds

- Example: 100 MHz (10 ns) SGI Challenge
- Cycle times for L1:2, L2:14, Mem:145

Studies on protocol memory performance

- Previous work focuses on reducing copies
- Our research: understanding cache behavior

Memory Reference Behavior: Research Issues

Characterizing Behavior of Network Protocols:

- How much time spent waiting for memory?
- What are miss rates?
- How do instruction refs compare to data?

Evaluate Sensitivity:

- Varying cache size
- Varying associativity
- Investigating future architectures

Example Instruction Cache Optimization:

- Using profile-guided code positioning

Memory Reference Behavior: Research Approach

Examine TCP/IP Network Protocol Stack

- Implemented in the x -kernel
- TCP code based on BSD Reno
- Runs in user space on SGI Machines
- Performance metric: *Latency* through stack

Construct Architectural Simulator

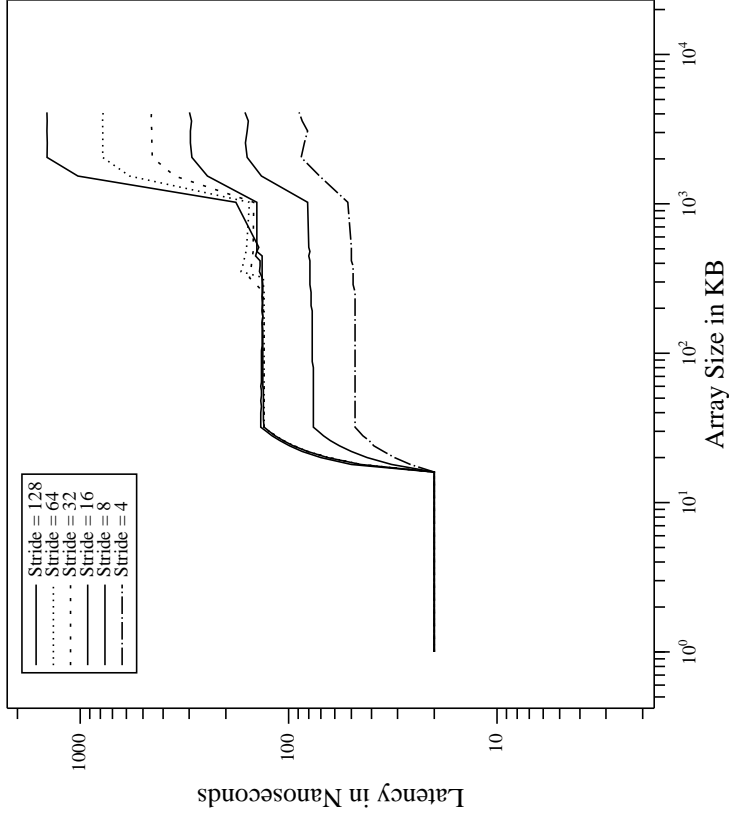
- Execution-driven simulation
- Use MINT (Veenstra 94) for trace generation
- Build trace consumer to model SGI
- *Validate* by comparing real times with simulated

⇒ *Run Protocols on Simulator and Evaluate*

Validation: Simulation vs. Reality

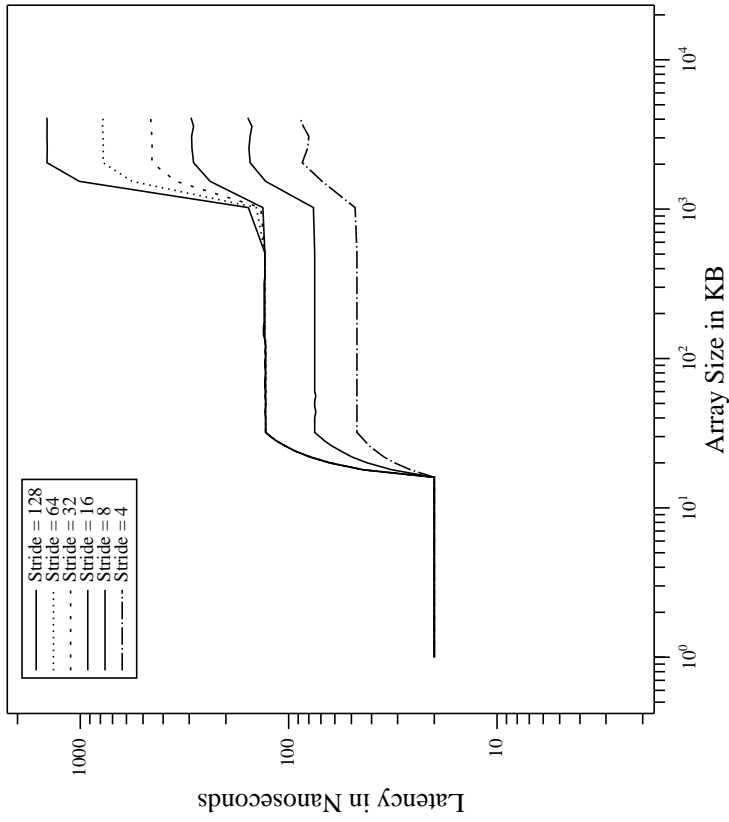
Actual

Predator (SGI IP19) READ times, 100MHz R4400 (10 ns clock),
16KB Split I/D 1st-level cache, 1MB 2nd-level unified
Direct-mapped, write-back



Simulated

Simulated Predator (SGI IP19) READ times, 100MHz R4400 (10 ns clock),
16KB Split I/D 1st-level cache, 1MB 2nd-level unified
Direct-mapped, write-back

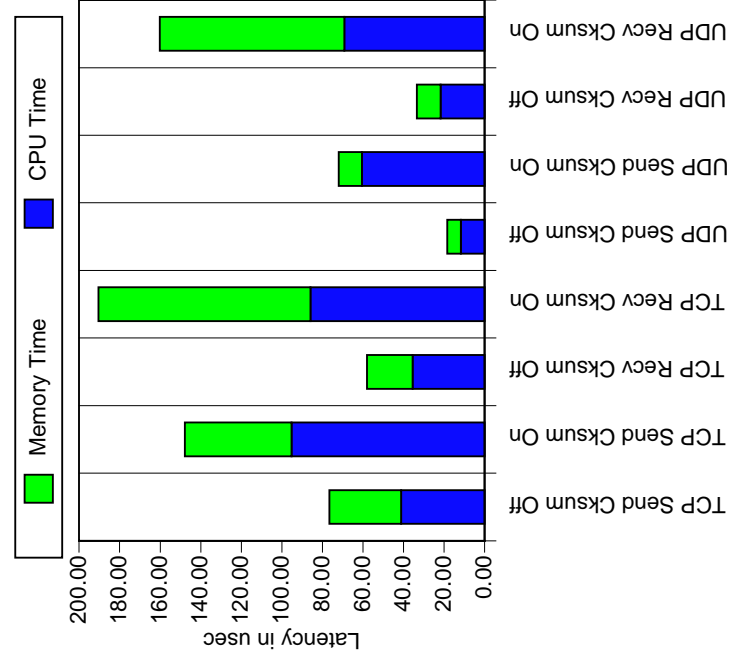


Memory Signature for an SGI R4400 Challenge

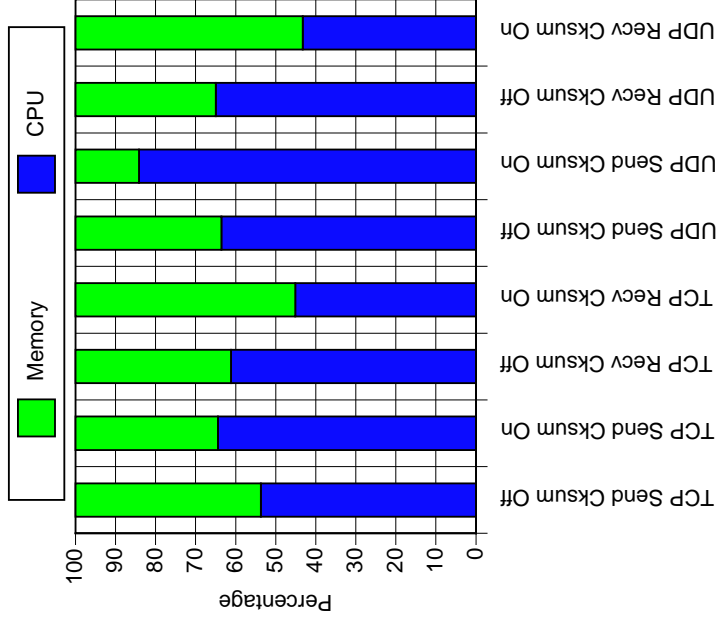
Validation: Simulated vs. Real Latencies

Protocol Benchmark	Simulated (μsec)	Real (μsec)	Error (%)
TCP Send, cksum off	76.63	78.58	-2.48
TCP Send, cksum on	147.84	146.66	0.81
UDP Send, cksum off	18.43	15.97	15.40
UDP Send, cksum on	71.99	70.30	2.41
TCP Recv, cksum off	58.06	62.65	-7.33
TCP Recv, cksum on	190.47	198.39	-3.99
UDP Recv, cksum off	33.80	32.84	2.95
UDP Recv, cksum on	161.78	158.84	1.85
Average Error			4.65

Q: Time Spent Waiting for Memory?



Latencies



Percentage

- Memory time varies from 15 to 58 %

Instructions vs. Data

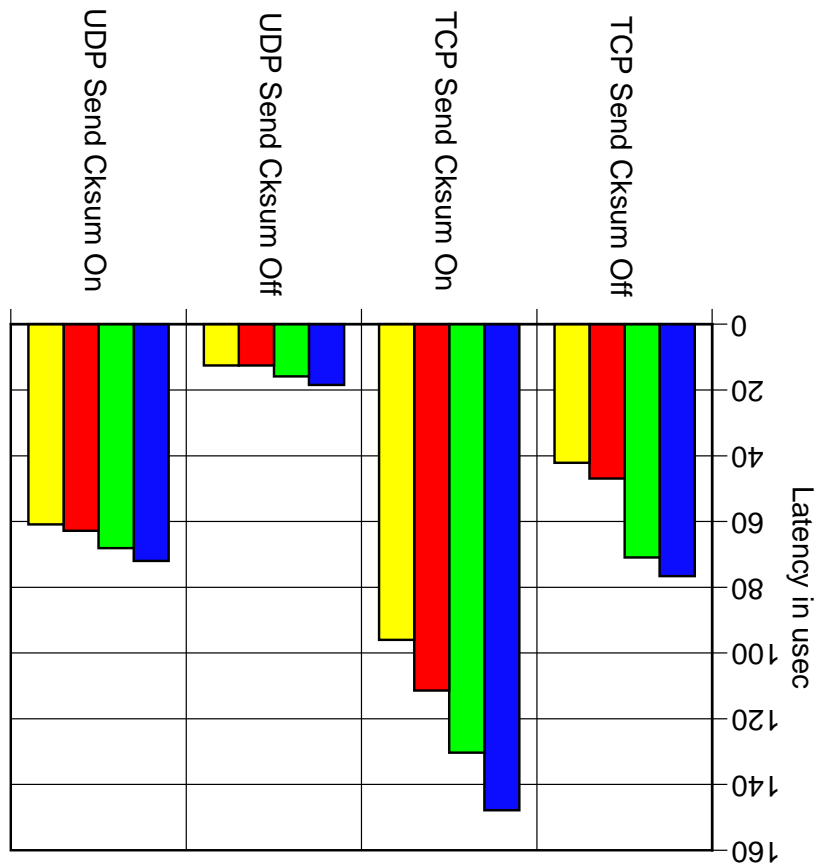
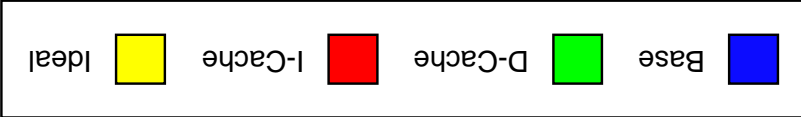
Memory References to Both Instructions and Data

- Machines have separate on-chip I + D caches

Which References have Larger Performance Impact?

- Baseline machine (same simulator)
- Perfect D-Cache (all *data* refs hit)
- Perfect I-Cache (all *instr* refs hit)
- Idealized Cache (*all* refs hit)

Instructions vs. Data



Latency vs. Cache Size

- I-Cache has bigger impact

Impact of Cache Size

Our SGI Machine uses MIPS R4400's

- R4400 has 16 KB on-chip L1 caches

Chip Densities are Increasing

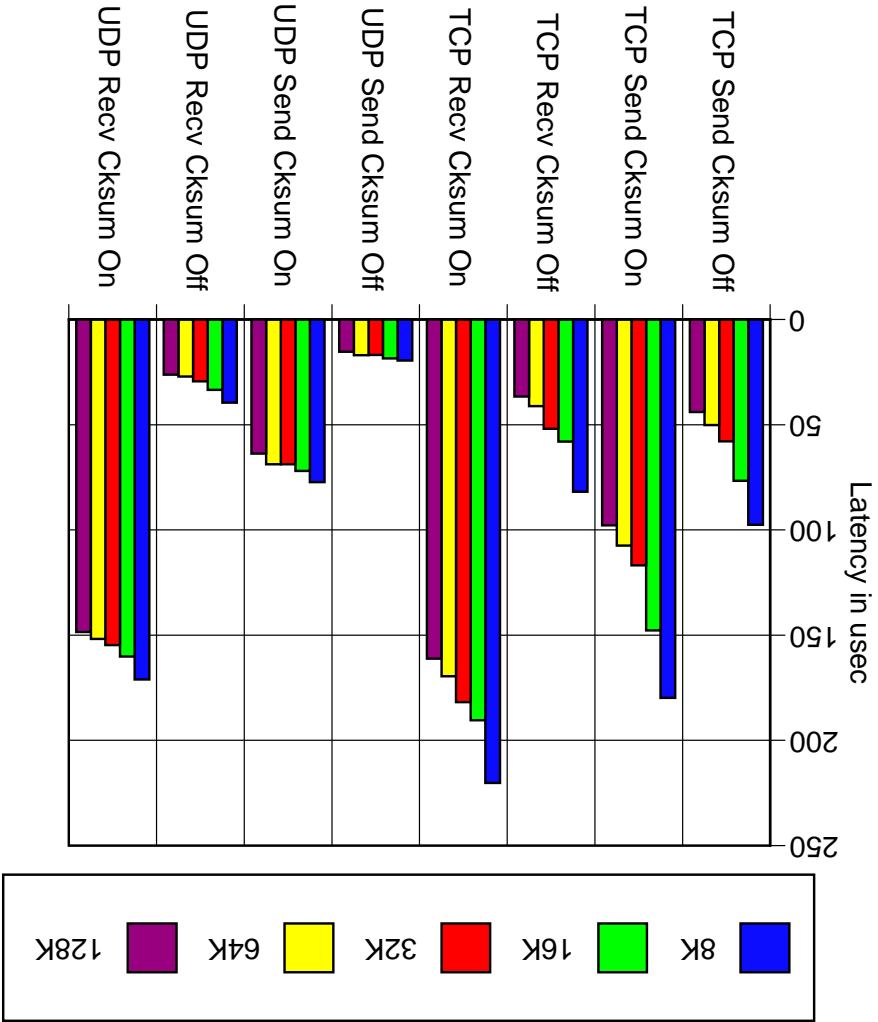
- On-chip cache sizes are growing

How Sensitive is Protocol Performance to Cache Size?

- Vary L1 size from 8 KB to 128 KB
- Leave L2 at 1 MB

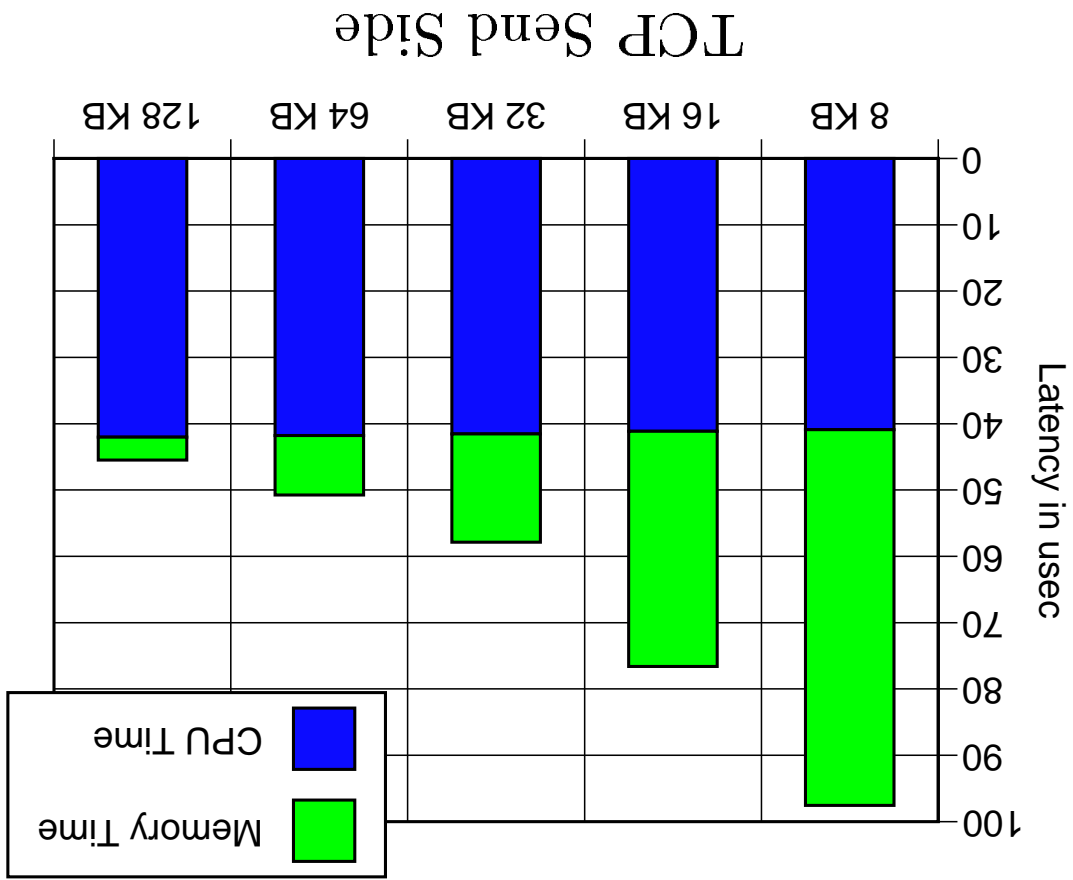
- TCP more sensitive to cache size

Latency vs. Cache Size



Varying Cache Size

- Less time spent waiting for memory



Varying Cache Size

Impact of Cache Associativity

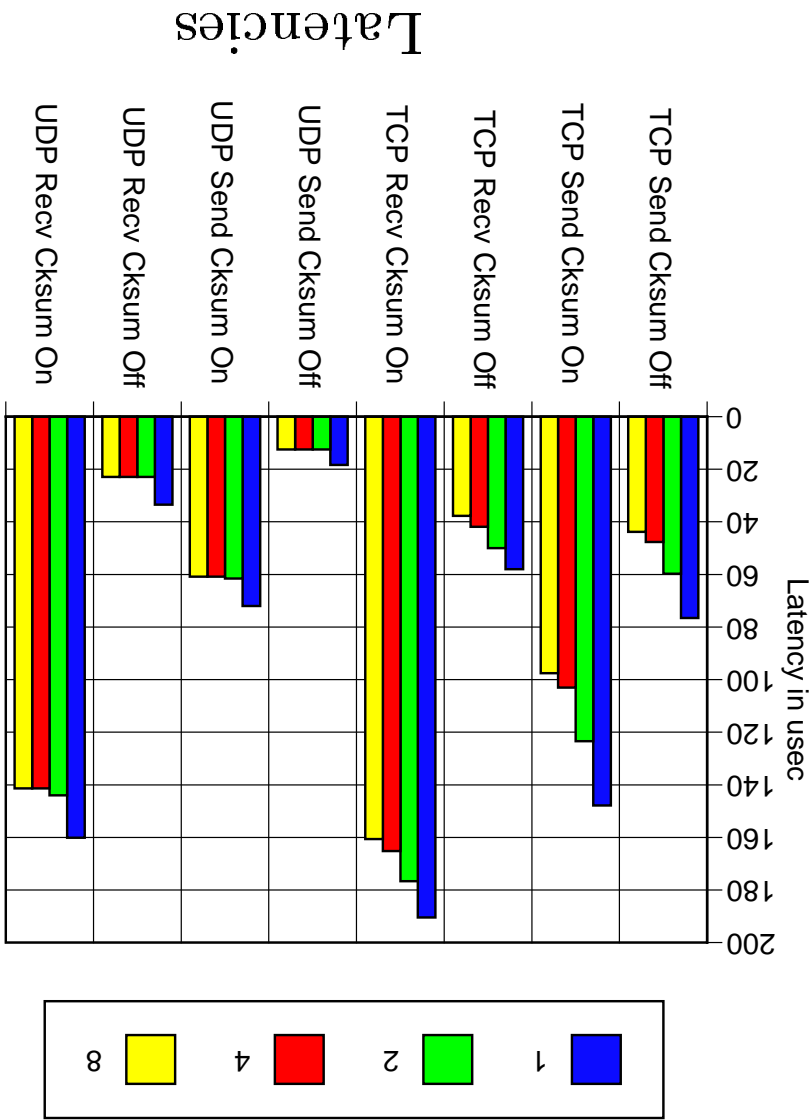
Associativity Varies by Chip

- Direct mapped \rightarrow data maps into 1 possible slot
(MIPS R4400, DEC Alpha)
- 2-way set assoc \rightarrow 2 possible slots
(Intel Pentium, MIPS R10000, UltraSparc)
- 4-way set assoc \rightarrow 4 possible slots
(Intel 486, Motorola 68040)
- 8-way set assoc \rightarrow 8 possible slots
(PowerPC 601, 620)

How Sensitive are Protocols to Associativity?

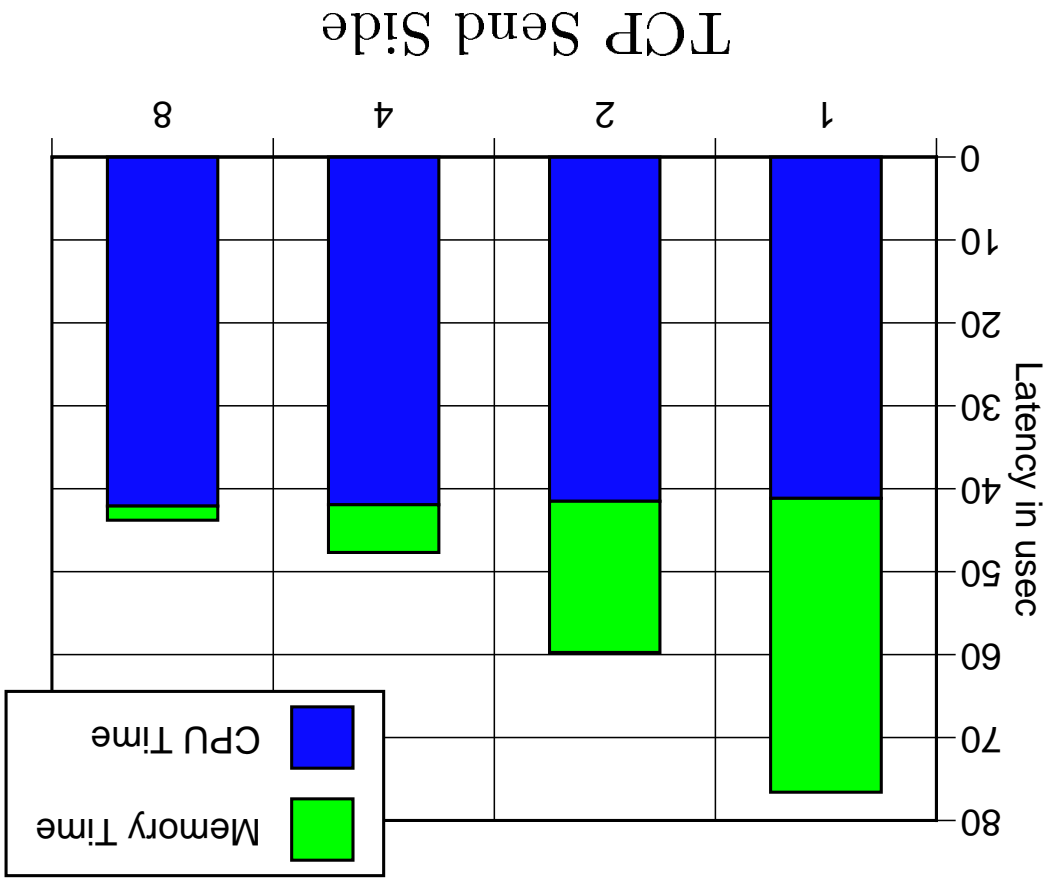
- Vary associativity from 1 to 8

- TCP more sensitive to associativity



Varying Associativity

- Less time spent waiting for memory



Varying Associativity

Impact of Future Architectures

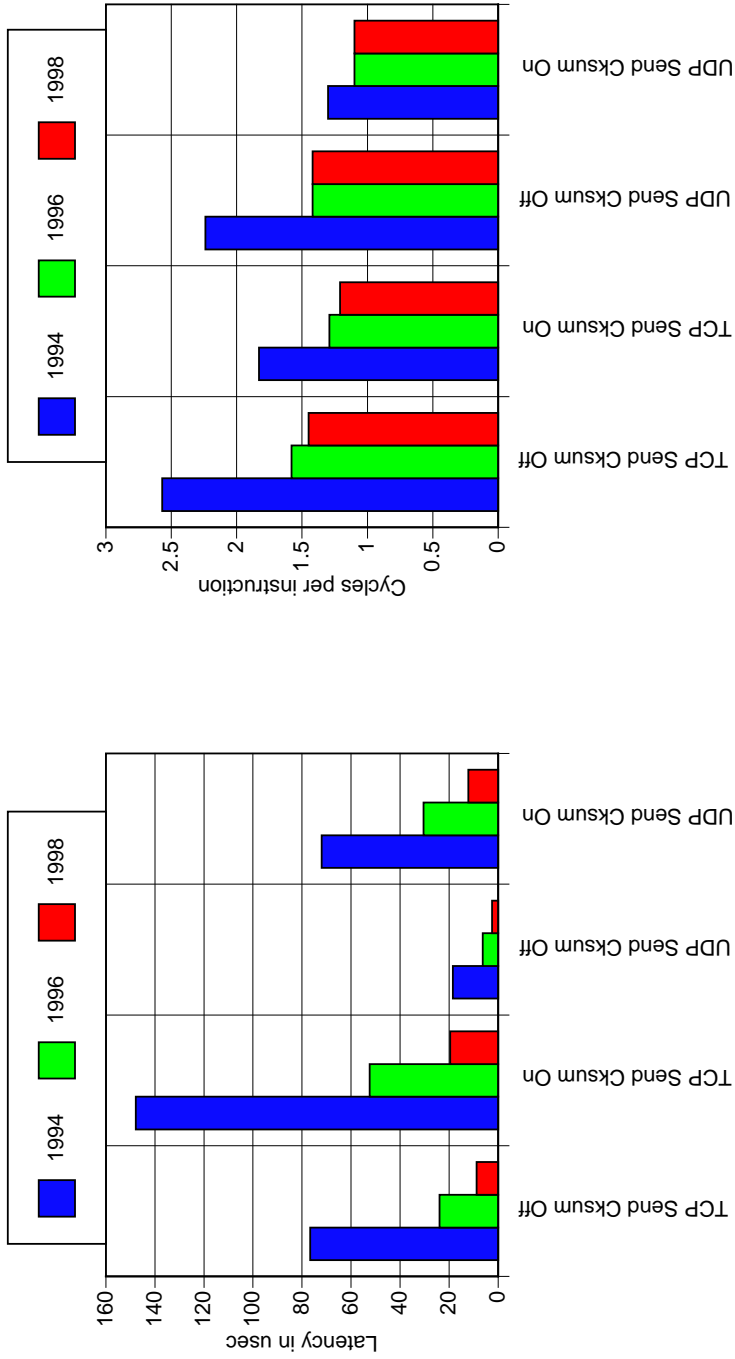
Architectures Change over Time

- Clock speeds increasing exponentially
- On-chip cache sizes are increasing
- Main memory speeds falling further behind
- Cache miss penalties getting worse

How will Protocols run on Future Machines?

- 1994: 100 MHz, 16 KB, direct-mapped
- 1996: 200 MHz, 32 KB, 2-way set assoc
- 1998: 400 MHz, 64 KB, 2-way set assoc

Future Architectures



Latencies

Machine CPI's

- Latencies and CPIs improve over time

Improving Performance Using Code Layout

Restructuring Executables to Improve Performance:

- Profile-guided code positioning (Pettis 90)
- Generate a *profile* from a previous run
- Determine frequently used procedures
- Re-order instructions to avoid cache conflicts

How effective is Code Layout for Network Protocols?

- Use Pixie to produce profiles of protocol code
- Use CORD to generate new binaries
- Evaluate CORDED binaries on simulator

Impact of Using Code Layout

Protocol Configuration	Original Latency	CORD Latency	Diff (%)
TCP Send Cksum Off	76.61	72.61	5
TCP Send Cksum On	147.80	148.38	0
TCP Recv Cksum Off	58.04	54.33	6
TCP Recv Cksum On	190.42	186.61	2
UDP Send Cksum Off	20.92	12.53	40
UDP Send Cksum On	77.45	65.59	15
UDP Recv Cksum Off	33.46	27.03	19
UDP Recv Cksum On	160.15	148.76	7

- CORD improves performance up to 40 %

Research Summary: Protocol Cache Behavior

Protocols spend 16-75 % waiting for memory

Miss rates vary from 0 - 28 %

Instruction cache is most significant

Protocols use load, store, add, branch instr.

TCP very sensitive to cache behavior

TCP has a lot of conflicts

Protocols will scale with CPU speed

Code layout improves protocol performance
(CORD)

Future work

Modern Architectural Features

- Superscalars
- Non-blocking caches
- Speculative execution

CISC CPU's (Intel x86)

Multiprocessors

More Information

Available at <http://www.cs.umass.edu/~nahum>

- Full paper with results
- Tech report on simulator design and validation
- Ph.D. thesis

Mosberger *et. al.* 96:

Also focuses on I-Cache

Uses Scout on DEC Alpha

Focuses on improving protocol latency

Compiler-related approaches:

- Outlining
- Cloning
- Path-inlining

Shows up to 40 % improvement

Does not examine cache sensitivity

Does not distinguish instr vs. data

Does not predict future performance

Blackwell 96:

Also focuses on I-Cache
Uses traces of NetBSD on DEC Alpha
Proposes locality-driven layer processing
Process all packets at layer for I-Cache behavior
Evaluates using a simulation

Network Protocol Instruction Costs

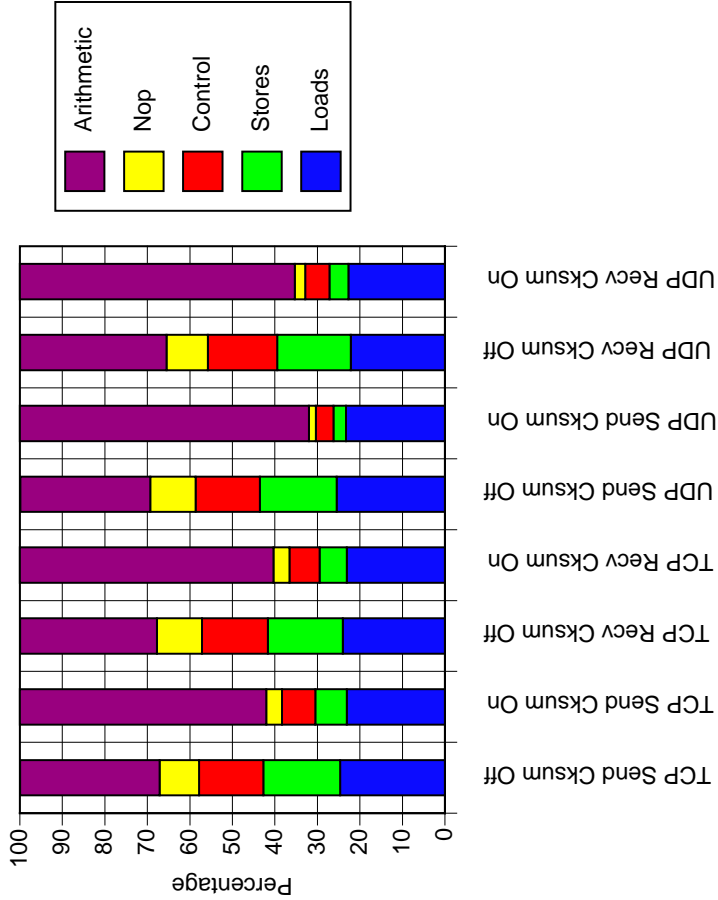
Instructions fall into roughly 4 categories:

- Load/Store: moving data to/from memory
- Arithmetic: add, sub, mult, div
- Control: conditional branch, jump
- Nop: do nothing

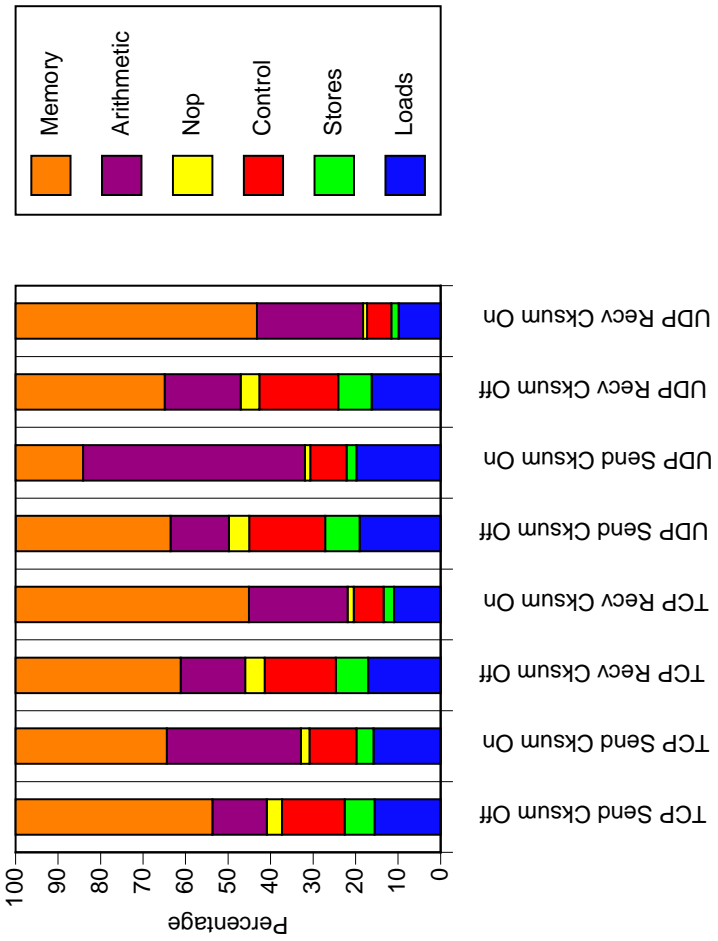
What Instruction Costs do Protocols Incur?

- Evaluate instruction *usage*
- Determine instruction *costs*

Instruction Costs



% of Instructions

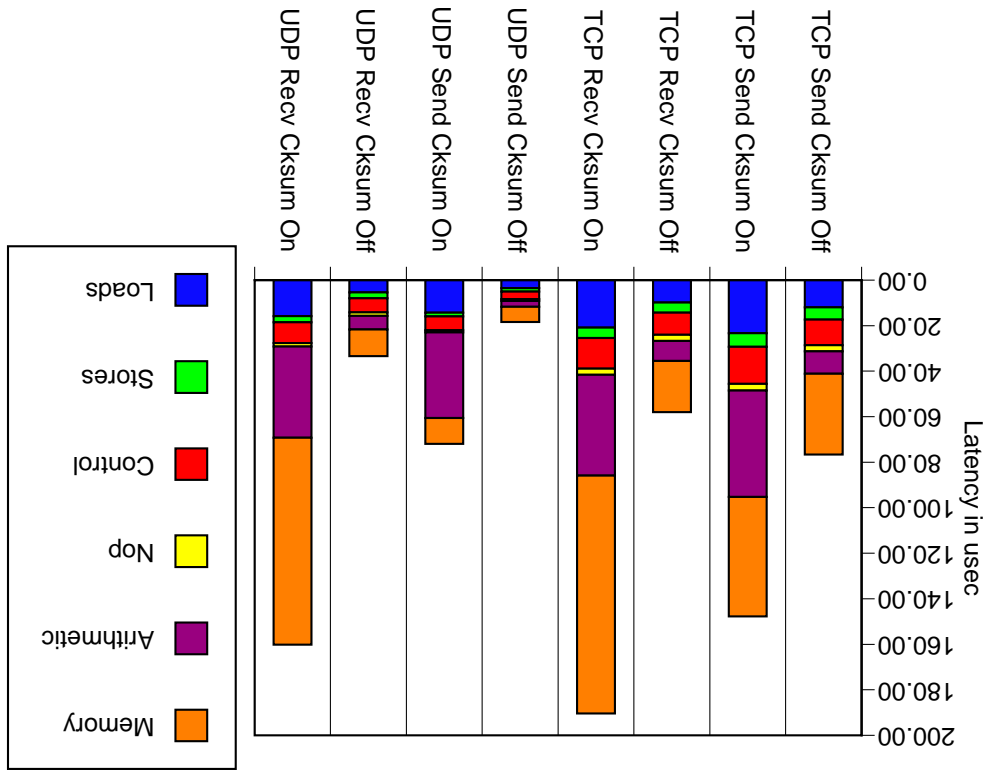


% of Cycles

- Instruction costs vary from 42 to 85 %

- Branch delay not a significant factor

Latency



Contribution to Latency

MINT Simulator Issues

Features:

- Supports multiple cache levels, sizes
- Supports inclusion, invalidation, associativity
- Some pipeline interlocks:
 - * Load delay
 - * Branch delay

Assumptions:

- Dedicated machine
- Most instructions take a single cycle
- Virtual addr = physical addr
- Heap allocator same in MINT vs. IRIX

Validation Lessons

Start Small, Work Your Way Up:

- Use microbenchmarks to capture costs
- Run microbenchmarks on simulator to validate

Event Frequency is Key:

- Capture cost of *frequent* events
- Rare/nonexistent events can be ignored
- Exception: *expensive* events

Diminishing Returns:

- Capturing cost of an event slows the simulator
- Accuracy gets harder to improve over time

Iterate until Satisfied:

- Went through about 15 iterations