Analysis of Techniques to Improve Protocol Processing Latency

David Mosberger, Patrick Bridges,

Larry L. Peterson, and Sean O'Malley

The University of Arizona

e-mail: {davidm,bridges,llp}@cs.arizona.edu

sean@netapp.com

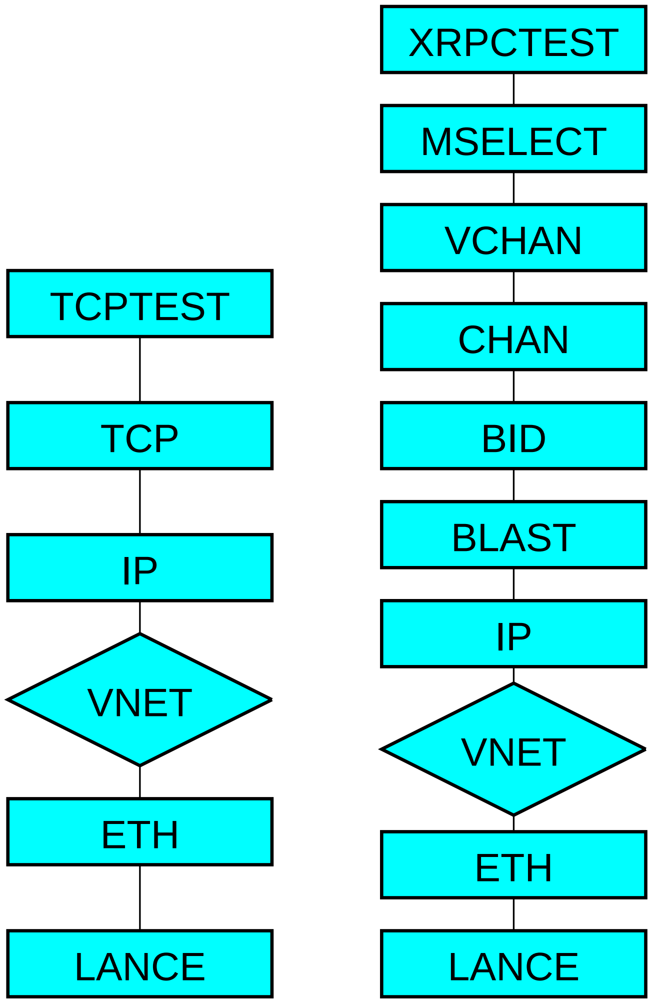
www: http://www.cs.arizona.edu/scout

Latency: Where does it come from?

- Speed of light
- Data touching overheads?
  - No: messages (data) are small.
- Execution overheads?
  - Too much code.
  - Badly structured code.

Test Environment

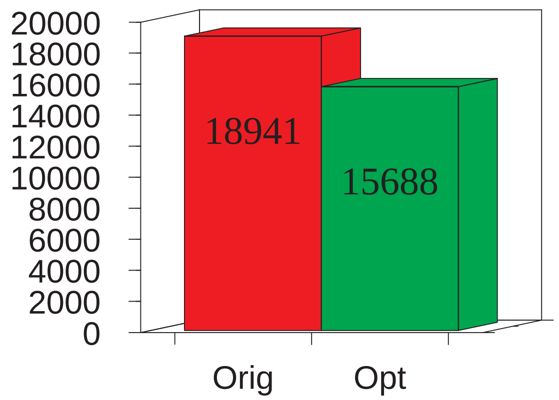
- Protocol stacks
  - TCP/IP
  - RPC
- Hardware platform
  - 175MHz Alpha
  - 100MB/s memory
  - TURBOchannel bus
  - 10Mbps Ethernet



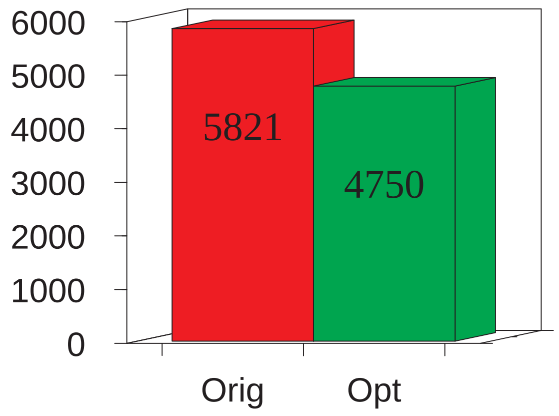
Starting Point

- Data cache footprint
  - padding
  - stack switching
  - info duplication
- Tiny functions
- Machine idiosyncracies
  - byte load/store
  - integer division

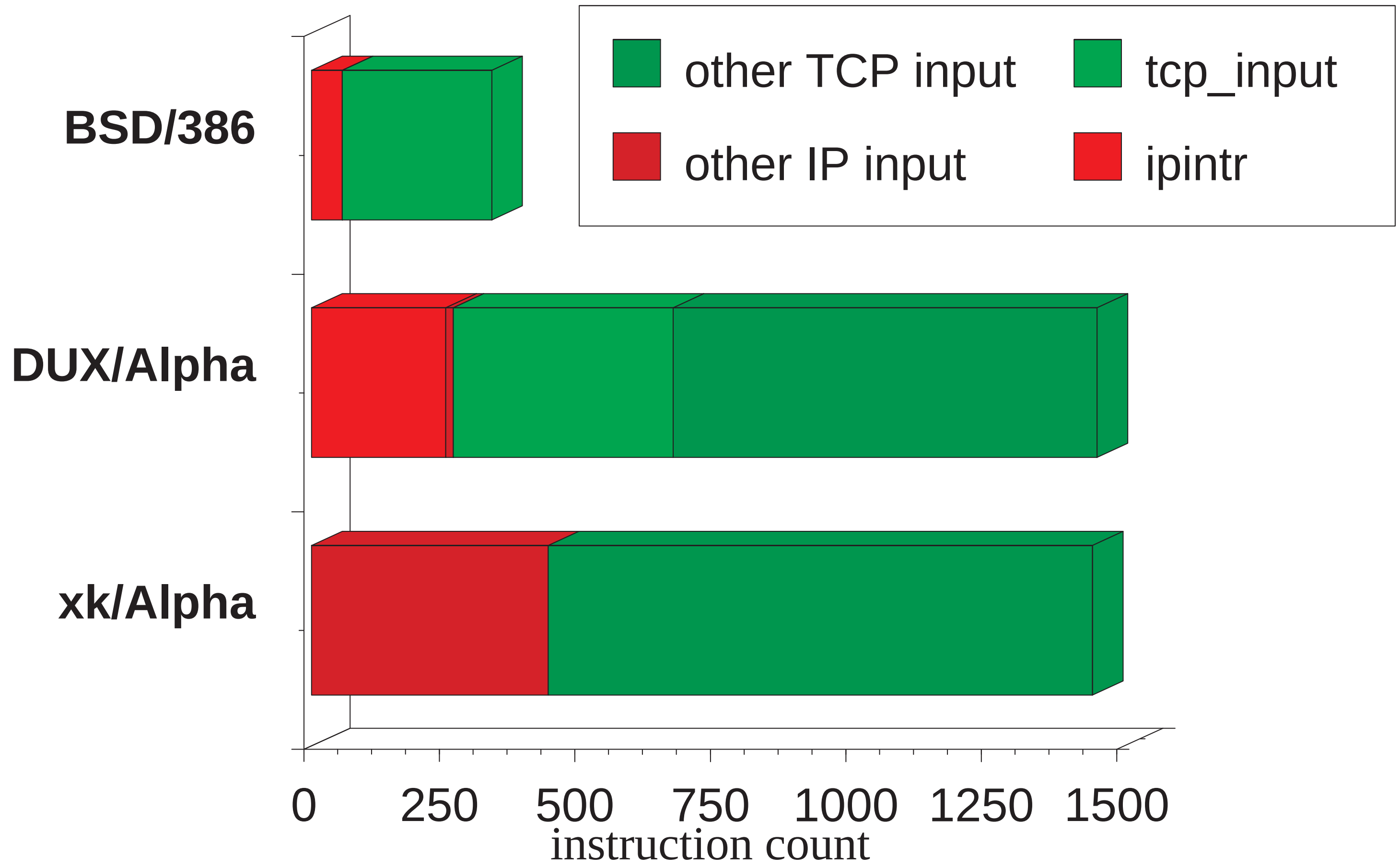
cycle count



instruction count



How fast is TCP/IP?



Latency Bottlenecks

Suspects

- Frequent branching
- Instruction-cache gaps
- Cache collisions
- Layering overheads

Not instruction/data translation buffer.

Techniques

- Outlining attacks:
  - frequent branching
  - i-cache gaps
- Cloning attacks:
  - cache collisions
- Path-inlining attacks:
  - layering overheads

Outlining

- Exception-handling code
  - lots of it (up to 50%)
  - dilutes instruction-cache
  - causes taken branches
- Remove from fast path
  - annotate if-statements with branch probability
  - move unlikely code to end of function

Outlining Example

```
            :
if (bad_case @ 0) {
    panic("bad day");
}
printf("good day");
            :
```

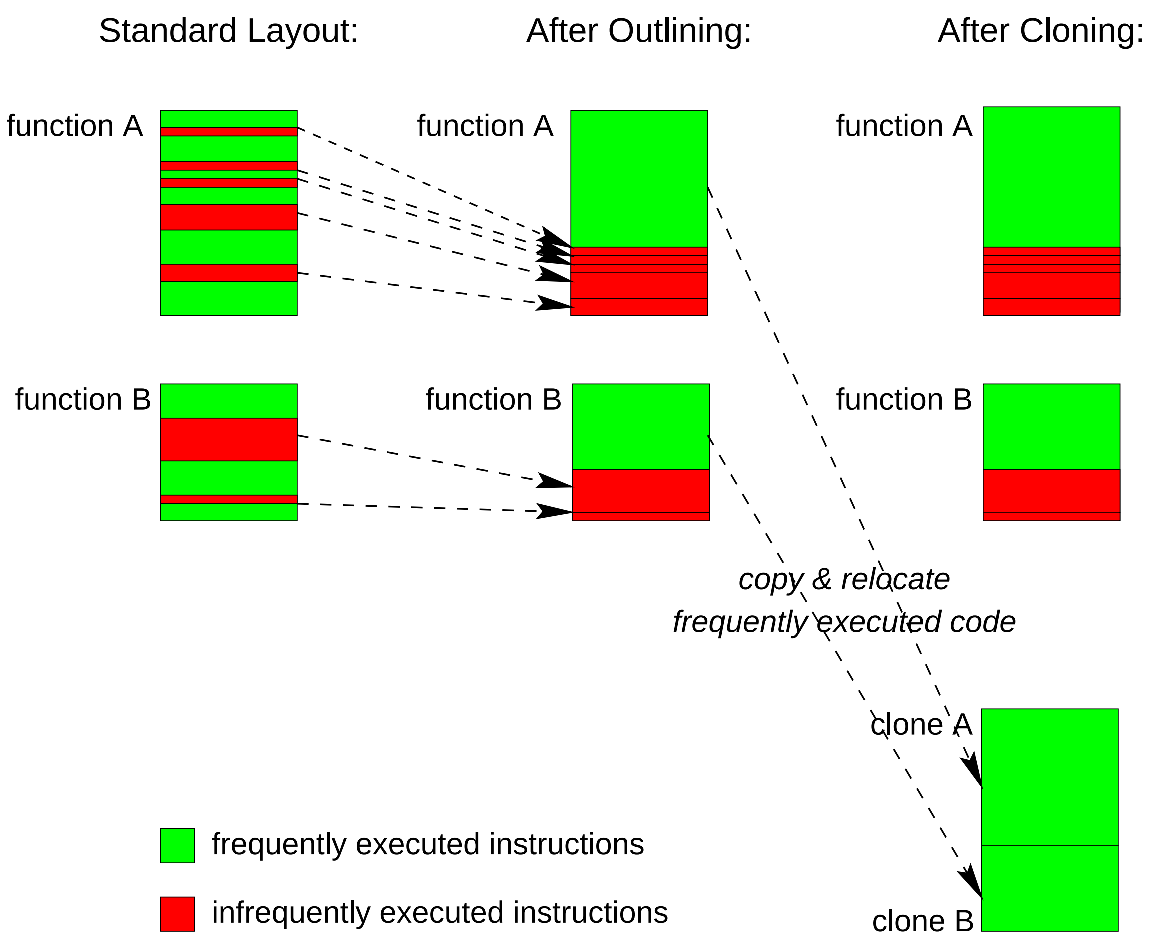
```
    :
  load          r0, (bad_case)
  jump_if_0     r0, good_day
  load_addr     a0, "bad day"
  call          panic
good_day:
  load_addr     a0, "good day"
  call          printf
    :
```

```
    :
  load          r0, (bad_case)
  jump_if_not_0 r0, bad_day
  load_addr     a0, "good day"
  call          printf
continue:
    :
  return
bad_day:
  load_addr     a0, "bad day"
  call          panic
  jump          continue
```

Cloning

- Make copy of functions on fast path
  - relocate to avoid conflict misses
  - specialize for a particular use (partial evaluation)
- Alternative layout algorithms
  - micro-positioning
  - bipartite layout

Outlining & Cloning Summary



Path-Inlining

Collapse deeply-nested functions

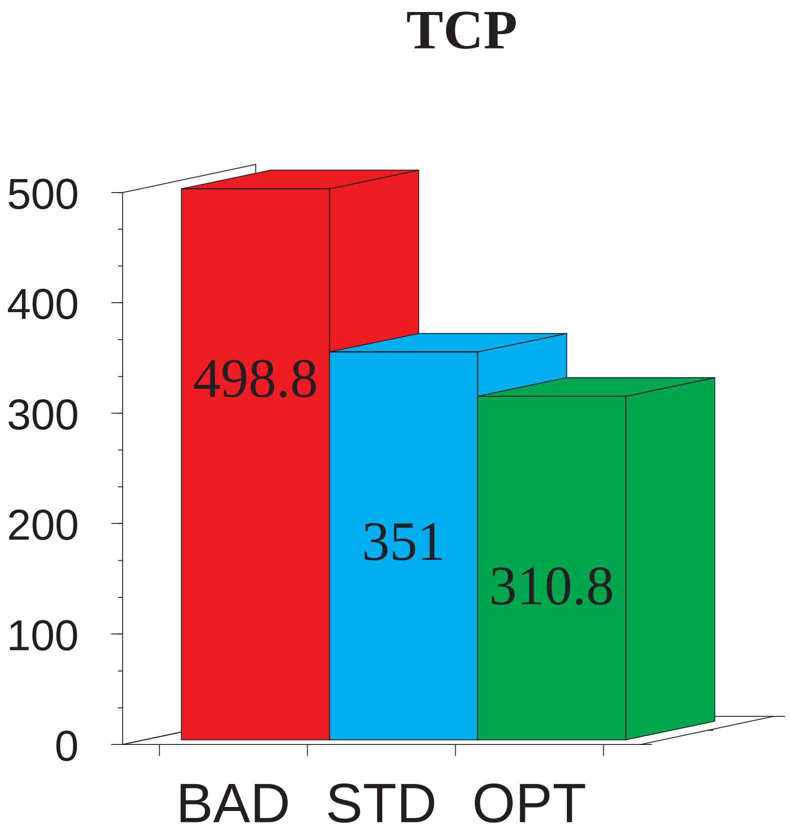
- Assume fast path is known
- Compile entire function as single unit

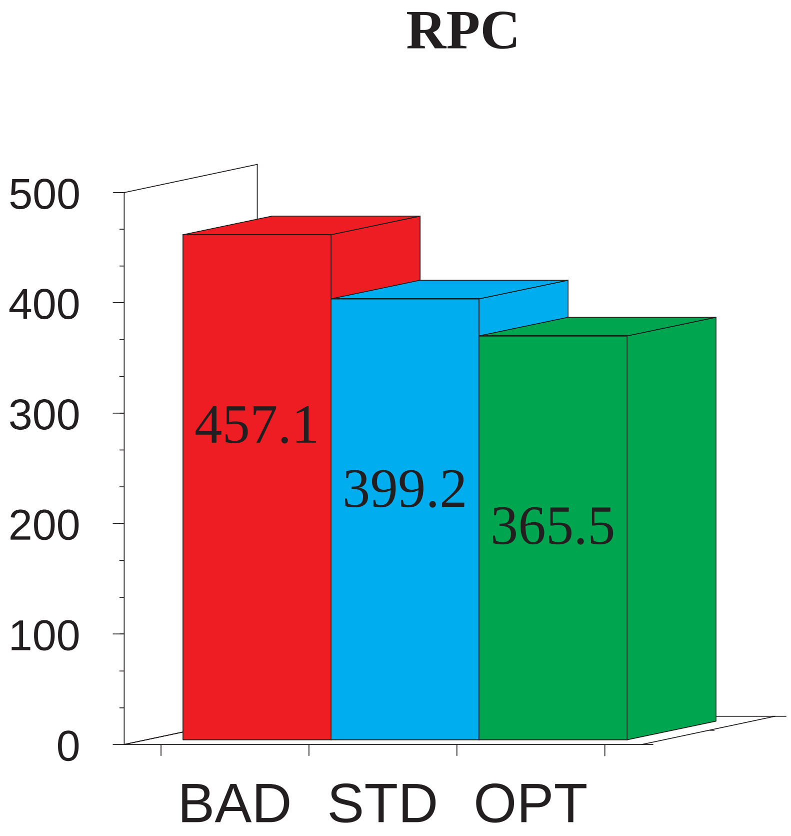
Advantages

- Removes call-overheads
- Increases context for optimizer

End-to-End Latency

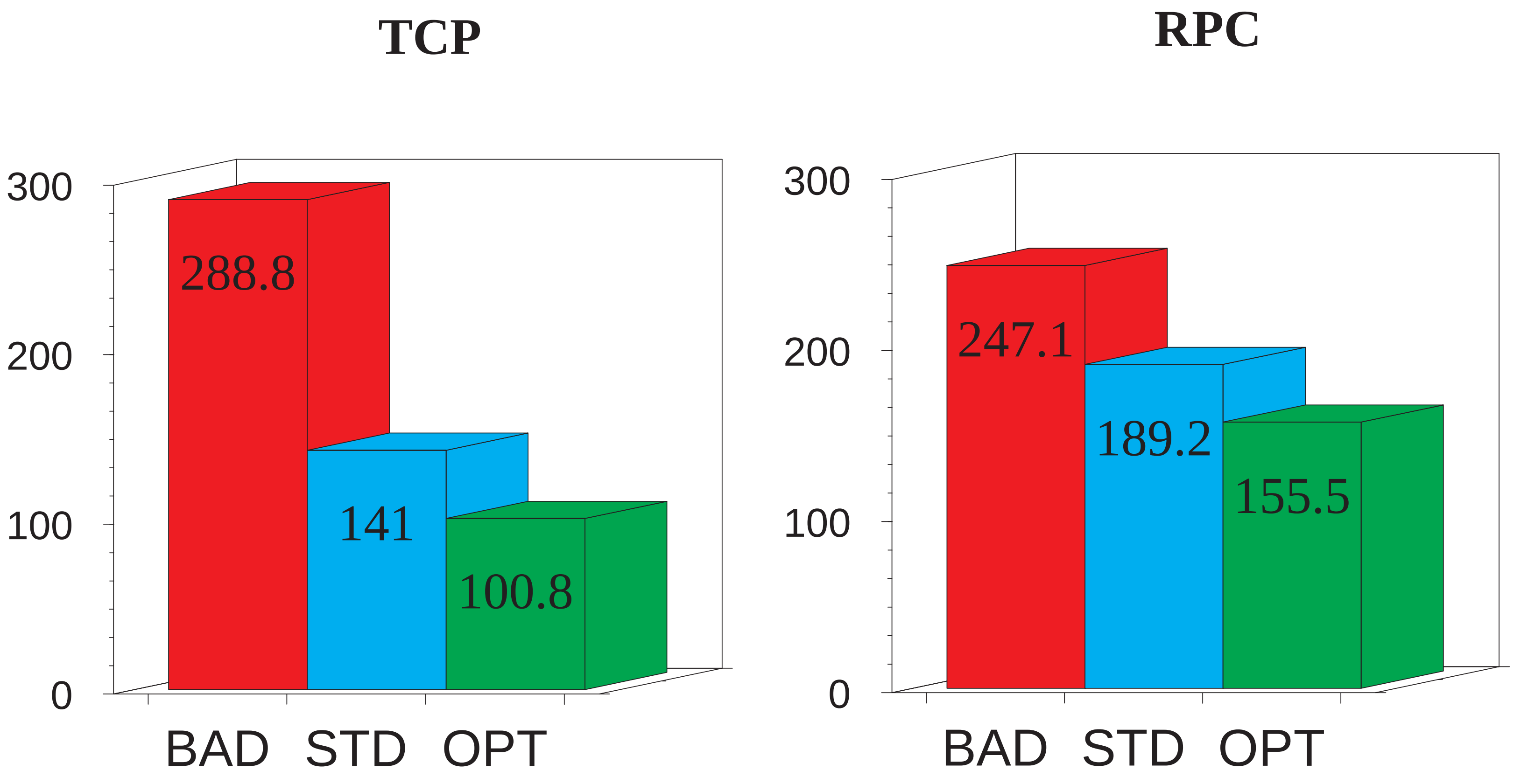
Roundtrip time in μs:



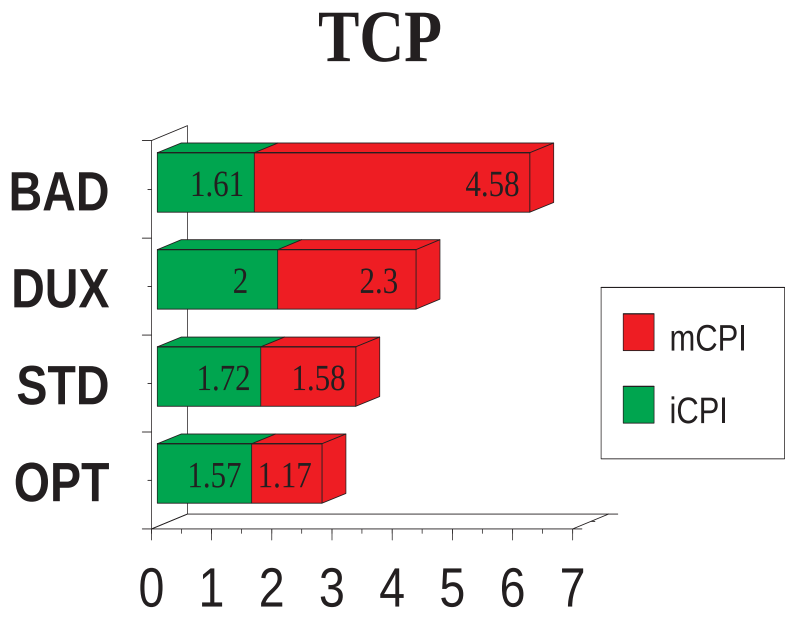


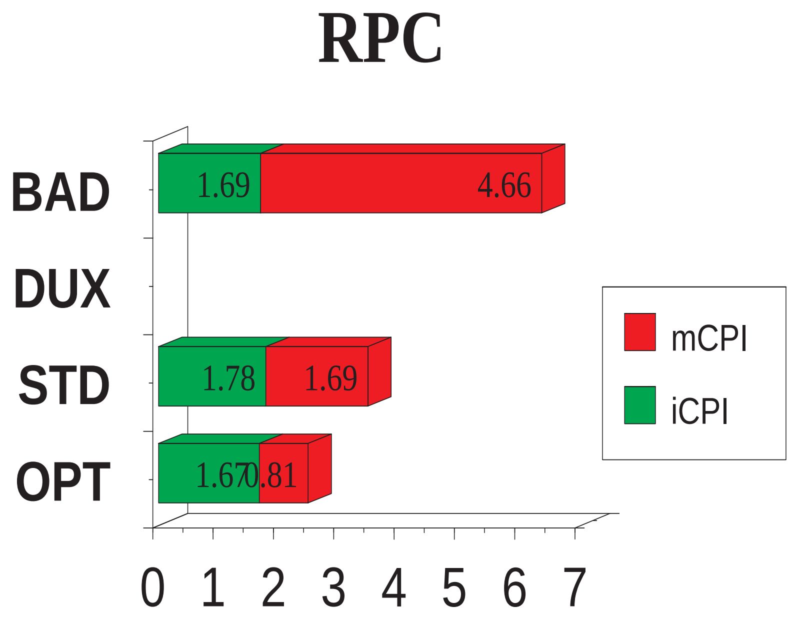
Processing Latency

Processing-time per roundtrip in μs:



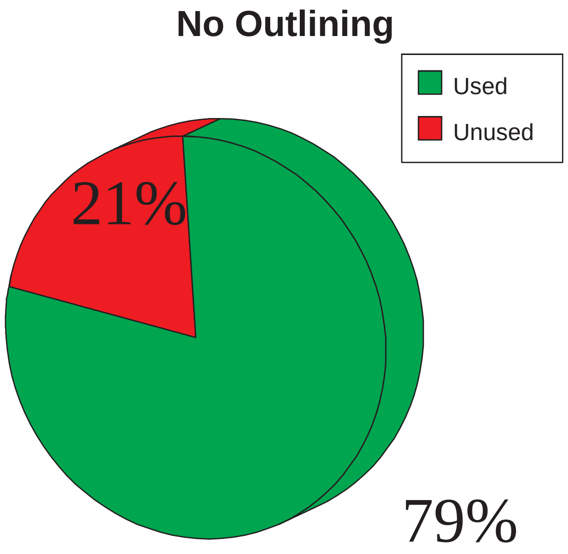
Memory System Performance

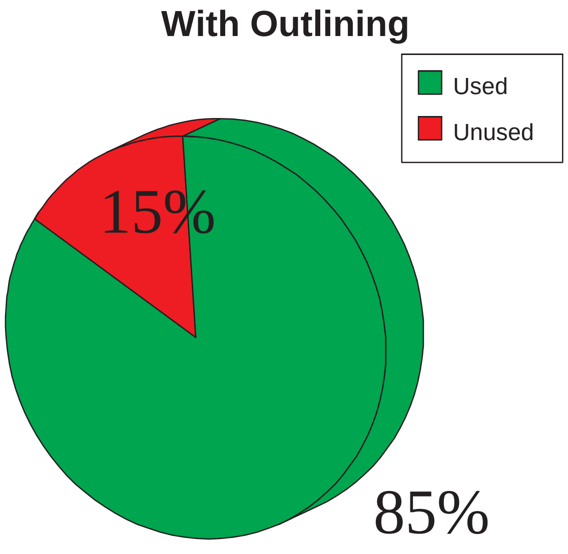




Outlining Effectiveness

- TCP





- RPC
  - Essentially identical performance.

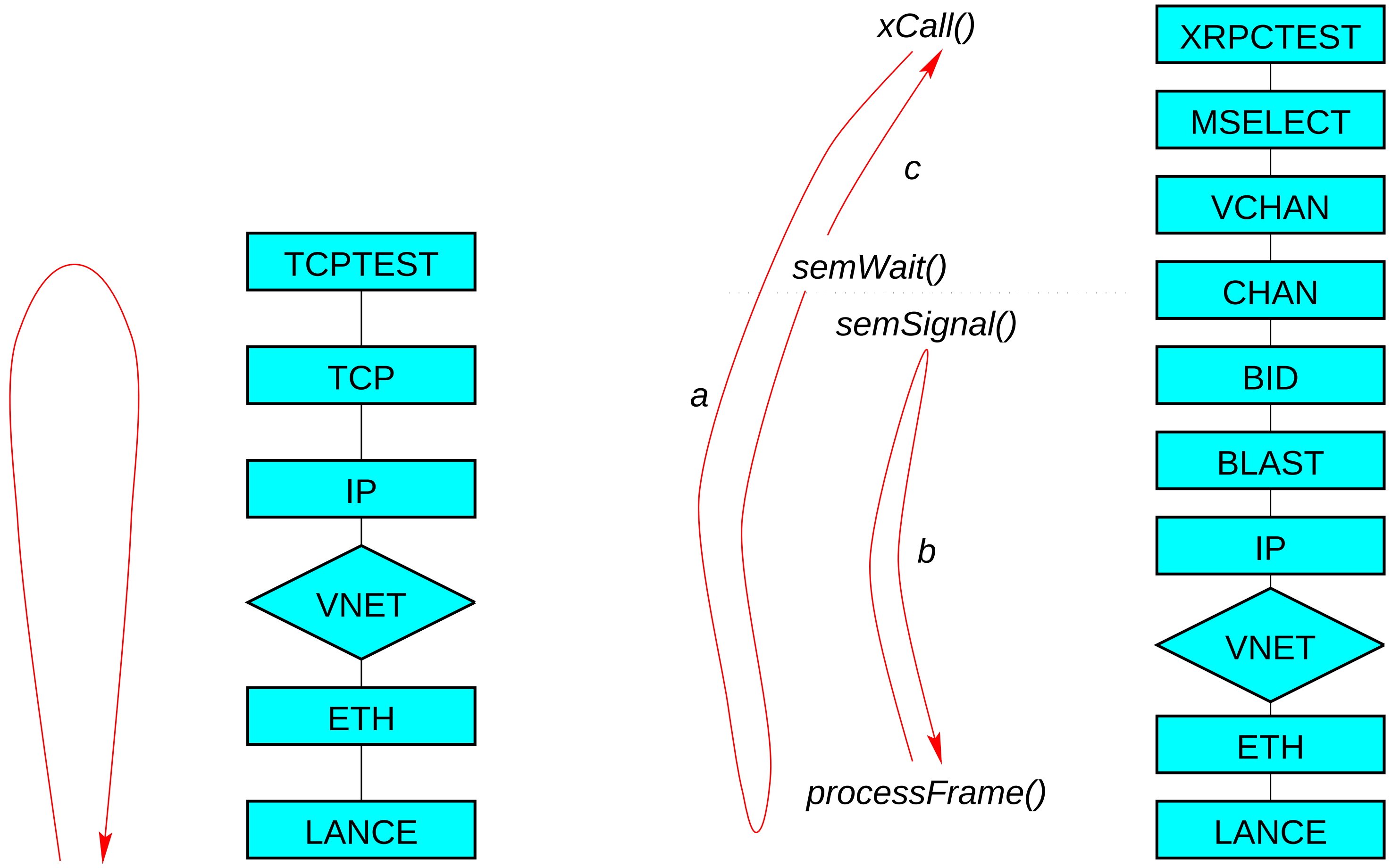
Conclusions

- Instruction cache bandwidth major bottleneck
- Cache collisions not particularly bad
- Processor/Memory gap still growing; now:
  - 300MHz processor
  - 100Mbps Ethernet
  - 80MB/s memory system

Conclusions

- Outlining
  - Readily applicable
  - Relatively convenient
- Cloning and path-inlining
  - Requires “path” notion: see Scout OS
  - Need better (automatic) tools

Dynamics


TCPTEST
TCP
IP
VNET
ETH
LANCE
xCall()
c
semWait()
semSignal()
a
b
processFrame()
XRPCTEST
MSELECT
VCHAN
CHAN
BID
BLAST
IP
VNET
ETH
LANCE