

# TCP Automatic Tuning Techniques for Distributed Computing



Eric Weigle and Wu-chun Feng  
RADIANT Team, CCS-1  
Los Alamos National Laboratory  
{ehw, feng}@lanl.gov

“Hi everybody!” at HPDC02, July 2002

# Introduction

- For good performance, distributed/grid applications need efficient networking—this commonly means optimizing TCP.
- The most important tuning parameter is buffer sizes; With an ideal size we:
  - Are not flow window limited (BIG buffers)
  - Maintain interactivity (small buffers).
  - Avoid loss— TCP AIMD *painful*; purposefully confuse flow/congestion control (buffer size below “instantaneous”  $bandwidth \times delay$ ).

With TCP, you can't have it all.

- This talk compares various techniques that perform this tuning automatically.  
Outline:
  - Taxonomy
  - Compare/contrast
  - Experiments
  - Results

## Who cares? (The Joy of Buffer Tuning)

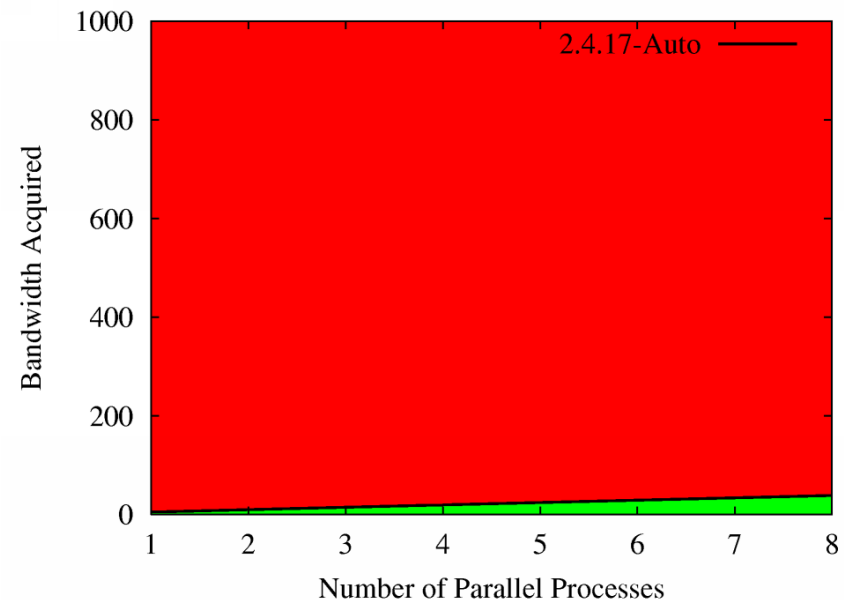
- We need to ensure our buffers are large enough to “fill the pipe”— That is, they must be able to hold a *bandwidth*  $\times$  *delay* product’s worth of data.
- On a 1Gbps link with 100ms delay that’s

$$1Gbps \times 100ms \times \frac{125MB}{1Gb} \times \frac{0.001s}{1ms} = 12.5MB$$

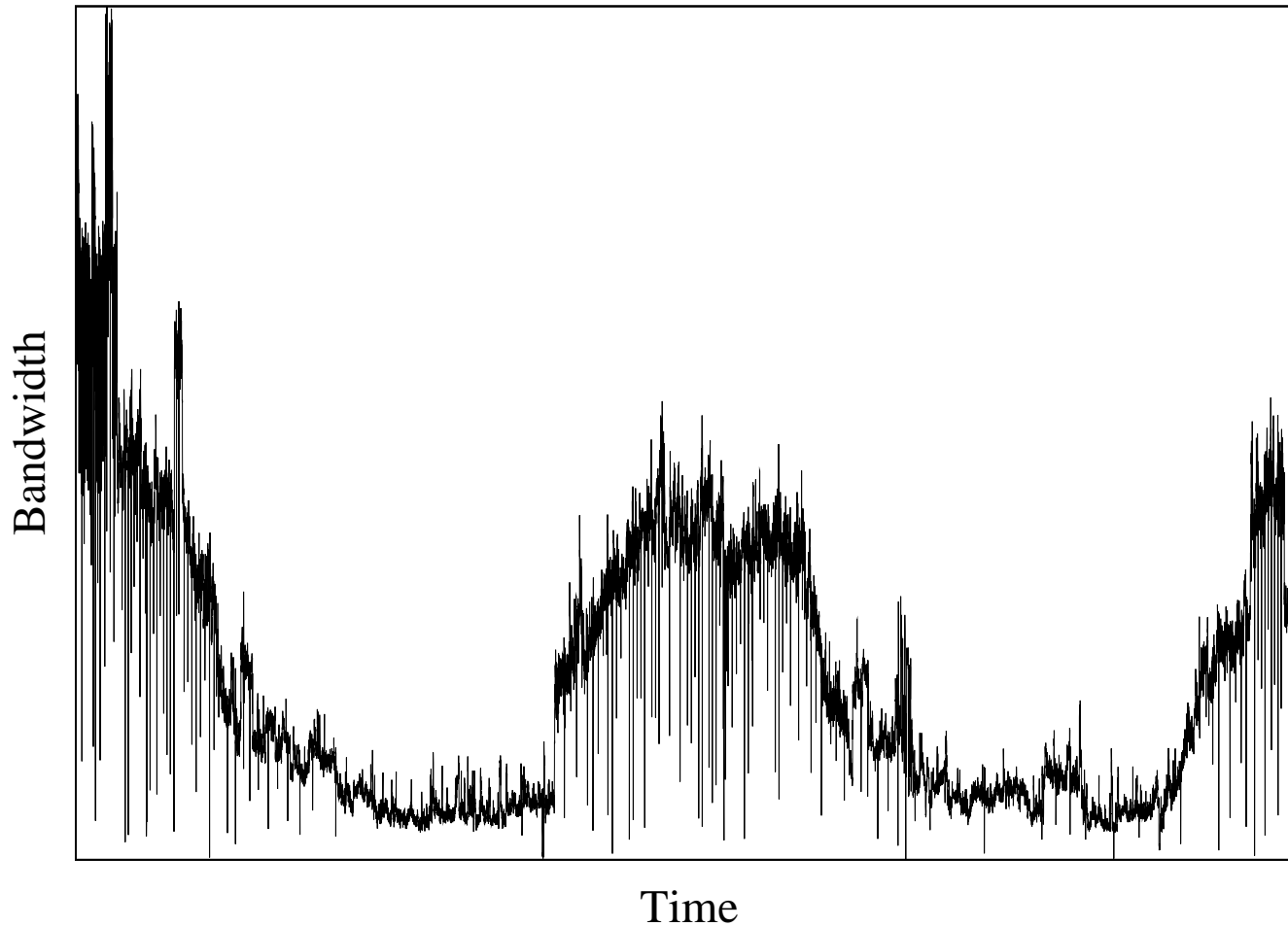
- But stock buffers are far too small - only 64KB! Stock flows will use:

$$\frac{64KB}{12.5MB} \approx .005, \text{ or } \frac{1}{200} \text{th of available bandwidth.}$$

Here we see the bandwidth wasted (Red) by a stock Linux 2.4.17 kernel (Green) for this case.



# Why isn't static hand-tuning good enough?



48 hours (noon-noon) on the LANL backbone at 1s granularity

# Taxonomy of Tuning Techniques (1 of 2)



## 1. No tuning

- High Performance? What's that?

## 2. Manual tuning – our baseline.

- A human gathers data on the network and sets “best guess” buffer sizes.

## 3. PSC's Automatic Tuning – in NetBSD 1.2

- Mostly sender-based approach. Sender uses header info to guess  $bandwidth \times delay$  product. The receiver simply advertises the maximal possible window.

## 4. Dynamic Right-Sizing (DRS) – in Linux 2.2 and 2.4;

- Mostly receiver-based approach where the receiver estimates  $bandwidth \times delay$  and congestion-control state of sender; receiver advertises a window large enough that the sender is not flow-window limited.

## Taxonomy of Tuning Techniques (2 of 2)

### 5. Linux 2.4 Auto-tuning

- Network unaware memory management technique; increases/decreases buffering based on available system memory and utilized socket buffer space.

### 6. *Enable* tuning

- A daemon collects data on network state and saves it to a database. Hosts query the database to set their buffer sizes.

### 7. NLANR's Auto-tuned FTP (in ncFTP)

- FTP program probes network at start of connection and sets buffer sizes appropriately.

### 8. LANL's DRS FTP (in wuFTP)

- FTP program uses a new control language command to gain network information, which is used to tune buffers during the life of a connection.

## Tuning Techniques– Features

Tuning	Level	Changes	Band	Visibility
PSC	Kernel	Dynamic	In	Transparent
Linux 2.4	Kernel	Dynamic	In	Transparent
DRS	Kernel	Dynamic	In	Transparent
Enable	User	Static	Out	Visible
NLANR FTP	User	Static	Out	Opaque
DRS FTP	User	Dynamic	Both	Opaque
Manual	Both	Static	Out	Visible

**User-level versus Kernel-level:** where does the buffer tuning occur– via a user `setsockopt()` call or in the kernel itself?

**Static versus Dynamic:** are tuned buffers a constant size set at the start of a connection, or does the size change with network “weather”?

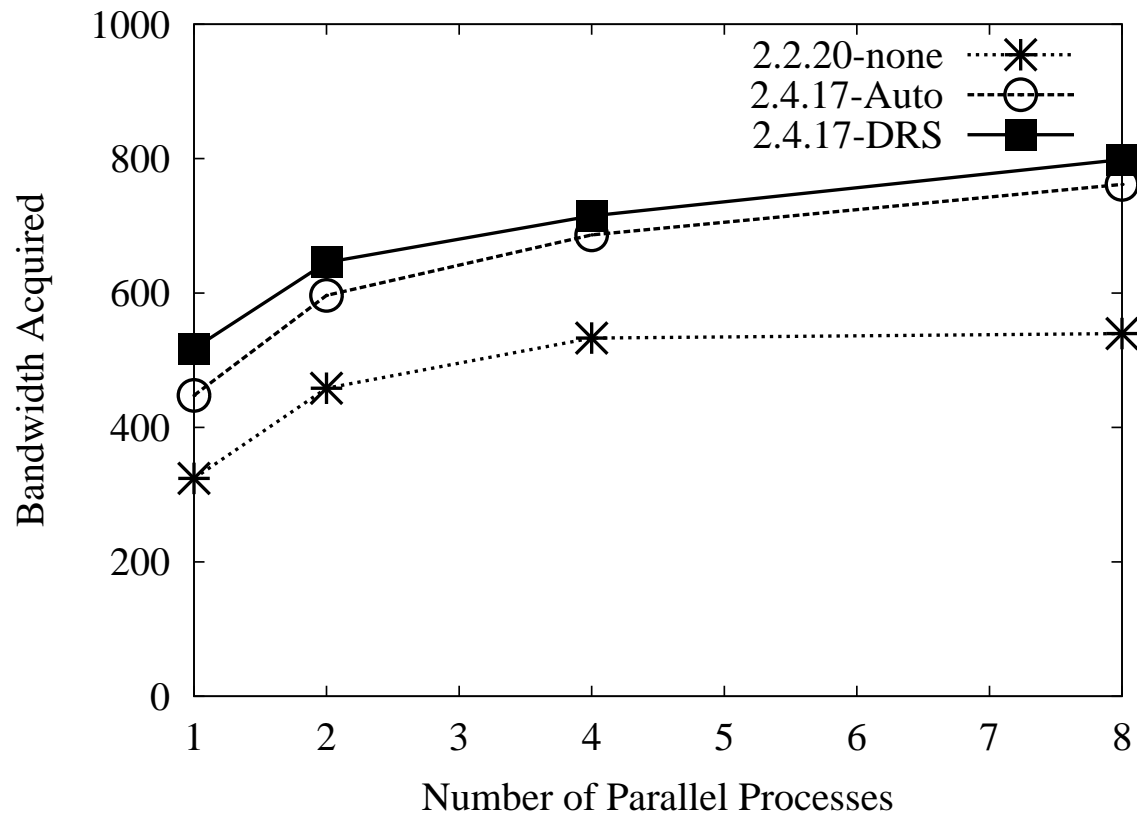
**In-Band versus Out-of-Band:** is tuning information gathered via the connection itself or by some other means?

**Transparent versus Visible:** how inconvenient is it for a user to deal with a tuning method?

# Experiments – Parameters

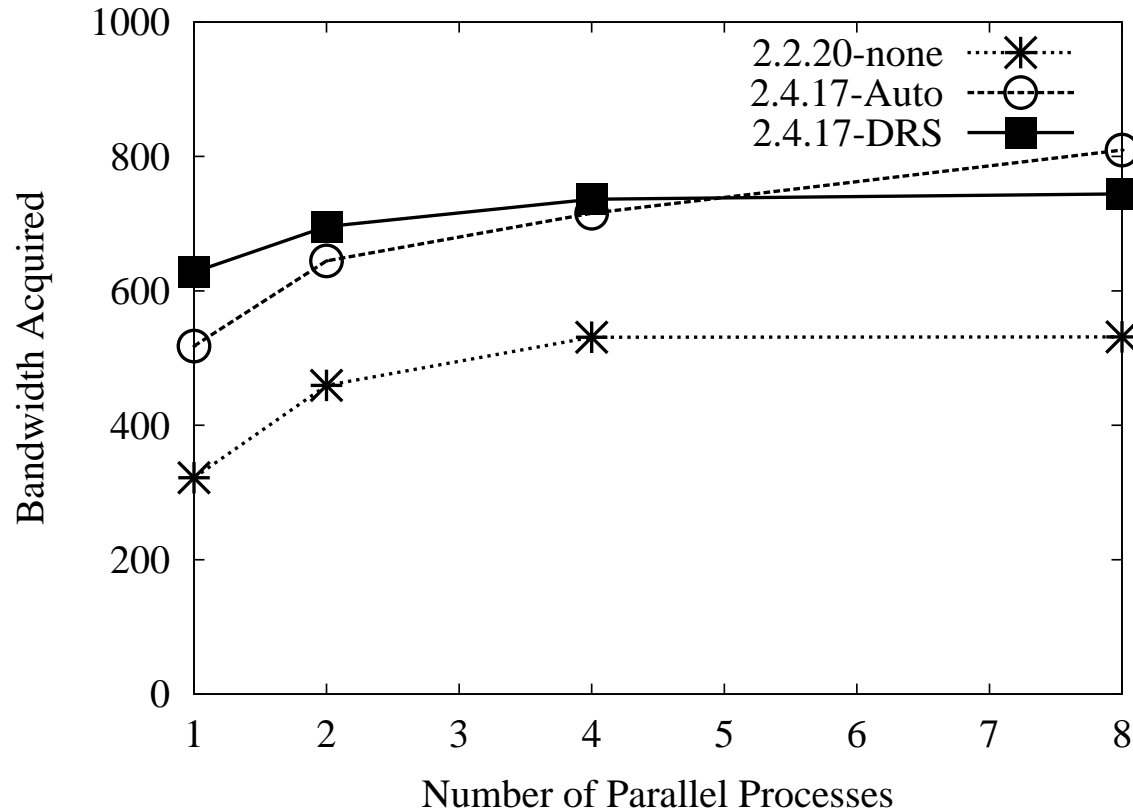
- Constant Parameters
  - **Topology:** 1 host to 1 host; possibly multiple streams per host.
  - **Unidirectional Transfers:** Source to destination.
  - **Loss:** None on purpose; may occur on-host (buffer over-runs/scheduling).
  - **Data Transfer:** Simple TCP-based program to mimic message-passing traffic (1MB message, 128 times).
  - **Hardware:** Two machines: dual 933MHz PIII, 512MB memory, 64bit/66MHz PCI bus, Alteon Tigon II Gigabit Ethernet card.
- Varied Parameters
  - **Tuning:** None, Manual, 2.4-Autotuning, Dynamic Right-Sizing (DRS).
  - **Max/Default Buffer Sizes:** 32KB to 32MB.  
May be set at user level (`setsockopt()`), kernel level (`/proc`), or both.
  - **Network Delay:**  $\approx$ 0.5ms, 25ms, 50ms, 100ms.  
Shows differences between LAN and WAN environments.
  - **Parallel Streams:** 1, 2, 4, 8.  
Shows tuning scalability, effectiveness of this commonly-used technique.

## No Tuning, 0.5ms



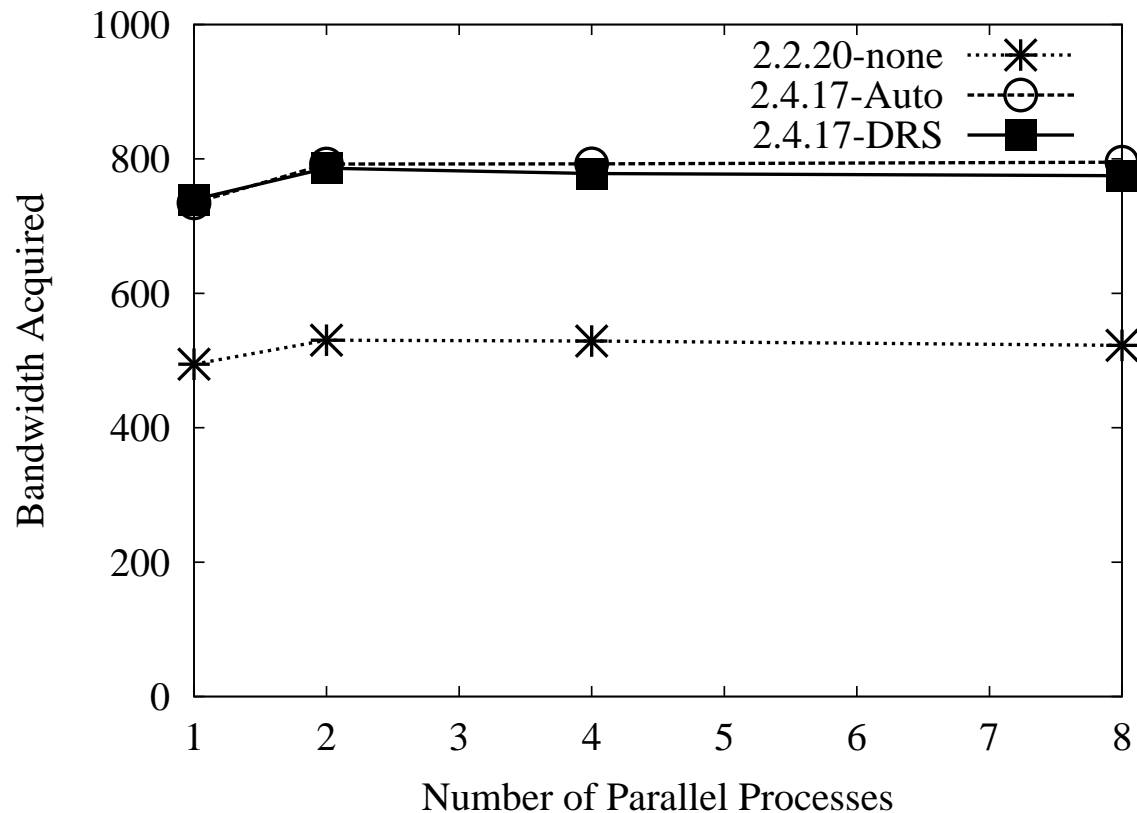
- With such short delays, default 64KB buffers suffice. ( $bw \times delay = 62.5KB$ )
- Software bottlenecks in NIC firmware/driver/stack cause peak at 800Mbps.
- DRS outperforms stock 2.4 stack which outperforms 2.2 stack. due to stack improvements.
- All benefit from use of parallel streams – due to effectively super-exponential slow start and additive increase by N.

## Kernel-Only Tuning, 0.5ms



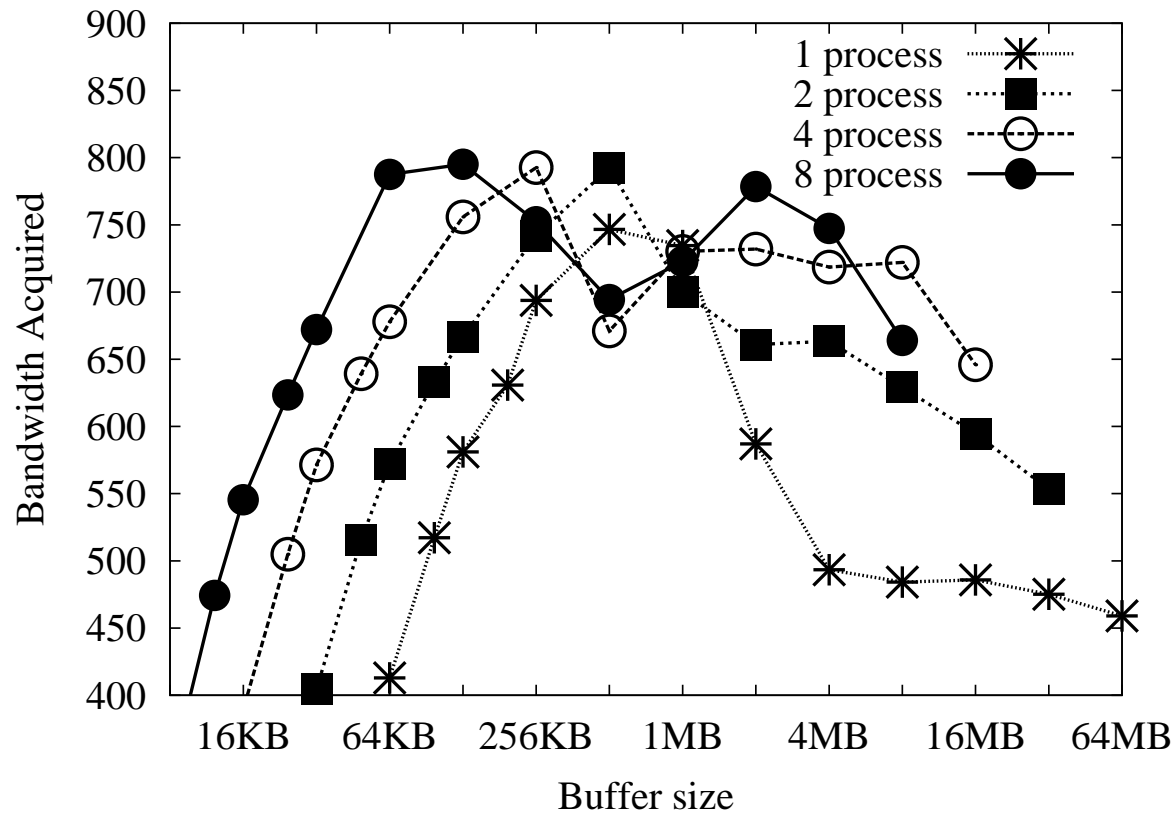
- Increase the maximum memory the kernel may allocate to a connection.
- No significant change in results for 2.2.20 (which does no autotuning).
- One or two 2.4.17-Auto connections perform better than in the untuned case, showing that the default 64KB buffers were insufficient (due to host delays).
- As the number of processes increases, DRS performs worse!  
Too aggressive in buffer allocation; over-allocations can be bad.  
TCP congestion-control & starvation of one or more processes.

## User/Kernel Tuning with Ideal Sizes, 0.5ms



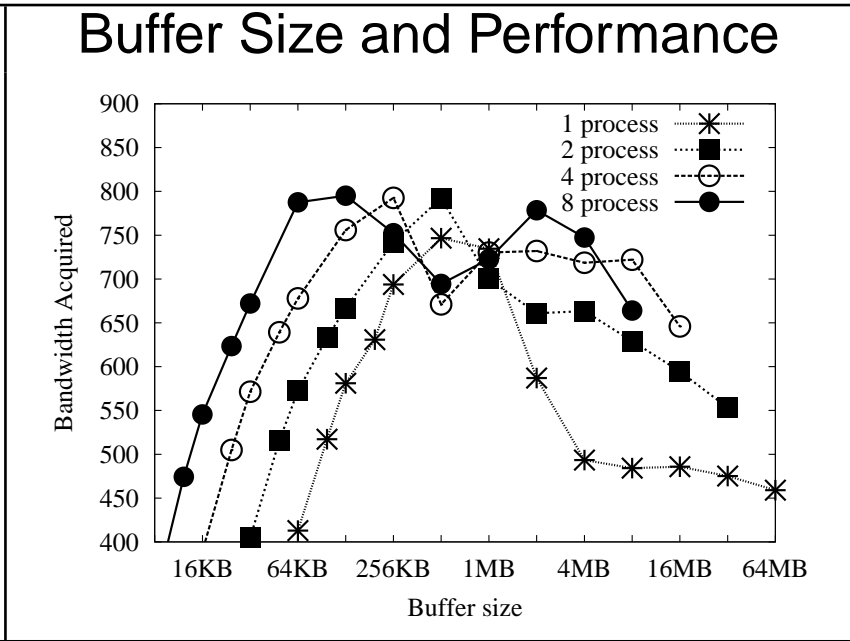
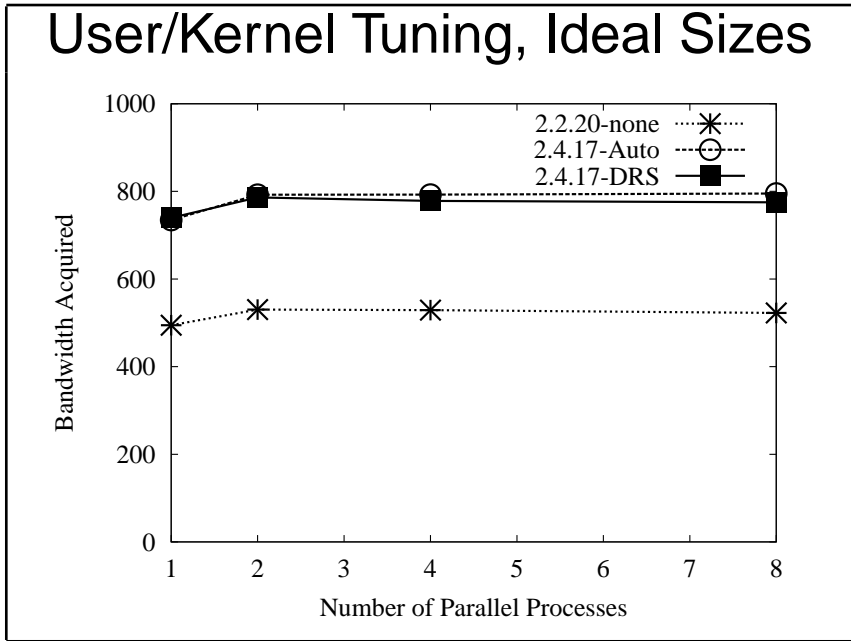
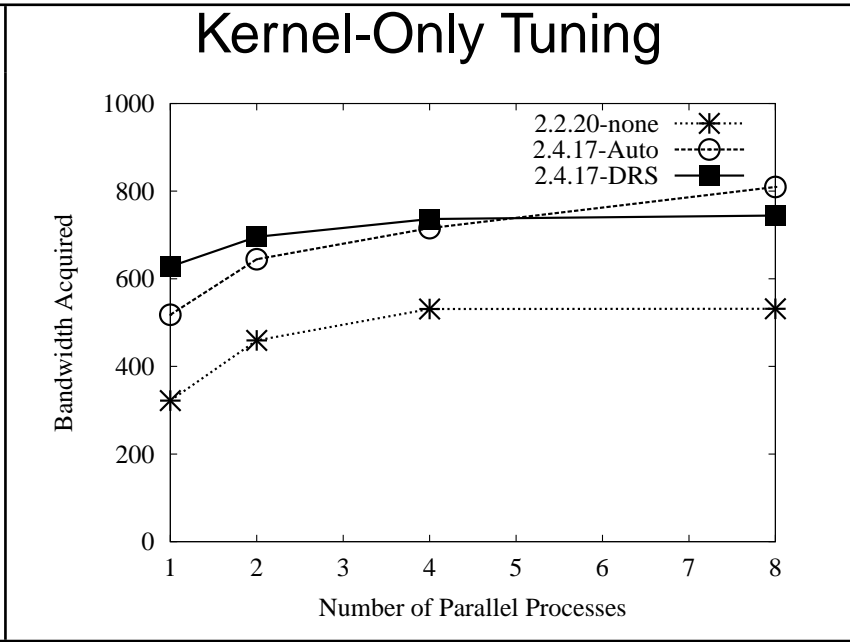
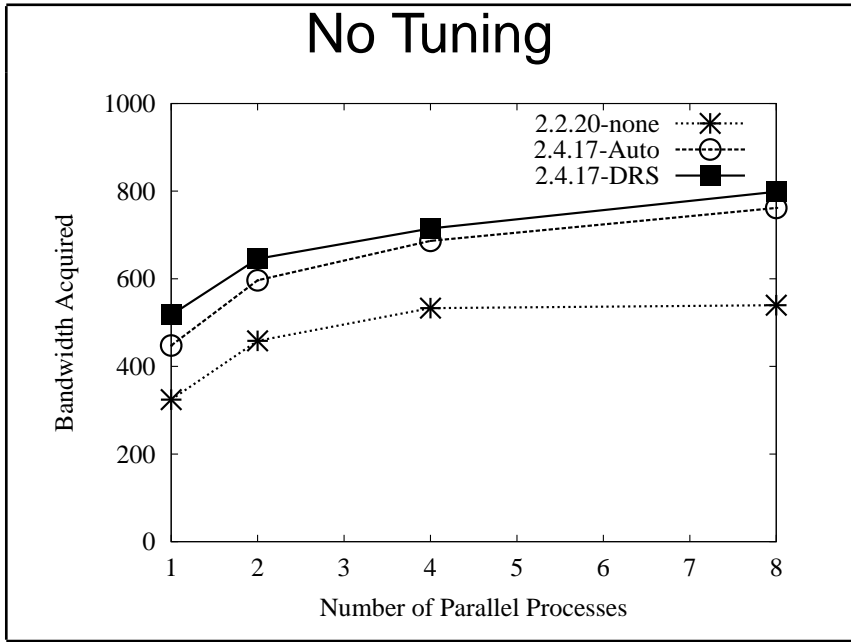
- DRS obeys the user when buffers are set by `setsockopt()`; 2.4.17-`{Auto,DRS}` use the same buffer sizes and perform about the same. DRS memory checks make *slightly* less efficient stack.
- Stack improvements in 2.4.17 give better performance than 2.2.20.

## Buffer Size and Performance, 2.4.17-Auto, 0.5ms

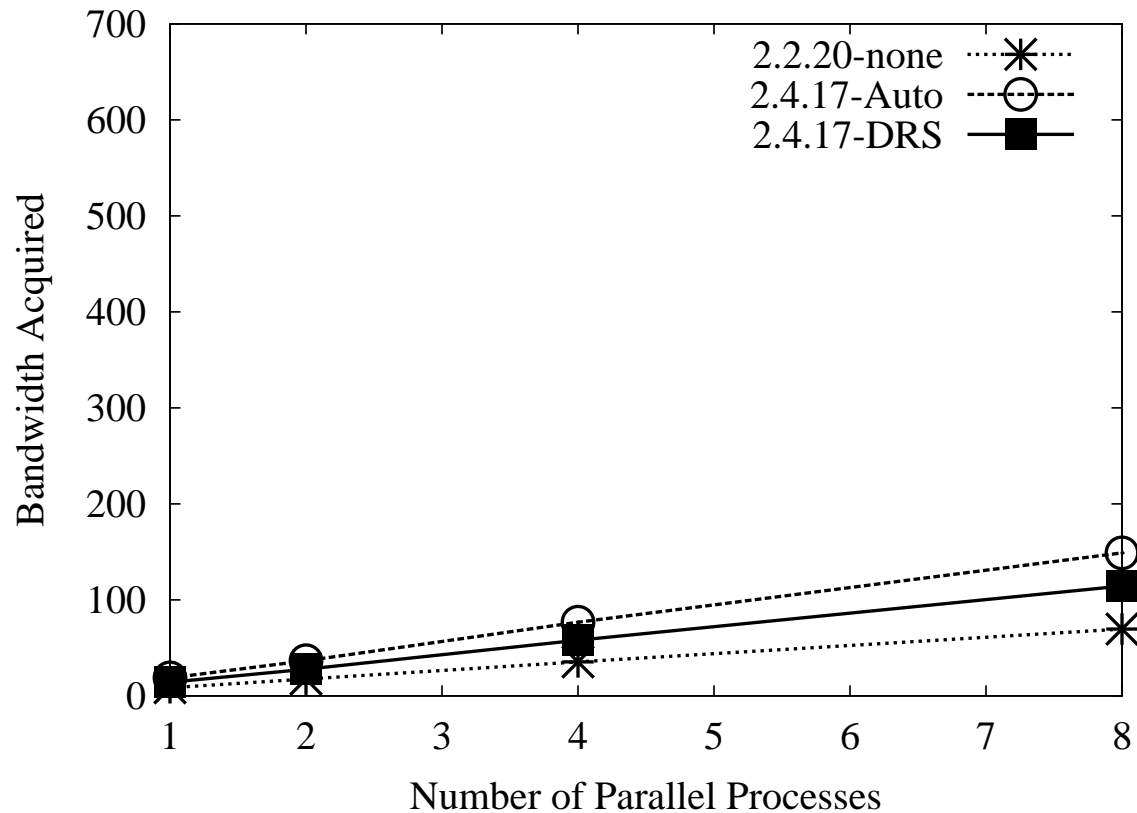


- “Ideal” buffer sizes are larger than one might expect (64KB).
- We need larger buffers to handle bursty (AIMD) TCP traffic.
- Feedback: large buffers can increase effective delay, and AIMD problems more dramatic with large buffers.
- Scheduling problems: buffers may fill (and block) while we process other sockets. This causes the dip seen with 4 and 8 processes around 1MB.

# Summary, $\approx 0.5\text{ms}$ Delay

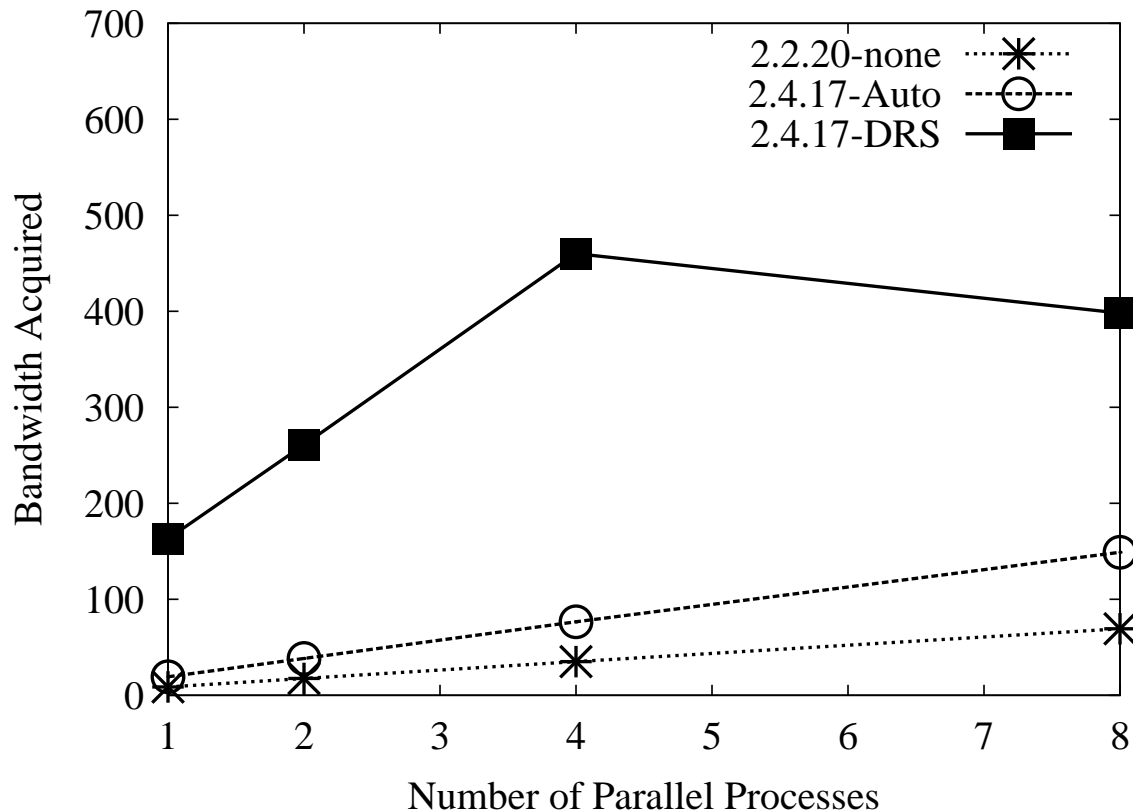


## No Tuning, 25ms



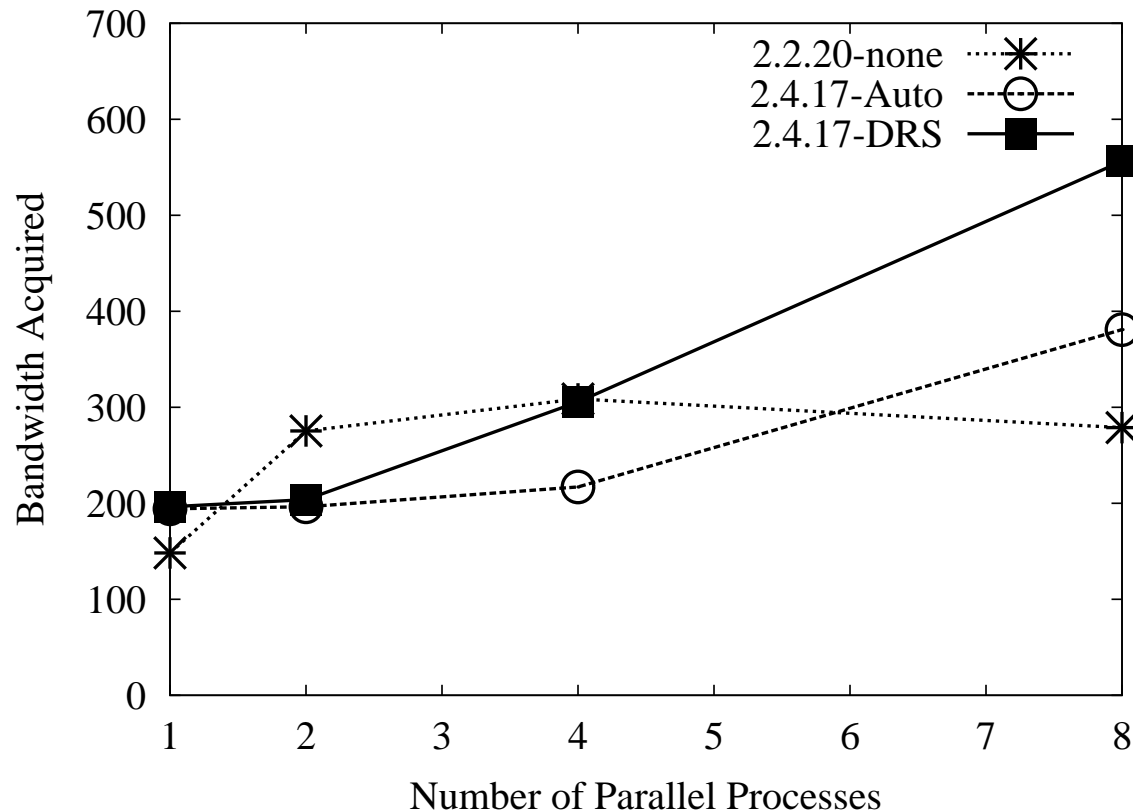
- With more realistic delay, we have  $bandwidth \times delay \approx 3MB$ .
- Reveals default configuration inappropriate for high performance.
- Linear speed-up with more processes, as effective flow window grows.
- Simple autotuning outperforms DRS; more effective with small windows.

# Kernel-Only Tuning, 25ms



- DRS improves, simple autotuning and stock connections are constant.
- Again the performance of DRS fall as the number of processes grows.
- The advertised window scaling factor in Linux 2.4.x is based on initial buffer size, not the maximal buffer size up to which Linux can tune.  
With only a *maximum* value set, Autotuning advertises no window scaling!

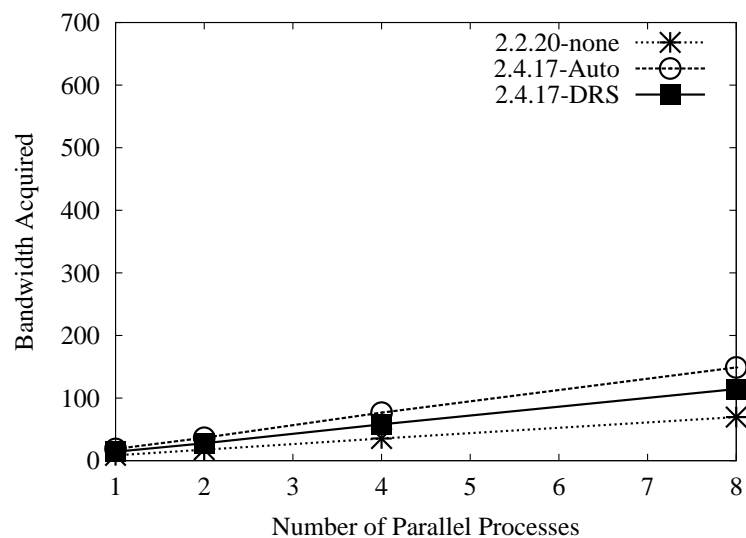
## User/Kernel Tuning with Ideal Sizes, 25ms



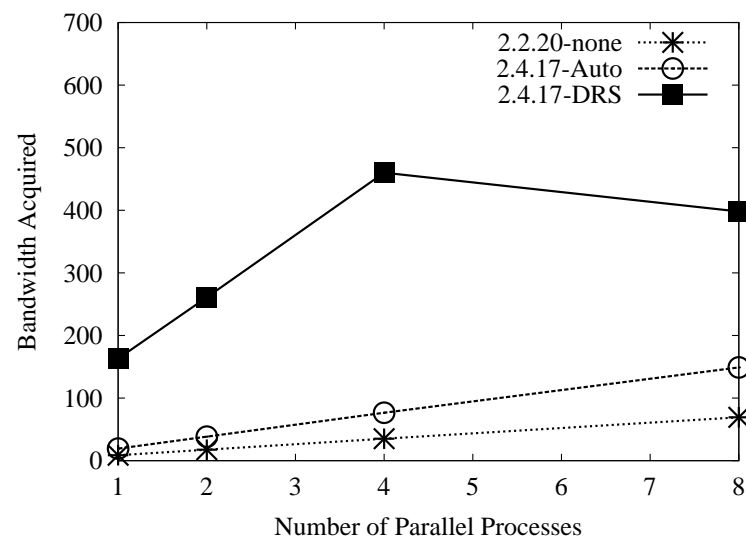
- With user and kernel tuning, maximal performance increases for all stacks.
- Performance does fall for DRS in the two and four process case – second-guessing the kernel can cause problems.
- DRS significantly outperforms 2.4 Autotuning because of window advertisement algorithm.

# Summary, 25ms Delay

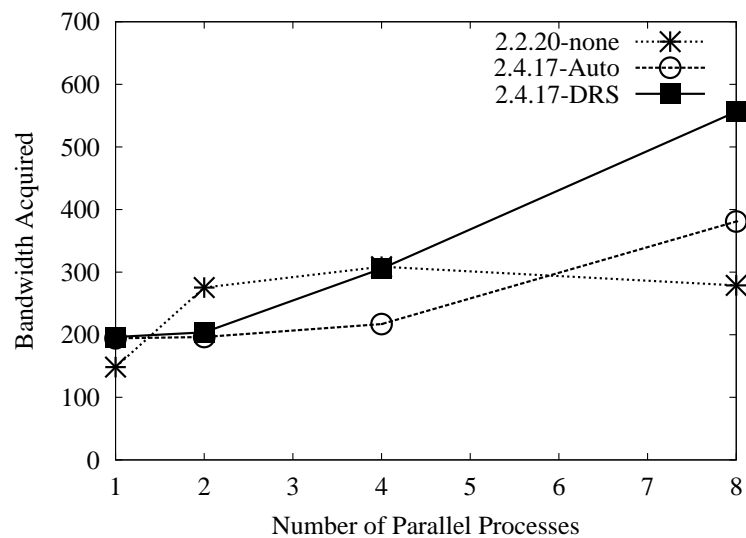
## No Tuning



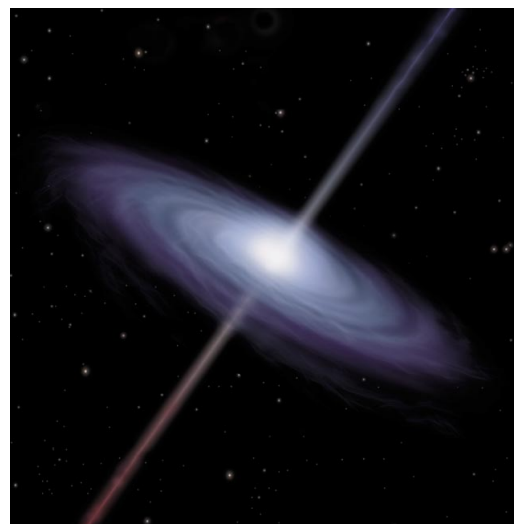
## Kernel-Only Tuning



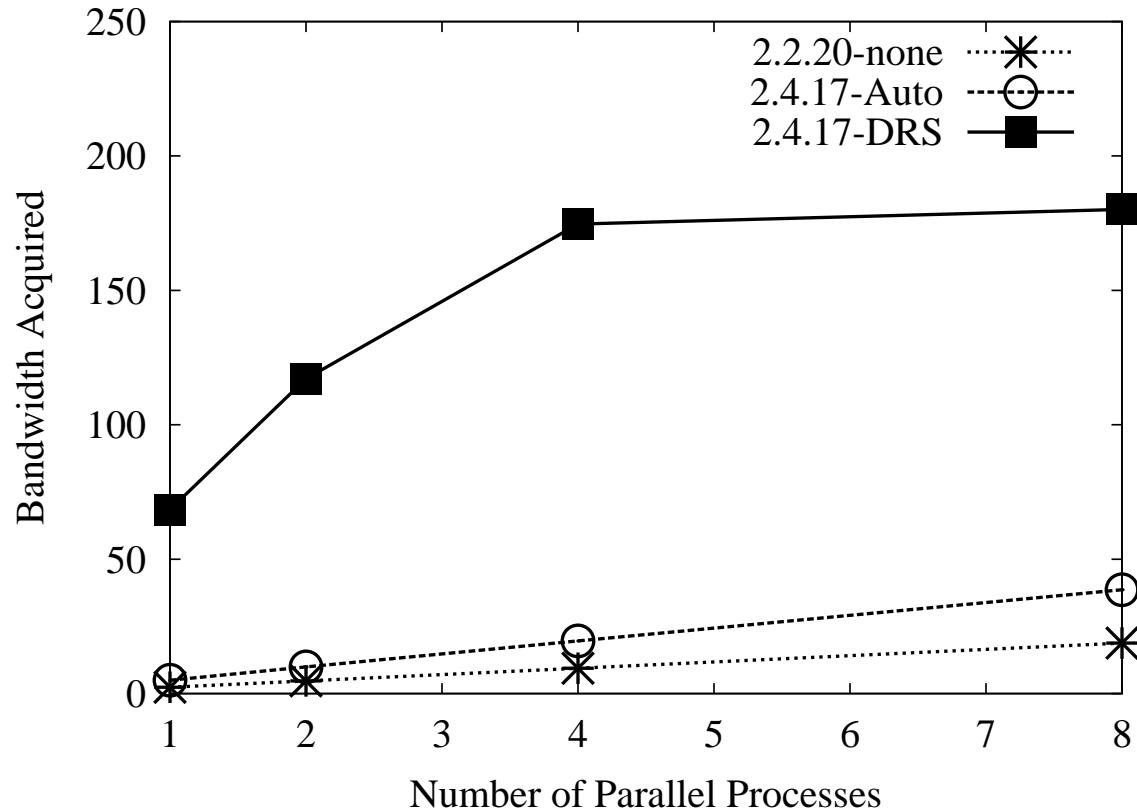
## User/Kernel Tuning, Ideal Sizes



## This Space Intentionally Left Blank

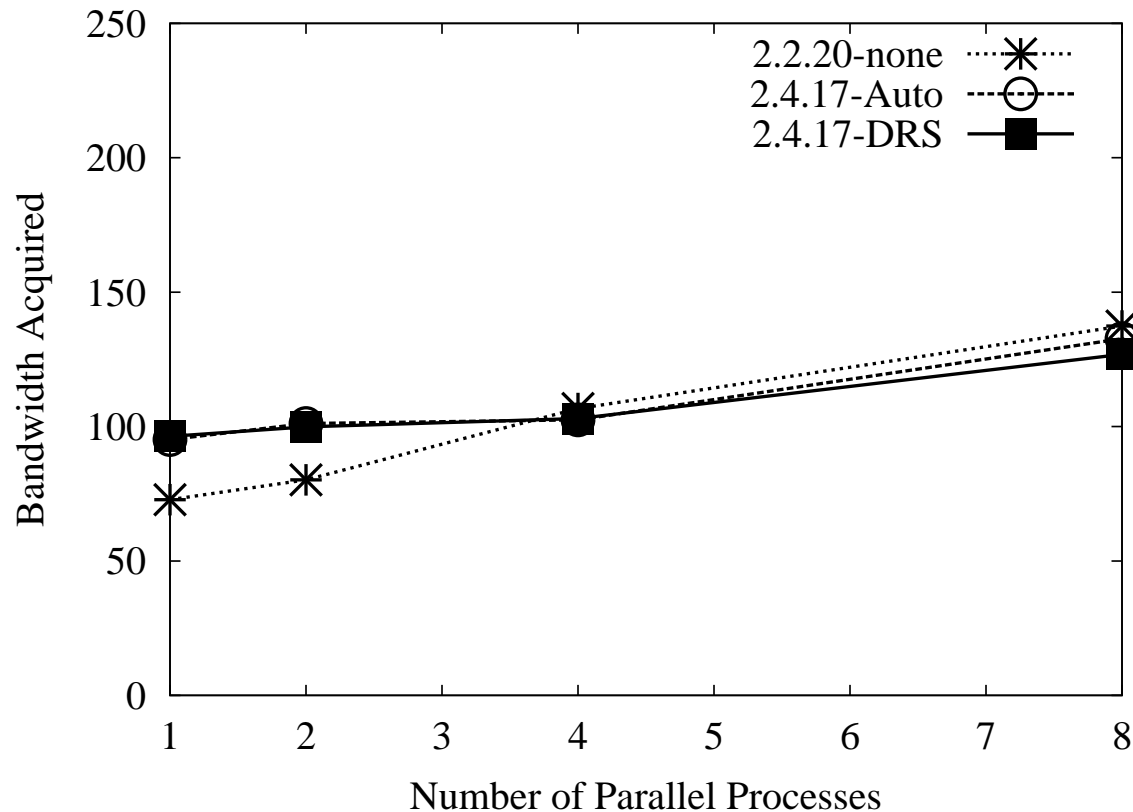


## Kernel-Only Tuning, 100ms



- We skip repetitive graphs of untuned cases; this equation captures their behavior with error below 20%:  
 $\text{Bandwidth(Mbps)} = \alpha(\text{processes}/\text{delay})$ , where  
 $\alpha = \{ 2.2.20\text{-None: } 214, 2.4.17\text{-Auto: } 467, 2.4.17\text{-DRS: } 355 \}$ .
- For delay  $> 25\text{ms}$ , TCP behavior dominates NIC/OS/etc. features.
- As earlier, DRS outperforms the other stacks. (Recall 2.2.20-None and 2.4.17-Auto do not benefit from kernel-only tuning).

## User/Kernel Tuning with Ideal Sizes, 100ms



- We see similar behavior to earlier graphs; 2.4.17-DRS and 2.4.17-Auto perform about the same, with 2.2.20-None slightly worse.
- When rates fall to  $\approx 100\text{Mb}$  (“old” Ethernet), the 2.2.20 stack performs well.
- Very large (multi-gigabyte, minimum) transfer sizes would be required to more fully utilize the network.

# Guidelines on Selecting an Auto-Tuned TCP

1. Willing & able to modify kernel, and run NetBSD or Linux?  
Use a kernel-level solution.
  - Like NetBSD?  
*Use PSC's approach.*
  - Many small connections, or willing to tune parallel streams?  
*Use Linux 2.4 autotuning.*
  - Few, large connections, unwilling to tune?  
*Use Linux with DRS.*
2. Unwilling or unable to modify kernel, don't use NetBSD or Linux?  
Use a user-level solution.
  - Just need FTP?  
*Use LANL's DRS FTP or NLANR's Auto-tuned FTP.*
  - Require multiple applications?  
*Use Enable.*

## When tuning, remember to....

- Ensure specific TCP options are enabled/disabled:
  - Window scaling on
  - Timestamps on
  - Selective acknowledgements on
  - Nagel algorithm off
- Set the maximum memory available to allocate per connection or for user-level tuning. Greater than the maximum  $bandwidth \times delay$  expected
- Set min/max memory for Linux 2.4 autotuning.
- (Optional) Flush caches in between runs so inappropriately set slow-start thresholds are not re-used.
- Beware of security (DoS) and performance issues with large buffers!

## Conclusion

- There are many approaches to getting good network performance.
- TCP auto-tuning techniques are one important facet of a larger problem.
- Select a technique appropriate for your circumstances.



Thanks! Questions?

<http://public.lanl.gov/ehw>  
<http://www.lanl.gov/radiant>

## Bonus slide: Performance is still atrocious, what now?

- Try other protocols:
  - Other versions of TCP; TCP Vegas, FACK, etc.
  - A reliable UDP (Only if you have a dedicated line).
  - Use RAPID.
  - “Never underestimate the bandwidth of a truck full of tapes hurling down the highway” – Andrew S. Tannenbaum
- Check for (proxying) firewalls between your source and destination; they may need tuning too.
- Slack. Network speeds will catch up.
- Buy more fiber.