Full-System Critical Path Analysis

Ali Saidi, Nathan Binkert[†], Steve Reinhardt[‡], Trevor Mudge

Advanced Computer Architecture Lab
University of Michigan

[†]Hewlett-Packard Labs
[‡]Reservoir Labs

Full-System Performance Analysis

- Today's complex workloads may not be CPU limited
  - E.g. webservers, databases, OLTP, etc
  - Multiple layers of HW (disk, network) and SW (app, OS)
  - No single metric to identify bottleneck
- No off-the-shelf tools analyze these systems
  - Local per-component analysis fails to account for overlapped latencies
  - Ad-hoc methodologies are workload specific
  - State-space exploration is slow
- How can we find bottlenecks across multiple layers?

Solution: Global Critical Path

- Global critical path indicates non-overlapped latencies
  - Directly identifying problem areas
- Used successfully in past in isolated domains
  - Fields et al. developed a model for out-of-order CPU
  - Barford and Crovella developed a critical-path model TCP
  - Yang and Miller extracted critical path information from message and synchronization calls

Solution: Global Critical Path

- **Challenge:** Create global dependence graph
  - Requires detailed knowledge across many domains!

- **Our solution:** automatically extract dependence graph from interacting state machines
  - Extract underlying state machines from HW and SW
  - Identify local interactions
  - Build an execution graph that captures the interactions
  - Find time where overlapping latencies not hidden

Contributions

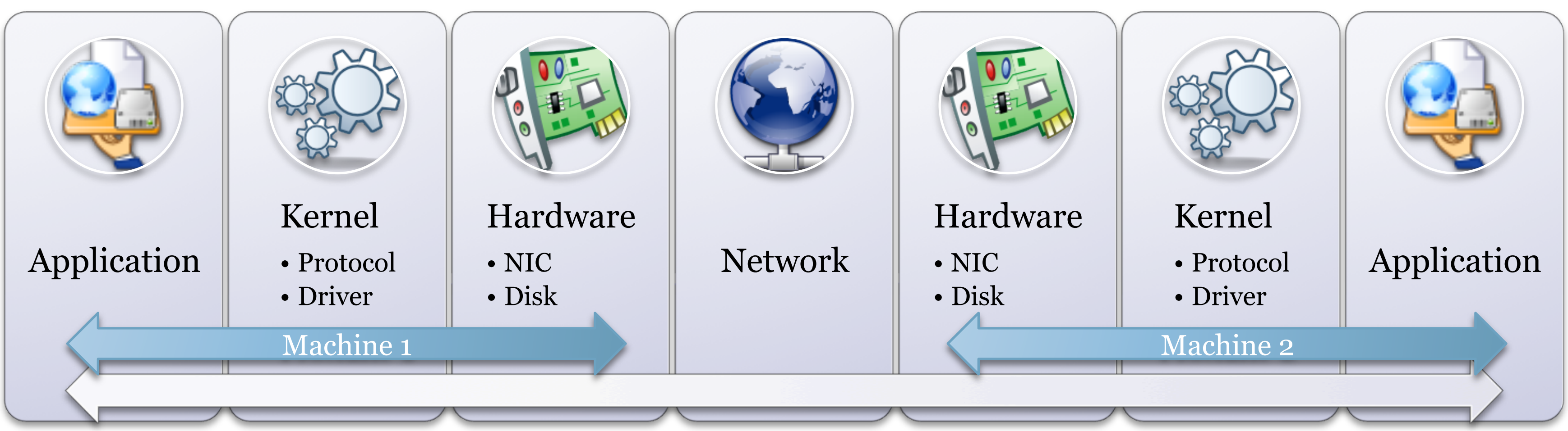
- Methodology to convert cooperating state machines into global dependence graph
  - Only local understanding required
  - Use dependence graph to find critical path
- Proof of concept implementation
  - Locate both hardware and software bottlenecks
- Sample analysis of UDP protocol
  - Found performance bugs in Linux 2.6.13

Outline

- Introduction
- Motivation
- Using critical path analysis
- Implementation
- Results
- Conclusion

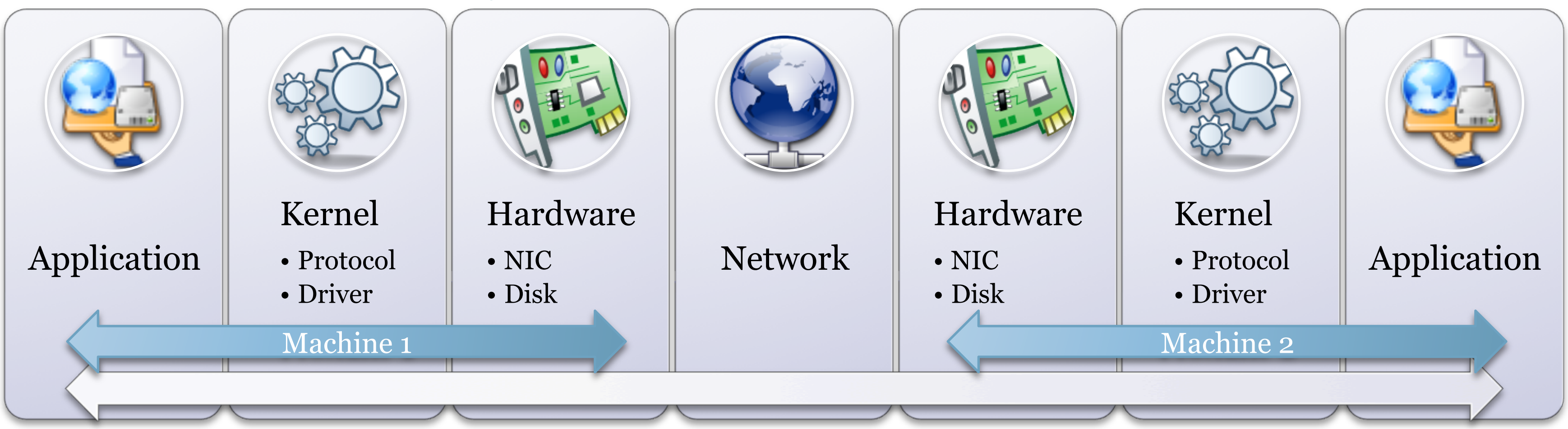
Performance Analysis Challenge: High-speed Networking

- Finding bottlenecks is particularly problematic in end-to-end networking
- Performance losses come from a combination of overheads and non-overlapped latencies



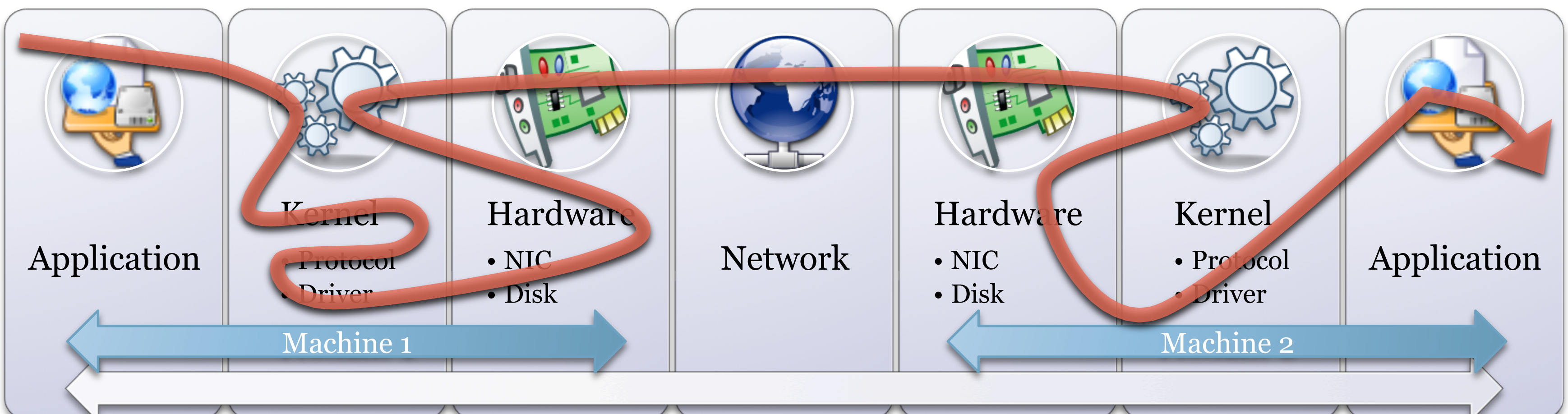
Conventional Tools Don’t Work

- Why not use a profiler?
  - Where most time spent ≠ bottleneck
    - Higher level dependence or protocol requirements that limit performance
  - Profilers only work on software



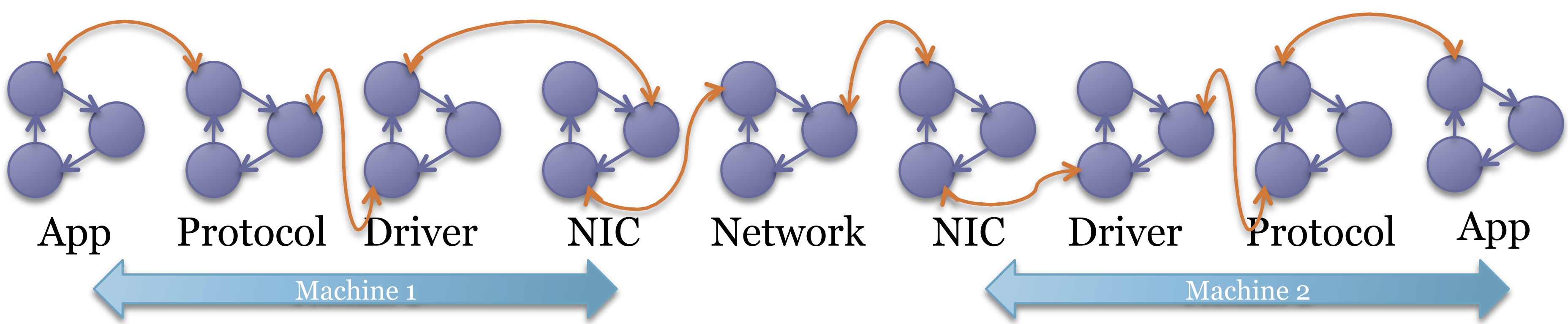
Critical Path Analysis

- Powerful technique to find performance bottlenecks in concurrent systems; However:
  - Requires a dependence graph
  - Detailed, domain-specific knowledge to create one

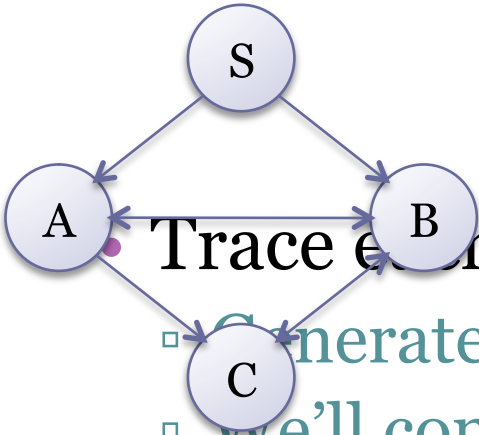


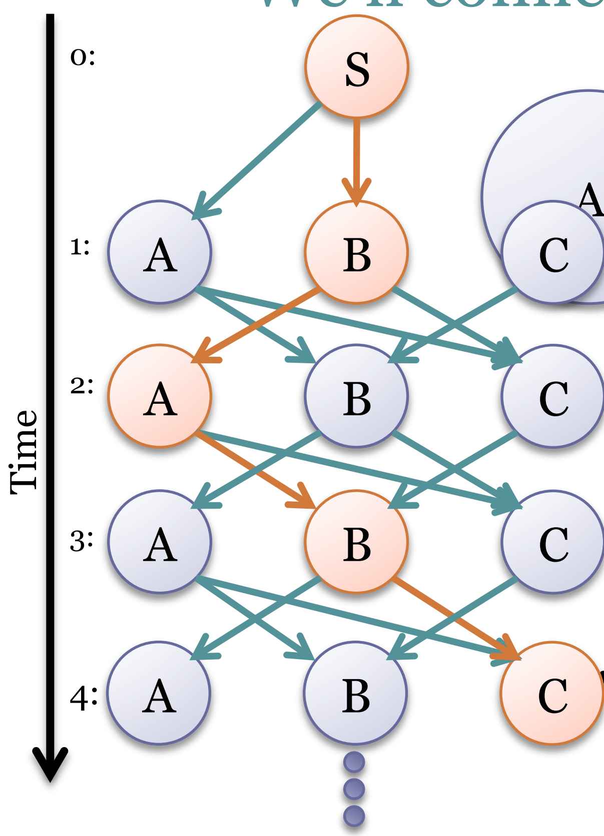
Constructing a Dependence Graph

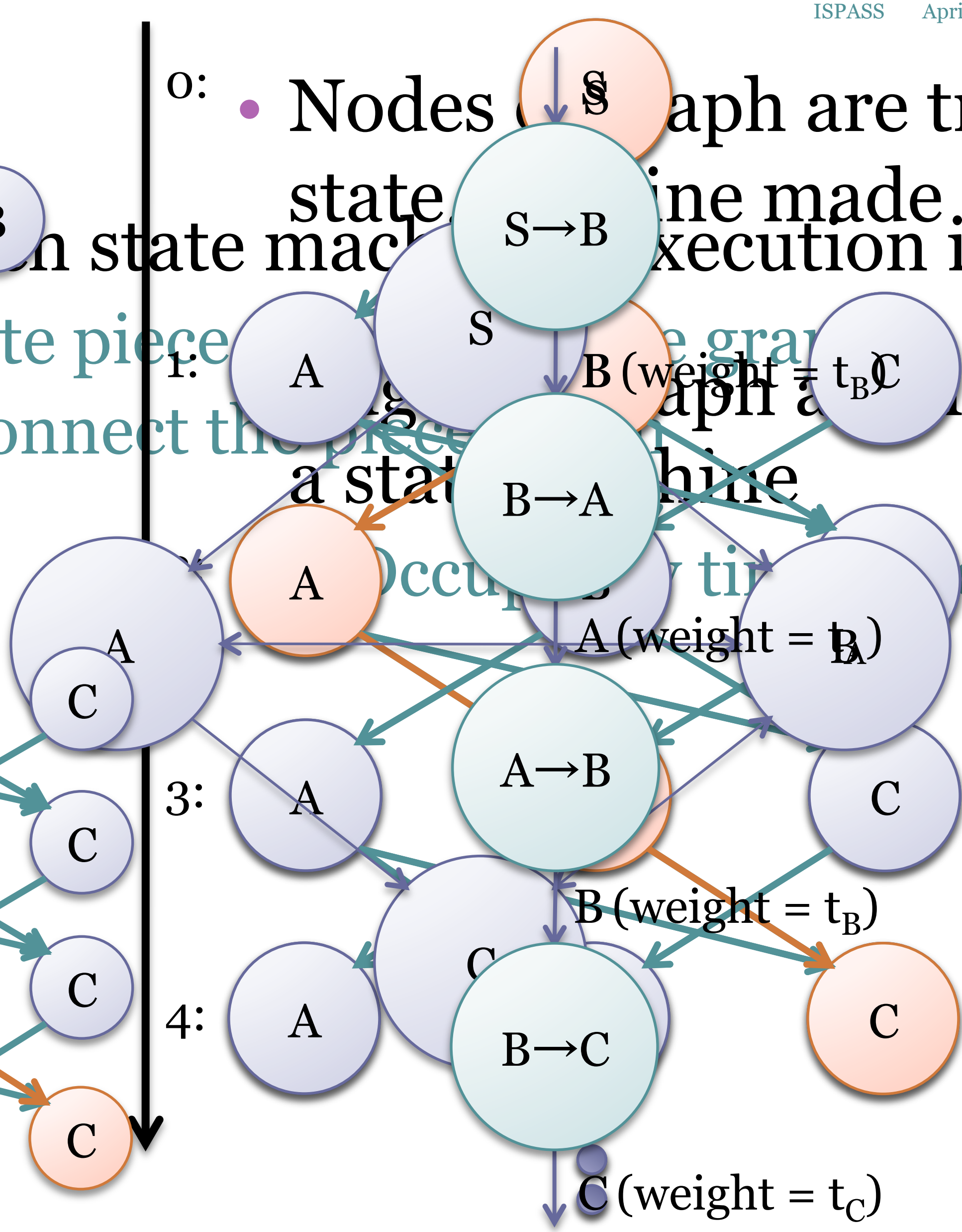
- **Key insight:** Systematically map state machines into a global dependence graph
  - Most HW is already specified as a state machines
  - Extract implicit state machines from SW
    - More on this later



- Nodes … aph are transitions state … ine made.
- Trace e…n state mac… xecution individually
  - G…nerate piece… e gra…
  - We'll connect the piece…
- … g … aph a… me spent in a stat… hine
  - Occu… y ti… dge weight







State Machine Interactions

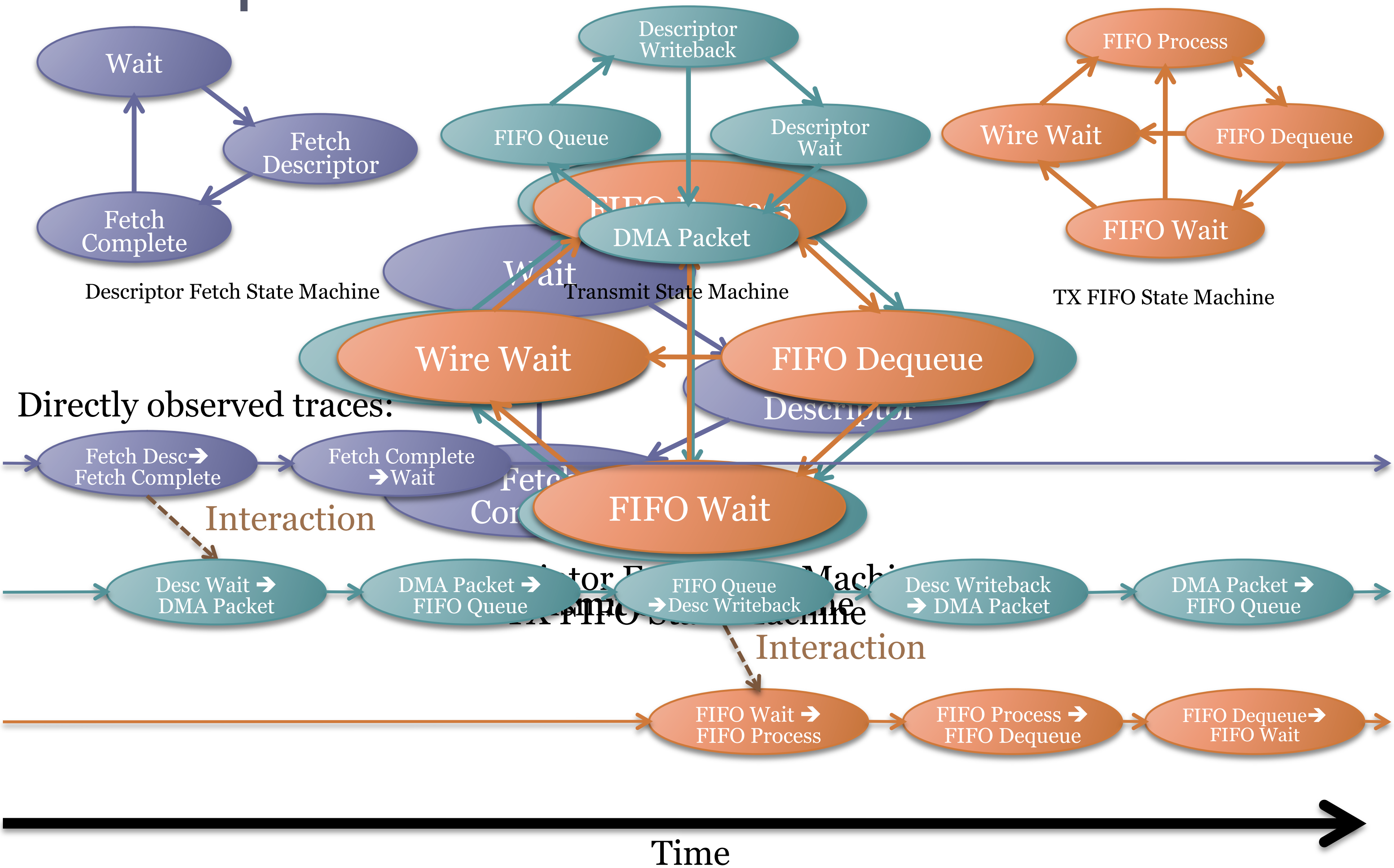
- Example illustrates conversion of single state machine to dependence graph
- State machines interact when one SM induces a transition in another SM
  - Insert edge between transition nodes in graph
  - Only way that subgraphs from different SMs interact
- SM transition plus inter-SM interaction events sufficient for global dependence graph

Identifying State Machines

- HW design often based on state machine
  - Mark interactions between state machines
- SW state machines typically implicit in code
  - Automatic function-based decomposition
  - Annotations for further refinement
    - Interactions between SMs
    - Multiple SMs in single code base (e.g., kernel)

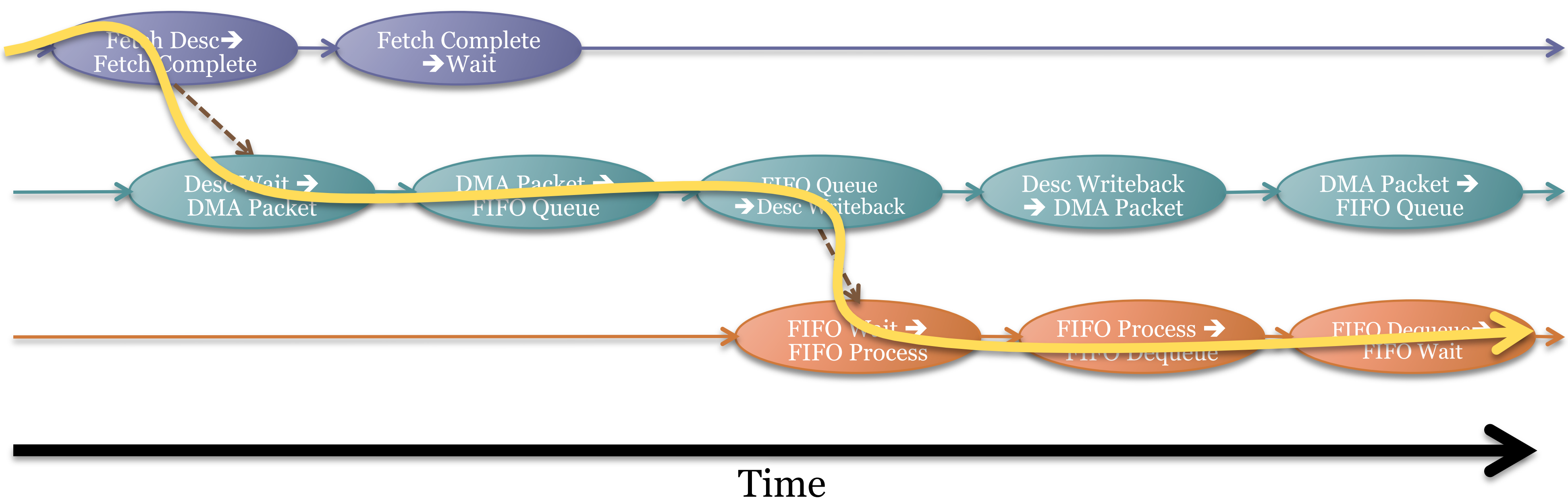
- Incremental: analysis finds incorrect assertions
- Less than 100 annotations for results in this talk

Example: NIC Transmit



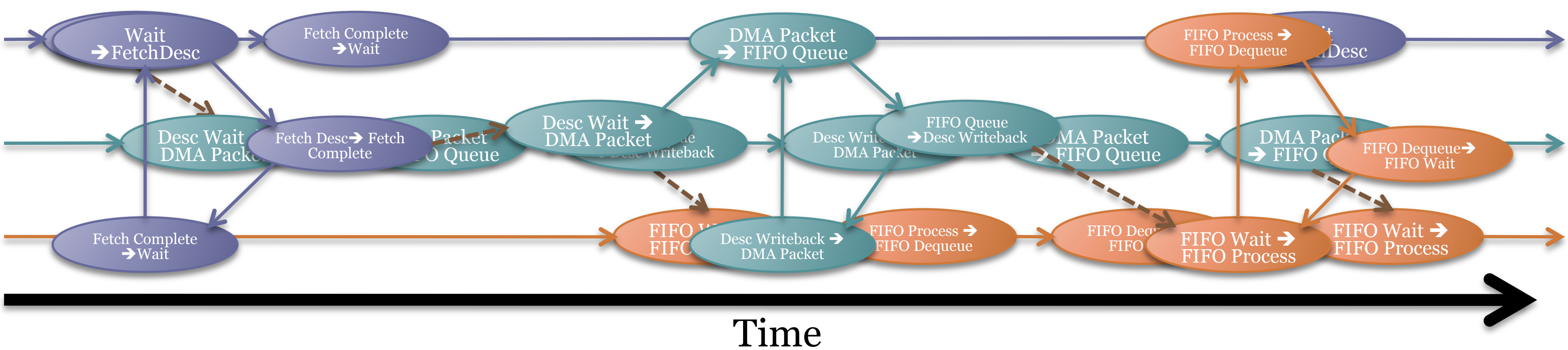
Finding Global Critical Path

- Use standard graph analysis techniques
- Path will likely go through same state multiple times
  - A state's criticality is time spent in that state divided by total critical path length
  - Use criticality to guide optimizations

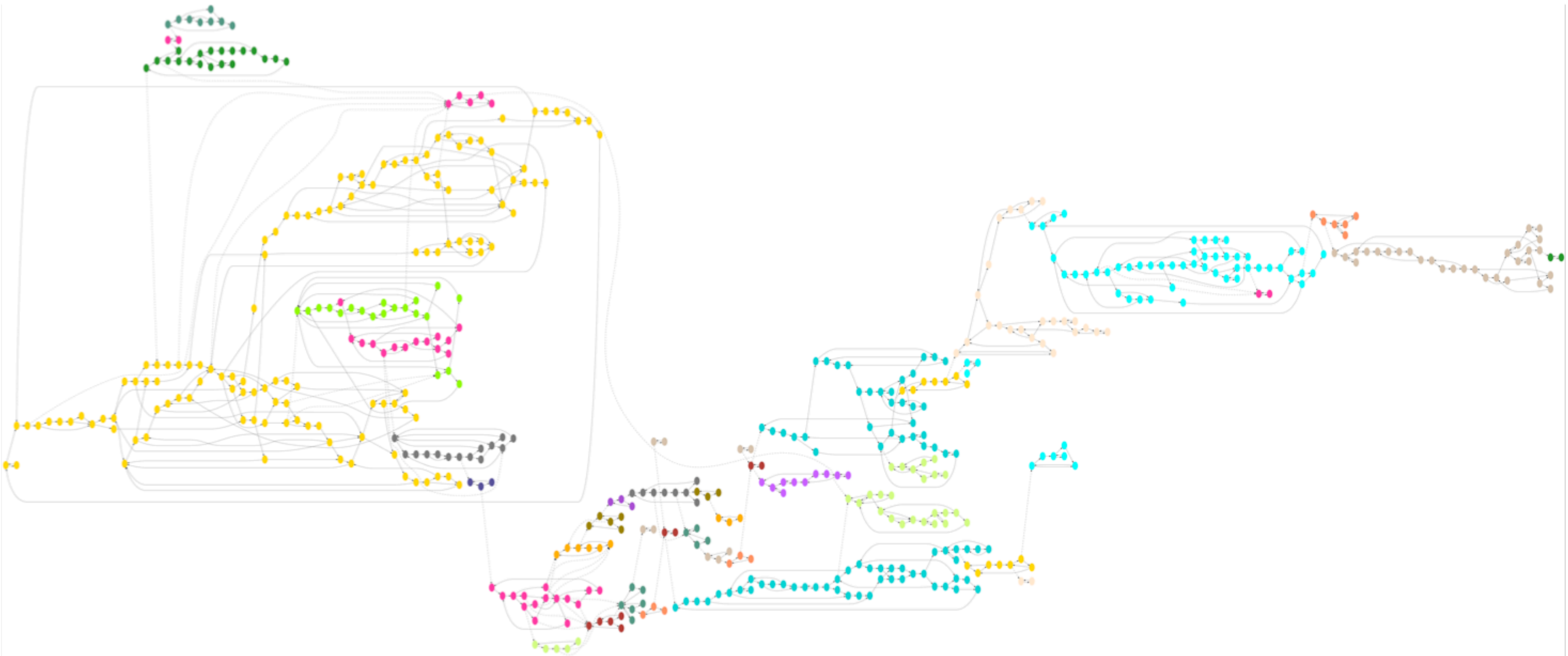


Result Visualization

- Analysis produces large graph
  - Unbounded in size
  - Fold traces back into "Combined Graph"
  - No node is ever repeated



Example UDP Protocol Graph



Started with ~1.5M nodes, this graph has 545

Evaluation Goals

- Artificially constrain the system
  - Verify that the analysis program can find the constrained resource
- Apply tools to an unconstrained system
  - See what problems our tool can find

Methodology

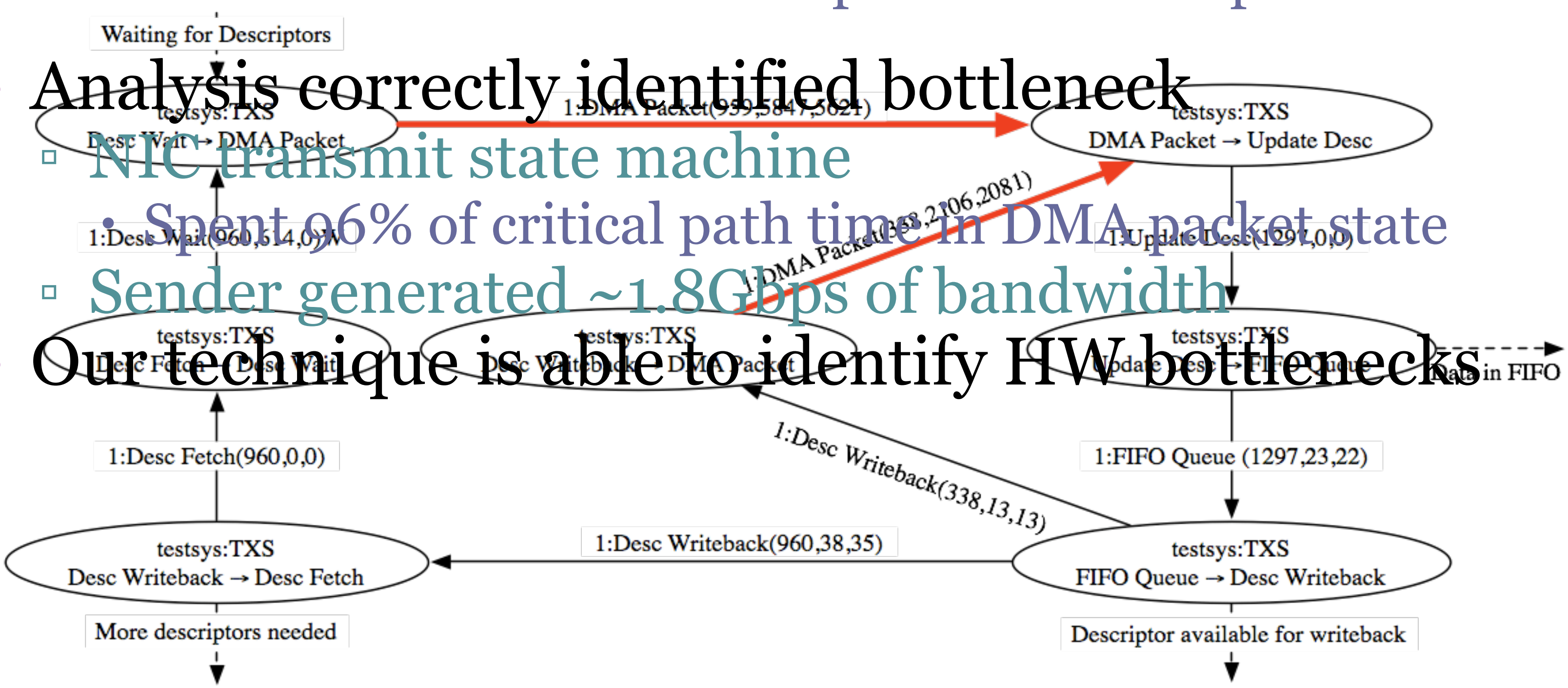
- Used M5 simulator
  - Provides deterministic results & visibility
  - Technique is not restricted to simulation
- Separate recording and processing
  - Allows for interactive analysis
- Records hardware state machine events directly
- Software state machine transitions are observed by simulator
  - Source code annotations used to capture other required data

Workloads

- Linux 2.6.13
- Netperf UDP stream test
  - Metric is bandwidth produced
- Hardware Evaluation
  - Artificially constrained system
  - Technique identified constrained resource
- Software Evaluation
  - Ran a variety of configurations
  - Found some interesting results

Constrained I/O Bandwidth

- Artificially made bottleneck I/O bandwidth
  - Limited I/O bus bandwidth to ~2Gbps
    - Not all of which is available for DMAing packets
    - Unconstrained benchmark produces 2.2Gbps
- Analysis correctly identified bottleneck
  - NIC transmit state machine
    - Spent 96% of critical path time in DMA packet state
  - Sender generated ~1.8Gbps of bandwidth
- Our technique is able to identify HW bottlenecks



Kernel Performance Bugs

- Ran a variety of different parameters
  - Payload size
    - 1480 bytes
    - 16KiB
  - Netfilter
    - Enabled
    - Disabled
- Different paths through the kernel
  - Resulted in different critical paths
- Expected UDP (user-to-kernel copy) to dominate
  - However, IP layer sometime does

UDP Sender w/Netfilter; 16KiB

State	Criticality	Profiler Rank
`IpSend:ip_defrag`	12.78%	4
`IpSend:memcpy`	9.59%	7
`IpSend:ipt_do_table`	6.94%	8
`IpSend:ip_copy_metadata`	6.90%	9
`IpSend:ip_fragment`	6.51%	10
`IpSend:ip_fast_csum`	4.87%	13

- Analysis shows IP layer dominates
- IP code fragments and then immediately defragments packet

UDP Sender w/Netfilter; 1480B

State	Criticality	Profiler Rank
`IpSend:ipt_do_table`	11.36%	2
`IpSend:csum_partial`	10.23%	3
`IpSend:nf_iterate`	7.38%	4
`IpSend:_read_unlock_bh`	3.61%	17
`IpSend:_read_lock_bh`	3.56%	19
`IpSend:memcpy`	3.47%	20

- Here again IP layer dominates
- Netfilter erroneously assumes it needs to checksum
- Lots of time used walking table of 0 netfilter rules

Found Problems in Real Software

- All problems fixed in 2.6.16 kernel
  - But we had no idea they existed
- When fixed, critical path returns to normal location of user-to-kernel copy

Conclusion

- Shown how to find bottlenecks in complex hardware and software systems
- Applied these techniques with a simulator
  - Hardware issues
  - Software issues
- Future
  - Expand to more complex workloads
  - Analyze software on real systems

Questions?