Performance Forecasting: Finding bottlenecks before they happen

Ali Saidi†◊, Nathan Binkert‡, Steve Reinhardt§†, Trevor Mudge†

† – University of Michigan
‡ – HP Labs
§ – AMD Research
◊ – ARM R&D

June 23, 2009
Today’s workloads & systems are complex
- Many layers of HW (disk, network), SW (app, OS)

How to evaluate systems in design stage?
- Where are the bottlenecks?

Conventional tools inadequate
Solution: Global Critical Path

- Directly identifies true bottlenecks
  - Accounts for overlapped latencies
- Used successfully in past in isolated domains
  - Fields et al. → out-of-order CPU
  - Barford and Crovella → TCP
  - Yang and Miller → MPI
Building a Global Critical Path

- Requires global event dependence graph

**Challenge:**
typically requires detailed knowledge across many domains!

**Solution:**
automatically extract dependence graph from interacting state machines
Our simulation technique directly identifies:
- The current bottleneck
- How much improvement until next bottleneck
- What the next bottleneck will be

Conventional simulation approach:
- Hypothesize bottleneck
- Prototype solution
- Simulate solution
- Test hypothesis, repeat if incorrect
Systematically map state machines into a global dependence graph
- Most HW is already specified as a state machines
- Extract implicit state machines from SW
Explicit State Machine Conversion

State Machine | Dependence Graph

Edges ➔ Nodes

Nodes ➔ Edges

dependence edge weight = time spent in state
Explicit State Machine Conversion

Start \[ \xrightarrow{t_1-t_0} \] A\(\rightarrow\)B \[ \xrightarrow{t_2-t_1} \] B\(\rightarrow\)D \[ \xrightarrow{t_3-t_2} \] D\(\rightarrow\)B \[ \xrightarrow{t_4-t_3} \] B\(\rightarrow\)C
What about software?

F() {
    int a;
    ...
    H();
    ...
    ...
    S();
}

H() {
    int z;
    ...
    ret L();
}

L() {
    int z;
    ...
    ...
    ...
    ...
    ...
}

Start -> H
F -> H
H -> L
L -> F
F -> S

\[ t_1 - t_0 \]
\[ t_2 - t_1 \]
\[ t_3 - t_2 \]
\[ t_4 - t_3 \]
State Machine Interactions

- Link up piece of dependence graph through these interactions
- Queues are interaction points
  - Without them back pressure can’t be modeled
  - Abstract entities
- Annotated in models and code
  - Developed iteratively
  - Analysis can pinpoint problems
Interaction Example

Simplified IP stack state machine

- Get Packet
- Enqueue in TXQ
- Find Dest
- Append hdrs

Simplified TX NIC driver state machine

- Process Packet
- Enqueue in TX ring
- Wait on TXQ
- Dequeue from TXQ
Interaction Example

Simplified IP stack state machine

1. Get Packet
2. Append hdrs
3. Find Dest
4. Enqueue in TXQ

Simplified TX NIC driver state machine

1. Process Packet
2. Enqueue in TX ring
3. Wait on TXQ
4. Dequeue from TXQ

Flowchart:
- Get Pkt→ App hdrs
- App hdrs→ Find dest
- Find dest→ Enqueue
- Enqueue→ Get pkt
- Get pkt→ App hdrs
- App hdrs→ Find dest

Time:
- EnQ→Wait TXQ
- DeQ→ Process Pkt
- Process Pkt→EnQ

Current Time: 28
Finding Global Critical Path

- Use standard graph analysis techniques
- Locate longest path through the graph
Critical States & Predicting Speedup

- Aggregate states on critical path
  - Most frequent state is the bottleneck
- Dependence graph contains all transitions and interactions
  - Not just the ones that compose critical path or where waiting occurred
- Modify weights on the critical path
  - Re-analyze data to see how critical path changes
  - Next critical path length → potential speedup
Critical path can sometimes be improved without reducing latency of any tasks.

In resource constrained environments critical path can be shorted by providing more resources.
Resource Dependence Loops

- Analysis automatically find candidates
- Addition of buffering changes critical path
Workloads

- Linux 2.6.18
- SinkGen – Streaming benchmark from CERN
  - Analyzed the transmit side
- Lighttpd – High-performance web server
  - Uses non-blocking I/O to manage connections
  - Used by large websites
- Metric is bandwidth achieved
TCP Transmit

- Start with default M5 system parameters
  1. Capture bottleneck data from that system
  2. Locate current bottleneck
  3. Predict performance when bottleneck is removed
  4. Repeat steps 2 and 3 for successive bottlenecks
  5. Verify that the locations and predictions are correct
TCP Streaming Benchmark

How did we do?

Run experiments making the above suggested changes
TCP Streaming Benchmark

- Predict performance again, this time starting with configuration 2

- Config 3: 1%
- Config 4: 15%
TCP Streaming Benchmark

Predict performance one last time, starting with configuration 3

3% error
Experiments and Errors

- Additional experiments are in the paper
  - Multi-core speed up of web server
- Describe why errors occurred
  - Compare modified dependence graph to observed graph from new simulation
Conclusion

- Architects are increasing looking at system-level issues for performance

- Apply critical path analysis to complete systems composed of concurrent components
  - Span multiple layers of HW & SW
  - Automate extraction of dependence graphs

- Identify end-to-end bottlenecks in network systems
  - Critical tasks
  - Resource dependence loops
  - Performance of hypothetical systems
  - Minutes not hours
Questions?