Improving Scalability of Processor Utilization on Heavily-Loaded Servers with Real-Time Scheduling

Eiji Kawai, Youki Kadobayashi, Suguru Yamaguchi
Nara Institute of Science and Technology
JAPAN
Motivation

- Network processing essentially incurs high overhead
  - Event-driven processing
- High processor utilization on servers
  - degrades service quality (large latency)
  - increases power consumption
  - worsens MTBF (mean time between failures)
- Serious issue especially in large service providers
  - Data centers
  - ISPs / ASPs
- Make processor utilization proportional to request rate (service throughput)
Traditional network server architecture

- Multi-process / Multi-thread architecture
  - Simple programming semantics
  - like inetd daemon
  - Poor scalability
  - context switches, process / thread management

- Polling I/O based architecture
  - select() / poll()
  - Better performance
  - Complex programming semantics
  - High processor utilization
Polling I/O

- Basic functions
  - Receives a socket set as an argument
  - Watches events on the socket set
    - Waits until one of the sockets becomes I/O ready

- Typical process flow
  1. Create a socket list
  2. Invoke polling I/O
  3. An event occurs (some sockets become I/O ready)
  4. Process the I/O ready sockets and return to Step 1.
Why polling I/O based architecture wastes processor cycles?

- Polling I/O scans a large socket list
  - A busy network server often processes many concurrent sockets
    - Large processing cost
  - Many researches attack this problem
    - Replace polling I/O with a special event notification mechanism
    - Difficult to use in actual operation (not a defact standard solution)

- No interruption in process flow cycle
  - Consider the situation that a web server handles 4,000 concurrent sockets and receives 1,000 requests per second
    - To make an interruption (blocking) in the polling I/O, the process flow cycle should be under 1ms
Our approach: Fine-Grained Interval Control with Self-Blocking

- Put a short self-blocking in the process flow cycle
  1. Create a socket list
  2. Check the interval and block itself
  3. Invoke polling I/O
  4. Process I/O ready sockets, and return to Step 1.

- Use RT-scheduling for accurate execution control
  - No fine-grained resource accounting system on Unix
  - Time-stamping to measure processing time does not work well with context switches
  - Except for Step 2, put the server process under RT scheduling to avoid context switching
Calculation of self-blocking time

- Two approaches on calculating self-blocking time (B)

  - Static process flow cycle
    - $B = C - P$ ($C$: process flow cycle, $P$: processing time)
    - Constant service delay
    - Linear processor utilization

  - Dynamic process flow cycle
    - $B = kP$ ($k$: configurable constant, $P$: processing time)
    - Linear service delay
    - Constant processor utilization on the polling I/O
Performance evaluation

- Benchmark tests
  - Implements the interval control mechanism in a polling I/O based web accelerator (reverse proxy server)
  - Evaluate the performance with Web Polygraph (http://www.web-polygraph.org/)
  - One of industry standard benchmark tools
  - Emulate realistic load
    - Content model, user access model, network delay, service delay on origin servers, etc...
  - Kernel profiling with kernprof (from SGI)
Performance evaluation (cont’d)

![Diagram of network architecture with nodes labeled as clients and servers, showing 40ms delay with 0.05% loss and 300ms delay.]

- Target accelerator relays the requests from clients to servers with contents cache
  - Cache hit ratio is about 70%
Static process flow cycle
CPU utilization

- Linear processor utilization against request rate
- Good scalability
- Process flow cycle of 20ms is the best configuration
Static process flow cycle
Response time

- Mean Response Time
- Traditional architecture
- poll() interval = 10 ms
- poll() interval = 20 ms
- poll() interval = 30 ms
- poll() interval = 40 ms

- Linear response time against request rate
- Good Scalability
- The process flow cycle should be small enough
  - 10ms or 20ms
Dynamic process flow cycle

CPU Utilization

Linear processor utilization against request rate
- Slightly higher utilization compared with static cycle approach
Dynamic process flow cycle
Response Time

- Linear response time against request rate
- Better performance than static cycle approach
Summary

- High processor utilization is a serious issue especially in large scale service providers.

- Fine-grained interval control with RT-scheduling linearize processor utilization against service throughput:
  - The self-blocking time should be small enough.