Managing Heavy Network Load: Eliminating Receive Livelock

Brad Karp
UCL Computer Science

CS 0133
1st November 2019
Engineering for Performance

• Much of the work in distributed systems concerns designing for
  – Consistency
  – Availability
  – Performance

• Performance is **multi-faceted**
  – Not just determined by design of distributed system itself (algorithms, protocols)
  – Low-level hardware, OS behavior play major role

• Achieving high performance requires deep understanding of **all layers:** hardware, OS, all the way through algorithms and protocols!
Engineering for Performance

**Systems Thinking**: the ability to reason about complex interactions among many layers, to find problems (and (re)design to avoid them)

- Availability
- Performance

- **Performance is multi-faceted**
  - Not just determined by design of distributed system itself (algorithms, protocols)
  - Low-level hardware, OS behavior play major role

- **Achieving high performance requires deep understanding of all layers**: hardware, OS, all the way through algorithms and protocols!
Heavy Load Happens

• Servers have limited CPU, network link capacity, memory, disk bandwidth

• Demand often approaches or exceeds a server’s capacity, e.g.,
  – Flash crowds at web server
  – Busy NFS server as client population grows
  – IP router or firewall carrying flash crowd traffic (or denial of service attack traffic!)

• But **software design** can limit performance at loads lighter than where these hardware limits kick in...
Example:
IP Packet Forwarding Performance

• Hardware: commodity workstation (DECstation 3000/300; PC-like), two 10 Mbps Ethernet interfaces
• Software: Digital UNIX 3.2 OS, screend firewall application in userspace
• Workload: forward IP packets from one Ethernet to another (UDP packets, 4 bytes of payload each)
• Packet-generating host has faster CPU than forwarder
Example: IP Packet Forwarding Performance

**Question:** How well does whole system scale as load increases?

**Experiment:** vary input packet rate to forwarder; observe output packet rate

firewall application in userspace

- Workload: forward IP packets from one Ethernet to another (UDP packets, 4 bytes of payload each)
- Packet-generating host has faster CPU than forwarder
Example: 
IP Packet Forwarding Performance

- Peak output rate w/o firewall: ~4700 pkt/s
- Beyond ~4700 pkt/s, output rate decreases with further increasing load!
Example: IP Packet Forwarding Performance

Suppose hardware’s capacity is 4700 pkt/s. What would ideal system behavior be beyond that input rate?
Example: IP Packet Forwarding Performance

Suppose **hardware’s capacity** is 4700 pkt/s. What would ideal system behavior be beyond that input rate?

---

From these tests, it was clear that with **screend** transmitted on the output interface. When the system is running, the router suffered from poor overload behavior at rates above 2000 packets/sec., and complete livelock set in at about 6000 packets/sec. Even with **screend**, the router peaked at 4700 packets/sec., and would probably livelock somewhat below the maximum Ethernet packet rate of about 14,880 packets/second. We solved the livelock problem by doing as much work as possible in a kernel thread, rather than in the interrupt handler, and by eliminating the IP input queue and its associated queue manipulations and software interrupt (or thread dispatch) once we decide to take a packet from the receiving interface, runs at interrupt priority level (IPL) = SPLIMP, and we try not to discard it later on, since this would represent wasted effort. **SPLNET**, which is lower than SPLIMP. The queue we also try to carefully ''schedule'' the work between the driver and the IP code is named in this thread. It is probably not possible to use ''ipintrq,'' and each output interface is buffered by a queue of its own. All queues have length limits; each packet, so we instead had this thread use a polling technique to efficiently simulate round-robin scheduling of packet processing. The polling thread (if it has not already been scheduled), recording its need for the rate at which the device driver can pull new packets into its input queue, the IP code never runs. Thus, it never sets its flag, so the system will not be distracted with additional interrupts until the polling thread has processed all of the pending packets.

The system's CPU resources are saturated by the time the modified interface drivers cause it discards each packet after a lot of CPU time has been invested in it at elevated IPL. This is foolish; once a packet has made its way through the device driver, it represents an investment and should be processed to completion if at all possible. In a router, this means that the packet should be transmitted out of the interface and add them to the IP input interrupt. It does not set the device’s interrupt-enable register themselves with the polling system, causing it discards each packet after a lot of CPU time register themselves with the polling system, providing this thread use a poll-

In 4.2BSD follows the model described in section 4.1, and depicted in figure 6-2. The device driver decide to take a packet from the receiving interface, runs at interrupt priority level (IPL) = 0, instead of as a software interrupt handler. Instead, it is now quite obvious why the system suffers from overload problems. Once the input rate exceeds 6000 packets/sec., the device driver does almost no work at all. Instead, the system implement interrupt batching, so at high input rates very few interrupts are actually taken. The system implement additional heuristics to help meet our performance goals.

**6.3. Why livelock occurs in the 4.2BSD model**

In the new system, the interrupt handler for an interface driver does almost no work at all. Instead, the system simple schedules the polling thread (if it has not already been scheduled), recording its need for the rate at which the device driver can pull new packets into its input queue, the IP code never runs. Thus, it never sets its flag, so the system will not be distracted with additional interrupts until the polling thread has processed all of the pending packets.

The system's CPU resources are saturated by the time the modified interface drivers cause it discards each packet after a lot of CPU time has been invested in it at elevated IPL. This is foolish; once a packet has made its way through the device driver, it represents an investment and should be processed to completion if at all possible. In a router, this means that the packet should be transmitted out of the interface and add them to the IP input interrupt. It does not set the device’s interrupt-enable register themselves with the polling system, providing this thread use a poll-

Suppose **hardware’s capacity** is 4700 pkt/s. What would ideal system behavior be beyond that input rate?
Background: I/O Device Hardware

- I/O devices need to notify CPU of events
  - Packet arrival at network interface
  - Disk read complete
  - Key pressed on keyboard

- Two main ways CPU can learn of events:
  - **Polling**: CPU “asks” hardware device if any events have occurred (synchronous)
  - **Interrupt**: hardware device sends a signal to CPU saying “events have completed” (asynchronous)

- Key concerns: **event latency** and **CPU load**
Polling

- Requires **programmed or memory-mapped I/O** (relatively slow; over I/O bus)
- CPU “blindly” polls device **explicitly in code**
  - to guarantee low latency, must **poll very often**
  - high CPU overhead to poll very often
- For rare I/O events, CPU overhead of polling unattractive
- Disk I/Os complete only 100s of times per second; in 1980s-90s, only hundreds of network packets arrived per second
- OSes in that era **eschewed polling**
Interrupts

- I/O devices have dedicated wire(s) that they can use to signal interrupt(s) to CPU
- On interrupt, if interrupt priority level (IPL) > CPU priority level:
  - CPU saves state of currently running program
  - jumps to interrupt service routine (ISR) in kernel
  - invokes device driver, which asks device for events
  - returns to previously running program
- CPU priority level: kernel-set machine state specifying which interrupts allowed (others postponed by CPU)
- On modern x86_64, interrupt latency of ~3 us from device interrupt to start of ISR
Interrupts

Interrupts well-suited to **rare I/O events**: lower latency than rarely polling, lower CPU cost than constantly polling

Interrupts asynchronous—they **preempt other system activity**

- invokes device driver, which asks device for events
- returns to previously running program

- **CPU priority level**: kernel-set machine state specifying which interrupts allowed (others postponed by CPU)

- On modern x86_64, interrupt latency of ~3 us from device interrupt to start of ISR
Interrupts and Network I/O

- Disk I/O requests come from OS itself; completion interrupts inherently rate-controlled
- Network packets come from other hosts; no “local” rate control for received packet interrupts
- Remember: interrupts take priority over all other system processing (over other kernel execution, user-space applications)
- What will happen when received packet rate extremely high?
  - Answer depends on detailed software structure...
Interrupts and Network I/O

Receive Livelock:
When event rate (pkt arrival rate) so high, system spends all its time servicing interrupts, gets no other work done!

- Remember: interrupts take priority over all other system processing (over other kernel execution, user-space applications)
- What will happen when received packet rate extremely high?
  - Answer depends on detailed software structure...
Design Goals for Network I/O System

• Goals:
  – Low latency for responding to I/O events
  – Low jitter (variance in latency)
  – Fairness: resources allocated evenly among tasks
  – High throughput for I/O (e.g., achievable packet receive rate, transmit rate)

• What are the tasks for a network server?
  – Packet reception
  – Packet transmission
  – Protocol processing (often in kernel)
  – Other I/O processing
  – Application processing
Background: OS Architecture for Interrupt-Driven Networking

- Packet arrives
- Network card interrupts at “high” IPL
- ISR looks at Ethernet header, enqueues packet for further processing, returns
- “Low” IPL software interrupt dequeues packets from queue, does IP/UDP/TCP processing, enqueues data for dst process
- Process reads data with read() system call
- Queues denote scheduling and priority level boundaries
Background: OS Architecture for Interrupt-Driven Networking

Queues are **scheduling** and **priority level** boundaries

Minimizing work in ISR **reduces service latency** for other device I/O interrupts

- “Low” IPL software interrupt dequeues packets from queue, does IP/UDP/TCP processing, enqueues data for dst process
- Process reads data with read() system call
- Queues denote **scheduling and priority level boundaries**
Interrupt-Driven Networking, UNIX Style

receive ISR

input queue

IP forwarding/reception software interrupt

socket buffer

application (e.g., firewall)

transmit complete ISR

output queue

socket buffer

kernel user

increasing priority level
Interrupt-Driven Networking, UNIX Style

Design prioritizes **packet reception** above all else
Original motivation: small buffers on network interfaces (no longer a concern)
Interrupt-Driven Networking, UNIX Style

How will this system behave as packet receive rate increases—what will output packet rate do?
Receive Livelock Pathologies

- As input rate increases beyond maximum loss-free receive rate, output rate decreases
- System wastes CPU preparing arriving packets for queue, all of which dropped
- For input burst of packets, first packet not delivered to user level until whole burst put on queue (e.g., leaves NFS server disk idle!)
- In systems where transmit lower-priority than receive, transmit starves
Livelock Avoidance Technique 1: Minimize Receive Interrupts

• Goal: limit the receive interrupt rate
• Receive ISR:
  – sets flag indicating this network interface has received one or more packets
  – schedules kernel thread that polls network interfaces for received packets
  – does not re-enable receive interrupts
• That’s it! Set flag, schedule kernel thread, and return, leaving receive interrupts disabled.
Liveloop Avoidance Technique 2: Kernel Polling Thread

• When scheduled, checks all network interfaces’ “packets received” flags
• For such interfaces:
  – process packet all the way through kernel protocol stack (IP/forwarding/UDP/TCP), ending with interface output queue or socket buffer to application
  – maximum quota on packets processed for same interface on one invocation for fairness
  – round-robins among interfaces and between transmit and receive
  – Re-enable interface’s receive interrupts only when no pending packets at that interface
Livelock Avoidance Technique 2: Kernel Polling Thread

Under overload, where do packets go?
Dropped by network interface card when buffering exhausted (either in card, or in host RAM), at no CPU cost!

- When scheduled, checks all network interfaces' "packets received" flags
- For such interfaces:
  - process packet all the way through kernel protocol stack (IP/forwarding/UDP/TCP), ending with interface output queue or socket buffer to application
  - maximum quota on packets processed for same interface on one invocation for fairness
  - round-robin among interfaces and between transmit and receive
  - Re-enable interface’s receive interrupts only when no pending packets at that interface
**Performance Evaluation: Techniques 1 and 2**

- No screend firewall
- Without *quotas for input processing*, big trouble! (Why?)
What about screend?

- User-level application still cannot run under heavy receive load!
- Technique 3: disable receive interrupts when queue to user application fills
Receive Livellok: Summary

• Scheduling vital to performance of a busy server
  – may be implicit (e.g., interrupts), not explicit (e.g., OS scheduler)
• Understanding cross-layer behavior vital to finding performance limitations and designing for high performance
• General lessons:
  – Don’t discard data after doing work on it
  – Poll while busy, interrupt while lightly loaded