IsoStack – Highly Efficient Network Processing on Dedicated Cores

Leah Shalev
Eran Borovik, Julian Satran, Muli Ben-Yehuda
Outline

- Motivation
- IsoStack architecture
- Prototype – TCP/IP over 10GE on a single core
- Performance results
- Summary
TCP/IP End System Performance Challenge

- TCP/IP stack is a major “consumer” of CPU cycles
  - “easy” benchmark workloads can consume 80% CPU
  - “difficult” workloads cause throughput degradation at 100% CPU

- TCP/IP stack wastes CPU cycles:
  - 100s of "useful" instructions per packet
  - 10,000s of CPU cycles
Long History of TCP/IP Optimizations

- Decrease per-byte overhead
  - Checksum calculation offload
- Decrease the number of interrupts
  - Interrupt mitigation (coalescing)
- Decrease the number of packets (for bulk transfers)
  - Jumbo frames
  - Large Send Offload (TCP Segmentation Offload)
  - Large Receive Offload
History of TCP Optimizations cont` – Full Offload

- Instead of optimizations - offload to hardware
  - TOE (TCP Offload Engine)
    - Expensive
    - Not flexible
    - Not robust - dependency on device vendor
    - Not supported by some operating systems on principle
  - RDMA
    - Requires support on the remote side
    - not applicable to legacy upper layer protocols

- TCP onload – offload to a dedicated main processor
  - Using a multiprocessor system asymmetrically
TCP/IP Parallelization

- Naïve initial transition to multiprocessor systems
  - Using one lock to protect it all

- Incremental attempts to improve parallelism
  - Use more locks to decrease contention
  - Use kernel threads to perform processing in parallel
  - Hardware support to parallelize incoming packet processing – Receive-Side Scaling (RSS)
Parallelizing TCP/IP Stack Using RSS

Theory (customized system)

- CPU 1:
  - Conn A
  - Conn B
  - Conn C
  - Conn D
  - TCP/IP

- CPU 2:
  - Conn A
  - Conn B
  - Conn C
  - Conn D
  - TCP/IP

- NIC

Practice

- CPU 1:
  - Conn A
  - Conn B
  - Conn C
  - Conn D
  - Rx 2
  - Tx

- CPU 2:
  - Conn A
  - Conn B
  - Conn C
  - Conn D
  - Rx 2
  - Tx

- NIC
So, Where Do the Cycles Go?

- No clear hot-spots
  - Except lock/unlock functions

- CPU is “misused” by the network stack:
  - Interrupts, context switches, cache pollution
    - due to CPU sharing between applications and stack
  - IPIs, locking and cache line bouncing
    - due to stack control state shared by different CPUs
Our Approach – Isolate the Stack

- Dedicate CPUs for network stack
- Use light-weight internal interconnect
  - Scaling for many applications and high request rates
- Make it transparent to applications
  - Not just API-compatible – hide the latency of interaction
IsoStack Architecture

- **Split socket layer:**
  - front-end in application
    - Maintains socket buffers
    - posts socket commands onto command queue
  - back-end in IsoStack
    - On dedicated core[s]
      - With connection affinity
    - Polls for commands
    - Executes the socket operations asynchronously
    - “Zero-copy”

- **Shared-memory queues for socket delegation**
  - Asynchronous messaging
  - Flow control and aggregation
  - Data copy by socket front-end
IsoStack Shared Memory Command Queues

- Low overhead multiple-producers-single-consumer mechanism
  - Non-trusted producers
- Design Principles:
  - Lock-free, cache-aware
  - Bypass kernel whenever possible
    - Problematic with the existing hardware support
  - Interrupt mitigation
- Design Choices Extremes:
  - A single command queue
    - Con - high contention on access
  - Per-thread command queue
    - Con - high number of queues to be polled by the server
- Our choice:
  - Per-socket command queues
    - Aggregation of tx and rx data
  - Per-CPU notification queues
    - Requires kernel involvement to protect access to these queues
IsoStack Prototype Implementation

- Power6 (4x2 cores), AIX 6.1
- 10Gb/s HEA
- Same codebase for IsoStack and legacy stack
- IsoStack runs as single kernel thread “dispatcher”
  - Polls adapter rx queue
  - Polls socket back-end queues
  - Invokes regular TCP/IP processing

- Network stack is [partially] optimized for serialized execution
  - Some locks eliminated
  - Some control data structures replicated to avoid sharing

- Other OS services are avoided when possible
  - E.g., avoid wakeup calls
  - Just to workaround HW and OS support limitations
TX Performance

![Graph showing TX performance with message size on the x-axis and CPU utilization and throughput on the y-axis. The graph compares Native CPU and IsoStack CPU as well as Native Throughput and IsoStack Throughput.]
Rx Performance

The diagram illustrates the CPU utilization and throughput for different message sizes. The x-axis represents the message size, and the y-axis shows the CPU utilization and throughput in MB/s. The graph compares Native CPU and IsoStack CPU performance. The Native Throughput is represented by a solid line, and the IsoStack Throughput is represented by a dashed line. The diagram shows that IsoStack provides better throughput compared to Native CPU for most message sizes.
Impact of Un-contended Locks

- Impact of unnecessary lock re-enabled in IsoStack:
  - For low number of connections:
    - Throughput decreased
    - Same or higher CPU utilization
  - For higher number of connections:
    - Same throughput
    - Higher CPU utilization

- Even when un-contended, locks have tangible cost!

Transmit performance for 64 byte messages

- Throughput decreased
- Same or higher CPU utilization
- Same throughput
- Higher CPU utilization

Even when un-contended, locks have tangible cost!
IsoStack – Summary

- Isolation of network stack dramatically reduces overhead
  - No CPU sharing costs
  - Decreased memory sharing costs

- Explicit asynchronous messaging instead of blind sharing
  - Optimized for large number of applications
  - Optimized for high request rate (short messages)

- Opportunity for further improvement with hardware and OS extensions
  - Generic support for subsystem isolation
Questions?
Backup
Using Multiple IsoStack Instances

- Utilize adapter packet classification capabilities
- Connections are “assigned” to IsoStack instances according to the adapter classification function
- Applications can request connection establishment from any stack instance, but once the connection is established, socket back-end notifies socket front-end which instance will handle this connection.
Potential for Platform Improvements

- The hardware and the operating systems should provide a better infrastructure for subsystem isolation:
  - Efficient interaction between large number of applications and an isolated subsystem
    - In particular, better notification mechanisms, both to and from the isolated subsystem
  - Non-shared memory pools
  - Energy-efficient wait on multiple memory locations