IsoStack – Highly Efficient Network Processing on Dedicated Cores

Leah Shalev

Eran Borovik, Julian Satran, Muli Ben-Yehuda

Haifa Research Lab

28 May 2010

Accepted to USENIX ATC 2010

Outline

- TCP Performance Challenge
- IsoStack Architecture
- Prototype Implementation for TCP over 10GE on a single core
- Performance Results
- Summary

TCP Performance Challenge

- Servers handle more and more network traffic, most of it is TCP
- Network speed grows faster than CPU and memory speed
- On a typical server, TCP data transfer at line speed can consume 80% CPU
  - In many cases, line speed cannot be reached even at 100% CPU
- TCP overhead can limit the overall system performance
  - E.g., for cache-hit access to storage over IP
- TCP stack wastes CPU cycles:
  - 100s of "useful" instructions per packet
  - 10,000s of CPU cycles

Long History of TCP Optimizations

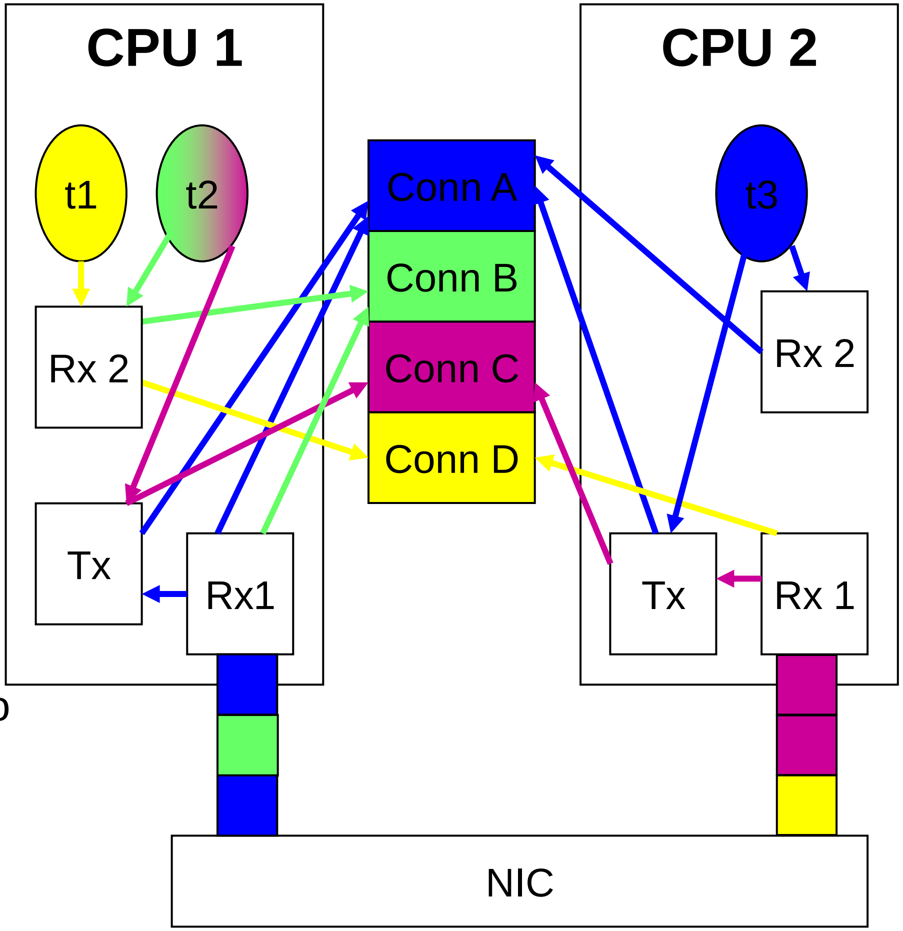
- Decrease per-byte overhead
  - Checksum calculation offload
- Decrease the number of interrupts
  - interrupt mitigation (coalescing) – increases latency
- Decrease the number of packets (to decrease total per-packet overhead, for bulk transfers)
  - Jumbo frames
  - Large Send Offload (TCP Segmentation Offload)
  - Large Receive Offload
- Improve parallelism
  - Use more locks to decrease contention
  - Receive-Side Scaling (RSS) to parallelize incoming packet processing
- Offload the whole thing to hardware
  - TOE (TCP Offload Engine) – expensive, not flexible, not supported by some OSes
  - RDMA – not applicable to legacy protocols
- TCP onload – offload to a dedicated main processor

Why so Much Overhead Today?

- Because of legacy uniprocessor-oriented design
- CPU is “misused” by the network stack:
  - Interrupts, context switches and 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
- Where do the cycles go?
  - CPU pipeline flushes
  - CPU stalls

Isn't Receive Affinity Sufficient?

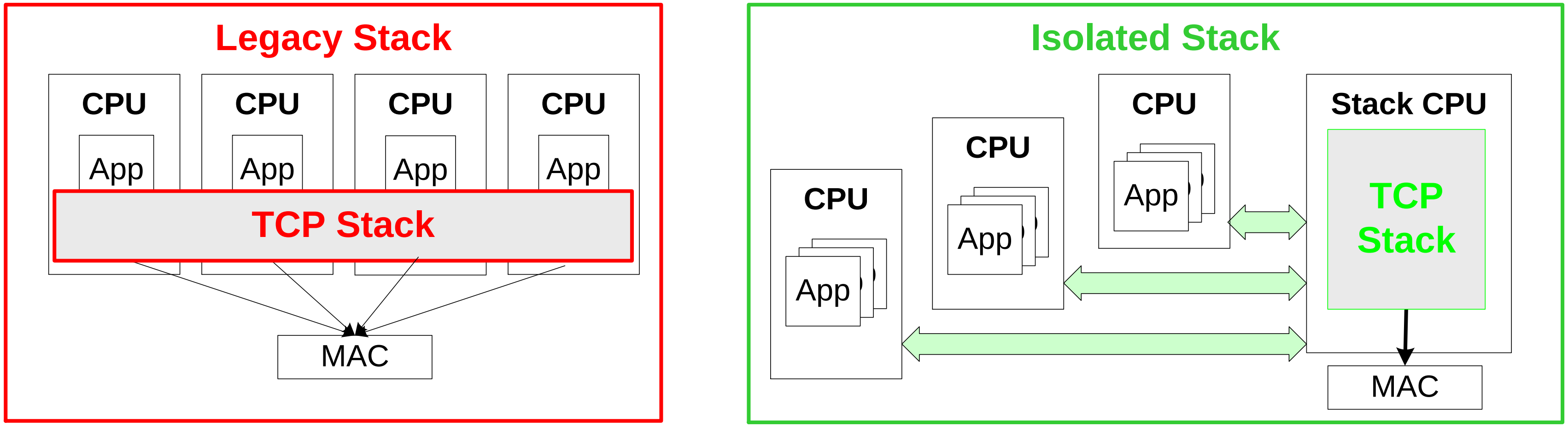
- Packet classification by adapter
  - Multiple RX queues for subsets of TCP connections
- RX packet handling (RX1) affinitized to a CPU
- Great when the application runs where its rx packets are handled
  - Especially useful for embedded systems
- BUT#1: on a general-purpose system, the socket applications may well run on a "wrong" CPU
  - Application cannot decide where to run
    - Since Rx1 affinity is transparent to the application
  - Moreover, OS cannot decide where to run a thread to co-locate it with everything it needs
    - Since an application thread can handle multiple connections and access other resources
- BUT#2: even when co-located, need to "pay" for interrupts and locks



- t1: recv on connD
- t2: recv on connB, send on connC
- t3: send and recv on conn A

Our Approach – IsoStack

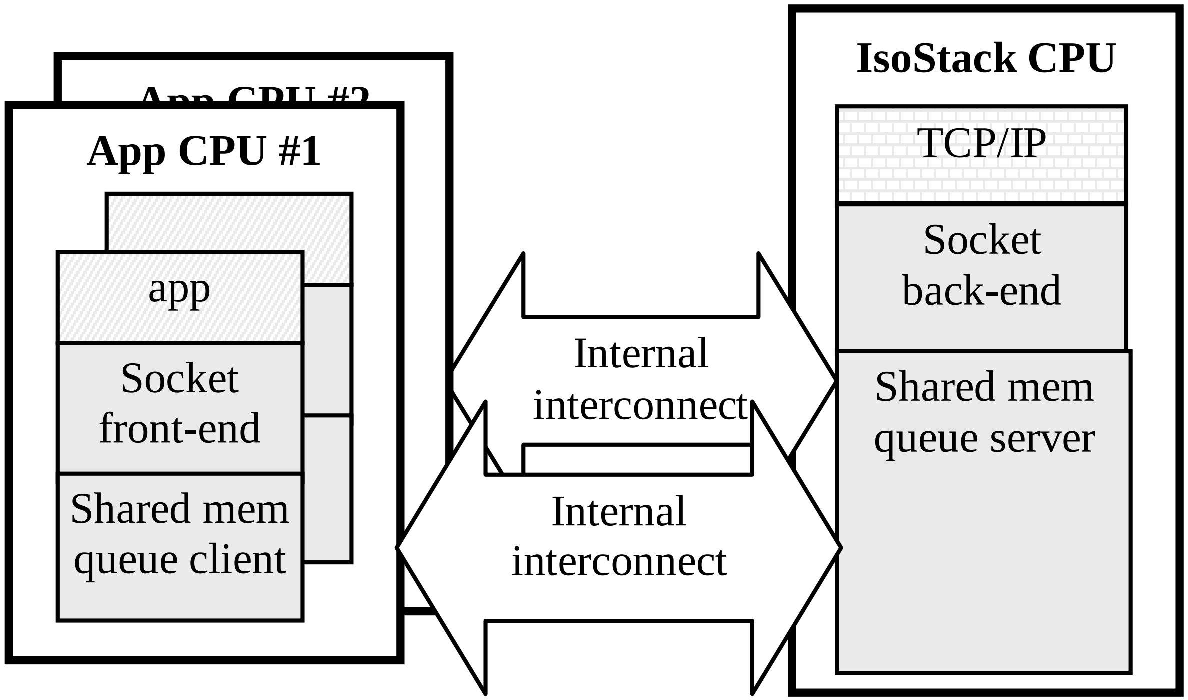
- Isolate the stack
  - Dedicated CPUs for network stack
    - Avoid sharing of control state at all stack layers
  - Application CPUs are left to applications
  - Light-weight internal interconnect



IsoStack Architecture

- Socket front-end replaces socket library
- Socket front-end "delegates" socket operations to socket back-end
  - Flow control and aggregation
- Socket back-end is integrated with single-threaded stack
  - Multiple instances can be used
- Internal interconnect using shared memory queues
  - Asynchronous messaging
  - Similar to TOE interface
- Data copy by socket front-end

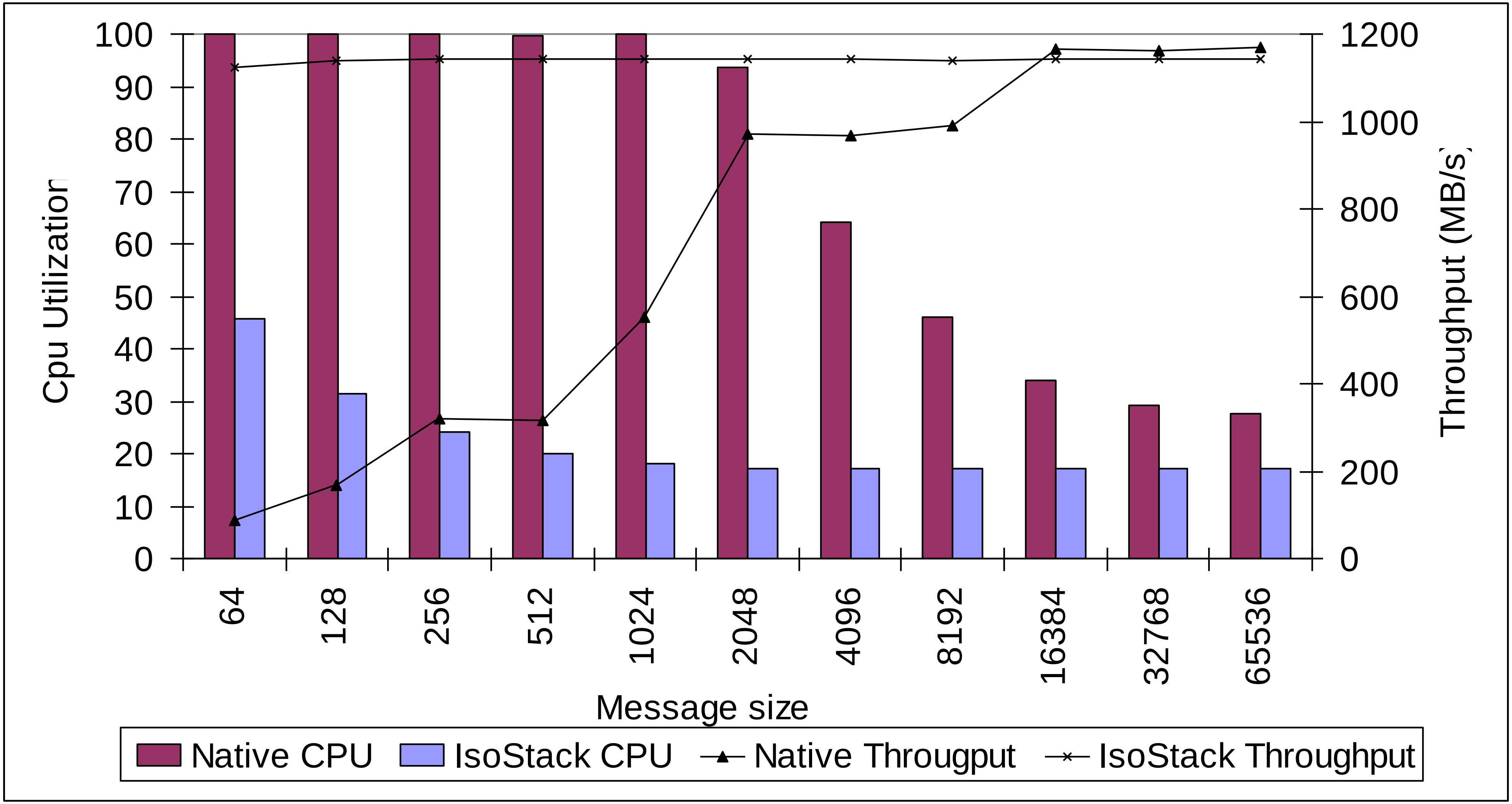
- Socket layer is split:
  - Socket front-end in application
  - Socket back-end in IsoStack



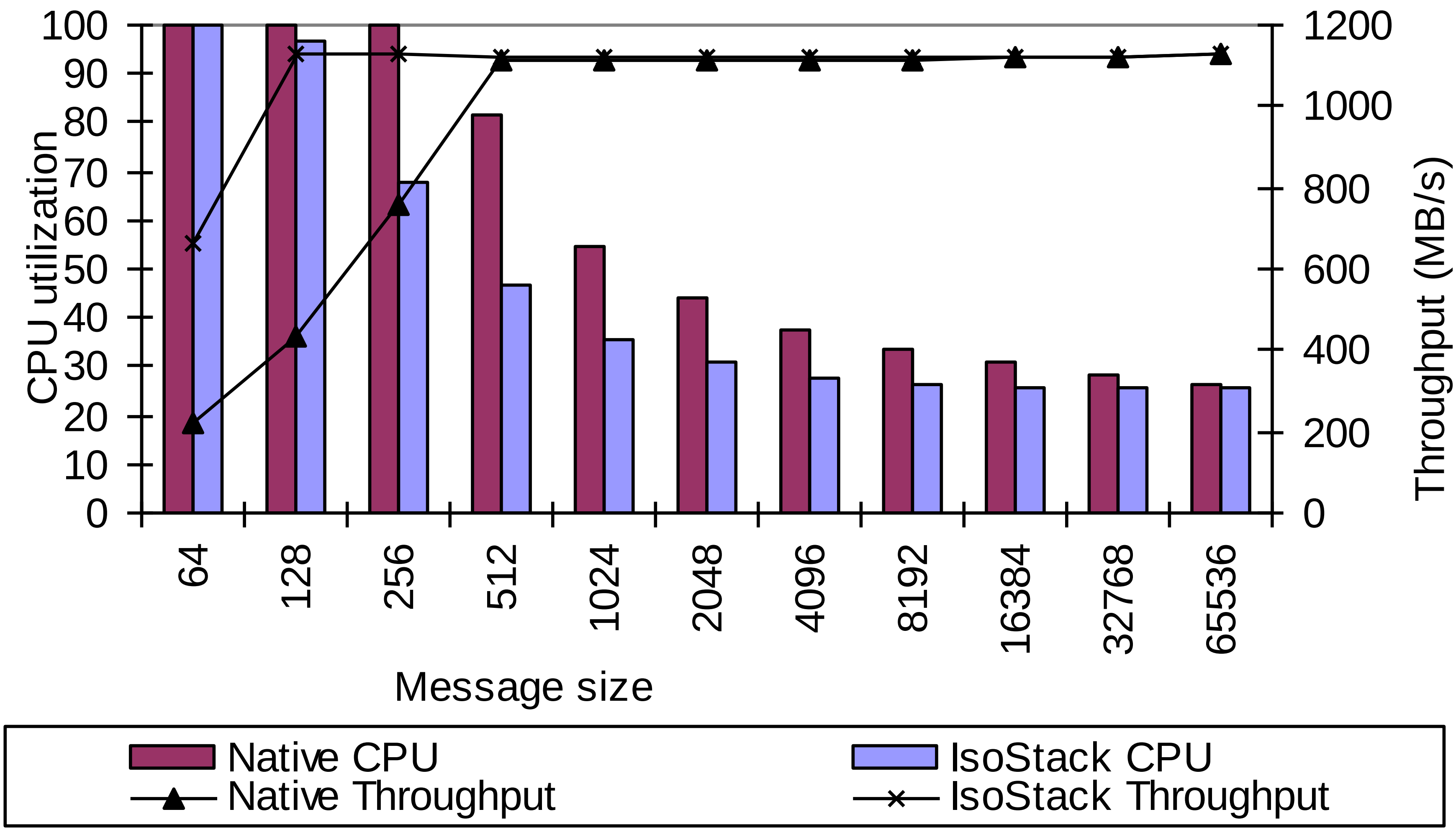
Prototype Implementation

- Power6 (4x2 cores), AIX 6.1
- 10Gb/s HEA
- IsoStack runs as single kernel thread "dispatcher"
  - Polls adapter rx queue
  - Polls socket back-end queues
  - Polls internal events queue
  - 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



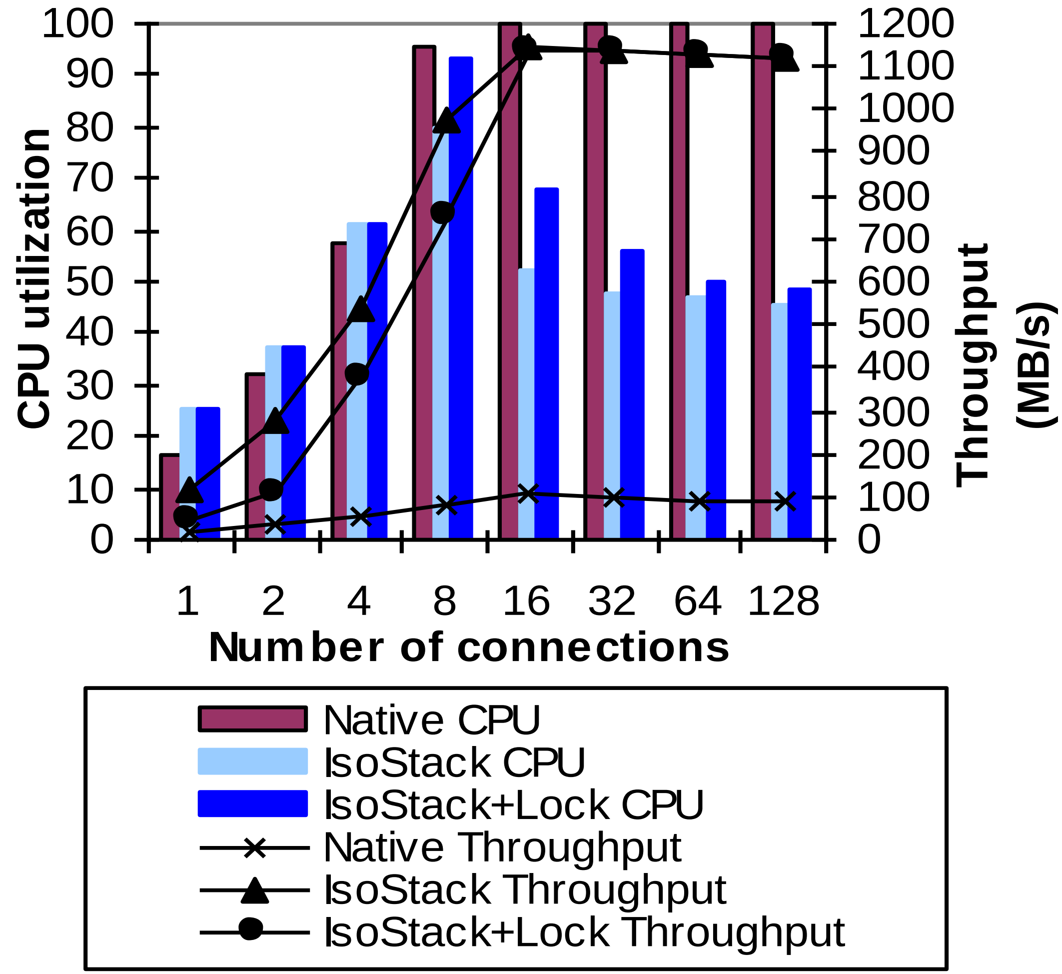
Rx Performance



Impact of Un-contended Locks

- Impact of un-necessary 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

Transmit performance for 64 byte messages



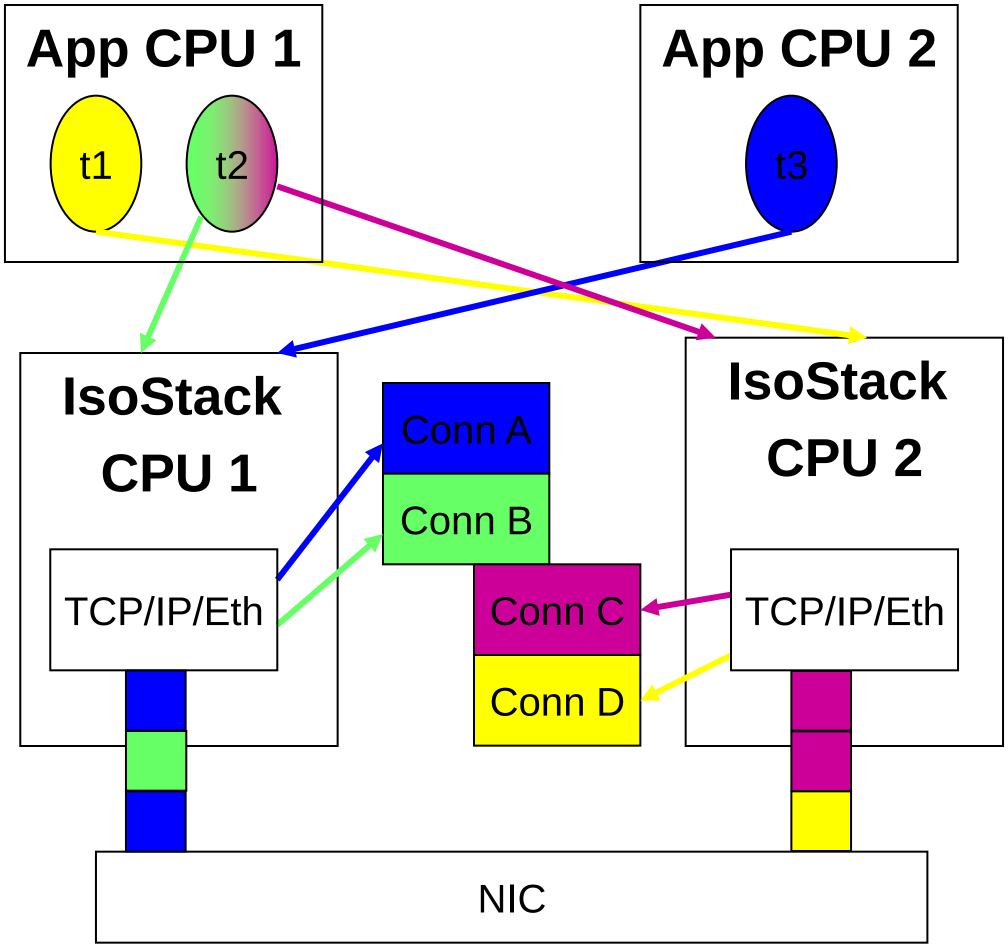
Isolated Stack – Summary

- Concurrent execution of network stack and applications on separate cores
- Connection affinity to a core
- Explicit asynchronous messaging between CPUs
  - Simplifies aggregation (command batching)
  - Allows better utilization of hardware support for bulk transfer
- Tremendous performance improvement for short messages
  - and nice improvement for long messages
- Un-contended locks are not free
  - IsoStack can perform even better if the remaining locks will be eliminated

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.



Internal Interconnect Using Shared Memory

- Requirement – low overhead multiple-producers-single-consumer mechanism
    - Non-trusted producers
- Design Principles:
  - Lock-free, cache-aware queues
  - Bypass kernel whenever possible
    - problematic with the existing hardware support
- 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
    - With flow control
    - Aggregation of tx and rx data
  - Per-logical-CPU notification queues
    - Requires kernel involvement to protect access to these queues

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

Performance Evaluation Methodology

- Setup:
  - POWER6, 4-way, 8 cores with SMT (16 logical processors), 3.5 GHz, single AIX LPAR
  - 2-port 10Gb/s Ethernet adapter,
    - one port is used by unmodified applications (daemons, shell, etc)
    - another port is used by the polling-mode TCP server. The port is connected directly to a “remote” machine.
- Test application
  - A simple throughput benchmark – several instances of the test are sending messages of a given size to a remote application which promptly receives data.
  - “native” is compiled with regular socket library, and uses the stack in “legacy” mode.
  - “modified” is using the modified socket library, and using the stack through the polling-mode IsoStack.
- Tools:
  - nmon tool is used to evaluate throughput and CPU utilization

Network Scalability Problem

- TCP processing load increases over years, despite incremental improvements
- Adjusting network stacks to keep pace with the increased link bandwidth is difficult
  - Network scales faster than CPU
  - Deeper pipelines increase the cost of context switches / interrupts / etc
  - Memory wall:
    - Network stack is highly sensitive to memory performance
    - CPU speed grows faster than memory bandwidth
    - Memory access time in clocks increases over time (increasing bus latency, very slow improvement of absolute memory latency)
  - Naïve parallelization aproaches on SMPs make the problem worse (locks, cache ping-pong)
- Device virtualization introduces additional overhead

Why not Offload

- Long history of attempts to offload TCP/IP processing to the network adapter

- Potential advantage: improved performance due to higher-level interface
  - Less interaction with the adapter (from SW perspective)
  - Internal events are handled by the adapter and do not disrupt application execution
  - Less OS overhead (especially with direct-access HW interface)

- Major disadvantages:
  - High development and testing costs
    - Low volumes
    - complex processing in hardware is expensive to develop and test
    - OS integration is expensive
  - No sustainable performance advantage
    - Poor scalability due to stateful processing
    - Firmware-based implementations create a bottleneck on the adapter
    - Hardware-based implementations need major re-design to support higher bandwidth
  - Robustness problems
    - device vendors supply the entire stack
      - Different protocol acceleration solutions provides different acceleration levels
    - Hardware-based implementations are not future-proof, and prone to bugs that cannot be easily fixed

Alternative to Offload – "Onload" (Software-only TCP offload)

- Asymmetric multiprocessing – one (or more) system CPUs are dedicated to network processing
- Uses general-purpose hardware
- Stack is optimized to utilize the dedicated CPU
  - Far less interrupts (uses polling)
  - Far less locking
- Does not suffer from disadvantages of offload
  - Preserves protocol flexibility
  - Does not increase dependency on device vendor
- Same advantages as offload:
  - Relieves the application CPU from network stack overhead
  - Prevents application cache pollution caused by network stack
- Additional advantage: simplifies sharing and virtualization of a device
  - Can use separate IP address per VM
  - No need to use virtual Ethernet switch
  - No need to use self-virtualizing devices
- Yet another advantage – allows driver isolation