IX: A Protected Dataplane Operating System for High Throughput and Low Latency

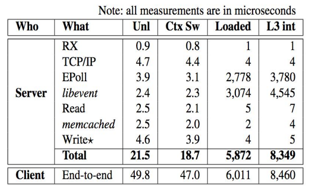
Adam Belay *et al*. *Proc. of the 11th USENIX Symp. on OSDI*, pp. 49-65, 2014.

Presented by Han Zhang & Zaina Hamid

Challenges

Datacenter applications raise the following challenges in OS:

- Low tail latency
  - One user request involves hundreds of servers.
  - Tail latency aggregates to the overall performance.
  - OS plays an important role in exacerbating tail latency.
  - Need tight bounds on 99th percentile latency.



Note: all measurements are in microseconds

Who	What	Unl	Ctx Sw	Loaded	L3 int
Server	RX	0.9	0.8	1	1
	TCP/IP	4.7	4.4	4	4
	EPoll	3.9	3.1	2,778	3,780
	libevent	2.4	2.3	3,074	4,545
	Read	2.5	2.1	5	7
	memcached	2.5	2.0	2	4
	Write⋆	4.6	3.9	4	5
	Total	**21.5**	**18.7**	**5,872**	**8,349**
Client	End-to-end	49.8	47.0	6,011	8,460

Table 1. Latency breakdown of an average request when the server process is unloaded (Unl), when it is context-switching with another process (Ctx Sw), when it is fully loaded (Loaded), and when it is subjected to heavy L3 cache interference while fully loaded (L3 int). All measurements are in microseconds. "End-to-end" is the time reported by mutilate on the clients. ⋆For brevity, we include TCP/IP and TX time in Write.

Table: J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In Proceedings of the 9th EuroSys Conference (Eurosys '14), page 4, 2014.

Challenges

- Low tail latency
- High packet rates
  - Facebook: Short requests (key: 50 bytes, values: 500 bytes).
  - Millions of requests per second (RPS) per node.
  - Impractical to use TCP for all connections.
  - Leverage UDP and aggregation proxy.

Challenges

- Low tail latency
- High packet rates
- Robust protection
  - Multiple services nodes share servers.
  - Network stack isolations provided by kernel or hypervisor.
- Resource efficiency
  - Allocate minimal resources to service nodes to meet requirements.
  - To mitigate diurnal patterns and spikes in traffic,
  - Allocate extras on demand, otherwise save the power.

Bottleneck in Network

- Hardware is FAST!
  - 10 GbE NIC is prevalent.
  - We are marching towards 40GbE, and 100 GbE.
  - Multi-core CPUs and high-speed channels to storage.

Bottleneck in Network

- Hardware is FAST!
- Operating systems have different(~~wrong~~) hardware assumptions.
  - Multiple applications share a single core.
  - Packet inter-arrival time > interrupts and system calls latency
  - Scheduling >> latency and throughput
  - Overheads in buffering and synchronization hurts CPU and memory system.

Alternative Attempts - To Save OS

- Kernel bypass
  - Using user-space networking stacks
  - Example: mTCP
  - Feature: Dedicated threads for TCP stack
  - Issues:
    - Switching overheads --, latency ++
    - Horrible security protection. Only count on NIC supports.

Alternative Attempts

- Replace TCP
  - Example: RDMA, UDP (as mentioned in Facebook)
  - Issues:
    - RDMA: Specialized hardware.
    - UDP: applications need to handle congestion control and reliability check.
- Replace POSIX API
  - Lightweight sockets
  - Still suffer from drawbacks of using kernel-based networking stacks.
- “Repair” OS
  - New socket options, new polling drivers.
  - Only receive incremental benefits.

Outline

- Motivation
- IX Design Approach
- IX Implementation
- Evaluation
- Discussion

Adopting Dataplane

- Middleboxes also require microsecond-level latency and high packet rates.
- Middleboxes use dataplane.
- Why not using dataplane?

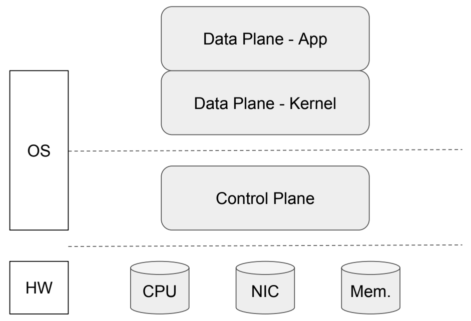
- Before jumping to dataplane, let’s point out several differences.

Difference against Middleboxes

- Middleboxes are different than DC OSes in that:
  - MBs run each packet to completion.
    - New packet -> Protocol processing -> Application processing -> Next packet
  - OS kernels decouple protocol processing from application.
    - Interleave between those two.
    - ACK a packet even though application hasn’t process it yet.

  - MB dataplanes are optimized for synchronization-free operations.
    - This method scales well.
  - OS kernels rely on coherence traffic and synchronization locks.

IX Design Peek



Design Principles - I

- Separation of control and data plane
  - Control plane:
    - Resource configuration, provisioning, management (Coarse grained).
    - Elastically allocate entire cores, large page memory, and NIC queues to dataplanes.
  - Dataplane:
    - A single application in a single address space.
    - Similar to guest OSes in virtualized systems.
    - Direct pass-through access to NIC queues.
  - Three-way virtualized via Dune (control plane, dataplane, untrusted code).

Design Principles - II

- Run to completion with adaptive batching
  - Run through all stages needed to transmit a packet without interrupts.
  - Interleave between protocol processing (*kernel mode*) and application processing (*user mode*) at well-defined transition points.
  - No need for intermediate buffering mechanism.
  - Batching packets throughout the network stack processing.
  - Start batching only under congestion. Set max # packets to be batched.
  - Queues only build up at NIC edge, before packets reaching to dataplane.
  - NIC ACK's parameters (speed, window size) are a reflection of dataplane processing power.

Design Principles - III

- Zero-copy API with explicit flow control
  - Step 1: Get rid of POSIX API.
  - Step 2: Use memory to communicate.
  - Dataplane has two components (non-root kernel and application).
  - Messages between those two are stored in memory prior to transition.
  - Incoming and outgoing packets are kept immutable in memory.
  - Dataplane kernel is in charge of flow control and trim transmission if necessary.
  - Application controls buffers.

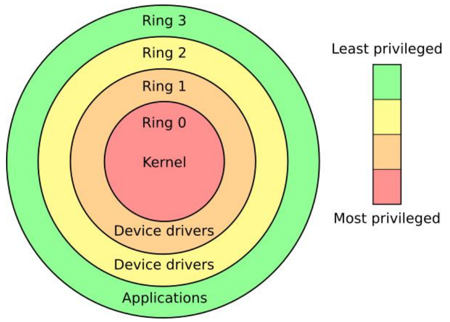
Design Principles - IV

- Flow-consistent, synchronization-free processing
  - Flow-consistent hashing, with receive-side scaling, of incoming traffic to NICs' queues.
  - Each hardware thread serves a single queue per NIC.
  - Eliminates needs for synchronization and coherence traffic between cores.
  - Memory also organized in distinct pools for each hardware thread.
  - Sounds more like incorporating existing methods into new system design.

Outline

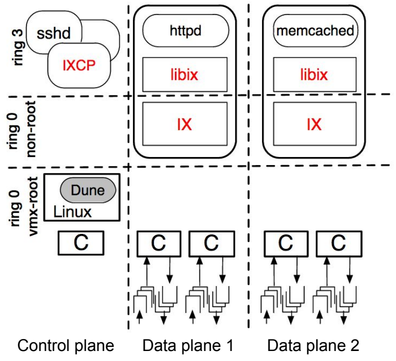
- Motivation
- IX Design Approach
- IX Implementation
- Evaluation
- Discussion

VMX Rings



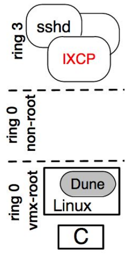
By Hertzsprung at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8950144

IX Overview



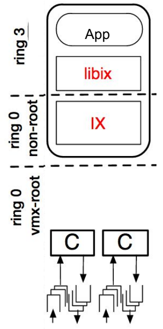
Control Plane

- Consist of Linux kernel and IXCP
  - Kernel: Initialization devices and basic resource allocation; System calls and services.
  - IXCP: Monitor dataplanes and enforce resource allocation policies.
- Allocate resources at coarse-level
- Linux and Dune in root ring 0
  - Similar to hypervisor.
- IXCP is at user-mode



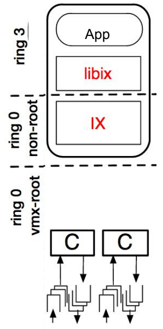
Data Plane

- A single, multithreaded application
- Direct access to hardware features
  - IX ⟷ NIC
- Internal memory management
- Own virtual address translations
  - Large page size (2MB) to reduce translation overhead
- Hierarchical timing wheel for network timeouts
- Implemented with DPDK, lwIP (lightweight IP), and Dune



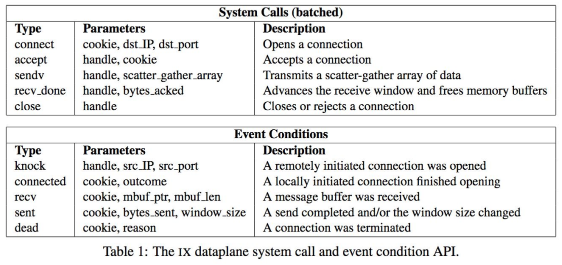
Data Plane - Threads

- Support two types of threads
  - **Elastic threads** (will talk more later)
    - Interact with IX for network I/O
    - Provide high performance with good latency
  - **Background threads**
    - Can issue blocking system calls
    - Timeshare an allocated hardware thread
  - Both can issue POSIX system calls
- Upon changes in hardware threads from CP:
  - Adjust # of elastic threads & background threads



Dataplane API

- Elastic threads interact with IX through:
  - Batched systems calls
  - Event conditions generated by dataplane
  - Direct, restricted memory access to incoming payloads



System Calls (batched)		
Type	**Parameters**	**Description**
connect	cookie, dst_IP, dst_port	Opens a connection
accept	handle, cookie	Accepts a connection
sendv	handle, scatter_gather_array	Transmits a scatter-gather array of data
recv_done	handle, bytes_acked	Advances the receive window and frees memory buffers
close	handle	Closes or rejects a connection

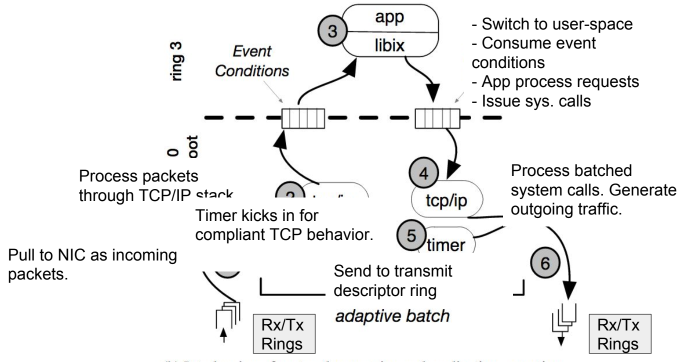
Event Conditions		
Type	**Parameters**	**Description**
knock	handle, src_IP, src_port	A remotely initiated connection was opened
connected	cookie, outcome	A locally initiated connection finished opening
recv	cookie, mbuf_ptr, mbuf_len	A message buffer was received
sent	cookie, bytes_sent, window_size	A send completed and/or the window size changed
dead	cookie, reason	A connection was terminated

Table 1: The IX dataplane system call and event condition API.

Dataplane API

- Sys. calls and events conditions are passed through shared memory
- Expose flow control to application
  - Notify application about bytes sent
  - Current OSes embed flow control in kernel
- `libix` provides simple interface to application developer
- Coalescing in `libix` improves locality and flow control performance

Dataplane API



app
libix
3
ring 3
Event Conditions
- Switch to user-space
- Consume event conditions
- App process requests
- Issue sys. calls
0
oot
Process packets through TCP/IP stack
4
tcp/ip
Process batched system calls. Generate outgoing traffic.
Timer kicks in for compliant TCP behavior.
5
timer
Pull to NIC as incoming packets.
6
Send to transmit descriptor ring
adaptive batch
Rx/Tx Rings
Rx/Tx Rings

(b) Interleaving of protocol processing and application execution.

Scalability and Security

- Elastic threads are (almost) synchronization and coherence free.
  - Commutative IX APIs - Each elastic thread has unique identifier namespace.
  - Optimized API - No concurrent execution, without synchronization primitives.
  - Disjoint subset of TCP flows per thread - Due to flow-consistent hashing.
  - Still need synchronization for shared structures and when CP reallocates resources.
  - But not a scaling bottleneck in system and protocol processing code.
- Better protection than user-level stacks.
  - Application in user-mode, dataplane in protected ring 0.
  - Dataplane can implement firewalls and ACLs.
  - Secure virtual memory protection between dataplane and application.
  - Timeout mechanism to interrupt non-responsive elastic threads. Hand over to CP.

Outline

- Motivation
- IX Design Approach
- IX Implementation
- Evaluation
- Discussion

Setup

- IX compared against Linux Kernel and mTCP
- TCP = networking protocol used throughout
- Methodology
  - 24:1 (clients:server)
  - Each socket has 8 cores, and 16 hyperthreads
  - Intel x520 10GbE NICs
  - Power management features are disabled
  - Scheduling jitter & background tasks - Avoided
  - IX max batch size = 64 packets per iteration

Latency & Single-flow b/w

- Goodput achieved for different message sizes
- 2 IX servers
  - Latency : 5µs / 64B message
  - Goodput 5Gbps / 20KB message
- 2 Linux servers
  - Latency : 24µs
  - Goodput 5Gbps / 385 KB message
- IX's dataplane model polls queues & processes packets to completion
- Linux - Interrupt model that wakes up blocked processes
- mTCP - aggressive batching to offset cost of context switching

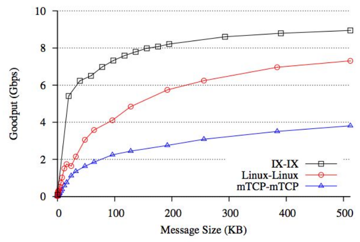


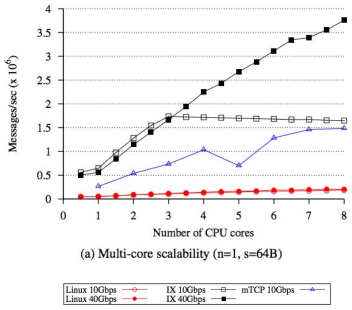
Figure 2: NetPIPE performance for varying message sizes and system software configurations.

Throughput & Scalability

- 18 clients connect to a single server, listening on a single port, send a remote request of size *s*, and wait for an echo
- IX scales more aggressively in all the 3 cases

Core scalability

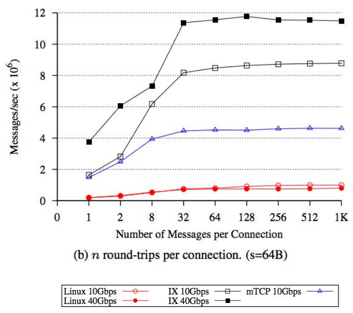
- IX only uses 3 cores to saturate the 10GbE link
- mTCP requires all 8
- IX linearly scales & delivers 3.8 M TCP connections per s on 4x10GbE



(a) Multi-core scalability (n=1, s=64B)

Message count scalability

- IX delivers 8.8 mil messages / s = 1.9 x the throughput of mTCP & 8.8 x the throughput of Linux.
- Also scales in a 4x10 GbE configuration
- Speedup of 2.3x with n=1 and 1.3x with n=1024 over 10 GbE IX

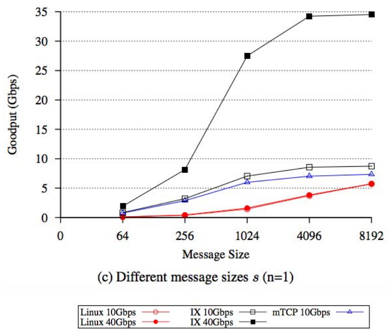


(b) n round-trips per connection. (s=64B)

Message size scalability

- IX can deliver 8KB messages with a Goodput of 34.5 Gbps

Overall IX can scale protected TCP/IP beyond 10GbE even with a single socket multi-core server



(c) Different message sizes s (n=1)

Connection Scalability

- At its peak, IX performs 10x better than Linux
- With all 250k connections @ 4x10GbE IX delivers 47% its own peak
- Drop in throughput is attributed to Performance of memory subsystem & Not an increase in IC.

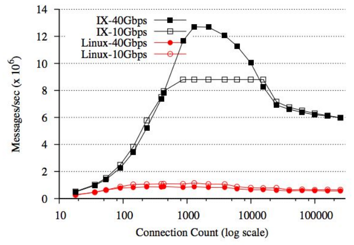


Figure 4: Connection scalability for the 10GbE and 4x10GbE configurations.

Memcached Performance

Background on Memcached

-memcached deployed on **top of the libevent framework**

-**high throughput, low latency** caching tier in front of persistent database servers

-**network bound** application, with 75% of execution time in kernel mode for n/w processing

- 'mutilate' generates load in terms of RPS & measures latency
- 23 client m/cs & 1476 connections
- 2 Representative workloads
  - ETC : 20B - 70B keys, 1B - 1KB values, 75% GET requests
  - USR : <20B keys, 2B values, 99% GET requests (minimum sized TCP packets)
    - Pipeline 4 requests per connections

- 8 cores with linux & only 6 cores with IX (*lock contention*)
- IX latencies are reduced to ~half as compared to Linux (*Linux is running on clients*)

Configuration	Minimum latency @99th pct	RPS for SLA: < 500µs @99th pct
ETC-Linux	94µs	550K
ETC-IX	45µs	1550K
USR-Linux	85µs	500K
USR-IX	32µs	1800K

Table 2: Unloaded latency and maximum RPS for a given service-level agreement for the memcache workloads ETC and USR.

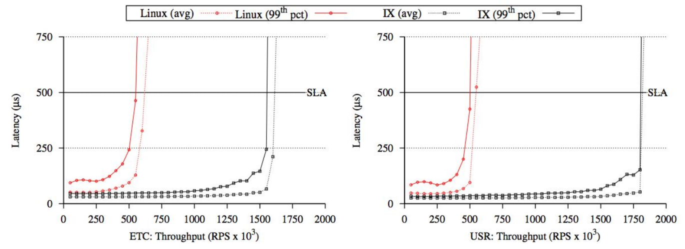


Figure 5: Average and 99th percentile latency as a function of throughput for two memcached workloads.

Outline

- Motivation
- IX Design Approach
- IX Implementation
- Evaluation
- Discussion

What makes IX fast?

- Tight coupling of dataplace architecture that uses minimal amount of batching
- Lack of intermediate buffers
- Zero copy approach
- Tuned for multi-core scalability
- Can also be implemented at the user-level networking stack in general

Adaptive Batching

- Different upper bounds of B = batch size
- At low load B does not impact tail latency
- Larger values of B improve throughput - 29%
  B ≥ 16 maximizes throughput

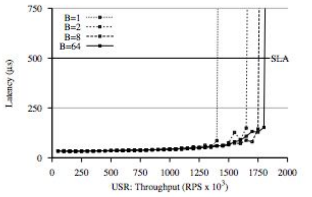


Figure 6: 99th percentile latency as a function of throughput for USR workload from Fig. 5, for different values of the batch bound B.

Another observation

- For high packet rates with smaller batch sizes, high rate of PCIe writes reqd. To post fresh descriptors at every iteration => performance degradation with core scalability
- Coalesced PCIe writes on the receive path to atleast replenish 32 descriptor entries at a time

Current Limitations:

- No exploitation of IOMMUs or VT-d : instead maps descriptor rings to IX memory using Linux pagemap to determine physical addresses
- No advantage of NIC’s SR-IOV capabilities
- Add support for interrupts to IX dataplane

Future Work:

- Explore control plane issues
- Dynamic runtime that rebalances network flows between available elastic threads
- Synergies between IX & networking protocols
- Can also be applied to other network protocols
- Library support for alternative API on top of the low level interface

Further Discussion

What we like:

- Security isolation at NIC queue level
- While measuring memcached performance Linux was being run on clients : Good & bad

Room for improvement:

- Run to completion with adaptive batching - not ideal for small loads
  - Though only starts batching upon congestion.
- How often is resource reallocation?
  - Impact performance due to synchronization at control plane.