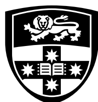
DLibOS - Performance and Protection with a Network-on-Chip

Stephen Mallon, Vincent Gramoli, Guillaume Jourjon

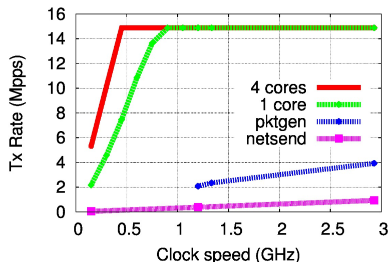
University of Sydney, Data61

November 21, 2017



Performance vs Protection

features	Kernel	kernel bypass
Packet rates	$<$ 1M pkt/s	10-100M pkts/s
99.9th latency (raw packets)	$>$20 us	1 us
Memcached tail latency	$>$20ms	$<$500us
Isolates IO from application	**yes**	**no**



Source of mismatch

IO stack designed around assumption that IO takes milliseconds.

- **System calls** and **memory copying** to perform IO safely in kernel context
- **Excessive locking** within kernel - Memcached profiled to spend 25% time in kernel spinlocks
- Lack of core locality, app may process on different cores to where packets arrive
- Batching and queueing used to amortise context switch cost greatly impacts tail latency

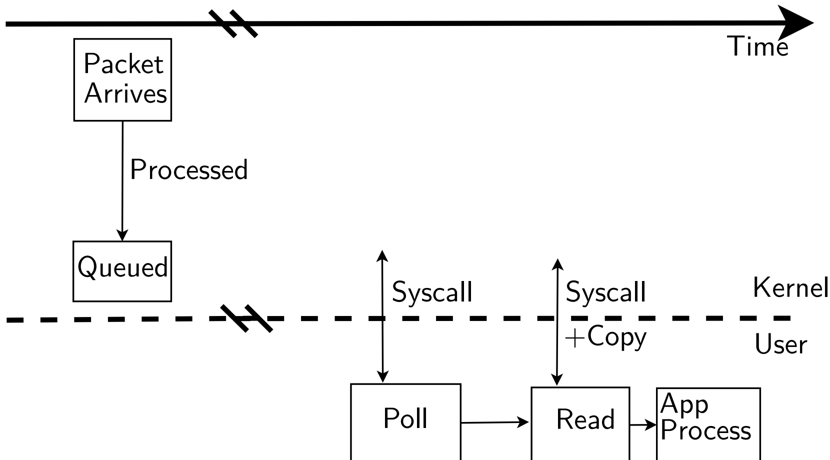
Modern Networking hardware mismatch

Modern hardware offers features allowing building a more efficient network stack.

- SR-IOV: Virtualisation allows partitioned, fast and safe access to NICs
- RSS: core selected from hash(src{ip,port}, dst{ip,port}) - cores do not need to share networking state
- Packets placed directly into L3 cache, avoid batching/queueing

[1] ATC'12

Berkeley Sockets - Example: Receiving



Existing approach - Bypassing the OS

Avoid Copies and Context Switches by processing network in userspace

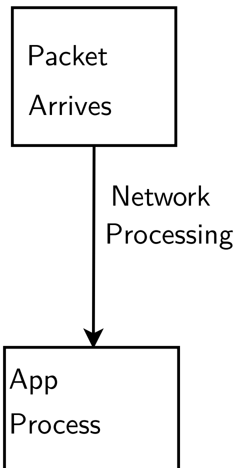
- Existing approaches require dedicating NIC to application
- No Memory Isolation between application and network
- Application is trusted not to corrupt network, send invalid packets

Run To Completion

Run to Completion (RTC) processes packets on same core to 'completion' without queueing

- Best performance according to conventional wisdom
- Best temporal locality (packets already in CPU cache)
- No timing guarantees (handling TCP timeouts/acks) within appropriate timebound

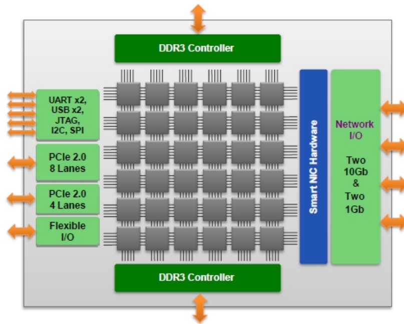
Run To Completion



Manycore Architectures

- Increased core count at expense of per-core performance
- Promises improved energy efficiency
- Power consumption and cooling are a limiting factor in data centres
- Manycore *should* offer better aggregate performance

Many Core TileGX



Cores communicate over network on chip instead of shared memory

.	latency
Shared Memory	>40ns
NoC messages	1 cycle per hop

Current Approaches

- Current approach: Net privileged, app unprivileged - Slow
- Net and app in same process (unprivileged) - fastest, no isolation
- Net in VM (semiprivileged), app unprivileged - isolation, fast

Sandstorm

Userspace TCP/IP stack and webserver on top of netmap.

- Presegmented webpages and stored them in memory.
- Order of magnitude improvement over Nginx + Linux.

Problems:

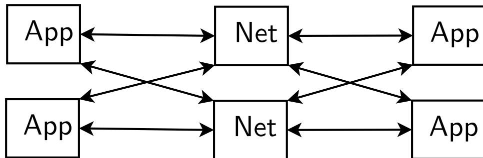
- Doesn't work for different MTUs, changing window sizes
- Hardware offloads make presegmentation unnecessary
- No memory isolation between netstack and application.
- No timing guarantees (no preemption)

IX

- Run network stack in VM ring 0, application in VM ring 3
- Replace Berkeley sockets with a more efficient interface
- Use context switching to enforce memory isolation
- Batching to amortise context switching cost
- Interrupts and preemption to handle timers/ack processing
- Requires dedicated entire NIC to IX (kernel cannot share access)

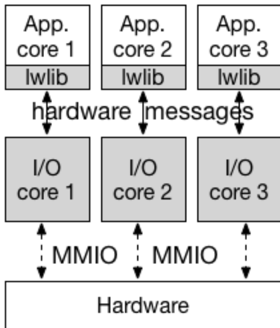
Improves on problems of Sandstorm, but is it possible to achieve memory isolation and time guarantees without context switching?

Design - New Approach

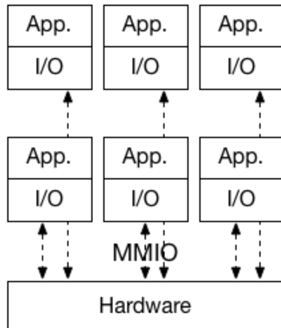


- Run network processing in separate user process from application.
- Some cores dedicated to network processing, some to application processing.
- Explicitly share only some memory between net, app and NIC to enforce memory isolation.
- Communicate using Network on Chip - faster than context switching.
- Scheduling becomes a layout problem instead of a time multiplexing problem.

Design



(c) DLibOS (shaded area)



(d) Traditional user-level libraries run-to-completion (RTC)

Advantages

- Better use of per-core resources (Cache, TLBs, instruction cache)
- Timing/correctness guarantees - acks/timeouts within a timebound - even when application is stuck
- Low latency, Memory isolation without context switching
- Doesn't interfere with Linux kernel (Kernel stack uses separate MAC address on same NIC)
- Can run multiple stacks - unique MAC address per stack

Disadvantages

- Complexity - Application must hold onto memory until remotely acknowledged
- Limited by hardware resources (NIC, cores, memory)
- On-chip network, care needs to be taken to avoid deadlocks, stalling
- Less cores for app and network than traditional approach, can lead to over/under provisioning

Shared Memory

App Cores

Net Cores

NIC

App Mem

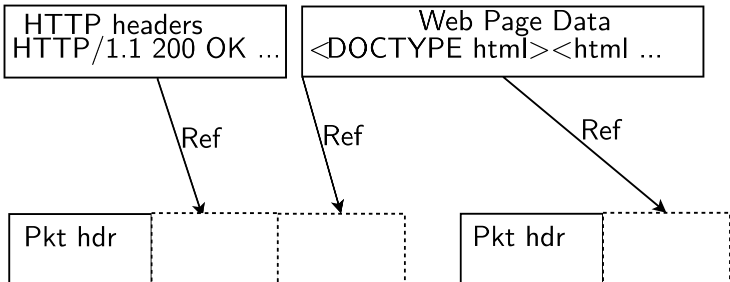
TX Mem

RX Mem

Read/Write

Read only

Scatter Gather IO



Zero touch

- Webserver saves webpages to shared memory region
- Pass reference to http response to net core
- Net core passes this to hardware for DMA, without ever examining the contents of memory
- Web page memory never touches processor cache (directly DMAd)
- Checksums offloaded to hardware

Share Nothing Design

- Run multiple Network cores
- Consistent flow hashing means we run a separate network stack on every net core
- Each stack does not share any data (ARP, flowtables)
- Allows network processing to scale linearly with number of net cores
- Connections identified by (netcore, id) instead of fd - don't need to synchronise an fd table

Intercore Messaging

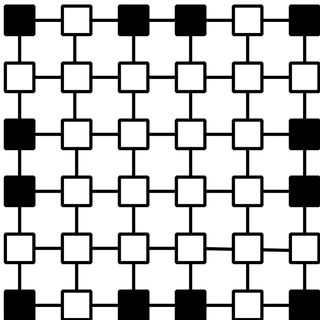
Events (net -> app)	Commands (app -> net)
new_conn	accept
new_data	recv_done
data_acked	send
remote_closed	shutdown
cleanup	close

- Netcore sends events over on chip network
- App send commands to net core
- recv_done and data_acked needed as net/app cannot free buffer until other side is finished

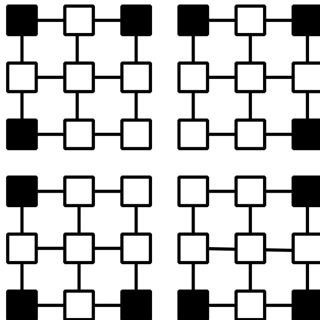
Core layout

- Can Adjust number/ratio of network and application cores based off workload
- Spread network cores out to minimise hops
- Restrict net/app pairs by policy e.g. only within same quadrant - reduces path contention

Core Layout



(e) All-to-all 12:24 (d12-all)



(f) quadrant 12:24 (d12-q)

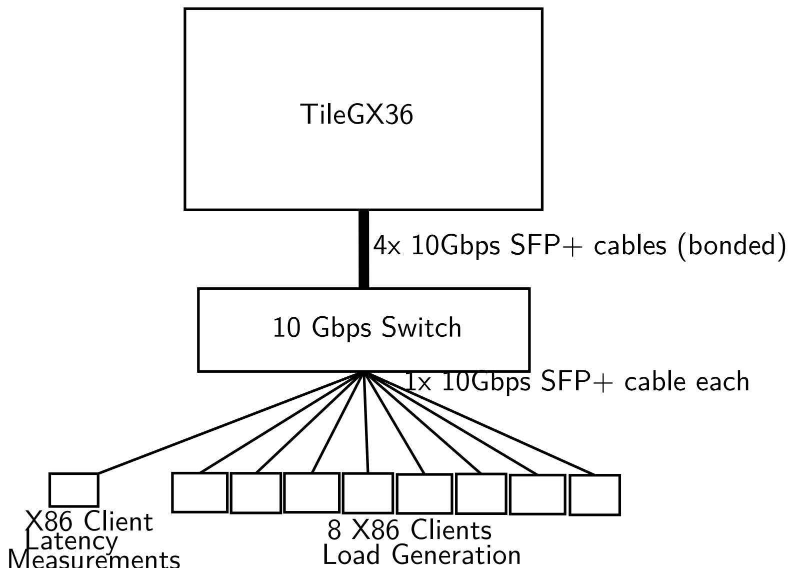
Contribution

- Driver + Network stack (eth, arp, ip, icmp, tcp) + custom socket layer (not compatible with Berkeley sockets)
- Network stack can run as both run-to-completion or dedicated cores transparent to application
- Ported Memcached to run on stack (both run-to-completion and dedicated cores)
- In-memory webserver, and some microbenchmarks
- Benchmarking results

Evaluation

- Evaluated performance of solution 'DLibOS' against 'RTC' same code in run-to-completion mode
- Calls function directly instead of messaging
- Expect RTC to perform better, but without isolation

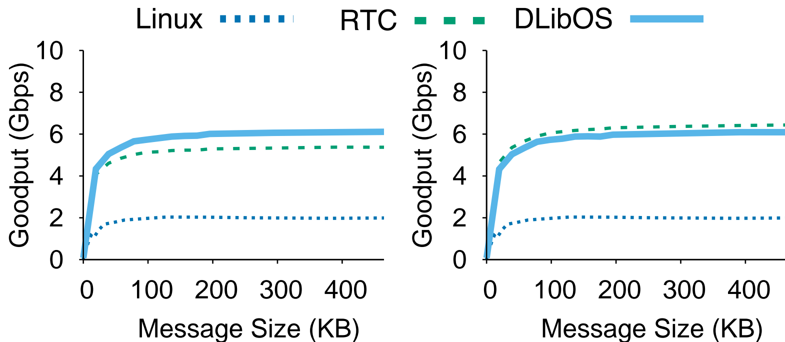
Testbed



NetPIPE

- Tests Throughput and latency of a single connection
- Used 1 TileGX socket as a client, another as a server
- Also evaluated NetPIPE without performing a copy per request
- 6 us one-way latency 64 byte TCP payload
- Linux 31 us one-way latency

NetPIPE



(j) NetPIPE performance

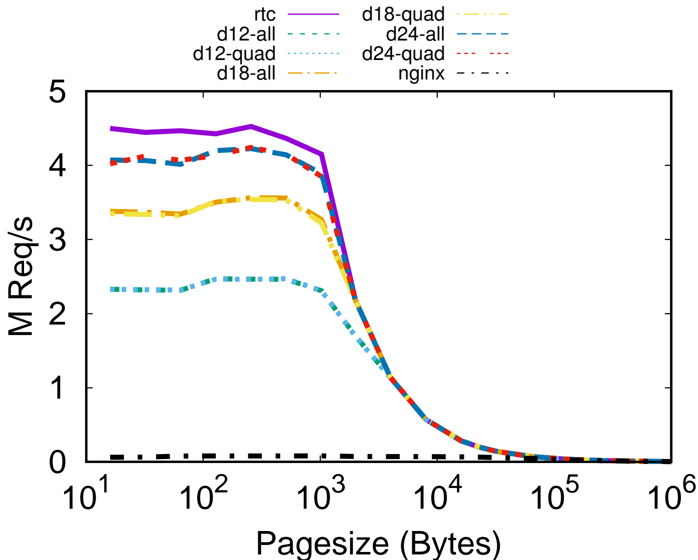
(k) NetPIPE (no copy)

Figure: DLibOS outperforms RTC when more work per request is involved

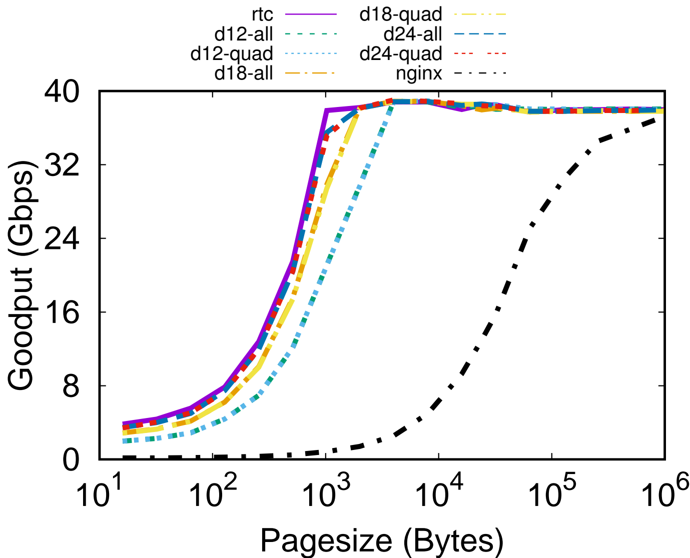
Webserver

- Simple in memory webserver - only supports get
- NGINX used as baseline, serve files from ramdisk, disabled logging
- Run-to-Completion as upper bound
- wrk used to generate workload, 2048 connections
- keepalives enabled, pipelining disabled

Webserver Requests per Second



Webserver throughput

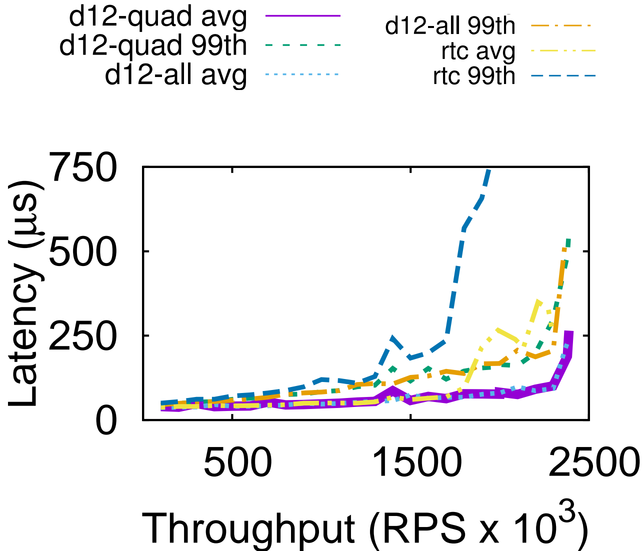


Memcached

- 'Mutilate' benchmark client running on x86 machines, using kernel stack
- Facebook Memcached workloads [1]
- Baseline memcached running on kernel netstack
- Modified Memcached to enable 'zero-touch' (only IO, no improvements to core)
- Set 500us upper bound for 99th percentile - find highest throughput at that bound
- Linux was unable to meet the SLA under any load, also present 5000us 99th percentile
- 1392 client connections, 32 measuring latency

[1] SIGMETRICS'12

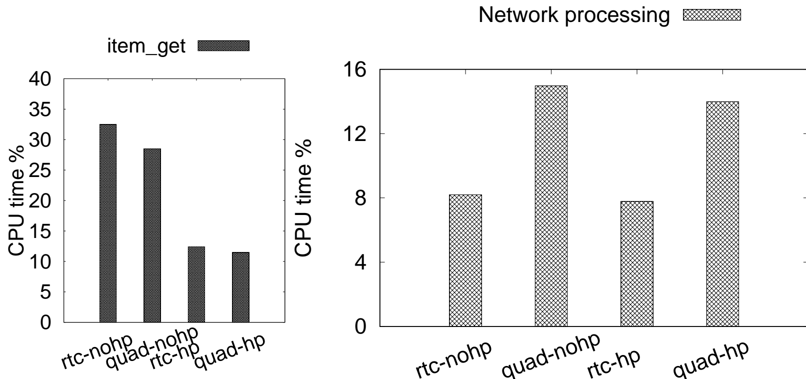
Memcached results



RTC vs DLibOS

- Expect RTC to outperform specialised cores (Significantly less work)
- RTC better in webserver, but worse in Memcached
- Webserver has almost no application work, no lock contention
- Before optimising Memcached, RTC performed even worse - Increased lock contention increases tail latency
- RTC greater impact from lock contention due to using 50% more cores

Memcached Profiling



- Profiling of RTC vs DLibOS shows a fairness issue, network processing starved of CPU time by application processing (no pre-emption)
- Lock contention dominates workload

Future Work - X86

Port netstack to x86

- Use lock free ringbuffers instead of Network on Chip
- DPDK instead of custom driver
- Test effect of NUMA architectures

Storage

- NVMe SSDs have a similar userspace polling mechanism (see Intel SPDK)
- Could run Net, Storage and App as three classes of cores, or combine Net/Storage into IO core
- Port database to IO stack
- Optimise for DB workload, e.g. append only Write-ahead-log

Conclusion

- Invalidate commonly held belief that user-level IO cannot achieve protection and performance
- It is possible to achieve memory isolation, correct time guarantees without context switches and preemption
- Significantly outperforms Linux kernel
- Run-to-Completion can be outperformed by specialised cores - contradicts conventional wisdom