SCYLLA

SCYLLA.

Seastar @ Core C++

Tel Aviv, May 2019

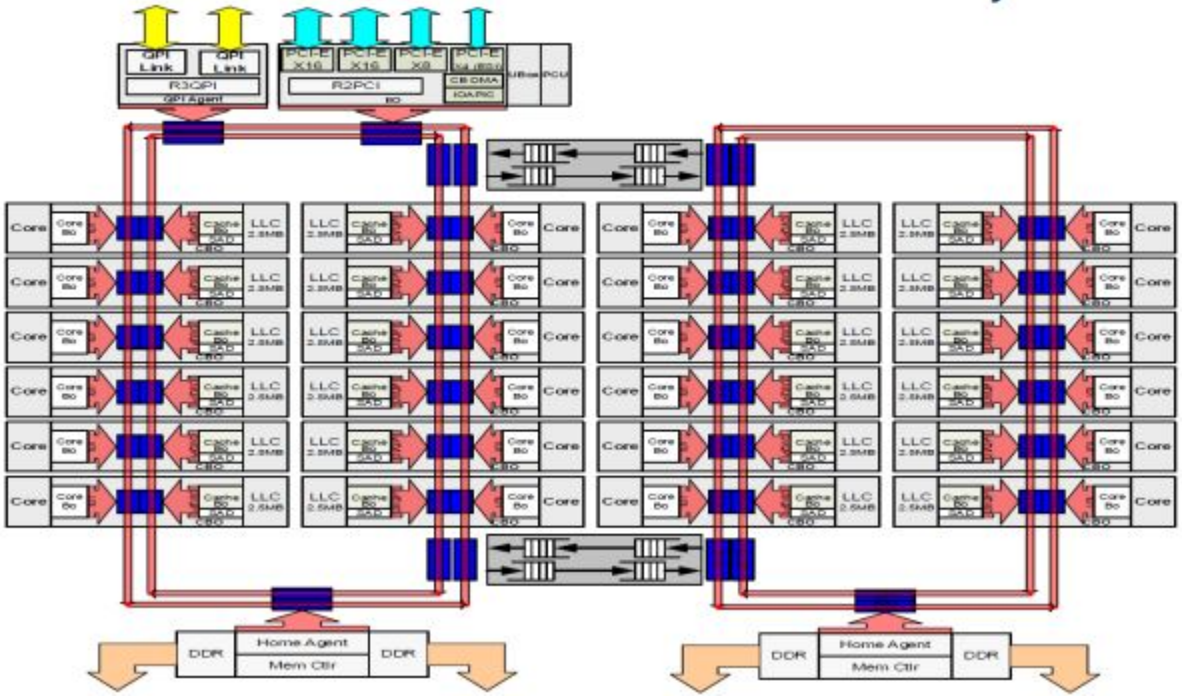
Avi Kivity (@AviKivity)

jobs@scylladb.com

Seastar:
A C++ Asynchronous Programming Framework

SCYLLA

Intel® Xeon® Processor E5 v4 Product Family HCC



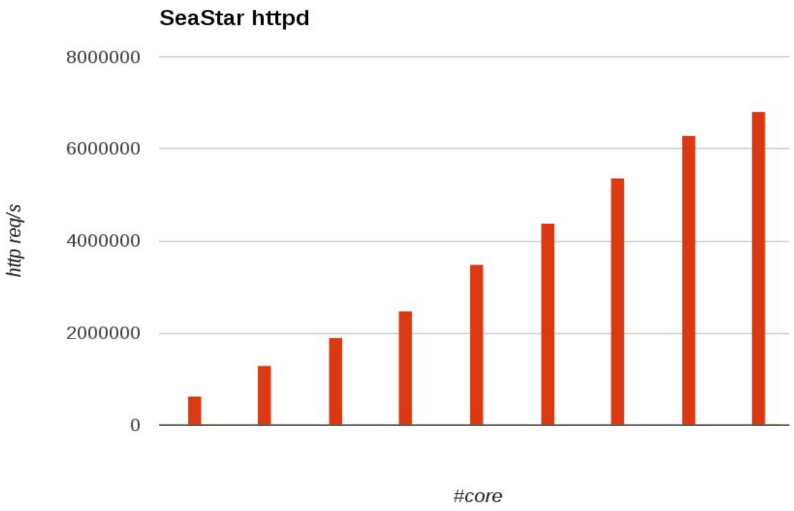
Multi-domain async programming

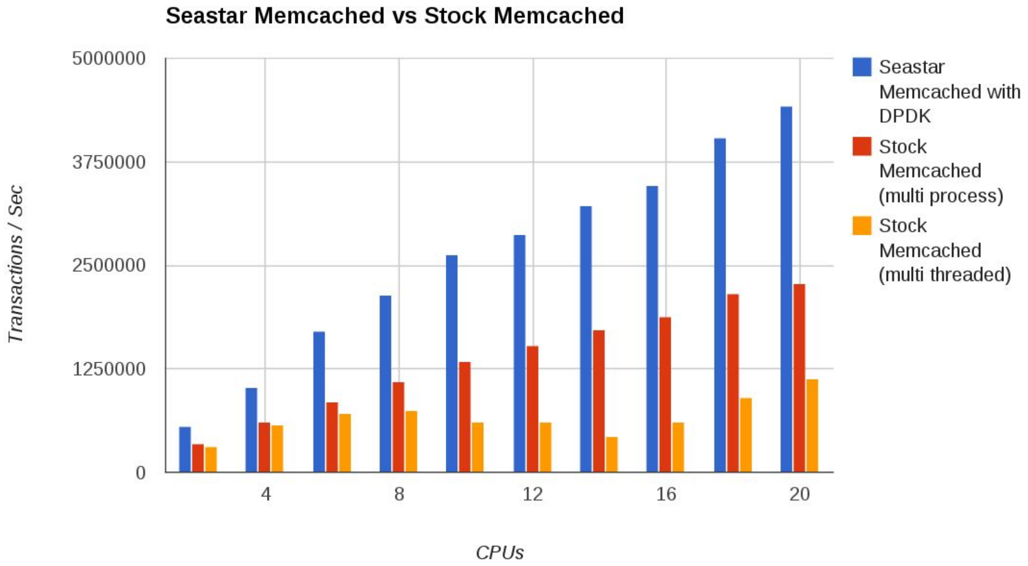
Async networking

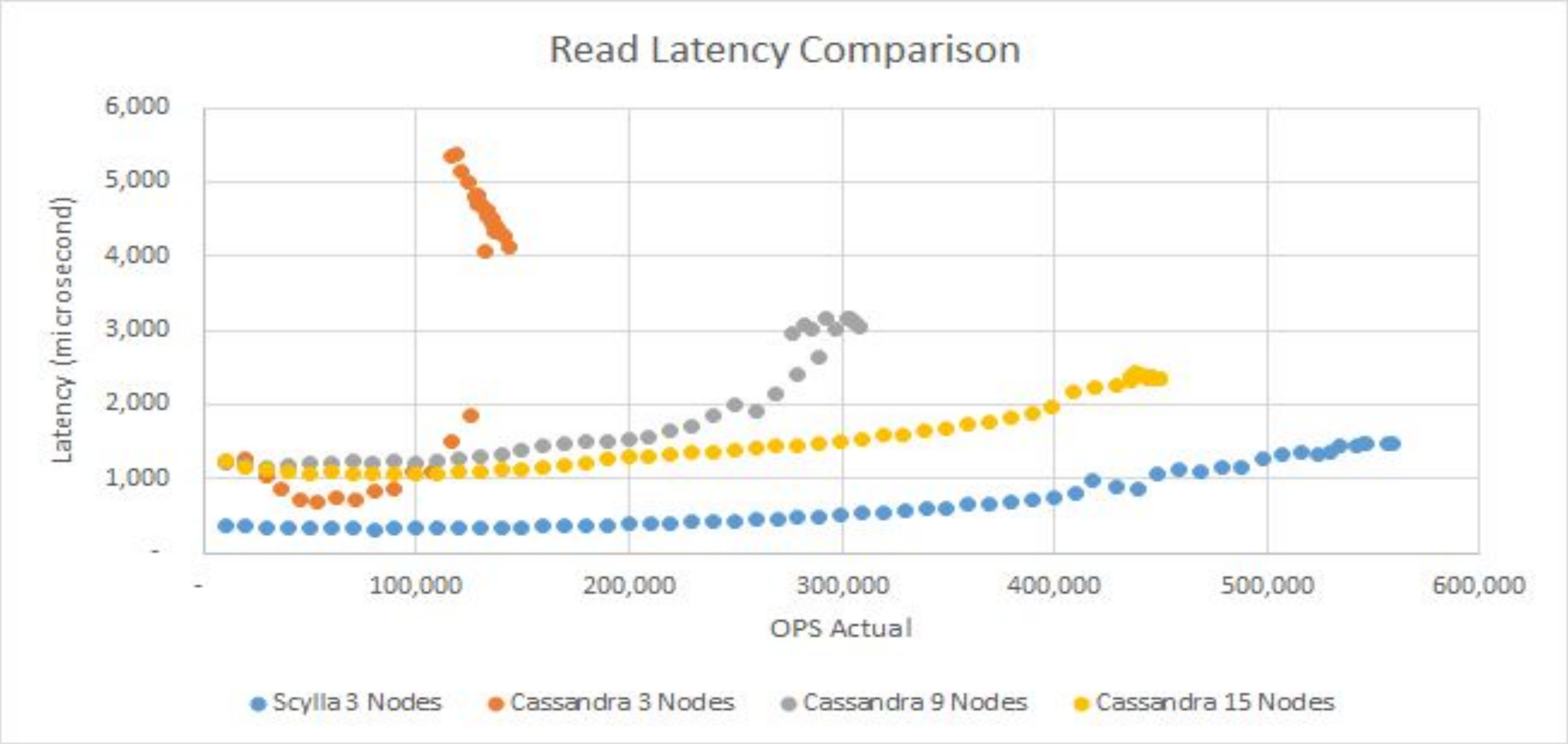
Async storage I/O

Async communications for multi-core, NUMA

RESULTS


SeaStar httpd
8000000
6000000
4000000
2000000
0
http req/s
#core


Seastar Memcached vs Stock Memcached
5000000
3750000
2500000
1250000
0
Transactions / Sec
4
8
12
16
20
CPUs
Seastar Memcached with DPDK
Stock Memcached (multi process)
Stock Memcached (multi threaded)



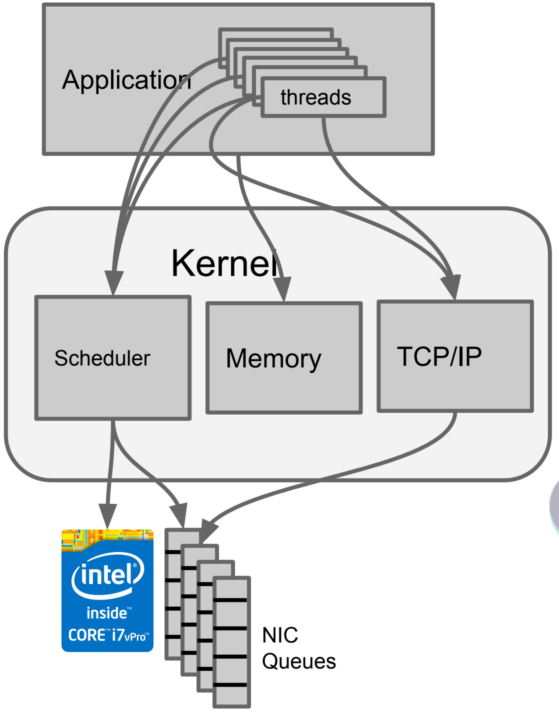
Read Latency Comparison
6,000
5,000
4,000
3,000
2,000
1,000
-
Latency (microsecond)
-
100,000
200,000
300,000
400,000
500,000
600,000
OPS Actual
Scylla 3 Nodes
Cassandra 3 Nodes
Cassandra 9 Nodes
Cassandra 15 Nodes

THREADING MODELS

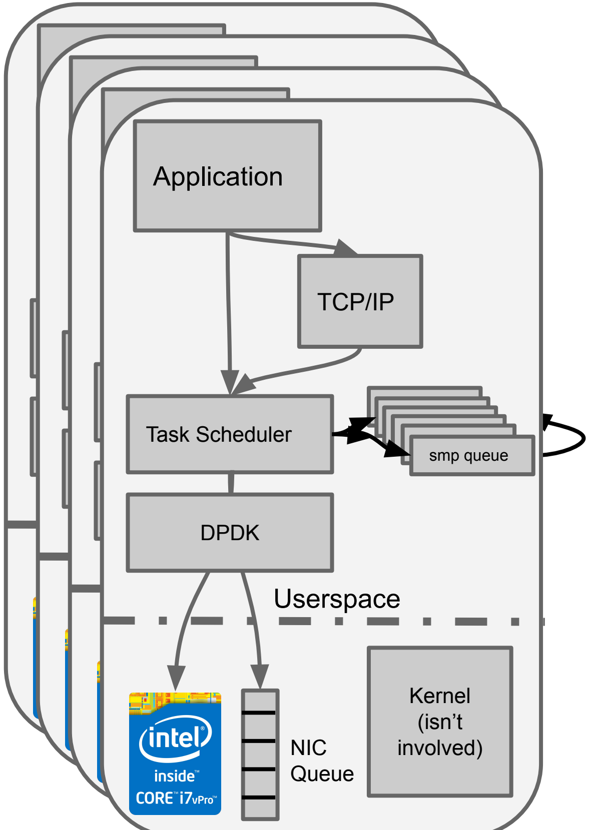
Before: Thread model

After: Seastar shards

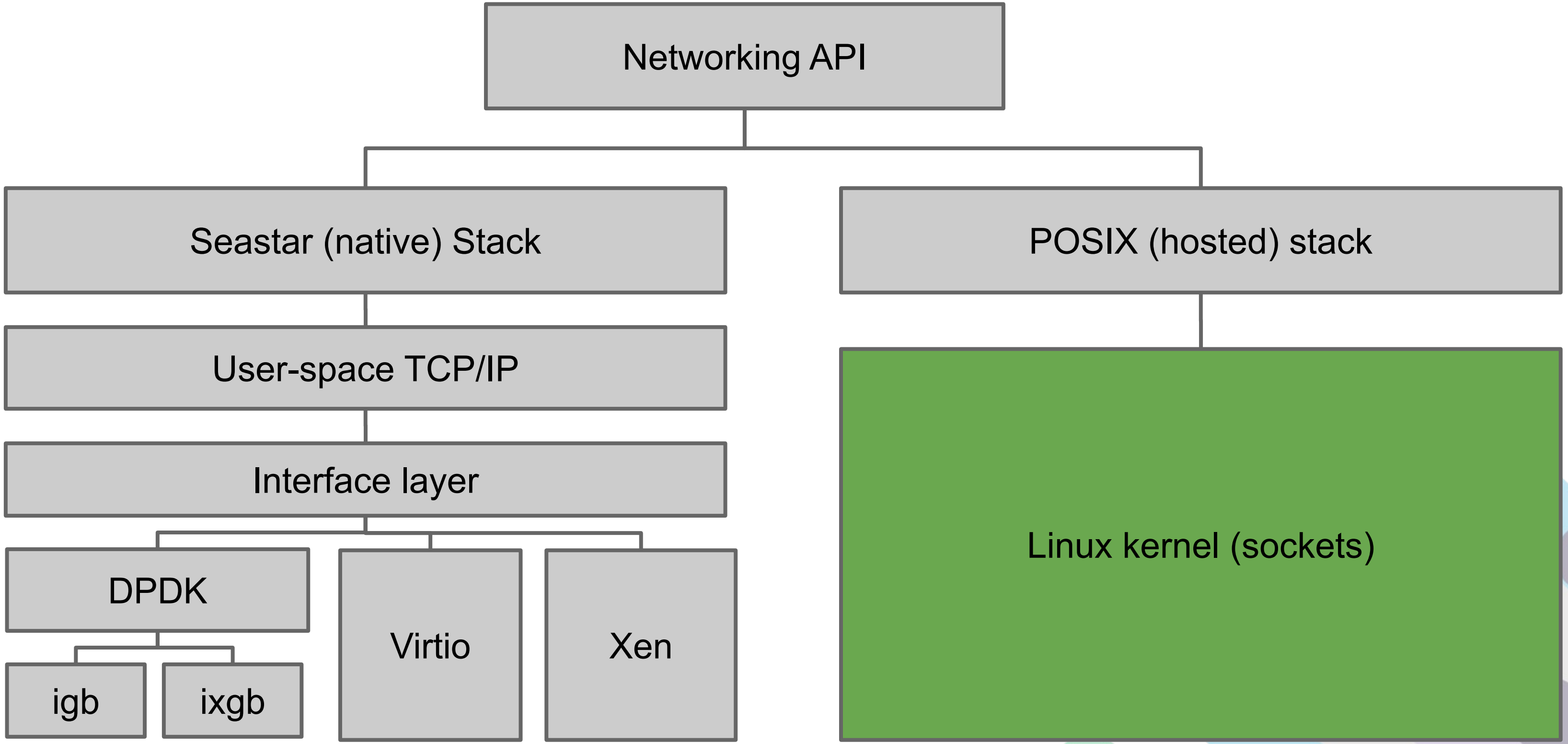
Traditional threading model



Seastar model



Dual networking stacks



Seastar model summary

- Each logical core runs a shared-nothing run-to-completion task scheduler
- Logical cores connected by point-to-point queues
- Explicit core-to-core communication
- Shard owns data
- Composable Multicore/Storage/Network APIs
- Optional userspace TCP/IP stack

CODING IT:
Futures and promises

BASIC MODEL

- Futures
- Promises
- Continuations

F-P-C Defined: Future

A future is a result of a computation that may not be available yet.

- Data buffer from the network
- Timer expiration
- Completion of a disk write
- Computation on another core
- Result of computation that requires the values from one or more other futures.

F-P-C Defined: Promise

A promise is an object or function that provides you with a future, with the expectation that it will fulfil the future.

F-P-C Defined: Continuation

A continuation is a computation that is executed when a future becomes ready (yielding a new future).

Basic Future/Promise

```
future<int> get(); // promises an int will be produced eventually
future<> put(int)  // promises to store an int

future<> f() {
    return get().then([] (int value) {
        return put(value + 1).then([] {
            std::cout << "value stored successfully\n";
        });
    });
}
```

std::future<>?

seastar::future

- Single threaded
- Embedded state
- No locks

std::future

- Thread-safe
- Allocated state
- Locks

Parallelism

```
void f() {
    std::cout << "Sleeping... " << std::flush;
    using namespace std::chrono_literals;
    sleep(200ms).then([] { std::cout << "200ms " << std::flush; });
    sleep(100ms).then([] { std::cout << "100ms " << std::flush; });
    sleep(1s).then([] { std::cout << "Done.\n"; engine_exit(); });
}
```

Zero-copy

```
future<temporary_buffer<char>> connected_socket::read(size_t n);
```

temporary_buffer points at driver-provided pages if possible

discarded after use

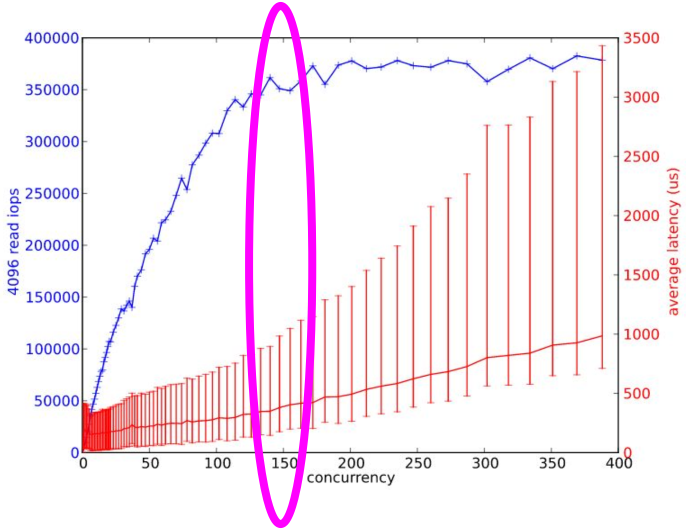
Threads

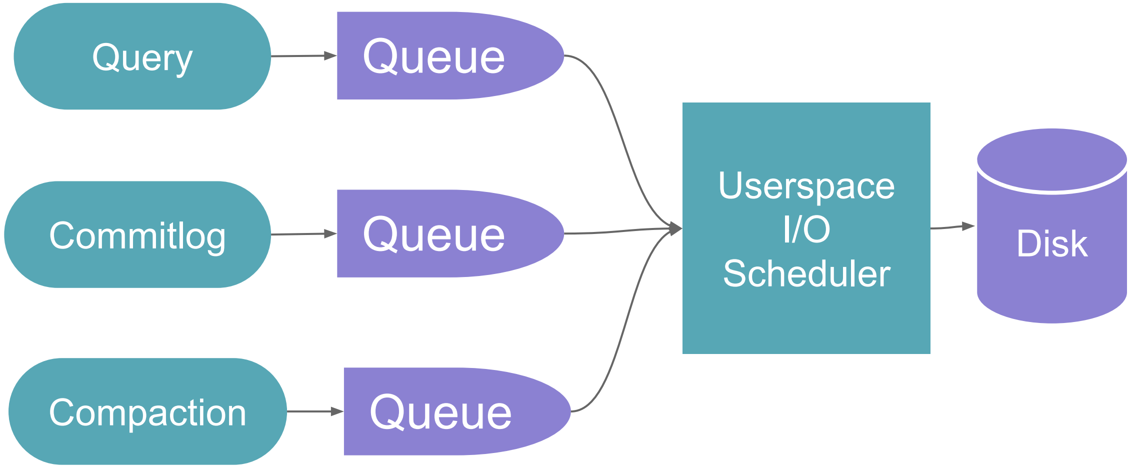
- Cooperative
- Lightweight
- Compose with continuations
- Useful for compute-intensive work interleaves with small amounts of I/O
  - Like processing sequential files

```
future<> f() {
    return seastar::async([] {
        auto v = read().get();
        write(v).get();
        std::cout << "value stored\n";
    });
}
```

I/O Scheduling

Max useful disk concurrency





CPU Scheduling

- Assign continuation chains and threads to scheduling groups
- Assign shares to scheduling groups
- Step back and let them fight it out

Coroutines

```
future<> f() {
    auto value = co_await read();
    co_await write(value + 1);
    std::cout << "value stored\n";
    co_return;
}
```

Rich APIs

- HTTP Server
- HTTP Client
- RPC client/server
- map_reduce
- parallel_for_each
- iostreams
- iosched
- Threads!

- sharded<>
- when_all()
- timers
- sleep
- semaphore
- gate
- pipe/queue
- Memory reclaimer

- stream<>
- execution_stage<>
- TLS
- Prometheus/collectd
- DNS

Ports

- x86_64
- aarch64
- IBM p
- IBM z

C++ features used

- Lambdas (extensively)
- Variadic templates
- Constexpr if
- Coroutines
- Custom operator new + sized deletes
- Fold expressions
- Concepts (pre-C++2a version)
- Metaprogramming (constexpr, old-style)

SCYLLA

USE CASES

Applicability

- High I/O to compute ratio
- High concurrency
- Mix of disk and network I/O
- Complex loads
- Cluster (sharded) applications

Applicability

- Distributed databases
- Object stores, file systems
- Complex proxies/caches

MORE INFORMATION

http://github.com/scylladb/seastar

http://docs.seastar.io

http://seastar.io

http://docs.seastar.io/master/md_doc_tutorial.html

@ScyllaDB

SCYLLA

Thank you.