Improving ivy scalability

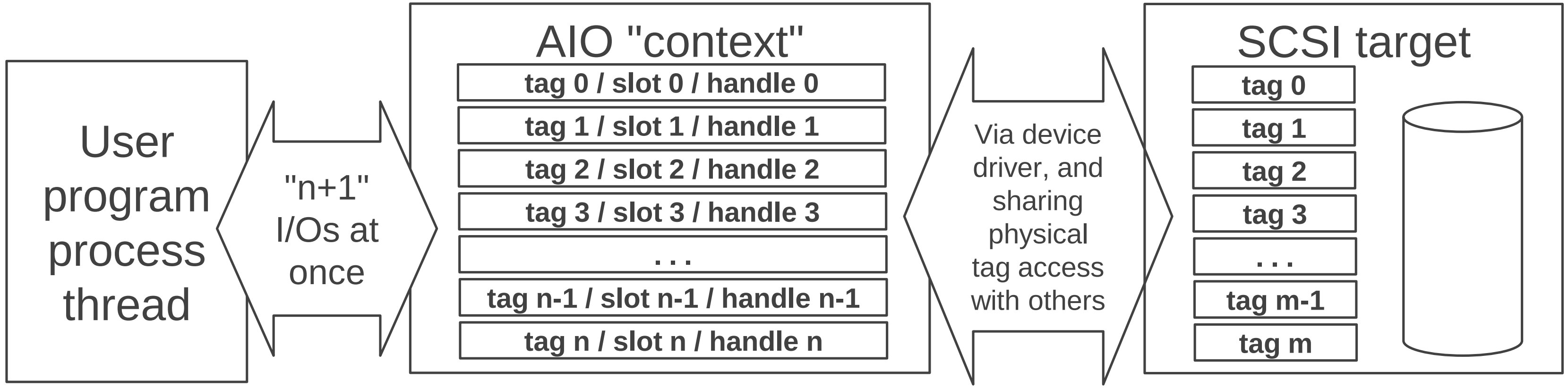
by doing everything with one thread per LUN.

https://github.com/Hitachi-Data-Systems/ivy

Hitachi Vantara - 2018-05-29

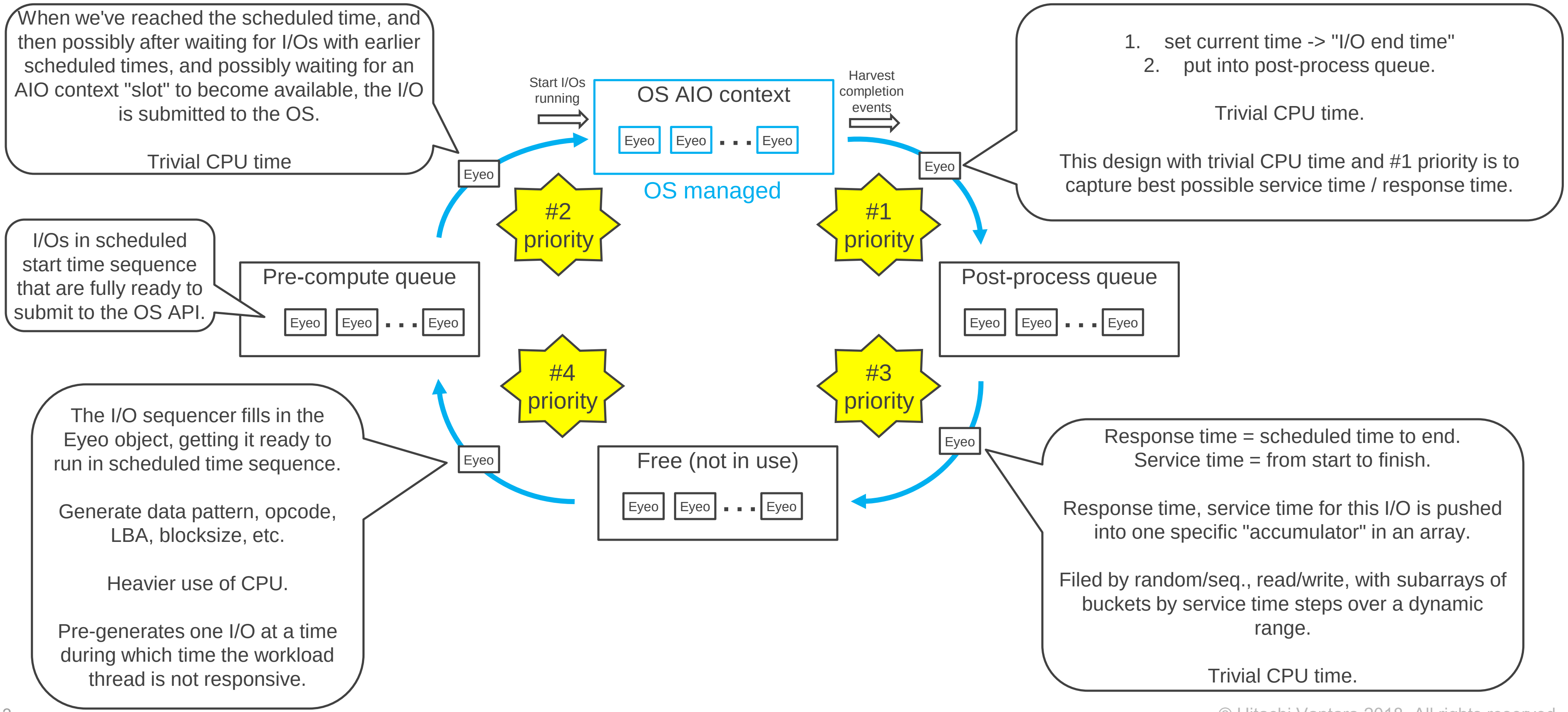
Allart Ian Vogelesang ian.vogelesang@hitachivantara.com

WorkloadThread uses AIO


User program process thread
"n+1" I/Os at once
AIO "context"
tag 0 / slot 0 / handle 0
tag 1 / slot 1 / handle 1
tag 2 / slot 2 / handle 2
tag 3 / slot 3 / handle 3
. . .
tag n-1 / slot n-1 / handle n-1
tag n / slot n / handle n
Via device driver, and sharing physical tag access with others
SCSI target
tag 0
tag 1
tag 2
tag 3
. . .
tag m-1
tag m

- Asynchronous I/O (AIO) uses the concept of a "context" which allows a single user process thread to launch and keep track of as many simultaneously running I/Os as you have slots in the AIO context.
- The AIO context can't "see" any underlying mechanisms, such as physical I/O tag numbers.
  - The device driver may use more tags and may use less tags than the number of AIO slots in a context.
- The AIO context is not aware of other AIO contexts and other users sharing the underlying device.

Life cycle of an ivy I/O



When we've reached the scheduled time, and then possibly after waiting for I/Os with earlier scheduled times, and possibly waiting for an AIO context "slot" to become available, the I/O is submitted to the OS.

Trivial CPU time

Start I/Os running
OS AIO context
Eyeo Eyeo . . . Eyeo
OS managed
Harvest completion events
Eyeo
#2 priority
#1 priority

1. set current time -> "I/O end time"
2. put into post-process queue.

Trivial CPU time.

This design with trivial CPU time and #1 priority is to capture best possible service time / response time.

I/Os in scheduled start time sequence that are fully ready to submit to the OS API.
Pre-compute queue
Eyeo Eyeo . . . Eyeo
Post-process queue
Eyeo Eyeo . . . Eyeo
#4 priority
#3 priority
Free (not in use)
Eyeo Eyeo . . . Eyeo
Eyeo

The I/O sequencer fills in the Eyeo object, getting it ready to run in scheduled time sequence.

Generate data pattern, opcode, LBA, blocksize, etc.

Heavier use of CPU.

Pre-generates one I/O at a time during which time the workload thread is not responsive.

Response time = scheduled time to end.
Service time = from start to finish.

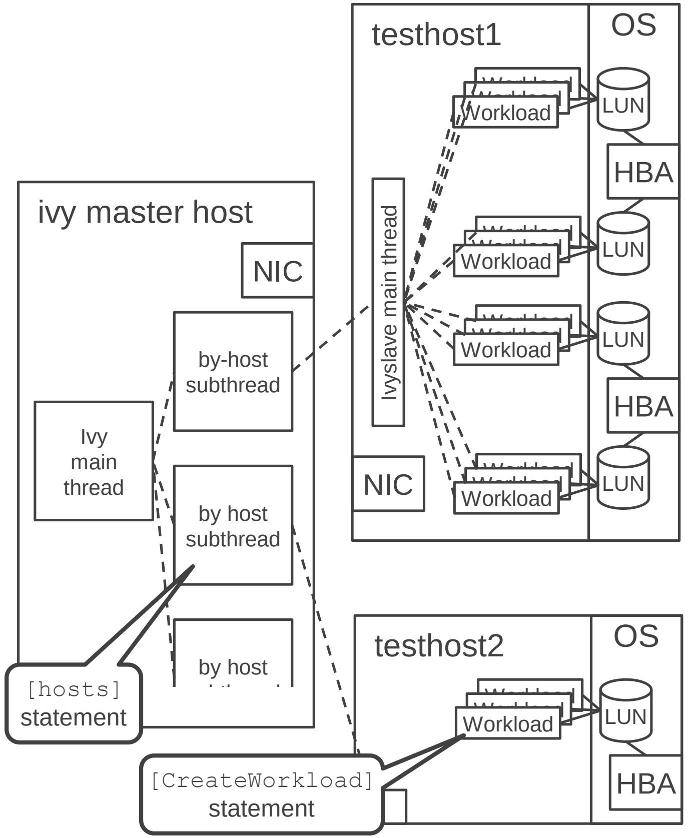
Response time, service time for this I/O is pushed into one specific "accumulator" in an array.

Filed by random/seq., read/write, with subarrays of buckets by service time steps over a dynamic range.

Trivial CPU time.

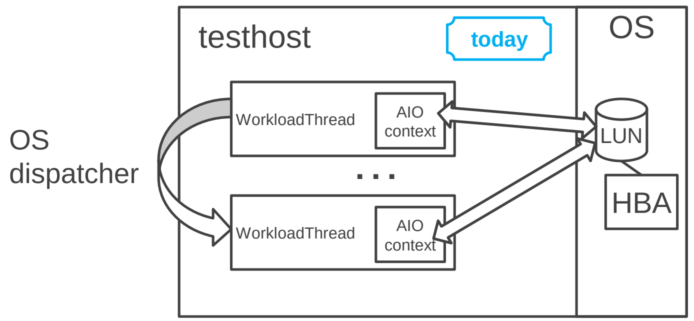
Today one thread for each layered workload on each LUN

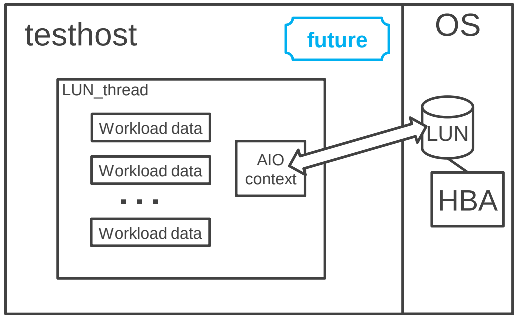
HITACHI Inspire the Next



- "Create Workload" creates a named "workload" subthread attached to a particular LUN.
- It's normal to create a workload of the same name on a bunch of LUNs, and to manage that group subsequently by workload name.
- Each workload connected to a LUN is implemented as a subthread using an AIO context to drive I/O to the LUN.
- If you layer multiple (differently named) workloads on the same LUN, each workload thread will have its own AIO context, each of which competes for use of the underlying device driver / HBA interface.

Idea to further reduce OS dispatcher use





- Move to a model where within a single workload thread bound to a LUN, we could merge the precompute queue I/O streams from multiple, independent named subworkload iosequencers to drive the LUN's sole AIO context.
- Where multiple workloads are layered on a LUN, additional OS dispatcher overhead is avoided.
- Each workload would have its own I/O sequencer, and these I/O sequence streams would merge onto the single AIO context in scheduled start time sequence.
- Where there are multiple sub-workloads each running IOPS = max, the merge point mechanism would use a fixed "IOPS skew" or weighting factor.

Experienced performance engineers know

- It can be very misleading when you are not measuring what you think you are measuring.
- For example, if we had overwhelmed the Linux dispatcher with thousands of ivy threads, all expecting to be dispatched promptly, we might think that we are measuring storage IOPS and service time, but what we are really measuring is OS dispatcher latency.
- Ivy now produces an "interlock_latencies" output folder with csv files recording OS dispatching latencies as part of a full set of interlock protocol latency data covering both test host and command device interlock protocol latencies.
- It is anticipated that the data in these csv files will inform us about the quality of the measurement data. If the interlock protocol is highly responsive, the data will be more "clean".

How much will this help?

- Extra threads only come when we layer multiple workloads on each LUN, like we do in the VMware workload.

- If you have 64 ports, each with 16 LUNs and each have 3 workloads.
  - With today's ivy, that would be 64 x 16 x 3 = 3072 workload threads spread over the test hosts. The new method would cut that to 1024 threads.

- You can detect if you are suffering from OS dispatcher latency by looking at the ivy interlock latency csv files in the `interlock_latencies` folder.

- You could also run some experiments with the same IOPS but spread over different numbers of workloads on each LUN.

- This looks like a good upgrade. Implementation depending on priorities. This would be a "big" development item, but not "really big".

How would we do this?

- As part of switching to JSON format workload and Go statement parameters.
- An ivy `WorkloadThread` has an AIO context to a LUN.
- A workload has an iosequencer type, and parameters for the sequencer.
- Using JSON would let us very easily define workloads using a mix of multiple sub-iosequencers.
- The output stream of I/Os from multiple iosequencers would be merged or mixed together by scheduled execution time to be run over the AIO context.
- Where workload components are running at IOPS = max, a skew % would be used where the workloads merge.

In some cases, we may need *more* threads on a test host

- For example, you have small number of enormous LUNs backed by a pool.
- You want enough threads doing the work to take advantage of the number of cores in the test host to get the highest I/O rate.
- Just make multiple workloads and layer them in ivy just like you do today.
- This new multiple sub-iosequencer capability is additional to the existing mechanism, not a replacement.

It's a case of why didn't I think of that before?

- Once you have a parsed set of iosequencer settings for each of multiple iosequencers being used by the one per-LUN thread, which will come from the switch to JSON and matching against a schema, the update to the ivy AIO engine is fairly simple, by I/O decide which of the sequencers will be used to generate the I/O.
- But the there all the machinery in ivy that magically juggles the data and makes the csv files. Hard to say how much of a rebuild vs. a rewrite.
- I'm thinking of maybe using same WorkloadID concept, but instead of host-/dev/sdxx-workload_name, extending to host-/dev/sdxx-0-workload_name.
  - The new layer –0 would be the number of the thread on this LUN
  - Each workload name may only appear once on any thread on a LUN.
  - This idea is to "adapt" existing mechanisms, this would be one way of packaging it.
  - Each LUN thread (there wouldn't be a WorkloadThread any more) could have multiple sequencers
- I wish I would have thought of this from the beginning.

HITACHI
Inspire the Next

Not looked at yet

- How we will print csv files the same way whether or not it was a workload thread like today, or a sub-workload. Look at data structure refactoring.
- Both should inhabit the same name space.