HITACHI
Inspire the Next

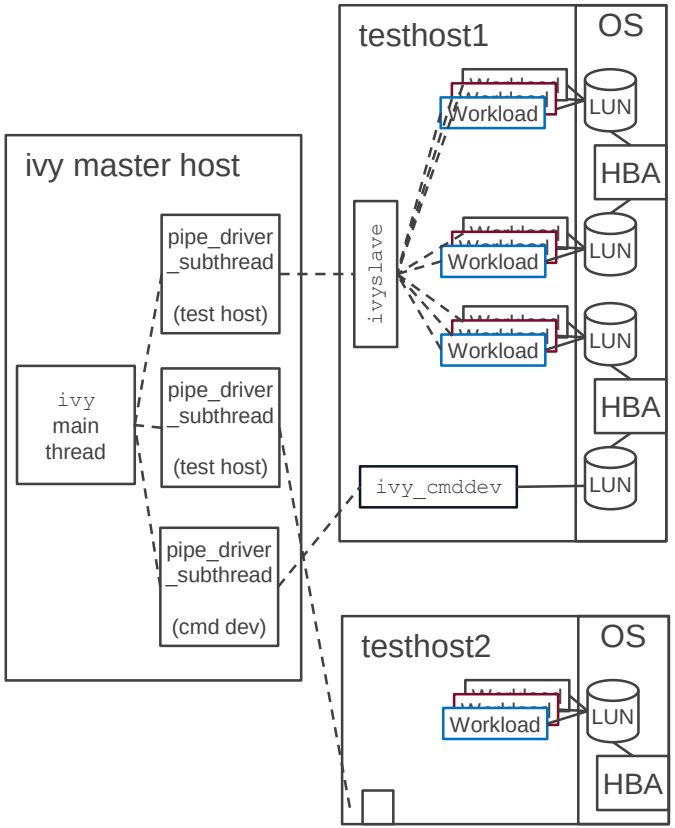
ivy thread interlock protocol

For
1) Users looking to understand how ivy works to better use it
2) Users examining ivy interlock latency measurement csv files,
3) ivy development contributors

Allart Ian Vogelesang

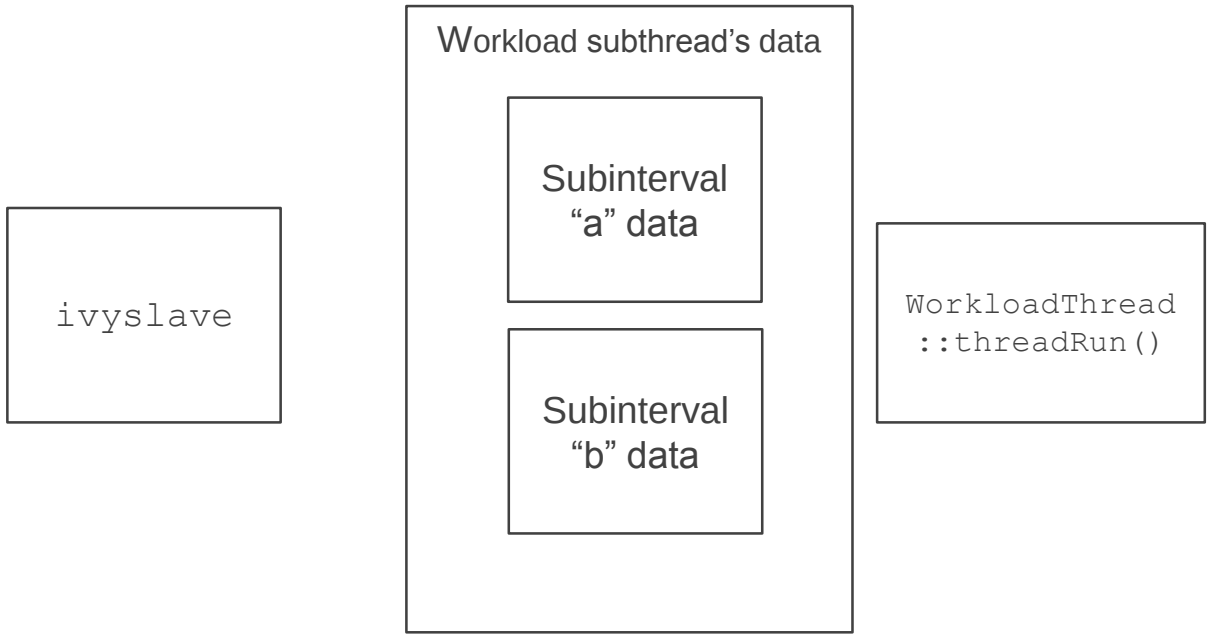
June 11, 2018

Four levels of threads, plus one more running ssh



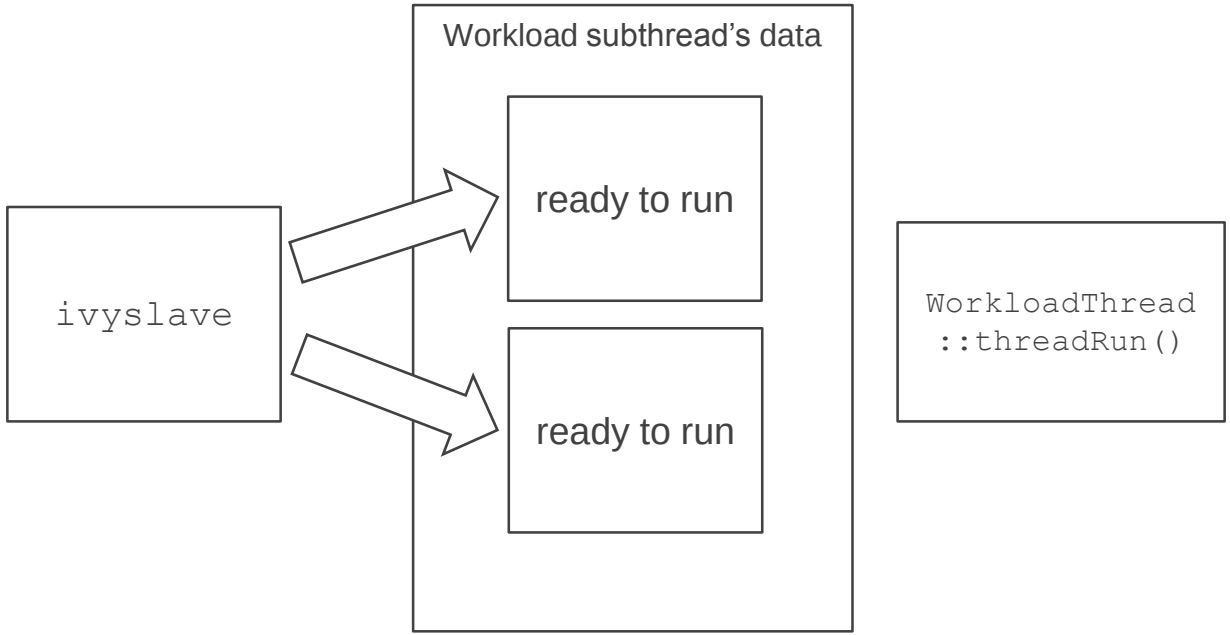
- The ivy main thread creates a subthread running `pipe_driver_subthread::threadRun()` to operate each test host.
  - It is possible to use one of the test hosts as the ivy master host.
- The pipe driver subthread in turn creates a subthread and connects "pipes" to stdin & stdout of the new subthread. Then the new subthread (not shown in diagram) runs ssh to fire up the remote executable. pipe_driver_subthread talks to the remote via the pipes.
- There are two types of remote host executable that pipe driver subthread can operate
  - The `ivyslave` executable operates workload subthreads, each of which drives I/O to a LUN using an Asynchronous I/O (AIO) context enabling a single workload thread to issue multi-threaded I/O to its LUN.
  - Multiple workload threads with different names can be layered on each LUN. Workload names must be unique on a particular LUN, but it's normal to use the same workload name on each of a set of LUNs across multiple test hosts.
  - The `ivy_cmddev` executable operates a Hitachi RAID subsystem command device to retrieve configuration data and real-time performance data. `ivy_cmddev` is a proprietary Hitachi internal-lab-use-only tool that is not part of the ivy open source project.

Workload threads flip back and forth between two objects



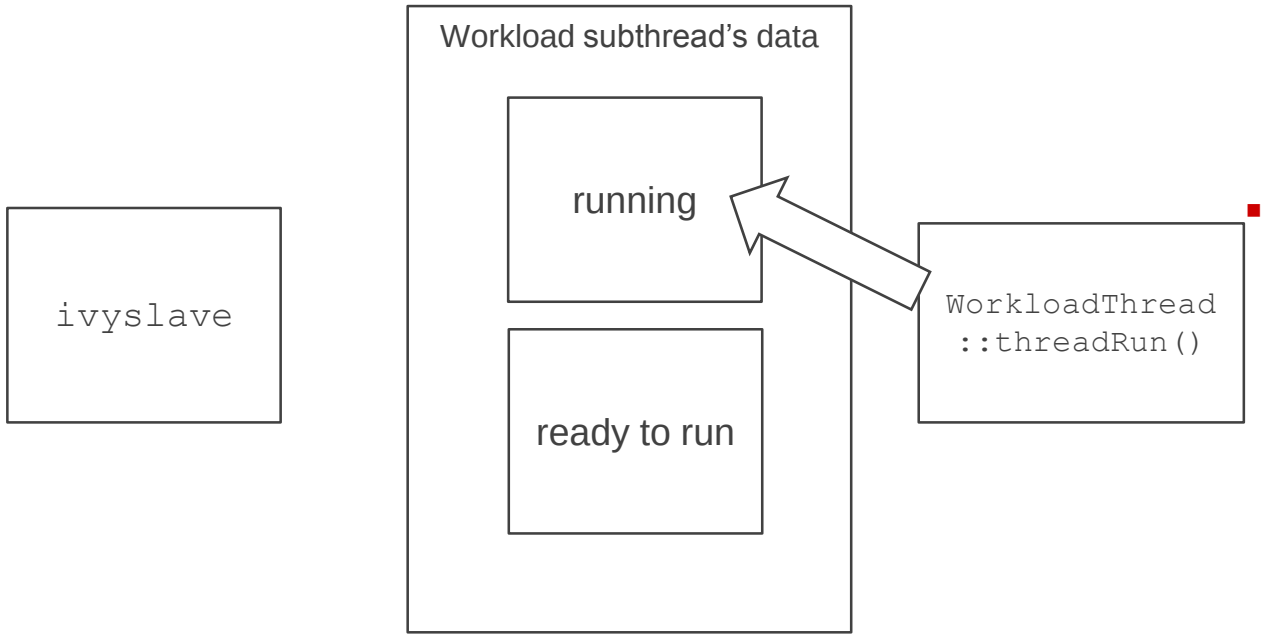
- When a workload thread is running driving I/O, it switches from one set of subinterval input and output data to the other at the end of each subinterval.
- Each of the two sets of subinterval data can be marked either
  - "ready to run"
  - "running"
  - "read to send"
  - "sending"

At time zero before the first subinterval starts running



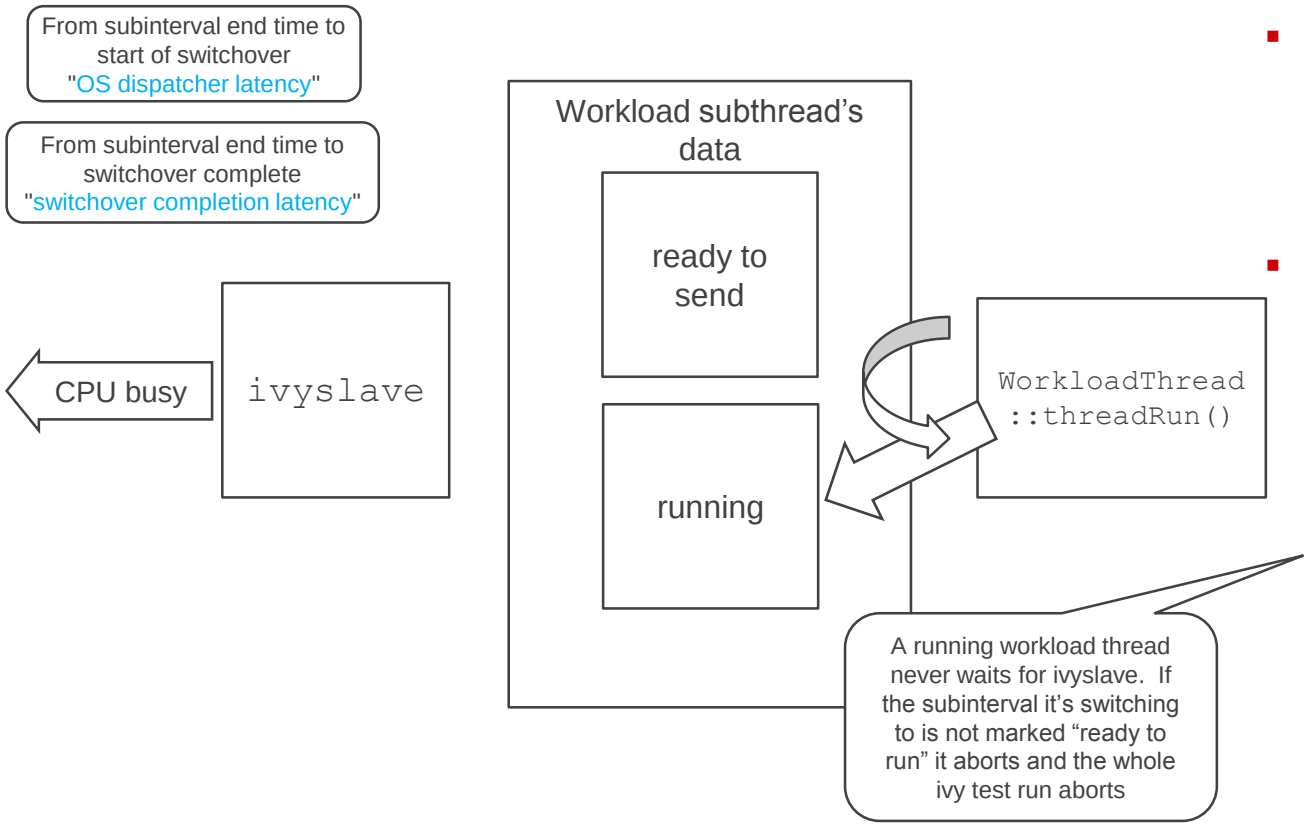
- I/O sequencer input parameters were already set in both subinterval data objects by "create workload" or "edit workload".
  - "edit rollup" at the ivy main thread results in an "edit workload" being issued at the ivyslave level.
  - Both subintervals are marked "ready to run"

During first subinterval



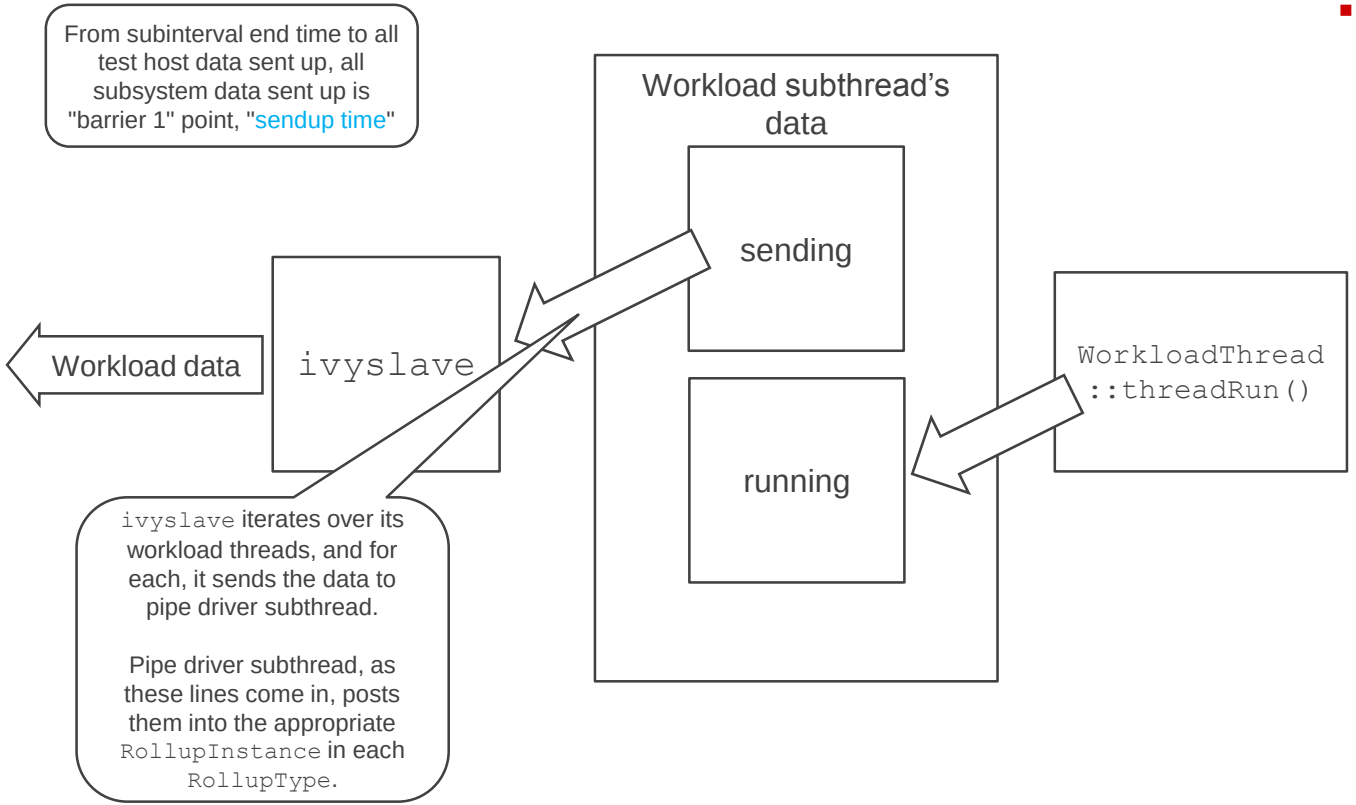
- `WorkloadThread::WorkloadThreadRun()` is running I/O and posting statistics into the subinterval data object.
- `ivyslave` is waiting for the end of the subinterval.

At the end of a subinterval



- `ivyslave` harvests Linux CPU busy data from /proc and sends "CPU" line to pipe_driver_subthread
- Workload thread interrupts driving/harvesting I/O for an instant to mark the subinterval data "ready to send" and then it switches to the other subinterval, **which must already be marked "ready to run"**, and marks it "running" and resumes driving I/O and recording in the new running subinterval

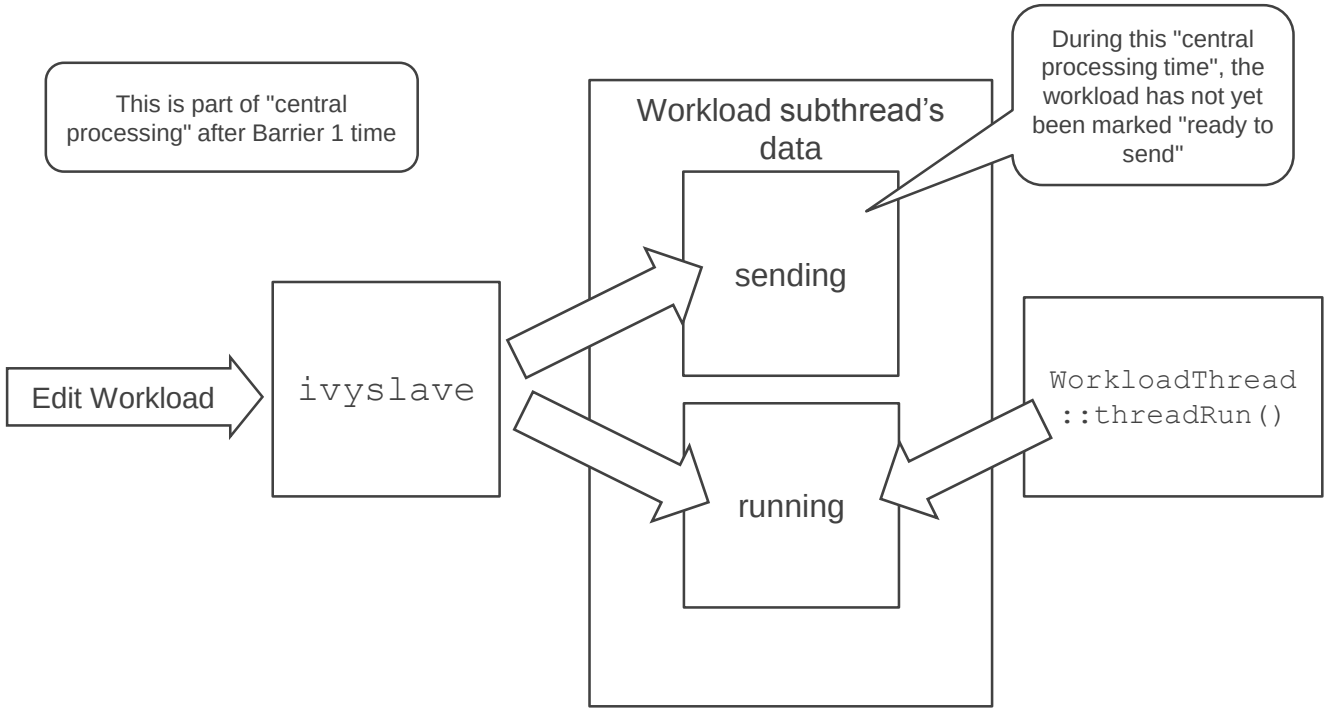
A few milliseconds after the end of a subinterval



- `ivyslave` waits for the `WorkloadThread` to have been dispatched and has marked the just-completed subinterval as "ready to send".
  - If it takes over ¼ of a subinterval for this to happen, ivy aborts the test.
  - This would mean either a workload thread as stopped operating, or that the test host is so overloaded that the workload thread didn't get dispatched for a LONG time.

Optional Dynamic Feedback Control iosequencer update

HITACHI
Inspire the Next



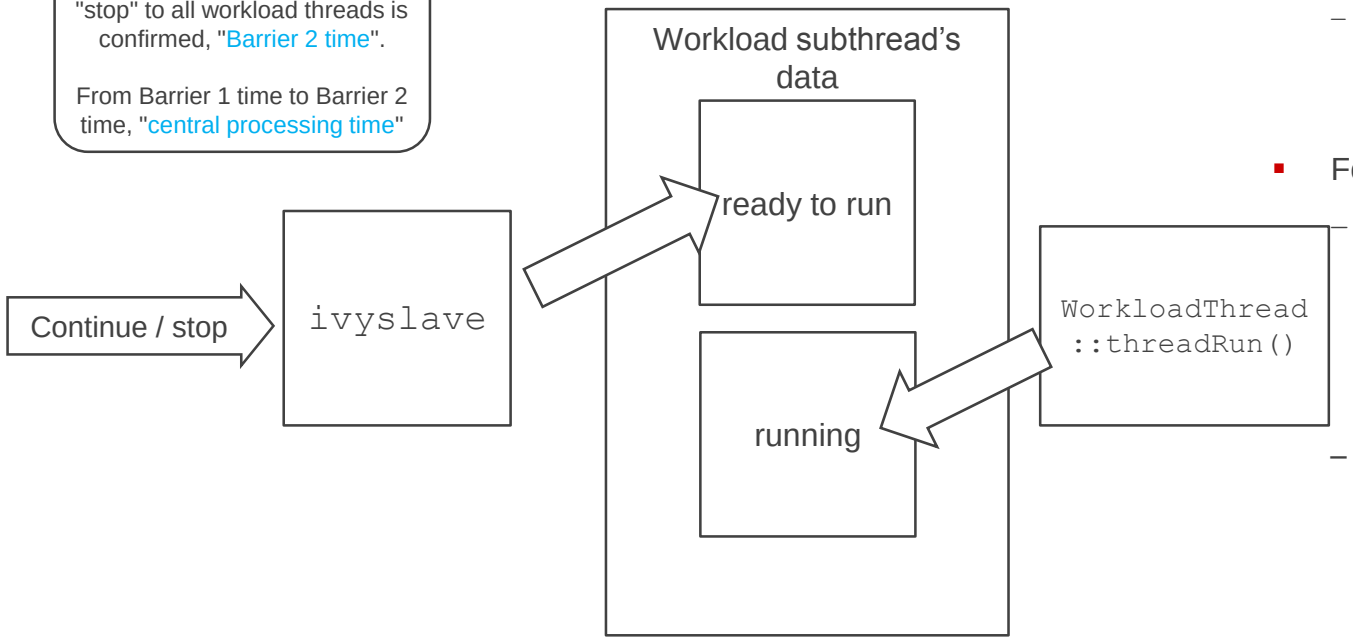
- `ivyslave` posts any iosequencer input parameter updates into BOTH subintervals, *while the workload thread is running*.
- This makes DFC updates calculated from the previous subinterval's data to take effect early in the very next subinterval.
- At present, only the IOPS parameter is being updated, and the current iosequencer types are OK with changing on the fly.

After optional DFC updates, “continue” or “stop”

HITACHI
Inspire the Next

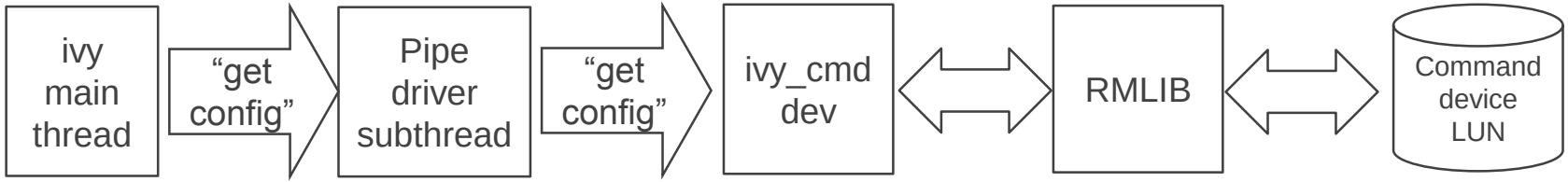
When delivery of "continue" or "stop" to all workload threads is confirmed, "Barrier 2 time".

From Barrier 1 time to Barrier 2 time, "central processing time"



- For "continue"
  - `ivyslave` clears out the old output data, and posts the inactive subinterval as “ready to run”.
- For "stop"
  - `WorkloadThread` stops at the end of the running subinterval, and after first catching any in-flight I/Os at the end of the last subinterval, goes to "waiting for command" state.
  - For "stop", `ivyslave` will still send up the data from the last subinterval at the end of the subinterval.

Command device gather for configuration info

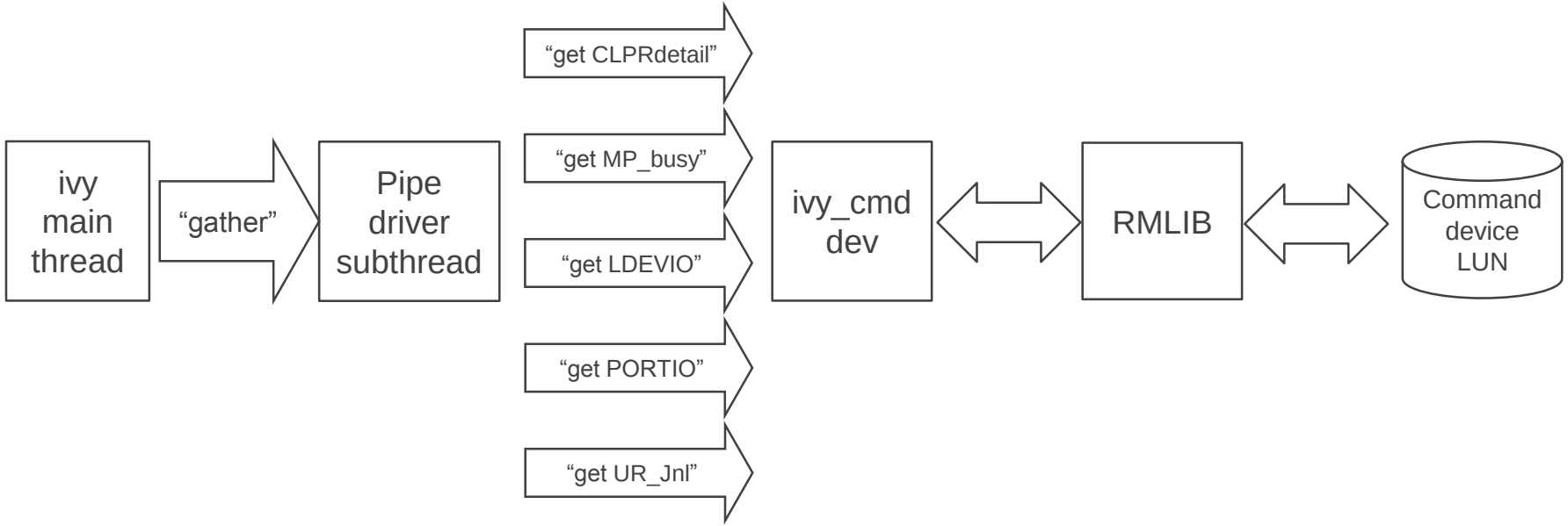


- In response to a “get config” command, `ivy_cmddev` sends back a set of four-tuples:
  - <element type>, <element instance>, <attribute name>, <attribute value>
  - E.g. “LDEV”, “00:FF”, “drive type”, “NFHAE-Q3R2SS”.

This is done once, when ivyslave first starts up

- A subsystem configuration csv file set is printed in a subfolder in the output root folder by `GatherData::print_csv_file_set`(). The subsystem serial number goes into the subfolder name, and then there is a csv file for each element type, with rows by element instance, and columns by attribute name.
- `pipe_driver_subthread` does some post-processing of the configuration data, such as making a list of the Pool Volumes for each Pool ID.
- Then the ivy main thread in `ivy_engine_startup.cpp` also does some post-processing, such as propagating LDEV attributes from subsystem config data to any available test LUNs mapped to this subsystem serial number and LDEV ID.

Command device gather for real time performance data



- For each, `ivy_cmddev` sends back a set of four-tuples: <element type>, <element instance>, <metric name>, <metric value>
- These are stored in the `GatherData` object at the end of a `std::vector` of `GatherData` objects by subinterval.
- A csv file is printed in a subsystem performance subfolder of the “step” folder. There is a subfolder for each element type, and a csv file for each element instance, with one row for each subinterval, and one column for each metric name.

Filtering real time performance data by Rollup Instance

HITACHI
Inspire the Next

- For certain metrics defined in `subsystem_summary_metrics` in `ivy_engine.h`, the real time performance data is filtered by rollup instance according to a configuration filter built for each rollup instance before a test step starts running.
- This configuration filter is built from the attributes of the LUNs underlying the workloads comprising the `RollupInstance`, and in turn from the attributes of the LDEVs behind the LUNs.
- This includes both “direct” attributes such as Pool_ID, as well as derived indirect metrics like PG, which for DP-Vols is derived by looking at the PG attributes of each Pool Volume in the DP Pool.
- The “focus metric” for the “measure” feature or for the PID loop (DFC) if it’s a subsystem metric must be one of the metrics in `subsystem_summary_metrics`, because the measure feature and DFC are done at the granularity of the `RollupInstance`, so the metric has to be filtered by `RollupInstance`.
- The real time subsystem data is also filtered for those “non participating” subsystem elements that are not represented in any `RollupInstance`. This shows up in columns in ivy csv files. This is so you can see if, for example, there was any significant amount of MP core % busy for those MPUs that are theoretically not being used. This could also show if there was non-ivy activity going on.

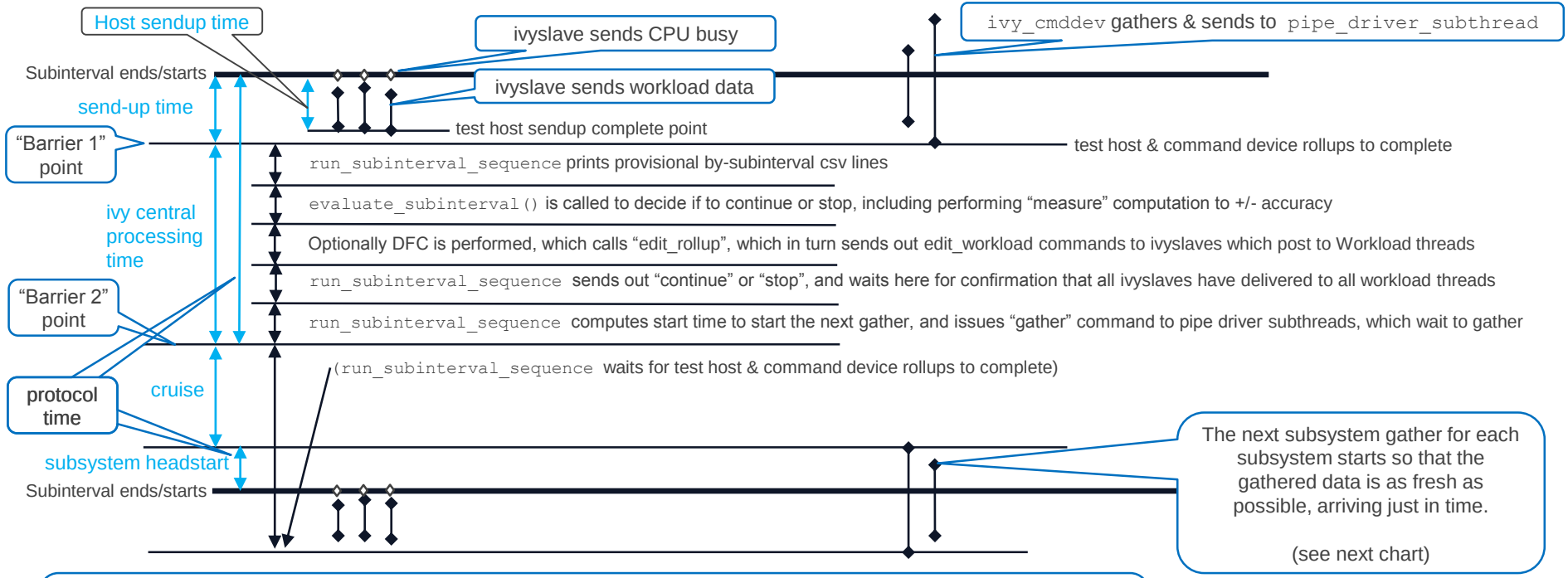
Ivy main thread waits for gathering & rollups to complete

HITACHI
Inspire the Next

- At the end of a subinterval
  - Test host data are rolled up and posted in `RollupInstance` objects.
  - Filtered subsystem data are rolled up and posted into `RollupInstance` objects.
  - (These two kinds of rollup are done independently in parallel.)
- In the main thread interlock timing diagram, you will see where the main ivy thread in `run_subinterval_sequence()`
  1. waits for test host data to rollup. This is the “host rollup complete” point.
  2. Waits for subsystem gathers to complete, if they are late. This is called “Barrier 1” on the diagram.
- This can happen in a few milliseconds to a few hundred ms after the end of a subinterval, but erratic subsystem gather times can slow things furthe.

ivy thread interlock protocol diagram (not to time scale)

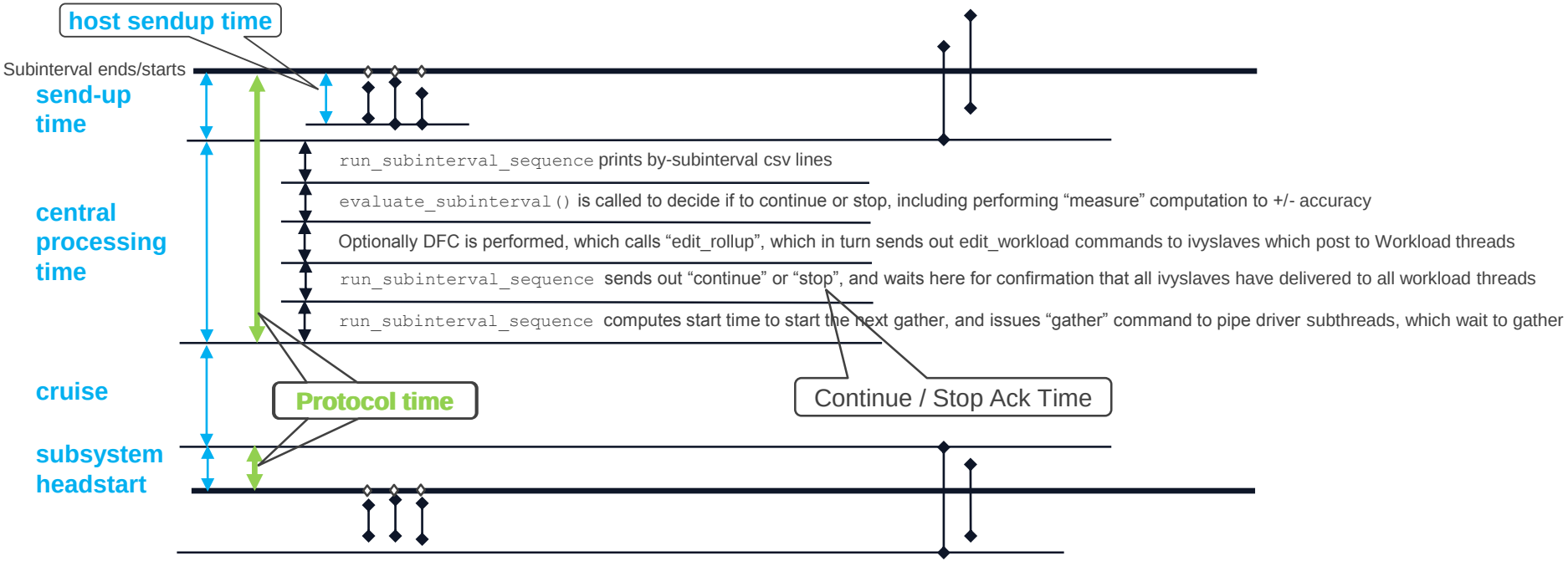
HITACHI
Inspire the Next



The "cruise time" after Barrier 2 and before the first subsystem gather starts and before the end of the subinterval is the amount of time by which the length of the subinterval could be reduced without the interlock protocol breaking.

Of course you want to have a some spare time in case of any delays.

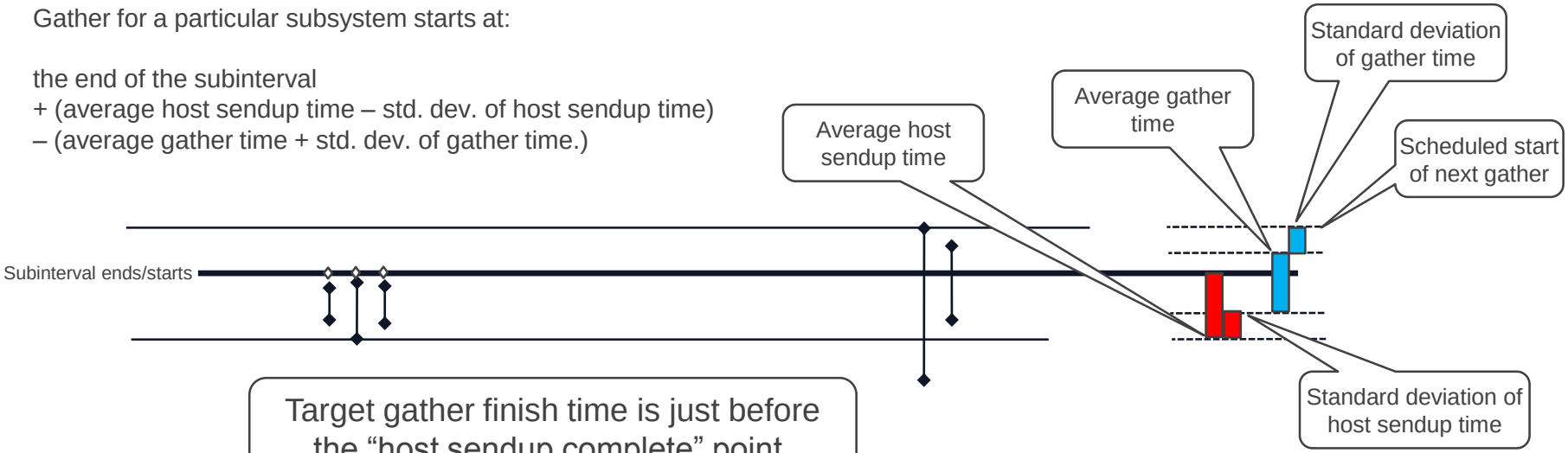
Ivy interlock latency csv file terminology



Scheduling just-in-time subsystem gathers

Gather for a particular subsystem starts at:

the end of the subinterval
+ (average host sendup time – std. dev. of host sendup time)
– (average gather time + std. dev. of gather time.)



Target gather finish time is just before the “host sendup complete” point.

Workload thread interlock latency metrics

HITACHI
Inspire the Next

- OS dispatcher latency
  - The time from the end of the subinterval until when WorkloadThread starts running
- Lock acquisition delay
  - The time it took to get the lock to talk to ivyslave
- Switchover complete time
  - The time from the end of the subinterval until when the switchover to the next subinterval is complete and we go back to driving I/O. OS dispatcher latency and lock acquisition are part of this.
- “Distribution over” metrics
  - The idea is to show if some threads are showing more responsiveness (e.g. fixed dispatching order) or if workload thread latencies are different across test hosts.

Using ivy interlock protocol latencies csv data

HITACHI
Inspire the Next

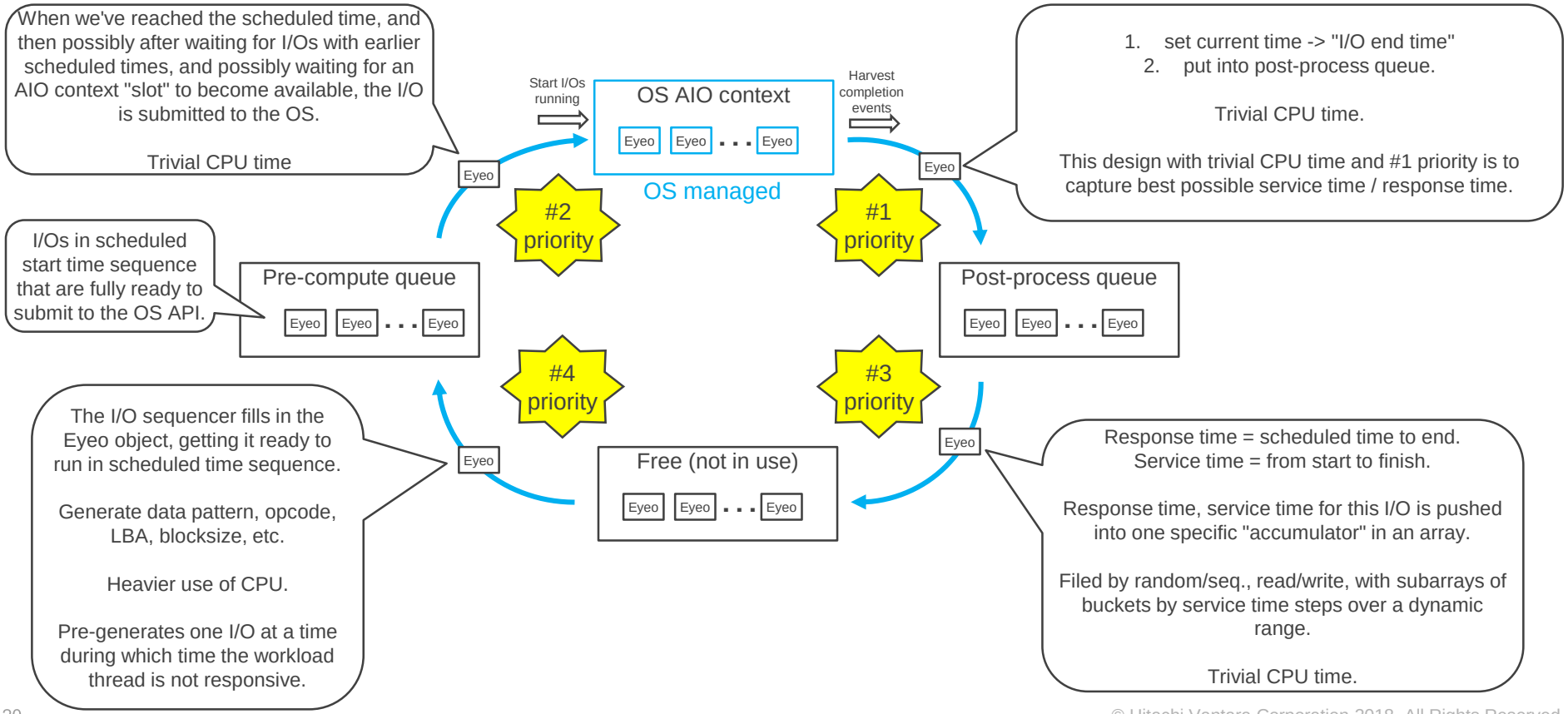
- First, after reading this material, you will be able to understand what the metric names in the csv files mean.
- The ivy interlock protocol latency data documents to what extent the operation of the ivy measurement mechanism itself is operating on a timely basis, or if interlock protocol latencies have become excessive.
- *If latencies have become excessive, then the measurements being made are not as "solid".*
- One example might be if a particular test host had lots of LUNs with lots of layered workloads on each LUN, and if thousands of such workload threads were running, it's possible that dispatcher latency may become big.
- Another example we have seen already is if the command device is on a very heavily used port operating at maximum MB/s with a large blocksize and a deep queue, subsystem gather times suddenly take seconds instead of milliseconds.
- We may in future define “limits” beyond which the test result will be invalidated.

HITACHI
Inspire the Next

Appendix: more detail

Life cycle of an ivy I/O

HITACHI
Inspire the Next



The ivy AIO engine human-friendly description

HITACHI
Inspire the Next

- "LOOP" means come back to here
  - Get "now" timestamp (nanosecond resolution)
  1. If there are pending I/O completion events, harvest a batch, timestamp them, and put in post-process queue. LOOP
  2. If it's time to submit the next I/O(s) in the precompute queue, and we have one or more available AIO slots, submit a batch of I/Os. LOOP
  3. If the post-process queue is not empty, take one I/O out, and post the results into a single selected bucket out of an array. Put Eyeo on the free queue. LOOP
     - There is "service time bucket spectrum" with 0.1 ms wide buckets up to 1.0 ms service time, then 1 ms wide up to 10 ms service time, etc. up to increments of seconds that can occur with error recovery delays. This gives good coverage for both SSD and HDD scenarios, and makes sporadic very high service time events clearly visible.
     - You get 4 sets of these service time bucket spectra - random read, random write, sequential read, and sequential write. Then there are virtual bucket spectra for "random", "overall", etc. This.
  4. If there is room in the precompute queue, and the scheduled time of the most recently added I/O is less than ¼ second in the future, take an Eyeo off the free queue, apply the I/O sequencer to generate a new precomputed I/O. LOOP
  5. Wait until either an I/O completion event arrives (LOOP), or it's time to drive the next I/O and we are not yet at max queue depth (LOOP), or if the test is over (exit).

The way it actually works

HITACHI
Inspire the Next

- It's all about figuring out how long to wait.
- That's because the highest priority thing is harvesting I/O completion events, and when you go to read them, you say how long you want it to wait before timing out if there are no pending I/O completion events.
- If there are I/Os in the post-process queue, we won't wait.
- If there is room in the pre-compute queue, and the most recent previously generated I/O is less than ¼ second into the future, we won't wait.
- If it's time to submit the next I/O, and we have an AIO slot, won't wait.
- Otherwise the wait time is until the end of the test if we are already at max AIO slot count, or if we have an available AIO slot, the wait time is until the scheduled time of the next I/O.

Communicating with `WorkloadThread`

HITACHI
Inspire the Next

- Each WorkloadThread has a mutex & condition variable used to synchronize access to:
  - thread state { waiting_for_command, running, stopping, died, exited_normally }
  - "command posted" flag
  - command { start, keep_going, stop, die }
  - "dying_words", a string filled in by any "explode and die" error handling
  - Each of the two subintervals – states { ready_to_run, running, ready_to_send, sending, stop }
- ivyslave sets the command, turns on "command posted".
- WorkloadThread waits for command posted, then executes command and turns off flag.
  - When running, WorkloadThread briefly gets the lock when switching to the next subinterval.
- If an error condition caused WorkloadThread to exit, ivyslave can look at "dying_words" to see what happened.

WorkloadThread states

HITACHI
Inspire the Next

- Waiting for command
  - Accepts "start", "die"
- Running
  - Accepts "keep_going", "stop", and "die"
  - When a running WorkloadThread reaches the end of a subinterval, it must see both
    a. Either "keep_going" or "stop" had been posted, and
    b. The "other" subinterval that we are switching into is marked "ready to run"

How to operate ivyslave

HITACHI
Inspire the Next

- "pipe_driver_subthread" on the main host, ivyslave on the remote test host speak to each other over the "pipe" in entire lines ending in a "newline" character.

- pipe_driver_subthread says "command" and ivyslave says "response", which is often simply "OK".

- The same pipe_driver_subthread mechanism operates both remote ivyslave executables and remote ivy_cmddev executables. However, the command processing sections are separate for the two applications. What's different is the names of the commands used, and what to do when you get them. But all the minutia of operating the link is common.

- `ivy_cmddev` when it first starts up says the identical "Hello, whirrled!" thing giving what this host thinks its hostname is, regardless of the path through DNS, aliases, hard coded IP address, etc. Once it's up and running and ready to receive and execute commands, there are separate code sections to process commands for the two applications.

ivyslave commands

HITACHI
Inspire the Next

- "send LUN header", "send LUN"
  - Ivyslave says "<eof>" to "send LUN" to indicate there are no more LUNs
- "[CreateWorkload]", "[DeleteWorkload]", "[EditWorkload]" (not showing the parameter values that come with the command.)
  - Answer: "OK", go back to waiting for command.
- "get subinterval result"
  - Waits until the end of the subinterval, then ivyslave records the test host CPU % busy, and says the CPU line right away.
  - Then it iterates over all the workload threads, interlocking with them waiting for the data to be ready to send, and sends a detail line for each including both a serialized IosequencerInput object and a SubintervalOutput object from the subinterval that just ended.
  - There's a few more lines you get too, not shown, for sequential fill status, and for interlock protocol latencies measured by the workload thread.
- "Go!<5,0>", "continue", "cooldown", "stop"
  - For Go!, the ivytime number in seconds and nanoseconds is the subinterval_seconds parameter.
  - Answer: "OK", which confirms delivery to all WorkloadThreads, marking the "other" subinterval as "ready to run", except for "stop" which tells the workload thread at subinterval switchover to not run another subinterval, but instead run "catch in-flight I/Os", and then go back to waiting for command state.
- "[Die, Earthling!]"
  - Answer: "[what?]", followed by a line for each thread with its "dying_words", if it had any and regardless of whether it exited normally or no. Then ivyslave exits.

ivy_cmddev commands

HITACHI
Inspire the Next

- "get config"
  - We say "get config" to ivy_cmddev
  - Config data comes in 4-tuples <element type> <element instance> <attribute name> <attribute value>
  - We do post-processing of received config data to connect indirect attributes like making a list of pool vols for each pool.
- "gather"
  - We wait until the specified gather start time calculated by the master thread for each subsystem in "run_subinterval_sequence"
  - We say "get CLPRdetail" then "get MP_busy", "get LDEVIO", "get PORTIO", "get UR_Jnl " to ivy_cmddev.
    - Each of these is individually timed, and these times are printed in the interlock_latencies csv file for the subsystem.
  - Performance data also comes in 4-tuples <element type> <element instance> <attribute name> <attribute value>
  - We do some post-processing of the performance data gather, including transforming counter value deltas over successive gathers into rates. For better or worse, the idea was to do this work on the ivy master host to put the minimum burden on test hosts. That does clutter ivy up with some stuff that is peculiar to Hitachi, that is not usable by open source users of ivy, outside Hitachi internal labs.
- “die”

HITACHI

Inspire the Next