Experiment: Effectiveness of Workqueues

The RDMA interface in the Linux kernel added a new "done" mechanism some time between kernel version 4.3 and 4.8. This new mechanism is in parallel with the older mechanism, but there is an indication that the older API will eventually be deprecated. So, ticket https://jira.hpdd.intel.com/browse/LU-8875 was opened by this author to move the o2iblnd module to the new mechanism before the deprecation occurs. As a part of the new done mechanism was introduced three runtime models:

- Direct callback
- Callback via IRQ Poll SoftIRQ
- Callback via WorkQueue

When I did the existing patches to LU-8875 (not yet landed), I chose to use the SoftIRQ model. I am now questioning if that was the correct approach.

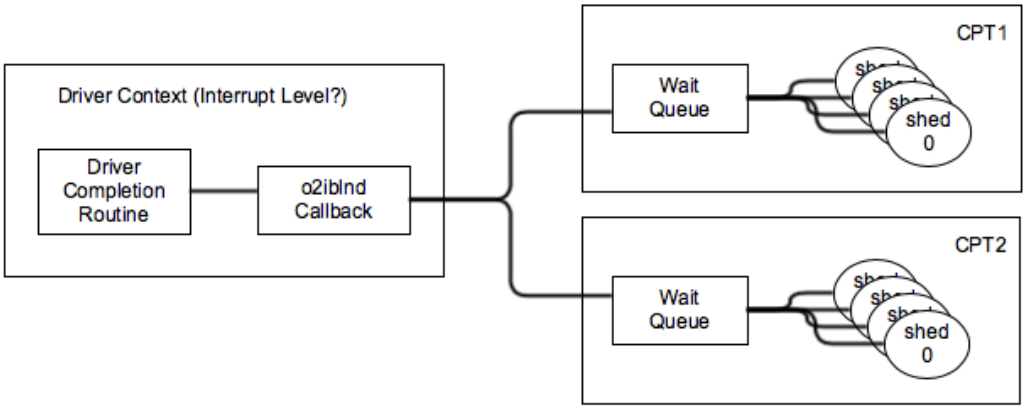
This document looks at the model we use today in o2iblnd, how the three new RDMA models work and how they are different, and looks at a strategy of how we can choose which one the patches LU-8875 should deploy.

Background: o2iblnd Today

With the RDMA runtime model being used by o2iblnd today, we have the fabric driver directly call our o2iblnd callback when a completion is put on a CQ (Completion Queue). This means that the o2iblnd callback runs in the context of the driver which, presumably is interrupt level. As such, the o2iblnd callback cannot do much work and should complete as soon as possible. The o2iblnd callback is passed the CQ identifier which has a completion and does the following:

- By design, the CQ notifier is disabled for the connection before the o2iblnd callback is invoked. We can get callbacks for other CQs, but not for the one we are being notified about. This promotes the batching of completions as we do not re-arm the notification system until we have processed all completion items on the CQ.
- Note: our current design assigns one CQ to each QP (queue pair) which in turn, represents a single peer connection.
- The connection which owns the CQ is checked to see if it should be process by a specific CPU Partition (CPT). If not, we treat the entire set of cores as a single CPT.
- Each CPT has a unique set of scheduler threads who all are waiting on the same CPT-based wait queue. The callback puts the connection identifier for the CQ onto the wait queue for the chosen CPT.
- Callback returns back to driver notifier.

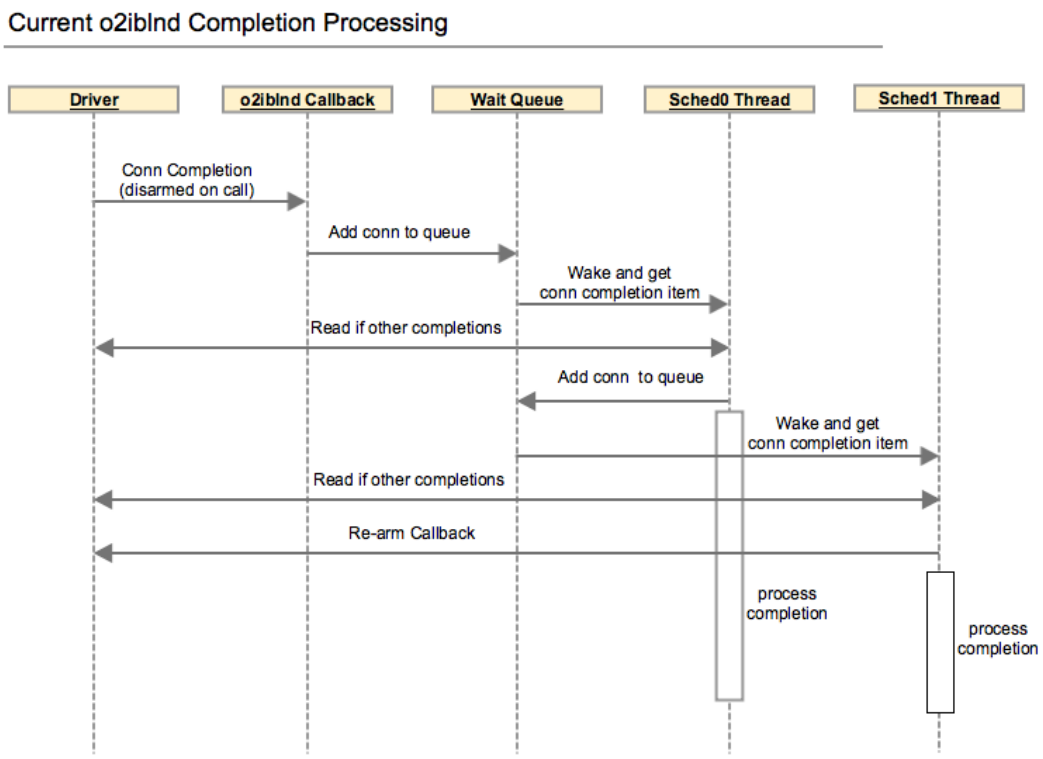
The digram below shows these steps:



So the scheduling of the completion processing is done by the o2iblnd callback running in the driver's context and is picked up by a scheduling thread running in a specific CPT. The scheduling threads do the following:

- Grab one completion work item off the connection's CQ
- Check if there are any more items on the that CQ. If so, put another connection identifier on the wait queue so another scheduler thread can start processing the next completion item for this connection,
- If there are no more items on the connection's CQ, re-enble the notifier for this CQ.
- Process the completion item.
- Go back to wait on the wait queue (waking up right away if something is there).

To demonstrate the sequence being following, let's assume two items are put onto a connection's CQ. The following diagram shows how these two items get scheduled and processed:



The whole point of this design is to maximize parallelism for processing completions. There are pros and cons to this approach:

Pros:

- Parallelism scales with core density.
- Allows lower latency when many completions are scheduled at same time.
- Makes best use of highly dense but slow cores like Xeon Phi.
- Gives better performance than what one core can achieve by itself (i.e. does not limit bandwidth to what a single core can do).

Cons:

- Causes a lot of context switching. Many in the networking world prefer to do most/all completion processing on a single core to avoid context switching. This is considered "more efficient".
- The scheduler threads need to have locks to prevent collisions due to the high degree of parallelism.
- All completion processing being done by threads rather than more efficient tasklets.

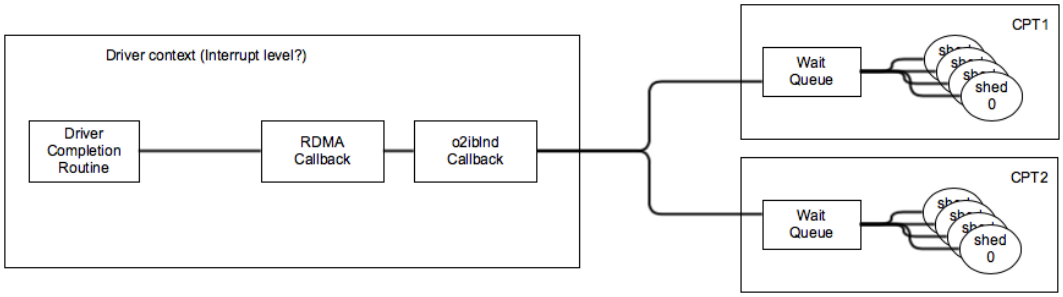
The debate with regards to the o2iblnd runtime seems to come down to this: Do we go with a high level of parallelization at the cost of expensive context switching, or do we focus on efficiency of minimizing context switching with the risk of hitting a bandwidth ceiling associated with core speed? I do not believe there is an easy or quick answer without experimentation.

With the addition of 3 new models to the RDMA interface via the new done mechanism, we have an opportunity to do such experiments. The following sections looks at the three new models we should be trying out.

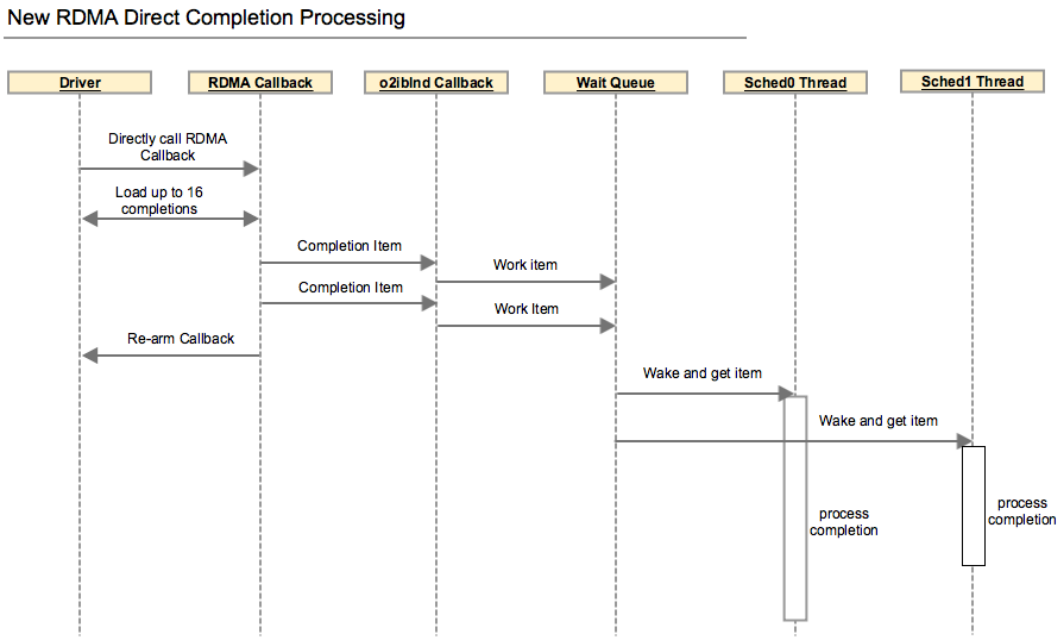
Change to the RDMA Interface

Callback Method: Direct

The first of the three new runtimes is very similar to what we are currently using. The driver notifier/completion routine calls an RDMA callback rather than directly calling our o2iblnd callback. This RDMA callback does some extra housekeeping so we don't have to in our own callback (I'll get into this later on). So, the runtime diagram for the direct method becomes:



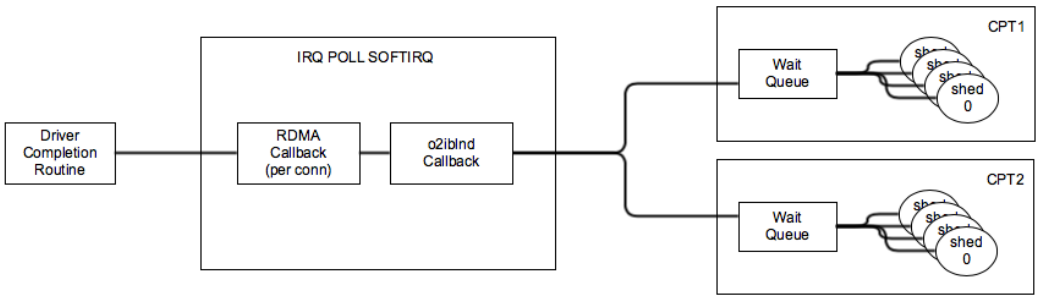
It does all the same things as before. The big difference is the extra work being done by the RDMA callback and what it passes to the o2iblnd callback. Where the previous o2iblnd callback just got the CQ as a parameter, the callback now gets the actual work item which was on the CQ. The o2iblnd callback then puts this work item on the wait queue rather than a connection identifier. The sequence diagram below shows how this works when two completion items appear on a CQ at the same time:



The RDMA callback pre-loads up to 16 completion items off the CQ at a time and then loops over these calling the o2iblnd callback. As before, the o2iblnd callback determines what CPT should be processing this completion and puts a work item on the wait queue for that CPT. Unlike before, the work item contains the completion items that was pre-loaded by the RDMA callback rather than a connection identifier. The scheduler threads wake up and process the given work item calling the appropriate completion routines. The scheduler threads no longer re-arm the notifier when the CQ is empty. That is done by the RDMA callback. This simplifies the processing done in the scheduler threads. Because of this simplification as well as pre-loading batches of completion items, I would expect to see a small performance improvement over what we do today.

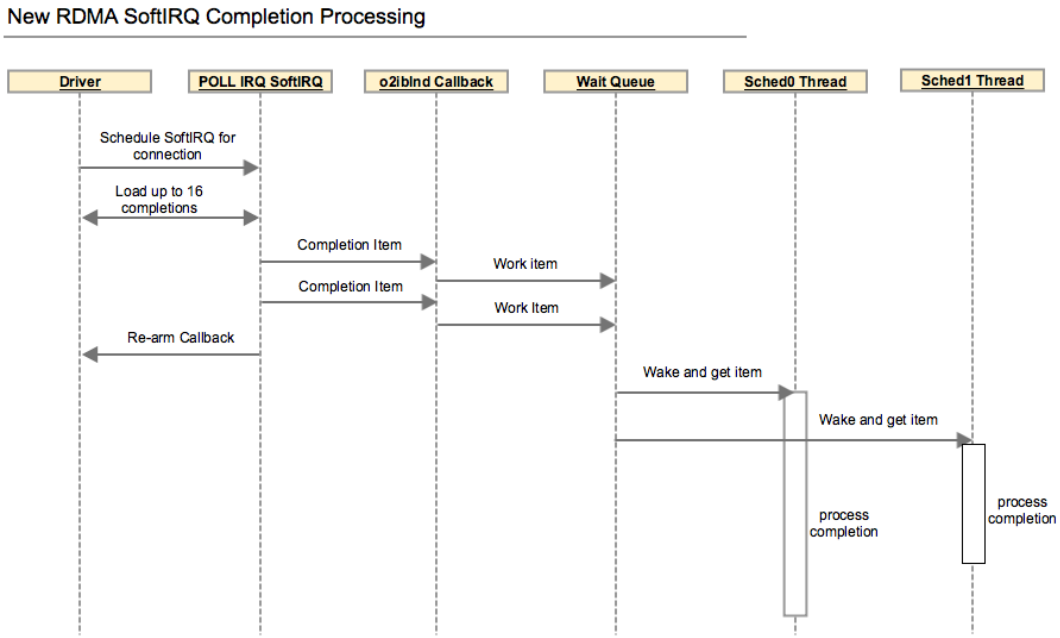
Callback Method: SoftIRQ

The second new runtime method takes a slightly different approach to what we do today. The driver notifier/completion routine schedules the IRQ Poll SoftIRQ to run for the CQ which now has completion items. The SoftIRQ runs in its own context (different from the driver) and executes the same RDMA callback as used in the Direct method above. So, the main difference between the SoftIRQ method and Direct method is the context the RDMA callback and o2iblnd callback run. The runtime diagram then becomes:



SoftIRQ's are restricted in that a given poll item can only be running on one core at any given moment. The scheduling of the completion items for a given CQ are serialized. However, each CQ is viewed as a unique poll item so you can have two CQ completion items being scheduled in parallel (two different cores).

The sequence diagram below for SoftIRQ is the same as for the Direct method and shows the processing of two items:

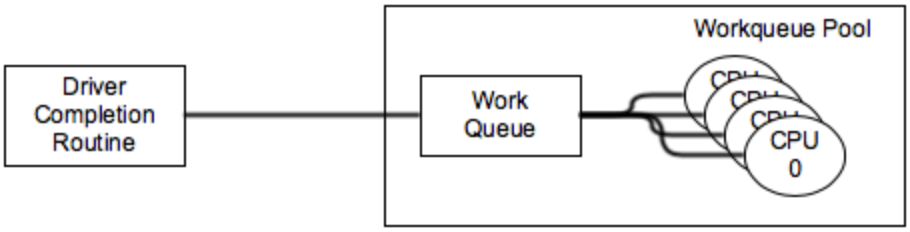


This is the approach which was coded in the current patches for LU-8875. I would expect the performance to be the same as Direct and what we do today, or perhaps a bit more latency due to the extra context switching to use SoftIRQ. We need to test this to see.

Callback Method: Workqueue

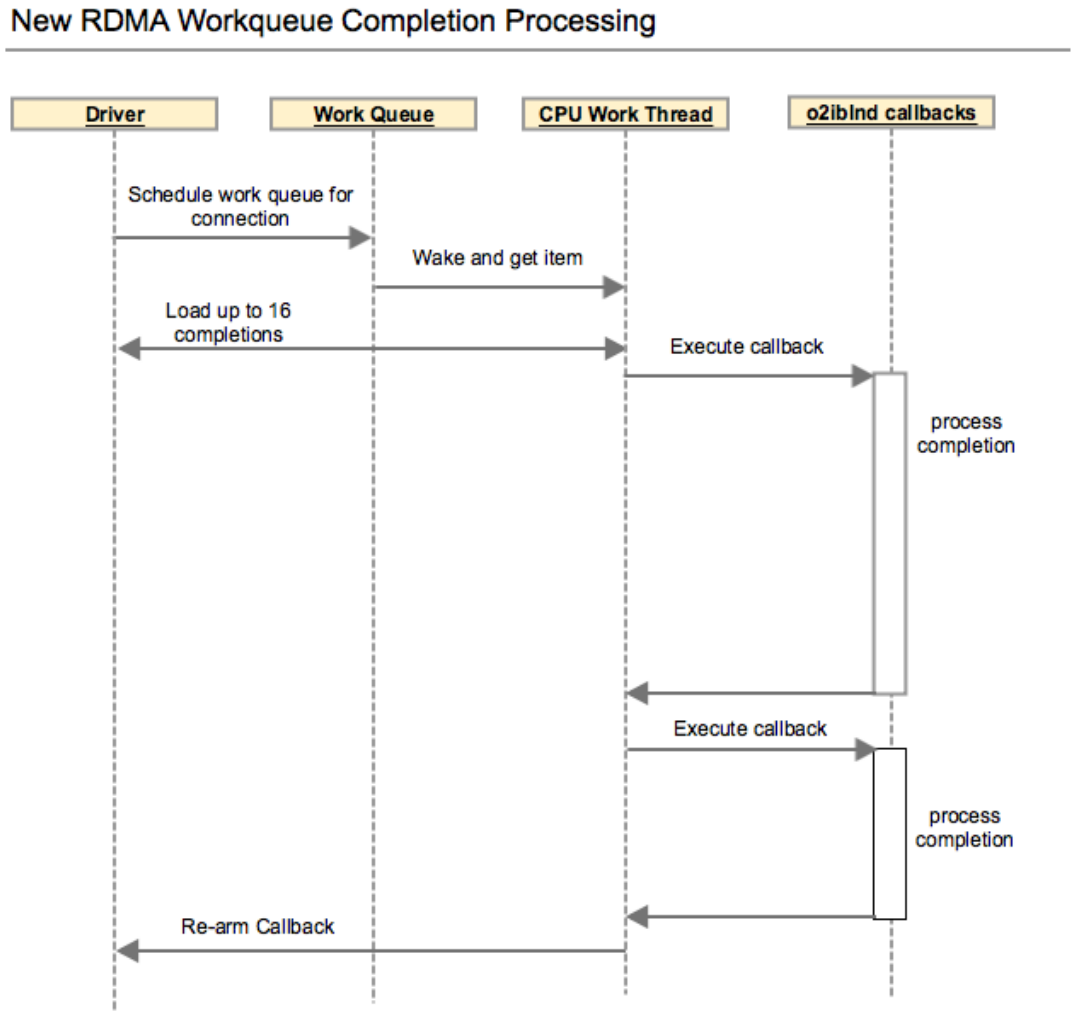
The third new runtime method is very different than all the others. It makes use of the Linux kernel Workqueue system. The Workqueues are a set of configured queues you can schedule work on which is picked up by a pool of worker threads who are assigned to the queue. Typically, one worker thread is created for each core in the system and the preference is to use the core which put the work item on the queue be the worker thread which processes the item. The assumption is this reduces context switching.

If we were to change o2iblnd to use the Workqueue method, we could get rid of our own scheduler threads and associated code (there is quite a lot). The Workqueue thread system would replace them entirely. All completion processing would take place in the Workqueue threads. Here, the driver notifier/completion routine puts a work item on the work queue we are using for RDMA processing (all done in the RDMA codebase for us). This triggers one of the core worker threads to wake up and process the item. The runtime diagram becomes:



As can be seen, we have lost the CPT scheduling feature. However, there is a Workqueue attribute which specifies what cores should be used by worker threads. You can set that attribute when creating the work queue. As work queue creation is being done by the RDMA code, we would need to change it to expose the core parameter to o2iblnd. That being said, we may not need to do this. Since the system favors the worker thread running on the core which assigned the work item to the queue. we would expect the NIC is directing interrupts to the best cores and those would then be processing the work items. Only by testing will we know if this is true.

The sequence diagram for two completion items is below:



As with the previous two methods, up to 16 completion items are loaded at at time and then processed in succession. With Workqueues, the loading and processing is done in the context of a single Workqueue worker thread. This ends up serializing the processing of the completions for each CQ.

Argument to Change o2iblnd to Workqueue Interface

I have heard others argued that Workqueues have too much overhead to be used in HPC. But, if we assume we are sticking to the high parallelism model, we are going to be using kernel threads to run in parallel and process completions. We already have this high overhead today with our scheduler threads. I don't believe that Workqueues are going to be any different for overhead. And, the intelligent core scheduling

Workqueues do could turn out to be more efficient than what our scheduling threads do today with CPTs.

Certainly we would get one big advantage from Workqueues: we can delete all scheduler thread code from o2iblnd (which has been the source of some bugs and misunderstandings). Replacing all of that with a standard kernel system would be very advantageous to code comprehension.

If it is believed that avoiding parallelism and minimizing context switches is the better way to go, then I still believe Workqueues will work. Remember that each CQ will generate a work item when completion items are added to it. The work items will trigger all completion items on the CQ to be processed in a single worker thread as a batch. This is different from today where the scheduler threads try to process the completion items on the CQ in parallel. This approach taken by Workqueues will prevent context switches between worker threads for a given CQ and should be more efficient for individual CQs.

If there are many CQs and we find that is triggering many context switches between the worker threads, we can then merge the per-connection CQs into one single o2iblnd CQ. This will serialize all completions under the Workqueue method. If the opposite is true and we want to get back some of the parallelism we have today with our scheduler threads, we can do that by having multiple CQs per connection via the conns_per_peer module parameter.

Overall, I beliefe we can manipulate the Workqueue runtime approach to give us what we want and make it a better overall solution than the scheduler threads we use today or the Direct and SoftIRQ methods.

How to do this

All of this is good theory, but will require experimentation to see what the reality is. I propose the following game plan:

- Produce 3 patches for LU-8875: using Direct, SoftIRQ (done already), and Workqueue methods as presented by the RDMA interface.
- Do performance testing with each patch.
- Pay attention to the results to see if bumping up conns_per_peer may help. This is especially true for Workqueues.
- Analyze what cores are used with the Workqueue method to ensure it makes sense. If not, then generate a new patch to the RDMA codebase which exposes the core attribute and use that when setting up the Workqueue. Change the Workqueue patch to use this attribute to lockdown the cores to match our CPT configuration.

At the end of this, we should be able to select the proper patch for LU-8875.