Photon: Remote Memory Access Middleware for High-Performance Runtime Systems

Ezra Kissel and Martin Swany
Center for Research in Extreme Scale Technologies
Indiana University
{*ezkissel, swany*}*@indiana.edu*

***Abstract*—We introduce the Photon RDMA middleware library that enables consistent remote memory access semantics over a number of network interconnect technologies. A primary goal of Photon is to expose a lightweight and flexible network abstraction that minimizes communication and message handling overheads for high-performance applications and runtime systems, in particular those that require the manipulation of objects within a global address space. Both one-sided and rendezvous communication models are supported and asynchronous network progress is exposed at a fine granularity. Photon implements a novel communication pattern called put-with-completion (PWC) that optimizes a completion notification path with variable size data for realizing active message-driven computation. The results of our performance evaluation show that our PWC model is comparable, and often improves upon, existing one-sided RDMA libraries in message latency and throughput metrics.**

I. Introduction

A number of common design features have emerged as research efforts in high-performance computing (HPC) runtime systems advance toward exascale. Key among these are active messages (AM) [1] and event-driven computation supported by global address spaces (GAS), which facilitate the movement of work to data, exploit communication and computation overlap, and support the use of dataflow-style programming. In part, these techniques promise to enable implementations of the parallel GAS SPMD model to achieve effective scaling of HPC applications on extreme-scale distributed system architectures. A major factor in the success such distributed runtimes is the messaging and event handling performance achievable across current and future network interconnects. More specifically, the design of an integrated network library should strive to minimize the latency for AMs and GAS update events while also maximizing the throughput of data movement between communicating processing elements.

Overheads inherent in event or message-driven computation are partially addressed by Remote Direct Memory Access (RDMA) support such as that found in InfiniBand [2] or Cray Gemini (XK5, XE6) [3] and Aries (XC3, XC40) [4] environments, among others. RDMA exposes *put* (write to memory address) and *get* (read from memory address) semantics for data movement and performs these operations without OS perturbation using a network interface adapter with appropriate on-board logic. To transfer data, an association must be established between the endpoints and the appropriate memory must be registered with the system so that it can be "pinned" with a consistent virtual to physical address mapping. Zero-copy RDMA eliminates local memory-to-memory copying and enables direct data placement in the memory of a remote system, without interrupting the CPU and incurring the overhead of a context switch.

While RDMA may eliminate the system cost of fast data movement, the one-sided nature of put and get operations introduces an additional challenge, namely the handling of completion notifications for synchronization and progress. The completion of RDMA operations is generally communicated by means of polling interconnect-specific completion queues (CQ), the result of which may optionally contain a small amount (4-8 bytes) of "immediate data" in the case of put requests. Received completion events, plus any optional data, must subsequently be interpreted and mapped by the caller to an appropriate action. For example, the event may correspond to the lookup of an outstanding request, or result in the update of some synchronization state. Due to the limited information contained within an event descriptor, significant infrastructure is required to develop higher-level messaging features commonly exposed in existing message passing libraries. We posit that by simply extending the CQ concept of *event*-plus-*data* to support variable size completion identifiers coupled to remote memory access operations, and directly exposing the completion progress through a middleware library, we can provide a powerful new mechanism in the realization of message-driven runtime systems. We call this abstraction put-with-completion (PWC), and its counterpart get-with-completion (GWC).

To evaluate this idea we have developed an RDMA middleware library called Photon. Photon is a major evolution of the Gravel MPI companion library [5] that enables automatic replacement of selected MPI calls with decomposed RDMA equivalents to exploit the potential communication with computation overlap in applications. Photon currently exists as a standalone library and includes the following key features:

- Rendezvous API for RDMA transfers, providing *function separation* in that the handshake protocol is decoupled from the data transfer.
- One-sided RDMA API for PWC and GWC, plus collectives, supporting variable length completion data.

- Network progress exposed at a fine granularity for both local and remote completion events with data.
- Abstracted core API to allow for consistent semantics on top of numerous network "backends." To date, Photon has implementations for InfiniBand, Gemini, Aries, and libfabric [1]. An SMP shared memory backend is in development.

The remainder of this paper describes the development and evaluation of the one-sided Photon PWC and GWC messaging model, henceforth PWC unless explicitly referring to GWC functionality. Section II provides an overview of related work followed by PWC implementation details in Section III. We evaluate the performance and scalability of PWC in Section IV, and Section V briefly surveys our ongoing integrating of PWC into the High Performance Parallex 5 (HPX-5) [6] implementation of the Parallex [7] runtime model. We describe future work and conclude this paper in Section VI.

II. Related Work

Photon is similar in spirit to a number of existing network interface libraries, perhaps most of all to GASNet [8], [9]. The GASNet extended API exposes one-sided RDMA put and get semantics with a number of synchronization mechanisms for non-blocking calls; however, remote notification of direct memory operations at the destination is not exposed. Their core API also directly implements active messages, which provides the ability to execute a handler that may operate on sent message data (and send an optional AM reply) at the destination. What primarily differentiates Photon is that we integrate the notification and completion progress path with the data transfer operation itself, allowing for direct control of the notification event and data during network progress.

The Portals Network Programming Interface [10], [11] has numerous implementations focused on the scalability of Partitioned-GAS (PGAS) programming models on large-scale systems. Completion events for put and get operations, including remote notifications, are exposed via Event Queues but limited to an 8-byte header value. The Portals specification is also clear that performance in terms of scalability is their primary metric of success, and Portals message passing performance is difficult to compare using micro-benchmarks.

The integration of RDMA techniques in MPI is an area of significant research, and emerging one-sided MPI-3.0 [12] implementations such as foMPI [13] in particular. Early usage of InfiniBand in MPI [14], and independent performance studies [15] of RDMA completion signaling provided insight into the communication challenges Photon attempts to address. A recent novel use of remote notification events in foMPI is called *notified access* [16]. This scheme extends the MPI-3.0 one-sided interface with "notified" versions of put and get calls, greatly reducing the synchronization and message matching overheads. Unfortunately, the notified access implementation relies solely on hardware CQ support and is tied to the MPI specification with integer tagging, whereas PWC attempts to generalize the event plus data model as a single abstraction with granular completion data access.

Finally, *active access* [17] has been proposed as a mechanism to merge AM semantics with RDMA operations at a hardware level. Proposed extensions to an endpoint's IOMMU would allow AMs to invoke a user-defined CPU handler directly, bypassing OS interactions through the direct passing of memory requests. We envision that PWC could leverage work in this direction as hardware support matures in support of active RDMA messaging handling.

III. Design and Implementation

At a fundamental level, a Photon PWC operation is defined as on operation (put or get) on remote memory accompanied by a user-defined local and remote completion identifier. The local and remote identifier data is subsequently *probed* for and acted upon at both the source and destination of the data transfer, respectively. The user-defined completion data remains opaque to Photon with the goal being to transfer the data from source to destination with minimal latency and with as few overheads as possible. The act of probing for completion events also serves to progress the network.

As noted above, most RDMA interconnect implementations already provide CQs necessary for progressing the underlying network, and when polled, they return short identifiers that were set during the initiating RDMA calls via event descriptors. These completion events are typically used internally by a higher level network library to implement the desired API behavior, and the same holds true for Photon. The insight with PWC is that the event notification with data mechanism may be generalized and exposed as an RDMA primitive, enabling a thin and lightweight abstraction over a number of native RDMA network interfaces. An inherent advantage of this approach is that it allows a runtime system to immediately act upon probed completion data without invoking any external handler routines, eliminating a level of indirection while also simultaneously progressing the network.

The transmission of completion data is facilitated through the use of internally allocated Photon bounded buffers called "ledgers." A number of Photon ledgers are allocated at library initialization time and they may vary in size and purpose based on configuration parameters. A PWC ledger may come in *packed* and *slotted* variations, but both are used in the exchange and accounting of remote completion data. Before we describe the operation of PWC and the use of ledgers, a number of Photon features must first be introduced.[2]

A. Network backends

Internally, Photon defines an interface to common data movement and management operations to ensure the successful integration of multiple network interconnect interfaces.

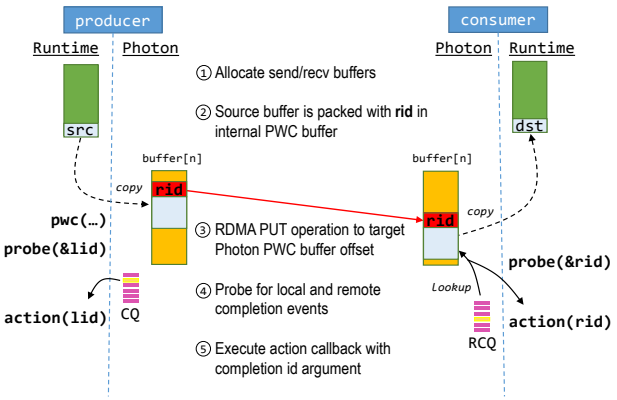
[1] OFI-WG libfabric: http://ofiwg.github.io/libfabric/

[2] While Photon API implementation details are omitted in the interest of space, we will mention that with the exception of explicit *wait* calls to support rendezvous RDMA, the Photon API is fully asynchronous (non-blocking). Photon is also designed to be fully thread safe and reentrant, making it suitable for runtimes that support lightweight threading.

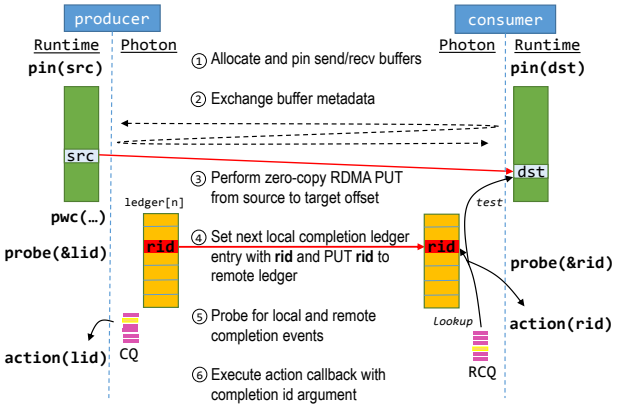
Photon calls these "backends" and each backend must, at a minimum, implement a buffer management interface, put and get remote memory access operations, and methods for querying local and remote completion events. Additionally, each backend should provide methods to inform Photon about the state of its available transmit and receive queues. If the network is saturated, new operations must either be queued or else the runtime must be notified of a resource limitation and back-off accordingly. The network backend is also responsible for registering the internal ledger buffers in use by Photon before any RDMA operations take place.

B. Buffers and handshake

The Photon API exports a buffer registration interface that allows runtimes to register, or "pin", externally allocated buffers for use with direct memory operations over the network. Photon passes these registration calls to the necessary configured backend(s) while creating and saving a buffer descriptor with reference counting for future lookups. In the process of registering memory regions, the network backend generates unique memory descriptor handles, or private access keys, that are required by the initiator of a remote memory access call. In the rendezvous protocol, dedicated "info" ledgers are used to exchange this registered memory metadata, along with the specifics of the RDMA request such as transfer size and address offset. For one-sided operations, this handshake is required before transfers can be made, but the same rendezvous protocol may be used during an initialization phase. For example, a runtime system may make use of a global heap that is registered with Photon and the exchange made once prior to any subsequent RDMA operations within the registered address space.



(a) Packed put-with-completion



(b) Ledger put-with-completion

Fig. 1. Photon PWC operations, highlighting the difference between packed and ledger modes.

C. Request tracking

Request descriptors in Photon are used to track requests made to the non-blocking API with the subsequent backend events generated on completion of a given RDMA operation. Photon requests may also be queued if insufficient ledger or backend resources are available. The default behavior is to provide strict ordering of requests, meaning any queued requests will be processed before any newly posted operations, but the ordering policy is configurable if the runtime takes steps to avoid potential starvation. Request descriptors are maintained on a per-peer basis as multi-level, directly indexed circular buffers, and Photon will scale with the number of queued requests without significant slowdown. For PWC operations in particular, the request descriptors keep a pointer to the local completion data to be returned via a probe call when the associated RDMA operation has completed.

D. Flow control

The presence of bounded ledger resources necessitates a level of synchronization between communicating peers to indicate when the next available offset or completion slot is available. Ledger space is partitioned on a per-peer basis and Photon maintains a 1:1 mapping between local and remote ledger progress, using monotonically increasing counters. As ledger space fills beyond a threshold, the sending peer will perform an eager get operation on the remote progress counter, updating its local progress counter equivalent. New requests to a peer are queued until additional *credits* become available as subsequent remote progress get updates complete. Ledger sizes are configurable at library initialization time based on the needs of the calling runtime, although we are currently investigating dynamically sized ledgers to address systems where minimizing the networking memory footprint is important.

E. Put with completion

With the preliminaries covered, we may now describe the details of PWC operation as shown in Figures 1(a) and 1(b). Two PWC modes are supported: a) *packed*, and b) *ledger* or 2-put PWC. The motivation behind packed PWC is simply that a single put is less costly than two puts and imposes less contention on the network backend. There is a practical limit to this coalescing, however, as the cost of copying local memory into the packed PWC buffer outweighs the additional RDMA put cost. The packed PWC threshold is configurable using a `small_pwc_size` parameter and we use a value of

128 bytes in our evaluation below, although more appropriate values for a given system may be empirically determined.

As shown in Figure 1(a), the invocation of a PWC call at the producer will result in the specified source buffer being copied into the packed PWC ledger at the next available offset. A header is written to the ledger ahead of the data, which specifies the target address and length of the payload (i.e., the source buffer) and also the layout of the remote completion data (rid), which is written directly following the header and before the payload offset. The header also begins with a *head* byte, and we append a *tail* byte at the end of the payload to facilitate memory polling at the target where supported. On 64 bit systems, the total size of the packed PWC header is 20 bytes in length and we ensure that the total transferred data size ($tail - head$) is 8 byte aligned. One additional advantage of the packed PWC mode is that the source buffer need not be registered as the payload is copied to a pinned ledger buffer before being sent.

Before the backend put method is invoked, immediate data is set with a flag indicating that the remote event should be interpreted as indicating the completion of a packed PWC operation. On the consumer side, probing the network results in the remote CQ (RCQ) being polled for events and the immediate data inspected and unpacked. The PWC probe then performs a *lookup* at the current offset within that producer's local PWC ledger and unpacks the data, copies the payload to the destination buffer address, and returns the completion data header via a runtime-defined callback. The consumer may then perform some direct action on the rid or copy out the completion data for later processing.

On the producer side, a similar sequence of events takes place when probing for the local completion data (lid) that indicates the successful placement of the data on the target. The generation of a local completion event signals that it is safe to re-use the source buffer, with the exception of the packed PWC mode where the producer buffer is copied internally by Photon and is safe to re-use immediately at the return of the PWC call.

In contrast to packed PWC, ledger PWC uses an RDMA put to transfer the source buffer directly to the destination buffer address followed by a second put for the remote completion data, Figure 1(b). The rid is preceded by a short 8 byte header to indicate the type and length of the completion data. By default, all completion events are of type *user*, meant for delivery via PWC completion probing; however, the Photon PWC implementation also defines a set of completion types for internal messaging.

As a one-sided put operation, ledger PWC requires that the source and destination buffers be registered with Photon and that previously exchanged remote buffer metadata be passed in via the PWC call as described above. Also, since the network backend may not guarantee ordered delivery of both puts, immediate data indicating that the remote event should be interpreted as a ledger PWC operation has an additional impact when probed at the consumer. Before probe returns the rid for a ledger PWC, a *test* is made to determine if the first

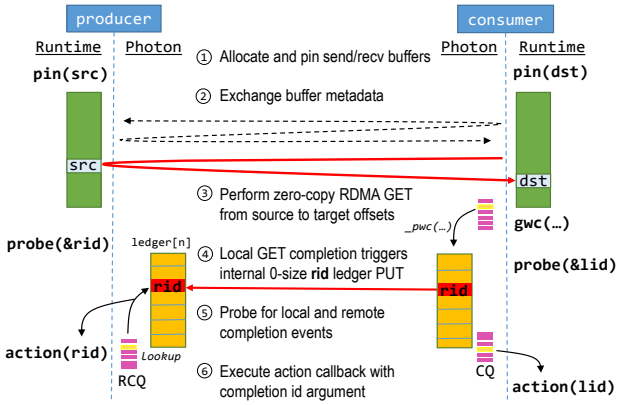


Fig. 2. A ledger GWC is consumer-initiated and requires an internal PWC reply to indicate that the producer data has been successfully read.

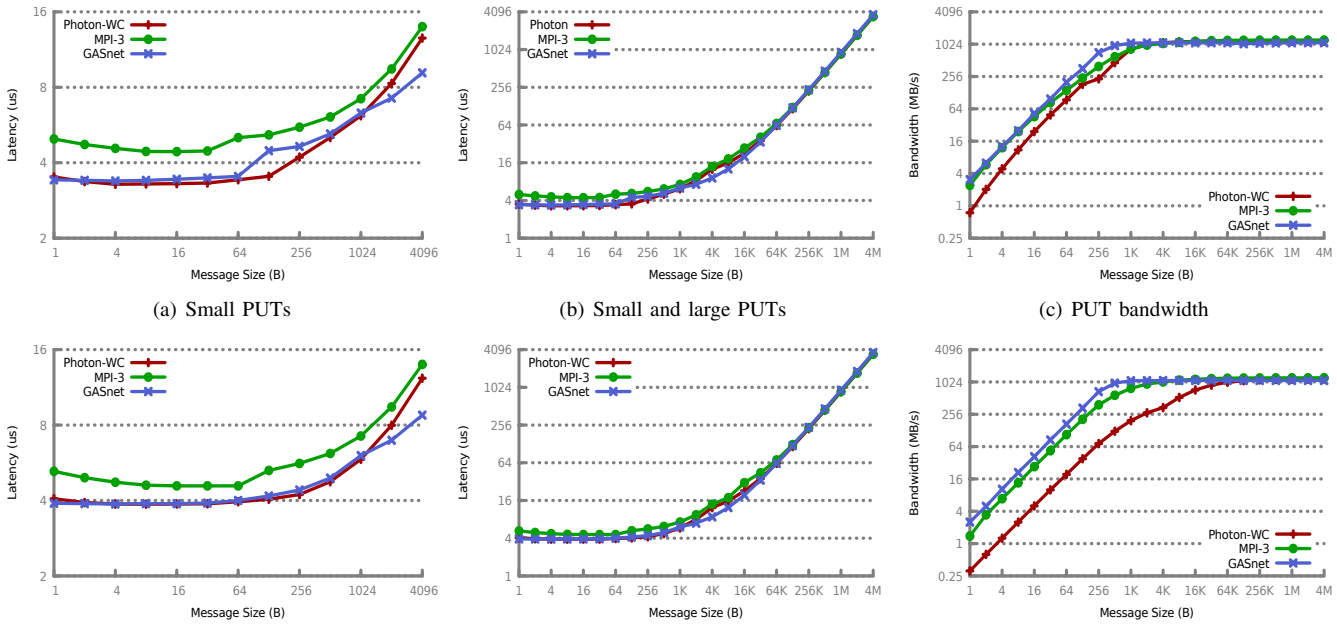
payload put completed successfully. A similar test is made if the second (ledger) put arrives first, atomically incrementing a PWC out-of-order ledger with a completed bit to be tested on reception of the first put completion event. The immediate data indicating whether a remote event is a first or second put is encoded at the producer, and we require this mechanism to ensure that probed completions guarantee the availability of the expected source data in the destination buffer.

F. Get with completion

A ledger GWC is essentially the inverse of the ledger PWC, Figure 2. A GWC is consumer-initiated and the completed event notification of a successful source buffer transfer would ordinarily be sufficient to complete a typical one-sided RDMA get. With GWC, however, we wish to inform the producer that a memory read was completed from a given peer. Thus, on the consumer, the local CQ completion event triggers the invocation of an internal PWC to the producer's ledger with the desired completion data. The local completion data is not returned via the consumer probe until the internal PWC event has been reaped.

The return of local and remote completion data during probing is controlled via flags to both PWC and GWC calls. If the runtime implementation does not wish to act on the completion of a locally called PWC, setting `PWC_REQ_NO_LCE` will prevent the generation of any local completion data for that operation. Similarly, setting `PWC_REQ_NO_RCE` will avoid the sending of remote completion data. In the latter case, no remote completion is the same as the familiar one-sided put operation. These options are also useful for internal PWC messaging. For example, ledger GWC avoids the return of a second local completion event by setting `PWC_REQ_NO_LCE` when sending the remote notification to the producer.

Finally, we note that GWC and PWC requests of size zero that specify remote completion data are allowed. Such requests simply communicate the rid to the destination without the put or get of external source and destination buffers and are useful for encoding short messages and actions between peers.



(a) Small PUTs (b) Small and large PUTs (c) PUT bandwidth

(d) Small GETs (e) Small and large GETs (f) GET bandwidth

Fig. 3. One-sided PUT and GET performance over InfiniBand comparing GASNet and MPI with Photon 8 byte remote completion data.

IV. Performance Evaluation

Our evaluation is designed to observe and quantify the overheads involved in coupling variable local and remote completion data using Photon PWC compared with traditional one-sided put and get operations supporting only local completion events. We chose the well-supported and widely used GASNet networking layer as our baseline with which to compare PWC against. As another point of comparison, we also include Ohio MicroBenchmark Suite (OSU-MB) results for MPI one-sided operations using the implementations available on our evaluation systems.

GASNet results were generated from the included conduit tests, for both InfiniBand and Aries, and compiled for sequential GASNet libraries. We report results from both the AM and non-blocking implicit put and get latency and bandwidth benchmarks. Similar benchmarks were created for Photon PWC that replicate the functionality of each GASNet test. All latency results show the result of repeated *post*-then-*wait-for-completion* loops over 10000 iterations to accurately represent the cost of a single operation. For bandwidth results, we used a flood approach to pipeline the posting of all iterations before probing for the first completion. All data points represent the mean values produced over 5 consecutive runs. Throughout our evaluation, reported *Photon-WC* lines use an 8 byte completion data size unless otherwise specified. Finally, all benchmarking software was compiled with "-O3" optimization flag.

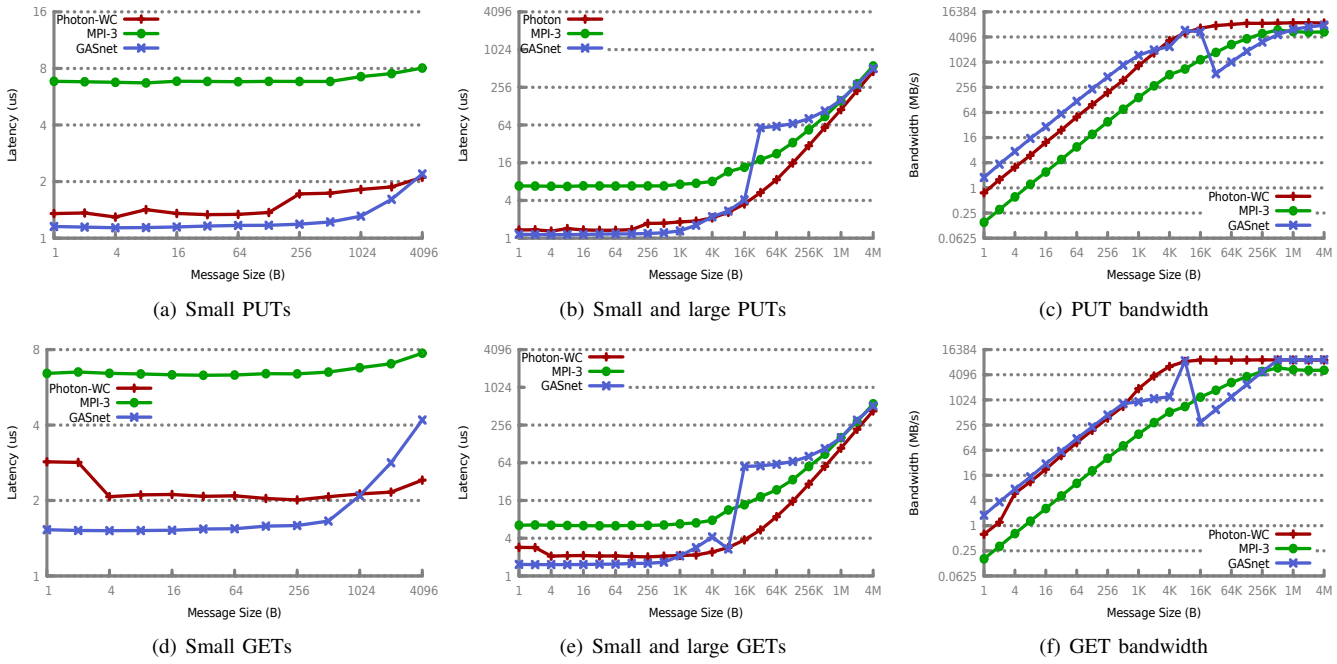
A. Network backend discussion

We note that no amount of optimization will help the core Photon API achieve better performance unless the network backends are correctly implemented, which includes making full use of what a given interconnect exposes for accelerating RDMA put and get operations. Message size thresholds, inlining of data where possible, and alignment considerations may all have a subtle but clearly observable impact on performance numbers as becomes apparent in our results.

Two implementation details in the Cray Aries backend, in particular, warrant a closer look. First, the Aries interconnect requires 4-byte alignment on remote address and length for get operations, but no such restrictions on puts. Since Photon places no restrictions on the length of source and destination buffers, an internal PWC protocol is used to transform a get request into an explicit put from the source. This naturally incurs some additional overhead in terms of an extra round-trip for the GWC-to-PWC handshake and synchronization. An alternative approach is taken by GASNet, where an assumption is made about the source and destination buffer lengths that allows for the *overfetching* of the necessary number of bytes to ensure correct alignment.

Second, Aries supports direct memory operation using Fast Memory Access (FMA) for the efficient transfer of small messages and the Block Transfer Engine (BTE) designed for offloading overheads involved in the transfer of larger messages. The message threshold for the use of BTE in Photon is configurable and was set to 1KB for our evaluation. In contrast, the GASNet version we used sets a default BTE cutover at 4KB RDMA puts and gets.

We are hopeful that recent efforts such as the OpenFabric Working Group's libfabric library will help mitigate many of these low-level interconnect considerations by providing a consistent and complete abstraction for RDMA operations



(a) Small PUTs (b) Small and large PUTs (c) PUT bandwidth

(d) Small GETs (e) Small and large GETs (f) GET bandwidth

Fig. 4. One-sided PUT and GET performance over Cray Aries comparing GASNet and MPI with Photon 8 byte remote completion data.

over a wide range of interconnects. Both a Photon libfabric backend, and a GASNet libfabric conduit, are available but have not been fully evaluated as of this writing.

B. *InfiniBand Results*

InfiniBand experiments were run on a cluster of 16 SuperMicro compute servers. Each node contains two Intel Xeon E5-2670 processors with 8 cores, running at 2.60GHz. Additionally, each node is installed with 32GB of memory and contains a dual-port Mellanox ConnectX-3 EN 10 Gbs Ethernet network adapter. Each compute node was running Ubuntu 14.04 (kernel version 3.19) and used the OS-provided *ibverbs* and *rdmacm* library packages. Additional installed software versions include OpenMPI 1.10.1, GASNet 1.26, and OSU-MB 5.1.

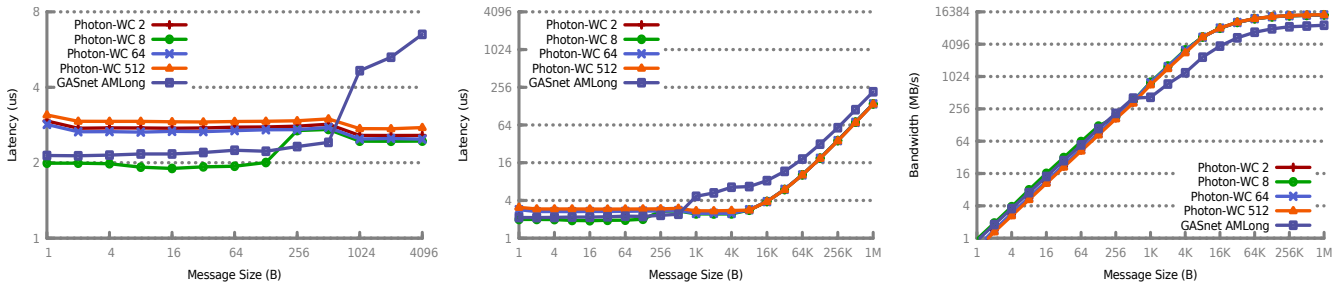
Figure 3 shows put and get results for small and large message sizes. In each row of charts, we reproduce a scaled version of the middle chart to highlight small message low-latency variations. For both put and get latency, we see PWC performance remain generally similar with GASNet across message sizes, even with 28 bytes (20 bytes header + 8 bytes rid) additional data transferred per operation in the packed PWC case for puts up to 128 bytes. We expect this is due to the inlining of small message data (`IBV_SEND_INLINE`) employed by both Photon and GASNet InfiniBand interconnect support. For PWC, we see a latency jump at 256 byte messages that coincides with the `small_pwc_size` value configured at 128 bytes, switching from packed to 2-put ledger completions. We expect that an inlining threshold is exceeded in GASNet to account for the latency increase at 128 bytes. Overall, it is promising to see PWC match, and in some cases, improve upon the GASNet latency numbers. GWC latency is more consistent in matching GASNet as there is no data inlining or packed versus ledger modes to differentiate small message performance, although we see both PWC and GWC take longer at 2KB messages before converging again at 32KB. Smaller one-sided MPI latency is significantly higher than both PWC and GASNet but converges at approximately 64KB messages.

Bandwidth results tell a different story altogether as the flooding test takes advantage of pipelined operations. In Figures 3(c) and 3(f), we see Photon-WC lines lag behind both GASNet and one-sided MPI. Here, the overheads incurred by both the PWC headers and completion data, plus the extra put in ledger PWC and GWC modes, are accumulative over sustained put and get operations. Essentially this means our observed "goodput" of measured data decreases as we do not account for the additional data communicated through the PWC completion notification path, although this difference is amortized as PWC approaches 4KB and 64KB messages for PWC and GWC, respectively.

C. *Cray Aries Results*

Experiments with the Aries interconnect were run on the NERSC Edison supercomputer [3]. Edison is a Cray XC30 with 5,576 compute nodes connected via a Dragonfly topology. Each node contains two 12-core Intel Xeon "Ivy Bridge" processors running at 2.4GHz and 64GB of installed memory, 32GB per socket. Installed software versions include cray-

[3]Edison at NERSC: http://www.nersc.gov/users/computational-systems/edison/



(a) Small Msgs (b) Small and large Msgs (c) Bandwidth

Fig. 5. Round-trip pingpong performance over Cray Aries with varying PWC completion sizes.

MPICH 7.3.0 and uGNI 6.0. GASNet and OSU-MB versions were the same as in our InfiniBand tests.

As shown in Figure 4, Cray Aries affords both Photon and GASNet much lower latency numbers.[4] Without the inlining of small data as on InfiniBand, we see that small PWC and GWC operations take up 0.25µs to 0.5µs longer than equivalently-sized GASNet puts and gets, as expected. Unaligned GWC requests are also more costly, as described in Section IV-A above. Interestingly, we found that GASNet exhibits a latency spike for puts after 16KB, and at after 8KB for gets after an initial reduction when cutting over to BTE. We are currently investigating if this is a known issue with the Aries conduit. Bandwidth measurements follow the same trend as on InfiniBand with the exception of GWC completion events being more efficiently pipelined on Aries after the initial un-aligned GWC-to-PWC overhead.

In Figure 5, we also show the results of an active message pingpong test, directly comparing GASNet *AMLong* messages with a PWC AM benchmark that sends various completion data sizes along with the message data. For small messages, we see round trip, *request-reply*, performance for Photon-WC and 8 byte completion data out-performing GASNet AMs up to 256B messages. This is in large part due to the effect of un-aligned memory copies for storing local completion data in Photon request descriptors. Contrast this with 2 byte completion data that exhibits more overhead than even 64 byte completions! GASNet AM latency jumps at 1KB messages and remains consistently higher compared to PWC for all completion data sizes, which exhibits a smooth increase in latency and bandwidth as message sizes increase.

D. PWC scaling

Figure 6 shows the results of a GWC scaling test up to 2048 nodes (49,152 cores) on Edison. We wrote a multi-threaded GWC benchmark that launches individual threads for probing local completions, remote completions, and a main thread for posting GWC operations, simulating a typical multi-threaded runtime integration usage. For each GWC put size, each peer randomly selects a neighbor and floods the remote side with get requests. For all GWC message sizes, we see that latency remains relatively consistent as the node count increases, with some variability expected due to the threaded nature of the test (threads were not pinned to cores). This result indicates that we can expect Photon to scale well to our early target of 10k nodes, although we must address the number and size of ledger buffers allocated by PWC as future work. The well-known tradeoff between space and time will need to be managed effectively in order to make use of PWC on current and emerging extreme-scale systems.

[4]Surprisingly, we saw significantly worse performance by one-sided MPI using cray-MPICH on Edison. We are in contact with NERSC to see if our reported numbers are due to a recent software update on Edison or if some other unknown factors may be playing a part.

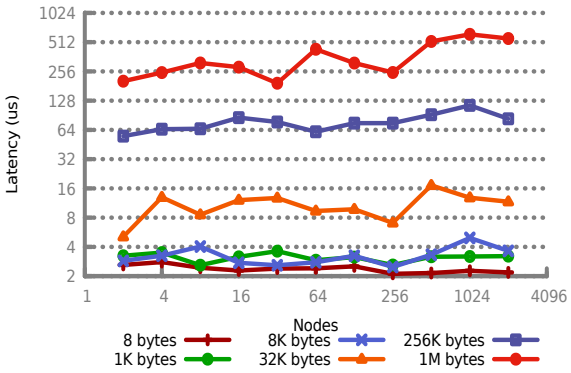


Fig. 6. Photon GWC scaling to 2048 nodes

V. Runtime Integration

The design of Photon PWC has been in large part influenced by our ongoing integration and co-development efforts with the HPX-5 [6] runtime system. As an implementation of Parallex, HPX-5 is an active message driven asynchronous runtime, which operates upon a shared GAS distributed memory model. Active messages in Parallex are known as *parcels* and a common target action of parcels are synchronization primitives known as lightweight control objects (LCOs). For example, parcels may target and set a value in an N-bit wide *and* LCO to ensure synchronization within, or completion of, N active distributed lightweight tasks.

The HPX-5 Photon network is designed to leverage both local and remote completion data to "activate" parcel messages,

as well as to operate on finer grained "interrupt" messages, which avoid the overheads in invoking the task scheduler to spawn a parcel action. Listing 1 highlights the key calls HPX-5 uses to effectively pull work off the network using PWC. Each rank progresses its network by calling `pwc_probe()` with a `rid_handler` argument, invoked when new completion data is available from the PWC ledger. Within the completion data, a remote HPX-5 peer has encoded an operation (`op`) and the address of the parcel that now resides on the target system. At the target, the `rid_handler` decodes and looks up the appropriate action `handler` to invoke with the parcel address encoded within the rid, `f(src,rid)`.

As of this writing, Photon PWC has become the default network available in HPX-5 due to its low overhead. We are currently working with our HPC applications team to develop new computational models that take advantage of the active runtime features enabled by the evolving PWC integration.

```
/* process completion data */
void rid_handler(int src, photon_cid rid) {
  /* decode HPX command from rid */
  hpx_addr_t op = cmd_get_op(rid);
  /* lookup and set action handler */
  hpx_action_handler_t handler =
       action_table_get_handler(here->actions, op);
  cmd_handler_t f = (cmd_handler_t)(handler);
  /* execute handler method with rid data */
  f(src,rid);
}

/* network progress loop thread */
void *progress(void *args) {
  photon_cid rid;
  int flag, src;
  do {
    /* probe for rid completion in pwc ledger */
    pwc_probe(&flag,&src,&rid,rid_handler,LEDGER);
  } while (!DONE);
  return NULL;
}
```

Listing 1: Simplified HPX-5 PWC network progress and completion handler routines.

VI. Conclusion and Future Work

In this work, we demonstrated how a put-with-completion (PWC) model of RDMA event notification *with data* enables a powerful mechanism for implementing message-driven computation. We introduced Photon as the middleware library that enables PWC and described the design and underlying methodology of our approach, including an overview of how Photon is being integrated into an existing runtime system. Our evaluation shows that PWC successfully addresses overheads inherent in exposing user-defined completion data coupled to RDMA *put* and *get* operations. In both direct memory and active messaging scenarios, PWC is comparable to, or exceeds, the performance of commonly used one-sided RDMA libraries across two widely deployed network interconnects.

As future work, we plan to address the efficient scaling of Photon. We have identified key areas in data alignment optimizations and dynamic ledger sizing that can be immediately addressed. Finally, We expect that continued runtime integration efforts will necessitate additional PWC-enabled RDMA operations, for example, atomics and vectored operations.

References

[1] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ser. ISCA '92. New York, NY, USA: ACM, 1992, pp. 256–266.

[2] "The InfiniBand trade association," Infiniband Architecture Specification Volume 1, Release 1.2., InfiniBand Trade Association, 2004.

[3] R. Alverson, D. Roweth, and L. Kaplan, "The gemini system interconnect," in *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, ser. HOTI '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 83–87.

[4] G. Faanes *et al.*, "Cray cascade: A scalable hpc system based on a dragonfly network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 103:1–103:9.

[5] A. Danalis, A. Brown, L. Pollock, and M. Swany, "Introducing gravel: An mpi companion library," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–5.

[6] T. Sterling *et al.*, "Towards exascale co-design in a runtime system," in *Exascale Applications and Software Conference*, Stockholm, Sweden, Apr 2014.

[7] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ser. ICPPW '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401.

[8] D. Bonachea, "Gasnet specification, v1.1," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep., 2002.

[9] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 84–84.

[10] D. S. Greenberg, R. Brightwell, L. A. Fisk, A. Maccabe, and R. Riesen, "A system software architecture for high-end computing," in *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, ser. SC '97. New York, NY, USA: ACM, 1997, pp. 1–15.

[11] B. W. Barrett *et al.*, "The portals 4.0.2 network programming interface," Sandia National Laboratories, Tech. Rep., 2014.

[12] M. P. I. Forum, "MPI: A Message-Passing Interface Standard Version 3.0," 09 2012, chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[13] R. Gerstenberger, M. Besta, and T. Hoefler, "Enabling Highly-Scalable Remote Memory Access Programming with MPI-3 One Sided," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, Nov. 2013, pp. 53:1–53:12.

[14] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda, "Rdma read based rendezvous protocol for mpi over infiniband: Design alternatives and benefits," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '06. New York, NY, USA: ACM, 2006, pp. 32–39.

[15] P. MacArthur and R. D. Russell, "A performance study to guide rdma programming decisions," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, ser. HPCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 778–785.

[16] R. Belli and T. Hoefler, "Notified Access: Extending Remote Memory Access Programming Models for Producer-Consumer Synchronization," in *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium (IPDPS'15)*. IEEE, May 2015.

[17] M. Besta and T. Hoefler, "Active Access: A Mechanism for High-Performance Distributed Data-Centric Computations," in *Proceedings of the 29th International Conference on Supercomputing (ICS'15)*. ACM, Jun. 2015, pp. 155–164.