Breaking Band: A Breakdown of High-performance Communication

Rohit Zambre
rzambre@uci.edu
EECS, University of California, Irvine, USA

Megan Grodowitz
Megan.Grodowitz@arm.com
Arm Research, USA

Aparna Chandramowlishwaran
amowli@uci.edu
EECS, University of California, Irvine, USA

Pavel Shamis
Pavel.Shamis@arm.com
Arm Research, USA

ABSTRACT

The critical path of internode communication on large-scale systems is composed of multiple components. When a supercomputing application initiates the transfer of a message using a high-level communication routine such as an `MPI_Send`, the payload of the message traverses multiple software stacks, the I/O subsystem on both the host and target nodes, and network components such as the switch. In this paper, we analyze where, why, and how much time is spent on the critical path of communication by modeling the overall injection overhead and end-to-end latency of a system. We focus our analysis on the performance of small messages since fine-grained communication is becoming increasingly important with the growing trend of an increasing number of cores per node. The analytical models present an accurate and detailed breakdown of time spent in internode communication. We validate the models on Arm ThunderX2-based servers connected with Mellanox InfiniBand. This is the first work of this kind on Arm. Alongside our breakdown, we describe the methodology to measure the time spent in each component so that readers with access to precise CPU timers and a PCIe analyzer can measure breakdowns on systems of their interest. Such a breakdown is crucial for software developers, system architects, and researchers to guide their optimization efforts. As researchers ourselves, we use the breakdown to simulate the impacts and discuss the likelihoods of a set of optimizations that target the bottlenecks in today's high-performance communication.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; • **Networks** → Network measurement; • **Software and its engineering** → *Software performance.*

KEYWORDS

analytical modeling, performance analysis, what-if analysis, breakdown, high-performance communication, Arm-based server, ThunderX2, InfiniBand

ACM Reference Format:
Rohit Zambre, Megan Grodowitz, Aparna Chandramowlishwaran, and Pavel Shamis. 2019. Breaking Band: A Breakdown of High-performance Communication. In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337910

1 INTRODUCTION

"To measure is to know." — Lord Kelvin

Internode communication is the crux of supercomputing. We can classify the various components involved in sending a message into one of three categories: CPU, I/O, or network fabric, as shown in Figure 1. Software stacks on the CPU include the Message Passing Interface (MPI) and the communication protocol processing in the underlying communication frameworks. I/O encompasses subsystems on the processor chip such as PCI Express (PCIe). Network components are the high-performance interconnect's switches and physical wire. Each of these components on the critical path of communication poses an opportunity for optimization. However, blindly optimizing all of the components is impractical considering the technical challenges associated with each and the wide variety of use cases. For example, the latency of sending a large message is driven by the time spent in the network components. Hence, optimizing the software stack for this case would be a futile effort. On the other hand, the time spent in the software stack during the propagation of a small message is a considerable portion of the overall latency and, hence, optimizing the time spent in the CPU would be beneficial. Therefore, it is important to understand where to focus our optimization efforts.

With the apparent end of Moore's law, the architectures of recent servers are now featuring a large number of cores per node [11, 15], a trend that is likely to continue moving forward [25]. Furthermore, other on-node resources such as memory, translation lookaside buffers, and network-hardware registers are not growing at the same rate. Since developers desire to solve the same problem faster on newer machines, they need to rely on strong scalability with the decreasing amount of memory per core (assuming a static split with a process per core). At the limits of strong scaling lies fine-grained communication, that is, each core participates in communication, eliminating the need to synchronize with the cores on a node. Since each core communicates independently of the others, the size of the messages involved in communication is small. Hence, we focus our analysis on the communication performance of small messages since it is a critical factor in overall performance.

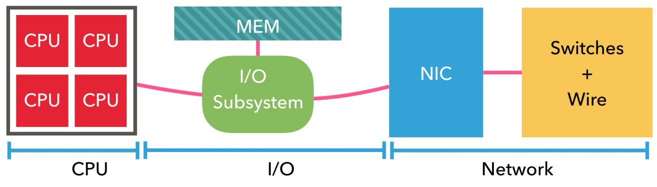


Figure 1: Components involved in the transmission of a message (on one end).

CPU, I/O, and network equally contribute to the communication performance of small messages; the times spent in each of the categories are on the same order of magnitude on state-of-the-art systems (we demonstrate this in § 6). Hence, optimizations of each category's constituents would be beneficial. This raises the question: *how much will optimizing component X improve the overall communication performance?* The answer to this question can guide the research and engineering efforts of software developers, system architects, and the HPC community at large. Typically, one measures the limits of a system's communication performance using injection-rate and latency tests. But such measurements do not inform the researcher where time is being spent or why the performance of one version of the system varies from that of another.

In this paper, we answer the posed question by analyzing the time spent in state-of-the-art software and system components during the transmission of messages. We classify the components into two levels: low and high. Low-level components include those that are not exposed to a typical end-user of an HPC system. These include the low-level communication framework (*e.g.* Verbs), the I/O subsystem (*e.g.* PCIe), and network components (*e.g.* Mellanox InfiniBand fiber). High-level components include programming model frameworks such as MPI.

Contributions and findings. This paper makes the following contributions.

(1) **Detailed breakdown**. As a first step to answer the posed question, we construct analytical models of the overall injection overhead and end-to-end latency of a system. Our models explain where and why time is spent during the transmission of a message. By attributing times to the models' constituents using precise CPU timers and traces from a PCIe analyzer, we show how much time is spent in low-level (§ 4) and high-level (§ 5) components, and thus present a detailed breakdown of high-performance communication. Our analytical models estimate the observed performance within a 5% margin of error on Arm ThunderX2. This work is the first of its kind on Arm. We use the breakdown to provide key insights in § 6.
(2) **Measurement methodology**. We present a detailed methodology to measure the overhead of each component such as the PCIe wire, the interconnect's wire, etc. Researchers with access to a similar analysis infrastructure described in § 3 can then measure overheads for components of their interest using our methodology.
(3) **Simulated optimizations**. Finally, we answer the aforementioned question in § 7 through a what-if analysis. We discuss the impact and likelihood of a set of optimizations that target the CPU, I/O, and network components of high-performance communication.

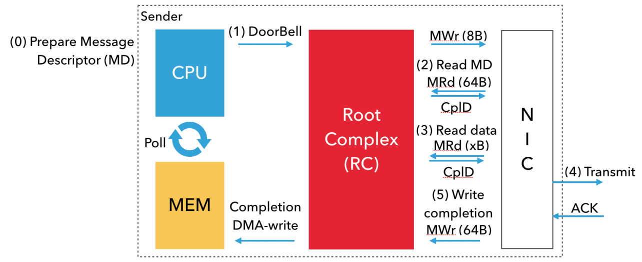


Figure 2: PCIe transactions and mechanisms on sender node to transmit data over wire.

2 BACKGROUND

The Network Interface Cards (NICs) of modern interconnects are typically connected to the processor chip on the node as a PCI Express (PCIe) device. In this section, we delineate the transmission and completion of messages in the context of the PCIe fabric.

PCI Express. The main conductor of the PCIe subsystem is the Root Complex (RC). It connects the processor and memory to the PCIe fabric. The peripherals connected to the PCIe fabric are called PCIe endpoints. The PCIe protocol consists of three layers: the Transaction layer, the Data Link layer, and the Physical layer. The first, the upper-most layer, describes the type of transaction occurring. In this paper, two types of Transaction Layer Packets (TLPs) are relevant: Memory Write (MWr), and Memory Read (MRd). Unlike the standalone MWr TLP, the MRd TLP is coupled with a Completion with Data (CplD) transaction from the target PCIe endpoint which contains the data requested by the initiator. The Data Link layer ensures the successful execution of all transactions using Data Link Layer Packet (DLLP) acknowledgements (ACK/NACK) and a credit-based flow-control mechanism. An initiator can issue a transaction as long as it has enough credits for that transaction. Its credits are replenished when it receives Update Flow Control (UpdateFC) DLLPs from its neighbors. Such a flow-control mechanism allows the PCIe protocol to have multiple outstanding transactions.

Mechanisms of a high-performance interconnect. From a CPU programmer's perspective, there exists a *transmit queue* (TxQ) and a *completion queue* (CQ). The user posts their message descriptor (MD) to the transmit queue, after which they poll on the CQ to confirm the completion of the posted message. The user could also request to be notified with an interrupt regarding the completion. However, the polling approach is latency-oriented since there is no context switch to the kernel in the critical path. The actual transmission of a message over the network occurs through coordination between the processor chip and the NIC using memory mapped I/O (MMIO) and direct memory access (DMA) reads and writes. We describe these steps below using Figure 2.

(0) The user first enqueues an MD into the TxQ. The network driver then prepares the device-specific MD that contains headers for the NIC, and a pointer to the payload.
(1) Using an 8-byte atomic write to a memory-mapped location, the CPU (the network driver) notifies the NIC that a message is ready to be sent. This is called *ringing the DoorBell*. The RC executes the *DoorBell* using a MWr PCIe transaction.
(2) After the *DoorBell* ring, the NIC fetches the MD using a DMA read. A MRd PCIe transaction conducts the DMA read.

(3) The NIC will then fetch the payload from a registered memory region using another DMA read (another MRd TLP). Note that the virtual address has to be translated to its physical address before the NIC can perform DMA-reads.
(4) Once the NIC receives the payload, it transmits the read data over the network. Upon a successful transmission, the NIC receives an acknowledgment (ACK) from the target-NIC.
(5) Upon the reception of the ACK, the NIC will DMA-write (using a MWr TLP) a completion (64 bytes in Mellanox InfiniBand) to the CQ associated with the TxQ. The CPU will then poll for this completion to make progress.

In summary, the critical data path of each post entails one MMIO write, two DMA reads, and one DMA write. The DMA-reads translate to round-trip PCIe latencies which are expensive.

A faster way to send a message that eliminates the PCIe round-trip latencies is *Programmed I/O (PIO)*. With PIO, the CPU copies the MD as a part of the *DoorBell*. Thus, the NIC doesn't need to DMA-read the MD. Another feature for small payloads is *inlining* which means that the payload is a part of the MD. Hence, when the NIC receives the MD, it does not need to DMA-read the payload. Typically, communication frameworks, such as UCX, combine PIO with inlining. This eliminates both the DMA-reads (steps (2) and (3)). In Mellanox InfiniBand, the PIO occurs in 64-byte chunks. Note that the CPU does more work in PIO (64-byte copy instead of an 8-byte write) and inlining (`memcpy`). However, the increase in CPU's work compared to the benefit gained from elimination of PCIe round-trip latencies is minimal.

3 EVALUATION SETUP

To measure the breakdown of time spent in components we use a system of two nodes, node 1 and node 2, that are connected to each other using a high-performance interconnect. Node 1 plays the role of the initiator in our following experiments. We use the CPU's timers to measure the time spent in software. To measure the time spent in other components, we use traces from a PCIe analyzer. Note that one can use this analysis infrastructure for any CPU or interconnect of interest.

We choose a state-of-the-art ThunderX2-based (TX2) server (running at 2 GHz) for the nodes and TOP500-popular Mellanox InfiniBand [2] as the high-speed interconnect. Specifically, we use ConnectX-4, a recent Mellanox InfiniBand adapter, and attach it to the node through a PCIe slot. A Lecroy PCIe analyzer sits just before the NIC on node 1, as shown in Figure 3. The overhead of the PCIe analyzer is negligible as we did not observe any difference in performance with and without it. Larsen et al [16] observe the same. The analyzer is a passive instrument that allows data to pass through fully unaltered [1].

For our software stack, we use the CH4 device of MPICH [21] with Unified Communication (UCX) [23] as the underlying communication framework. Specifically, we use UCX's *rc_mlx5* transport which is UCX's implementation of the data-path operations, such as posting to the transmit queue and polling from the completion queue, for modern Mellanox InfiniBand adapters.

To measure time spent in the CPU, we instrument relevant code with UCX's UCS profiling infrastructure [4], which internally reads the `cntvct_el0` register timer preceded by an `isb` for `aarch64`.

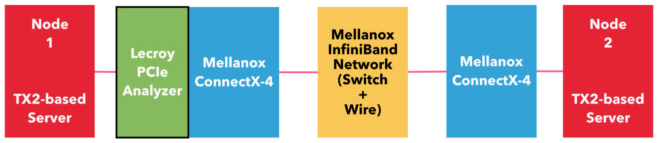


Figure 3: Two-node setup with PCIe analyzer on node 1.

The mean overhead of this infrastructure is 49.69 nanoseconds (a standard deviation of 1.48 for 1000 samples); we report software measurements in the rest of the paper after removing this overhead.

Each reported CPU or PCIe analyzer measurement is a mean of at least 100 samples. While measuring time of a component, we do not simultaneously measure time in any other component to minimize any effects of artificial slowdowns caused by the timer infrastructure. Hence, we do not require synchronized timers.

4 BREAKDOWN OF THE LOWER LEVEL

In this section, we present a detailed breakdown of time spent in the low-level components. These include the low-level communication protocol (LLP), the I/O subsystem, and network components. The LLP software drives the I/O and network hardware. We first define terminology for time spent in each of the low-level components.

- *LLP_post* – LLP performing a PIO post of one 8-byte message.
- *LLP_prog* – LLP dequeuing one entry of the completion queue during the progress of an operation.
- *PCIe* – payload traversing PCIe between RC and NIC.
- *Wire* – payload traversing the physical wire of the interconnect.
- *Switch* – overhead added by a network switch.
- *Network* – the total time in the interconnect (*Wire* + *Switch*).
- *RC-to-MEM(xB)* – RC writing an x-byte payload to memory.

We use UCX's low-level transport API, UC-Transports (UCT) for our LLP driver. It abstracts the capabilities of the various hardware architectures with minimal software overhead. The UCT driver runs the UCX `perftest`'s injection-rate and ping-pong style latency microbenchmarks, namely the `put_bw`, and `am_lat` tests, with a single thread. The `put` test corresponds to RDMA-writes while the `am`* test corresponds to send-receive semantics. Each message is 8 bytes, the size of a `double`.

4.1 Breakdown of the LLP

The LLP implements the HW/SW interface required to transmit a message and confirm its completion. The network driver (software) invokes the NIC (hardware) directly after correctly preparing resources and registers needed by the NIC during an *LLP_post*. The following details the steps involved in an *LLP_post*.

(1) Prepare MD – this involves the time taken to write the control segment of the descriptor. It also involves a `memcpy` of the small payload when inlining is used.
(2) A store memory barrier – this ensures that the MD is completely written before the CPU signals the NIC. This barrier is relevant only for a weak memory model (`dmb st` on `aarch64`).
(3) *DoorBell* counter increment – the NIC reads a *DoorBell* counter to perform speculative reads. The CPU updates this counter before writing to the NIC.

*`am` is short for active messages, terminology that describes send-receive style messaging in UCX.

(4) A store memory barrier – this ensures that the NIC sees the update to the *DoorBell* counter before any subsequent write to its device memory.
(5) PIO copy – this is the CPU's write to the memory-mapped device memory instructing the NIC to transmit the message. Device memory is typically an uncached, buffered memory region that supports out-of-order writes. For the TX2-based server in our setup, we use Device-GRE memory for the memory-mapped location. Though there would be a store memory barrier (`dsb st`) after the PIO copy to flush the data to the NIC, we observed experientially that this flush is not necessary for the microarchitecture of the TX2-based server. The PIO copy of an 8-byte message is one 64-byte chunk in Mellanox InfiniBand (see § 2).

Similarly, the LLP reads the designated memory location (where the NIC DMA-writes its completions) during an *LLP_prog*, the progress of an operation. This progress operation constitutes a load memory barrier for `aarch64`'s weak memory model to ensure that the read for a completion queue entry occurs before subsequent updates to data structures.

Measuring LLP and its breakdown. We measure *LLP_post* and *LLP_prog* by wrapping the UCS profiling infrastructure around the calls to `uct_ep_put_short` and `uct_worker_progress`. We use the same technique around the relevant regions of code in the implementation of `uct_ep_put_short` to measure the time in each of the categories of an *LLP_post*. While these categories are critical components of an *LLP_post*, they do not account for other miscellaneous time such as the function call overhead, branches to decide code path, etc. We compute this time by taking the difference of *LLP_post* and the sum of the times spent in the categories. Table 1 reports the times for *LLP_post*, *LLP_prog*, and each category of *LLP_post*. Figure 4 shows the breakdown of *LLP_post*. Since the *LLP_prog* contains only one critical category (the load memory barrier), we don't show its breakdown.

4.2 Injection overhead

Injection is the insertion of a message into the network. The message is injected when the payload reaches the NIC. We study the case when the user is transmitting messages continuously since this represents a system's injection limit. Then, the system's injection overhead, *Inj_overhead*, is the time difference between messages arriving at the NIC. This *Inj_overhead* explains why all the messages in a burst do not reach the NIC at time zero. We first model the injection overhead of PIO posts for a small message, then measure the overhead according to the model, and finally validate it.

Modeling injection overhead. Since the depth of the transmit queue (TxQ) is finite, the user cannot post indefinitely. Polling the completion queue (CQ) serves as the dequeue semantic for the TxQ. Hence, the user must poll in between their posts to inject messages into the NIC. Say, the user polls after every p posts. If $p = 1$, the depth of the TxQ is not utilized and the post translates to a synchronous post, that is, the user will be able to post the next message only after the previous message has reached the target node (since the completion is generated only when the host NIC receives an ACK from the target NIC (see § 2)).

To remove the overhead of waiting for a previous message to complete, the user must choose a value of p such that the completion

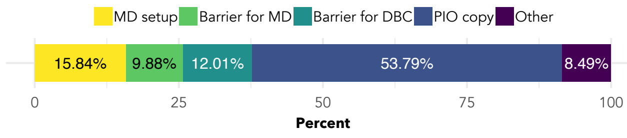


Figure 4: Breakdown of time in an *LLP_post* (MD: message descriptor; DBC: *DoorBell* counter).

for an earlier message is available during a poll. Such a value of p depends on the value of *LLP_post* and the time taken to generate a completion, *gen_completion*. From § 2, we can deduce that

$$gen\_completion = 2 \times (PCIe + Network) + RC\text{-}to\text{-}MEM(64B)$$

since the PCIe wire and the interconnect's network fabric are traversed twice: first while transmitting the message to the target NIC, and second while receiving the ACK from the target NIC and writing the corresponding completion. A completion in InfiniBand is 64 bytes and hence, the RC conducts a 64-byte write to memory on behalf of the host NIC. Then, to remove the overhead of waiting for a previous message, the lower bound on p is

$$p \geq gen\_completion / LLP\_post$$

In our modeling of the injection overhead, we assume that the user meets this lower bound on p.

Typically, the API of the network driver allows the user to poll a batch of completions, reducing the overhead of expensive memory barriers and function calls [14]. Say the user polls b number of completions in each batch. This means that the user can post only b posts in the next round of posts since only b entries have been dequeued from the TxQ. Note that b meets the lower bound mentioned above. Additionally, the user could perform some miscellaneous operations during the window of b posts or b polls. Let *tot_misc* demarcate the cumulative time spent in these other operations. Then, the overhead of the CPU to post a message is

$$\begin{aligned} CPU\_time &= \frac{b \times LLP\_post + b \times LLP\_prog + tot\_misc}{b} \\ &= LLP\_post + LLP\_prog + Misc \end{aligned}$$

where $Misc = tot\_misc/b$ is the miscellaneous overhead amortized for each message.

Hence, on average, messages arrive at the RC every *CPU_time*. Since PCIe supports multiple outstanding requests (see § 2), the RC initiates MWr PCIe transactions targeting the NIC as soon as it receives messages from the CPU. Considering that the RC is implemented with hardware logic, the time it takes to generate a transaction would be in the order of a few cycles. Hence, we ignore its contribution to the injection overhead. Note that the RC can generate transactions only if it has enough credits. Otherwise, it needs to wait for an UpdateFC DLLP from the NIC which would incur the overhead of the PCIe wire between the NIC and the RC (*PCIe*). Experientially, we observe that a single core does not exhaust the credits for MWr transactions. Hence, we do not model for the overheads imposed with exhausted credits in this paper.

Once the message leaves the RC, it incurs *PCIe* before arriving at the NIC. Hence, the injection overhead of a *single* message is

$$Msg\_inj\_overhead = CPU\_time + PCIe$$

While *Msg_inj_overhead* describes the time taken by each message to reach the NIC, it is not the same as the injection overhead

Table 1: Measured times of various components.

Component	Time (ns)
Message descriptor setup	27.78
Barrier for message descriptor	17.33
Barrier for *DoorBell* counter	21.07
PIO copy (64 bytes)	94.25
Miscellaneous in *LLP_post*	14.99
LLP_post (total of above)	175.42
LLP_prog	61.63
Busy post	8.99
Measurement update	49.69
Misc in *Inj_overhead* (total of above)	58.68
PCIe for a 64-byte payload	137.49
Wire	274.81
Switch	108
Network (total of above)	382.81
RC-to-MEM(8B)	240.96
`MPI_Isend` in MPICH	24.37
`MPI_Isend` in UCP	2.19
Callback for a completed `MPI_Irecv` in MPICH	47.99
Successful `MPI_Wait` for `MPI_Irecv` in MPICH	293.29
Callback for a completed `MPI_Irecv` in UCP	139.78
Successful `MPI_Wait` for `MPI_Irecv` in UCP	150.51

observed by the NIC, *Inj_overhead*, as we shall see next. When the system is issuing messages continuously, the *CPU_time* of the next message overlaps with the *PCIe* of the previous one (see Figure 5). Hence, the time difference between the initiation of messages is *CPU_time*. This holds true for any relation of *PCIe* with *CPU_time* (assuming that *PCIe* is not long enough to exhaust the RC's credits). When *PCIe* > *CPU_time*, *PCIe* of the next message can also overlap with *PCIe* of the previous one. Hence, from the perspective of the NIC, the time difference between the arrival of messages is the same as that between the initiation of messages, that is,

$$\begin{aligned} Inj\_overhead &= CPU\_time \\ &= LLP\_post + LLP\_prog + Misc \end{aligned} \tag{1}$$

Next, we measure the constituents of *Inj_overhead*. In § 4.1, we reported the times measured for *LLP_post* and *LLP_prog*. To account for *Misc*, we first explain what occurs between consecutive posts in UCX's `put_bw` benchmark.

Every message in the benchmark generates a completion. However, the benchmark polls for one completion every 16 posts. Hence, eventually the finite depth of the TxQ is fully utilized after which an *LLP_post* results in a "busy" post, that is, an *LLP_post* fails since an *LLP_prog* must occur before the next successful *LLP_post*. Thus, in the average case, after every successful *LLP_post*, there occurs a busy post. Additionally, the benchmark records a timestamp and updates its injection-rate measurements after every *LLP_post*. Table 1 reports the times for a "Busy post" and a "Measurement update" measured using the UCS profiling infrastructure wrapped around the relevant code paths; *Misc* = 56.58 nanoseconds.

Breakdown of injection overhead. The PCIe trace of the `put_bw` test shows the observed injection overhead of the system.

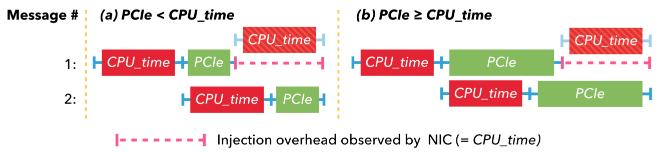


Figure 5: Injection overhead observed by the NIC.

Figure 6 shows a snippet of the PCIe trace after filtering for downstream (RC to NIC) transactions. The data in each downstream transaction is 64 bytes corresponding to the PIO post of an 8-byte payload. Every transaction is associated with a timestamp. This timestamp corresponds to the time when the PCIe analyzer observes the transaction. Since the PCIe analyzer is sitting just before the NIC, these timestamps correspond to the times at which the messages reach the NIC. Hence, calculating the delta of the timestamp of consecutive transactions would result in the observed *Inj_overhead*. Figure 7 shows the distribution of this overhead. The modeled injection overhead of **295.73** nanoseconds is within **5%** of **282.33** nanoseconds, the mean observed injection overhead. Figure 8 shows a percentage breakdown of *Inj_overhead*.

4.3 Latency

Latency is the total time incurred by a message starting from the time the host node initiates the transfer to the time of writing the payload in the destination buffer on the target node.

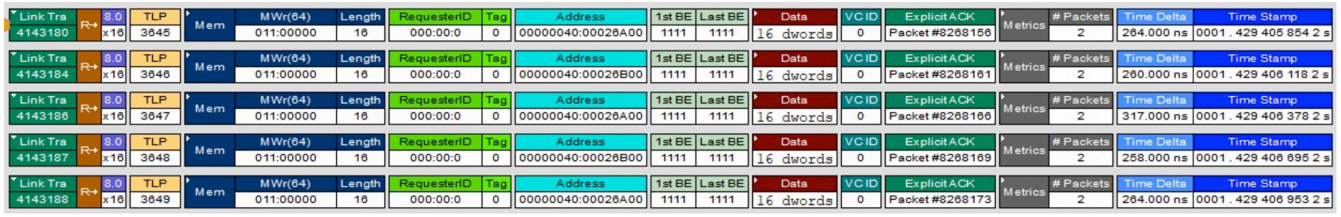
Modeling latency. We study the latency of a short message transmitted using send-receive semantics. The initiation of the transmission begins with an *LLP_post*, after which the message traverses the PCIe fabric and reaches the NIC. The NIC then transmits the message over the network fabric to reach the target node. On the target node, the NIC performs a MWr PCIe transaction, which traverses the PCIe wire and instructs the RC to write the payload into the target node's memory. Meanwhile the CPU on the target node has been polling for its posted receive to complete. The user can use its receive buffer only after a successful poll. Thus, for a payload of size, x, the time for latency is derived as follows,

$$Latency = LLP\_post + 2(PCIe) + Network + RC\text{-}to\text{-}MEM(xB) + LLP\_prog$$

Now, we measure the individual components that contribute to *Latency*. The value of an *LLP_post* is the same as the one measured in § 4.2, that is, 175.42 nanoseconds.

Measuring *PCIe*. To measure *PCIe*, we first measure the round-trip latency of the PCIe wire between the NIC and the RC. Since the PCIe analyzer sits just before the NIC, any transaction initiated by the NIC and the corresponding ACK DLLP from the RC will give us the start and end time of the required round-trip. For this purpose, we use the MWr transactions initiated by the NIC during the DMA-write of completions. The timestamp in the MWr transaction is the start time of the round trip and that in the corresponding ACK DLLP is the end time. Dividing this round-trip value by two is *PCIe* (the size of this MWr transaction is the same as that of the PIO copy: 64 bytes). We measure *PCIe* to be 137.49 nanoseconds.

Measuring *Network*. One way to measure *Network* would be to first measure the time difference between when a PIO post reaches the NIC and when the NIC receives an ACK from the target node for that PIO post. Then, dividing that difference by two would correspond to *Network* since the difference entails a round-trip



Link Tra	R→	8.0	TLP	Mem	MWr(64)	Length	RequesterID	Tag	Address	1st BE	Last BE	Data	VC ID	Explicit ACK	Metrics	# Packets	Time Delta	Time Stamp
4143180		x16	3645		011:00000	16	000:00:0	0	00000040:00026A00	1111	1111	16 dwords	0	Packet #8268156		2	264.000 ns	0001 . 429 405 854 2 s
4143184		x16	3646		011:00000	16	000:00:0	0	00000040:00026B00	1111	1111	16 dwords	0	Packet #8268161		2	260.000 ns	0001 . 429 406 118 2 s
4143186		x16	3647		011:00000	16	000:00:0	0	00000040:00026A00	1111	1111	16 dwords	0	Packet #8268166		2	317.000 ns	0001 . 429 406 378 2 s
4143187		x16	3648		011:00000	16	000:00:0	0	00000040:00026B00	1111	1111	16 dwords	0	Packet #8268169		2	258.000 ns	0001 . 429 406 695 2 s
4143188		x16	3649		011:00000	16	000:00:0	0	00000040:00026A00	1111	1111	16 dwords	0	Packet #8268173		2	264.000 ns	0001 . 429 406 953 2 s

Figure 6: PCIe trace of downstream PCIe transactions for UCX's RDMA-write injection-rate benchmark (put_bw).

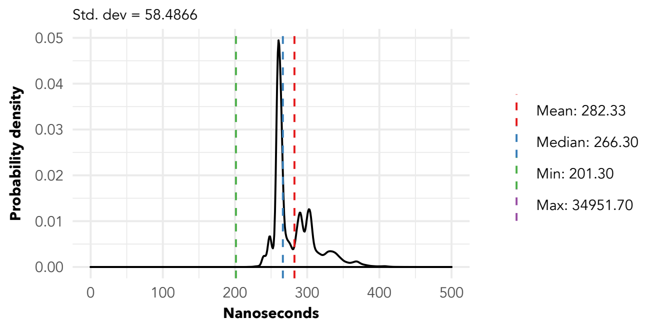


Figure 7: Distribution[†] of the observed injection overhead.

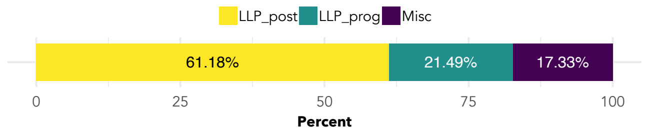


Figure 8: Breakdown of injection overhead with the LLP.

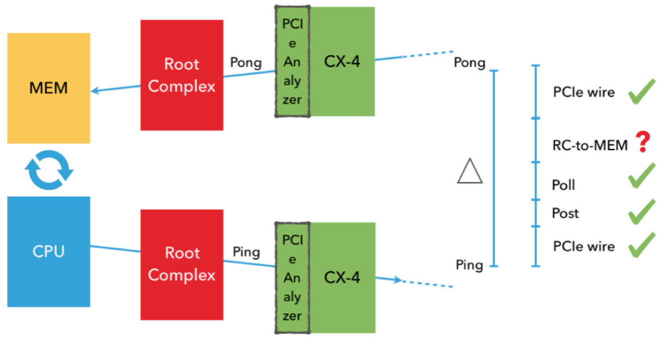


Figure 9: Measuring *RC-to-MEM(xB)* using the time delta between an inbound pong and outbound ping on node 1.

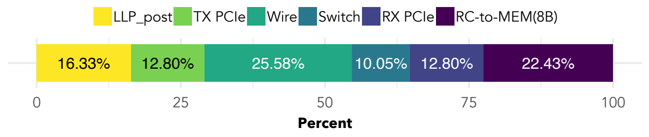


Figure 10: Breakdown of latency with the LLP.

Network latency. The timestamps on the PCIe trace of the ping-pong style `am_lat` benchmark allow us to employ this method. A downstream 64-byte PCIe transaction corresponds to a ping and the next upstream 64-byte PCIe transaction corresponds to the ping's completion which is generated upon reception of the ACK. Doing so, we measured the value of *Wire* to be 274.81 nanoseconds for a direct NIC-to-NIC connection. If the NICs are connected via a switch, the overhead of *Switch* is 108 nanoseconds. We measured this by taking the difference between two latency measurements: one with a switch involved and one without.

Measuring *RC-to-MEM(8B)*. To measure *RC-to-MEM(8B)*, we utilize the timestamps on the PCIe trace data of the `am_lat` ping-pong benchmark. As shown in Figure 9, the time difference between an incoming pong and outgoing ping entails an *RC-to-MEM(8B)*, two *PCIe*s (one for the inbound pong and the other for the outbound ping), a *LLP_prog* (successful poll), and a *LLP_post* (the ping). Once we measure the pong-ping difference from the PCIe trace, we can compute the value of *RC-to-MEM(8B)* since we have measured the values of the other components. This way, we measured the value of *RC-to-MEM(8B)* to be 240.96 nanoseconds.

Plugging in our measured values (reported in Table 1) into the latency model of a short message transmitted with send-receive semantics, we have *Latency* = **1135.8** nanoseconds.

[†]Max is not shown in the figure due to the large value.

Breakdown of latency. The observed latency from UCX's `am_lat` test is 1215 nanoseconds. The benchmark measures a round-trip latency and then divides the measurement by two to report the latency. Since a measurement update occurs before the target responds with a pong, we need to deduct half of "Measurement update" from Table 1 from the observed latency, which results in **1190.25** nanoseconds. The modeled latency is within **5%** of this observed latency. Figure 10 shows a percentage breakdown of latency.

5 BREAKDOWN OF THE HIGHER LEVEL

In this section, we present a breakdown of the time spent in the high-level components of high-performance communication. This comprises of the high-level communication protocols (HLP). The most commonly used programming model for large-scale parallel systems today is MPI [25]. Hence, at the highest level of the software stack sits an MPI library that implements the MPI standard. Modern implementations, such as the CH4 device of MPICH, rely on abstract communication frameworks, such as UCX, so that the MPI libraries do not need to maintain separate critical paths for all interconnects.

UCX in turn is composed of multiple components such as UC-Transports (UCT) and UC-Protocols (UCP). UCT is the LLP that we analyze in § 4.1. UCP implements high-level communication protocols such as collectives, message fragmentation, etc. using the low transport-level capabilities exposed through UCT. MPI libraries then use UCP to implement the specifications of the MPI standard.

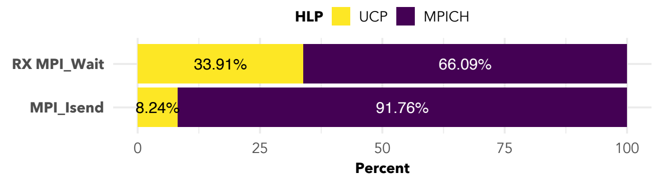


Figure 11: Breakdown of time in HLP.

We present a breakdown of time spent in the HLP for a communication-initiation operation such as `MPI_Isend`, and a communication-progress operation such as a successful (*i.e.* no busy waiting) `MPI_Wait` corresponding to an `MPI_Irecv`.

Measuring HLP and its breakdown. In an `MPI_Isend`, the MPI library first decides how to best execute the operation by checking if the data is contiguous, computing which communication interface to use, etc. Ultimately it will call into the UCP layer (`ucp_tag_send_nb`) which will eventually execute the LLP in the UCT layer (`uct_ep_am_short`). To measure the time spent in MPICH and UCP for an `MPI_Isend`, we first measure the total time of `MPI_Isend`, the total time of `ucp_tag_send_nb` inside MPICH, and the total time of `uct_ep_am_short` inside UCP by wrapping them with the UCS profiling infrastructure. We can then measure the time spent in MPICH and UCP by taking the differences of times between the upper and lower layers. For example, subtracting the total time of `ucp_tag_send_nb` from that of `MPI_Isend` gives us the time spent in MPICH.

Similarly, in an `MPI_Wait`, the MPI library executes its progress engine which ultimately calls into the UCP layer (`ucp_worker_progress`). UCP will then ensure progress on all outstanding operations that have posted by progressing the low-level UCT layer (`uct_worker_progress`). When an operation completes, UCT executes a registered callback into the upper UCP layer to update data structures that indicate the completion of the operation. Similarly, the UCP callback also executes a registered callback into the upper MPICH layer to indicate that the operation has completed. Note that these callbacks are executed before returning from `uct_worker_progress`. To measure the time spent in MPICH and UCP for an `MPI_Wait`, we measure the times spent in the registered MPICH and UCP callbacks in addition to measuring the total times of `MPI_Wait`, `ucp_worker_progress`, and `uct_worker_progress`. Since the UCP callback entails the MPICH callback, we can measure the time spent in the UCP callback alone by taking the difference between the total times spent in the callbacks. We can then measure the time spent in MPICH and UCP by taking the differences of times between the upper and lower layers and adding in the time for the upper layer's registered callback. For example, subtracting the total time of `ucp_worker_progress` from that of `MPI_Wait` and adding in the time of the MPICH callback gives us the time spent in MPICH.

Table 1 reports the time spent in MPICH and UCP on top of the LLP's HW/SW interface for an `MPI_Isend` (26.56 nanoseconds in total), and a successful `MPI_Wait` for an `MPI_Irecv` (443.8 nanoseconds in total). Figure 11 shows their percentage breakdown.

6 THE COMPLETE PICTURE

In this section, we first present a breakdown of the overall injection overhead and end-to-end latency including all the software, I/O, and network components for send-receive communication. Then, analyzing the breakdown, we note a number of insightful findings.

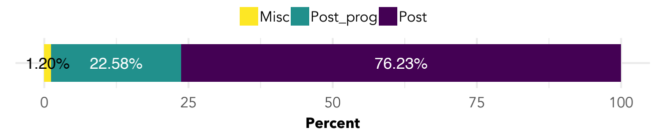


Figure 12: Breakdown of the overall injection overhead.

Overall injection overhead. § 4.2 shows that the injection overhead observed by the NIC for a single core is governed by the rate at which the CPU can send messages to the RC. Equation 1 defines this injection overhead. *CPU_time* in § 4.2 involved only the overhead of the LLP; in this section, we add in the overheads of the HLP to complete the picture. We redefine *CPU_time* as follows.

$$CPU\_time = Post + Post\_prog + Misc \tag{2}$$

where *Post* is the total time taken by the HLP and LLP to initiate an operation, and *Post_prog* is the total overhead imposed by both the HLP and LLP for the progress of a send-operation. *Post* is the sum of *LLP_post* and *HLP_post*, the time spent in the HLP during the initiation of a message. For our setup, *HLP_post* equals 26.56 nanoseconds, which implies *Post* equals 201.98 nanoseconds. Before attributing times to *Post_prog* and *Misc*, we delineate certain caveats.

First, UCP schedules the successful execution of *LLP_post* for busy posts (see § 4.2) during the progress of operations. Second, progress for a bunch of initiated operations is typically conducted with a batch-progress operation in the HLP such as `MPI_Waitall`. MPICH executes its progress engine until all the operations listed in `MPI_Waitall` complete. More important, UCP reduces the overhead of progress using unsignaled completions [14], which means the NIC DMA-writes a completion only every *c* operations to indicate the completion of all *c* operations ($c = 64$ in UCX). Hence, the overhead of progress is amortized over *c* operations.

The first caveat implies that the progress of some operations includes the overhead of initiation in the LLP. Since we already account for the successful posts of busy posts in *Post*, we deduct the cumulative *LLP_posts* corresponding to the busy posts from the total time of `MPI_Waitall` for analytical purposes. We do so by keeping track of the number of busy posts occurred before `MPI_Waitall`. Dividing the resulting total time by the number of operations progressed, we measure *Post_prog* to be 59.82 nanoseconds. Less than a nanosecond of *Post_prog* (due to the aforementioned amortization) occurs in the LLP; the rest occurs in the HLP (*HLP_tx_prog*).

We include the time incurred in busy posts under *Misc*. Using the tracked number of busy posts we can compute the total time spent in busy posts during an `MPI_Isend`-`MPI_Waitall` window. Dividing this total time by the number of operations in the window gives us an average of 3.17 nanoseconds per operation in *Misc*.

We use OSU Micro-Benchmark's [3] message rate test[‡] to measure the observed injection overhead. By taking the inverse of the message rate, we measure the mean injection overhead to be **263.91** nanoseconds. The injection overhead computed with Equation 2 is **264.97** nanoseconds which is within **1%** of the observed overhead. Figure 12 shows the breakdown of the overall injection overhead.

[‡]We remove the send-receive sync after every window of posts for a clear analysis.

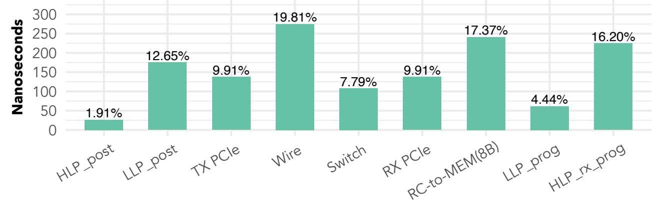


Figure 13: Breakdown of the end-to-end latency.

End-to-end latency. § 4.3 describes the constituents of *Latency* with minimal software involvement. To complete the picture, we add in the latencies of the HLP as follows.

$$\begin{aligned} Latency &= HLP\_post + LLP\_post + 2(PCIe) + Network \\ &\quad + RC\text{-}to\text{-}MEM(xB) + LLP\_prog + HLP\_rx\_prog \end{aligned}$$

HLP_rx_prog refers to the overhead of progressing the reception of an incoming message with MPI (after it has been written to memory by the RC). We assume the initiation of the receive (such as `MPI_Irecv`) overlaps with the rest of the constituents and, hence, do not account for its time in the end-to-end latency.

HLP_rx_prog is the sum of the times spent in the registered callbacks of MPICH and UCP along with the remaining time spent in MPICH after `ucp_worker_progress` returns. Note that the latter is not the equivalent of the total time spent in MPICH for a successful `MPI_Wait` minus the time spent in the MPICH callback. `MPI_Wait` is a blocking call and incurs a portion of the 293.99 nanoseconds before even progressing UCP. MPICH internally loops on `ucp_worker_progress` until the operation is complete. Hence, we specifically measure the time spent in MPICH after a successful `ucp_worker_progress` and observe this time to be 36.89 nanoseconds. The value of *HLP_rx_prog* then is 224.66 nanoseconds. Adding in the values of *LLP_prog*, *HLP_post*, and *HLP_rx_prog* to the modeled latency in § 4.3, the end-to-end latency is **1387.02** nanoseconds. This is within **4%** of the observed latency of **1336** nanoseconds measured by OSU Micro-Benchmark's point-to-point latency test. Figure 13 shows a detailed breakdown of this latency.

Insight 1. § 4.2 describes that the programmer cannot indefinitely initiate messages. Hence, the progress of a send operation serves as a "semantic bottleneck". Once the performance overheads imposed by this bottleneck is minimized through optimizations like unsignaled completions, Figure 12 shows that *Post* dominates (more than 70% of total) the overall injection overhead. Within *Post*, the LLP dominates as seen in "Initiation" of Figure 14.

Insight 2. Figure 15 presents the overall percentage breakdown of the end-to-end latency of a small message in the three categories: CPU, I/O, and network. The constituents of the software and I/O categories contribute almost equally (within 4% of each other) to their respective total times. In the case of *Network*, the latency of *Wire* dominates the overall off-node time. Note that none of the three categories dominates the overall latency. However, we observe that the network fabric constitutes less than a third of the overall latency while CPU and I/O components together contribute towards 72.4% of the latency. Hence, most of the overhead in the transmission of a small message is incurred on the node.

Insight 3. Figure 16 shows a high-level breakdown of the time spent on the node during the transmission of the message. The majority of this time occurs on the target node. Out of the time on the target node, the majority occurs during I/O, the majority of which is comprised by the RC writing the payload to memory. On the contrary, software comprises the majority of the time spent on the initiator node. This is due to the use of Programmed I/O (see § 2) for short messages. Consequently, I/O on the initiator node comprises only of a PCIe transaction unlike that on the target node.

Insight 4. Figure 14 shows that the HLP dominates the progress of both send and receive operations. The progress of a receive operation is 4.78× higher than that of a send operation.

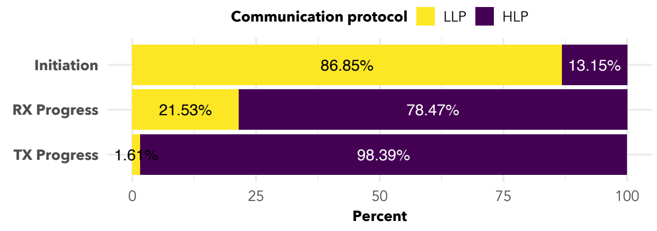


Figure 14: Breakdown of time in HLP and LLP during the initiation and progress of communication.

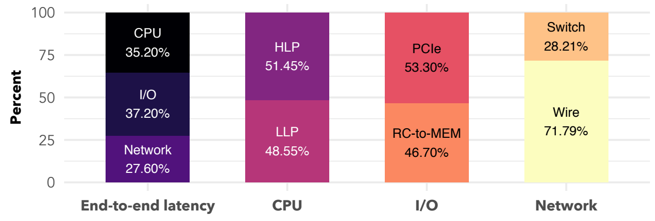


Figure 15: High-level breakdown of the end-to-end latency.

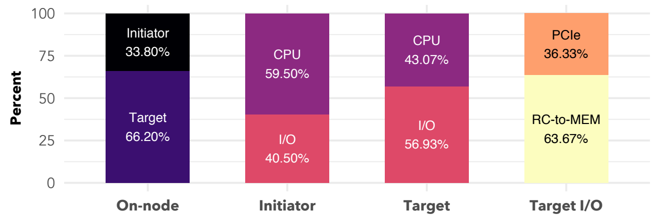
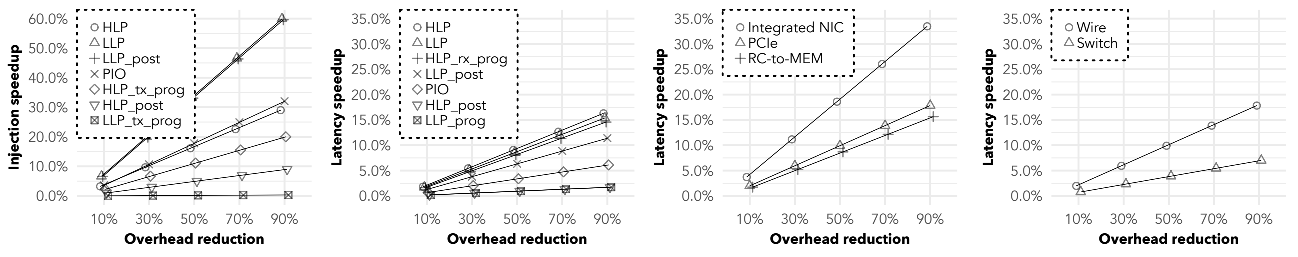


Figure 16: Breakdown of time spent on node.

7 SIMULATED OPTIMIZATIONS

In this section, we use the insights gained from the breakdown of the complete picture in § 6 to study the effects of optimizing the CPU, I/O, and network fabric components on the injection and latency of small message transfers. In the figures that follow, we aim to answer the following question: if we optimize component X by Y%, what is the corresponding reduction in injection overhead and latency? The horizontal axis of Figure 17 represents the degree of optimization for the component of interest. It consists of five evenly spaced reductions in overhead, starting from 10% (1.1× faster) to 90% (10× faster). The vertical axis represents the speedup in the overall injection or end-to-end latency as a result of reducing the component's overhead. Note that the components of our models are not concurrent, that is, their executions do not overlap. Hence, evaluating the impacts of reductions in overheads on benchmarks such an MPI stencil kernel through a distributed system simulator



(a) CPU (b) CPU (c) I/O (d) Network

Figure 17: Simulated speedups in overall injection (a) and end-to-end latency (b, c, d) by reducing overheads of CPU, I/O, and network components (note the differences in y-axis scales).

(such as SimGrid [9]) results in exactly the same linear speedups that we generate through a manual what-if analysis in Figure 17. We organize our discussion into a set of relevant optimizations that target the different components. For each optimization we discuss their likelihood and evaluate their impact. We consider speedups more than 5% to be substantial.

7.1 On-node optimizations

In § 6 we learn that most of the time in the transmission of a small message is spent on the node. CPU and I/O components make for the on-node time. Below we discuss three relevant optimizations.

NIC integrated into a System-on-Chip (SoC). The idea of this optimization is that the NIC sits on the same die as that of the processor. The deployment of such a solution would be in the form of an SoC so that instead of interfacing with the CPU through the PCIe subsystem, the NIC would connect to the network-on-chip (NoC). Such a tight integration of the NIC and the CPU would eliminate a majority of the I/O subsystem's overhead, which accounts for the majority of the time in the latency of a small message. While integrated NICs are not commonplace in today's HPC systems, they are more than likely to become ubiquitous in the future given the potential of their impact. There have been multiple works [8, 17] that argue for and evaluate the performance of SoC-integrated NICs showing their benefits in terms of better performance and higher CPU availability for all message sizes. More recently, Arm-based supercomputers are on the rise [13] since they allow HPC vendors to integrate their custom solutions (such as an integrated NIC) with Arm IP on SoCs. The Tofu interconnect D [5] on Fujitsu's post-K machine is a prominent example of this optimization. With Tofu's NIC integrated into a post-K-node, the RDMA-write latency has been improved by nearly 400 nanoseconds.

Impact. "Integrated NIC" in Figure 17c shows the impact of a solution that simply brings the NIC closer to the TX2-based SoC. While one can expect such a solution to eliminate most of the I/O overhead, we can observe over a 15% improvement in overall latency even with a modest 50% reduction in I/O time. In fact, a tightly integrated NIC allows for opportunities to reduce the involvement of the CPU in the LLP's HW/SW interface and thereby increase its availability for computational tasks. Recall that the reason for the use of PIO for small messages is expensive PCIe round-trip latencies with the communication-offloading approach (see §§ 1 and 2). Since an integrated NIC would sit close to memory, round-trip latencies performed by the hardware logic of the NIC would most likely be faster than involving the CPU in PIO.

Improving the initiation of a message in LLP. This optimization deals with how writes to device memory occur in the microarchitecture of a processor. Ideally, writes to `aarch64`'s *Device memory* [18] should be as fast as writes to its *Normal memory* [18]. Such an optimization is likely since the current difference between 64-byte writes to Normal and Device memory is more than 90%, hinting that there exists room for optimization. It would reduce the time spent in the PIO copy, which accounts for more than 50% of the time in *LLP_post* (see Figure 4).

Impact. "PIO" in Figure 17a and Figure 17b shows the impact of improving the 64-byte PIO copy on the overall injection and end-to-end latency, respectively. A regular 64-byte `memcpy` on the TX2-based server takes less than a nanosecond as expected. If we modestly project the overhead of PIO to reduce to 15 nanoseconds (84% reduction), overall injection can improve by more than 25% and end-to-end latency can improve by more than 5%.

Reducing software overheads. This optimization deals with software engineering targeted to reduce overheads in the HLP. However, unlike the previous optimizations, it is unlikely that this optimization would reduce overheads by more than 50%. For example, the current implementation of MPICH is highly optimized [21], reducing the number of instructions by 76% from its previous implementation for an `MPI_Isend`. We conjecture that software optimizations would reduce overheads by less than 20%.

Impact. Figure 17a and Figure 17b show the what-if analysis for the different components in the HLP and LLP. The "HLP" and "LLP" lines in the figures reflect the upper bound on speedups that would result from optimizing the components that constitute the HLP and LLP, respectively. For both injection and latency, optimizing the progress of operations in the HLP (*HLP_tx_prog* and *HLP_rx_prog*) can achieve speedups close to HLP's upper bound. Similarly, optimizing *LLP_post* can achieve speedups close to LLP's upper bound. If we consider software overheads would be reduced at most by 20%, the upper bounds reflect a less than 5% speedup in the end-to-end latency. On the other hand, a 20% reduction in overhead in the HLP can speedup injection by up to 6.44% while that in the LLP can do so by up to 13.33%.

7.2 Off-node optimizations

Figure 15 shows that 27.6% of the end-to-end latency is spent on the interconnect's *Wire* and in the *Switch*. Our foresight is that the reduction in off-node overheads is less than likely and that the resulting speedups with off-node optimizations alone would not be substantial. We explain our foresight below.

The reduction in *Wire*'s overhead is less than likely due to engineering complexities at the physical layer. In fact, it is possible that the latency will increase in future interconnects in order to accommodate for higher throughput. The conversion between the parallel PCIe signals and the serial signals on the interconnect's fiber transmission link occurs through SerDes (serializer/deserializer) integrated circuits. For throughputs higher than 100 Gb/s, the SerDes unit needs to be able to deliver higher throughput. While higher degrees of pulse amplitude modulation (PAM) deliver higher signal rates, they require more complex forward error correction (FEC), which increases the latency of the transmission in some cases by 300 nanoseconds [7, 12, 24].

The current latency of a high-performance interconnect's switch is already an order of magnitude lower than that of an Ethernet's switch [22]. New technologies like GenZ forecast their switch latencies to be 30-50 nanoseconds [10]. However, such low latencies are yet to be demonstrated. Only an optimistic reduction to 30 nanoseconds (72% overhead reduction) would correspond to a substantial speedup (5.45%) in end-to-end latency according to Figure 17d.

8 RELATED WORK

Prior research (described below) show the effects of optimizing certain components on the overall communication performance. We take an inverse approach that first explains the observed performance and then showcases the potential of optimizations. To the best of our knowledge, this paper's detailed breakdown encompassing all CPU, I/O, and network components is the first of its kind. Additionally, such work is the first for an Arm-based server.

Communication breakdown. Papadopoulou et al. [20] present a detailed instruction breakdown of initiation and progress functions between UCP and UCT to identify engineering and abstraction overheads. Similarly, Raffenetti et al. [21] analyze the overheads in the MPICH library using instruction analysis. Both reduce the number of instructions used in commonly used functions, resulting in higher communication performance. However, they only focus on one level of the stack. Our work spans both the MPICH and UCX stacks in addition to I/O and network components. Ajima et al. [5] present a breakdown of an RDMA-write latency on the post-K system using simulation waveforms of hardware emulators. Our work presents a breakdown with measured times using our described methodology as opposed to instructions or simulations to explain the observed communication performance.

Relevance of I/O. Like us, several others also mention the bottlenecks imposed by PCIe in datacenter networking systems. Kalia et al. [14] emphasize the need to consider the low-level details and features of Verbs and the PCIe subsystem while designing RDMA-based systems. In fact, R. Neugebauer et al. [19] and Alian et al. [6] contribute PCIe models to evaluate the impact of improvements to current I/O subsystems. We quantitatively compare the I/O overheads against those of CPU and network components. In addition to PCIe, we profile the time spent by the RC to write to memory. Unlike prior work, we use PCIe traces to validate software measurements.

9 CONCLUSION

Our analytical models of the injection overhead and latency of high-performance communication on state-of-the-art components explain observed performance with a 5% margin of error. The models and their resulting breakdown give the reader insights into where, why, and how much time is spent during the transfer of small messages. As the importance of small, fine-grained communication is rising, we believe that such a breakdown can guide the efforts of software developers and system architects alike to address the bottlenecks present today. More importantly, researchers and engineers can identify bottlenecks on their own systems using our detailed methodology described in this paper.

ACKNOWLEDGMENTS

We thank Giri Chukkapalli and Ham Prince from Marvell Technology Group, Yossi Itigin from Mellanox Technologies, and Pavan Balaji from Argonne National Laboratory for their aid and support.

REFERENCES

[1] [n. d.]. Teledyne LeCroy Summit T3-16 Analyzer. https://teledynelecroy.com/protocolanalyzer/pci-express/summit-t3-16-analyzer

[2] 2018. Top 500 High Performance Computing Platform Interconnect. Retrieved June 7, 2019 from http://www.mellanox.com/solutions/hpc/top500.php

[3] 2019. OSU Micro-Benchmarks 5.6.1. http://mvapich.cse.ohio-state.edu/benchmarks/

[4] 2019. UCS profiling. https://github.com/open/ucx/wiki/Profiling

[5] Yuichiro Ajima et al. 2018. The Tofu Interconnect D. In *2018 IEEE Intl. Conf. on Cluster Computing (CLUSTER)*. IEEE, 646–654.

[6] Mohammad Alian et al. 2018. Simulating PCI-Express Interconnect for Future System Exploration. In *2018 Intl. Symp. on Work. Char. (IISWC)*. IEEE, 168–178.

[7] Sudeep Bhoja et al. 2014. FEC codes for 400 Gbps 802.3 bs. *IEEE P802. 3bs* 400 (2014).

[8] Nathan L Binkert et al. 2006. Integrated network interfaces for high-bandwidth TCP/IP. *ACM Sigplan Not.* 41, 11 (2006), 315–324.

[9] Henri Casanova et al. 2014. Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms. *J. Parallel and Distrib. Comput.* 74, 10 (June 2014), 2899–2917. http://hal.inria.fr/hal-01017319

[10] Greg Casey. 2018. Gen-Z: High-performance interconnect for the data-centric future. https://www.opencompute.org/files/OCP-GenZ-March-2018-final.pdf

[11] Eric G. 2014. What public disclosures has Intel made about Knights Landing? Retrieved June 7, 2019 from https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing

[12] Ali Ghiasi et al. 2012. Investigation of PAM-4/6/8 signaling and FEC for 100 Gb/s serial transmission. *IEEE P802. 3bm* 40 (2012).

[13] Adrian Jackson et al. 2019. Evaluating the Arm Ecosystem for High Performance Computing. In *Proc. of the Platform for Advanced Scientific Computing Conf.* ACM.

[14] Anuj Kalia et al. 2016. Design Guidelines for High Performance {RDMA} Systems. In *2016 {USENIX} Annual Technical Conf. ({USENIX}{ATC} 16)*. 437–450.

[15] Patrick Kennedy. 2018. Cavium ThunderX2 Review and Benchmarks a Real Arm Server Option. Retrieved June 7, 2019 from https://www.servethehome.com/cavium-thunderx2-review-benchmarks-real-arm-server-option/

[16] Steen Larsen et al. 2015. Reevaluation of PIO with write-combining buffers to improve I/O performance on cluster systems.. In *NAS*. 345–346.

[17] Guangdeng Liao et al. 2009. Performance measurement of an integrated NIC architecture with 10GbE. In *2009 17th IEEE Symp. on High Perf. Inter.* IEEE, 52–59.

[18] Arm Ltd. 2019. ARMv8-A Memory types. Retrieved June 7, 2019 from https://developer.arm.com/docs/100941/latest/memory-types

[19] Rolf Neugebauer et al. 2018. Understanding PCIe performance for end host networking. In *Proc. of the 2018 Conf. of the ACM Special Interest Group on Data Communications*. ACM, 327–341.

[20] Nikela Papadopoulou et al. 2017. A performance study of UCX over InfiniBand. In *Proc. of the 17th IEEE/ACM Intl. Symp. on Cluster, Cloud and Grid Computing*. IEEE Press, 345–354.

[21] Ken Raffenetti et al. 2017. Why is MPI so slow?: Analyzing the fundamental limits in implementing mpi-3.1. In *Proc. of the Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*. ACM, 62.

[22] Stephen M Rumble et al. 2011. It's Time for Low Latency.. In *HotOS*, Vol. 13. 11–11.

[23] Pavel Shamis et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Ann. Symp. on High-Perf. Inter.*. IEEE, 40–43.

[24] Phil Sun. 2017. 100Gb/s Single-lane SERDES Discussion. *IEEE P802.3 New Ethernet Applications Ad Hoc* (2017).

[25] Rajeev Thakur et al. 2010. MPI at Exascale. *Proc. of SciDAC* 2 (2010), 14–35.