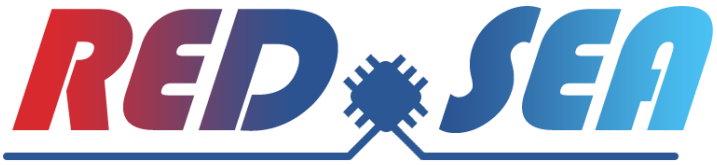
Network Solution for Exascale Architectures



D2.3 RoCE and IPoverBXI Evaluation Report

Document Properties

Contract Number	955776
Contractual Deadline	M18 (30/09/2022)
Dissemination Level	Public
Nature	Report
Edited by:	Nikolaos D. Kallimanis (FORTH), Gregoire Pichon (ATOS)
Authors	Giorgos Saloustros, Nikolaos D. Kallimanis, Nikolaos Chrysos, Jonathan Espié Caullet, Sylvain Goudeau, Grégoire Pichon
Reviewers	Gilles Moreau, CEA Jesus Escudero-Sahuquillo, UCLM
Date	29 September 2022
Keywords	
Status	Final
Release	1.3

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955776. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Greece, Germany, Spain, Italy, Switzerland.

History of Changes

Release	Date	Author, Organization	Description of Changes
0.1	01/05/2022	N. Chrysos	Template Creation.
0.2	01/05/2022	G. Saloustros	Adding material for the Tebis key-value store.
0.3	30/05/2022	N. Kallimanis	Text style fixes. Adding material for the GSAS environment.
0.4	14/06/2022	N. Kallimanis	Adding more text for the GSAS environment.
0.5	18/07/2022	J. Espié, S. Goudeau, G. Pichon	Adding material for IPoverBXI section.
0.6	01/09/2022	N. Kallimanis, N. Chrysos	Adding material for the intro.
0.7	05/09/2022	N. Kallimanis, G. Saloustros	Adding more material for the intro, various fixes.
0.8	05/09/2022	N. Chrysos, N. Kallimanis, G. Salustros	Executive summary.
1.0	06/09/2022	N. Chrysos, N Kallimanis	Final changes.
1.1	07/09/2022	N. Chrysos	Minor changes in executive summary, added conclusions.
1.2	09/09/2022	G. Salustros, N. Kallimanis, G. Pichon	Adding some entries in the Acronyms and Abbreviations section.
1.3	29/09/2022	N. Kallimanis	Addressing the comments and corrections proposed by the reviewers, i.e. Gilles Moreau (CEA) and Jesus Escudero-Sahuquillo (UCLM).

Table of Contents

List of Figures

List of Tables

Executive Summary

In the RED-SEA, we study new interconnect technologies built around BXI. Designing new solutions for RDMA interconnects, it is important to know the performance levels of existing solutions. Beside raw performance numbers, it is important to know how existing interconnects behave under real applications.

In this deliverable, we report the outputs of Tasks 2.1 and 2.2. In Task T2.1, we examine the performance of RoCE and Infiniband interconnects under relevant workloads. In particular, we evaluate how they perform under a distributed persistent key-value store framework named Tebis. In addition, we evaluate their performance under GSAS environment. In Task T2.2, we evaluate the performance of IPoverBXI. Supporting IP inside the network allows important applications and frameworks to run unmodified, which is a key offering of HPC interconnects.

In this deliverable we also report enhancements in IPoverBXI that are implemented in Ptlnet code and which have brought significant performance improvements for unmodified socket-based applications that run on top of BXI. Our key findings are: 1) with regards GSAS applications, the bandwidth of modern interconnects, i.e. Infiniband, significantly suffers in case that the traffic consists of a high number of small-size packets (i.e., a few hundreds of bytes at most); and 2) with regards Tebis and key-value stores, frameworks cannot easily saturate the available network throughput with small (64 B – 512 B) message sizes due to saturation of the packet rate of the network card, although higher-level bottlenecks do exist in today's systems for such small packets. Regarding IPoverBXI, our software enhancements have increased the TCP/UDP bandwidth from 10-40 Gbits/s to 70-80 Gbits/s almost independent from the MTU.

1 Introduction

In RED-SEA, the consortium envisions to support widely adopted protocols over BXI. For instance, the vision is to support RDMA over Converged Ethernet (RoCE) framework by supporting Ethernet over BXI. Additionally, the consortium is working for IP over BXI, in order to allow unmodified applications to run in new BXI networks.

Designing new solutions for RDMA, its useful to know the performance levels of existing solutions. In this deliverable, we report the output of Task 2.1 which evaluates the performance of InfiniBand and RoCE under datacenter relevant workloads and report results from experiments on a real testbed. Additionally, RDMA-capable NICs, and also the one offered by BXI, may support unmodified socket-based applications. In this deliverable, we evaluate the current offering by BXI and improve its performance by optimizing the software infrastructure.

Tebis (Vardoulakis, Saloustros, Gonzalez-Ferez, & Bilas, 2022) is a distributed persistent key-value store. Tebis targets flash storage devices (SSDs, NVMe) and uses RDMA for its client-server and server-server communication. Tebis can operate atop of protocols such as Infiniband or RoCE. Internally, it uses an LSM-Tree index structure to organize its data on the flash storage devices. The main characteristic of the LSM-Tree is its constant data reorganization. Furthermore, it keeps replicas of its data through a primary-backup protocol. Tebis instead of reorganizing data both on primary and the backups to minimize network traffic, it uses efficient bulk data transfers of RDMA to reorganize data only in the primary and sends the index to the backups. Sending the index saves CPU cycles and device I/O leading to overall better system performance.

GSAS environment (Kallimanis, Marazakis, & Chrysos, 2019) defines an API that provides a shared memory abstraction model to distributed applications. More specifically, GSAS gives the ability to processes running across remote servers/nodes to communicate in a way resembling a system that provides shared memory communication. It allows the applications to allocate/de-allocate memory (local and remote), and to perform several atomic operations (i.e., read, write, Fetch&Add, etc.) on the allocated space by using the appropriate API calls that the environment provides. Since v2.0, GSAS provides a high-performance OpenSHMEM v3.0 (Chapman, et al., 2010) implementation.

The HPC fabrics offer in-band communications between nodes using a socket-based network programming interface. This capability is provided by the IPoverBXI at the network interface. The bandwidth must be optimized for all message sizes. Atos has run and analysed the results of benchmarks and optimized the IP over BXI driver to obtain a stable and expected performance required in HPC systems. After evaluation of the bottlenecks, we report enhancements in IPoverBXI software that are implemented in Ptlnet code. Our enhancements have increased the TCP/UDP bandwidth from 10-40 Gbits/s to 70-80 Gbits/s almost independent from the MTU.
.

2 Evaluation of Infiniband/RoCE (FORTH)

2.1 The Tebis Key-Value Store

Key-value (KV) stores are the heart of modern datacenter storage stacks (Apache HBase, 2022) (Chodorow, 2013) (DeCandia, et al., 2007) (Lakshman & Malik, 2010) (Matsunobu, Dong, & Lee, 2020). These systems typically use a log structured merge (LSM) tree (O'Neil, Cheng, Gawlick, & O'Neil, 1996) index structure because it achieves: 1) fast data ingestion capability for small and variable size data items while maintaining good read and scan performance, and 2) low space overhead on the storage devices (Dong, et al., 2017). However, LSM-based KV stores suffer from high compaction costs for reorganizing the multi-level index (Lu, Pillai, Arpaci-Dusseau, & Arpaci-Dusseau, 2016) (Papagiannis, Saloustros, González-Férez, & Bilas, 2018), including both I/O amplification and CPU overhead.

To provide reliability and availability, state-of-the-art KV stores (Chodorow, 2013) (Lakshman & Malik, 2010) replicate KV pairs in multiple, typically two or three (Borthakur & others, 2008), nodes. Current designs perform costly compactions to reorganize data in the primary and backup nodes to ensure: (a) minimal network traffic by moving only user data across nodes and (b) sequential device accesses by performing only large I/Os. However, this approach comes at a significant increase in read I/O traffic, CPU utilization, and memory use at the backups. Given that all nodes function both as primaries and backups at the same time, eventually this approach hurts overall system performance. In Tebis, we rely on two observations: 1) The increased use of RDMA in the datacenter (Gao, et al., 2021) (Singhvi, et al., 2021) which increases available throughput at low CPU utilization. This makes it viable to trade network traffic for CPU and device I/O. 2) The use of KV separation in state-of-the-art KV stores (Lu, Pillai, Arpaci-Dusseau, & Arpaci-Dusseau, 2016) (Papagiannis, Saloustros, González-Férez, & Bilas, 2018) (Chan, Li, Lee, & Xu, 2018) (Facebook, 2018) that reduces, depending on the KV pair sizes, the size of the index.

Instead of storing values and keys in the LSM tree, KV separation places values in a separate log and uses additional metadata to point in the value log. This technique reduces I/O amplification by up to 10x (Batsaras, Saloustros, Papagiannis, Fatourou, & Bilas, 2020) by introducing small and random read I/Os, which are not as harmful for fast storage devices. Additionally, recent work (Xanthakis, Saloustros, Batsaras, Anastasios, & Bilas, 2021) (Li, et al., 2021) significantly improves garbage collection overhead for KV separation.

Based on this we design Tebis (Vardoulakis, Saloustros, Gonzalez-Ferez, & Bilas, 2022) an efficient replicated LSM-based KV store. The main novelty in Tebis is that it reduces compaction overhead at the backups by shipping a pre-built index from the primary. This approach reduces read I/O amplification, CPU overhead, and memory utilization in backup nodes. The design of Tebis addresses a main and two secondary challenges. The main challenge is an efficient rewrite mechanism of the index at the backup nodes: The index received at the backups contains segment offsets of the device in the primary. Tebis creates mappings between aligned primary and backup segments. Then, it uses these mappings to rewrite device locations at the backups efficiently.

Tebis replicates the data value log, using an efficient RDMA-based primary backup communication protocol that does not require the involvement of the backup CPUs in communication operations (Taleb, Stutsman, Antoniu, & Cortes, 2018). In addition, to reduce CPU overhead for client-server communication, Tebis uses one-sided RDMA write operations. The protocol of Tebis supports variable-size messages that are essential for KV stores using a single round trip to reduce the processing overhead at the server. We evaluate Tebis using a modified version of the Yahoo Cloud Service Benchmark (YCSB) (Cooper, Silberstein, Tam, Ramakrishnan, & Sears, 2010) that supports variable key-value sizes for all YCSB workloads, similar to Facebook's (Cao, Dong, Vemuri, & Du, 2020) production workloads.

2.1.1 One-sided RDMA Client-Server Protocol

The main design points that Tebis addresses are the management of RDMA buffers without synchronization at the server and support for variable-size messages.

2.1.2 RDMA Buffer Management

Tebis performs client-server communication via one-sided RDMA write operations (Kalia, Kaminsky, & Andersen, 2016) to avoid network interrupts and reduce the CPU overhead in the server. After connection establishment, the server and the client allocate a pair of buffers with configurable size (currently 256 KB). The region server frees these buffers when a client disconnects or fails. A thread monitors inactive queue pairs and checks if the queue pair is still in a valid state. Currently, this thread spins on RDMA buffer but could also use a sleep-wakeup approach. Clients manage both request and reply buffers to avoid synchronization among workers in the server. Clients allocate a pair of messages for each KV operation; one for their request and one for the server reply (step 1 in Figure 2.1). Each request header includes the buffer offset where the region server can write its reply. Workers complete requests asynchronously and respond to the client out of order. For put requests, the reply is of fixed size, so the client allocates the exact amount of memory needed prior to the operation. On the other hand, for get and scan requests, the reply size is variable and unknown a priori to the client. If the value size is larger than the buffer size of the reply, the region server sends part of the reply and informs the client to increase its allocation size for reply buffers to avoid similar cases in subsequent requests. Then, it retrieves the rest of the value from an offset provided by the server. As a result, the penalty, in this case, is a round trip with a small impact on overall latency

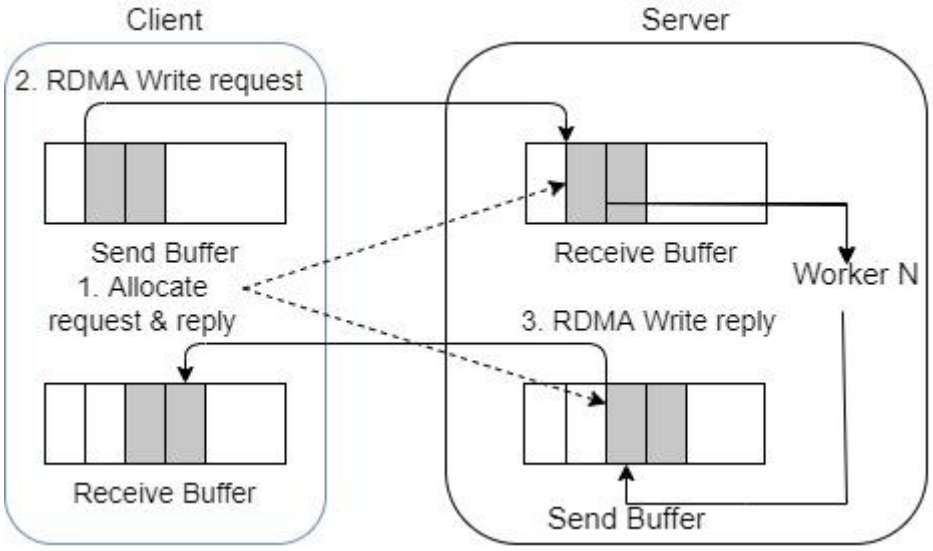


Figure 2.1. Allocation and request-reply flow of Tebis RDMA Write-based communication protocol.

Scaling the RDMA protocol of Tebis to large numbers of clients requires using more memory for RDMA buffers and polling for new messages in more rendezvous points. To limit the required memory for RDMA buffers, Tebis could divide this memory elastically between more and less active clients. Also, other approaches such as LITE (Tsai & Zhang, 2017) could be appropriate for persistent LSM KV stores since the 90-percentile tail latency of LSM KV stores is in the order of hundreds of μs. Polling a large number of rendezvous points can be mitigated by adjusting the number spinning threads in Tebis and distinguishing hot from cold clients to reduce the polling frequency. We leave these as extensions for future work.

Variable Size Messages and Task Scheduling. The main design challenge with variable size messages is how to detect their arrival at the region server without using network interrupts. All messages in Tebis consist of a message header of size 128 B and a variable size payload. To support variable size payloads, Tebis pads the payload to a multiple of the message header size. To detect incoming messages each region server uses a spinning thread, as shown in Figure 2.2. The spinning

thread polls a fixed memory location in each RDMA buffer it shares with a client. The spinning thread detects a new message by checking for a rendezvous magic number at the last four bytes of the current message header. Then, it reads the payload size from the message header to determine the end of the variable size message and the next header. A second rendezvous point is used at the end of the payload to check that the whole message has arrived. Upon receiving a message, the spinning thread creates a new client request for one of its workers, zeroes of the message in the RDMA buffer, and advances its rendezvous point to the next message header. The fact that all messages are multiple of message header size has the benefit that the spinning thread does not have to zero the whole message memory area. Instead, it only zeroes the possible locations of message header size in the area where future message headers may arrive.

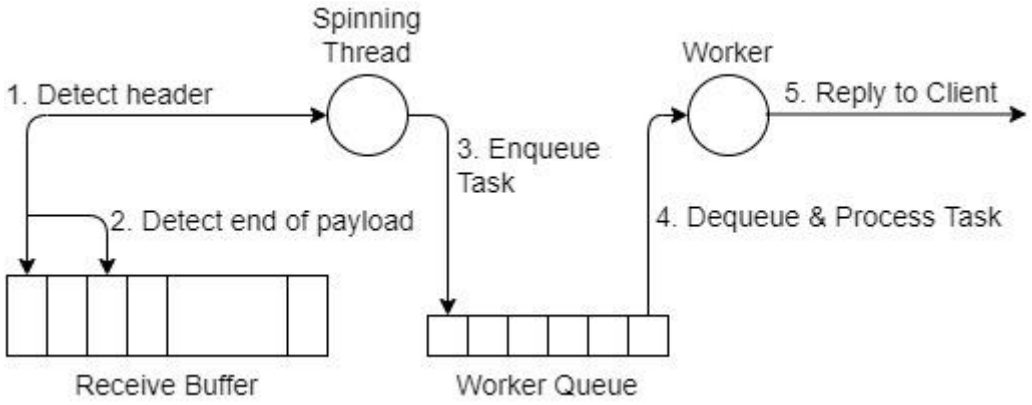


Figure 2.2. Message detection and task processing pipeline in Tebis. For simplicity, we only draw one circular buffer and a single worker.

When clients reach the end of the RDMA buffer, the client informs the server spinning thread to reset the rendezvous points at the start of the buffer. There are two possible cases:
(a) When the last message received reaches the end of the buffer, the spinning thread sets automatically the rendezvous point without any communication with the client. (b) When the remaining space in the circular buffer is not enough for the current message, the client sends a NOOP request message to the server with a size equal to the remaining space in the buffer. The spinning thread detects it and assigns it to a worker. The worker then sends a NOOP reply. When the clients detect the NOOP reply, it proceeds as in case a. Tebis uses a configurable number of workers. Each worker has a private task queue where the spinning thread places new tasks (Figure 5). Workers poll their queue to retrieve a new request and sleep if the spinning thread does not assign a new task within a period (currently 100 μs). To limit the number of wake-up operations, the spinning thread assigns a new task to the same worker as long as its task queue has fewer pending tasks than a threshold (currently set to 64). Then, the spinning thread selects the next running worker with fewer than threshold tasks. If none exists, it proceeds to wake-up a sleeping worker.

2.1.3 Evaluation

Our experimental setup consists of three identical servers equipped with two Intel(R) Xeon(R) CPU E5-2630 running at 2.4 GHz, with 16 physical cores for a total of 32 hyper-threads and with 256 GB of DDR4 DRAM. All servers run CentOS 7.3 with Linux kernel 4.4.159. Each server has a 1.5 TB Samsung PM173X NVMe SSD and a 56 Gbps Mellanox ConnectX 3 Pro RDMA network card. To ensure our experiments exhibit significant I/O activity, we use cgroups to limit the buffer cache used by memory-mapped I/O to 25% of the dataset size in all cases as shown in Table 2.

In our experiments, we run the YCSB benchmark (Cooper, Silberstein, Tam, Ramakrishnan, & Sears, 2010) and its workloads Load A and Run A to Run D. Table 1 summarizes the operations run during each workload. We run Tebis with 32 regions equally distributed across all servers. Furthermore, each server has two spinning threads and eight worker threads in all experiments.

Workloads	Operation Mix
Load A	100% inserts
Run A	50% inserts, 50% Updates
Run B	95% reads, 5% Updates
Run C	100% Reads
Run D	95% Reads, 5% Inserts

Table 1. Operation mix for YCSB. Workloads use Zipfian distribution and Run D uses the latest distribution

In all experiments, we use two separate servers to run the clients. In each server, we run four client processes with four threads per process. To generate enough outstanding requests for each server, each client process uses four queue pairs per server which are shared among each client's threads. Clients send requests asynchronously to all 32 regions as long as there is space in the RDMA buffers of the channel to each server, therefore, the outstanding number of requests is limited by RDMA buffer size. Each client generates the same number of operations. The total number of operations is 100 million requests for Load A and 50 million operations for each of the Run A – Run D phases in YCSB.

In our evaluation, we also vary the KV pair sizes according to the KV sizes proposed by Facebook (Cao, Dong, Vemuri, & Du, 2020), as shown in Table 2. We first evaluate the following workloads where all KV pairs have the same size: either Small (S), Medium (M), or Large (L). For this purpose, we use a C++ version of YCSB (Ren, 2016) and we modify it to produce different values according to the KV pair size distribution we study.

	KV Size Mix S% - M% - L%	#KV Pairs	Dataset Size (GB)
S	100 – 0 - 0	100 M	3.0
M	0 – 100 - 0	100 M	11.4
L	0 – 0 - 100	100 M	95.2
SD	60 – 20 - 20	100 M	23.2
MD	20 – 60 - 20	100 M	26.5
LD	20 – 20 - 60	100 M	60.0

Table 2. KV size distributions we use for our YCSB evaluation. Small KV pairs are 33 B, medium KV pairs are 123 B, and large KV pairs are 1023 B. We report the record count, dataset size, and cache size per server used with each KV size distribution

In addition, we evaluate workloads that use mixes of small, medium, and large KV pairs. We use a small-dominated (SD) KV size distribution as proposed by Facebook (Cao, Dong, Vemuri, & Du, 2020), as well as a medium dominated (MD) and a large dominated (LD) workload. We summarize these KV distributions in Table 2.

We examine the throughput (ops/s), efficiency (cycles/op), I/O amplification, and network amplification of Tebis for the following three setups: (1) without replication (No Replication), (2) with replication, using our mechanism for sending the index to the backups (Send-Index), and (3) with replication, where the backups perform compactions to build their index (Build-Index), which serves as a baseline. In all three configurations we use an L_0 size that stores 96K KV pairs. We note that Build-Index uses one L_0 for each replica, whereas Send-Index uses a single L_0 for the primary replica only. Thus, Send-Index is more memory-efficient than Build-Index.

We measure efficiency in cycles/op and define it as:

$$efficiency = \frac{\frac{CPUutilization}{100} * \frac{cycles}{s} * cores}{\frac{averageOps}{s}}$$

where CPUutilization is the average of CPU utilization among all processors, excluding idle and I/O wait time, as given by mpstat. As $cycles/s$ we use the per-core clock frequency. Finally, averageOps/s is the throughput reported by YCSB, and $cores$ is the number of system cores, including hyperthreads.

I/O amplification measures the excess device traffic generated due to compactions (for primary and backup regions) by Tebis, and we define it as:

$$IOAmplification = \frac{deviceTraffic}{datasetSize}$$

where deviceTraffic is the total number of bytes read from or written to the storage device and datasetSize is the total size of all key-value requests issued during the experiment.
We measure network amplification as traffic to all servers over application data written and read by the clients.

$$networkAmplification = \frac{networkTraffic}{datasetSize}$$

where networkTraffic is the total number of bytes sent and received by the server(s). Note that application data do not include network overhead (headers, acknowledgements), therefore, network traffic is always higher than application data. In addition, our RDMA client-server protocol uses a minimum payload of 256 B to reduce CPU use for detecting variable size messages in the servers, since for small messages the bottleneck is the packet rate in the NICs. This minimum payload is reflected in client-server network traffic for all experiments, including the No-Replication configuration.

2.1.4 Tebis Performance and Efficiency

We run Load A and Run A for all six KV distributions with a growth factor of 4, which minimizes I/O amplification (Batsaras, Saloustros, Papagiannis, Fatourou, & Bilas, 2020). Figure 2.3 and Figure 2.4 show that compared to Build-Index and for all KV size distributions, Send-Index increases throughput by 1.1 − 1.41×, CPU efficiency by 1.06 − 1.36×, and reduces I/O amplification by 1.13 − 1.45×. This happens because Send-Index 1) eliminates reads for L_i and L_{i+1} levels and 2) replaces in-memory sorting with index rewriting in backup regions. However, this trade-off favors Tebis since it uses available network throughput to reduce device I/O traffic and CPU usage.

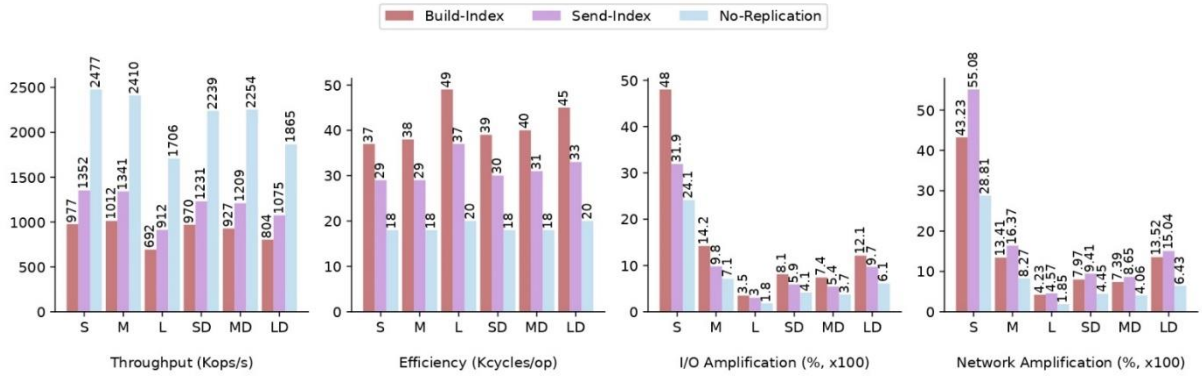


Figure 2.3 . Throughput, efficiency, I/O amplification, and network amplification for the different key-value size distributions during the YCSB Load A workload.

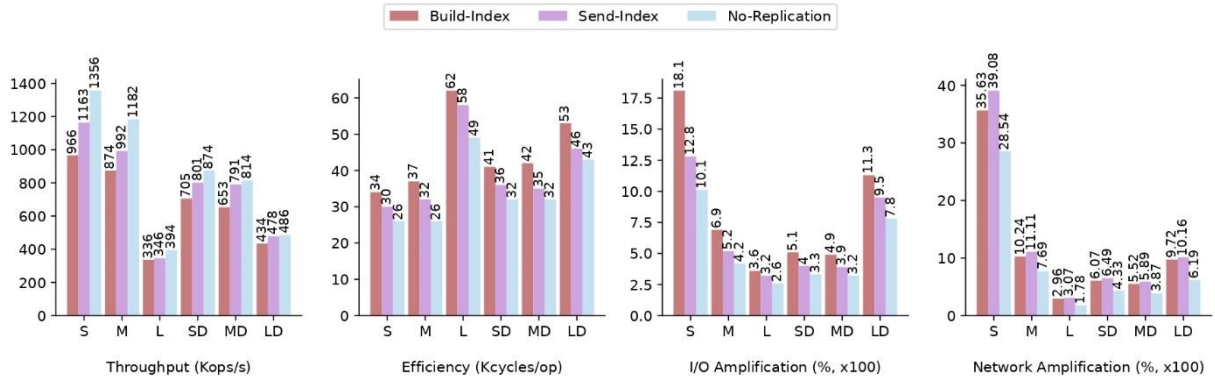


Figure 2.4. Throughput, efficiency, I/O amplification, and network amplification for the different key-value size distributions during the YCSB Load A workload.

- Evaluate InfiniBand network architecture under stressing and relevant workloads

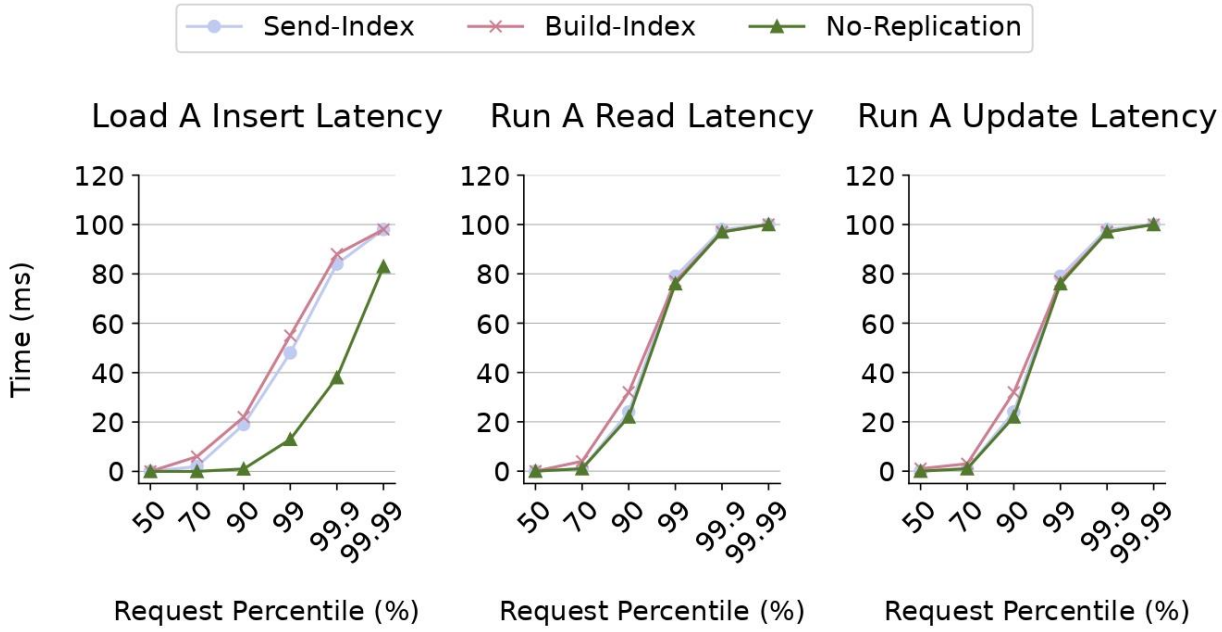


Figure 2.5. Tail latency for Load A and Run A using SD.

We also measure the tail latency for YCSB workloads Load A and Run A using SD (Figure 2.5). Compared to Build-Index, Send-Index improves the 50, 70, 90, 99, 99.9, and 99.99% tail latency between 1.05 − 1.77× for insert, 1.1 − 1.9× for read, and 1.02 − 1.65× for update. The reduction we observe in tail latency is due to the more efficient compactions in the backup regions. Send-Index releases device, CPU, and memory resources which the system uses for its primary regions and reduces stalls in L_0 where requests wait for compaction to finish.

2.1.5 RoCE Evaluation

In this experiment, we evaluate raw RoCE performance using the ibv_send_bw Mellanox tool with different message sizes. We use two machines that are equipped with a 56 Gbps Mellanox ConnectX 3 Pro RDMA network card and AMD EPYC 7551P CPUs with 32 cores at 2 GHz. We run two setups (1)

Unidirectional and (2) Bidirectional. On unidirectional bandwidth benchmarks, the client measures the bandwidth. On bidirectional bandwidth benchmarks, each side measures the bandwidth of the traffic it initiates, and at the end of the measurement period, the server reports the result to the client, who combines them together.

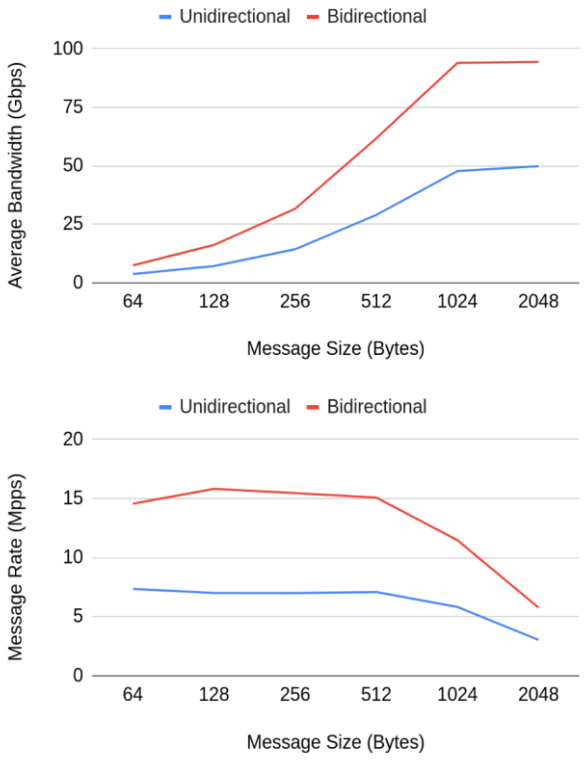


Figure 2.6: RoCE performance in throughput and message rate using ibv_write Mellanox for various message sizes.

As we see in Figure 2.6, in the Unidirectional setup the bandwidth for message sizes from 64-512 B is up to 25 Gbps. For these message sizes RoCE cannot saturate the network link because it reaches the available message rate (Mpps) of the network card (NIC) which is approximately 15 Mpps. For each RDMA write operation the NIC of the server sends an acknowledgment packet which the NIC of the client receives. As a result, for each RDMA write operation client NIC consumes 2 packets.
We further confirm this case from the bidirectional bandwidth and message rate reports in Figure 2.6 . Compared to the unidirectional setup, we see that for message size from 64 – 512 B the bandwidth and message rate is 2x. As a result, from small message sizes the server produces almost the same traffic compared to the client.

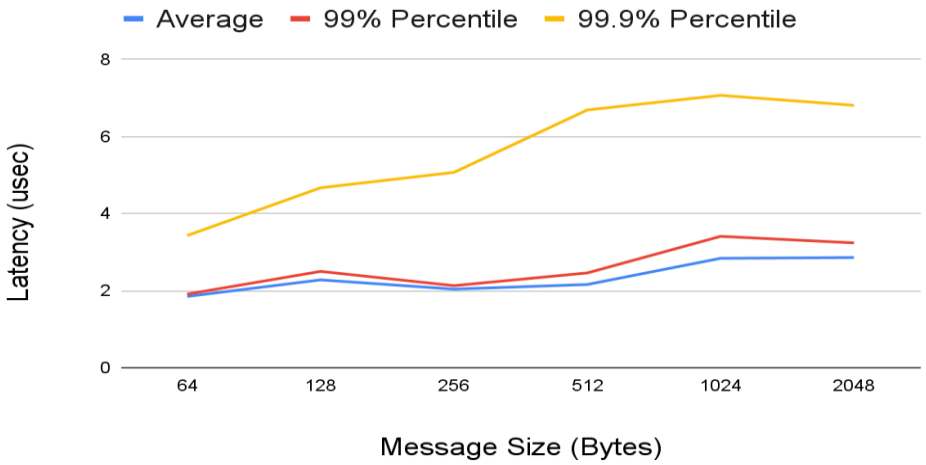


Figure 2.7. RDMA write message latency for various message sizes

As we see in Figure 2.7, Comparing 64 B messages, for 2 KB messages average latency increases by up to 1.5x, 99% percentile latency increases by 1.7x, and 99.9% percentile latency increases by 2x.

2.1.6 Tebis Evaluation with RAM Disk

In this experiment to further stress the network subsystem, we configure Tebis to use a large L_0 in order not perform data reorganizations. Also we use a ram disk for writing the write ahead log of Tebis to avoid all device I/O. We then run all YCSB workloads from Load A – Run D for message sizes 128 B, 256 B, 512 B, and 1024 B respectively.

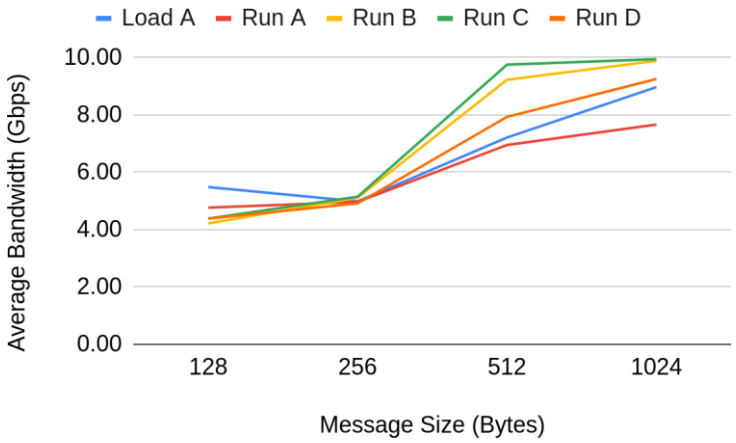


Figure 2.8. Load – Run D YCSB average bandwidth for various message sizes.

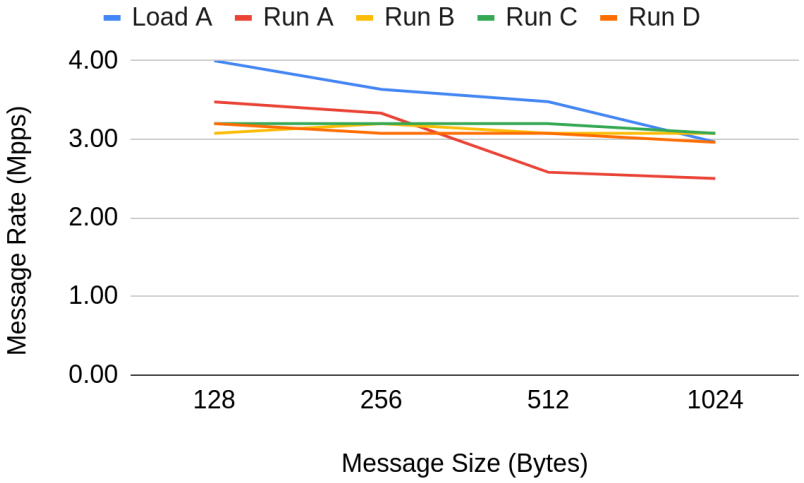


Figure 2.9. Load – Run D YCSB message rate for various message sizes.

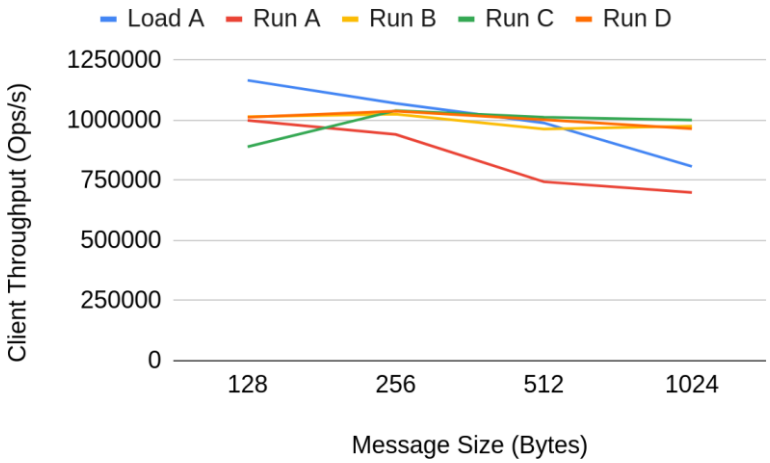


Figure 2.10. Load – Run D YCSB client throughput for various message sizes.

In Tebis, each protocol operation consists of two RDMA write operations. Client initially send a request message to the server via a one-sided RDMA write operation. Then, servers replies to the client via an RDMA write operation.

2.2 The GSAS environment

In order to evaluate the OpenSHMEM performance of GSAS for all of its supported network connectivity, GSAS is compared against OpenMPI and MPICH that both support OpenSHMEM. For an Ethernet network, MPICH provides a native implementation of OpenSHMEM, whilst OpenMPI utilizes the Unified Communication X (UCX) framework. For supporting Infiniband (IB) both MPICH and OpenMPI require the UCX framework. Since both OpenMPI and MPICH use similar underlying network stack, we only compare our GSAS implementation that supports IB against OpenMPI. Both MPI implementations are configured with the default configuration. Both OpenMPI and MPICH use TCP sockets, while the OpenSHMEM implementation provided by GSAS supports RDS and UDP sockets. In the case of the IB network, all the compared OpenSHMEM implementations use the same transport protocol, which is Reliable Connection Queue Pairs.

Our testing environment consists of two identical compute nodes equipped with both 1 Gbit Ethernet and 50 Gbit IB network interfaces. Each compute node is equipped with 2 Intel CPU Xeon E5-2620 processors with 12 physical cores and 24 hyperthreads, running at the 2.6 GHz frequency. Moreover, each compute node is equipped with 128GB of DDR4 DRAM. The operating system for both machines is Centos 7 with a Linux kernel of version 4.14 that has RDS sockets enabled. In order to perform a fair comparison, in all the experiments we bind processes to CPU cores by following a very similar policy. Specifically, we bind each process to a single physical CPU core and we avoid to utilize hyperthreading for avoiding running two threads on the same processing core that can pollute the experiments. We also bind our processes to processing cores in NUMA nodes that are closer to our network adapters in order to achieve the best performance.

2.2.1 Evaluation of Get and Put operations

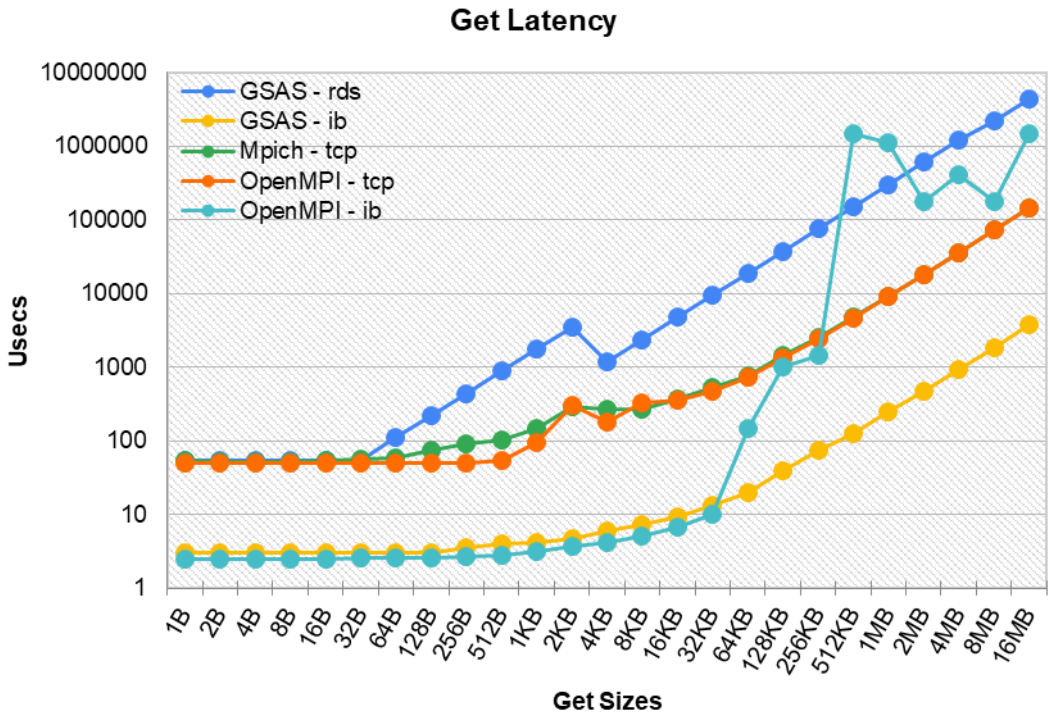


Figure 2.11 Latency performance of the OpenSHMEM Get operations.

In the experiments of Figure 2.11- Figure 2.14, the performance (i.e. latency and throughput) of Remote Memory Access (RMA) operations performance is evaluated. In the first experiment of Figure 2.11, we study. More specifically, we measure the throughput and latency for GET operations. We perform 10^6 GET operations for variable sizes of data. We calculate latency as the time taken for the operations to finish divided by the number of operations. The vertical axis of the Figure shows the latency measured in microseconds (usecs) while the horizontal axis displays the size of data transfers performed. In the case of Ethernet, we compare GSAS' RDS and UDP implementations with the TCP implementation of OpenMPI and MPICH. In the IB cases, we compare GSAS against OpenMPI. Figure 2.11 shows that the IB implementation of GSAS is competitive, starting at 1.55 times behind the MPI which uses UCX and progressively reducing the gap, but for sizes of 64KB and more, it manages to be almost 6 times better, maintaining a greater and a more consistent latency. This is a result of OpenMPI's complex architecture. OpenMPI splits data in packets called fragments. The maximum size of a fragment is 64KB. When data sizes are greater than 64KB, OpenMPI switches to a more complicated protocol that introduces overheads causing the performance to drop. Oppositely, GSAS is using a simpler approach of only having two types of messages. For sizes up to 128 bytes, we are transmitting inline IB messages. The inline feature is an implementation extension that is not strictly defined in any specification and its support varies based on the network adapter's manufacturers. We chose 128B as our maximum inline

message size, since it is a low enough size that all manufacturers support. For sizes greater than this, we switch to standard IB messages. For the commodity Ethernet cases, our implementation falls behind and is only able to stay close to the MPI implementations for sizes up to 32B. This is due to the limitations of GSAS design. In the Ethernet cases, GSAS was designed to provide very low latency for small messages up to 64B, with half of it being our headers and the rest consisting of the actual payload. This explains the linear increase for sizes of 64B and more, since more than one packet need to be transmitted. Although we provide a fast path mechanism for sizes of 4KB and more, explaining the drop at the 4KB mark, it is still unable to keep up with both MPI implementations. In addition to this, our testing environment does not support native RDS sockets, which explains RDS being worse than UDP and TCP. RDS natively is implemented to work over Infiniband. In our implementation we are forcing it over Ethernet, which leads it to be simulated over TCP sockets, adding some more overhead on top of it.

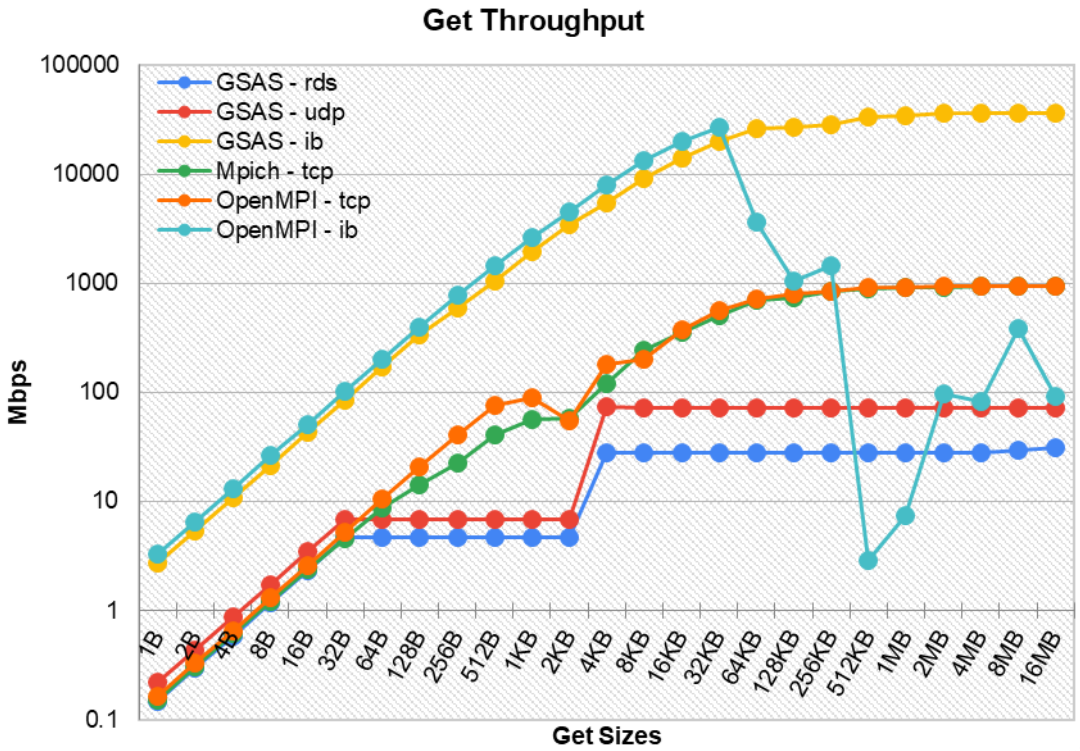


Figure 2.12 Throughput performance of the OpenSHMEM Get operations.

In Figure 2.12, we evaluate the throughput performance of GET operations. The amount of operations and data sizes are large enough to allow us to measure the maximum possible performance in terms of throughput. The vertical axis of the figure shows the transmission rate measured in Megabits per second (Mbps) and the vertical axis contains the size of transmitted data. In the IB cases, the initial throughput is relatively small, which is expected since transfer sizes are not large enough to saturate the network. Figure 2.12 shows that GSAS' throughput is very close to OpenMPI for data sizes up to 64KB, where it surpasses it. The reason behind the OpenMPI behaviour after 64KB is that it switches its standard transmission protocol, to a more complex one for large packets introducing heavy overhead. GSAS utilizes 72% of the links capability, peaking at a throughput of 36 Gbps. However, GSAS is not able to fully utilize the link's capability, since in the current implementation it utilizes a single thread for servicing operations. In the Ethernet cases, OpenMPI has better performance utilizing 100% of the links capability. The Ethernet implementation of GSAS is only optimized for small data transfers and its performance does not advance in larger data sizes.

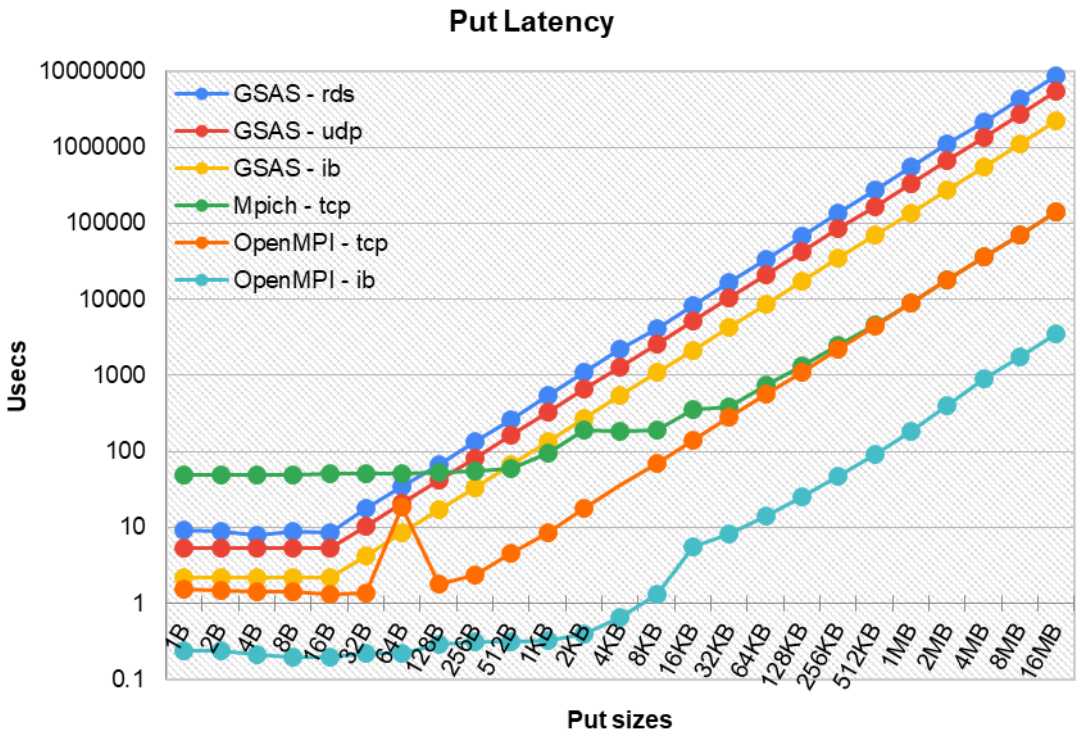


Figure 2.13 Latency performance of the OpenSHMEM Put operations.

In Figure 2.13, we evaluate the latency of PUT operations. Similarly to Figure 2.11, the vertical axis displays the latency in microseconds while the horizontal axis displays the size of data transfers. Figure 2.13 shows that the performance of PUT operations in GSAS falls behind the OpenMPI and MPICH in both IB and Ethernet. The reason behind this is due to its design. More specifically, the processes of an application exchange packets of fixed size (i.e. 32 bytes) with the atomic service instances of GSAS. Moreover, out of these 32 bytes, 8 bytes are used as our header, leaving 16 bytes as the payload. Thus, for transmitting more than 16 bytes, more than 1 packets are needed, which increases the latency. This is the reason of latency increase for sizes of 16 bytes or more. Contrary, In the case of GET operations, a client requests from a re mote node to receive the requested data. The remote node splits the requested data in MTU size packets and transmit them in a burst. However, in the case of a PUT operation, a client cannot use the MTU size to transmit to the server and instead is limited to transmitting packets of 32 bytes. This leads in an increased number of packets required that affect negatively both the latency and the throughput. Given that this is a design constrain, all supported networking protocols of GSAS for PUT operations are affected, i.e. IB, UDP and RDS as depicted on the Figure 2.13.

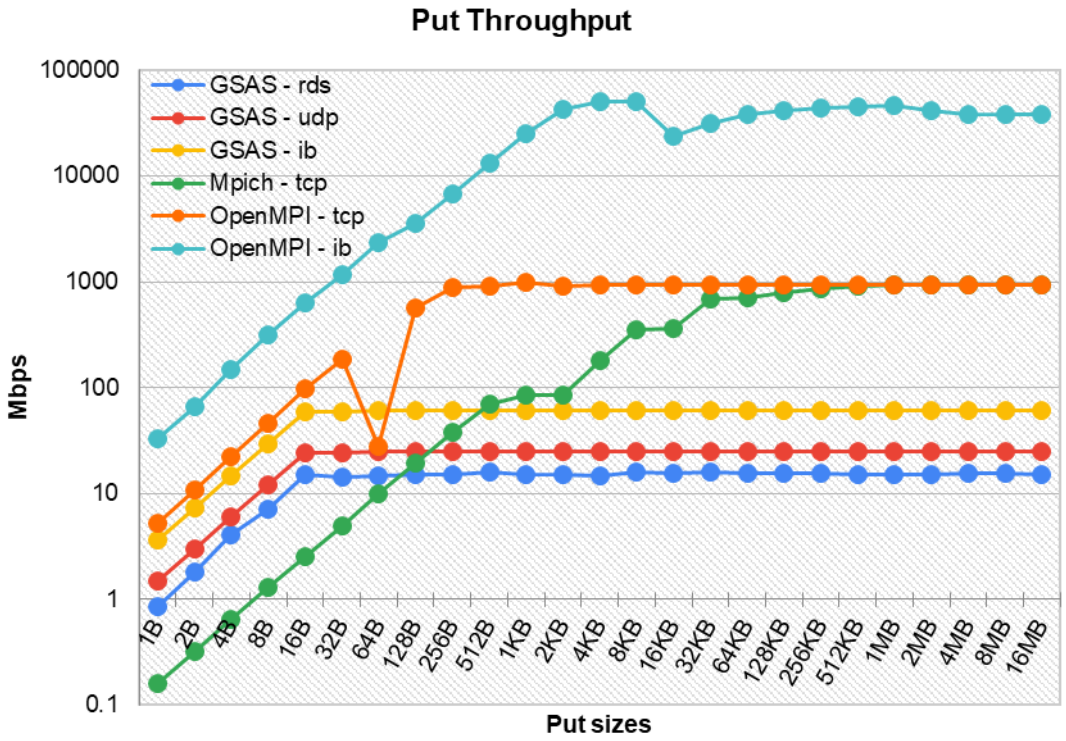


Figure 2.14 Throughput performance of the OpenSHMEM Put operations.

Figure 2.14 displays the throughput of PUT operations. Similarly to Figure 2.13, the IB implementation of GSAS stays on par with OpenMPI up to sizes of 16B and degrades linearly afterwards. OpenMPI's latency on PUT operations is lower than the network's capability. This phenomenon is due to their optimized PUT operations, in which they use both multi-threaded transmissions as well as multiple connections per pair of nodes. For the Ethernet cases, GSAS's design limitation persist on both UDP and RDS transmissions displaying the same behaviour as IB. MPICH follows a similar fixed sized packet transmission for packets with size less than 512 bytes, and although it starts with a worse performance than GSAS, it manages to outperform it for packet sizes greater than 128 bytes. OpenMPI's TCP manages to outperform both GSAS and MPICH for all packet sizes.

2.2.2 Evaluation of atomic operations

Moving on to our next set of experiments, we evaluate the performance of atomic operations defined by OpenSHMEM for GSAS, OpenMPI and MPICH. We evaluate the Compare&Swap operations, Fetch&Add and AtomicAdd operations. For each operation, we perform 10^6 operations both for local and remote nodes, and we measure the average latency of each operation. For measuring the performance of local atomic operations, we utilize two processes running on the same node and performing atomic operations on local shared memory. For the case of atomic operations on remote memory, we utilize two processes running on two remote nodes and performing atomic operations on remote shared memory.

Figure 2.15 displays the latency performance of Compare&Swap operations. The vertical axis shows the average latency measured in microseconds. Our horizontal axis is split in two parts. The first one contains the results for atomic operations on local memory and the second one for atomic operations on remote shared memory. In the case of atomic operations performed locally, GSAS manages to achieve extremely low latency regardless of the underlying networking protocol used. This is due to the fact that in this case GSAS applies the atomic operations by avoiding to utilize the network stack. More specifically, in GSAS, processes that are spawned on the same node are able to communicate by utilizing shared memory and avoiding communication via the network stack. Both OpenSHMEM

implementations provided by the MPI environment seem to lack this feature and perform local operations performed through the network stack. This is the reason the performance varies depending on the underlying networking protocol. On the remote cases, GSAS's IB implementation remains ahead, being 33% better than OpenMPI's performance. Whenever GSAS utilizes UDP, it manages to outperform both OpenMPI and MPICH, while the RDS implementation is slightly better than MPICH and worse than OpenMPI. This is due to the limitation of our testing environment's RDS is implemented over Ethernet instead of IB.

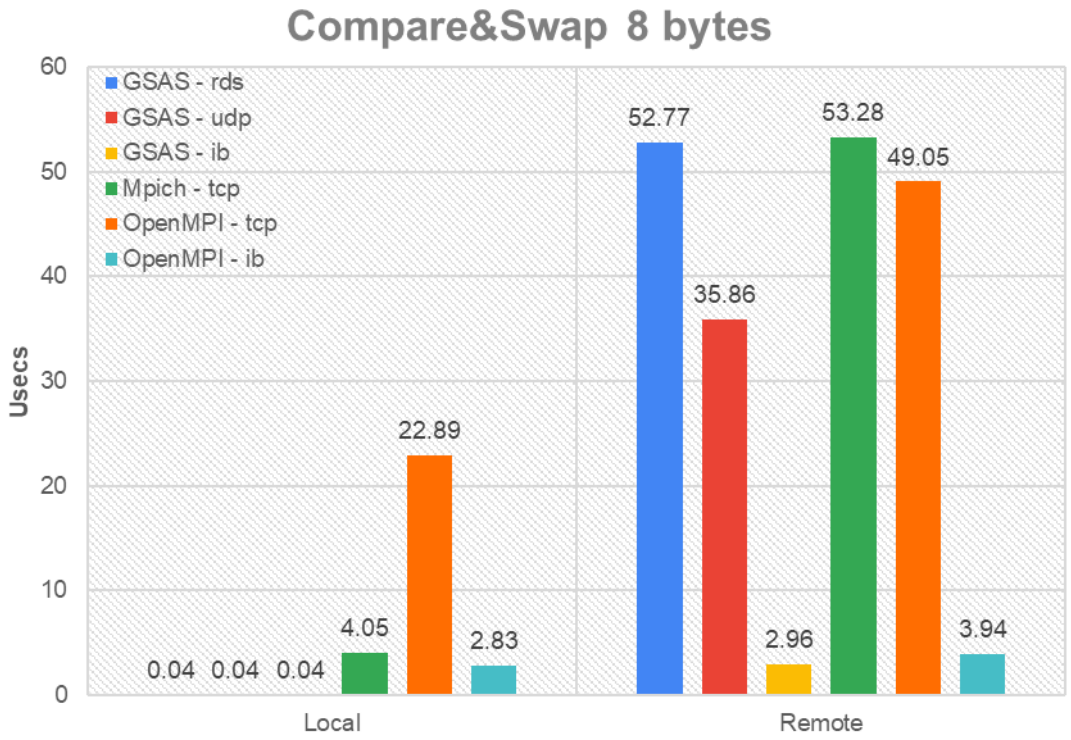


Figure 2.15 Latency of the OpenSHMEM Compare&Swap operations.

In Figure 2.16, we measure the performance of the Fetch&Add operations. The vertical axis shows the latency measured in microseconds, while the horizontal axis displays the various test cases. For our local operations, similarly to Figure 2.15, GSAS performs better than the OpenSHMEM implementations provided by OpenMPI and MPICH regardless of the underlying networking protocol used. Both OpenMPI and MPICH are utilizing the network stack for inter-process communication, and they are suffering from the overheads that a network stack imposes on local communication. In the case of performing atomic operations on remote memory, the IB implementation of GSAS outperforms OpenMPI by achieving 35% better latency. Moreover, the UDP implementation of GSAS achieves lower latency than both OpenMPI and MPICH, while the RDS implementation has almost equal performance to MPICH and slightly worse performance than OpenMPI.

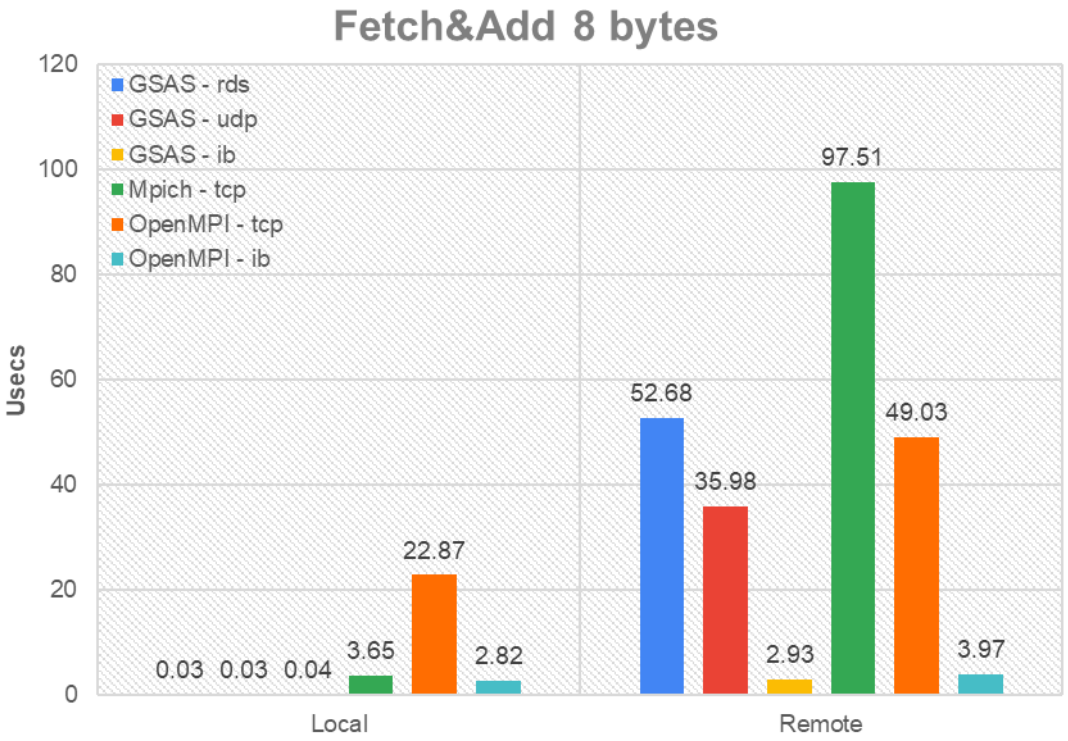


Figure 2.16 Latency of the OpenSHMEM Fetch&Add operations.

Figure 2.17 displays the performance of the AtomicAdd operation. The vertical axis displays the latency of operations measured in microseconds, while the horizontal axis displays the performance of the various implementations and network protocols. Similarly to Figure 2.15 and Figure 2.16, we observe that in the local operations, GSAS achieves better results compared to its competitors. Moreover, GSAS manages to outperform OpenMPI being 8x better in Ethernet transmissions and 6x better in IB by completing avoiding the communication through the network stack. On the remote cases, OpenMPI and MPICH perform better than the OpenSHMEM implementation provided by GSAS.

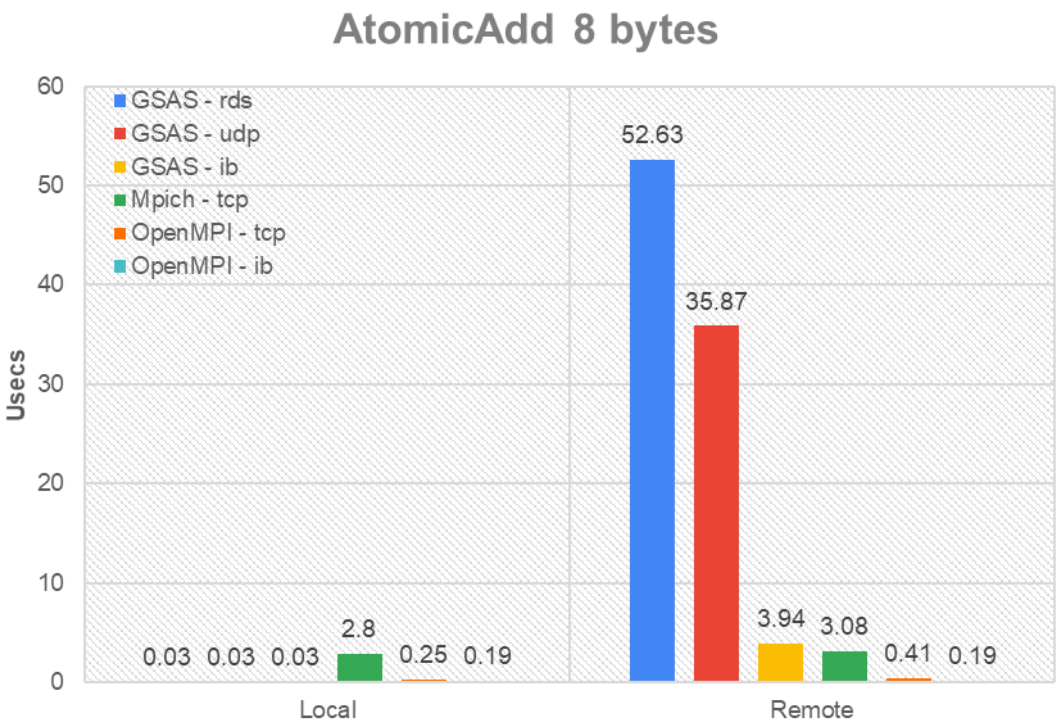


Figure 2.17 Latency of the OpenSHMEM Atomic operations.

3 IP over BXI Performance Improvements (ATOS)

3.1 Introduction

ATOS provides BXI (Bull eXascale Interconnect), an HPC interconnect allowing a large number of nodes to communicate efficiently within a cluster. BXI is a fabric network including NICs (one or several chips per node) and hardware switches connecting the NICs together, implementing the portals4 communication model.

BXI comes with a software stack that enables system services and user applications to exploit BXI hardware components (see Figure 3.1).

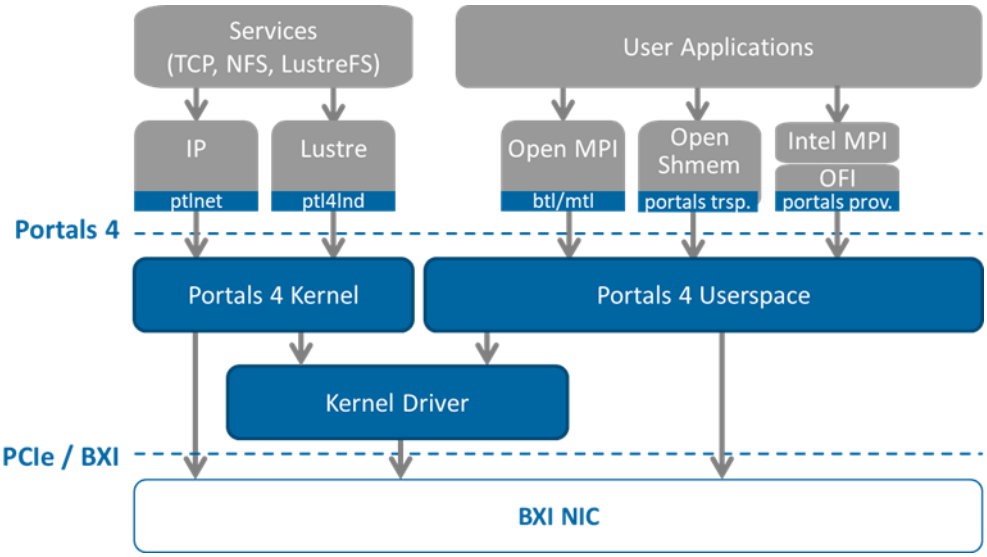


Figure 3.1 BXI software (compute) stack

Applications may offload most of their communication operations onto the BXI NIC with minimal processing of the host CPU, by using the portals4 user-space API directly (bxi-portals library). Alternately, applications may seamlessly leverage existing socket-based communication model using Ptlnet software component, which exposes a network interface (e.g ptl0) similar to usual ethernet devices. Any applicative IP traffic routed to/from ptl0 will be processed by Ptlnet component (IP over BXI).

Ptlnet is a Data Link Layer (L2) Linux network device (netdevice) over Portals4 and as such allows any Network Layer (L3) protocol such as IP to be transmitted over BXI network by the Linux kernel. In order to leverage existing tools such as ifup, ifdown, iptools, tcpdump and ethtool, the L2 frame format has been chosen to be the same as the Ethernet one.

This socket-based network is used by the nodes of an HPC system to communicate with storage services like NFS appliances, resource schedulers and job management services.

Since BXI network has no support for broadcast, it is emulated using a broadcast server ptlnet_bcastd. This allows any L2 broadcast protocol (ARP, DHCP, etc.) to be used over ptlnet.

Figure 3.2 presents a summary of how data / packets are processed across the different layers, from the application down to the BXI physical (hardware) layer:

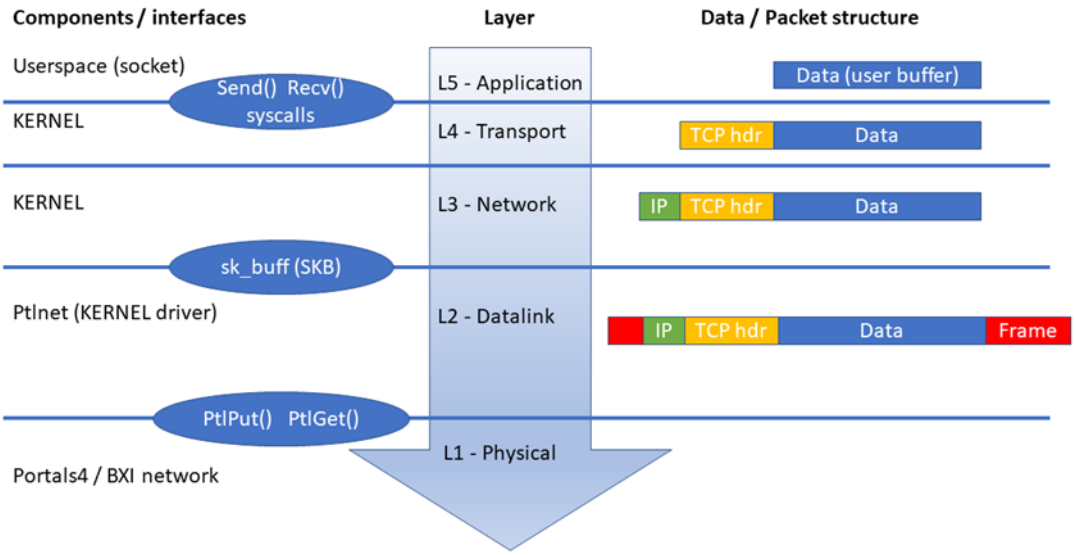


Figure 3.2 Data/Packet processing across the different network layers

The portals4 communication protocol is detailed in the following references:

https://www.researchgate.net/publication/336221144_The_Portals_42_Network_Programming_Interface
https://hal.archives-ouvertes.fr/hal-02972297/document

3.2 Current Ptlnet performance state

The current implementation of the Ptlnet module has a limited bandwidth performance. Indeed, in the best scenario with a 64KB MTU, the bandwidth peaks at 40Gbit/s. With a more realistic scenario where packets are routed through an IP gateway to and from a 1500B MTU external network, the bandwidth peaks at 5Gbit/s. This is not in line with expectations for a 100Gbit/s NIC. In order to make better use of the available raw BXI bandwidth, multiple aeras are worked on, from using multiple transmission and reception queues to leveraging TCP and UDP offloads.

3.3 Multiqueue

Netdev multiqueue interface had already been tested during previous Ptlnet performance investigations. It showed promising performance results on Intel Knights-Landing machines. This interface allows to use simultaneously multiple send/receive communication queues. It increases the overall processing capacity of ptlnet packets by allowing multiple processes to handle events and memory buffers and scattering them over multiple cpus.

This prototype has been integrated in Ptlnet module, and its performance has been tested on CER machines with AMD Milan cpus, with the following setup: AMD EPYC 7763 64-Core Processor, RHEL 8.4, bxi-module 2.2.5, bxi-portals 2.1.7, Ptlnet baseline 2.1.5.

The highest performance is achieved with 4 queues. Portals library lock contention decreases performance when more than 4 queues are used.

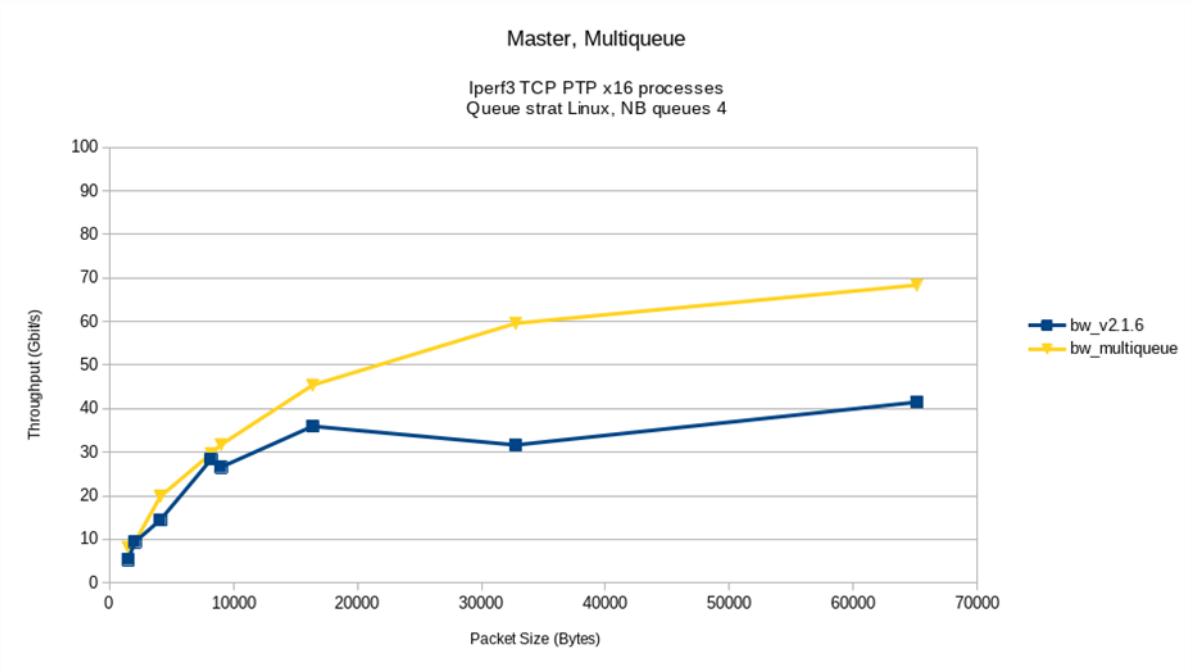


Figure 3.3 Multiqueue (4 queues) bandwidth improvement (yellow line) vs baseline (blue)

As shown by Figure 3.3, bandwidth is improved when packet size is greater than 16 KB, but we observe no significant gain for 1500 bytes packets. For this low packet size, on CER machines, the main bottleneck is the processing capacity of PUT IOVEC commands by the BXI NIC.

3.4 Queue selection policies

The algorithm selecting which queue is used to send a given packet should ensure a fair load balancing between queues while minimizing the overall packet processing cost.

Three selection policies have been tested:

- Round-Robin
- CPU
- Linux

Round-robin policy provides the best sharing of packets over all queues. However, it doesn't guarantee reception ordering of packets. The reordering of packets by Linux on receive side is a costly operation, which makes this policy less efficient overall.

CPU policy distributes packets according to the CPU id used to send them, i.e. the communication flow of a given applicative process will land on a given queue. As there are in general more processes involved than transmission queues, several flows may be assigned to the same queue. As such, sharing of packets between queues may not be optimal. However, the resulting performance is generally better than round-robin policy.

Linux policy is the default policy implemented in kernel. It creates a flow id from the IP:Port source and destination addresses, and each flow is assigned to a queue. It guarantees send ordering of packets, and Ptlnet will receive all packets of a flow on the same receive queue. The drawback of this policy is that the dispatch of flows over queues is not always optimal like the CPU policy. Linux policy provides the best overall performance, though.

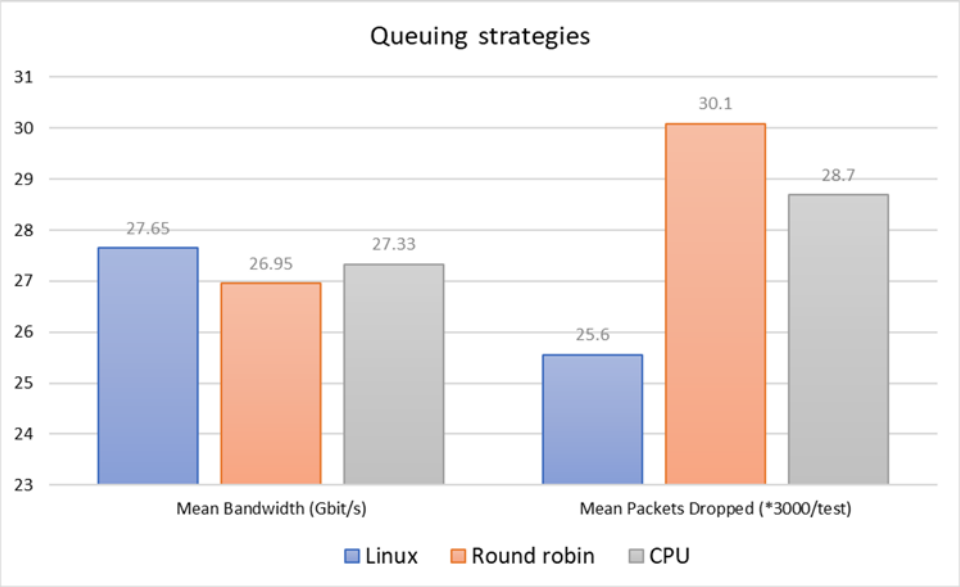


Figure 3.4 Mean bandwidth and mean number of dropped packets for each queue selection policy

As shown by Figure 3.4, on an average of 1188 iperf3 tests, the Linux kernel strategy generates less drops and provides the best bandwidth.

3.5 Transmission queues flow control

Ptlnet transmission (TX) queues need two resources to send a packet over the network: a transmission descriptor, and a slot in the BXI NIC transmission Command Queue (CQ).

When these resources are not available, several options may be envisioned to proceed:

- Silently drop the packet (it can be reemitted later in the framework of a TCP connection, for example)
- Give the packet back to the Linux kernel with NETDEV_TX_BUSY return code, to be reemitted later
- Disable the transmit queue as soon as resources get exhausted and reenable it when resources are available again.

Dropping packets is very detrimental to performance, as it negatively impacts TCP flow control and congestion algorithms. UDP flows are also affected because dropped packets will not be retransmitted.

Giving the packets back to Linux kernel was the solution implemented by Ptlnet so far. It is efficient but not optimal: it prevents packet dropping in most cases by inserting them back in the kernel transmit queue, but if there is no room left in that queue, packets are still dropped.

The third option is actually recommended by the Linux kernel and is the one retained in the improved Ptlnet implementation, as detailed below.

In order to guarantee the availability of TX descriptors, the queue is disabled as soon as the last TX descriptor is reserved to send a new packet. It will be reenabled when a Portals SEND or ACK event is received and the associated descriptor is freed, so that Linux kernel only send packets to queues where at least one TX descriptor is available. We also carried out experiments limiting the number of TX descriptors to 16, so that there is no risk to post a command on an already full BXI NIC TX CQ (a CQ has 16 slots).

Our performance measurements have shown that even with that limited number of 16 TX descriptors, performance stays correct as unnecessary PCI accesses and traffic of packets back and forth between kernel and Ptlnet are suppressed. Performance is similar to what was observed with large numbers of TX descriptors, only for low packet sizes we observe a slight loss of bandwidth (around 5 to 10%). The BXI NIC is slightly underused for small packet transfers, but packet transmission always succeeds.

However, better performance is obtained with 512 TX descriptors (128 per queue) when the NIC is fully exploited while reducing occurrences of TX CQ full to an acceptable level. This is why we implemented a limit of 512 TX descriptors instead of only 16.

3.6 Scatter-gather

The processing cost of a packet is composed of a fixed part and a part that depends on the size of the payload contained in the packet. The main size-dependent costs are the computation of layer 4 checksums (L4: TCP/UDP) and memory copies. Scatter-gather capability allows to optimize out such costs, both at transmission (TX) and at reception (RX) side.

3.6.1 TX Scatter-gather

TX scatter-gather capability allows to handle fragmented packets. It also allows to offload the packet linearization operation to the BXI NIC. Without offloading, the Linux kernel would have to resort to copy all fragments end-to-end into a linear buffer before putting them on emission.

Hardware linearization

BXI NIC features the IOVEC capability that allows to send fragmented data over the network. This capability was used so far by Ptlnet to add padding before and after payload, in order to build socket buffers (SKB) efficiently during reception of data on target machine. With the additional scatter-gather capability, BXI IOVECs are also used for packet linearization. Note that Linux kernel computes L4 checksums during linearization stage, so L4 checksums are no longer computed. This results in a shorter transmission path of packets in Linux kernel.

Software linearization

Performance measurements show that hardware linearization offload didn’t increase Ptlnet bandwidth. Indeed, IOVEC PUT commands execute slower than linear PUT on BXI NIC. IOVEC PUT requires more PCI transfers, first to access the IOVEC descriptor, then the actual data. The smallest packet sizes incur the greatest cost overhead of IOVEC PUT command, but the smallest memory copy overhead.

This is why a software linearization was implemented directly in Ptlnet module, allowing to use linear PUT commands for packets smaller than 32 KB. For packets larger than 32 KB, it is more efficient to use PUT IOVEC commands and avoid memory copies by the CPU.

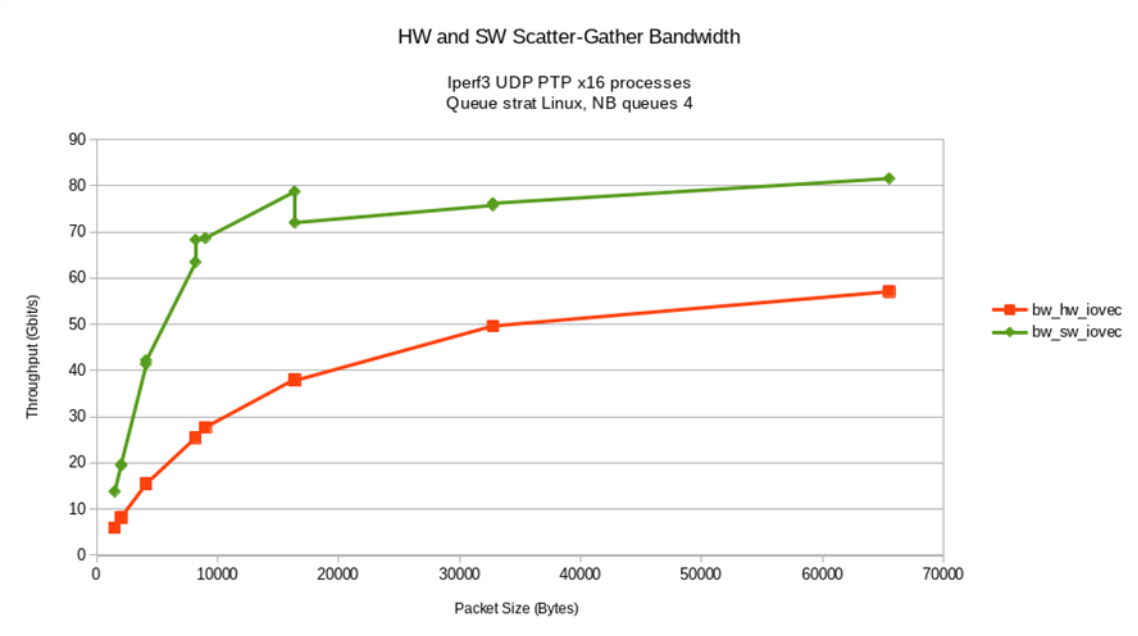


Figure 3.5 Hardware vs Software linearization

3.6.2 RX Scatter-gather

The use of fragmented RX socket buffer (SKB) allows to reduce the amount of padding sent on the network. In order to build a fragmented SKB, a new SKB is allocated, then headers of received packet are copied to it. Packet data are linked in the first fragment. This fragment doesn't require padding.

On BXI NIC, several consecutive packets may be received in the same memory buffer, since packet reception is done with matching entries using MANAGE_LOCAL option. Each packet needs to be aligned on a cache line, ie 64 bytes. This requires some padding at the end of packet data, but this padding is less (63 bytes max) than the SKB padding needed before this change (around 400 bytes), and it doesn't incur data transmission overhead.

This change also allows to ease the construction of packet at sending side. Some specific conditions also allow to shorten the emission path of packets in Ptlnet. For specific traffic patterns, Linux kernel generates linear packets. If their size is a multiple of 64 bytes, they can be sent as is using a PUT command without memory copies.

3.7 Performance evaluation

Ptlnet performance evaluation has been carried out on CER Milan nodes equipped with BXI V2 NIC version 1.3, rev4. The software setting consists of Atos BXI v2.4.1 software compute stack (bxi-portals v2.1.8, bxi-module v2.2.6) over RHEL8.4. Ptlnet performance is evaluated using 16 parallel Iperf3 benchmark processes.

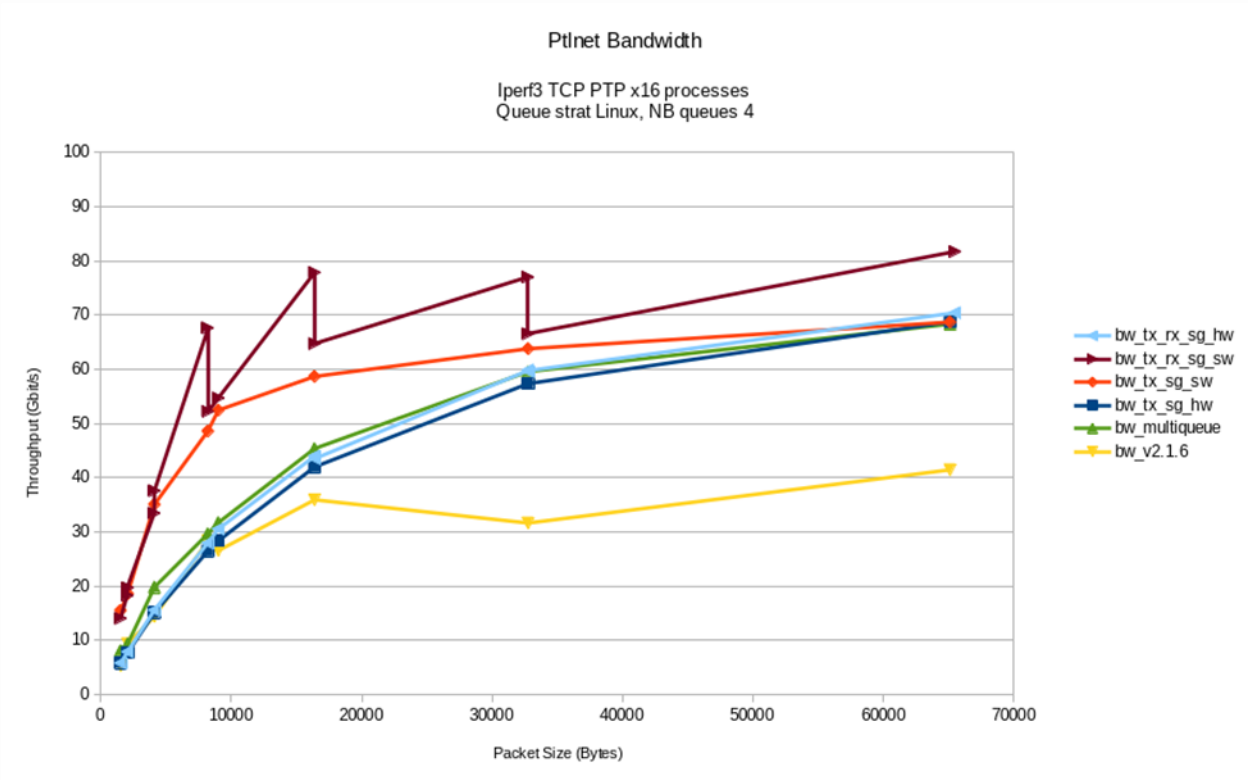


Figure 3.6 Ptlnet bandwidth improvements for successive Ptlnet prototypes

Figure 3.6 above displays incremental performance improvements associated with each Ptlnet code enhancement:

- Yellow line: baseline ptlnet v2.1.6, without change.
- Green line: Ptlnet with multiqueue enhancement
- Dark blue line: Multiqueue + TX Scatter-Gather using BXI IOVEC
- Orange line: Multiqueue + TX Scatter-Gather using software linearization
- Light blue line: Multiqueue + TX Scatter-Gather using BXI IOVEC + RX Scatter-Gather
- Purple line: Multiqueue + TX Scatter-Gather (software linearization) + RX Scatter-Gather

3.8 TCP/UDP offloads

3.8.1 TCP segmentation offload

This optimization aims at minimizing the fixed costs of packet processing by reducing the number of packets and increasing their size in Linux kernel. Such packets are called GSO (Generic Segmentation Offload); their size is up to 64 kB. If the MTU (Maximum Transfer Unit) of the physical link is lower, GSO packets have to be broken into smaller packets by the NIC before being sent over the network. They are rebuilt on the target NIC after transfer and given to the Linux kernel thanks to a mechanism named GRO (Generic Receive Offload).

We have implemented this optimization into Ptlnet to be able to transmit GSO packets directly, regardless of network layer MTU size. After transfer, these packets are given as is to the Linux kernel by Ptlnet on target side. BXI NIC doesn't need to break/rebuild packets because it can transfer more than 64kB at once, without being limited by the network MTU. Since it is more efficient to transfer bigger packets, this optimization allows to reach 70Gbits/s for all MTU sizes during TCP transfers, as shown in Figure 3.7.

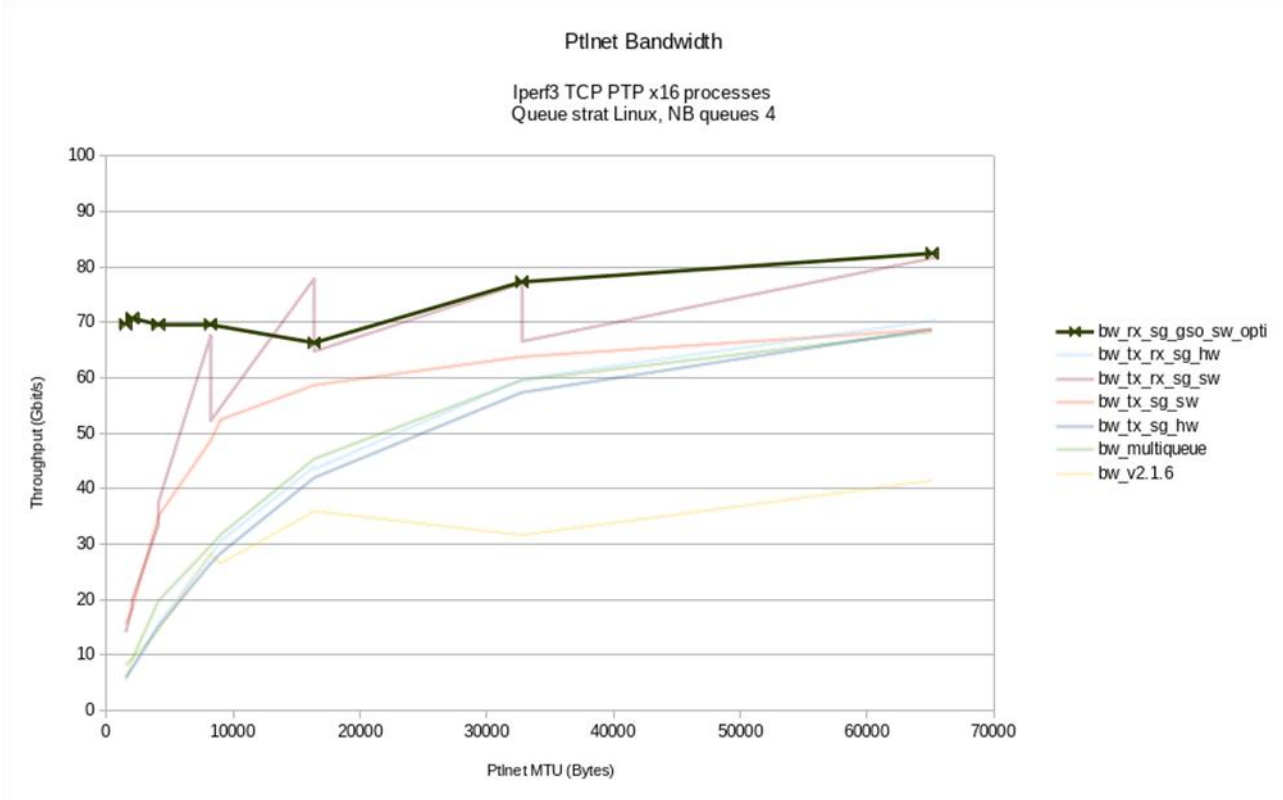


Figure 3.7 Successive optimizations + GSO

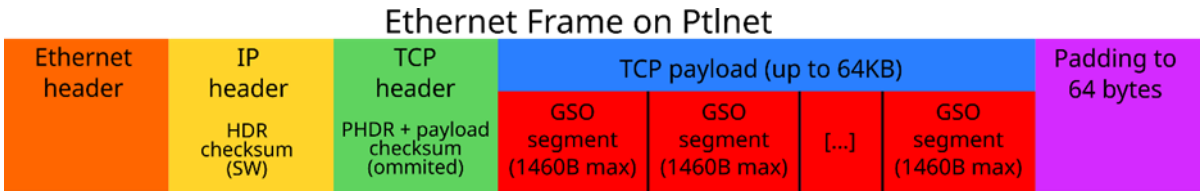


Figure 3.8 representation of a GSO TCP packet on a Ptlnet network with 1500 bytes MTU

Figure 3.8 displays a representation of a GSO TCP packet on a Ptlnet network with 1500 bytes MTU.

3.8.2 TCP/UDP checksum remote offloading

As stated above, Linux kernel doesn't compute L4 checksums at send time since scatter-gather is implemented in Ptlnet. It has no effect on traffic inside the same Ptlnet network. Ptlnet marks packets as valid when they are received without checksums so that Linux always accepts them. This does not hold across an IP gateway to a different network. The packets would be dropped on the target machine outside Ptlnet network because of their invalid checksum.

The packets leaving Ptlnet network should have valid checksums. However, BXI NIC doesn't have hardware acceleration to compute them quickly. Two workarounds may be envisioned:

- Tell explicitly the Linux kernel to compute L4 checksums at send time. It was the behavior of Ptlnet so far.
- Offload the computation of checksums to the IP gateway.

We implemented the second solution by marking the checksums of outbound packets as partially computed. This forces the Linux kernel on the gateway to recompute L4 checksums before giving packets to the target NIC driver. If that NIC is able to offload checksum calculation from the kernel, Linux kernel will use this capability, and the processing of packets will be more efficient on the gateway.

3.9 Ptlnet performance with IP gateway

Ptlnet performance over IP gateway has been measured using two setups:

- Infiniband network with 1500 bytes MTU
- Infiniband network with 2044 bytes MTU

The hardware configuration is the following:

- CER nodes with BXI NIC v1.3 rev4 for Ptlnet network
- X431 node with BXI NIC v1.3 rev4 and Infiniband ConnectX-6 NIC for IP gateway
- One node on Infiniband network with Infiniband ConnectX-6 NIC and two AMD EPYC 7402 CPUs

Several Ptlnet optimizations have been tested:

- Multiqueue (4 queues TX et RX)
- TX Scatter-Gather with software linearization for packets less than 32Ko and BXI IOVECs for packets larger than 32Ko
- RX Scatter-Gather
- TCP segmentation offload
- TCP/UDP checksum remote offload

The resulting bandwidth is not symmetric, because GSO packets transmitted by Ptlnet have a size close to 64 kB, while those received from Infiniband network have a size between 26 kB and 36 kB, which is detrimental to ptlnet bandwidth.

Figure 3.9 shows Infiniband network to/from Ptlnet network traffic bandwidth with all Ptlnet enhancements enabled:

- Dark green line: Ptlnet to Infiniband (outgoing), 1500 bytes MTU
- Purple red line: Infiniband to Ptlnet (incoming), 1500 bytes MTU
- Light green line: Ptlnet to Infiniband (outgoing), 2044 bytes MTU
- Orange line: Infiniband to Ptlnet (incoming), 2044 bytes MTU

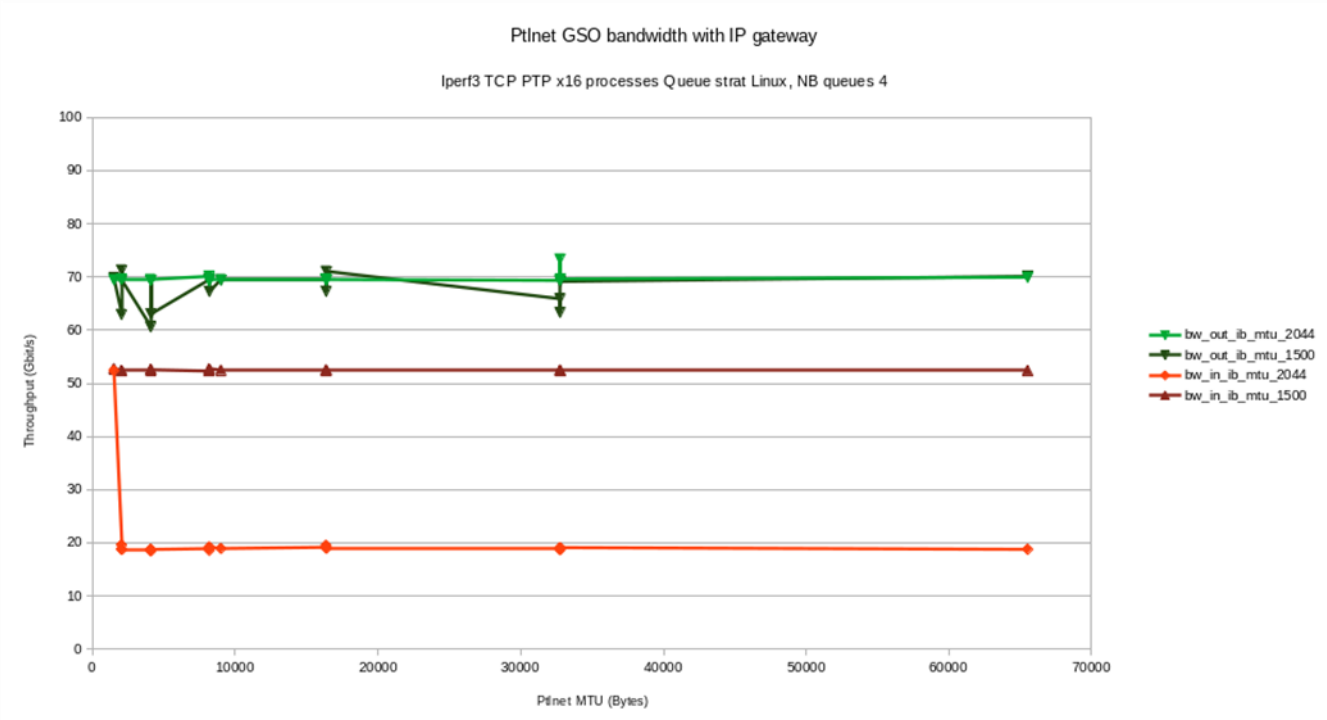


Figure 3.9 Ptlnet network to/from Infiniband network traffic bandwidth through IP gateway

3.10 **Conclusion**

As a summary, the enhancements described in this document and implemented in Ptlnet code have brought significant performance improvements. TCP/UDP bandwidth has increased from 10-40 Gbits/s, strongly MTU dependent, to 70-80 Gbits/s almost independent from the MTU. This increased performance level and stability is in line with expectations for a 100 Gbit BXI NIC in HPC environment.

These Ptlnet enhancements have been integrated and delivered as part of the Atos BXI software compute stack version 2.5 (July 2022 release).

4 Conclusion

In this deliverable, we evaluated how Infiniband-based solutions perform under a distributed persistent key-value store framework named Tebis. In addition, we evaluated their performance under GSAS environment. Additionally, in T2.2, we evaluated and improved the performance of IPoverBXI.
Our key findings are: 1) with regards GSAS applications, the bandwidth of well-known HPC interconnects suffers in case that the traffic consists of a high number of small-size packets (i.e., a few hundreds of bytes at most); and 2) with regards Tebis, the same trends exist: systems cannot easily saturate the available network throughput with small (64 B – 512 B) message sizes due to saturation of the packet rate of the network card. Regarding IPoverBXI, our software enhancements have increased the TCP/UDP bandwidth from 10-40 Gbits/s to 70-80 Gbits/s almost independent from the MTU. These results are useful in planning new enhancements and optimizations in RED-SEA, while developing new network interfaces based on European technologies.

5 Acronyms and Abbreviations

Term	Definition
API	Application Programming Interface
ARP	Address Resolution Protocol
BXI	Bull Exascale Interconnect
CER	Code name of the Atos blade server used as compute node
CPU	Central Processing Unit
CQ	Command Queue
DDR	Double Data Rate memory
DHCP	Dynamic Host Configuration Protocol
GRO	Generic Receive Offload
GSAS	Global Shared Address Space
GSO	Generic Segmentation Offload
HPC	High Performance Computing
IOVEC	Input/Output Vector
IP	Internet Protocol
MPI	Message Passing Interface
MTU	Maximum Transfer Unit
NFS	Network File System
NIC	Network Interface Controller
NVMe	NonVolatile Memory Express
NUMA	Non-Uniform Memory Access
PCI	Peripheral Component Interconnect
RDS	Reliable Datagram Socket
RHEL	RedHat Enterprise Linux
RX	Reception
SKB	Socket Bufer
SSD	Solid State Drive
TCP	Transmission Control Protocol
TX	Transmission
UCX	Unified Communication - X framework library
UDP	User Datagram Protocol
X431	Code name of the Atos rack server used as gateway node
LSM-tree	Log Structured Merge Tree
RoCE	RDMA over Converged Ethernet

Table 3: Acronyms and Abbreviations

References

(2022). Retrieved from Apache HBase: https://hbase.apache.org/

Batsaras, N., Saloustros, G., Papagiannis, A., Fatourou, P., & Bilas, A. (2020). VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores. *VAT: Asymptotic Cost Analysis for Multi-Level Key-Value Stores*.

Borthakur, D., & others. (2008). HDFS architecture guide. *Hadoop apache project, 53*, 2.

Cao, Z., Dong, S., Vemuri, S., & Du, D. H. (2020, February). Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. *18th USENIX Conference on File and Storage Technologies* (pp. 209–223). Santa: USENIX Association.

Chan, H. H., Li, Y., Lee, P. P., & Xu, Y. (2018). HashKV: Enabling Efficient Updates in KV Storage via Hashing. *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (pp. 1007–1019). Berkeley: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=3277355.3277451

Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., & Lauren, S. (2010). openshmem: Shmem for the pgas community. *Proceedings of the Fourth Conference on Partioned Global Address Space Programming Model, PGAS 10.* New York, NY.

Chodorow, K. (2013). *MongoDB: The Definitive Guide (second edition).*

Commission Européenne, 7. C. (2016, 10). *Grant Agreement.* Retrieved from https://shirocommunity.bull.com/ext/fpop/clouddbappliance/shareddocuments/GRANT/Grant%20Agreement-732051-CloudDBAppliance.pdf

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing* (pp. 143–154). New York, NY, USA: ACM. doi:10.1145/1807128.1807152

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., . . . Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review, 41*, 205–220.

Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., & Strum, M. (2017). Optimizing Space Amplification in RocksDB. *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings.* Retrieved from http://cidrdb.org/cidr2017/papers/p82-dong-cidr17.pdf

Facebook. (2018). BlobDB. *BlobDB*.

Gao, Y., Li, Q., Tang, L., Xi, Y., Zhang, P., Peng, W., . . . Wu, J. (2021, April). When Cloud Storage Meets RDMA. *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)* (pp. 519–533). USENIX Association. Retrieved from https://www.usenix.org/conference/nsdi21/presentation/gao

Kalia, A., Kaminsky, M., & Andersen, D. G. (2016). Design Guidelines for High Performance RDMA Systems. *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, (pp. 437–450).

Kallimanis, N. D., Marazakis, M., & Chrysos, N. (2019). GSAS: A Fast Shared Memory Abstraction with Minimal Hardware Support. *EuroExaScale workshop - HiPEAC 2019.* Valencia, Spain.

Lakshman, A., & Malik, P. (2010, April). Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev., 44*, 35–40. doi:10.1145/1773912.1773922

Li, Y., Liu, Z., Lee, P. P., Wu, J., Xu, Y., Wu, Y., . . . Cui, Q. (2021, July). Differentiated Key-Value Storage Management for Balanced I/O Performance. *2021 USENIX Annual Technical Conference (USENIX ATC '21)* (pp. 673–687). USENIX Association.

Lu, L., Pillai, T. S., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2016, February). WiscKey: Separating Keys from Values in SSD-conscious Storage. *14th USENIX Conference on File and Storage Technologies (FAST 16)* (pp. 133-148). Santa: USENIX Association. Retrieved from https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

Matsunobu, Y., Dong, S., & Lee, H. (2020, August). MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow., 13*, 3217–3230. doi:10.14778/3415478.3415546

O'Neil, P., Cheng, E., Gawlick, D., & O'Neil, E. (1996, June). The Log-structured Merge-tree (LSM-tree). *Acta Inf., 33*, 351–385. doi:10.1007/s002360050048

Papagiannis, A., Saloustros, G., González-Férez, P., & Bilas, A. (2018). An Efficient Memory-Mapped Key-Value Store for Flash Storage. *Proceedings of the ACM Symposium on Cloud Computing* (pp. 490–502). New York, NY, USA: ACM. doi:10.1145/3267809.3267824

Project, 7. C. (2016, 11 07). *Consortium Agreement.* Retrieved from https://shirocommunity.bull.com/ext/fpop/clouddbappliance/shareddocuments/Consortium%20Agreement/DESCA%20CloudDBAppliance.Final.2016-11-07.pdf

Ren, J. (2016). YCSB-C. *YCSB-C*. GitHub.

Singhvi, A., Akella, A., Anderson, M., Cauble, R., Deshmukh, H., Gibson, D., . . . Vahdat, A. (2021). CliqueMap: Productionizing an RMA-Based Distributed Caching System. *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (pp. 93–105). New York, NY, USA: Association for Computing Machinery. doi:10.1145/3452296.3472934

Taleb, Y., Stutsman, R., Antoniu, G., & Cortes, T. (2018). Tailwind: Fast and Atomic RDMA-based Replication. *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (pp. 851–863). Berkeley: USENIX Association. Retrieved from http://dl.acm.org/citation.cfm?id=3277355.3277438

Tsai, S.-Y., & Zhang, Y. (2017). LITE Kernel RDMA Support for Datacenter Applications. *Proceedings of the 26th Symposium on Operating Systems Principles* (pp. 306–324). New York, NY, USA: Association for Computing Machinery. doi:10.1145/3132747.3132762

Vardoulakis, M., Saloustros, G., Gonzalez-Ferez, P., & Bilas, A. (2022). Tebis: Index Shipping for Efficient Replication in LSM Key-Value Stores. *Eurosys.* Rennes, France.

Xanthakis, G., Saloustros, G., Batsaras, N., Anastasios, P., & Bilas, A. (2021). Parallax: Hybrib Key-Value Placement in LSM-based Key-Value Stores. *Proceedings of the ACM Symposium on Cloud Computing.* New York, NY, USA: ACM.