Tyche: An efficient Ethernet-based protocol for converged networked storage

Pilar González-Férez[1]
Institute of Computer Science (ICS) - FORTH
Heraklion, Greece
Email: pilar@ditec.um.es

Angelos Bilas[2]
Institute of Computer Science (ICS) - FORTH
Heraklion, Greece
Email: bilas@ics.forth.gr

***Abstract*—Current technology trends for efficient use of infrastructures dictate that storage converges with computation by placing storage devices, such as NVM-based cards and drives, in the servers themselves. With converged storage the role of the interconnect among servers becomes more important for achieving high I/O throughput. Given that Ethernet is emerging as the dominant technology for datacenters, it becomes imperative to examine how to reduce protocol overheads for accessing remote storage over Ethernet interconnects.**

In this paper we propose Tyche, a network storage protocol directly on top of Ethernet, which does not require any hardware support from the network interface. Therefore, Tyche can be deployed in existing infrastructures and to co-exist with other Ethernet-based protocols. Tyche presents remote storage as a local block device and can support any existing filesystem. At the heart of our approach, there are two main axis: reduction of host-level overheads and scaling with the number of cores and network interfaces in a server. Both target at achieving high I/O throughput in future servers. We reduce overheads via a copy-reduction technique, storage-specific packet processing, preallocation of memory, and using RDMA-like operations without requiring hardware support. We transparently handle multiple NICs and offer improved scaling with the number of links and cores via reduced synchronization, proper packet queue design, and NUMA affinity management.

Our results show that Tyche achieves scalable I/O throughput, up to 6.4 GB/s for reads and 6.8 GB/s for writes with 6 x 10 GigE NICs. Our analysis shows that although multiple aspects of the protocol play a role for performance, NUMA affinity is particularly important. When comparing to NBD, Tyche performs better by up to one order of magnitude.

I. Introduction

Today, storage in datacenters is typically a separate tier from application servers and access happens mostly via a storage area network (SAN). Current efforts to improve efficiency of datacenters in terms of capital expenses, e.g. reduced energy consumption, and operational expenses, e.g. less expensive storage, dictate bringing storage closer to applications and computation by converging the two tiers. *Converged storage* advocates placing storage devices, most likely performance-oriented devices, such as solid state disks or non-volatile memory, in all servers where computation occurs and adapting the current I/O stack to the new model. In the converged storage model many storage accesses require crossing the network for various reasons: additional storage capacity, reliability, and sharing. Therefore, storage requests are exchanged between all compute servers and the network protocol used plays an important role.

TABLE I. Network storage protocol and generic network protocol based on Ethernet.

	Software	Hardware
Storage	NBD, iSCSI AoE, FCoE	iSER, SRP gmblock
Generic	PortLand [1]	iWARP, RoCE JNIC

Today, it is generally accepted that there are many advantages to using Ethernet-based physical networks for storage as well. A single Ethernet network for network and storage data traffic reduces cost and complexity. In the past, there has been a lot of research work on interconnects, such as Infiniband, which scale and impose low overheads. However, it is unlikely that such interconnects will dominate and displace Ethernet in the datacenter. For this reason, the network protocol used on top of Ethernet plays a significant role in achieving high efficiency for remote storage access.

Table I provides a summary of storage-specific and general-purpose network protocols based on Ethernet. We further classify these protocols in two categories, whether they need hardware support or not. Software-only protocols typically exhibit relatively low throughput for small requests and incur high overheads. A main reason is that they mostly either use TCP/IP or they are not optimized for storage. TCP/IP inherently incurs high overheads due to its streaming semantics. On the other hand, hardware-assisted protocols usually obtain maximum link throughput at lower CPU overheads, but they require custom NICs or other extensions to the underlying interconnect, which is a significant impediment for deployment and adoption.

In this paper, we examine the issues associated with networked storage access over Ethernet, and we design Tyche a network storage protocol that achieves high efficiency, without requiring any hardware assistance. Our proposal can be deployed in existing infrastructures and to co-exist with other Ethernet-based protocols. To the best of our knowledge, our approach is the first to achieve 90% of link efficiency for 16 kB request sizes without any specialized hardware support.

[1]Also with the Department of Computer Engineering, University of Murcia, Spain

[2]Also with the Department of Computer Science, University of Crete, Greece

978-1-4799-5671-5/14/$31.00

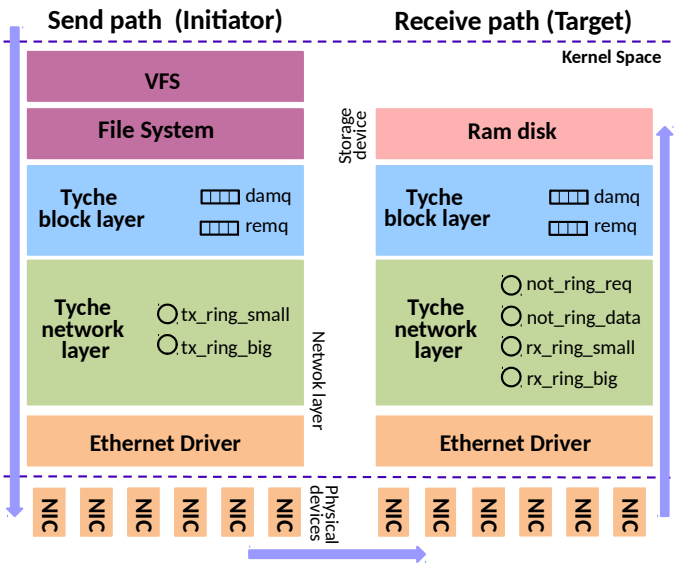


Fig. 1. Overview of send and receive path.

Given current technology trends, the expectation is that future servers will host hundreds of cores and hundreds of GBits/s of network throughput and will mainly access remote storage devices. Already today, typical servers employ 64 cores and multiple 10 GBits/s Ethernet NICs. Tyche provides low-overhead communication by carefully considering protocol operations, structures in the common path, memory management, and synchronization required to access networked storage. We identify two main challenges:

1) Host-overhead for remote storage access. The I/O throughput provided by current network storage protocols, such as Network Block Device (NBD), is limited to a small percentage of the network throughput. For instance, NBD achieves around 600 MB/s for sequential reads and writes (Section IV-A), which is far from the 1.2 GB/s provided by the NIC. In addition, NBD requires requests of 1 MB in size and incurs a 100% of CPU utilization to provide this 600 MB/s; in contrast, Tyche achieves the same throughput at 4 kB requests and only 50% CPU utilization.
2) Transparent use of multiple NICs. Current network storage protocols are only able to use a single NIC. Transparently sharing multiple NICs from many cores increases synchronization and memory management overhead. For instance, if a fine-grain approach is used for assigning NICs to cores at the packet level, this leads to significant synchronization at high network speeds.

To reduce host overheads, Tyche efficiently maps I/O requests to network messages, pre-allocates memory, and handles NUMA affinity. The preallocation and placement of buffers is done per connection. We use a copy reduction technique based on virtual memory page remapping to reduce packet processing cost. Indeed, Tyche avoids all copies for write requests by taking advantage of storage semantics, but requires a single copy for reads at the initiator side, due to OS kernel semantics for buffer allocation.

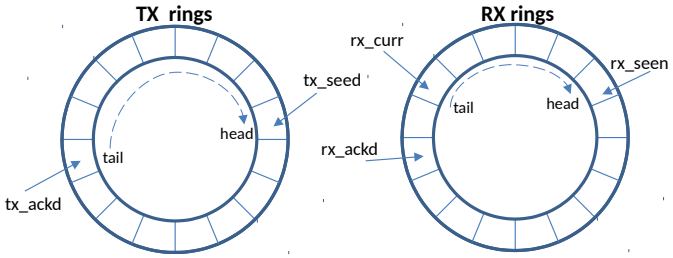


Fig. 2. The network ring protocol structure used in the send and receive path.

To limit synchronization, Tyche uses the familiar connection abstraction to privatize the main structures (rings and memory buffers) and thread processing context, which are typically shared across in today protocols. Mapping cores to connections and connections to links allows for flexibility managing the amount of throughput available to each application.

Our results show that Tyche achieves scalable throughput of up to 6.4 GB/s and 6.8 GB/s for sequential reads and writes, respectively, on 6x10Gbits/s network devices. Results also show that, to achieve maximum throughput, NUMA affinity should be taken into account, otherwise, throughput drops by up to 2x. When comparing Tyche against NBD, for sequential reads and writes, our proposal outperforms NBD by one order of magnitude when using 6 NICs, and by about 2x with a single NIC. For actual applications, Tyche significantly increases throughput, for instance, for Psearchy Tyche achieves 2x and 8x better throughput compared to NBD, when using 1 and 6 NICs, respectively.

The rest of this paper is organized as follows. Sections II and III present Tyche and the main decisions taken to achieve the previous issues. Section IV discusses our results. Section V describes related work and Section VI concludes this work.

II. System Design

The overall design of Tyche is depicted in Figure 1. Tyche is a connection-oriented protocol that allows the creation of multiple connections between the client (initiator) and the server (target). Each Tyche connection uses its own private resources to minimize synchronization for shared structures and to allow scaling with the number of NICs and cores. Tyche handles several NICs transparently and each connection can span a single or multiple NICs.

To initialize the network stack, Tyche opens several connections, one per available NIC, between the initiator and the target. For each connection, the initiator and target exchange information about buffers and resources. Then the initiator is ready to receive I/O requests the network storage device. The new storage device can be mounted and used as a regular block device, and can support any existing file system.

A. Network messages

Tyche supports two different message types, one for transferring I/O requests/completions and one for transferring data. An I/O request/completion message uses a single packet. Request packets are small, less than 100 bytes in size, and they are transferred in small Ethernet frames of the corresponding size.

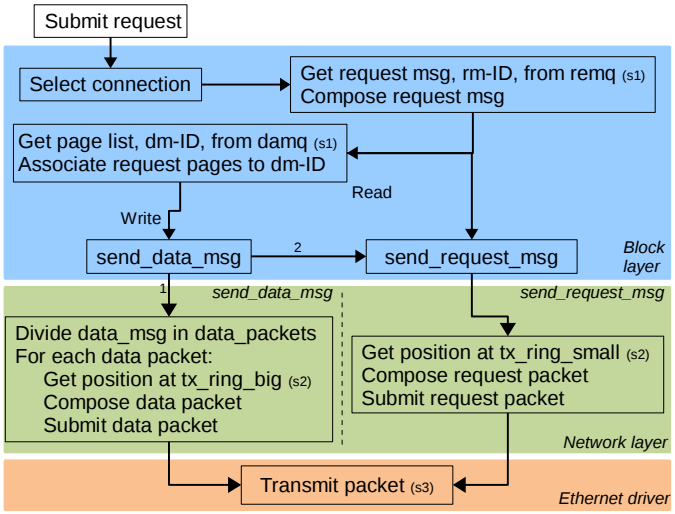


Fig. 3. Overview of the send path at the initiator.

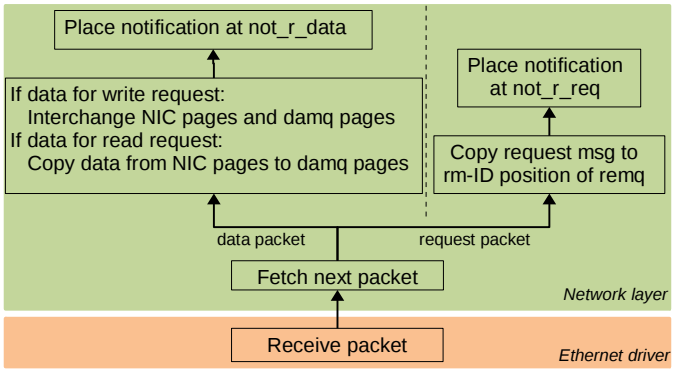


Fig. 4. Overview of the receive path in the target at the network layer.

In this work, we do not batch I/O requests to avoid increasing I/O response time. Data messages are sent via RDMA-type messages by using scatter-gather lists of memory pages (I/O buffers). The corresponding (data) packets are transferred in separate Jumbo Ethernet frames of 4 or 8 kB. An Ethernet jumbo frame of 9000 bytes carries two 4 kB pages, so, a data message for an I/O request of N pages is split into $\frac{N}{2}$ data packets.

Tyche allows out-of-order transfer and delivery of packets over multiple links or different network paths. For this reason, the Ethernet header of each data packet includes the message id, the position of the packet in the message, and the number of packets that compose the message.

B. Main data structures

The target allocates memory for each new request and data message. To reduce the overhead of memory management, Tyche uses two separate and pre-allocated queues (Figure 1), one for request messages (`remq`) and one for data messages (`damq`). `damq` contains lists of pre-allocated pages for sending and receiving data messages. The lists point to memory pages that the target uses for issuing write and read requests to the local device. Although both queues are allocated by the target, they are managed directly by the initiator (during the connection negotiation phase, the queue information is exchanged, so the initiator knows all necessary identification handlers of the target queues). The initiator specifies fixed positions in the message header for the target queues. For instance, the initiator specifies on-behalf of the target the position (pages) where data packets has to be placed when they arrive (for writes), and the target uses these pages for submitting the regular I/O write requests. Completion messages are prepared in the same buffer where the corresponding request was received. The initiator also uses similar queues with the exception that `damq` lists refer to pointers of pages, because the pages are already provided by the user I/O requests.

To process messages and packets, our protocol uses three rings (Figures 1 and 2), one for transmitting, `TX_ring`, one for receiving, `RX_ring`, and one for notifications, `Not_ring` (not shown in Figure 2). Since Tyche handles two kinds of packets (request and data packets) for each ring, it also has two ring instances. Therefore, a request packet is transmitted using the `TX_ring_small`, it is received in the `RX_ring_small`, and the corresponding notification is placed in the `Not_ring_req`. In the same way, a data packet uses the `TX_ring_big`, `RX_ring_big` and `Not_ring_data` rings.

To reduce synchronization, each ring uses only two pointers (Figure 2), a head and a tail. The head points to the last packet sent or received. The tail points, for the transmission ring, to the last packet currently acknowledged by the remote node, and, for the receive and notification rings, to the last packet currently processed. In addition, the receive ring has a third pointer, `rx_ackd`, to the last packet currently acknowledged towards the remote node. Other implementations usually use more pointers for handling rings. For example, the transmission ring has a third pointer for controlling packets currently sent by the NICs. However, this pointer can be avoided, because a position can only be re-used when its corresponding ACK has been received, and if a packet is acked, it is because the NIC has sent it. This approach delays certain protocol processing, but allows us to reduce synchronization overhead.

Each cell of the transmission ring has two fields to denote its state: the packet is ready to be sent or it has been sent. They are used when the NIC is busy and sending packets has to be delayed. Each cell of the receive ring also has a field to denote that the packet has arrived and to control packets that arrive several times due to re-transmissions. These two fields are updated by atomic operations.

C. Networked I/O path

Figures 3, 4, 5, 6, and 7 summarize the flow path of our network storage protocol. For simplicity, we do not include error handling and retransmission paths. Numbers on the arrows denote the order of execution when several actions are run after a previous one. In Figure 3, we mark some actions with labels s1, s2, and s3 to indicate that a synchronization point is required to execute the corresponding action.

At the initiator side, for each new I/O request, Tyche selects one connection. On this connection, Tyche gets one request message (rm-ID) in `remq` and one position (dm-ID) in `damq`. The initiator composes the request message and associates the

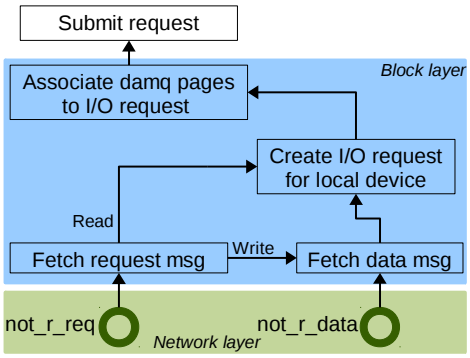


Fig. 5. Overview of the receive path in the target at the block layer.

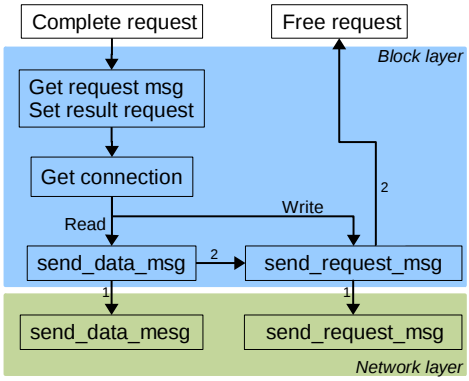


Fig. 6. Overview of the completion path in the target.

pages of the I/O request to the dm-ID position in `damq`. For writes, these pages are sent to the target, and for reads, upon arrival, data is directly placed in the proper pages.

Tyche can operate in two different modes. In the first, "inline" mode (Figure 1), the application context issues I/O requests to the target, without requiring any context switch in the issue path. In the second, "queue" mode, regular I/O requests are inserted in a Tyche queue, and several Tyche threads dequeue these I/O requests and issue them to the target. With the queue mode, the issuing context blocks just after enqueuing the request. We evaluate both modes in Section IV-D. Figure 3 shows how the initiator issues a network I/O request in the inline mode. The queue mode is similar, with local I/O requests being inserted in a queue and a Tyche thread executing the issue path for each request.

At the target side, dedicated network threads, one per NIC, process incoming packets, compose messages, and generate a notification to Tyche. The interrupt handler of the NIC is only used for waking up the corresponding network thread. Note that the request message is placed in the rm-ID of the `remq`, and the data pages, if any, are placed in the dm-ID position of the `damq`. As a consequence of the notification, Tyche processes the request message and fetches the corresponding data message in case of a write request. Then, Tyche constructs a proper Linux kernel I/O request and issues it to the local block device. The receive path at the target is summarized in Figure 4 for the network layer and in Figure 5 for the block layer.

The target uses work queues, one per core, to send back

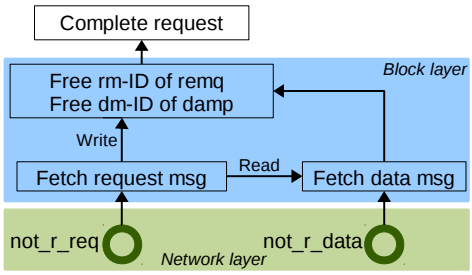


Fig. 7. Overview of the receive path in the initiator.

completions to the initiator. Local I/O completions run in an interrupt context, which is not able to perform network send/receive requests that can block. For this reason, the local I/O completion schedules a work queue task that executes the required Tyche operations. When a completion arrives at the initiator, the network thread constructs the message and generates the notification to Tyche. Finally, the corresponding regular I/O request is completed. Figure 6 depicts the completion path at the target. Figure 7 presents the receive path of the block layer at the initiator which corresponds to the completion of a request. Figures 6 and 7 do not include the network path for sending/receiving, because both paths are already shown in Figures 3 and 4, respectively.

D. Storage-specific network protocol

We include some fields on the Ethernet header to provide end-to-end flow-control, to facilitate communication between the block layer and the network and to allow several connections per NIC.

For request packets, the header includes the corresponding connection, the local position in the transmission ring what also denotes the same position on the receive ring, the position on the request message queue, and positive acknowledgements. For data packets, the header includes the connection, the position on the receive ring, the position on `damq`, the number of pages of the data packet and the number of pages on the data message.

By using the same position on the transmission and receive rings, we reduce packet processing overhead in the receive path. By including for each packet message its position in `remq`, upon its arrival the corresponding message is placed in its final position and we avoid the copy from the network rings to the block data. By including the position in `damq` for a data packet, the data pages are directly placed from the NIC ring in the pages of the I/O requests.

III. Main Challenges

In our design, we deal with the following main challenges: i) synchronization; ii) memory management overhead; iii) NUMA affinity; and iv) many cores accessing a single NIC. Next, we discuss how Tyche addresses these challenges.

A. Synchronization

We minimize the synchronization when accessing shared Tyche structures by reducing the number of spin-locks and mutexes used and by using atomic operations whenever is

possible. Here, we describe the synchronization points needed in our proposal.

In the send path several threads can submit requests concurrently. Therefore, Tyche synchronizes access to all queues, rings, and the NIC itself. In the initiator, the block layer uses two mutexes for exclusive access to `remq` and `damq` (s1 in Figure 3). The network layer has spin-locks to control accesses to transmission rings (s2 in Figure 3), and to update the header. The tail of the transmission rings is updated by an atomic operation. When a data message corresponds to several data packets, several positions are requested in a single operation to acquire just once the lock of the transmission ring for data packets. An additional spin-lock, one per NIC, is used to transmit the packets through the NIC (s3 in Figure 3).

In the receive path, the network threads concurrently poll NICs for reception of events, but, to reduce synchronization, a single thread processes pending events and cleans up the receive rings. An atomic operation controls the access to these two functions.

In the (uncommon) case that overlapping messages are processed concurrently, a per-buffer lock is required to avoid concurrent remapping of a single buffer. This lock is defined as an atomic operation, and it has to be acquired every time a message is processed (note that a message is normally processed just once). To avoid synchronization, each packet has already assigned by the sender its position in the receiving ring. The three pointers used for controlling the receiving ring are updated by atomic operations.

For positive and negative acknowledgments we use a per-connection lock for ensuring that only a single thread sends the corresponding packet. Since several threads can simultaneously set/get notifications, each notification ring has two spin-locks, one for the head and one for the tail.

B. Memory management

As already mentioned, Tyche has pre-allocated request messages (`remq`) and buffer pools (`damp`) to reduce memory management overhead when issuing I/O requests to the target and when receiving/sending data messages.

At the target, the pre-allocated pages are used for sending and receiving data as well as for issuing regular I/O requests to the storage device. Note that, at the initiator, there are no pre-allocated pages because the kernel allocates pages at higher layers when creating I/O requests.

Network protocols over Ethernet involve a copy of data in the receive path from NIC buffers to the actual data location. The reason is that arriving data is placed in the physical pages belonging to the NIC's receive ring, however, these data should be placed eventually in the pages of the corresponding request. The copy of data occurs in the target for write requests and in the initiator for reads. To avoid the overhead of the memory copy in the target, from NIC buffers to Tyche pages, we interchange pages between the NIC receive ring and `damq`.

For reads, at the initiator, this interchange technique cannot be applied. When a read is sent over the network, the layer that initially issued the request expects specific non-sequential *physical* pages (`struct page` objects in the Linux kernel) to be filled with the received data. Therefore, exchanging pages does not work, and a memory copy is required.

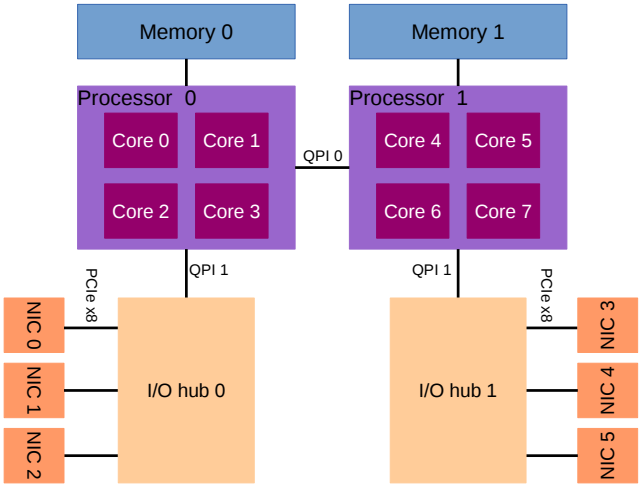


Fig. 8. Internal data paths in our NUMA servers.

C. NUMA affinity

For scalability purposes, modern servers employ NUMA architectures, such as the one depicted in Figure 8 that corresponds to the servers in our work. In such architectures, there is significant difference in performance when accessing local or remote memory [2], [3].

In the I/O path, there are four elements related to NUMA affinity: application buffers, protocol data structures, kernel (I/O and NIC) data buffers, and placement of NICs on server sockets. To achieve maximum performance, the relative placement (or affinity) among application, protocol data structures, kernel buffers, and NICs must consider the system topology and use resources that are close to each other.

We consider two types of affinity that affect the Tyche behavior. The first, that we call threads-core affinity, implies that threads always run in the same NUMA node where the memory they use is allocated and the NIC they use is attached. This affinity includes application threads, protocol threads, work queues, and interrupt handlers. The second, called memory-NIC affinity, implies that memory used by Tyche in the send/receive path is allocated in the same NUMA node where the NIC is located. For instance, at the NIC level, pages of the corresponding Ethernet ring are allocated in the same NUMA node where the NIC is attached, and the connection that uses this NIC allocates their pages, rings, and data structures in the same NUMA node. Therefore, in the architecture of Figure 8, data structures for NICs 0, 1 and 2 and data structures for the connections associated to these NICs are all allocated in "Memory 0".

We do not study the impact of the threads-cores affinity, for several reasons: i) controlling application thread placement can have adverse effects on application performance; ii) in the Linux kernel version used in our implementation, it is not possible to force placement of I/O completions nor the assignment of jobs to work queues; and iii) it is not possible to control placement of page cache. Nevertheless, as future work, we plan to study the affinity for Tyche send/receive threads.

In our design we have considered and analyzed two variants of the memory-NIC affinity. The first, called kmem-NIC affinity, allocates in a specific NUMA node all pages, rings, and data structures of each connection, kernel buffers, and the pages of the NICs. Then, Tyche ensures that each connection will only use NICs located in the same NUMA node. In this case, we do not consider the affinity for user I/O requests: sending data for writes and receiving data for reads, involves no NUMA affinity decisions; we rather select the connection, and therefore the NIC, in a round robin manner. So, at the initiator, the pages of the I/O requests might not be in the same node as the connection-NIC used.

The second one, called full-mem affinity, is kmem-NIC affinity plus affinity at I/O request level. For each user I/O request issued to the initiator, Tyche checks in which node its pages are allocated and then selects a connection-NIC in the same node. The pages for I/O requests are in the same NUMA node as the connection ring and the NIC ring used to transmit or receive the request.

To achieve both types of affinity, Tyche opens a logical connection per NIC and allocates the resources of each connection on the physical memory of the NUMA node where the NIC is attached. Similarly, the NIC rings are allocated on the NUMA node where the NIC is attached. Finally, Tyche selects connections depending on the NUMA node where the buffers of the user I/O request are located.

For writes, kmem-NIC affinity affects the receive path and full-mem affinity both the send and receive path. For reads, kmem-NIC affinity affects the send path at the target and the receive path at the initiator, whereas full-mem affinity also affects the receive path when copying the data from the NIC pages to the pages of the user I/O request. We examine the impact of both types of affinity in Section IV-B.

D. Many cores accessing a single network link

The increasing number of cores in modern servers increases also contention when threads from multiple cores access a single network link. In the send path, the initiator uses the queue mode, where multiple threads place requests in a queue, and Tyche controls the number of threads that can access each link. At the target, work queues send completions back, limiting the number of contexts that interact with each NIC by using one work queue thread per physical core. In the receive path, Tyche uses one thread per NIC to process incoming data. Our measurements show that one core can sustain higher network throughput than a single 10 GigE NIC, and therefore does not limit the maximum throughput (Section IV-C).

IV. Experimental evaluation

We implement Tyche in Linux kernel 2.6.32. We use as baseline NBD (Network Block Device) that is a popular, software-only solution for accessing remote storage. NBD can only use one NIC per remote storage device. We have used iSCSI as well, however, NBD performs better than iSCSI, so we only include NBD in our graphs. For evaluation purposes, and as an intermediate design point, we also implement a version of Tyche that uses TCP/IP. In this version, called TSockets, Tyche creates a socket per connection, and communicates with the remote node through the socket. TSockets uses all available NICs by creating a connection per NIC.

Our experimental platform consists of two systems (initiator and target) connected back-to-back with multiple NICs. Both nodes have two, quad core, Intel(R) Xeon(R) E5520 CPUs running at 2.7 GHz. The operating system is the 64-bit version of CentOS 6.3 testing with Linux kernel version 2.6.32. Each node has six Myricom 10G-PCIE-8A-C cards. Each card is capable of about 10 Gbits/s throughput in each direction for a full-duplex throughput of about 120 Gbits/s. The target node is equipped with 48 GB DDR-III DRAM and the initiator with 12 GB. The target uses 12 GB as RAM and 36 GB as ramdisk. Note that we use ramdisk only for avoiding the overhead of the storage devices, since we are interested in focusing on the network path.

To understand the basic aspects of our approach, we evaluate its main features with two micro-benchmarks zmIO and FIO. zmIO is an in-house micro-benchmark that uses the asynchronous I/O API of the Linux kernel to issue concurrent I/Os at low CPU utilization [4]. FIO is a flexible workload generator [5]. In addition, we analyze the impact of Tyche with the following applications.

IOR [6] emulates various checkpointing patterns that appear in the high performance computing domain. IOR uses MPI and typically exhibits moderate user time, whereas the I/O issued by several concurrent MPI processes results in significant I/O wait time.

BLAST [7] is an application from the domain of comparative genomics. We run multiple instances of BLAST each with a separate set of queries on a separate database. We use random query sequences of 5 kB, which is a common case in proteome/genome homology searches. BLAST is I/O intensive and the execution time is primarily dominated by user time. We use BLAST for Nucleotide-Nucleotide sequence similarity search.

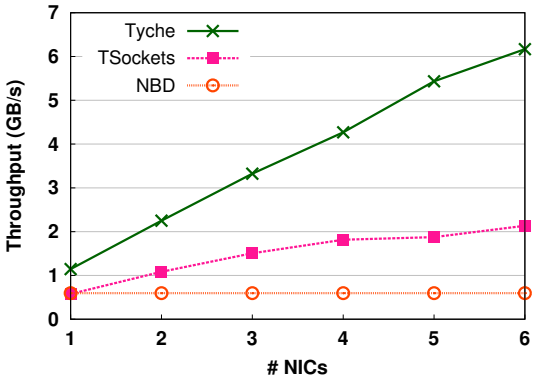
Psearchy [8] is a file indexing application. We run it using multiple processes where each process picks files from a shared queue of file names. We modify the original Psearchy to use block-oriented reads instead of character-oriented reads to improve I/O throughput.

HBase is a NoSQL data store that is part of the Hadoop framework. We use the YCSB benchmark [9]. We first build a database using the YCSB load generator with a workload that makes only insert operations. We then run a workload that does 100% read. We also run a workload that makes 100% insert operations, but without the load phase.

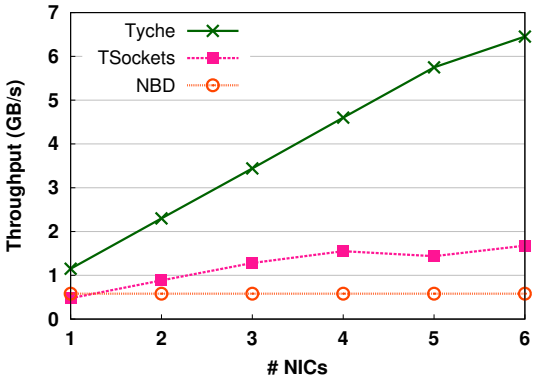
A. Baseline performance

First, we analyze the baseline performance with zmIO. We run zmIO with sequential reads and writes, synchronous operations, direct I/O, 32 threads submitting requests and 2 outstanding requests per thread, a request size of 1 MB, and a run time of 60 seconds. The remote storage device is accessed in a raw manner (there is no file system). The test is run for 1 to 6 NICs, with one connection per NIC.

Figure 9 depicts results for Tyche, TSockets, and NBD. For reads, when 1, 2, and 3 NICs are used, Tyche achieves

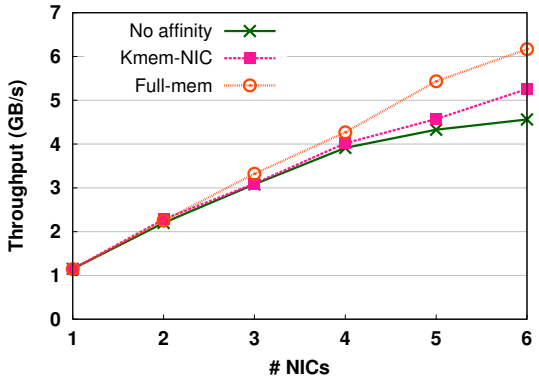


(a) Read requests

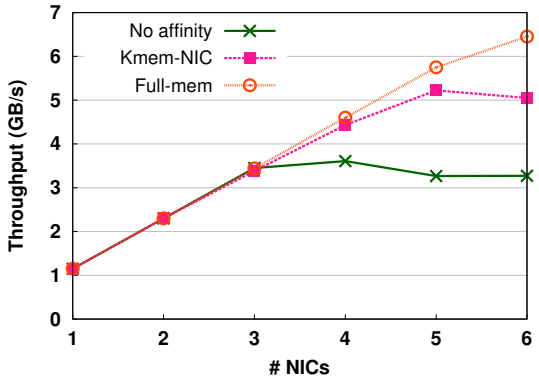


(b) Write requests

Fig. 9. Throughput, in GB/s, achieved by Tyche, TSockets and NBD with zmIO, for sequential reads and writes, and a request size of 1 MB.



(a) Read requests



(b) Write requests

Fig. 10. Throughput, in GB/s, achieved by Tyche depending on the affinity, with zmIO for 32 threads, sequential reads, and writes and 1 MB request size.

the maximum throughput of the NICs. When using 4, 5, and 6 NICs, Tyche provides a bit lower throughput, 4.3 GB/s, 5.4 GB/s, and 6.2 GB/s, respectively. This is due to the overhead of copying pages in the initiator that becomes noticeable at high rates. For writes, Tyche achieves the maximum throughput provided by the NICs except for 6 NICs, that it obtains 6.5 GB/s. With 6 NICs, when running this benchmark, the initiator is almost a 100% CPU utilization. TSockets achieves a throughput of 2.1 GB/s and 1.7 GB/s for reads and writes, respectively. NBD obtains a throughput of 609 MB/s because it is only able to use a single NIC.

We see that Tyche throughput scales with the number of NICs, and our proposal achieves between 82% and 92 % of NIC throughput. NBD is only able to use a single link. TSockets does not scale with the number of NICs, and by using 6 NICs, it is able to saturate at most 2 NICs. Tyche achieves 6.5 GB/s, compared to 2.1 GB/s and 609 MB/s for TSockets and NBD respectively, so Tyche has about 10x the throughput of NBD and more than 3x the throughput of TSockets. We also see that TSockets is more than 3x better than NBD, which shows that TCP/IP is responsible only for part of the overheads when accessing remote storage.

B. Dealing with NUMA

To analyze the impact of the NUMA architecture we consider three configurations of Tyche: no affinity ("No affinity" in Figure 10); only kmem-NIC affinity; and full-mem affinity. We run zmIO with the same configuration as in Section IV-A.

Figure 10 depicts throughput achieved by Tyche depending on placement. Up to 3 NICs there is almost no difference among the three configurations. However, for 4 or more NICs results vary significantly. Maximum throughput is only achieved when all types of affinity are considered, and both the send and receive path use pages that are in the same NUMA node where the NIC is located. With kmem-NIC affinity, throughput is higher than without placement, but, for writes the difference between no affinity and kmem-NIC affinity is higher than for reads. The reason is that for writes this affinity has impact on the receive path, whereas for reads, the impact is more on the send path due to the copy done at the initiator receive path.

When comparing the results of full-mem placement to no affinity at all, Tyche improves the performance up to 35% and 97% for reads and writes, respectively. If the comparison is with the kmem-NIC affinity, the improvement is up to 15% and 54% for reads and writes, respectively.

Results show that Tyche achieves the maximum throughput only when the right placement is done. The kmem-NIC placement is particularly important for writes due to the interchange of pages made between the NIC and the list of pages of Tyche, since the NIC uses these new pages for receiving the data. Therefore, when receiving write data, our protocol checks if the pages to interchange are in the same node, and the interchange is done only in this case. If the pages are allocated

TABLE II. THROUGHPUT, IN MB/S, OBTAINED BY TYCHE WITH ZMIO, WITH 32 THREADS, SEQUENTIAL READS/WRITES FOR A VARYING NUMBER OF CONNECTIONS AND NICS.

	# NICs					
	1			6		
#Connections	1	3	6	1	3	6
Read	1,174	1,175	1,050	4,530	4,699	6,316
Write	1,177	1,153	998	4,493	3,910	6,654

in different NUMA nodes, the protocol will copy the data from the NIC page to the Tyche page.

We also evaluate the affinity impact by using four different configurations that depend on the number of logical connections and NICs. We compare the throughput obtained when 1, 3, and 6 connections are in use, but only over 1 and 6 NICs. The resources (rings, data structures and pages) of the first three connections are allocated in NUMA node 0 ("Memory 0" in Figure 8), whereas, for the other connections, they are allocated in the node 1 ("Memory 1" in Figure 8). When 6 NICs are in use and only 1 or 3 connections, Tyche selects the NIC in a round robin manner.

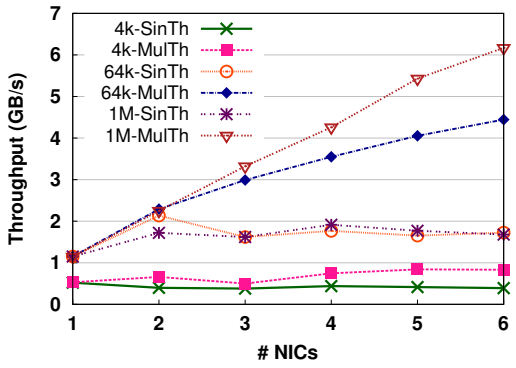
We run zmIO with the same configuration and workloads as in Section IV-A. Table II presents the throughput achieved depending on the number of NICs and connections. Results show the impact of NUMA affinity between the memory allocation of the connections and the NIC. With a single NIC there is almost no difference between using 1 or 3 connections, since all the resources are allocated in the same node. But, with 6 connections, there are affinity problems, since three connections have their resources allocated in a different node than the NIC is attached. Consequently, the throughput drops by 10% for read operations and by 15% for writes. This problem is worse for 6 NICs. The maximum performance is obtained with 6 connections. With 1 or 3 connections, only 3 NICs are attached to the same NUMA node where the connections have allocated their resources. With 1 or 3 connections, the throughput drops to the level of no affinity.

C. Receive path processing requirements

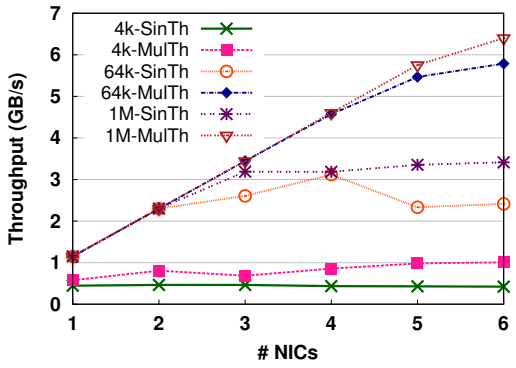
Typically, receive-path processing is heavier than send-path processing in network protocols. To properly understand trade offs with today's CPUs and high speed links we examine processing requirements of the Tyche receive path. Figure 11 depicts Tyche throughput when there is a single network thread for all NICs (curves with "X-SinTh", where X is the request size) and when there is a thread per NIC (curves with "X-MulTh"). We use zmIO with sequential reads and writes, synchronous operations, direct I/O, 32 threads issuing requests, 2 outstanding requests, and a run time of 60s. We use 4 kB, 16 kB, 64 kB, 128 kB, and 1 MB request sizes. Tyche uses affinity optimizations. We only show results for 4 kB, 64 kB and 1 MB, since the other results are similar. We see that a single thread can process requests for two NICs, so about 20 GBits/s. Therefore, using a thread per NIC, Tyche can achieve maximum throughput as well as reduce receive path synchronization.

D. Efficiency of the send path at initiator side

As mentioned, the Tyche initiator can operate in two different modes. In the inline mode, there are many threads



(a) Read requests



(b) Write requests

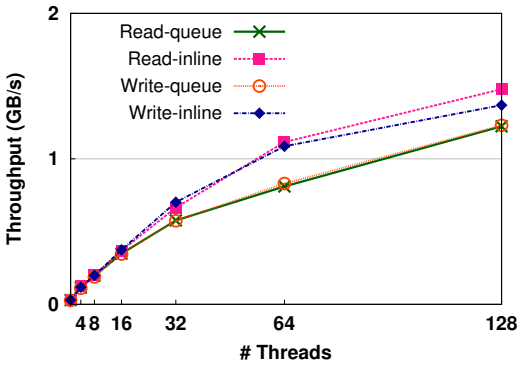
Fig. 11. Throughput, in GB/s, obtained by Tyche when a single network thread processes packets from all NICs (SinTh) or when a thread per NIC is used (MulTh), with zmIO, sequential reads and writes, and 4 kB, 64 kB, and 1 MB request sizes.

submitting requests but the system incurs no context switch overhead. In the queue mode, a context switch is used to avoid having many threads access a single NIC and incur the associated synchronization overhead.

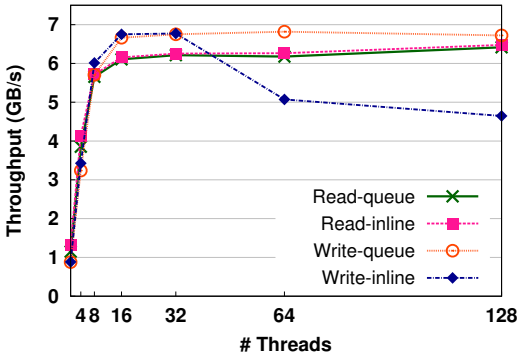
Figure 12 depicts throughput achieved by Tyche as a function of the number of threads and the mode of the send path. We use FIO with sequential reads and writes, direct I/O, a 256 MB file size, request sizes of 4 kB and 512 kB, a run time of 60s. It is run for 1 - 128 tasks, each one with its own file. XFS is used as file system. With this test, Tyche obtains its maximum throughput: 6.48 GB/s for reads with 128 tasks and the inline mode, and 6.81 GB/s for writes with 64 tasks and the queue mode.

For writes, with a request size of 4 kB, there is no difference between both modes up to 16 threads. However, for 32, 64, and 128 threads, the inline mode outperforms the queue one by up to 31%. For a request size of 512 kB, both modes achieve the same throughput up to 32 threads. For 64 and 128 threads, the throughput significantly drops by up to 31% for the inline mode, whereas, the queue mode achieves maximum throughput, due to the increased contention for the NIC lock. The queue mode pays the cost of a context switch but uses 18 Tyche threads for submitting packets (3 per NIC), and lock contention is reduced.

For reads, the inline mode outperforms the queue one up



(a) 4 kB request size



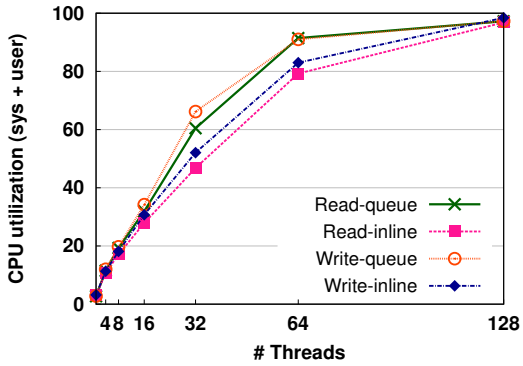
(b) 512 kB request size

Fig. 12. Throughput, in GB/s, of Tyche depending on the send path mode, with FIO, sequential reads and writes and 4 kB and 512 kB request sizes.

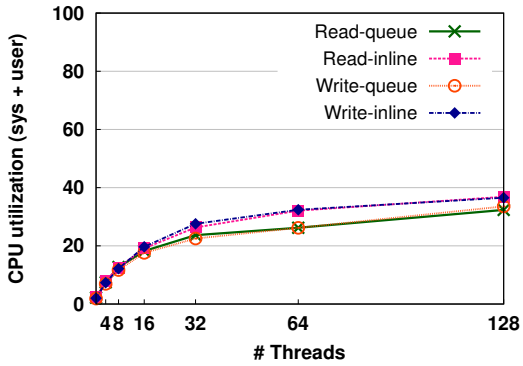
to 27% (for a request size of 4 kB and 64 tasks), because the latter pays the overhead of a context switch when there is just a single thread submitting requests. The exception is for a 512 kB request size and 16 threads or more, in which case both modes achieve similar throughput. For large requests the throughput obtained depends more on the delay of the target than on the overhead at the initiator.

Figure 13 depicts the CPU utilization, calculated as system time utilization plus user time utilization at the initiator and target sides for both modes, depending on the number of application threads and on the request size. At the initiator and with a request size of 4 kB, due to the context switch, the CPU utilization for the queue mode is higher by 29% and 15% for 32 and 64 threads respectively. However, at the target, the queue mode makes less processing, it drops up to 19%, because this mode achieves lower throughput.

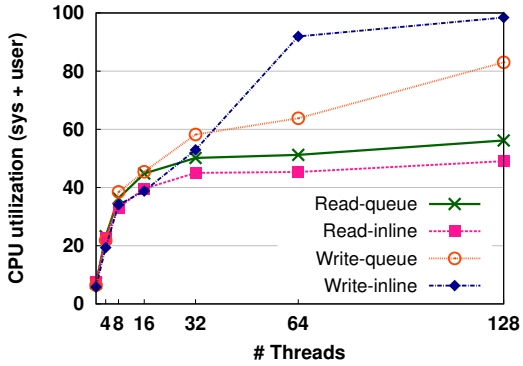
When the request size is 512 kB, for reads, the initiator incurs higher processing, up to 31% in the queue mode, due to the context switch. At the target, both modes use almost the same CPU and the throughput achieved is similar. For writes, up to 32 threads the queue mode incurs more processing at the initiator side, up to 16% more, whereas, at the target, both modes have similar CPU utilization. For 64 and 128 threads, at the initiator, the inline mode makes up to 30% more processing than the queue one, due to the synchronization overhead and lock contention. At the target, since the inline achieves lower throughput, its CPU utilization is also lower, up to 40%.



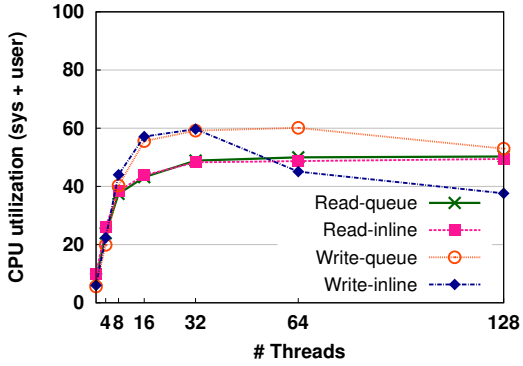
(a) Initiator - 4 kB request size



(b) Target - 4 kB request size



(c) Initiator - 512 kB request size



(d) Target - 512 kB request size

Fig. 13. CPU utilization of Tyche depending on the send path configuration, with FIO for sequential reads and writes and request sizes of 4 kB and 512 kB.

TABLE III. THROUGHPUT, IN MB/S, ACHIEVED BY TYCHE, NBD AND TSOCKETS FOR PSEARCHY, BLAST, IOR, AND HBASE.

	Throughput (MB/s)				
	Tyche		NBD	TSockets	
NICs	1	6	1	1	6
Psearchy	1,154	4,117	499	488	1,724
Blast	775	882	438	391	564
IOR-R 512k	573	1,670	212	226	745
IOR-W 512k	603	1,670	230	243	751
HBase-Read	303	295	154	168	229
HBase-Insert	106	112	99	54	92

Figure 12 shows that for reads both modes scale with the number of application threads. For writes, the queue mode mode scales with the number of threads, whereas the inline mode only scales for small request sizes.

E. Application results

Table III shows the throughput for Psearchy, Blast, IOR and HBase. We choose these tests because they perform a significant amount of I/O and allow us to observe differences at the network protocol level. Tyche always performs better than NBD and TSockets, even with a single NIC. For Psearchy and IOR, the difference between Tyche and NBD is remarkable, Tyche achieves more than 2x and 8x better throughput than NBD with 1 and 6 NICs respectively. For Blast and HBase-Read, the differences are smaller, but Tyche is still up 2x better than NBD. For HBase-Insert, Tyche outperforms NBD by 7% and 10% with 1 and 6 NICs, respectively. When comparing with TSockets, the differences are smaller but still significant, with more than 2x improvement for Psearchy and IOR. For Blast, HBase-Read, and HBase-Insert, Tyche outperforms the vanilla version by 36%, 22%, and 18%, respectively, when 6 NICs are used.

V. RELATED WORK

Regarding network storage protocols iSCSI and NBD are built over TCP/IP and are widely used in Linux. In contrast, Tyche uses its own Ethernet-based transport, which incurs less overhead. HyperSCSI [10] modifies iSCSI to use raw Ethernet instead of TCP/IP. It turns Ethernet into a usable storage infrastructure by adding missing components, such as flow control, segmentation, reassembly, encryption, access control lists and security. Compared to HyperSCSI, Tyche is designed to transparently use multiple NICs, it deals with NUMA and synchronization issues, it uses RDMA-like operations that reduce packet processing, and it employs a copy reduction technique. All the techniques used in Tyche can eventually be incorporated in HyperSCSI as well.

RDMA has been used extensively by protocols, such as iSER (iSCSI Extension for RDMA) [11], SCSI RDMA Protocol (SRP), and RDMA-assisted iSCSI [12] which improve the performance of iSCSI by taking advantage of RDMA-operations. For instance, Burns *et al.* [13] implement an extension iSCSI to support RDMA through iSER. Other protocols are Internet Wide Area RDMA Protocol (iWARP) and RDMA over Converged Ethernet (RoCE) which are the two commonly known RDMA technologies over Ethernet. The former defines how to perform RDMA over TCP. The latter defines how to perform RDMA over a Ethernet link layer. SMB2 Remote Direct Memory Access (RDMA) Transport Protocol of Microsoft is an example of network storage that requires iWARP, Infiniband or RoCE protocols to provide RDMA operations [14]. However, all these protocols focus on providing RDMA capabilities by using hardware support. The focus of Tyche is to use existing Ethernet and to explore issues at the software interface between the host and the NIC, which emerges as an important bottleneck for high-speed communication in networked storage.

Regarding the copy reduction technique, several authors proposed similar techniques [15], [16], [17], [18]. Typically, they use a technique that avoids the copy between the kernel and user space. For instance, Rizzo proposes to remove data-copy costs by granting applications direct access to the packet buffers [17]. Our approach avoids the copy at kernel space by ensuring that Ethernet frames are prepared properly and then interchanging pages between the Ethernet ring and the Tyche queues, specifically targeting our storage protocol that transfers multiples of 4 kB.

A lot of work has been done for NUMA-aware process scheduling and memory management in the context of many-core processors and systems. For instance, Moreaud *et al.* [19] study NUMA effects on high-speed networking in multi-core systems and show that placing a task on a node far from the network interface leads to a performance drop, and especially bandwidth. Their results show that NUMA effects on throughput are asymmetric since only the target destination buffer appears to need placement on a NUMA node close to the interface. In our case, NUMA affects both sides, target and initiator. Ren *et al.* [20] propose a system that integrates an RDMA-capable protocol (iSER), multi-core NUMA tuning, and an optimized back-end storage area network. They apply NUMA affinity by using the `numactl` utility for binding a dedicated target process to each logical NUMA node. They use iSER that relies on hardware support to provide RDMA capabilities. In contrast, Tyche provides RDMA-operations without hardware support. They achieve an improvement of up to 19% in throughput for write operations, whereas our proposal achieves an improvement of up to 2x. Dumitru *et al.* [21] also analyze, among other aspects, the impact of NUMA affinity on NICs capable of throughput at the range of 40 GBits/s, without, however, to propose a solution.

Several authors [22], [23], [24], [25] have studied tightly-coupled NIC architectures and on-load software on Ethernet. For example, the JNIC project replaces one of the four sockets of a server multiprocessor server with a NIC [25]. By closely attaching the NIC to CPU and memory, the NIC can be accessed using coherent memory as opposed to PCI transactions, which reduces latency of accessing the NIC from the processor. This approach requires extensive hardware support, whereas Tyche uses general purpose Ethernet NICs.

The gmblock project is a block-level storage sharing system over Myrinet which transfers data directly between the storage device and the network, bypassing the CPU and main memory bus of the storage server [26]. Although for sending request data is sent directly from the storage device to the network, when receiving, a copy operation is needed between the NIC SRAM and the Lanai RAM. Tyche does not aim to by-pass the target, but rather to optimize the communication path to

the target, allowing for storage functions, such as I/O caching to be performed by the target.

Multipath TCP/IP [27] allows TCP to run over multiple paths, i.e. NICs, transparently to applications by presenting a single TCP interface. Although it can scale throughput with the number of NICs, it still incurs high overheads. In addition, its target is general purpose networking and is not optimized for storage access.

VI. Conclusions

In this paper we present the design, implementation, and evaluation of Tyche, a networked storage protocol that is deployed directly on top of Ethernet and provides RDMA-like operations without requiring hardware support from the network interface. Tyche reduces overheads via a copy-reduction technique, pre-allocation of memory, custom network queues and structures, and storage-specific packet processing. In addition, our approach is able to transparently and simultaneously use multiple NICs and to scale with the number of links and cores via proper packet queue design, NUMA affinity management, and reduced synchronization.

Our results show that Tyche achieves scalable throughput, of up to 6.4 GB/s for reads and 6.7 GB/s for writes on 6x10Gbits/s network links, without requiring any hardware support. This is 89% and 93% respectively of the peek throughput available with 6 NICs. Tyche performs about 10x better than NBD for 6 NICs and by about 2x for 1 NIC. Compared to TSockets, Tyche improves throughput more than 3x. We also find out that, if not taken into account, NUMA affinity can hurt throughput by almost 2x, especially for writes.

Future work needs to consider how Tyche can co-exist with other types of network protocols over Ethernet, and how it can support dynamic policies, e.g. for batching and switching between the inline and queue modes of operation when issuing requests. Overall, we believe that future storage nodes in datacentres will use similar techniques to increase the degree of storage consolidation and to improve data-centre efficiency.

Acknowledgment

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the NanoStreams (FP7-ICT-610509), HiPEAC3 (FP7-ICT-287759), and SCALUS (FP7-PEOPLE-ITN-2008-238808) projects. We are thankful to Dimitris Apostolou for his contribution on earlier versions of this work, and to Manolis Marazakis for his helpful comments.

References

[1] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, Aug. 2009.

[2] M. Dobson, P. Gaughen, M. Hohnbaum, and E. Focht, "Linux Support for NUMA Hardware," in *Ottawa Linux Symposium*, 2003.

[3] C. Lameter, "Local and Remote Memory: Memory in a Linux/NUMA System," in *Ottawa Linux Symposium*, 2006.

[4] "zmIO Benchmark," http://www.ics.forth.gr/carv/downloads.html.

[5] "FIO Benchmark," http://freecode.com/projects/fio.

[6] "IOR Benchmark," http://ior-sio.sourceforge.net.

[7] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, pp. 403–410, 1990.

[8] S. Boyd-Wickizer, A. T. Clements, A. P. Y. Mao, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *Proceedings of the Conference on Operating systems design and implementation*, 2010.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the Symposium on Cloud computing*, 2010.

[10] W. Y. H. Wang, H. N. Yeo, Y. L. Zhu, T. C. Chong, T. Y. Chai, L. Zhou, and J. Bitwas, "Design and development of Ethernet-based storage area network protocol," *Computer Communications*, vol. 29, no. 9, pp. 1271–1283, 5 2006.

[11] M. Ko, J. Hufferd, M. Chadalapaka, U. Elzur, H. Shah, and P. Thaler, "iSCSI Extensions for RDMA Specification (Version 1.0)," http://www.rdmaconsortium.org/home/draft-ko-iwarp-iser-v1.PDF.

[12] J. Liu, D. K. Panda, J. L. D. K. P, and M. Banikazemi, "Evaluating the Impact of RDMA on Storage I/O over InfiniBand," in *Proceedings of the SAN Workshop*, 2003.

[13] E. Burns and R. Russell, "Implementation and Evaluation of iSCSI over RDMA," in *Proceedings of the IEEE International Workshop on Storage Network Architecture and Parallel I/Os*, 2008.

[14] "[MS-SMBD]: SMB2 Remote Direct Memory Access (RDMA) Transport Protocol," http://msdn.microsoft.com/en-us/library/hh536346.aspx.

[15] H. Jin, M. Zhang, and P. Tan, "Lightweight messages: True Zero-Copy Communication for Commodity Gigabit Ethernet," in *Proceedings of the international conference on Emerging Directions in Embedded and Ubiquitous Computing*, 2006.

[16] D.-J. Kang, Y.-H. Kim, G.-I. Cha, S.-I. Jung, M.-J. Kim, and H.-Y. Bae, "Design and Implementation of Zero-Copy Data Path for Efficient File Transmission," in *Proceedings of the international conference on High Performance Computing and Communications*, 2006.

[17] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *Proceedings of the USENIX Annual Technical Conference*, 2012.

[18] P. Shivam, P. Wyckoff, and D. Panda, "EMP: zero-copy OS-bypass NIC-driven gigabit ethernet message passing," in *Proceedings of the Conference on Supercomputing*, 2001.

[19] S. Moreaud and B. Goglin, "Impact of NUMA effects on high-speed networking with multi-opteron machines," in *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, 2007.

[20] Y. Ren, T. Li, D. Yu, S. Jin, and T. Robertazzi, "Design and Performance Evaluation of NUMA-aware RDMA-based End-to-end Data Transfer Systems," in *Proceedings of international conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[21] C. d. L. Cosmin Dumitru, Ralph Koning, "40 Gigabit Ethernet: Prototyping Transparent End-to-End Connectivity," in *Proceedings of the Terena Networking Conference*, 2011.

[22] N. L. Binkert, A. G. Saidi, and S. K. Reinhardt, "Integrated Network Interfaces for High-bandwidth TCP/IP," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 315–324, Oct. 2006.

[23] C. Dalton, G. Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner [Network-independent Card for Protocols]," *Netwrk. Mag. of Global Internetwkg.*, vol. 7, no. 4, pp. 36–43, Jul. 1993.

[24] S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood, "Coherent Network Interfaces for Fine-grain Communication," in *Proceedings of the Annual International Symposium on Computer Architecture*, 1996.

[25] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell, Jr, L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, and N. P. Jouppi, "High-performance Ethernet-based Communications for Future Multi-core Processors," in *Proceedings of the Conference on Supercomputing*, 2007.

[26] E. Koukis, A. Nanos, and N. Koziris, "GMBlock: Optimizing Data Movement in a Block-level Storage Sharing System over Myrinet," *Cluster Computing*, vol. 13, no. 4, pp. 349–372, Dec. 2010.

[27] "Linux kernel MultiPath TCP project," http://www.multipath-tcp.org/.