J Supercomput (2016) 72:4129–4159
DOI 10.1007/s11227-016-1726-7

Mitigation of NUMA and synchronization effects in high-speed network storage over raw Ethernet

Pilar González-Férez[1,2] · Angelos Bilas[2,3]

Published online: 3 May 2016

Abstract Current storage trends dictate placing fast storage devices in all servers and using them as a single distributed storage system. In this converged model where storage and compute resources co-exist in the same server, the role of the network is becoming more important: network overhead is becoming a main limitation to improving storage performance. At the same time, server consolidation dictates building servers that employ non-uniform memory architectures (NUMA) to scale memory performance and bundling multiple network links to increase network throughput. In this work, we use Tyche, an in-house protocol for network storage based on raw Ethernet, to examine and address (a) performance implications of NUMA servers on end-to-end path and (b) synchronization issues with multiple network interfaces (NICs) and multicore servers. We evaluate NUMA and synchronization issues on a real setup with multicore servers and six 10 GBits/s NICs on each server and we find that: (a) NUMA effects have significant negative impact and can reduce throughput by almost 2× on our servers with as few as eight cores (16 hyper-threads). We design protocol extensions that almost entirely eliminate NUMA effects by encapsulating all protocol structures to a "channel" concept and then carefully mapping channels and their resources to NICs and NUMA nodes. (b) The traditional inline approach where each thread accesses the NIC to post-storage requests is preferable to using a queuing approach that trades locks for context switches, especially when the protocol is NUMA-aware. Overall, our results show that dealing with NUMA affinity and syn-

✉ Pilar González-Férez
pilar@ditec.um.es

[1] University of Murcia, Murcia, Spain

[2] FORTH-ICS, Heraklion, Greece

[3] University of Crete, Heraklion, Greece

chronization issues in network storage protocols allows network throughput between the target and initiator to scale by a factor of 2× and beyond 60 GBits/s.

Keywords NUMA · Memory affinity · Synchronization · Network storage · Tyche · I/O throughput

1 Introduction

Current technology trends for efficient use of infrastructures dictate that storage converges with computation by placing storage devices, such as non-volatile memory (NVM)-based cards and drives, in the servers themselves. With converged storage, compute servers are used as a single distributed storage system, in a departure from traditional storage area network (SAN) and network attached storage (NAS) approaches. In this model, where computation and storage are co-located, the role of the network becomes more important for achieving high storage I/O throughput.

Although there has been a lot of research on high-speed interconnects, such as Infiniband, Ethernet-based networks today dominate the data center due to management reasons, cost-efficiency, and the software stack that is already in use. In addition, Ethernet has caught up with other technologies, especially in terms of throughput. In fact, most interconnects today use the same physical layer technology with small differences. However, an area where Ethernet still lacks significantly is protocol overhead, as exhibited in terms of CPU cycles. Technologies, such as Infiniband, are able to support more lightweight protocols than IP-based protocols used over Ethernet. As Ethernet is starting to be used for accessing storage in the data center, protocol overheads are becoming a main concern. For instance, with high-end storage devices, a server can be spending as many cycles for serving remote storage requests as serving the application itself. Therefore, the network protocol used on top of Ethernet plays a significant role in achieving high efficiency for storage access.

In our previous work, we design Tyche [1,2] a network storage protocol over raw Ethernet that achieves high throughput without hardware support. We argue that raw Ethernet is cost-effective and Tyche delivers high I/O throughput and low I/O latency using several techniques: bundling multiple NICs transparently, copy reduction, storage-specific packet processing, remote direct memory access (RDMA)-type communication primitives, memory pre-allocation. However, our previous analysis observes that (1) non-uniform memory access (NUMA) affinity is an important issue that spans the whole I/O path and impacts significantly performance; and (2) for some workloads, a lock contention on the send path can reduce performance.

Modern servers tend to use NUMA architectures (Fig. 1) for efficiency and scalability purposes. These servers use multiple processor sockets or nodes with memory attached to each socket, resulting in non-uniform latencies from processor to different memories. Each device is attached to a specific node via an I/O hub. Processors, memories, and I/O hubs are connected through high-speed interconnects, e.g., QPI (Intel ® QuickPath Interconnect) [3]. Accessing remote memory (different NUMA node) incurs significantly higher latency than accessing local memory [4,5], up to a factor of 2×. The distance among devices, memories, and cores used to process I/O

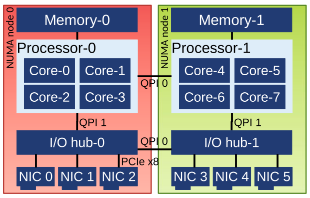

Fig. 1 Internal data paths in NUMA servers

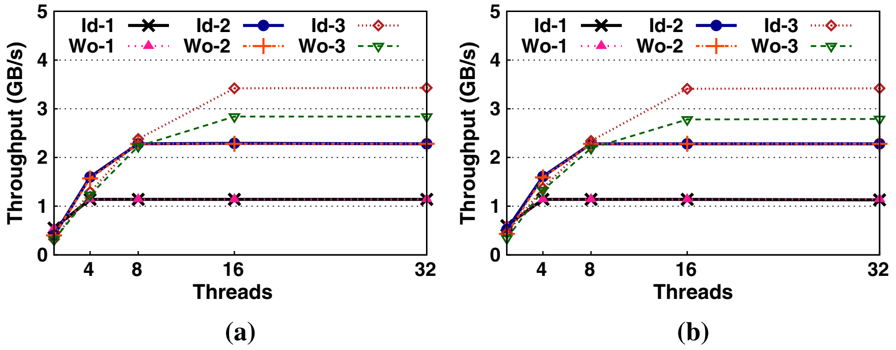

Fig. 2 FIO [11] throughput with direct I/O, random writes and reads, 128 kB and several threads, with (Id) and without (Wo) Tyche affinity over and 1–3 NICs. Id-1, Id-2, and Id-3 correspond to the Ideal NUMA configuration with 1–3 NICs, respectively, whereas Wo-1, Wo-2, and Wo-3 correspond to the Worst NUMA configuration with 1–3 NICs, respectively. Table 2 describes the NUMA configurations of the Ideal and Worst. Note that curves for Id-1, Wo-1, Id-2, and Wo-2 are overlapping. **a** Writes. **b** Reads

requests must be optimized to achieve high I/O efficiency and performance [1,6]. Scaling networked storage throughput on such servers is becoming an important challenge. Therefore, it is essential to explore how network protocols can be designed to cater to affinity among memory, cores, and network interfaces (NIC) for data and protocol data structures.

The second important issue is that making transparent use of multiple NICs and network links to increase I/O throughput from a large number of cores introduces synchronization and lock contention issues. Locks cause serialization, reduce parallel efficiency over multiple NICs, and eventually hurt performance and scalability [7–10], especially as the number of cores and NICs increases.

In this work, we use Tyche to analyze in detail the impact of NUMA affinity and synchronization on networked storage protocols. We analyze two important issues:

1. Can we mitigate performance effects of modern NUMA architectures by properly re-designing the network protocol?
2. Is lock contention an issue in optimized implementations and should we examine alternative designs that trade locks for other overheads?

Our analysis shows that NUMA effects have a very large negative impact on performance. Figure 2 shows this behavior with two different placements, Ideal and Worst (see Table 2), and three NICs attached to the NUMA node 0. With one and two NICs there is no difference in performance between both configurations. However, with three NICs Ideal significantly outperforms Worst by up to 23.7 %. With six NICs Ideal can improve performance by up to 2×. This difference in performance is reflected in the data traffic through the QPI links. With NUMA affinity, data traffic mainly comes through the local links. Without affinity, a large amount of traffic comes through the remote links.

To mitigate these NUMA effects at high network throughput, we carefully design the I/O issue and completion paths, including network send and receive paths. We encapsulate structures and flow control in a "channel" concept that essentially corresponds to the end-to-end I/O path. We map channels to NICs and to NUMA nodes. For each I/O request, we dynamically select the appropriate channel based on the location of its buffers. Thanks to bundling several channels/NICs, Tyche not only improves throughput but also mitigates NUMA effects. Our approach aligns buffers and NICs and almost entirely eliminates NUMA effects.

Our lock analysis shows that when there is a high concurrency of large write requests, throughput can drop by up to 2× due to a lock contention, especially if the application does not consider memory placement. Indeed, NUMA affinity can significantly impact on the lock overhead as well. We avoid this lock contention with a queue approach: application threads insert I/O requests in a queue, and a few threads dequeue and issue them. This approach pays a context switch, but significantly reduces lock contention for large writes.

Overall, our results show that network storage protocols for modern servers with multiple resources need to be designed for NUMA affinity and synchronization to achieve high network throughput. Otherwise, when they are not taken into account, the performance is significantly downgraded.

The rest of this paper is organized as follows. Section 2 presents a background on Tyche. Section 3 describes our experimental environment. Section 4 analyzes the NUMA impact on Tyche, Sect. 5 describes our NUMA-aware design, and Sect. 6 evaluates this design. Sections 7 and 8 present and evaluate the synchronization points and lock contention problem. Section 9 analyzes the impact of NUMA and synchronization on solid-state drives (SSD) and NVMs. Finally, we present related work in Sect. 10 and draw our conclusions in Sect. 11.

2 Background

Tyche [1,2] is an end-to-end network storage protocol on top of raw Ethernet that achieves high I/O throughput and low latency without hardware support. The overall design of Tyche is depicted in Fig. 3. Tyche presents the remote storage device locally by creating at the client (initiator) a virtual device that can be used as a regular device. Tyche is independent of the storage device and supports any file system. It provides reliable delivery, Ethernet-framing, and transparent bundling of multiple NICs.

To reduce message processing overhead, Tyche uses a copy reduction technique based on virtual memory page remapping, reduces context switches, and uses RDMA-type operations. The server (target) avoids all copies for writes by interchanging pages between the NIC receive ring and Tyche. The initiator requires a single copy for reads, due to OS-kernel semantics for buffer allocation. Tyche does not use RDMA over Ethernet, instead our protocol uses a similar, memory-oriented abstraction that

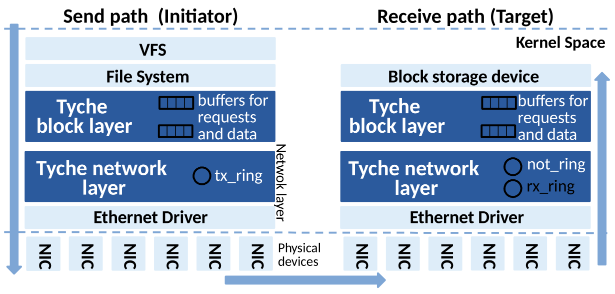

Fig. 3 Overview of the send and receive path from the initiator to the target

allows Tyche to reduce messaging overhead by avoiding packing and unpacking steps that are required over streaming-type abstractions, such as sockets. Tyche reduces overheads for small I/O requests by avoiding context switches for low degrees of I/O concurrency. Additionally, there are several optimizations, such as avoiding dynamic memory allocations that are typical in network protocol implementations.

Tyche uses small request messages for requests and completions, and data messages for data pages. A request message corresponds to a request packet that is sent using a small Ethernet frame. A data message corresponds to several data packets that are sent using Jumbo Ethernet frames of 4 or 8 kB, i.e., each data packet sends one or two data pages of the regular I/O request.

Tyche uses two pre-allocated buffers to send/receive messages. The initiator handles both buffers by specifying in the message header its position on them, and, on its reception, a message is directly placed on its buffer's position. At the target, the buffer for data messages contains lists of pre-allocated pages for sending/receiving data and for issuing requests to the local device. The initiator has no pre-allocated pages and uses the pages of the I/O requests.

The initiator send path can operate in two modes. In the “inline” mode (Fig. 3), the application context issues requests with no context switch. In the “queue” mode, the application inserts requests in a queue at the block level and a thread dequeues them and issues them.

The target uses a work queue for sending completions back, because local I/O completions run in an interrupt context that cannot block. At the receive path, a network thread per NIC processes packets and messages. At the block layer, several threads process I/O requests.

Figure 4 depicts the end-to-end I/O path of a write request when Tyche works in inline mode. Reads follow the same path, but the target sends back the data messages. In the figure, the red lines and numbers are the steps of the path and their execution order. The solid lines mean that there is a copy of data, and the dashed ones that there is no copy of data.

3 Experimental environment

In this work, we conduct a detailed analysis of NUMA effects and lock synchronization overheads up to 60 Gbits/s over bundled 10 GigE Ethernet links.

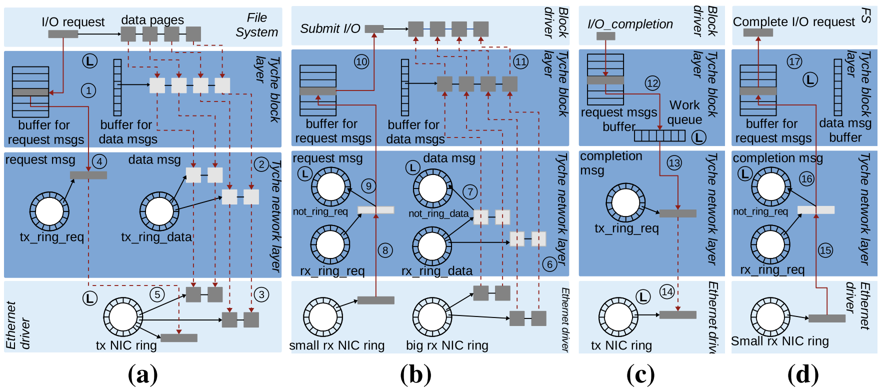

Fig. 4 End-to-end I/O path of a 16 kB write request when Tyche works in the inline mode. **a** Initiator send path. **b** Target receive path. **c** Target send path. **d** Initiator receive path

Our experimental platform consists of two systems (initiator and target) connected back to back. 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. The target has 48 GB DDR-III DRAM and the initiator has 12 GB. The target uses 12 GB as main memory and 36 GB as Ram disk. Each node has six Myricom 10G-PCIE-8A-C cards that are capable of about 10 Gbits/s throughput in each direction.

Note that although our experimental platform uses a single initiator (client), our results show that this single initiator can generate enough traffic to saturate the target's links. In addition, in this work we are interested in the impact of NUMA affinity and synchronization on network storage protocols. Scale out does not change our results and conclusions: in a network setup with multiple clients and servers, the same problems appear since NUMA effects are due to the individual architecture of each node and the synchronization problems arise when there is a large number of cores using a small number of NICs.

We use two micro-benchmarks: flexible I/O (FIO) [11] is a workload generator with many parameters, including the number of threads, synchronous and asynchronous operations, request size, access pattern, etc. zmIO [12] is an in-house benchmark that uses the asynchronous I/O API of the Linux kernel to issue concurrent I/Os with minimal CPU utilization. We use direct I/O, random reads and writes, several request sizes from 4 to 512 kB, and 1–128 application threads. Along the experiments, when FIO or zmIO apply NUMA affinity, we explicitly and manually place threads and resources.

We start from the Tyche [1] version implemented in Linux kernel 2.6.32. The NUMA analysis uses the queue mode of Tyche. The synchronization analysis compares the inline and queue modes.

We analyze the QPI traffic with the open-source Intel ® Performance Counter Monitor (PCM) [13] that provides estimations of traffic transferred through QPI links. Table 1 describes the estimations analyzed.

Table 1 Estimations of data traffic through the QPI links given by PCM

Name	Node	Traffic	Direction
Q1-N0	0	Local	I/O-hub-0 ⇔ Processor-0
Q1-N1	1	Local	I/O-hub-1 ⇔ Processor-1
Q0-N0	1	Remote	Processor-0 ⇒ Processor-1
Q0-N1	0	Remote	Processor-1 ⇒ Processor-0

Note that the PCM tool does not provide the QPI traffic between I/O hubs

Table 2 Test configuration for the baseline NUMA study

Test	NUMA nodes		
	NICs	Tyche	Application
Ideal	0	0	0
Worst	0	1	1

We specify the NUMA node in which the NICs are attached. For Tyche and the application, we specify the node in which memory and threads are allocated or pinned

We analyze the synchronization overhead with the utility `lock_stat` [14] of the Linux kernel that provides statistics on locks.

4 Degradation of I/O throughput due to NUMA

We first analyze the NUMA effects on Tyche using the baseline version that has no support for NUMA. To analyze QPI traffic due only to Tyche and exclude traffic due to the storage device, the target completes requests without performing the actual I/O during this test. We attach three NICs to NUMA node 0. Table 2 describes the two extreme configurations, Ideal and Worst, analyzed.

Figure 2 provides, for 128 kB requests, throughput, in GB/s, achieved by Tyche as a function of the number of application threads, and Figs. 5 and 6 depict for the target and initiator, respectively, the QPI traffic, in GB/s, as a function of the number of application threads and the NUMA affinity applied.

Figure 2 shows that, with 1 and 2 NICs Ideal and Worst obtain the same throughput. However, with three NICs Ideal significantly outperforms Worst by up to 23.7 %. The reason is the traffic in the QPI interconnects.

Figure 5 shows that, with Ideal, the target has only local QPI traffic through Q1-N0. With Worst the behavior is different, and the target has remote QPI traffic through Q1-N1. The data go through QPI-1 that connects I/O hub-1 with Processor-1 and through the QPI link that connects the I/O hubs (this traffic is not reported by the tool).

With three NICs, the amount of data traffic generated saturates this path that becomes a bottleneck. Indeed, QPI and Tyche throughput are improved by up to 24.5 and 22.6 %, respectively, when NUMA affinity is taken into account.

Figure 6 shows that, at the initiator, the QPI traffic follows the same paths as at the target. In addition, writes have traffic through QPI-0 due to cache line invalidations in the remote node. Reads do not induce this kind of traffic.

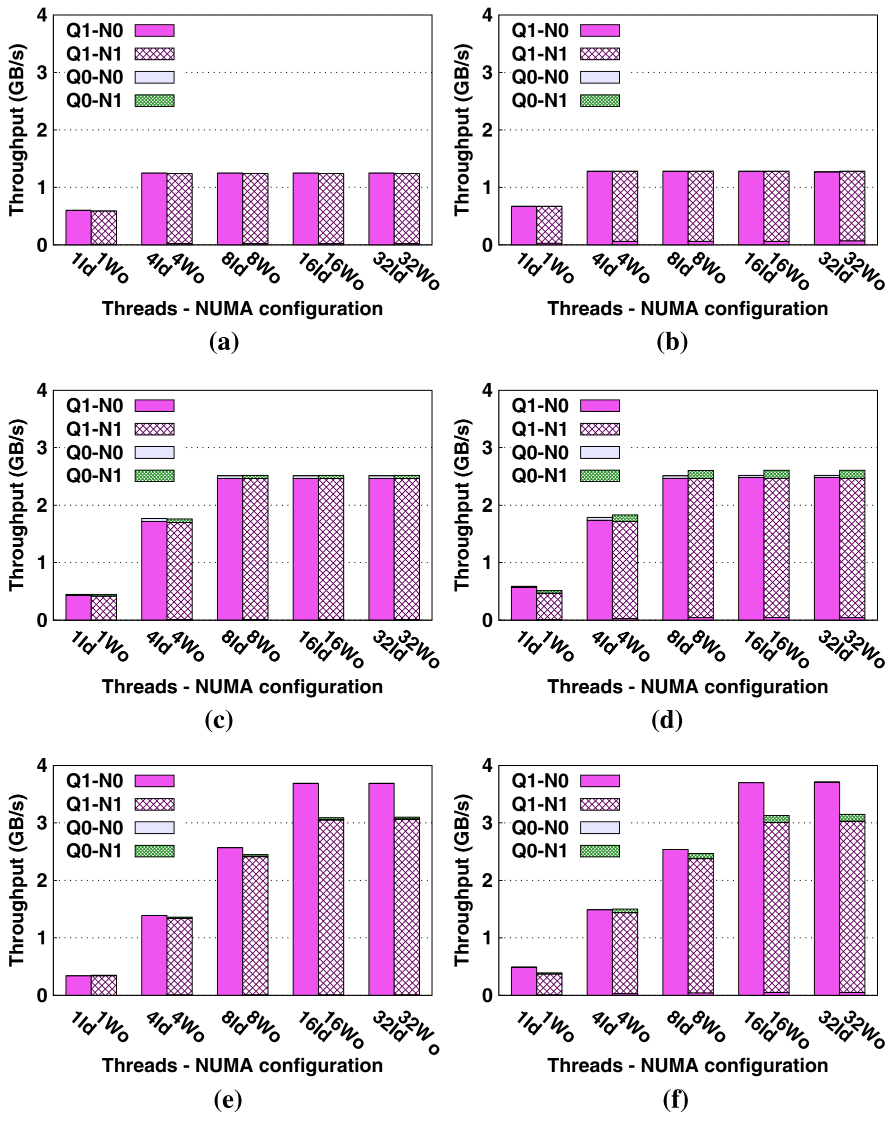

Fig. 5 At the target QPI traffic, in GB/s, for Ideal (Id) and Worst (Wo) NUMA configurations, and 1, 4, 8, 16, and 32 application threads, with FIO, 128 kB requests, random writes and reads, and 1–3 NICs. Table 1 describes the QPI traffic measured. **a** 1 NIC—writes. **b** 1 NIC—reads. **c** 2 NICs—writes. **d** 2 NICs—reads. **e** Three NICs—writes. **f** Three NICs—reads

For 4, 8, 16, 32, and 64 kB request sizes, Fig. 7 depicts throughput, in GB/s, achieved by Tyche, and QPI traffic, in GB/s, at initiator and target, as a function of the NUMA configuration and request size, with three NICs and 32 FIO threads. We only provide results with 32 threads, because results are quite similar to those obtained with 128 kB requests.

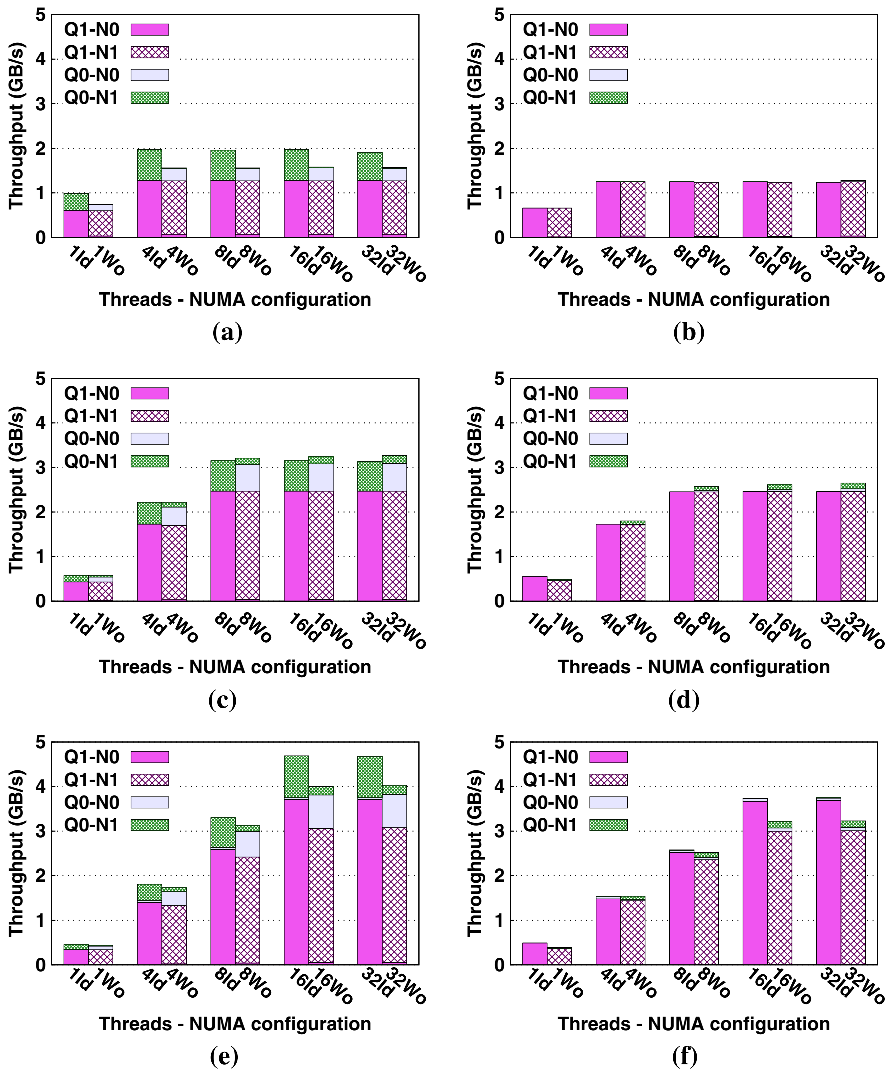

Fig. 6 At the initiator QPI traffic, in GB/s, for Ideal (Id) and Worst (Wo) NUMA configurations, and 1, 4, 8, 16, and 32 application threads, with FIO, 128 kB requests, random writes and reads, and 1–3 NICs. Table 1 describes the QPI traffic measured. Table 2 describes the NUMA configurations of Ideal and Worst. **a** 1 NIC—writes. **b** 1 NIC—reads. **c** 2 NICs—writes. **d** 2 NICs—reads. **e** Three NICs—writes. **f** Three NICs—reads

Figure 7 shows that Ideal significantly outperforms Worst in throughput by up to 22.1 % as well, although the improvement depends on the request size. Similar to the 128 kB request, Ideal mainly has data traffic through Q1-N0 and Worst through Q1-N1.

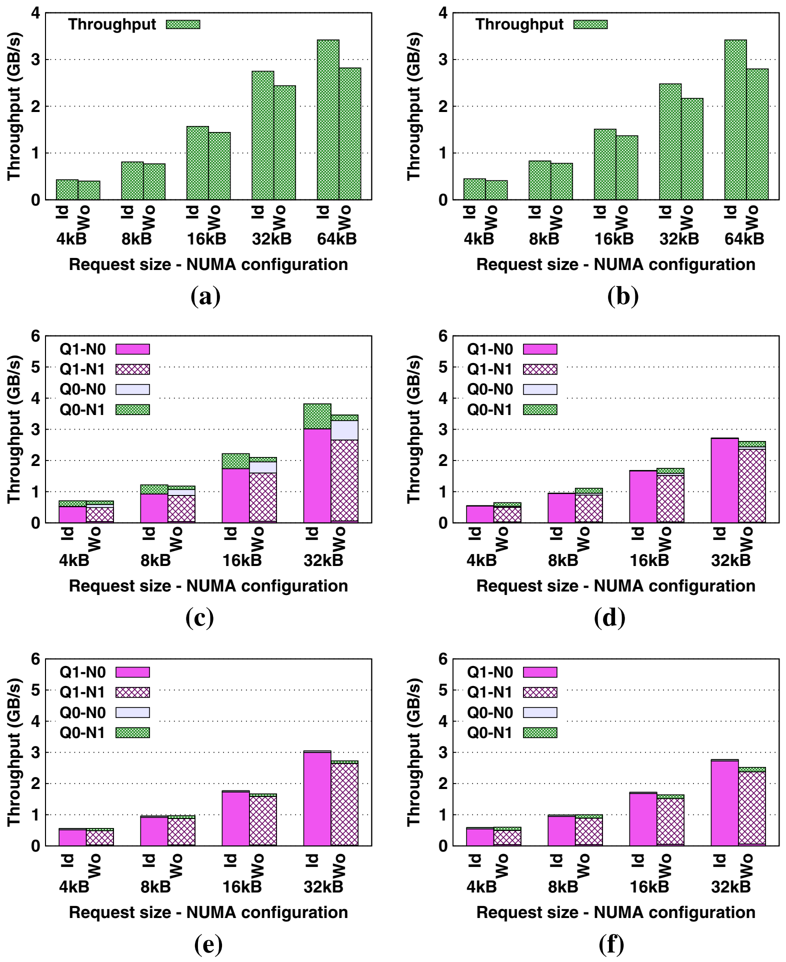

Fig. 7 Throughput (**a**, **b**), in GB/s, and QPI traffic (**c**–**f**), in GB/s, for Ideal (Id) and Worst (Wo) NUMA configurations and three NICs, with FIO, 32 threads, direct I/O, random writes and reads, and several request sizes. Table 1 describes the QPI traffic measured. Table 2 describes the NUMA configurations of Ideal and Worst. **a** Throughput—writes. **b** Throughput—reads. **c** Initiator QPI—writes. **d** Initiator QPI—reads. **e** Target QPI—writes. **f** Target QPI—reads

5 Protocol design for NUMA affinity

Results provided in Sect. 4 confirm that we need to consider affinity among different resources to achieve high throughput in NUMA architectures; otherwise, performance can be significantly downgraded [4,5]. In the network I/O path, there are four resources related to NUMA affinity: application buffers, protocol data structures, kernel (I/O and NIC) data buffers, and NIC location in server sockets. The placement of threads plays a role as well, and it affects application threads, protocol threads, work queues, and interrupt handlers. In addition, avoiding cross-transfer over NUMA nodes and choosing a NIC on the proper node are also important to do the right placement and provide an efficient I/O path. Tyche orchestrates affinity of memory and threads by considering the system topology and the location of NICs.

Tyche uses the concept of communication "channel" to establish a connection between initiator and target. A channel allows a host to send/receive data to/from a remote host. We directly associate a channel with the NIC that is used for sending/receiving data. Although a channel is mapped to a single NIC, several channels can be mapped to the same NIC. Tyche is able to simultaneously manage several channels and it creates at least a channel per NIC. Transparently managing several channels/NICs allows Tyche not only to increase the throughput, but also to provide a better NUMA affinity.

Tyche associates resources exclusively with a single channel. Each channel allocates memory for all purposes and pins its threads to the same NUMA node where its NIC is attached. For instance, in the architecture of Fig. 1, a channel mapped to NIC-0 uses memory in Memory-0 and runs its threads in cores within Processor-0.

The NIC driver uses per NIC data structures: a transmission ring and two receive rings. We force the allocation of these rings in the same node where the NIC is attached as well, making them part of the NIC channel.

We implement a NUMA-aware work queue in Tyche, because in the Linux kernel we use in our experiments, work queues do not allow placement of tasks. Our work queue launches a thread per core that is pinned to its corresponding core. The target submits completion messages to the work queue using its NUMA information. Conceptually, there is a work queue per channel.

There are a few remaining parts that are not affinity aware. In the Linux kernel, (a) it is not possible to control the placement of buffer cache pages, (b) controlling application thread placement may have adverse effects on application performance, and (c) it is not possible to control the placement of the device I/O completions context (in the target). Next, we discuss how we deal with (a) and (b), whereas our experimental results show that the impact of (c) is not significant.

To deal with the affinity of the I/O request buffers that are allocated before the request is passed to Tyche, we use an "assignment" approach. We allow requests to arrive with pre-allocated buffers anywhere in the system. Then, we dynamically detect where buffers are allocated, we identify a NIC that is located in the same NUMA node as the request buffers, and we assign the request to a channel that uses this NIC. For this purpose, Tyche implements a scheduler to select a channel through which the next I/O request will be issued. If there are several channels on this node, it uses a fairness metric, by default equal kBs to each channel, to select one of them. Therefore, a request will be transmitted through a local NIC, i.e., a NIC located in the same NUMA node than the request's buffers. Figure 8 depicts a Tyche system composed of six NICs, three per NUMA node, three channels per NIC, and the scheduling of two I/O requests. Our evaluation contrasts this affinity-based scheduling to a simple round robin approach that does not consider buffer location and merely distributes I/O requests to channels in a round robin manner.

Our affinity-based scheduling policy does not deal well with load imbalance among NUMA nodes. For instance, if a single application is using Tyche, and this application is placed in NUMA node 0, it would only use channels mapped to NUMA node 0,

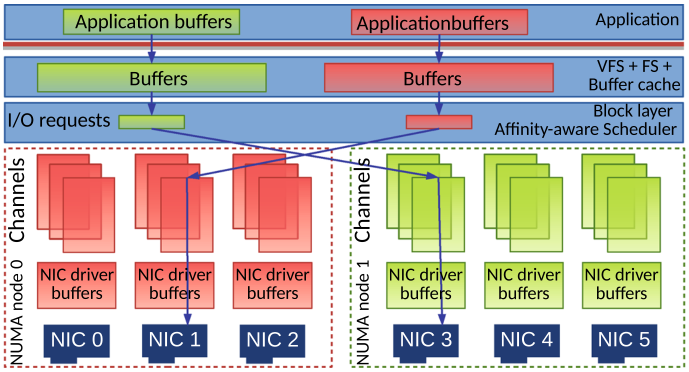

Fig. 8 Affinity-aware scheduler selecting a channel for two I/O requests in a Tyche system with six NICs, three per NUMA node and three channels per NIC

whereas the channels mapped to NUMA node 1 will be idle. In this case, a round robin policy might achieve higher throughput using more NICs. We should note that in consolidated servers, this and similar situations are not common, because Linux allocates memory appropriately and also because typically servers run multiple applications.

To address such load imbalance situations, we implement a channel scheduler that is able to dynamically select the scheduling policy depending on the throughput achieved by each NUMA node and the current load in terms of requests. Our dynamic channel scheduler uses by default the affinity-based policy. It switches to a round robin policy when it detects that better balancing of requests across NUMA nodes may lead to higher throughput. It then switches back to affinity scheduling when it finds that balancing requests across NUMA nodes creates unnecessary cross-NUMA traffic.

For each NUMA node, we calculate the NUMA link utilization by taking into account only the local requests transmitted through the node. As local requests, we count those requests with buffers allocated in this NUMA node.

Whenever the difference in NUMA link utilization between both nodes is larger than 60 %, we choose round robin as policy. In the other case, the affinity-based scheduling is used.

6 Improvement due to protocol NUMA extensions

We now analyze the impact of NUMA depending on the placement applied by Tyche and the application. Table 3 describes the three configurations evaluated: Ideal, T-Numa, and No-Opt. With Ideal, we manually place half of the application threads on each NUMA node. Now, we only analyze QPI traffic at the initiator, because the target performs the actual I/O and has QPI traffic of the network and storage devices. We use six NICs, three on each NUMA node.

Maximum performance is only achieved when both Tyche and the application apply NUMA affinity. For this reason, we examine two cases: (a) regular applications that are not tuned for NUMA, and (b) hand-tuned applications.

Table 3 For the NUMA study, NUMA affinity applied and channel scheduler used on the configurations tested

Test	NUMA affinity		Channel scheduler
	Tyche	Application	
Ideal	Yes	Yes	Affinity aware
T-Numa	Yes	No	RR
No-Opt	No	No	RR

RR round robin

6.1 Impact in performance of protocol extensions for NUMA

We perform this analysis with zmIO that allocates 99 % of writes and around 75 % of reads in NUMA node 0. Thus, almost all writes issued to channels in node 1 have their buffers in node 0, whereas for reads, this rate is only 50 %.

Figure 9 depicts throughput, in GB/s, achieved by Tyche as a function of the number of application threads. Figure 10 depicts the percentage of the total QPI traffic as a function of the application threads and configuration.

For writes, Fig. 9 shows that only by applying the right placement (Ideal) Tyche achieves its maximum throughput, about 6.8 GB/s. Figure 10 shows that with Ideal, almost all the data traffic comes through the QPI-1 links, having a similar amount of traffic in both nodes.

No-Opt obtains only up to 4.7 GB/s and Ideal outperforms No-Opt by up to 85 %. Figure 10 shows that for writes, No-Opt has data traffic through Q1-N0, but not through Q1-N1, since almost all the I/O requests are allocated to node 0. There is also a significant amount of traffic through the QPI-0 links.

T-Numa achieves only up to 5.0 GB/s and Ideal improves its throughput up to 37.6 %. Figure 10 shows that T-Numa behaves like No-Opt and the data traffic through the QPI-1 link is mainly on node 0.

Therefore, for writes, due to the QPI traffic, No-Opt and T-Numa are not able to provide better performance. T-Numa outperforms No-Opt because at the target T-Numa is applying NUMA affinity, whereas No-Opt is not.

For reads, Ideal and T-Numa achieve up to 6.8 and 6.9 GB/s, respectively, whereas No-Opt achieves only up to 4.8 GB/s. Applying NUMA affinity, Ideal improves throughput by up to 84.8 % compared with No-Opt.

This difference in performance between Ideal and No-Opt is due to the QPI traffic. With reads, Ideal has all the data traffic through the QPI-1 links, with both nodes having the same amount of traffic. However, No-Opt has up to 33 % of the total traffic coming through the QPI-0 links.

Ideal and T-Numa exhibit a quite similar behavior. But at the initiator, T-Numa has a similar QPI traffic than No-Opt. Two reasons explain the QPI traffic with T-Numa. First, application buffers are allocated (only) at the initiator and zmIO does not apply NUMA. Second, Tyche applies NUMA placement; however, at the initiator the amount of memory used by Tyche is small, since Tyche uses data pages of I/O requests for data messages.

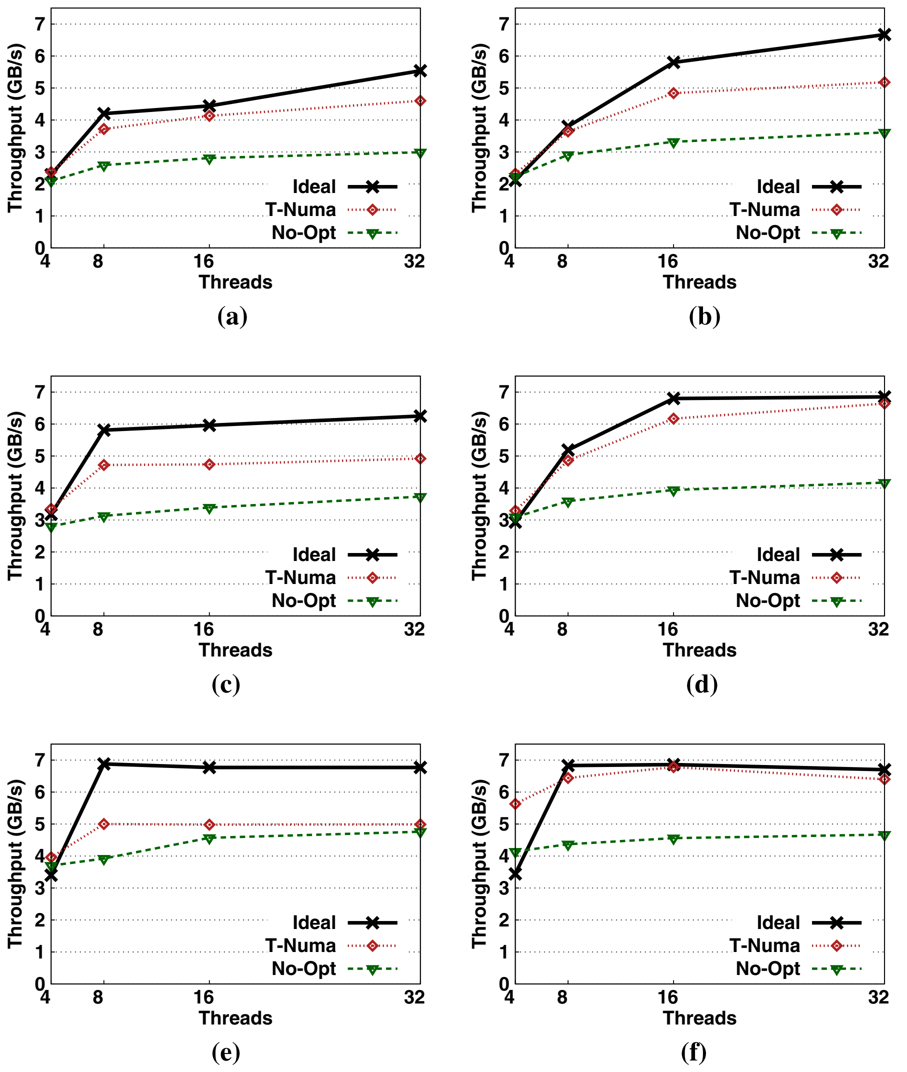

Fig. 9 Throughput, in GB/s, achieved by Tyche depending on the NUMA affinity applied by Tyche and the application. We use zmIO for 64, 128, and 512 kB requests, and random writes and reads. Table 3 describes the three NUMA configurations tested. **a** 64 kB writes. **b** 64 kB reads. **c** 128 kB writes. **d** 128 kB reads. **e** 512 kB writes. **f** 512 kB reads

At the target, Ideal and T-Numa have the same QPI traffic, because both apply affinity. With No-Opt, the target does not apply affinity and a significant amount of traffic goes through the QPI-0 links.

Note that with four threads, threads and resources are allocated in NUMA node 0. For this reason, Ideal has only traffic through the QPI-1 link of node 0.

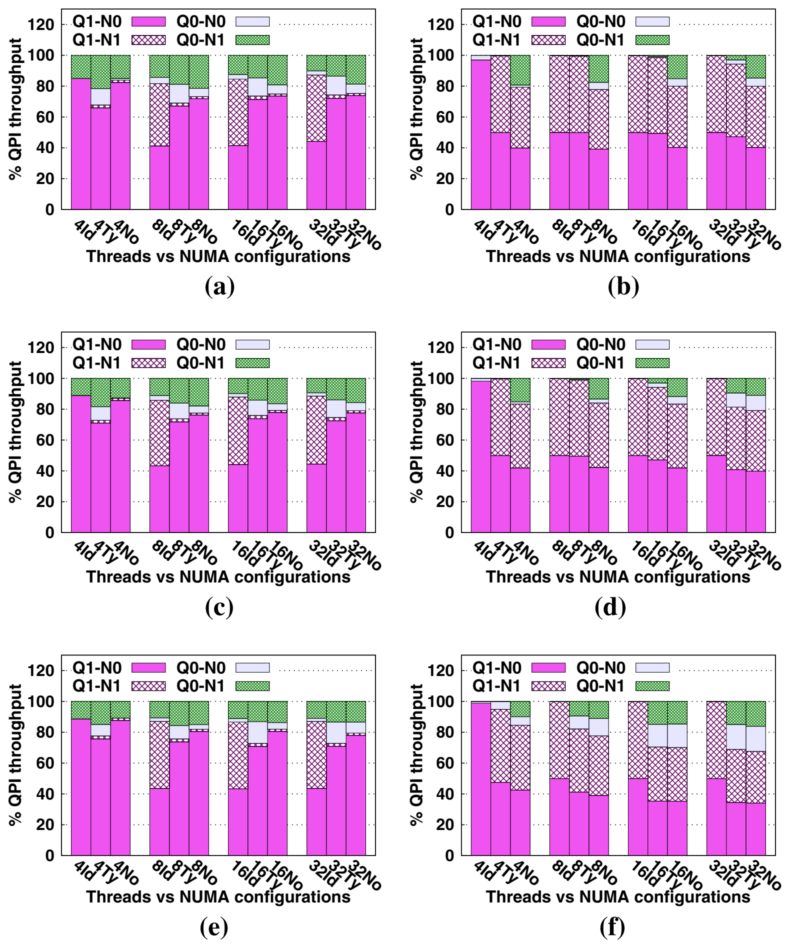

Fig. 10 Percentage of QPI for Tyche depending on the NUMA affinity applied by Tyche and the application. We use zmIO for 64, 128, and 512 kB request size, and random writes and reads. Id, Ty and No stand for Ideal, T-Numa and No-Opt configurations, respectively (see Table 3), and the number indicates the number of threads used. Table 1 describes the QPI data traffic measured. **a** 64 kB writes. **b** 64 kB reads. **c** 128 kB writes. **d** 128 kB reads. **e** 512 kB writes. **f** 512 kB reads

6.2 Impact on the performance of NUMA protocol extensions on hand-tuned applications

To answer this question, we use FIO that allocates I/O buffers in a balanced manner across NUMA nodes. Around 50 % of the requests are issued through a channel allocated in the same node where the request's buffers are allocated.

Figure 11 depicts the throughput, in GB/s, achieved by Tyche as a function of the number of application threads. Figure 12 depicts the percentage of the total QPI traffic as a function of the application threads and configuration.

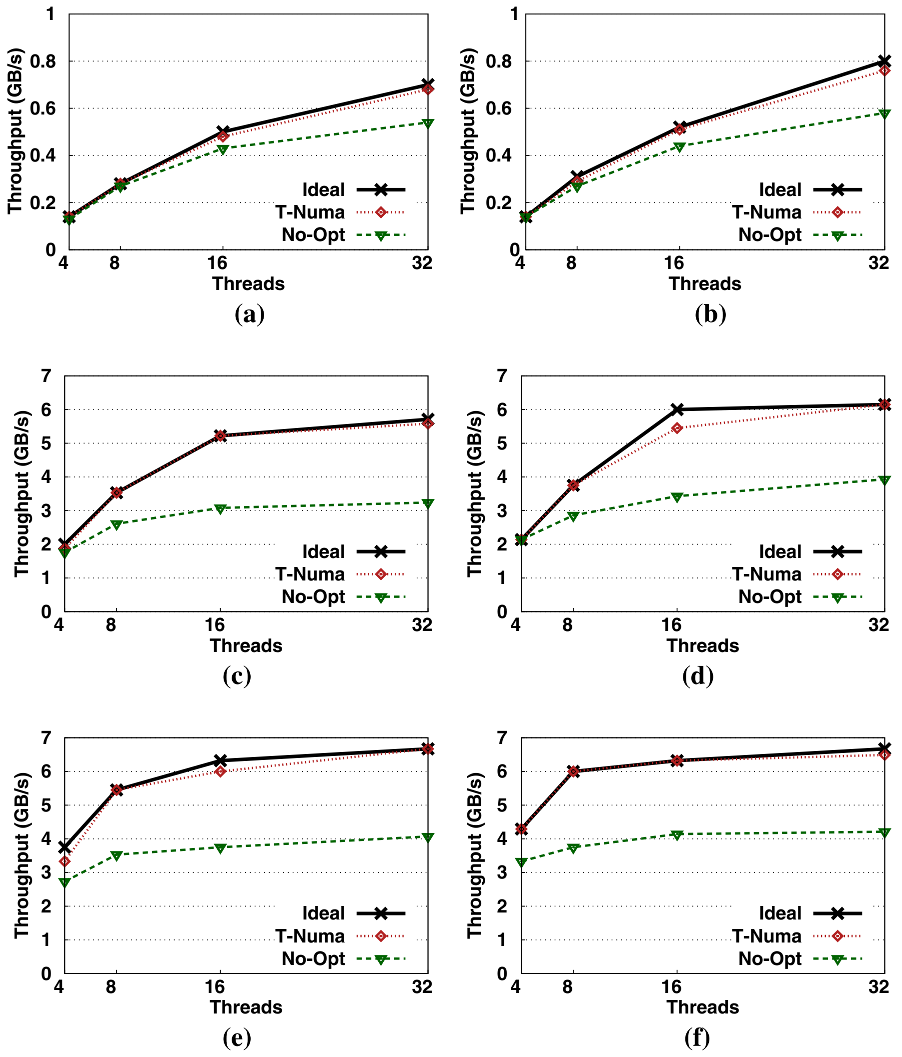

Fig. 11 Throughput, in GB/s, achieved by Tyche depending on the NUMA affinity applied by Tyche and the application. We use FIO for 4, 128, and 512 kB requests, and random writes and reads. Table 3 describes the three NUMA configurations tested. Note that sometimes curves for Ideal and T-Numa are overlapping. **a** 4 kB writes. **b** 4 kB reads. **c** 128 kB writes. **d** 128 kB reads. **e** 512 kB writes. **f** 512 kB reads

Figure 11 shows that with Ideal Tyche obtains maximum throughput, up to 6.7 GB/s. With No-Opt, Tyche obtains only up to 4.2 GB/s. With the right placement, Ideal improves throughput up to 76 % when compared with No-Opt.

Figure 12 shows that this difference in performance is due to the QPI traffic, as explained for reads with zmIO. Ideal has almost only local traffic through the QPI-1 links, and both nodes have a similar amount of traffic. However, No-Opt has a significant amount of remote traffic through the QPI-0 links, up to 48 % of the total traffic.

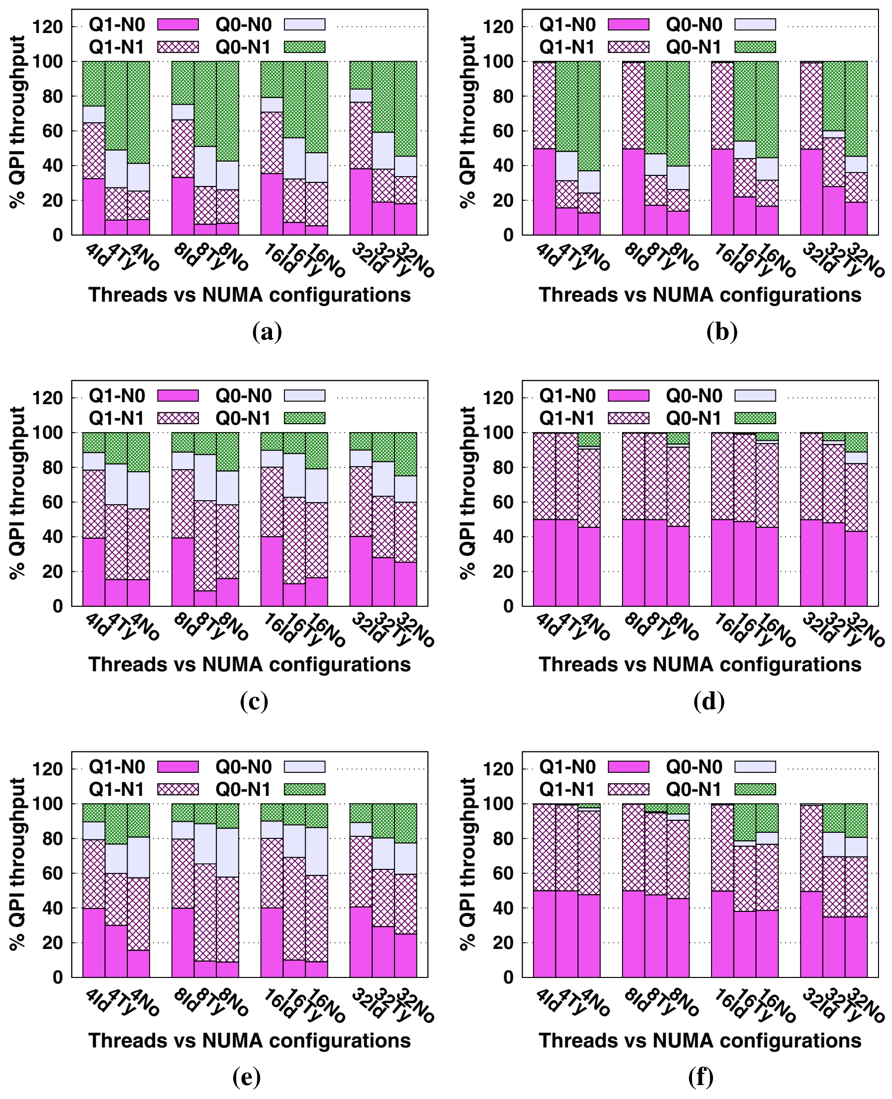

Fig. 12 Percentage of QPI for Tyche depending on the NUMA affinity applied by Tyche and the application. We use FIO for 4, 128, and 512 kB request size, and random writes and reads. Id, Ty, and No stand for Ideal, T-Numa, and No-Opt configurations, respectively (see Table 3), and the number indicates the number of threads used. Table 1 describes the QPI data traffic measured. **a** 4 kB writes. **b** 4 kB reads. **c** 128 kB writes. **d** 128 kB reads. **e** 512 kB writes. **f** 512 kB reads

T-Numa behaves as Ideal due to the QPI traffic at the target, as explained for zmIO, because the target applies affinity with both.

For small requests of 4 kB, memory placement has also a significant impact and Ideal improves throughput by up to 66.2 % compared to No-Opt. Similar to large requests, T-Numa is within 6 % of Ideal.

Although Ideal exhibits perfect placement, for writes there is traffic at QPI-0 due to cache line invalidations to the remote NUMA node.

6.3 Dynamic selection of channels

Finally, we show how our dynamic channel scheduler works and allows Tyche to achieve the maximum throughput even when application buffers are placed in a single NUMA node. Figure 13 shows the throughput achieved by Tyche when using the dynamic and affinity-based schedulers. The dynamic scheduler is set to check for load imbalance every 5 s. We run FIO and zmIO with several application threads and random reads and writes. For this test, we choose a configuration that easily creates load imbalance by using four NICs: three NICs are in NUMA node 0 and one NIC in node 1. Every 60 s, we change the placement of the application threads by placing them as follows:

1. an equal number of threads placed in each node;
2. all threads placed in node 1;
3. all threads placed in node 0;
4. all threads placed again in node 1;

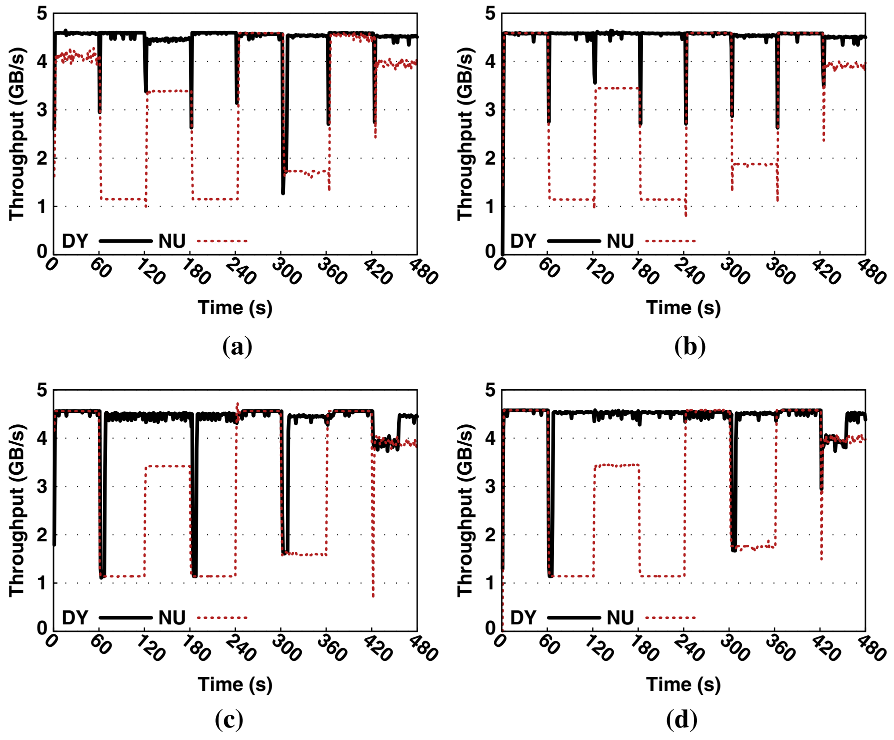

Fig. 13 With four NICs, throughput, in GB/s, achieved by Tyche with an affinity-aware scheduler (NU) and a dynamic scheduler (Dy). **a** FIO writes. **b** FIO reads. **c** zmIO writes. **d** zmIO reads

5. threads placed proportionally between the two nodes, based on the number of NICs in each node: 3/4 of the threads in node 0 and 1/4 in node 1;
6. threads placed in reverse proportion (to the number of NICs) between the two nodes, which is the reverse of interval 5: 1/4 of the threads in node 0 and 3/4 in node 1;
7. an equal number of threads placed again in each NUMA node;
8. threads placed again proportionally between the two nodes, based on the number of NICs in each node, similar to interval 5: 3/4 of the threads in node 0 and 1/4 in node 1.

The request size is always 128 kB, except during intervals six and eight, when 3/4 of the threads issue 128 kB requests, whereas the rest 1/4 issues 4 kB requests.

Figure 13 shows that affinity-based scheduling is not able to achieve the maximum throughput when application threads are placed only in a single node. For instance, during intervals two, three, and four (between seconds 61 and 240), only NICs in a single NUMA node are used, and therefore Tyche only achieves the maximum throughput provided by these NICs. Similarly, during intervals six and eight, NICs from one node are underutilized, and Tyche cannot achieve the maximum possible throughput.

The dynamic scheduler detects during these intervals a large difference in performance between the two NUMA nodes, and switches to a round robin policy. The idea is that using more NICs, even by breaking NUMA affinity, will result in higher throughput. During the rest of the intervals, the dynamic scheduler selects the affinity-aware policy, since it detects that requests are issued in both NUMA nodes and a round robin policy creates unnecessary cross-NUMA node traffic. The short deeps in the black lines signify the points where the dynamic scheduler is adjusting its policy.

Finally, we note that during the first half of interval eight, with zmIO the dynamic scheduler is not able to choose the round robin policy, because the difference in NUMA link utilization between both nodes is smaller than the currently used threshold of 60 %.

7 Synchronization and lock contention

The initiator path can operate in two different modes. In the inline mode, application threads simultaneously submit their requests without performing a context switch. The queue mode uses a context switch to avoid having many threads accessing the send path and incurring a significant synchronization overhead. The queue mode introduces a new queue at the initiator block layer. The application thread inserts its requests in this queue, instead of issuing them directly to the device queue. Tyche threads dequeue requests and send them to the target by following the regular process (see Fig. 4a). The rest of the protocol paths behaves as explained in Sect. 2.

Our previous results [1] show that for reads and writes of small size, the inline mode outperforms the queue mode, whereas the queue mode may outperform the inline mode when there are many outstanding writes of large size. The problem is lock contention in the send path.

Table 4 Locks on the Tyche end-to-end I/O path for the inline and queue modes

Name	Layer	Path	Host	Data structure/task	Held in steps (Fig. 4)
Mes	Block	S, R	I	Message buffers	1, 17
NIC	Net. driver	S	I, T	NIC transmission ring	3, 5, 14
Not	Network	R	I, T	One per notification rings	7, 9, 10, 11, 16, 17
Work	Block	S	T	Work queue	12, 13
Pos	Network	S	I, T	To send positive ack	Not included
Neg	Network	S	I, T	To send negative ack	Not included
Que	Block	S	I	Request queue (queue mode)	Not included

S send, *R* receive, *I* initiator, *T* target

Although Tyche minimizes the synchronization with private resources per channel, each channel still has to grant exclusive access to its data structures, because several threads can simultaneously use the same channel. For the same reason, the NIC driver has to grant exclusive access to its rings as well.

For a write request, Fig. 4 marks with "L" the locks of the end-to-end I/O path in the inline mode. A read request uses the same locks, since the only difference is that data are sent as a completion from the target to the initiator. Table 4, for each lock, gives its name, the layer, path, and host in which the lock is used, the data structure or task protected, and the steps in Fig. 4 in which it is held. All the locks are spin locks.

In addition, Tyche uses atomic operations to control access to other data structures such as its transmission rings. An atomic operation per buffer avoids the (uncommon) case of concurrently processing overlapped messages as well.

For packets and messages, Tyche assigns in advance its position on the receive rings or buffers, so the receive path does not require locks. Sending completions neither requires locks, since the target sends a completion message by using the corresponding request message.

For the initiator send path, a lock contention problem arises with the NIC lock when there is high concurrency, and many threads are issuing write requests of large size. The problem is the large number of data packets that are simultaneously sent, because, for instance, one 128 kB data message corresponds to 16 data packets. Section 8 shows that performance can drop up to 2× due to the lock contention. The contention only appears for the NIC lock because its critical section is larger than the critical section of other locks and it is held per packet, i.e., more times than those that are held per message.

The queue mode solves the contention problem thanks to a queue at the initiator block layer and a lock to control this queue. The lock contention disappears, since the critical section for handling this queue is significantly smaller than the critical section for transmitting a packet though the NIC.

The receive path and the target send path do not have lock contention, because only a few threads are in charge of these paths. Reads have no such problem either, because the target sends data messages back with the work queue.

Table 5 Configurations evaluated in the analysis of the lock contention problem

	Tyche mode	NUMA affinity	
		Tyche	Application
I	Inline	Yes	No
Q	Queue	Yes	No
I-NU	Inline	Yes	Yes
Q-NU	Queue	Yes	Yes

8 Analysis of I/O throughput degradation due to lock contention

We study the wait and lock times per acquisition. The wait time per acquisition is the total time waiting on a lock divided by its number of acquisitions (number of times the lock is held). The hold time per acquisition is the time a lock is held divided by its number of acquisitions. For the NIC lock, we count as acquisitions only the times that the packet is sent. If there is no room on the NIC ring, no work will be done, and the lock will be released very quickly.

We use Tyche with NUMA affinity and we run FIO with and without NUMA affinity. Table 5 describes the configurations and Tyche mode tested.

With three NICs on NUMA node 0, one channel per NIC, and for writes, Fig. 14 provides throughput, in GB/s, achieved by Tyche, and wait and hold times, in μs, per acquisition at the initiator and target. We do not provide wait and hold times for one and four threads, since they are pretty similar to eight threads.

When FIO does not apply affinity, throughput drops significantly with the inline mode. As a result, the queue mode outperforms the inline mode by up to 64.9 %. When FIO applies affinity, there is no difference, although with 128 threads the queue mode still outperforms the inline mode by up to 9.9 %.

The problem of the inline mode is the increased contention for the NIC lock due to the large number of application threads that simultaneously issue requests. However, the queue mode avoids the lock contention by having only three threads per channel that simultaneously issue requests.

At the initiator, the wait time is usually close to zero. However, for 128 threads and the inline mode, the wait time for the NIC lock is significantly increased to 16.99 μs when FIO does not apply affinity and 4.14 μs when it does. This increase confirms that there is contention for the NIC lock.

The hold time for the NIC lock increases with contention for the inline mode and FIO with no affinity. The reason is that the hold time includes the time of false retries (times that the NIC lock is acquired, but released without doing any work because there is no room on the NIC ring). This extra time should be counted as wait time, since no work is done.

The NIC lock has smaller hold times when FIO applies affinity. The critical section of this lock performs the PCI operations to map the memory region of the data to transmit for DMA in streaming mode. These operations are more expensive and the transmission process becomes slower when there is no NUMA affinity, and the pages to send are in a different node than the NIC.

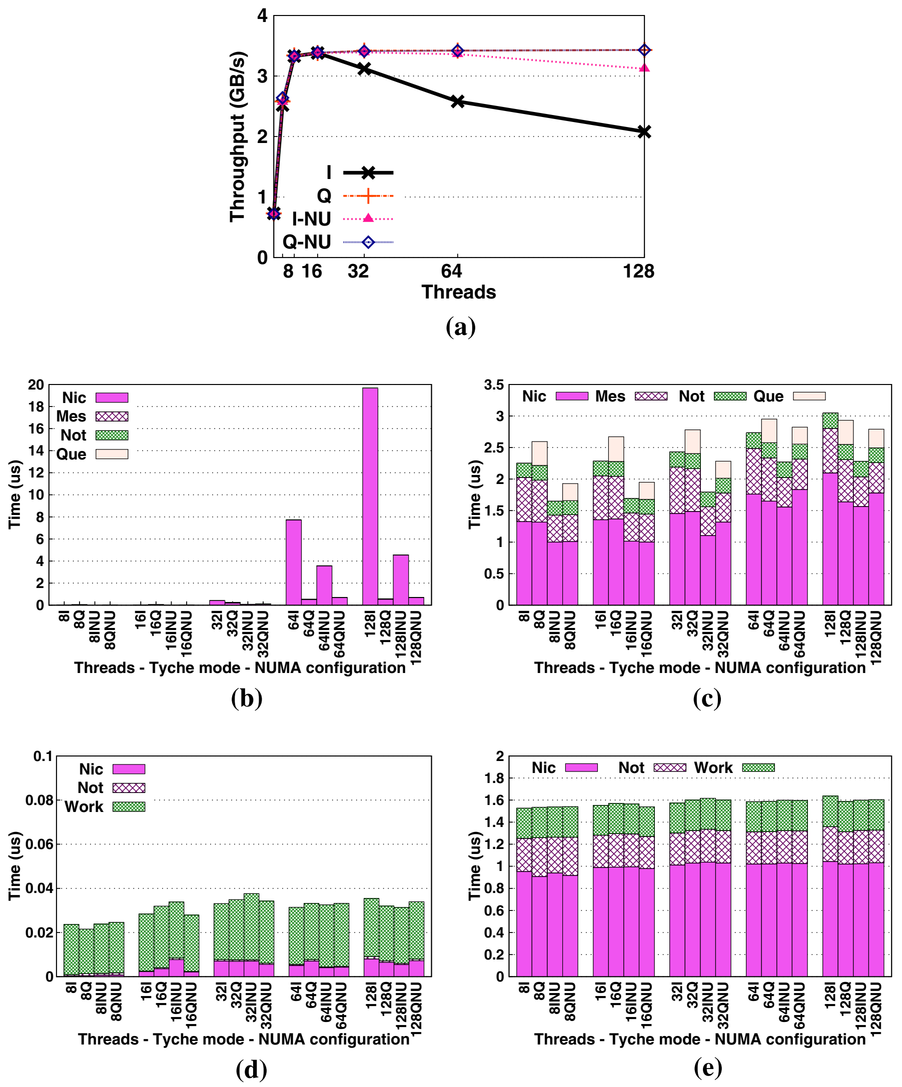

Fig. 14 Throughput, in GB/s, achieved by Tyche, and wait and hold times, in μs, for the locks at the initiator and target, with random writes and three NICs. I and Q evaluate the inline and queue modes, respectively, when the application does not apply NUMA affinity, and I-NU and Q-NU when the application applies affinity. Note that for **c**, the *Y* axis range is [0, 3.5], for **d**, it is only [0, 0.1] and for **e** it is [0, 2]. **a** Write throughput. **b** Wait time at the initiator. **c** Hold time at the initiator. **d** Wait time at the target. **e** Hold time at the target

In addition, when two threads running on different NUMA node acquire a lock, one after the other, the overhead of acquiring the lock is larger than when both threads are running on the same NUMA node [15,16]. The problem is lock migration [15, 16] that occurs when threads run on different nodes, which leads to increase cache invalidations.

Therefore, NUMA is also important for locks and lock contention increases when the application does not take into account NUMA affinity.

The target (Fig. 14d, e) has no lock contention problem, and the wait and hold times are small or negligible. Indeed, the wait time is always smaller than 0.04 μs, and the hold time is always smaller than 1.7 μs.

The lock contention is even worse when using more NICs. With six NICs, a channel per NIC, and for 128 and 512 kB write request sizes, Fig. 15 provides throughput, in GB/s, achieved by Tyche, and wait time and hold time, in μs, per acquisition for the locks at the initiator, as a function of the number of application threads and the Tyche mode.

When FIO does not apply memory placement, for 128 application threads with 128 kB requests and for more than 32 application threads with 512 kB requests, the queue mode significantly outperforms the inline one. The queue mode improves by up to 2$\times$ the throughput obtained by the inline mode. When FIO applies affinity, the difference in throughput is small, and only for 512 kB requests and 128 threads the queue mode slightly outperforms the inline mode by up to 5 %.

The increase in the wait time (see Fig. 15c, d) confirms the lock contention problem. With the inline mode, the wait time of the NIC lock raises up to 19.7 and 4.6 μs when FIO applies NUMA or not, respectively. In the queue mode, NIC lock wait time raises only up to 0.7 μs.

In the inline mode, the hold time per acquisition (Fig. 15e, f) is slightly increased as well, for the reason as with three NICs.

Figure 15a, b shows that, for 128 kB requests up to 64 threads and for 512 kB requests up to 32 threads, the inline mode outperforms the queue mode by up to 24 %. For sizes smaller than 128 kB (not shown), the inline mode also outperforms the queue mode [1]. The context switch made by the queue mode implies an overhead that increases the time needed to issue a request.

Finally, reads have no lock contention problem. With three NICs, one channel per NIC, and reads, Fig. 16 provides Tyche throughput, in GB/s, and wait and hold times, in μs, per acquisition for the locks at the target. There is no difference in performance between both modes. The target sends completion and data messages back. Thanks to the work queue, there is no lock contention. At the initiator (times not shown), the wait times are almost zero, and the hold times are small and do not increase with the number of threads. Note that when more NICs are in use, the inline mode outperforms the queue mode due to the overhead of the additional context switch in the queue mode [1].

9 Impact of NUMA and synchronization when using real storage devices

As discussed, NUMA effects and lock appear when traffic saturates three 10 GbE NICs. The same behavior appears with storage devices if they are able to provide throughput to saturate three 10 GbE NICs. Currently, there are already solid state drives (SSD) that are able to easily provide 500 MB/s of throughput [17,18], whereas high-end devices operate above the 1 GB/s regime [17,19]. Therefore, servers with a small number of devices can generate the amount of traffic required to saturate several

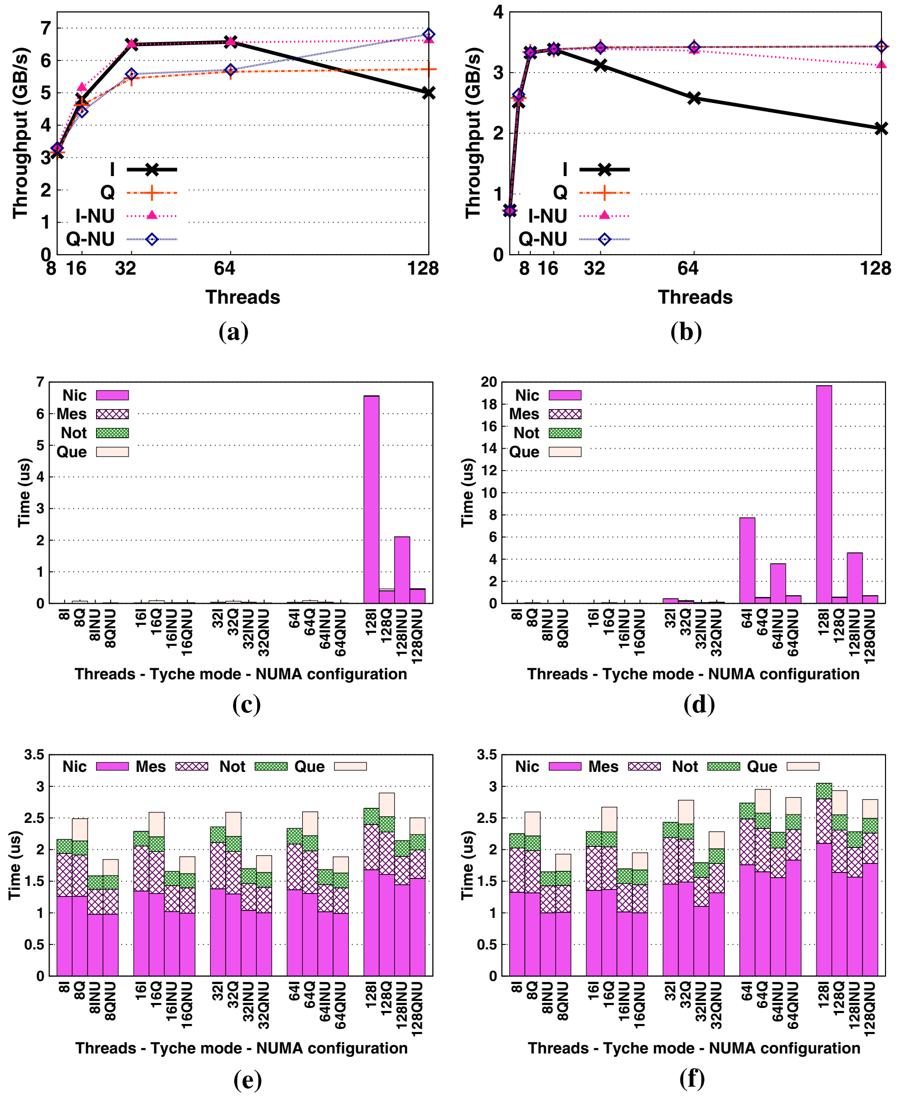

Fig. 15 With six NICs throughput, in GB/s, achieved by Tyche and wait and hold times, in μs, for the locks at the initiator. I and Q evaluate the inline and queue modes, respectively, when the application does not apply NUMA affinity, and I-NU and Q-NU when the application applies affinity. Note that for **e** and **f**, the Y axis range is [0, 3.5]. **a** 128 kB. **b** 512 kB. **c** 128 kB—wait time. **d** 512 kB—wait time. **e** 128 kB—hold time. **f** 512 kB—hold time

NICs. Projections for NVM [20–24] suggest that future devices will exhibit bandwidth and latency characteristics similar to DRAM.

In this section, we analyze the impact of NUMA and lock contention on NVM devices by using an NVM emulator layer [25,26]. We emulate a 12 GB NVM device and use three NICs attached to NUMA node 0. We run FIO with direct I/O, random writes and reads, 128 kB request size, and several threads.

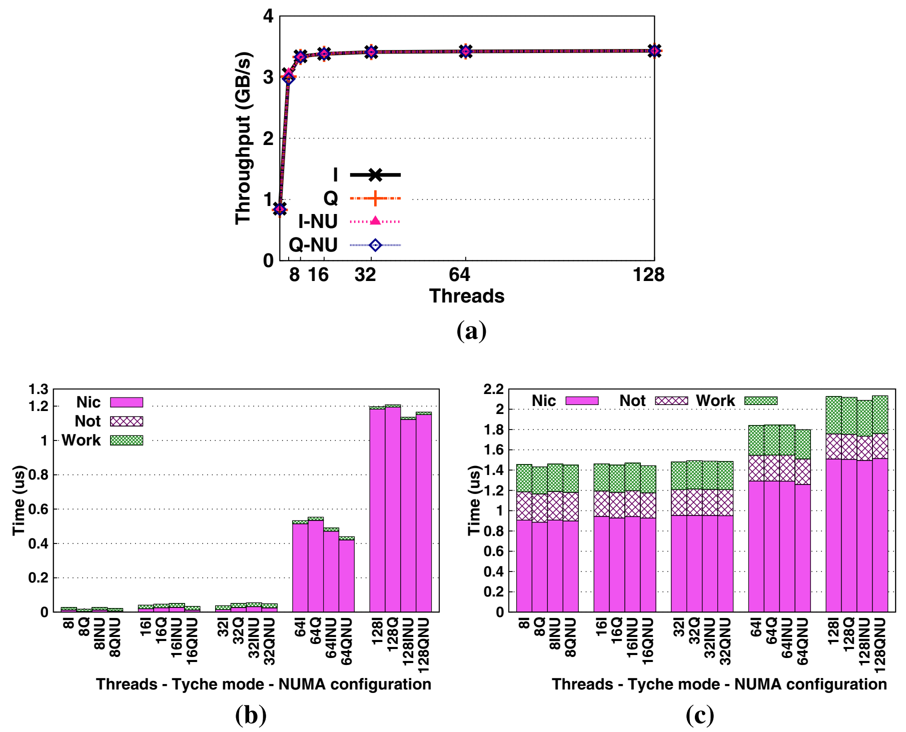

Fig. 16 Throughput, in GB/s, achieved by Tyche, and wait and hold times, in μs, for the lock at the target, with random reads of 512 kB and three NICs. I and Q evaluate the inline and queue modes, respectively, when the application does not apply NUMA affinity, and I-NU and Q-NU when the application applies affinity. For **a**, the curves are overlapping. For **b**, the Y axis range is only [0, 1.3], and for **c** it is [0, 2.2]. **a** Read throughput. **b** Wait time at the target. **c** Hold time at the target

We analyze the NUMA impact with the Ideal and Worst NUMA configurations (see Table 2). Figure 17 depicts the throughput achieved by Tyche depending on the NUMA configuration when using the NVM device. With one and two NICs, there is no difference in performance between both configurations. However, with three NICs, Ideal (perfect placement) significantly outperforms Worst by up to 21.4 %. The difference in performance is due to the data traffic through the QPI links (not shown). When applying perfect placement (Ideal), data traffic mainly comes through the local links. However, when considering the Worst configuration, a significant amount of data traffic comes through the remote links.

We analyze the lock contention by comparing the inline and queue modes, when Tyche applies affinity, and we run FIO with and without affinity. Table 5 describes the configurations and Tyche mode tested. With three NICs on NUMA node 0, one channel per NIC, and for writes, Fig. 18 provides throughput, in GB/s, achieved by Tyche. We do not provide the wait and hold times for the locks since the behavior is quite similar to that shown in Sect. 7. When FIO does not apply affinity, throughput drops significantly with the inline mode, and the queue mode outperforms the inline mode by up to 55.4 %. When FIO applies affinity, there is no difference between

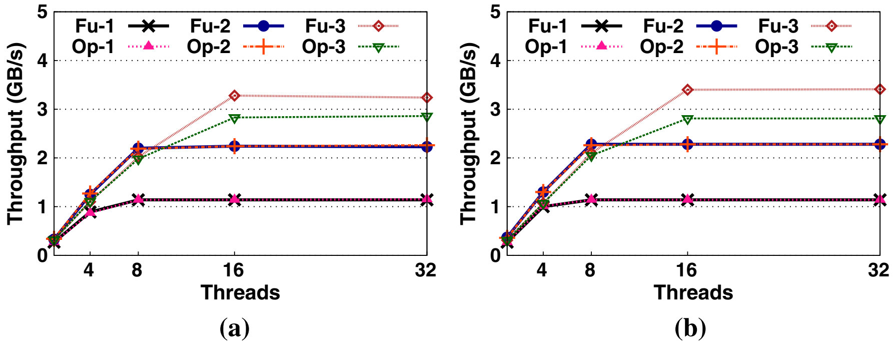

Fig. 17 With the NVM simulator [25,26] throughput achieved by Tyche with direct I/O, random writes and reads, 128 kB request size, and several threads, with (Id) and without (Wo) Tyche affinity over and 1–3 NICs. Id-1, Id-2, and Id-3 correspond to the Ideal NUMA configuration with one to three NICs, respectively, whereas Wo-1, Wo-2 and Wo-3 to the Worst NUMA configuration with one to three NICs, respectively. Table 2 describes the NUMA configurations of Ideal and Worst. Curves for Id-1, Wo-1 and Id-2, Wo-2 are overlapping. **a** Writes. **b** Reads

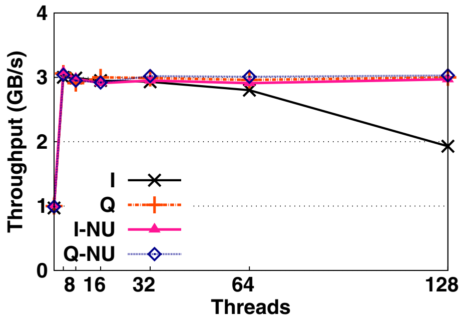

Fig. 18 With the NVM simulator [25,26], throughput, in GB/s, achieved by Tyche when analyzing the lock contention problem. We run FIO with direct I/O, random writes, and 128 kB request size. We use three NICs. I and Q evaluate the inline and queue modes, respectively, when the application does not apply NUMA affinity, whereas I-NU and Q-NU when the application applies affinity

both modes. As explained in Sect. 7, with the inline mode, the contention for the NIC lock is significantly increased due to the large number of application threads that simultaneously issue requests. The queue mode solves this problem since only a few threads per channel issue simultaneously requests.

10 Related work

Recently, there has been a lot of work on for NUMA-aware process scheduling and memory management in the context of many-core and systems. Regarding I/O performance, for instance, Mavridis et al. propose Jericho [6], an I/O stack that consists of a NUMA-aware file system and a DRAM cache organized in slices mapped to NUMA nodes. Their results show that Jericho improves performance up to 2× by doing more than 95 % of memory accesses locally. Zheng et al. [10] propose a scalable user-space cache for NUMA machines. By partitioning the cache by processors, they break the page buffer pool into a large number of small page sets and manage each set independently. Dashti et al. [27] show that the performance is dramatically hurt by congestion on interconnect links and in memory controllers, which results from high volume of data flowing across the system. They propose an algorithm called Carrefour that places threads and memory so as to avoid traffic hotspots and prevent congestion in memory controllers and on interconnect links. Lepers et al. [28] show that the asymmetry of the interconnect in NUMA systems also impacts performance, since performance is more affected by the bandwidth between nodes than by the distance between them. They develop AsymSched, a thread and memory placement algorithm that maximizes the bandwidth for communicating threads. Note that, in this work, we show that NUMA placement is a key aspect to achieve maximum throughput with *network storage* protocols as well.

Moreaud et al. [29] study NUMA effects on high-speed networking in multi-core systems and show that placing a task on a node far from the NIC leads to a performance drop. Their results show that NUMA effects on throughput are asymmetric, since only the target destination buffer appears to need placement on a node close to the interface. In our case, NUMA affects both sides, target and initiator.

Ren et al. [30] propose a system that integrates an RDMA-capable protocol (iSER), multi-core NUMA tuning, and an optimized back-end storage area network. They use the `numactl` utility for binding a dedicated target process to each logical NUMA node and achieve an improvement of up to 19 % in throughput for write operations. Tyche applies NUMA affinity to both the initiator and the target, by defining channels that are mapped to NICs and NUMA nodes, and their resources are allocated in the same node where the NIC is attached. This approach almost entirely eliminates NUMA effects and achieves an improvement of up to 85 %.

Dumitru et al. [31] also analyze, among other aspects, the impact of NUMA affinity on NICs capable of throughput at the range of 40GBits/s, without, however, proposing a solution. Pesterev et al. [32] analyze NUMA effects on TCP connections by proposing Affinity-Accept that ensures that all processing for a given TCP connection occurs on the same core. They reduce the time spent in the TCP stack by 30 % and improve the overall throughput by 24 %. They use the NICs to spread incoming packets among many RX DMA rings to ensure packets from a single flow always map to the same core. However, our study shows that even the NIC resources should be allocated in the same NUMA node where the NIC is attached to obtain maximum performance.

Lock contention has long been recognized as a key impediment to achieve high performance for shared-memory parallel programs [7–10]. Today, the risk for performance loss in shared memory servers due to synchronizations is well understood. For instance, Bjørling et al. [8] demonstrate that, in the Linux block layer, the single lock used for protecting the I/O request queue can become a bottleneck, as device performance approaches 800 thousand IOPS. They also apply NUMA, but only by considering a queue per core and issuing requests in the queue of the corresponding core, but without verifying in which node requests are allocated. To achieve a million random read IOPS, Zheng et al. [10] propose a user-space file abstraction running on an array of commodity SSDs. The file abstraction builds on top of a local file system on each SSD to aggregate their IOPS. To eliminate lock contention in the file and device interfaces they use, for I/O to each SSD, dedicated threads that access the SSD and file exclusively. In our case, we use a queue approach to avoid contention at the NIC lock when there is a high concurrency of large write requests and the application has not applied memory placement.

IX [33] is an operating system that optimizes bandwidth and latency by, among other aspects, eliminating multi-core synchronization. Each thread controls its own flow, having private resources such as memory pools or hardware queues. Each thread serves a single receive and transmit queue per NIC by using multi-queue NICs with receive side scaling (RSS) [34], and eliminating, in this way, the need for synchronization.

Synchronization overhead can also be reduced by multiqueue NICs and per queue processing threads, as proposed by RSS [34]. RSS provides the ability to schedule reception of packets on different processors by using a hash function to select the same core for packets of a single network flow, resulting in both load balance and memory locality, while scaling receive-path processing to multiple cores.

Our previous results [1] show that Tyche is able to achieve maximum NIC performance by using a single NIC queue (at 10 GBits/s) and without using the RSS extensions of our NICs. In fact, in Tyche, a single thread is able to handle the full receive path processing for about two 10 GBit/s NICs, because the processing involved in the receive path is significantly more lightweight compared to TCP. Also, the receive path in Tyche is structured differently with the interrupt handler required to only perform minimal processing and accessing very limited protocol data structures, while the rest of the receive path processing happening in multiple threads that are able to access local-only (to their NUMA node) data structures, because in Tyche receive path buffer allocation and processing is done by channels in a NUMA-aware manner. Therefore, we expect that Tyche can scale receive path processing with single-link speed without requiring any RSS extensions.

Several authors analyze NUMA effects on locks. NUMA-aware locks [15,16,35] are general-purpose mutual-exclusion locks that improve the locality of reference on NUMA architectures. These locks encourage threads with high mutual cache locality to acquire the lock consecutively, thus reducing the overall level of cache coherence misses when executing instructions in the critical section.

With respect to our previous work, we first propose Tyche [1] as a network storage protocol directly on top of Ethernet that does not require hardware support and can be deployed over existing infrastructure. Tyche reduces overheads via a copy reduction technique, storage-specific packet processing, pre-allocation of memory, and using RDMA-like operations without requiring hardware support. Tyche transparently bundles multiple NICs and offers improved scaling with the number of links and cores via reduced synchronization, proper packet queue design, and NUMA affinity management. In our subsequent work [2], we examine how Tyche can achieve low host CPU overhead and high network link utilization for small I/O requests. We show that context switches significantly increase overhead, and we propose a new Tyche design that reduces context switches for low degrees of I/O concurrency. For high degrees of I/O concurrency, and to achieve high link utilization, we propose an adaptive batching mechanism that dynamically calculates the degree of batching based on the throughput obtained. Next, we use Tyche [36] to examine the performance implications of NUMA servers on end-to-end network storage performance. We analyze the impact of memory placement by studying the amount of data traffic through the QPI links. In this work, we extend [36] by providing a more thorough analysis in NUMA effects and by proposing a new dynamic channel scheduler to solve imbalance placements that applications could make. In addition, we also examine in detail the impact of synchronization on network storage protocols and how Tyche handles synchronization to avoid lock contention.

11 Conclusions

In this paper, we analyze and evaluate the impact of NUMA affinity and lock synchronization on networked storage over raw Ethernet, at link speeds beyond 60 Gbits/s.

We examine the amount of data traffic through QPI links. Our analysis shows that NUMA effects can have a large negative impact on performance, reducing network throughput up to 2×. To mitigate NUMA effects, we encapsulate all protocol data structures and flow control in a "channel" that essentially corresponds to end-to-end I/O path. We carefully map channels to NICs and NUMA nodes to ensure proper affinity. Our approach improves throughput by up to 85 %, to a large extent, eliminating inter-processor QPI traffic and NUMA effects. We propose a dynamic channel scheduler that allows Tyche to achieve the maximum throughput even when application buffers are allocated and used in an imbalanced manner across NUMA nodes. Finally, we analyze the impact of synchronization when using multiple cores and NICs. We find that the inline approach where application threads issue requests with no context switch is preferable to using a queuing approach that trades locks for context switches, when NUMA affinity is enabled end to end. Indeed, in theses cases, the inline approach outperforms by up to 24 % the queuing approach.

Overall, network protocols for converged storage over raw Ethernet without hardware support are a viable approach, but NUMA affinity and synchronization should be considered to achieve high network throughput.

Acknowledgments We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the NanoStreams (FP7-ICT-610509) project, the HiPEAC3 (FP7-ICT-287759) Network of Excellence, and the COST programme Action IC1305, 'Network for Sustainable Ultrascale Computing (NESUS)'.

References

1. González-Férez P, Bilas A (2014) Tyche: an efficient Ethernet-based protocol for converged networked storage. In: Proceedings of the IEEE 30th international conference on massive storage systems and technology (MSST)
2. González-Férez P, Bilas A (2015) Reducing CPU and network overhead for small I/O requests in network storage protocols over raw Ethernet. In: Proceedings of the IEEE 31st international conference on massive storage systems and technology (MSST)
3. Intel (2009) An introduction to the Intel® QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html. Accessed 29 Apr 2016
4. Dobson M, Gaughen P, Hohnbaum M, Focht E (2003) Linux support for NUMA hardware. In: Ottawa Linux symposium

5. Lameter C (2006) Local and remote memory: memory in a Linux/NUMA system. In: Ottawa Linux symposium
6. Mavridis S, Sfakianakis Y, Papagiannis A, Marazakis M, Bilas A (2014) Jericho: achieving scalability through optimal data placement on multicore systems. In: Proceedings of the IEEE 30th international conference on massive storage systems and technology (MSST)
7. Anderson TE (1990) The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans Parallel Distrib Syst 1(1):6–16
8. Bjørling M, Axboe J, Nellans D, Bonnet P (2013) Linux block io: introducing multi-queue SSD access on multi-core systems. In: Proceedings of the 6th international systems and storage conference
9. Tallent NR, Mellor-Crummey JM, Porterfield A (2010) Analyzing lock contention in multithreaded applications. SIGPLAN Not 45(5):269–280
10. Zheng D, Burns R, Szalay AS (2013) Toward millions of file system IOPS on low-cost, commodity hardware. In: Proceedings of the international conference on high performance computing, networking, storage and analysis
11. FIO benchmark. http://freecode.com/projects/fio. Accessed 29 Apr 2016
12. zmIO benchmark. http://www.ics.forth.gr/carv/downloads.html. Accessed 29 Apr 2016
13. (Intel) TW (2012) Intel® performance counter monitor—a better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor. Accessed 29 Apr 2016
14. Lock statistics. https://www.kernel.org/doc/Documentation/locking/lockstat.txt. Accessed 29 Apr 2016
15. Dice D, Marathe VJ, Shavit N (2012) Lock cohorting: a general technique for designing numa locks. In: Proceedings of the 17th ACM SIGPLAN symposium on principles and practice of parallel programming. ACM, New York, pp 247–256
16. Dice D, Marathe VJ, Shavit N (2015) Lock cohorting: a general technique for designing NUMA locks. ACM Trans Parallel Comput 1(2):13:1–13:42
17. Intel (2016) Intel SSD data center family. http://www.intel.com/content/www/us/en/solid-state-drives/data-center-family.html. Accessed 29 Apr 2016
18. Samsung (2016) SSD 850 pro 2.5 sata iii 1 tb. http://www.samsung.com/us/computer/memory-storage/MZ-7KE1T0BW. Accessed 29 Apr 2016
19. Samsung (2016) SSD 950 pro nvme 512 gb. http://www.samsung.com/us/computer/memory-storage/MZ-V5P512BW. Accessed 29 Apr 2016
20. (2009) International technology roadmap for semiconductors: emerging research devices
21. Breitwisch M (2008) Phase change memory. In: Proceedings of the interconnect technology conference
22. Coburn J, Bunker T, Schwarz M, Gupta R, Swanson S (2013) From aries to mars: transaction support for next-generation, solid-state drives. In: Proceedings of the twenty-fourth ACM symposium on operating systems principles
23. Dieny B, Sousa R, Prenat G, Ebels U (2008) Spin-dependent phenomena and their implementation in spintronic devices. In: Proceedings of the VLSI technology, systems and applications
24. Meena JS, Sze SM, Chand U, Tseng TY (2014) Overview of emerging nonvolatile memory technologies. Nanoscale Res Lett 9(526):1–33
25. (2016) Persistent memory block driver. https://github.com/linux-pmbd/pmbd. Accessed 29 Apr 2016
26. Chen F, Mesnier MP, Hahn S (2014) A protected block device for persistent memory. In: Proceedings of the IEEE 30th international conference on massive storage systems and technology (MSST)
27. Dashti M, Fedorova A, Funston J, Gaud F, Lachaize R, Lepers B, Quema V, Roth M (2013) Traffic management: a holistic approach to memory placement on NUMA systems. In: Proceedings of the eighteenth international conference on architectural support for programming languages and operating systems
28. Lepers B, Quema V, Fedorova A (2015) Thread and memory placement on numa systems: asymmetry matters. In: 2015 USENIX annual technical conference (USENIX ATC'15). USENIX Association, Santa Clara, pp 277–289. https://www.usenix.org/conference/atc15/technical-session/presentation/lepers. Accessed 29 Apr 2016
29. Moreaud S, Goglin B (2007) 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
30. Ren Y, Li T, Yu D, Jin S, Robertazzi T (2013) 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

31. Dumitru C, Koning R, Laat C (2011) 40 gigabit ethernet: prototyping transparent end-to-end connectivity. In: Proceedings of the Terena networking conference
32. Pesterev A, Strauss J, Zeldovich N, Morris RT (2012) Improving network connection locality on multicore systems. In: Proceedings of the 7th ACM European conference on computer systems
33. Belay A, Prekas G, Klimovic A, Grossman S, Kozyrakis C, Bugnion E (2014) IX: a protected dataplane operating system for high throughput and low latency. In: Proceedings of the 11th USENIX symposium on operating systems design and implementation (OSDI'14)
34. Microsoft Coorporation (2014) Introduction to receive side scaling. https://msdn.microsoft.com/library/windows/hardware/ff556942.aspx. Accessed 29 Apr 2016
35. Radovic Z, Hagersten E (2003) Hierarchical backoff locks for nonuniform communication architectures. In: Proceedings of the 9th international symposium on high-performance computer architecture. IEEE Computer Society, Washington, DC, pp 241–252
36. González-Férez P, Bilas A (2015) NUMA impact on network storage throughput over high-speed raw Ethernet. In: Proceedings of the international workshop of sustainable ultrascale network