View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universidad Carlos III de Madrid e-Archivo

Proceedings of the Second International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2015)

Krakow, Poland

Jesus Carretero, Javier Garcia Blas
Roman Wyrzykowski, Emmanuel Jeannot.
(Editors)

September 10-11, 2015

NUMA impact on network storage protocols over high-speed raw Ethernet

Pilar González-Férez† and Angelos Bilas‡

†Universidad de Murcia, Spain, pilar@ditec.um.es

‡FORTH-ICS and University of Crete, Greece, bilas@ic.forth.gr

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. In our previous work we have designed Tyche, a network protocol for converged storage that bundles multiple 10GigE links transparently and reduces protocol overheads over raw Ethernet without hardware support. However, current technology trends and server consolidation dictates building servers with large amounts of resources (CPU, memory, network, storage). Such servers need to employ Non-Uniform Memory Architectures (NUMA) to scale memory performance. NUMA introduces significant problems with the placement of data and buffers at all software levels.

In this paper, we first use Tyche to examine the performance implications of NUMA servers on end-to-end network storage performance. Our results show that NUMA effects have significant negative impact and can reduce throughput by almost 2x on servers with as few as 8 cores (16 hyper-threads). Then, we propose extensions to network protocols that can mitigate this impact. We use information about the location of data, cores, and NICs to properly align data transfers and minimize the impact of NUMA servers. Our design almost entirely eliminates NUMA effects by encapsulating all protocol structures to a "channel" concept and then carefully mapping channels and their resources to NICs and NUMA nodes.

Keywords NUMA, memory affinity, network storage, Tyche, I/O throughput

I. Introduction

Technology trends for efficient use of infrastructures dictate that storage converges with computation by placing storage devices, such as NVM (Non-Volatile Memory) 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 SAN (Storage Area Network) and NAS (Network Attached Storage) 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.

For efficiency and scalability purposes modern servers tend to employ multiple resources of each kind, namely processors, memories, and storage/network links, in Non-Uniform Memory Access (NUMA) architectures (Figure 1).

Scaling networked storage throughput on such servers is becoming an important challenge. NUMA servers use multiple processor sockets with memory attached to each socket, resulting in non-uniform latencies from processor to different memories. In NUMA architectures each I/O device is attached to a specific NUMA node via an I/O hub (Figure 1). Processors, memories, and I/O hubs are connected through high-speed interconnects, e.g. QPI [1]. I/O requests as well DMA transfers to and from devices are routed through the memory-processor interconnect. Accessing remote memory (in a different NUMA node) incurs

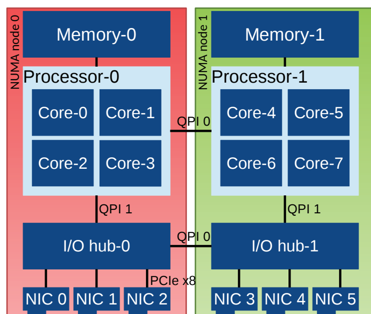
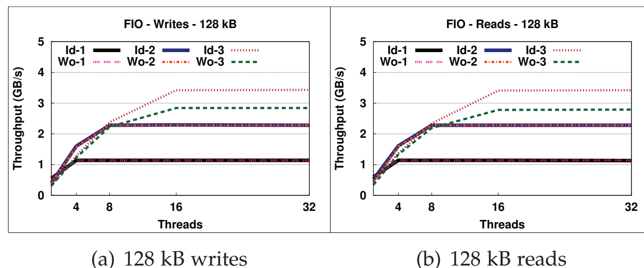


Figure 1: Internal data paths in NUMA servers.



(a) 128 kB writes (b) 128 kB reads

Figure 2: FIO [7] throughput with direct I/O, random writes and reads, 128kB and several threads, with (Id) and without (Wo) Tyche affinity over and 1, 2 and 3 NICs. Note that curves for Id-1, Wo-1 and Id-2, Wo-2 are overlapping.

significantly higher latency than accessing local memory [2, 3], up to a factor of 2x. In addition, it consumes throughput from the inter-processor link(s). Thus, for I/O performance and scalability purposes, it is important to explore how the network protocol can be designed to cater for affinity among memory, processing cores, and network interfaces (NIC) for data and protocol data structures.

In our previous work Tyche [4, 5] we examine the design of network storage protocols over raw Ethernet to achieve 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, RDMA (remote direct memory access)-type communication primitives, memory pre-allocation, and transparent NIC bundling.

In our previous work we observe that NUMA affinity is an important issue that spans the whole I/O path and has a significant performance impact.

Therefore, in this work, we use Tyche to analyze in detail the impact of NUMA affinity on networked storage access. In addition, we examine whether we can mitigate these performance effects on NUMA servers by properly re-designing the network protocol.

We evaluate this issue on two Linux servers with 8 cores (16 hyper-threads) and 6 x 10GigE Myricom 10G-PCIE-8A-C NICs attached to each server. We analyze the data traffic transferred through the QPI links with the Intel ® Performance Counter Monitor (PCM) [6].

Our results show that NUMA effects indeed have a very large negative impact on performance and they can reduce throughput by almost 2x. Our re-designed protocol that detects and uses NUMA affinity across buffers, threads, and NICs, improves I/O throughput by up to 85%.

This behavior is shown in Figure 2 for two configurations: Ideal (Id) case that allocates all the resources in NUMA node 0, and Worst (Wo) case that allocates them in node 1. We use 1, 2, and 3 NICs, all of them attached to the node 0. For 1 and 2 NICs, there is no difference in performance between both configurations. However, when using 3 NICs Ideal significantly outperforms Worst up to 23.7%. When using 6 NICs, this difference is larger. For instance, when the in-house micro-benchmark zmIO [8] is run with direct I/O and random 64kB read requests, Tyche obtains only 3.61 GB/s when no affinity is applied, whereas it achieves 6.67 GB/s when NUMA is taken into account. Additionally, our analysis shows that this difference in performance is reflected in QPI traffic. With NUMA affinity, data traffic mainly comes through the local QPI-1 link. Without NUMA affinity, a large amount of traffic comes through QPI-0 links or the remote QPI-1, and performance drops significantly. Sections IV and V present these results in more detail.

To mitigate these NUMA effects at high network throughput, we carefully design the send and receive paths. We encapsulate important 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 and allocate their resources in the same node where the NIC is attached.

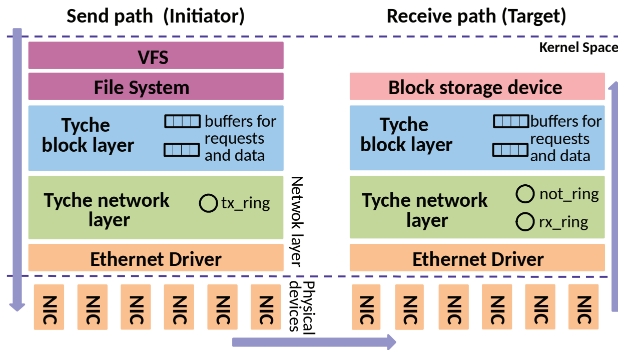


Figure 3: Overview of the send and receive path from the initiator (client) to the target (server).

I/O requests can use any channel. We dynamically detect the appropriate channel for each request, based on the location of the request data, and we direct each request accordingly. This approach aligns buffers and NICs and almost entirely eliminates NUMA effects.

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

The rest of this paper is organized as follows. Section II presents the necessary background on Tyche. Section III describes how Tyche deals with NUMA affinity. Sections IV, V and VI present our experimental results. Finally, we present related work in Section VII and we draw our conclusions in Section VIII.

II. Background

Tyche [4, 5] 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 (Figure 3). Tyche presents the remote storage device locally by creating at the client (initiator) a virtual local 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 reduces overheads for small I/O requests by avoiding context switches for low degrees of I/O concurrency and by dynamically batching messages for high degrees of I/O concurrency. Tyche does not use RDMA over Ethernet, instead our protocol uses a similar, memory-oriented abstraction that allows us to reduce messaging overhead by avoiding packing and unpacking steps that are required over streaming-type abstractions, such as sockets. 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 8kB.

Tyche uses the concept of a communication "channel" to establish a connection between initiator and target. Each channel allows a host to send/receive data to/from a remote host. A channel is directly associated to the NIC that uses 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 single channel per NIC.

Each channel has two pre-allocated buffers, one for each kind of message, for sending and receiving 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 and receiving data messages, and for issuing I/O requests to the local device. The initiator has no pre-allocated pages, it uses the pages of the I/O requests.

The initiator send path can operate in two modes. In the "inline" mode (Figure 3), the application context issues requests to the target with no context switch. In the "queue" mode, requests are inserted in a queue at the block level, and a thread dequeues them and

issues them. There is no other difference. Regarding performance, the inline mode outperforms the queue mode for small requests; the queue mode significantly outperforms the inline mode when there are many outstanding writes of large size. 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. When several channels are mapped to the same NIC, this thread will process packets for all the channels. At the block layer, several threads per channel process I/O requests.

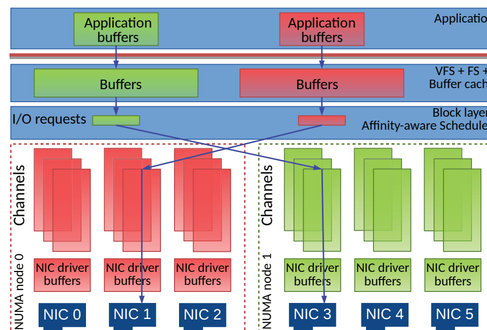


Figure 4: Affinity-aware scheduler selecting a channel for two I/O requests in a Tyche system with six NICs, three per NUMA node, and three channels per NIC.

III. Protocol design for NUMA affinity

To achieve high throughput in a NUMA architecture such as the one depicted in Figure 1 we need to consider affinity among different resources [2, 3]. In the 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. Tyche orchestrates affinity of memory and threads by considering the system topology and the location of NICs. It creates a communication channel per NIC, and associates resources exclusively with a single channel.

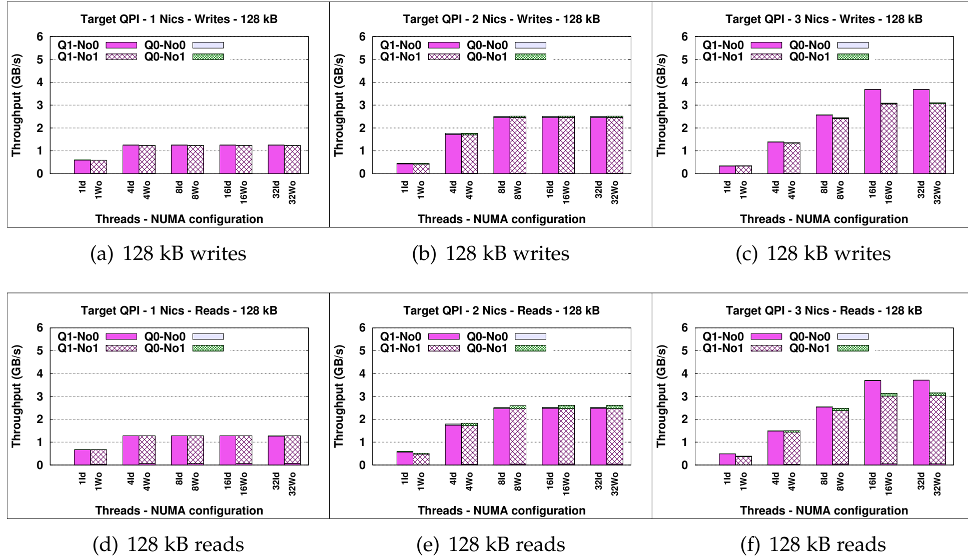
Each channel allocates memory for all purposes and pins its threads to the same NUMA node where its NIC is attached. For instance, in the architecture of Figure 1 a channel mapped to NIC-0 uses memory in Memory-0 and runs its threads in cores within Processor-0.

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 because in the Linux kernel we use it is not possible to apply affinity during assignment of tasks to work queues. Our work queue launches a thread per core that is pinned in its corresponding core. The target submits completion messages to the work queue by using its NUMA information. Conceptually, there is a work queue per channel.

There are a few remaining parts of the end-to-end path that are not affinity-aware: In the Linux kernel (a) it is not possible to control 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 placement of device I/O completions (on the target side). Next, we discuss how we deal with (a) and (b), whereas our results show that the impact of (c) is not significant.

To deal with affinity of 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. Figure 4 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.



(a) 128 kB writes (b) 128 kB writes (c) 128 kB writes

(d) 128 kB reads (e) 128 kB reads (f) 128 kB reads

Figure 5: QPI traffic, in GB/s, for Ideal (Id) and Worst (Wo), with FIO, 128 kB requests, and random writes and reads, and 1, 2 and 3 NICs.

IV. Experimental environment

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 node is equipped with 48 GB DDR-III DRAM, and the initiator with 12 GB. The target uses 12 GB as main memory, and 36 GB as ramdisk. Each node has 6 Myricom 10G-PCIE-8A-C cards that are capable of about 10 Gbits/s throughput in each direction.

We use the open-source Intel ® Performance Counter Monitor (PCM) [6], that provides core-level CPU information, and supports different kind of metrics. We use the estimation of data traffic transferred through the Intel ® QuickPath interconnect links. For each node, the tool provides data traffic for "QPI-1" (inside the same node), and for "QPI-0" (traffic coming from a remote node). We analyze **Q1-No0**, **Q1-No1**, **Q0-No0** and **Q0-No1**, that correspond to the traffic for QPI-1 and QPI-0 in NUMA nodes 0 and 1, respectively. The traffic coming through "QPI-0" link is the traffic between processors (Figure 1), Q0-No0 corresponds to the data traffic to Processor-0 from Processor-1, and Q0-No1 to the traffic in the other direction. The tool does not provide the traffic between I/O hubs.

We use two micro-benchmarks: FIO (Flexible I/O) is a workload generator with many parameters, including number of threads, synchronous and asynchronous operations, request size, access pattern, etc. [7]. zmIO 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 [8]. In this work we start from the Tyche version implemented in Linux kernel 2.6.32 [4]. Tyche uses the queue mode (see Section II) to avoid lock contentions [4].

V. Degradation of I/O throughput due to NUMA

We first perform an analysis for the impact of NUMA effects on the throughput of networked storage I/O and the associated QPI traffic.

We use the baseline version of Tyche that has no support for NUMA effects. We examine the extent of

NUMA impact on throughput as follows. We attach 1, 2, or 3 NICs to NUMA node 0 and create one channel per NIC with round-robin scheduler. Given that all the NICs are on the NUMA node 0, the role of the scheduler is minimal. Then, we create two extreme configurations: Ideal and Worst. In Ideal, we manually allocate memory and threads for Tyche and the benchmark in node 0 where the NICs are also attached. In Worst, we allocate all memory and threads for Tyche and the benchmark in NUMA node 1.

We use FIO with direct I/O, random reads and writes of 128kB, and 60s of runtime. We run the test with 1, 4, 8, 16, and 32 application threads. The storage device is accessed in a raw manner (no file system). Figure 2 provides throughput, in GB/s, achieved by Tyche as a function of the number of application threads.

To analyze the QPI traffic due only to our networked storage protocol and exclude traffic due to the storage device, the target does not use the ramdisk during this test, and it completes the requests without performing the actual I/O. Figure 5 depicts, in GB/s, the traffic through each QPI-node link at the target during the test execution as a function of the number of application threads.

Regarding performance, with 1 and 2 NICs, both configurations obtain the same throughput, and there is no difference between them. With 3 NICs, Ideal significantly outperforms Worst by up to 23.7%. As we explain below, the reason is a bottleneck that appears on the QPI path followed by the data.

Regarding the QPI analysis, with Ideal, since both, NICs and resources, are in NUMA node 0, the QPI traffic is only through Q1-No0 (the Q1 link at NUMA node 0), and the throughput provided by this link is quite similar to the throughput provided by Tyche. There is no data traffic through Q0-No0, Q0-No1, and Q1-No1.

With Worst, the behavior is rather different, and the QPI traffic is through Q1-No1 (the Q1 link at NUMA node 1). The data goes through QPI-1 that connects I/O hub-1 with Processor-1 (Q1-No1) and through the QPI link that connects the I/O hubs (this traffic is not reported by the tool).

With 1 or 2 NICs, the data traffic generated does not saturate this path and the system achieves maximum performance. With 3 NICs, the amount of data traffic generated saturates this path and QPI-0 becomes the bottleneck. QPI and Tyche throughput drop by up to 26.2% and 23.9%, respectively.

Table 1: Configuration of the tests run for the NUMA study. RR stands for round robin scheduling

Test	NUMA affinity		Channel
	Tyche	Application	scheduler
Ideal	Yes	Yes	Affinity-aware
TyNuma	Yes	No	RR
Worst	No	No	RR

VI. Improvement due to protocol NUMA extensions

We now analyze the impact of NUMA effects depending on memory placement applied by Tyche and the application. To perform this analysis, we evaluate three configurations: Ideal, TyNuma and Worst. Table 1 summarizes these configurations. With Ideal, we manually configure the NUMA placement of the application: half of the application threads and their corresponding resources are allocated in each NUMA node. With TyNuma and Worst, we run the application without any affinity hint.

To perform this set of experiments, we use six NICs, three on each NUMA node, and we open one channel per NIC. Now, the target uses the ramdisk, and it performs the actual I/O. Consequently, at the target, there is data traffic due to the network traffic and due to the copy of the ramdisk, and we are not able to distinguish between them. Therefore, we only analyze the QPI traffic at the initiator.

To achieve maximum performance, NUMA affinity should be applied not only by Tyche but also by the application. Therefore, the performance achieved depends also on the placement performed by the application. In addition, it is interesting to see if our protocol extensions can hurt performance for hand-tuned applications. For this reason, we examine two cases: (a) Regular applications that are not tuned for NUMA. For this purpose we use zmIO that allocates resources (threads and buffers) without any particular attention to NUMA. (b) Hand-tuned applications. For

this purpose we use FIO that allocates resources in a balanced manner.

Do protocol extensions for NUMA help performance? We perform the analysis with zmIO, because when zmIO is run without affinity hint, it allocates 99% of writes and around 75% of reads to a single NUMA node (node-0). Therefore, almost all writes issued to channels allocated in node 1 have their resources allocated in node 0, and for reads, this rate is only 50%. Consequently, with zmIO, performance also depends on the request type.

We run zmIO with random reads and writes, direct I/O, request sizes of 64 kB, 128 kB, and 512 kB, and a runtime of 60 s. The remote storage device is accessed as a raw device (no file system). We run 1, 8, 16, and 32 application threads. Since this test is base on time, each time a different amount of data is read or written. Figure 6 provides throughput, in GB/s, achieved by Tyche as a function of the number of application threads. Figure 7 depicts the percentage of the total traffic through each QPI-node link as a function of the application threads and configuration.

For writes, Figure 6 shows that only by applying the right placement, Ideal configuration, Tyche achieves its maximum throughput, being 6.77 GB/s with 32 threads and 512 kB request size. Figure 7 shows that with Ideal, almost all the data traffic comes through the QPI-1 link, having a similar amount of traffic both nodes.

With Worst (the opposite case), Tyche only obtains up to 4.67 GB/s again with 32 threads and 512 kB request size. Indeed, by applying affinity, Ideal outperforms Worst by up to 85%. Figure 7 shows that for writes, Worst only has data traffic through the QPI-1 link on node 0, since almost all the user requests are allocated in this node. There is no data traffic through QPI-1 on node 1, and there is a significant amount of traffic through QPI-0 as well.

With TyNuma, Tyche only achieves up to 5.00 GB/s again with 32 threads and 512 kB request size. Ideal improves throughput up to 37.60%. Figure 7 shows that TyNuma behaves like Worst, and the data traffic through the QPI-1 link is mainly on node 0.

For writes, due to the QPI data traffic (see Figure 7), Worst and TyNuma are not able to provide better performance. TyNuma outperforms Worst because, at the Target, TyNuma is applying NUMA affinity, whereas Worst is not.

For reads, Ideal and TyNuma achieve up to 6.86 GB/s and 6.78 GB/s, respectively, whereas, Worst only up to 4.79 GB/s. Ideal improves throughput by up to 84.8% comparing with Worst.

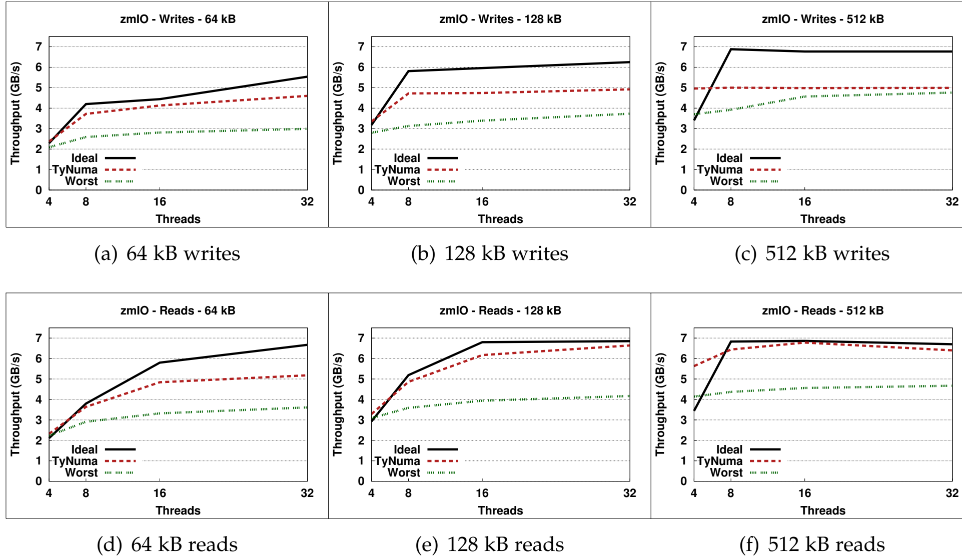
As we can see in Figure 7, this difference in performance between Ideal and Worst is due to the QPI traffic. With reads, Ideal has all the data traffic through the QPI-1 links, having both nodes the same amount of traffic. However, Worst has up to 33% of the total traffic coming through QPI-0.

When comparing Ideal and TyNuma, there only is a small difference in throughput and they exhibit a quite similar behavior. However, regarding QPI traffic, TyNuma behaves more similar to Worst. With TyNuma, at the initiator the QPI traffic is quite similar to with Worst, but at the target, the QPI traffic is the same as with Ideal. At the target, Ideal and TyNuma are applying memory and threads placement, and having a quite similar behavior. With Worst, the target does not apply affinity, so a significant amount of traffic goes through QPI-0 links and consequently the throughput drops significantly. At the initiator, since application buffers are allocated there, QPI traffic with TyNuma behaves like with Worst, since, with both, the application is not applying NUMA affinity, and the application buffers are only allocated at the initiator.

Note that with only 4 threads, all of them, and their resources, are allocated in NUMA node 0, for this reason, with Ideal, there is only data traffic through the QPI-1 link of node 0.

Do protocol extensions for NUMA hurt performance for hand-tunned applications? We analyze the QPI traffic with FIO. When we run FIO with no affinity hint, FIO, by itself, makes a quite balanced distribution of resources. Consequently, even selecting the channel in a round-robin order, around 50% of the requests are issued through a channel allocated in the same node where the request's buffers are allocated.

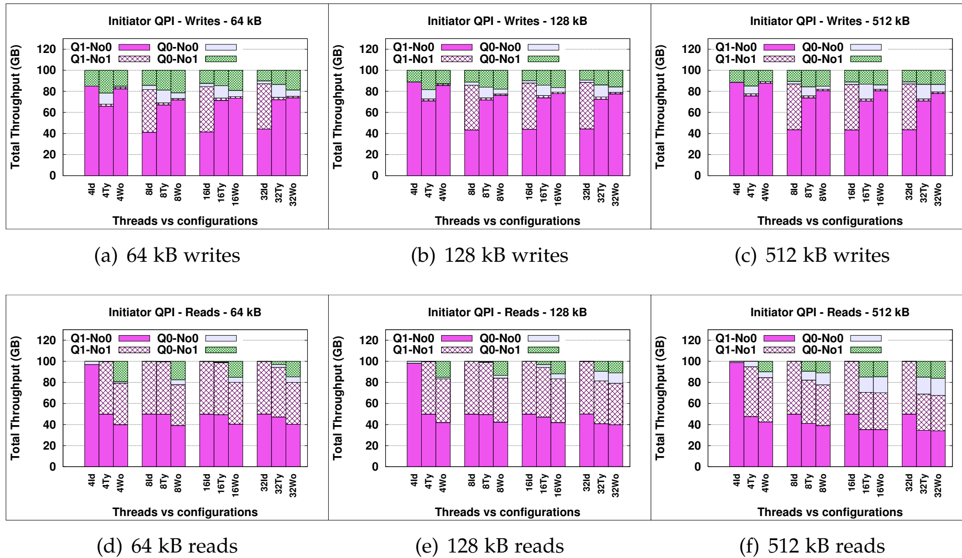
We use FIO with random reads and writes, direct I/O, a 256 MB file size, and 4 kB, 128 kB, and 512 kB request sizes. We run 1, 4, 8, 16, and 32 application threads. Each thread has its own file and makes 30



(a) 64 kB writes (b) 128 kB writes (c) 512 kB writes

(d) 64 kB reads (e) 128 kB reads (f) 512 kB reads

Figure 6: Throughput, in GB/s, achieved by Tyche depending on the affinity, with zmIO for 64 kB, 128 kB and 512 kB requests, and random writes and reads.

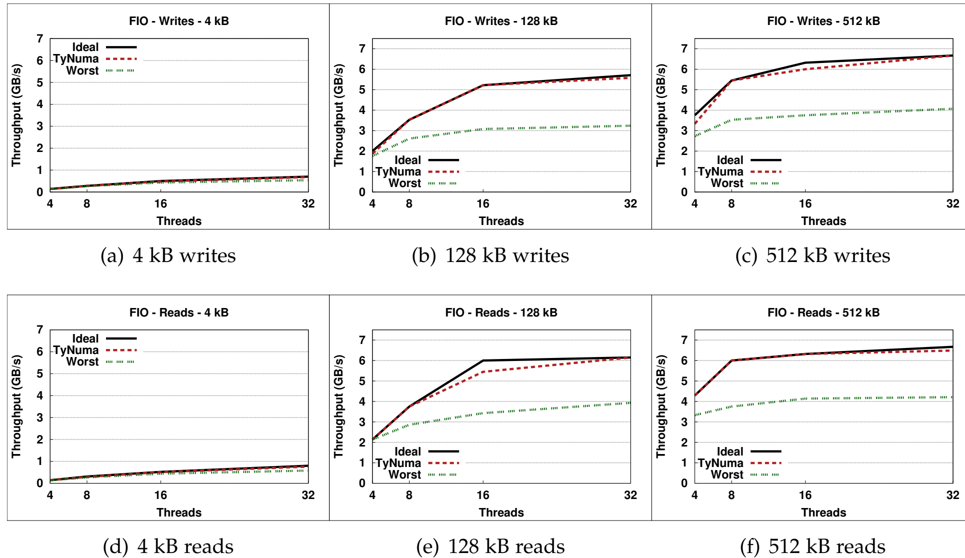


(a) 64 kB writes (b) 128 kB writes (c) 512 kB writes

(d) 64 kB reads (e) 128 kB reads (f) 512 kB reads

Figure 7: Percentage of QPI for Tyche depending on the affinity, with zmIO for 64 kB, 128 kB and 512 kB request size, and random writes and reads.

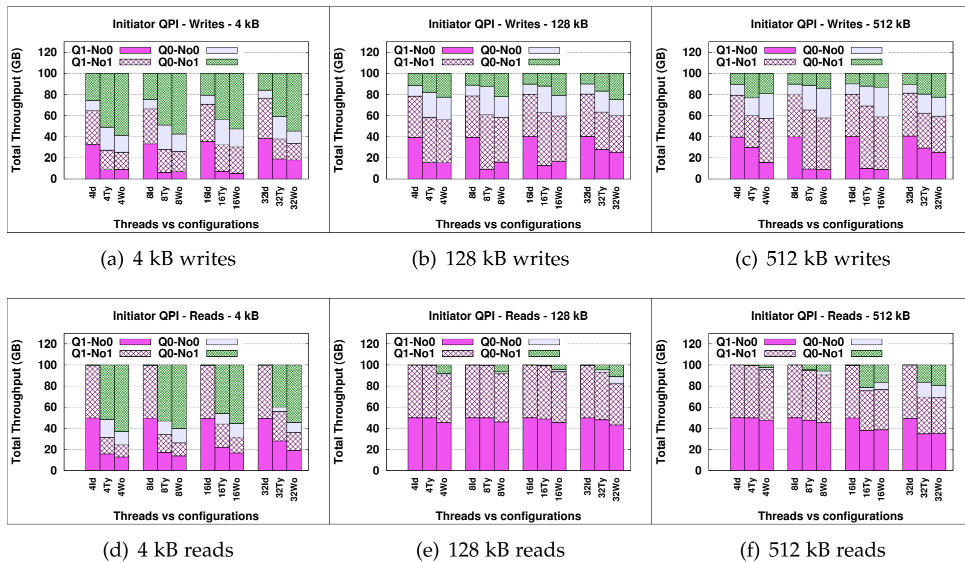
iterations over it, thus, with all the request sizes, the same amount of data is always read or written. We use XFS as the file system. Figure 8 provides throughput, in GB/s, achieved by Tyche as a function of the number of application threads. Figure 9 depicts the percentage of the total traffic through each QPI-node link as a



(a) 4 kB writes (b) 128 kB writes (c) 512 kB writes

(d) 4 kB reads (e) 128 kB reads (f) 512 kB reads

Figure 8: Throughput, in GB/s, achieved by Tyche depending on the affinity, with FIO for 4 kB, 128 kB and 512 kB request size, and random writes and reads.



(a) 4 kB writes (b) 128 kB writes (c) 512 kB writes

(d) 4 kB reads (e) 128 kB reads (f) 512 kB reads

Figure 9: Percentage of QPI for Tyche depending on the affinity, with FIO for 4 kB, 128 kB and 512 kB request size, and random writes and reads.

function of the application threads and configuration.

Figure 8 shows that with Ideal, Tyche obtains its maximum throughput, being 6.67 GB/s for writes and reads, with 32 threads and 512 kB request size. Whereas, with Worst, Tyche obtains only up to 4.21 GB/s and 4.07 GB/s for writes and reads, re-

spectively, also with 32 threads and 512 kB request size. Indeed, by applying the right placement (Ideal configuration), Tyche improves throughput up to 76% (32 threads and 128 kB writes) when comparing with the Worst configuration.

As we can see in Figure 9, this difference in performance is due to the QPI traffic, as explained for reads with zmIO. With Ideal, almost all the traffic comes through the QPI-1 link, and both nodes have a similar amount of traffic. But with Worst, a significant amount of traffic, up to 48% of the total, comes through the QPI-0 link.

Note that although Ideal exhibits perfect placement, for writes, there is traffic at QPI-0 due to cacheline invalidations to the remote NUMA node. With reads, this type of traffic is not present.

TyNuma behaves similar to Ideal due to the QPI traffic at the target, as explained for zmIO. The target applies affinity for both Ideal and TyNuma, so they behave quite similar. In Worst, the target does not apply affinity, therefore a significant amount of traffic goes through QPI-0 and the throughput drops significantly.

This behavior is presented also for small requests. For 4 kB requests, memory placement has also a significant impact, and Ideal improves throughput up to 66.2% and 44.6% for writes and reads, respectively, compared to Worst. Again, there is a small difference in performance between Ideal and TyNuma.

VII. Related work

A lot of work has been done for NUMA-aware process scheduling and memory management in the context of many-core processors and systems. Regarding I/O performance, for instance, Mavridis *et al.* propose Jericho [9], 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 local. 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 manages each set independently. Note that, in this work, we show that NUMA placement is a key aspect to achieve maximum throughput with *network storage* protocols as well.

Regarding NUMA and networking, Moreaud *et al.* [11] 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.* [12] 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, and achieve an improvement of up to 19% in throughput for write operations. In contrast, Tyche applies NUMA affinity, at both initiator and target, by defining channels that are mapped them to NICs and to 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.* [13] 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. Pesterev *et al.* [14] analyze NUMA effects on TCP connections by proposing Affinity-Accept that ensures that all processing for a given TCP connection to occur on the same core. They reduce time spent in the TCP stack by 30% and improves 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.

VIII. Conclusions

Here, we analyze and evaluate the impact of NUMA affinity on the network layer supporting the converged storage paradigm over high-speed Ethernet. We analyze the impact of memory placement by studying

the amount of data traffic through the Intel ® QPI links. This analysis shows that NUMA effects can have a large negative impact on performance, reducing network throughput up to 2x.

To mitigate NUMA effects, we encapsulate all protocol data structures and flow control in "channels" that essentially correspond to the structures required to serve a request through the full end-to-end I/O path. Then, 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.

Acknowledgment

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] An Introduction to the Intel® QuickPath Interconnect. http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html, 2009.

[2] Matthew Dobson, Patricia Gaughen, Michael Hohnbaum, and Erich Focht. Linux Support for NUMA Hardware. In *Ottawa Linux Symposium*, 2003.

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

[4] Pilar González-Férez and Angelos Bilas. Tyche: An efficient Ethernet-based protocol for converged networked storage. In *Proceedings of the IEEE Conference on MSST*, 2014.

[5] Pilar González-Férez and Angelos Bilas. Reducing CPU and network overhead for small I/O requests in network storage protocols over raw Ethernet. In *Proceedings of the IEEE Conference on MSST*, 2015.

[6] Thomas Willhalm (Intel). Intel ® Performance Counter Monitor - A better way to measure CPU utilization. https://software.intel.com/en-us/articles/intel-performance-counter-monitor, 2012.

[7] FIO Benchmark. http://freecode.com/projects/fio.

[8] zmIO Benchmark. http://www.ics.forth.gr/carv/downloads.html.

[9] Stelios Mavridis, Yannis Sfakianakis, Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Jericho: Achieving Scalability through Optimal Data Placement on Multicore systems. In *Proceedings of the IEEE Conference on MSST*, 2014.

[10] Da Zheng, Randal Burns, and Alexander S. Szalay. 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*, 2013.

[11] Stéphanie Moreaud and Brice 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.

[12] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas 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.

[13] Cees de Laat Cosmin Dumitru, Ralph Koning. 40 Gigabit Ethernet: Prototyping Transparent End-to-End Connectivity. In *Proceedings of the Terena Networking Conference*, 2011.

[14] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.