Reducing CPU and network overhead for small I/O requests in network storage protocols over raw Ethernet

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

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

***Abstract*—**

Small I/O requests are important for a large number of modern workloads in the data center. Traditionally, storage systems have been able to achieve low I/O rates for small I/O operations because of hard disk drive (HDD) limitations that are capable of about 100-150 IOPS (I/O operations per second) per spindle. Therefore, the host CPU processing capacity and network link throughput have been relatively abundant for providing these low rates. With new storage device technologies, such as NAND Flash Solid State Drives (SSDs) and non-volatile memory (NVM), it is becoming common to design storage systems that are able to support millions of small IOPS. At these rates, however, both server CPU and network protocol are emerging as the main bottlenecks for achieving large rates for small I/O requests.

Most storage systems in datacenters deliver I/O operations over some network protocol. Although there has been extensive work in low-latency and high-throughput networks, such as Infiniband, Ethernet has dominated the datacenter. In this work we examine how networked storage protocols over raw Ethernet can achieve low, host CPU overhead and increase network link efficiency for small I/O requests. We first analyze in detail the latency and overhead of a networked storage protocol directly over Ethernet and we point out the main inefficiencies. Then, we examine how storage protocols can take advantage of context switch elimination and adaptive batching to reduce CPU and network overhead.

Our results show that raw Ethernet is appropriate for supporting fast storage systems. For 4kB requests we reduce server CPU overhead by up to 45%, we improve link utilization by up to 56%, achieving more than 88% of the theoretical link throughput. Effectively, our techniques serve 56% more I/O operations over a 10Gbits/s link than a baseline protocol that does not include our optimizations at the same CPU utilization. Overall, to the best of our knowledge, this is the first work to present a system that is able to achieve 14μs host CPU overhead on both initiator and target for small networked I/Os over raw Ethernet without hardware support. In addition, our approach is able to achieve 287K 4kB IOPS out of the 315K IOPS that are theoretically possible over a 1.2GBytes/s link.

[1]Also with the Department of Computer Engineering, University of Murcia, Spain
[2]Also with the Department of Computer Science, University of Crete, Greece

I. Introduction

Traditionally, small and typically random I/O requests have been bound by the latency of storage devices, especially magnetic hard disk drives (HDDs). Therefore, techniques for improving throughput have mostly focused on large requests, for which HDDs can achieve high throughput. Given that HDDs dominate all I/O overheads, the server I/O path has not been optimized for small I/Os.

However, small I/O requests are important for a large number of modern workloads in data centers. Although large files account for a large fraction of space, most files are 4kB or smaller [1]. In addition, metadata requests are typically small in size (4kB and 8kB), they follow a random access pattern for both reads and writes, and they account for about 50% of the I/O operations performed [2]. In some occasions, metadata occupy similar space as regular files in modern file systems and account for most I/O traffic.

Today, solid state drives (SSDs) and emerging persistent memory technologies exhibit performance characteristics similar to DRAM [3]. These technologies shift the bottleneck from the devices to the rest of the I/O path. For the first time, it now becomes possible to achieve high I/O operation rates with few storage devices, commodity servers, and commodity interconnects.

Applications today predominately use some form of networked storage, therefore, the network path is an essential component of the I/O path. Although there has been extensive work in low-latency and high-throughput networks, such as Infiniband, Ethernet has dominated the datacenter, and it is also presently dominant in networked storage compared to other networking technologies.

In our previous work, Tyche [4], we examine the design of network storage protocols over raw Ethernet to achieve high throughput without hardware support. Tyche employs numerous techniques to improve throughput and reduce overheads: copy-reduction, storage-specific packet processing, preallocation of memory, NUMA (Non-Uniform Memory Access) affinity management, and RDMA (remote direct memory access)-like operations. However, Tyche still exhibits high host CPU overhead and low network link utilization for small I/Os. This behavior is shown in Figure 1 that depicts the theoretical

978-1-4673-7619-8/15/$31.00 © 2015 IEEE

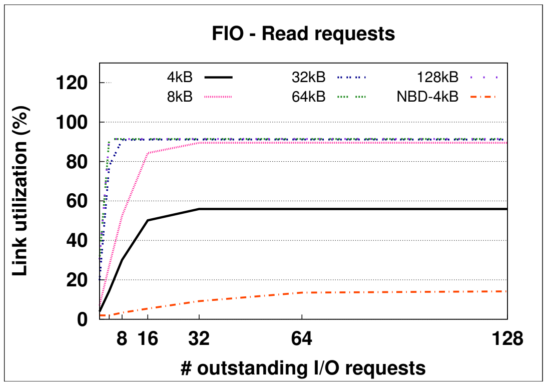


Fig. 1. Link utilization achieved by Tyche and NBD depending on the number of outstanding I/O requests, with FIO, direct I/O and random reads.

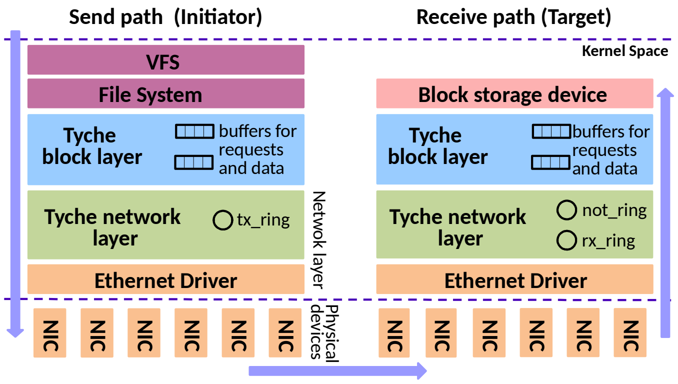


Fig. 2. Overview of the send and receive path from the initiator (client) to the target (server).

link utilization achieved by Tyche when the micro-benchmark FIO (Flexible I/O) [5] is run with direct I/O, random reads, and several request sizes. Section III describes the testbed we use in more detail. For a baseline comparison, Figure 1 also includes the link utilization achieved by NBD (Network Block Device). Tyche provides up to 5x the link utilization of NBD. However, Tyche is able to achieve only up to 56% of link utilization for 4kB requests, whereas, for 8kB requests link efficiency is 90%.

In general, in today's systems, small I/Os incur high server CPU utilization and low network link utilization, therefore reducing the effective rate of I/O operations per second (IOPS). As network link speed increases and storage device latency drops, the host I/O stack emerges as the main bottleneck in the I/O path. The overheads introduced by the host I/O stack for small I/Os can eliminate any potential benefits from improved network speed and storage device latency.

In this paper, we analyze in detail the host CPU overheads in the networked I/O path for small requests. We find out that there are two fundamental limitations to achieve high IOPS for small requests: Context switches impose significant CPU overhead whereas network packet processing dominates over link throughput.

Context switches cost on average $4\mu s$. Considering only the context switches along the common path of our baseline protocol, they represent 27.5% of the total CPU overhead for serving a 4kB request. Host overhead costs 65% of the total CPU overhead for serving a 4kB request. Considering only the processing done by Tyche, it costs 20% with no context switches and 47% by taking into account the context switches done during its path.

We design a protocol that reduces context switches for small I/Os. This design is particularly effective for low degrees of I/O concurrency and reduces host CPU overhead by 30.8% per 4kB-I/O request, about equally divided between the client (initiator) and server (target), and by up to 61% if we take into account only the Tyche send/receive path, excluding processing done by layer above Tyche.

We also incorporate a batching technique to achieve high link utilization under high degrees of I/O concurrency. Batching has been used extensively in both networking and storage systems. The novelty of our contribution is a dynamic technique that varies the degree of batching to match the load at a fine grain, without a negative impact on I/O overhead and response time. We dynamically determine the number of requests to include in a batch message by analyzing the throughput obtained. Results show that our batching technique can improve link utilization by up to 56.9% and 53% for reads and writes, respectively, when comparing with the link utilization obtained without batching.

Overall, to the best of our knowledge, this is the first work to present a system that is able to achieve $14\mu s$ host CPU overhead on both initiator and target for small networked I/Os over raw Ethernet without hardware support (including both initiator and target CPU cycles) and to reach 88% of the theoretical link utilization at high concurrency. In addition, Tyche is able to achieve 287,695 4kB IOPS over a single 10 GigE link, again with no hardware support.

This paper is organized as follows. Section II presents an overview of Tyche. Section III describes our experimental environment. Section IV analyzes in detail the overheads of Tyche. Section V and VI propose and evaluate our techniques to reduce overheads. Section VII discusses related work and Section VIII draws our conclusions.

II. Background Overview

Tyche [4] is a network storage protocol on top of raw Ethernet that achieves high I/O throughput without hardware support (Figure 2). Tyche presents the remote storage device locally by creating at the initiator a virtual local device, that can be used as a regular device. Tyche is independent of the storage device, and supports any existing file system.

Tyche uses RDMA-like operations, reliable delivery, framing, and transparent bundling of multiple NICs. It is an end-to-end protocol that does not require any hardware support. Tyche uses a copy reduction technique based on virtual memory page remapping to reduce processing cost for data packets. The 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. We would like to remark that Tyche does not use RDMA in Ethernet, instead our protocol uses a similar, memory-oriented abstraction that allows us to perform RDMA-like operations.

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 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, 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 issuing I/O requests to the local device. The initiator has no pre-allocated pages, it uses the pages of the user I/O requests.

The initiator send path can operate in two modes. In the "inline" mode (Figure 2), 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 to the target. There is no other difference between them. The inline mode outperforms the queue mode for reads and for writes of small size; the queue mode may outperform the inline mode when there are many outstanding writes of large size.

The target uses a work queue for sending completions back to the initiator, because local I/O completions run in an interrupt context that cannot block. So, the local I/O completion schedules a work queue task to send the required network responses.

In each node, the receive path uses a network thread per NIC to process packets and messages. A block layer thread (per connection) processes messages and issues local I/O requests (in the target) or completes requests (in the initiator).

Figure 3 depicts the end-to-end I/O path of a request message through the different layers, buffers, and rings. To simplify the figure, we only include a single message buffer. Arrows correspond to the flow of an I/O operation in the I/O path.

III. 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 2.7GHz. The operating system is the 64-bit version of CentOS 6.3 testing with Linux kernel version 2.6.32. Each node uses a Myricom 10G-PCIE-8A-C card that is capable of about 10Gbits/s throughput in each direction. The target node is equipped with 48GB DDR-III DRAM, and the initiator with 12GB. The target uses 12GB as RAM, and 36GB as ramdisk. The ramdisk provides throughput and latency of DRAM and the storage device is not anymore the slowest part of our network-storage system. The NIC device is configured for not delaying the delivery of interrupts. For the evaluation, we use the micro-benchmark FIO (Flexible I/O) [5] that allows us to generate several workloads.

IV. End-to-end overhead analysis

This section analyzes the cost of the I/O path for a request. Figure 3 marks with arrows and labels the parts of the I/O

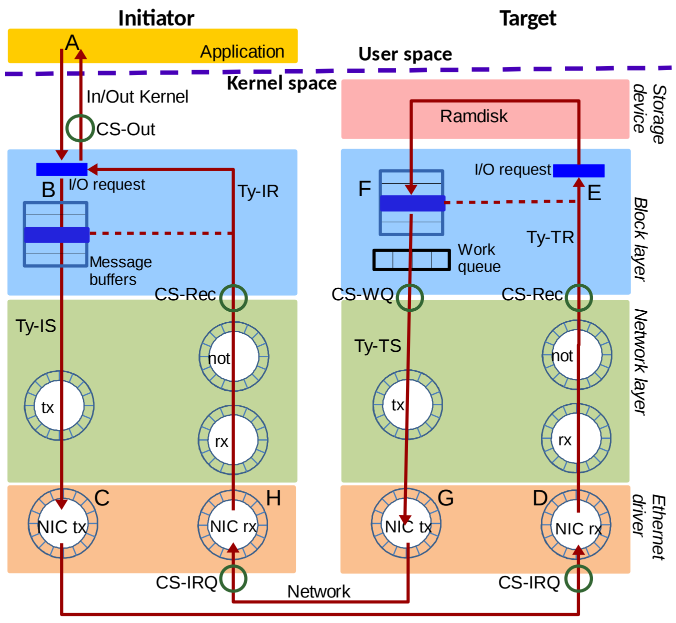


Fig. 3. End-to-end I/O path showing the components we measure, and the associated context switches that are marked with a green circle. Labels A, B, C, etc. just help us describe the computed values.

path that we measure. Table I summarizes the individual and cumulative overheads of Figure 3. To study the overhead of our protocol, we have modified the Tyche version implemented in Linux kernel 2.6.32 [4] by including the overhead computation.

Table II presents overheads, in μs, and throughput, in MB/s, obtained by Tyche, for reads and writes of 4kB, 8kB, 64kB, and 128kB. We use FIO with direct I/O, random requests, and 60s of runtime. The storage device is accessed in a raw manner (no file system). There is a single application thread issuing I/O requests, and a single outstanding request. We apply NUMA affinity by allocating memory and pinning all threads in the same NUMA node where the NIC is connected. In addition, as a baseline comparison between Tyche and NBD, Table III provides the total overhead, in μs, obtained by both protocols, for read and write requests, of 4kB, 8kB, 64kB and 128kB, when running the same experiment.

Table III shows that Tyche already reduces by more than 2x the total overhead achieved by NBD. However, the overhead introduced by Tyche for small requests is still large and the main bottleneck when using fast storage devices.

We note that message processing is the most important source of per-I/O request overhead, being up to 65% of the total overhead. Considering only Tyche overheads (Ty-IS, Ty-TR, Ty-TS, Ty-IR, CS-WQ, CS-Rec, and CS-IRQ), for a 4kB request, they represents 47%, taking into account the context switches done along the I/O path, and 20% without these context switches (without CS-WQ, CS-Rec, and CS-IRQ). Similar percentages are true for other request sizes.

The send and receive paths are more expensive when data is involved, since a request packet and several data packets are sent or received. Receiving data is more expensive for reads than for writes, since, reads copy data pages from the NIC driver to the I/O request, whereas writes interchange pages [4].

TABLE I. COMPONENTS MEASURED DURING THE END-TO-END I/O PATH OF TYCHE (SEE FIGURE 3 FOR A DETAILED GRAPH).

Name	Path	Description
Total	A - B - E - B - A	Overhead, reported by the application, of serving the request measured as the time delay between the time the application issues the request until the request is completed.
Tyche	B - E - B	Overhead measured by Tyche as the time between the arrival of the request to its block layer until its completion. Effectively, this is the overhead of our protocol excluding the above layers.
Ty-IS	B - C	Overhead of the Tyche (Ty): send path at the initiator (IS) and target (TS) and receive path at the initiator (IR) and target (TR).
Ty-TR	D - E	
Ty-TS	F - G	
Ty-IR	H - B	
CS-WQ		Cost of the context switch due to work queues.
CS-Rec		Cost of the context switch between the network layer and block layer threads (includes context switches of both sides).
CS-IRQ		Cost of the context switches done when a NIC's IRQ is raised. Measured as the time spent since the IRQ handler function executes the wake up function until the network thread starts its execution.
Ramdisk	E - F	Overhead of the ramdisk from submitting a bio until receiving its completion. Ramdisk is synchronous so, IO happens inline with no context switches.
In/Out kernel	A - B and B - A	Time needed by a request to arrive from the application to Tyche and time needed to complete the request from Tyche. This overhead is calculated (not measured) as the difference between total and Tyche overheads.
Network	C - D and G - H	Overhead of the network interface and the network link(s). This overhead is calculated (not measured) as the difference between Tyche overhead and the sum of Ty-IS, Ty-TR, Ty-TS, Ty-IR, CS-WQ, CS-Rec, CS-IRQ, and Ramdisk. It includes the overhead of the corresponding driver at the host, which however, is low compared to the rest of the host overheads.

TABLE II. OVERHEAD, IN μs, AND THROUGHPUT, IN MB/S, CALCULATED FOR TYCHE, BY RUNNING FIO WITH A SINGLE THREAD AND A SINGLE OUTSTANDING I/O, DIRECT I/O, RANDOM READS AND WRITES, AND SEVERAL REQUEST SIZES.

Overhead (μs)		**Read requests**				**Write requests**			
		4 kB	**8 kB**	**64 kB**	**128 kB**	**4 kB**	**8 kB**	**64 kB**	**128 kB**
Software	**In/Out kernel**	13.19	13.13	16.25	15.33	12.80	14.66	26.38	40.96
	Ty-IS	2.75	2.00	3.00	2.00	4.75	5.00	15.00	26.25
	Ty-TR	3.00	3.00	4.00	4.25	5.00	6.00	15.00	24.25
	Ty-TS	4.00	5.00	12.00	22.00	3.00	4.00	4.00	3.00
	Ty-IR	5.00	6.00	24.00	45.00	2.25	2.00	2.00	2.00
	CS-WQ	4.00	4.00	4.00	4.00	4.00	3.00	5.00	3.00
	CS-Rec	8.00	8.00	8.00	7.00	8.00	8.00	8.00	7.00
	CS-IRQ	8.15	7.02	20.30	30.54	8.13	8.02	23.82	37.90
Hardware	**Ramdisk**	1.00	2.00	16.00	30.75	1.00	2.00	17.00	31.00
	Network	24.60	32.73	45.20	60.21	24.87	30.98	47.18	63.35
Total		73.69	82.88	152.75	221.08	73.80	83.66	163.38	238.71
Throughput (MB/s)		52.50	94.00	408.50	565.00	52.50	92.75	382.00	523.25

The In/Out kernel overhead is high, and depends not only on the request size but also on its type. For reads, this overhead has a larger impact in small requests, being a 17.9% and 15.8% of the total overhead for 4kB and 8kB requests, respectively. For larger requests, the overhead is less important, being 10.6% and 6.9% for 64kB and 128kB requests, respectively. For writes, its impact in the total overhead does not depend on the request size, and represents, on average, 17.0% of the total overhead.

A significant component of the In/Out kernel overhead is the overhead due to the context switch done when the application thread is waken up to complete the request (marked as CS-Out in Figure 3 and but not included in Table I). We further examine this issue by creating two "fake" block devices: BD-NCS (Block Device No Context Switch) and BD-WCS (Block Device With Context Switch). The block layer of BD-NCS, when it receives a request, just completes the request, without any further processing. The block layer of BD-WCS forces a context switch before completing the request, similar to a regular request. The overhead of BD-WCS corresponds to the In/Out kernel overhead. For both devices, Table IV provides the total overhead, in μs, when running the same test as previously.

BD-NCS has significantly smaller overhead than BD-WCS: In BD-NCS the overhead of crossing the user-kernel boundary from application to Tyche is $4\mu s$ for 4kB and 8kB requests, $7\mu s$ for 64kB, and $10\mu s$ for 128kB. For BD-WCS, this overhead is closer to In/Out kernel in Table II: About $10.70\mu s$ for 4kB reads (vs. $13.19\mu s$ for In/Out kernel). For large requests, however, BD-WCS exhibits significantly lower overhead compared to In/Out kernel: For 64kB and 128kB writes, BD-WCS incurs an overhead of $13.80\mu s$ and $18.00\mu s$, respectively, whereas, In/Out kernel overhead is $26.38\mu s$ and $40.96\mu s$, respectively. The reason of this difference is that In/Out kernel incurs more cache misses, since it runs the full Tyche I/O path, whereas BD-WCS, as a fake configuration,

TABLE III. TOTAL OVERHEAD, IN μs, CALCULATED FOR TYCHE AND NBD, WITH FIO, A SINGLE THREAD AND ONE OUTSTANDING I/O, DIRECT I/O, RANDOM READS AND WRITES.

		4 kB	8 kB	64 kB	128 kB
Reads	**Tyche**	73.69	82.88	152.75	221.08
	NBD	152.35	162.42	433.84	561.48
Writes	**Tyche**	73.80	83.66	163.38	238.71
	NBD	151.79	153.78	332.23	520.02

TABLE IV. TOTAL OVERHEAD, IN μs, FOR FIO WITH THE TWO FAKE BLOCK DEVICES THAT JUST COMPLETES THE REQUEST.

		4 kB	8 kB	64 kB	128 kB
BD-NCS	**Reads**	3.60	3.80	6.40	9.00
	Writes	3.40	3.60	6.60	9.70
BD-WCS	**Reads**	10.70	11.00	13.40	16.10
	Writes	10.50	10.70	13.80	18.00

runs only a small part of the path. Therefore, In/Out kernel overhead is mainly due to blocking and waking up of the application thread.

A. *Context switch overhead*

Overall, each context switch costs around $4\mu s$. For 4kB and 8kB requests, at Tyche level, there are five context switches that represent 27.5% and 23%, respectively, of the total overhead. For larger requests, there are more context switches since a large number of data packets are sent/received, and more NIC IRQs are raised. For 64 kB and 128 kB requests, they represent up to 22.5% and 20.0% of the total overhead, respectively.

We confirm the cost of a context switch in our testbed system with a simple test. We launch two kernel threads, A and B. Each one has its own wait queue, and they communicate through a control variable. A sets the control variable, it wakes up B, that is sleeping in its wait queue, and then A blocks. When B is waken up, it resets the variable, wakes up A, and it blocks. We measure the cost of a context switch as the time between A waking up B until B is executed again (and the other way around).

When both threads run in the same NUMA node, the context switch costs around $2.5\mu s$, but when they run in different NUMA nodes, it costs around $5\mu s$. Similar numbers were obtained by Li *et al.* [6], whose experiments in the same NUMA node showed $3\mu s$ per context switch. Note that for our protocol the context switches are more expensive compared to this simple test, because the cost of a context switch also depends on the program data size, system caches, memory allocation, and other factors [6], [7].

This test shows that NUMA affinity matters as well. The cost of a context switch is smaller when the threads are running in the same NUMA node. In addition, we have made a test in which the application and network threads are pinned in a node that does not have the NIC attached, and the block layer thread is pinned in the same NUMA node of the NIC. For a 4kB read, the overhead of the whole send/receive path is increased by up to 75%, and the context switch cost increases by up to 45% to $5.8\mu s$.

These results show that context switches are expensive compared to other overheads in the I/O path. Next, we discuss our techniques for reducing the number of context switches

V. REDUCING CONTEXT SWITCHES

Serving a request involves, at least, six context switches:

- Two on the receive path, one at the initiator and one at the target, marked as CS-Rec in Figure 3. The receive path has two threads: one thread that runs the tasks of the network layer, and another one that runs the tasks of the block layer. At least a context switch occurs between these threads. The roundtrip of an I/O request involves two request messages (request and completion) and a data message. For writes, the initiator sends the request and data messages and the target sends back the completion message. For reads, the initiator sends the request message and the target sends the completion and data messages. Since a request message that has associated a data message cannot be processed without its data, Tyche always sends the data message before its request message. Therefore, in a controlled experiment as the one run here, a data message will always arrive before its corresponding request message[1]. Thus, when the request message arrives, there is a context switch between the network thread and the block layer thread. The block layer thread fetches first the request message and then the data message without doing an additional context switch.
- One context switch at the target send path, marked as CS-WQ in Figure 3. This one is done because a work queue sends the completion back to the initiator (see Section II).
- Two context switches, one on each side, due to the NIC interrupts, marked as CS-IRQ in Figure 3. When a NIC's IRQ is raised, its handler function just wakes up our network thread to receive the packets. Once this thread is running, it continuously polls the NIC for new packets. The thread will block, waiting for a new IRQ, when the NIC receive ring is empty. For small requests, two context switches are done. For large requests, more context switches are expected because the number of data packets sent/received depends on the request size. It is worth emphasizing that Tyche cannot avoid these context switches. The reason is that interrupt handler functions cannot do anything that would sleep, such as waiting an event or getting a lock. Previous works [8], [7] have examined how to avoid this context switch by using polling instead of interrupts. They propose to have threads spinning and waiting for the arrival of packets. However, we have not considered this option because the spinning time could be significant large, specially for large requests.
- One context switch, CS-Out in Figure 3, is done to complete the request. After issuing a request, the application thread is blocked waiting for its completion.

[1]Since Tyche allows out-of-order transfers, a request message may arrive before its corresponding data message.

On the reception of a completion message, Tyche runs a function to complete the request that ends up waking the application thread and doing a context switch. To avoid this context switch, layers above Tyche should be modified, but, this is beyond the scope of this paper, since this context switch is out of the control of Tyche. One option could be that the application thread waits for the completion spinning [9], [8]. But, when there are a large number of threads concurrently issuing I/O requests, the spinning option is not viable.

Tyche works in the inline mode (see Section II) that outperforms the queue mode for small requests. Therefore, at the initiator send path requests are issued with no context switch.

Finally, we use a ramdisk to emulate fast storage devices so we omit the context switch on the target side. This context switch is related to the interrupt handler that completes the I/O to the local device and then performs a context switch to the corresponding Tyche thread.

We now discuss two variants of the base protocol that reduce the number of context switches. The first one avoids context switches on the receive path. The second one, in addition, avoids the context switch due to the work queue.

A. Tyche-Rec

To avoid the context switch between the network and block threads (CS-Rec in Figure 3), we propose using a single thread to run the whole receive path (Tyche-Rec). Currently, the network thread runs its own tasks, and the tasks of the block layer as well. The network thread processes a packet, composes the message, and checks whether any data message related to the one just composed has arrived. When all the messages (request message and data message if any) have been received, the network thread runs the block layer tasks by using a callback function. By using callback functions the network thread also checks whether all the messages that compose an I/O request have been received. This way, the thread will not block when a message is missing because it has been not received it yet.

The notification rings (marked as not in Figure 3) that communicate both threads are not used. Consequently, we also reduce overhead by avoiding, these rings and, for instance, the lock to ensure exclusive access.

B. Tyche-NoWQ

Next, we eliminate, in addition to Tyche-Rec, the context switch in the target send path due to the work queue (CS-WQ in Figure 3). This version, Tyche-NoWQ, attempts to send the response to the initiator from the completion context of the local I/O. If it succeeds, there will be no context switch. But, if the operation needs to block, which is not allowed in the completion context, it will fall back to the work queue of the base version. Note that, the completion context will block if, for instance, there is no space in the transmission ring. Avoiding the work queue results in avoiding the management associated. For instance, a lock is required to insert/dequeue a task into/from the work queue.

C. Evaluation of Tyche-Rec and Tyche-NoWQ

To study the context-switch costs we implement Tyche-Rec and Tyche-NoWQ based on the Tyche version that computes the overhead. Tables V and VI provide the overhead breakdown, in μs, and throughput, in MB/s, for reads and writes. We use again FIO with the same configuration as in Section IV.

Total overhead is reduced by up to 27.6% and 17.6% for 4kB and 8kB reads respectively, and throughput is improved by up to 39.1% and 21.3%, respectively. For 64kB and 128kB reads, overhead is reduced by up to 12.7% and 8.1%, respectively, and throughput is improved by up to 14.5% and 8.8% respectively. For writes, this reduction is 30.8% and 21.3% for 4kB and 8kB requests, respectively, and throughput is improved by up to 44.8% and 26.4%, respectively. The overhead reduction is 12.3% and 5.2%, respectively. Throughput is improved by up to 13.9% and 5.5%, for 64kB and 128kB writes, respectively.

By analyzing values in detail, and taking into account only Tyche send/receive path, we reduce overhead by up to 56% and 61.1% for reads and writes, respectively, for 4kB requests.

Results for Tyche-Rec show that CS-Rec is reduced to zero, since no context switch is done on the receive path. These results also show that Ty-TR and Ty-IR are significantly reduced. There are two reasons: i) the notification rings are not used because a single thread runs the whole receive path; ii) the locks to protect these rings are not required.

For Tyche-NoWQ, CS-WQ is reduced to zero, since the context switch due to the work queues is not performed. Ty-TS is also reduced, because the management of the work queue is avoided, for instance we avoid the lock to add a job to the work queue.

These results show, as expected, that avoiding locks also reduces overhead. However, only versions such as Tyche-Rec and Tyche-NoWQ, where a single thread executes exclusively one path, allow us to eliminate locks. When several threads can send/receive simultaneously packets, locks are needed to ensure exclusive access to the shared data structures.

The In/Out kernel overhead is also slightly reduced. In this case, there is no difference in the way the test is run. We believe that the reduction is due to the system caches as Li *et al.* point out [6], since there are fewer threads running and fewer context switches.

VI. Adaptive Batching

At high concurrency when there is a large number of outstanding requests, the previous designs cannot achieve high link utilization and consequently high IOPS. This section discusses how adaptive batching can improve significantly link utilization for small requests. The novelty of our proposal is a dynamic technique that varies the degree of batching without increasing I/O overhead and response time.

We introduce a new request message, called batch request message or batch message. A batch message is a single network message that includes several I/O requests, reads or writes, issued by the same or different threads. When a batch message is received, the target issues to the local device a regular I/O request per request included in the batch message.

TABLE V. OVERHEAD, IN μs, AND THROUGHPUT, IN MB/S, CALCULATED FOR TYCHE-REC, BY RUNNING FIO WITH A SINGLE THREAD AND A SINGLE OUTSTANDING I/O, DIRECT I/O, RANDOM READS AND WRITES, AND SEVERAL REQUEST SIZES.

Overhead μs)		Read requests				Write requests			
		4 kB	8 kB	64 kB	128 kB	4 kB	8 kB	64 kB	128 kB
Software	In/Out kernel	12.63	13.72	17.13	14.98	12.80	13.70	25.76	42.55
	Ty-IS	2.00	2.00	2.25	2.00	4.00	4.00	12.25	23.50
	Ty-TR	1.00	1.00	2.00	3.75	2.00	2.00	12.75	23.00
	Ty-TS	4.00	5.00	12.75	23.00	3.00	4.00	4.00	3.00
	Ty-IR	2.00	2.00	19.50	41.50	0.00	0.00	0.00	0.25
	CS-WQ	4.00	4.00	3.25	3.00	3.50	4.00	4.00	3.00
	CS-Rec	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	CS-IRQ	8.12	8.28	21.03	31.11	8.02	8.04	23.74	38.35
Hardware	Ramdisk	1.00	2.00	16.00	30.25	1.00	2.00	16.25	30.00
	Network	26.88	35.72	49.47	61.39	25.98	36.21	53.01	68.15
Total		61.63	73.72	143.38	210.98	60.30	73.95	151.76	231.80
Throughput (MB/s)		63.00	105.25	435.75	592.00	64.00	105.00	411.25	538.75

TABLE VI. OVERHEAD, IN μs, AND THROUGHPUT, IN MB/S, CALCULATED FOR TYCHE-NOWQ, BY RUNNING FIO WITH A SINGLE THREAD AND A SINGLE OUTSTANDING I/O, DIRECT I/O, RANDOM READS AND WRITES, AND SEVERAL REQUEST SIZES.

Overhead (μs)		Read requests				Write requests			
		4 kB	8 kB	64 kB	128 kB	4 kB	8 kB	64 kB	128 kB
Software	In/Out kernel	11.38	13.83	14.43	14.77	12.11	14.27	25.61	42.36
	Ty-IS	2.00	3.00	2.00	2.00	3.00	4.00	11.00	23.75
	Ty-TR	1.00	1.00	1.00	2.50	2.00	2.00	12.00	20.25
	Ty-TS	3.00	3.00	11.00	20.00	1.00	1.00	2.00	2.00
	Ty-IR	1.00	2.00	20.00	40.00	0.00	0.00	0.00	0.25
	CS-WQ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	CS-Rec	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	CS-IRQ	8.09	8.16	20.4	30.71	8.01	9.10	23.77	38.92
Hardware	Ramdisk	1.00	2.00	16.00	30.50	1.00	2.00	16.00	31.00
	Network	25.91	35.34	48.6	62.79	23.99	32.90	52.98	67.83
Total		53.38	68.33	133.43	203.27	51.11	66.27	143.36	226.36
Throughput (MB/s)		73.00	114.00	467.75	614.75	76.00	117.25	435.25	552.00

Completions are sent as batch messages as well. The target sends the completion message when all the requests associated with the batch message are completed.

Batch messages significantly reduce the number of messages and the associated message processing. For instance, a batch message with 32 requests sends two messages (one in each direction) instead of 64 concurrent messages (32 in each direction). Equally important, batching reduces the number of context switches required.

There are two general remarks about batching. First, batching makes sense up to some request size. Large I/O requests already make efficient use of the network, and batching offers no benefit. Given the low overhead of Tyche, this size is 8kB, so in our case we only batch 4kB requests. Other systems, with higher overheads, will benefit from batching larger requests as well. Second, static batching does not work in practice, as our results show. Thus, we employ an adaptive technique that constantly adjusts the number of requests batched. When our technique chooses a batch degree of one, it incurs no additional overhead compared to the optimized inline mode of the previous section.

A. Batch mechanism

Our batching mechanism is built around a batch queue introduced in the send path of the inline mode. Figure 4 depicts the initiator with the batch mechanism (to simplify the figure, we have omitted the message buffers). Now, at the initiator, each I/O request is inserted into the batch queue. Then, a single (batch) thread dequeues requests and includes them in a batch request message. If there is no batch message pending, a new one will be created. Although we could use directly the batch message to place all requests, it is preferable to use a separate queue. The reason is that managing the batch message results in a larger critical section than enqueuing a request to the batch queue. So, it is better to have the many user contexts insert requests to the batch queue and a single thread dequeues and prepares batch messages.

A key aspect of our batching approach is to decide when to wait for new requests or when to send the batch message immediately. We use a parameter, the current batch level (`current_batch_level`), to determine the number of requests to include in a batch message. We send a batch message when it has `current_batch_level` requests. We dynamically calculate `current_batch_level` based on the link throughput achieved. If by increasing or decreasing the

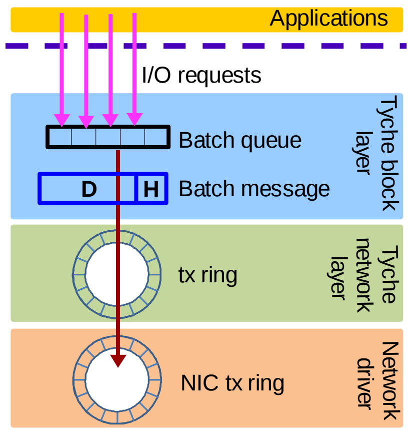


Fig. 4. Initiator send path with batching. To simplify, we have omitted the message buffers.

batch level compared to the current value results in increased throughput, we will keep moving in the same direction.

The value of current_batch_level varies between 1 and max_batch_level. Where max_batch_level corresponds to the maximum number of requests that a batch message can carry.

To compute the value of current_batch_level, our batch mechanism calculates two values:

- The throughput achieved (Xput and Xput_p) in the last and previous intervals by using the number of I/O requests completed and their size.
- The average number of outstanding I/O requests (a_out_r) in the batch queue during the last interval.

We then calculate the improvement in throughput of the last interval over the previous interval, and we set current_batch_level as:

- If the improvement is larger than 3% the new value of current_batch_level will be increased to:

  $$\frac{\texttt{current\_batch\_level} + min(\texttt{a\_out\_r}, \texttt{max\_batch\_level})}{2}.$$

- If the improvement is smaller than -3%, the value of current_batch_level will be reduced to:

  $$\frac{1 + min(\texttt{a\_out\_r}, \texttt{current\_batch\_level})}{2}.$$

- Otherwise, no change is made to the batch level.

To avoid delaying batch messages too long, we use a maximum amount of time (max_delay) that the first request of a batch message may be delayed. A batch message will be sent if current_batch_level is reached or max_delay expires.

Finally, we avoid the case where the batch level remains unmodified because throughput is stable, although there is potential for better link utilization. For this reason, if after 10 consecutive intervals there are no changes, we compare the throughput of current_batch_level to the throughput of current_batch_level $- 1$ and current_batch_level$+1$. If for one of these new values, there is an improvement of at least 3%, we start adjusting current_batch_level again.

B. *Batching data messages*

We also use a simple batch mechanism for data messages. The data of a 4kB request is sent by using a single data packet in a Jumbo Ethernet frame of 4kB. By batching data messages, the data messages of two 4kB requests are transmitted by using a single data packet of 8kB and a single data batch message. The method to batch data messages may be applied to larger requests as well, but the main benefit occurs when 4kB requests are batched and sent in 8kB packets.

C. *Overhead of dynamic batching*

The batch mechanism increases overhead. For instance, a lock is required to ensure exclusive access to the batch queue, and new context switches are done by the batch thread to handle the batch messages. However, tasks can be overlapped: the thread adds a request to a batch message, while an application thread inserts its requests into a batch queue. Batching may increase processing at the layer block, but it reduces the number of messages, so it reduces processing at the network layer of our protocol. As our results show, this overhead increase is offset by a significant improvement in throughput.

D. *Evaluation of adaptive batching*

To study the effects of batching, we have implemented a version called Tyche-Batch that batches requests and applies the dynamic algorithm to choose the batch level. This version is based on the Tyche version implemented in Linux kernel 2.6.32 [4]. In addition, we have implemented a static version with a fixed batch level during the whole execution. Next, we examine to questions: i) how our dynamic algorithm chooses the right batching level depending on the throughput achieved; ii) how dynamic batching compares to static batching.

We run FIO with direct I/O, random reads and writes, 4kB request size, and 1, 2, 4, 8, 16, 32, 64, and 128 threads issuing requests, 4 outstanding requests per thread, and a runtime of 60s. The remote storage device is accessed in a raw manner (there is no file system). We have also tested that our batch technique provides no benefit for other sizes, such as 8kB, 16kB, 32kB, and 64kB, but it does not hurt. For Tyche threads we apply NUMA affinity, by pinning them in the same NUMA node where the NIC is connected. The application threads are not pinned, since they are too may to pin all of them in the same node.

The dynamic version is configured with 1s as check interval, 64 requests as max_level_batch, and 5ms as max_delay. We run the static version with 2, 4, 8, 16, 32, and 64 requests as max_batch_level. However, we only present results for 2, 8, and 64 requests, since all of them have similar behavior.

Figure 5 depicts the theoretical link utilization, depending on the number of outstanding requests, achieved by the dynamic Tyche-Batch version (DyB in the figure), and the static version with 2, 8, and 64 requests as batch level (B-2, B-8, and B-64), and Tyche with no batching (NoB).

When there is high concurrency, dynamic Tyche-Batch outperforms the no batching version by up to 44.2% and 49.5% for reads and writes, respectively. At 32 outstanding requests, it improves over the no batch version by 28.1% and 35.8% for reads and writes, respectively. By batching requests, Tyche is able to achieve up to 80.7% and 79.2% of link utilization for reads and writes, respectively.

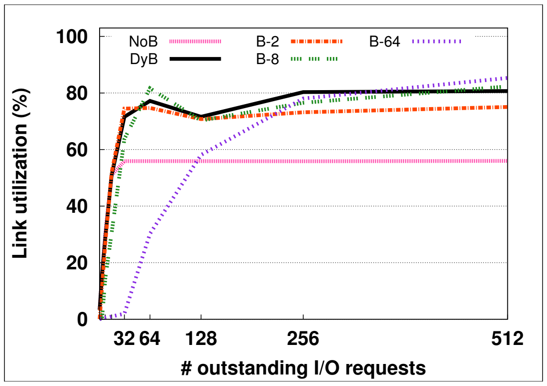
Comparing the dynamic version to the static, we see that the dynamic version achieves the best performance and follows the static version providing the best behavior. Sometimes, a static version outperforms the dynamic version, but the larger difference between them is only up to 8.6%. In these cases, the algorithm took a conservative decision when by batching a large number of requests, it could achieve a higher throughput.

We also note that the static version achieves poor performance and link utilization at low concurrency. The reason is that batch messages are sent to the target because `max_delay` expires and not because the batch level is reached.

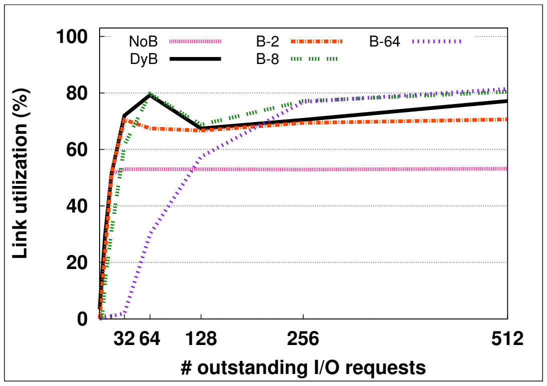
Figure 6 depicts the batching level used during the execution of the test depending on the number of outstanding requests. For reads, the dynamic version increases the batch level as the number of outstanding requests increases. For writes, the batch level is kept constant, and when there are more than 256 outstanding requests, the algorithm starts to increase the number of requests included in a batch message. Although the batch level is small, the dynamic version achieves better performance than without batching. With low concurrency, the static version obtains a small batch level, since `max_delay` forces to send the batch messages without reaching the fix batch level.

Figure 7 depicts the theoretical link utilization, depending on the number of outstanding requests, when we also batch data messages, achieved by the same configurations and running the same test. Now, dynamic Tyche-Batch outperforms the no batch version up to 56.9% and 53% for reads and writes, respectively. For reads, our proposal achieves up to 88.0% of link utilization, and for writes up to 81.1%. The dynamic version is close to the static version providing the best performance. Although the graph for the batch level is omitted due to space constraints, the algorithm chooses similar batch levels when data messages are also batched.

To show that the batching technique works well with different simultaneous request sizes, we run FIO with direct I/O, random reads and writes, and two different request sizes. During 360s, there are 32 threads issuing 4kB requests and 4 outstanding requests per thread. Then, every 60s, we launch a new thread that issues 128kB requests with a single outstanding request. So, during the test, the first 60s, 32 threads issue 4kB requests, the second 60s, 32 threads issue 4kB requests and one thread issues 128kB requests, the third 60s, 32 threads issue 4kB requests and two threads issue 128kB requests, and so on. Figure 8 depicts the theoretical link utilization, depending on time, achieved by the dynamic version when data messages are also batched and by Tyche with no batching. The batch mechanism significantly outperforms the version without batching, by improving the link utilization by up to 3× and 2× for reads and writes, respectively. For reads, the batching approach achieves up to 91.3% of link utilization, and on average 88.8%, whereas no batching achieves up to 74.4%,



(a) Read requests



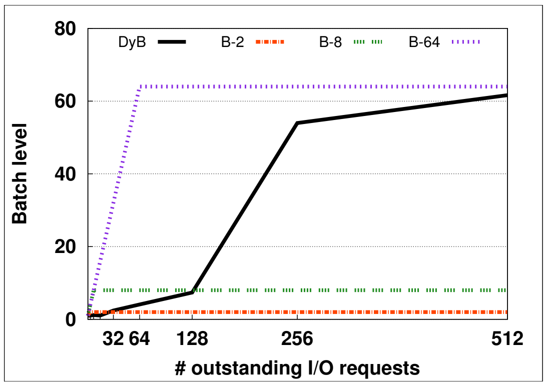
(b) Write requests

Fig. 5. Link utilization achieved by the dynamic and static Tyche-Batch versions and Tyche with no batching, with FIO with direct I/O and random reads and writes of 4kB.

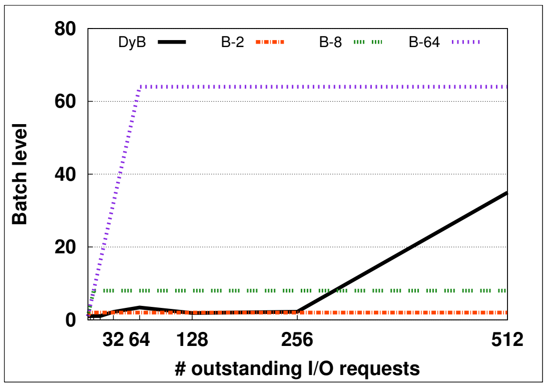
but on average only 56.7%. Similar numbers are obtained also for write requests. Note that there are few points in which the algorithm is not able to get the right decision and its performance momentarily drops to the performance of the version without batching.

VII. Related Work

Lately, there have been renewed interest in latency and overhead for storage access due to SSDs and emerging storage devices. Rumble *et al.* [10] analyze the latency problem of network protocols, and they claim that operating systems should implement a new networking architecture and new protocols to solve the latency problem end-to-end. Flajslik and Rosenblum [11] provide a network interface solution for low latency request-response protocols. They reduce latency by minimizing the number of transitions over the PCIe interconnect, particularly for small requests. HERD [12] is a key-value cache that leverages RDMA features to deliver low latency and high throughput. Pilaf [13] is a distributed in-memory key-value store that leverages RDMA to achieve high throughput with low CPU overhead. They only use RDMA to perform reads from server's memory, whereas writes are handled via traditional messaging. FaRM [14] is a main memory distributed computing platform that uses RDMA to improve latency and throughput. It uses RDMA writes to implement a fast message passing primitive. HERD, Pilaf, and FaRM rely in RDMA hardware to achieve low latency, whereas Tyche provides low latency without any hardware support.

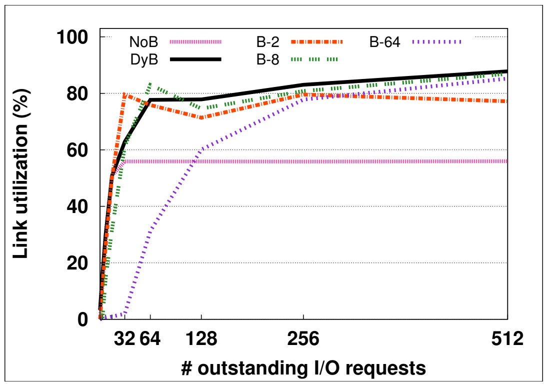


(a) Read requests

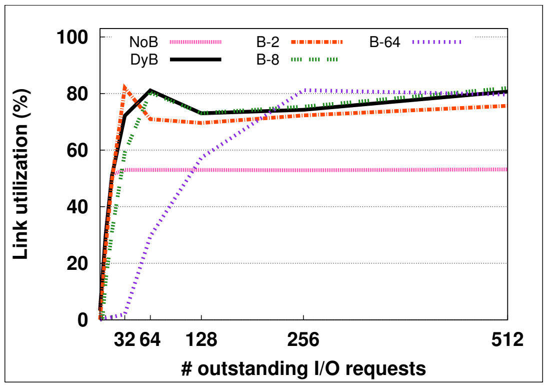


(b) Write requests

Fig. 6. Batch level used by the dynamic and static Tyche-Batch versions, with FIO with direct I/O and random reads and writes of 4kB.



(a) Read requests



(b) Write requests

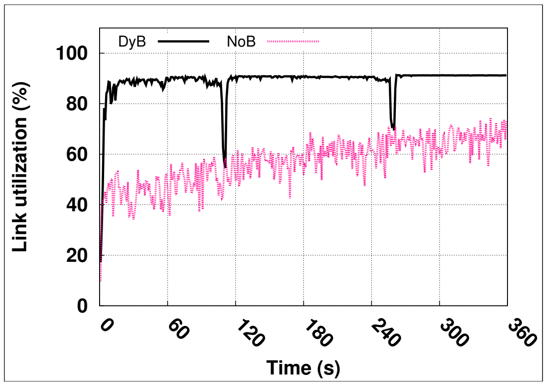
Fig. 7. Link utilization achieved by the dynamic and static Tyche-Batch versions and Tyche with no batching, when data messages are also batched, with FIO with direct I/O and random reads and writes of 4kB.

Regarding the reduction of latencies inside the data center, there are several works, such as jVerbs [15] and Chronos [16]. jVerbs offers low latencies to applications running inside a Java Virtual Machine by mapping the network device directly into JVM. Chronos reduces latency by removing the kernel and network stack from the critical path of communication.

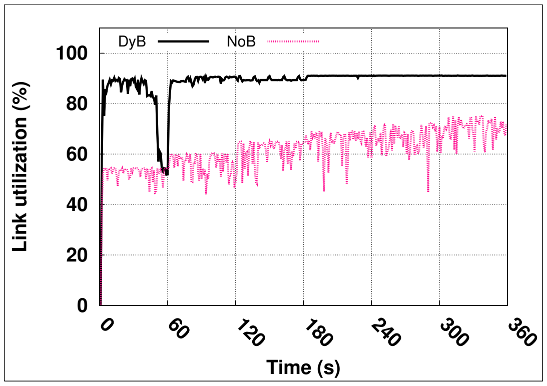
Recently, several works have proposed to bypass the kernel and to run in user-space I/O stacks to reduce latency by eliminating kernel crossing overheads. Moneta Direct [17], [9] bypasses the operating and file systems while preserves their management and protection functions to reduce latency. For each process, it provides a private, virtual interface to the device. Aerie [18] is a file system architecture that allows user-mode programs to access files without kernel interaction. The kernel provides only coarse-grained allocation and protection, while most functionality is distributed to client processes. Arrakis [19] is an operating system that removes the kernel from the I/O path by separating the control and data plane. In the data plane, I/O devices are accessed without kernel mediation by delivering I/O directly from the device hardware to a user-level library. The control plane uses the kernel to manage naming and coarse-grain allocation. Bypassing the kernel complicates resource sharing and creates security tradeoffs since application bugs can corrupt the I/O stack. IX [8] is an operating system that also reduces latency by separating the control and data plane. Its control plane is responsible for coarse-grain resource provisioning between applications, whereas its data plane runs the networking stack and application logic. IX does not bypass the kernel, instead, it leverages hardware virtualization to run the data plane kernel. Our proposal is to reduce latency without modifying the operating system, by redesigning the network I/O path.

Li *et al.* [6] show that the cost of a context switch depends not only on the regular context switch process of saving/restoring registers, execution of the scheduler, flushing processor pipeline, etc., but also on the program data size and even on the access pattern of the task. Gim *et al.* [7] show that the overhead of a context switch becomes relatively more significant for faster storage devices and that the overhead of a context switch mostly comes from the pollution of the data cache. They propose an intelligent context switch mechanism called SmartCon. To serve a given I/O requests, SmartCon decides whether to switch context over another thread (interrupt driven manner), or to simply stall (busy-wait based manner). The decision is based on the device characteristics, I/O latency, request size, and CPU utilization. Our approach, however, is to eliminate context switches for 4kB requests by using a single thread that runs the whole path and not by having threads spinning.

Currently, many network storage protocols are using batch messages or batch operations. For instance, NFSv4 reduces latency for multiple operations by bundling together different RPC calls [20], a lookup, open, read, and close can be sent once, and the server can execute the entire compound call as a single entity.



(a) Read requests



(b) Write requests

Fig. 8. Link utilization achieved by dynamic Tyche-Batch when data messages are also batched and Tyche with no batching, with FIO with direct I/O and random requests of 4kB and 128kB size.

Automatic TCP corking [21] is another example. It is included in Linux kernel 3.14 to improve performance for help applications doing small write/sendmsg operations on TCP socket. Although it is not a proper batch operation, it allows to delay the dispatch of messages in a socket in order to coalesce more bytes in the same packet, reducing the number of packets to send. Previous versions of Linux allow the use TCP corking, however, Linux kernel 3.14 is the first one to include automatic TCP corking.

IX [8] batches network requests in the presence of network congestion and allows application threads to issue batched system calls. Similarly, we batch I/O requests based on the observed I/O concurrency by examining the queues in the I/O path and additionally we use a performance feedback mechanism (achieved throughput) to adapt batching, regardless of the network conditions.

VIII. Conclusions

Our work shows how networked storage protocols over raw Ethernet can achieve low host CPU overhead and high network link utilization for small I/O requests. We analyze in detail the overheads of Tyche, our in-house networked storage protocol deployed directly over Ethernet. We point out that context switches significantly increase overhead, especially for small I/Os. Then we propose a new design for network storage protocols that reduces context switches for small I/Os and that is particularly effective 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. The novelty is that our batching mechanism dynamically calculates the degree of batching based on the throughput obtained without a negative impact on I/O overhead and response time.

Our results show that host CPU overhead is an important source of overhead, and it represents up to 65% of the total overhead. Considering only our protocol, for a 4kB request, its Tyche overhead represents 47%.

We reduce total overhead by up to 31% with our optimized protocol that reduces context switching. Considering only the overhead introduced by Tyche, avoiding context switches reduces overhead by up to 61.1% (from 36.1μs to 14μs) per 4kB-I/O request, about equally divided between the initiator and target. Adaptive batching improves link utilization for small I/Os by up to 56.9% and 53% for reads and writes, respectively, achieving 88% and 81% of the theoretical link utilization for 4kB reads and writes, respectively. Overall, our approach is able to achieve 287K 4kB IOPS out of 315K IOPS possible on a 1.2GBytes/s link, without any hardware support.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the NanoStreams (FP7-ICT-610509), HiPEAC3 (FP7-ICT-287759) projects, and the COST programme Action IC1305, 'Network for Sustainable Ultrascale Computing (NESUS)'. We are also thankful to Manolis Marazakis for his helpful comments.

References

[1] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A Five-year Study of File-system Metadata," *Transactions on Storage*, vol. 3, no. 3, Oct. 2007.

[2] D. Roselli, J. R. Lorch, and T. E. Anderson, "A comparison of file system workloads," in *Proceedings of USENIX Annual Technical Conference*, 2000.

[3] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2013.

[4] P. González-Férez and A. Bilas, "Tyche: An efficient Ethernet-based protocol for converged networked storage," in *Proceedings of the IEEE Conference on Massive Storage Systems and Technology (MSST)*, 2014.

[5] FIO Benchmark. [Online]. Available: http://freecode.com/projects/fio

[6] C. Li, C. Ding, and K. Shen, "Quantifying the Cost of Context Switch," in *Proceedings of the Workshop on Experimental Computer Science*, 2007.

[7] J. Gim, T. Hwang, Y. Won, and K. Kant, "SmartCon: Smart Context Switching for Fast Storage Devices," *Transactions on Storage*, vol. 11, no. 2, pp. 5:1–5:25, Mar. 2015.

[8] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "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)*, 2014.

[9] S. Swanson and A. M. Caulfield, "Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage," *Computer*, vol. 46, no. 8, pp. 52–59, 2013.

[10] S. M. Rumble, D. Ongaro, R. Stutsman, M. Rosenblum, and J. K. Ousterhout, "It's Time for Low Latency," in *Proceedings of the USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[11] M. Flajslik and M. Rosenblum, "Network Interface Design for Low Latency Request-Reponse Protocols," in *Proceedings of the USENIX Annual Technical Conference*, 2013.

[12] A. Kalia, M. Kaminsky, and D. G. Andersen, “Using RDMA Efficiently for Key-value Services,” in *Proceedings of the ACM Conference on Special Interest Group on Data Communicatio (SIGCOMM)*, 2014.

[13] C. Mitchell, Y. Geng, and J. Li, “Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store,” in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[14] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro, “FaRM: Fast Remote Memory,” in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[15] P. Stuedi, B. Metzler, and A. Trivedi, “jVerbs: Ultra-low Latency for Data Center Applications,” in *Proceedings of the Annual Symposium on Cloud Computing*, 2013.

[16] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, “Chronos: Predictable Low Latency for Data Center Applications,” in *Proceedings of the ACM Symposium on Cloud Computing*, 2012.

[17] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, “Providing Safe, User Space Access to Fast, Solid State Disks,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[18] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, “Aerie: Flexible File-system Interfaces to Storage-class Memory,” in *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[19] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, “Arrakis: The Operating System is the Control Plane,” in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[20] A. MacDonald, “Nfsv4,” *login:*, vol. 37, no. 1, Feb. 2012.

[21] (2014) Linux 3.14. [Online]. Available: http://kernelnewbies.org/Linux 3.14/