WZBLK: Network Block Device for Commodity Ethernet LANs

undergraduate thesis of
Dimitrios Apostolou
jimis@csd.uoc.gr, jimis@gmx.net

supervised by
Prof. Angelos Bilas
bilas@csd.uoc.gr

May 9, 2014
Computer Science Department
University Of Crete

Contents

Abstract

In this document the design and implementation of WZBLK, a network block device for commodity Ethernet LANs, is presented. WZBLK is a network block device implemented in the Linux kernel. Concept, design and specific optimisations are described in depth. Finally it is benchmarked against the industry standard, iSCSI, and is shown more performant under the conditions of the test.

1 Introduction

1.1 Problem Definition - Motivation

Ongoing improvements in Ethernet, the most prevalent commodity network technology, are promising link rates in the range of 40-100 Gbps, matching those of highly expensive specialised interconnects. From the perspective of a storage software system engineer, major challenges appear when using Ethernet, even more concerning the usual combination of Ethernet with TCP/IP, primarily because such rates are stressing the limits of the CPU and the memory controller's bandwidth capabilities.

Commodity networking technology (Ethernet) has traditionally been lagging in speed advances compared to CPU clock frequencies, as a result factors such as memory copies and interrupt handling have not been an issue until lately that *CPU clock advances have mostly stalled*, and development paradigm has been shifting to taking advantage of multiple CPU cores. It does not come as a surprise that most traditional solutions are not designed to take advantage of multiple cores, or multiple parallel links, and implementations have been mostly unoptimised regarding resource utilisation.

Finally, a common way to achieve economy of scale in big HPC (High Performance Computing) environments is to try utilising multiple commodity links in parallel with the ultimate goal of presenting a single logical connection at the application layer. With no support from the network, either at its core (switches) or at its edge (NICs), and given the current trends in multi-core CPUs, it is a major challenge to find a solution that scales good in both areas, i.e. multi-core and multi-link utilisation.

1.2 Historical Overview

In the short history of computing the need for accessing remote data has existed since almost the beginning. In both personal and scientific computing there have been a multitude of use cases with different requirements in performance, reliability and ease of use.

Especially within the High Performance Computing (HPC) community, the requirements have never seized to augment. Today enormous computer rooms provide space for thousands of nodes, running applications demanding access to PetaBytes of data.

For the purpose of accessing remote data, many different protocols have been developed, custom APIs (Application Programming Interfaces) and user-level abstractions, each one featuring a different set of characteristics and tradeoffs. However the *default* way of accessing local data has always been the filesystem, and to take advantage of the network each application must be separately redesigned and refactored for that purpose, or the data must be fetched to the local filesystem before, and sent back afterwards using separate syncing tools, which imposes extra overhead and is not always possible.

In order to take advantage of the multitude of existing programs designed to access data on *local storage*, abstractions have been developed that allow transparent access to remote storage, via performing standard system I/O (Input / Output) operations. These abstractions generally fall into two categories, (1) *network filesystems* and (2) *network block devices*.

Network filesystems directly provide filesystem semantics, i.e. a hierarchical tree of directories containing files with data of arbitrary size, each one with a specific set of attributes like type and permissions. It is a high level approach to the problem, and the most popular historically, with successful examples like NFS and CIFS.

Network block devices on the other hand only provide the semantics of a block device, i.e. *read*, *write* and *seek* over a given amount of space. Due to this simplicity, it is most times necessary to create and mount a filesystem on top of it so that applications can have access to the remote data. This low-level abstraction has only become popular lately, with the most prevalent example being iSCSI. *The implementation presented here, WZBLK, falls into this category.*

The network back-end in both cases can be implemented in a variety of ways, with the most important detail being the communication protocol. Since *reliability* is a necessity, TCP/IP is the obvious choice and is indeed used in both NFS[1] and iSCSI.

1.2.1 Case Study: iSCSI

iSCSI is a protocol that specifies the encapsulation of SCSI (Small Computer System Interface) over IP. SCSI commands are carried on top of IP, allowing the connection of storage devices (*Targets*) to servers (*Initiators*) over any network distance without special cabling, but utilising existing networking infrastructure. Since the SCSI commands are largely unmodified, this network layer is plugged on top of

[1] NFS can also be configured to connect over UDP/IP since it is implemented via Remote Procedure Calls (RPC), a protocol which handles retransmissions at a higher level.

existing SCSI support in operating systems, thus allowing the devices to be seamlessly managed as local ones.

1. *Initiator* is the name given to the *client-side* of an iSCSI connection, i.e. the side that initiates the I/O requests. It might be confusing since in the common case the iSCSI Initiator is a server machine in a datacenter.

2. *Target* is the name that represents the *server-side* of the connection, i.e. the one that replies to all requests sent by the Initiator. For the iSCSI case the Target is basically a machine with storage devices, which upon reception *completes* the I/O requests to the actual block devices and notifies the sender.

Both Target and Initiator are usually implemented in software as a layer on top of the operating system SCSI or block subsystem. However there exist expensive hardware implementations for both sides, that combine their power with embedded TCP offload engines to overcome most performance problems. [1] Only software implementations have been examined for the work presented in this paper.

The recent surge in popularity of iSCSI can be attributed mostly to the increasing rates of networking speeds in Ethernet and WiFi (802.11) equipment, creating the common need for cheap but high performance storage servers. Storage servers of Gigabit speeds and upwards today are considered commodity equipment and are found in frequent use among consumers. And of course HPC needs are even higher, as already discussed. Given such high requirements, the simplicity of a network block device (like iSCSI) in comparison to a network filesystem (NFS) has been a primary motive in the adoption of the former. The focus in performance is prevalent [2] and indeed there has been a difference in speed due to a variety of factors. Today iSCSI is implemented in most NAS and SAN devices, purely in software or with acceleration modules in hardware for enterprise grade devices [3], and is also implemented in software in most current operating systems, e.g. Linux, Solaris, Windows.

As mentioned, iSCSI is leveraging TCP/IP as communication protocol. The TCP/IP protocol is designed to serve all kinds of links from low to high latency, and from low to high bandwidth as well, in the same computer room or at the other end of the world. Despite its advantages, some of the functionalities TCP/IP provides are mostly unneeded to the simple idea of the network block device, for example ordered delivery and stream oriented connections. In addition TCP/IP is so widely used, from low-latency, high-bandwidth to high-latency, low-bandwidth links, that it is bound to show inefficiencies under certain conditions, like the high bandwidth - low latency interconnects of modern HPC clusters that today support transfer rates in the order of 10 to 100 Gbps[2] which highly stress every component to the limits.

[2] 1 Gbps == 10^9 bits per second see A.1

In addition, the use of TCP/IP allows the protocol to be transparently routed at the IP layer, giving the possibility of having the block devices far away from the system. Nevertheless it is notable that iSCSI is mostly in use within the same LAN, since any transient network errors that trigger TCP retransmissions have devastating effect in performance.

Finally, it must be clarified that in the end iSCSI does not provide a filesystem to the Initiator, but only a block device. It is the common case that a filesystem is created on top of that block device, and usually that filesystem is generic and network-agnostic. Therefore it is not uncommon for the filesystem to behave erratically in the case of TCP retransmissions and other major delays, which creates another argument for placing the Target within the same LAN as the Initiator.

1.2.2 Emerging technologies

Other relevant technologies that are currently on the rise are ATA over Ethernet (AoE), and Fibre Channel over Ethernet (FCoE).

The former boasts a very simple specification and is being marketed towards personal or small-business consumers. The latter inherits all the enterprise characteristics of Fibre-Channel and is targeting mostly datacenters. They both encapsulate the respective block device protocols directly over Ethernet. As a result they are not routeable at the IP layer, however they overcome inefficiencies that iSCSI inherits from its dependence on the TCP/IP stack.

Both protocols are still far from enjoying the popularity and wide acceptance that iSCSI has.

1.3 Introducing WZBLK

Although iSCSI has been serving the need for cheap and performant storage up to Gigabit speeds very well, it has not been coping well with recent advancements in technology. The introduction of 40 Gbps and 100 Gbps Ethernet and the ongoing commoditisation of 10 Gbps in combination with the wide availability of Solid State Storage, has brought demand for very higher performance. We'll try to enumerate the reasons, most have been discussed above.

- TCP provides features like connection-oriented communication and ordered delivery, that impose a burden to the software implementations for very high rates, and are not really needed in the scatter/gather, multiple request I/O model that a block device serves.

- IP allows routing the storage devices over the entire Internet, which is contradictory to high-performance needs in an office or datacenter network. Furthermore, possible packet loss in the network routers due

to competition for bandwidth among different protocol stacks, leads to devastating performance problems in the network-agnostic filesystems created on top of the iSCSI block devices. Thus, connecting to iSCSI endpoints outside of LAN's borders is already being avoided.

- Link-level parallelism (e.g. via bonding interfaces) is a common technique to further enhance network performance by taking advantage of multiple commodity hardware instances. For it to happen transparently though, it has to multiplex one TCP stream to multiple links, which TCP/IP does not provision for thus it is bound to the limitations of the generic TCP/IP protocol stacks. (proven to not scale good by our measurements)
- Multi-CPU parallelism is also a common technique to enhance performance, since today advancements in CPU clock speeds have mostly stalled and multi-core CPUs are prevalent. TCP/IP is not friendly to that concept either, especially given that transparent usage of established APIs (e.g. Berkeley sockets) encourages synchronous handling of requests.

In short, Ethernet speed and cheap Solid State Storage (SSD, see A.1) reaching the memory subsystem's bandwidth limits together with the need for exploiting multi-core and multi-link parallelism for advancing performance, create the need to architect a solution from the ground up able to handle all these issues.

As a proof of concept that these issues can be dealt with efficiently using only commodity hardware, we implemented WZBLK.

2 WZBLK Description

2.1 Technologies used

2.1.1 Multiedge

WZBLK relies on Multiedge [4], a thin protocol that sits directly over Ethernet and provides reliable and scalable data transfer.

Multiedge is essentially a reliable Ethernet transport protocol, designed to take advantage of multiple parallel links with no support from the network infrastructure, and to efficiently parallelise its work over multiple CPU cores [5] [6].

RDMA (see A.1) protocols have the advantage that they do not require significant buffering at the receiving NIC, as data arriving can always be delivered to a specific memory buffer. However, they require some type of protocol to agree in advance on the buffers used for communication and their size, establishing in essence credit-based flow control for buffer management.

Another advantage of RDMA protocols has traditionally been the *zero-copy* capability. The term *zero-copy* characterises software design, where data is moved in-between application buffers, but there is no actual copying performed by the CPU. For this case involving network and storage devices, it usually means that *only DMA transfers* take place, either between the NIC and system's RAM, or between the storage controller and system's RAM, but no extra RAM-to-RAM copies. This has proven to be complex, especially on the receiving side, meaning that special hardware support has been used and that this feature is generally unavailable with commodity Ethernet network interfaces.

The sender part of the protocol is straightforward since the kernel context that initiates the send operation can also issue the DMA to the NIC directly without any further context switches, and block this context (thread) until completion.

The receiving side is significantly more complex, due to the need of honouring RDMA semantics without performance penalty. When a packet arrives, it usually carries on its header the destination address, and needs to be written directly there. Over commodity Ethernet network interfaces, since controllers don't support RDMA semantics, the packet upon arrival can only be immediately transferred to the next free buffer in the Rx ring (see A.1). The complication is that each arriving packet will simply use the next available buffer, since the destination RDMA address can only be determined *after the CPU parses the packet.* Thus each arriving packet will be transferred to a buffer in memory, however not the buffer specified by the RDMA protocol.

Multiedge solves this issue for user-space applications using a complex memory remapping and buffer replenishing technique: Physical pages belonging to the NIC's Rx ring are *swapped* with the proper pages from the user-space application. This swapping involves page table traversal and modification, so that the physical page in the Rx ring's buffer is mapped to the process's address space, the existing page in the process's address space is unmapped, and is finally put in the Rx ring and marked as free. It's possible that it also involves TLB flushing, though it's avoided in the common case.

For kernel-space usage, especially for I/O, there are further issues faced that will be described later.

2.1.2 Violin

WZBLK also leverages the Violin [7] filesystem framework to present a block device to the end user, allowing to transparently access multiple block devices within the same Ethernet LAN.

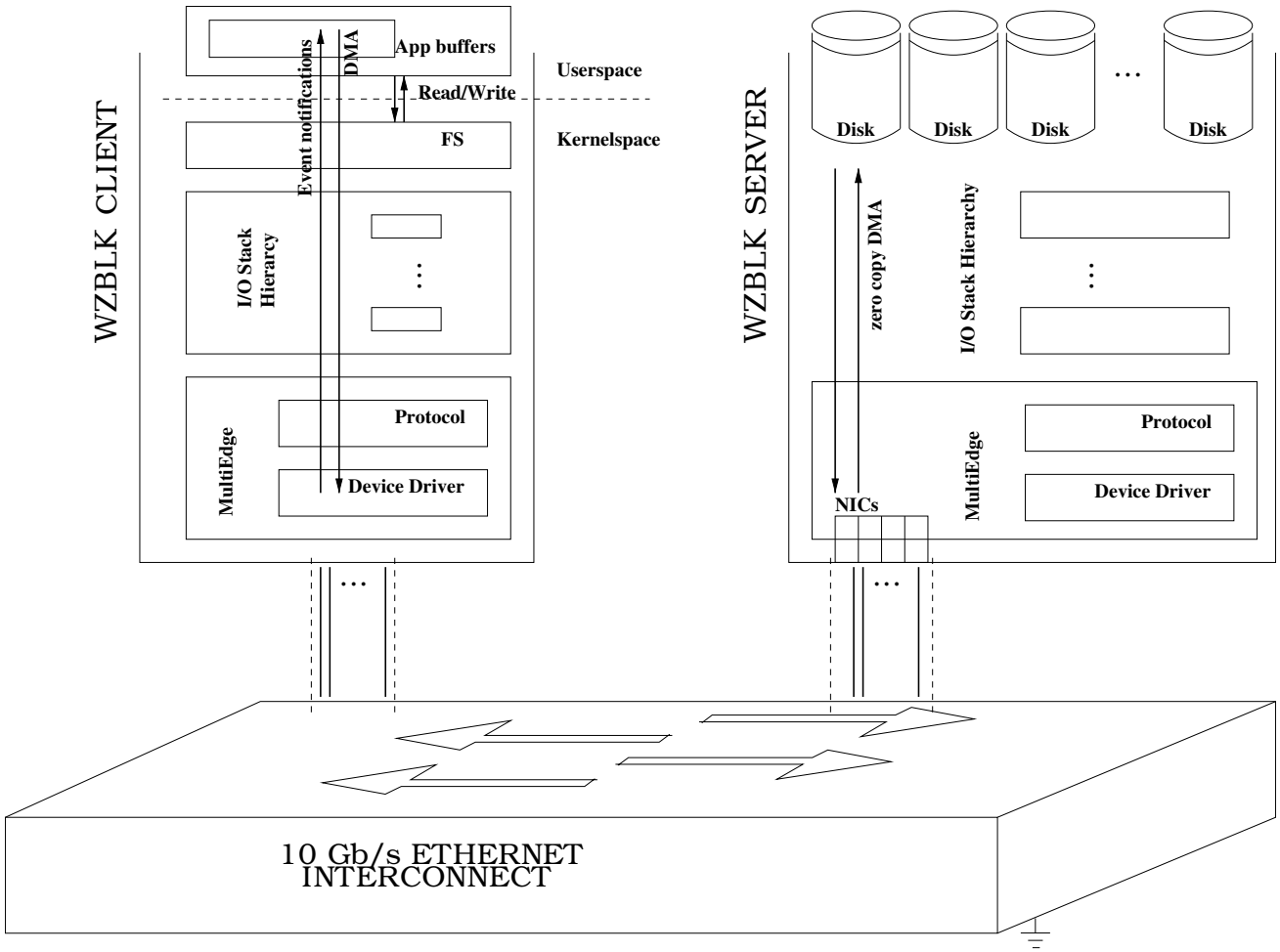
Violin [7] is a block-layer virtualisation framework within the Linux kernel. Essentially it combines multiple block-layer mechanisms into one, and ultimately exports the resulting block device as `/dev/violin1`. New mechanisms can easily be implemented by adhering to a simple but powerful API capable of both synchronous and asynchronous implementation.

Ultimately, Violin allows storage administrators to [8] [9] create arbitrary, acyclic graphs that fully define the dependency hierarchy of a complex storage system that would be hard to create otherwise. Combining heterogeneous devices within the same system or across different ones is made feasible with relatively-simple, completely kernel-space high performance module implementations. Example Violin layers have already been implemented for functions like RAID redundancy, block checksumming, network transparency - the latter was used and extended as part of WZBLK.

Overall, for WZBLK Violin provides a fully-featured abstraction layer that handles all possible block-layer combinations of operations. As a result WZBLK implementation does not care about the details of each operation, just deserialises / serialises from the network all the relevant structures of the Violin API, and makes the respective calls.

2.2 WZBLK

2.2.1 Overview



WZBLK, in summary, is a thin glue layer, between Multiedge [4], utilised as the communication protocol and Violin [7] block-layer virtualisation framework, which facilitates the integration of networking support to the exported block device. With primary goal the saturation of today's fastest Ethernet network technology, it boasts simplicity in order to achieve low overhead per operation and scalability in both CPU cores and network links.

Multiedge [4], as described, is a thin protocol stack directly on top of Ethernet that provides reliability. Thus WZBLK is restricted to devices within the same Ethernet LAN but because of Multiedge's design characteristics, WZBLK is inherently capable of scaling on multiple parallel links between two hosts thus providing *link aggregation* capability, as well as scaling to multiple cores, which proves of great importance given that current CPU technology is primarily advancing in core count rather than clock frequency.

WZBLK presents *zero-copy* characteristics on most of the data it transfers. Utilising technologies of high performance Ethernet, like *jumbo frames*, and using a minimum transfer unit of x86 architecture's *page size*, i.e. 4 KB, which is also the preferred block size for most filesystems, it is capable of transferring data without unnecessary copies where possible.

The advantages of this approach have been measured in a variety of experiments, where WZBLK is shown to consistently surpass its competitors. Performance is shown in many cases to be multiple times that of iSCSI, demonstrating scalability in utilisation of all the testbed's resources, including multiple 10 Gbps Ethernet links as well as multiple cores.

2.2.2 Design and implementation

In a typical TCP/IP network block device implementation [3], the client (Initiator) includes a layer below the block layer that simply forwards I/O requests to the server (Target). The Initiator sends either a read request (expecting data as the reply to the request), or a write request together with data (expecting a simple *completion* reply).

The Target typically uses a thread that blocks waiting requests from the network, listening to the specific TCP/IP socket. This thread receives all incoming I/O requests and re-issues them to the block layer of the target system. Thus, in the target system, all data crosses the memory hierarchy at least twice, once from the Ethernet network adapter to the listening thread's address space, and another from there to the storage devices (disks, SSDs etc). The target thread is

[3] NBD [10] being the simplest example, embedded in the Linux kernel source tree

then notified via callback for its request and the data, in the case of read request.

It is typical for the processing thread on the Target to be receiving a large number of outstanding requests as in high-end I/O workloads all operations need to be asynchronous. When the Initiator receives the response from the Target - either a completion in the case of write or actual data in the case of read - the network layer below the block device wakes up the kernel context that has issued the I/O request, which in turn will complete the I/O all the way up to userspace. So on the Initiator side, multiple concurrent I/O operations can occur like on the Target side, with the primary difference that they originate from different process contexts, be it applications or kernel modules.

All of this is valid for WZBLK, with the major differentiator being that it not built on top of TCP/IP but on top of Multiedge network protocol, not using socket but *RDMA operations* for efficient meta-data (request) transfers, and *pagelist operations* (see 2.2.6) for transferring the actual data with minimal overhead. Consequently, reduced CPU utilisation and I/O response time is achieved while leveraging important attributes of Multiedge, like transparent utilisation of multiple network links, out-of-order request processing and completion, and aggressive coalescing of network interrupts.

Ideally, in a *zero-copy* design, the major burden to the memory subsystem should consist only of data transfers, in-between the system and the network or storage controllers, which are performed via Direct Memory Access (DMA) and do not impose burden to the CPU. That is the case for WZBLK, however, there is one exception that is very hard to alleviate without specialised hardware, and RAM-to-RAM copying of data has to be performed.

2.2.3 Data copy for data received on the Initiator

This case is on the Initiator, when *read* requests are sent, and replies with the data are received from the network. Due to the way Ethernet is designed, the network controller does not know in advance neither the order in which the packets will arrive nor the multitude of them, so it can only perform DMA directly to a part of memory called the *Rx ring*.

The complication is that when the completion with the data arrives over the network, the NIC will store the pages received, in the order received, in the Rx ring. On the Initiator side a *memory copy* is needed, since the `bio_req` request that is about to be completed already has a `struct bio` (see 2.2.6) with the needed empty pages attached, as per the Linux Kernel's block layer requirements. These pages are decoupled from any mapping to virtual address space, and that means the pages might be mapped to some user-space buffer that initiated a `read()` operation, or might be part of in-kernel read-ahead activity with no user-space reserved address space, i.e. the pages are not mapped in any process address space. In short: The only way to complete the read I/O request, is to manually copy all data to its reserved pages.

Our measurements have shown that this single copy imposes significant overhead for the CPU on Multiedge's receiving threads at high rates (exceeding 1 GB/s). Multiedge's scalable design alleviates this issue by having the possibility for multiple receiving threads (currently 4 maximum defined as compile-time constant) performing the copies. Each of the threads can deal with data arriving to any of the available network interfaces.

See 4.2.1 for proposal on how to overcome this overhead.

2.2.4 Freeing memory on the Target, for sent data

A further complication is that freeing memory pages used by the network layer is challenging for read requests. After the *Target* has received the *read request*, it uses preallocated pages to form a request and submit it to the `bio` layer, and of course marks these pages as reserved. When the I/O layer completes the request and data requested is available in those pages, a network reply with this data is sent to the Initiator, however these pages can not be freed and returned to the preallocated pages pool. Reason is that it may be the case that *retransmission is needed*, in the event of packet loss or other network transient error, since Multiedge is a *reliable* protocol.

This is resolved by implementing a new feature in Multiedge, which enforces freeing of data pages when *positive acknowledgement* of the specific or later Multiedge packet arrives. This mechanism can be generalised to support calling arbitrary functions on the event of verified delivery, but this imposes further complexity since it involves passing a hook down to the lowest layer of Multiedge.

2.2.5 Interrupt mitigation

A problem for Ethernet at over Gigabit speeds, is the overhead induced by serving interrupts which typically occur on a per-packet basis. When the rate of received packets grows high, the kernel has to serve thousands, even millions of interrupts per second in order to receive them. Besides the danger of dropping packets because of missed interrupts, the overhead to the the CPU is too high, mostly because of CPU time spent in the uninterruptible top half interrupt handler (see A.1) and the constant context switching incurred because of this.

We handle this issue by taking advantage of interrupt moderation which is a key feature of Multiedge. Quoting [4] section 2.6 "Reducing Overheads":

> The design of Multiedge tries to minimize interrupts both in the send as well as the receive path in the following way: When an interrupt arrives, the interrupt handler disables

subsequent interrupts and notifies the protocol layer. When the receive or send protocol path is invoked by an interrupt handler, it processes all pending interrupt related events, e.g. send frame completions or newly received frames, by polling each network interface. The protocol layer enables interrupts when there are no more interrupt related events and no protocol kernel thread is active.

2.2.6 Modifications with pagelists and simplifications for kernel use

Performing an RDMA operation consists of transferring data from a continuous region of local virtual address space to a specific continuous region in remote process's virtual address space. However an I/O request to or from a storage device typically consists of a scatter-gather list of memory areas, not necessarily continuous. Furthermore, as mentioned (see 2.2.3) an I/O request does not necessarily consist of pages mapped in any process's address space, since it is common for the kernel to initiate I/Os as part of read-ahead operations, i.e. data which will be later used, or even not used at all by user-space. This is made obvious from the `struct bio` implementation in the Linux kernel, which is a scatter-gather list describing the data of each I/O.

```
struct bio_vec {
    struct page     *bv_page;
    unsigned int    bv_len;
    unsigned int    bv_offset;
};
struct bio {
    unsigned short bi_vcnt;
    struct bio_vec *bi_io_vec;
    /* ... various accounting fields ... */
};
```

As can be seen it is basically a list of physical pages, i.e. `struct page`, completely decoupled from any virtual address space mapping. Given that data received from the network controller is put via DMA to unexpected (due to Ethernet's controller limitation) physical pages in system's RAM, there are basically two options for implementing the RDMA operation: Either perform a separate RDMA operation for each memory page, or map the scatter-gather list at some virtual address space, perform the RDMA operation, and unmap in the end.

Both options are overly costly at high data rates, primarily because *data copying* would still be necessary to fill the proper physical pages that the `struct bio` was paired with in advance, when the I/O operation was issued, and secondly because of heavy page table traversal operations that thrash the TLB.

Multiedge, having been originally designed for being used from user-space applications specifically for RDMA purposes, had to be ported for in-kernel use and had to be tuned specifically for our use case, i.e. network to block layer communication and vice-versa. Its original RDMA operation type is *not* being used for actual data but for transferring only *metadata*, i.e. requests. Data is being handled by introducing a new type of operation, the *pagelist transfer*.

```
struct pglq
{
    /* an array of PGLQ_SIZE preallocated
       pagelists. */
    struct page ***pgl;
    /* tail+len is the head of the queue */
    int tail, len;
    /* bitmap showing if each pgl is used */
    DECLARE_BITMAP(bitmap, PGLQ_SIZE);
    struct completion comp;
    struct mutex mtx;
};
```

A *pagelist* is equivalent to `struct page **` essentially a scatter-gather list of memory pages which correspond to the blocks of data a request refers to, so that DMA can be initiated without further processing, either to or from the devices, for write or read requests respectively. During the connection negotiation phase, a list of pre-allocated pagelists is exchanged so that both Initiator and Target know the total number and all relevant identification handles of each other's pagelists. At the Initiator, pre-allocation refers solely to the list of pointers to pages, and not to the pages themselves, as these are provided from/to the kernel per each request's context. At the Target the pagelists refer to lists of pointers to pages, plus the memory pages themselves (see 2.2.8 for the reasoning).

These pages are separate from the ones in Multiedge's network Rx ring (see A.1), and serve the purpose of providing a supply of pages ready to replenish the Rx ring, where data arrives in random order. As explained, they are the exact amount that will be needed in worst-case scenario, given that the Initiator knows in advance this amount and will never overflow this area (credit-based communication). With this strategy, remote I/O requests can be issued with DMA transfers only and without RAM-to-RAM copies, in most cases.

The pre-allocation and exchange of *pagelists* for each logical connection in WZBLK is a flavour of *credit based flow-control* that leads to great simplifications both in the buffer management implementation and at the protocol level. More details on this subject are explained in the next section (see 2.2.7).

2.2.7 No allocations are needed during operation

Given that the resources of each node are strictly determined and communicated once during the initial handshake, and given that a storage resource is in use always between exactly two nodes, Initiator and Target, with careful design it is possible to avoid overflowing any of the two nodes, so congestion of requests is a non-issue.

In particular, during the initial handshake, the following sequence of events takes place:

Initiator	Target
	Blocking for connection
Open Multiedge connection to Target	
	Register memory region for RDMA operations (mostly meta-data transfers)
	Block for notification of RDMA operation
Register memory region for RDMA operations (mostly meta-data transfers)	
Allocate local pagelists (pointer to pages), empty (pointing to NULL)	
Send pagelists information via RDMA operation	
Block for notification of RDMA operation	
	Receive and store remote pagelist information
	Allocate local pagelists, including all physical pages
	Send pagelists information via RDMA operation
Receive and store remote pagelist information	
WZBLK ready	WZBLK ready

All slots for requests are preallocated, and each end always keeps account of free slots in both his own and the remote pagelist queues.

In such way a major issue with Ethernet is alleviated: the possibility that at any time data might arrive and network ring will overflow does not exist. When the Initiator sends a page-list it is known in advance that there is an already allocated free page on the other side, and the same goes in the other direction. That way if too many outstanding I/O operations are taking place, the Initiator will automatically withhold them until completions of previous requests have arrived, thus slots are freed on the Target, and this will cause blocking up to userspace, or cause an error in case the user is utilising non-blocking I/O API.

In addition, generic allocations (`kmalloc()`) are avoided during operation of WZBLK. These allocations can be overly expensive under memory pressure and can even fail when the VM subsystem is under a lot of stress. On the contrary handling our preallocated pages is fast since it mostly involves scanning a small bitmap to reserve memory of given size (`PAGE_SIZE`), unlike the generic `kmalloc()`.

2.2.8 VM page remapping

Multiedge boasts a complex page-remapping implementation (mentioned again in 2.1.1), in order to forward to userspace all the data coming from the network via an RDMA operation. In short, already mapped pages in the address space of the process are being unmapped, and the pages containing the data in the kernel's RX ring are being mapped to those virtual addresses. The pages unmapped are being used to replenish the RX ring. See [6] section 3.2 "Context Independent Page Remapping" for full details.

As mentioned, WZBLK does not use RDMA operations for data transfer since it is implemented in kernel-space and most I/O operations involve unmapped physical pages (see 2.2.3). Thus, we completely disable the page-remapping functionality and use a fairly simple procedure. As an example, on the *Target* when a *write* request is received accompanied with data, the following actions are taking place:

1. on reception of the pagelist, all relevant pages are detached from the RX ring and attached to a `struct bio`.
2. The RX ring is replenished from the preallocated pages pool. This is the reason that the Target needs actual pages preallocated.
3. The I/O request, together with the attached pages, is submitted to the block layer according to the metadata received.

Nowhere in this path is a context-switch or a page table traversal needed. As a result, most of Multiedge's original overhead, related to frequent TLB flushing because of page-table modifications, is not an issue for WZBLK.

3 Evaluation

3.1 Experiment description

3.1.1 Hardware setup

For this experiment we have set up two machines to play the role of the Initiator and the Target. They are both of nearly identical characteristics: 4 AMD Opteron dual-core CPUs, for a total of 8 cores per host. For network connectivity we equipped each host with 4 MyriCom 10 Gbps Ethernet PCI-express NICs. These 4+4 NICs are connected back-to-back (no switch) between the two hosts and are *dedicated* to our experiments, since we perform the necessary management activities through the on-board interfaces.

The only difference in hardware configuration is that the Initiator machine is equipped with 4 GB of RAM while the Target boasts 32 GB of RAM, for the purpose of having a sizeable ramdisk as a block device to export, plus 8 Intel 32 GB SATA single-cell SSDs for exporting real block devices.

3.1.2 Execution

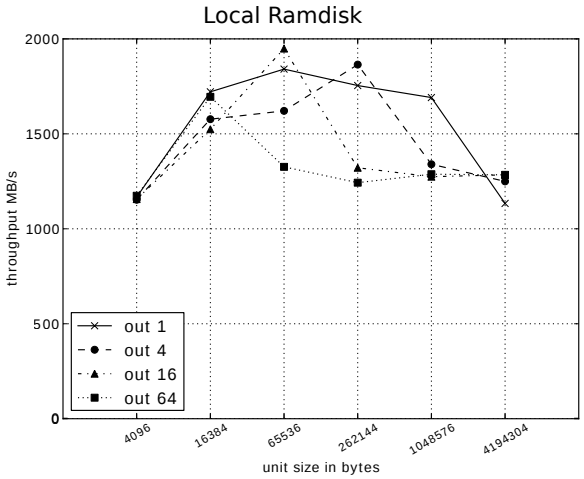
We want to measure low level performance of WZBLK. In order to avoid application overhead and results differentiation due to complex workloads, we choose to utilise a micro-benchmark, zmIO.

zmIO [11] is a low level block device micro-benchmark. It issues read or write (or mixed) I/O requests asynchronously, using the Linux AIO API [12], with the ability to maintain a given queue depth of submitted incomplete requests.

To maximise the network bandwidth utilisation we experimented with various setups for the exported block device at the *Target* machine, including various ramdisk implementations, exporting multiple targets – one for each SSD, exporting a single block device created via RAID-0 over all SSDs etc. In all these setups it was common to often identify as primary limiting factor either the total disk bandwidth or even the system's memory bandwidth (measured using `memstress` [13] custom utility), and almost never the network bandwidth which in our case is ample and theoretically surpasses 4 GB/s[4].

A quick benchmark shows that it will not be easy to bring our network setup to its limits. The following graph shows throughput with read-only workload on the local RAM drive, with no network involved, for various queue depths (1, 4, 16 and 64) of outstanding I/O requests.

[4] 1 GB/s == 2^{30} Bytes per second see A.1

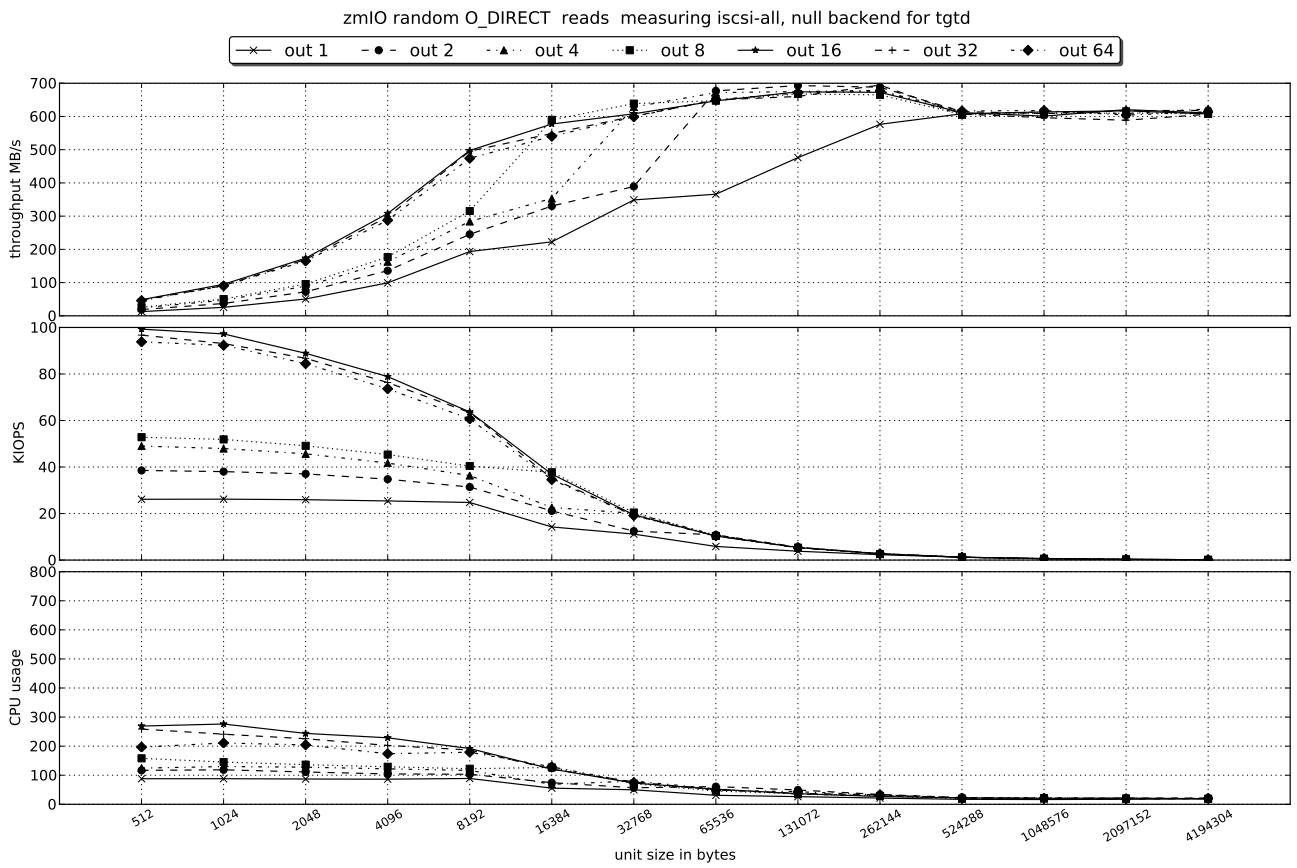


Thus the results presented here involve special setup at the *Target* that avoids reading or writing actual data. For iSCSI, we use *tgtd*'s (see A.1) `null` back-end. For WZBLK we export a block device created by *ramdrv.ko* (see A.1) RAM drive, using its special `no_memcpy` option that avoids copying any data to or from the ramdisk. Even though these back-ends result in garbage data being read or written, thus are useless for real use cases, they are a very good way of stressing our implementation (and the iSCSI implementation as well) to the maximum. In fact they emulate a system with very fast storage subsystem, since that would involve no in-memory copies (only DMA transfers) for reads/writes to the storage device, as opposed to multiple memory copies involved with a regular ramdisk.

3.1.3 Measuring iSCSI

Next we setup the Target machine's four network interfaces with different IP addresses and use tgtd's *null* back-end [14] to export four iSCSI targets, one on each network interface. All four of them are mounted separately at the Initiator machine. This was the most performant way among other setups tried, e.g. bonding all four interfaces and exporting a single mount-point, or exporting 4 of them and using software RAID-0 on the Initiator to unite them to one block device.

We use zmIO at the Initiator, configured to issue random I/O requests concurrently to all remote block devices, using request sizes of 4 KB to 4 MB and queue depths of 1 to 64 outstanding requests, measuring all possible combinations in separate experiments. The results are presented in the following graphs, including measurements for bandwidth, IOPS (I/O Operations per second), Initiator and Target CPU utilisation. Different curves have been drawn for varying number of outstanding I/O requests. Results were similar for both reads and writes, so only the former are shown.



Multiple lines present the varying queue depths for outstanding I/O operations It is notable that performance with only one outstanding operation is low, but scales well while increasing it up to 16 outstanding operations. Performance also increases by increasing the I/O operation unit size (data being transferred per I/O request) up to 256 KB. Nevertheless, *The performance cap for tgtd seems to be around 700 MB/s and 100 K IOPS.*

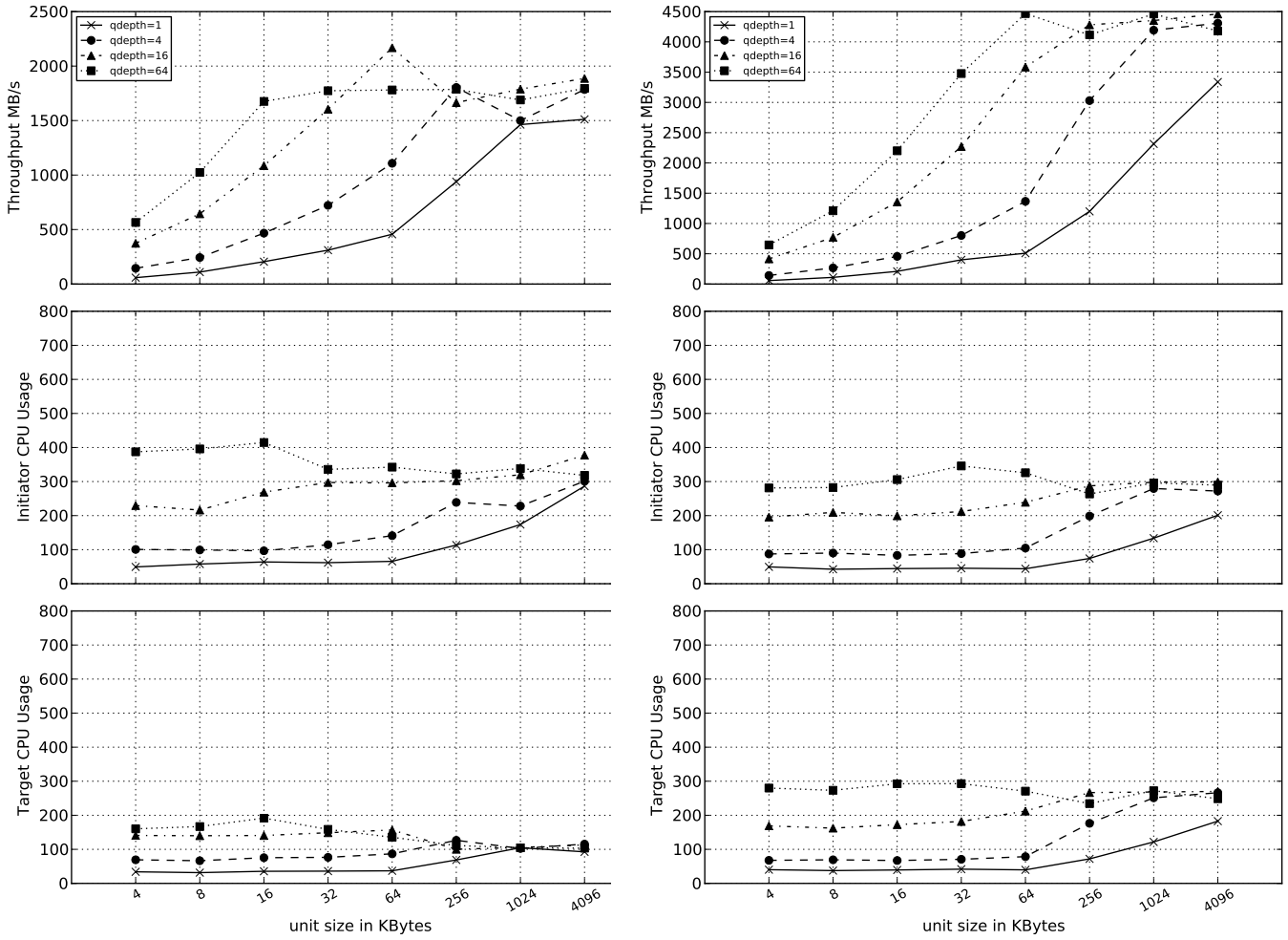
Further investigation of the complete accounting data of the experiment reveals that this is not a limit imposed by the iSCSI protocol itself, but most probably an inefficiency of the tgtd implementation: the tgtd daemon on the Target machine never utilises more than 120% of CPU (maximum is set to 800% for all 8 cores), and even though there are multiple threads running, the bottleneck seems to be always at one thread which always utilises one core at 100%. This was the case in all of our experiments, no matter which tgtd back-end we used, or how many mount-points we exported (reminder: this experiment involves four, one per network interface).

3.1.4 Measuring WZBLK

Next we present the relevant results for WZBLK. We create the 28 GB ramdisk using the `no_memcpy` option of ramdrv.ko (see A.1) and export it to the network. All four NICs are **not** configured with any IP address as WZBLK uses Multiedge, a thin protocol directly on top of Ethernet (see 2.1.1). In addition no interface bonding or multiple mount-point exports are needed as Multiedge is inherently capable of scaling across interfaces. On the Initiator side we mount the exported mount-point, which appears locally as `/dev/violin1`.

The set of results shown here present WZBLK configured (via a compile-time constant) to use 4 threads for receiving data on the Initiator (full data is also available for 1 or 2 receiver threads configuration). This is the most performant configuration because the extra memory copies taking place at the initiator are distributed among multiple processor cores.

The two set of graphs that follow involve random reads and random writes respectively. Each one shows throughput measurements for random read/write operations of various sizes (x-axis), continuously keeping a load of 1, 4, 16, or 64 outstanding operations (different curves). Each triplet of graphs has its primary data on the top graph, with CPU usage (800% peak when all 8 cores are fully utilised) client-side and server-side plotted on the two bottom graphs.



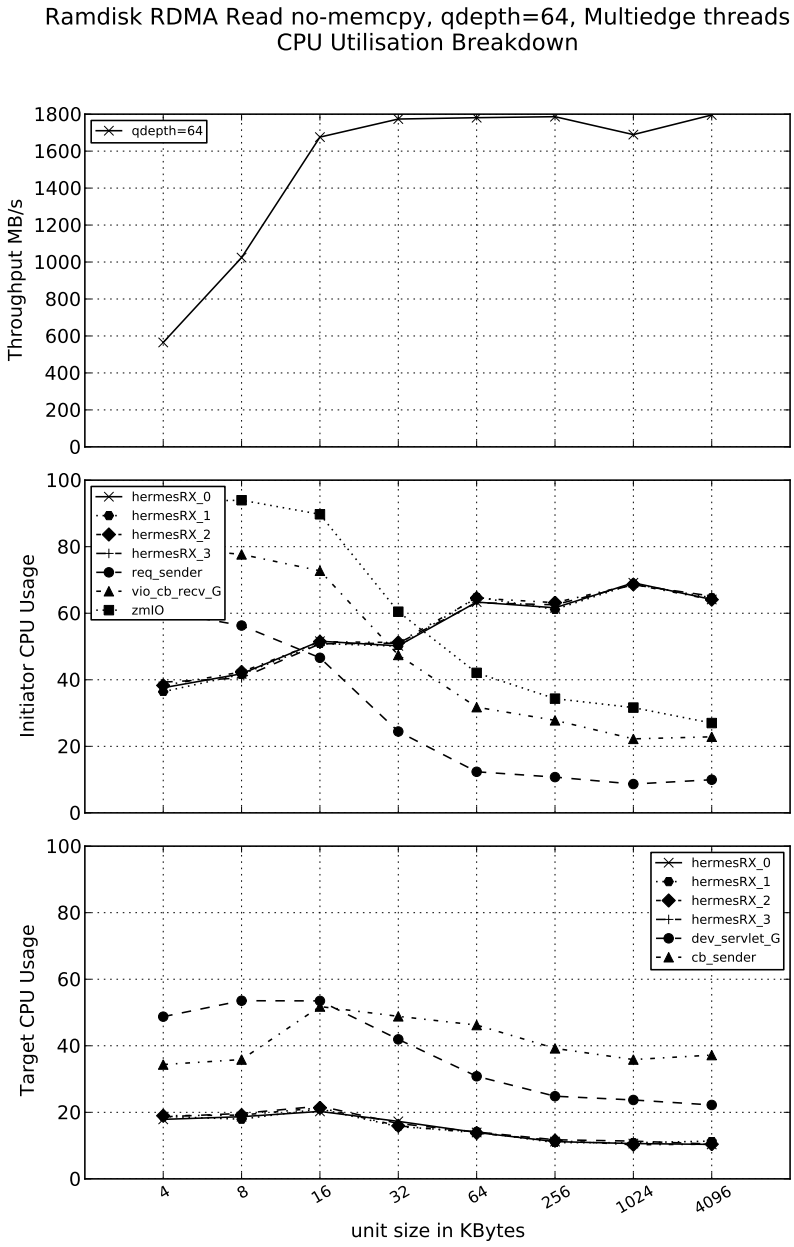
In comparison to the previous graph that referred to measurements with ideal non-realistic iSCSI setup (*null* backend, separate exported targets each accessed through different interface using different connections) the numbers are much in favour of WZBLK. Where iSCSI peaked at around 650 MB/s read or write throughput, WZBLK is able to read at a rate more almost 2 GB/s and write at more than 4 GB/s. Where iSCSI peaked at a maximum of 100 K IOPS, WZBLK exceeded 170 K IOPS. Finally, while iSCSI stopped scaling after 16 outstanding requests, WZBLK kept increasing throughput and IOPS up to 64 outstanding requests. Finally we notice that WZBLK makes use of multiple cores in both Target and Initiator, utilising 300% - 400% CPU under most of the test parameters.

Even in comparison to the first graph that presented local ramdisk measurements, WZBLK shows much higher speeds, since it avoids the RAM-to-RAM copies that the regular ramdisk implies.

It is notable that given a unit size big enough, and high number of outstanding I/O requests, WZBLK is able to saturate the network when *writing* data to the remote block device, reaching the peak of 4 GB/s.

On the other hand, when *reading* from the block device we can see that bandwidth, even though much better than what iSCSI achieves, is still limited at about 2 GB/s. We have explained (see 2.2.3) that *reading* on the Initiator is heavier because of the extra copy of the data happening from the Rx ring to the `struct bio` attached memory pages. However one would expect that using 4 threads would be enough to saturate the network.

The cause was not evident until a specific behaviour was noticed during various stress tests: The 4 receiving Multiedge threads (named *HermesRX*) at the Initiator that perform the necessary memory copies were almost always stuck at 60% CPU usage, never surpassing this number, while it easily reached up to 100% when WZBLK was compiled with only one HermesRX thread (however performance in total was lower for the single-threaded case, as expected). To further clarify the situation, all involved processes and threads were plotted for both Target and Initiator.



These results make it obvious that although memory copying is distributed among cores, the cores are not being utilised to their full potential. While the primary load for the HermesRX threads is copying data (by means of Linux kernel's `copy_page()`) there is a factor that is hindering their execution, preventing them to utilise more than 60% CPU no matter how many outstanding I/Os we set (*the four curves are one on top of the other*). We assume that because of *lock contention* – mostly on the Rx ring, possibly other structures as well – during extremely high transfer rates, those threads are staying interlocked for a significant amount of time. Further investigation and optimisation is necessary.

4 Conclusion

4.1 Summary

We presented the implementation of a network block device in the Linux kernel, optimised to work efficiently over high-speed Ethernet interconnects, that is based on custom reliable transport protocol and is capable of exploiting parallelism over multiple network links as well as multiple CPU cores. Problems inherent to using such commodity hardware were presented, together with the solutions we chose to implement.

In addition, a brief introduction was given to other solutions, and iSCSI was explored as it was the most common

solution in use. Our solution was measured against iSCSI using block-level micro-benchmarks and was shown to be much more performant in both sequential and random I/O workloads.

4.2 Improvement areas, future work

4.2.1 Improving on extra copy

A very interesting field for improvement would be to turn into optional the requirement of attaching the physical pages when submitting an I/O request in the Linux kernel. Today, even when issuing a read I/O request, the `struct bio` argument passed must have already attached the necessary unused physical pages that will be filled with the data when reading is complete. As noted (see 2.2.3), when physical pages are filled from the Ethernet controller in the order that arrive in the network, they must obviously be *copied* to the pages attached to the `struct bio`. By enabling an I/O request to be submitted without attached pages, e.g. with a new `BIO_NO_PAGES` flag, the kernel can choose to return whatever pages the data is in without any copying whatsoever, or allocate the pages itself if zero-copy is impossible.

The complication however is that Linux kernel's block-layer API has been many years like this and introducing deep changes in the API will certainly break a lot of existing code. Nevertheless it is certain that many block-layer drivers and filesystems will benefit from such a feature.

4.2.2 Packing of data with relevant commands

As currently meta-data requests are sent via RDMA operations and data is sent via pagelist operations (see 2.2.6), which are two different calls in the API of Multiedge, they are sent as different Ethernet frames. However meta-data requests are small, and it can easily fit in Ethernet's Jumbo frames of 9000 bytes together with one or two pages of actual data. Such improvement would reduce the machine's IRQ load and the load of the per-NIC receiver threads (HermesRX threads).

4.2.3 Issuing DMA transfers directly between Ethernet controller and storage

WZBLK's zero-copy characteristic means that there are no copies performed (exception 2.2.3) from main memory to main memory. However it is unavoidable that copies will take place from the devices to the main memory, so called data *transfers*. For example when data comes from the network it is copied with the use of DMA to main memory, and to write to disk another DMA is issued that copies the data to the block device.

Such operations are not considered copies traditionally since they are mostly unavoidable and do not impose burden to the CPU due to the use of direct memory access (DMA). However they are a burden to the memory controller, and given the extremely high rates of today's Ethernet interconnects and SSDs it is the case that they may be *limited by the memory bus total bandwidth* capabilities.

Thus, it makes great sense to reduce the data transfers. In particular, since both network and storage controller are connected to the same PCI bus, it *should be possible* to initiate a DMA request from the Ethernet card to the storage controller and vice-versa, avoiding completely the main RAM.

4.3 Acknowledgements

Deep appreciation to FORTH-ICS for providing scholarships that give the opportunity to work in such advanced fields. Greetings to all former, current, and permanent residents of CARV laboratory. Special mention to Manolis Marazakis for devoted "mentoring".

A Appendix

A.1 Terminology

In this document there is frequent use of network layer and block layer terminology. The most important terms are explained below:

RDMA Remote Direct Memory Access is an operation that involves moving copying specific data from main memory of one computer to specific place in the main memory of another computer, without imposing burden to the CPU.

SSD (Solid State Storage) is a type of block device that provides non-volatile memory using flash memory. As opposed to hard disk drives, this technology has no moving parts, resulting to faster access speeds by two or three orders of magnitude in comparison to hard disks.

Initiator is the side that issues the I/O request. Frequently mentioned also as *Client*, on the *Initiator* the user-space (usually) programs run, and issue the I/O requests most often via common `read()` and `write()` operations.

Target or *server* receives the request that the Initiator sent. The actual storage devices are on the *Target*, and it is common that target is part of a high-end NAS or SAN system.

Page is the basic unit of system memory, which in our case (x86 architecture) is of 4 KB size. Of high importance is that it presents the minimal amount of memory that can be used in DMA operations.

Block is the minimal I/O unit in block devices and filesystems. 512 B (i.e. equal to sector size) or 4 KB are the most common block sizes in today's disks. However 4 KB is the most prevalent block size that filesystems present, which handily matches the *page* size on x86 architectures. In our case both *block* and *page* are always referring to 4 KB so there should be no confusion, a design choice aiming to avoid complexity.

zero-copy a characteristic of high performance systems that allows passing memory buffers from/to different components without the actual data being copied.

Rx ring is a circular buffer allocated by the Ethernet adapter driver module. When data arrives from the network, the NIC performs DMA and puts the whole packet in free space of the Rx ring. As such it is of primary importance for this buffer to never be full, as in such case packets will be dropped.

MB/s, GB/s 2^{20}, 2^{30} **Bytes** per second, as opposed to

Mbps, Gbps 10^6, 10^9 **bits** per second.

tgtd is the iSCSI target shipping with Redhat Enterprise Linux 6. It is implemented completely in userspace and comes with several back-ends that differentiate the read/write interface to the kernel, e.g. `rdwr` back-end for common `read()`, `write()` system calls, and `mmap` back-end for mmap'ed I/O. The `null` back-end is equivalent to reading zeros and writing nothing.

top half is the part of the interrupt handler that actually responds to the hardware interrupt, as opposed to the *bottom half* part (or "soft" IRQ) that is scheduled by the top half to do additional processing.

ramdrv.ko is a Linux kernel RAM drive with special performance tuning features, developed at CARV laboratory, FORTH-ICS, as part of the IOLANES [15] EU-funded project.

A.2 Bibliography

[1] *Chelsio Demonstrates Next Generation 40G iSCSI.* URL: http://www.chelsio.com/chelsio-demonstrates-next-generation-40g-iscsi-at-snw-spring/.

[2] Dimitrios Xinidis, Angelos Bilas, and Michail D Flouris. "Performance evaluation of commodity iSCSI-based storage systems". In: *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on.* IEEE. 2005, pp. 261–269.

[3] Maciej Brzezniak, Norbert Meyer, Michail Flouris, and Angelos Bilas. "Evaluation of Custom vs Commodity Technology-based Storage Elements." In: *Grid and Services Evolution.* Springer, 2009, pp. 1–9.

[4] Sven Karlsson, Stavros Passas, George Kotsis, and Angelos Bilas. "Multiedge: An edge-based communication subsystem for scalable commodity servers". In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International.* IEEE. 2007, pp. 1–10.

[5] Stavros Passas, George Kotsis, Sven Karlsson, and Angelos Bilas. "Exploiting spatial parallelism in Ethernet-based cluster interconnects". In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on.* IEEE. 2008, pp. 1–8.

[6] Stavros Passas, Kostas Magoutis, and Angelos Bilas. "Towards 100 gbit/s ethernet: multicore-based parallel communication protocol design". In: *Proceedings of the 23rd international conference on Supercomputing.* ACM. 2009, pp. 214–224.

[7] Michail D Flouris and Angelos Bilas. "Violin: a framework for extensible block-level storage". In: *Mass Storage Systems and Technologies, 2005. Proceedings. 22nd IEEE/13th NASA Goddard Conference on.* IEEE. 2005, pp. 128–142.

[8] Michail D Flouris, Renaud Lachaize, and Angelos Bilas. *Shared & Flexible Block I/O for Cluster-Based Storage.* Tech. rep. 2006.

[9] Michail D Flouris, Renaud Lachaize, Konstantinos Chasapis, and Angelos Bilas. "Extensible block-level storage virtualization in cluster-based systems". In: *Journal of Parallel and Distributed Computing* 70.8 (2010), pp. 800–824.

[10] *NBD, a Network Block Device in the Linux kernel.* URL: http://nbd.sourceforge.net.

[11] *zmIO Linux block layer microbenchmark.* CARV laboratory, FORTH-ICS. URL: http://www.ics.forth.gr/carv/downloads.html.

[12] *Linux AIO kernel API,* `man 2 io_submit`.

[13] Dimitrios Apostolou. *`memstress` – a memory subsystem stress tool.* URL: https://github.com/jimis.

[14] *Red Hat Enterprise Linux 6 – Storage Administration Guide.* URL: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html-single/Storage_Administration_Guide.

[15] *IOLanes project.* Advancing the Scalability and Performance of I/O Subsystems in Multicore Platforms. URL: http://iolanes.eu.