Project N^{o}: 610456

D4.2: VIO specification, Implementation of and low-overhead I/O access model, and thin servers for resource virtualisation

September 30, 2015

Abstract:

This document describes the various techniques and mechanisms to reduce the overhead to specific I/O resources and allow the EUROSERVER system to access data required by modern applications.

Document Manager – John Thomson (ONAPP)	
Author	**Affiliation**
John Thomson	ONAPP
Julian Chesterfield	ONAPP
Michail Flouris	ONAPP
Thanos Makatos	ONAPP
Manolis Marazakis	FORTH
John Velegrakis	FORTH
Dimitris Poulios	FORTH
Rémy Gauguey	CEA
Emil Matus	TUD
Reviewers – For draft version	
Paul Carpenter	BSC
Emil Matus	TUD
Fabien Chaix	FORTH
Nikolaos Chrysos	FORTH
Manolis Marazakis	FORTH
Reviewers – For final version	
Paul Carpenter	BSC
Emil Matus	TUD

Document Id N°:		**Version:**	2.0	**Date:**	30/09/2015

Filename:	EUROSERVER_D4.2_v2.0.docx

Confidentiality

The EUROSERVER Consortium consists of the following partners:

Participant no.	Participant organisation names	Short name	Country
1	Commissariat à l'énergie atomique et aux énergies alternatives	CEA	France
2	STMicroelectronics Grenoble 2 SAS	STGNB 2 SAS	France
3	STMicroelectronics Crolles 2 SAS	STM CROLLES	France
4	STMicroelectronics S.A	STMICROELECTRONICS	France
5	ARM Limited	ARM	United Kingdom
6	Eurotech SPA	EUROTECH	Italy
7	Technische Universitaet Dresden	TUD	Germany
8	Barcelona Supercomputing Center	BSC	Spain
9	Foundation for Research and Technology Hellas	FORTH	Greece
10	Chalmers Tekniska Hoegskola AB	CHALMERS	Sweden
11	ONAPP Limited	ONAPP LIMITED	Gibraltar
12	NEAT Srl	NEAT	Italy

Revision history

Version	Author	Notes
0.01	Rémy Gauguey (CEA)	Initial version prepared
0.02	John Thomson (ONAPP)	(Major) Transferred document ownership. Added draft status.
0.03	Manolis Marazakis (FORTH)	Added text about sockets-over-RDMA and NIC sharing.

Version	Author	Notes
0.04	John Thomson (ONAPP)	Addition of ATAoE + Storage caching. TUD section added. Manolis M changes folded in. Preliminary version of intro + conclusion
0.05	John Thomson (ONAPP)	Preparing for first draft
0.05b-B	Paul Carpenter (BSC)	Internal review
0.05b-BT	Emil Matus (TUD)	Internal review
0.05b-BTF	Fabien Chaix, Nikolaos Chrysos, Manolis Marazakis (FORTH)	Internal review
0.05c	Rémy Gauguey (CEA)	Changes made related to comments
0.06	John Thomson (ONAPP)	Preparation for release
0.07	John Thomson (ONAPP)	Lock down for submission. Submitted to EC before interim review, 19th June 2015
0.08	John Thomson (ONAPP)	Adding changes from TUD for CRAN Adding changes from ONAPP for caching and ATAoE Addressing reviewers comments
0.09	John Thomson (ONAPP)	Included the results for ATAoE
0.99	John Thomson (ONAPP)	Final release for internal reviewers
1.00	John Thomson (ONAPP)	Final version
2.0		Final version for submission

Contents

List of Abbreviations

Term	Definition	Term	Definition
API	Application program interface	NIC	Network interface card
ASIP	(CPU) Application specific instruction set processor	NoC	Network-on-chip
ATA	(Storage) Advanced technology attachment	NUMA	Non-uniform memory access
ATAoE	ATA over Ethernet	NVM	Non-volatile memories
AXI	(ARM tech) Advanced eXtensible Interface	OFA	Open Fabric Alliance
BIOS	Basic input output system	OFED	OpenFabrics Enterprise Distribution
CCI	Cache coherent interconnect	OS	Operating System
CCN	Cache coherent network	PCI	Peripheral component interconnect
CDMA	Central direct memory access	PCIe	PCI express
CPU	Central processing unit	PF	(Network) Physical function
CQ	(InfiniBand) Completion queue	PHY	(Network) Physical layer of OSI
CRAN	Cloud radio access network	PMU	Performance management unit
DDR	(RAM) Double data rate	QoS	Quality of service
DFE	(CRAN) Dataflow runtime engine	QP	(InfiniBand) Queue pair
DMA	Direct memory access	RAM	Random access memory
DRAM	Dynamic random access memory	RDMA	Remote direct memory access
DSP	(CPU) Digital signal processor	RDS	Reliable datagram sockets
FPGA	(CPU) Field programmable grid array	RNIC	RDMA-capable network interface
GIC	Generic interrupt controller	RX	(Network) Receive
HCA	Host channel adapter	SAN	Storage area network
HDD	Hard disk drive	SDP	Sockets direct protocol

Term	Definition	Term	Definition
HMC	(RAM) Hybrid memory cube	skb	Struct sk_buff in Linux (fragmented network packet)
HPC	High-performance computing	SMMU	System memory management unit
ID	Identifier	SoC	System on a Chip
I/O	Input / output	SR-IOV	Single root IO virtualisation
IOMMU	Input / output memory management unit	SRP	SCSI RDMA protocol
IOV	Input / output virtualisation	SSD	(Storage) Solid state drive
IP	Internet protocol	SSD	(RAM) Secure state determination
IPoIB	Internet protocol over InfiniBand	TCO	Total cost of ownership
IS	(ONAPP) Integrated Storage	TCP	Transmission control protocol
iSER	iSCSI extensions for RDMA	TLB	Translation lookaside buffer
KVM	Kernel-based Virtual Machine	TPA	Transport physical address
LUN	Logical unit number	TX	(Network) Transmit
MAC	(Network) Media access control	UDP	(Network) User datagram protocol
MDIO	Management data input/output	ULP	(InfiniBand) Upper layer protocol
MMU	Memory management unit	USB	Universal Serial Bus
MPI	Message passing interface	VBS	Virtual block switch
MR	(InfiniBand) Memory region	VF	(Network) Virtual function
MTU	(Network) Maximum transmission unit	VIO	Virtual input / output
NAND	(RAM) Type of memory using NAND gates	VLAN	Virtual local area network
NFS	Network file system	VM	Virtual machine
NFS-RDMA	Network filesystem over RDMA		

List of Figures

Executive Summary

The EUROSERVER project is paving the way for a novel hyperconverged, next-generation, microserver architecture that will advance the state of the art for both software and hardware in the data center. The efforts are focused around energy efficient platforms that produce more useful computation for less energy. The efforts of Work Package 4 are concentrated on improving the software stack for microserver platforms. In particular, the software looks to create a holistic software stack using low resource, densely integrated nodes and benefitting from the unique aspects of the EUROSERVER architecture such as share-everything that set it apart from existing platforms.

This document contains a summary of the efforts by the EUROSERVER consortium on working towards improved Virtualised Input/Output (VIO) mechanisms at all layers of the software stack. The focus of the efforts has been on reducing contention to resources that are typically bottlenecked in current platform implementations. Access to modern, high bandwidth memories (Hybrid Memory Cubes – HMC) and storage (Solid State Drives – SSD and Non-volatile Memories – NVM) across the backplane has been an area of research in recent years including in EC funded projects such as IOLanes[1]. This has led to the recommendation as captured in the data center requirements document (D2.1) of >1GB/s memory bandwidth per core and that 'future servers, under optimistic scaling assumptions, will need to support up to 500GB/s of I/O throughput'.

To address these issues we have investigated and developed more efficient virtualisation techniques. Specifically in this deliverable we cover improvements to network, storage and memory shared I/O. Given that virtualisation is used by converged architectures such as EUROSERVER we have investigated limitations to current virtualisation approaches and suggested some improvements that can be made that will increase the overall efficiency of the platform. Rather than create a new optimised network layer we have proposed to adopt some of the existing work from the OFED stack that provides the primitives required for the shared memory solution, UNIMEM that is described in more detail in D4.3. The efficient sharing of virtualised network devices is looked into in more detail with modifications performed to both the kernel and the virtual network driver to support higher throughput. For shared storage, the limitations of current network attached storage solutions has also been investigated. To improve the storage performance for small packet sizes that can affect the aggregate performance in a cloud based data center, an implementation of ATA over Ethernet has been investigated with promising results. To validate and test the improved I/O we have started work on mapping one workload that seems promising for exposing the EUROSERVER proposed architecture, which is CRAN. Work has been performed on virtualising the CRAN workload and it is expected that improved virtual I/O that has been worked on in other parts of the deliverable, will present, in combination with the other pieces of work in WP4, an efficient platform for operating such workloads.

The EUROSERVER proposed hardware solution utilises a combination of direct attached, high-bandwidth HMC memories connected to processor cores and high bandwidth interconnects for being able to share I/O resources between cores and off-board. Although there is an improvement to the

[1] http://www.iolanes.eu/

underlying hardware that allows improved I/O performance it is important to support the hardware with the software to determine the locality of resources to reduce the performance inefficiencies of transferring data over long distances. The software mechanisms as proposed in this Deliverable and the other Deliverables within WP4 are therefore an integral part for meeting the objectives of the EUROSERVER Project.

The work being carried out in WP4 is being coordinated into a single software stack to enable integration into the complete EUROSERVER prototype in WP6.

Regarding deviations from the planned work

- A software only approach for network I/O virtualisation on 10GbE was dropped in favour of porting OFED to the EUROSERVER platform.

1. Introduction

EUROSERVER is a holistic project aimed at providing Europe with leading technology for capitalising on the new opportunities required by the new markets of cloud computing. Benefitting from Europe's leading position in low-power, energy efficient, embedded hardware designs and implementations, EUROSERVER provides the next generation of software and hardware required for next-generation data centers. The efforts of Work Package 4 are focused on improving the software stack for microserver platforms. In particular, the software looks to create a holistic software stack using low resource nodes and benefitting from the unique aspects of the EUROSERVER architecture. Instead of utilising a small number of 'fat' cores the EUROSERVER software platform will support multiple 'thin' cores that scale our and share resources. To enable support of such an architecture we propose fundamentally different concepts, tools and techniques. These go some way towards addressing some of the scalability issues that arise as the number of nodes in the platform increases. Virtualisation has led to massive consolidation of software workloads on servers. As this consolidation continues, the overheads of existing solutions need to be mitigated through optimised resource sharing.

Another part of the share-all vision of EUROSERVER is that all the resources should be shared between all microservers that might need those resources, subject to fair access control mechanisms and accounting of resource usage.

As shown in Figure 1, components and techniques such as UNIMEM, the MicroVisor, shared and optimised I/O, optimised virtualisation techniques and orchestration and energy aware orchestration tools address some of the scalability issues that arise as the number of nodes in the platform increases. The work being done in WP4 is being coordinated into a single software stack to enable integration into the complete EUROSERVER prototype in WP6.

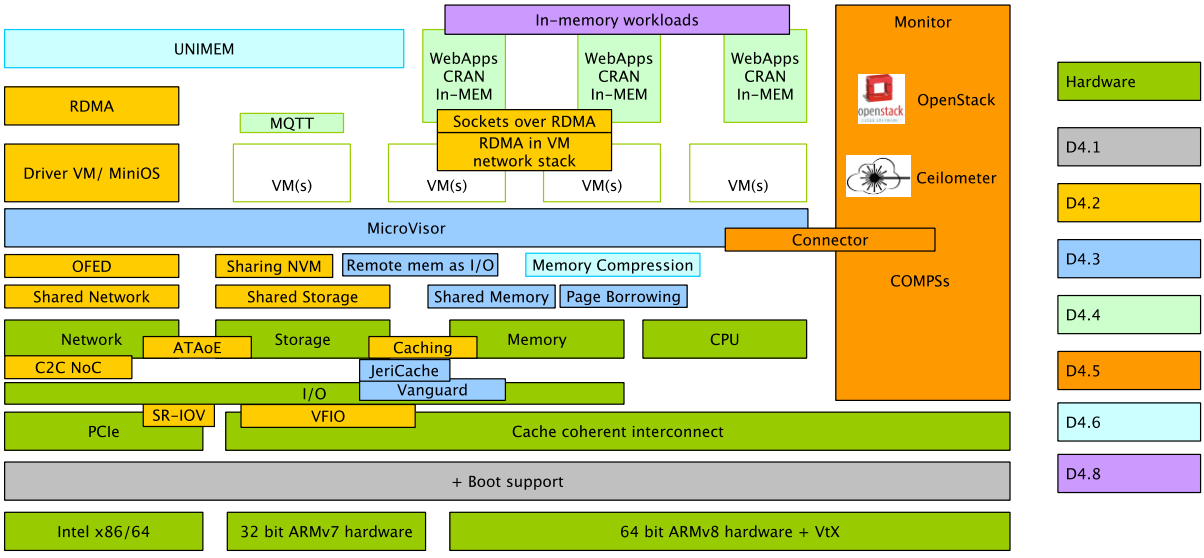


Figure 1: Diagram showing the areas being worked on in EUROSERVER in WP4

In this Deliverable, the authors seek to investigate whether various layers of the software stack can be improved to generally increase the virtualisation performance for I/O resources. The work has been carried out with the proposed EUROSERVER Final Prototype as the target platform.

Improvements in the software stack as described are a mixture of platform specific technologies as well as more generic technologies that can be applied to other platform types. The MicroVisor is an example of one of the more generic technologies that is expected to work on x86 and ARM-based hardware. The key concept in the MicroVisor technology is to remove the control domain stack that incurs high overheads for resource constrained micro-servers. It is covered in part in this Deliverable with the full description in D4.3. Building up from a more efficient hypervisor platform there have been efforts in improving the I/O performance for RAM, storage and network I/O.

In contrast to global I/O Ethernet links, chiplet-to-chiplet communication will benefit from UNIMEM as proposed in the EUROSERVER architecture. The EUROSERVER platform will also rely on IOMMUs to implement an RDMA protocol, and could then bypass the legacy TCP/IP stack that is not efficient with respect to latencies. This work on chiplet-to-chiplet communication path is closely related to the pending and patented hardware block developed by CEA to optimise the usage of IOMMUs.

To improve the performance, efforts have been carried out on the hypervisor platform (Section 2) up through to the Network (Section 3) and Storage virtualisation (Section 4) improvements. There have also been improvements on I/O virtualisation (Section 5) in general. Finally we conclude this Deliverable by describing the work that has gone into improving support for the virtualisation platforms. Specifically we look at how virtualisation performance can be assisted through the use of hardware drivers contained in specific, specialised domains as well as looking at improving the performance of Cloud-RAN applications (Section 7).

2. MicroVisor platform – for efficient resource virtualisation (ONAPP)

The MicroVisor platform is an innovative hypervisor technology for microserver hardware architectures, including the EUROSERVER proposed architecture. The platform has been designed to work with resource-constrained, scalable, low-power, energy-efficient servers. The platform is based on Xen and looks to remove any dependency on a local control and management domain (known in Xen as dom0) for generic functionality and instead promote this functionality into the Xen hypervisor layer directly. This helps to reduce the management overhead that has grown to accommodate the functionality required of large cloud deployments based on powerful, cache-coherent CPU cores. Cache coherency though is reaching its limit, with workarounds such as NUMA being used to allow fast access of regions of memory to individual sockets whilst also maintaining access to the global memory through slower remote accesses.

Hardware driver support is maintained via a driver domain interface, the current implementation being device drivers in MiniOS instances. This is described in more detail in Section 6.2 of this deliverable. By moving the hardware interaction responsibility to instances that can be spawned as

helper VMs we allow for the number of devices to scale by spawning new instances as the load on the existing hardware increases beyond a threshold.

Although the MicroVisor platform improves various aspects of storage, network and general I/O, the main detail is described in D4.3. In this Deliverable we describe the MiniOS framework that is used for supporting hardware drivers and helping to reduce the number of context switches required in the MicroVisor platform. This work is described in Section 6.2.

3. Network virtualisation (CEA/FORTH/ONAPP)

At the heart of any modern computing platform that is based on multiple CPUs and cores is the underlying communication network. Inter-processor communication has been handled with various architectures and approaches. These have normally been designed to work at near if not full bandwidth between the components. Bus speeds and connections to other resource components have typically been the restriction for performance for generic cloud type workloads. Where the hardware architecture is uniform and well known there are specific optimisations that can be and are typically carried out for high-performance computing (HPC) type workloads. Remote memory access speeds are becoming more of an issue for non-specialised systems, as NUMA type systems are being developed with the workloads possibly not being NUMA-aware. The performance impact for remote memory accesses is becoming more of an important issue and as such vendors are now producing NUMA aware systems that can localise a workload's RAM placement to the same NUMA-set. Network virtualisation also has the issue that aside from being integral for communication there are a lot of packets that all require context switches and as such when the TCP/IP stack is used for elements such as storage transfer there is a high overhead for processing such packets resulting in decreased efficiency.

Deviation from planned work

Related to Subtask 4.2.1 of the DoW: *Without hardware assisted I/O virtualisation, the hypervisor has to perform a pre-packet analysis in order to dispatch incoming network packets to the correct VM, perform a memory copy to appropriate VM address space to ensure isolation, and optionally perform some minimal QoS shaping to ensure correct resource sharing. During T4.1 CEA performed some profiling of such software-only approach on a Cortex-A15 CPU (with virtualisation extensions) running KVM and associated IOV framework "virtio-net".*

Observations lead us to the conclusion that a software only approach would require at least a x10 performance improvement in order to be efficient on a 10GbE interface, which, however, would still require a full EUROSERVER CPU core. Instead of trying to optimise or to offload the Linux network stack to improve any IOV para-virtualisation software approach, CEA focused on using next generation EUROSERVER IOV acceleration hardware blocks such as IOMMU especially in the context of inter chiplet communication path. In this context, CEA decided to port the standard OFED (Open Fabric Enterprise Distribution) RDMA software stack from the Open Fabric Alliance (OFA) on top of UNIMEM/NoC/sMMU architecture and then inherit many upper layers, middlewares and libraries widely deployed among high-performance computing (HPC) and Cloud computing distributed applications. This is described in more detail in Section 5.3.

3.1. Network virtualisation software techniques and limitations (CEA)

The challenge in network virtualisation is to provide efficient, shared, and protected access to the network interface. In order to achieve these upper three goals, two I/O virtualisation technologies are currently deployed on modern systems:

- Software techniques like I/O para-virtualisation (ex: KVM Virtio [4][8]) already covered in Deliverable D4.1 "Architecture Software Support"
- Hardware assisted techniques like Direct Device Assignment, (PCI pass-through or SR-IOV [1]), where each VM has a direct access to a hardware device, bypassing the host software on the I/O path.

Experiments described in D4.1 and performed on an Arndale board (dual ARM Cortex-A15 SoC running at 1GHz) and its USB3 to 1Gbps Ethernet adapter clearly highlighted the limitations of such a software only approach.
Indeed, in any software I/O virtualisation technology, the hypervisor needs to:

- Perform additional network packet analysis and filtering to identify the targeted VM
- Perform memory copy of every packet to and from the VM address space, since only the hypervisor can access the device driver and the system physical memory (this is sometimes called shadow translation tables or bounce buffering)
- Emulate or forward the interrupts to the appropriate VM

In addition, each of those hypervisor interventions will cause a "VM exit", i.e. a VM to hypervisor software context switch.

The results in D4.1 results showed that even with a relatively "low bandwidth" 1GbE link and a state-of-the-art paravirtual I/O implementation (KVM virtio and vhost-net infrastructure), the CPU could quickly saturate and TCP/IP performance drop by 20 to 30%.

It becomes obvious that such software IOV approaches cannot fit the requirement of high speed interconnects like the 80Gpbs inter-chiplet serial link, or even any 10Gpbs Ethernet that will be used in the next EUROSERVER microserver.

For this reason, we consider that, if software optimisations of such an approach cannot reach one order of magnitude performance gain, any software only I/O virtualisation becomes irrelevant regarding the targeted EUROSERVER interconnect capacity.

As described in Deliverable D3.3 "Next Generation EUROSERVER System Architecture":
The EUROSERVER system architecture must avoid the complexity and poor performance (see D4.1) of shadow translation tables (software IOV approach). For this reason, the hardware must have full hardware virtualisation support, including one or more IOMMUs. In Section 6, we will explain how the software can take benefit of this full hardware virtualisation support.

3.2. Sharing a network interface among nodes (FORTH)

With increasing core counts in tight packaging, as envisioned in the EUROSERVER project, the physical placement and management of I/O resources (esp. network interfaces and storage

controllers and devices) becomes a great concern and a challenge for hardware and software designers, as it is not cost-effective, and soon not even feasible, to dedicate separate I/O resources to each compute node. Instead, expensive and high-performance I/O resources are distributed among groups of compute nodes, necessitating efficient sharing arrangements, which require both hardware and software support.

FORTH has implemented a Linux kernel driver that allows sharing of a high-speed network interface (10 Gbps Ethernet NIC, optical transceiver). Virtualisation of the network interface and the multiplexing of network traffic from/to different Compute Nodes are required for efficient and secure usage of the interface. Sharing of a network interface is particularly important for scale-out workloads, such as web services, where many Compute Nodes have the same entry/exit point to the Internet. By installing this NIC driver at each of the compute nodes, each compute node is presented with a (virtual) L2 networking device that is compatible with the Linux kernel's TCP/IP stack. Applications using the popular Berkeley Sockets API can run unmodified on the nodes and communicate with the external network. Figure 2 presents a flow-chart of send/receive operations initiated by user-space processes that use the Berkeley Sockets API and the software and hardware components involved. A socket send() library call causes a system call trap, which gives control to the Linux network stack (implemented entirely in kernel-space). For outbound network traffic, the data produced by application processes are processed by each TCP/IP layer, until the kernel calls the driver's transmit function hard_start_xmit(). At this point, the driver that manages the underlying hardware initiates a DMA transfer to the network device. The receive path, for the inbound network traffic, follows a similar flow.

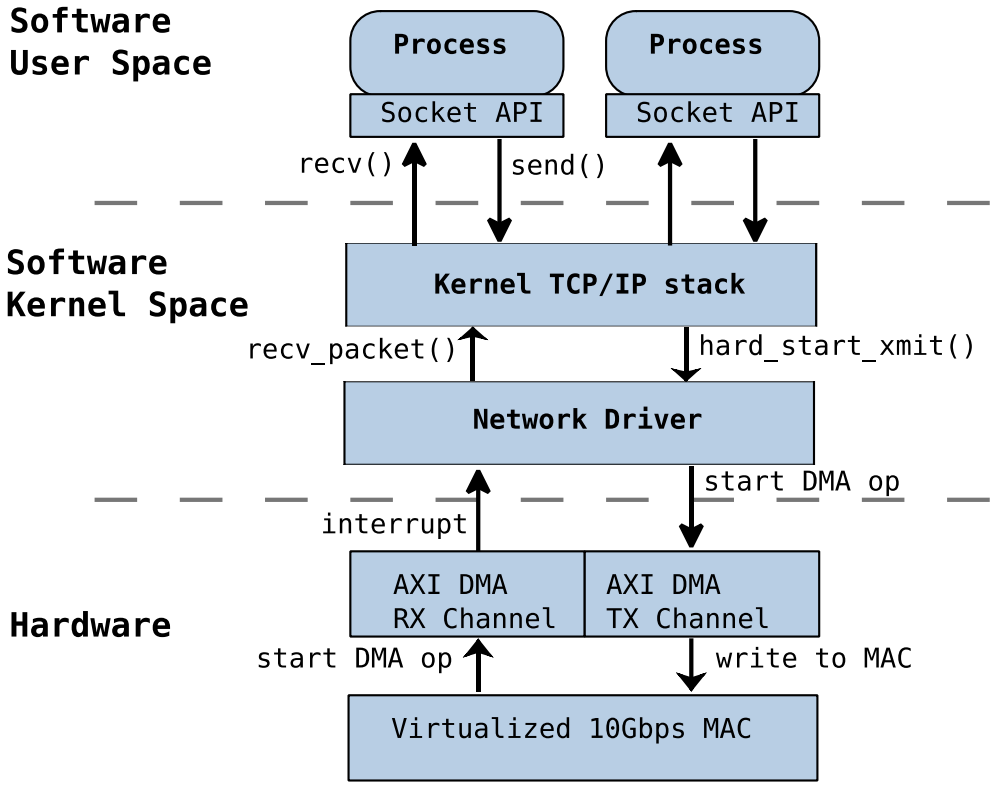


Figure 2: Network traffic send/recv flow chart in Linux.

Packet processing in the kernel is done with the use of the sk_buff (socket buffer) structure (skb for short), which is depicted in Figure 3. The sk_buff structure contains pointers to various segments of the packet for fast access. As each layer takes control of the packet processing inside the kernel stack, it uses the corresponding pointer from the sk_buff structure to add/modify segments of the packet. The common case is for every layer to add its corresponding header and some checksum. The Payload is the segment of the packet that contains actual user space application data.

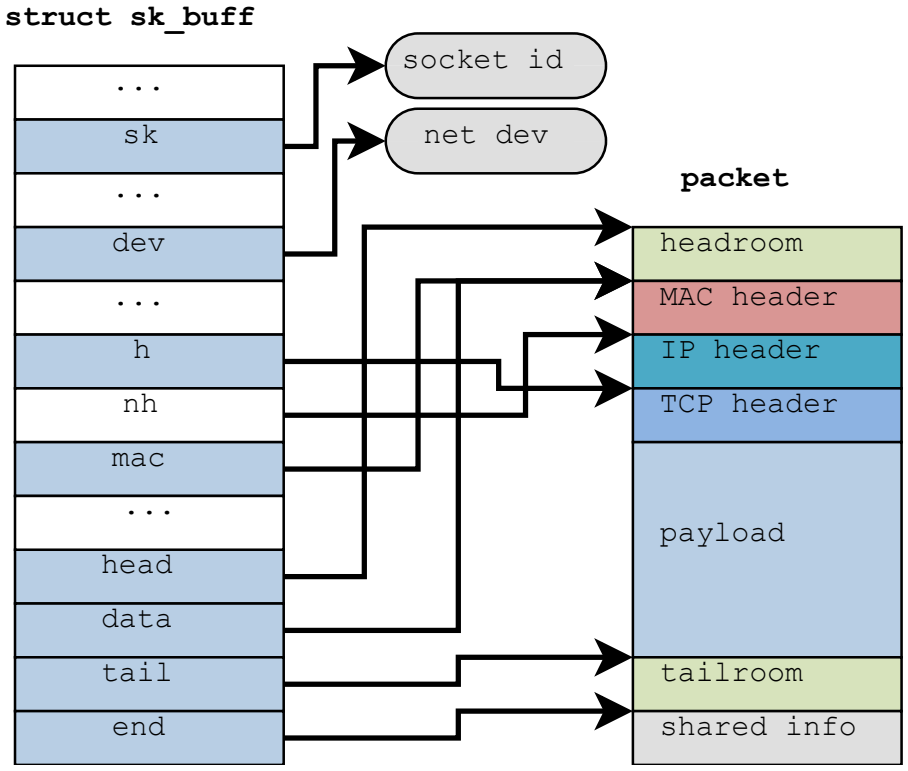


Figure 3: The Linux kernel's sk_buff structure.

Efficient network drivers work on packets in scatter/gather mode, allowing packets to be split into several fragments that reside in different memory pages. This arrangement allows the kernel to pass fragmented packets to the network driver without having to marshal their contents first, thus avoiding memory copy operations. The common case in scatter/gather operation is for the kernel network stack to place the headers of different layers into different memory pages as the packet travels through the layer stack and the layer headers are created. The sk_buff structure can describe this fragmentation of packets, because it includes pointers to each of the fragments of a packet, as shown in Figure 4.

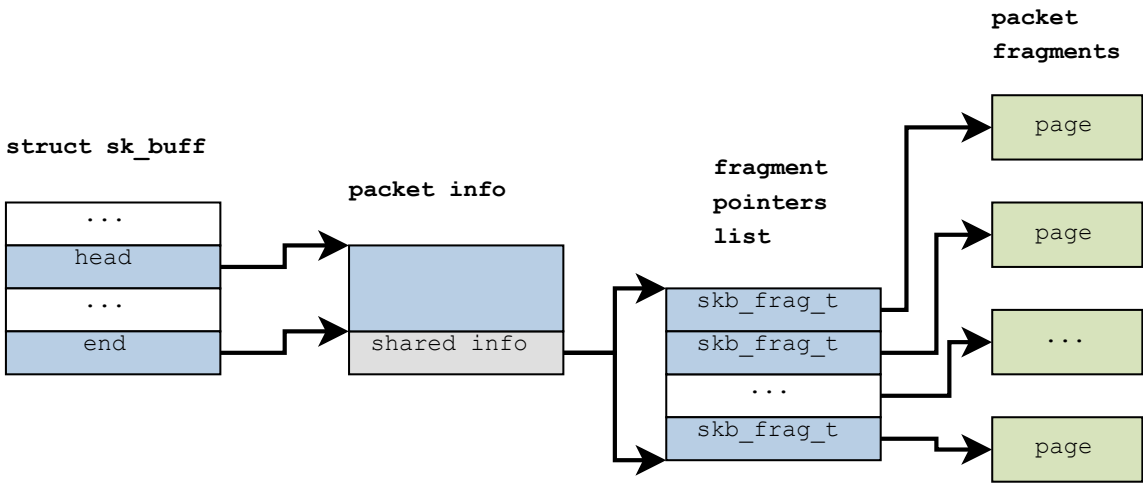


Figure 4: sk_buff describing a fragmented packet.

Driver data structures

In our implementation on the 32-bit discrete prototype of EUROSERVER, we used the AXI DMA hardware block to transmit and receive data to/from the 10 Gbps MAC block. The AXI DMA resides in the FPGA of the Compute Node, whereas the 10 Gbps MAC resides in the FPGA of the central board. The DMA engine is configured to operate in scatter-gather mode. It can accept a linked list of descriptors, with a head and tail pointer. Once the linked list has been properly set up, the head and tail descriptors physical addresses are given as arguments to the DMA. Then, the DMA will start fetching all packet fragments by reading each descriptor in the linked-list. Each such descriptor has a length of sixteen 32-bit words (64 bytes) and it contains several pointers and control registers describing the packet fragments. The following work-length fields in the DMA descriptor are used by our driver:

- next desc phys addr: physical address of the next descriptor (allows the driver to maintain a linked list of related descriptors)
- frag phys addr: physical address of the packet fragment.
- ctrl: setting for the DMA control register for the specific fragment, containing the size of the fragment and some additional tags.
- status: written by the DMA engine with either a success or error indication upon transfer completion
- app4: DMA user field, not used by the hardware, but used by the driver to keep the virtual address of the sk_buff that describes the series of fragments making up a packet.

Note the physical addresses stored in the fragment descriptors must each be in a DMA-capable memory region, i.e. the DMA engine must be able to read/write in burst-mode from/to those memory addresses. Such memory regions are made available to the driver by the kernel, via calls to the kernel's dma_map_single() function.

Our driver maintains two circular arrays of DMA descriptors, called the descriptors rings. The transmit (TX) path has 64 such descriptors in its ring, while the receive (RX) path has 128. Furthermore, we maintain a head and tail pointer for each ring, to keep track of used and free

descriptors. This implementation allows simultaneous access by the hardware (CPU and DMA engine) and the software to the descriptor rings and their corresponding fragments in physical memory.

Driver Initialisation

When the driver module is loaded by the kernel it sets up the descriptors for the TX/RX path and configures the DMA engine. For the TX path, an array of 64 descriptors, each of 64 bytes, is allocated by calling the kernel's dma_alloc_coherent() function, which returns the physical and virtual addresses of the allocated space. The next desc phys addr field in each descriptor is set up in such a way to create a circular array (ring), with the last descriptor pointing back to the first in the array. The RX descriptors ring is set up in a similar fashion. For the RX path, a similar memory region is allocated, to enable the DMA engine to store the incoming frames. For each descriptor in the RX ring we therefore allocate a buffer with size equal to the network's maximum transmission unit (MTU), using the kernel's netdev_alloc_skb_ip_align() function, which allocates and prepares space for an skb structure. The virtual addresses of the allocated skb structures are kept in the sw_id_offset field of their corresponding descriptors. Finally, a DMA-capable physical address is requested for each allocated skb structure, using the kernel's dma_map_single() function. That physical address is stored in the frag phys addr field of the corresponding descriptor.

Finally, the DMA engine is activated upon setting the TX/RX head pointers. The RX DMA channel starts running and is ready to accept frames. The TX DMA channel is waiting to transmit when its register for the DMA descriptor tail pointer is written.

Transmitting a Frame

When a whole or fragmented frame is ready to be sent by the network stack, the kernel calls the driver's hard_start_xmit() function with the corresponding sk_buff pointer as an argument. This function finds out the number of fragments in the packet and it allocates that many free descriptors in the TX ring. If enough descriptors are available, it sets their fields to point to the corresponding fragments. Finally, it marks the first and last fragments with START_OF_FRAME and END_OF_FRAME respectively. If the frame consists of only one fragment, then its descriptor is marked with both START_OF_FRAME and END_OF_FRAME. The driver then updates the tail pointer accordingly, and writes it to the DMA engine's TX TAIL register. Upon this write, the DMA engine starts fetching descriptors and their corresponding fragments for transmission.

When the DMA engine has finished transmitting the frame, it will raise an interrupt to the CPU. The driver's TX interrupt handler will run, check for DMA errors in each transmitted descriptors' status field and finally release all the used descriptors, by updating the head pointer. If an error has occurred, it will schedule a task that will later reset the DMA engine and re-initialise the TX descriptors ring. The flow chart of a transmit operation from the user-space process to the hardware (including interrupt generation) is shown in Figure 5.

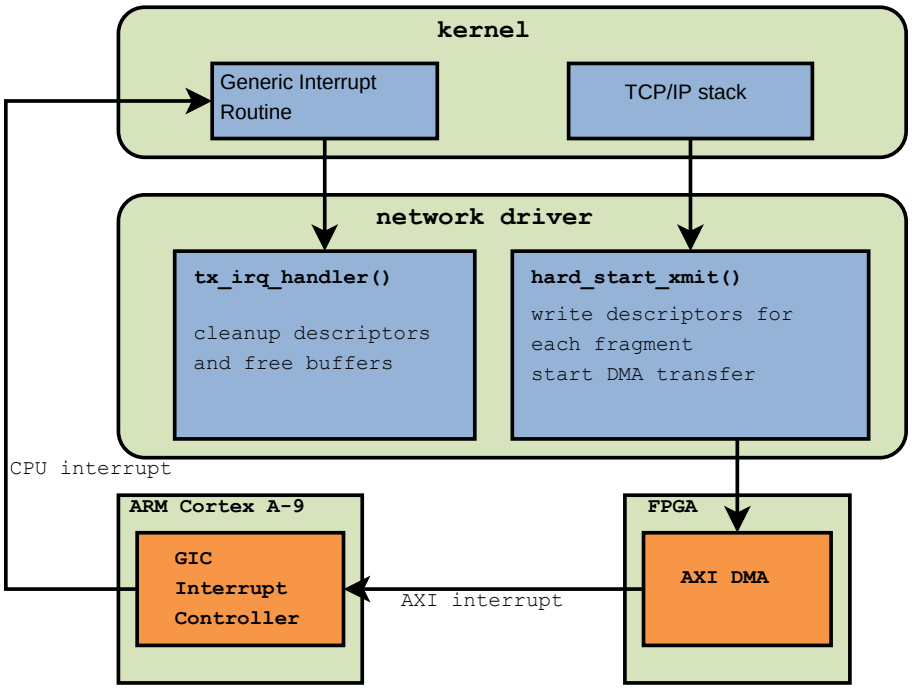


Figure 5: Flow chart for the TX path.

Figure 6 depicts the descriptors ring containing the fragment pointers. The head pointer points to the oldest (first) descriptor of the fragment subset that may still have to be processed by the DMA, from the point of view of the CPU, while the tail pointer points to the newest (last) fragment. In this figure, curr is a register in the DMA engine that points to the descriptor currently being processed by the DMA – this register is updated automatically by the hardware. The head pointer is updated by the driver's transmit interrupt handler, while the tail pointer is updated by the kernel's hard_start_xmit() function.

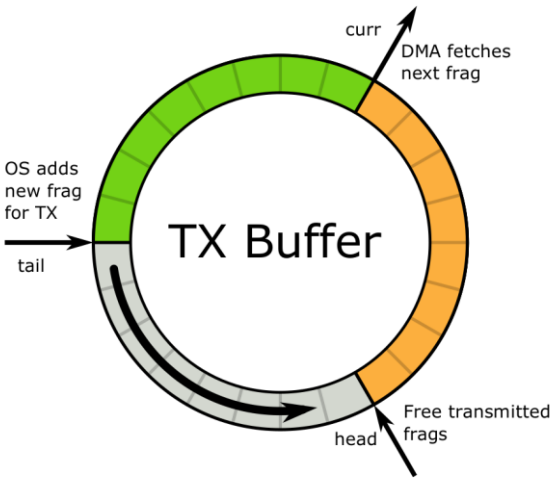


Figure 6: View of the TX descriptor ring in operation – source adapted from Wikimedia[2]

[2] https://commons.wikimedia.org/wiki/File:Ring_buffer.svg

The following subsets of descriptors (as delineated by the head, tail, and curr pointers) can be present in the TX ring at the same time:

- head – curr: descriptors for fragments that have been transmitted by the DMA engine, but have not yet been released by the driver.
- curr – tail: descriptors for fragments that have been submitted to the DMA engine, but have not yet been transmitted.
- tail – head: free descriptors, available for new transmission requests by the kernel's hard_start_xmit() function.

The function hard_start_xmit() takes the following steps when called by the kernel:

1. Finds out how many fragments make up the frame (described by a sk_buff structure) to be transmitted.
2. Checks if there are a sufficient number of free descriptors in the TX ring.
3. Reads the tail pointer and starts setting up descriptors for each frame fragment.
    a. In the frag_phys_addr field, it sets the physical address of each fragment as obtained by a call to dma_map_single().
    b. Stores the size of the fragment in the ctrl field, and the virtual address of the sk_buff structure in the app4 field, to remember the sk_buff of each fragment.
    c. For each descriptor completed, it advances the tail pointer (in a circular fashion: tail modulo NUMBER_OF_DESCRIPTORS).
4. Marks the first descriptor of the frame as START_OF_FRAME, and the last one as END_OF_FRAME.
5. Starts the DMA transfer, by writing the physical address of the new tail descriptor to the appropriate DMA register (The DMA engine starts operation automatically upon write on this register).

Receiving a frame is handled in a similar manner to that of sending a frame. To avoid repetition of familiar elements and processes the reader may choose to read the receive frame section in the Appendix (see Section 0).

Checksum offloading

In Linux it is required to have hardware support for TCP and IP header checksums, in order to enable scatter/gather operation. This policy of the Linux kernel ensures that the scatter/gather operation will not be slowed down by the calculation of TCP and IP header checksums in the network stack. FORTH is developing a TCP and IP header checksum hardware block that will reside in the I/O FPGA of the 32-bit discrete prototype. This hardware block was not available at the time of preparing this document. Therefore, we had to find a workaround in order to enable scatter/gather operation and to also have the kernel calculate the TCP and IP header checksums. This was achieved by enabling two flags that are conflicting according to the Linux network driver implementation guidelines. The first flag enables the scatter/gather operation (tells the kernel that the driver can handle fragmented frames) and the latter enables checksum calculation in the kernel's network stack. Enabling both of

those flags is not an issue in the Linux 3.12.0 kernel version, but we are not sure whether this will apply to other kernel versions. We will use the hardware checksum block in the near future.

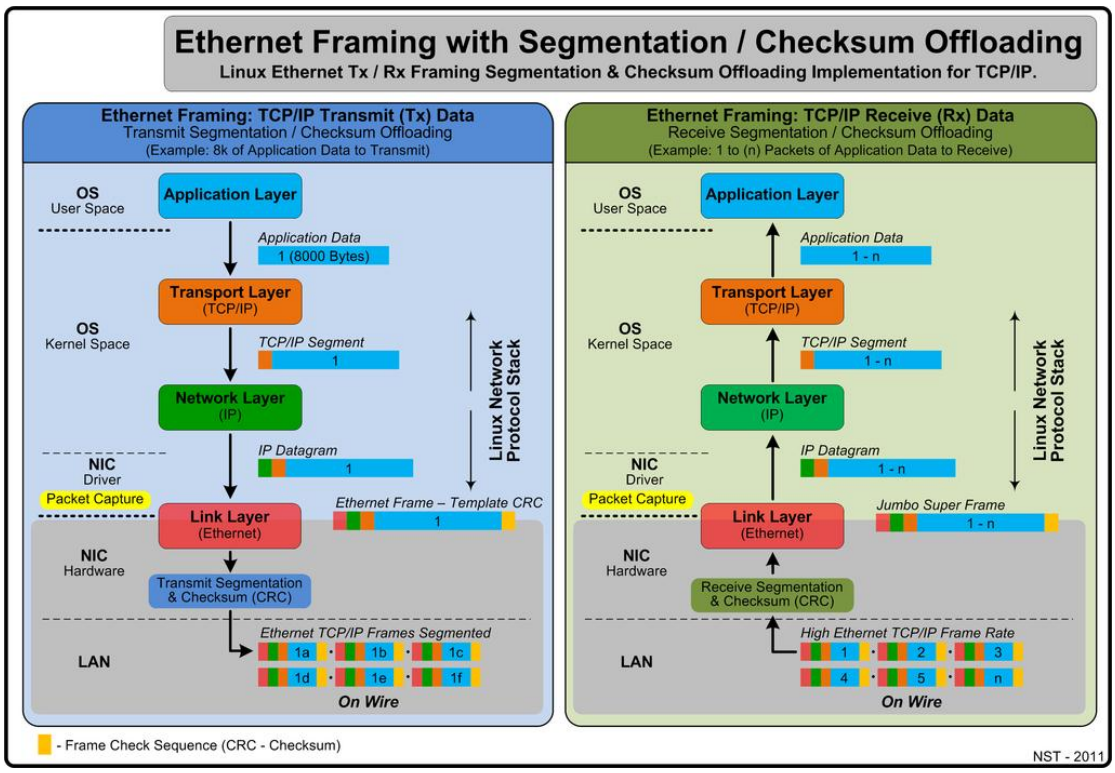


Figure 7: TX and RX flow with hardware offload for checksum computations – source networksecuritytoolkit[3]

Figure 7 shows the TCP transmission and reception flows when using checksum and offloading. Segmentation offloading can further improve performance, because it allows the kernel network stack to bypass many steps in the TCP layer that perform checks and segment the packet if necessary. TCP packet segmentation is a costly process, because the kernel must correctly split the packet into segments with appropriate headers, sequence numbers, etc.

Interrupt Coalescing

Interrupt handling by the kernel is a high-overhead operation (roughly 2 µs for a 'trivial' interrupt handler in our prototype) that can saturate CPU utilisation at increased incoming/outgoing frame rates. When the CPU utilisation is high enough, packet drops occur and throughput is limited.

Modern network drivers and devices, support multiple interrupt rates. For example, a network device can be configured to raise interrupts every ten frame arrivals, as opposed to interrupting on every frame arrival. This leads to a reduced (amortised) interrupt service time per frame, and most importantly, it increases throughput, because of the reduced CPU utilisation at a given frame rate. This method is called Interrupt Coalescing, because it groups events of many frame arrivals into a single interrupt. Of course, this comes at the cost of increased response latency for both TX/RX paths.

[3] http://wiki.networksecuritytoolkit.org/

To mitigate this effect, the hardware blocks employ a timer, and they also raise an interrupt if it expires.

Our network driver and the underlying hardware blocks support interrupt coalescing. The transmit, receive and interrupt handler functions are implemented in such a way that they can operate with all interrupt coalescing settings, i.e. from one interrupt per frame to one interrupt per many frames. The AXI DMA engine that we use can be configured to a required interrupt coalescing threshold. Moreover, a timer-based limit can also be configured.

Management features of the virtual network driver

Our network driver implements a set of functions and exports them to the user space for usage by the ethtool utility. Supporting those features is a standard in modern network drivers, since one can access and configure many settings of the network device's MAC and PHY blocks and the network driver. Our implementation involves access of the 10 Gbps MAC and PHY hardware blocks. The custom underlying hardware interconnect, together with the Physical Address Translation block, enables the software of a Compute Node to access the 10 Gbps MAC layer that resides in the main board's FPGA, since it is mapped in to the node's physical memory address space.

To expose the MAC configuration and statistics registers to the user space environment, we used the Linux procfs file system (/proc). For each MAC register, the network driver creates a procfs entry that can be either writable or non-writable. When a user space process reads or writes from/to that procfs entry (just like an ordinary file), the appropriate method of our network driver is called, that either reads the requested register from the MAC block and returns its value, or writes the register with the data given by the process. All the procfs directories and files for the MAC registers are created when the driver is loaded.

MDIO (Management Data Input/Output) is a serial bus interface and protocol that is used in modern network devices to transfer management information between the MAC and the PHY hardware blocks in modern network devices. With our hardware, software running on a Compute Node can access the registers of the network device that contain the information of the MDIO. Our driver can access those registers and expose them to the user space, using the procfs file system. One can view information or configure the PHY hardware block by reading or writing the appropriate bits of the according MDIO registers.

The 10 Gigabit Media Independent Interface (XGMII) is a standard defined in IEEE 802.3 for connecting full duplex 10 Gigabit Ethernet (10GbE) physical blocks between them and to other electronic devices on a printed circuit board. The XGMII is exposed to the Compute Nodes in our Discrete Prototype, through the MDIO interface.

The network driver implements methods for reading and writing to MDIO registers that result in reading or writing raw byte data from/to the PHY block through the XGMII. Writing raw data to the PHY eventually results in the actual physical transmission over the 10Gbps optical cable. Those read/write methods are exposed to the user space through the procfs system and are useful for the debugging of the hardware prototype.

During its operation, the network driver collects statistics about the IP and Ethernet traffic that passes through it. It also counts the numbers of dropped packets and transmission reception errors that have occurred. Update of those counters is done in the transmit and receive functions and interrupt handlers. Those statistics can be obtained via the standard ifconfig utility.

To load the network driver and map the DMA and MAC register space into physical memory, the kernel needs to know about them at boot time. In ARMv7 architectures, an appropriate Device Tree file must be used, that replaces the function of BIOS in traditional x86 architectures. More details about how to load drivers for the ARM Device Tree are given in the Appendix (see Section I.b).

Performance Evaluation of the Shared Ethernet NIC

The following figure outlines the organisation of the 32-bit discrete prototype, as used in a series of evaluation experiments. The discrete prototype, with four compute nodes, is connected back-to-back to an x86-based network endpoint (Intel Core i7 CPU, 8GB of DDR3 memory, PCIe NIC, running Xubuntu Linux v.14.04), via a 10Gbps-capable optical fibre link.

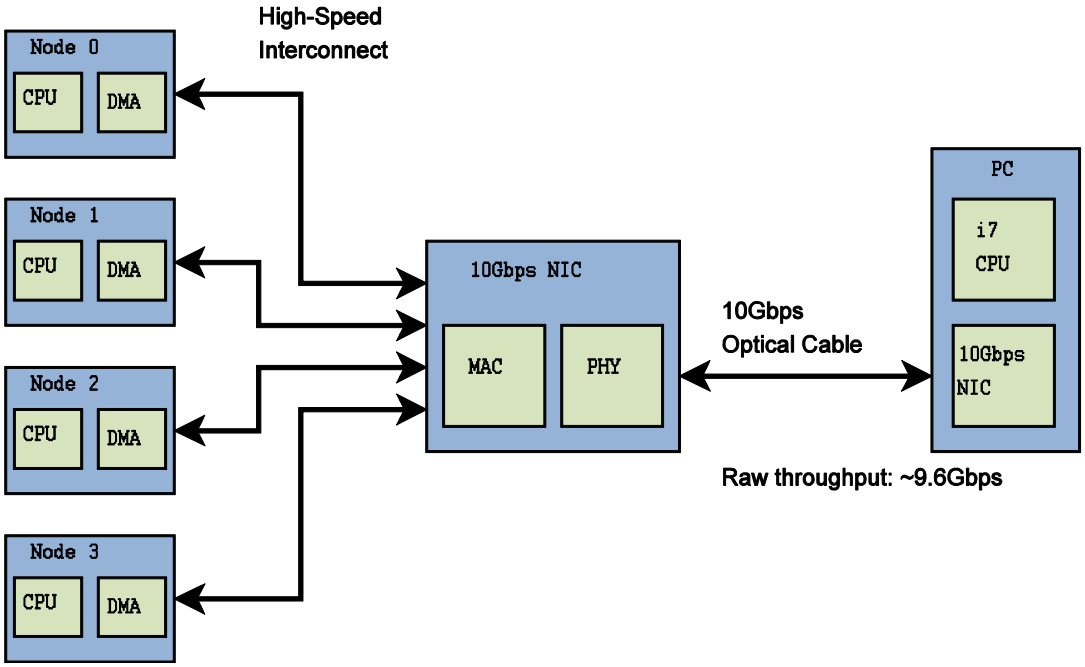


Figure 8: Experimental testbed for performance evaluation of NIC sharing.

To test whether we can reach the throughput limit of the 10Gbps NIC, we first implemented a bare metal application that runs in each Compute Node and uses the AXI DMA engines in the nodes in scatter/gather configuration. This mode allows the AXI DMA to read/write ring buffers without stopping its operation. When used for transmission, it reads and transmits the data in the ring. By running this experiment, we confirmed that we can get very close to the theoretical throughput limit, by achieving 9.8 Gbps total throughput when all Compute Nodes generate traffic (blue line in Figure 9). The throughput was equally shared among the Compute Nodes, because of the round-robin TX scheduler in the MAC hardware block. Each node contributes around 2.2 Gbps of throughput. When only one node is active, the maximum throughput can achieve is 3.2 Gbps. The bottleneck is mostly

due to the CPU: A compute node cannot generate a higher volume of packets per unit time, so traffic from only a single node cannot saturate the 10 Gbps link. This is a demonstration of the value that I/O resource sharing is capable of bringing to a modern microserver design.

The most important evaluation result from the point of view of application software is the throughput achievable by user space applications using the TCP protocol, which is the most common for network communications. We measured the TCP throughput achieved by a user space process in a full system with Linux kernel-space and user-space environment in each node, using the standard iperf utility measuring TCP throughput.

First we measured the maximum transmission throughput achievable by a node. An iperf server is set up on a host PC listening for incoming TCP connections. Then at a node we set up an iperf client that connects to the server and transmits process payload using TCP sockets (STREAM). The maximum usable TCP throughput achieved by each node is 960 Mbps, for large packet sizes that are close to the Ethernet MTU size. During transmission, the CPU utilisation was always at 150% (as shown by the top utility), meaning that both CPU cores of the compute node were utilised. In a similar manner, we measured the maximum achievable node receive throughput by setting up an iperf server on a node and an iperf client on the PC. The result was 880 Mbps for packet sizes close to the Ethernet MTU. In these measurements, CPU utilisation was also high (150%).

Using all four nodes together, each one transmitting TCP packets, the aggregate throughput seen at the 10 Gbps NIC is the sum of the individual throughputs achieved by the nodes: The aggregate throughput is 4 x 880 Mbps = 3.52 Gbps. The sum of the individual throughputs is also confirmed when all four nodes receive packets.

Figure 9 shows the aggregated TCP throughput as a red line. The horizontal axis shows the number of nodes generating network traffic and the vertical axis is the aggregated throughput achieved in Gbps for that number of nodes. We see that when only one node is engaged in network transactions, the aggregated throughput is 880 Mbps, which is the maximum TCP throughput obtainable by a single node. When two nodes generate traffic, the aggregated throughput is 2×880 Mbps = 1.76 Gbps and when three nodes generate traffic the aggregated throughput is 3×880 Mbps = 2.64 Gbps. Finally, when four nodes generate network traffic, the aggregated throughput is 4×880 Mbps = 3.52 Gbps. We can see that each time a node is added in the experiment it adds its maximum TCP throughput to the aggregated throughput. The aggregated TCP throughput with four nodes is far from link saturation (~9.6 Gbps), due to high CPU utilisation.

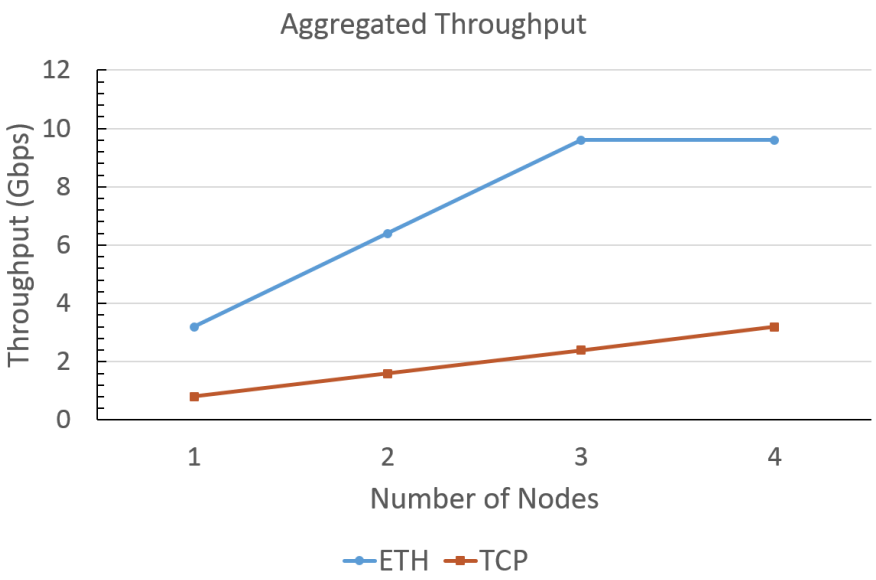


Figure 9: Aggregate throughput for TCP and raw Ethernet frames (one to four nodes sharing one 10GbE NIC). Blue line – NIC level throughput using bare-metal application. Red line – TCP level throughput.

CPU utilisation is very high due to the TCP and IP header checksum calculations in the kernel TCP/IP stack. As explained earlier in this section, in the current version of the prototype we had to resort to a workaround that fools the kernel into allowing scatter/gather operation while still performing the checksum computations without hardware offload. We extracted the TCP checksum calculation function from the Linux kernel sources and ran that code in a user space application to measure the CPU clock cycles (or time) it takes to calculate one TCP checksum. The result is that for Ethernet frames that have the MTU size, the TCP checksum calculation accounts for more than 9000 CPU cycles (about 13 ms) per packet. The high CPU utilisation puts a limit to the throughput achieved when using TCP packets, so running the experiment with multiple concurrent TCP flows does not increase performance as it should. When we run two iperf client threads in a node (to equal the number of cores) that transmitted packets to the PC, the throughput achieved was only 20 Mbps higher than that achieved with single-threaded iperf client, resulting in CPU utilisation of 200% (both cores fully utilised). The same situation occurs when measuring reception throughput with two iperf client threads running in the PC. The reception throughput seen in the node was only 20 Mbps higher compared to the single-threaded experiment. Under conditions of high CPU utilisation, using interrupt coalescing as supported by our driver implementation does not bring performance gains. We ran the same transmission and reception experiments with different coalescing settings, but the throughput achieved is almost the same as when interrupt coalescing is not set.

When TCP transmission and TCP reception flows are concurrent, the total throughput (both TX and RX) achieved in the node is 880 Mbps (the same as the TCP transmission throughput). The throughput is almost equally split between the transmit and receive flows. Again, performance is limited by the high CPU utilisation.

To confirm that TCP throughput is limited by high CPU utilisation, caused by the checksum calculation, we ran raw Ethernet frame transmission and reception experiments. We implemented a

kernel module that is loaded in the nodes and the PC. The module uses the Layer 2 packet transmission function dev_queue_xmit() that is a pointer to our driver's hard_start_xmit() function. In order to send raw Ethernet frames, we had to create the packets in the “raw” form of sk_buff structures. Our test kernel module runs a loop transmitting Ethernet frames (with random payloads).

The first experiment measures the transmission throughput achieved at the nodes, so the raw frames module is loaded at the nodes. The PC runs the ethstats utility that measures the incoming data and packet rate at a specific network interface. The result was that the single node achieved 3.0 Gbps of transmission throughput were very close to the throughput seen by a bare-metal application. In a similar manner, we measured reception throughput at the nodes. This time, the raw frames module is loaded in the PC and the ethstats utility runs at the node. The reception throughput is 2.8 Gbps. These measurements confirm that the high CPU utilisation because of TCP and IP header checksum calculation is the dominant factor for the throughput limitations when using the TCP protocol.

When using all four nodes together, we can saturate the 10 Gbps link, achieving almost 9.8 Gbps of raw Ethernet throughput as seen in the figure above by the blue line. When only one node generates raw Ethernet traffic it achieves 3.2 Gbps of throughput. When two nodes generate traffic, the aggregated throughput seen is 2×3.2 Gbps = 6.4 Gbps. When three nodes are connected the aggregated throughout reaches the link saturation limit, which is 3×3.2 Gbps = 9.6 Gbps. When four nodes generate traffic the aggregated throughput is still the link saturation but the throughput per node is decreased, so each node gets the same portion of the available data rate, due to the round-robin transmit scheduler in the FPGA.

3.3. Running Unmodified socket-based applications (FORTH)

In this section we focus on internal communication among microserver nodes. Microservers are designed to undertake specific workloads that can easily execute in parallel, using large numbers of nodes (scale-out workloads) and demand relatively low processing power. Popular examples are web serving applications or data analytics workloads like those using MapReduce. Workloads like these, even though they are not always demanding in terms of processing power, often perform a great deal of communication among the running nodes and their scalability can be compromised by a slow internal network. Low throughput and, even more critically, high latency can lead to underutilisation of the cores. Solutions for achieving low latency, like TCP Offloading in the Network Interface (NIC), are usually too expensive to be employed in these low-cost systems.

Internal traffic is typically seen by the operating system of each node as normal network traffic simply heading to nearby destinations; most of the time, the OS is not even aware of their vicinity. Interconnecting server nodes using networks originally designed with larger area specifications is probably overkill. This statement applies not only to the hardware side of the system, but also to the software side. Network protocols like the commonly used TCP/IP are supposed to handle Wide Area Network connections, thus having features not needed for small-scale environments.

Nevertheless, optimising a microserver's internal network is not a simple task. Utilising existing high-speed interconnection technologies can be tricky and expensive. Simpler custom hardware solutions can be designed, but this requires effort and extra software support in the OS. Except for this and even more challenging, is that in order to reduce the software induced overhead, the whole programming model of the applications may have to be changed; this basically means rewriting them.

Our goal is to allow unmodified applications to efficiently utilise an RDMA-capable internal network in order to achieve low-latency and high-throughput communication. Our implementation consists of two separate parts executing in user and kernel space:

- In user space, we intercept system calls related to the popular Berkeley Sockets API, to bypass the kernel TCP/IP stack and avoid its overhead.
- In kernel space, we handle data transfers by means of high-speed Remote Direct Memory Access (RDMA) transactions, using a custom RDMA driver.

So far in this effort, we support only *stream sockets* (i.e. TCP connection-based streams) and not datagram sockets (i.e. UDP connectionless transfers).

In this section, we present the following contributions:

- Design of protocol to run Sockets over RDMA, using per-connection transfer buffers and a Mailbox mechanism to send notifications.
- Bypass of kernel TCP/IP stack and implementation of RDMA-based network sockets within the Standard C Library.
- Interception of system calls in user space, within the Standard C Library, to allow unmodified applications to transparently utilise our system.
- Analysis of overheads and a detailed breakdown of latency when using RDMA-based sockets.

System call interception

Applications use the networking subsystem in Linux through the system calls of the Berkeley Sockets API. Since we need to leave applications unmodified, any intervention has to be made after these system calls have been issued.

One potential solution is to modify the Linux kernel and use kernel space interception. Considering the fact that we are going to use a custom communication scheme, any intervention has to happen as soon as possible, close to the system calls' entry points in the kernel. Doing this avoids adding latency from the TCP/IP stack, resulting in a more lightweight network access path. However, another significant source of overhead is caused by simply entering and leaving the kernel space. The processor has to switch mode and the total delay can be thousands of cycles. User space interception is necessary to avoid this overhead as well. Unfortunately, the use of the DMA engine imposes some restrictions. The device could in fact be controlled directly by a process from user space, but this does not allow efficient arbitration among many users. Second, and more important, we cannot allow exposing the physical addresses expected by the DMA engine to the user space. From the

above it seems inevitable, that even if we intercept execution from user space, eventually we will not able to avoid the kernel completely.

We have opted to rely on user-space interception for two reasons:

- To explore the potential benefits of avoiding the kernel, even for only a subset of possible execution scenarios. For example, a send call may not be able to send data if the remote side is not ready for reception.
- The potential future availability of new-sophisticated DMA engines combined with I/O MMUs working with virtual addresses.

Making a system call, means triggering a software interrupt (or a software trap) to force the processor to enter kernel space. This procedure demands low-level assembly programming and thus, is different for every architecture. For this reason, system calls are almost never handled by the programmer. In Unix and Unix-like systems this is responsibility of the Standard C Library (libc): Applications always call simple libc functions, which have the same name as the original system call and are called System Call Wrappers. Figure 10 illustrates the execution flow of a system call. Initially, a system call wrapper prepares the call. The type of the call and the arguments it needs, have to be placed at specific locations, which are usually processor registers. This is where the kernel will look for them. Afterwards, it can trigger the trap. When the kernel returns, in the case of an error, the result is handled according to the errno interface. Eventually, execution returns to the normal flow of the user application.

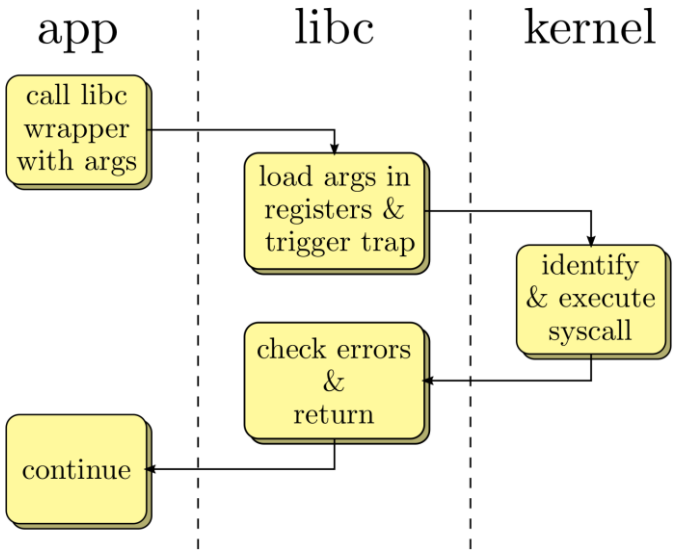


Figure 10: System call execution flow.

Our approach is to intercept system calls by injecting code within the system call wrappers. There is hardly any software that doesn't interact with the kernel using the libc functions. This is true, even when using other languages like Java or OCaml or scripting languages like Python, Perl, and Ruby. These languages usually define their own socket interface, either built-in or as an extension library,

but ultimately, they all end up using libc. One case, though, that we obviously cannot deal with, is the case of statically-linked executable binaries.

Rather than modifying libc as a whole, there is the alternative method of LD_PRELOAD method. LD_PRELOAD is a parameter of the dynamic linker that is given as environmental variable and allows a custom library to be interposed before other linked libraries. Typically, the first library to be examined for symbols is libc, so by placing the custom library before, even system call wrappers can be overridden. However, employing the preload method is more of a temporary solution not suitable for normal use, so we have chosen not to follow it.

The most commonly used libc version in Linux world today is the GNU C Library (glibc) created by the GNU Project. In our prototype we use version 2.15 of glibc. On this version, ARM support is not included in the normal distribution of the library, but it needs to be downloaded separately as an add-on. In newer versions though, ARM support has been fully incorporated in glibc. Glibc features a very complex building environment due to the fact that it supports several different architectures, kernels and Unix specifications. The directory named socket, for instance, contains most of the socket system call wrappers but these are just stub versions of the real functions; they always return an error. This kind of code is included in the final library, only when something is not supported in the target architecture. The real implementation of system call wrapper functions is usually found under the sysdeps directories that contain system dependent code. Generally, they can be divided in three different types: assembly, macro and bespoke.

Most of the system calls are handled by assembly wrappers, which simply follow a specific pattern to convey the system call number and arguments to the kernel. The only things that differ from one system call to another are the system call number and the number and type of arguments. For this reason, there are not source files for every specific system call, but the code is generated at build time, using assembly templates. The use of assembly eliminates any unnecessary overhead caused by a compiler. The sh script in sysdeps/unix/make-syscalls.sh is used to parse the various syscalls.list files and export necessary information. These files exist in many locations inside the sysdeps folders and have a special format, with each line representing one call and its attributes. When the library is being built, several syscalls.list files, either generic or architecture-specific, are read and this way the whole list of assembly system call wrappers is determined. For each one, the above script compiles the file sysdeps/unix/syscall-template.S having passed at the same time suitable variables and values to the preprocessor. The macros in this file produce the final assembly code with the help of other macros defined in the various sysdep.h files.

Some system call wrappers require more work to be done before or after the system call. For example, in some system calls, a slightly different interface is exposed by libc than the one the kernel actually uses. Such cases are handled by ‘macro wrappers’. Their code is defined in C language files and the actual call is made by inline macros again defined in a sysdep.h file. For example, the file sysdeps/unix/sysv/linux/sendmmsg.c contains the Linux implementation of sendmmsg. This wrapper will be different for any architecture because it uses the INLINE_SYSCALL() macro that includes inline assembly code.

Finally, there are a few system call wrappers that do not use the standard assembly or C inline macros. In the future, probably these will be changed, too. No socket-related calls belong to this category.

There is some additional complexity for multi-threaded applications. The Native POSIX Thread Library (nptl) is a Linux implementation of the POSIX Threads (pthreads) API that is now integrated in glibc. Besides the standard libc library, libpthread is also produced by the glibc build process for use by multithreaded applications. libpthread redefines several system calls and because of this, it has its own sysdeps directory structure (nptl/sysdeps). The sysdep.h files we saw earlier are now called sysdep-cancel.h. This is because these system calls act as a pthread cancellation point when used in libpthread -- they are checking if their thread has been cancelled. Thus, their implementation is a little different than the normal one. An interesting fact is that, although two distinct libraries exist, libc also includes some functions that normally belong to libpthread. Besides this, even some system call wrappers inside libc use nptl's versions -- those created by the macros of sysdep-cancel.h files.

In order to intercept the system calls inside glibc, we must inject our code within the wrappers. This is quite easy for macro wrappers, since they are written in C. For assembly wrappers, we have to call our own functions from the assembly code. This step relies on making changes to the build process, and more importantly on changes to the assembly wrapper templates (based on the calling conventions for ARM processors).

Additional work is needed for situations (e.g. for the accept system call) when what we need is to interpret the results of a system call, i.e. allow the system call to proceed and intervene before its results become available to the caller. This entails running the call normally in the kernel, and then inspecting what it returns when it is finished. For post-kernel interception, our assembly code injected in the wrappers' building templates is different and custom for every case. Luckily, the cases where this handling is needed are not many. Again, we call our functions from there, but this time the arguments passed to them are different. Depending on the call, the kernel's return value is passed along with some of the original arguments. These arguments may be output arguments of the system call, using the pass-by-reference technique. At the end, the value returned to the caller of the wrapper is the value returned from our function. The following figure illustrates the handling of system calls by our modified libc library, showing interception before and after the kernel.

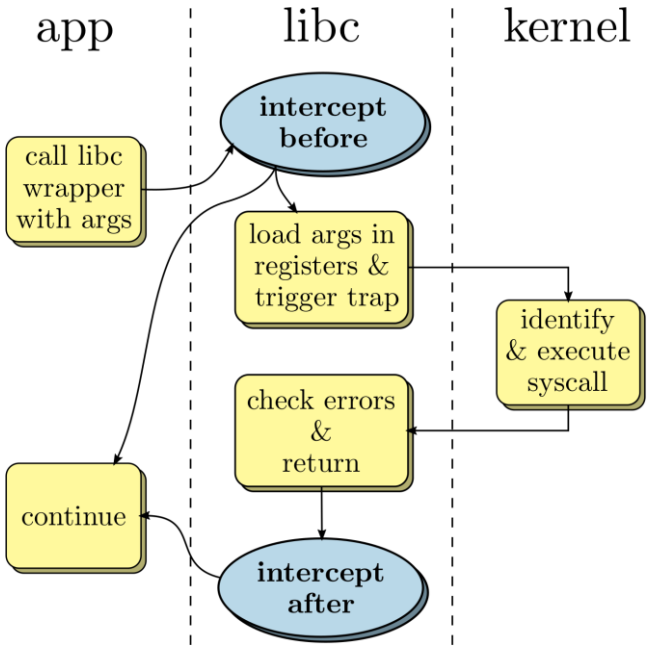


Figure 11: Intercepting a system call before or after the kernel.

One final difficulty that we encountered with glibc was building the run-time dynamic loader. This is the loader of all the other dynamically linked libraries to a binary. Therefore, it cannot contain functions from other libraries. Unfortunately though, it needs some system calls and those are included in the loader (the wrappers). One of them is write, for instance, which is one of our intercepted calls. Thus, our own modified assembly version wrapper was to be included and this caused an error. To overcome this difficulty, we relied on the build system defining a preprocessor variable (IS_IN_rtld) that is set during the build of the loader. Checking for this value in sysdeps/unix/syscall-template.S}, ensures that the non-modified wrappers enter the loader.

Kernel driver to support TCP socket connections over RDMA

There are two main reasons why it is necessary to control RDMA transfers from kernel space:

1. Arbitrating multiple users of the DMA engine
2. Avoid exposure of physical addresses to user space.

The driver assumes it is the sole user of the AXI-CDMA engine (Xilinx LogiCORE IP), and takes full control of it. Moreover, extensive use of the mailbox mechanism is made to send remote interrupts and exchange messages with the remote side. During module initialisation, page table entries for the CDMA engine and the Mailbox are created to provide access to their configuration registers. Then, the devices are reset and configured. For the CDMA engine, regular and error interrupts are enabled and the scatter-gather mode is selected. For the mailbox, two interrupt types are enabled: the ``enqueue'' interrupt, and the ``full FIFO'' interrupt. Finally, these interrupts are registered to be handled by the interrupt handlers of the driver. Several types of messages are served by the mailbox. Each one of them has a message ID and usually a connection ID. These are found in the four most significant bits and in the next 12 bits of the 64-bit message, respectively. With this configuration, we

can support a maximum of 16 message types and 4096 simultaneous connections. Currently, only six message types are used and the rest are reserved for future use.

Before the RDMA driver can be used, RDMA descriptors have to be created. A set of pre-allocated descriptors is used for all transactions. These are permanently connected together. In a 4KB space, 64 descriptors are created. Each one of them has a size of 32 bytes but also requires to be aligned to a 64-byte address. As a result, 32 bytes of padding are added. Every descriptor points to the next via the NXTDESC_PNTR field and the last one to the first, forming a circularly linked list. The descriptors are protected by a lock. This is a kernel semaphore that one must hold to edit any descriptors. The number of descriptors for every transaction is not known beforehand. Therefore every sender has to lock the list to avoid overwriting descriptors used by others.

Processes that create TCP connections within the local network have to use the RDMA driver. Calls to the driver are, most of the time, simple writes, passing information to the driver in the buffer argument. For instance, a "c" string makes a connect request whereas an "r14" means that data must be received from connection #14 (actually, connection IDs are passed in binary format). Before the driver can be used by a process it must be opened. This occurs the first time a process calls the socket system call to create a new Internet socket. In the intercepted call, it is checked that the domain argument refers to a network socket and then the driver is opened. The returned file descriptor is stored in a global variable so that every other intercepted system call can use it. In case the driver is not loaded in the system, the normal TCP/IP network path will be followed. Apart from opening the driver, a special configuration file is also read during the socket call. This file must contain the IP address of the remote node. This address is stored in libc, as a global variable (peers), and is checked to determine if a destination is local or not. To support more than one remote nodes, peers has the form of a linked list.

Connection establishment

Before any data transactions can happen, a connection must be established between the server and the client. The client must call connect, with the prerequisite that the server is listening to the same port. The accepted connection will then be handed over to the server by using the accept system call. The main issue when establishing a connection via RDMA is that a local and global address mapping must be maintained. To overcome the global consistency issue, data structures that maintain the active connections have been introduced to the modified libc and the RDMA driver. These structures have had to be carefully designed to avoid adding too much additional latency when establishing and destroying connections. More information about the techniques implemented and the issues discovered are detailed in the Appendix (see Section II.b).

Connection buffers

Data buffers are kernel buffers, also memory-mapped to user space. The benefit of this is that the user process can avoid calling the RDMA driver and undergo the user-to-kernel switch overhead. When data are already available in the receive buffer, they will be consumed immediately by a read call and when there is no need to send any data yet, a write will only store in the send buffer, hopefully coalescing data with data to arrive later. The use of buffers per connection allows this user-space buffer access. Otherwise, one process would be able to watch data belonging to other

processes. For more information about the specific implementation choices and background into connection buffers please see Appendix (Section II).

As a matter of fact, this user mapping takes place at the end of the connection establishment phase. A call to mmap is issued right after returning from the RDMA driver in accept and connect, but before leaving the wrapper. mmap is a system call that creates new user mappings. For example, memory regions of a driver or even real files from disk can be mapped inside the memory space of a process. The real system call is issued (with no interception) and then the mmap implementation (drivers have to implement a specific function in order to support mmap; the same thing happens with write, read, etc.) inside the driver takes over. Most of the work though, like the page tables modification, is still done by the kernel.

The process calls mmap passing the driver's file descriptor as an argument. However, a connection ID must also be given: the ID of the newly created connection. The trick we do here is passing the ID via the offset argument of mmap. Then the driver performs the correct mapping and finally mmap returns a user pointer from where the buffers can be accessed. This pointer is stored in buffers table, which is another of our global libc tables, where buffers of all local connections of a process are kept.

Sharing resources between user and kernel space can lead to races with unpredictable results. For instance, the user process could read a value and then before it makes an action an interrupt could appear that stores something else, ruining the procedure. In our case, such risks do not exist because every important variable is always written by only one and because of the way the buffers are updated. For example, the tail of a send buffer is always written by the intercepted user space code of write, whereas its head is only updated by the kernel after a successfully completed transaction. Before the driver begins the RDMA operation, it reads both the head and the tail. The only thing that user space can do at this moment is write new data after the old that will not affect the current transaction. Data up to the old tail will be sent and moreover, the new data cannot overwrite the old, because the head pointer is controlled by the kernel side.

Receiving data

With the current implementation, the actual data transfers with RDMA operations are initiated by the receiver. A normal recv call blocks until data are available, so the other side must be informed of this and unblock it as soon as possible. The read request message has been created for this purpose. The sender of the read request message also sends the current values of the head and tail pointers of the local receive buffer. With the current communication scheme, the remote side can deduce the values of these pointers since it is the sole writer of the receive buffer and because the local side will not issue a read request before consuming all the previously received data. However, the pointers are included in this message as a safety measure and for possible future use. The following figure (Figure 12) shows the flow chart for receiving data.

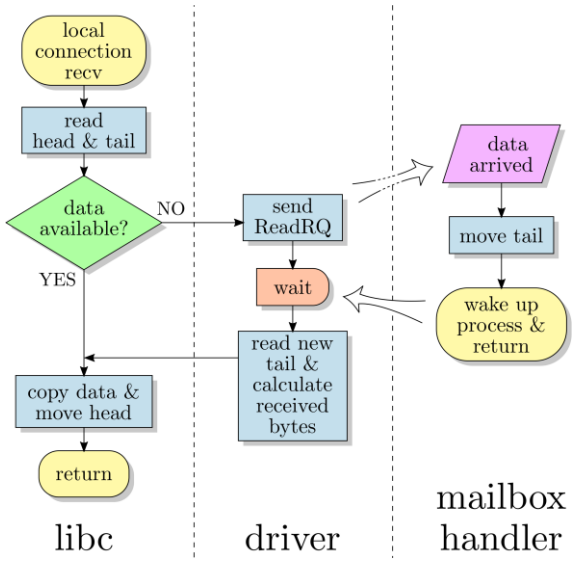


Figure 12: Flow chart for receiving data.

There are two main parts playing a role in the receive procedure. The synchronous part, consisting of the intercepted system calls in libc and the RDMA driver in the kernel. Their actions are initiated by the user process, whereas the asynchronous part is the mailbox interrupt handler, which is executed depending on the data arrival moment. The mailbox interrupt handler is part of the RDMA driver, as well, but is not executed on the context of a user process.

The intercepted recv performs the following steps when a local connection is encountered:

1. Check for data. The first thing to do is check whether there are already data available in the receive buffer, by reading the head and tail pointers. This could happen if a previous recv had not consumed the whole available payload. If this is the case, the next step is bypassed.
2. Send a read request. If head equals tail, then there are no data and a read request must be sent. The RDMA driver is called and the message is sent by performing a simple store command to the remote mailbox. Afterwards, the process is put to sleep in the kernel (special care has to be taken to avoid the lost wake-up problem). This happens when a wake-up event arrives after it was decided to put the process to sleep, but before this procedure is completed. As a result, the event is lost.), waiting to be woken up when data arrive. After the wake up, the tail pointer is read again to find out and return the number of received bytes to user space.
3. Consume data. The available data or a part of them, if there are more than recv has requested, are then copied in the buffer given with the system call. Afterwards, they can be released from the receive buffer. This is followed by writing the suitable new head in the buffer.

The receiver does not inform the other side on how much data it wants (the count argument of the system call), but how much data it is able to receive. The sender is free to fill the available space with as much data as it has available.

An interrupt is issued upon data arrival. After identifying the message type and the referring local connection, the following steps take place:

1. Move the tail pointer. Data are already copied in the correct position of the receive buffer, so now the buffer tail pointer has to be updated to show this. The count of received bytes is included in the interrupt message.
2. Wake up the process. The driver knows the process associated with this particular connection, so the interrupt handler reactivates it (this includes changing the process state to TASKRUNNABLE and placing it in the task scheduler's list).

Sending data

The flow chart for sending data is shown in Figure 13. In order to send data to the remote side, a send call has to be made. The time of actual sending through the network though, is not always known. Initially, the available space in the send buffer must be checked. This is deduced by the head and tail pointers. If the send buffer is full of unsent data, the system call has to block until some space is freed. As usual, the process sleeps in the driver. Subsequently, the correct amount of data is copied to the send buffer and then the tail pointer is increased respectively. The read request flag has to be checked at this point, because if a request had arrived earlier the transaction would not happen. If the flag is set, the RDMA driver is called to perform the RDMA operation.

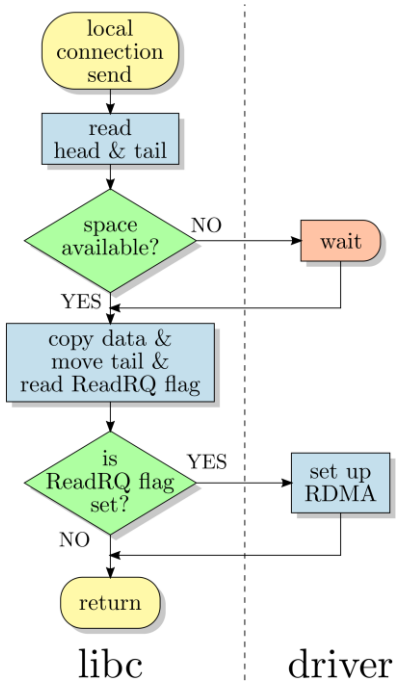


Figure 13: Flow chart for sending data (libc, RDMA driver).

There are two types of interrupts affecting the send procedure (see Figure 14):

1. Arrival of a read request. The head and tail pointers are read and then one of the next two steps is performed: (a) Set the ReadRQ flag. If the send buffer is empty, the first subsequent send system call has to know that the remote side is waiting for data and send them immediately. (b) Schedule RDMA. If there are data to be sent, an RDMA operation must occur. Unfortunately, the interrupt handler cannot perform this action (it

may sleep), so it is scheduled in the kernel's default workqueue to happen as soon as possible.

2. Completion of an RDMA operation. (a) Move the head pointer. When an RDMA operation finishes successfully, the send buffer has to be updated to free the reserved space. (b) Wake up process. If the process is blocked on a send call, it can now be unblocked since there is free space in the send buffer again.

The arrival of a read request can occur between the two events of moving the tail pointer and checking the read request flag in the wrapper. This rare situation, however, cannot lead to an erroneous outcome. The handler will see that data exist and will schedule a transaction, without setting the flag. Consequently, the code in the wrapper will not begin another RDMA operation. Additionally, no extra read request can be delivered, because the remote side is blocked, waiting for data.

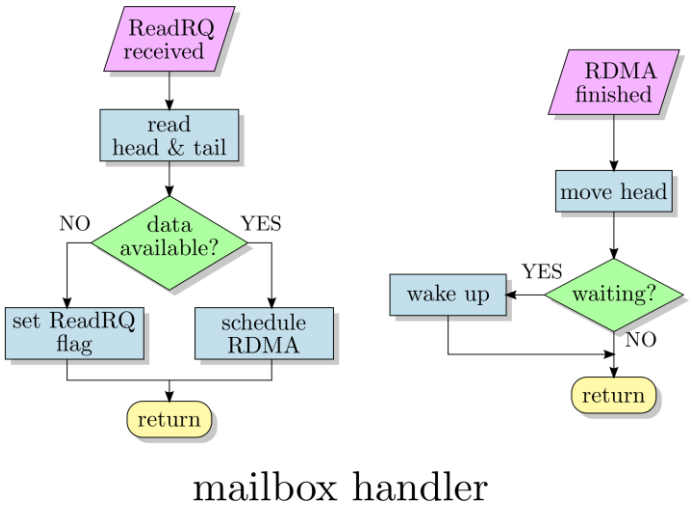


Figure 14: Mailbox interrupts affecting sending of data.

Data transfer via RDMA

Making an RDMA operation requires preparing the descriptors that are going to be used and writing the last of them to the TAILDESCPNTR field of CDMA. To perform these two actions the descriptors semaphore must be held and as a consequence, this procedure may sleep in the case of a contended semaphore. This is why the mailbox handler schedules an RDMA rather than carrying it out at once. In the general case, copying data between two ring buffers normally requires from one up to three different transactions, depending on the data position at each side (if they wrap around). In our case, because the receive buffer of one side always follows the send buffer of the other, only one or two transactions may have to be performed. The latter case occurs when data expand beyond the physical end of the buffer to its start. The first transaction will copy the data from the head pointer, up to buffer's end and the second from its start, up to the current tail. The following figure (Figure 15) illustrates these two cases.

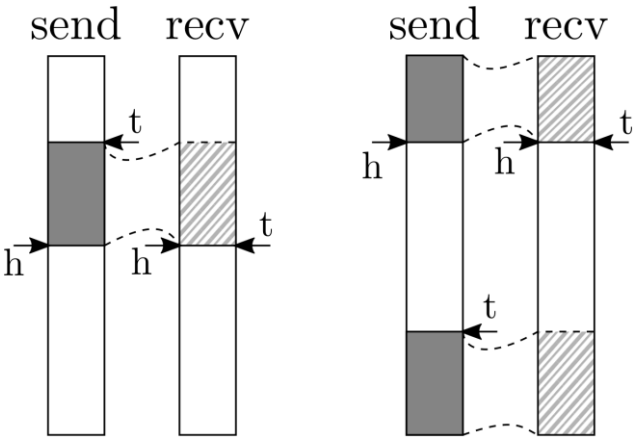


Figure 15: Data transfer via RDMA: either 1 or 2 transactions are needed.

After a successful RDMA operation, both connection sides have to be notified. The remote side needs to be notified in order to be unblocked from the read call and to wake up the local process possibly waiting for space in the send buffer. The only way to perform a remote interrupt is by using the mailbox mechanism along with the address translation feature. A simple (64-bit) store to the address of the remote mailbox will trigger the interrupt, which will also be secure against other simultaneous interrupts possibly coming from other nodes, due to mailbox's fifo list. As far as the local interrupt is concerned, the interrupt caused by CDMA could be used. However, this is not easy because of the way this interrupt is triggered. When operated in scatter gather mode, CDMA produces an interrupt after a programmable but also fixed number of completed descriptors. Unfortunately, as we have mentioned above, the number of transactions needed can be either one or two. Except for this, we would also have to identify the connection that the interrupt belongs to. For these reasons, we use the mailbox again to produce the local interrupt as well.

A complete RDMA operation consists of three or four descriptors: The one or two data copying descriptors and another two producing the local remote interrupts. Preparing data descriptors is straightforward. On the other hand, for the interrupt descriptors, the DMA engine needs to read the content of the mailbox messages from somewhere. For this, we take advantage of the free 32 bytes after each descriptor, where the 8-byte messages are written.

An example of a complete send/recv procedure is given in the timing diagram is shown in Figure 16. The DMA engine is also added in this diagram, with the thick line indicating the data transfer in this case. In this example, recv has been called before send. Furthermore, only one send is issued, so there could not be a case of blocking in a send call.

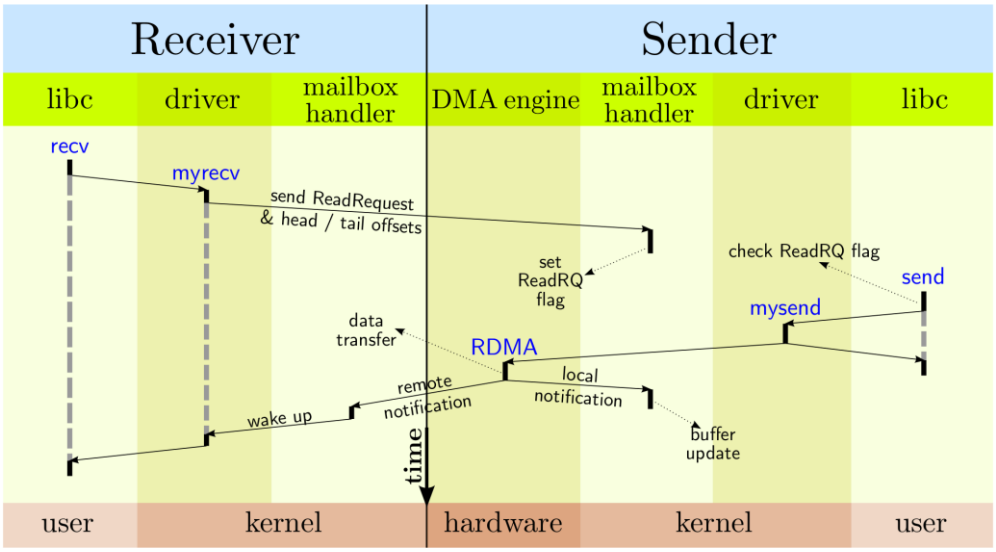


Figure 16: Sending and receiving data.

Closing a connection

A socket is terminated either by explicitly calling the close system call in the user code or when the process ends. In our injected code in libc, there are no dynamically allocated resources to be released. A close call updates our global tables to forget this local connection and it closes the real accepted socket descriptor, which was left open, so that the kernel can reuse its number. However before this, a call to the RDMA driver is also needed to release the connection resources allocated by the driver. These include the local buffers and structures like the connection struct. Finally the driver's global connection table must also be modified. If close is not called at all, the driver cleanup still takes place with the help of the kernel's cleanup procedure of the user process. The kernel will call two registered cleanup functions of our driver: The first when the mapping created by mmap is undone, and the second when eventually releasing the driver from the process. As a result, the connections belonging to that process are again normally released preventing memory leaks in the driver's memory.

The remote connected side is more complex. If a connection is simply closed unilaterally, several problems can emerge, either affecting the remote or the local side. First of all, the remote side could have been waiting for data and consequently, would be stuck there forever. This is easily handled: During the connection cleanup is checked whether there is any read request already. If this is true, a remote interrupt message is sent with a value of zero in the bytes field, causing read to also return zero. This is exactly what TCP/IP does in a similar case. If the request arrives after the closing procedure has completed, this message is sent by the interrupt handler. On the other hand, if the closing side has any data in the send buffer, these are sent normally and the cleanup continues thereafter.

Another possible difficult situation occurs when the remote side is preparing an RDMA operation and the connection is closed locally. This could result in overwriting memory that no longer belongs to the driver, risking the whole system's integrity. An RDMA happens only if the local side is waiting for it and as a result, the connection could be closed safely if this is not the case. Unfortunately, though,

nothing can be done if it is already waiting though. It is not possible to determine whether and when the remote side will send data. On this occasion, some kind of negotiation must be done between the two sides.

Finally, another case showing that this negotiation is inevitable is the following: the local side closes a connection without any of two above problems happening. The connection ID is then released and after a while it is used again for a connection with another peer. However, the original connected peer still thinks that it is connected, so it can send messages at any time; the local side will not be able to determine where the messages came from.

The NACK mailbox message is used to abort an internal network connection by informing the remote side before any actual cleanup. This is not enough, however. The aborting side must be sure that the message was delivered and handled. As a result, a disconnect procedure actually takes place.

The two sides perform a handshake using NACK messages. The remote side receives the first NACK and cancels any upcoming RDMA operations and waits for any ongoing ones. Afterwards, it responds with another NACK message and when it is delivered, the other side can finally release the connection's resources. After a timeout, the original aborting side would resend the message if it did not receive a response from the remote side. The sender can also include an integer value in the message representing the reason for aborting. Furthermore, his node ID must also be sent, handling situations like the following: Assume that PeerA first send a NACK. PeerB receives it but sends his response delayed, so that a resend from PeerA happens. However, as soon as PeerB sends his response, he considers the connection as over and could have even started a new one with the same ID. Therefore, after receiving the new NACK the node ID in the message will clarify who sent it. Beginning a new connection with PeerA using the same connection ID is not problem, because PeerA keeps a list of uncompleted NACK procedures.

Aborting the connection establishment procedure is again carried out with NACK messages. This could occur in the case of a system error (e.g. out of memory error), a response timeout, or the user interrupting the procedure. On this occasion, the value field of the message is also used to give the reason, using standard defined values of the Socket API like ECONNREFUSED or ECONNABORTED. These values are then returned from the system call of the remote side. Additionally, a special case exists that requires different handling. If a node sends a CONNRQ message to begin a connection and for some reason the host does not respond or the client decides to abort before the first reply, then the client does not yet know the remote connection ID to send a NACK. In this situation, connect returns to the user with an error code, but the driver holds the ID for a longer period, until it can presume that the connection has failed.

Support for forked and multithreaded applications

Multithreaded and forked applications result in sockets being shared among multiple processes or threads. This sharing could easily cause many problems to our system. For example, a process issues a read and afterwards, a child process issues another read on the same socket. How will two read requests be handled? Who will eventually receive the data in this case? The RDMA driver could have

been designed to serialise such accesses, but unfortunately, due to the mixed user/kernel space architecture of our system, this would not be enough.

However, normal socket programming does not commonly involve concurrent use of sockets. For example, a server usually accepts a new connection and then creates a new thread to deal with it; no other thread will use this socket. In forked applications, usually either the parent or the child keeps a socket, while the other one closes it immediately after the fork takes place. However, since it can happen, it must be handled. The same thing is done by the kernel network stack. It ensures that all data are transferred, even mixed with each other sometimes.

Therefore, we have implemented custom locks to protect socket operations on our internal network. In general, the RDMA driver is designed to prevent events that could compromise the integrity of the whole system (the OS and other running applications), whereas data integrity of our local sockets is handed over to this locking procedure in libc. We could not use the mutexes offered by libpthread because our injected code - where the locks are needed - is part of libc. Some elements of libpthread are also integrated in libc, but features like shared mutexes among processes - necessary for forked applications - are only found in the first. So, libc would have to be linked to libpthread and this would require significant changes in the build system of these libraries. Apart from this, we need the locks to specifically lock connections, so our RDMA driver can be directly used, rather than employing the kernel futex subsystem that is more generic (pthread mutexes use the futex system call to wait on a particular memory location; all these locations are kept in a hash table in the kernel so processes can sleep or woken up).

Our locks are implemented in a similar way to pthread mutexes. A memory location is used to perform atomic operations (we use the atomic compare-and-swap primitive provided by gcc) to. The first to modify this location gets the lock. Others have to call the driver to wait. Before this though, they modify - again atomically - the location so that it now indicates that not only it is locked, but contended, as well. Afterwards they are put to sleep in the driver (again the lost wakeup problem must be dealt with). When the lock owner releases the lock, he calls the driver to wake up one, if it is contended. What needs to be locked in our case, are the different connections. Each connection has buffers and these buffers are shared among all users of it, the RDMA driver included. Consequently, it is convenient to place the locks in these buffers. Furthermore, because every connection has two independent paths, one for sending and one for receiving (and hence two buffers per side), two locks are used per connection.

The RDMA driver keeps a list of all processes that use a particular connection. This is a linked list inside every connection struct. Its elements are also structs, including a pointer to the process along with information referring to its current state. This pointer has the type of a kernel struct called taskstruct, as processes are called tasks in kernel terminology. An interesting thing is that threads are also represented with the same struct and are in fact, actual tasks. They differ from normal processes in that they share many parts with other tasks. As a result, the tasks list of a connection can include threads as well.

The state we keep for every task is its state in relation to the connection. That is for example, if it is waiting on a recv call, or on the send buffer lock, etc. This information is used by the driver to wake up the correct tasks. Except for this, the other reason that the tasks list is required for, is security. Most of the calls to the driver also contain a connection ID. One of the first things done by the driver when serving these calls, is to check out if the calling process has indeed access right for the connection. Otherwise, it would be easy for any user process to steal data from others. However, because the size of the tasks list could become very large, while usually one only task is using the connection, there is actually a second list. The tasks list contains all the legitimate potential users of the connection, while at the users list only the actual users of it are inserted.

For every system call that involves sending data (send, write, sendmsg et al.), the intercepted call acquires the send lock of the connection before doing anything and releases it at the end. In the same manner, all receiving system calls lock the receive path until getting and consuming the data. For the case of connect and accept, locking primitives are not needed. Concurrent connects will safely - due to driver's locking - create multiple new connections, whereas with accept, again different connections will be accepted, with the kernel this time handling concurrency. Finally, locks are being used in the intercepted close system call, but this is done to protect the procedure of dup.

In Linux, creating threads and forking processes are actually implemented with the same system call, clone (for compatibility, the old fork and the vfork variation still exist as separate system calls, but in the kernel they still use the subsystem of clone). The difference between these two is the amount of resource sharing between parents and children. For example, threads share memory space, while in fork a copy-on-write technique is employed.

Our custom locks would suffice for running multithreaded and forked applications, without any interception at the cloning procedure. However, we have also intervened there, for two reasons:

1. The clone table. We want to avoid the overhead of locks, when they are not needed. Single-threaded applications would always run with uncontended locks, not entering the kernel, but still the usage of atomic primitives wastes many CPU cycles. For this purpose, another global table has been added in our modified libc and libpthread code. The clone table uses file descriptor numbers as indices, like the table of local connections, and gets updated after cloning procedures. The values of this table show if a particular local socket needs locking or not. To give an example, let us assume that a process has a local connection, to which the file descriptor 3 is assigned. If the process forks itself, both the parent and the child will now have the value of 1 at the 4^{th} element of their clone table. This value instructs any data transferring operation to use the locks, because the connection is shared. On the other hand, if the child later creates another local connection, its value on the table will remain 0, as the parent has nothing to do with it. Therefore, a send call will see this value and will not acquire or release the lock to send the data.
2. Updating the task list. Only tasks included in the driver's tasks list of a connection can use the connection. But how does this list get its elements? The first user is simply added during the connection establishment procedure. When cloning occurs, the new child has to be announced to the driver. One possible solution, could be the child to call the RDMA driver,

which could confirm its parent identity, using the kernel struct describing the child. However, it is possible that the parent has already terminated before this. In this case, the pointer to the parent at the kernel's struct would be NULL. The exact same thing could happen, if the parent was the one to enter the driver to announce its child. Consequently, to solve this problem, both of them call the RDMA driver, with the first waiting the second. Thereafter, the tasks lists of all local connections the parent has, are updated. An interesting detail is that we do not use a lock to protect a read of the list from a possible update of it - concurrent updates are not allowed, however. Traversing the tasks list occurs very often and this would cause overhead. In fact, we use the double-ended linked lists provided by the kernel and we only add elements in the end. The way that the kernel function of adding an element is written could only ruin the reverse traversal of a list. Since we use only use the normal direction, each read of the list is protected.

Support for socket options

There are lots of socket options, controlling various parameters of socket connections. Currently, only the most common of them is supported, the SOCKNONBLOCK. This is actually an attribute of file descriptors and not only sockets. It can be set either in the socket call, or later using fcntl.

The non-blocking attribute makes all calls to a socket return immediately. For example, if no client has yet connected to a server, accept returns an error and so does a recv when no data are available. A noteworthy change that was implemented when dealing with non-blocking local sockets is the following: At the sender's side of a data transaction, a send would never block; if the send buffer is full, an error is returned. Therefore, the local interrupt is useless in this case. What we do is replace the RDMA descriptor causing the local interrupt, with a new that only moves the head of the local send buffer. This way, we avoid the interrupt handling overhead.

Support for the dup system call

The dup family of system calls (dup, dup2, dup3) is used to create aliases of file descriptors, including socket file descriptors. For example, a dup(3) call that returns 4, has created a new file descriptor (number 4), that is exactly the same with the old descriptor (number 3). Intercepting a dup call referring to one of our local sockets is simple. We let the real system call run in the kernel and intercept its result. This is another case of post-kernel interception. The file descriptors that are passed to the kernel as the argument of dup, are the original socket created by the socket call at the client, and the socket created by the real accept at the server. The kernel thinks that the first is still unconnected and that the other had a localhost connection and is now waiting to be closed. Both of them though, are not closed, so their descriptor numbers are still valid. In our previous example, our intercepted call will just copy the value of the 4th element of libc connections table and buffers table to the 5th element. Now both these file descriptor numbers will refer to the same local connection.

A problem arises here with the close system call. Having duplicated descriptors means that only the last close call, actually closes them. All previous just remove the aliased descriptor numbers. As a result, another table has to be maintained, to keep the count of descriptor numbers assigned to local sockets. The socketcount table has elements with indices referring to connection IDs. Their values

are the count of each connection and are updated while holding the send lock - the recv lock could also be used – to prevent errors.

Early evaluation results

We have validated our current design and implementation using several micro-benchmarks. A complication in presenting a thorough evaluation is that the internal RDMA-capable interconnect is of considerably higher throughput than the standard 1GbE NICs available on the compute nodes of our prototype. In this section, we focus on latency measurements, and present a detailed breakdown of latency components.

Our evaluation considers both the latency imposed by the physical link and the latency added by our protocol and the operating system. Our test is a micro-benchmark that performs a ping-pong style communication between the two peers. One of them initially does a send operation, while the other waits for the data with a recv. As soon as this transaction is completed, another one is carried out in the opposite direction this time. These transactions do not overlap, because each time one is sending and the other is waiting. As a consequence, if we measure the total time that one side needs to perform these two actions we have the aggregate latency of the two transfers. Supposing they are symmetrical, the half of this time is the latency we want: The latency from the time the sender starts sending data until the receiver consumes them. We collect measurements from 500,000 iterations, for message sizes ranging from 16 bytes to 4KB. We do not include the connection establishment delay in this measurement. The following table (Table 1) and figure (Figure 17) summarise the results.

Table 1: Latency evaluation for sockets-over-RDMA.

Transfer Size (Bytes)	Latency (microseconds)	
	Sockets-over-RDMA	TCP/IP NIC
16	13.99	62.35
32	13.97	62.49
64	14.04	62.89
128	14.58	64.38
256	14.80	67.18
512	15.20	72.54
1024	17.33	84.78
2048	21.17	85.11
4096	28.06	85.44

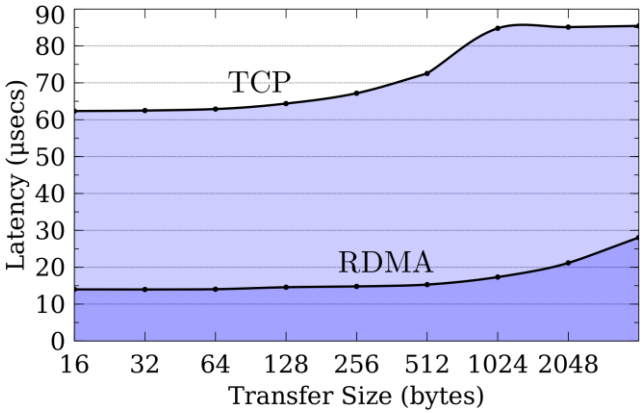


Figure 17: Latency evaluation of sockets-over-RDMA.

Compared to the TCP/IP latency, our Sockets over RDMA system is three to five times better. We can see that for small sizes, data copying and transferring is not so important and latency is mostly determined by factors like the interrupt handling or the user-to-kernel and kernel-to-user switch overheads.

We have also used the Performance Management Unit (PMU) of the ARM A9 processor to measure the overhead of connection establishment. The time needed to establish a new connection is 100 microseconds, compared to the 180 microseconds with TCP/IP. While it was not our primal concern, creating new connections rapidly can benefit significantly modern servers, which often service numerous short-lived clients. As explained earlier, our connection establishment procedure includes establishing a 'dummy' TCP localhost connection, and this is also included in the total of 100 microseconds. Still, our connection establishment sequence is faster though. Our measurement takes place at the client side and is essentially the time that the connect system call blocks.

The following diagram (Figure 18) summarises the breakdown of latency into components: both at the sender and at the receiver sides, including both user-space and kernel-space processing. The measurement is for 16-byte messages. For this message size, our current implementation achieves a latency of 14 microseconds. However, many of the following latency factors do not depend on transfer size and are always the same.

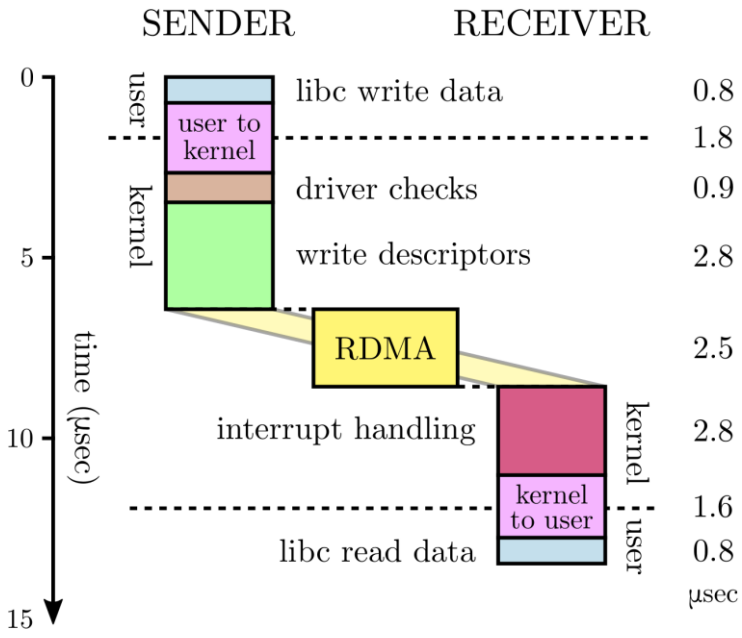


Figure 18: Latency breakdown for sockets-over-RDMA (16-byte messages).

We begin this breakdown from the sender's side and end it on the receiver, as shown in Figure 18. The breakdown of latency into components is as follows:

1. At the sender, the time spent in user-space in our libc code, before entering the kernel, is around 0.8 microseconds. During this interval, data are copied to our buffers. Thus this latency component depends on the size of the transfer.
2. A context switch from user- to kernel-space costs around 1.8 microseconds.
3. After entering the kernel, the RDMA driver takes about 0.9 microseconds to perform certain checks before initiating the transfer (using the DMA engine and other hardware mechanisms).
4. To prepare the RDMA descriptors, another 2.8 microseconds are spent in the driver. This interval reflects the preparation of 3 descriptors (which is the common case in our current implementation).
5. The data transfer itself costs around 2.5 microseconds (this interval depends on the number of bytes transferred).
6. At the receiver side, the time to server an interrupt (triggered via the mailbox mechanism) is around 2.8 microseconds.
7. Following the interrupt, there is a kernel-to-user context switch, taking around 1.6 microseconds.
8. Finally, after returning from the RDMA driver to the receive system call wrapper in libc we need to copy the data just received (16 bytes) to the buffer indicated by the receive call. This step costs another 0.8 microseconds (and depends on the actual transfer size).

One important detail of the 16-byte transfer is that the interrupt handling interval we have mentioned, concerns 2 interrupts on the receive side. After the data transfer has completed, the local interrupt is sent to the sender and the remote interrupt to the receiver. However, due to the delay of the interconnect, the local one is delivered first. As a result, the sender returns from the write call and continues to the subsequent read. This causes a read request to be sent to the receiver. We have verified (by using counters in the driver's code) that 99% of the time the read

request is served together with the remote interrupt. After completing a particular interrupt, the interrupt handler always checks if a new interrupt has arrived, to avoid the overhead of exiting and re-entering the interrupt context.

Extensions

The current implementation achieves the primary goal of allowing unmodified applications to communicate over the RDMA-capable internal interconnect in the EUROSERVER microserver prototype. Basic functionalities like creating TCP connections and performing data transactions through them have been implemented, along with support for more advanced features like multithreaded and forked applications. One important direction for improvements is adding support for more functionality available in the sockets API – specifically, event-driven calls (such as select and epoll) to support popular server applications, and support for passing and interpreting socket options (setsockopt call) such as dynamically setting buffer sizes for sending and receiving data, and operating sockets in a non-blocking mode (e.g. use the fcntl call to set the O_NONBLOCK option for an open socket). This line of work will broaden the applicability of our sockets-over-rdma implementation, to cover more workloads and allow application to fine-tune parameters and capabilities. Moreover, we are considering optimisations to improve the scalability of our implementation, especially in the context of large numbers of open connections.

3.4. Improving network attached virtual block storage performance – ATAoE integration

Introduction

The MicroVisor (described briefly in Section 2 and in detail in D4.4) is a highly-tuned hypervisor kernel that requires a minimal footprint to execute. It is designed to support virtualisation across many small, low powered SoCs. Traditionally all management and control for a hypervisor running across a set of cache coherent cores is handled through a local control domain that provides virtual device communication for virtual machines including network, storage and console access. Mapping these functions onto physical device hardware requires some multiplexing of virtual I/O streams onto the physical device driver I/O queues, typically handled using generic layers in the Linux system stack such as attaching a corresponding Virtual Interface to the software bridge component with a VLAN interface for network I/O. For storage I/O, virtual block I/O requests are mapped onto a generic storage device, typically either a network attached block LUN or a file on a shared network filesystem such as NFS.

The MicroVisor adopts a novel approach to virtualising device I/O by embedding native Ethernet frame handling in the hypervisor layer. By implementing a native virtual Ethernet switch in the hypervisor, packet handling between virtual domains is significantly optimised, reducing packet forwarding latency by almost an order of magnitude by not having to context switch into the control domain in order to make a forwarding decision and then switch back to the hypervisor to trigger a hypercall into the receiving domain. A more significant benefit in the context of microservers however is the ability to remove any dependency on a localised HostOS or Domain 0, thus allowing a

centralised controller to communicate with the distributed low power nodes efficiently using a lightweight raw Ethernet protocol, and to control state and actions that are executed across them.

Motivation

Virtual block I/O handling across the end-to-end path typically involves a translation through a few logical block layers as well as encapsulation into a TCP/IP stream for remote storage access. A block request originates in the VM via the virtual block device driver (e.g. netfront on Xen), is communicated to the backend handler in the control domain, then passed to a Linux kernel block device such as a device mapper layer device or a loopback device to map to a file. The block request is then translated onto a network communication path such as iSCSI or NFS both of which typically utilise a TCP/IP connected stream protocol. The block request must be acknowledged from the remote device back over the network and then translated back through the block layers to the virtual block device driver in the guest in order for the VM's application layers to continue.

One of the core design principles of the MicroVisor however is to maintain an extremely small footprint with very low processing overhead. Given that the MicroVisor must provide a similar type of block semantic in the VMM layer since the HostOS no longer exists, an alternative lightweight approach has been sought. Integrating an iSCSI protocol stack client into the MicroVisor was not considered an option due to the dependency on a full TCP/IP stateful stack to be handled in the VMM layer. Instead, a very lightweight native Ethernet block protocol client, referred to as ATA over Ethernet (ATAoE), is implemented to provide block I/O to network block request translation directly in the MicroVisor layer.

Architecture

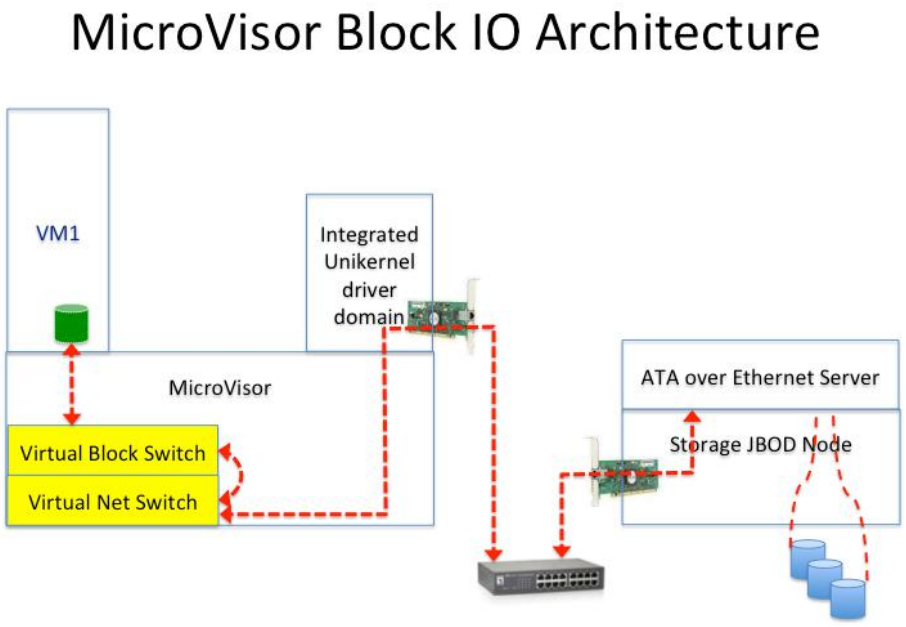


Figure 19: MicroVisor block I/O architecture

The virtual block I/O handler is implemented as a logical 'blkback' handling port in the MicroVisor virtual block switch (VBS). The VBS is responsible for implementing the ATA over Ethernet client side

logic which involves forwarding block requests as raw Ethernet frames to the correct destination MAC address, processing responses from the appropriate ATAoE server, and handling any block request retransmissions in the event of packet loss between the VBS and the ATA over Ethernet server.

The blkback component processes block I/O requests handed to it by the virtual machine paravirtual device driver over a shared memory I/O ring data structure. The VBS then takes a block request and inserts it into an MTU-sized ATA over Ethernet frame and transmits it to the destination ATA over Ethernet server. Once the acknowledgement frame is received from the server, the virtual block request can be completed for the virtual guest domain. If an acknowledgement is not received within a very short timeout interval, the request can be retransmitted up to a configurable number of times, *N*. In the unlikely event that the block I/O request is not completed after *N* times, it is eventually failed returning an error to the guest. An outcome such as this would only occur in the event of a network partition, server or drive failure.

The simplicity of the protocol and communication between the embedded VBS component, the virtual network switch and the remote ATA over Ethernet server helps to build the lowest latency I/O path possible with an extremely small packetisation overhead. Converting block requests natively to Ethernet frames also provides great flexibility in the addressable destination location as frames can be forwarded to either a local storage driver domain instance, a remote Ethernet connected storage server or a MicroVisor to MicroVisor virtual encapsulated network path remote storage driver domain. In short, the ATAoE client direct integration into the VMM layer provides all the flexibility of network attached block storage with the performance of native drive access.

Results

In the interests of providing a fair comparison of measurements but at the same time highlighting one of the key benefits of the integrated I/O path in the MicroVisor VMM layer, we present here results taken from a VM residing on different hypervisor types: Xen (CentOS6, 3.10.20 Linux kernel), KVM (CentOS6, 2.6.32 Linux kernel) and the MicroVisor as is being developed by the OnApp development team. The assumption when we began the evaluation was that the most useful measurement was to compare network attached storage over a TCP/IP link vs the MicroVIsor ATAoE network attached storage technology as this is a fundamental requirement for workload mobility. Furthermore, we measure the effect of accessing the same storage over both a local network forwarding path (i.e. co-located storage appliance VM) vs a remote forwarding path (i.e. remote located storage appliance VM). The early MicroVisor results demonstrate a great deal of potential, particularly bearing in mind that there is still significant forwarding path optimisations to be made.

In Figure 20 we compare the performance results of raw device read rates against the virtualised storage performance of a volume manager accessed over the TCP/IP network path (Figure 21 compares write performance). It is important to note that this is the standard method of accessing networked storage in a virtualised environment as it is a critical property for supporting fluidity of workloads. The raw storage drive benchmark is included here for illustrative purposes as it helps to highlight what percentage of the best case raw measurement can the virtualised network block I/O path achieve. The improved performance of the MicroVisor over the KVM and Xen hosts utilising the

Network Block Device protocol confirms that the TCP/IP processing overhead for virtualised, network storage becomes a limiting factor for performance when handling small packets.

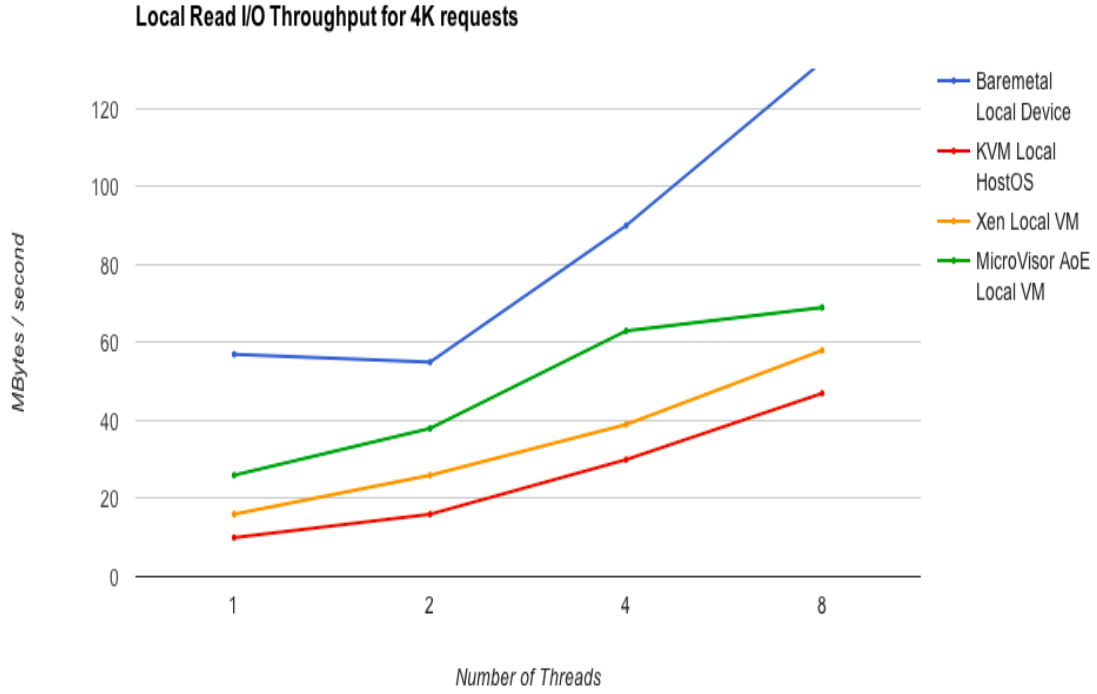


Figure 20: Local read performance of storage as measured from a VM relative to raw for varying amounts of outstanding I/O

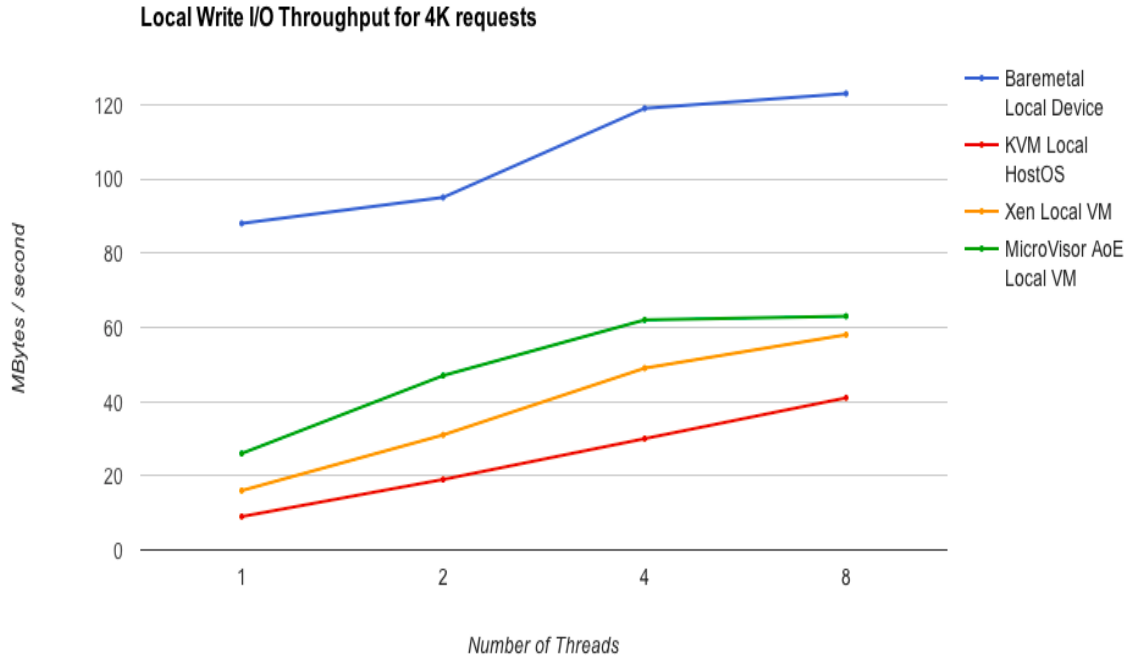


Figure 21: Local write performance of storage as measured from a VM relative to raw for varying amounts of outstanding I/O

Utilising ATAoE we note a significant performance advantage for small 4KB packet sizes when compared against the more traditional network attached storage target. The read performance comparison over the physical network link can be seen in Figure 22 with the write performance in Figure 23.

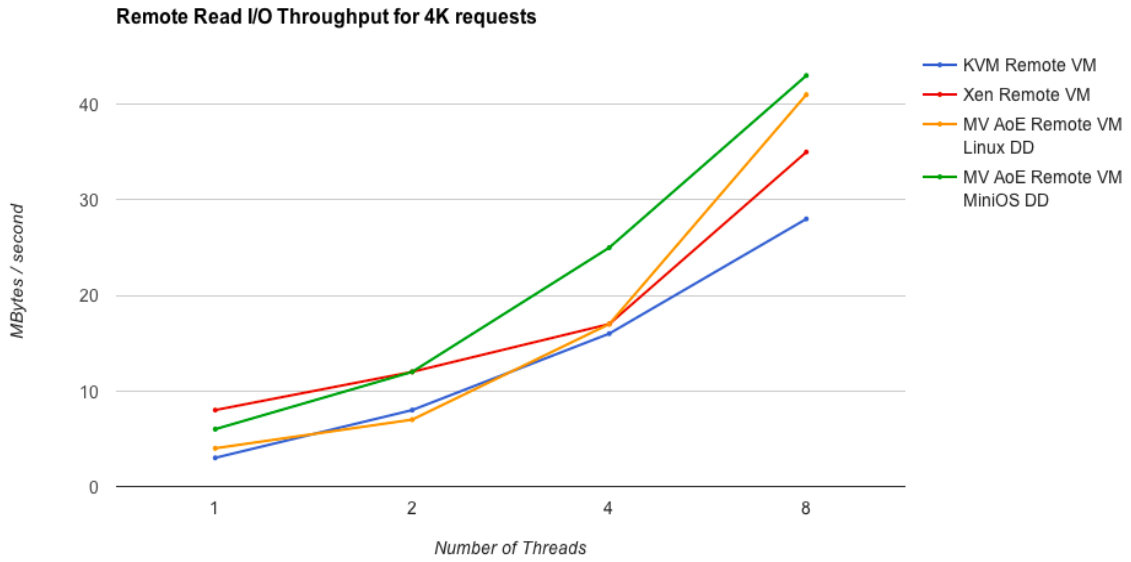


Figure 22: Remote read performance of storage as measured from a VM relative to raw for varying request amount of outstanding I/O

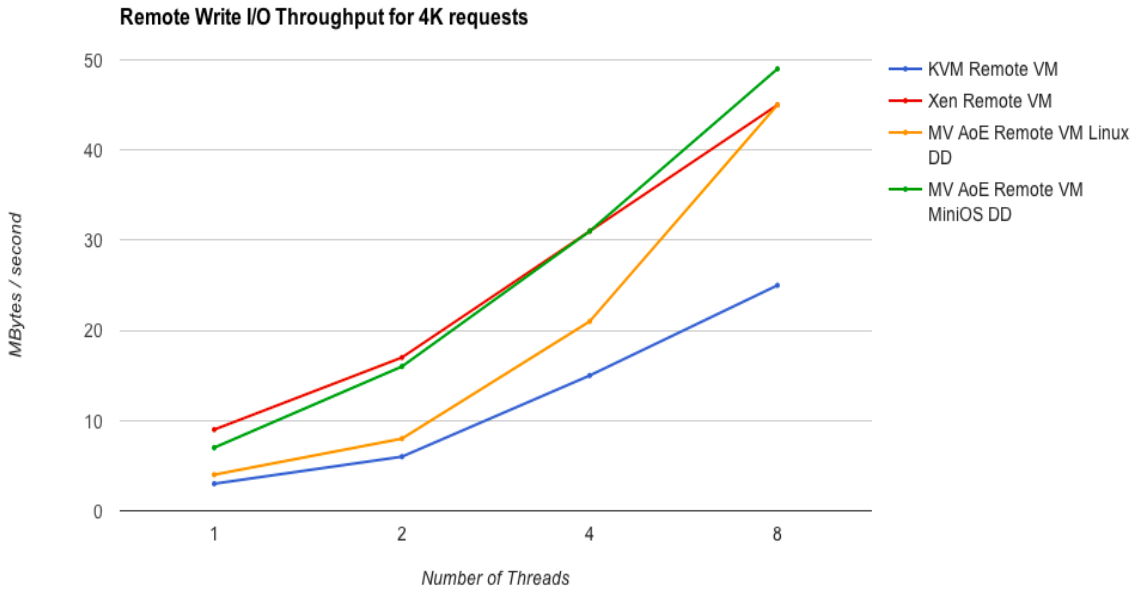


Figure 23: Remote write performance of storage as measured from a VM relative to raw for varying amount of outstanding I/O

In the remote measurement case we note that all results are much closer due to the dominating effect of the physical network transmission medium. The ATAoE path fairs well, particularly over larger thread numbers (amount of outstanding I/O) but not as performant as stock Xen yet for the 1

thread, 4K block measurement. This is believed to be due to some inefficiencies in the packet striping algorithm in the current MicroVisor implementation where multiple links are presented to the VMM layer for bandwidth aggregation. We expect to resolve these discrepancies quickly and move the ATAoE performance significantly higher, closer to raw drive performance. Note however that as the outstanding I/O level increases the ATAoE protocol implementation begins to outperform all other platform measurements.

4. Storage virtualisation (ONAPP + FORTH)

Storage is often considered a persistent form of memory that is used for data that needs to be preserved, long after the initial process that created them has expired. Information is preserved to one or more media using different mechanisms relevant to the type of media. Figure 24 characterises various memory technologies in terms of price and bandwidth. SSD would roughly fit in the area highlighted by NAND in that graph and HMC is promising the bandwidth of L3 memories for a price similar to that of DRAM. The increasing need for permanent storage has been highlighted by a number of research studies including a report (in 2012) by IDC[4] that suggests a growth from 7.91 ZB of global digital data in 2015 to 40 ZB by 2020.

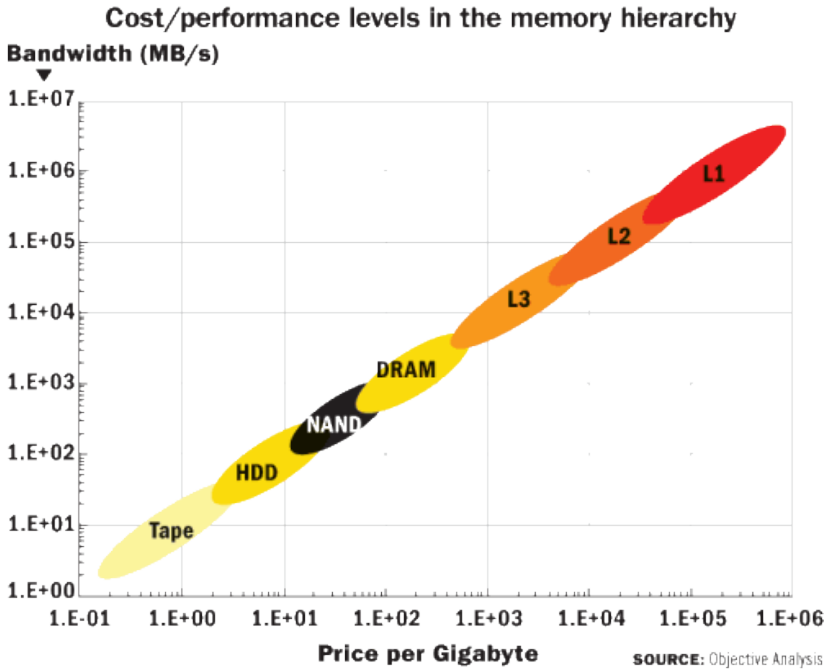


Figure 24: Memory hierarchy price and bandwidth for different technologies[5]

Supporting this growth only encourages the vendor market in its pursuit of achieving increasingly greater bandwidth to storage devices to meet the demands for storage I/O. Intelligent mechanisms are therefore needed to improve the accessibility of storage, especially when distributed storage systems are becoming common place. Given that only a small proportion of data is regularly accessed (a fact that is taken advantage of by CPU caches), intelligent caching mechanisms can be used for

[4] *Expected growth of digital data until 2020 - source (IDC) Digital Universe Study, sponsored by EMC, December 2012*

[5] Original source Objective Analysis, 2012. Image from http://goo.gl/RShlnV

providing faster access speeds whilst keeping the overall platform costs and TCO manageable. This is extremely relevant and in fact considered to be an essential requirement in the EUROSERVER architecture, particularly for resource constrained environments in which storage accesses are likely to be often remote over limited network throughput links. Significant work therefore has been conducted in EUROSERVER to measure and integrate system level content caching (described in Section 4.1). As this is highly relevant to the MicroVisor platform, which utilises the OnApp distributed storage platform, known as Integrated Storage the following information presented specifically compares the effect of caching enabled and disabled on the Integrated Storage platform with a view to enabling this support natively on the MicroVisor. Improvements in shared memory access that are also applicable to the storage domain have been discussed in more detail in Section 3.3.

4.1. Non Volatile Memory storage / Caching (ONAPP + FORTH)

Introduction

The goal of this work on block I/O caching is to reduce the access times to persistent storage and help to consume less energy overall by storing and retrieving data from a local drive close to the owning VM workload, thereby reducing the dependency on network packet transmissions. While there are many ways to improve the I/O performance of a storage system, in order to provide a consistent platform for comparison, we examine the benefits of the caching technology applied against the OnApp cloud storage architecture that is shown in Figure 26. To better understand the effect of caching we first describe the architecture of the system.

Virtualised workload infrastructure can use local direct attached storage drives on each hypervisor for VM workload data, however this is ineffective in a production system since there is no opportunity for fault tolerance to fail over a VM in the event of a system failure or to support proactive virtual machine migration. Multi-node hypervisor environments therefore must utilise some shared storage resource that can be accessed by any hypervisor at any time. There are two common approaches to this:

a) Utilising a centralised storage disk array as a SAN or NAS target. Virtual disk storage is accessed over a network-attached iSCSI LUN or NFS partition. This requires a generic Ethernet/IP/TCP stack on both ends of the network path (hypervisor client side and storage target server side) in order to access the data.
b) Utilising a hyper-converged methodology where all resources are distributed across the hypervisor infrastructure and shared as a cooperative service. Also referred to as Software Defined Storage, this approach often provides efficiencies both in cost (utilises commodity storage drives accessed via the host OS stack) and in performance (storage locality can be managed for the VM to ensure that a local copy of data is provided). Note that this only applies to data READs and not for data WRITEs which require synchronous distribution of copies in order to maintain data availability.

The OnApp cloud stack provides a choice in storage deployment to accommodate both options a) and b) above. The hyper-converged architecture as described in option b) is fast becoming the most

popular deployed architecture on all new OnApp cloud infrastructure. The OnApp cloud Integrated Storage (IS) architecture provides a running VM's data to be *replicated* on one or more hosts such that in case of failure there is a set of hosts that the VM can run on. Specifically this is achieved by replicating each write I/O request to the other replication hosts *before* completing it. Read I/O requests are served from the local disk, avoiding the potentially expensive network trip. Therefore there is an *asymmetry* between read and write I/O requests.

This design works well because it achieves functionality similar to the one offered by a SAN without incurring its high financial cost. On the other hand, a VM's perceived disk I/O performance is affected. In particular, converting local write I/O requests to remote, synchronous network writes significantly increases write disk I/O latency. In this project we focus on mitigating the penalty of these synchronous writes I/O requests. While overall storage I/O performance can be improved simply by using faster local disks or a faster network, we examine less efficient but more cost-effective solutions based on a localised cache SSD device as illustrated in Figure 27.

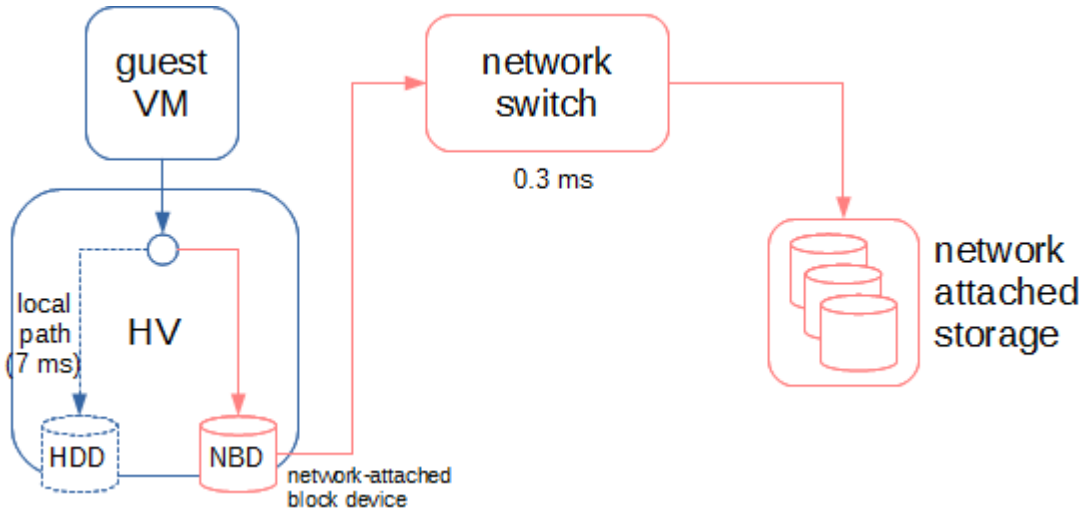


Figure 25: Traditional SAN storage access architecture

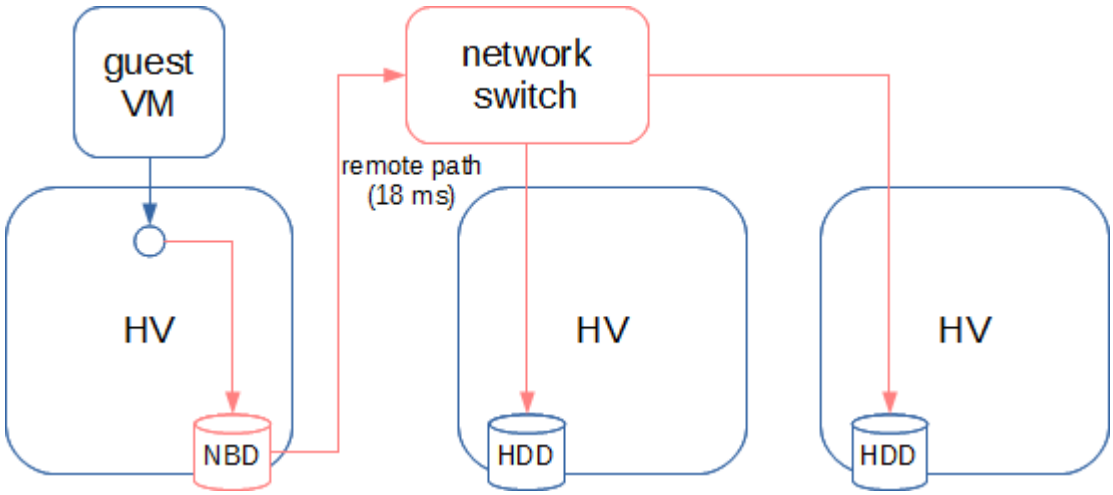


Figure 26: Hyper-converged storage access architecture

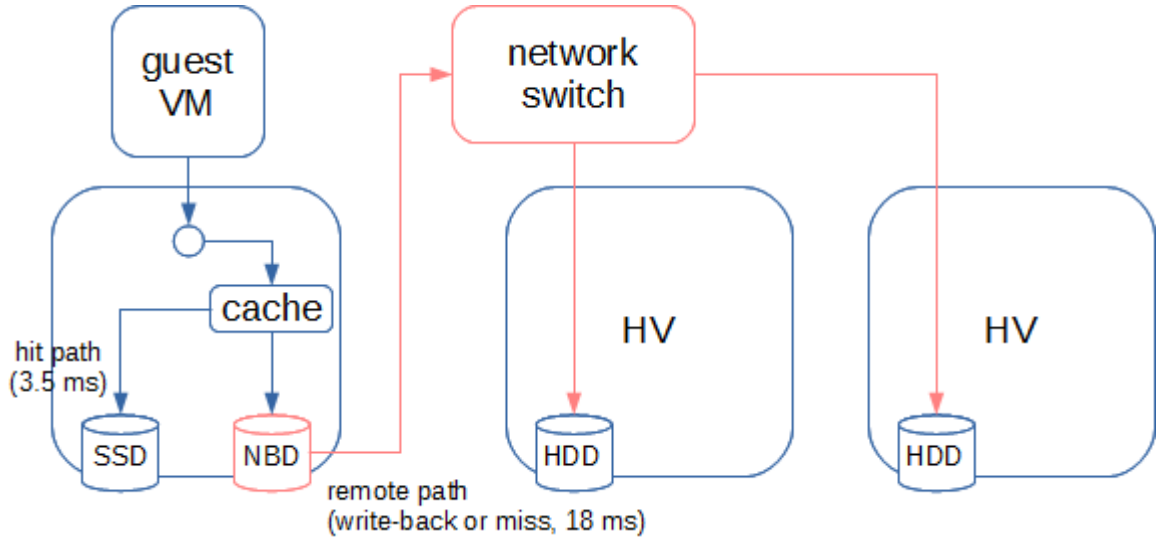


Figure 27: Hyper-converged system architecture with a local cache block device

One way to reduce the latency of a synchronous write I/O request is to defer it to a later time when the network will be less saturated or enough write I/O requests can be batched to create more throughput-efficient network requests. Deferring the requests must be done in a persistent way otherwise the remote VM data disk replicas will lag, so DRAM-based solutions are precluded. Solid state disks (SSD) offer much lower latency compared to traditional rotating hard disk drives (HDD) and are an excellent candidate for persistently buffering write I/O requests. SSDs have a significantly higher cost per capacity ratio compared to HDDs and they have a higher failure ratio, so their use as primary storage is still prohibitive. Due to the low latency characteristics, SSDs can be an excellent candidate for buffering write I/O requests:

- Their exceptionally high read I/O performance can mitigate the performance penalty of reading the buffered write I/O requests for sending them to the remote replicas.
- Their small size is not a concern because only a limited number of write I/O requests needs to be buffered, the whole workload does not need to fit in the SSD.

The drawback of buffering write I/O requests is that if the host fails then there is no up-to-date remote replica on any host that would allow the VM to be restarted on. Trading the ability to run a VM on any host against disk I/O performance can be a sacrifice some customers are willing to make. This feature can be configured on a per-VM basis.

Recently a number of open-source solutions that employ SSDs on top of HDDs as I/O *caches* have been developed. We can achieve our desired buffering behaviour by configuring such a cache in write-back mode. These existing solutions perform well enough that, in EUROSERVER, instead of devising our own implementation we focus on integrating such an open-source solution into the Integrated Storage architecture. There do exist hardware solutions of the same architecture as hybrid disks, however they are unusable because they sit far too low in the storage I/O stack, making them unsuitable.

There have been a few open source implementations that achieve caching using SSDs:

- dm-cache (upstream)
- bcache: requires the virtual device to be formatted
- dm-cache (non-upstream)
- FlashCache
- EnhanceIO

We choose dm-cache (upstream) over the other solutions because it is the de facto SSD cache implementation in the Linux kernel and because other solutions have substantial disadvantages: bcache requires the HDD to be re-formatted which complicates integrating it in the existing Integrated Storage architecture and it is not stable enough. dm-cache (non-upstream), FlashCache (based on dm-cache non-upstream), and EnhanceIO (based, in turn, on FlashCache) are not actively maintained.

System Architecture

In this section we examine the Integrated Storage architecture, the open source dm-cache solution, and how we integrate dm-cache into Integrated Storage.

Figure 28 illustrates in detail the positioning of dm-cache in the existing architecture, focusing on the affected part of the disk I/O datapath. dm-cache is a virtual block device driver in the Linux kernel that sits on top of the HDD and SSD and intercepts requests routing them to the appropriate device. The fact that dm-cache comes as a layered device mapper block device makes it very easy to integrate as it is essentially just another block device in the I/O pipeline. dm-cache offers three caching modes:

- write-through: read I/O requests can be cached, write I/O requests are never cached
- write-back: both read and write I/O requests can be cached
- pass-through: neither read nor write I/O requests are cached.

The write-through mode improves performance by placing frequently-read data on the SSD. Whenever such a piece of data is re-read the expensive I/O read from the HDD is skipped. The write-back mode further improves I/O performance by buffering write I/O requests on the SSD and writing them back to the HDD at a later, more convenient time. Finally, the pass-through mode is used to temporarily disable caching for whatever reason; it does not affect disk I/O performance.

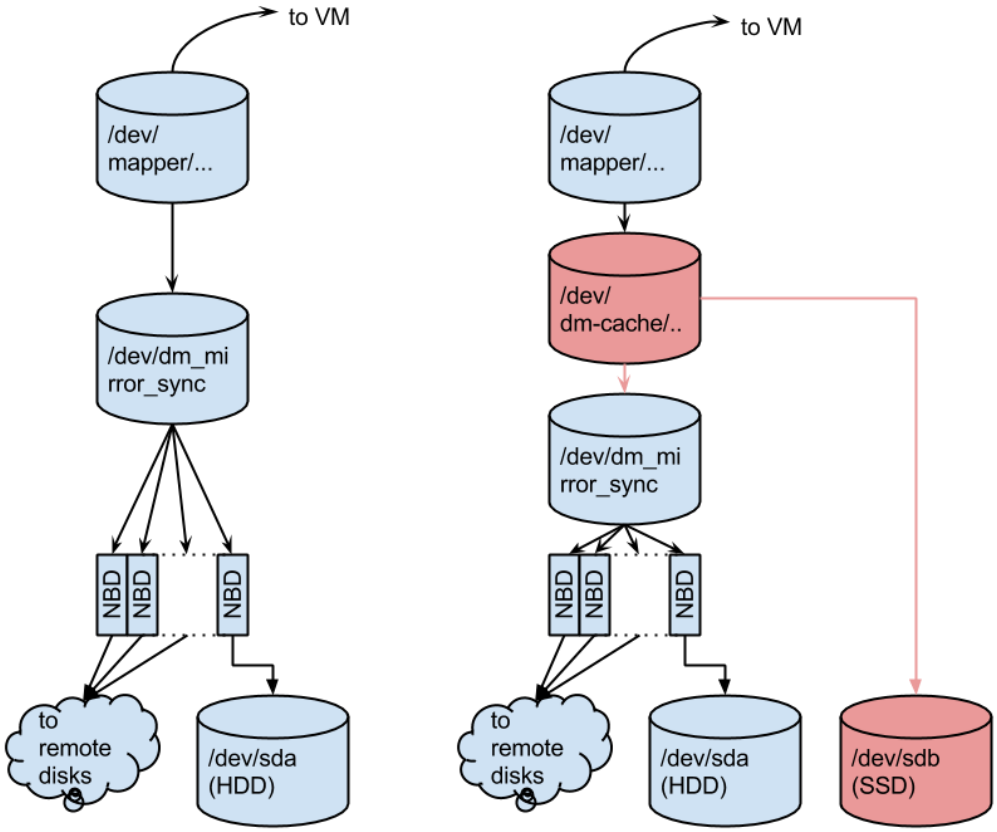


Figure 28: dm-cache in Integrated Storage

Integrating dm-cache into the datapath is relatively straightforward. However, when write-back caching is used the control plane is substantially affected, because in the case of host failure the remote replicas would be out of sync, so they should not be used. Also, taking a snapshot of, migrating, or shutting down the VM require any pending data to be flushed and the cache to be switched to write-through mode before proceeding.

The main reason for placing dm-cache on top of the kernel driver responsible for synchronously replicating write I/O requests is that the data-path semantics remain the same as before: if write I/O request completes then *all* of the replicas contain the same data. Placing dm-cache anywhere else on the data-path would unnecessarily complicate semantics and break the existing assumption without any actual benefit in return.

Another advantage of dm-cache is improving disk I/O performance when a VM is run on a host that does not have a local replica of the VM's data. In this case local caching can substantially increase disk I/O performance as it reduces the number of remote read I/O requests, even when write-through caching is used.

Preliminary Performance

In this section we examine how dm-cache improves disk I/O performance. We used flexible I/O tester (fio[6]) natively on the host with a 100% read or 100% write pattern utilising both sequential and random I/O patterns. Since the proportionality of results varied very little between both sequential and random, we compare only the random I/O benchmark results below. In the interests of simplifying the benchmark to highlight the most important characteristics of the caching effect, the SSD cache is provisioned to be large enough to fit the entire workload. The cache is warm and the cache line size is 64 KB. The first set of performance results can be seen in Figure 33, focusing in particular on read path behaviour. The results demonstrate significant benefit in applying a local cache target, particularly for the larger request size values as the block size matches the cacheline size. It is assumed that the performance of the cache could be improved for the smaller 4 KB block requests if the cacheline size was reduced to the minimum supported value of 32 KB, however note that in general this is not considered a valuable optimisation for average workloads.

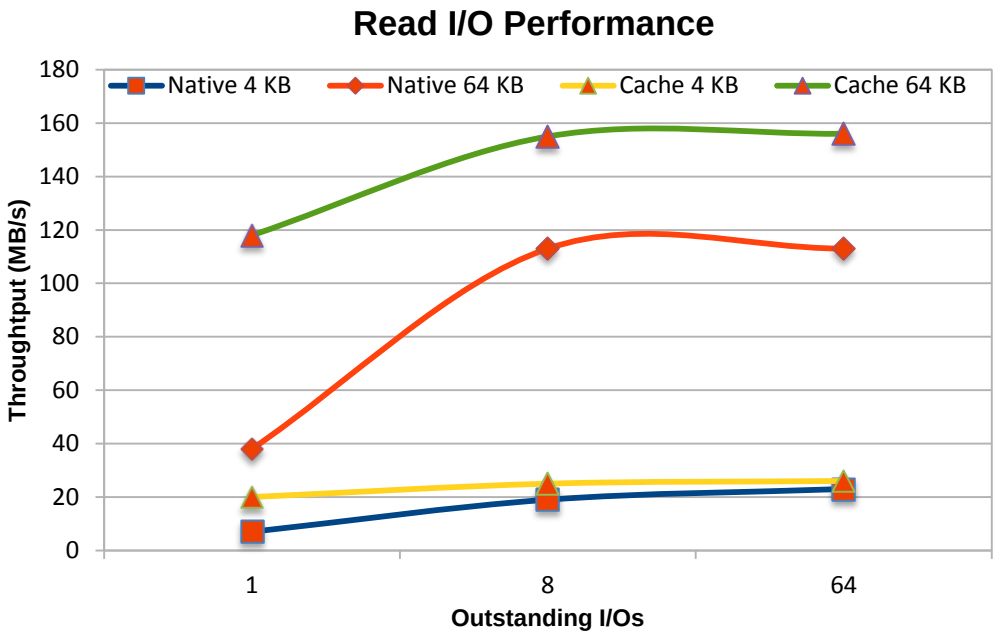


Figure 29: Read I/O performance of different block sizes for varying levels of outstanding I/O requests

Moving on to the write path, we can see in Figure 30 that the write path is even more significantly affected by applying a local cache target, particularly at the 64KB blocksize and above. For the results shown here, the cache flushing was disabled entirely to demonstrate the benefits of a 100% cache target hit for writes vs no cache. In a practical system deployment it is important to tune the cache to decide how fast the dirty blocks should be migrated which is really a personal deployment choice as to how much risk a system admin is prepared to tolerate. The trade-off is clear in that the slower the migration rate of the cache, the faster the live I/O stream performance but with a higher loss risk in the event of a crash or cache server failure in which the dirty cache data may become unavailable. The effect of disabling the flushing of dirty data is illustrated in Figure 31.

[6] https://github.com/axboe/fio/

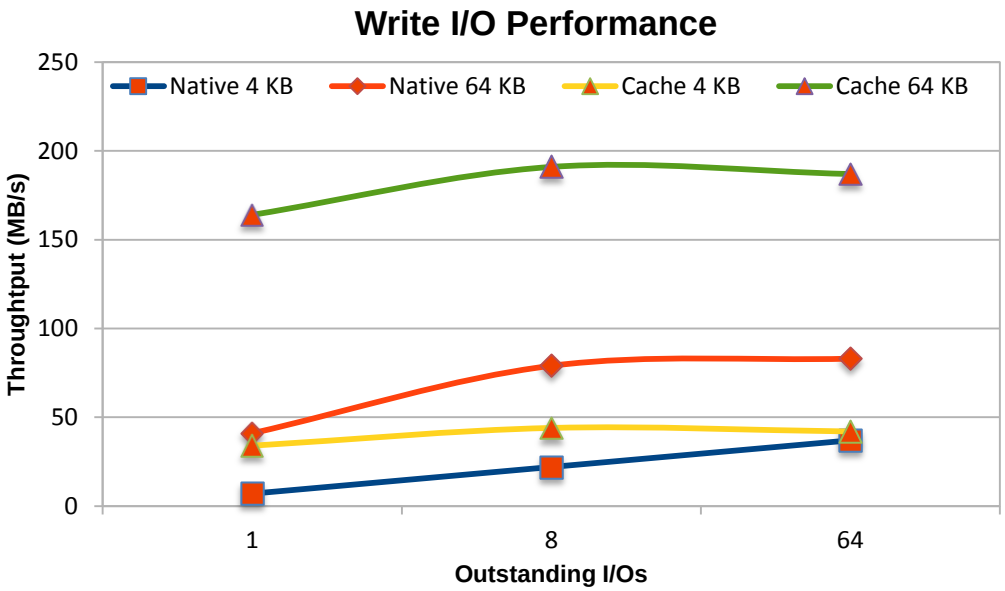


Figure 30: Write I/O performance for varying levels of outstanding I/O requests

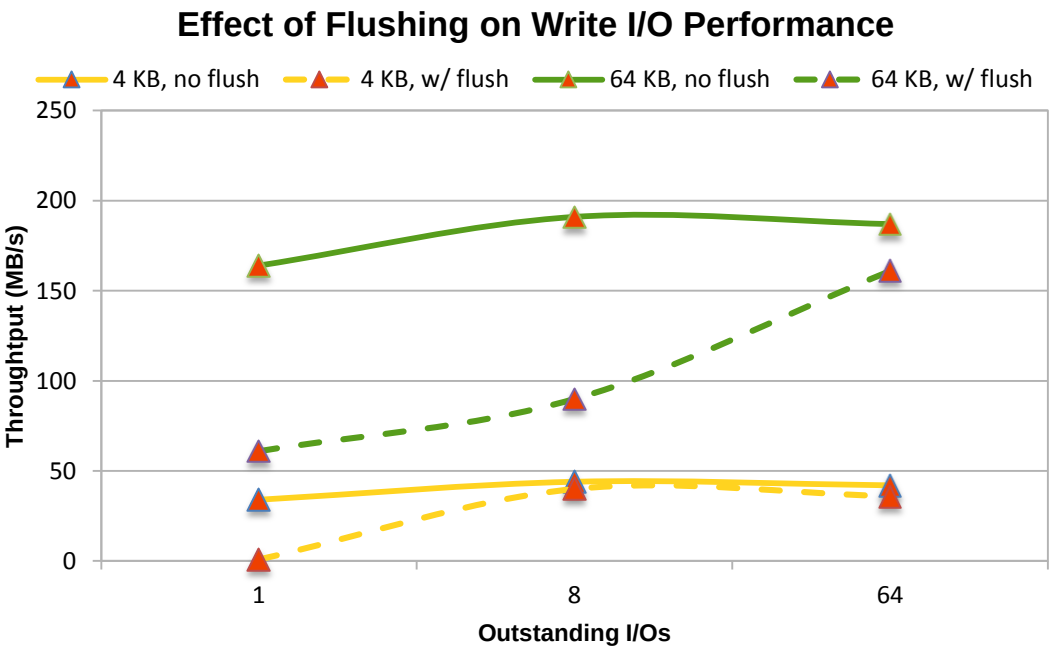


Figure 31: Effect of flushing on write I/O performance for varying levels of outstanding I/O requests

Additional effects of the cache that are intuitively true and have been verified independently from the results presented above are that:

1. The raw cache target (once warm) imposes very little overhead in the best case read baseline measurement.

2. Writing data where the writeback (data migration threshold) is completely disabled demonstrates that writing into the local cache device achieves something approaching the same performance as the raw disk baseline measurement.

Performance of dm-cache is significantly higher than native because dm-cache converts random write I/O requests to sequential ones. This is an artefact of how sequentialisation improves storage performance for sequential reads and writes compared to random I/O workloads. The actual performance, given sufficient load and time is expected to deteriorate to just under the native SSD speeds. For aggregate VMs, the speed of a cache backed HDD though is expected to be greater than that of just HDD.

Comparison of SSD only implementation vs platform that uses caching

To analyse the performance improvement on a more realistic production environment, a comparison was made between an SSD only solution and a system that used a combination of hard disk drives backed with SSD caching.

Each server had dual Intel Xeon CPUs, 60GB RAM, Dual 10GbE, Quad port 1GbE and a MegaRAID SAS controller. The platform was tested with one Control Panel server, three KVM hypervisors and one Backup Server. The SAN network was connected via one of the 10Gb ports on each hypervisor (HV) and used an MTU of 9000. The management network used one of the 1GbE interfaces. The network topology can be seen in Figure 32. The datastore on Integrated Storage was set up to use two replicas and two stripes. Of six drives on each HV, two were used for the cache target and four for the persistent datastore.

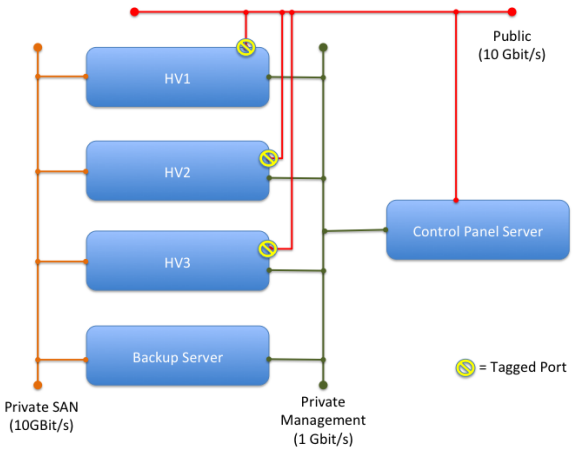


Figure 32: Network topology for testing caching performance

The tests were run for over 48 hours with a variety of workloads running on them. The I/O performance test results are captured in Table 2. The performance results indicate that although there is an expected hit to the maximum write speed (synched across the network), the read speed is roughly the same as the I/O for the SSD only solution.

Table 2: Caching performance of SSD only solution vs Integrated Storage

	READ	WRITE
Raw Drive Baseline (2 SSDs, RAID-0)		
Max 4KB IOPS	156K	80K
Max Throughput	1090 MB/s	780 MB/s
Integrated Storage measurements (caching enabled)		
Max 4KB IOPS	128K	114K
Max Throughput	1115 MB/s	564 MB/s

In conclusion therefore we find that deploying a localised cache target on a fast storage device such as an SSD helps to significantly reduce I/O latency by avoiding a network round trip time for every I/O request, and also to improve the performance and throughput as experienced by an application that is accessing the storage device. Based on these observations, and the context in which the cache technology is being evaluated, we can further assume that accessing storage locally rather from a remote device that involves network transmission support helps to reduce the energy overhead incurred in reading from and writing to persistent storage. This becomes particularly significant in low power systems such as the EUROSERVER platform where resources must be metered and controlled at a very low level in order to achieve the sorts of performance vs energy consumption trade-offs envisioned in the architecture.

5. I/O Virtualisation acceleration (CEA)

VM Guests not only share the CPU and memory resources of the host system, but also the I/O subsystem. Because software I/O virtualisation techniques deliver less performance than bare metal, hardware solutions that deliver almost native performance have been developed recently.

One of them is direct assignment: When using direct assignment, instead of using software emulation or paravirtualisation techniques as described in Deliverable D4.1, we can allow the guest OS to directly access the peripheral using its native device driver. The result is a significant gain in performance, almost reaching native, since VM exits to hypervisor are no more needed to emulate the device.

Direct assignment provides compatibility, since the guests can use their native driver and performance, since it does not require hypervisor intervention to process I/O requests.

However, in order to support direct access, each device requires the presence of an IOMMU in front of it, as shown in Figure 33. In effect, this component is necessary to provide VM integrity and isolation as well as address translation. [2]

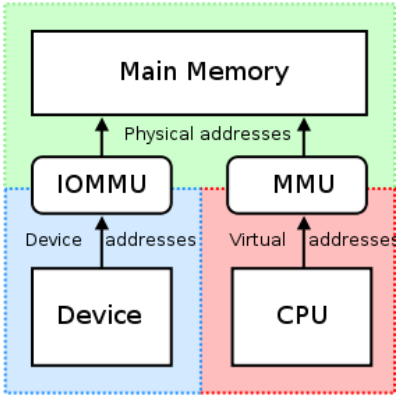


Figure 33: Direct memory access of guest VM to peripheral using IOMMU

Direct assignment on its own does not allow a device to be shared among several VMs. For this reason the technique is usually associated with other technologies such as SR-IOV or any other multi-channel interface technology.

5.1. PCI-SRIOV introduction and VFIO (CEA)

SR-IOV [1] is a technology that allows a single PCIe (PCI Express) device to emulate multiple separate PCIe devices. The emulated PCIe functions are called "virtual functions" (VFs), while the original PCIe device functions are called "physical functions" (PFs). SR-IOV is typically used in an I/O virtualisation environment, where a single PCIe device needs to be shared among multiple virtual machines.

A typical example is its use in Network Interface Cards (NIC): SR-IOV network cards provide multiple Virtual Functions (VFs), up to 64, that can each be individually assigned to a guest virtual machine using PCI device assignment, as demonstrated in Figure 34. Once assigned, each will behave as if it were a full physical network device. This permits many guest virtual machines to gain the performance advantage of direct PCIe device assignment, whilst only using a single slot on the host physical machine.

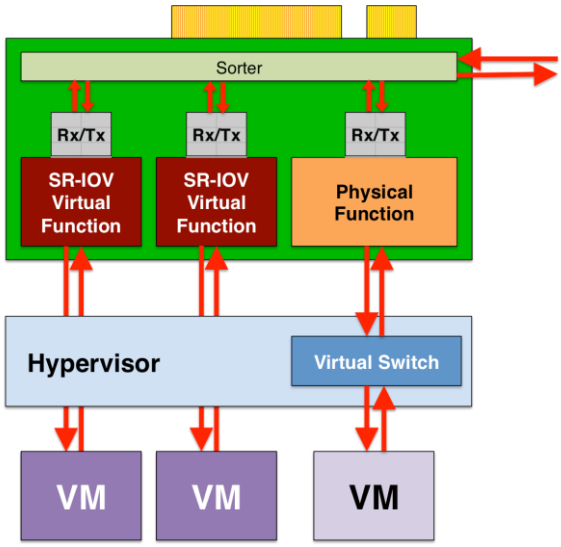


Figure 34: SR-IOV NICs provide multiple Virtual Functions that can be directly assigned to guest VMs

As mentioned earlier, the direct device assignment of Virtual Functions requires an IOMMU (MMU for I/O devices) to perform isolation and address translation in a transparent manner from the VM's point of view.

The hypervisor or more precisely the controller of the hypervisor (Dom0 when using Xen, Qemu when using KVM) will be responsible to configure this IOMMU accordingly to the selected VF and VM. In the case of the MicroVisor a specialised controller VM will be used for interacting with the IOMMU.

When using the Linux/KVM hypervisor, Qemu will perform this configuration through a new user-level driver framework for Linux: VFIO (Virtual Function I/O) [9][10][11].

This VFIO driver is an IOMMU/device agnostic framework for exposing direct device access to userspace, in a secure, IOMMU-protected environment. In the context of KVM, this is a replacement of the KVM PCIe specific device assignment, which was limited to PCI and x86 only, and had no notion of IOMMU granularity.

Compared to KVM PCI device assignment, the VFIO interface has the following advantages:

- Resource access is compatible with secure boot (e.g. TrustZone security model)
- The device is isolated and its memory access protected
- It offers a user-space device driver with a more flexible device ownership model
- It is independent from KVM technology, and is not bound to the x86 architecture only, which is of interest in the context of our ARM64 ecosystem

Summing up, the VFIO driver exposes direct device access to user space in a secure memory protected environment, and can either be used in a virtualisation environment (considering the VM as a user space driver from the device and host perspective) or a low-overhead direct device access from user space as we can find them in high performance computing field (e.g. TCP/IP kernel bypass and user space NIC drivers).

5.2. Virtual Function IO (VFIO) for non PCI devices (CEA)

In contrast to the x86 platform, where PCIe is used as the main interconnect, ARM platforms are generally designed around an on-chip interconnect (e.g. Cache Coherent Interconnect CCI-400/500) or a network-on-chip (e.g. Cache Coherent Network, CCN-50x). A major difference between PCIe interconnects and on-chip interconnects is the configuration interface: the on-chip interconnects generally do not provide an auto configuration interface like PCI, so they require other mechanisms, such as the Linux "device tree" to provide system information (memory map, interrupts, device type, etc.) to the Linux kernel at boot time. In Linux terminology, those non-PCI devices are named "platform devices", in the sense that they typically appear as autonomous entities in the system. This includes legacy port-based devices and host bridges to peripheral buses, and most controllers integrated into a system-on-chip (SoC) platform.

In order to support VFIO on ARM SoC, recent work has been performed by the Linaro team[12][13]: the VFIO framework has been enhanced to support "non PCI" platform devices (VFIO_PLATFORM) and extended to support IOMMUs found in the ARM ecosystem (ex: MMU-400, MMU-401, MMU-500).

Using the recently purchased Juno (r0) ARM boards, we were able to validate the VFIO framework by using a small user space program driving the DMA controller trough VFIO, and performing some memory copies using user space virtual addresses [14]. This experimentation allowed us to validate VFIO on the ARMv8 architecture (which has not yet been performed by the Linaro team), but also to confirm that the selected hardware configuration (SMMU + CCI-400) which is almost identical to the one found in the EUROSERVER chiplet architecture is able to conform regarding the cache coherency aspects, as well as in a SMMU multi-context configuration.

Juno boards contain :

- A dual-core ARM Cortex-A57 cluster
- A quad-core ARM Cortex-A53 cluster
- A quad-core ARM Mali-T624 cluster
- A Cache Coherent Interconnect (CCI-400)
- A DMA controller (PL-330) and its associated IOMMU (sMMU-401)

Without entering into details, to perform this user space VFIO experiment, we had to:

- Add the SMMU devices in the Juno device tree source (juno.dts) along with proper mmu-masters, i.e DMA device with various MMU StreamID values (see MMU spec [3], and Juno spec [5])

```
smmu0: smmu@7fb00000 {
       compatible = "arm,mmu-401";
       reg = <0x0 0x7fb00000 0x0 0x10000>;
       #global-interrupts = <1>;
       interrupts = <0 95 4>,
                    <0 95 4>;
```

```
        mmu-masters = <&dma0 0x0 0x1 0x2 0x3 0x4 0x5 0x6 0x7 0x8>;
}
```

- Add the DMA pl-330 device in the Juno device tree source, and ensure that the DMA and SMMU configurations guarantee the I/O cache coherency.

```
dma0: dma@0x7ff00000 {
    compatible = "arm,pl330", "arm,primecell";
    reg = <0x0 0x7ff00000 0 0x1000>;
    interrupts = <0 88 4>,
             <0 89 4>,
             <0 90 4>,
             <0 91 4>,
             <0 108 4>,
             <0 109 4>,
             <0 110 4>,
             <0 111 4>;
    #dma-cells = <1>;
    #stream-id-cells = <9>;
    #dma-channels = <8>;
    #dma-requests = <32>;
    clocks = <&soc_faxiclk>;
    clock-names = "apb_pclk";
    dma-coherent ;
};
```

- Fix some default security settings in the SMMU SSD (Secure State Determination) Registers for the PL330 manager channel.
- Ensure that the caching property of the DMA engine was set to outer cacheable, as we observed that the SMMU can overwrite cacheability attributes but only to reduce the cacheability level/strength. In other words, if the DMA engine is programmed to generate non-cacheable/non-bufferable transactions, cache coherency will not be guaranteed whatever are the SMMU TLB memory attribute settings.

This experiment of DMA transactions between two user space addresses, will also be performed on EUROSERVER RTL using hardware emulation, and is crucial from a SMMU/DMA integration verification perspective.

This VFIO framework will be used for the shared NIC between nodes and between VMs. Considering a maximum of one VM per core (to avoid inefficient contexts switching), the shared NIC should support a minimum of 32 MAC addresses and DMA channels.

Since the shared NIC is located in the FPGA at board level integration, and is currently under definition, we mainly focused on the inter-chiplet communication path: the RDMA. As we will explain in the following section, the IOMMUs and its software integration inside the Linux kernel (MMU API and VFIO) are key components to take benefit of the EUROSERVER's UNIMEM architecture.

5.3. Application to inter node communication: RDMA (CEA)

Traditional socket-based networking technologies are limited by copies on the data path, which not only increase latency, but also add considerable processing overhead. One decade ago, the remote direct memory access technology (RDMA) was developed by the high-performance computing (HPC) community to provide ultra low-latency and high throughput on top of InfiniBand infrastructure. RDMA removes copies from the data path and it enables direct isolated application access to RDMA-capable network interfaces (RNICs). This allows data path operations to be performed without involving the operating system, thus keeping network computation overhead to a minimum.

The EUROSERVER system architecture exhibits a “to remote” and “from remote” link on each chiplet (a.k.a. “node”) which, once tightly coupled with an integrated IOMMU, will allow RDMA transaction between nodes.

Two approaches are studied regarding the integration of RDMA into existing applications:

- The sockets over RDMA approach
- The integration of RDMA over NoC in the standard RDMA stack in Linux

The latter approach is supported by the Open Fabric Alliance Enterprise Distribution (OFED [15]), the porting of which will be discussed in the following section.

5.3.1. OFED porting (CEA)

Introducing RDMA and InfiniBand terminology

Remote direct memory access (RDMA) is a network technology that enables low-latency and high-throughput data transfer between network nodes with minimal processing overhead.

This is achieved by allowing an RDMA-capable network interface (RNIC) also called a Host Channel Adapter (HCA) to directly access application buffers, even when running within the OS virtual address space. As opposed to socket-based networking, where the OS kernel has to perform copies of the application buffers, applications can be given direct and isolated access to the HCA, allowing data transfers to be performed without any operating system intervention.

RDMA supports various types of transactions:

- Two-sided Send and Receive operation: in this case both emitter and receiver nodes are involved in the transaction with their respective send and receive buffers
- One-sided RDMA read or write operation: in this case the initiator node application can read or write from/to a remote memory buffer located in another node, without any involvement of the application or the operating system on the remote side.

This second type of operation is of particular interest as it totally fits the EUROSERVER “UNIMEM” architecture, and it is much more efficient in term of bandwidth efficiency and latency.

HCA are typically programmed by RDMA Verbs, which are an abstract definition of the functionality provided by an adapter, in other words they describe the semantics of RDMA operations.

Verbs include work queues, completion queues, registration of memory, and how they interact with each other.

Verbs have been specified by the InfiniBand Trade Association (IBTA [17]) and the RDMA Consortium [16].

RDMA operations specified by Verbs are always performed on memory buffers, which have to be registered once before they can be used, to make them uniquely identifiable (using a key) and enable direct data access. Once associated with a memory buffer (i.e. an offset), the key is used to identify a local or remote memory location in a data transaction.

Verbs also specify the network resources involved in the data path operations.

There are two types of resources: work queues, which are used to queue work requests from the applications, and completion queues, which are used to inform about completed work.

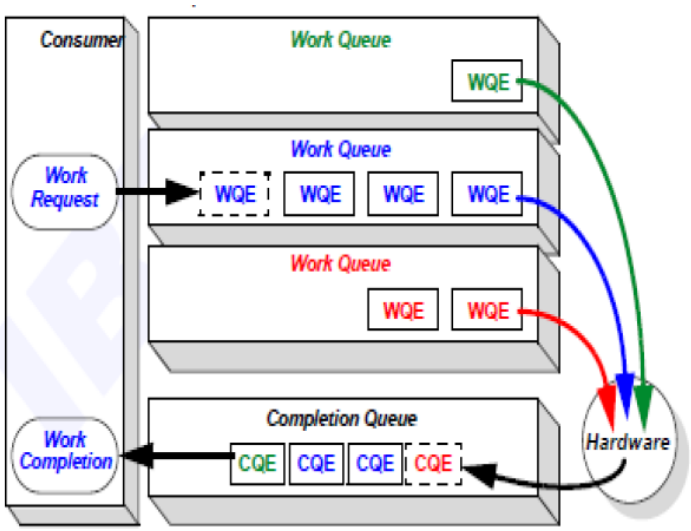


Figure 35: The two types of resources; work and completion queues in Infiniband

For each connection there are two queues: a send queue and a receive queue, forming one queue pair (QP). The QP number associated with the node ID uniquely identifies the connection.
For each pair of work queues, there can be an associated completion queue (CQ). If required, a work request can generate a work completion event when (successfully or not) completed.

Table 3: RDMA over InfiniBand hardware; operations (verbs) and descriptions

Create QP	create a send and receive queue pair
Modify QP	change internal state of QP
Destroy QP	Destroy a QP
Create CQ	create a completion queue CQ

Poll CQ	polls for work completions on a CQ
Req notify CQ	request completion event on new work completions
Destroy CQ	Destroy a CQ
Reg User MR	Register a user memory region (MR)
Dereg User MR	deregisters user memory region (MR)
Post Send	posts a work request on a send queue (send, write or read operation)
Post Receive	posts a work request on a receive queue

Verbs call Description

RDMA is available on Infiniband (IB) and Ethernet (RDMA over Converged Ethernet aka RoCE).

Introduction (OFA / OFED stack)

The OpenFabrics Alliance (OFA) provides an open source implementation for the abstract Verbs interface called the OpenFabrics Enterprise Distribution (OFED). The OFED stack follows the IBTA standard verbs definition, and is widely deployed on many Linux and Windows distributions, including Red Hat Enterprise Linux (RHEL), Novell SUSE Linux Enterprise Distribution (SLES), Oracle Enterprise Linux (OEL) and Microsoft Windows Server operating systems. OFED is in production today in more than 60% of the TOP500 high performance computing sites.

The OpenFabrics Enterprise Distribution (OFED) implements channel I/O and enables RDMA on fabrics and networks in Linux and Windows environments. The software stack includes drivers for 10 Gigabit Ethernet and 10/20/40 Gigabit InfiniBand® interconnects as well as a rich set of Upper Layer Protocols (ULPs) and associated libraries including :

- SDP – Sockets Direct Protocol. This ULP allows a sockets application to take advantage of an InfiniBand network with no change to the application.
- SRP – SCSI RDMA Protocol. This allows a SCSI file system to directly connect to a remote block storage chassis using RDMA semantics. Again, there is no impact to the file system itself.
- iSER – iSCSI Extensions for RDMA. iSCSI is a protocol allowing a block storage file system to access a block storage device over a generic network. iSER allows the user to operate the iSCSI protocol over an RDMA capable network.
- IPoIB – IP over InfiniBand.
- NFS-RDMA – Network File System over RDMA. NFS is a well-known and widely-deployed file system providing file level I/O (as opposed to block level I/O) over a conventional TCP/IP

network. This enables easy file sharing. NFS-RDMA extends the protocol and enables it to take full advantage of the high bandwidth and parallelism provided naturally by InfiniBand.

- Lustre support – Lustre is a parallel file system enabling, for example, a set of clients hosted on a number of servers to access the data store in parallel
- RDS – Reliable Datagram Sockets offers a Berkeley sockets API allowing messages to be sent to multiple destinations from a single socket. This ULP, originally developed by Oracle, is ideally designed to allow database systems to take full advantage of the parallelism and low latency characteristics of InfiniBand.
- MPI – The MPI ULP for HPC clusters provides full support for MPI function calls.

In addition to kernel-level drivers and RDMA send/receive operation libraries, OFED provides services for message passing (MPI), sockets data exchange (e.g., RDS, SDP), NAS and SAN storage (e.g. iSER, NFS-RDMA, SRP) and file system/database systems.

OFED offers a layered network architecture (see Figure 37), which allows any application using the IBTA verbs definition to run independently of the underlying network infrastructure (e.g. InfiniBand, 10GbE).

Taking this advantage into account, rather than proposing a single middleware solution, CEA decided to put the effort into porting the OFED lower layers to the EUROSERVER "UNIMEM" RDMA architecture.

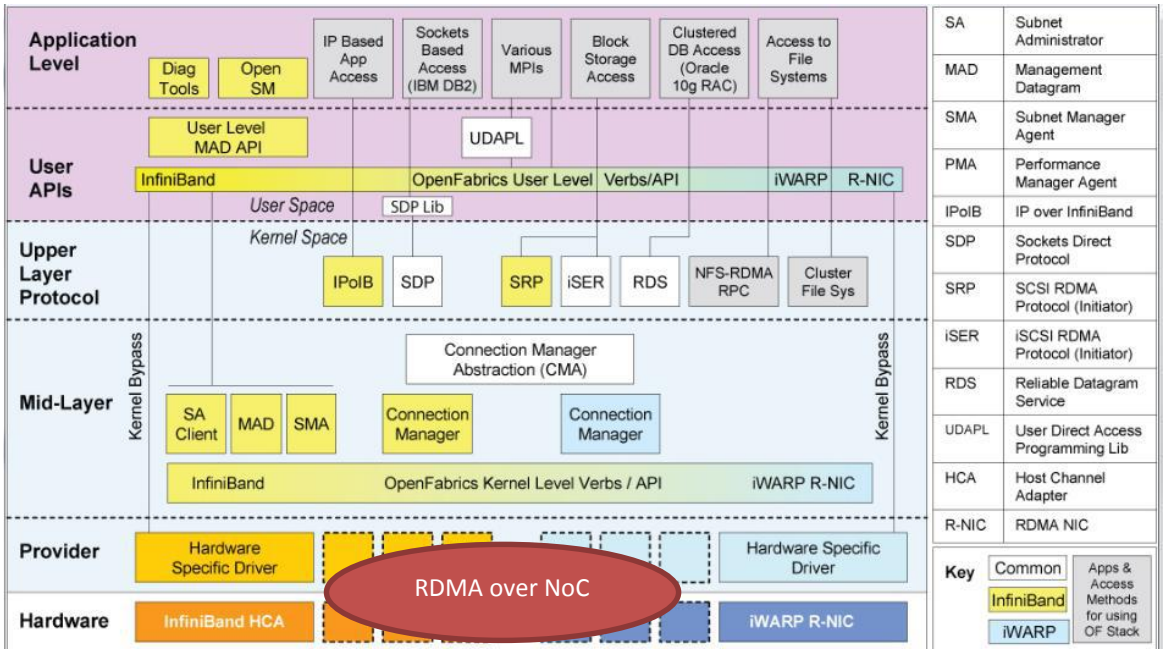


Figure 36: Architecture of OFED and the areas where RDMA over NoC cover – source (modified from) OFED[7]

[7] OFED interoperability Presentation. www.openfabrics.org

CEA approach: Porting to a new HCA: the Chip-to-Chip NoC

By adding a new Channel Adapter (Network on Chip), as well as new Data/Network layers, in the OFED framework, we will leverage all the existing upper layers and protocols (IP over IB, NFS-RDMA, MPI, etc.) without additional porting effort.

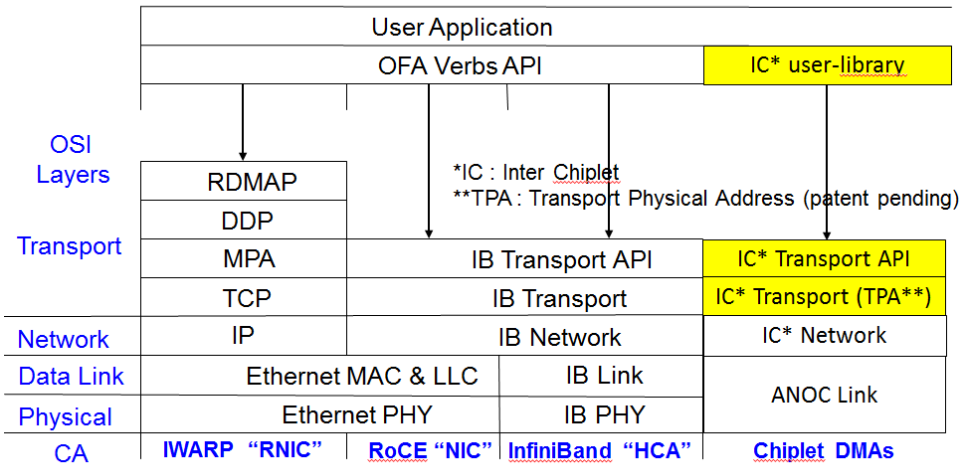


Figure 37: OFED over NoC implementation

OFED provides two APIs: a kernel-level API (kverbs) and a user-level API (uverbs) so the RDMA over NoC (RNoC) implementation is composed of:

- A user-space library supporting all configuration path verbs: e.g. query_device, query_port, reg_mr, etc.
- A kernel level driver supporting data path verbs implementation: e.g. post-send, post-receive, etc.

Unlike the complex InfiniBand HCA, the EUROSERVER RDMA hardware block does not provide hardware support for RDMA queue semantics. This means that the OFED queue management (send/receive queues and completion queue) has to be performed in software.

For this reason, the kernel driver will be responsible for queue management, and will interact with DMA through Linux kernel "dma engine" support. This DMA engine module provides a hardware agnostic API to various DMA controllers that will greatly facilitate the porting from the ARM Juno prototype to the EUROSERVER architecture.

It is important to note that even if the Linux RNoC kernel driver is still involved in RDMA transaction (there is currently no kernel bypass) DMA channels are still programmed with the application's buffer virtual addresses.

Status of OFED porting

The porting of OFED to EUROSERVER architecture is being performed in parallel with the design of a new patented hardware block closely connected to SMMU and called TPA (for Transport Physical Address). The idea is to take benefit of both SMMU and NoC routing capabilities to emulate a new class of Host Channel Adapter called RNoC (RDMA over NoC).

Both DMA and remote link's SMMU will have to be configured accordingly to registered RDMA memory regions (MR), Queue Pair (QP) and destination ID (routing destination on the NoC). I/O Virtual Address (IOVA) will be amended with a QP/MR key identifier to identify the address translation context to be used by the "from remote" SMMU as well as authenticate the requester.

Due to this non-standard usage of the SMMU, rather than the previously described VFIO framework, the Linux kernel IOMMU API is used inside our OFED kernel module.

However, in case of virtualisation, the second stage IOMMU will still be controlled with VFIO as described earlier.

The work has been started to get OFED working on the ARM Juno board.

Running Ubuntu 14.10 with an arm64 distribution, we were able to relatively easily compile basic RDMA libraries (libibverbs) and the new RDMA over NoC (librnoc) library with elementary verbs.

Two approaches are currently being studied for the development of lower layers driving the new TPA hardware block:

- Continue to use the ARM Juno platform with a (purchased) FPGA board implementing the TPA logic and connected to the ARM SoC interconnect as well as the DMA PL330 and its SMMU.
- Emulate the real EUROSERVER's chiplet on Mentor Veloce hardware emulator.

The first approach is much faster than hardware emulation, but it will require additional resources for FPGA development. For this reason, we will likely work on the hardware emulator despite the limited execution speed.

Linux is now booting on the Mentor hardware emulator, and we are currently bringing up the DMA engine and its SMMU. The OFED low level code (TPA kernel driver) is co-designed with the hardware team, and will be first tested in a "loop back" mode, preventing to emulate more than one chiplet. The current objective is to have this running by Q4 2015.

6. User space support for virtualisation (ONAPP + TUD)

6.1. Introduction

In the preceding sections we have described how to improve the virtualisation efficiency and functionality for the EUROSERVER platform. In this section we describe how improvements can be

made from the user-space side to further improve these metrics. In Section 6.2 we will describe how by moving driver level controls from the hypervisor control domain to specific, specialised guest domains we can avoid context switching and improve performance.

In Section 6.3 we describe how a particular application Cloud Radio Access Networks (CRAN) can utilise cloud-like virtualised infrastructure for hosting workloads and benefit from the type of architecture that EUROSERVER is proposing. There are also some changes that are required from the application side that look to benefit from particular hardware optimisations given the hardware configuration on offer.

6.2. Porting Unikernels, i.e. MiniOS to MicroVisor (ONAPP)

The original goal of this task as outlined in the DoW was to leverage a unikernel architecture such as the Mirage platform to provide lightweight application instantiation coupled with very efficient resource usage. The MicroVisor platform being developed in EUROSERVER extensively leverages the MiniOS unikernel capability from the Xen project to create very lightweight but efficient function domains to provide specific processing capability in an isolated VM.

The original MicroVisor design always envisaged providing hardware driver support through a super-thin driver domain interface that removed as much of the standard packet processing system overhead as possible. An early proof of concept implementation has produced a MiniOS domain for certain drivers to act as a high performance helper function with dedicated access to a particular piece of hardware. By avoiding the context switch to the control domain (Dom0) for each packet between local domains as well as for each hardware access will help improve the performance for network I/O operations that require a high frequency of context switches with the current implementation in Xen. This is particularly relevant for packet streams involving many small and frequent packets.

Every hardware NIC is assigned to a dedicated, high-performance unikernel domain (MiniOS) by the MicroVisor at boot time by passing through the Ethernet PCI device. Each MiniOS domain efficiently receives packets from either the internal virtual interface (MicroVisor switch) or an external interface (hardware NIC). These domains act as the interface between the hardware and the MicroVisor. Each MiniOS domain receives interrupts from either the MicroVisor for a virtual network queue packet or from the hardware device for an Ethernet frame from the network. This creates a dedicated single-purpose unikernel domain purely responsible for fast packet handling between Ethernet queues without any of the Linux OS scheduling, bridge forwarding, MAC learning logic required. Currently we bootstrap every available hardware NIC with MiniOS and forward packets between the hardware and the MicroVisor that then forwards the packets to the relevant interfaces based on the addressing path.

By creating several, specialised MiniOS domains we can implement specific network functions as they can efficiently forward packets between virtual interfaces and therefore allow for network service chaining.

Currently we have ported the e1000e, ixgbe and IGB Intel network cards to work on MiniOS. When MiniOS is started it detects the hardware, Ethernet device when it scans the PCI entries in xenstore. If a NIC hardware device that corresponds to a supported driver then MiniOS will call the appropriate driver's probe function. To determine which interfaces are connected to this particular instance, xenstore is scanned for VIF (Virtual Interface) entries. At this point a call is made to the netfront probe that create a thread for listening to incoming traffic on the physical and virtual interface queues.

MiniOS has three threads running;

1. A thread listening to xenstore events
2. A thread listening for incoming packets on the physical network interface
   Interrupt driven
3. A thread listening for incoming packets on the virtual network interface
   Polls the netfront ring for incoming packets and yields when there are no further requests

When a packet is received on either the virtual or physical network interface it is directly transmitted to the other interface. The threads run asynchronously to listen for packets on the physical and virtual interfaces. When either the physical interface or the netfront ring destination queues are full the thread will yield and try to retry packet transmission in the next iteration. The forwarding logic is implemented by copying the packet data to the destination queue. Only a single device pass-through is currently supported.

Each of the MiniOS domains are non-preemptive and they do not support multi-cores, so each one supports only one virtual core. They do however run threads which helps latency as a MiniOS domain is always able to receive and potentially handle requests.

Potentially MiniOS driver domain instances can be used to forward traffic to particular accelerators, dependent on the type of processing that is required. In the context of EUROSERVER, each MiniOS driver domain can be loaded with a specific, specialised hardware driver that helps increase the performance and reduce the overhead for handling different resource types.

6.3. CloudRAN Application Abstraction (TUD)

This section describes the work that has been done by TUD in the context of Task 4.2 (Subtask 4.2.3) The primary goal of this activity was to specify and develop an application programming interface (API) that i) enables user space abstraction of telecom data plane and signal processing applications from the underlying hardware and ii) allows sharing of the available resources by multiple applications. In contrast to existing frameworks like e.g. Data Plane Development Kit (DPDK) and OpenDataPlane (ODP), which are primarily dedicated to packet processing, the developed API shall be more general in order to support a broader class of dataflow applications. Particularly, API shall reflect and support:

- dataflow computation model of data plane and signal processing telecom applications,

- high performance and strict timing requirements of telecom software stack (L1 frame processing requirements: GSM 40ms, TD-SCDMA 5ms, TD-LTE 1ms)
- platform extension using heterogeneous accelerators,
- task and data migration across multiple memory domains (e.g. CPU to accelerator),
- application partitioning and distribution across multiple compute nodes enabling e.g. pipelined processing.
- POSIX-based real-time and non real-time operating systems as well as bare metal solutions.

In the reported time period, TUD developed an initial version of data-flow programming interface and its associated runtime engine enabling parallel execution of CRAN benchmarks on multi-core platforms. In addition to this, an appropriate simulation platform has been developed for the purpose of API implementation, evaluation and validation. The development activities were coordinated in conjunction with requirements analysis in task T2.2 as well as the activities related to the design and implementation of dataflow runtime engine (DFE) in T4.3.3. Results of this part of the work have been used within T3.1 for EUROSERVER CRAN benchmark implementation, system exploration and demonstration as well as in T4.3.3 for DFE design. The work and the contributions of TUD within the reported project period are summarised as follows:

- Definition of the hierarchical dataflow model of computation and the requirements analysis for CRAN data plane applications (in conjunction with T2.2, T3.1 and T4.3.3).
- Definition of computation API incl. dataflow modeling elements and parameters for CRAN data plane application specification.
- Concept of user-space resource virtualisation including the definition of logical computing resources and the mapping methodology to physical resources allowing seamless application portability and CRAN system adaptation to various hardware arrangements.
- Implementation of computation API and a set of associated benchmark applications.
- Development of API capable system simulator for API evaluation and validation as well as tools for automated model-driven application development.

The Computation API has been developed and implemented jointly with the task related to data flow runtime engine as reported in D4.4 section 2 using common hardware platforms and software stack.

System Concept

This part of the work focuses primarily on the implementation aspects of radio access protocols for CRAN applications using a common computation API. More specifically, the goal is to provide a unified framework enabling modeling, representation and implementation of data plane applications of radio access protocol (e.g. baseband processing). In this regards, the data plane and signal processing applications can be well represented by dataflow computation model. The dataflow model defines actors, i.e. nodes of computation and channels between actors associated with data transmission. An actor consumes data (tokens) produced by its predecessors, executes a function on this data set and produces the data to its successors. In contrast to sequential program execution typical for imperative models, the dataflow paradigm enables actor execution once all the necessary input data produced by its predecessors are available and the placeholders for produced data are

available. The dataflow model is inherently parallel, and hence, well suited for simultaneous actor processing on multi-core hardware. In order to reflect the design and representation of more complex applications associated with the CRAN scenario, a hierarchical dataflow model (Figure 38) has been proposed within this project. In this case, the actor is associated with i) "task" representing a kernel function or ii) "system", which is a composition of tasks and systems.

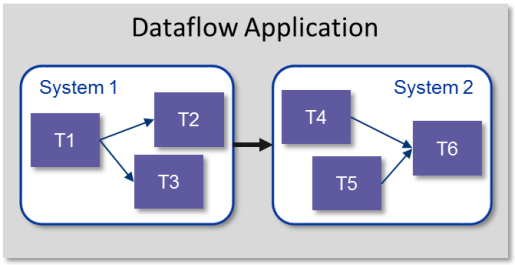


Figure 38: Concept of hierarchical dataflow computation model

In the CRAN concept, multiple dataflow applications can be defined and executed simultaneously on the common hardware resources as depicted in Figure 39. In this simplified model, CRAN control API enables to configure and manage the multiple dataflow applications while computation API translates the dataflow specific requests to dataflow engine runtime calls. In order to allow the development and implementation of dataflow applications, extensible dataflow programming model has been proposed within this task reflecting all requirements associated with the data plane processing of CRAN applications. Note that the research related to data flow runtime engine (DFE) is addressed in T4.3.3.

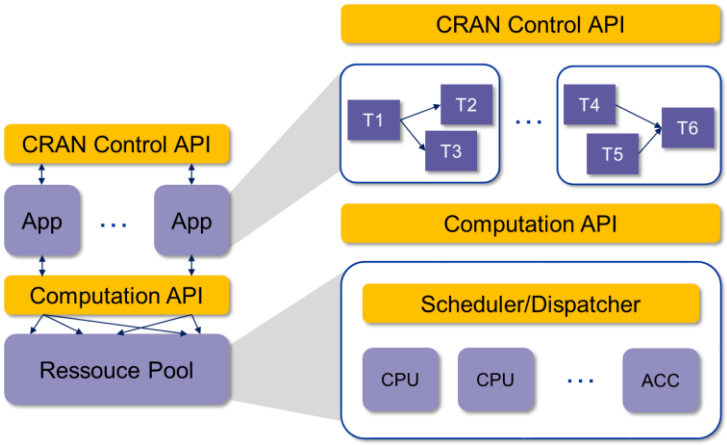


Figure 39: Principle of CRAN dataflow computing system

Dataflow Programming Model

In the initial phase of the project the existing open-source frameworks dedicated to dataflow and/or task-centric processing have been investigated (e.g. Ptolemy, Simulink, GNU radio, DPDK, ODP, OpenMP, StarSs, …). Motivation was to find relatively simple, well-documented and analysed tool that could be extended and adapted to specific needs of CRAN processing. Due to the lack of simplicity, platform independence, support, clear documentation, seamless integration with TUD's Tomahawk MPSoC framework and comprehensive performance analysis for fine-granular data-

intensive workloads, TUD proposed to modify and extend its Tomhawk-MPSoC related TaskC programming interface for purpose of the EUROSERVER project. Based on this, an extensible dataflow programming model and programming interface (computation API) have been introduced. Moreover, the close relationship and compatibility to TaskC API will in the future allow the integration of CRAN specific accelerators (e.g. channel decoder, detectors, DFT etc.) into GPP based multi-core platform.

According to the definition of hierarchical model of dataflow application, a "task" represents an atomic unit of computation that can be executed on a specific type of processing element (CPU or accelerator). In addition to tasks, "system" elements are defined for the purpose of building a hierarchy of elements, i.e. a system may comprise tasks and systems. Task and system migration has to be supported, e.g. for EUROSEVER multi-node operation and/or offloading task to a specific accelerator. Therefore, the task/system specification has to encapsulate all relevant parameters allowing their execution on the remote node e.g. reference to function, description of containers associated with input and output function arguments i.e. reference, size, type etc. (Figure 40). In addition to this, a task/system can be associated with various processing resources e.g. software programmable CPUs/DSPs/ASIPs or specific hardware accelerators (ACC).

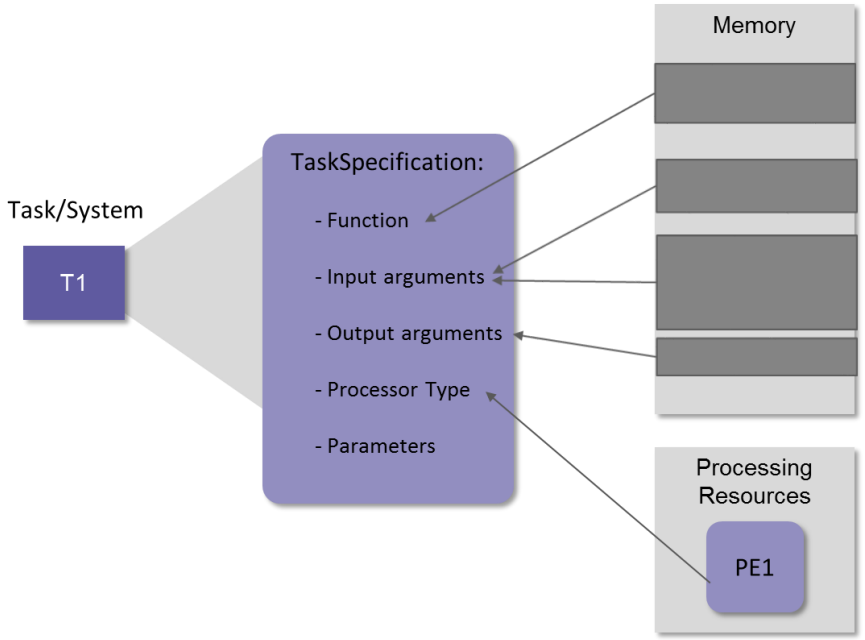


Figure 40: Principle of the task specification and the encapsulation of task arguments

In order to cope with the variability of system arrangements with regards to CRAN software portability, computation API shall decouple the application from the underlying hardware by the definition of logical computation resource types e.g.: "SP" (system processor) and "PE" (processing element). The runtime environment maps SPs and PEs to the execution threads of operating system that are further managed by operating system according to user configuration and OS scheduling policy. The principle of associating tasks/systems to logical PEs/SPs and their mapping to physical cores is illustrated in Figure 41. The framework assumes three levels of mapping:

- *Computation API level*: application's system/task elements are associated with virtual processing types SPs/PEs in order to preserve application granularity. In addition to this, processor type value is introduced in order to differentiate multiple SP/PE types i.e. PE1, PE2, PE3 in this example. Task/System affinity to virtual cores is controlled by user using specific computation API parameters. Note that multiple-to-one and one-to-multiple mapping is allowed, i.e. multiple types of virtual cores can be associated with specific task or system, e.g. task T6 is assigned to core types PE2 and PE3, respectively. This poses a challenge to the DFE scheduler to optimise core type selection according to specific criteria.
- *DFE level*: virtual core types (SP/PE) are mapped to thread pool clusters of DFE runtime. Thread affinity is controlled by System and Task managers within DFE (this part of work is subject of T4.3.3). Number, size, priority and type of thread pool clusters is optimised at design time according to application requirements and hardware capabilities.
- *OS level*: Mapping of threads to physical cores is managed by operating system. In addition to this, tuning of the operating system in conjunction with optimising thread priority and physical core affinity allows user to optimise responsiveness and throughput of CRAN system. Note that mapping of threads to specific accelerators is also supported and is inevitable in order to meet the real time constraints.

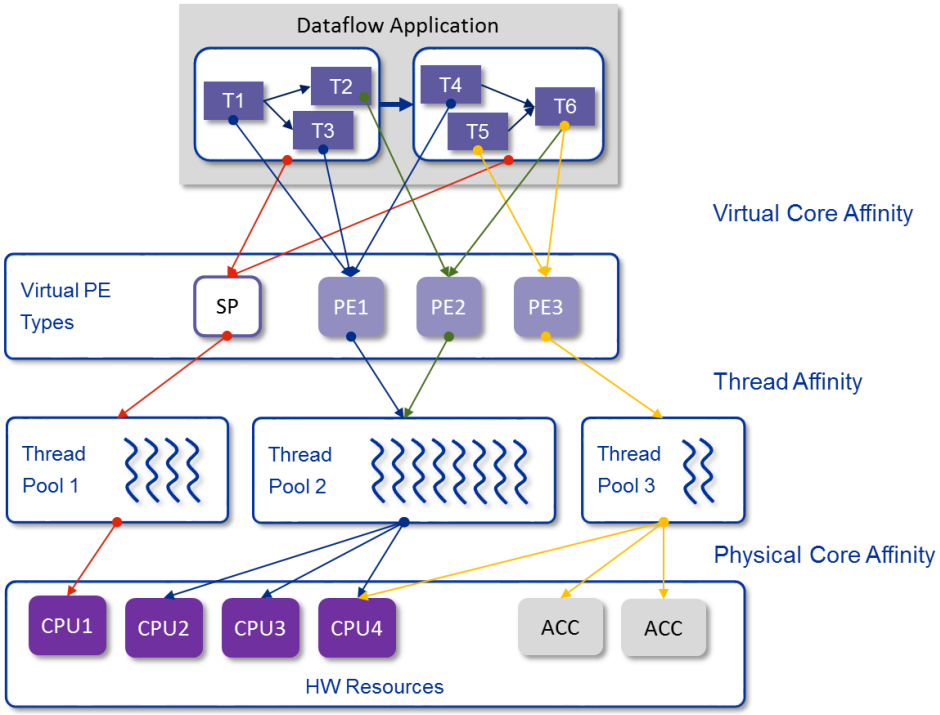


Figure 41: Concept of resource virtualisation and mapping strategies

Computation API

Based on the concept of programming model proposed in the previous part, an associated computation API have been defined and implemented. The API allows a set of tasks/systems to be specified using a dedicated data structure "TaskSpecification". TaskSpecification is extensible and, depending on the system configuration, it allows additional parameters to be defined; e.g.

input/output arguments, task/system dependencies, priority, worst-case execution time, and deadline. In the following part, the basic elements of computation API are described. Note that the full API specification will be described in the DFE programming documentation.

- *Creating and configuring task*: New task is created using the "task" function by characterizing default parameters e.g. name, type, function, input/output arguments.
  - Create task:

```
TaskSpec* t = task(fnc_t* fnc, arg_t* arg1, arg_t* arg2,…);
```

Function returns pointer to task specification structure. In order to minimise the overhead associated with allocation/deallocation of data structures, pre-allocated pool of task specification with thread save pop-push mechanism have been exploited (Figure 41).

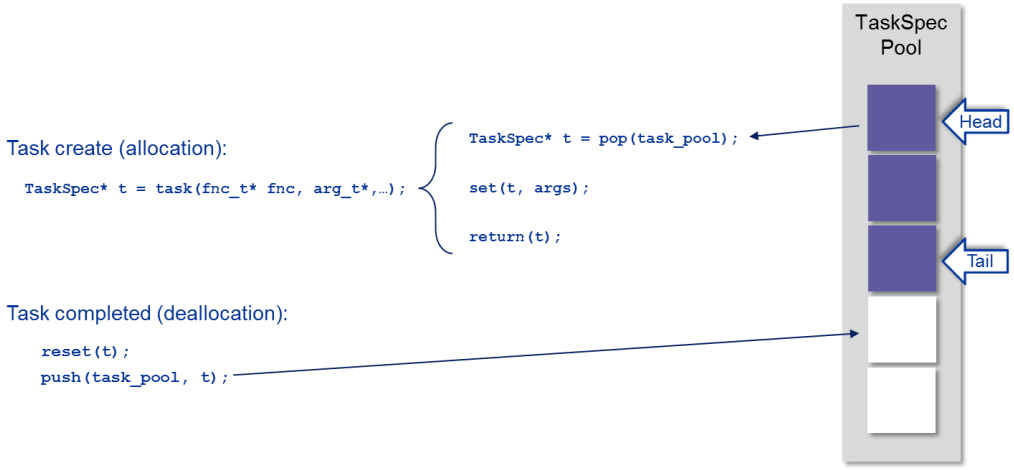


Figure 42: Principle of using preallocated data structures for task specification

  - Setting additional task parameters:

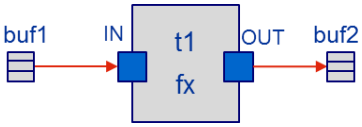
```
set_task(t, Property, Value);
```

  - Send request for task execution to DFE:

```
push_task(t);
```

Note that task request in unblocked and asynchronous. Moreover, run-to-completion model is employed for task execution in order to better optimise the utilisation of resources by centralised management units (i.e. SystemManager and TaskManager in DFE).

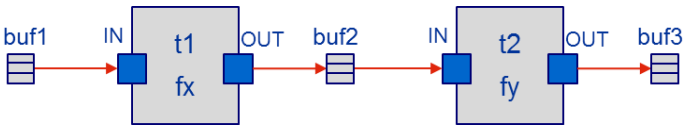
- Example: task “t1” executing function “fx()” with input/output arguments “buf1”/”buf2”. Note, that macros IN/OUT translates argument specific parameters (e.g. pointer, size, type, …) to argument data structure.




```
TaskSpec* t1 = task(fx, IN(buf1, BUF1_SIZE), OUT(buf2, BUF2_SIZE));
set_task(t1, task_name, "t1");
push_task(t1);
```

- *Define task dependency*: In order to exploit precedence constraints in the DFE scheduler, task dependency can be specified by programmer. Computation API allows to define dataflow dependency between source and destination tasks as follows:

```
predecessors(TaskSpec* dest_task, TaskSpec *source_task,…);
```



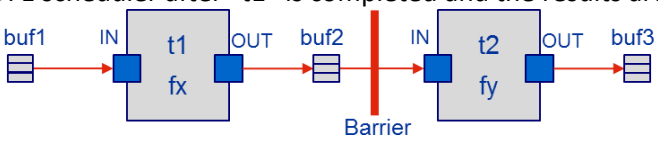
- Example: destination task (consumer) “t2” depends on data produced by source task “t1”:

```
TaskSpec* t1 = task(fx, IN(buf1, BUF1_SIZE), OUT(buf2, BUF2_SIZE));
TaskSpec* t2 = task(fy, IN(buf2, BUF2_SIZE), OUT(buf3, BUF3_SIZE));
predecessors(t2, t1);
push_task(t1); push_task(t2);
```

- *Task synchronisation:* *DFE* employs an asynchronous task execution approach, i.e. the task creation is completely decoupled from the task execution. In order to guarantee data consistency at task inputs in specific situations (e.g. memory boundary crossing in distributed memory architecture, sync with accelerator etc.), barrier synchronisation mechanism has been introduced:

```
synchronize(TaskSpec* S);
```

- Example: In system S, DFE scheduler suspends the execution of tasks behind barrier until all task of system S issued before barrier are completed. In this example, the “t2” is passed to DFE scheduler after “t1” is completed and the results are stored in buf2.



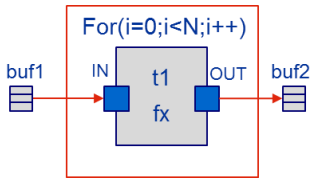

```
TaskSpec* t1 = task(fx, IN(buf1, BUF1_SIZE), OUT(buf2, BUF2_SIZE));
```

```
push_task(t1);

synchronize(s);

TaskSpec* t2 = task(fy, IN(buf2, BUF2_SIZE), OUT(buf3, BUF3_SIZE));
push_task(t2);
```

- *For/While loop:* Loops allow to realise program iterations. Both for and while loops with static and dynamic iteration bounds are supported.

```
for(init; condition; update){  …; task(…); …   }
```

  - Example:

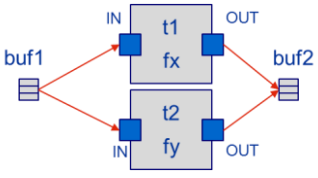



```
for(int i=0; i<N; i++){
    TaskSpec* t1 = task(fx, IN(&buf1[i], 1), OUT(&buf2[i],1));
    push_task(t1);
}
```

- *Conditional execution:* Conditional execution using if/else or switch statements is allowed:

```
if (condition){
        …
}
else{
        …
}
```

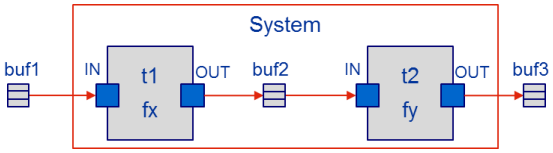
  - Example:




```
if(condition){
        TaskSpec* t1 = task(fx, IN((buf1, BUF1_SIZE), OUT((buf2,BUF2_SIZE));
        …
        push_task(t1);
}
else{
        TaskSpec* t2 = task(fy, IN(buf1, BUF1_SIZE), OUT(buf2, BUF2_SIZE));
```

```
            ...
            push_task(t2);
    }
```

- *System-Task hierarchy:* Hierarchy of systems and tasks is supported. In the example, tasks t1 and t2 belong to system S. System structure is defined dynamically by system generator function assigned to system during system creation phase.
    - Example:




```
...
TaskSpec* s = task(system_fnc);
push_task(s);
...

void system_fnc(){
    TaskSpec* t1 = task(fx, IN(buf1, BUF1_SIZE), OUT(buf2, BUF2_SIZE));
    push_task(t1);
    TaskSpec* t2 = task(fy, IN(buf2, BUF2_SIZE), OUT(buf3, BUF3_SIZE));
    push_task(t2);
    system_end();
}
```

Computation API Implementation

For the purpose of portability (note that a EUROSERVER platform was not available at this time), the computation API has been developed and implemented in conjunction with T4.3.3 for standard C/C++ Linux environment. The GNU C/C++ based tool-chain and associated libraries libc, libpthread and glibc have been used for the implementation (reported in D4.4).

In the initial phase of project, the computation API has been developed and validated on the host XEON x86 (two Intel Xeon E5-2650v3, 10-core, 2.3GHz) platform. Later, the API has been ported to ARM Cortex-A15/A7 (Odroid XU3) and integrated with the DFE runtime from T4.3.3 (reported in D4.4) and CRAN benchmark from T3.1. Moreover, compatibility with the POSIX specification enables seamless portability to the EUROSERVER discrete prototype and full prototype in a later phase of project as well as to real-time OS powered systems.

A methodology for the evaluation of the implemented API as well as associated utilities has been developed in order to analyse performance and identify critical sections. The proposed approach has been evaluated on x86 development platform using untuned real-time Linux. The most important factor affecting the throughput and responsiveness of the system is the critical path of API-runtime control path. For this purpose, TUD analysed most important sections of the control pipeline. Particularly, the average latency per task of four following segment have been analysed:

1. Latency associated with the generation of task specification data structure (TaskSpec) of new task or system within application controller based on the pre-allocated TaskSpec pool.
2. Segment (1) followed by setting the task related parameters of TaskSpec.
3. Segment (1)-(2) followed by push of TaskSpec into the input queue of TaskManager.
4. Segment (1)-(3) followed by inserting of TaskSpec into input queue of the thread pool. Note that the effect of scheduler have been suppressed by bypassing the scheduler.

The mentioned segments of control path and associated measured average latencies pre task are illustrated in Figure 43. The measured results are summarised in Figure 44 to Figure 47. For each measurement, the average latency of 100 successive trials (upper figure) and the distribution in the form of histogram (lower figure) are provided. The main observations about latency on control path can be summarised as follows:

- Experiments on untuned real time Linux system cause reasonable variance. Hence, the improving of the determinism of Linux configuration as well as runtime support is necessary for telecom applications.
- New task allocation using pre-allocated buffers with 210ns mean and 300ns maximum latency is acceptable.
- Setting TaskSpec data structures with task/system specific parameters takes too much time. Note that lot of debugging parameters are included into TaskSpec in this phase of the project. This, however, can be easily reducible in the later phase of the project.
- The latency in TaskManager pop – bypass – push operation of about 200ns results in 5MTask/s throughput which could be acceptable for CRAN use case. However, the queue access shall be improved in the future by making use of lockless techniques (note that currently the mutexes and spinlocks are supported)
- Overall max latency of control path bellow 900ns results in Task rate above 1MTasks/s. This can be acceptable for small-scale telecom scenarios. However, in order to improve the scalability of fine granular application scenarios (i.e. lot of short-time tasks) the latency of control path should be reduced. The main impact on the latency reduction largely depends on the optimisation of TaskSpec data structure and synchronisation mechanism of data queues.

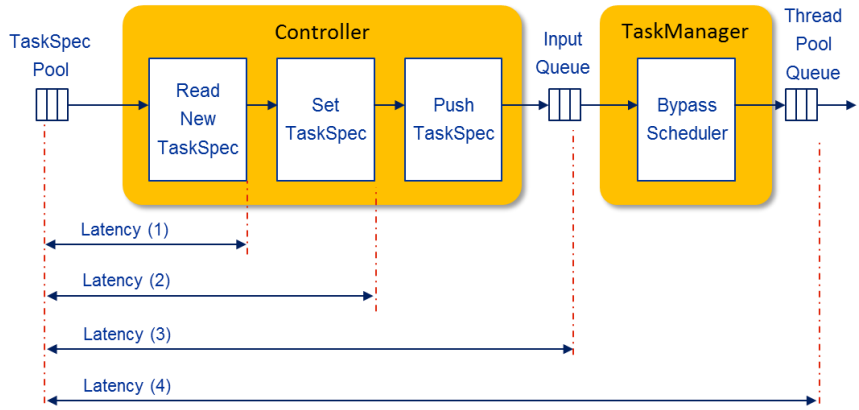


Figure 43: The arrangement of the system for the analysis of average latencies per task of four segments in the control path.

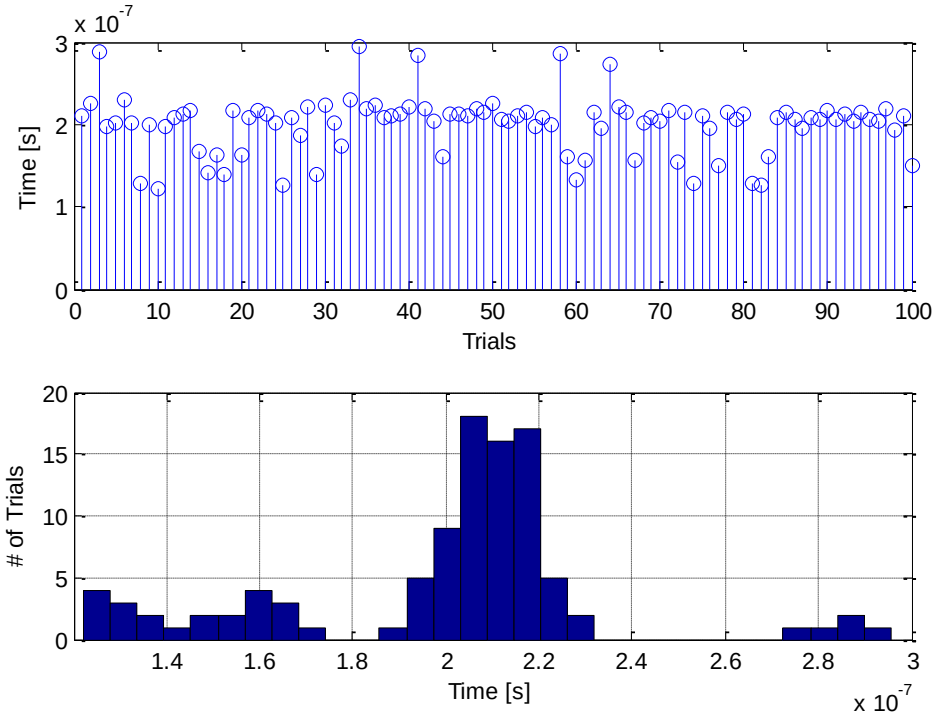


Figure 44: Average latency of segment (1) in Figure 43 of 100 successive trials and associated histogram.

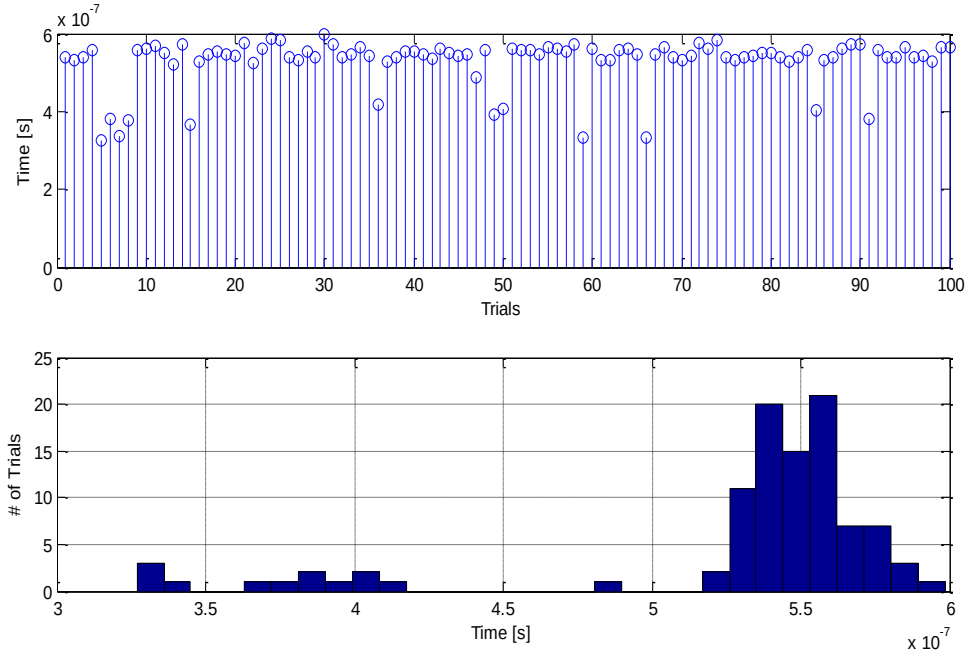


Figure 45: Average latency of segment (2) in Figure 43 of 100 successive trials and associated histogram.

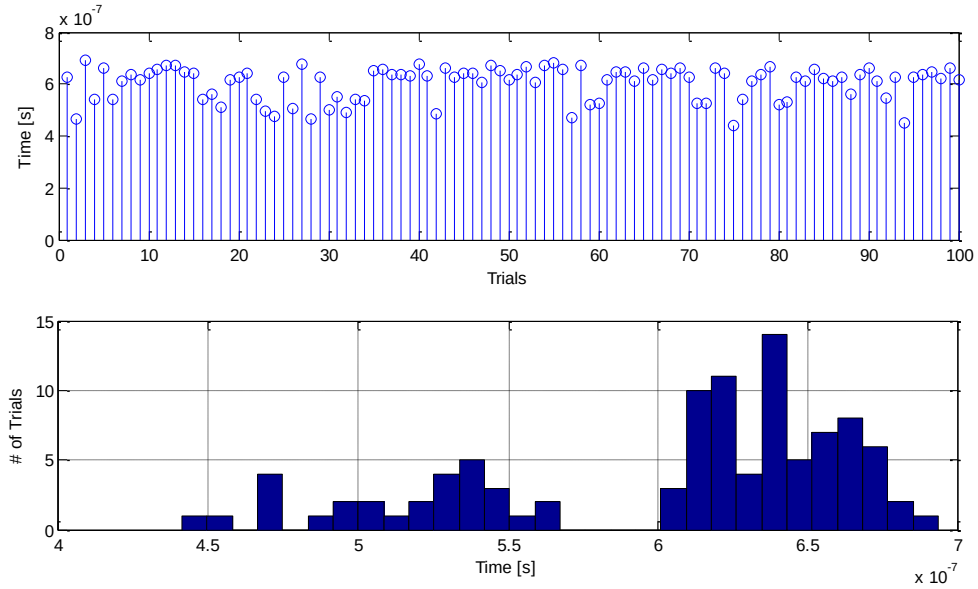


Figure 46: Average latency of segment (3) in Figure 43 of 100 successive trials and associated histogram.

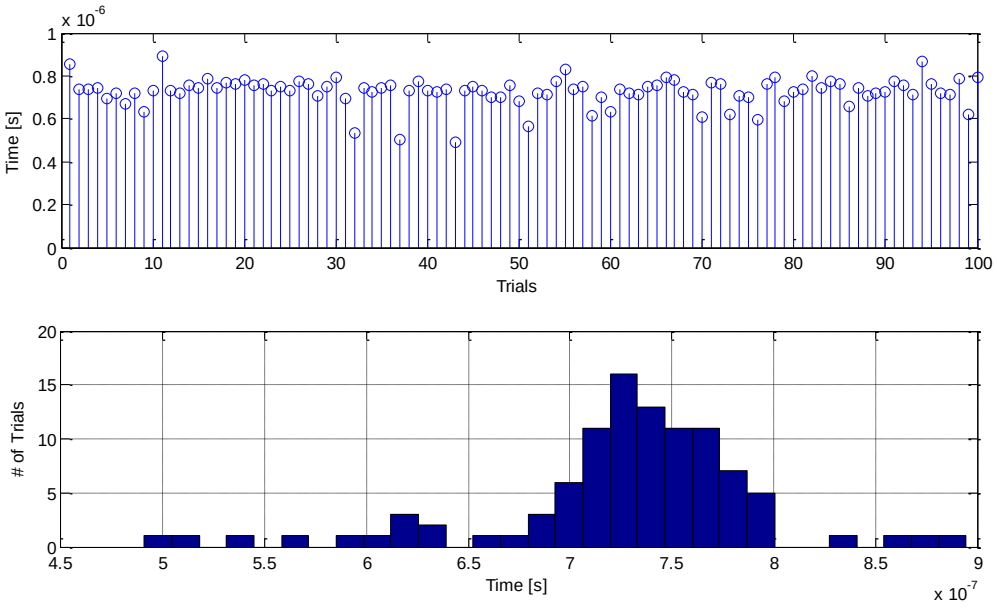


Figure 47: Average latency of segment (4) in Figure 43 of 100 successive trials and associated histogram.

Ongoing and future work

In the current phase of the project, the computation API and associated DFE have been extensively analysed. The bottlenecks have been identified within DFE as well as CRAN benchmark application (limited queue throughputs and fine granular application partitioning). Due to this fact, API extensions are proposed e.g. definition of task set and task fusion capability in order to reduce request rate and scheduler overhead. Moreover, additional task parameters are introduced enabling implementation of low overhead fine granular events monitoring for the purpose of system analysis and debugging.

In the next phase of project, the optimisation and extension of computation API in conjunction with DFE and CRAN benchmark and system capabilities will be undertaken. Particularly, the goal is to:

- Improve API in order to reduce granularity and DFE request rate in critical CRAN sections,
- Exploit EUROSERVER specific instruction set e.g. Cortex-A53 SIMD NEON instructions for implementation of critical API functions
- Extend API allowing transparent access to the accelerators subsystem using the same API directives
- Optimise critical API data structures in terms of their size and memory alignment
- Reduce dynamic allocation of critical data structures and utilising pre-allocated structures
- Validation of API on EUROSERVER discrete and full prototype (in cooperation with FORTH, CEA, ST).
- API extensions enabling scale-out solutions of CRAN architecture for multi-node (chiplet) operation in EUROSERVER platform (in cooperation with FORTH).

In addition to planned work items, the TUD's dataflow modeling and code generation tool (Figure 48) dedicated to the development of dataflow applications will be extended in order to support additional API functionality. This will enable the automation of CRAN application development and will simplify benchmark development and maintenance.

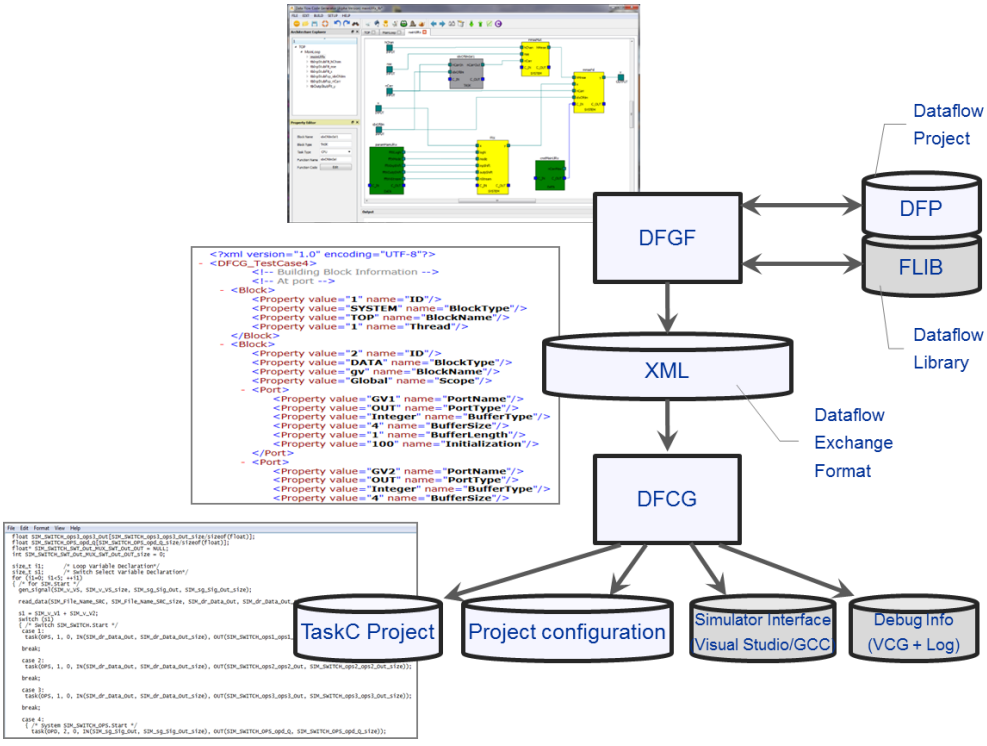


Figure 48: Dataflow modeling and code generation tool flow of TUD supporting CRAN application development and automated code generation according to CRAN computation API (DFGF - dataflow graphical frontend, DFCG - dataflow code generator)

7. Conclusion and future work

In this deliverable we have described how there can be improvements to I/O virtualisation at various layers of the software stack, from the hypervisor level through to the applications running on the hardware. These techniques offer performance improvements, generally at the sacrifice of hardware generality by using knowledge of the way that the system stack is used and improving the performance of frequently performed 'expensive' operations.

The work on the MicroVisor platform is described in more detail in D4.3 'Kernel-level memory and I/O resource sharing across chiplets'. The other techniques are being tied in together with the other software techniques, which are produced in the scope of Work Package 4, in the Integration and Evaluation Work Package (WP6). The techniques and technologies detailed in this Deliverable offer performance and functional improvements that will allow the EUROSERVER platform to reduce the overhead for I/O resources and move towards reaching the overall project objectives.

Improving the sharing of resources has been a core principle behind the efforts reported in this document. To exploit the benefits of the scale-out microserver architecture we need for resources to be addressable and shareable between EUROSERVER boards. RDMA has been used to access and share memory resources between boards with efforts also going into adapting the platform such that programs already using Linux Sockets APIs can benefit from RDMA without being modified. Improvements have also been made to the access of peripheral hardware via the PCIe and CCI buses for x86 and ARM systems respectively. Sharing of network resources in a fair and scalable manner is an important hurdle for sharing all the other resources and has been investigated with a driver created for the EUROSERVER 32-bit demonstration platform to allow efficient sharing of the NIC. This will have to be repeated for the final 64-bit EUROSERVER platform.

While working on the profiling of shared network resources it became apparent that one of the operations that limited the scalability was that of the TCP and IP header checksum calculations. These checksum operations will be carried out by the FPGA on the final EUROSERVER platform and by doing so offload the work from the core platform.

We looked at removing the TCP/IP overhead for memory and storage communication on remote nodes. This was carried out by improving RDMA access via UNIMEM, a mechanism that describes memory resources as a shared unit and utilising part of the OFED stack (through porting) for handling communication primitives. For storage sharing and faster access to storage primitives the overhead of TCP/IP was also re-evaluated and instead direct ATA over Ethernet commands were favoured to help improve the throughput for smaller requests. To also help improve the apparent speed of the system for 'bursty' requests we have also implemented caching mechanisms that will take benefit of high speed storage while waiting for slower-speed, rotational or remote network path based disk drives to synchronise.

The mechanisms presented in this document will be incorporated into the final EUROSERVER platform and as such the improvements made will also be included in the final platform.

Appendix

I. Implementation details on 'sharing a network interface'

a. Receiving a frame

Once the RX descriptors ring has been initialised, with appropriate buffer space for new sk_buff frames, and the DMA engine's RX channel has been configured, the driver is ready to accept incoming frames. It is important to note that the RX path is asynchronous, i.e. that the driver is expected to always be able to accept an incoming frame regardless of whether any user-space process is waiting to receive it. For a process to actually receive data from network, it must notify the kernel network stack that it is waiting for incoming data, by calling the recv() system call (or variants – e.g. recvfrom()). Reception of data is a synchronous operation from the user-space perspective, meaning that calls to recv() are (usually) blocking. The following flow-chart (see Figure 49) illustrates the events that occur at data reception, both from the process' and the driver's perspective. The figure also shows the hardware and software components involved in receiving data from the network.

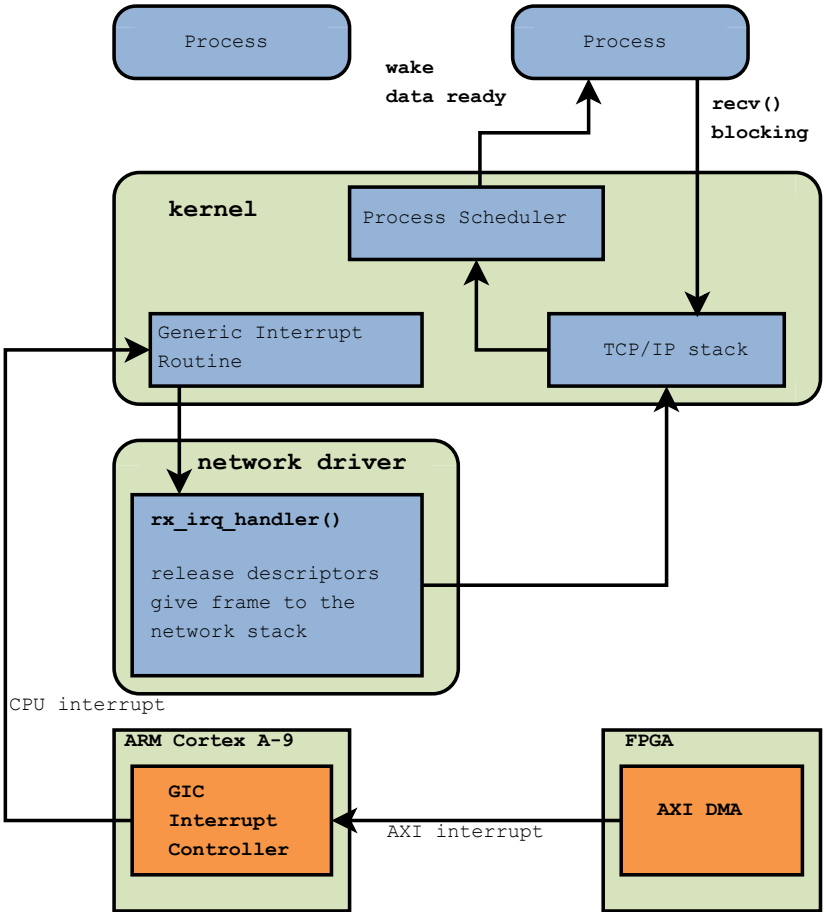


Figure 49: Flow chart for the RX path.

In the EUROSERVER 32-bit discrete prototype, the 10Gbps MAC hardware block uses the DMA engine to write an incoming frame to the predefined memory segments. When the DMA engine has finished

writing the frame into memory it raises an interrupt that causes execution of the driver's receive interrupt handler. The descriptors in the RX ring are updated by the DMA automatically when writing the frame into memory. The interrupt handler checks the appropriate status DMA register for errors and passes the received frame to the upper network stack in an sk_buff format, using the netif_rx() function. Then the driver releases the corresponding descriptors and allocates new sk_buff space for each descriptor, the same way as in the initialisation phase. Note that the DMA engine writes an incoming frame to a single buffer space, using only one descriptor for each frame. If errors have occurred during frame reception, the driver will schedule a task to reset the DMA engine and reinitialise the RX descriptors ring at a later time (while running in task context).

After the frame has been given to the kernel, the network stack does some processing of the frame and checks whether any processes are waiting for packets with that destination information. If there are such processes, it copies the received packet payload to the process's user space buffer and it wakes up the process, which is blocked in the recv() system call. There are no frame copies between the driver and the kernel.

The following figure shows the RX descriptor ring in operation. The driver maintains a head and a tail pointer in the RX ring. The head pointer points to the first descriptor written by the DMA engine that has not yet been processed by the receive path interrupt handler. The tail pointer points to the last free descriptor that can be used by the DMA engine. The DMA engine also maintains a pointer that keeps track of the current descriptor being written. Head and tail pointers are updated by the driver's receive interrupt handler, when it is invoked upon frame arrivals. The updated head and tail values are given to the DMA engine.

As for the TX ring, three descriptor subsets (as delineated by the values of the head, tail, and curr pointers) can exist in the RX ring during operation:

- head – curr: descriptors recently written by the DMA engine, but have not yet started processing in the driver's interrupt handler
- curr – tail: remaining free descriptors, available to the DMA for writing
- tail – head: descriptors written by the DMA that are being processed by the interrupt handler.

In our implementation, each descriptor represents a whole Ethernet frame, since the DMA writes the received Ethernet frame in its entirety (without fragmentation) in one memory buffer.

The receive-path interrupt handler performs the following operations when invoked:

4. Checks for errors reported by the DMA engine in the status register. If an error has occurred, it schedules a task that will reset the DMA engine and reinitialise the RX ring in the near future.
5. Reads the tail and head pointers, and then reads all descriptors in that range.

a. For each descriptor, it reads its status field for errors.

b. Marks the frame as CHECKSUM_UNNECESSARY, to tell the kernel not to re-evaluate the checksum.

c. Delivers the frame to the kernel network stack, by calling the netif_rx() function.

d. Allocates a new buffer and sets it up as an sk_buff space, using the netdev_alloc_skb_ip_align() function.

6. Advances the tail pointer.

7. Submits the physical address of the tail descriptor to the DMA engine.

b. Device tree description of the virtual network device

To share the network interfaces in the EUROSERVER 32-bit discrete prototype, two device tree entries must be added, so that the Linux kernel can load our driver and map the hardware register spaces. The relevant device tree segments are shown in the following listing.

```
axi-dma@82000000 {
    eusrv-connected = <0x6>;
    compatible = "xlnx,axi-dma-6.03.a";
    interrupt-parent = <0x2>;
    interrupts = <0x0 0x1e 0x4 0x0 0x1d 0x4>;
    reg = <0x82000000 0x10000>;
    xlnx,family = "zynq";
    xlnx,generic = <0x0>;
    linux,phandle = <0x7>;
    phandle = <0x7>;
};
eusrv-eth@41000000 {
    eusrv-connected = <0x7>;
    compatible = "eusrv,eusrv-ethernet-1.00.a";
    device_type = "network";
    interrupt-parent = <0x2>;
    local-mac-address = [00 00 1a 12 34 56];
    reg = <0x41000000 0x10000>;
    linux,phandle = <0x6>;
    phandle = <0x6>;
  };
```

The first entry named axi-dma@82000000 is the segment that describes the AXI DMA Engine hardware block that resides in the FPGA of each compute node. Its member field “reg” instructs our network driver to map the register space of the DMA into the physical memory region 0x82000000 - 0x82010000. The second entry, named eusrv-eth@41000000, describes the MAC block that resides in the main board's FPGA. Its field “compatible” instructs the Linux kernel to load our network driver (the driver has the same tag in its source code). The Field “local-mac-address” tells the network driver to create a standard Ethernet interface in the Linux kernel with MAC address

00:00:1a:12:34:56. The field “reg” instructs the driver to map the MAC register space into physical memory region 0x41000000 - 0x41010000. Finally, the field “interrupt-parent” defines that the interrupt controller responsible for interrupts generated from the device is the Generic Interrupt Controller (GIC) of the Xilinx Zynq-7000 SoC.

II. Implementation details on 'sockets-over-rdma'

a. Connection buffers

For all data transfers a pair of send and receive buffers exist in each end of a connection. We allocate separate buffers per connection to avoid synchronisation problems among processes. Processes can modify these buffers from user space, so enabling accesses from multiple sources would require complex and time-consuming buffer protection. The buffers come in two identical pieces, one for sending and one for receiving data, which are allocated during the connection establishment phase in contiguous memory space. Subsequently, their physical addresses are exchanged between the peers. Physical addresses are needed because the DMA engine operates with these. They are only stored in kernel space, within the connection struct. Kernel memory is used for the allocation of the buffers and then they are memory-mapped in user space. This way avoids the overhead of "pinning" the correspondent pages in memory. DMA operations use physical addresses and for this reason it must be assured that data source or destination pages do not get swapped out during a transfer.

Figure 50 illustrates the organisation of the buffer pair at one side of a connection.

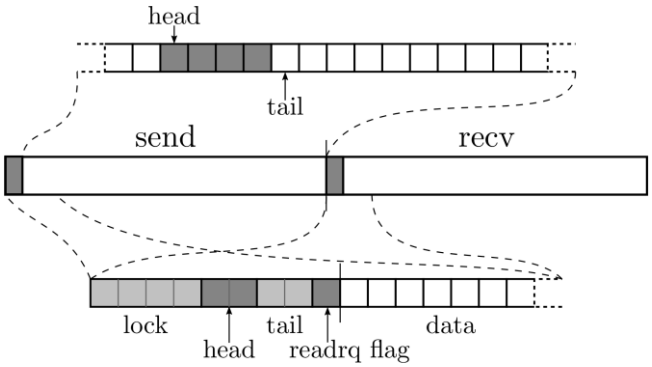


Figure 50: Organisation of send/recv buffers for a connection.

Both send and receive buffers of one connection are organised in a ring buffer (or circular buffer) scheme. This means that although they have physical ends, they do not have logical ends. Data reaching the end border can continue (wrap around) from the beginning, provided that there is adequate space. By using ring buffers, data can be consumed and produced simultaneously, with each action occurring at each end. In our implementation it is possible that the user process stores more data to be sent, while at the same time the driver performs a DMA operation sending older data. For this reason, ring buffers organisation was followed.

Data boundaries inside a ring buffer are pointed to by the head and tail pointers. The first indicates the position where data starts and the latter always points to the first free byte after the data. Any new insertion will happen there. When both pointers have the same value, the buffer is assumed

empty. According to the previous definition though, a full buffer would again have the pointers with the same value. To avoid confusion, one byte always stays free. Therefore, a full buffer (minus the one byte) has its head pointing at the next byte of its tail.

The location where the head and tail pointers are stored is on the buffer itself. Actually, a few bytes at the beginning of both send recv buffers are for holding special values and not data. Each pointer takes up two bytes, with the values stored there representing offsets within the buffer (from its start, not from the data's start). As a result, buffers of up to 64KB are supported with this configuration. The reason why these special "metadata" are stored together with the actual data is that all these (data + metadata) must be shared between the RDMA driver and the user space process. On the other hand, although each connected side knows the values of head and tail pointers of the remote recv buffer, these are only used by the driver and therefore are stored in the connection struct.

Except for the head and tail pointers, two other values are also kept in the beginning of each buffer. The read request flag and the buffer lock. The read request flag is a single byte that can hold either 0 or 1. It is used only at the send buffer. When the flag is set (its value is 1), it means that the remote side has already sent a read request message to ask for new data and is actually waiting for them. As soon as new data arrive, they will be sent after checking this flag.

b. Connection establishment

Figure 51 illustrates the timing for connection establishment. Each connection must have a unique connection ID, in order to be distinguishable among others. We aim to support networks with more than two local nodes and therefore each node could have multiple local open connections. As a result, these connection IDs cannot be global -- this would require everybody to be informed about a new connection. Another solution would be to have dedicated connection IDs for each remote peer. This, apart from wasting resources, would require every sent message to include both the connection and the sender ID. Depending on how many users and how many connections per remote user we would have to support, this information could be many bits per message. Instead, we chose to use a global set of connection IDs for each node. When PeerA tries to connect with a remote side, it will initially search for the first available of its IDs, let's for example assume this is number 3. Subsequently, it will inform PeerB that it will identify this particular connection as connection 3. PeerB then, will follow the same procedure and choose its own ID, let's say number 5, which must be sent to PeerA. Now, for every message referring to this connection, each one will write the ID of the remote side as the message's connection ID. PeerA will send messages with ID 5, while PeerB will use number 3.

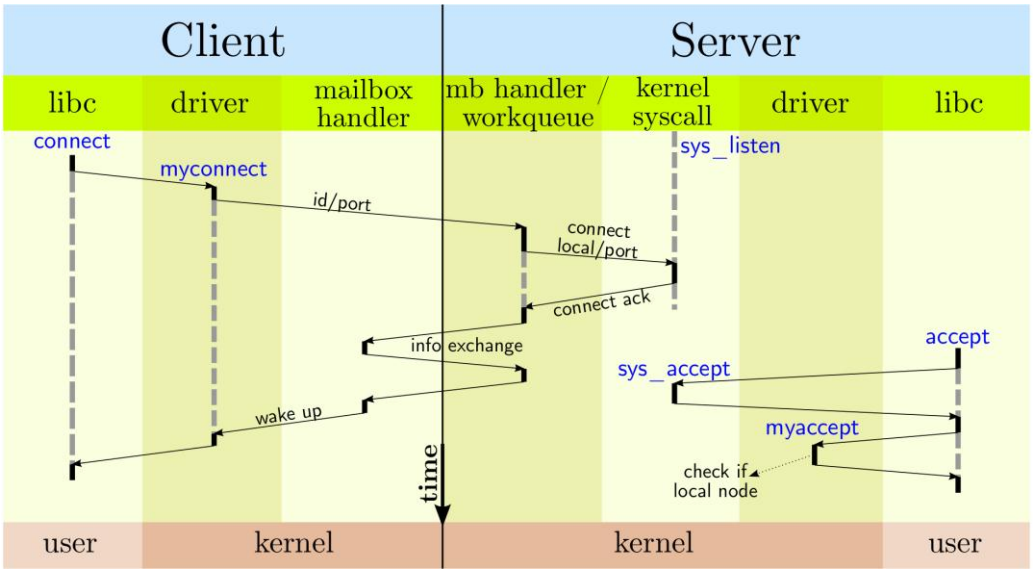


Figure 51: Connection establishment (between nodes on the internal network).

Several data structures are used to keep the state of an open local connection. Some belong to the RDMA driver, while others are global variables in our modified libc. The RDMA driver has a table of active connections. Each element of the table is a pointer to a connection struct. Its index number is the connection's ID. Null pointers denote available IDs. When a new ID must be obtained, the accompanying semaphore of the table is acquired and then the first empty element is used. Most calls to the RDMA driver refer to a connection, so this table is searched. This operation requires only a read to an element of the table and therefore does not need holding the semaphore. Except for the connections table, another data structure held in kernel space is the newconnections structure

In user space, connections are kept in a libc connections table. The difference here is that the connection ID is not the index of the element but the element's value. In this table the indices denote a file descriptor number. Consequently, when an intercepted system call is issued, the file descriptor is checked in libc connections table. A value of zero indicates that the descriptor does not belong to a local connection (or is not a socket descriptor, at all). A positive value is a local connection. Because connection IDs begin from 0, the value stored in the table is actually connID+1. Moreover, since it is very rare for a process to use more than a few dozen descriptors, the libc connections table has a length of 64 elements and then is continued as a linked list. Traversing the whole list is rarely needed. The connection struct is a structure used by the RDMA driver. All information regarding a local connection is gathered here: the ID of the remote peer, the remote connection ID, or the port and the remote port numbers. Moreover, additional information like the connection status (e.g. whether it is connected or still connecting) and the processes related to this connection are stored here. Finally, the details of the connection's local and remote buffers are also part of this struct.

Every connection has a pair of send and receive buffers on which all remote DMA operations occur. These buffers are allocated during the connection establishment phase and are then also mapped to user space. The driver keeps pointers to these buffers in the connection struct, whereas in libc there is another global table that resembles the libc connections table, using file descriptor numbers as

indices. The values of its elements are though, pointers to the mapped data. Both tables are updated at the same time.

When the client calls connect, our injected code checks whether this destination belongs to the local network by reading the peers structure. If it does, the original system call is aborted and a negotiation with the remote (within the local network) side is initiated by exchanging a series of mailbox messages. All these actions take place in kernel space, by the two RDMA drivers.

Before contacting the server side, the client creates a new connection struct and finds a new connection ID. Then a connection request (ConnRQ) mailbox message is sent and the client gets to sleep until a respond arrives. The 0C value on the left is the message ID, from which the server's mailbox interrupt handler will understand that it is a connection request message. The connection request is the only type of message where its connection ID, the second field, does not refer to an ID belonging to the receiver of the message, but the sender. In other words, the client here says: "I want a new connection that I will identify with this ID". Of course, its node ID must also be included so the server knows who to respond to. The last piece of information needed by the server is the port number, to associate the request with one of its listening sockets. The difficulty at this point is that these sockets must be able to connect to both internal and external destinations. Listener waits for incoming connections inside the kernel and we do not want to modify the kernel. We could intercept listen and have both the kernel and our driver waiting at this port number, the first waiting for network packets and the second for mailbox messages. Unfortunately, this way no arbitration of the connections could happen. Due to the multiple waiting sides, we would not be able to tell which connection is the first in a case of close arrivals.

Therefore, when the connRQ message arrives a connection to the real socket is attempted. This connection is a "dummy" because it is not actually going to be used, but it serves only the purpose of confirming a connection to the real socket. The RDMA driver creates a new socket and issues a normal localhost connect to the same port. By localhost connection we mean a connection from within the node itself, via its loopback network interface that has the special IP address 127.0.0.1. Because connect is a blocking call, it cannot be executed from the interrupt handler (sleeping inside an interrupt handler is not allowed) that received the message. So, this action is actually scheduled to the kernel's default workqueue (Workqueues are special kernel mechanisms where tasks can be scheduled to run as kernel threads).

If the localhost connection succeeds, our connection procedure can be resumed. The server allocates a connection struct and gets its own connection ID for this connection. Furthermore, it allocates a pair of connection buffers. Afterwards, a response is sent to the client (i.e. the first server message). From now on, every message sent by either side until the connection is established will be a connection reply (ConnRPL) type message, having a 0A as message ID. Additionally, all messages will contain the receiver's connection ID, in the connection ID field, so the receiver can recognise it. As a result, the server has to send his local connection ID, which the client is not aware of, on his first response.

At the client side the (intercepted) connect has not returned yet. The process is sleeping inside the call to the RDMA driver. When the server sends a respond, the client's interrupt handler knows which process has to wake up (which is in kernel space -the RDMA driver - and will remain there until the end of the connection procedure) because this information is already stored in the connection struct of the connection. On the other hand, at the server side everything is still done by the driver on its own and not on behalf of the server process. The mailbox interrupt handler receives the messages and continues the procedure (or schedules to the workqueue when it cannot sleep).

To finish the connection establishment procedure, the two sides exchange the physical addresses of their local connection buffers. Finally, the server acknowledges this last reception and then at the client side, the call to the RDMA driver returns the connection ID of the new connection, to be stored in libc connections table. The connect call can then return with a successful return value and the client is ready to use the connection from this point.

The server process is handed over a connection with a call to accept. Unfortunately, the same problem we faced with listen, applies to accept as well. We cannot intercept accept and ignore external accepted connections. To overcome this difficulty, the "dummy" localhost connection will also be used here. We will just let the server process accept it normally.

However, two issues arise with this approach. First, how can we identify our local connection? When accept succeeds, it returns a new socket descriptor for the connection. We need to know this number in order to intercept all upcoming data transactions. The solution here is to use post-kernel interception. We let all accept calls run in the kernel and then before returning to the process, we match the new sockets with our localhost connected sockets. The second problem is synchronisation; accept can be called at any time, before or after the client calls connect. When called before, as soon as the dummy localhost connection succeeds, both connect of the workqueue and accept of the process will return. The latter should not be allowed to happen this moment because the process could then try to use the connection before our connection establishment has finished.

The newconnections structure is used to synchronise these events. Before any "dummy" localhost connection is issued, newconnections gets updated to show there is a pending connection. If it succeeds, it remains there, in a list of completed connections. On the other hand, when the real accept returns from kernel, our intercepted wrapper checks if the accepted address belongs to localhost. In this case, our driver is called and the newconnections structure is examined. If completed connections exist there, each one of them is also examined. Its pair is identified by the port number given to the "dummy" socket of the RDMA driver. However, accept may get there first, before connect has even returned. For this reason we keep the pending connections count. If no completed connection matches the accepted connection but there are pending connections, the accept side must wait for them to complete and then check them as well.

After a match of an accepted connection with a "dummy" socket, the rest of the connection establishment procedure must complete, if not yet, and then the intercepted accept can finally return to the process. Before this, the RDMA driver has returned the value of the new connection ID. Generally, because it is unknown whether accept will have to wait or not, completions are used.

Completions are a kernel mechanism to wait on an event. If the event is already completed there will be no waiting. Ultimately, the dummy socket is closed and released, as it is not useful anymore. On the other hand the accepted descriptor is not closed since this could result in the kernel giving the same descriptor number to a new file.

One side effect of accepting local connections this way is that some irrelevant connections will be delayed with no reason. However, only accepted connections from localhost are examined. Furthermore, dummy connections are separated by port number in newconnections - here we mean the listening port. These two facts minimise the possibility of false waiting radically. For example, if the listening port is number 50000, only accepted connections from localhost will make the call to our driver and then in newconnections only connections for port 50000 will be looked. The matching criterion will be the other port number.

As soon as the connection establishment procedure has completed, the local sockets are ready to be used for data transactions. The two connected sides behave in the same manner from this point and the distinction between server and client is not applied anymore. Data transferring occurs when calling the pair of send/recv system calls or the more general write/read. Some variations of these calls exist, but there is not any substantial difference on the way that data are transferred.

References

[1] PCI-SIG Single Root I/O Virtualization www.pcisig.com

[2] ARM White Paper "Virtualization is Coming to a Platform Near You: The ARM® Architecture Virtualization Extensions and the importance of System MMU for virtualized solutions and beyond" Roberto Mija, Andy Nightingale 2011

[3] ARM Ltd. ARM System Memory Management Unit (MMU) architecture ARM IHI 0062B, 2012.

[4] Rusty Russell "virtio: towards a de-facto standard for virtual I/O devices" ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel

[5] Juno ARM Development Platform (ADP) SoC Technical Reference Manual ARM DDI 0515B

[8] Virtual I/O Device (VIRTIO) Version 1.0 Committee Specification Draft 01 Public Review Draft 01 03 December 2013 http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf

[9] VFIO "Virtual Function I/O" https://www.kernel.org/doc/Documentation/vfio.txt

[10] VFIO: A user's perspective (Alex Williamson) http://www.linux-kvm.org/wiki/images/b/b4/2012-forum-VFIO.pdf

[11] vfio. VFIO driver: Non-privileged user level pci drivers, 2010. http://lwn.net/Articles/391459/

[12] Platform Device Assignment to KVM-on-ARM Virtual Machines via VFIO Antonios Motakis, Alvise Rigo, Daniel Raho , 2014 IEEE DOI 10.1109/EUC.2014.32 978-0-7695-5249-1/14, 2014 International Conference on Embedded and Ubiquitous Computing

[13] VFIO support for platform devices http://lists.linuxfoundation.org/pipermail/iommu/2015-March/012352.html

[14] Testing VFIO_PLATFORM with the PL330 DMA Controller by Virtual Open Systems http://www.virtualopensystems.com/en/solutions/guides/vfio-on-arm/

[15] OpenFabrics Alliance, http://www.openfabrics.org

[16] RDMA Consortium, http://www.rdmaconsortium.org

[17] InfiniBand Trade Association, http://www.infinibandta.org