Project Nº: 610456

D4.3 Kernel-level memory and I/O resource sharing across chiplets

September 30, 2015

Abstract:

This deliverable describes the results of Task 4.3 (UTILIZE) for re-architecting and targeting the systems software stack to allow both scaling by adding more resources and efficient use of all available resources.

Document Manager	
Authors	AFFILIATION
John Velegrakis, Manolis Marazakis	FORTH
Luis Garrido, Paul Carpenter	BSC
Per Stenstrom	CHAL
John Thomson, Michail Flouris, Anastassios Nanos, Xenia Ragiadakou, Julian Chesterfield	ONAPP
Reviewers	
John Thomson	OnApp
Paul Carpenter	BSC
Nikolaos Chrysos, Fabien Chaix	FORTH

Document Id N°:	D4.3	**Version:**	2.0	**Date:**	30/09/2015

Filename:	EUROSERVER_D4.3_v2.0.docx

Confidentiality

The EUROSERVER Consortium consists of the following partners:

Participant no.	Participant 2rganization 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

Revision history

Version	Author	Notes
0.5		June 2015: Submitted as working draft before interim review
0.6		Updated material on MicroVisor (provided by OnApp)
0.7		New material on tmem (provided by BSC)
0.8		New material on virtualized storage (provided by FORTH)
0.81		Added 'global view for WP4' in Introduction.
0.9		Added summary of achievements
0.91		Edits based on review by OnApp
0.92		Edits by BSC in "memory sharing across coherence islands"
0.93		Edits in Executive Summary (common), edits in Appendix (BSC), added 'deviations' (FORTH)
1.0		Finalized text (for the PO and project reviewers)
2.0		All modifications accepted (no track change)

Contents

List of Abbreviations

Term	Definition
ACP	Accelerator Coherency Port
API	Application Programmer Interface
ATAoE	ATA (AT Attachment) over Ethernet
AXI	Advanced Extensible Interface
CDMA	Central DMA (Xilinx hardware IP block for direct memory access)
DDR	Double Data Rate
DMA	Direct Memory Access
DP	Discrete Prototype
DRAM	Dynamic Random Access Memory
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
GPL	GNU General Public License
JTAG	Joint Test Action Group
KVM	Kernel-based Virtual Machine
HP	High-Performance Port (non-coherent)
HV	Hypervisor
I/O	Input/Output
iSCSI	Internet Small Computer System Interface
IOPS	I/O Operations Per Second
LVDS	Low-Voltage Differential Signal
MCVMM	Memory Compression Virtual Memory Management
NFV	Network Function Virtualisation
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Access
NVM	Non-Volatile Memory
OS	Operating System
PCI	Peripheral Component Interconnect
PL	(Xilinx) Programmable Logic
PS	(Xilinx) Processing System
PV	Paravirtual
REST	Representational State Transfer
RDMA	Remote DMA
SD	Secure Digital
SDN	Software Defined Network
SMP	Symmetric Multiprocessor
SoC	System on Chip
TCP	Transmission Control Protocol
TMEM	Transcendent Memory
TPC	Transaction Processing Performance Council
VM	Virtual Machine
VMM	Virtual Machine Monitor

List of Figures

Executive Summary

The EUROSERVER project is paving the way for a novel converged, microserver architecture that will advance the state of the art for both software and hardware in the data center. 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 such as *share-everything* that set it apart from existing platforms.

This deliverable describes the outcomes of Task 4.3 (UTILIZE), and specifically of subtasks T4.3.1 and T4.3.2. The objective of these tasks is to re-architect the systems software stack, so as to achieve a more efficient use of available resources, and also to improve scalability. The resources under consideration are primarily memory pools, but also include fast I/O devices (storage and network).The accomplishments described are as follows:

- Exploration and evaluation (in terms of latency) of a control and data path from a compute unit to a fast storage device, and techniques to share this device
- Hypervisor support for sharing remote resources (the MicroVisor concept, based on Xen)
- Mitigation techniques for performance interference in the storage I/O path
- Use cases for utilizing remote memory: (a) memory capacity extension in the OS, (b) swap device, (c) memory allocator on top of an I/O device.
- Evaluation of remote memory access primitives: load/store, DMA
- Emulation of memory capacity sharing across coherence islands in the Xen hypervisor (basis of MicroVisor), using the Transcedent Memory (tmem) mechanism
- Progress towards transparently compressed main memory (topic of a separate upcoming deliverable: D4.6, due by M30).

This deliverable starts with a detailed exploration of a case study in I/O resource sharing in the content of the EUROSERVER 32-bit discrete prototype: the physical control and data path for accessing a remote storage device, specifically a NVM-Express SSD. We provide a breakdown of the overall latency in servicing I/O requests, and identify the high relative contribution of software layers. We identify alternatives for sharing this access path, using software to mediate accesses to the shared storage device.

We then describe the architecture and implementation of MicroVisor, a multi-node hypervisor (based on the Xen Type-I hypervisor) that aims to utilize efficiently remote memory and I/O resources. The core idea behind the MicroVisor is to remove the dependency on the local control domain (dom0) for control operations such as virtual machine setup, booting and resource assignment, and to instead move this functionality into the Xen hypervisor layer itself. Hardware driver support is still maintained outside the Xen layer through a lightweight driver domain interface to provide ultra-low overhead for accessing hardware. As a further optimization, each hardware component can actually utilize its own super-thin isolated driver domain to handle the virtualized access to its resource. NIC driver functionality, for example, can be migrated into a dedicated helper domain (e.g. via Mini-OS integration) that only has access to that hardware in order to process packets over the wire, and place them onto the Xen Hypervisor switch via a generic Xen netfront interface.

With the capability to share resources comes the concern of performance interference among co-located workloads. Future servers will include large numbers of cores, memories, and I/O resources, and will host a large number of applications and services. Currently, resource management in servers has to handle shared control and data paths in the OS kernel (or the hypervisor), resulting in severe interference, non-determinism, increased overheads and contention, all of which ultimately limit efficiency and scalability. To address the unwanted interference between workloads, we introduce the

concept of "system slices". Each system slice consists of private control and data paths in the OS kernel and in the hypervisor, as well as an allocation of one or more processing cores, memories, storage and network access paths. In the context of the EUROSERVER architecture, each "system slice" may include resources from different chiplets and parts of the overall server chip. Each slice contains cores from a single coherence island; however memory and I/O resources may belong to different coherence islands. We present a summary of experiments that stress the I/O path using data-center workloads. Our focus is on performance interference as well as on the impact of non-uniform memory access times. These experiments were conducted on mainstream x86 servers, but the results provide useful insight into severe performance degradation effects when multiple workloads share server resources. The partitioned I/O path design evaluated in our experiments is shown to improve isolation across collocated workloads, by assigning them to different system slices.

We present an evaluation of methods supported by EUROSERVER's UNIMEM memory architecture to utilize remote memory located across multiple coherence islands. The evaluation is conducted on the 32-bit discrete prototype of EUROSERVER, and includes results for both fine-grained memory accesses (load and store instructions) and coarse-grained memory transfers (DMA operations). Moreover, we evaluate the case where remote memory is used as a swap device. We also consider how to use remote memory resources through a standard memory allocator, via an I/O device interface.

In the UNIMEM memory architecture, memory sharing across coherence islands is seen as an essential capability for improving performance and energy efficiency. Particularly, for data-center workloads that require varying amounts of memory, over-provisioning memory allocation per server to handle the peak requirement is expensive, both in terms of capital costs and power consumption. We present a summary of our ongoing efforts towards sharing of non-cache coherent memories across chiplets, as per the UNIMEM memory architecture, using virtualisation and page-level accesses, on the discrete prototype and the MicroVisor platform.

Finally, we provide a brief outline of work recently initiated on a software module for managing compressed memory, addressing the need to offer increased main-memory capacity at low cost, low energy and high performance. A key assumption is that it will transparently manage the compressed memory without any changes to existing system software. One approach to be investigated is to expose the memory freed up by compression to a swap ramdisk, called victim cache. This will allow pages selected for eviction to stay in system memory longer, without being written back to disk.

The mechanisms described in this deliverable are integral parts of the proposed EUROSERVER microserver platform. The complete work being done 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

- Initial work on virtualisation involved porting the Xen and Xvisor hypervisors to the 32-bit discrete prototype. These efforts provided valuable technical insights, but reached limitations of the hardware platform. In view of the planned transition to a 64-bit discreet prototype and the final chiplet-based prototype, these lines of work are no longer active, to better consolidate WP4 efforts.

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 (summarized in Figure 1) are focused on improving the software stack for microserver platforms. In particular the software stack looks to create a holistic software stack using low resource nodes and benefitting from the unique aspects of the EUROSERVER architecture, such as UNIMEM. 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. 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 architectures 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.

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 key 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.

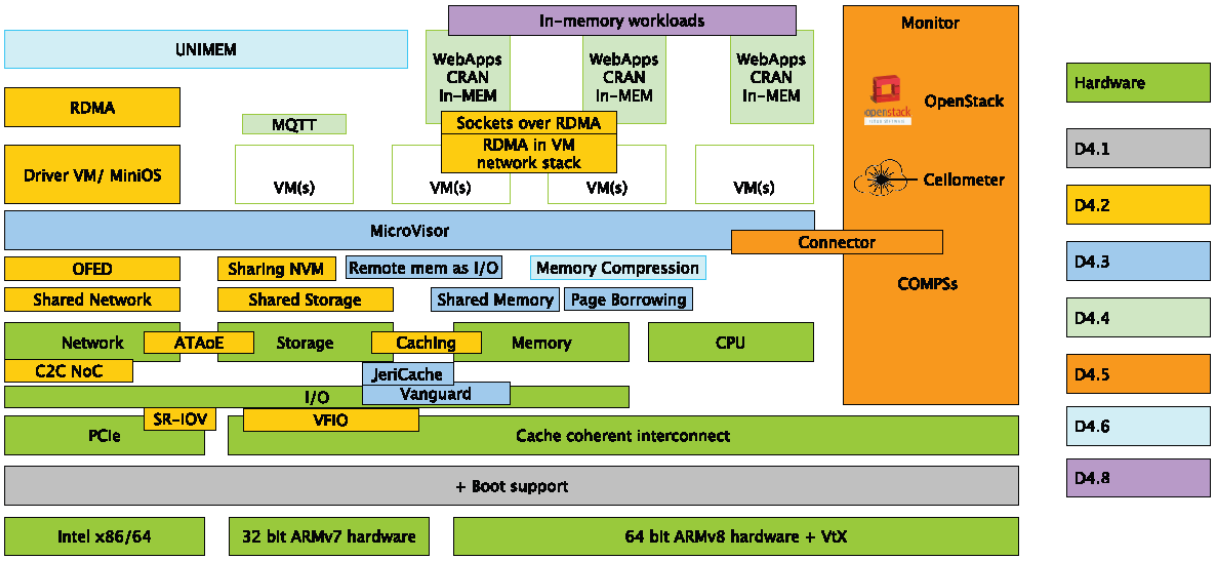


Figure 1: Diagram showing the areas being worked on in EUROSERVER in WP4.

This deliverable describes the outcomes of Task 4.3 (UTILIZE), and specifically of subtasks T4.3.1 and T4.3.2. The objective of these tasks is to re-architect the systems software stack, so as to achieve a more efficient use of available resources, and also to improve scalability. The resources under consideration are primarily memory pools, but also include fast I/O devices (storage and network). The work has been carried out with the proposed EUROSERVER Final Prototype as the target platform.

This deliverable is organized as follows. Section 2 reports on work on sharing I/O resources, such as fast storage and network devices. We start this section with a case study developed on the EUROSERVER 32-bit discrete prototype: support for access to a fast storage device shared among microservers. We then present the design and initial implementation and validation of a hypervisor that takes into account remote resources within a microserver environment. NUMA effects and performance interference are also discussed. Section 3 focuses on memory resources, specifically the problem of utilizing remote memory pools for expanding the available memory capacity without the currently common coherence mechanisms. Section 4 provides an introduction to the problem of transparently applying compression to increase the memory capacity available to workloads in the EUROSERVER microserver environment. Finally, Section 5 concludes this deliverable by summarizing key findings and outlining directions for future work.

2. Resource Sharing(FORTH, OnApp, BSC)

Sharing a Remote NVM Storage Device (FORTH)

Emerging flash-based storage devices provide access latency in the order of a few microseconds. Currently available storage devices (e.g. PCI-Express Solid-State Drives) provide read and write latencies in the order of 68 µs and 15 µs respectively (numbers specifically for Fusion I/O). These performance numbers are projected to become significantly lower in next-generation non-volatile memory (NVM) devices. Given these trends, the software overhead of the host I/O path in modern servers is becoming the main bottleneck for achieving microsecond-level response times for application I/O operations. In addition, these overheads translate to fewer I/O operations being served per core, making the software overhead in the I/O path the limit in increasing the number of I/O operations per second (IOPS), rather than the storage devices. Therefore, in this new landscape, it becomes imperative to re-design the I/O path in a manner that it will be able to keep up with shrinking device and network latencies and to allow applications to benefit from such devices.

We have worked on the design of a storage stack for efficient, low-overhead access to fast storage devices in the context of a microserver, where devices are shared among compute nodes. The storage device we selected is a Samsung XS1715 SSD, with a PCI-Express connector, that supports the NVM-Express (Non-Volatile Memory Express) specification. There are two major advantages for this class of storage devices:

(a) Lower latency, as the storage stack in the Linux kernel is more streamlined (i.e consists of fewer layers) and relies on in-memory command and completion queues, rather than hardware-level (command-register) accesses. Devices are still presented as block-layer devices to the kernel's VFS layer, so that they can host standard file-systems.

(b) Higher concurrency levels, via support for multiple independent command and completion queues, with higher queue depths (as the queues are kept in the host's main memory).

Figure 2 illustrates the key components of our experimental system configuration for exploring a remote storage I/O path, implemented within the 32-bit discrete prototype (see EUROSERVER deliverable D4.1). Table 1 lists the major components sued, and explains their role in the overall storage I/O path.

Table 1: Components of storage I/O path to remote SSD, in the EUROSERVER 32-bit discrete prototype.

Component (H/W or S/W)	**Location**	**Functionality**
Chip-to-Chip connection (custom H/W IP block)	Compute Unit, I/O FPGA	Connectivity to I/O FPGA, via AX4 system interconnect and FMC connector. (a) Interfaces to ARM processing system within the MicroZed FPGA board (ACP memory access port). (b) Interfaces to address translation logic within the I/O FPGA.
PCI-Express Root Complex (H/W IP block, from Xilinx)	I/O FPGA	Orchestrates device discovery protocol (as per PCI-Express specifications), and mediates accesses to devices (via address translation windows). Supports 4 lanes of 5.0 GT/sec (PCI-Express Gen.2), and Message-Signaled Interrupts (MSI).

Samsung XS1715 SSD (H/W device)	I/O FPGA (via daughter card)	High-performance NAND Flash storage device, with PCI-Express connectivity (Gen.3, x4)
Address Translation – Forward direction (compute unit-to-device) and Reverse direction (device-to-compute unit) (custom H/W IP blocks)	I/O FPGA	Translates addresses in PCI-Express device accesses (load/store, DMA) from compute unit to PCI-Express Root Complex, as well as in the reverse direction. Configuration via firmware running on embedded microcontroller (MicroBlaze soft-IP core) in the I/O FPGA.
PCI-Express bridge driver (mainline Linux kernel driver)	Compute Unit	Controls the PCI-Express Root Complex – unmodified, as the underlying hardware convinces the Linux kernel on the compute unit that the device is locally attached.
NVM-Express storage drivers (mainline Linux kernel drivers)	Compute Unit	Present (a portion of) the SSD storage device as a block-device to the Linux kernel (i.e. suitable to serve as the backing storage for a filesystem) Unmodified Linux kernel drivers: nvm-core, nvm-scsi.

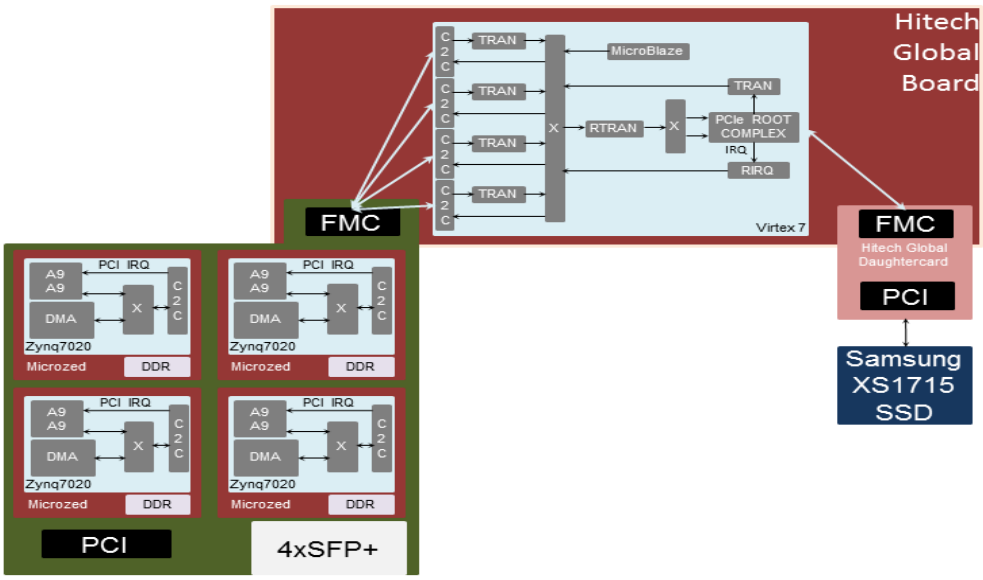


Figure 2: Storage I/O path to PCI-Express SSD on the EUROSERVER 32-bit prototype.

We have instantiated a PCI-Express Root Complex combined with a AXI Bridge IP block within the central I/O FPGA (Hitech Global board), connected via a FMC connector to daughter-card with a PCI-

Express connector. A MicroBlaze processor core (soft IP block) within the I/O FPGA runs the prototype's monitor program, implementing essential firmware functions and configuration support. Compute units, implemented as MicroZed boards, are connected to the I/O FPGA via a FMC connector. For each of the compute units, there is a dedicated physical path to/from the I/O FPGA (3.2 Gbps raw bandwidth, full-duplex), implemented with instances of our custom Chip-to-Chip I/P block connected to an AXI4 system interconnect.

Addresses are translated via address translation IP blocks, both in the direction from the compute units to the (remote) PCI-Express Root Complex (marked 'TRAN' in the figure) as well as in the reverse direction (marked 'RTRAN' in the figure). Moreover, a custom IP block implements the logic needed for forwarding interrupt signals from the PCI-Express Root Complex to the (remote) compute units. With these arrangements in place, we have successfully implemented a storage I/O path from a compute unit (MicroZed board) to the SSD device.

Using the fio utility with queue-depth=1, i.e. with a single I/O request at a time, we have measured that the total time to service a page-sized (4KB) request from a remotely-attached storage device is 147 microseconds. Using hardware tracing, we have measured components of this total time due to the data path from the compute unit to the storage device (and back) and the raw data transfer time. These components are listed in the following table. There are differences in the collected timings between reads and writes, specifically in the steps for interrupt handling and in the actual transfer of data to/from memory, as in marked Table 2. For read I/O, the components add up to 30.5 microseconds, whereas for write I/O the corresponding total is 32.6 microseconds.

Table 2: Timing of steps in servicing a 4KB I/O request from remote SSD.

I/O processing step	Time (microseconds)	Comment
2 writes from compute unit to the SSD's PCIe BAR	2 x 0.79 = 1.58	(1) Update of NVM-Express submission queue tail after the SCSI command has been placed in memory (2) Doorbell (causes device to pick up SCSI command)
Fetch SCSI commands from main memory of compute unit	1.088	Submission queues in NVM-Express devices are kept in the host's main memory, thus drivers do not have to write directly to device memory.
Transfer of data to/from to the main memory of the compute unit	Read I/O: 22.06 Write I/O: 35.58	For read I/O, the storage device writes data to the compute unit's main memory. For write I/O, the storage device reads from the compute unit's main memory. 4KB transferred in bursts: 128 bytes per burst for read, 512 bytes per burst for write.

I/O processing step	Time (microseconds)	Comment
I/O completion	1.208	Completion queues in NVM-Express devices are kept in the host's main memory.
MSI-(X) interrupt generation	1.208	Storage device posts a write to a FIFO in the PCI-Express Root Complex.
Interrupt handling	For Read I/O: 4 x 0.76 + 2 x 0.79 = 4.62 For Write I/O: 2*0.76 + 2*1.8 2*0.79 = 6.7	• 4 read accesses to PCI-Express Root Complex: interrupt decode register, interrupt mask register, FIFO (x2) • 2 write accesses: FIFO, interrupt decode register (clear) • 2 of the FIFO reads are slower for write I/O than in the case of read I/O (1.8 instead of 0.79 microseconds)

We have measured that only 22% of the total time to service a page-sized (4KB) read I/O request from a remotely-attached storage device is spent in the hardware data path and the storage device itself. For write I/O, the corresponding portion is 32%, due to transfers of data to the main memory of the compute unit being slower than transfers in the opposite direction. The remaining 68-78% of the observed I/O latency is attributed to software layers (OS kernel and I/O-issuing application).

We expect the I/O latency attributed to (physical) access to the storage device to almost halve in the near-term. However, if the software layer overheads are not proportionately reduced, we will not be seeing a significant reduction of the observed I/O latency. This motivates further work in two directions: (a) reducing the complexity and overhead of the storage I/O path in the OS kernel, and (b) sharing of such fast storage devices among several processing cores. For the first direction, we would need to radically reconsider the interface to storage, bypassing as much of the OS kernel layers as possible while still maintaining adequate protection boundaries between processes. The protection challenge is already being considered by the EUROSERVER consortium in the context of work for I/O virtualisation. For the second direction, the most promising guideline for system design is to maintain several concurrent streams of I/O requests, originating from multiple compute engines.

In the EUROSERVER 32-bit discrete prototype, the processing cores (ARM A9, 32-bit, designed for the embedded market rather than servers) cannot issue I/O requests fast enough to fully utilize a fast non-volatile memory device such as the SSD used in our experiments. As a reference point, the datasheet for the Samsung XS1715 SSD reports a read latency of 90 microseconds (4KB random reads, using the fio utility with queue-depth=1) as measured on current-generation mainstream servers (x86-based). In our experimental environment, this latency is 147 microseconds (47% higher). The hardware data path between compute units and devices is affected by the underlying interconnection technologies, and we feel that its contribution to I/O latency is rather over-represented in our current FPGA-based testbed (for example, we are using a Gen.2 PCI-Express root Complex with a Gen.3 storage device). Assuming for simplicity (first-order approximation) that with a more performant compute unit we will not change the hardware data path to/from the (remote) storage device, we would expect that the

software components of I/O latency would be in range 48-65%. In this scenario, the hardware data path to remote storage would account for 35% of the overall latency in the case of read I/O, and 52% in the case of write I/O.

The hardware infrastructure in the current prototype provides a single compute unit (which we call the 'storage host') with a remote access path to the storage device, allowing the use of unmodified Linux kernel drivers. To share the storage device among compute units, we have relied on the partitioning feature of Linux block devices: specific, non-overlapping block ranges of a block device are presented as separate (logical) devices to the kernel at the 'storage host'. In this manner, the 'storage host' with the physical path to the storage device can export as many virtual storage devices as there are partitions. The other compute units access their corresponding storage device partitions over a network storage protocol, either at the block level (e.g. nbd, iscsi) or at the filesystem level (e.g. nfs, cifs, sshfs). This centralized software-based configuration makes sharing among the compute units relatively easy to setup, but limits system scalability up to the limits of the 'storage host' (primarily CPU and network I/O capacities, but also affected by filesystem and I/O protocol overheads).

For future work, we are considering an evolution of the current design, within the upcoming 64-bit discrete prototype, where the partitioning of block ranges is accomplished via a hardware IP block within the I/O FPGA in a transparent manner. Each of the compute units already has a dedicated physical path to the I/O FPGA, so the CPU and network I/O capacities of a single compute unit would no longer be the limiting factor. This design requires that NVM-Express protocol messages are captured and transparently modified at the I/O FPGA.

Hypervisor Support for Remote Resource Sharing (ONAPP)

Virtualisation of server hardware is a commonly used practice to provide scalable resource management and it is essentially the enabling technology for cloud computing. There are many benefits to virtualizing hardware resources, primarily to enable efficient resource sharing (CPU, memory, NICs) across a multi-tenant platform hosting a variety of Operating Systems (OSes).

A Hypervisor (HV) or virtual machine monitor (VMM) is a piece of software that creates, manages and runs Virtual Machines (VMs). Each VM presents an isolated environment for booting a standard Operating System (OS) kernel and running applications within that OS virtual container. The computer on which a hypervisor is running one or more VMs is defined as the host machine, while each VM is called a guest machine. The Hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems.

In addition to commonly deployed commercial hypervisors (HVs), there are two dominant open-source virtualisation platforms: Kernel-based Virtual Machine (KVM) [1] and the Xen Hypervisor [2]. The platforms have mature support for established Intel x86 architecture chipsets and are both making fast advances towards mature support for ARM architecture hardware.

Alternative architectures for the hypervisor in EUROSERVER

Traditional server architectures, having reached frequency scaling limits, and are now improving performance through increasing the number of cache-coherent logical cores and available caches. To improve scalability in a power-efficient manner, recent server architectures implement non-uniform memory access (NUMA), where each CPU core is directly attached to a portion of memory, but is also able to address all system memory, albeit at different speeds and latencies. Due to the power and performance scaling limitations, even with NUMA architectures, new alternative server architectures

like EUROSERVER [3] are emerging, having many non-cache coherent memories between groups of power-efficient processor cores. Relaxing the need for global cache coherency, these new architectures have better scaling properties that allow increasing numbers of CPU cores, while maintaining a high performance-to-power ratio, which is the key metric if hardware is to continue to scale to meet the expected demand of exascale computing and Cloud growth. Additionally, due to their lower power usage and small footprint, these architectures, commonly referred to as microservers, are also able to increase server density in the datacenter, thus lowering costs.

Figure 3 shows the EUROSERVER architecture with multiple compute nodes (light blue color), each having several cache-coherence areas (in red color). It also shows (in yellow) the three types of remote memory that UNIMEM allows to be mapped and accessed from remote CPU cores.

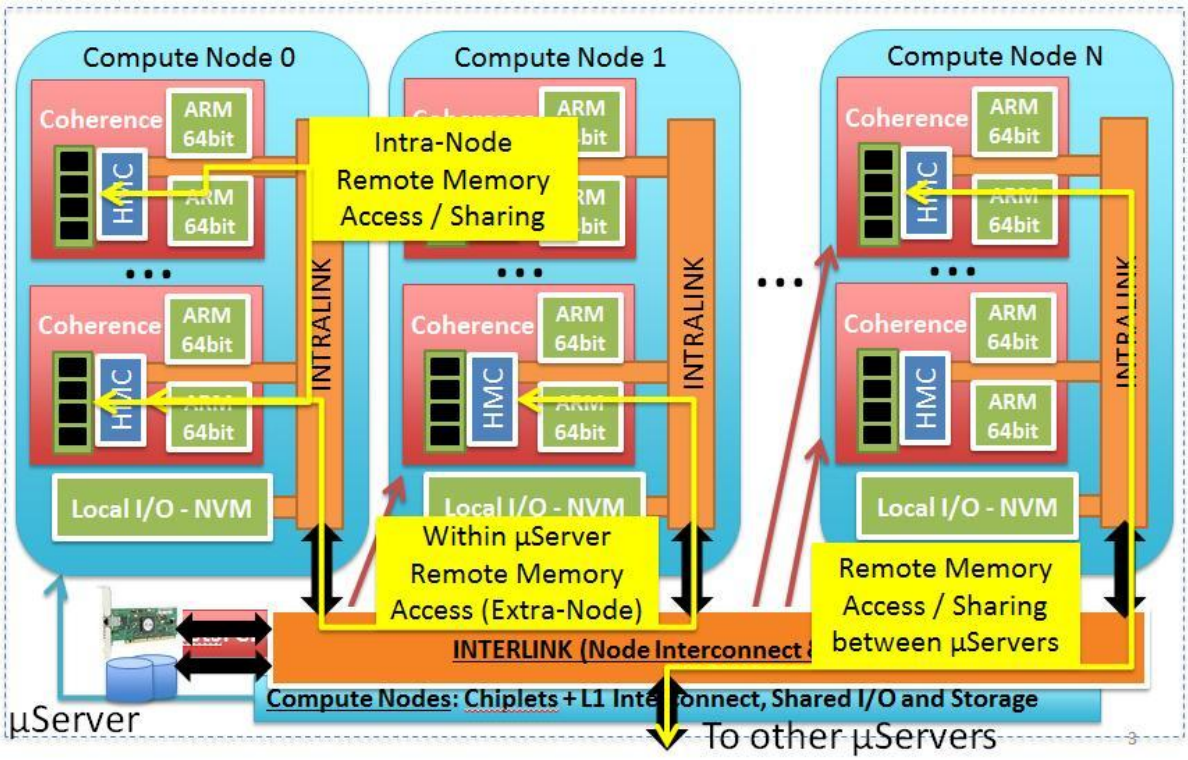


Figure 3: Proposed EUROSERVER architecture showing compute nodes (light blue), cache-coherent areas (light red) and 3 types of remote memories (yellow).

This new paradigm of non-cache-coherent architectures like EUROSERVER, however, imposes changes on system and application software stacks. New OS system architectures are required to enable applications to execute across multiple, low-power, independent, non-uniform memory access (NUMA) and non-cache-coherent islands (which in EUROSERVER corresponds to silicon dies, also denoted as chiplets) with access to shared hardware I/O device resources. All current dominant operating systems and hypervisors (that we are aware of) operate in a globally-shared, cache-coherent memory environment, where a single OS or Hypervisor manages all CPU cores and every CPU core is able to access all memory of the system in a cache-coherent manner. Thus, the EUROSERVER architecture requires a new virtualisation approach with the goal of providing efficient multi-tenant utilization of the microserver hardware, with low overhead management of resources leading to high power efficiency.

One straightforward virtualisation approach for the EUROSERVER architecture is to run a separate hypervisor per cache-coherent island (i.e. chiplet), using current unmodified hypervisor platforms. This

is similar to a multi-node system and can support existing Hypervisor platforms, such as KVM and Xen. Unfortunately, this approach of independent hypervisors per chiplet does not take advantage of the characteristics of the architecture, such as fast inter-processor communication or remote memory management via UNIMEM or remote interrupts. Additionally this approach requires a full host OS (for KVM) or control domain (Dom0 for Xen) to run on each cache-coherent chiplet, leading to increased overheads on both the power and the performance of the architecture. Furthermore with this approach there is no coordinated VM and power management at the compute node or microserver levels. In Figure 4, the left-hand side shows a compute node with many independent hypervisor platforms per chiplet (coherence island), whereas the right-hand side shows a global hypervisor running over all chiplets on a compute node.

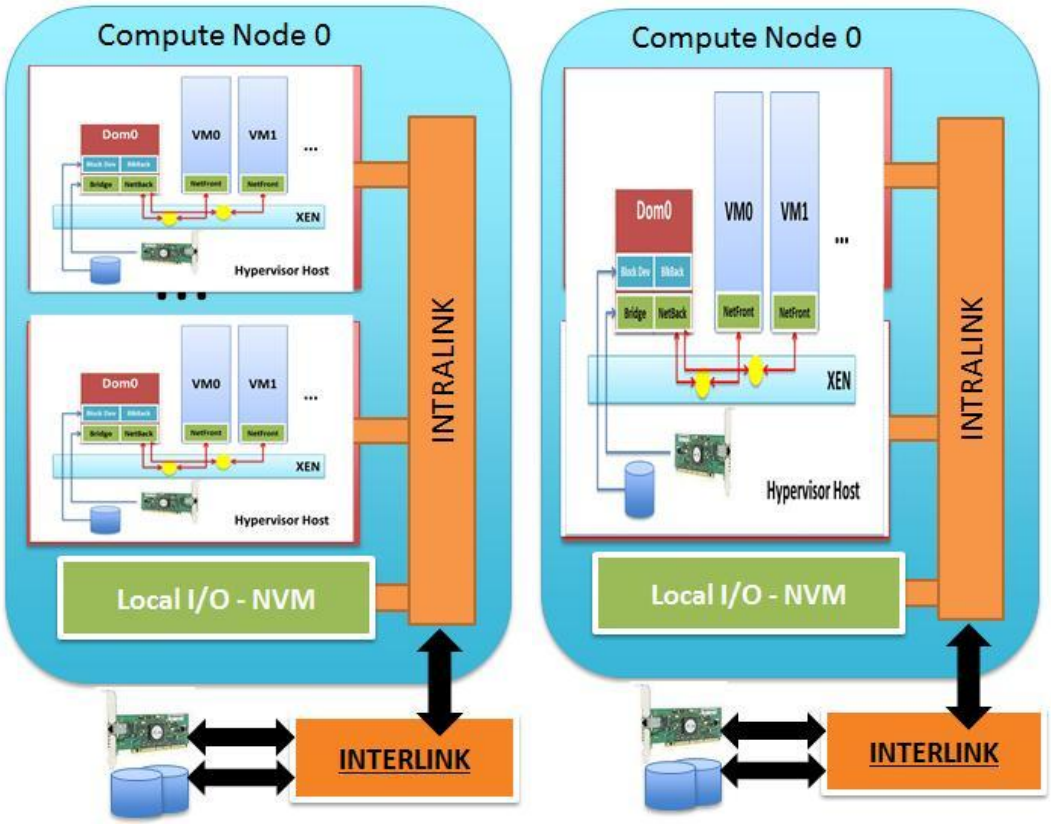


Figure 4: (a) Left side: the EUROSERVER platform running an independent Hypervisor for each chiplet. (b) Right side: the EUROSERVER compute node being managed by a single global Hypervisor.

Our goal for EUROSERVER is to provide virtualisation for a whole compute node (many non-cache-coherent chiplets) with as low overhead as possible, which will allow higher performance and power efficiency. This is demonstrated on the right-side of Figure3. The advantages of this approach is that it will offer a single management domain, the flexibility to utilize all CPUs and memory resources for virtual machines and a global view for performance and power management. However, running a single hypervisor for the whole compute node is not possible using an existing hypervisor platform, since these assume a cache-coherent shared memory model. Building a completely new hypervisor platform designed for the EUROSERVER architecture requires significant redesign and effort that is beyond the scope and timeframe of this project. Below we present and discuss different approaches in providing such virtualisation for the compute node and also present the approach we have taken.

One approach to node-level virtualisation (i.e. one hypervisor per compute node) is to run a single hypervisor on one of the chiplets, which will then control all remote chiplets, managing virtual machines on them. We can name this approach the "Central-Visor", because there is a central hypervisor on one chiplet managing all resources. This approach, however, requires remote interrupts and remote memory mappings to manage VMs, as well as hardware support to map local and remote memory to VMs, handle efficiently remote hypercalls and manage local and remote interrupts. We suggest that, while this "Central-Visor" approach provides a single hypervisor per compute node, it is inefficient, incurring high overheads for VMs that are not local to the hypervisor chiplet, due to remote interrupts and high latencies to remote memory, compared to fast local memory access. The described "Central-Visor" approach is depicted on the left-hand side of Figure 5.

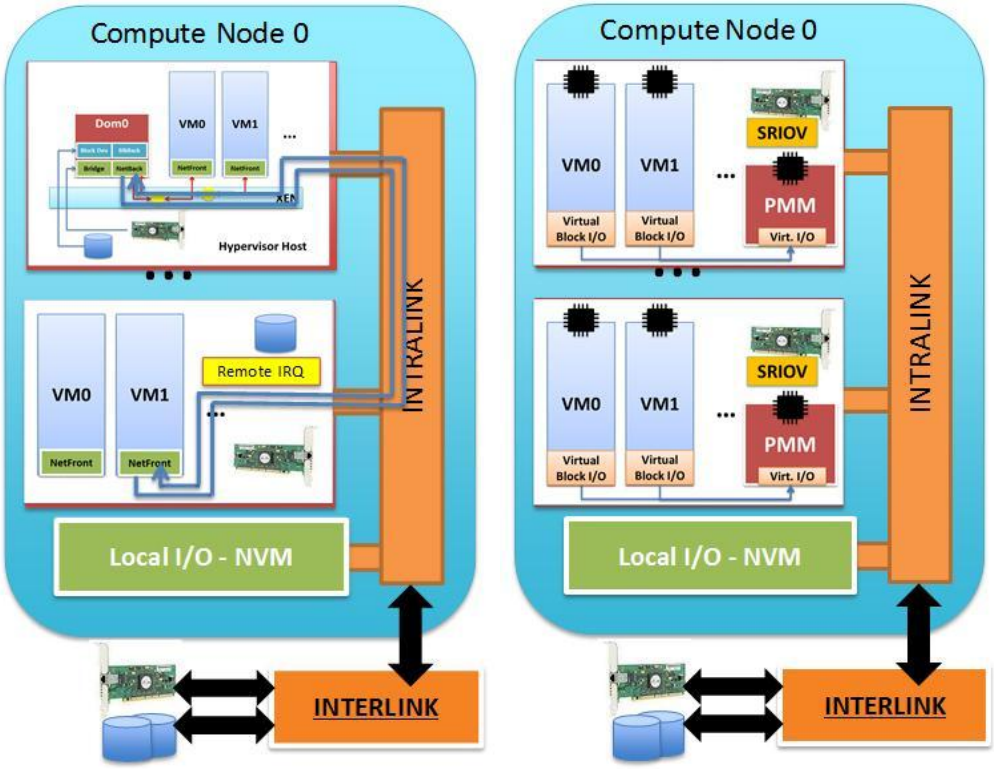


Figure 5: (a) Left side: the "Central-Visor" approach running one main Hypervisor on one chiplet. (b) Right side: depicts the "physicalization" approach, which uses dedicated hardware resources for virtual machines.

Another virtualisation approach for sharing resources to many virtual machines is to restrict the virtual machines to execute on specific physical CPU cores. As can be seen on the right-hand side of Figure 5, in this approach VMs are pinned to CPU cores and each CPU core is dedicated to one VM (but not the opposite, since one VM can use many CPU cores). We call this approach "physicalization", since it dedicates physical resources to virtual machines. In the "physicalization" approach, there is a central "Physicalization Machine Manager" that runs on one CPU core on one chiplet and it manages hardware resource access, mapping physical resources to VMs when they are created or terminated. The "physicalization" approach has certain advantages, such as high performance for VMs, while allowing consolidation of VMs with different operating systems to share the hardware resources. On the other hand, there are several drawbacks to this approach, such as reduced flexibility (e.g. no VM migration), no CPU sharing for workloads, which results in lower resource utilization overall and waste of resources. In general, while "physicalization" can be great for some workloads, it is a backwards step from virtualisation to dedicated resources per OS.

We find all the previously described approaches towards a single hypervisor per compute node (and across coherence islands) as having significant drawbacks that make them inappropriate for efficiently virtualizing a platform such as EUROSERVER. Next, we present our virtualisation approach, which is based on the established Xen hypervisor platform.

In contrast to the KVM hypervisor platform, the Xen hypervisor provides a true Type I Hypervisor. That is to say that the hypervisor layer runs directly on the bare-metal hardware, managing guest OS instances directly above it. There is no Host Operating System required and in this way the Type I architecture is considered to be a minimal, high performance shim. In parallel to the virtualized guest systems running on the Type I Hypervisor, traditional Xen systems utilize a Control domain, known as Dom0, which has privileged access to the hypervisor and is responsible for various tasks including; administering guest virtual machines, managing resource allocation for guests, providing drivers for directly attached hardware, and offering network communication support. Guest Virtual Machines (DomUs) do not typically have direct access to real hardware, and thus all guest domain network and storage communication is managed through paravirtual device drivers hosted in the Control domain (Dom0), which in turn handles safe resource access through multiplexing the physical hardware. Below we provide more details on the current standard Xen architecture and the role of the Control Domain, and then we present our microserver-optimized approach, which we call the “MicroVisor”.

Current Xen Architectural Design and the Control Domain

In the traditional Xen platform, the control domain (Dom0) boots immediately after the Xen Hypervisor and provides user-space tools to signal to the hypervisor via defined hypercall interfaces for the management of guest domains. A user-space application can start, stop, suspend or resume a guest domain (VM), as well as migrate the live state of a guest VM to another physical hypervisor in order to continue its execution. The traditional Xen architecture is shown in Figure 6.

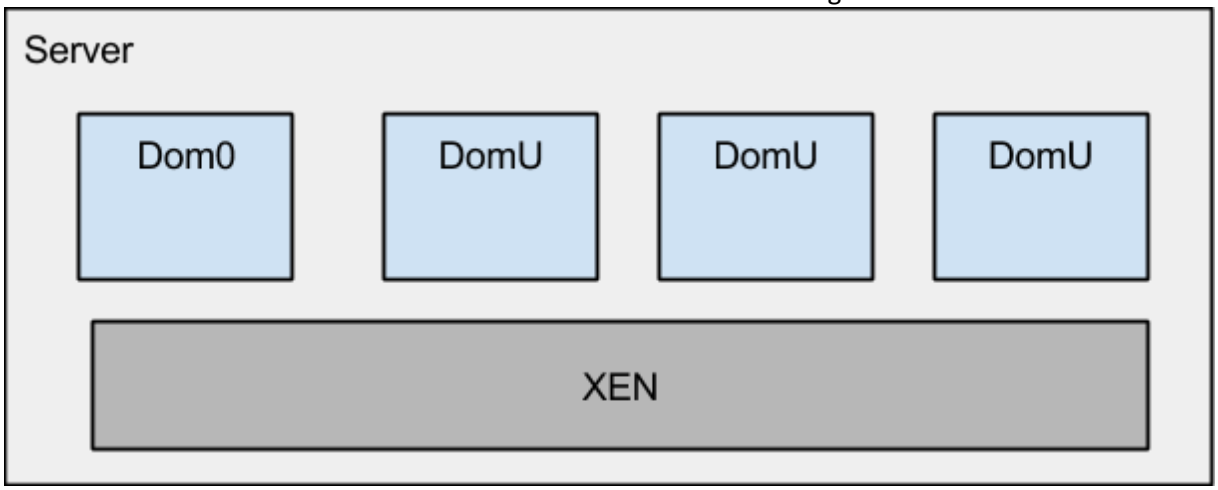


Figure 6: Xen reference architecture with the control domain, Dom0, and guest domains, DomU's.

Additionally, the control domain is responsible for managing the Xenbus communication channel, which is primarily used to signal virtual device information to a guest VM. It also manages the physical memory pool that is shared across virtual machines, as well as signalling to the hypervisor what the physical-to-virtual CPU mapping should be per VM, including features such as the CPU scheduler algorithm.

For every guest VM paravirtualized device that is active, there is a corresponding driver in the control domain that allocates resources and handles the communication ring over which virtual I/O requests are transmitted. The control domain is then responsible for mapping those I/O requests onto the

physical hardware devices behind them such as Ethernet frames over a NIC, or block I/O requests to an iSCSI block device. The drivers, known as frontend (guest-side) and backend (Dom0-side), are implemented using ring buffers between Dom0 and the associated DomU PV device instance.

For accessing data and communicating values over Xenbus, the control domain user-space operating system implements a key/value filesystem-like interface called XenStore. It is managed by a user-space daemon called Xenstored and allows user-space applications to access fields and communicate information to/from any guest hosted on the physical server. It is used for configuration and status information of different guest virtual machines and is accessible with access restrictions to relevant data only from all domains. Xenstore is allocated by the kernel driver of xenbus at the initialization process (xenbus probe).

Concepts of the MicroVisor architecture

There are a number of reasons motivating improvements and changes to the traditional architecture of the Xen hypervisor. First of all, in a platform like EUROSERVER, where there exist many low-power chiplets with limited resources, local memories and no global cache coherence (i.e separate cache-coherent islands), providing full Dom0 management capability for each chiplet (coherence island) is too expensive, wasteful and unnecessary. Secondly, the network performance in such a system is limited, due to context switches to the control domain (Dom0) that handles the network packet switching for all VMs. Finally, network function virtualisation is increasingly being handled on commodity hardware as individual worker VMs to provide enhanced network packet filtering, middlebox functionality and packet forwarding. A new model is required to enhance this architecture by making the packet forwarding layer as fast as possible, with slow path network functions offloaded to separate processing engines (SDN Network Function Virtualisation model).

To address these limitations of the current Xen architecture on low-power microserver architectures, potentially with separate cache-coherence islands we propose a new microserver-optimized approach, which we call the “MicroVisor”.

The core idea behind our approach is to remove any dependency on a local control domain (dom0) for virtual machine setup, booting and resource allocation and instead move this generic functionality into the Xen hypervisor layer itself. Hardware driver support is still maintained outside the Xen layer through a lightweight driver domain interface to provide ultra-low overhead for accessing hardware. As a further optimisation, each hardware component can actually utilise its own super-thin isolated driver domain to handle the virtualized access to its resource. NIC driver functionality for example, can be migrated into a dedicated unikernel helper domain (e.g. via mini-OS integration) that only has access to that hardware in order to process packets over the wire, and place them onto the Xen Hypervisor switch via a generic Xen netfront interface.

Implementation of MicroVisor

Based on the ideas presented in the background and motivation sections, we design and implement the “MicroVisor”, a true *Type I* Hypervisor platform, which compresses all essential HostOS functionality into the HyperVisor microkernel. This is a novel, super-lightweight Hypervisor architecture that is significant for a number of reasons:

1. It allows a smaller software footprint, requiring no control domain
2. Provides optimized network forwarding path between VMs
3. Requires no inter-VM event processing by the control domain
4. Has potential for true zero-copy operations between VMs

5. Does not requires TCP/IP control stack
6. Reduces system attack surface, improving system security

This new architecture has two main benefits for the EUROSERVER platform:

1. True virtualisation support with a Type I Hypervisor that allows isolation and flexible hardware resource sharing to many guest VMs running unmodified operating systems.
2. Resource utilization efficiency. Microservers are servers with limited resources and high performance-to-power ratios, which are built and managed as a high-density cluster of cache-coherent CPU islands. The MicroVisor is lightweight and highly efficient in utilizing system resources and this lowering power usage.
3. Performance and scalability. To meet the performance and scale demands of emerging Network Function Virtualisation (NFV) platforms, a significantly re-architected paravirtual network IO path is required. The Microvisor architecture offers lower network overheads and latencies by removing the dependency on control domain context switching.
4. The UNIMEM hardware interface can be transparently managed by the VMM layer without requiring any guest domain modifications, allowing the MicroVisor to intelligently optimise the mapping of virtual machine memory addresses to physical memory addresses, either local or remote.

The Microvisor platform, depicted in Figure 8, is based on the Open Source Xen Hypervisor but presents significant architectural changes. The core idea behind the MicroVisor is to remove any dependency on a local Dom0 domain for inter-domain paravirtual device communication, but maintain support for unmodified virtualized guest operating systems and kernels with minimized footprint and low overheads. Additional features include the support of VM setup, bootstrap and resource allocation without any control domain dependency.

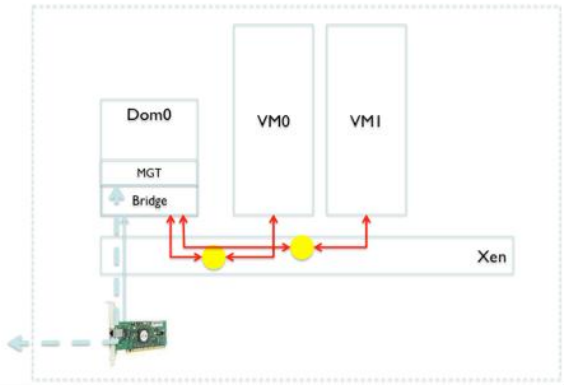
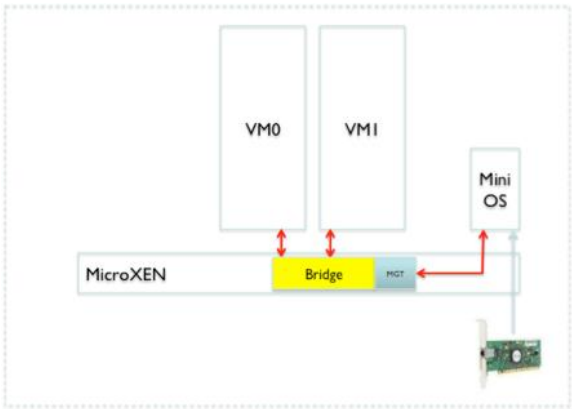
Current Xen Architecture	MicroVisor Architecture
	

Figure 7: Comparison between current Xen architecture and the MicroVisor.

Figure 7 shows guest domUs, the network hardware control and driver management and packet forwarding handling. Note that inter-VM packet forwarding is handled directly in the VMM layer.

Implementation of the MicroVisor HV architecture

For the MicroVisor implementation we have moved this generic functionality into the Xen Hypervisor layer directly whilst maintaining hardware driver support through a super-lightweight driver domain interface, which we implement as a unikernel using mini-OS integration with the hardware driver(s). This results in an extremely efficient, minimal virtualisation shim that provides all the generic inter-

domain virtualisation functions. More details on the driver domains, implemented in the EUROSERVER platform as unikernels using mini-OS integration with the hardware driver(s) and the Linux network stack, are provided in EUROSERVER Deliverable D4.2. Figure 8 compares the MicroVisor to the standard Xen hypervisor architecture.

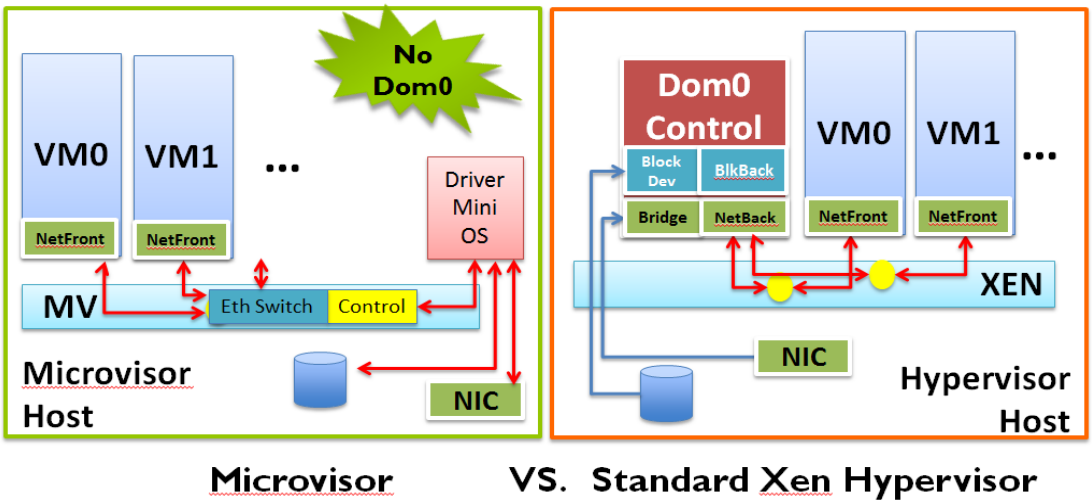


Figure 8: The MicroVisor architecture compared to the standard Xen architecture: MicroVisor has no control domain.

The new hypervisor implementation boots as a standalone image without loading any control domain. Various major changes were required to complete the implementation of the proposed vision. Specifically, the first version of the MicroVisor architecture has been designed and implemented to provide the following:

1. Support for creating and booting multiple guest VMs without control domain in Xen 4.4.0.
2. A minimal set of Xenbus functionality inside the core Xen hypervisor, implemented by migrating the Xenbus functionality from dom0 to the Xen Hypervisor layer.
3. An implementation of a simple key/value store (Xenstore) interface to communicate values to guest domains via Xenstore.
4. The Xen netback functionality (i.e. the Xen network handler in Dom0) was implemented as a fast packet soft switching layer integrated directly in the Xen Hypervisor, including the tx/rx queues for virtual interfaces.
5. PCIback functionality was embedded in the hypervisor layer to support PCI device passthrough for any guest domain (DomU).
6. A simple packet forwarding soft switch is integrated in the Xen Hypervisor layer itself, providing simple and fast packet forwarding functionality between frontend guest netfront instances using a zero-copy mechanism. Packet forwarding off the host is provided via a hardware driver domain, plugged directly into the logical soft switch.
7. Console and debugging support at the hypervisor level.
8. RDMA over Ethernet was implemented between MicroVisor nodes and is used to remotely load guest domain kernels and memory images for booting VMs.
9. Remote booting of guest domains using RDMA over Ethernet to transfer ramdisk and kernels. Our current prototype provides basic flow control, which will be improved in future releases.
10. VM Block storage is provided by migration of the corresponding dom0 block I/O layer handler (blkback) to the Hypervisor layer, providing support for hot-pluggable domU block devices. The blkback component in the HV layer translates block I/O requests to ATA-over-Ethernet requests that are served from ATAoE storage targets (server-side) that can reside anywhere on the local Ethernet.

11. A raw Ethernet packet command interface implemented for the software switch virtual MAC address supporting the following management operations to local/remote Microvisor nodes:
    a. Start/stop VMs
    b. Hotplugging network VIFs to running VMs
    c. Hotplugging PCI devices to running VMs
    d. Management of Xenstore entries for configuration and status information
    e. lspci type output for detection of hardware devices
    f. Hotplugging of virtual block devices to VMs, connecting via ATAoE to target block servers
    g. vcpuinfo for VMs
    h. Getting and setting the CPU affinity for vcpus

The Block I/O architecture of the MicroVisor is depicted in Figure 9, showing the I/O path from the VM to the actual storage device through the unikernel driver domain and ATA-over-Ethernet.

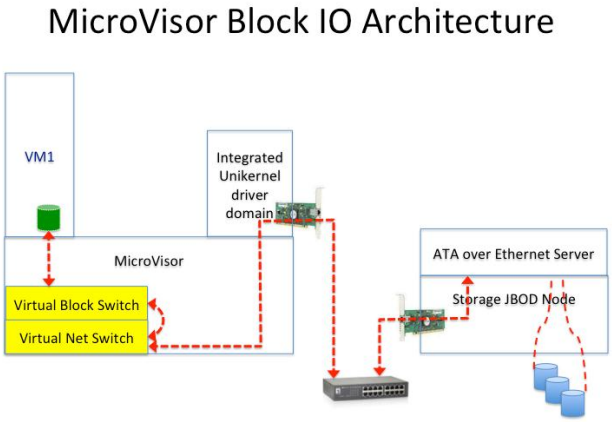


Figure 9: The MicroVisor block I/O architecture.

Based upon the raw Ethernet command interface, we are implementing a centralized management tool (or service) to monitor and manage virtualized resources over a cluster of MicroVisors. This tool will provide also a REST API and command-line interface for easier integration into existing management suites. Our goal is to remotely manage a cluster of cooperative MicroVisors that execute directly on bare-metal hardware, as simple commodity raw Ethernet appliances, using this management tool, running anywhere in the cluster and using the raw Ethernet command interface. Secure access is physically enforced using Layer 2 Ethernet mechanisms to ensure that only trusted elements can communicate directly with the MicroVisor, as shown in Figure 10. A multi-node MicroVisor runs one hypervisor per cache-coherent chiplet, but is managed by a single control tool running anywhere on the cluster. The control tool controls all VMs on the cluster via Ethernet commands to the MicroVisors.

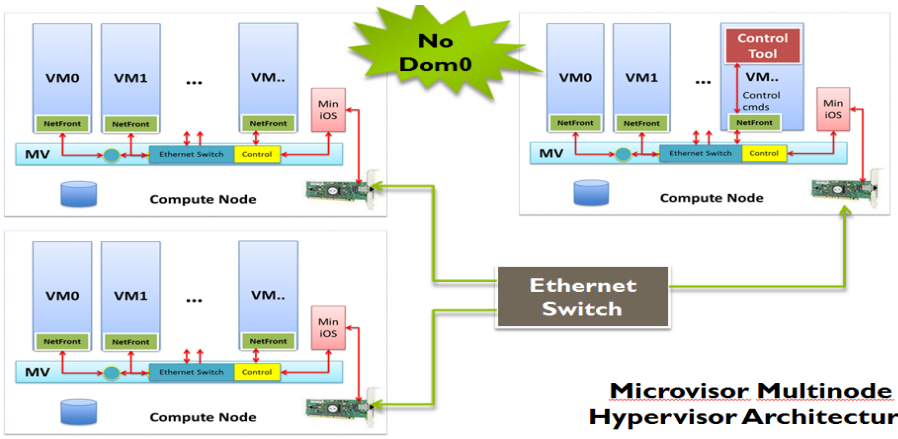


Figure 10: Multi-node MicroVisor architecture, running one hypervisor per cache-coherent chiplet.

There is currently ongoing work in the following areas:

1. Expanding the functionality and optimizing the performance of the network soft switch in the hypervisor layer.
2. Porting the Xen modifications to Xen on ARM.

Memory sharing between MicroVisors

One architectural feature of the MicroVisor is for the ability to share memory resources between the MicroVisors and therefore allow remote memory sharing. This involves a combination of allocating and communicating the available memory regions as well as trapping when this memory is accessed and during the transfers.

There are a set of components required to get memory sharing to work of which we have two currently implemented.

Currently implemented

1. Memory region allocation/freeing/registering
2. One-sided transfer mechanism (RDMA read/write)

Not currently implemented

3. Trap mechanism to trigger the relevant operation (read/write)

In the MicroVisor we currently have implemented an API that allows the local exported and remote memory regions to be gathered from the controller interface, virtxd. In this way all local and remote memory regions can be exposed for each individual MicroVisor node. We have currently implemented the memory allocation, registering/de-registering, freeing, identification and addressing for any memory space within a single MicroVisor. (This corresponds to 1) above)
When a MicroVisor announces the (memory) resources available it includes a listing of active exports and remote maps and is the authoratitive entity for the current memory mappings for the resources it controls.

In addition there is an RDMA over Ethernet implementation where we can read/write data to/from a specific MicroVisor given a virtual address that is relevant only for a specific MicroVisor. (This corresponds to 2) above).

To get memory sharing working fully it will be necessary for a controller to create the memory allocations and then create and control the mappings. The controller therefore needs to be able to allocate memory per individual MicroVisor and to record the local ID of the local-allocated memory region. The controller will then need to register this region on the remote MicroVisor that will also then have a unique mapping for that MicroVisor. Once the regions are mapped there will need to be a

synchronisation and access control mechanism that determines how MicroVisors can read/write to the shared memory regions. In the first instance the system will likely look to claim memory resources from remote regions to have access to a larger memory pool and the remote nodes not be able to use those remote-allocated regions. Further development will allow better utilisation of local and remote memories via transcendent memory allocation techniques and will be investigated and reported in later Deliverables, in particular D4.8 «Performance of in-memory data center workloads on the EUROSERVER platform».

Particular benefits of this architecture are that the memory mapping is done in a dynamic manner with the synchronisation made at the time of the request. This avoids the need for a-priori memory mappings that might be required in a hardware based approach. The architecture also lends itself towards graceful degradation of resources and the ability to increase or decrease the shared memory, resource pool. Particular attention will need to be made for the latency and throughput of transfers between different MicroVisor nodes that may be on the same or different board, system or rack. It will be important for the topology of the MicroVisor network including the resources be determined and kept up to date.

A truncated code snippet that shows the API operation of exposing local and remote memory resources for a single MicroVisor is included in the Appendix (Section «Remote memory sharing in the MicroVisor code listing»).

MicroVisor on heterogeneous hardware architectures

The MicroVisor architecture has been designed with microservers as a key target implementation platform. To proceed with the software development work, in advance of the final EUROSERVER platform being ready, initial platform development for the MicroVisor platform was carried out on x86 systems. This has allowed for the implementation of the platform on existing hardware and allowed for experimentation and validation of the approach in advance of utilising ARM based hardware. As has been described in D2.1, there are efforts by the major CPU vendors to move to low-power, high-efficiency computing platforms. The MicroVisor implementation on x86 therefore will lend itself to evaluation on Intel's Xeon D-1500 and future[1], high-efficiency platforms.

To move towards a single EUROSERVER platform, efforts have also started into getting the MicroVisor platform implemented on ARM 64-bit hardware architecture. An analysis of available and forthcoming 64-bit boards led to us utilising a high-end server board from Gigabyte (see 11) that will be launched to market that has server-level specifications[2], which will be a fairer comparison against the proposed D-1500 Xeon platform boards. Given that the platform is very new, the features may change and the board specifications may vary before the final released product. It has an 8-Core X-Gene1, support for 128GB DDR3 RAM, dual 10GbE, dual 1GbE, IPMI, 4x SATA III that are very similar to available D-1500 hardware[3]. Platform evaluation based on this board is also currently unavailable but will be included in a subsequent deliverable, namely D6.6 «Micro-server evaluation and assessment».

[1] Intel Roadmap and the Xeon D-1500 – accurate as of September 2015 (http://www.intel.co.uk/content/dam/www/public/us/en/documents/roadmaps/public-roadmap-article.pdf)
[2] Specifications subject to change (http://b2b.gigabyte.com/products/product-page.aspx?pid=5422#ov)
[3] Supermicro Intel Xeon D-1500 based platform (https://www.supermicro.com/products/motherboard/Xeon/D/X10SDV-8C-TLN4F.cfm)

Figure 11: Gigabyte MP3-AR0 ARM-64 server board.

The implementation up to now includes;

- MicroVisor ported to Xen 4.6.0-rc and modifications made to support the platform on arm64.
- Driver domain uses unmodified Linux Kernel 4.2.0-rc6+ with passthrough support for the 1GbE xgene1-sgenet NICs, 10GbE xgene1-xgenet NICs and the SATA controllers.
- A RAMDisk has been prepared that uses a minimal busybox image. The MicroVisor platform is fully functional on the ARM boards including off-host management commands with the exception of the advanced volume management storage controller virtual service.

The modifications that are needed to get the platform to work are included in the Appendix (see MicroVisor – modifications required for XGene-1 ARM-64 board to work).

MicroVisor evaluation

We present results comparing the performance of the MicroVisor vs. standard Xen hypervisor architecture, using two Intel servers (two identical machines), where each server has 12 Intel Xeon CPU cores and 16GB RAM. Standard Xen is Xen version 4.4.2, contained in mainline Linux kernel version 3.13. The MicroVisor platform for the measurements is also based on Xen 4.4.

First we present in Figure 12 performance results for booting an increasing number of virtual machines (VMs). The results show that MicroVisor is able to boot the same number of VMs in much less time. This demonstrates the much lower overhead of MicroVisor compared to standard Xen. Please note that the maximum number of VMs we have used in these measurements is limited to 40 due to the RAM size of the servers used (i.e. 16 GB) and is not due to any limits of the software platforms.

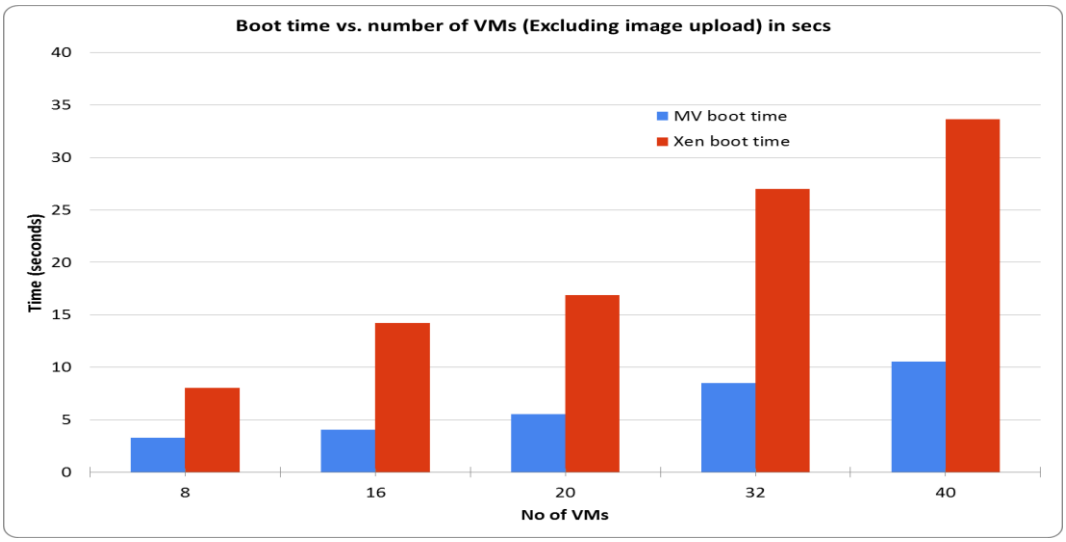


Figure 12: Boot time for varying number of VMs.

Next, we present in Figure 13 a breakdown of the time taken to boot 32 VMs, both for standard Xen and the MicroVisor. It is apparent that the MicroVisor takes much less time for all phases of VM resource allocation, VM spawning, and VM boot. The time to upload the image is shown in the figure, but is excluded from the total boot time. Currently the MicroVisor is using RDMA to upload the image in remote memory and this operation requires further work on the network flow control operations to be efficient. In the next version of our prototype, it is expected that the image upload time for the MicroVisor will be the same as the standard Xen case.

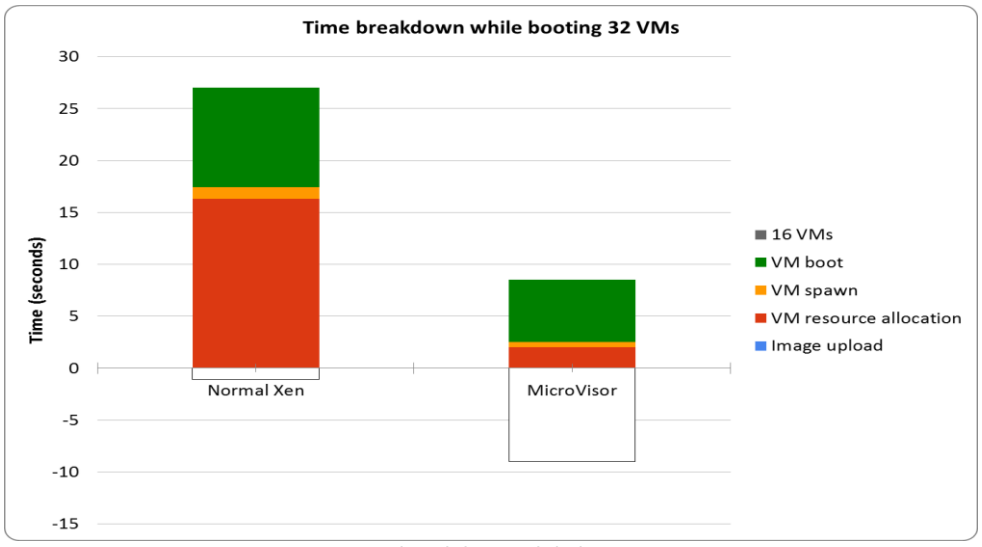


Figure 13: Time breakdown while booting 32 VMs.

Next, we compare MicroVisor to standard Xen in terms of network performance. We use the following methodology:

1. Xen: control domain (Dom0) uses four CPU cores and 2 GB RAM
2. All guest VMs (in both Xen and MicroVisor (MV)) use a single CPU core, 256 MB RAM and all use the same kernel and ramdisk for booting.
3. MV (MicroVisor): our driver domain uses 2 CPU cores and 512 MB RAM
4. The benchmark we use for TCP throughput and latency measurements is NPtcp (netpipe).
5. The reported “netpipe” results are averaged for 2000 messages for each message size.

Figures 14 and 15 summarize latency and throughput results, respectively, from "netpipe" experiments where pairs of VMs communicate over TCP sockets. TCP latency with MicroVisor is 30% of the standard Xen, for up to eight pairs of VMs. MicroVisor outperforms standard Xen in throughput as well (10-25% higher).

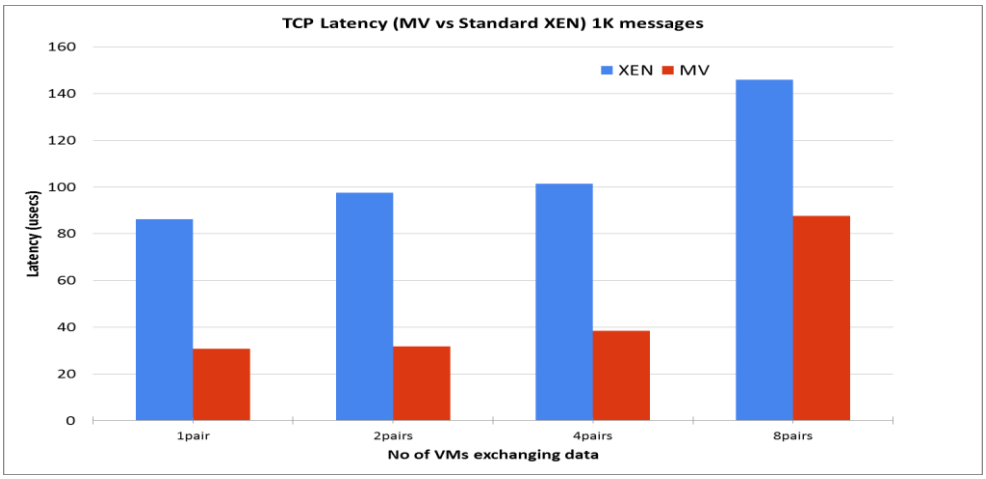


Figure 14: TCP Latency (1 K messages).

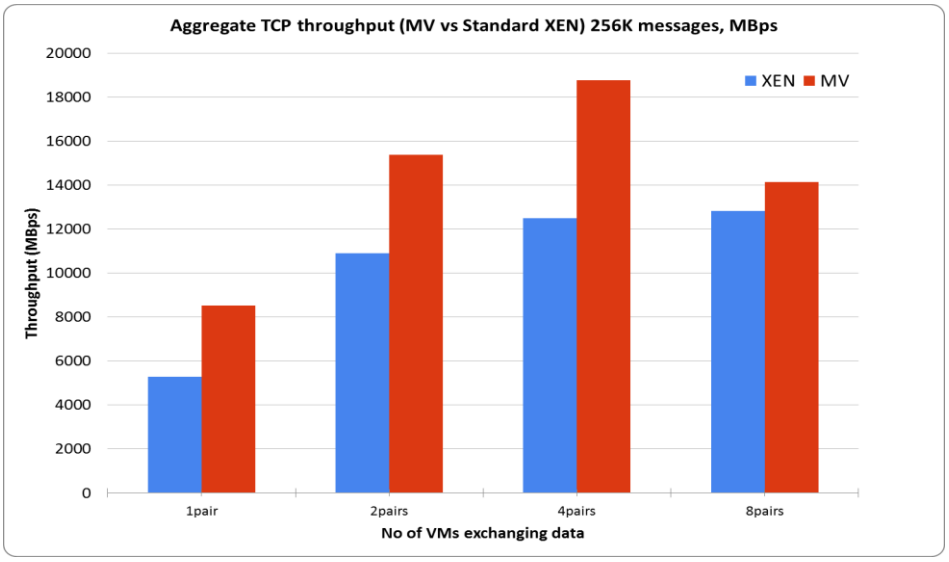


Figure 15: Aggregate TCP Throughput (256K messages).

In Figure 16 we show the aggregate throughput between nodes that have 4x 1GbE links between them when using an MTU of 9000. What is important to show here is that we are close to the physical line rate through efficient encapsulation. The link utilization gets closer to the maximum possible as we increase the number of parallel threads. Future work will compare this result against hardware assisted link bonding on the network switch v.s. this implementation. This will be included in D6.6 "Micro-server evaluation and assessment".

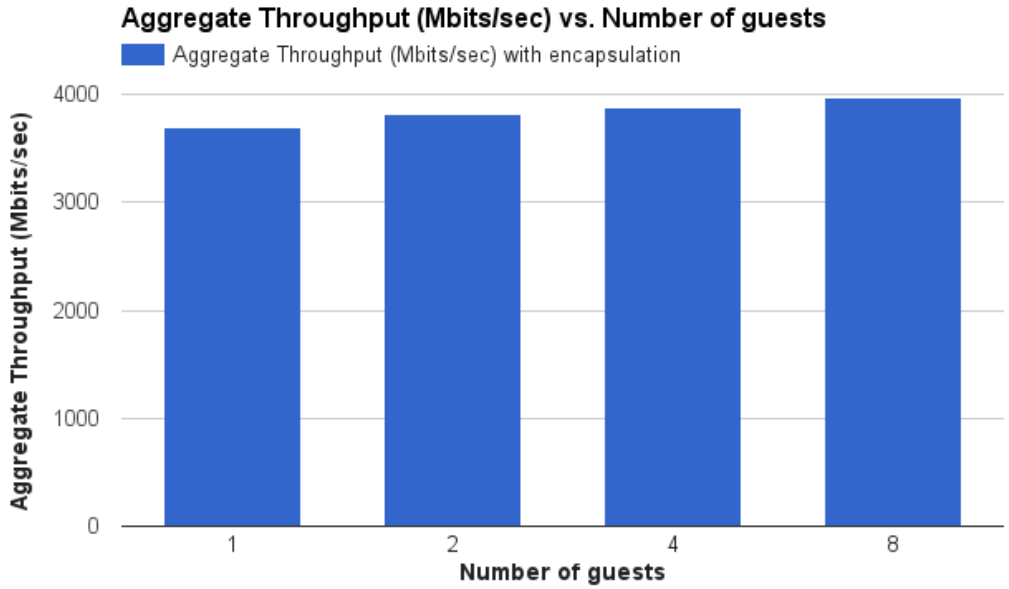


Figure 16: Aggregate throughput v.s. number of guests.

The inter-node latency between guests is shown for guests running on different platforms in Figure 17. The results are only valid for message sizes below the MTU of 1500. Values beyond 1500 message size are not important when measuring the packet forwarding latency as we are not considering fragmentation issues that are handled differently by different parts of the platform. The main thing to consider though is that the latency is much less than that of a stock Xen-4.4 Hypervisor and is very close to bare-metal performance. For the encapsulated case a small overhead above that of the non-encapsulated case is expected.

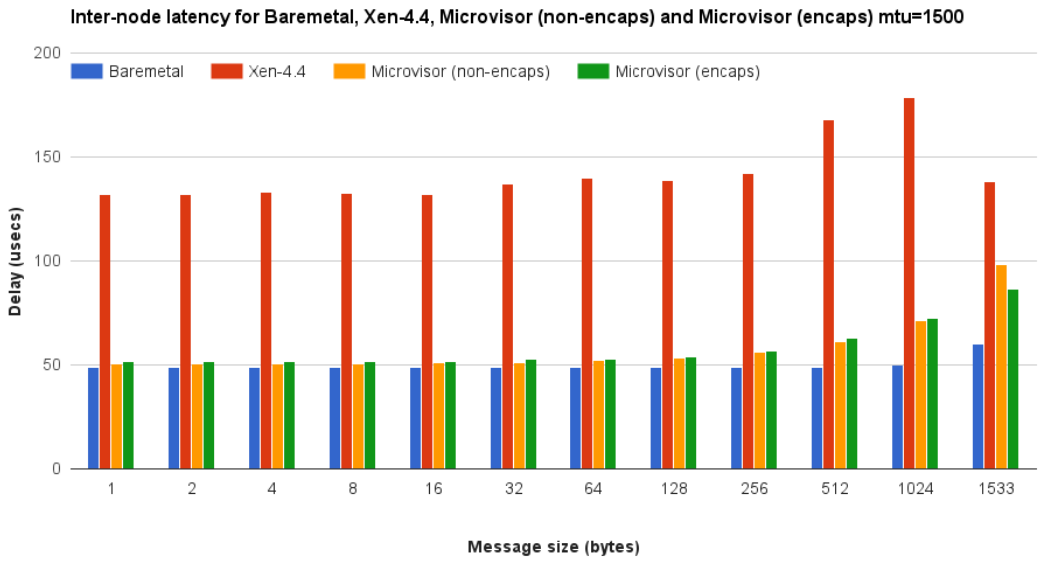


Figure 17: Latency of internode communications for different platforms and message sizes.

In Figure 18 we show the relative latency of the MicroVisor platform in comparison to normalized Xen results. For small (<4K bytes) message sizes the latency is about 1/3 of that of when running on Xen. This holds true as the number of pairings increases. This would suggest that the greatest benefit of the

MicroVisor platform against current platforms is when there are small message sizes. Even at larger message sizes there are still significant improvements over the baseline latency.

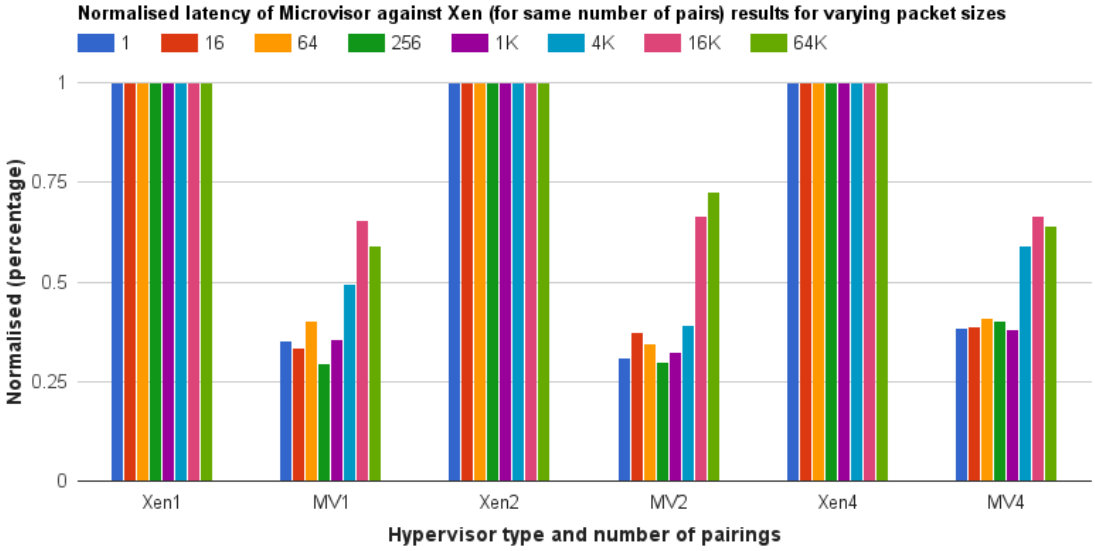


Figure 18: Relative comparison of latency of MicroVisor vs Xen.

Performance Interference Considerations in the Storage I/O Path (FORTH)

Modern servers have switched from SMP to NUMA memory architectures, for scaling the number of cores as well as the capacity and performance of memory. NUMA architectures consist of nodes, each node having its own local memory controller and high speed interconnect to other remote nodes. By increasing the NUMA nodes in a system, we can scale memory in terms of capacity as well as aggregate throughput. All these design choices result in different memory latency and bandwidth towards local and remote memory, with remote accesses being significantly slower. Application performance varies over a disturbingly wide range, depending on the affinity of threads to memory pages. Moreover, due to possible contention in memory controllers and queuing delays in the system interconnect, applications may experience further performance degradation.

These effects have been observed and addressed in current-generation servers, and we expect them to be even more pronounced in upcoming server designs, including the microserver environment of EUROSERVER. We provide a summary of our previous work in addressing the impact of NUMA effects.

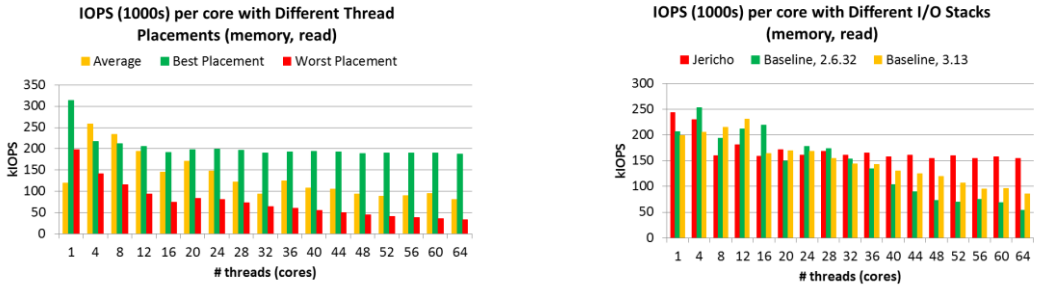
The Jericho [4]prototype features a redesigned I/O stack aiming to achieve optimal affinity of data and processing contexts in many-core systems. Jericho consists of a NUMA-aware filesystem and a replacement for the Linux page cache. This approach is necessary to enforce placement constraints for both tasks and I/O buffers. A related prototype, Vanguard[5], builds on the same infrastructure to address the issue of performance interference between co-located workloads. We show that I/O interference across independent workloads is an important problem in modern servers and can degrade performance many times compared to running workloads in isolation. Running workloads with varying levels of interference in an unregulated system severely impacts application performance, from 52% up to two orders of magnitude. Existing mechanisms in the Linux kernel, such as cgroups, fail to address performance interference effectively, particularly at high load.

[4] Details published in S. Mavridis et al, "Jericho:Achieving Scalability through Optimal Data Placement on Multicore Servers", in proceedings of IEEE MSST, 2014.
[5] Details published in I.Sfakianakis et al, "Vanguard: Increasing Server Efficiency via Workload Isolation in the Storage I/O Path", in Proceedings of ACM SoCC, 2014.

The Jericho Prototype

Applications seriously affected by NUMA effects include scientific applications using MapReduce and similar frameworks, virtual machine workloads, and other I/O-intensive applications, especially when application threads access independent file sets. Focusing on the later, a major cause of performance degradation is the *weak buffer affinity* policies in the Linux page cache. The Linux kernel has been NUMA-aware since version 2.5. System memory is organized into zones, each of them corresponding to a single NUMA node. Accordingly, the page cache became NUMA-aware by using the first-touch policy. When a miss occurs, the newly allocated buffer is allocated from the issuer's local NUMA node. This eliminates remote accesses as long as all later accesses happen from a CPU core in the same NUMA node. The problem with current versions of the Linux kernel is that the task scheduler implements a weak affinity task placement policy. This means that task migrations will not always keep tasks in the same NUMA node as their buffer, breaking affinity.

In Figure 19, part (a) shows results from an I/O-intensive microbenchmark running on top of RAM-disks, as measured on a 64-core testbed (with AMD Opteron 6272 cores), and different thread placement policies. This figure shows the number of IOPS achieved per core, with up to 64 I/O issuing threads. The default, out-of-the-box performance ('average' in the plot) is dropping with increasing core count. The best-case performance, which corresponds to placing the I/O issuing threads so that all their accesses are to local memory, is 2-4x better than the worst-case performance, and moreover remains almost constant with increasing thread count. In part (b) of this figure, we compare the I/O performance (IOPS per core) for two versions of the Linux kernel (2.6.32, 3.13) with the Jericho prototype. With Jericho, I/O performance remains constant with increasing thread count.



(a) (b)

Figure 19: Impact of Thread/Data Affinity: (a) worst-case vs best-case vs default placement, (b) Comparison with the Jericho prototype.

The Jericho prototype features a redesigned I/O stack aiming to achieve optimal affinity of data and processing contexts in many-core systems. Jericho consists of a NUMA-aware filesystem and a replacement for the Linux page cache. This approach is necessary to enforce placement constraints for both tasks and I/O buffers. In order to accomplish our design goals a radical redesign of the I/O stack is necessary, providing alternatives for the layers of the common Linux VFS I/O stack. The existing page cache is bypassed and our own JeriCache is used instead. With the existing page cache, although the first-touch policy results in buffers being allocated locally to the I/O-issuing tasks, later task migrations (after context switches) may break affinity. Our page cache replacement, JeriCache, is organized as a group of independent caches. We use the term 'slice' to describe a single cache instance, limited to using memory from a single NUMA node and caching a specified range of blocks from an underlying

storage device. Mapping storage block ranges to NUMA nodes allows us to implement policies based on the actual data set of each application, instead of tracking I/O requests and I/O buffers in the Linux kernel. Having a custom filesystem, JeriFS, allows us to determine at the time of each I/O request whether the I/O-issuing task is placed at the same NUMA node as the requested data. The storage space managed by this filesystem consists of a set of block ranges from the underlying storage that correspond to the JeriCache slices. This arrangement allows placement of files to use storage from a specific storage block range and, moreover, I/O buffers from a single NUMA node.

With the Jericho prototype, we have identified scalability limitations of the Linux kernel due to NUMA effects, using targeted intensive tests of the common I/O path. We present an evaluation of NUMA effects with our Jericho I/O stack that supports slices. We demonstrate significantly improved scalability, for both I/O throughput-intensive and IOPS-intensive tests. Most of these limitations are not observed at relatively low core counts (8-12), which is the currently common core count for servers, but become severe with more than 24 cores. With 64 cores, our Jericho I/O stack improves sequential read I/O throughput by 2.9x over the baseline system (unmodified Linux kernel) and sequential write I/O throughput by 3.4x. We demonstrate similar improvements for random IOPS performance. Overall, our approach and results will be more useful and relevant with upcoming larger-scale NUMA server platforms, with more pronounced non-uniformity in remote memory access times and cross-core synchronization overheads.

The Vanguard prototype

Figure 20 shows the behavior of six workloads under low and high background load (one and five additional applications running concurrently in other VMs, respectively). We plot, for each workload, two bars, one for the low and one for the high background load, for one-hour runs. The metric plotted is the average performance score reported by each application normalized to an undisturbed run. We observe that even with low background load the performance penalty is at least 50%, while with high background load performance drops up to 30x.

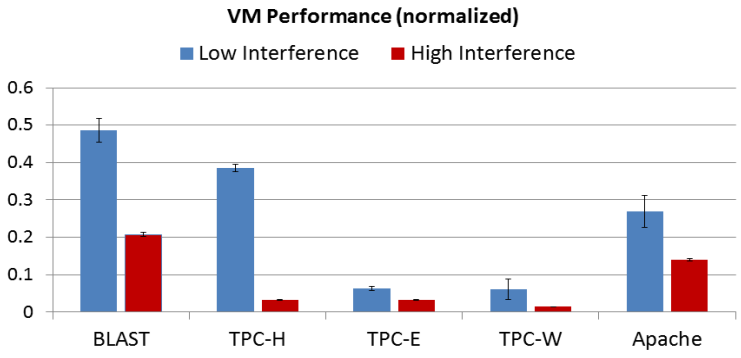


Figure 20: Performance score for the native system with varying background load (Low, High BgL), normalized to the case of no background load.

With the Vanguard prototype, we tackle the issue of performance degradation on a virtualisation server operating at high utilization levels. Our approach is based on isolating hardware resources and dedicating them to competing workloads. We focus on two key resources: in-memory buffers for the

filesystem, and space on SSD devices that serve as a transparent cache for block devices. Our approach effectively mitigates performance interference, for several mixes of transactional, streaming, and analytical processing workloads. We find that with our approach a server can run more workloads close to their nominal performance level as compared to the unmodified Linux I/O path, by careful allocation of I/O path resources to each workload. At excessive load levels, i.e. when the aggregate load exceeds the capabilities of the server, our approach can still provide isolated slices of the I/O path for a subset of the co-located workloads yielding at least 50% of their nominal performance. In addition, Vanguard is shown to be 2.5x more efficient in terms of service resource usage for a 4-workload mix, taking into account utilization and power consumption. With an I/O-heavy mix 6-workload mix, Vanguard is 8x more efficient than the unmodified baseline Linux system.

Figure 21 shows a comparison between the baseline system (unmodified Linux kernel) and the Vanguard prototype, for a workload mix where a 'primary' workload is co-located with a set of 'background/excess' workloads. Every 300 seconds, one additional workload is initiated, adding to the resource demands by the aggregate 'background/excess' load. We plot over time the normalized performance, as compared to the performance of the 'primary' workload running on an uncontended system with the unmodified Linux kernel. Two 'primary' workloads are considered in this experiments: TPC-E (online transaction processing), and TPC-H (online analytical processing). For both cases, the Vanguard prototype effectively isolates the 'primary' workload from the 'background' load, offering consistent, predictable performance.

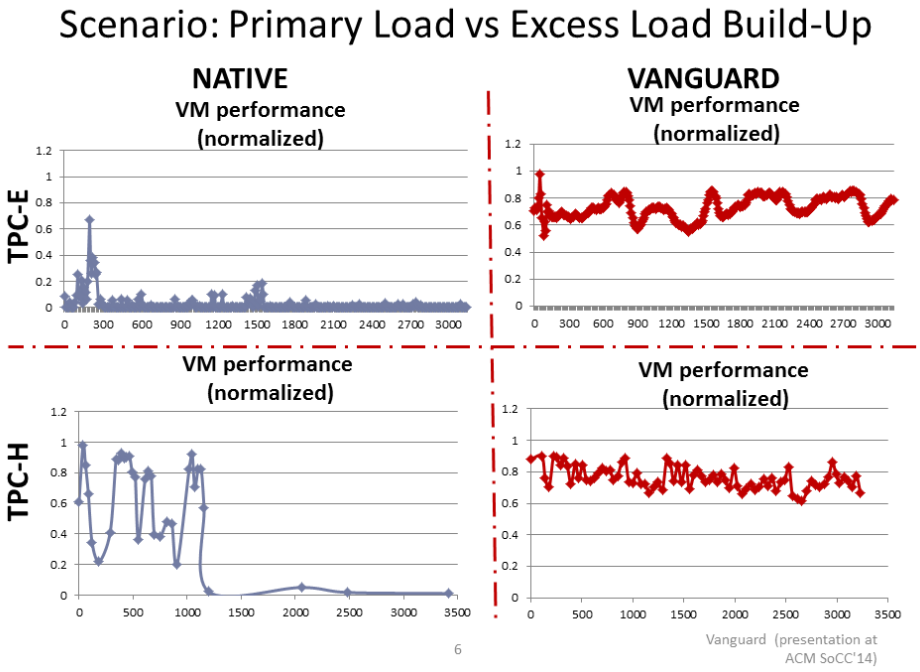


Figure 21: Primary Load vs Excess Load Build-Up.

In summary, with the Jericho and Vanguard prototypes we have demonstrated the effectiveness of workload isolation enforcement mechanisms in the storage I/O path. Currently, we are limited to statically defined assignments of workloads to I/O path slices. We are currently considering dynamic control policies that adjust the parameters of each I/O path slice based on observations of per-workload performance. Dynamic policies will also alleviate the current shortcoming of capping the performance of a workload in a slice even when resources in other slices are available. Overall, we

believe that our I/O partitioning approach is promising, and that future consolidated servers will employ similar techniques to reduce performance variability and improve efficiency.

3. Page-based non-coherent memory capacity sharing (FORTH, BSC)

Many modern Datacenter workloads maintain a large portion of their working set in memory in order to eliminate I/O accesses. These workloads will perform poorly if memory resources are scarce. In microserver environments, each Compute Node has limited physical local memory; therefore, it is essentially that nodes can utilize available remote memory. In EUROSERVER, we want to utilize unused remote memory segments before falling back to slow storage devices. The stipulation made here is that the latency and throughput characteristics of remote memory accesses are far better than that of storage devices.

Page borrowing and capacity sharing (FORTH)

With EUROSERVER *Remote Page Borrowing*, a Compute Node can borrow a memory page from another Compute Node to expand its available physical memory. Since page borrowing occurs between *different Coherence Islands*, which are not connected via a coherent interconnect, we must constrain its operation in order to avoid memory coherency issues for software. There are two ways to implement safe Page Borrowing under the UNIMEM framework, depicted in the Figures 22 and 23:

1. Remote pages can be cached only by the user Compute Node (i.e. the node issuing load/store accesses to these pages).
2. Remote pages can be cached by the owner Compute Node (i.e. the node that is physically closer to the memory hosting these pages).

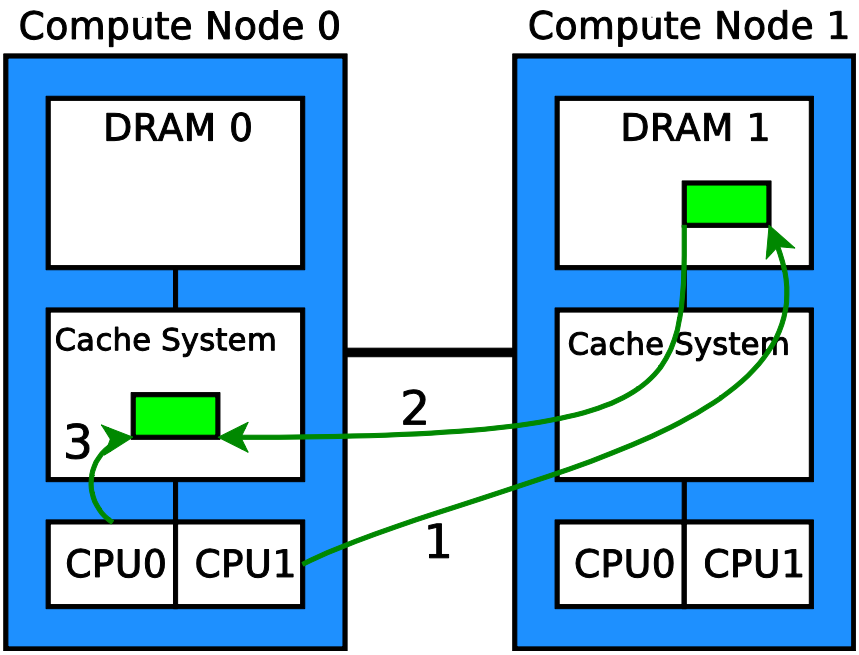


Figure 22: Page borrowing where pages are only cached at the node using them.

The first alternative shown in Figure 22 is preferred when a Compute Node 0 borrows a page from a remote Compute Node 1, at a time when the latter does not use this page. By caching the page only at node 0, the latency and throughput performance will increase, since only the first access will have to go through the interconnect. In the corresponding figure below, (Step 1) Compute Node 0 accesses via the interconnect a remote page residing in the DRAM of Compute Node 1, (Step 2) bringing it to its local cache; subsequent accesses are served by the local cache (Step 3).

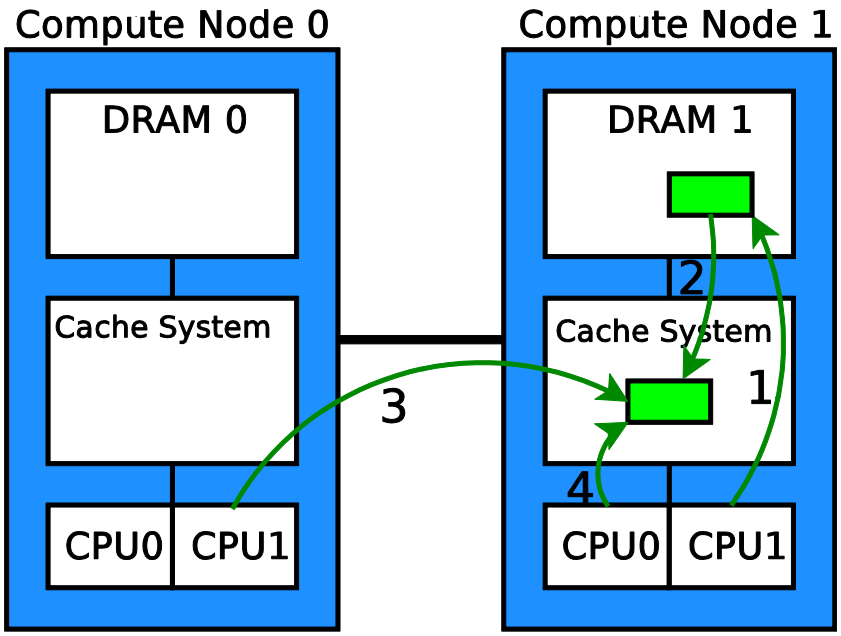


Figure 23: Page borrowing where pages are only cached at the owner node.

The second alternative addresses better situations where a page is shared by two or more Compute Nodes. In this case, it is beneficial to keep the page in the cache of its owner, although this will inevitably increase experience the latency of remote accesses. To achieve this, the user node(s) must mark the page as 'uncacheable'. In the Figure 23, Compute Node 1 (owner) accesses a shared page that resides in its local DRAM (Step 1), bringing it in its cache. Later, Compute Node 0 (user) accesses the shared page through the interconnection network; this remote access passes through the ARM ACP port, which will first search if the page is in the owner's cache. However, the user does not cache the page. Finally, Compute Node 1 accesses again the shared page from its local cache, because the cached copy is not dirty (Step 4).

There is another way to implement sharing of a memory page, using ARM's HP port instead of ACP. Accesses that pass through the HP port bypass the cache coherence system and talk directly to the DRAM controller. With this third alternative, all nodes have to set the shared page as ``uncacheable'', so that all accesses are served directly from DRAM; hence, none will benefit from caching.

In our work so far, we have been using the first alternative, i.e. a remote page is only cached at the node using it.

Remote Memory and the Operating System (FORTH)

In a modern microserver system, both a full Operating System and a user-space environment are needed. Thus, it is essential to explore the ways we can utilize remote physical memory, without compromising security or requiring excessive development efforts by application programmers. In this environment, remote physical memory can be seen in three different ways, as follows:

1. The remote physical memory is seen an extension of the available physical memory, managed by the OS.
2. The remote physical memory is seen a swap device, managed by the OS.
3. The remote physical memory is seen as an I/O character device. With the use of a kernel driver, the remote memory access and management is done by the user space applications or a user space runtime system.

These alternatives differ in the way the OS and the user-space applications see and manage the available remote physical memory. Any software application (OS, user space, or bare metal program) that runs in a processor can access memory in two ways: either with Load/Store instructions or with DMA operations. I/O Read/Write system calls by a user-space application also end up as Load/Store instructions or DMA operations implemented by the low-level kernel driver, which manages these system calls for a specific device.

In our 2-node prototype, each Compute Node runs its own Operating System that is a Linux kernel 3.6.0 as provided by Digilent Inc. We use the higher 256 MBytes of Compute Node 1's DRAM as remote memory for Compute Node 0. The lower 256 MBytes of Compute Node 1 are dedicated to its local processes. Using the available ACP and HP ports, we have several memory mappings available for use by the software, as shown in the Figure 24.

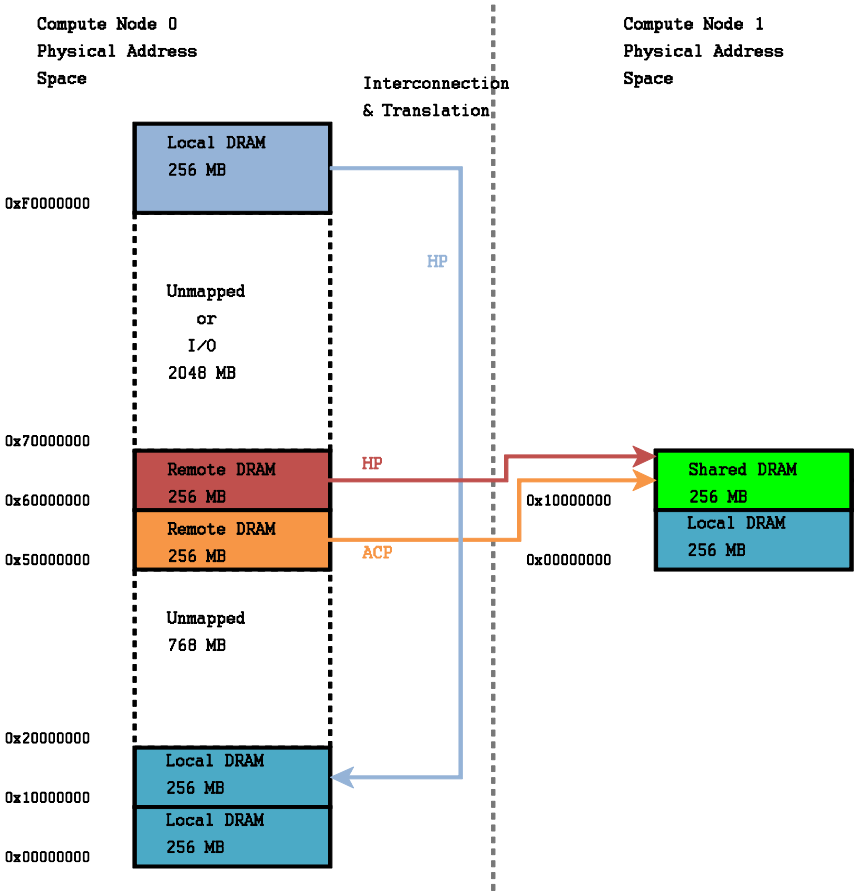


Figure 24: Memory Mappings in a Sparse Memory Model, as it is used in our 2-node prototype.

Remote Memory as Main Memory Extension (FORTH)

We can use a remote physical memory segment as an extension of the local physical memory of a Compute Node, and include it in the memory pool that is available to the Operating System. This means that the kernel structures that describe the physical memory (table *memmap*) will also include the remote physical memory segment. This remote memory will also be fragmented to physical page

frames as well, just like local physical memory does. These physical page frames are available to store virtual pages belonging to user space applications or the kernel itself.

Figure 25 depicts the mapping of virtual pages to physical frames as implemented by page tables. In the left side, the linear virtual address space of a user space application is shown. Available physical memory is shown in the right, along with its valid physical address ranges. Virtual to physical address translation is achieved by the use of multilevel page tables, which in the case of ARMv7 have two levels: Page Directory and Page Table. The address of the Page Directory resides in a register and changes each time a context switch occurs. Using this register, a Page Walk procedure can be initiated in order to find the corresponding physical address of a virtual one, when such a mapping does not exist in the hardware Translation Look-aside Buffer (TLB).

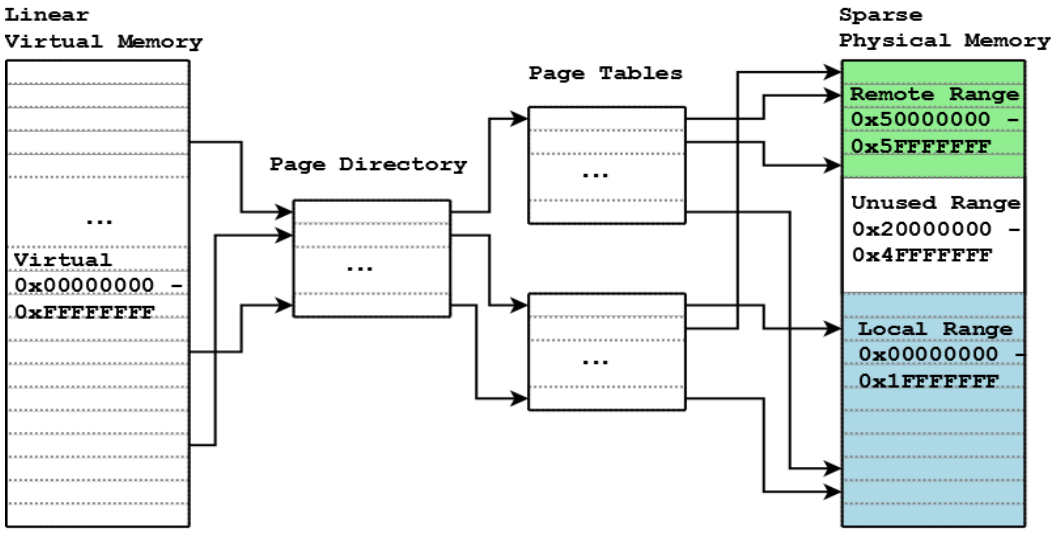


Figure 25: remote memory as main memory extension.

For the OS to see the remote physical address range, we must describe it in the appropriate device tree segment. The Device Tree is a Linux Kernel binary file, which in ARM architectures describes the memory and I/O peripherals of the specific board. The Device Tree file is loaded by the u-boot (boot loader for ARM), and, during OS boot time, it is passed to the Linux kernel. In our case, we had to add a segment in the device tree of Compute Node 0 (shown in the following listing) that describes the physical address range that corresponds to Compute Node 1 physical memory (through the underlying hardware physical address translation).

```
memory@00000000 {
  reg = <0x0 0x20000000>;
  device_type = "memory";
};

memory@50000000 {
 reg = <0x50000000 0x10000000>;
 device_type = "memory";
};
```

Figure 26: Device Tree memory segments. Remote memory access through ACP port.

The listing in Figure 26 shows the device tree segments that describe the available physical memory. The first segment describes 512 MB of local DRAM that start at physical address 0x00000000 with

length 0x20000000 bytes. The latter describes remote physical memory that start at physical address 0x50000000 with length 0x10000000 bytes, resulting in 256 MB of Compute Node 1's DRAM. Note that addresses in this second range are handled by the (coherent) ACP port. We can alternatively map the same remote memory segment using the HP port, by defining the device memory as shown in the listing of Figure 27.

```
memory@00000000 {
  reg = <0x0 0x20000000>;
  device_type = "memory";
};
memory@60000000 {
 reg = <0x60000000 0x10000000>;
 device_type = "memory";
};
```

Figure 27: Device Tree memory segments. Remote memory access through HP port.

Using remote physical memory, borrowed from Compute Node 1, as an extension of physical memory for Compute Node 0, the Linux Operating System that we run at node 0 sees a total of 768 MB of physical memory. We confirm that by running the free utility, as shown in the Figure 28.

```
root@zedboard0:~# free
             total       used       free     shared    buffers     cached
Mem:        775168     129752     645416          0      10556      59600
-/+ buffers/cache:      59596     715572
Swap:            0          0          0
```

Figure 28: Output of free utility on a system with borrowed memory.

We observe that the available physical memory is larger than 512 MB (of local DRAM), and close to 768 MB. Some MBytes are reserved by the OS to maintain memory-resident chunks of its own code and data structures. In ARM architectures, these reserved MBs are almost always located at physical address 0x8000. The OS-reserved memory is not available for user-space applications, and cannot be swapped to a storage device. We can also view the valid physical address ranges by reading the special file /proc/iomem (shown in the Figure 29).

```
root@zedboard0:~# cat /proc/iomem
00000000-1fffffff : System RAM
00008000-00429b8f : Kernel code
00450000-0049f66f : Kernel data
50000000-5fffffff : System RAM
e0001000-e0001ffe : xuartps
e0002000-e0002fff : /axi@0/ps7-usb@e0002000
e0002000-e0002fff : e0002000.ps7-usb
e000a000-e000afff : e000a000.ps7-gpio
e000d000-e000dfff : e000d000.ps7-qspi
e0100000-e0100fff : mmc0
f8000000-f8000fff : xslcr
f8003000-f8003fff : pl330
f8007000-f8007fff : xdevcfg
```

Figure 29: Valid physical memory ranges on a system with borrowed memory.

The output of the special file shows that physical address ranges 0x00000000 to 0x1FFFFFFF and 0x50000000 to 0x5FFFFFFF are labeled as *System RAM*, and are available to applications. Memory ranges 0x00008000 to 0x00429B8F as well as 0x00450000 to 0x0049F66F are reserved by the kernel.

The remaining memory ranges are used for I/O peripheral mappings. The following two listings, in Figures 30 and 31 show respectively the available memory and valid physical memory ranges on the system (Compute Node 1) that owns the borrowed memory.

```
root@zedboard1:~# free
             total       used       free     shared    buffers     cached
Mem:        254828      65760     189068          0       6948      34052
-/+ buffers/cache:      24760     230068
Swap:            0          0          0
```

Figure 30: Free memory on Compute Node 1 (owner of memory borrowed by Compute Node 0).

```
root@zedboard1:~# cat /proc/iomem
00000000-0fffffff : System RAM
00008000-00480323 : Kernel code
004aa000-004fcfcf : Kernel data
e0001000-e0001ffe : xuartps
e0002000-e0002fff : /axi@0/ps7-usb@e0002000
e0002000-e0002fff : e0002000.ps7-usb
e000a000-e000afff : e000a000.ps7-gpio
e000d000-e000dfff : e000d000.ps7-qspi
e0100000-e0100fff : mmc0
f8000000-f8000fff : xslcr
f8003000-f8003fff : pl330
f8007000-f8007fff : xdevcfg
```

Figure 31: Valid physical memory ranges on Compute Node 1 (owner of memory borrowed by Compute Node 0).

Compute Node 1 has only the lower 256 MB of its DRAM available to Its OS. This memory region resides in address range 0x00000000 to 0x0FFFFFFF.

Accessing Remote Memory from User Space (FORTH)

With what has been described so far, user space processes do not have direct access to the remote memory. Ultimately, they do not even know that remote memory exists. They only get access to virtual addresses, which the OS sets up for them. User space processes request for additional memory using the malloc() library function, which, in turn, triggers system calls that request space from the OS. Only the OS can modify the page table of each user space process. Effectively, a user-space virtual page can be mapped to either a local or a remote physical memory, without the process getting to know these details. The main advantage of this approach is that unmodified user-space processes can run on the new platform. Effectively, we can run any application, allowing to access remote physical memory.

To confirm that user-space processes can get more memory than the 512 MB of the local DRAM, we developed a program that uses the malloc() library call to request large memory blocks from the Operating System, sliced in several smaller chunks of varying size. Since the OS implements memory allocation for processes' request in a lazy way (lazy allocation), we must ensure that the process indeed got the memory size it requested. To confirm that, the process accessed all of the requested pages. We know that all accesses were served by physical memory, because we did not have any swap device enabled. Furthermore, as shown in Figure 32, running the *free* utility during the operation of our utility,

we can see that a very large portion of the available (local plus remote) physical memory is in use. Finally, we can also see that no swap device is enabled.

```
             total       used       free     shared    buffers     cached
Mem:        775168     764116      11052          0       4760      19860
-/+ buffers/cache:     739496      35672
Swap:            0          0          0
```

Figure 32: Available physical memory during large memory allocation test.

Specifically, we observe that 764116 KB are in use, out of the 775168 KB that are available in total.

Sparse Memory Model (FORTH)

The ARM processor within the Xilinx Zynq-7000 SoC of our discrete prototype does not allow AXI Requests to be produced on the GP0 and GP1 ports for physical addresses lower than 0x40000000. As a result, we cannot map a remote physical memory segment into a Compute Node's physical address space lower than this limit. Thus, in a Compute Node with remote memory, the available physical memory may contain an unused (invalid) address range, which is not mapped anywhere. This physical memory model is called Sparse Memory. In our discrete prototype, an unmapped physical address range exists between 0x20000000 and 0x4FFFFFFF, creating an unused segment of 768 MB. The next usable address range is 0x50000000 to 0x5FFFFFFF which is mapped to the higher 256 MB of DRAM at Compute Node 1, using the ACP port. Using the HP port, we again map the higher 256 MB of Compute Node 1, but this time to address range 0x60000000 to 0x6FFFFFFF. A large 2 GB segment follows, which contains an unmapped segment as well as I/O mappings for the board's peripherals. Finally, we mapped the higher 256 MB of DRAM of Compute Node 0, so that it can access its own DRAM segment through HP port, instead of the normal way that occurs inside the CPU chip. That segment is mapped at 0xF0000000 to 0xFFFFFFFF.

Remote memory as swap device (FORTH)

Another way of utilizing remote physical memory is as a remote swap device. In this case, remote memory is used only when the local physical memory is exhausted. In this way, the remote physical memory will not be directly available to processes. Instead, remote memory is seen as an I/O device by the OS, and is managed by the OS threads responsible for page swapping, as well as by the appropriate block drivers. Access to the swap device is initiated when a swapper kernel thread decides to store an unused physical memory frame or when a page fault occurs and the OS must bring the stored physical frame back to memory. Figure 33 illustrates the page fault handling procedure in Linux.

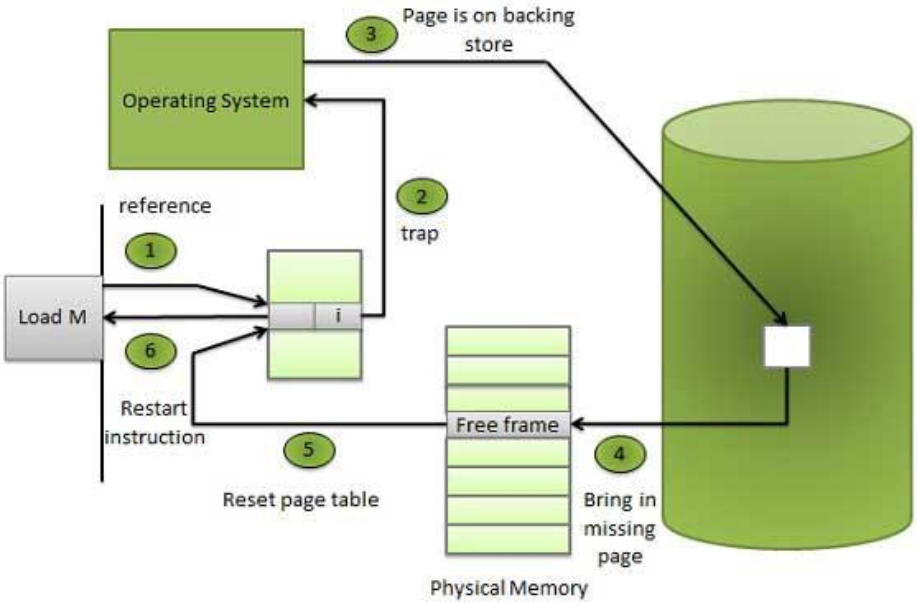


Figure 33: Page fault and physical frame recovery.

The processor issues a Load/Store instruction to a virtual address, which corresponds to a page that does not have a corresponding physical frame present in memory. Effectively, a page fault exception is raised, generating a trap to the Operating System. The latter, finds the corresponding physical frame from the swap device, and brings it to a (free) local physical memory frame. If there is not any free frame in the physical memory, the Operating System, will swap the least-recently-used physical frame, thus causing a write to remote memory.

In order to use physical memory of Compute Node 1 as a remote swap device in Compute Node 0, we developed a kernel driver that creates a ramdisk for the address range corresponding to remote memory. The ramdisk is represented by a block device entry (/dev/krama) in the /dev directory of the Linux. When a read or write is occurred on this device, our driver is invoked to service that request. Note that, besides swapping, the ramdisk device can be used for additional purposes as well. For example, one can use the ramdisk device and our driver to create an EXT4 file system. The driver's main component is an array of struct page structures, which describe the physical page frames the driver is responsible of. Reading from and writing to these physical page frames is done by a function called kram_make_request, which is invoked every time a block read or write operation is performed upon/dev/krama. The function completes a read or write operation by reading data from the appropriate physical address and returning them data to the caller, or by writing its input data to the appropriate physical address.

Since we use physical pages that are not allocated by the kernel, we must ensure that these pages are accessible by the kernel source code macros and helper functions. To achieve this, we must add those remote pages to the kernel's physical memory tables, but reserve them for use only by our swap driver. We modified the kernel source code and added a reservation segment during boot time. Specifically, we added the following code segment in function arm_memblock_init() of arch/arm/mm/init.c in the Linux kernel source tree: memblock_reserve(0x60000000, 0x10000000). Having this physical address range reserved for our remote ramdisk driver by the kernel, we use standard kernel API calls to populate the driver's table of pages (that emulate a block device) with valid physical page frame references.

In order to use the remote memory segment as a swap device, we must first describe it in the device tree. This step is essential, in order to reserve these physical page frames at kernel boot time. After the Linux boots at Compute Node 0, running the free utility will show that the available physical memory is 512 MB, since the remote 256 MB have been reserved at boot time. We then can load the ramdsik driver into the running kernel, and use the mkswap utility to make our ramdisk block device available for use by the Linux swapper. After enabling the swap, running the free utility again shows the available memory for the system, as shown in the Figure 34.

```
             total       used       free     shared    buffers     cached
Mem:        513012     127040     385972          0      10864      57200
-/+ buffers/cache:      58976     454036
Swap:       262140          0     262140
```

Figure 34: Available memory after enabling swap to remote borrowed memory.

We observe that the physical memory is still 512 MB, but this time we have an additional swap space of 256 MB.

We have also implemented the remote swap driver, utilizing the DMA engine for Remote DMA (RDMA) data transfers, instead of Remote Load/Store instructions.

User space processes do not know about the existence of a swap device. When the physical memory (DRAM) is full, the OS starts storing pages from different processes into the swap. Running a test application performing large memory allocations, we can confirm that the (remote) swap is indeed used by the OS, when a process requests memory larger than the available physical DRAM. Running the free utility while this is happening, we can confirm that the swap is used indeed. Figure 35 shows that almost all physical memory is used, and, furthermore, that a large portion of the swap device space is used as well.

```
             total       used       free     shared    buffers     cached
Mem:        513012     501852      11160          0         68       2248
-/+ buffers/cache:     499536      13476
Swap:       262140     136564     125576
```

Figure 35: Swap device (remote ramdisk) in use.

Explicit Access of Remote Memory (FORTH)

It is possible for user-space processes to explicitly access physical memory. The Linux kernel provides a set of special device files inside the /dev directory that can be used for that purpose. In particular, the /dev/mem special file gives access to physical addresses (physical RAM and I/O peripherals). Using that special file, a process can perform read/write operation to physical addresses, using either I/O read/write system calls (just like file operations) or using the mmap() system call to map a physical address range to its virtual memory space. Calling mmap() upon /dev/mem requests from the OS to set up the process's page tables and to add mapping of physical frames to virtual pages of that process. An example of a mmap() call is shown in Figure 36.

```
  memfd = open("/dev/mem", O_RDWR | O_SYNC);
  mem = mmap(0, mapsize, PROT_READ | PROT_WRITE, MAP_SHARED,
memfd,  phys_addr & ~MAP_MASK);
```

Figure 36: Example of call to mmap().

Argument mapsize is the size in Bytes that we request to map into our process address space; this size must always be a multiple of a page. Argument PROT_READ | PROT_WRITE defines the permissions for the mapped pages. The MAP_SHARED argument tags the mapped pages as shareable among processes. We use it in our experiments to tag the mapped page as uncached, in order to measure latency and throughput of the remote physical memory itself. The last argument is the file descriptor of /dev/mem, which is page aligned using the & ~MAP_MASK bitwise operation. The mmap() function returns a pointer to the base address of the first virtual page that corresponds to the requested physical address range. If the mmap() is successful, the process can read or write any byte of the physical memory with Load/Store instructions.

Explicit Remote DMA Operations (FORTH)

Via memory mapping of DMA control registers, it is possible for user-space applications to initiate DMA operations to/from remote memory. A process can initiate a DMA transfer by defining the physical memory source and destination addresses, the size of the transfer, and some control flags. The latter are given as arguments to the DMA engine, by writing their values in the appropriate DMA control registers. After the size register has been written, the DMA engine starts the transfer. The process can wait for the DMA transfer to complete by polling the status register of the DMA engine.

Remote Memory as I/O Device (FORTH)

The remote memory can alternatively be presented to applications as a character (i.e. byte-addressable) I/O device. We have omplemented a kernel driver that manages a character device, /dev/remotemem, which represents the available remote memory. Our driver implements a set of functions that service read/write requests to the /dev/remotemem character device file. It also supports the mmap() function, which enables accessing the remote memory directly from user space. Therefore, a user-space process can access the remote memory through the /dev/remotemem device file, either as a file, using read/write system calls, or directly via Load/Store instructions using the mmap() function. This device driver can provide the basis for a user-space memory allocator (i.e. an implementation of malloc/free calls). Specifically, we have modified jemalloc, an alternative implementation of the standard libc malloc/free library calls.

Considerations for NUMA support in Linux for ARM-based platforms (FORTH)

From the three alternatives that we describe above for accessing remote memory, the first one, memory extension via borrowing, is the most convenient for both the OS and user-space applications, as it requires no driver support. However, with this method the OS and user-space applications do not know which segments are local and which are remote. In this way, data may end up in the remote memory, although local memory space is also available, resulting in performance penalties. Instead, we would like the OS to use remote physical memory only when the local physical memory is getting full, similarly to the conditions when the swapper is activated. This requirement means that the OS should be aware of the underlying non-uniform memory access (NUMA) topology, rather than (simplistically) assuming that all memory regions are equally fast. Unfortunately, Linux kernels for ARM platforms so far do not have this capability (NUMA awareness). We are considering what is required to achieve NUMA awareness in our prototype, having studied NUMA support on mainstream x86 server machines. An essential feature of such systems is the use of distance vectors that provide a latency characterization for different physical memory segments. In this way, the standard Buddy

Algorithm of the Linux kernel, which performs page frame placement (and replacement), has information about the topology of the underlying memory system. Effectively, it can always prefer local memory, and use the remote segments only used when the local memory is getting full. In NUMA machines, the Linux kernel views memory as a collection of (connected) NUMA nodes, each being closer to a specific subset of the available processing cores. The Linux kernel exports APIs for threads (in-kernel as well as user-space) to affect the placement of threads to processing cores and memory buffers to specific 'NUMA nodes'.

Evaluation of mechanisms for sharing remote memory (FORTH)

We implemented some micro-benchmarks both in bare-metal and Linux process forms, to measure latency and throughput achieved when using remote memory. We implemented both forms (bare-metal and Linux process) to confirm that running a full system with Operating System and User Space environment does not negatively impact the performance seen by processes.

Bare-metal evaluation

Running a bare metal application at Compute Node 0, we can access remote memory of Compute Node 1, either with Load/Store instructions or with DMA operations and we can measure latency and throughput. It is important to measure the latency of a simple Read or Write operation for one word (4 Byte). With a basic setup we can measure the Round Trip Time for such an operation. That word must **not be cached anywhere**, in order to measure the latency of the interconnection network itself. A single Load or Store instruction to a remote memory region produces an AXI Read/Write requests and its corresponding response from the remote processor. Thus, a single Load or Store to an uncached remote segment suffers the Round Trip latency of the interconnection network. We created an application that does a large number of Loads/Stores to the same memory address and then calculate the time it takes for a pair of Load/Store instructions to complete by dividing the total time to the number of iterations.

In our 32-bit Discrete Prototype we measured: **1450 ns for a Load-Store pair to complete.** Thus the AXI Request Round Trip Time is about 725 ns. The pair of instructions produces 2 AXI Requests: one Write AXI Request and one Read AXI Request. Although the latency seems quite large, it is measured in the Discrete Prototype in which the interconnection logic is implemented in the FPGAs that run with a slow clock of 100 MHz. Many of the FPGA clock cycles are spent to the axi2axi blocks that handle the LDVS pairs of the FMC-to-FMC cable. When eight LVDS pairs are utilized this subcircuit consumes almost 39 to 40 clock cycles. The remaining 33 to 34 clock cycles are spent in the essential AXI protocol conversion blocks and the processor's memory-to-ACP port and memory-to-GP port subsystems.

The measurement is in accordance with the results seen using a Logic Analyzer (Xilinx ChipScope). With a Logic Analyzer we can see an AXI Read or Write Request and their corresponding responses, determining the FPGA clock cycles it took to complete. (FPGA clock runs at 100 MHz). The Logic Analyzer logic probes (ChipScope FPGA blocks) are put inside the FPGA logic in the Zedboard that is the initiator of the experiment (Compute Node 0). In particular they are inserted at the point the GP ports of ARM processor connects to the interconnection logic. Each time an AXI Request is produced to the GP port by the processor the probe captures it. The logic probes connect to the Logic Analyzer software of a PC via the JTAG port of the boards. We can measure the FPGA clock cycles needed for an AXI Read Request to complete by measuring the time difference between the signals ARVALID (Read Address Valid) and RVALID (Read Valid). In a similar fashion, we check the signals AWVALID (Write Address Valid) and BVALID (Write Valid). The AXI Read Request takes 73 clock cycles to complete, while the AXI Write Request takes 65. As a result a pair of a load and store instructions needs 138 clock cycles

to complete. The 145 clock cycles we obtained from software experiments contain the additional overhead of the GP and remote ACP ports latency that cannot be measured with the Logic Analyzer.

We also measured the throughput achieved for data transfers from local to remote memory and vice versa, with a bare-metal application that uses a loop of Load/Store instructions. We used the *memcpy()* function that is included in the bare metal libraries that come with the Xilinx SDK suite. Using either ACP or HP ports we made some experiments, the results shown in the following table.

Table 3: Data transfer throughput (for bare-metal application).

	Direction	Destination Port	MB/s	Gbps
1	Local Read – Remote Write	ACP	110	0.88
2	Local Read – Remote Write	HP	112	0.90
3	Remote Read – Local Write	ACP	45	0.36
4	Remote Read – Local Write	HP	40	0.32

The *Direction* column shows the data transfer direction. *Destination Port* is the ARM slave port used for that data transfer. The last two columns shows the result for the data transfer in *MB/sec* and *Gbps*. Experiments 1 and 2, read from local physical memory and write to the remote memory - the first using the ACP and the latter the HP port. The last two experiments read from the remote physical memory and write to the local memory - experiment (3) through ACP and (4) through HP port. Note that data transfers that copy data word-by-word or byte-by-byte, do not require the AXI Requests for each word or byte to be completed. Instead, the hardware produces multiple interleaved AXI Requests, since it is allowed by the AXI protocol.

Besides data transfers using Load/Store instructions, we can utilize DMA operations to the mapped remote physical memory segment (RDMA operations). The RDMA is a *Data Mover*, which transfers the content of a physical memory segment to another. Four kinds of data transfers can be performed with RDMA:

1. From local to remote
2. From remote to local
3. From local to local
4. From remote to remote

There is an FPGA block that implements the Xilinx CDMA in each Zedboard (Compute Node). It resides outside of the CPU cores, thus it has to create 2 AXI requests for the transfer of a single word (or Byte). It produces an AXI Read Request for reading the data from the source physical address and an AXI Write Request to write those data to the destination physical address. The DMA engine utilizes several techniques to improve performance, such as multiple interleaved AXI Requests, Read/Write bursts, etc. Using a bare metal application that handles the DMA engine, we measured the throughput achieved for data transfers in various directions and different source and destination port utilizations (ACP or HP). These results are summarized in the following table. Columns *Source* and *Source Port* show the port used for reading data, while columns *Destination* and *Destination Port* mentioned the port used for writing data. The last two columns show the measured throughput in MB/sec and Gbps for each data transfer. We observe that when using the HP Port for reading, the throughput almost doubles. It is reasonable, since use of the HP port bypasses the cache coherency system of the ARM processor.

Table 4: DMA Data Transfer Throughput (for Bare-Metal Application).

Source	Source Port	Destination	Destination Port	MB/s	Gbps

1	Local	ACP	Remote	ACP	310	2.48
2	Local	ACP	Remote	HP	305	2.44
3	Local	HP	Remote	ACP	610	4.48
4	Local	HP	Remote	HP	507	4.06
5	Remote	ACP	Local	ACP	264	2.11
6	Local	ACP	Local	ACP	263	2.10
7	Local	HP	Local	HP	716	5.73

Linux evaluation

We measured the latency for a single read/write of one word (4 bytes) which belongs to a remote memory page and the throughput achieved for data transfers from the local physical memory to the remote and vice versa. The latency for a Load/Store pair of one word of the remote memory is: **1450 ns or 725 ns per read or write operation. The results are the same to the measurements taken with the bare metal applications.** As a result, we observe that we can use remote memory without penalty in a full Linux environment. We measured throughput achieved for data transfers in various directions: from local to remote physical memory, remote to local and local to local. The results are shown in the following table and in Figure 37.

Table 5: Data transfer throughput (for Linux run-time environment).

	Direction	Destination Port	MB/s	Gbps
1	Local Read – Remote Write	ACP	110	0.88
2	Local Read – Remote Write	HP	112	0.90
3	Remote Read – Local Write	ACP	45	0.36
4	Remote Read – Local Write	HP	40	0.32

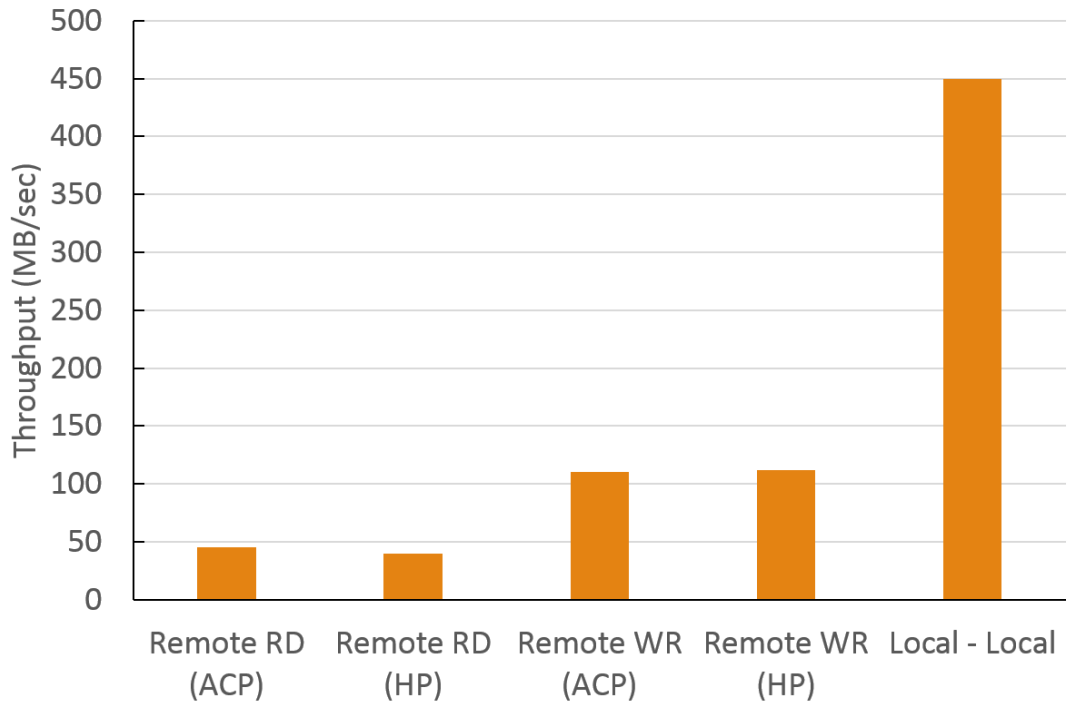


Figure 37: Data transfer throughput using load/store instructions (for Linux run-time environment).

The throughput achieved with load/store instructions is relatively small compared to local-to-local memory data transfers inside the processor. However, the throughput achieved is still far better than data transfers from/to I/O storage devices. Transferring data from remote to local memory gives a maximum throughput of 45 MB/sec. *(If the ARM processor had no read buffers so that no outstanding AXI read requests could exist, then the maximum throughput that could be achieved would be 5.2 MB/sec.)* Writing data from local to remote memory gives a higher throughput of 110 MB/sec.

For completeness of the benchmarks we include an evaluation of the local DRAM and create a machine profile for the Compute Nodes. Since Zedboards contain a DDR3 DRAM memory module clocked at 553 MHz utilizing a 32 bit *bus*, the maximum theoretical data rate the memory can handle is *(Clock Frequency) x (Bus Width) = 533 MHz x 32 bit = 15.89 Gbps or 2033 MB/sec.* This is the maximum raw data rate the DRAM can handle. Of course, the Read or Write bandwidth differs when the Load/Stores refer to sequential or random memory addresses. The data transfer throughput, achieved by an application when copying a buffer to another one, is even smaller because it consists of pairs of load/store instructions per word. The following table shows memory throughput measurements using a variety of buffer copying methods. Methods (1) and (2) use the default library functions *memcpy()* and *bcopy()* and achieve high throughputs of 7.6 and 5.5 Gbps respectively. Methods (3), (4), (5) and (6) use for loops that copy four-byte words using load/store instructions. Method (3) copies one word per loop iteration, while methods (4), (5) and (6) implement unrolled loops that copy four, eight and 32 words per loop iteration, respectively.

Table 6: Local memory throughput for various buffer copying methods (Linux run-time environment).

	Method	MB/sec	Gbps
1	memcpy()	950	7.6
2	bcopy()	688	5.5
3	loop()	250	2
4	Unrolled loop() x4	650	5.2
5	Unrolled loop() x8	763	6.1
6	Unrolled loop() x32	938	7.5

Overall, we observe that performance of remote memory in our Discrete Prototype is comparable to the performance of local DRAM.

The following table summarizes the results from experiments measuring the throughput achievable by initiating DMA transfers from user-space. The first four lines address data transfers from local to remote node, with all port configurations (reading from local DRAM and writing to remote DRAM). Lines 5 to 8 address remote to local data transfers (reading from remote DRAM and writing to local DRAM). Lines 9 to 12 the transfer throughput when the DMA is used on local DRAM only and finally, the last four lines (13 to 16) show throughput for DMA transfers from remote DRAM to remote DRAM again (also called *3rd party DMA*).

Table 7: Data transfer throughput using DMA (Linux run-time environment).

	Source	Source Port	Destination	Destination Port	MB/s	Gbps
1	Local	ACP	Remote	ACP	310	2.48
2	Local	ACP	Remote	HP	305	2.44
3	Local	HP	Remote	ACP	610	4.48
4	Local	HP	Remote	HP	507	4.06
5	Remote	ACP	Local	ACP	264	2.11
6	Remote	ACP	Local	HP	280	2.24
7	Remote	HP	Local	ACP	280	2.24
8	Remote	HP	Local	HP	701	5.60
9	Local	ACP	Local	ACP	250	2.00
10	Local	ACP	Local	HP	241	1.92
11	Local	HP	Local	ACP	715	5.72
12	Local	HP	Local	HP	716	5.73
13	Remote	ACP	Remote	ACP	213	1.70
14	Remote	ACP	Remote	HP	230	1.84
15	Remote	HP	Remote	ACP	605	4.84
16	Remote	HP	Remote	HP	501	4.00

These results are plotted in a graph form as well in Figure 38, where the horizontal axis describes the data transfer direction and port configuration and the vertical axis shows the DMA throughput achieved. We observe that DMA transfers are faster than simple load/store operations, for transfer sizes above a few hundred bytes.

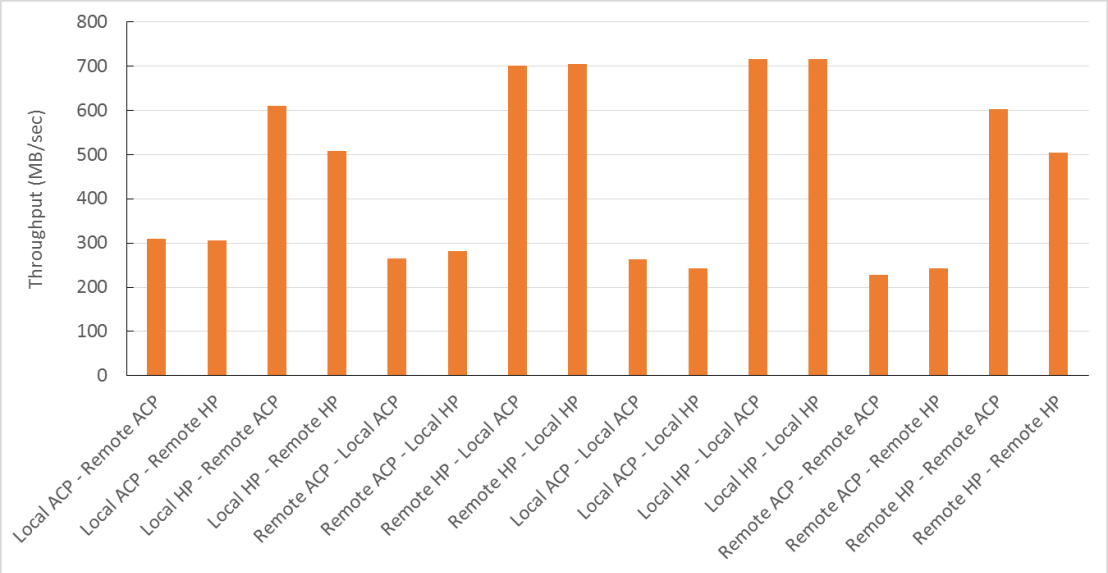


Figure 38: Data transfer throughput using DMA (Linux run-time environment).

We have confirmed that having a full environment with Operating System and user-space applications does not have a negative impact on performance of DMA. We also, observe that in general, when reading through the ACP port (regardless if the source is the local or remote DRAM) the performance is decreased compared to transfers that read through the HP port. The Discrete Prototype and the CDMA hardware IP block give the software engineer and the systems designer many options to choose

from. Depending on the memory allocation policy (which node uses what part of DRAM) and the type of memory coherency needed, one can use the appropriate configuration to maximize performance.

Figure 39 illustrates the impact of transfer size on throughput for transfers from local DRAM (HP port) to remote DRAM (via the ACP port). As expected, there is little benefit to be gained when using DMA operations for small transfer sizes. The performance is decreased due to DMA setup time and the fact that one cannot fully utilize the DMA features, such as large bursts, etc. As a result, the throughput of small data DMA transfers is comparable to throughput achieved with Load/Store data transfers for the same size, so one can use Load/Stores instead. The critical transfer size is 512 bytes, which is the first transfer size that DMA operations start to gain greater throughput than load/stores. As seen in Figure 39, DMA data transfer throughput for 512 bytes is 122 MB/sec. The load/store upper throughput limit when writing to remote memory via HP or ACP ports is 110 MB/sec as seen in Figure 38. Therefore, for smaller data sizes it is beneficial to use load/store instructions instead of DMA operations.

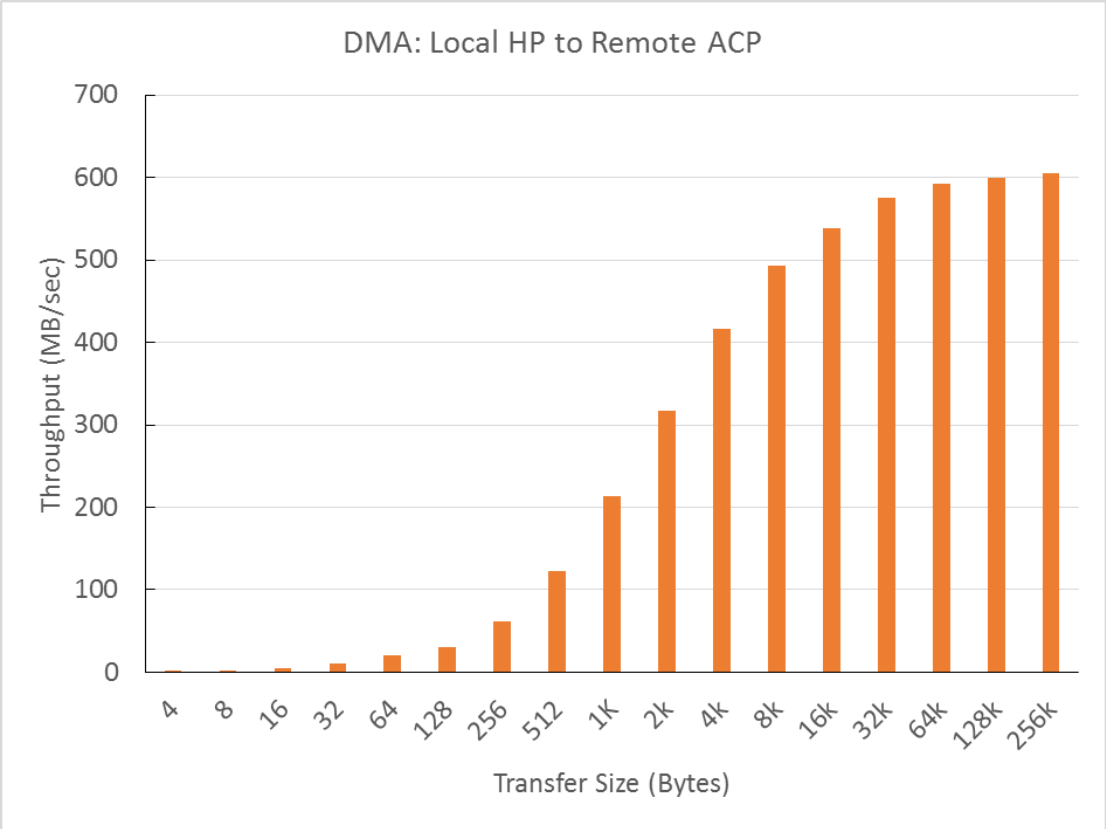


Figure 39: DMA data transfer throughput (local HP port to remote ACP port, Linux run-time environment).

Figure 40 shows the impact of transfer size for DMA transfers in the opposite direction, i.e. from remote DRAM (through the ACP port) to local DRAM (through the HP port). The upper limit for DMA throughput using this direction and port configuration is 280 MB/s, is reached only for transfers with large data sizes. For small data sizes, 4 up to 128 Bytes, data transfers using Load/Store instructions give better results in terms of throughput, because as shown earlier Load/Store data transfers in this configuration can achieve 45 MB/s.

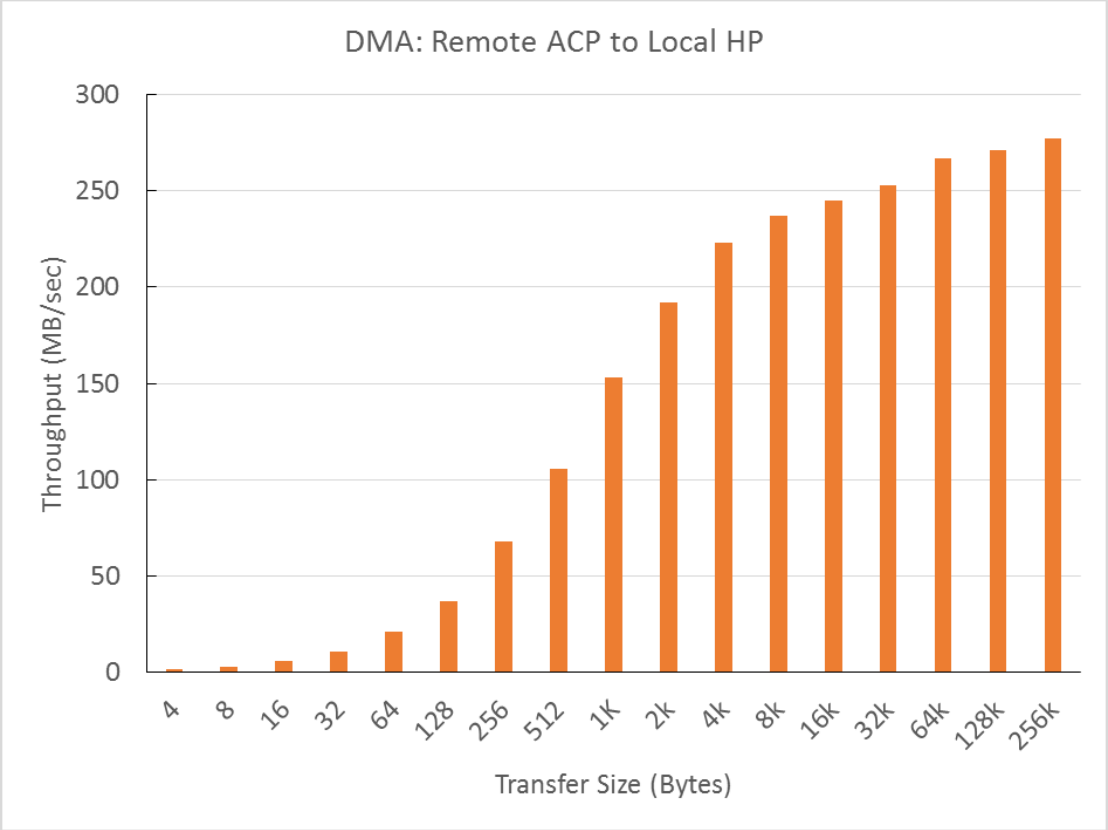


Figure 40: DMA data transfer throughput (remote ACP port to local HP port, Linux run-time environment).

Finally, we present results from an evaluation of remote swap. Viewing remote memory as a remote swap device requires the block I/O stack of the Linux kernel to intervene for each *read()* or *write()* from/to that device. First of all, we measured the I/O throughput the swap device can achieve using the standard *dd* utility.

We ran *dd* for different block sizes and the results are show in Figures 41 and 42. In those two figures, the Gray line represents throughput achieved when using remote swap over Ethernet with the *Network Block Device (nbd)*. The orange line shows throughput for a swap partition that resides in the SD card that contains the Linux kernel and the Ubuntu 12.04 root file system.

We implemented remote swap over Ethernet using the *nbd* in the following way. Using the native 1 Gbps Ethernet interface of the Zedboard platform, we connected the node with a PC with a Linux OS *back-to-back*, using a *cross* Ethernet cable. The swap partition on the PC resides in main memory (*ramdisk*), in order not to suffer from hard disk throughput limitations. Using the *nbd* driver we created a block device in the Compute Node's Linux. The client driver is invoked when *read()/write()* occur at that device and sends those requests over the Ethernet to the PC, where the server side of the *nbd* accesses the swap ramdisk and services the requests.

Our implementation achieves double the data transfer throughput for Write requests for all block sizes. For Read throughput, block size has a huge impact on the *nbd* throughput, resulting in a better performance than our implementation for large block sizes. However, at the most important block size, that is 4KB - the size of a page, our implementation achieves three times greater throughput.

In general, the TCP throughput achieved using the native 1 Gbps Ethernet interface on our testbed's boards is limited to 450 Mbps Half-Duplex, as we measured in similar experiments using the *iperf*

utility. Using a swap partition in the SD card, we measured Write data transfer throughput at 50 Mbps and Read throughput from 50 to 100 Mbps depending on the block size. We observe, that the swap I/O throughput is limited by the SD card device I/O throughput that is much smaller than throughput achieved in our implementation.

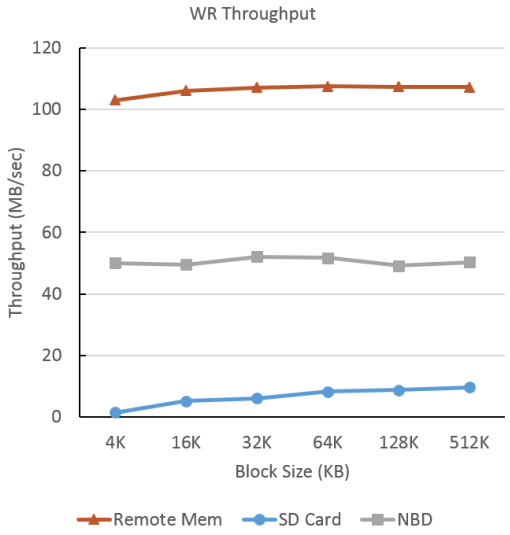


Figure 41: Remote swap evaluation: write throughput.

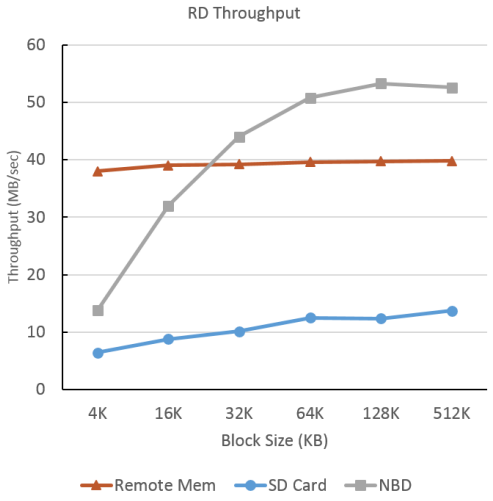


Figure 42: Remote swap evaluation: read throughput.

Memory capacity sharing across coherence islands (BSC)

This task focuses on hypervisor support for page-based memory capacity sharing, based on the hardware UNIMEM support. In addition to implementing the necessary hypervisor support on the final prototype, the main research direction addresses how to dynamically assign the global shared memory capacity among multiple hypervisors.

The initial efforts at BSC, from M1 to M18, were focused on building a viable development platform using the 32-bit discrete prototype (DP). An initial evaluation of OnApp's port of Xen to the 32-bit DP showed that it was too unstable for use as a development platform. We therefore ported Xvisor, a hypervisor with full support for ARM Cortex-A9, to the AVNET MicroZed. This work was successful, in that this Xvisor port was able to run multiple guests on a single MicroZed board, without encountering stability issues. Unfortunately, however, this Xvisor port did not work on the 32-bit DP. When one or more MicroZed boards were connected to the HiTech Global board and the Zynq FPGA configured to use the appropriate bitstream, Xvisor could still be booted, but attempts to boot a guest OS failed with a mysterious external data abort.

In light of the M18 Review in May 2015, during preparations for the Second Interim Review in June 2015, the consortium decided to consolidate the various development platforms used in WP4. In addition, the ongoing development of a 64-bit DP means that software development to work around the limitations of the 32-bit DP will soon become obsolete. BSC therefore decided to align its ongoing work with the OnApp MicroVisor platform, using RDMA support to emulate the UNIMEM hardware – an approach which will also allow the work to be more easily integrated into the single project demonstrator. This decision, unfortunately, means that we had to abandon our work on virtualisation on the 32-bit discrete prototype. Further details of this initial work are given in the Appendix.

Background: Transcendent Memory

Transcendent memory (Tmem) is an approach to improve physical memory utilization in a multi-guest virtualized environment [21]. Underutilized ("fallow") memory, such as OS disk caches, is collected into one or more pools, and managed by the hypervisor, with accesses only through an explicit paravirtualized interface. The guests perform read ("get"), write ("put") and delete ("flush") operations using hypercalls, at page granularity. Pages are either *persistent* (a later get of the same page must succeed), or *ephemeral* (the hypervisor may reuse the memory, in which case a subsequent get operation would fail). Pages are also either *private* to a single guest or *shared* between multiple guests.

The Linux kernel employs Tmem using two abstractions. The first, *cleancache*, acts as a victim cache for clean pages, for example for clean file-backed pages. When such a page is evicted from physical memory, it is put to Tmem. When its contents are later required, cleancache attempts to get the page from Tmem. If the get operation fails, then the kernel will re-read the data from disk, as it would have without Tmem. The second abstraction, *frontswap*, acts like a fast swap device, albeit of unknown and variable size. When a dirty page is evicted from physical memory, frontswap first tries to put its contents as a persistent page to Tmem. If this put fails, then the page is written as normal to the swap device. If the put succeeds, however, the only up-to-date copy of the page will be the one stored in Tmem, so the later get of the same page must be successful.

Figure 43 illustrates how transcendental memory is implemented in Linux and Xen. On initialization, the tmem kernel module (`drivers/xen/tmem.c`) registers itself as a back-end driver for both cleancache and frontswap. Cleancache and frontswap operations are passed to the Tmem kernel

module and translated into the appropriate hypercalls, of which the most important are TMEM_PUT_PAGE and TMEM_GET_PAGE. This design is therefore a synchronous interface with one hypercall per page put or get.

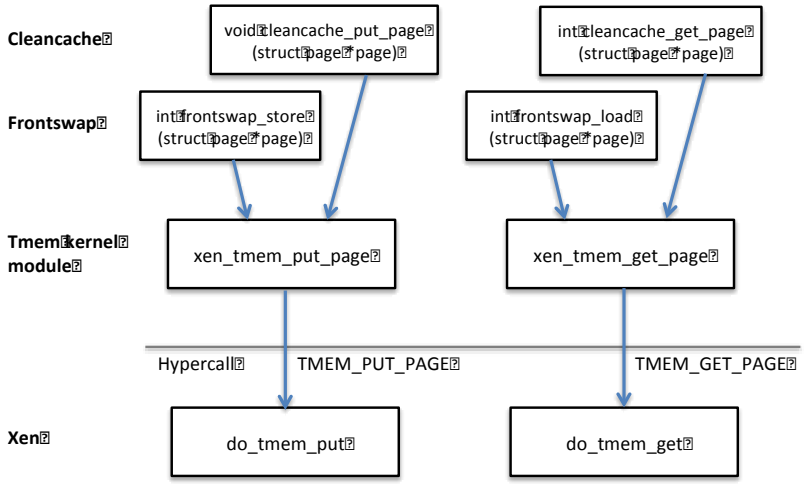


Figure 43: Transcendent Memory implementation

Tmem prototype using remote memory emulation

We are prototyping the hypervisor's memory capacity sharing using Tmem in Xen. As mentioned above, we initially planned for the work in this task to be done using hardware UNIMEM support on the 32-bit DP. The intention was to grant remote memory directly to guest domains using either the balloon driver or memory hotplug. With the new plan to use software emulation, guest domains will access remote memory only indirectly through Tmem. This enables functional evaluation, but not performance evaluation, since the software emulation approach will significantly increase execution time overheads. We will be able to perform performance evaluation once MicroVisor has been ported to the 64-bit DP.

The plan is as follows:

1. *(At BSC)* Emulate remote memory on x86 using a user-space program in Dom0 and socket communication.
2. Emulate remote memory on x86 using MicroVisor and remote RDMA emulation. This will be joint work of BSC and OnApp.
3. Mechanisms to dynamically reassign memory among nodes. This is the focus of the research activities at BSC. Functional evaluation will be done using steps 1 and/or 2 above.
4. When MicroVisor has been ported to 64-bit ARM, allocate remote pages directly using hardware UNIMEM support. This will allow us to perform performance evaluation (still using Tmem).

Part 1: Remote memory emulation using user-space program in Dom0

Figure 44 shows a high-level view of the software remote memory emulation approach. The diagram shows one node with one (or more) DomU guests ("clients") using Tmem, as well as a single Dom0 ("server") that assists in remote memory emulation. A user program in Dom0 (the *remote memory server*) communicates with its counterparts in other nodes.

When a Tmem put operation is allocated to remote memory, Xen generates a virtual interrupt to Dom0. Dom0 performs a hypercall to obtain the global remote memory address and page contents, and it passes this information to the user-space remote memory server using procfs. This user program stores the page within its own address space. The user program may additionally send the page to the appropriate remote node using socket communication (we have prototyped this but not yet integrated it with the rest of the infrastructure). The sequence of steps is similar when Xen performs a remote Tmem get operation. The get request is sent to the user program in Dom0, which may have to obtain the page from its counterpart in another node. The page contents are eventually passed to the Tmem kernel module using procfs, and to Xen using a hypercall. The data is finally returned to DomU using a second virtual interrupt and hypercall.

A remote Tmem get is therefore an asynchronous operation, meaning that the original Tmem get hypercall performed by DomU should return before the requested page is available. The get operation will also be asynchronous when using remote RDMA on MicroVisor. We therefore modified frontswap to use an asynchronous interface to Tmem. We will modify cleancache similarly, but have not done so yet.

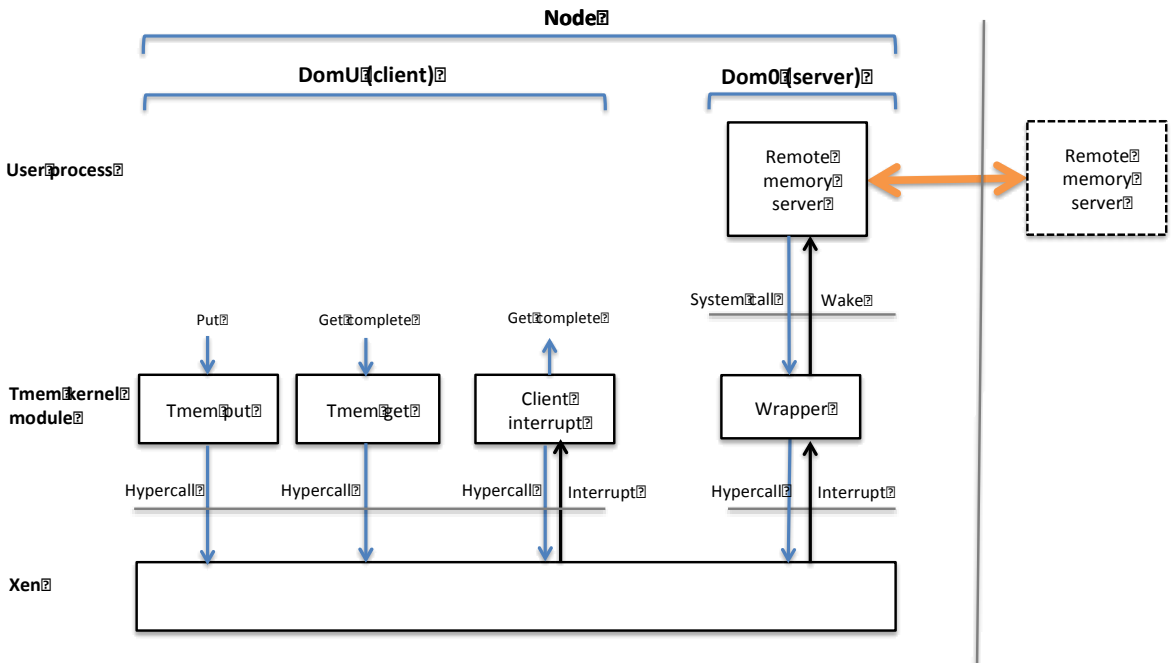


Figure 44: Remote memory emulation using user process

We have implemented step 1 (above) using Linux 3.19 and Xen 4.5, running on a laptop using nested virtualisation inside Oracle VirtualBox. We configured DomU to run with 512 MB and a 1 GB swap device, and we configured Dom0 to use 2 GB in order to be able to store the "remote" pages.

Our current status is that the step 1 implementation described above is functional and stable, with the following limitations:

- Only frontswap has been modified to be compatible with asynchronous gets. It is still necessary to modify cleancache.
- Socket communication among multiple Dom0s has been prototyped but it has not yet been integrated with the Tmem remote emulation infrastructure.

Figure 45 shows a screenshot of an SSH connection to Dom0 inside VirtualBox on a 2013 Apple MacBook Pro (13-inch). After a few minutes of operation, there have been 326,745 remote puts, and 266,643 remote gets have been initiated, of which 266,635 have been serviced (the remainder are pending due to the asynchronous interface). The average latency on a get operation, as measured at the DomU end is 1,158 microseconds.

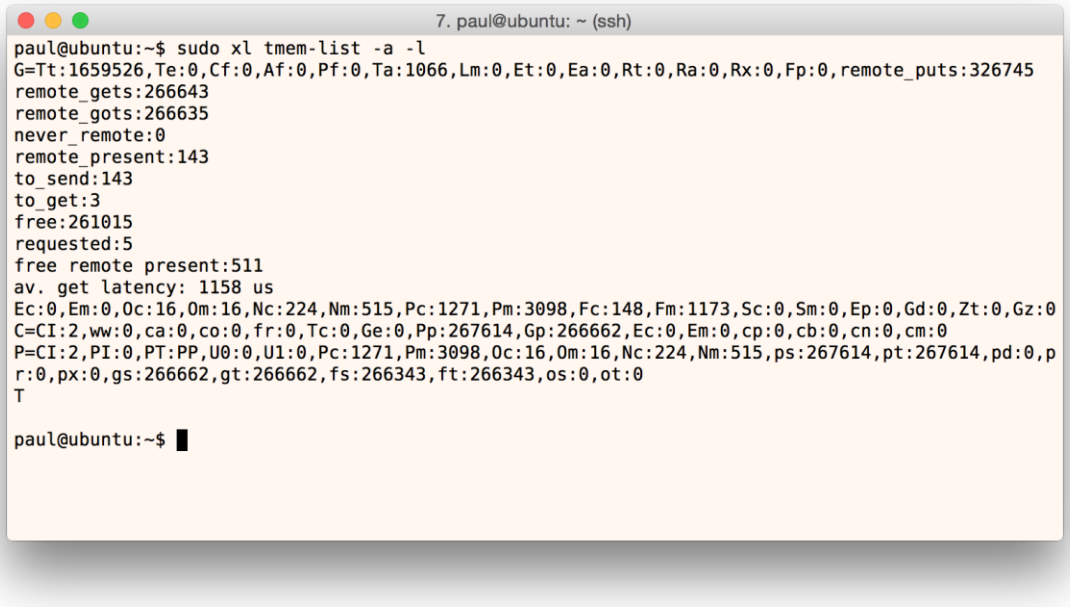


Figure 45: Tmem status with remote memory support

We are currently working to make progress on step 3 (dynamic memory reassignment), in order to have publishable progress in our research directions. Work on step 2 (emulation using MicroVisor remote DMA) will start soon.

Summary

In conclusion, BSC made a serious effort in the first half of the project to use the 32-bit discrete prototype. Significant effort was expended, as described in Appendix A, but it was eventually decided, with agreement from the rest of the consortium and apparent appreciation from the reviewers, to change direction to use software emulation. In the 2.5 months since the Interim Review at the end of June 2015, BSC has implemented the majority of a viable development platform, consistent with the rest of the project, which is sufficient to start making progress on the real work in the task. This work has been considerably delayed compared with our original intentions, but there is sufficient time

remaining within the project to successfully complete the work in the task. The aim is to develop and demonstrate intelligent and adaptive sharing of the total memory capacity among multiple hypervisors.

4. Considerations for Transparent Memory Compression (CHAL)

Main memory accounts for a major cost, especially in server systems. The fraction of the total hardware cost of a server system devoted to memory varies but it surfaces around 20%. On top of that, main memory also consumes a significant fraction of energy, on the order of 10%. In addition, main memory consumes space, proportional to its capacity. At the same time, many data center applications today, especially data analytics, are plagued by limited memory capacity. Hence, there is a need to offer increased main-memory capacity at low cost, lo*w energy and high performance.*

The memory compression technology being developed in WP3 compresses memory content continuously "under the hood", typically by a factor of three, and uses the extended memory capacity to increase the swap space so as to reduce the number of disk accesses. It uses statistical memory compression to enable aforementioned values. Unlike other attempts in the past, the approach is to compress all memory content using canonical Huffman coding, based on value frequency sampling. At the time a page is brought into the system from disk, it is compressed into main memory. The hardware support needed for memory compression comprises fast and effective compression/decompression engines, value frequency tables and an address translation mechanism that maps physical pages to compressed memory. These mechanisms are developed and integrated in a prototype for proof-of-concept in WP3.

Apart from the hardware support needed, a key challenge is related to management of compressed memory. The memory resources released by compression have to be exposed to the OS to be utilized. A key assumption is to manage the compressed memory "under the hood" by requiring no changes to the system software (the OS and virtualisation layers). The memory compression technology being developed does this using a separate software Memory Compression Virtual Memory Management (MCVMM) module. In this new task, work is proposed for the development of MCVMM.

Chalmers, in collaboration with FORTH, is planning to develop a software module for management of compressed memory. A key assumption is that it will transparently manage the compressed memory without any changes needed to existing system software. One approach to be investigated is to expose memory being freed up by compression to a swap space called victim cache that allows pages selected for eviction to stay in system memory longer without being written back to disk. MCVMM is responsible for several functions including 1) monitoring of how much memory is available 2) the mapping of physical pages to compressed pages 3) address translation between physical and compressed memory 4) address translation exceptions 5) management of the swap space and 6) reclamation of pages being stored in the swap space. A prototype of MCVMM will be built and evaluated by M30 (Delivarables D4.6 and D4.7).

5. Summary

This deliverable describes the outcomes of Task 4.3 (UTILIZE), and specifically of subtasks T4.3.1 and T4.3.2. The objective of these tasks is to re-architect the systems software stack, so as to achieve a more efficient use of available resources, and also to improve scalability. The resources under consideration are primarily memory pools, but also include fast I/O devices (storage and network).
The accomplishments described are as follows:

- Exploration and evaluation (in terms of latency) of a control and data path from a compute unit to a fast storage device, and techniques to share this device. This exploration shows that software overheads dominate the observed I/O latency.
- Hypervisor support for sharing remote resources (the MicroVisor concept, based on Xen)
- Mitigation techniques for performance interference in the storage I/O path. We find that with our techniques a server can run more workloads close to their nominal performance level as compared to the unmodified Linux I/O path, by careful allocation of I/O path resources to each workload.
- Use cases for utilizing remote memory: (a) memory capacity extension in the OS, (b) swap device, (c) memory allocator on top of an I/O device.
- Evaluation of remote memory access primitives: load/store, DMA
- Emulation of memory capacity sharing across coherence islands in the Xen hypervisor (basis of MicroVisor), using the Transcedent Memory (tmem) mechanism
- Progress towards transparently compressed main memory (topic of a separate upcoming deliverable: D4.6, due by M30).

The results of the work described in this deliverable will be carried over to the more advanced 64-bit discrete prototype, and eventually to the final chiplet-based final prototype.

Appendix

I. Remote memory sharing in the MicroVisor code listing

Running from a virtxd controller;

```
curl –X GET 'http://192.168.1.1:8000/mvNode'
```

The response would be (truncated to only show relevant elements)

From A:
Local node:

```
"RDMA": {
      "localExported": {
            "2a8f7d4d": {
                  "mv": "cccc32",
                  "vaddr": "0xffff8302011aa000",
                  "size": "0x00000000002000"
                  }
      },
      "remoteAccessed": {
            "8fab045c": {
                  "mv": "cccc02",
                  "vaddr": "0xffff8302185a4000",
                  "size": "0x00000000001000"
            }
      }
}
```

Remote node:

```
"RDMA": {
      "localExported": {
            "f96d20f1": {
            "mv": "cccc02",
            "vaddr": "0xffff8302185a4000",
            "size": "0x00000000001000"
            }
      },
      "remoteAccessed": {}
}
```

II. MicroVisor – modifications required for XGene-1 ARM-64 board to work

Toolchain
The toolchain used for building Xen, Linux and the RAMDisk are available at
http://releases.linaro.org/14.11/components/toolchain/binaries/aarch64-linux-gnu/gcc-linaro-4.9-2014.11-x86_64_aarch64-linux-gnu.tar.xz

Linux Kernel 4.2.0-rc6
Linux source code from git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
Linux image built using the following commands;

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- mp30ar0_defconfig
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image
```

The device tree that is used is apm-mustang.dts (found under arch/arm64/boot/dts/) after applying the necessary modifications for the specific mp30ar0 platform based on a device tree provided by Gigabyte. The device tree blob was built by running:

```
$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- apm-mustang.dtb
```

The Linux image created is used for both the driver domain and guest VMs.

Xen 4.6.0-rc
Xen Source code from git://xenbits.xen.org/xen.git
Initial development work towards ARM platform required porting of MicroVisor to a more recent Xen version. Changes were then applied to the MicroVisor to allow it to run on the arm64 platform. Examples of the changes involved include;

- Pass the xenstore/xenconsole mfn and evtchn via guest's HVM parameters list instead of start info page
- Hide all the devices from guests, except those explicitly passed-through during the creation of driver domains, by assigning them to xen and not adding their nodes to the guest device tree
- Adjust MMIO address and interrupt info in vtimer and vgic nodes for guests etc.

The Xen uboot image was built with the following commands;

```
$ make dist-xen XEN_TARGET_ARCH=arm64 debug=y
CONFIG_EARLY_PRINTK=xgene-storm CROSS_COMPILE=aarch64-linux-gnu-
$ mkimage -A arm -C none -T kernel -a 0x0200000 -e 0x00200000 -n Xen
-d xen/xen uXen
```

Busybox RAMDisk
Busybox image source was from http://busybox.net/downloads/busybox-1.23.2.tar.bz2
Libraries
To support the MicroVisor on ARM the following libraries have been compiled in a manner similar to that of libc that is shown below; (lib/libm.so.X, libnss_compat.so.X, libnss_db.so.X, libnss_dns.so.X, libnss_files.so.X, libnss_hesiod.so.X, libnss_nis.so.X, libnisplus.so.X)

Libc sources from http://ftp.gnu.org/gnu/libc/glibc-2.21.tar.xz

```
$ wget http://ftp.gnu.org/gnu/libc/glibc-2.21.tar.xz
$ tar -xJf glibc-2.21.tar.xz
$ mkdir glibc-build
$ cd glibc-build/
$ CC=aarch64-linux-gnu-gcc CFLAGS=-static ../glibc-2.21/configure --
host=aarch64-linux-gnu --target=aarch64-linux-gnu --enable-add-ons -
-enable-static-nss
$ make
```

```
$ make install install_root=<rootfs>
```

In addition for building dropbear the following was performed;

```
$ wget https://matt.ucc.asn.au/dropbear/dropbear-2015.67.tar.bz2
$ tar -xjf dropbear-2015.67.tar.bz2
$ cd dropbear-2015.67/
# CC=aarch64-linux-gnu-gcc LD=aarch64-linux-gnu-ld ./configure --
host=aarch64-linux-gnu --prefix= --disable-zlib

# make PROGRAMS="dropbear dropbearkey scp dropbearconvert dbclient"
STATIC=1 MULTI=1
```

Copy dropbearmulti into usr/bin then cd usr/bin and do

```
$ ln -s dropbearmulti dropbearconvert
$ ln -s dropbearmulti dropbearkey
$ ln -s dropbearmulti dropbear
$ ln -s dropbearmulti dbclient
$ ln -s dropbearmulti scp
```

Other libraries and binaries:

There are three compiler flags that should be enabled when building for the arm64 microvisor platform and all libraries should be linked statically.

```
--host=aarch64-linux-gnu --target=aarch64-linux-gnu  --built=<host
platform>
```

Driver support

To allow for driver support in the Linux Kernel to be enabled the following configuration options are required;

```
CONFIG_AHCI_XGENE=y CONFIG_NET_XGENE=y
```

The device tree also has to be updated with the device to be passed into the guest for the device to be considered and the driver loaded.

III. Memory capacity sharing across coherence islands

As mentioned in Section **Erreur ! Source du renvoi introuvable.**, the work of BSC in T4.3.2 initially ocused on building a viable development platform using the 32-bit discrete prototype. In light of project's plans to develop a 64-bit discrete prototype, as well as reviewer comments in the M18 review about a lack of coordination among the partners in WP4, BSC decided, with the agreement of the consortium, to align its work with the OnApp MicroVisor platform. This ongoing work is described in Section **Erreur ! Source du renvoi introuvable.**. The deprecated work is described in detail here, in the ppendix.

BSC spent considerable time evaluating the possibilities for virtualisation on the 32-bit discrete prototype. We first evaluated OnApp's port of Xen, finding it to be too unstable for our work. We also ported and evaluated Xvisor, a hypervisor with proper support for ARM Cortex-A9. Although we were able to get Xvisor to work and pass all our stability tests on a single MicroZed board, we have not been able to get it to boot a virtual machine on the 32-bit discrete prototype.

Familiarisation with toolchain and MicroZed started near the end of June 2014 (M10, slightly ahead of the M13 scheduled start time). During this time, significant time was spent reading the EUROSERVER documentation, getting to understand the details of the project, ARM cores, setting the trend for our

research contribution and other details. This period lasted until the end of August. Important landmarks for this period are:

1. Understanding ARM processors by reading the manuals and technical references
2. Understanding details of the EUROSERVER 32-bit discrete prototype and silicon prototype
3. Understanding the state-of-the-art in NUMA technology for multicore processors
4. Setting the trend for the research contribution: a virtualisation extension to enable aggregation of resources in a virtualized environment that is distributed, scalable and layered. This is recurrent and it is not yet complete. We have conceptualized the proposal, but we are still on the process of developing the theoretical aspects of it, as well as the high-level design.

Work with the MicroZed board started around September 2014, when some time was spent testing the board with the out-of-the-box Linux kernel and the Xilinx Vivado tools. An important landmark achieved by the end of September / beginning of October was to become familiar with the developer tools of the MicroZed.

This consisted of the following:

1. Able to install the Vivado suite and generate a hardware model for the programmable logic (FPGA) of the MicroZed board (Xilinx Zynq 7010 SoC).
2. Compilation of the Xilinx's Linux [5], a GPL kernel adapted from a vanilla kernel. The porting was done by the Xilinx developer team.
3. Testing functionality of the kernel in the MicroZed board. Many functional tests were performed, following the tutorials provided by Avnet.

Evaluation of OnApp's Xen-PV on MicroZedOctober to December 2014, we set up the Xen port developed by OnApp with the Samsung's PV kernel. OnApp had taken the effort of porting Xen to the ZedBoard, which use ARM Cortex-A9 cores, just like the MicroZed. They had also taken a kernel that was paravirtualized and ported to the Cortex-A9 cores by Samsung. However, OnApp personnel also made some contributions to the PV kernel itself.

By the end of November, Xen was able to run on the MicroZed, but with some limitations. In parallel with this, the team at the BSC also managed to make Xvisor work on the MicroZed. The porting of the Xvisor into the MicroZed consisted of adapting the Linux drivers and the QEMU device emulators for Xvisor. This was initially done as a "plan B", as we suspected that there may be serious problems with OnApp's port of Xen, whereas Xvisor has good support for Cortex-A9. For this reason, the initial work was done in a relatively "unclean" way. Xvisor does not require the kernel to be paravirtualized, so we used the port of the Xilinx's Linux kernel.

In summary, important landmarks from this period were:

1. Get Xen to boot on the MicroZed using Samsung's PV kernel as Dom0
2. Increase memory available for Xen and virtual machines
3. Enable persistent storage for Dom0 on the SD Card
4. Xvisor on MicroZed with one guest virtual machine.

However, the OnApp port of Xen has several limitations that will be detailed in a later section.

From mid-December 2014 until the beginning of February 2015, the time was spent mostly on making more utilities available for the MicroZed under the Xen/Dom0 arrangement. In addition, significant time was spent on choosing, compiling and running benchmarks for the MicroZed board, using both Xen and the Xilinx's Linux software. Important landmarks from this period:

1. Cross-compiling important utilities for the Xen/Dom0 arrangement: sftp, ssh, make, awk, zlib.
2. Benchmark survey to run over the microzed, cross-compilation of the benchmarks and execution of the benchmarks. The benchmarks used were:
a. lmbench
b. memtester
c. STREAM

IV. Xvisor on MicroZed

In February 2015, we decided that OnApp's port of Xen was not stable enough for our work. We therefore chose to work with "Plan B", which is Xvisor. We cleaned up the Xvisor code base, removed modifications to the kernel image, and with a more thorough understanding of the memory allocation mechanisms. The objective was to get the Xvisor to run using a cleaner approach to enable it over the MicroZed board. Important landmarks achieved were:

1. Thorough analysis of memory allocation for virtual machines
2. Booting up the Xvisor with a cleaner code base
3. Testing Xvisor running benchmarks inside a virtual machine.

V. Work on the 32-bit discrete prototype

From March 2015 until the beginning of May 2015, a lot of work was done on the 32-bit discrete prototype after it arrived at our laboratory at BSC. During this time, we focused on assembling the prototype, booting the MicroZed boards using the guidelines and kernel images provided by Forth. In addition, we were also able to run the Xvisor over the prototype using remote memory access through the FPGA. However, we encountered some limitations when trying to start up virtual machines on top of Xvisor. Important landmarks achieved were:

1. Assembling of the discrete prototype using the HMC board, the FPGA with the ANoC support and the MicroZed boards.
2. Able to run FORTH's Linux binary at the beginning of April (there were various small issues that had to be solved, with the help of FORTH).
3. Executing Xvisor on the discrete prototype and starting virtual machines in one of the MicroZed boards. We were able to read/write remote memory using Xvisor's command line.

Important limitations were found when starting a virtual machine on one of the MicroZed boards of the discrete prototype. This prompted us to reconsider other choices for virtualisation platforms.

By the end of April 2015 and up to June 2015, we started focusing our efforts on the newer version of Xen. The objective is to understand the memory management mechanisms implemented on Xen, and then to implement basic mechanisms to page memory in and out of reserved memory space transparently to the VMs.

In order to summarize this, we show the following timeline chart.

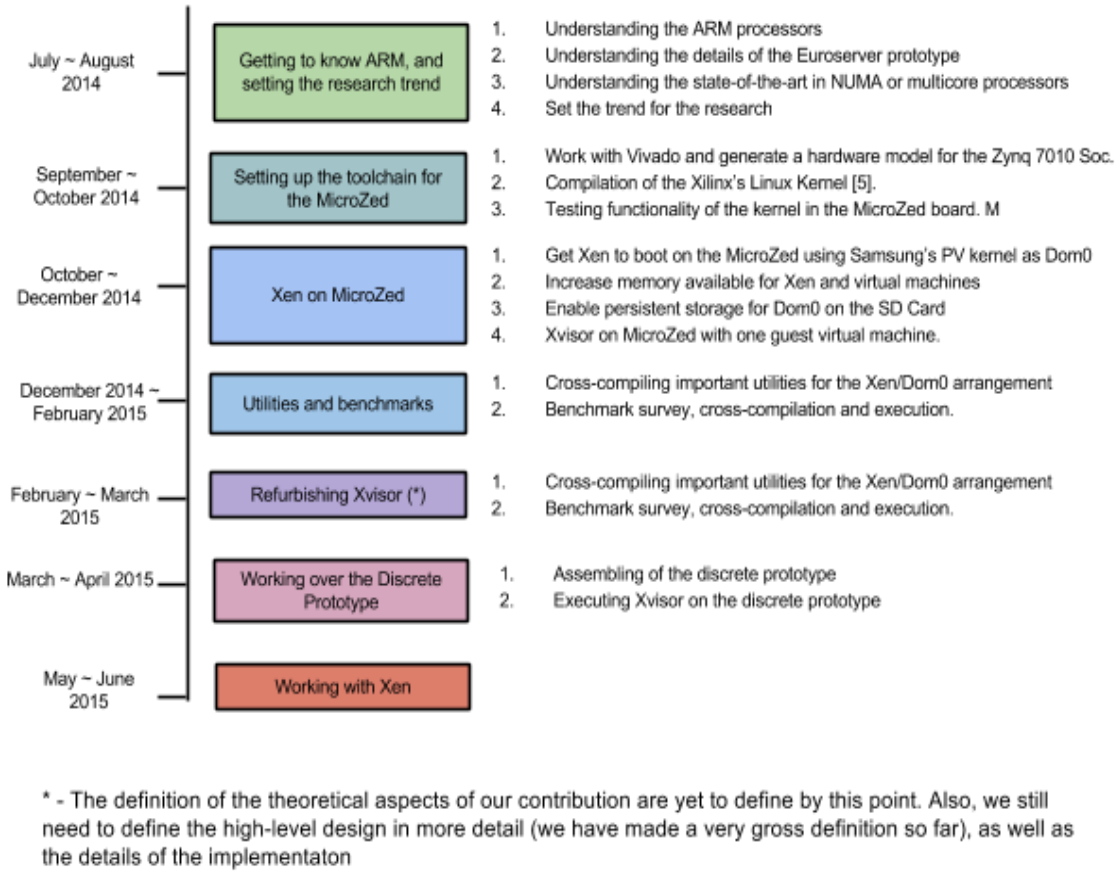


Figure 46: Timeline of BSC's work on hypervisors.

a. MicroZed evaluation

In this section, the evaluation of the MicroZed [1] is described. We first describe the characteristics of the MicroZed board and its most relevant hardware details. We describe the different test scenarios that we used to run the benchmarks in, and the motivation behind each one. Next, we describe the benchmarks used, which are very well known and common. Finally, we present a summary of the results so far, but we do not present the results of the benchmarks themselves because we have shifted our efforts into using Xen with a different platform for prototyping. So, we did not see any benefits on generating evaluation charts for a platform that will not be employed in the future stages of the project.

The MicroZed [1], developed by Avnet, is a low-cost development board based on the Xilinx Zynq 7000 All Programmable SoC [2,3]. The Zynq SoC hos an important flexibility feature that allows design engineers to enable the hardware blocks that the SoC can have and certain aspects of their hardware blocks' configuration. There are two versions of the board, the MicroZed Zynq 7Z010 and the 7Z020. In this work, we make use of the 7Z010. The MicroZed 7Z010 consists of a processing system (PS) block and a programmable logic (PL) block [3, 4]. The processing element is a dual-core ARM Cortex-A9 [5] processor, and the programmable logic is an FPGA logic fabric based on the Xilinx 7-series FPGA architecture. Thus, the MicroZed 7Z010 integrates an FPGA with an ARM processor in a single SoC. The presence of the FPGA fabric logic is what gives the system its flexibility.

The integration between the two blocks is made through an industry standard AXI interface [6], which enables communication between the PS and PL in a very fast and efficient way. The Zynq SoC also has

integrated memory (1 GB for the 7Z010) and peripherals. The PS is capable an application level processor capable of executing a Linux operating system, while the PL is ideal for implementing high-speed logic, arithmetic and data flow subsystems [4]. It is flexible enough to allow for the designer to create its own peripherals.

The programming of the PL inside the SoC and the access of the PS to the peripherals has to be programmed and configured using Xilinx's toolchain. The toolchain is made available when installing the Vivado Design Suite [7]. Using Vivado, the designer is able to generate a boot image that the board reads upon power-up. The boot image contains the hardware model for the SoC (the peripherals available, configuration for the peripherals and connections to the PS), a first-level boot loader [8] that is generated by the Xilinx's Software Development Kit [9] and a u-boot image which is the second-level boot loader. The u-boot image is an executable binary that has to be compiled for the Zynq SoC, and tells the PS where to fetch the software to execute. This could a stand-alone bare metal application or a fully fledged operating system. It is necessary to provide a device tree [10].

b. Test scenarios for the MicroZed evaluation

The MicroZed board was successfully booted using the Xilinx's tool chain, and tested using stand-alone applications and with the Linux kernel. Testing was also carried out using different hypervisor software.

Having a hypervisor over the MicroZed is vital for the validation of the EUROSERVER [11] prototype. Our contribution to the EUROSERVER consists of a new technology that will allow to aggregate resources (initially, we will focus on memory) in a distributed way across locally coherent and independent computation nodes interconnected among each other through a very fast interconnect. In order to increase the chances for our proposal to be integrated into state-of-the-art technology used in data-center environment, we decided to conceptualize it and design it as an extension to state-of-the-art virtualisation technology. Thus, the validation of our contribution needs hypervisor software to run over the MicroZed.

There are many GPL hypervisor software solutions available:

1. KVM [12]: not possible to use on the MicroZed 7Z010. The ARM Cortex-A9 does not have the virtualisation extensions necessary to run KVM. This option was discarded.
2. Xen [13]: a port was made by OnApp using a PV kernel as Dom0, ported by Samsung.
3. Xvisor [14]: a lightweight GPLv2 hypervisor developed by Anup Patel and other community members. This hypervisor is known to work on ARM Cortex-A9 cores.
4. OKL4 [15]: also GPL, developed by Open Kernel Labs. Also known to work on ARM Cortex-A9 cores. No serious attempts were made at getting the source code.

The MicroZed board was evaluated under the following scenarios:

1. Xilinx's Linux with ramdisk filesystem: the benchmarks were executed inside the Xilinx version of Linux. This would provide a reference for comparison of the other scenarios.
2. Xen hypervisor running only domain 0 with the filesystem mounted on persistent storage on an SD card: the benchmarks were run in the management domain without any guest domains. For short, we label this as: "Xen-Dom0-SD".
3. Xen hypervisor running only domain 0 with ramdisk filesystem: the benchmarks were run in the management domain without any guest domains, but the guest domain has no filesystem mounted in persistent storage. For short, we label this as: "Xen-Dom0-Ramdisk".
4. Xen hypervisor running DomU: the benchmarks were run in the guest domain that has a ramdisk filesystem. For short, we label this as: "Xen-DomU".
5. Xvisor with one guest VM: the benchmarks were run in a VM running on top of XVisor. It is not possible to run applications from the Xvisor management prompt.

6. Xvisor with no guests in the discrete prototype: once the prototype was assembled, we needed to test the capabilities for remote memory access somehow. To do this, we implemented one of the benchmarks as a command in Xvisor's command line in order to perform some testing.

All the scenarios described above are implemented using only a single MicroZed board, except for Scenario 6, which uses the full 32-bit discrete prototype.

Scenarios 2 to 5 are compared against Scenario 1. The purpose of making these comparisons was to make a decision on whether we choose Xen or Xvisor as our hypervisor for evaluation.

In Scenario 2, we evaluate the performance and functionality of the benchmarks running on the privileged domain of Xen mounting the filesystem on an SD Card. When mounting the filesystem on an SD Card, the RAM memory does not need to hold the filesystem, enabling for more memory to be available for the processes running on Dom0. This should allow for faster execution of the benchmarks and allow them to use more memory resources.

Scenario 3 was also tested for two reasons:
1. In order to understand the impact of limited to amount of RAM available to the process.
2. in Scenario 1, the filesystem was also mounted on ramdisk, so we also wanted to be able to make a more fair comparison. However, we are not sure how "fair" this is or not. After running the benchmarks, any differences would have been evaluated and further investigation would have ensued.

The objective of Scenario 4 was to test the performance and correctness of the benchmarks on an unprivileged domain in Xen. This would allow a better evaluation between Scenario 4 and Scenario 5, which basically consisted of running the benchmarks in an unprivileged domain over Xvisor.

Scenario 6 was executed using only a subset of the Memtester benchmark in order to functionally test the remote memory access capabilities.

c. Benchmarks used

So far, we have used three benchmark suites in the different testing scenarios. These are:
1. memtester [16]: authored by Charles Cazabon. A user-space utility for testing memory subsystem faults.
2. stream [17]: authored by John McCalpin. Measures sustainable bandwidth.
3. lmbench [18]: authored by Larry McEvoy and Carl Staelin. Also GPL, measures latency and bandwidth.

Memtester

There's a strong motivation behind using the memtester benchmark, and this is strongly related to the architecture of the EUROSERVER prototype. In the following figure, we show a block diagram of the architecture of the EUROSERVER prototype with our proposed software layers.

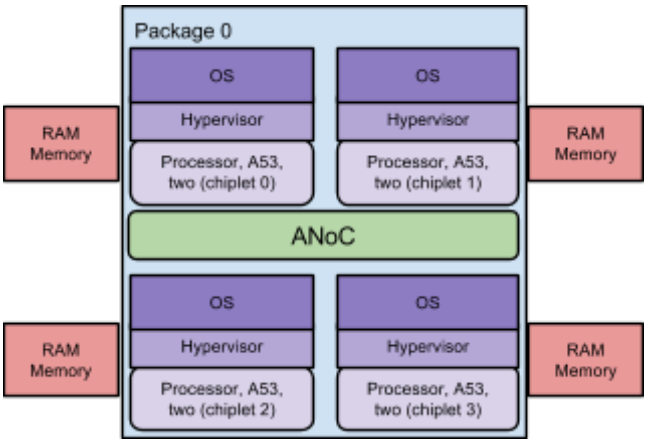


Figure 47: Block diagram of the architecture of the final EUROSERVER prototype.

The ANoC is the fast interconnect that links the chiplets (compute nodes) to each other. The address space of the processors inside each compute node is partitioned into local and remote addresses. This means that there is a subset of the addresses in the address space that will access the local memory of the chiplet, while the other subset is distributed across three or more remote chiplets.

When an access to an address belonging to the remote subset of the addresses is performed by the processor, the AXI interconnect will send it through the remote interface towards the ANoC. The data will traverse the ANoC towards the destination chiplet, and it will be inserted in the AXI interconnect of the remote chiplet through the ANoC-AXI interface. This allows for one chiplet to have access to a portion of the memory of any other chiplet. However, there are details regarding the management of the permissions for the utilization of remote memory as well as the access per se through the page tables and MMUs. These details are beyond the scope of this report.

So, to access the remote memory, we have to take into account the following:

1. Set up the page tables in order to map from virtual address to physical address. The process that sets this up needs to be aware of remoteness and localness
2. When a remote access takes place, it has to traverse the ANoC
3. Once it arrives at the destination chiplet, it has to be able to enter the physical memory through an additional stage of translation
4. The response is sent back.

The memtester benchmark becomes relevant in this environment because it tests that the whole memory subsystem of the architecture, including local and remote memory, is functionally correct (not faulty) regardless of any overhead associated to it. The types of tests that memtester performs are:

1. Tests to reveal memory faults due to bits that are unable to change values normally: Random value, XOR comparison, SUB comparison, MUL comparison, DIV comparison, OR comparison, AND comparison, Sequential Incremente, Block Sequential, Solid Bits, Bit Flip, Checkerboard, Walking Ones, Walking Zeros, Bit Spread (bitwise operations).
2. Tests to determine if memory allocations are addessable: performs things like “Stuck Address” [16].

Memtester tests correctness of the memory subsystem.

STREAM

We use this benchmark to test the sustainable bandwidth of the memory subsystem [17], i.e. the speed of the links from the perspective of the processor. STREAM uses computation kernels that stress the

memory subsystem in a number of ways, using enough data to exceed the capacity of the caches of the processor.

STREAM is particularly useful to us since it will allow to get measurements on the sustainable bandwidth for accesses that have to traverse the AnoC across the chiplets, and allow us to compare the latency costs of accessing remote memory versus the latency expected when accessing local memory. Taking these latencies into account are of extreme importance for our contribution.

Using STREAM, we could measure the latency cost in scenarios when the AnoC is managing a relatively low amount of traffic and when the ANoC is under a lot of stress. Hence, we will be able to understand the variability of the latency costs under different operating conditions. As an additional test for future evaluation, we would like to also tests power consumption and how much of it is due to ANoC traffic, but this will evaluated at a later stage.

lmbench
It runs as a user process within the runtime of an operating system. As mentioned before, it measures latency and bandwidth. It performs the following tests [19]:

- Bandwidth benchmarks
  - Cached file read
  - Memory copy (bcopy)
  - Memory read
  - Memory write
  - Pipe
  - TCP
- Latency benchmarks
  - Context switching.
  - Networking: connection establishment, pipe, TCP, UDP, and RPC hot potato
  - File system creates and deletes.
  - Process creation.
  - Signal handling
  - System call overhead
  - Memory read latency
- Miscellaneous
  - Processor clock rate calculation

Lmbench has been around since the mid 1990s. It still remains a relevant tool to test the performance of Linux-based systems, even when running in virtualized environments [20]. However, it is important to understand that lmbench by itself needs to be complemented with applications that are more relevant for our purposes.

d. Benchmark Evaluation Results

In the following table, we present the results obtained when running the benchmarks in each of the scenarios above. We were not able to execute the benchmarks under all the scenarios due to some limitations encountered in the respective environments when attempting the execution. There were major limitations when trying to run the benchmarks over a single MicroZed board. The major limitation was encountered by scenarios 2, 3 and 4. The main issue is that the Xen port for the ARM Cortex-A9 cores is unstable and it sometimes freezes when attempting to boot up a virtual machine.

Scenario 5 was very stable when attempting to test it, but when using the same Xvisor image with extended memory for Scenario 6, the system would again turn unstable issuing data aborts (among other errors) when attempting to start a virtual machine. This error seemed to be related to the interaction between the Xvisor and the new capabilities of the hardware. However, we were unable to determine the real source of the errors and unstable behavior.

Table 8: Summarizing the scenarios that were able to execute the benchmarks.

	Memtester	Stream	Lmbench
Xilinx' linux	OK	OK	OK
Xen/Dom0 (mmcbl0p2)	OK	OK (around 30% slower)	OK (results have been analyzed)
Xen/Dom0 (ramdisk)	OK	OK (around 30% slower)	(unfinished after 24 hours)
Xen/DomU	FAIL (Seg Fault)	FAIL (Seg Fault)	FAIL (Seg Fault)
Xvisor/Guest1	OK	OK (around 50% slower)	(capacity issue, revise guest1 settings)
Xvisor With no Guests over the prototype using remote memory	OK	-	-

e. Summary and Conclusion

In conclusion, BSC has made a serious effort to use the 32-bit discrete prototype developed in the EUROSERVER project. When we found that the port of Xen developed within the project was not stable for our work, we decided to switch to Xvisor, which has proper support for ARM Cortex-A9. However, the plan to develop a 64-bit discrete prototype, together with a renewed effort to align the work in WP4 towards a smaller number of development platforms means that the work with the obsolete 32-bit discrete prototype was seen as less valuable.

In ongoing work we will use a software emulation platform developed by OnApp as part of their MicroVisor development. We will therefore prototype our work on Intel architecture using the Xen hypervisor. We will develop the support for memory capacity sharing on top of RDMA support between MicroVisors. This is currently being prototyped using paging in and out of a reserved area, which will emulate the remote memory accessed only via RDMA.

References

[1] MicroZed Zynq Evaluation and Development and System on Module Hardware User Guide. Avnet, 2014
[2] Zynq-7000 All Programmable SoC: Concepts, Tools and Techniques (CTT). Xilinx, 2012
[3] L. Crockett, R. Elliot, M. Enderwitz, R. Stewart. The Zynq Book: embedded processing with the ARM Cortex A9 on the Xilinx Zynq-7000 All Programmable SoC, First Edition, Strathclyde Academic Media, 2014.
[4] Creating a Zynq Hardware Platform in Vivado. Avnet, 2014.
[5] ARM Architecture Reference Manual, ARM v7-A and ARM v7-R edition. ARM Limited, 2004-2008
[6] AMBA AXI and ACE Protocol Specification. ARM Limited, 2011.
[7] Vivado Design Suite User Guide. Xilinx, 2014.
[8] A.B. Benayas. Development of Embedded Linux Applications Using Zedboard. Julio, 2013
[9] Xilinx Software Development Kit (SDK) User Guide. Xilinx, 2014.
[10] http://www.devicetree.org/Main_Page
[11] Y. Durand, P. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, J. Thomson. EUROSERVER: Energy Efficient Node for European Micro-servers. In 17th Euromicro Conference on Digital System Design (DSD), 206 - 213, August, 2014.
[12] http://www.linux-kvm.org/
[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield. Xen and the Art of Virtualisation. SOSP, 2003, Bolton Landing, New York, USA
[14] http://xhypervisor.org/
[15] G. Heiser, B. Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. APSys 2010,.
[16] http://pyropus.ca/software/memtester/
[17] McCalpin, John D., 1995: "Memory Bandwidth and Machine Balance in Current High Performance Computers", IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.
[18] L. McVoy, C. Staelin. lmbench: Portable Tools for Performance Analysis. Proceedings of the USENIX 1996 Annual Technical Conference San Diego, California, January 1996
[19] http://lmbench.sourceforge.net/whatis_lmbench.html, lmbench info
[20] http://www.ok-labs.com/blog/entry/why-lmbench-is-evil/
[21] D. Magenheimer, C. Mason, D. McCracken, Kurt Hackel, Transcendent Memory and Linux. Proceedings of the Linux Symposium, 2009.