D5.2
HW-SW integration and tuning

Workpackage:	WP5	System Integration & Evaluation
Author(s):	Alvise Rigo	VOSYS
	Andrea Bartolini	ETHZ
	Valeria Bartsch	FHG
	Iakovos Mavroidis, Nick Kallimanis, Manolis Marazakis, Vassilis Papaefstathiou	FORTH
	Antoniu Pop	UOM
Authorized by	Antoniu Pop	UOM
Reviewer	Dirk Pleiter	JUELICH
Reviewer	Didier Lattard	CEA
Reviewer	Iakovos Mavroeidis	FORTH
Dissemination Level	PU	

Date	Author	Comments	Version	Status
2017-03-11	A Pop	Initial draft	V0.0	Draft
2017-06-26	A Pop	First complete draft	V0.6	Draft
2017-07-17	A Pop	Internally reviewed	V0.8	Draft
2017-07-18	A Pop	Final document	V0.9	Submitted

Executive Summary

This deliverable describes the system-level integration of the initial software environment (firmware, OS services, runtime, hypervisor services, virtualization activities), described in D3.1, and the multi-board hardware prototype designed in D5.1. Throughout this integration process, both the software and hardware platform have been tuned in order to provide efficient use of resources. Such tuning involves the design and implementation of the Ultrascale+ coherence island, runtime optimizations for locality and latency, high-speed low-latency inter-chip communication, low-power techniques, and design of scalable HW sensors monitoring infrastructure with minimal intrusiveness and overhead.

Table of Contents

Table of Figures

Table of Tables

No table of figures entries found.

List of abbreviations

Term	Definition
API	Application Programmer Interface
BW (MPI)	Busy Waiting MPI
CPU	Central Processing Unit
DoA	Description of the Action
DSA	Dynamic Single Assignment
DVFS	Dynamic Voltage and Frequency Scaling
EAW	Energy-Aware MPI Wrapper
ECED	Edge and Coherence-Enhancing Anisotropic Diffusion filter
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FSP	First Step Problem
GAS	Global Address Space
GASNet	Global Address Space Networking
GASPI	Global Address Space Programming Interface
GPL	GNU General Public License
GPU	Graphics Processing Unit
GSAS	Global Shared Address Space
HLS	High-Level Synthesis
IB (MPI)	Interrupt-based MPI
ILP	Integer Linear Programming
ISP	*i*-th Step Problem
MCTP	(Fraunhofer's) Multicore Thread Package
MPI	Message Passing Interface
MPSD / MPMD	Multiple Program Single/Multiple Data
NUMA	Non-Uniform Memory Access
OS	Operating System
OTC	Optimal Thermal Controller
PGAS	Partitioned Global Address Space
PoC	Proof of Concept (prototype)
RDMA	Remote DMA (Direct Memory Access)
RTM	Reverse Time Migration
SPSD / SPMD	Single Program Single/Multiple Data
SMP	Symmetric Multiprocessor
TDP	Thermal Design Power
TMC	Thermal-aware Task Mapper and Controller
UDP	User Datagram Protocol

1 Introduction

The ExaNoDe project is developing a unique HPC system architecture founded on the UNIMEM architecture, which is also the basis for the related projects EUROSERVER[1] and ExaNeSt[2]. A system that implements UNIMEM consists of a number of computational nodes connected through a custom network. Each node typically contains multiple processing cores, which communicate amongst themselves using coherent shared memory as provided by the hardware. Distinct nodes communicate using UNIMEM's global shared address space (GAS), which provides non-coherent load–store and RDMA access to any other remote node. The UNIMEM hardware architecture is exposed to user space via the Global Shared Address Space (GSAS), user-space RDMA, mailbox and remote allocator APIs defined in D3.6.

This deliverable describes the system-level integration of the initial software environment (firmware, OS services, runtime, hypervisor services, virtualization activities), described in D3.1, and the multi-board hardware prototype designed in D5.1. Throughout this integration process, both the software and hardware platform have been tuned in order to ensure efficient interactions between the different software and hardware components.

The design and implementation of the Ultrascale+ coherence island solves the issue of high inefficiency of the Juno PCIe link by employing Xilinx Ultrascale+ FPGAs which were made available to the project towards the end of the first year. We further describe the SerDes link architecture for high-speed low-latency inter-chip communication, as well as low-power techniques to reduce virtualization overhead and power consumption, and the design of scalable HW sensors monitoring infrastructure with minimal intrusiveness and overhead.

The GSAS environment API, an extension of the global address space communication mechanism enabled by the UNIMEM architecture, allows applications to allocate/de-allocate virtual shared address space, to perform atomic reads, writes and many other operations on the allocated space by using appropriate library calls. The user-space initiated DMA library, capable of transferring to/from any memory location throughout the whole global memory space, exposes the functionality of the DMA engines that UNIMEM provides to user space. Sockets over RDMA and a mailbox-style notification mechanism are also provided.

GPI is an open-source communication library that implements the GASPI standard PGAS API. It provides a portable and lightweight API that leverages remote completion and one-sided RDMA-driven communication, both being efficiently supported by the UNIMEM architecture. As such, GPI is an appropriate communication library to benefit from and evaluate the UNIMEM architecture.

1 http://www.euroserver-project.eu/
2 http://www.exanest.eu/

2 Prototype and Global Shared Address Space (FORTH)

The UNIMEM architecture, introduced in the Euroserver project, aims to provide a scalable distributed system solution enabling direct remote memory accesses and shared memory. The UNIMEM architecture can be deployed in a system consisting of several "coherence islands", where a coherence island includes one or more processors, a cache coherent memory, various peripherals and an external port for remote accesses. All accesses inside the coherence island are cache coherent. In the ExaNoDe architecture, a Xilinx Ultrascale+ FPGA is one such coherence island and a collection of FPGAs form the ExaNoDe system which supports the UNIMEM architecture. Each FPGA supports a physical address mapping for accessing its local memory, its local peripherals and the external world (i.e. memories and peripherals of other FPGAs in the UNIMEM architecture). This "window" to the external world provides direct memory and I/O accesses (through standard load/store instructions) to other coherence islands and it is used to provide a global address space, as shown in Figure 1. The physical interface to the remote world can be anything (PCIe, AXI, Ethernet, etc.).

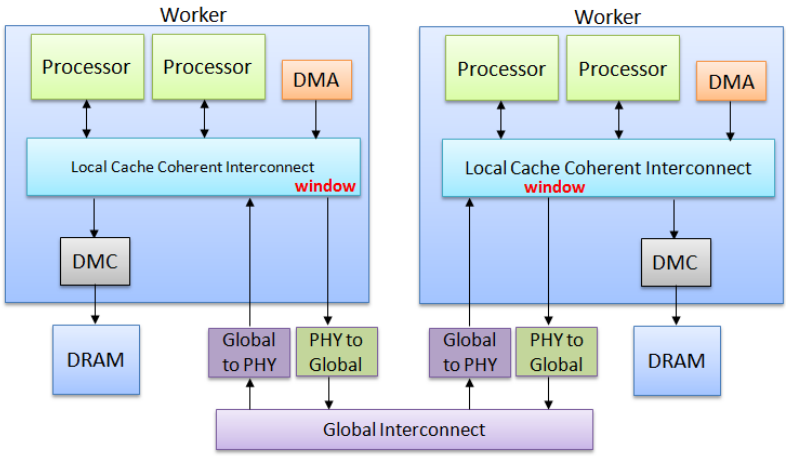


Figure 1: Accesses in the external "window" of the local address space are translated to global memory access.

Within ExaNoDe we first extended the UNIMEM architecture of the Juno-based prototype, as described in D5.1. Low-latency communication between the coherence islands is very critical for the UNIMEM architecture. The coherence island in the UNIMEM architecture of the Juno-based prototype is a Juno board. A Xilinx KCU105 board, supporting a Xilinx Ultrascale FPGA, is connected to the PCIe slot of the Juno board in order to provide external connectivity. Since the latency of the PCIe link is very long (about 200 CPU clock cycles) the communication latency between the Juno-based coherence islands is so high that the UNIMEM architecture is inefficient. When ExaNoDe decided to use and extend the Juno-based prototype there was no other option since it was the only system employing 64-bit ARMv8 processors which support an external window through the PCIe link.

ExaNoDe resolved the latency issue of the Juno-based prototype by employing Xilinx Ultrascale+ FPGAs which were made available to the project about one year ago. This effectively replaced the Juno prototype architecture, which uses a discrete CPU and FPGA connected through a PCIe link, with an FPGA with integrated CPU cores.

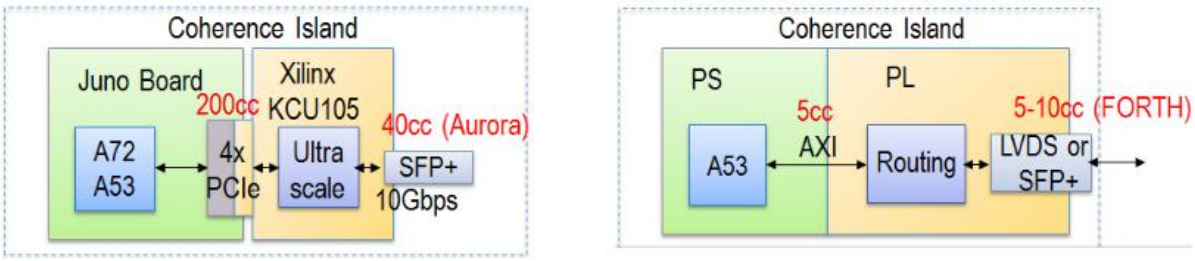


Figure 2: Latency of Juno coherence island and Ultrascale+ coherence island.

Thus ExaNoDe further ported the UNIMEM architecture onto the Trenz-based prototype. This is the second interim prototype that uses Xilinx Zynq UltraScale+ FPGAs and development boards marketed by Trenz Electronic GmbH, a company that provides development services for the electronics industry. Each Xilinx Ultrascale+ FPGA in this prototype is a UNIMEM coherence island. As shown in Figure 2 the latency to the external world is a few cycles and thus the communication latency between the coherence islands was highly improved compared to the Juno-based prototype.

This prototype is meant to assist in the development of a number of key components of the ExaNoDe system which are included in this deliverable. The major building blocks of this prototype are the TE808 Multi-Processor SoC module that mounts on a TEBF0808 baseboard. The prototype is comprised of multiple instances of those two components in an appropriate configuration. This prototype is served as a development platform for a number of ExaNoDe partners, hence, although based at FORTH's premises, it is also remotely accessible to those that need it.

The UNIMEM architecture has been reproduced using Trenz hardware in a setup such as the one presented in Figure 3. In effect, each Coherence Island is implemented using a single pair of the TE0808 module and TEBF0808 baseboard (as well as the JTAG/UART adapters, although their role secondary). This can be easily performed since the UltraScale+ FPGA of TE0808 features 64-bit quad application processors along with significant reconfigurable resources.

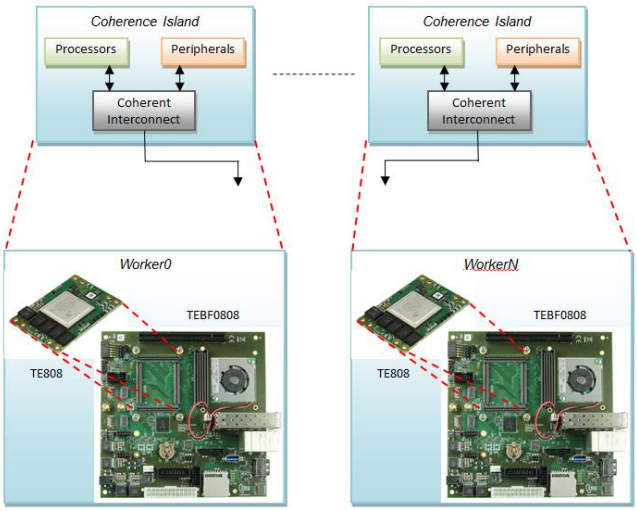
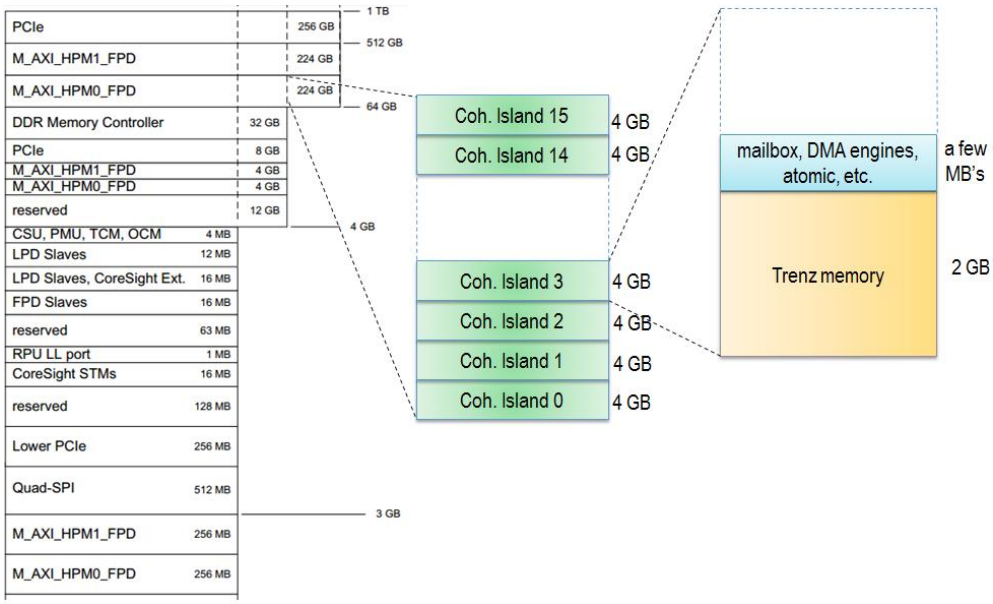


Figure 3: UNIMEM architecture where each coherence island is a Trenz kit.

The FPGA used on the Trenz-Node is the same as the one to be used in the final ExaNoDe prototype. The Trenz-based prototype will be actually a smaller version of the final prototype. It will connect several Xilinx Ultrascale+ FPGAs over high-speed links, in a similar way as the final ExaNoDe prototype. The only difference is that the final ExaNoDe prototype will be denser and it will support higher throughputs, more memory and it will demonstrate the use of system in package and interposer technology. Figure 4 shows the Trenz-based prototype at FORTH consisting of 8 Trenz kits.

Figure 4: The Trenz-based prototype at FORTH (on the left) and the Trenz kit (on the right).

Figure 5 shows a portion of the "Global System Address Map" of the physical space inside the Zynq Ultrascale+ FPGA (see Zynq Ultrascale+ Technical Reference Manual[3]). The Ultrascale+ Address Mapping provides regions of about 384 GB in total to the external world (M_AXI_HPM0_FPD and M_AXI_HMP1_FPD regions).



3 https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf

Figure 5: Address mapping of Xilinx Ultrascale+ and mapping to the UNIMEM global address space.

The external window of each Zynq Ultrascale+ FPGA of the Trenz-based prototype can be statically partitioned into memory domains, each one providing accesses to a Trenz-based coherence island as shown in Figure 5. Since each Trenz kit supports only 2GB of DRAM memory we statically map the external window to the global address space of the UNIMEM architecture allowing each Trenz board to access directly the whole UNIMEM space. In this way a Trenz-based coherence island can directly access any memory in the system without the need of any complicated translation mechanism in hardware.

3 Software Environment Porting to UNIMEM

The current implementation of the UNIMEM architecture exposes a set of programming frameworks as shown in Figure 6. These programming frameworks provide a powerful set of communication mechanisms to developers and to runtime systems, resulting in systems that are scalable for large numbers of nodes. These programming environments are the following:

1. The Global Shared Address Space environment (abbr. GSAS environment) and the communication mechanisms that it provides. The GSAS environment defines an application interface (API) that is an extension of the global address space enabled by the UNIMEM architecture. GSAS gives the ability to processes that run across remote nodes to communicate in a way resembling a system that provides coherent shared memory communication. More specifically, the GSAS environment allows the applications to allocate/de-allocate virtual shared address space, to perform atomic reads, writes and many other operations on the allocated space by using appropriate library calls.
2. User-space initiated DMA library. This library facilitates user-space initiations of DMA transfers. The DMA engine of the underlying system is capable of transferring to/from any memory location throughout the whole global memory space. The main part of this work aims to expose the functionality of the DMA engines that UNIMEM provides to user space.
3. Sockets over RDMA. With Sockets over RDMA, a UNIMEM system can utilize low-latency communication among local nodes, by means of fast RDMA transactions that bypass the kernel network stack.
4. Furthermore, we give a description of mailbox-style notification mechanism with which a kernel- or user-space application can send and receive messages to and from remote nodes, thus enabling remote notification capability.

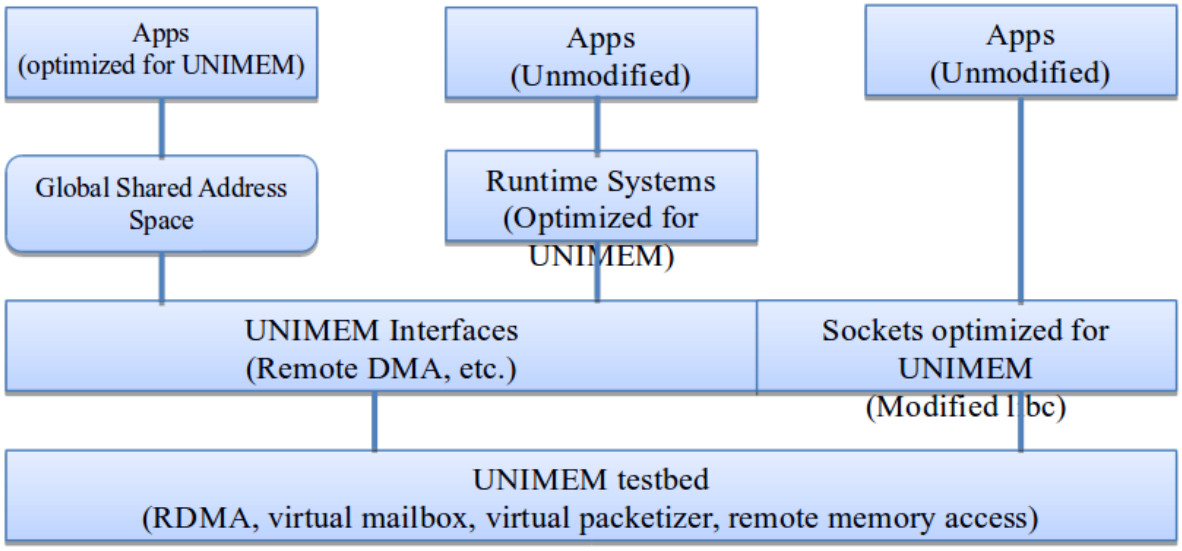


Figure 6 The UNIMEM programming interfaces, and their interactions.

In the following section, we provide an overview of the UNIMEM hardware and software interfaces. Details have already been documented in Deliverable D3.6 (submitted in M13). In this deliverable we focus on hardware and software co-design aspects of the global shared address space that we have designed and implemented on the current-generation of UNIMEM prototypes.

3.1 GSAS Environment (FORTH)

In this section, we describe the current generation of global shared address space (abbr. GSAS environment) and the provided mechanisms for inter-process communication across different nodes (i.e., Trenz prototype nodes).

Our global shared address space defines an application interface (i.e., API) that gives the ability to processes that run across remote nodes to communicate in a way resembling shared memory communication. More specifically, the GSAS environment allows the applications to allocate/de-allocate shared address space, to perform reads, writes and other atomic operations on the allocated space by calling the appropriate library calls.

In the GSAS environment, all the read, write and atomic operations on the allocated address space are performed via special user-level library calls and not via conventional load and store instructions provided by the ARM processors. Since these calls are user-level, they do not involve operating system's kernel, thus they are a low-latency, fast communication mechanism. Although this communication mechanism is efficient, local memory accesses performed by the conventional ARM processor instructions are more efficient. Thus, the communication mechanisms provided by the GSAS environment should not be used in cases that do not involve communication among remote processes. In Figure 7, the hardware and the software stack that the GSAS environment is built on is presented.

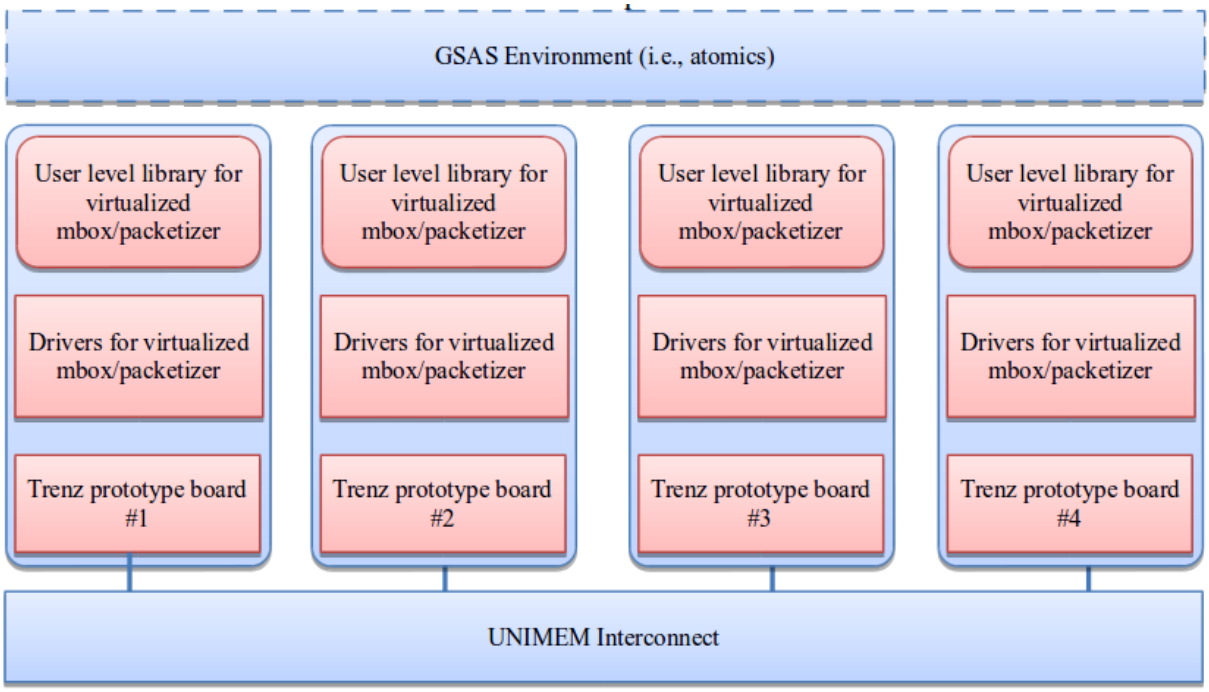


Figure 7 Hardware and Software Stack for the GSAS Environment.

3.1.1 Hardware Components (FORTH)

We first discuss the hardware components that are the major communication components of the GSAS environment. A thread in order to be able to issue an atomic request on the GSAS environment, i.e., to send a packet describing the atomic operation to some remote node and thus to some remote atomic service, it should be possible to allocate one interface of the local virtualized mailbox and one interface of the local virtualized packetizer. By using the allocated interfaces of the local mailbox and the local packetizer, the thread has the ability to send network packets (using the packetizer interface) describing atomic requests and receive responses (using the mailbox interface) for the issued requests. More specifically, a thread is able to send packets of size of 256 bits to an interface of some remote or local mailbox by using the allocated packetizer interface. Moreover, the issuer of the atomic operation is able to receive the response send by the atomic service to the allocated interface of the local virtualized mailbox.

The first interface of the virtualized mailbox starts at an address with suffix 0x0000 and it is only used by the atomic service of the node that the mailbox resides on. All other interfaces can be used by applications and they are allocated and de-allocated by the atomicity driver. Therefore, at most 63 threads are able to perform operations on the GSAS environment to each node. Each interface occupies 4096 bytes in address mapping, which is equal to operating system's (Linux) page size. Thus, it is possible for the operating system to safely map one interface of the virtual mailbox to the address space of exactly one thread. This kind of mapping allows threads to perform user-level accesses to the component.

The virtualized packetizer is another important hardware component for the GSAS environment. The virtualized packetizer allows the user threads to **atomically** send packets of 256 bits to any interface of any local/remote virtualized mailbox without having to make a memory mapping of the remote mailboxes to their address space. It is worth noting that by

avoiding to map remote mailbox interfaces directly to a thread's virtual address space, we disallow a thread to read the contents placed there by other threads. This and the ability to send packets to remote mailbox interfaces only through the packetizer enables protection on the data stored in remote mailboxes. Moreover, each packetizer interface adds a unique prefix to the transmitted packet indicating the sender thread. The atomicity driver, which is a kernel entity, sets up this prefix and thus, it is a trusted entity. This gives the ability to the atomic service to identify in a secure way the sender of the packet and thus, to decide if the atomic operation that is requested is either valid or not. Therefore, the packetizer component is not only necessary for atomically transmitting, but also for providing the appropriate features to the atomic service for secure detection the source of the transmitted packets.
Similarly to the virtualized mailbox, virtualized packetizer is equipped with 64 interfaces. The first interface of the virtualized packetizer starts at an address with suffix 0x0000 and similarly to the virtualized packetizer, it is only used by the atomic service. All other interfaces used by applications and they are allocated and de-allocated by using the functionality that the atomicity driver provides. Each interface occupies 4096 bytes in address mapping, which is equal to operating system's (Linux) page size. The virtualized packetizer is also equipped with an extra interface that is accessible only by the atomicity driver. This is the 64^{th} interface and starts at an address with suffix 0x40000. The functionality of this interface provides the ability to the atomicity driver to set up a unique identification number for each running thread in the whole system. This identification number is added as a prefix to each transmitted packet giving the ability to the receiver thread (i.e., any remote atomic service) to securely identify the origin of the packet.

3.1.2 Software Components (FORTH)

We now describe the main software modules that are used on the GSAS environment. These software modules are the following (see Figure 7 for the whole hardware and software stack used by the GSAS environment).

1. The atomicity driver.
2. The atomic service.
3. The software library that initiates atomic operations.

We first describe the role of the atomicity driver in the GSAS environment. The atomicity driver is responsible for distributing the appropriate hardware resources to applications' threads. The role of the atomicity driver is to grant one out of the 63 interfaces of the virtualized mailbox and one out of the 63 interfaces of the virtualized packetizer to each system thread that wants to perform atomic operations on the GSAS environment. At the first time that a thread that wants to use the functionality of the GSAS environment, it allocates one interface of the virtualized mailbox and one interface of the virtualized packetizer. Afterwards, the thread by using the functionality of the library that initiates the atomic operations is able to use the allocated interface in order to perform atomic operations. The atomicity driver guarantees that each thread owns at most one atomic interface of the virtualized mailbox and at most one interface of the virtualized packetizer. Furthermore, by setting the appropriate memory mappings the atomicity driver guarantees that each thread is not able to access the hardware resources (i.e., the interfaces of the virtual mailbox or the interfaces of virtual packetizer) of threads that are spawned by different processes. As it was

already pointed out, the atomicity driver set ups at the initialization of the GSAS environment, one globally unique identification number on each of the interfaces of the packetizer. This gives the ability to the atomic service that runs on some node to safely distinguish which thread issues any atomic request.
The main role of the atomic service is to serve the requests that are delivered on its local mailbox for the part of the address space that is responsible for. More specifically, the atomic service polls the interface 0 of the local mailbox until a packet that describes an atomic request arrives. Whenever such a packet arrives, the atomic service decodes the request, applies the described operation if it is valid (i.e., the target address and the operation code are valid, and the issuer thread has the appropriate access rights to perform the operation). Afterwards, the atomic service replies to the issuer by writing the response of the operation in the issuer's mailbox.
Apart from applying atomic operations on the address space of the GSAS environment, atomic service is responsible for servicing requests for memory allocation and for spawning new processes. It is noticeable that whenever no request is pending on a node for a long time, the atomic service of this node enters to *sleep* mode (i.e., gets the lowest priority among the other threads running on processing core 0) returning most of the processing resources to the other running processes. Whenever a new request is received, the atomic service exits from sleep mode. By following this kind of policy for sharing processing resources on core 0, in case that the address space of some node is not used, the atomic service of this node negligibly affects the performance of the other running applications.
Lastly, we describe the role of the user-level library that is responsible for initiating atomic operations. Whenever, an application thread wants to perform an atomic operation (i.e., Read, Write, CAS, etc.), it calls the appropriate function of the user-level library in order to initiate the respective atomic operation. At first, this call checks if the system is appropriately initialized, i.e., one interface of the virtualized mailbox and one interface of the virtualized packetizer are allocated for the calling thread. In case that the environment is not appropriately initialized, the library initializes it through the atomicity driver. Afterwards, the atomic operation is encoded in a network packet and this packet is send to the atomic service on the appropriate remote node.

3.2 OS & Virtualization (VOSYS, FORTH)

In the Linux ecosystem, there are several open source and proprietary solutions to achieve virtualization. Among these, KVM has been chosen as virtualization-enabling solution in the context of ExaNoDe. KVM for ARM relies on the ARM virtualization extension which is an hardware IP that implements all the hardware-related support required by virtualization. Such hardware features are accesses by the host OS via a Linux kernel driver (KVM).

KVM for ARMv8 (architecture of the target platforms) is part of the upstream (the official) Linux version and is maintained by the community. However, it was not enabled by default in the target boards configuration. In this context, VOSYS tested KVM support on the Trenz board, which required an ad-hoc kernel configuration for enabling the KVM module.

The use of virtualization as key component to improve reliability and manageability of the system makes more challenging the use of libraries that need to interface to physical resources. Among other things, virtualization introduces an additional layer that isolates the execution of the guest CPUs from the external devices, making the direct access to such devices not possible. Several solutions are available to tackle this problem, however, all of these add additional overhead that has to be quantified and kept to a minimum. As described in D3.4, an API remoting technology is being developed in order to enable the guest systems' processes to use the UNIMEM API, filling the gap introduced by the virtualization layer.

One of the key benefit related to the API remoting technique is about re-usability of the code already written: applications relying on UNIMEM do not require any adaptation to be run inside the virtual machine. From the functional perspective, this technique does not introduces any limitation and can be transparently integrated in the rest of the system. Being exclusively a software solution, the remoting technology can be easily integrated in the ExaNoDe prototype as it will affect mainly the virtualization layer being used (QEMU and KVM). The remoting technology will certainly introduce some overhead since some copy of data and processing is required by the transport layer (further details in D3.4). In Section 3.2 this overhead will be analysed and some direction to mitigate the platform's power consumption will be detailed.

3.3 Runtime Systems

A key aspect in providing a software environment on top of UNIMEM is to port and tune the runtime systems of the ExaNoDe programming models, taking into account the specificities of this architecture to optimize performance-critical aspects such as communication, synchronization and scheduling. This section first details the contributions of UOM on emulating UNIMEM atomics to allow early prototyping, porting and tuning of runtime systems for UNIMEM, as well as key advances in optimizing the performance and scalability of runtime algorithms. The last part of this section presents the contribution of FHG on porting and evaluating the GPI runtime system on UNIMEM.

3.3.1 Emulation of UNIMEM Atomics (UOM)

Porting OpenStream, runtime algorithms and data-structures to UNIMEM requires a controlled development environment where it is possible to distinguish between low-level bugs in the runtime system from bugs present in the prototype device drivers, as well as the capability to debug across multiple nodes. As this has not been possible on the prototype boards, relying on the emulation layer was the most efficient approach. However, many low-level algorithms for synchronization and communication require support for atomic operations and remote memory allocation, as well as the capability to emulate multiple boards and to dynamically deploy new boards. As mentioned in D3.1, these features were missing in the original emulation environment, which would not allow porting OpenStream to UNIMEM until the prototype is fully stable.

UOM has therefore developed an emulation layer that provides the missing functionality, that is fully compatible with the UNIMEM API and inter-operable with the RDMA emulation developed by FORTH. This emulation layer is designed to facilitate debugging and to allow an early evaluation of the impact of some key performance aspects, such as the latency of atomic operations across nodes.

As the emulation environment is meant to be used on desktop development machines, it is essential to ensure that despite emulating multiple nodes within a shared address space, the emulation environment does not allow shared-memory communication to occur by mistake. This is all the more important because the runtime system and algorithms being ported were originally designed to communicate through shared memory and this could otherwise have been silently ignored and only found out during deployment on the prototype. To this effect, each node is emulated by a separate process and the Global Shared Address Space is emulated using Linux shared memory segments for communication. Furthermore, to ensure that communication only occurs through the UNIMEM API rather than native memory accesses, the memory allocator only returns "tainted" addresses: invalid addresses which would give a segmentation fault if a program tried to access the location directly. Such tainted addresses can only be used within the emulator, which converts them back to valid addresses before performing any memory operations.

An added benefit of this emulation framework is that it allows to tune some important parameters in the early prototyping stages. In particular, most of the optimizations developed at UOM are sensitive to the latency of memory accesses and atomic operations. For this reason, the emulation of UNIMEM atomics allows to introduce artificial latencies and allows early detection of significant algorithmic design issues when porting concurrent data-structures or synchronization algorithms from shared memory to UNIMEM.

3.3.2 Locality-aware runtime support (UOM)

In this section, we discuss the performance-critical algorithms and data-structures used by the OpenStream runtime and which require special handling to ensure efficient operation on the UNIMEM architecture. We first describe the design of a state-of-the-art scalable load-balancing scheduler queue, then detail work on taking advantage of knowledge of machine-specific parameters (e.g., locality, latencies) to improve scheduling and memory allocation.

Hierarchical locality-aware Chase&Lev work-stealing

Concurrent deques (i.e., double-ended queues) are a key data structure in shared-memory parallel programming and play an essential role in work-stealing schedulers. Chase and Lev's algorithm is the current state-of-the-art, of which we previously offered the first proven implementation [4] that uses minimal atomic operations on relaxed memory consistency systems. The key idea of the Chase and Lev algorithm is that a worker thread only interacts with its own deque from one end (bottom in Figure 8) while the other worker threads can steal from the other end (top in Figure 8). The local worker can "put" and "take" tasks from the scheduler queue's bottom without requiring any atomic operation, only (remote) "steal" operations require atomics, when accessing the top of the deque. Under the hypothesis that

stealing only represents a small fraction of the total accesses to the scheduler deque, this delivers good performance and scalability.

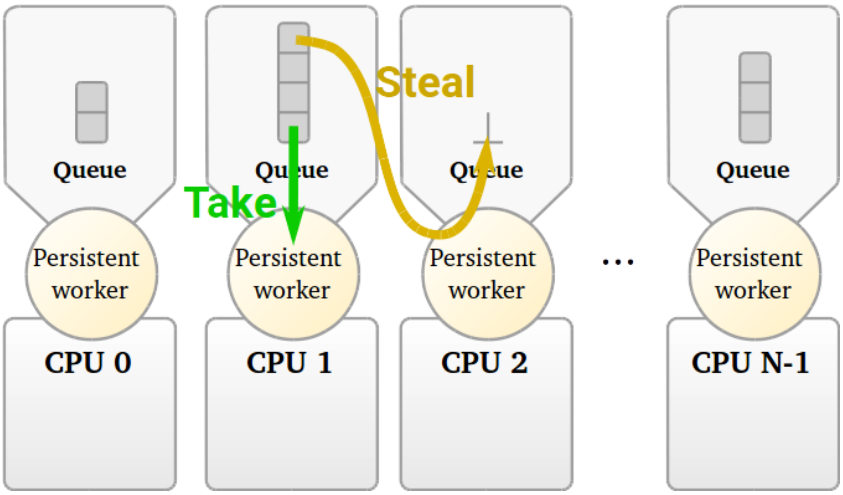


Figure 8: Load-balancing scheduler with work-stealing

However, this load-balancing scheduler queue comes with tradeoffs that have a substantial impact when porting to UNIMEM. These issues are essentially linked to the original design, which was targeting simple, uniform memory architectures offering sequential consistency. Porting this data-structure to UNIMEM requires addressing two main issues: the inefficient model for work discovery where every worker scans the scheduler queues of all other workers while looking for work (quadratic complexity and high potential for contention); and the impact of non-uniform intra-node vs. inter-node memory access and atomics latencies, which leads to an imbalanced access to work depending on node distance.

The first adjustment is necessary for scalability as OpenStream programs can be written without specifying work-distribution policies, which means that computation starts on one worker and is spread out across the machine through load-balancing. In this initial phase, random work-stealing means that all workers attempt to find work by polling any other worker in the system, which introduces a high amount of traffic on the interconnect. Correcting this issue has required a significant re-design of the original algorithm: (1) to adopt a hierarchical data-structure (Figure 9), separating inter-node load-balancing from intra-node operations; and (2) to switch from random work-stealing to a topology-aware variant [6] that further restricts traffic across the system, favoring local neighborhoods.

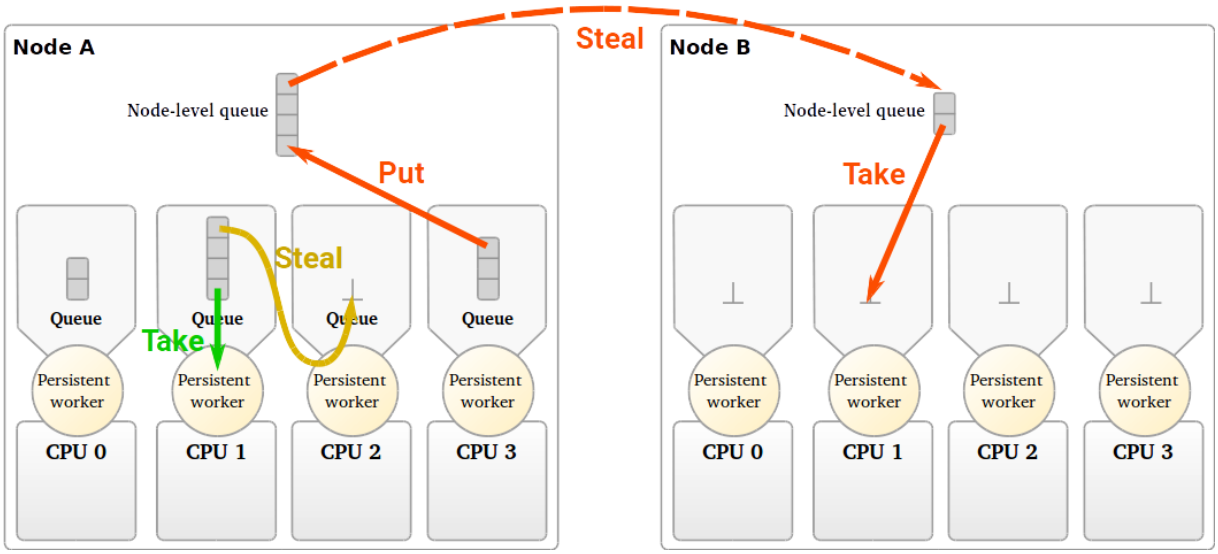


Figure 9: Hierarchical work-stealing

As shown in Figure 9, in addition to the worker threads' local work-stealing deque, a node-level deque is added which is where worker threads from remote nodes can steal work. As this structure's bottom is shared locally by all worker threads on the node and its top is shared by all nodes in the system, all operations must be synchronized with atomic operations. The illustration in Figure 9 corresponds to an imbalanced execution (e.g., when the program is starting and work has not yet been distributed), but the assumption remains the same as in the initial work-stealing algorithm, that the vast majority of tasks are not stolen and therefore expensive synchronized steal operations should not result in excessive overhead. This assumption has held for most non-trivial programs in our experiments with OpenStream.

In addition to this hierarchical data-structure, the choice of work-stealing victim can also be adjusted by defining neighborhoods of nodes, based on the topology of the machine or dynamically-defined, and attempting to steal from within a given neighborhood then progressively widening the search area. This has the advantage of allowing to reduce overall load-balancing traffic and reducing contention on shared data-structures, as well as allowing to map to potentially different levels of latency depending on the machine topology.

The second issue is due to the need to avoid using expensive atomic operations when polling remote nodes for available work prior to initiating a steal. Our previous work has shown that this is essential to maximize throughput in the load-balancing algorithm even on small-scale systems. However, this two-step approach can be problematic if the latencies of inter-node communication and atomic operations differ significantly between nodes and from intra-node latencies. Indeed, between the moment a victim has been identified and the effective acquisition of the work, workers that have a faster access can systematically acquire the available work before a slow-access worker has a chance. This hampers the distribution of work across the machine and can lead to very uneven distribution of work, reducing the effective parallelism exploited on the machine. This issue is addressed by introducing an additional work distribution mechanism, called work-pushing, which we detail below.

Locality-aware load-balancing scheduling and memory allocation

Dynamic task-parallel models are increasingly popular programming approaches for large-scale computing as they promise enhanced scalability, load balancing and locality. Yet these promises are undermined by non-uniform memory access (NUMA), a common feature of any large-scale system. In [2], we have shown that using NUMA-aware task and data placement, it is possible to preserve a uniform abstraction of both computing and memory resources for task-parallel programming models while achieving high data locality. This is achieved through a comprehensive strategy for memory allocation and work placement, and which we have implemented in OpenStream. The core of this strategy is composed of three techniques that complement the work-stealing load-balancing algorithm presented above:

1. Implicit privatization of all data produced by tasks, which is the default behavior in OpenStream, ensures that the runtime system has full control over memory management. This is also one of the key properties used to port OpenStream, which was initially a shared-memory programming model, on top of UNIMEM RDMA.
2. *Deferred allocation* is a technique that delays the allocation of memory for writing the outputs of a task until it starts executing. This is necessary because in the presence of dynamic scheduling, a task can be stolen or moved at any point until it starts executing. If its memory affinity (i.e. the location of the data it will access) is fixed too early, this can lead to sub-optimal runtime behavior. By delaying memory allocation until execution, we guarantee that all data produced will be written locally.
3. *Topology-aware work-pushing* inspects the *input* dependences of a task to determine on which node it is the most suitable to be executed by computing a weighted average of the cost to transfer the data it requires for execution. This is illustrated in Figure 10 where the worker on CPU N-1 has decided that a newly ready to execute task has best affinity with CPU 0 and therefore *pushes* the task instead of putting it in its own queue. This requires adding an additional reception queue for work-pushing on each worker (and at node-level once the hierarchical refinement described for work-stealing is also implemented).

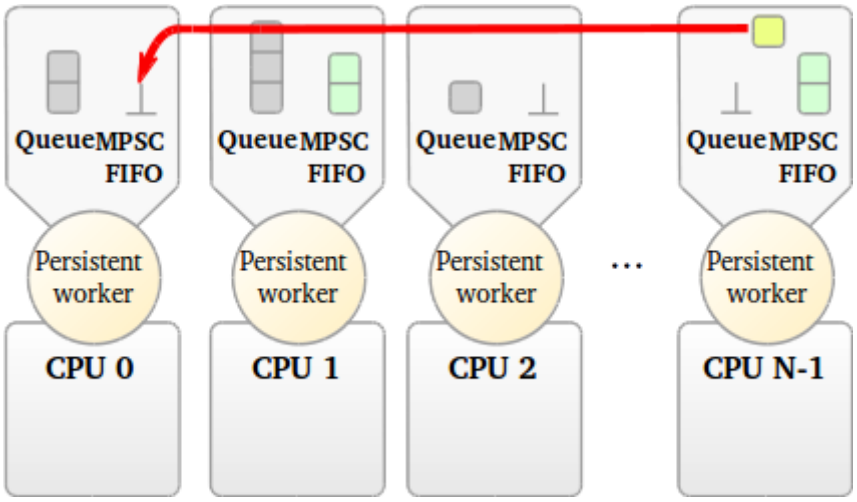


Figure 10: Locality-aware work-pushing

While work-pushing and work-stealing may appear to conflict in some cases, they are very complementary. Work-pushing can also include additional heuristics or user hints to accelerate the initial distribution of work. Our data placement scheme guarantees that all accesses to task output data target the local memory of the accessing core. The complementary task placement heuristic improves the locality of task input data on a best effort basis. These algorithms take advantage of data-flow style task parallelism, where the privatization of task data enhances scalability by eliminating false dependences and enabling fine-grained dynamic control over data placement. The algorithms are fully automatic, application-independent and performance-portable, automatically adapting to dynamic changes.

Placement decisions use information about inter-task data dependences readily available in the run-time system and placement information from the operating system or UNIMEM driver. Our experiments, so far with shared-memory systems up to 192 cores and 24 nodes, show that we achieve on average 94% of local memory accesses and up to 5× higher performance. We expect similar or better improvements in the case of larger systems because of a higher impact of locality when the memory latency of accessing remote nodes increases.

We have additional ongoing investigations into adapting previous work on restricting the reliance on atomic operations and mitigating their impact for software barrier synchronization [3] and FIFO queue communication [5]. We expect that the key benefits of the optimizations implemented for other forms of NUMA and for minimizing reliance on atomic operations will translate into significant efficiency gains on the UNIMEM architecture as well.

3.3.3 GPI runtime system (FhG)

GPI-2 is a PGAS-based runtime environment for distributed-parallel systems. GPI outperforms other parallel runtimes in almost all scenarios in terms of scalability and communication efficiency. To enable this kind of performance, all GPI-Building-Blocks (Figure 11) need to be optimized and highly tuned for the underlying transport hardware.

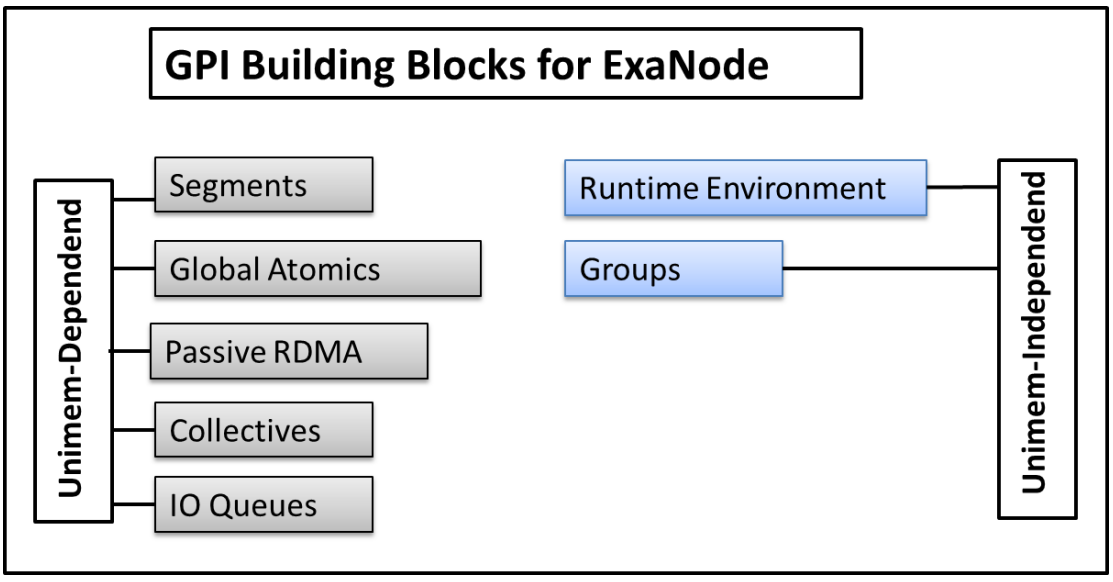


Figure 11: GPI Building-Blocks

In collaboration with Forth we co-designed a first and simple API for UNIMEM. This API served as a basis model for the implementation of a complex emulation-system on top of Infiniband, which was used to test the integration of GPI into the Umimem software environment. Today GPI is the first runtime that supports the defined UNIMEM API. In a next step different unit tests of the separated sub-modules will be implemented to improve the robustness of the implementation.

Tests on the UNIMEM Prototype Hardware

Parallel to the implementation on the emulation-system we started to test the UNIMEM API on the multi-board Prototype Hardware. The current test results are as follows:

- Multi-board prototype hardware is not stable and cannot yet be used for real-world tests
- Typical parallel startup mechanisms like mpi_run or gpi_run cannot be used
- Hand-started processes cannot communicate at all using the UNIMEM API
- The documented startup procedure (remoteFork) seems to implement some kind of process hierarchy/security which does not include other process-instances and has no functionality for environment variables and command line arguments as needed by e.g. mpi_run (undocumented)
- Strong size limitations for memory segments and communication calls

To overcome the current UNIMEM API limitations, we started to port the GPI-Submodules to a generic socket interface on top of UNIMEM. This workaround allows us to further improve the individual GPI-Submodules and to continue the developing process without time-consuming interruption and delays.

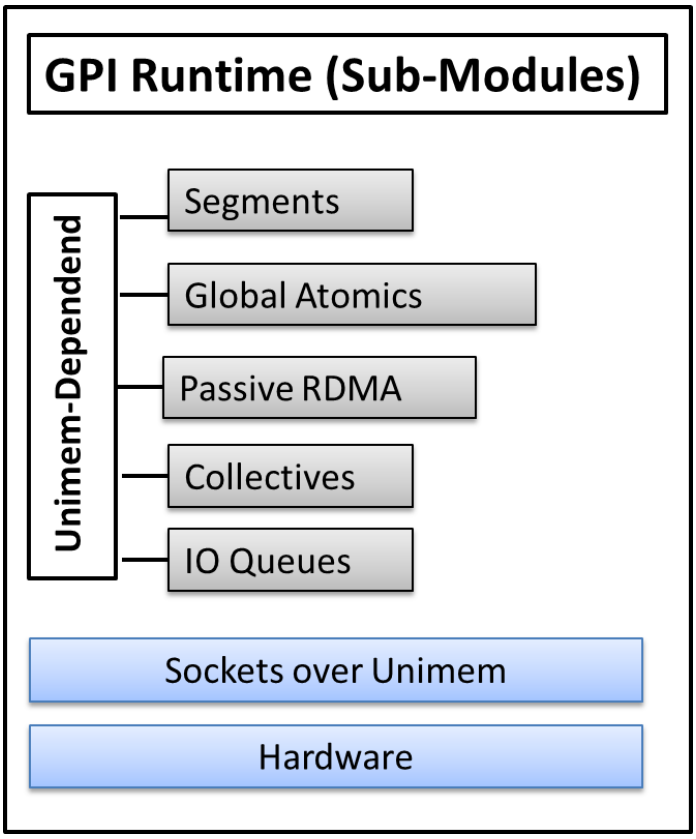


Figure 12**: GPI-Submodules on top of UNIMEM-Sockets**

Next steps

We have developed a small test-suite that can be used to test the UNIMEM API on the multi-board prototypes. This test-package was introduced to Forth in a live-session to show off the current UNIMEM API problems. The package is also installed on a network-share available to the Prototypes and can be used by all other developers as a starting point for own developments. Together with Forth we will work on the current issues to build up an optimized UNIMEM API for runtimes like MPI and GPI.

4 Tuning of platforms

4.1 High-speed low-latency inter-chip communication (FORTH)

The inter-chip communication between FPGA modules is supported by high-speed serial links based on SerDes transceiver hard blocks found in all high-performance FPGAs; high-end FPGAs may contain several tens of SerDes transceiver hard blocks. In some designs, SerDes transceivers are preferred over LVDS links because they have higher per-pin transfer rates (an order of magnitude) and thus require less pins for connections of the same bandwidth. Each SerDes transceiver hard block in our FPGA infrastructure (Xilinx UltraScale+ GTH[4]) supports rates of up-to 12.5 Gbps per lane per direction (2 TX and 2 RX pins). Although, these hard blocks have very high bandwidth, their latency depends on the configuration settings and the hardware controller (IP block implemented in FPGA logic) that drives them.

The typical IP block offered by Xilinx is called Aurora[5] and offers an AXI-Stream interface to the user logic. The Aurora IP block protocol uses 64B/66B encoding and introduces ~3% bandwidth overhead. The issue with Aurora IP is its high-latency for inter-chip communication. According to the Aurora IP manual, the latency from user-logic at the transmitter to the user-logic at the receiver ranges from 46 user clock cycles (minimum) to 55 user clock cycles (maximum); we have confirmed such latencies with simulations and tests on FPGA prototypes. For a 10.3125 Gbps link with a 64-bit datapath the user clock is 156.25 MHz (6.4 nanoseconds) and the one-way *latency ranges from 294 to 352 nanoseconds*. Consequently, the Aurora IP block is not suitable for low-latency inter-chip communication.

4.1.1 Low Latency SerDes Link IP (FORTH)

During the tuning efforts in task T5.2, FORTH has designed and implemented a streamlined *low-latency SerDes Link* IP with a custom protocol to achieve link latencies *below 100 nanoseconds*. The IP block customizes the existing Xilinx UltraScale+ GTH transceivers, offers the same AXI-Stream interface to user-logic, and uses a 64B/66B encoding scheme for data transmission similar to typical protocols.

Data Encoding

One important issue with all multi-gigabit transceivers and serial links is the encoding of data. The serial links are plesiochronous and the transmitters do not send their clock (via a dedicated pin) with the data, as in source-synchronous systems, but rather encode the clock "edges" in the actual data with bit transitions from 0 $\rightarrow$ 1 and from 1 $\rightarrow$ 0. The receivers perform clock-and-data recovery (CDR) and require frequent "edges". Some systems use 8B/10B fixed encoding which guarantees at least on "edge" every ~5 bits but the overhead is 20%. Modern CDR circuits are more immune to infrequent "edges", have higher bit-transition density margins, and can operate with an "edge" every ~100 bits. Several protocols (e.g. 10G

4 Xilinx Inc. Ultrascale Architecture GTH Transceivers User Guide (UG576)
5 Xilinx Inc. PG074 - Aurora 64B/66B v11.2 Product Guide

Ethernet) take advantage of these improvements and use encodings like 64B/66B which have only ~3% overhead.

Our IP block uses a custom 64B/66B encoding scheme where the two extra bits (called header-bits) provide framing and control information and guarantee at-least one "edge" every 66-bits. To generate more "edges" from the transmitted data, we use the scrambler/descrambler polynomial defined by 10G Ethernet: [$G(x) = x^{58} + x^{39} + 1$] which has very low area overhead. The encoding logic incurs only a single user clock cycle of latency at the transmitter (to scramble) and a single user clock cycle at the receiver (to descramble). For 10.3125 Gbps links the sum of scrambler and descrambler latency is 12.4 nanoseconds.

The encoding of header-bits is as follows:

- *Data (2'b01)*: indicates that the word is part of a frame/stream transmission.
- *Data End-of-Frame (2'b11)*: indicates the last word of a frame/stream transmission.
- *Control (2'b10)*: indicates that the next 64-bit word is user-defined information. We allow user logic to define and transmit custom control words of 64-bit. In our reference implementation, we have reserved 1-bit from the 64-bit control word to indicate IDLE, i.e. no transmission. Moreover, we have defined other control words for pause-based flow-control (XON/XOFF) and credit-based flow-control.
- *Reserved (2'b00)*: This value is reserved and considered an invalid header value. This value should not appear in a correct frame/stream and indicates transmission errors and/or that the link is down.

Based on the above header-bit encodings our IP block easily constructs framed data transmissions from the AXI-Stream user interface signals at the sender, and reconstructs the AXI-Stream user interface signals at the receiver. There is almost one-to-one correspondence of the "valid" and "last" AXI-Stream signals to the header-bits – "valid" drives header-bit 0 and "last" drives header-bit 1 at the sender and vice-versa at the receiver; our logic that handles this conversion has zero latency.

Moreover, our IP features an additional control port interface to allow the user-logic to transmit and receive user-defined control words, e.g. flow-control information. This control port is 64-bit wide with "valid" and "ready" signals. The transmission of control words takes precedence over the AXI-stream interface and may occur anytime. However, to ease fast-packet transmission (via the AXI-Stream interface) and enable cut-through operation without "hiccups", our reference design enables (acknowledges) control word transmission only on packet/frame boundaries.

Low Latency GTH Transmit and Receive Paths

To achieve low-latency in both the transmitter path and the receiver path we have carefully configured and optimized the internal paths in the GTH transceivers that are used by our custom link controller (IP block in the FPGA logic). In order to decide for the optimal configuration of the GTH transceivers, we have studied the latency of each internal

component using latency information by Xilinx[6]. The latency values are typically measured in unit intervals (UIs). The UI is defined as the time needed to transmit one bit and depends on the link-rate, e.g. for a 10 Gbps link the UI is 100 picoseconds and for a 12.5 Gbps link the UI is 80 picoseconds.

Our analysis of transmit path latencies reveals that the "TX Asynchronous Gearbox" component, which is typically used in many GTH transceiver configurations, incurs a latency of 309-340 UIs (~30 – 33 nanoseconds for 10.3125 Gbps links). This block compensates for the 64-bit user-data to 66-bit link-data frequency difference. Supporting 10 Gbps user rate with 64B/66B encoding (3.125% overhead) over a 10.3125 Gbps link requires a user clock-frequency of 156.25 MHz where the GTH internal transmitter frequency is 161.13 MHz.

In our design, we opted to use the "TX Synchronous Gearbox" component that has a lower latency of 64 – 128 UIs (~6.2 – 12.4 nanoseconds for 10.3125 Gbps links). The impact of using this path is that the transmitter operates at 161.13MHz and *must* pause transmission every 32 words to compensate for the difference. Figure 13 presents the internal path that is used in our GTH transmitter configuration and shows in blue boxes the latency incurred by each component. The minimum total transmit latency with our configuration is 271 UIs while the maximum latency is 335 UIs. For a 10.3125 Gbps link, the minimum and maximum transmit path latency values are ~26.3 and ~32.5 nanoseconds respectively.

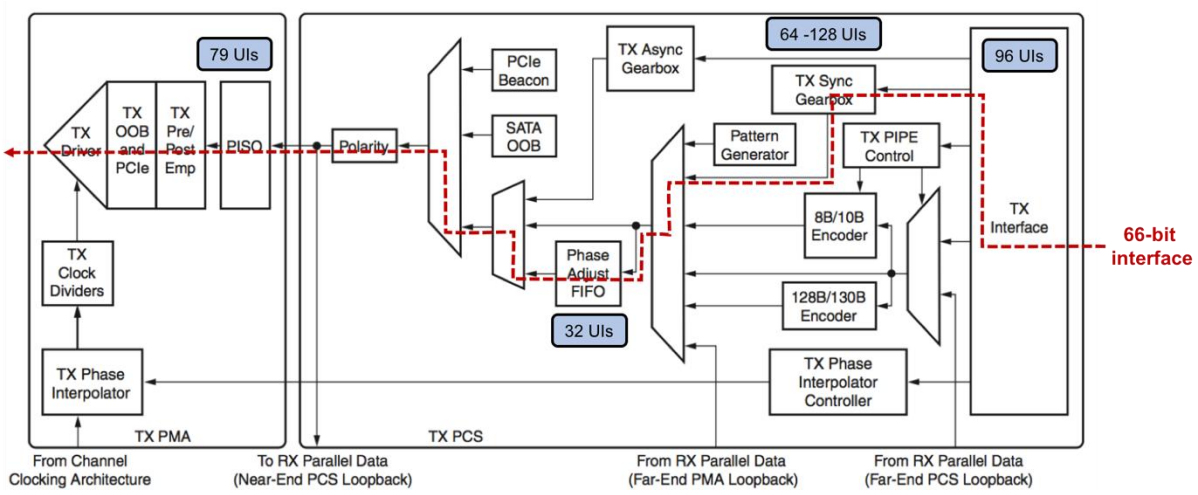


Figure 13: Customized transmit path in Xilinx GTH transceiver

Our analysis of receive path latencies reveals that the "RX Asynchronous Gearbox" component, which is typically used in many GTH transceiver configurations, incurs a latency of 348 UIs (~33.8 nanoseconds for 10.3125 Gbps links). This block compensates for the 64-bit user-data to 66-bit link-data frequency difference. Supporting 10 Gbps user data rate with 64B/66B encoding (3.125% overhead) over a 10.3125 Gbps link requires a user clock-frequency of 156.25 MHz where the GTH transmitter frequency is 161.13 MHz.

6 Xilinx Inc. AR# 64309 UltraScale GTH Transceiver: TX and RX Latency Values

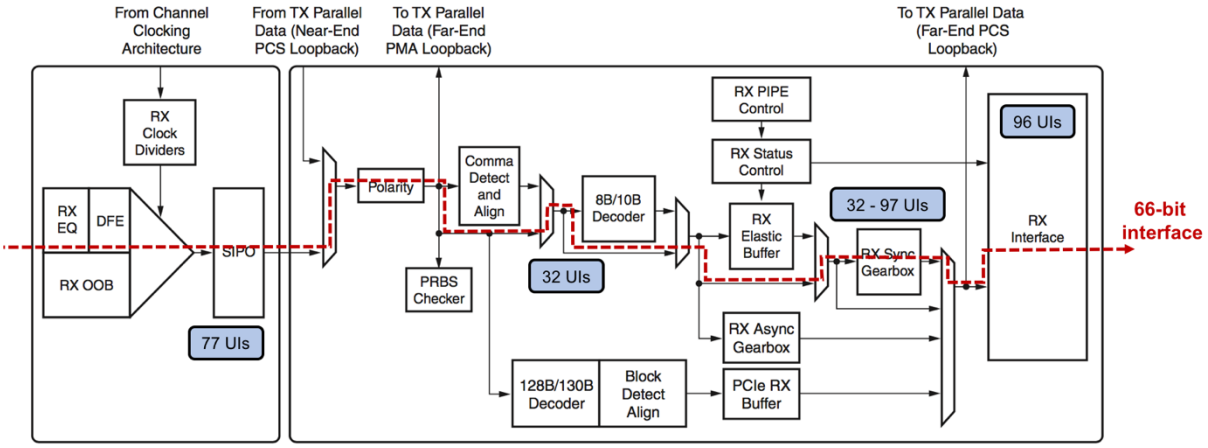


Figure 14: Customized receiver path in Xilinx GTH transceiver

In our design, we opted to use the "RX Synchronous Gearbox" component that has a lower latency of 32 – 97 UIs (~3.1 – 9.4 nanoseconds for 10.3125 Gbps links). The impact of using this path is that the receiver operates with the recovered transmitter clock, at 161.13MHz, and every 32 words one invalid words appears to the interface in order to compensate for the difference. Figure 14 presents the internal path that is used in our GTH receiver configuration and shows in blue boxes the latency incurred by each component. The minimum total receive latency path with our configuration is 237 UIs while the maximum latency is 302 UIs. For a 10.3125 Gbps link, the minimum and maximum receive path latency values are ~23 and ~29.2 nanoseconds respectively.

Clock Correction and Receive Interface

In serial link connections, the transmitter and the receiver have the same nominal clock frequency, but their clock source is coming from different crystal oscillators and have inevitably small differences in frequency and jitter (typically 100 ppm – parts per million) which effectively means that clock phases drift. In such plesiochronous systems the sender and receiver need to compensate for such frequency and phase differences. The typical technique used in such systems is to periodically transmit in the link "clock-correction" sequences. These clock correction sequences are used by the receiver to avoid overflows and underflows. The receiver may discard a "clock-correction" sequence to avoid overflow (the transmitter has faster clock) or duplicate one sequence to avoid underflows (the transmitter has slower clock). The periodicity of clock correction sequences depends on the maximum allowed clock frequency difference and the link-rate and often times limits the maximum packet size. The current GTH transceiver supports clock correction only when 8B/10B encoding is used and when the "elastic-buffer" component is enabled (the elastic buffer has at least 230 UIs latency) thus, we follow a custom approach.

Our approach is to operate the back-end part of the receiver FPGA logic with the recovered clock (transmitter's clock) and enqueue the link-data to an asynchronous FIFO queue. The queue at the receiver is already there for flow-control and we exploit it to avoid overflows due to clock differences. Moreover, the FIFO enables the receiver and the transmitter user-logic to operate with a common clock (AXI-Stream clock) as most designs do. The link-data are

enqueued to the FIFO by the back-end FPGA logic with the recovered clock and dequeued from the FIFO with the user-clock (AXI-Stream clock) from the front-end FPGA logic.

Typical asynchronous FIFOs have long latencies that range from 3 to 7 clock cycles. However, our design exploits the fact that the design is plesiochronous and optimizes for this case. We have developed a plesiochronous FIFO that uses a variant of the Even-Odd Synchronizer[7]. Our plesiochronous FIFO has a minimum latency of ~0.6 clock cycles and a maximum latency of ~1.6 clock cycles while the average is ~1.1 clock cycles. For a 10.3125 Gbps link the average plesiochronous FIFO latency is ~6.8 nanoseconds while the maximum latency is ~10 nanoseconds.

Total Latency

The total transmit latency of our IP block is the sum of the scrambler (encoding) latency and the GTH transmit path latency. For a 10.3125 Gbps link, the total minimum transmit latency is ~32.5 nanoseconds and the total maximum transmit latency is ~38.7 nanoseconds. The total receive latency of our IP block is the sum of the GTH receiver path latency, descrambler (decoding) latency, and the plesiochronous FIFO latency. For a 10.3125 Gbps link, the total minimum receive latency is ~39.2 nanoseconds and the total maximum receive latency is ~45.4 nanoseconds.

The total inter-chip latency from the transmitter's user logic (AXI-Stream) to the receiver's user logic (AXI-Stream) is *~72 nanoseconds at minimum* and *~84 nanoseconds at maximum*. Compared to the Aurora IP block, our SerDes Link IP *yields four times lower latency* with total *one-way latency under 100 nanoseconds*, which makes it suitable for low-latency inter-chip communication.

4.2 Low-power techniques (VOSYS, FORTH)

The overhead introduced by the API remoting technique and virtualization in general directly translates to a higher power consumption. This section will detail the directions that are being explored to reduce the virtualization overhead and thus the power consumption.

Related to the API remoting, two different synchronization methods are envisioned for the backend's thread involved in the handling of the guests' API calls. The first one, based on spinlocks, ensures the best performance since expects the threads to constantly poll on a shared memory – shared between host thread and guest – regulated by the spinlock. This solution, although ensuring the lowest latency to serve a guest's API call, is not power-efficient. As an alternative solution, a Virtio-based mechanism has been implemented, relying on a much more efficient approach based on asynchronous notification delivery. The two

7 W. J. Dally and S. G. Tell, "The Even/Odd Synchronizer: A Fast, All-Digital, Periodic Synchronizer," 2010 IEEE Symposium on Asynchronous Circuits and Systems, Grenoble, 2010, pp. 75-84.

solutions are not mutually exclusive, and can co-exist in the platform to guarantee the best compromise between power consumption and performance.

To further increment the efficiency of the API remoting solution, a direct attachment of an hardware mailbox to the virtual machine can be considered. By configuring the host MMU is such a way to access the device directly from the virtual machine, the virtualization layer can be skipped at a negligible price of an additional MMU translation stage. The power consumption resulting from this solution is the same as not having virtualization at all since the device will be operated directly by the guest Linux driver.

Virtualization however does not go only against power consumption of a system: it introduces benefits like virtual machine snapshot creation and incremental checkpointing that have been explored in the context of ExaNoDe. This features can have interesting applicability in the context of efficiently manage the system resources. When a virtual machine does not need to perform any computation, this can be suspended and, in case, a snapshot can be created. This will allow to restore the virtual machine when needed, avoiding the pointless idling of the guest system.

4.3 HW sensors monitoring infrastructure (ETHZ, FORTH)

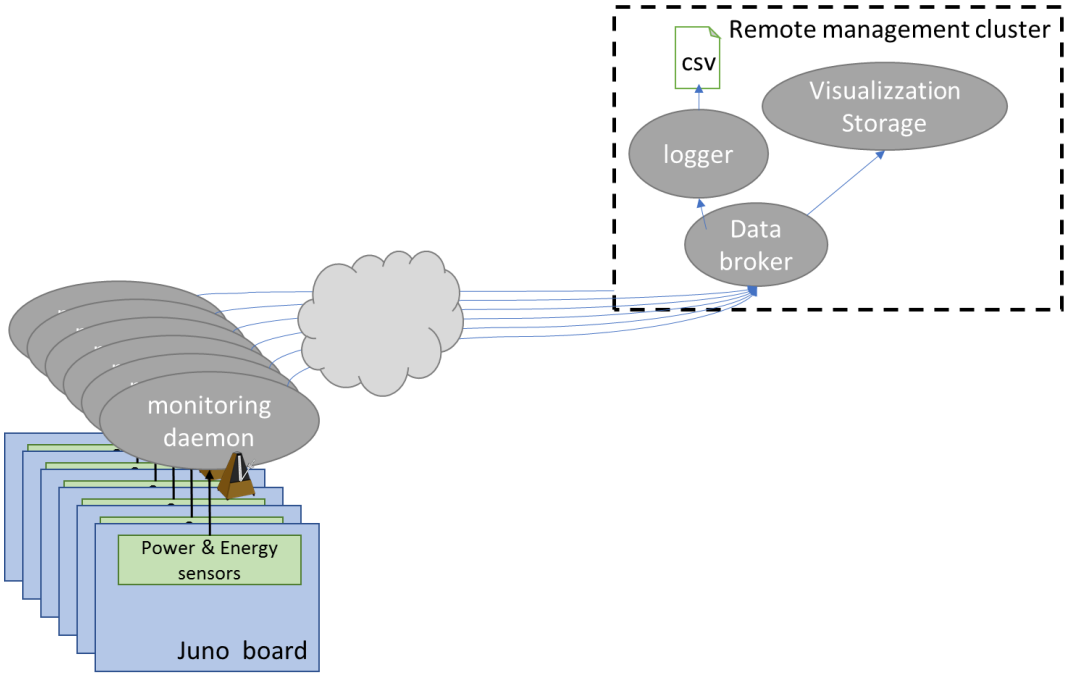


Figure 15: Scalable monitoring framework

Figure 15 shows the skeleton of the target monitoring framework which has been developed by ETHZ. The framework can be adapted to inspect several physical parameters by periodically reading available sensors and propagating them to the network thanks to a lightweight communication protocol based on sockets and out-of-band communication. Thanks to that the monitoring framework can be executed in multiple nodes and share the monitored events to a centralized entity which organizes the monitored values in topics and

forward them to a list of interested subscribers. The subscriber could be a data logger as well as a data visualization front-end.
ETHZ has been working on implementing the scalable monitoring framework for monitoring and profiling the power and energy consumption of the devices. The framework is composed by several software components:

1. A monitoring daemon which periodically monitors the energy and power consumption of the target device reading the built-in hardware senosors. It takes a timestamp and send these values to the network to a remote data collector
2. A data broker which receives the data from multiple sources organizes them in topics and propagates the messages received to a list of subscriber to the specific information topics. Topics are organized accordingly to measurement sources and types.
3. A logger than subscribes to the power and energy measurements and saves them to a csv file.

Current activities have focused on the creation of the data broker and logger. Future activities will focus on the monitoring daemon as well as visualization front-end.

5 Concluding Remarks

This deliverable has described the initial integration of the software and hardware environments of the ExaNoDe project and the tuning of the different components to improve the efficiency of interactions. Each of the key components is described in detail, together with a preliminary overview of the challenges of running a complete, high-level software stack (GPI) on the multi-board prototype and to take advantage of the unique characteristics of the UNIMEM architecture. The final integration and tuning will be documented and delivered in D5.4 at month M36.

The runtime systems and communication libraries have been prototyped and developed using remote access to the multi-board prototype hosted at FORTH in Crete, which provides functional verification on real hardware. A software emulation of the UNIMEM APIs, using a software layer provided by FORTH and UOM, provides the ability to perform substantial development work on a local machine.

6 References and Applicable Documents

[1] Nikolaos D. Kallimanis et al, D3.6 Design of the ExaNoDe Firmware, ExaNoDe project deliverable D3.6, 2016.

[2] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, Nathalie Drach: Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. PACT 2016: 125-137.

[3] Andrey Rodchenko, Andy Nisbet, Antoniu Pop, Mikel Lujan: Effective Barrier Synchronization on Intel Xeon Phi Coprocessor. Euro-Par 2015: 588-600.

[4] Nhat Minh Le, Antoniu Pop, Albert Cohen, Francesco Zappa Nardelli: Correct and efficient work-stealing for weak memory models. PPOPP 2013: 69-80.

[5] Nhat Minh Le, Adrien Guatto, Albert Cohen, Antoniu Pop: Correct and Efficient Bounded FIFO Queues. 25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2013).

[6] Drebes A, Heydemann K, Drach N, Pop A, Cohen A. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. ACM Transactions on Architecture and Code Optimization (TACO). 2014 Oct 27;11(3):30.