D3.6

Design of the ExaNoDe Firmware

Workpackage:	3	Enablement of Software Compute Node
Author(s):	Nikolaos D. Kallimanis	FORTH
	Manolis Marazakis	FORTH
	Antonis Psathakis	FORTH
	Babis Aronis	FORTH
	Dimitris Poulios	FORTH
	Nikolaos Chrysos	FORTH
Authorized by	Manolis Marazakis	FORTH
Reviewer	Etienne Walter	BULL
Reviewer	Dirk Pleiter	JUELICH
Reviewer	Francis Wray	SCAPOS
Dissemination Level	PU	

Date	Author	Comments	Version	Status
2016-08-31	N. D. Kallimanis	Description of the GSAS environment	V0.1	Draft
2016-09-03	A. Psathakis	Description of Hardware to Software Interface	V0.2	Draft

2016-09-10	N. D. Kallimanis	Document revision	V0.3	Draft
2016-09-12	B. Aronis	Description User-Initiated RDMA	V0.4	Draft
2016-09-14	D. Poulios	Description of sockets over RDMA	V0.5	Draft
2016-09-16	N. D. Kallimanis	Document revision	V0.6	Draft
2016-10-04	N. D. Kallimanis	Introduction to Unimem architecture	V0.8	Draft
2016-10-06	M. Marazakis	Document revision	V0.9	Draft
2016-10-07	N. D. Kallimanis, M. Marazakis	Document revision	V1.0	Release to ExaNoDe reviewers
2016-10-26	N. D. Kallimanis, M. Marazakis	Revision based on comments by Etienne Walter, Dirk Pleiter and Francis Wray	V1.1	Draft
2016-10-27	N. D. Kallimanis, M. Marazakis	Final edits, inclusion of an overview figure	V.1.1	Final

Executive Summary

In this deliverable, we describe the firmware and operating system support that we have developed to support the Unimem architecture on the current-generation ExaNoDe multiboard prototype. We present our design and implementation of a global shared address space (abbr. GSAS environment) and its communication mechanisms. We describe the GSAS architecture in detail, while giving a brief overview of the hardware and software components that it is based on. We also provide a description of the hardware-software interface of the hardware components co-designed for use by the GSAS environment.

We also describe two additional building-block capabilities available in our prototype: user-space initiated DMA and mailbox notifications. The DMA engine of the underlying system is capable of transferring to/from any memory location throughout the entire global memory space. A main goal of our effort is to efficiently and conveniently expose the functionality of the DMA engine to applications running in user space. Furthermore, we describe the custom mailbox mechanism with which a kernel- or user-space application can send and receive messages to and from remote nodes, thus offering a low-latency remote notification capability.

Finally, we provide an overview of the Sockets over RDMA feature. With Sockets over RDMA, a Unimem system can utilize low-latency communication among local nodes, by means of fast RDMA transactions, bypassing the kernel-space network stack.

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The Unimem Architecture plays a central role in the ExaNoDe project, and also has been the basis for related projects, i.e., EuroServer (1) and ExaNeSt (2). The Unimem architecture consists of a powerful set of mechanisms that provide efficient communication among the remote nodes of a large computational system. The main advantage of the Unimem architecture over conventional communication architectures (i.e., coherent shared memory systems and message passing computational systems), is that it offers more advanced communication mechanisms than the conventional message passing systems and eliminates the complexities, the performance overheads, and the costs that the large coherent shared memory systems induce.

The Unimem architecture, is a technology that was first developed within the EuroServer project (1), (3). A computational system that implements the Unimem architecture consists of a set of computational nodes that are connected through a custom network. Each computational node consists of a set of processing cores, which communicate among each other using some coherent shared memory protocol provided by the hardware. Unimem enables the nodes to directly access areas of memory located in remote nodes. More specifically, in the Unimem architecture, there is a global address space (abr. GAS) that it is accessible to any node inside the computational system. The local physical memory of each node is mapped to a portion of the GAS. Therefore, any node in the system has the ability to directly access the physical memory of any other remote node through the GAS. In order to eliminate the complexity and the costs that the system-level coherence protocols induce (4), the Unimem architecture imposes that each page of the physical memory can be cached by at most one node (see Figure 1 for such a use-case). In principle, the node that caches a page of memory can be the local node where this page is physically allocated or any other remote node. However, in practice, it is generally preferable that nodes do not cache remote memory pages.

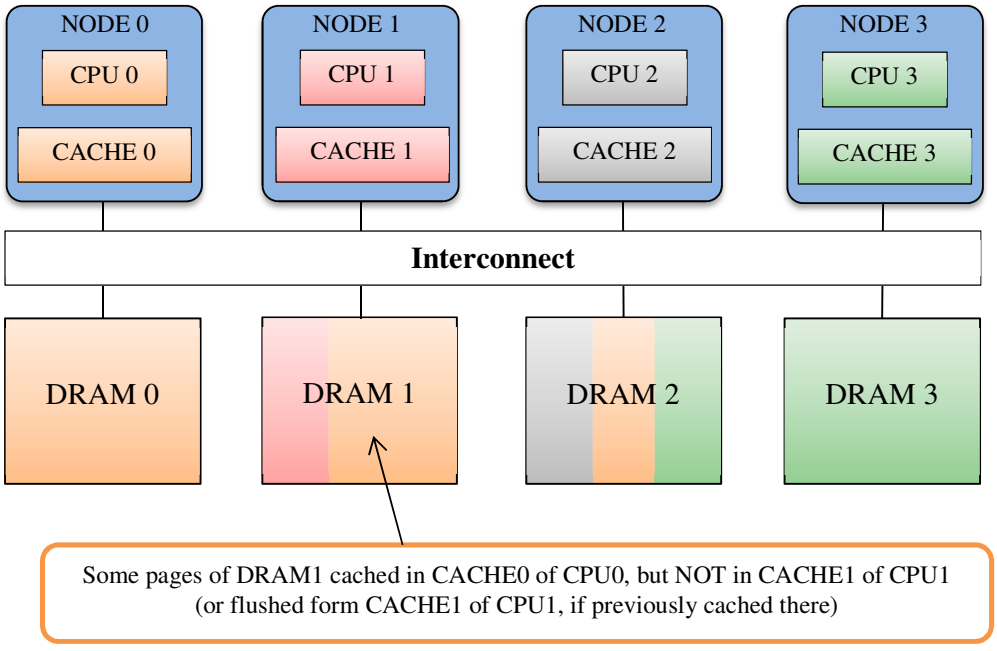

Figure 1 Overview of the Unimem architecture.

The most notable characteristics of the Unimem architecture are the following:

1. Load and store instructions are allowed across remote nodes, to any address within the GAS. Thus, any node of the computational system is able to access any part of the memory of any remote node via conventional load and store instructions (see Figure 1).
2. Every page of physical memory can only be declared *cacheable* in a single node at most, this node is called *owner node*. The owner node is usually the node whose local memory contains this page of memory, but it could also be any remote node that uses the page by borrowing memory from another node (see Figure 1).
3. Unimem provides the ability of efficiently copying large amounts of memory from/to remote nodes. This is achieved by using the Remote Direct Memory Access (RDMA) block transfers. This is a communication mechanism supported by the hardware and enables efficient zero-copy transfers of large portions of data. The Unimem architecture provides the ability that an RDMA block transfer can be initiated at user-level in a protected way, without paying the overhead of a system call. It thus drastically reduces the latency and the energy consumption of communication across remote nodes (see Figure 2). It is remarkable, that this type of communication has the advantage that it is performed by a separate, dedicated hardware (DMA) engine, which executes communication primitives in parallel with the main processor performing other, overlapping computations.
4. Unimem also offers the mailbox hardware primitive, which give processes that reside on remote nodes the ability to receive remote notifications. By using mailboxes, processes are able to send/receive synchronization mechanisms to/from processes that reside on remote nodes. The hardware gives processes the ability to access the mailboxes via user-level library calls in a protected way, without paying the overhead of a system call.

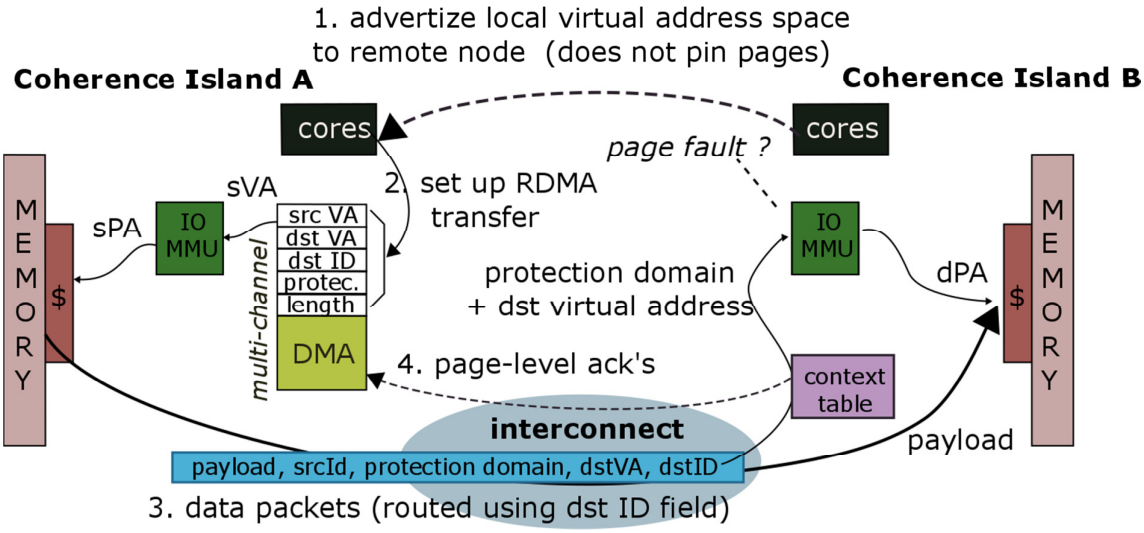

Figure 2 Overview of an RDMA Operation

Owing to these, Unimem behaves as an evolution of both shared memory and message passing parallel architectures:

- Shared Memory: Since all memory words within the entire GAS are accessible by any node using conventional load and store instructions, Unimem is a shared memory system. However, we avoid the high cost of system-wide hardware cache coherence protocols, by requiring that every memory page can only be cached within a single node at most. Thus, all the other remote nodes may also enjoy consistent accesses to the data by employing un-cached, remote memory accesses. This kind of accesses is more expensive, since it is un-cacheable. However, this cost is acceptable provided that remote accesses are infrequent, which is Unimem's response to the often made observation that "cache coherence is nice to have, provided you do not frequently use it".
- Message Passing: The Unimem architecture allows making bulk data transfers directly into the receiver's memory, i.e., zero-copy RDMA. Thus, additional copies of the data induced by conventional message passing systems that do not support this feature are avoided. In this way, this type of data copying (i.e., RDMA) becomes the multi-word generalization of (pipelined, posted) remote store instructions. Recall that this type of communication is performed by a separate, dedicated hardware (DMA) engine, giving the main processor the ability to execute other, overlapping computations. Therefore, system's performance is enhanced in terms of time and energy.

The Unimem architecture is already implemented in a system consisting of a few ARM-based micro-servers (i.e., nodes) designed and prototyped by the ongoing EuroServer project (FP7-ICT-610456, http://www.euroserver-project.eu). Each node is based on the Juno ARM development platform (5), and it has the following key properties:

- It consists of several processing cores (up to 6), which all of them consisting of a coherence island. Communication among the processing cores of a coherence island is performed through a coherent shared memory protocol provided by the hardware of the processor. Nodes are able to share I/O devices and accelerators that are attached to any other remote node resulting to better I/O performance and flexibility.
- There is a partitioned global address space (abr. PGAS), consisting of the aggregation of the physical memory of multiple nodes (i.e., coherence islands), where each memory page has a single owner. Thus, each node owns a part of the PGAS and the whole of its local physical memory can be accessed by any remote node through the PGAS. A processor of any node can access any page of the PGAS, by issuing conventional load and store instructions, which are transparently routed by the hardware to the appropriate node that the memory resides on. This is achieved by adding non-trivial extensions to the processor's data-path that can only be implemented in an open platform.
- Since we aim to implement an extremely scalable computational system, we have to reduce or even eliminate the overheads that are related to coherency protocols. Thus, our system imposes the following important property: From the point of view of a processor, a memory page can be either at the cache of a remote node or at the cache of local node, *but not at both*. This is the basis of the *Unimem* consistency model, which eliminates the need of maintaining global-scope cache coherence protocols.
- The Unimem consistency model (i.e., caching each memory page only among the nodes of a single coherence island) gives to application code the ability to be executed without the risk of data inconsistency. Furthermore, the consistency model of Unimem effectively pushes the application developers or runtime systems to place their computations close to data. In this manner, the locality of data is improved having as result the reduction of energy consumption and the mitigation of performance bottlenecks induced by the data movement.

- The current version of the platform also provides, today, via Unimem: Remote Direct Memory Access (RDMA) and asynchronous interrupts to remote nodes, via mailboxes. It is noticeable that the existing software consisting of either shared-memory or message-passing applications, can run on the platform with minimal or moderate modifications.

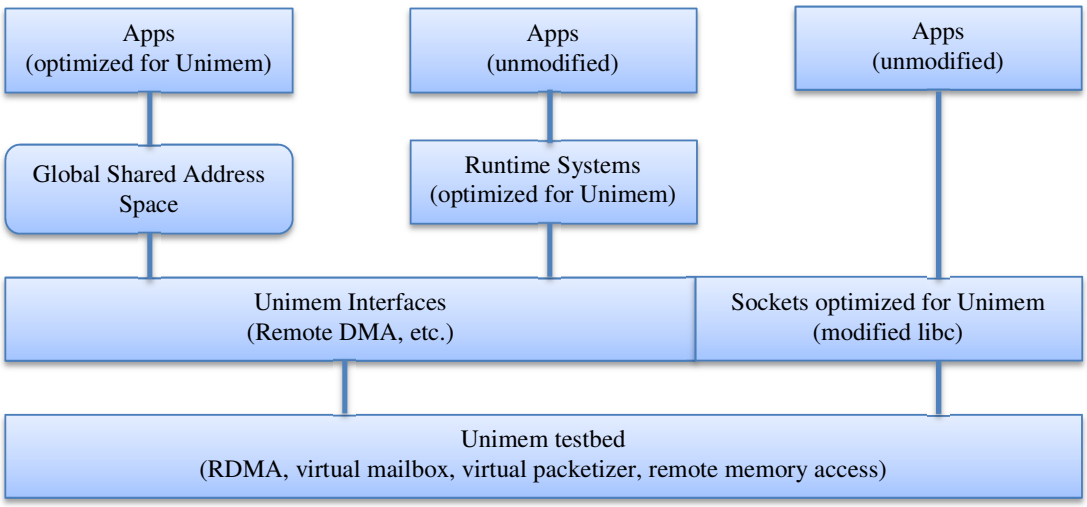

Figure 3 The programming interfaces that the Unimem provides and their interactions.

In this task, we leverage the advantages of a current implementation of the Unimem architecture by exposing a set of programming frameworks as shown in Figure 3. These programming frameworks provide a powerful set of communication mechanisms to developers and to runtime systems, resulting 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 provides. The GSAS environment defines an application interface (i.e., API) that is an extension of the GAS that the Unimem architecture provides. 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 reads, writes and many other atomic operations on the allocated space by using the appropriate library calls that the environment provides.
2. The user-space initiated DMA library that 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 the user space.
3. The sockets over RDMA. With Sockets over RDMA, a Unimem system can utilize low-latency communication among local nodes, by the means of fast RDMA transactions and bypassing of the kernel network stack. Furthermore, we give a description of the custom mailbox mechanism with which a kernel- or user-space application can

send and receive messages to and from remote nodes, thus enabling remote notification capability.

2 Global Shared Address Space

In this chapter, 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., Juno prototype nodes).

2.1 General

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 using 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. Note that in large scale system, the main overhead of an atomic instruction to some remote memory location is mainly the network latency, i.e., the order of magnitude of network latency is microsecs, while the order of magnitude of a user-level library call is nanosecs. Therefore, the overhead of a call to a user-level library is minimal.

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.

2.2 GSAS architecture overview

In this section, we provide an overview of the GSAS environment. We first give a brief overview of the hardware and software components. We next give a detailed description of the GSAS architecture. More specifically, in Section 2.2.2, we describe the scheme for addressing the remote space that the GSAS environment uses. Section 2.2.3, briefly presents the hardware components of the GSAS environment, and Section 2.2.4 presents its software components. Section 2.2.5 gives a detailed description on how a virtual addresses of GSAS environment is translated to a physical address. In Section 2.2.6, we describe the mechanisms involved in the execution of the atomic operations supported by the GSAS environment. Section 2.2.7 describes the mechanisms involved in remote memory allocation, and Section 2.2.8 presents how new processes are spawned in remote nodes.

2.2.1 Brief description

The current version of the prototype consists of a set of computing nodes (i.e. Juno development boards) that are connected through a custom network. Each of the computing nodes contains 6 processing cores, i.e., 2 ARM Cortex-A72 cores and 4 ARM Cortex-A53 cores. Furthermore, each computing node is equipped with 8 GB of DDR3 RAM. Any of the processing cores is able to communicate with any other local and/or remote processing core via a custom network that is described in (3). Figure 1 presents a high-level overview of the system description.

The main network mechanisms that are used by GSAS environment for communication among remote nodes are the virtualized mailbox and the virtualized packetizer. Each node of the system is equipped with a virtualized mailbox that contains 64 interfaces and a virtualized packetizer that also contains 64 interfaces. Thus, 64 threads per node are able to use the functionality of the GSAS environment at each point in time. Each of these threads is able to send a network packet to the mailbox of any other remote or local thread using one allocated interface of the local packetizer. Any remote thread is able to receive a network packet using one allocated interface of the local mailbox. This network packet contains the appropriate data describing the atomic operation that the sender thread wants to perform. The virtualization of the mailbox and packetizer hardware blocks enables system's threads to use a private instance (or interface) of them without noticing that more than one threads access the same hardware blocks. Thus, threads are able to directly use the functionality provided by these hardware blocks without involving kernel for sharing the hardware. The atomicity driver that runs on each system's node is responsible for managing the virtualized packetizer and the virtualized mailbox interfaces of the node.

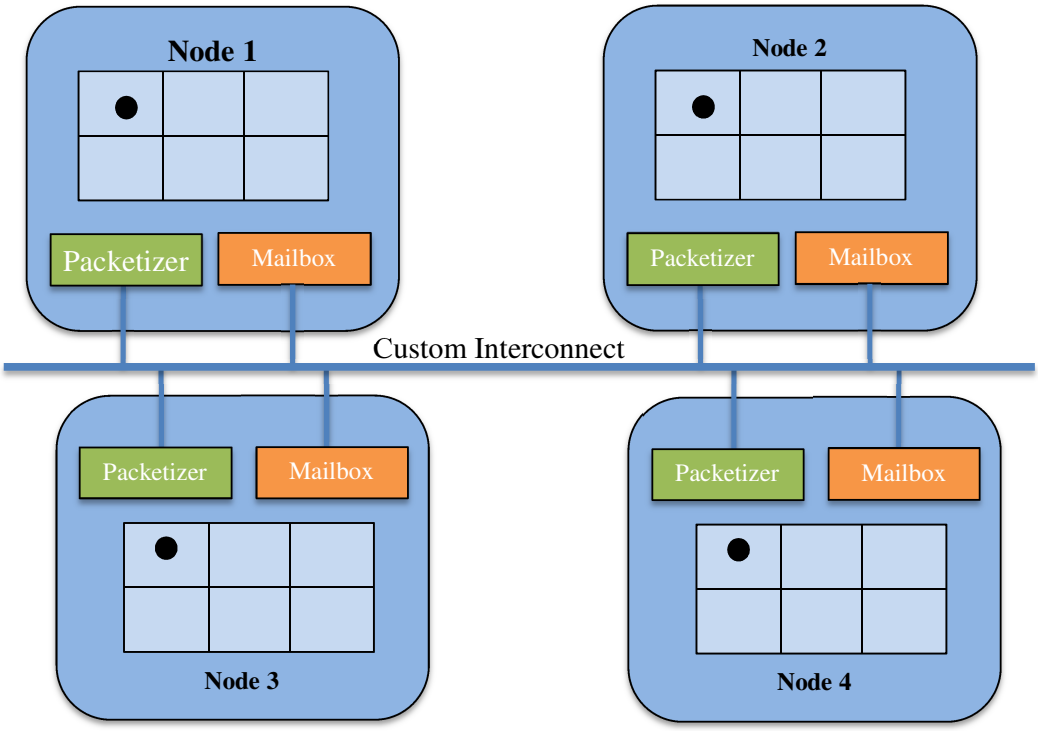

Figure 4. A high level overview of the architecture of the hardware prototype consisting of 4 nodes.

In the current generation of the GSAS environment, each node of the system is responsible for a distinct part of the global address space. On every node, there is a service, called atomic service, which is responsible for servicing atomic requests issued by remote or local threads to this part of global address space. Each thread that wants to issue an atomic operation prepares a network packet that describes the atomic operation (i.e., the kind of the atomic operation, the address that wants to apply the atomic operation, the arguments of the operation, etc.) and sends it (by using its allocated packetizer interface) to the mailbox of the responsible remote atomic service. The atomic service that receives the packet in its mailbox, applies the de-

scribed atomic operation to its local memory and responds to the issuer by sending to it a response packet.

2.2.2 Addressing on the GSAS environment

We first give a detailed description of the addressing policy that the GSAS environment follows. The GSAS environment supports addresses of 64 bits. The address space of GSA consists of N = 2^{16} partitions (see Figure 5). Each of these partitions is of size 2^{48} bytes and it is strongly related to at most one computing node. Thus, at most 2^{16} compute nodes are supported. In case that the system is equipped with $N < 2^{16}$ compute nodes, only the first N parts of the address space are used. In such a case, the address space of the GSAS environment is of size $N \cdot 2^{48}$ bytes. Therefore, the 16 most significant bits of an address contain all the appropriate routing information. Whenever a thread that wants to apply an atomic operation on some address of the GSAS environment, it can extract the destination node from the address itself.

The addressing policy of the global address space described above, leads us to allocate the first most significant 16 bits (out of the 64 bits) of an address for identifying the part of the address space the address belongs to. The remaining 48 bits are available to each node for addressing its local memory. The address space that is handled by a node is partitioned in pages, whereas each page is of size 4096 bytes (or 1000 in Hex). For example, the virtual address 0x0002-0000-0000-1001 (in Hex) points at the second word of the second virtual page of the second partition of the global address space. This address also states that the contained data are placed on the node of the system with id 1.

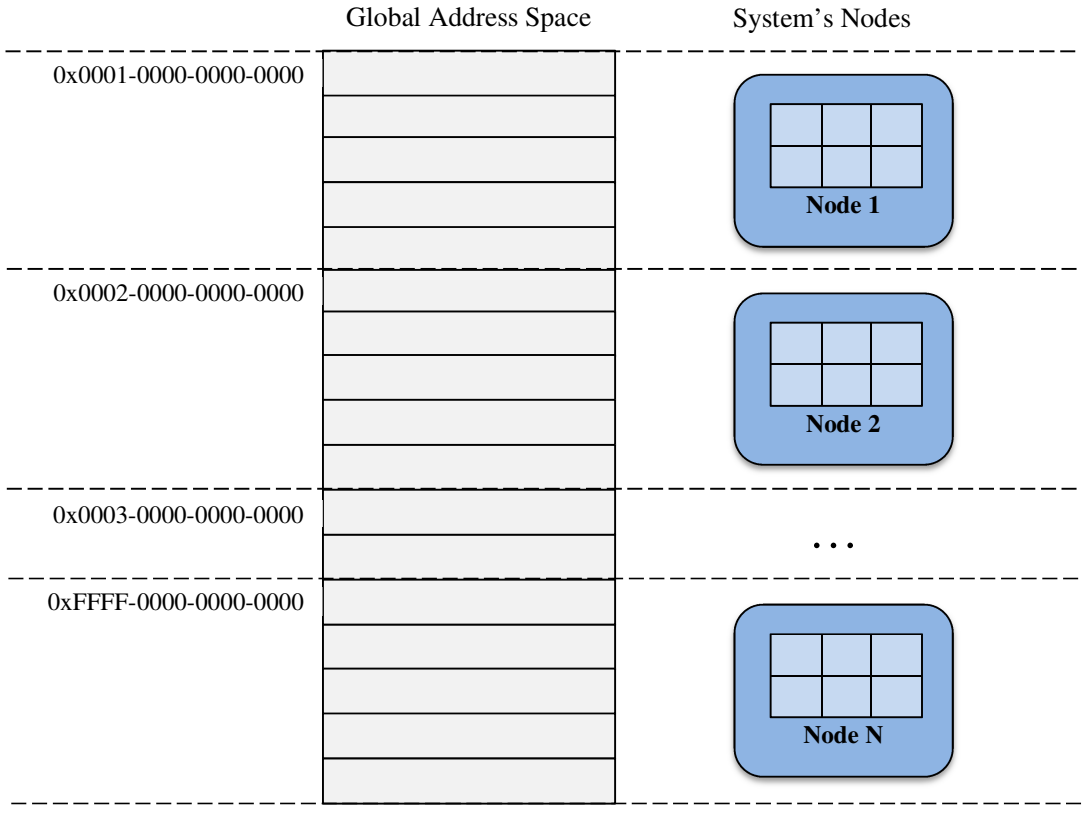

Figure 5. An overview of the addressing system that the GSAS environment offers (Addresses in Hex).

2.2.3 Hardware components

In this section, we discuss in detail the hardware components (i.e., virtualized mailbox and virtualized packetizer) 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 able 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[1]. 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.

Figure 6 presents the structure of the 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. A more detailed description of the virtualized mailbox is presented in Section 5.1.

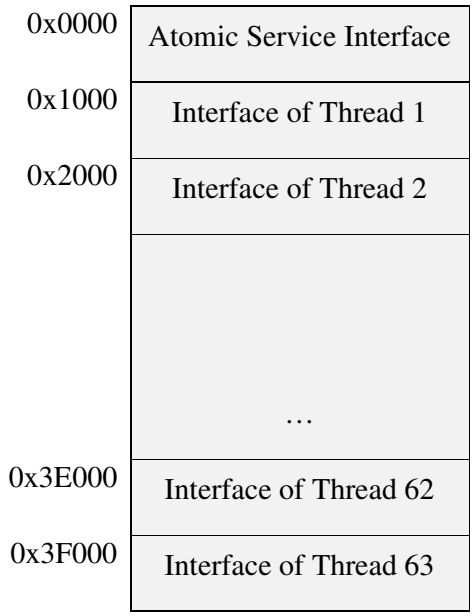

Figure 6. Memory mapping of the virtualized mailbox.

[1] The Unimem communication mechanisms presented in (3) give to a thread the ability to send small packets of size less or equal to 128 bits to some interface of a remote mailbox by issuing remote store operations. However, packets of 128 bits are not enough for describing an atomic operation.

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 noticeable 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. This prefix is set up by the atomicity driver, which is a kernel entity 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.

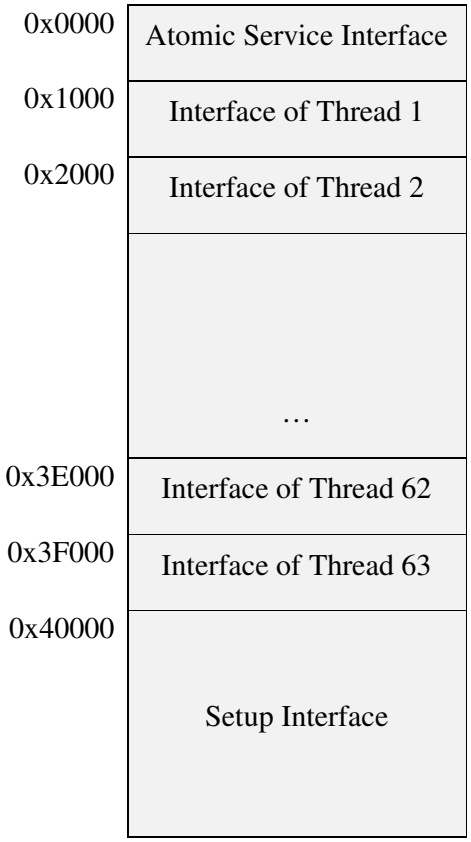

Figure 7. A memory map of a virtualized packetizer.

Similarly to the virtualized mailbox, virtualized packetizer is equipped with 64 interfaces (see Figure 7). 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

64th 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. A more detailed description of the virtualized packetizer is presented in Section 5.1.

We now discuss the format of network packets that describe atomic operations (see Figure 8). A request packet that describes an atomic operation consists of 256 bits and it is divided into 6 parts. The first part, which is named "Thread & Node ID", contains the identification number of the sender thread and at which node this thread resides on. This part is of size of 48 bits and it is written by the packetizer hardware block during its transmission to network. This identification number was set up during the initialization of the GSA environment by the atomicity driver and threads that are using the packetizer interfaces have no control on changing these identification numbers. Since only a trusted entity of the system (i.e., atomicity driver) is able to write the contents of this part, the atomic service can safely derive the origin of the atomic operation.

The second part of a packet is of size 8 bits and it is reserved for future use.

The third part of a packet, which describes the kind of atomic operation the atomic service should execute. This part is of size 8 bits and thus, at most 256 different types of operations could be supported. Currently, 12 different types of instructions are supported. At this generation of the GSAS environment the following types of instructions are supported.

- **allocSharedPage**: This type of operation commands atomic service to allocate a part of shared address space. The size of the shared address space to be allocated is written on field *Argument 1*.
- **freeSharedPage**: This type of operation frees a part of address space that starts on the address that is written of field *Argument 1*.
- **remoteFork**: This type of operation commands atomic service to spawn a new process on its local node. The description of the spawned process exists on an already allocated shared page that its address starts on the address written on filed *Argument 1*.
- **Read8**: This operation returns the contents of an 8-bit word whose virtual address is written in field *Instruction Address*.
- **Read32**: This operation returns the contents of a 32-bit word whose virtual address is written in field *Instruction Address*.
- **Read64**: This operation returns the contents of a 64-bit word whose virtual address is written in field *Instruction Address*.
- **Read4K**: This operation returns the contents of a shared page whose virtual address is written in field *Instruction Address*.
- **Write8**: This operation writes an 8-bit word whose virtual address is written in field Instruction Address and the contents are written in field *Argument 1*.
- **Write32**: This operation writes a 32-bit word whose virtual address is written in field Instruction Address and the contents are written in field *Argument 1*.
- **Write64**: This operation writes a 64-bit word whose virtual address is written in field Instruction Address and the contents are written in field *Argument 1*.

- **Write128**: This operation writes a 128-bit word whose virtual address is written in field *Instruction Address* and the contents are written in field *Argument 1* and in field *Argument 2*.
- **CAS**: This type of operation performs a *Compare&Swap* operation on the address that is written in field *Instruction Address* with arguments written in fields *Argument 1* and *Argument 2*.
- **FAD**: This type of operation performs a *Fetch&Add* operation on the address that is written in field *Instruction Address* with argument written in field *Argument 1*.
- **SWAP**: This type of operation performs a Swap operation on the address that is written in field *Instruction Address* with argument written in field *Argument 1*.

The fourth part of a packet describes the address that the atomic operation should be applied to (notice that some operations, such as remote spawn of a new process, do not make any use of this part). This part is of size 64 bits.

The fifth and the sixth parts of a packet that describes operations on GSAS environment, are used by some operations that their description has to send to the atomic service at most two arguments.

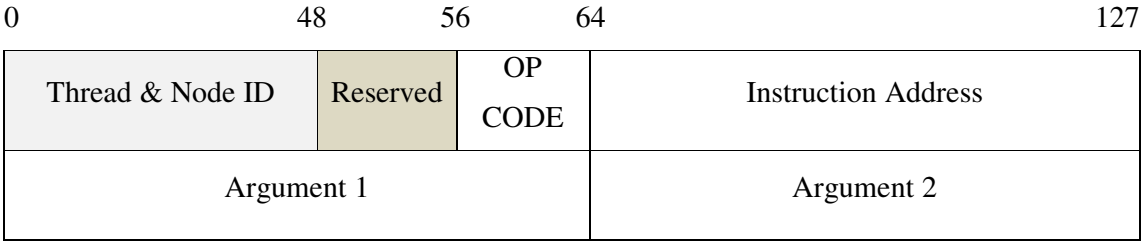

Figure 8. Description of the request packet of an atomic operation.

As we have already discussed for each atomic operation received by the atomic service, a response packet is send to the issuer of the atomic operation. This packet consists of two fields of 64 bits each totalling 128 bits. The first field reports an error code in case that the issued atomic operation failed. In case that the issued atomic operation is successful the value of this field is zero. The second field contains the result of the applied operation (e.g. the return value of a Read operation is placed in this field). The format of a response packet is presented in Figure 9.

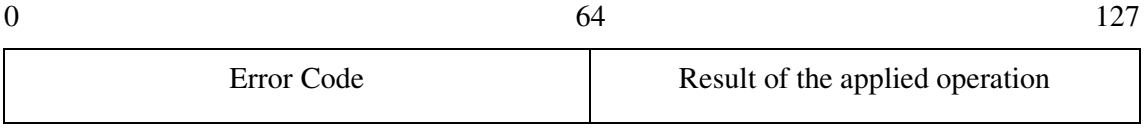

Figure 9. Description of the response packet of an atomic operation.

2.2.4 Software components

In this section, we describe the main software modules that are used on the GSAS environment. These software modules are the following.

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 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 (see Section 2.2.7 and Section 2.2.8, respectively). It is noticeable that whenever no request is pending on a node for 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) giving the 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 impacts 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.

2.2.5 Software MMU

The software MMU is a software data structure that every atomic service maintains an instance of it in its own local memory. The actual purpose of this software entity is to efficiently store and/retrieve pairs of <virtual global shared page address, physical page address on local node> (abbr. translation pairs). We first describe this data structure in detail and afterwards, we describe the basic mechanisms for storing and/or retrieving translation pairs to software MMU.

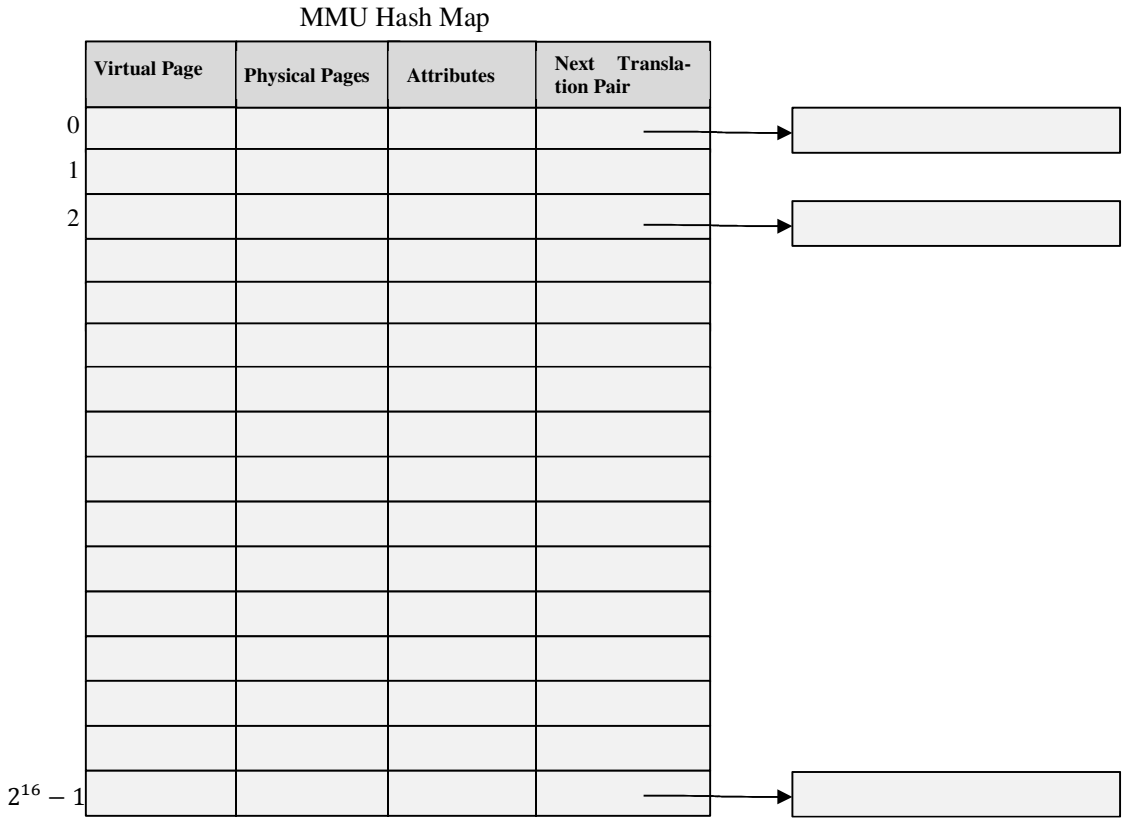

Figure 10. Hah map of the software MMU overview.

The basic data structure that the MMU uses to store the translation pairs consists of a hash map where the search key is the virtual address of a page (see Figure 10). Each entry of the MMU hash map has 4 fields. The first field, which plays also the role of the key, is the virtual address of the page to be translated. The second field contains the physical address on the local node of the page that its virtual address is assigned on. The third field of the software MMU contains the attributes of the page (i.e., which set of processes/threads is permitted to access the page, etc.[2]). The fourth field of the MMU hash map is a pointer to the next translation pair.

The key point for achieving efficient hash key search is that each row of the hash map should maintain a small number of records (i.e., the average number of key collisions should be

[2] The main use of the attributes field in the MMU hash map is for memory protection. However, in the current implementation of the GSAS environment, memory protection is not fully implemented yet.

small) in order to minimise the length of the hash map chains. Our experiments have shown that for a hash map of 2^{16} entries, the time required for hash map lookups was very small compared to network communication latency. Therefore, the hash map performance plays a minor role in the final performance of atomic operations.

We now describe the procedure that the atomic service follows in order to translate virtual addresses to physical. Whenever an atomic request is retrieved by the atomic service, the service decodes the request and in case this request dictates a memory access to some virtual address q, a translation of virtual to physical address should be performed. In such a case, the atomic service computes the virtual address ρ of the page that the requested address q resides on. It also computes the offset a of ρ in page q. The atomic service also computes a hash $0 \leq \hbar \leq 2^{16} - 1$ of ρ. Afterwards, the atomic service searches at the linked list beginning at the $\hbar$-th entry of the hash map. In case that ρ is successfully found in any of the buckets of the linked list, it retrieves the physical address of the page. Then, the atomic service adds the offset α to the address of the physical page and accesses the stored data.

At this point, we describe the procedure that the atomic service follows in order to install a newly allocated physical page in the hash map that is maintained by the software MMU. At first, the atomic service allocates a physical page of memory with some physical address ρ. Afterwards it allocates a virtual address q for the allocated page. Then, the atomic service computes the hash value $0 \leq \hbar \leq 2^{16} - 1$ of q and inserts a new entry to the linked list (i.e., a new node at the list) of the $\hbar$-th entry of the hash map. This entry contains the translation pair $\langle q, \rho \rangle$.

2.2.6 Performing atomic operations

In this section, we describe the actions that take place by the hardware modules (virtualized mailbox and packetizer) and the software modules (atomic service and the user level library that initiates the atomic operations) whenever an atomic operation (i.e., Read, Write, CAS, FAD and SWAP) of GSAS is performed.

Whenever an application wants to execute an atomic operation, the following sequence of steps is executed:

1. The user level library that initiates the atomic operations prepares a network packet describing the atomic operation.
2. It finds the destination of the network packet by appropriately hashing the global virtual address of the operation.
3. It writes the packet that contains the atomic operation and its destination to the local packetizer hardware block.
4. The local packetizer hardware block transmits the packet that contains the atomic operation, to the appropriate destination node.
5. The destination node receives the packet and pushes it to the first interface of the local virtualized mailbox.
6. The atomic service eventually removes the packet from the first interface of the virtualized mailbox.
7. The atomic service translates the global virtual address to a local virtual address by requesting the software MMU unit (see Section 2.2.5 for a detailed description of software MMU).

8. The atomic service checks if the atomic operation is valid and the target global virtual address is valid.
9. In case of success, the atomic service applies the operation to the local data.
10. Afterwards, the atomic service responds to the thread that requested the operation by directly writing the response to the appropriate entry at the remote node.
11. In case that the request is invalid, the atomic service drops the request.

2.2.7 Allocating remote memory

In this section, we describe the actions that take place by the hardware modules (virtualized mailbox and packetizer) and the software modules (atomic service and the user level library that initiates the atomic operations) whenever an allocation operation on GSAS is performed.

Whenever an application wants to execute *allocSharedPage* operation, the following sequence of steps is executed:

1. The user level library that initiates an *allocSharedPage* operation by preparing a network packet that contains the size (in number of pages) that the user wants to allocate on some remote node with id bid.
2. It writes the packet that contains the operation type, the size to be allocated and its destination to the local packetizer hardware block.
3. The local packetizer hardware block transmits the packet that contains the *allocSharedPage*, to the appropriate destination node.
4. The destination node receives the packet and pushes it to the first interface of the local virtualized mailbox.
5. The atomic service eventually removes the packet from the first interface of the virtualized mailbox.
6. The atomic service decodes the operation.
7. The atomic service checks if the arguments of the *allocSharedPage* are valid (i.e., valid size to be allocated etc.).
8. In case of success, the atomic service locally allocates the desirable physical space.
9. Afterwards, the atomic space assigns virtual addresses to the allocated physical space.
10. The atomic service installs the appropriate pairs <virtual page address, physical page address> in the software MMU (see Section *2.2.5* for a detailed description of software MMU).
11. Finally, the atomic service sends the virtual address of the allocated space to the thread that requested the space allocation by directly writing the response to the appropriate entry of the remote node.
12. In case that the request is invalid or there is not enough free memory in the node, the atomic service drops the request and responds NULL to thread that request the allocation.

2.2.8 Creating remote processes

In this section, we describe the actions that take place by the hardware modules (virtualized mailbox and packetizer) and the software modules (atomic service and the user level library that initiates the atomic operations) whenever a remote Fork operation on GSAS is performed.

Whenever an application wants to execute a Remote Fork, the following sequence of steps is executed:

1. The user level library that initiates atomic operations allocates a shared page on the remote node where the remote spawn operation will take place (see Section *2.2.7* for more details on how remote space allocation is performed).
2. The user library writes on the allocated shared page the filename of the executable that will be used for spawning a new process by performing write operations on it.
3. The user level library that initiates Remote Spawn operation by preparing a network packet that contains the address of the shared page that contains the filename of the executable that will be used for spawning a new process.
4. It finds the destination of the network packet by appropriately hashing the global virtual address of the allocated page.
5. It writes the packet that contains the atomic operation and its destination to the local packetizer hardware block.
6. The local packetizer hardware block transmits the packet that contains the Remote Spawn, to the appropriate destination node.
7. The destination node receives the packet and pushes it to the first interface of the local virtualized mailbox.
8. The atomic service eventually removes the packet from the first interface of the virtualized mailbox.
9. The atomic service translates the global virtual address that contains the filename to a local virtual address by using the software MMU.
10. The atomic service checks if the arguments of the Remote Spawn are valid (i.e., valid filename etc.) and the target global virtual address is valid.
11. In case of success, the atomic service spawns a new process imposed by the filename.
12. Afterwards, the atomic service responds to the thread that requested the process spawning by directly writing the response to the appropriate entry at the remote node.
13. In case that the request is invalid, the atomic service drops the request.

2.3 Application interface (API)

In this section, we provide a detailed description of the mechanisms that our GSAS environment offers to the user applications. At first, we describe the way that the application is able to explore the system's size (i.e., number of nodes) and at which node a process runs (Section 2.3.1). We next describe the allocation and de-allocation mechanisms (Section 2.3.2). In Section 2.3.3, we describe how usual read and writes are performed on the GSAS environment, and in Section 2.3.4, we describe how the remaining atomic operations behave. Finally, in Section 2.3.5 we outline how a process spawn new processes on remote boards.

2.3.1 Basic functionality

Before starting to write any c-code that uses the GSAS environment, include the following .h file in the code.

#include "atomic.h"

We now describe two basic calls that the GSAS environment provides to the applications giving the processes the ability to explore the number of available nodes of the system and the board id number that the process runs on.

- **uint64_t getNumberBoards(void)**: This function returns the number of available boards on the system that support the GSAS mechanisms.
- **uint64_t getBoardPid(void)**: This function returns the board id of the board that the process runs on.

2.3.2 Allocation and de-allocation mechanisms

- **void *allocSharedPage(uint16_t quantity, uint64_t bid)**: This function allocates a shared memory chunk of size quantity · PAGE_SIZE at board *bid*. In case that there is not enough memory NULL is returned, otherwise a pointer to the shared memory is returned. Notice that this portion of memory should not be accessed directly through load or store instructions. In case that the allocated memory is accessed through a store and/or load, the system behavior is unspecified (i.e., direct accesses of type *var_x = *addr* and/or of type **addr = var_y* lead to unspecified behavior). In the current development environment on the Juno prototypes PAGE SIZE equals to 4096 bytes.
- **uint64_t freeSharedPage(void *addr)**: This function frees the shared memory chunk pointed by *addr*. Notice that, in contrast to common behavior of *malloc/free*, the application is supposed to explicitly free all the allocated memory chunks that were allocated during its execution. Any memory chunk that it is not freed before the termination of the application occupies memory resources permanently (i.e., memory leak).

2.3.3 Read and Write operations

- **char READ8(char *addr)**: *READ8* returns the current 8-bit value that is stored in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **uint32_t READ32(uint64_t *addr)**: *READ32* returns the current 32-bit value that is stored in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **uint64_t READ64(uint64_t *addr)**: *READ64* returns the current 64-bit value that is stored in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **uint64_t READ4K(void *addr, char *buffer)**: *READ4K* returns the current value of the page that start in address *addr* in one atomic operation. It is imposed *addr* to be a multiple of 4096 (or 1000 in hex). In the opposite case, the three least significant bits of *addr* are ignored. In case that *addr* is an invalid pointer, application exits abnormally. The returned value is the number of bytes *read*, which under normal circumstances should be exactly 4096. Notice that in current implementation on the Juno prototype,

the contents of the memory that is read should not contain a pattern equal to the hex value CAFEBEBEDEADBEEF.

- **void WRITE8(char *addr, char val)**: *WRITE8* writes *val* (a 8-bit value) in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **void WRITE32(uint32_t *addr, uint32_t val)**: *WRITE32* writes *val* (a 32-bit value) in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **void WRITE64(uint64_t *addr, uint64_t val)**: *WRITE64* writes *val* (a 64-bit value) in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **void WRITE128(uint64_t *addr, uint64_t val[2])**: *WRITE128* atomically writes a vector of two words *val* of size 64-bit in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.

2.3.4 Other atomic operations

- **uint64_t CAS64(uint64_t *addr, uint64_t old val, uint64_t new val)**: A *CAS64* atomic operation on address *addr* stores *new_val* to *addr* if the current value at address *addr* is equal to *old_val* and returns 1 (true); otherwise, the contents at new val remain unchanged and 0 (false) is returned. Notice that all the arguments are of size 64-bits. CAS64 is also able to handle shared memory pointers by applying the appropriate type-casting, since pointers are of size 64-bits. In case that *addr* is an invalid pointer, application exits abnormally.
- **int64_t FAD64(int64_t *addr, int64_t val)**: A *FAD64* atomic operation atomically adds the (positive or negative) value *val* to the value stored on address *addr* and returns the value stored on address *addr* just before the addition. In case that *addr* is an invalid pointer, application exits abnormally.
- **uint64_t SWAP64(uint64_t *addr, uint64_t val)**: A *SWAP64* atomic operation atomically stores a 64-bit value *val* to address *addr* and returns the current value that is stored on address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.

2.3.5 Forking new processes

- **uint64_t remoteFork(char *filename, uint64_t arg, uint64_t bid)**: This function executes file *filename* on remote board *bid*. Keep in mind that *filename* points to a file that is accessible by the remote board with id bid on its attached filesystem. It is highly recommended for *filename* to reside on a location that is accessed by any node (i.e., a location on a shared filesystem). String filename should be of maximum size of _MAX_FORK_PATH_FILENAME_. Argument *arg*, which is a 64-bit unsigned integer, is also passed to the newly created process (afterwards, the newly create process is able to read this argument by calling function *readThreadArgument*). By appropriately using *arg*, on creation, an imaginary tree of processes could be created. Notice that the value of *arg* should not be equal to the hex value CAFEBEBEDEADBEEF. In case that *remoteFork* fails to spawn a process on a remote board, zero is returned, otherwise a value different than zero is returned. The standard output and error output of filename executable is redirected to file exec-*x*-stdout.log and file exec-*x*-stderr.log respectively; *x* is the id of the host board and exec is the filename of the executable. It is strongly recommenced to use absolute paths for identifying the executable filename.

- **readThreadArgument(void)**: This function returns the arguments of the running thread. In case that the running thread is not spawned by another thread (in such a case there is not any parent thread), *readThreadArgument* returns a value equal to _EMPTY_THREAD_ARGUMENT_. In any other case, *readThreadArgument* returns the argument of the current thread defined by the parent thread. In the current implementation of the GSAS environment, _EMPTY_THREAD_ARGUMENT_ is equal to the hex value CAFEBEBEDEADBEEF.

2.4 System limitations

In this section, we provide a few details on the limitations of the GSAS environment.

2.4.1 Unaligned accesses

In the current implementation of the GSAS environment, unaligned accesses on read, write and on all the other atomic operations are not supported. For example, all 32 bit read operations should be performed on addresses, for which modulo with 4 is equal to zero. In case that a read operation is performed at an unaligned address, the 2 least significant bits the of address are ignored. Similarly, 64 bit read operations should be performed on addresses, for which modulo with 8 is equal to zero; in the unaligned scenario, the 3 least significant bits are ignored. All shared memory accesses should be aligned following the rules of Table 1.

Atomic Operation	Address alignment (in bytes)	bits ignored in address
READ8	1	0
READ32	4	2
READ64	8	3
READ4K	4096	12
WRITE8	1	2
WRITE32	4	3
WRITE64	8	3
WRITE128	8	3
CAS64	8	3
SWAP64	8	3
FREE	4096	12

Table 1. Alignment rules for atomic instructions.

2.4.2 Double pointer accesses

Assume that in an application variable array is defined as follows.

*uint64_t **array;*

In an ordinary C application, a read on the 5th element of the array pointed by the 3rd pointer array is performed as follows.

*uint64_t tmp = *(*(array + 3)+5);*

Assume now that array is initialized as shared array of pointers to shared arrays. Now, a read on the 5th element of the array pointed by the 3rd pointed array is performed as follows.

*uint64_t *ptr = (uint64_t *)READ64(array + 3);*

uint64_t tmp = READ64(ptr + 5);

3 Mechanisms for RDMA, synchronization and remote memory allocation

In this Section we present the functionality and the application interfaces of the basic mechanisms that the Unimem architecture offers for performing RDMA, synchronization and remote memory allocation. Section 3.1 presents the available functionality of the RDMA transfers, while Section 3.2 presents the functionality of mailbox that is the major synchronization mechanism that the Unimem offers. In Section 3.3, the mechanisms for performing remote memory allocation are presented. Finally, in Section 3.4, we present the application interfaces for the Unimem mechanisms presented in Sections 3.1 - 3.3.

3.1 User-Initiated RDMA

In this section we present the user-space initiated DMA library that 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 point of this work is to expose this functionality to the user space.

In this first version of the user-space initiation of DMA transfers, all the underlying logic of the system is kept, thus the DMA engine management is done by kernel modules and the initiation of a DMA transfer requires the involvement of the whole OS stack (user-space, system calls and kernel-space calls). As this work evolves, the kernel space dependencies will be eliminated and all of the management will be done in the user space area.

3.1.1 DMA buffers

The memory areas used in DMA transfers should have some special characteristics. They should be contiguous and accessible by the DMA engines. In our system this is assured by using the functionality of the Remote Allocator Service, which has a global view of the system's memory layout. The RDMA library provides some functions for allocating and freeing DMA capable memory areas to be used as buffers for the DMA transfers. These buffers are categorized into local and remote.

Local buffers refer to memory regions that reside on the local node. These buffers are writable/readable from the user code that runs on the local node and the API provides the appropriate functionality for obtaining reference pointers to these locations.

Remote buffers refer to memory regions that reside on any remote node. These buffers are also writable/readable from the user code that runs on the local node (since the Unimem architecture supports load/stores to any memory region). The API provides the appropriate functionality for obtaining reference pointers to these locations.

A user space application is able to allocate two or more buffers on local and remote nodes having the ability to initiate transfers from the local node to a remote node and vice versa. Each buffer is assigned with a unique ID. In the first version of this API the system can have one buffer per node. Due to this limitation, the buffers have a maximum size of 256 MB. This will change in future versions. No special alignment is required applied during to allocation of such buffers.

3.1.2 DMA transfers

After the buffers have been allocated/retrieved, a DMA transfer can be submitted. This means that the transfer is stored in the library's transfer list and gets the DMA_PENDING status. There is a 'submitter' thread that handles the distribution of the transfers. The thread scans the list and submits all transfers that have the pending status.

The transfers are guaranteed to be submitted in the order that the transfers were requested. However, there is no guarantee that the transfers will be completed in the same order. The library provides certain functionality to users giving the ability to determine the status of the transfer. There is a polling mechanism, the 'status' function, as well as a way to register a callback function to be executed upon completion of the transfer. The library does not provide a mechanism giving the ability to the remote node determining the completion of a transfer. This issue will be addressed in a future version of the library.

3.1.3 Kernel modules

The user space RDMA library requires the functionality of some kernel modules. More specifically, it requires the driver for the Xilinx cDMA engine, the DMA buffers module and the remote allocator service module.

3.1.4 DMA buffers module

The dmabuffers module acts as an intermediate layer between the user space and the other kernel modules (the DMA driver and the remote allocator).

It is viewed as a character device from the system and receives commands through 'write' system calls. The user space library interacts only with this module it forwards the requests to the appropriate mechanism.

The first function of the dmabuffers module is the allocation of DMA capable memory regions. This allocation is done via the mmap system call. The user space part calls mmap and the dmabuffers module request a memory region from the remote allocator service. It then returns a readable/writeable pointer to this memory area. The second function of the module is the initiation of DMA transfers. The user space part issues a 'write' system call with the appropriate arguments (source address, destination address, size and transfer id) and the module initiates the transfer. When the transfer is completed the module is notified via the callback function and stores the transfer ID. When the user space part performs a 'read' system call, a buffer containing the status of the transfers is returned.

3.1.5 CDMA driver

For the initiation and completion determination of the transfers the cDMA driver is used. It exports symbols for initiating transfers and callbacks are registered to signal the completion. It implements all functionality required for the Xilinx cDMA hardware module to operate.

3.1.6 Remote allocator

The memory in the Unimem system is global. This means that any node in the system can access any memory part of any other node. For the administration of this feature the Remote Allocator Service has been developed.

Currently, each of the nodes has 8 GB of memory. This is mapped throughout the system with global physical addresses that depend on the node ID. So to access memory that reside in the node with ID 1, one must create a 40-bit address that contain this ID. The Remote Allocator Service is based on this mapping.

On initialization the allocator sets up regions on all the nodes. Then a character device is created from which a user space application can request memory regions. This is done with the *mmap* system call. In this call the user space part must specify the ID of the node to which the allocation will be performed. This way the user space application receives a normal pointer to that memory location that may reside on the local or a remote node. With this functionality, the dmabuffers module can create DMA capable memory regions to be used in transfers.

The other major function the Remote Allocator performs is translation. When an allocation is done the allocator keeps track of the process id of the owning process. This provides a way for the user space library to be able to use user virtual addresses when submitting transfers and the module is responsible for translating them to Unimem global physical addresses.

3.2 Mailbox

The mailbox driver provides an interface for the local and remote mailbox devices. It can be used by both user-space applications and kernel drivers. A mailbox device receives 64-bit messages from remote nodes and triggers interrupts to notify waiting applications. The 64-bit messages are classified into 16 different message types, recognized by the 4 most significant bits of the message, also called the 'message opcode'. Drivers or user-space applications can register free opcodes, so that the mailbox driver notifies them and delivers the messages.

3.2.1 Kernel-space Interface

- **typedef int (*mailbox_callback) (long long unsigned)**
- **int mailbox_register_callback(unsigned opcode, mailbox_callback cb,**
  char *description)
- **int mai lbox_unregister_callback (unsigned opcode)**

A kernel module can register/unregister a mailbox message type using the 2 previous functions. The message type is identified by the argument *opcode* which must hold a value between 0x0 and 0xf. On success, both functions return 0 and the given callback function is registered/unregistered. This function is being called when a message of the particular type has arrived. The whole 64-bit value of the message (including the *opcode*) is passed to the callback as an argument. The callback always runs in interrupt context.

- **int mailbox_send_message(int rnode , long long unsigned val)**

A message to the remote node *rnode* is sent with the previous function. The message contains the 64-bit *val* value.

3.2.2 User-space interface

A number of special device files (*/dev/Unimem/mailboxA*, */dev/Unimem/mailboxB*, and so on, with the letter indicating the node) represent the mailboxes. A write operation on a remote mailbox device sends the message to the remote node. The buffer size of the write system call must be equal to 8 bytes (64 bits).

A read operation on the local mailbox device reads any received message. The read buffer must be big enough to hold 8 bytes. If more than one messages are available, the older one is read first. If no messages are available, the read call either blocks or returns -EAGAIN, if the device is opened in non-blocking mode. The *select, poll*, and *epoll* system calls are also supported on read operation.

A message type must first be registered to the driver, for the driver to be delivering it to an application. In order to register/unregister a message type, the following struct must be given to the driver using a write call.

```
struct mailbox_action_t {
        char action;
        unsigned opcode;
        int val;
}
```

The action field can be:

- 'r': the message type identified by the field *opcode* (must have a value between 0x0 and 0xf in Hex) is registered with the write call to the local mailbox device. The field *val* is here ignored.
- 's': the message type identified by the field *opcode* is registered and the application is notified for the arrival of a new message via a signal. The signal number is equal to *val*.
- 'e': the message type identified by the field *opcode* is registered and the application is notified for the arrival of a new message via the *eventfd* system. A *eventfd* descriptor must previously be created and its number is passed to the driver using *val*. The value returned from a read call to the *eventfd* descriptor indicates the number of available messages.
- 'u': the message type identified by the field *opcode* is unregistered.

3.3 Remote allocator

The remote allocator module is responsible for allocating and distributing memory portions across coherency islands. In the user-initiated RDMA mechanism it plays the part of allocating local and remote buffers. It makes sure that the memory is contiguous and non-swappable, and thus, it is accessible by the DMA mechanism.

In this first version of the API the remote allocator service does not have a full image of the memory throughout the whole system and that leads to a limitation in the user RDMA mechanism. This limitation is that the user cannot have more than one buffer per node, because the mapping is static.

Because of the use of our remote allocation service no memory copies on the data are needed. The memory of the local and remote buffers is mapped and directly accessible from both the user-space and the cDMA engine. The only memory copies that are done (and are inevitable) is the writing of the source and destination address to the cDMA engine's transfer descriptors.

3.4 API reference

The definition of the user-space RDMA API follows:

3.4.1 Types

- **unim_dma_node_t**: Represents a node ID in the Unimem System.
- **unim_dma_transfer_t**: Represents a RDMA transfer object.
- **void (*unim_comp_cb_t)(void *)**: Function prototype for the transfer completion callbacks.
- **unim_dma_buffer_t**: Represents a RDMA buffer.
- **unim_dma_buf_id_t**: Respresents a RDMA buffer ID.
- **unim_dma_transfer_id_t**: Represents a RDMA transfer object. Transfer IDs are unique throughout the whole system.
- **unim_node_id_t**: Represents a Node ID in the system.
- **unim_dma_status_t**: Represents the sate of a RDMA transfer. There are four states for a transfer: DMA_PENDING, DMA_STARTED, DMA_COMPLETED and DMA_FAILED.

3.4.2 Functions for initialization/cleanup

- **int unim_dma_init(unim_node_id_t node_id)**: This function initializes the library. For the initialization it requires the ID of the current node. It returns 0 on success.
- **void unim_dma_cleanup(void)**: Frees up resources when the library is no longer needed.

3.4.3 Functions for buffer manipulation

- **unim_dma_buffer_t *unim_alloc_buf_l(uint32_t size)**: Allocates a new DMA capable of 'size' bytes buffer on the current node. It returns a new buffer object on success or NULL if there was an error.
- **unim_dma_buffer_t *unim_alloc_buf_r(uint32_t size, unim_dma_node_t node)**: Allocates a new DMA capable buffer of 'size' bytes on a remote node. The node is identified by the 'node' object. It returns a new buffer object on success or NULL if there was an error.
- **void unim_free_buf(unim_dma_buffer_t *buf)**: It frees a RDMA buffer.
- **unim_dma_buffer_t *unim_get_buf(unim_dma_buf_id_t id)**: This function retrieves a RDMA buffer object based on the buffer 'id'. Returns the buffer object or NULL if there was an error.
- **unim_dma_buf_id_t unim_get_buf_id(unim_dma_buffer_t *buf)**: It retrieves the buffer id for the specific RDMA buffer object 'buf'.
- **void *unim_buf_get_memory(unim_dma_buffer_t *buf)**: Gets a readable/writable pointer for a RDMA buffer's memory. Returns a pointer or NULL if there was an

error.

3.4.4 Functions for transfers

- **unim_dma_transfer_id_t unim_dma_submit_transfer(unim_dma_buffer_t *src, unim_dma_buffer_t *dst, uint64_t src_off, uint64_t dst_off, uint64_t size, unim_comp_cb_t comp_cb, void *cb_arg)**: This function submits a new transfer of data from buffer 'src' + 'src_off' to 'dst' + 'dst_off', of 'size' bytes. Optionally the user can register a function with prototype 'unim_comp_cb_t' and an extra argument to be called upon completion of the transfer. It returns a transfer ID or 0 on error.
- **unim_dma_transfer_id_t unim_dma_submit_transfer_from_mem(void *src, unim_dma_buffer_t *dst, uint64_t src_off, uint64_t dst_off, uint64_t size, unim_comp_cb_t comp_cb, void *cb_arg)**: This function submits a new transfer of data from the local memory pointer 'src' + 'src_off' to the RDMA buffer 'dst' + 'dst_off', of 'size' bytes. The transfer is done in two stages. First the library copies the data from 'src' to an internal RDMA buffer and then performs the transfer. This memory copy will be eliminated in later version with the extension of the remote allocator mechanism. Optionally the user can register a function with prototype 'unim_comp_cb_t' and an extra argument to be called upon completion of the transfer. It returns a transfer ID or 0 on error.
- **unim_dma_transfer_id_t unim_dma_submit_transfer_to_mem(unim_dma_buffer_t *src, void *dst, uint64_t src_off, uint64_t dst_off, uint64_t size, unim_comp_cb_t comp_cb, void *cb_arg)**: This function submits a new transfer of data ffrom buffer 'src' + 'src_off' to the local memory 'dst' + 'dst_off', of 'size' bytes. The transfer is done in two stages. First the library performs the transfer from 'src' to an internal RDMA buffer and then copies the memory from the buffer to 'dst'. This memory copy will be eliminated in later version with the extension of the remote allocator mechanism. Optionally the user can register a function with prototype 'unim_comp_cb_t' and an extra argument to be called upon completion of the transfer. It returns a transfer ID or 0 on error.
- **unim_dma_status_t unim_dma_status(unim_dma_transfer_id_t tID)**: Retrieves the status of the transfer with ID 'tID'.

3.4.5 Miscellaneous functions

- **unim_dma_node_t unim_dma_get_node(unim_node_id_t id)**: Retrieves the node object for the node with ID 'id'. Returns the node object or NULL if there was an error.

4 Sockets over RDMA

In the past few years, micro-servers have entered the data-center market, where power consumption is a critical factor of cost. Micro-servers consist of low cost and low power Compute Units that are usually found in embedded devices. Such compute units can be deployed in large numbers assembling a modern data-center, forming multiple Coherence Islands – groups of compute units that are coherent – connected together. I/O resources in micro-servers are unevenly scattered among compute units, providing a challenge to the OS to implement mechanisms for secure sharing and remote access, optimized for latency and throughput.

Sockets over RDMA deliver a low-latency communication mechanism on the Unimem architecture. In sockets over RDMA, the TCP connections among system nodes use RDMA transactions in order to transfer data. This gives the ability to bypass the whole kernel TCP network stack achieving high performance.

RDMA allows very fast data transfers, without blocking the processor. A DMA engine only needs to know a source and a destination address to transfer the amount of data directly from memory to memory, avoiding most of the copies between buffers that the network stack typically performs. By utilizing translation of physical addresses on remote nodes, the RDMA capability is generated. Together with the mailbox mechanism to enable remote notifications, a complete TCP communication solution is created.

The implementation of sockets over RDMA consists of two parts:

- In user space, we intercept system calls related to the popular Berkeley Sockets API, to bypass the kernel TCP / IP stack and avoid its overhead.
- In kernel space, we handle data transfers by means of RDMA transactions and remote mailbox notifications, using a custom RDMA driver.

With this approach, unmodified applications are able to efficiently utilize the RDMA-capable custom interconnection network. System call interception is executed within a custom standard C Library (libc), so that we can avoid entering the kernel every time, which suffers from time-consuming context switches. Furthermore, the kernel part of our system ensures data security for running applications by avoiding to expose physical addresses to the user space.

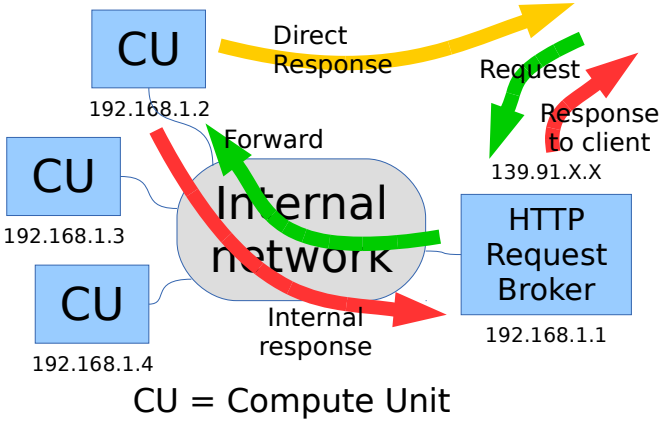

Figure 11 Overview of an example network configuration.

Figure 11 summarizes the network environment using the use-case of an HTTP request proxy/broker as an illustrating example.

We present a set of mechanisms that enhance the performance of network communication between compute units of different coherence islands. Internal communication is implemented using remote memory and RDMA scatter-gather operations, bypassing the traditional OS TCP/IP stack, without application modification.

Compute units share a virtualized Ethernet NIC that handles traffic to the external world. This I/O sharing is outside the scope of this deliverable, and is based on work carried out by FORTH during the EuroServer project (FP7-ICT-610456).

Our Sockets-over-RDMA mechanism allows unmodified applications running on compute units to communicate over the internal interconnection network. We intercept all system calls related to the Socket API (*connect*, *accept*, *socket*, *send*, *recv* etc.), by using a modified standard C library (libc). For internal IP addresses, the OS network stack is bypassed and our custom RDMA driver is used instead.

Per-connection buffers (organized as ring buffers) are allocated and handled by our driver to enable remote DMA transfers, using the CDMA engine. These buffers are mapped to the user-space side, as well, in order to avoid kernel overhead, when possible. Remote notifications to local peers are sent through the mailbox mechanism.

The current implementation achieves the primary goal of allowing unmodified applications to communicate over the RDMA-capable internal interconnect in the prototype. Basic functionalities like creating TCP connections and performing data transactions through them have been implemented, along with support for more advanced features like multithreaded and forked applications.

4.1 Supported features and limitations

In order to enable Sockets-Over-RDMA, an application could just use the custom libc library. Thus, there is no need to recompile the application. To run the application, one simply has to use the LD_LIBRARY_PATH environment variable to force the application to be linked with the custom libc and afterwards, all local TCP connections will employ RDMA sockets. A local node is distinguished by its IP address. Currently, all nodes have fixed IP addresses (192.168.1.10, 192.168.1.11...).

Statistics of the RDMA sockets usage on a node can be obtained from an entry of the proc filesystem. There, the number of local (RDMA sockets) connections, of RDMA (write) operations and of bytes received and sent from the node are available.

The following list presents the features and the limitations that the current sockets over RDMA implementation has.

- Multithreaded and multiprocess applications are fully supported.
- Non-blocking sockets are fully supported.
- *select*, *poll* and *epoll* system calls are fully supported.
- Socket-related system calls fully supported (see Table 2).

accept	*connect*	*socket*	*listen*
bind	*close*	*getpeername*	*getsockname*
read	*recv*	*recvfrom*	*recvmsg*
readv	*write*	*send*	*sendto*
sendmsg	*writev*	*accept4*	*dup*
dup2	*dup3*	*fork*	*clone*
select	*pselect*	*poll*	*ppoll*
epoll_wait	*epoll_pwait*	*epoll_create*	*epoll_create1*
epoll_ctl			

Table 2 System calls supported by the current implementation of sockets over RDMA

- Socket-related system calls partially supported:
  - getsockopt, setsockopt: SO_ERROR is supported, others are ignored
  - ioctl, fcntl: Can be used to set/unset the non-blocking socket mode
- Socket-related system calls not supported yet:
  - execve: sockets with the SOCK_CLOEXEC attribute are not supported
  - sendfile
  - splice

4.2 Supported Applications

The current implementation of the sockets over RDMA is successfully tested on the following applications.

- iperf
- memcached
- openssh
- openmpi

4.3 Support for event-driven socket calls

One important area of improvements over the original version (coming from the EuroServer project) has been to add support for event-driven socket calls (such as *select* and *epoll*) to support popular server applications, and support for passing and interpreting socket options (*setsockopt* call) such as dynamically setting buffer sizes for sending and receiving data, and operating sockets in a non-blocking mode (e.g. use the *fcntl* call to set the O_NONBLOCK option for an open socket).

Network servers are traditionally implemented using a separate process or thread per connection. For high performance applications that need to handle a very large number of clients simultaneously, this approach won't scale well, due to limitations in system resource con-

sumption and overheads in context-switching. An alternate method is to serve non-blocking I/O requests with a single thread, along with some readiness/completion notification method which inform threads when a connection (one out of many being monitored) is ready to deliver or consume more data. In Linux, this functionality is provided by the *select*, *poll* and *epoll* families of system calls. In our work, we built on top of this infrastructure in the Linux kernel. The *epoll* call is the more recent, and more comprehensive, of the available event notification facilities in Linux, but several applications still use *select* and *poll*. As specific examples relevant to the ExaNoDe project, the MPICH and OpenMPI implementations of the MPI standard rely on *select* and *epoll*, respectively. Furthermore, the libevent API provides a mechanism to execute a callback function when a specific event occurs on a file descriptor or after a timeout has been reached. Our work on sockets-over-RDMA supports this popular API, which is in use in several popular implementations of network and database servers (e.g. memcached, PostgreSQL).

This work has considerably broadened the applicability of our sockets-over-rdma implementation, to cover more workloads, and allows applications to fine-tune parameters and capabilities. Going forward, we are considering optimisations to improve the scalability of our implementation, especially in the direction of supporting very large numbers of open connections.

5 Hardware to software interface

In this section, we give a detailed description of the main hardware blocks provided by the Unimem architecture, i.e. virtualized mailbox and virtualized packetizer, that are used by the three application frameworks presented on Sections 2 – 4. The hardware blocks described in this section have been co-designed with the software infrastructure and modules described earlier, with their functional requirements derived from a use-case analysis performed in close cooperation with our hardware engineers (WP5).

5.1 Virtualized mailbox

The virtualized mailbox (abbr. vmbox) implements up to 64 Hardware FIFOs. As shown in Figure 12, each FIFO is accessible through the corresponding mailbox (abbr. mbox) interface (MIF). Each MIF provides separate access for each hardware FIFO. AXI reads to the preferred MIF number, dequeues the corresponding FIFO, whereas an AXI write request to the preferred MIF, enqueues the FIFO.

For example, Figure 12 illustrates 3 enqueues on FIFO #0 through a single-word write and a two-word write (2-words burst). Figure 12 also depicts a single-word write enqueuer on the FIFO #6, and 2 dequeues on the FIFO #6 (2 back-to-back read requests).

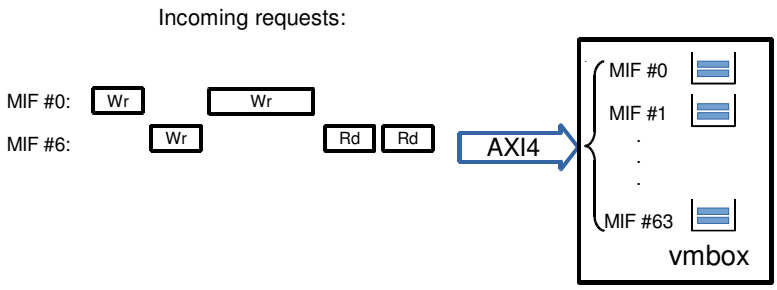

Figure 12: A General use-case of the vmbox block.

5.1.1 Peripheral Usage

As shown in Figure 15, the vmbox block maps each hardware FIFO to a single page at zero offset. In the current implementation the 64 FIFOs are consecutively mapped. Thus, the base address of the FIFO #0 is sufficient in order to map the rest of the hardware FIFOs.

Write requests to the correct offset and page, will make a successful enqueue as long as the corresponding FIFO is not full. A write request can also be a burst of multiple words. A single-word request to a full FIFO will always lead to a negative acknowledgment. A multi-word request to a full FIFO will always return a negative acknowledgment. However, a multi-word request on an almost full FIFO is likely to fail for the reasons described in Section 5.1.2 pg. 33.

Read requests to the correct offset and page, will indicate whether the FIFO is empty or not. A read request to an empty FIFO, will return "CAFEBEBEDEADBEAF-CAFEBEBEDEADBEAF" data. A read request to a non-empty FIFO will return the actual data of the FIFO and make a single dequeue. A multi-word read request will make as many

dequeues as the number of the burst size. For example, a 4-word read request will make 4 dequeues.

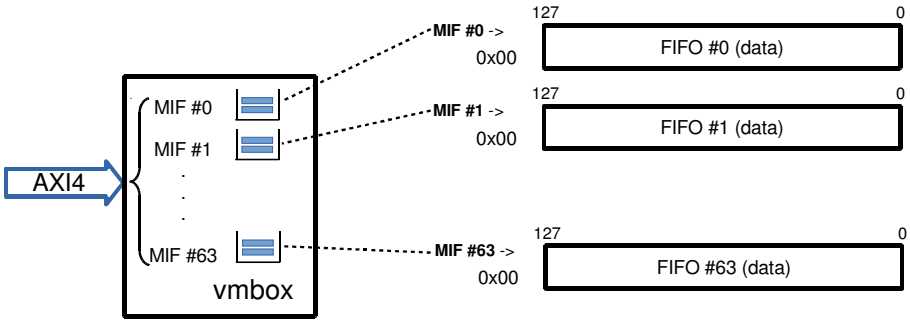

Figure 13: Vmbox mapping.

5.1.2 Known issues

When multi-word write requests are issued to an almost empty FIFO, there is case that the vmbox will discard the words that do not fit, and a positive acknowledgment will be returned. For example, assume a non-full FIFO with 3 elements left to be full. A 4-word write request will accept the first 3 words, and the fourth will be rejected. However, a positive acknowledgment will be returned.

5.1.3 Register space

This section depicts in table format the register space of MIF (Table 4). The *Offset* column is written in hex format, the *Valid* column shows the actual bits used by the hardware, the *Actual* column depicts the actual size of data seen by the software, and the *AT* describes the access type. R: Read, W: Write, WC: Write Clear (a single write to the corresponding register, clears the register).

5.1.4 Virtualized mailbox interface (MIF)

Offset per Pg.	Description	Valid	Actual	AT
0x00	• Each page corresponds to a separate FIFO. • Read or write requests are valid for this offset. Single word read requests return the actual data and dequeue the corresponding FIFO, when that FIFO is non-empty. Otherwise "CAFEBEBEDEADBEAD-CAFEBEBEDEADBEAD" is returned. Multi-word read request are also valid. • Single or Multi word write requests enqueue the corresponding FIFO. Negative acknowledgment is returned if the corresponding FIFO is full.	[127:0]	[127:0]	R/W

Table 3: MIF Register Space

5.2 Packetizer

The basic function of this block is to initiate an atomic packet to a user-defined destination, carrying user-defined data. The necessity of this block relies on to the fact that a source may not be able to initiate a transaction as a burst. For example, when a CPU is about to send a large message, it may create separate AXI transactions instead of a single one. This may cause interleaving of different messages sent by different sources and break the atomicity of the messages. The packetizer can gather a chuck of different fragments, and produce a single burst transaction to the corresponding destination.

As shown in Figure 14, the packetizer utilizes 65 pages. The 64 pages are assigned to the packetizer interfaces (PIFs) and the single page is assigned to the trust interface (TIF). Both can be read or written by the system software. Write transactions to any PIF, allows a portion of the atomic packet payload data (APPD) and destination address (DA) to be setup for that PIF. The other APPD portion is found stored in an internal structure, previously setup through TIF by a trust software.

Upon a PIF setup, an atomic packet is being triggered for transmission. A PIF setup is completed as soon as the appropriate number of write transactions for that PIF are completed.

Figure 14 illustrates 2 atomic packets transmission scenario. For each atomic packet, 3 small packets are needed (Pck1, Pck2, Pck3). Small packets may be injected to the AXI slave IF in any order, and destined for any PIF. Upon reception of the third small packet, an atomic packet may be initiated for that PIF. Both PIF and TIF stores the incoming small packets internally. The internal structure of PIFs and TIF is presented on Section 5.2.6.

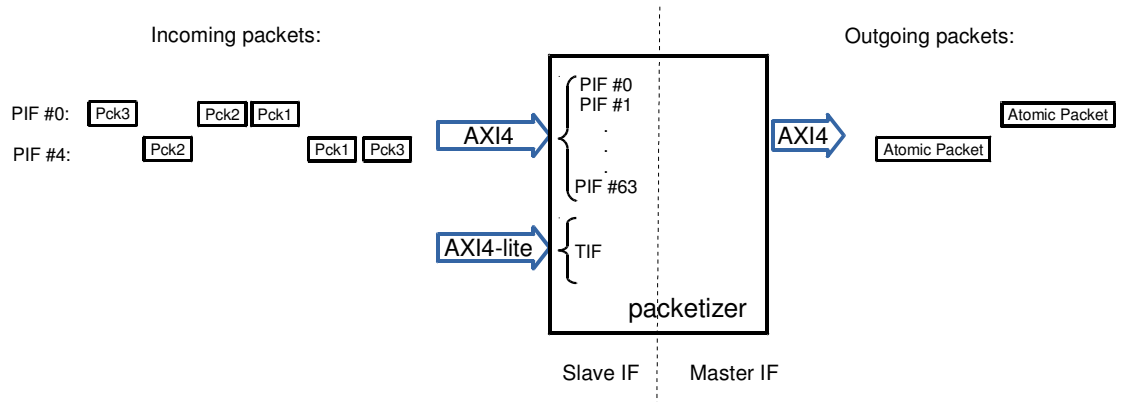

Figure 14: A General use-case of the packetizer block.

5.2.1 Peripheral usage

As shown in Figure 15, the packetizer utilizes two tables. The Packet Table (PCKT) and the PID Table (PIDT). Both can be read or written by the system software. The PCKT keeps a portion of the APPD and the DA, whereas the PIDT keeps the other part of the APPD.

The TIF allows access to the PIDT and some configuration registers whereas the PIF allows access to the PCKT and a packet initiation (PIF setup). Upon a PIF setup, a packet is being prepared using the payload data found on the corresponding entries of PCKT indexed by the PIF page number, and the PID data found in the corresponding entry of the PIDT. For i.e.,

when the 4th PIF has been setup, a packet is being prepared using the corresponding payload data and destination address found on the PCKT, plus the PID data found on the 4th entry of the PIDT. Sections 5.2.2 and 5.2.3 describe the TIF and PIF setup respectively.

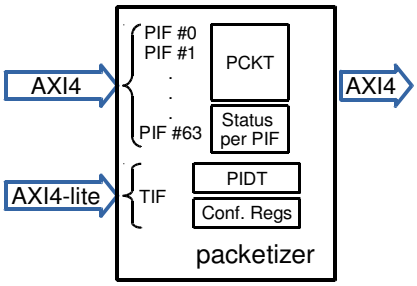

Figure 15: Internal structure of the packetizer.

5.2.2 TIF setup

A trusted process can setup the PIDT and configuration registers through the TIF. The trusted software is responsible for the correct initialization of the afford-mentioned structures. Figure 16 illustrates the TIF mapping. Each PID entry is available for each PIF. For example, the first PIDT entry is available to the PIF #0, the second PIDT entry is available for the PIF#1. Thus, 64 PIDT entries are utilized by 64 PIFs. The offset mapping starts from the begging of the TIF page. Beyond the PIDT range, three TIF configuration registers are mapped. These are: BASE_DEST, PID_CONF and CLR_REQ.

In particular, (i) the BASE_DEST keeps the base address of the destination. It is assumed that all destination addresses are consecutive, starting from the BASE_DEST address. (ii) The PID_CONF indicates which PID table entry is valid. Each bit (starting from the lsb to the msb) corresponds to a PIF page (64 pages). (iii) The CLR_REQ clears the corresponding PIF page. Each bit (starting from the lsb to the msb) corresponds to a PIF page (64 pages). This register should never be used in normal circumstances but for debug cases. Figure 16 illustrates TIF mapping.

A successful TIF setup includes PIDT, BASE_DEST and PID_CONF configuration from trusted software.

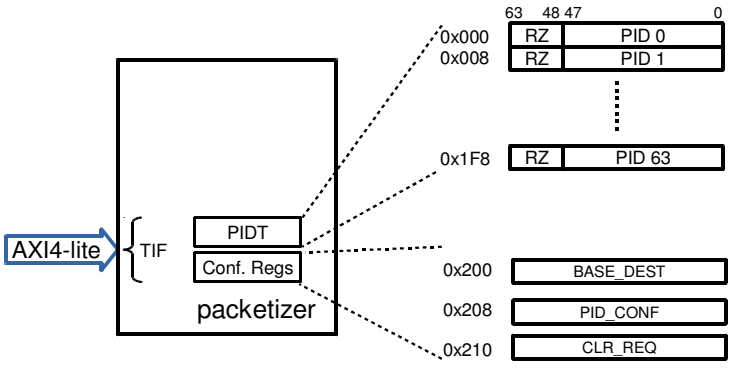

Figure 16: Trust Page Fields.

5.2.3 PIF setup and status

As shown in Figure 18, a PIF provides access to the corresponding entry of the PCKT and its status information. A successful packet initiation includes a successful PIF and TIF setup. For each packet initiation a PIF setup is always needed, but a TIF setup can happen just once. As soon as the PIF setup has been successfully completed, an atomic packet is transmitted carrying the data shown in Figure 17.

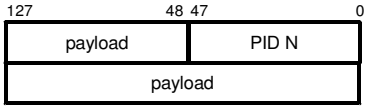

Figure 17: Atomic Packet Format.

A correct PIF setup is done by checking whether the desired PIF is ready. As long as the desired PIF page is ready the user may issue three writes in any order. The third write will trigger an atomic packet initiation. 2 writes are needed for the "payload" field of the atomic packet (offsets 0x00 and 0x10), and a third write for the destination address. A read to the desired PIF at offset 0x30, indicates the PIF status. 5 bits (***O, A, N, I, P***) per PIF are utilized for this purpose as shown in Figure 18.

- ***P***END: If asserted, there is a pending packet for preparation/transmission. As soon as it becomes zero, the user should check the O, A, N, I bits for detailed status.
- ***I***LLEGAL: The packet was about to be sent but the destination address was illegal.
- ***N***ACK: A negative ack has been received for the transmitted packet. It becomes zero again upon the next packet transmission.
- ***A***CK: A positive ack has been received for the transmitted packet. It becomes zero again upon the next packet transmission.
- ***O***VERWR_BUF: Debug bit. If asserted, a write to a PIF has been initiated during a packet preparation.

A user should make sure that the PID_CONF bit of TIF which maps to the corresponding PIDT entry and PIF page, is valid and the destination address written through the PIF does fall within the valid destination address range indicated by BASE_DEST of the TIF configuration register. If neither of the afford-mentioned rules are taken into account, the atomic packet initiation will fail.

5.2.4 Known issues

In the current version of the prototype, consecutive PIF setups at the same interface is not allowed because a single hardware buffer is utilized per interface. The user should check whether the desired PIF is ready, before he or she makes a consecutive PIF setup at the same interface. For example, if 3 PIF setups are needed for interface 14, the user should make the following steps:

- Setup PIF#14
- Check if PIF#14 is ready
- Setup PIF#14

- Check if PIF#14 is ready
- Setup PIF#14.

Notice that consecutive PIF setups on different pages are freely allowed without any check by the user, because each interface utilizes its own hardware buffer. For example, if a user needs to make 3 consecutive PIF setups on different interfaces he or she can do the following:

- Setup PIF#14
- Setup PIF#15
- Setup PIF#16.

In an updated version of the hardware that will be available soon, the packetizer block will be able to accept consecutive PIF setups for the same interface by activating the AXI4 back-pressure mechanism.

5.2.5 HW block diagram

Figure 19 illustrates the internal hardware sub-blocks of the packetizer hardware block.

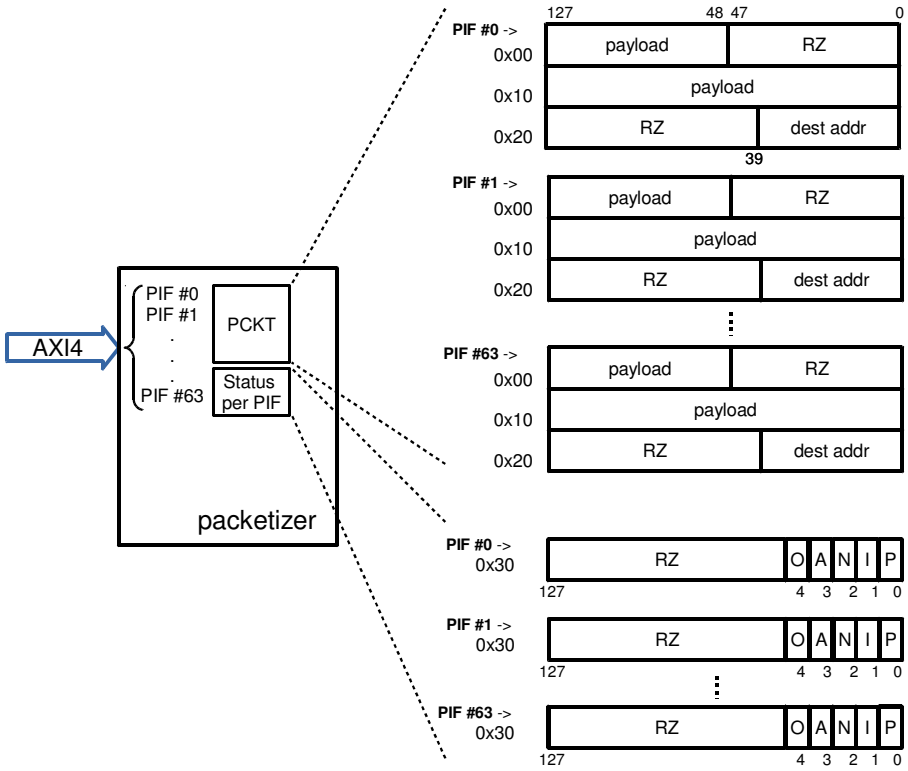

Figure 18: PIF mapping.

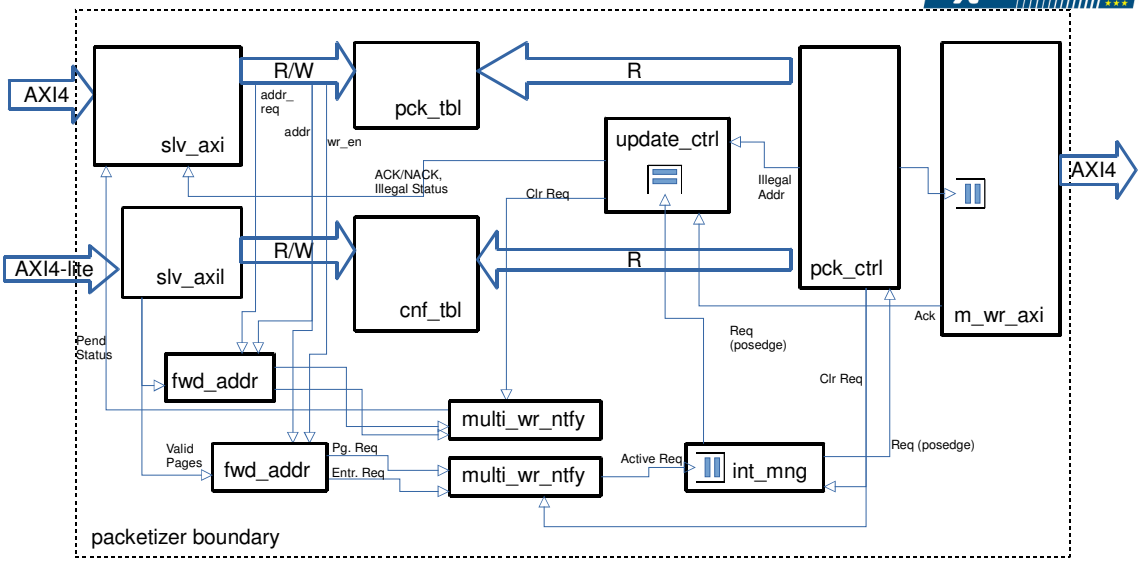

Figure 19: Packetizer HW Block Diagram.

5.2.6 Register space

This section depicts in table format the register space of the TIF and PIF respectively (Table 4 and Table 5). The Offset column is written in hex format, the Valid column shows the actual bits used by the hardware, the Actual column depicts the actual size of data seen by the software, and the AT describes the access type. R: Read, W: Write, WC: Write Clear (a single write to the corresponding register, clears the register).

Offset	Description	Valid	Actual	AT
0x00 (Entry 0) *0x08 (Entry 1)* *0x10 (Entry 2)* . *0x1F8 (Entry 63)*	Table memory of PIDs. For i.e., writing to the offset 0x08 the value 0x123456CABEBE, the second page of packetizer (PIF#1) will use the 0x123456CABEBE value as PID field of the packet.	[47:0]	[63:0]	R/W
0x200	**BASE_DEST**: This is a register. Keeps the base_address of the destination addresses.	[33:18]	[63:0]	R/W
0x208	**PID_CONF:** This is a register. Each bit corresponds to a valid PID. If none of the bits are set, (all zeros) any write to the PIF, will not correspond to any packet initiation.	[63:0]	[63:0]	R/W
0x210	**CLR_REQ**: This is a register. Clears any active requests of the corresponding PIF (Each bit corresponds to a PIF page) This should be used for debug only.	[63:0]	[63:0]	WC

Table 4: TIF Register Space.

Offset per Pg.	Description	Valid	Actual	AT
0x00	Payload0 data to be transmitted.	[127:48]	[127:0]	R/W
0x10	Payload1 data to be transmitted.	[127:0]	[127:0]	R/W

0x20	Destination address. This address is checked with [33:18] bits of **BASE_DEST** register. If check is ok, the packet is transmitted, else, **ILLEGAL** status is set.	[39:0]	[127:0]	R/W
0x30	**PEND**: Bit0. If asserted, the packet is under preparation/transmission. No ack has been received. **ILLEGAL**: Bit1. The packet was about to be transmitted but the address given by the SW was illegal. **NACK**: Bit2. A negative ack has been received for the transmitted packet. Becomes zero again upon the next packet transmission. **ACK**: Bit3. A positive ack has been received for the transmitted packet. Becomes zero again upon the next packet transmission. **OVERWR_BUF**: Bit4. Debug bit, if asserted, a write to a PIF has been initiated during packet preparation.	[4:0]	[127:0]	R

Table 5: PIF Register Space.

6 Concluding remarks

In this deliverable, we have described the firmware and operating system support developed at FORTH to support the Unimem architecture on the current-generation ExaNoDe multi-board prototype. We present our design and implementation of a global shared address space (GSAS) and its communication mechanisms. We describe the GSAS architecture in detail, while giving a brief overview of the hardware and software components that it is based on. We also provide a description of the hardware-software interface of the components co-designed for use by the GSAS environment. We also describe two additional building-block capabilities available in our prototype: user-space initiated DMA and mailbox notifications. The DMA engine of the underlying system is capable of transferring to/from any memory location throughout the entire global memory space. A main goal of our effort is to efficiently and conveniently expose the functionality of the DMA engine to applications running in user space. Furthermore, we describe the custom mailbox mechanism with which a kernel- or user-space application can send and receive messages to and from remote nodes, thus offering a low-latency remote notification capability. Finally, we provide an overview of the Sockets over RDMA feature. With Sockets over RDMA, a Unimem system can utilize low-latency communication among local nodes, by means of fast RDMA transactions, bypassing the kernel-space network stack.

Starting from M6, FORTH is offering a remote access facility to the current multi-board prototype, via a web-based reservation system (implemented using the Apache Virtual Computing Lab platform). Figure shows the structure of the current multi-board prototype, consisting of four ARM Juno/R2 development systems that are interconnected via both Ethernet and a custom interconnect based on high-speed serial links (as delivered by T5.1).

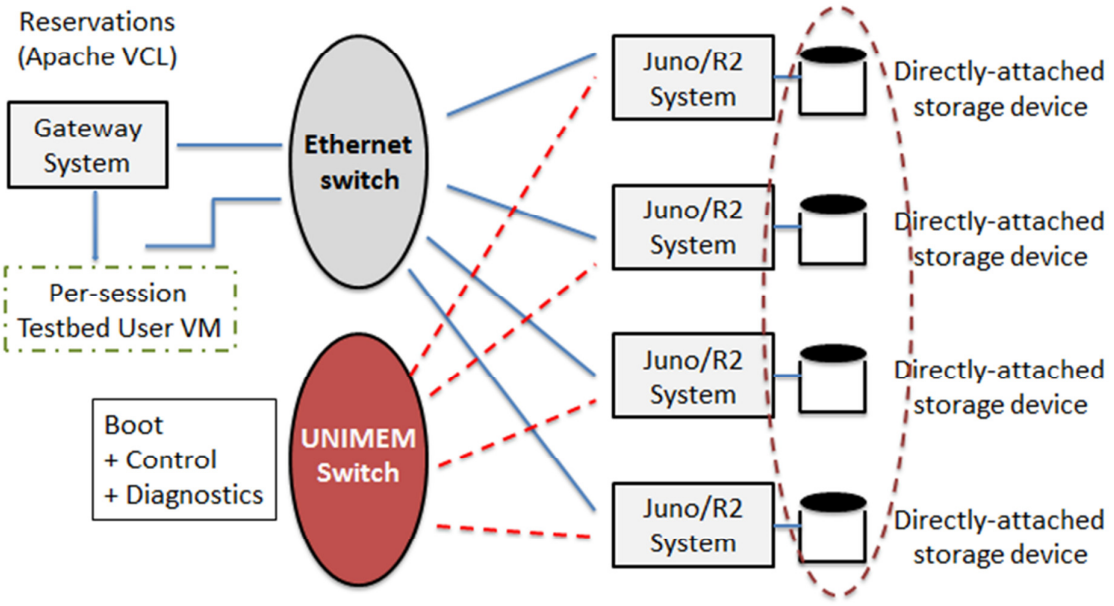

Figure 20: Overview of current multi-board prototype, with reservations and remote access gateway.

Partners of ExaNoDe WP3 (namely: BSC, University of Manchester, FHG) have been using the current ExaNoDe prototype via this remote access facility, working towards adapting their respective run-time environments (OmpSs, OpenStream, GPI/GASPI) to match the Unimem memory architecture.

For the next reporting period, we will work to provide the features and services described in this document in the upcoming prototype of ExaNoDe, following the timeline of the ExaNoDe DoA. Moreover, we will work towards the following enhancements:

- GSAS: We plan to incorporate more advanced protection features for the running applications. More specifically, an adversary or a faulty application will not be able to access (i.e., read or write) the shared data stored on GSAS of any other application that uses the functionality of the GSAS environment.
- Memory management services: We will offer additional interfaces for managing remote memory regions, including policies for selecting whether to allocate memory from the local coherence island or from "affiliated" remote ones.
- RDMA and mailbox services: We are working towards more sophisticated support for virtualization at the level of hardware resources such as RDMA engines and mailboxes, specifically considering alternatives for the hardware-software interfaces and the necessary infrastructure for allowing user-space access to such resources with minimal or even zero copies of data from user- to kernel-space.
- Sockets-over-RDMA: We will add support for kernel-space interception of socket-related functionality, so that we can transparently support not only user-space applications and services (as in the current prototype) but also kernel-space network-based services (such as network filesystems).
- NUMA support in Linux for ARMv8: We have started an effort towards providing NUMA-awareness in the Linux kernel for the Unimem memory architecture, allowing applications that use the standard libNUMA API to use remote memory regions in our prototype. Our OS adaptations are aiming to support NUMA-aware Linux instances running on each of the coherence islands. This work requires patches to the architecture-specific source files of the Linux tree, so that a Linux kernel instance running on one of the coherence islands can be configured to recognize and utilize remote memory regions offered by other coherence islands as NUMA nodes. Mainstream kernels for ARMv8 systems do not support NUMA, offering only a SMP view of the platform. With our work, we will provide support for dynamic memory management and user-level socket-based communication across servers using RDMA.

As stated in the DoA, OS development and refinements have started on the Unimem system prototype developed within the EUROSERVER project. By M9, this effort has moved to the ExaNoDe multi-board prototype from WP5 (output of T5.1), in close collaboration with the WP5 hardware developers. The operating system environment developed and tested on these two early hardware platforms will later be adapted and evaluated on ExaNoDe's final hardware platform.

7 References

1. **EuroServer.** http://www.euroserver-project.eu.

2. **ExaNest.** http://www.exanest.eu/.

3. *EUROSERVER: Energy efficient node for European micro-servers.* **Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, et al.** Verona, Italy : IEEE, 2014. Euromicro Conference on Digital System Design (DSD). pp. 206-213.

4. *The SGI origin: A ccNUMA Highly Scalable Server.* **J. Lenosk, and D. Laudon.** s.l. : ACM, 1997. ACM SIGARCH Computer Architecture News. pp. 241-251.

5. **Juno ARM Development Platform.** http://www.arm.com/products/tools/development-boards/versatile-express/juno-arm-development-platform.php.