D3.7
Operating System Support for ExaNoDe

Workpackage:	3	Enablement of Software Compute Node
Author(s):	Nikolaos D. Kallimanis	FORTH
	Manolis Marazakis	FORTH
	Manolis Skordalakis	FORTH
Authorized by	Manolis Marazakis	FORTH
Reviewer	Pierre-Yves Martinez	CEA
Reviewer	Kevin Pouget	VOSYS
Consolidating Reviewer	Elyes Zekri	ATOS
Dissemination Level	PU	

Date	Author	Comments	Version	Status
2017-07-19	N. D. Kallimanis	First version of the document, with section outlines.	v0.1	Draft
2017-07-28	N.D. Kallimanis	Material for the 2nd-generation of the GSAS environment added.	v0.2	Draft
2017-08-02	E. Skordalakis	Material for the KRAM remote swap space added.	v0.3	Draft
2017-08-28	M. Marazakis	Material for the NUMA support. Enhancements on Section 3.0.	v0.4	Draft

2017-08-31	N. D. Kallimanis, M. Marazakis	Document ready for internal review.	v0.5	Draft
2017-09-08	N. D. Kallimanis, M. Marazakis	Editorial changes, inclusion of an overview figure	v0.6	Draft
2017-09-09	N.D. Kallimanis	Editorial changes.	v0.7	Draft
2017-09-10	N. D. Kallimanis, M. Marazakis	A few editorial changes.	v0.8	Release to ExaNoDe reviewers
2017-09-25	N. D. Kallimanis, M. Marazakis	Revision based on comments by Kevin Pouget, Pierre-Yves Martinez, and Elyes Zekri.	v0.9	Draft
2017-09-26	N. D. Kallimanis, M. Marazakis	Final edits.	v1.0	Final

Executive Summary

In this deliverable, we describe the evolution of firmware and operating system support that we have developed to support the Unimem architecture on the current-generation ExaNoDe multiboard prototype. We focus on the extensions that we have designed and implemented in two major areas of functionality:

(a) The 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.

(b) The use of remotely-accessible memory as a swap device, to augment the memory space available to applications executing on a Unimem compute-node. This is one of several use-cases for remotely-accessible memory that we have explored. Other use-cases briefly described in this deliverable include using memory from remote nodes via allocator modules, both in user-space and in kernel-space, remote memory.

Finally, we describe our work towards adding NUMA (Non-Uniform Memory Access) support in the operating system.

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-scale 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.

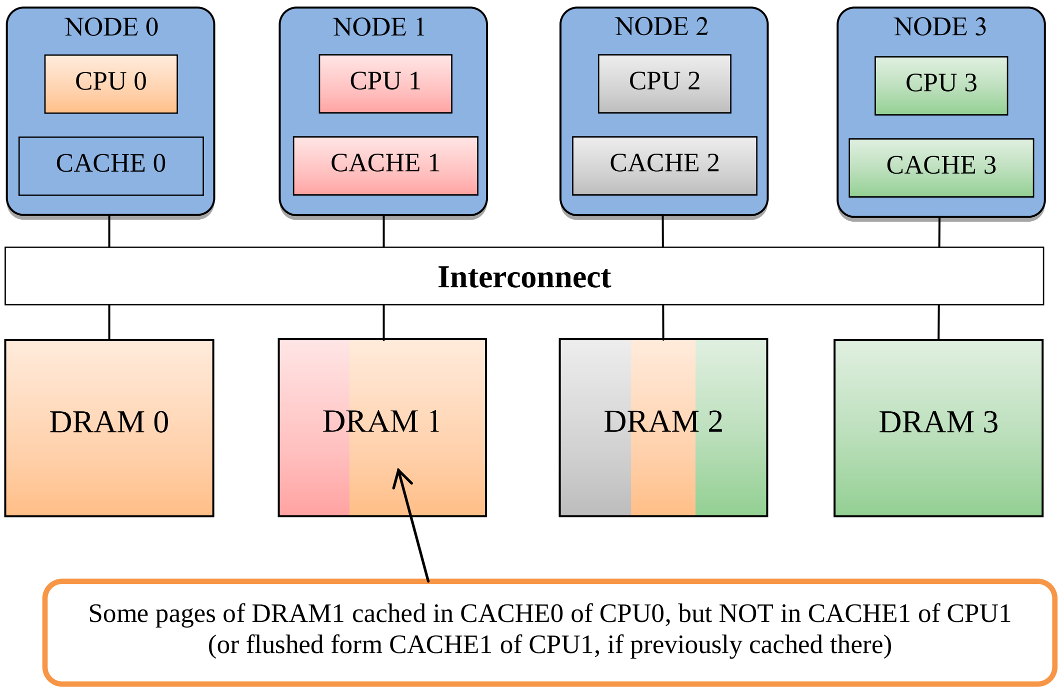
NODE 0
CPU 0
CACHE 0
NODE 1
CPU 1
CACHE 1
NODE 2
CPU 2
CACHE 2
NODE 3
CPU 3
CACHE 3
Interconnect
DRAM 0
DRAM 1
DRAM 2
DRAM 3
Some pages of DRAM1 cached in CACHE0 of CPU0, but NOT in CACHE1 of CPU1
(or flushed form CACHE1 of CPU1, if previously cached there)

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). Significantly, this type of communication has the advantage that it is performed by a separate, dedicated hardware (DMA) engine. This engine executes communication primitives in parallel with the main processor performing other, overlapping computations giving the main processor the ability to execute other, overlapping computations.
4. Unimem also offers the mailbox hardware primitive, which gives processes that reside on remote nodes the ability to receive 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.

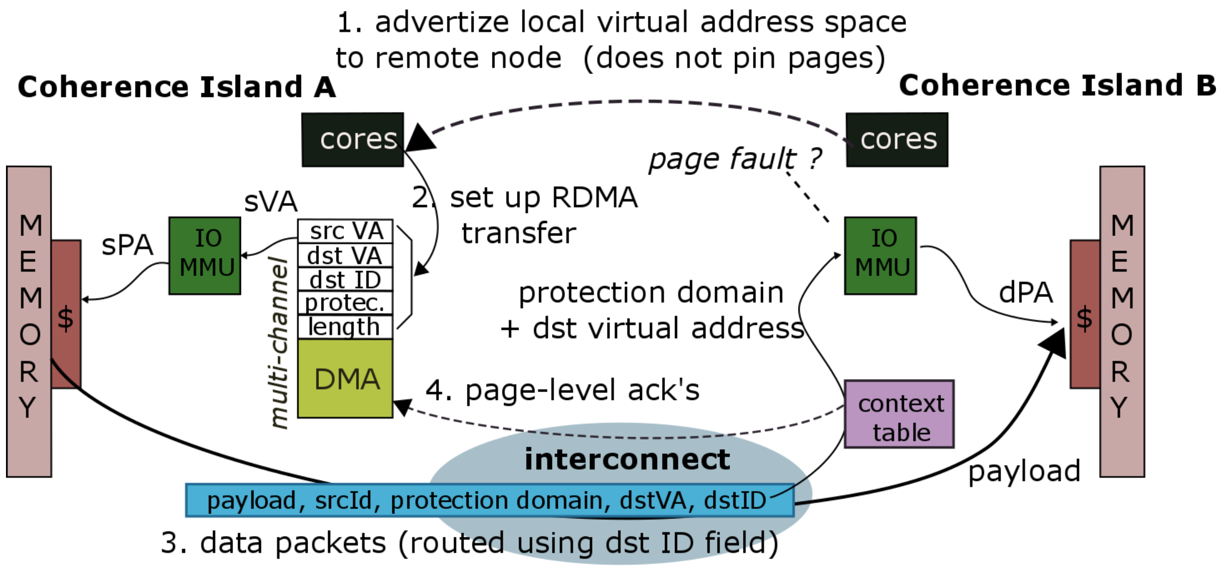

Figure 2. Overview of an RDMA Operation.

Thanks to these abilities, Unimem appears 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 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 in parallel. Therefore, the 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 EUROSERVER (1). Each node is based on the Trenz ARM development platform (5), and it has the following key properties:

- It consists of several processing cores (up to 4), 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 power 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 mail-

boxes. 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.

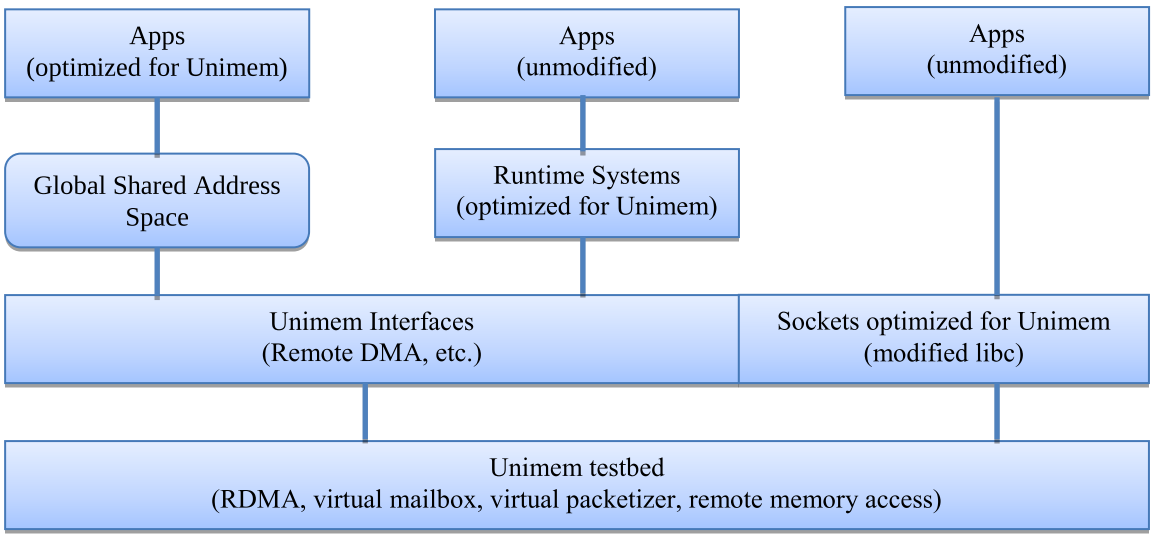

Figure 3. The programming interfaces that the Unimem provides and their interactions.

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 and the communication mechanisms that it provides. The GSAS environment defines an application interface (API) that is an extension of the GAS that the Unimem architecture provides. GSAS gives the ability to processes running 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 several atomic operations (i.e., read, write, Compare-And-Swap, etc.), on the allocated space by using the appropriate library calls that the environment provides.

2. A library that enables user-space initiation 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.

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 that bypass the kernel network stack. Furthermore, we give a description of the custom mailbox mechanism with which a kernel-space 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 the GSAS environment and the provided mechanisms for inter-process communication across different nodes (i.e. Trenz prototype nodes).

2.1 GSAS architecture overview

Our global shared address space defines an application interface (i.e. API) that gives the ability to processes running across remote nodes to communicate in a way that resembles shared memory communication. More specifically, the GSAS environment allows the applications to allocate/de-allocate remote parts of the shared address space and to perform reads, writes and other atomic operations on the allocated space 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 and do not involve operating system's kernel, they have a low-latency, fast communication mechanism. Note that in large scale systems, 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 microseconds, while the order of magnitude of a user-level library call is a few nanoseconds. Therefore, the overhead of a call to a user-level library is minimal.

Although the communication mechanism of the GSAS environment 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.

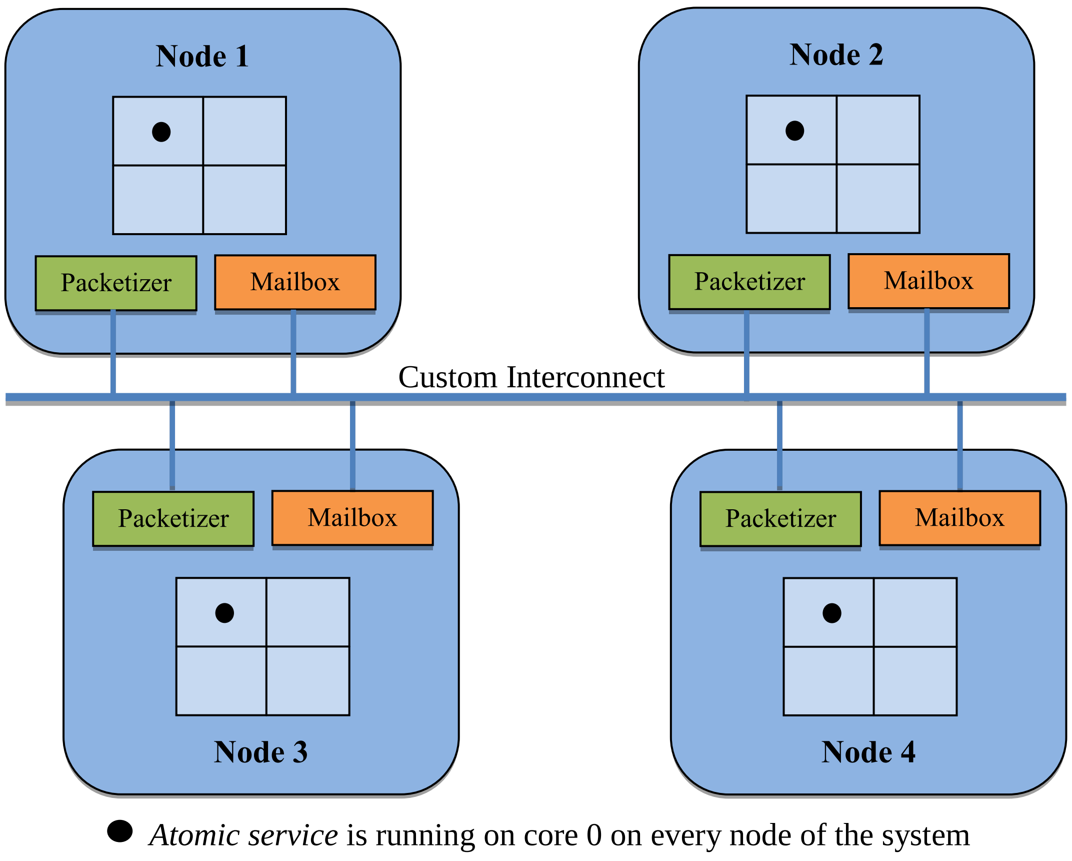

Figure 4. A high-level overview of the architecture of the hardware prototype consisting of 4 nodes.

The current version of the prototype consists of a set of computing nodes (i.e., Trenz development boards) that are connected through a custom interconnection network. Each of the computing nodes contains 4 ARM Cortex-A53 processing cores running at 1.2 GHz and is equipped with 2 GB of DDR4 RAM clocked at 600 MHz. 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 4 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 thread is able to receive a network packet using one allocated interface of the local mailbox. A 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 having to take into account that other threads may access the same hardware blocks concurrently. Thus, any thread is able to directly use the functionality provided by the virtualized packetizer and mailbox hardware blocks without involving the 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 5 shows a brief description of the hardware and software stack.

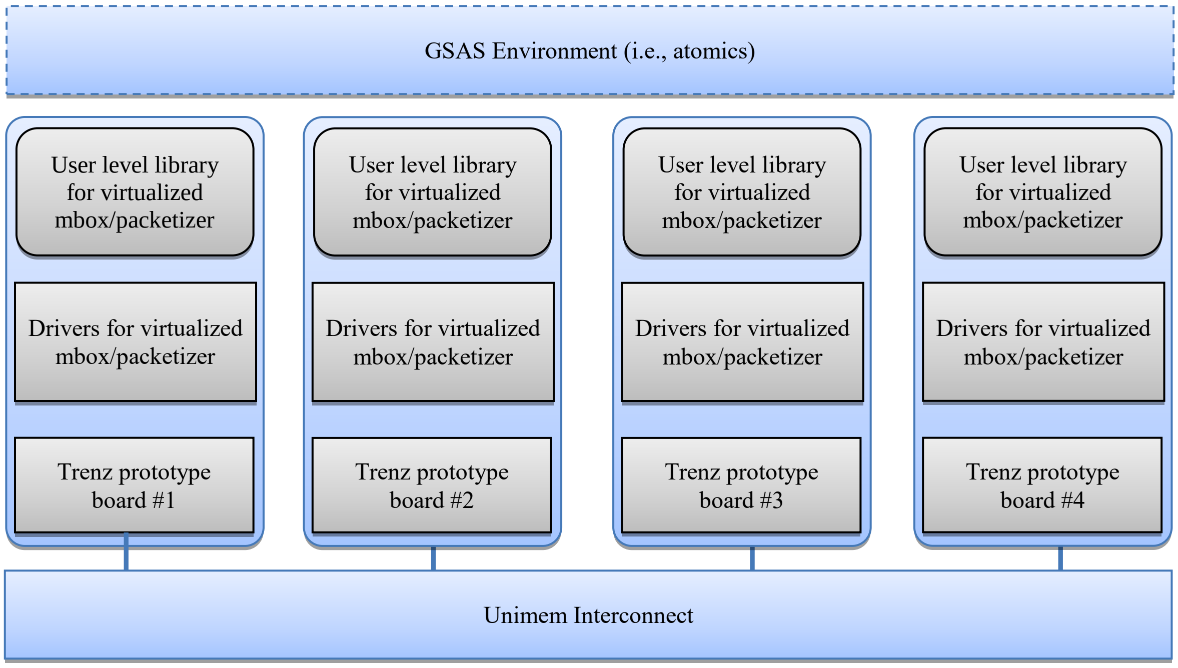
GSAS Environment (i.e., atomics)
User level library for virtualized mbox/packetizer
Drivers for virtualized mbox/packetizer
Trenz prototype board #1
Trenz prototype board #2
Trenz prototype board #3
Trenz prototype board #4
Unimem Interconnect

Figure 5. The hardware and the software stack for the GSAS environment.

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

The GSAS environment supports addresses of 64 bits. The address space consists of N = 2^{16} partitions (see Figure 6). 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 this 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 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 which node this address belongs to. The remaining 48 bits are available to each node for internal addressing of all of its local memory. The address space that is handled by a node is partitioned in pages, where 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.

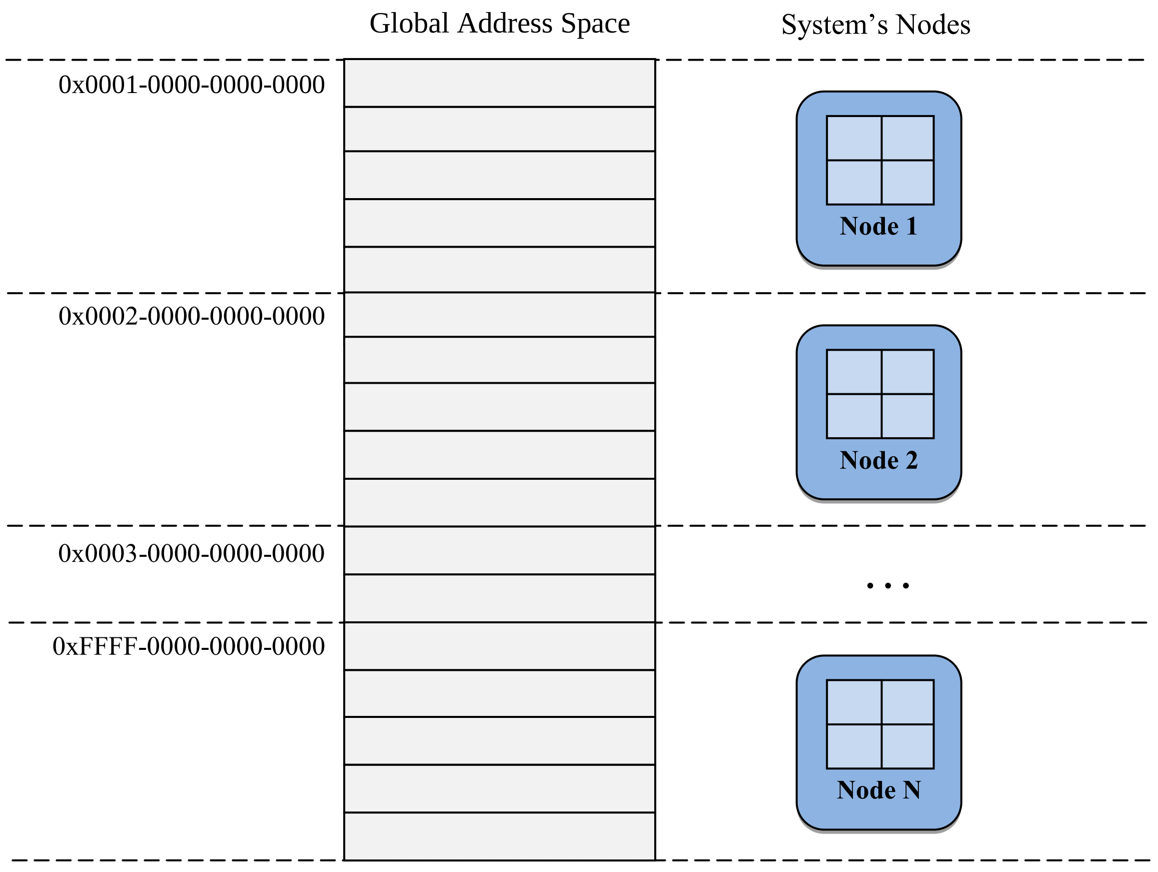

Figure 6. An overview of the addressing system that the GSAS environment offers (Addresses in Hex).

2.2 Protection on GSAS environment

In this Section, we present in detail the protection properties (see Section 2.2.1) that the GSAS environment imposes to the applications running on top of it. The implementation of the protection properties is discussed in Section 2.2.2.

2.2.1 Protection properties

We first describe the protection properties that are imposed to any multi-threaded application that runs on the GSAS environment.

Since each thread on the GSAS environment allocates its own virtual mailbox and packetizer channel, we refer to threads and not processes in our model. Let P some thread that executes the binary code of some file A. Denote by $P \rightsquigarrow P'$, the creation of a local or a remote thread P' by thread P. Thread P' could execute either the same binary of code A or any other binary code $B \neq A$. Denote by $T(P)$, the tree of threads that P resides on. Let P_r be any thread of the tree of threads $T(P_r)$. We say that P_r is the root node of tree $T(P_r)$, if there is not exist some thread P such that $T(P) = T(P_r)$ and $P \rightsquigarrow P_r$.

Denote by σ_k any sequence of $k > 0$ threads (let them be $P_0, P_1, \ldots, P_{k-1}$) such that $P_0 \rightsquigarrow P_1 \rightsquigarrow P_2 \rightsquigarrow \cdots \rightsquigarrow P_{k-1}$. We say that P is an *ancestor* of P', if there is a non-empty sequence σ_k of k>0 threads such that $P_0 \rightsquigarrow P_1 \rightsquigarrow \cdots \rightsquigarrow P_{k-1} \rightsquigarrow P'$ and $P_0 = P$.

Let Q and Q' be any two threads that are running in the system. It holds that $T(Q) = T(Q')$, if there is a thread P_r such that P_r is an ancestor for both Q and Q'. Therefore, both threads Q and Q' are members of the same tree of threads.

Denote as $mem(P)$ any *allocated chunk* of memory of some thread P. Denote by $P \rightarrow mem(P')$, the ability of thread P to execute atomic operations on any memory chunk allocated by thread P'. Denote also by $P \nrightarrow mem(P')$, the inability of thread P to execute atomic operations on any memory chunk allocated by some thread P' (i.e. process P' terminates its execution abnormally if tries to execute an atomic operation in such memory chunk).

Let Q and Q' be two threads that run on the GSAS environment, the following invariants hold:

i. $Q \rightarrow mem(Q)$.

ii. If $T(Q) = T(Q')$, then it holds that $Q \rightarrow mem(Q')$ and $Q' \rightarrow mem(Q)$.

iii. If $T(Q) \neq T(Q')$, then it holds that $Q \nrightarrow mem(Q')$ and $Q' \nrightarrow mem(Q)$.

Figure 7 and Figure 8 show a few examples of the protection properties (i.e., invariants i-iii) that are imposed to the applications running on the GSAS environment. Figure 7 illustrates a tree T of threads. Threads $P1$ to $P3$ (i.e. illustrated as blue nodes) execute the same binary file A, while threads $P4$ to $P6$ (i.e. illustrated as orange nodes) execute some other binary file $B \neq A$. Since all threads $P1$ to $P6$ are members of the tree T, any thread is able to access any other memory chunk allocated by some other thread of T. For example, Figure 7 shows that P_5 is able to access any memory allocated by P_2 and P_4 without having to take into account the binary file that P_2 and P_4 execute. Moreover, Figure 7 shows that P_1 is able to access any memory chunk allocated by P_4, without also having to take into account the binary that P_4 executes.

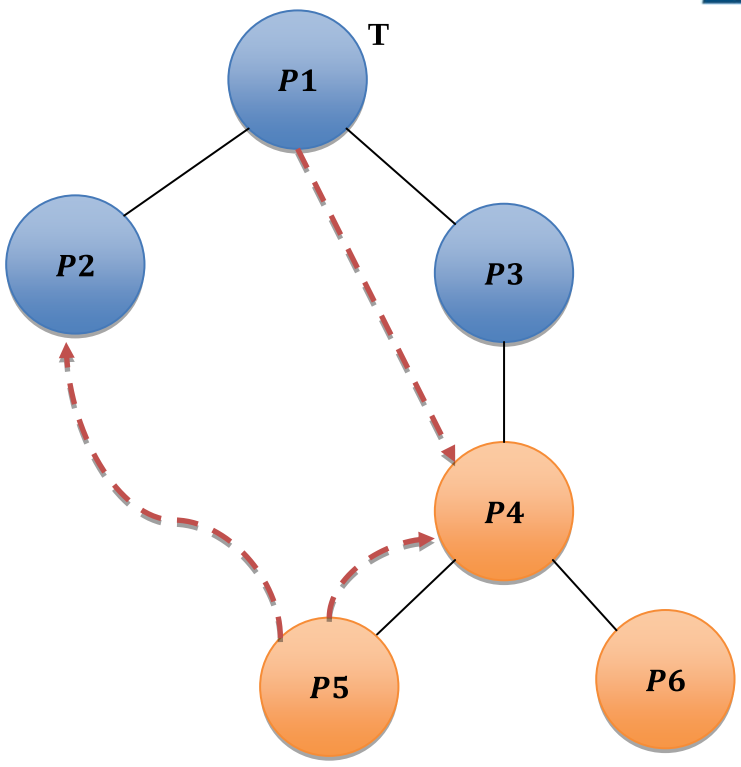

Figure 7. An example of an imaginary tree of *remoteFork* calls. Threads $P1 - P3$ are instantiated by executing binary A, while threads $P4 - P6$ are instantiated by executing binary B.

Figure 8 illustrates two trees of threads T and T'. Threads $P1$ to $P3$ and $P1'$ to $P3'$ (i.e. illustrated as blue nodes) execute the same binary file A, while threads $P4$ to $P6$ (i.e. illustrated as orange nodes) execute some other binary file $B \neq A$. By invariants i-iii, it follows that all threads $P1$ to $P6$ are members of the tree T, any thread is able to access any other memory chunk allocated by some other thread of T. Furthermore, invariants i-iii imply that any of threads P_1 to P_3 is not able to access any chunk of memory allocated by threads P_1' to P_3', ignoring the fact that both set of threads execute the same binary file. By invariants i-iii, it is also implied that threads P_1' to $P_3{}'$ cannot access any chunk of memory allocated by any thread P_1 to P_6 of tree T.

2.2.2 Implementation of protection in GSAS environment

In the GSAS environment, there are two kinds of entities in terms of protection, i.e. the *trusted* entities and the *untrusted* entities. The trusted entities are the following:

1. Hardware entities, i.e. Packetizer and Mailbox.
2. The atomicity driver that handles the Packetizer and Mailbox hardware entities.
3. The atomic service that runs on each system's node

The untrusted entities are the following:

1. Any thread of a user-space application.
2. The user-space library that handles the atomic requests on the issuer side.

The key architectural design-point on the GSAS environment is that the trusted entities provide a restricted way for communication to the untrusted entities (i.e. the GSAS environment users). Moreover, only the trusted entities are able to set-up/modify the protection ids. This results in a protection scheme that guarantees that a thread using the functionality of the GSA environment cannot impersonate some other thread. This gives the ability on threads that run on top of the GSAS environment to be able to access the memory areas that are implied by the protection model described on Section 2.2.1.

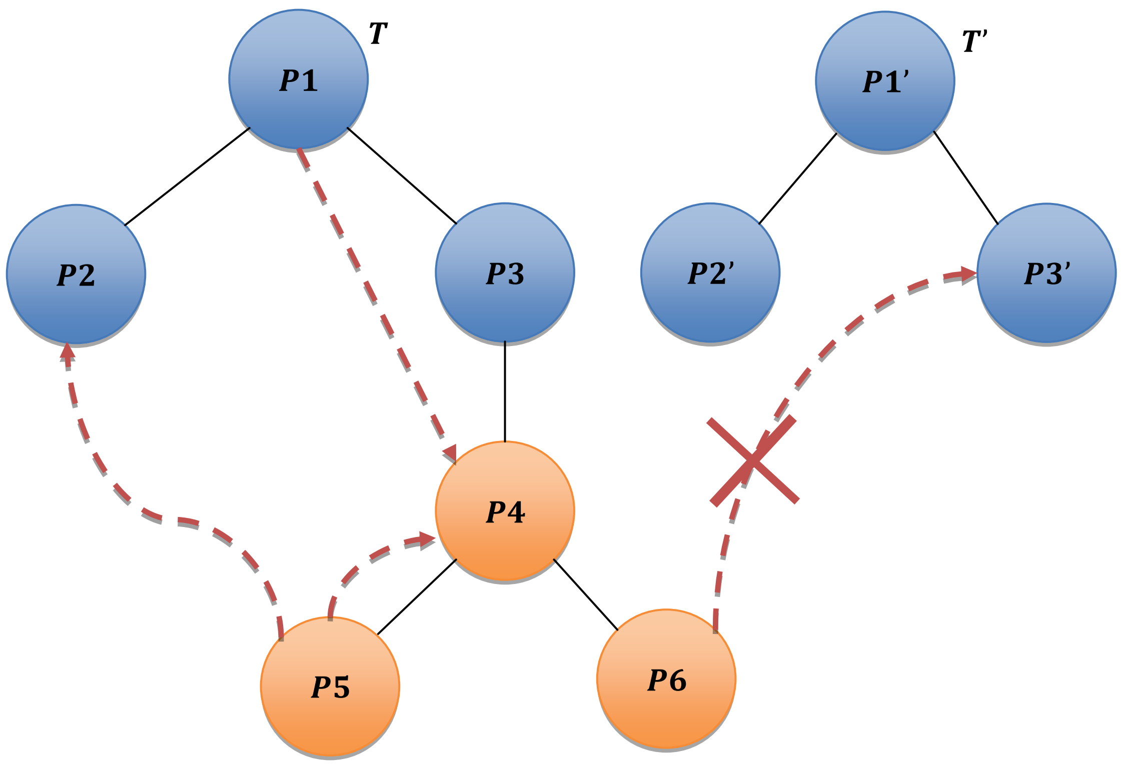

Figure 8. Thread P6 is not able to access the memory allocated by thread P6'.

We now describe how the protection scheme of the GSAS environment is implemented. As already pointed out, each atomic operation issued by some thread is described by a network packet that is transmitted to the atomic service of the appropriate remote node (see Section 2.1 for a more detailed description). All the packets that describe atomic operations consist of a header of 56 bits and a payload of 200 bits. The first field of the header is of size 24 bits and uniquely describes the sender of the packet (see Figure 9). This field consists of the interface id (each virtual interface of some has a unique id on the local board) and the board id (each board in the system has a unique id). The next field also consists of 24 bits and is the protection id of the thread. The protection id is a unique number that share all the processes of the same tree of threads (see Section 2.2.1 for a formal definition of the tree of threads). Every thread at its creation time is assigned a protection id that is equal to its thread id and therefore, it is unique. The payload is written by the user-space library that issues atomic operations. The role of the payload is to describe the destination, the arguments and the type of the atomic operation that will be applied. Since the user-space library that issues atomic operations could be sided over by an erroneous or malicious user application, it is not a trusted entity and thus, it is not allowed to write the first 56 bits that are important for the protection scheme of the GSAS environment. In contrast to packet's payload, the header of the packet (the first 56 bits) is written only by the Packetizer hardware block.

Each thread that is running on the GSAS environment allocates by requesting atomicity driver, a pair of virtual interfaces, one for the Mailbox and one for the Packetizer. The atomicity driver is responsible for setting-up the thread id and the protection id to the allocated virtual interface of the Packetizer. Whenever a thread allocates a pair of Packetizer and Mailbox hardware blocks, the atomicity driver set-ups the Packetizer hardware block in way that the Packetizer interface writes the thread and protection id of the thread on every packet that is transmitted over the network. In case that this thread is the first thread of the application that allocates the pairs of Packetizer and Mailbox interfaces, **its thread id and its protection id are the same** (the case that this thread is spawned by some other thread is described in a later part of this section). Therefore, any thread (and also the atomic service) that receives a packet from some remote thread, it has the ability to safely identify the origin of the packet. Since the atomic service bookkeeps the owner (i.e. the protection id) for each allocated page, it is able to safely deny to respond on requests to memory issued by threads that are not allowed to.

0	24	48	56	64 … 127
Thread & Node ID	Protection ID	Reserved	OP CODE	Instruction Address
Argument 1				Argument 2

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

Figure 10 shows a detailed description of the packet's flow. More specifically, some thread on node A issues an atomic operation by transmitting a network packet for a memory location that resides on node B. Since only the payload of the transmitted packet is written by the issuer thread (an untrusted entity), the header that contains the information of thread and protection ids is written by the Packetizer hardware block. Thus, the receiving endpoint, which is the atomic service on node B, is able to safely identify the sender thread that resides on node A. In case that the issuer thread of node A tries to apply an atomic operation on some memory location that it is not allowed to on node B, the atomic service on node B rejects the packet describing this operation.

For allowing two or more remote threads accessing the same chunks of memory, the GSAS environment provides special functionality to an application for spawning remotely threads, i.e. *remoteFork* operation (see (6) for a detailed description).

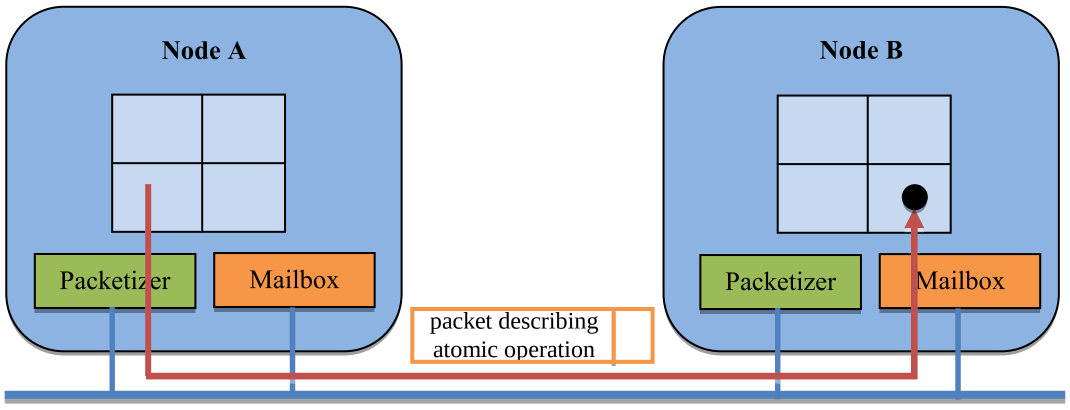

Figure 10. The protection architecture of the GSAS environment.

This functionality not only eases the creation of threads on remote nodes, but it also enables the use of the same protection id for threads that reside on different nodes. Whenever a thread wants to spawn a thread (or process) to some remote node, it calls *remoteFork* giving as arguments the node id (i.e. in which node the process should be spawned on) and the path of the binary file that the remote thread should execute. *remoteFork* creates a packet that describes the spawn operation and it sends it to the atomic service of the remote node. Whenever, the atomic service of the remote node receives the packet that describes the spawn operation, it creates a new thread. The newly created thread initially owns a protection id that is equal to its own thread id (the standard behaviour of atomicity driver when a thread allocates a pair of

Mailbox and Packetizer interfaces, is to assign a protection id to the thread equal to its id). Afterwards, the atomic service changes the protection id of this newly created thread to an id that is equal to the id of the thread that issued the spawn operation. Recall that the atomic service extracts the id of the issuer thread from the header of the received packet. Thus, it can safely set the protection id of the newly created thread to match the protection id of the issuer thread. This gives the ability to one or more threads (i.e. thread that are in the same tree of threads) to access the same chunks of memory.

2.2.3 Restrictions on protection

In the current version of the GSAS environment, the following restrictions on the implementation protection scheme hold.

- Let T be any tree of threads (see Section 2.2.1) and let p_r be its root thread. In the current implementation of GSAS environment, *invariants i-iii hold only if thread p_r waits for all of its descendant threads to finish their execution*. Thus, for any application that wants to strictly follow the protection model of Section 2.2.1, the root thread of the tree of threads should wait (e.g. on a barrier, etc.) all the spawned threads to finish their execution.
- At the current protection model, two threads that belong to different trees of threads are not able to communicate using the mechanism provided by the protection scheme of the GSAS environment. In future versions of the GSAS environment, we aim to provide some forms of communication of threads that belong to different trees of threads.

2.3 Application interface extensions

In this section, we provide a detailed description of the new functionality that the GSAS environment offers to the user applications. In this version of the GSAS API (i.e., v1.1), we provide some new operations that are able to read or write shared memory variables in chunks of 128 bits and/or 256 bits. By providing these new operations, a user application is able to access larger chunks of remote memory areas (Read and Write operations of 128 or 256 bits) with a single atomic operation, resulting significant performance gains.

- **uint128_t READ128(uint64_t *addr)**: *READ128* returns the current 128-bit value that is stored in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.
- **uint64_t READ256(uint64_t *addr, void *buffer)**: *READ256* stores on local memory pointed by *buffer*, a vector of 256-bit values that are stored 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 64 bit words *val* in address *addr*. In case that *addr* is an invalid pointer, application exits abnormally.

2.4 GSAS performance evaluation

We evaluated the performance of GSAS environment in the Trenz prototype. The Trenz prototype consists of 4 nodes (i.e. 4 Trenz boards), each being equipped with 4 A53 processing cores. The atomic service is pinned at core zero of each of the nodes. We used the gcc 4.9.2 compiler. The operating system was Linux with kernel version 4.4.0. To prohibit the linux scheduler from doing unnecessary kernel thread migrations, threads were pinned in all experiments: the i-th thread was bound on core (i+1)-th core.

For our experiments, we first consider a synthetic benchmark, called the Fetch&Add benchmark, which is similar to that presented in (7). In this benchmark, we measure the maximum number of atomic operations (i.e., Fetch&Add[1] operations) could be executed on a single, remote memory location. The Fetch&Add object is simple enough to exhibit the overheads of the GSAS environment induced by the consecutive execution of atomic operations on a single, remote memory location issued by large number of threads. Each instance of the benchmark simulates 10^8 Fetch&Add operations, in total, with each of the n threads simulating $10^8/n$ Fetch&Add operations out of the total Fetch&Add operations. For the measurement of the average throughput, each experiment is executed 10 times. The remote shared variable was allocated using the functionality of *AllocSharedPage*, and it was located on node with id equal to 1.

Figure 11 shows the performance of the GSAS environment for different number of nodes and different number of threads per node. This benchmark shows that the best throughput is achieved, when the Fetch&Add operations are issued by the local node where the memory location resides on. More specifically, the maximum achieved throughput is about 600k operations per second, and it is achieved by the use of either 2 or 3 threads. By increasing the number of nodes that issue Fetch&Add operations, the throughput gracefully drops at about 400k operations per second for the case of 4 nodes.

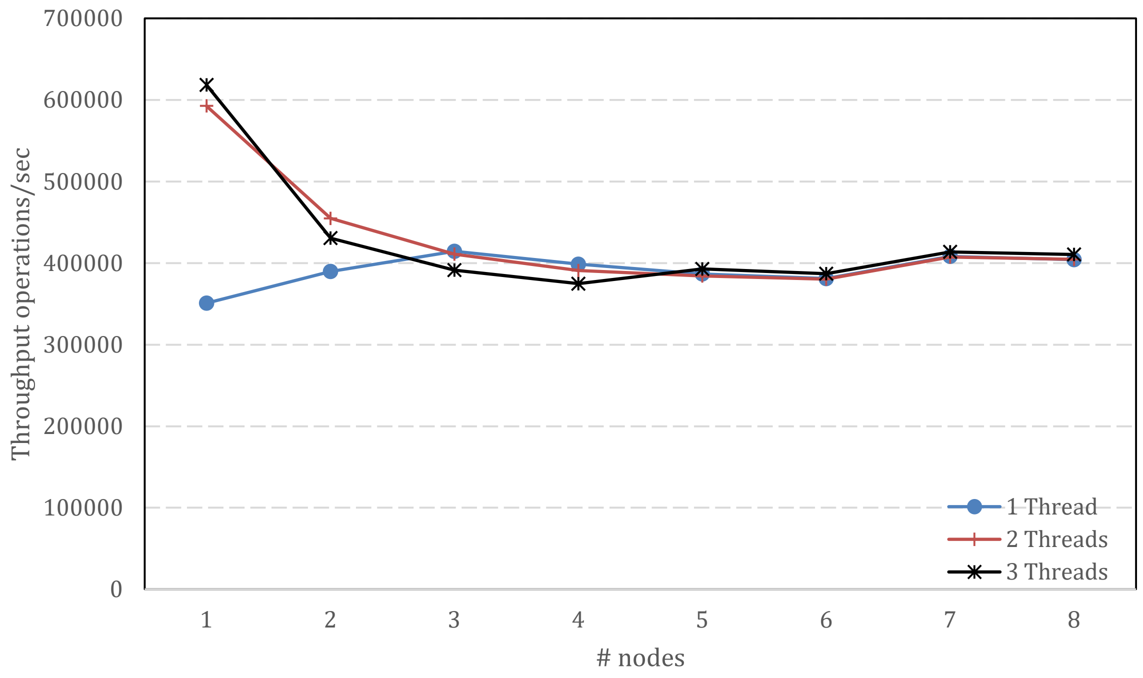

Figure 11. Average throughput of a Fetch&Add operation on a single variable for different numbers of nodes and threads.

In the next experiment, illustrated on Figure 12, we measure the throughput of Fetch&Add operations for different numbers of shared variables and not for a single remote location. More specifically, x-axis shows the number of the nodes that the experiment is performed, while the y-axis shows the achieved throughput. The remote variables are allocated evenly on

[1] A Fetch&Add atomically adds some (positive or negative) value to some memory location and returns the previous value.

all system's nodes and each thread randomly choses where to perform a Fetch&Add operation. This benchmark aims to present how the performance of the GSAS environment scales under load on many different remote locations. In the experiment of Table 1, we present the latency of an atomic operation (i.e. a Fetch&Add operation). More specifically, we ran a benchmark where a single thread running on node with id equal to 1 issued 10^8 Fetch&Add operations on a single, remote variable. This variable was allocated by using the *AllocSharedPage* functionality provided by the GSAS environment. We run 4 instances of the benchmark; in the first one, the remote variable was allocated on node 1 (i.e. local node), in the second one, the remote variable was allocated on node with id 2, etc. Again, for each of the configurations, the experiment was executed 10 times and average execution time was taken. We also repeated the above experiment by running the thread that issued the Fetch&Add operations on different nodes than 1.

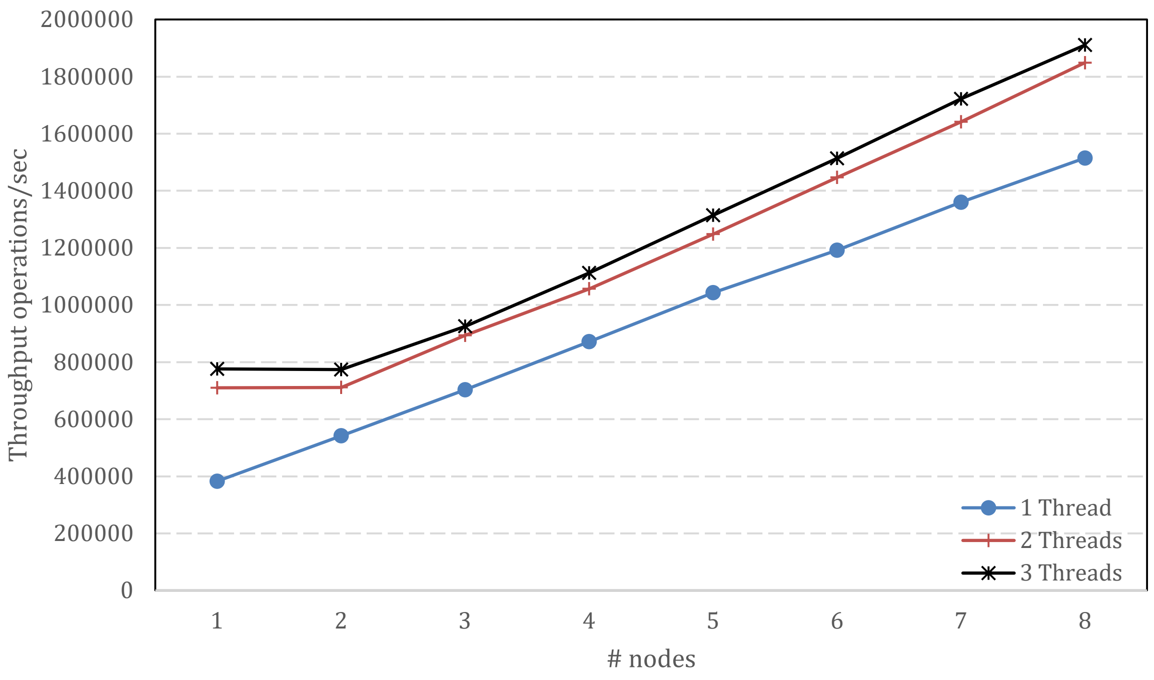

Figure 12. Average throughput of Fetch&Add operations on many different variables for different numbers of nodes and threads.

From/to	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Node 1	2.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6
Node 2	4.6	2.6	4.6	4.6	4.6	4.6	4.6	4.6
Node 3	4.6	4.6	2.6	4.6	4.6	4.6	4.6	4.6
Node 4	4.6	4.6	4.6	2.6	4.6	4.6	4.6	4.6
Node 5	4.6	4.6	4.6	4.6	2.6	4.6	4.6	4.6
Node 6	4.6	4.6	4.6	4.6	4.6	2.6	4.6	4.6
Node 7	4.6	4.6	4.6	4.6	4.6	4.6	2.6	4.6
Node 8	4.6	4.6	4.6	4.6	4.6	4.6	4.6	2.6

Table 1. Latency (in μsecs) of a Fetch&Add operation on some remote memory location.

Table 1 shows that the latency is minimal in cases that the thread that issues Fetch&Add operations resides in the same node (i.e. local node) that the remote variable is allocated. In this case, the achieved latency is 2.6 μsec. In cases that the thread issues Fetch&Add operations on variables located on other than local node, the achieved latency increases to 4.6 μsec. Given that each atomic operation, either on the local node or on some remote node, consists of two network messages (i.e. one packet that describe the operation and one response packet, see (6) for more details), it follows that the central switch adds about 1 μsec one-way latency. We aim to reduce this latency significantly in the future implementation of our interconnect.

2.5 A distributed in memory key-value store on GSAS environment

In this section, we present a GSAS use case example, i.e., a Distributed Hash Table (abbr. DHT). A DHT is a concurrent data structure that enables applications to store pairs of $\langle key, value \rangle$ items. It also allows applications to retrieve a $value$ associated with a given key. The presented DHT uses the communication mechanisms provided by the GSAS environment resulting in a DHT implementation that is highly-scalable.

2.5.1 Supported functionality

We first provide a detailed description of the functionality supported by our DHT on top of GSAS environment. In the current version of the DHT application, two operations are supported for storing and retrieving data, i.e. *dhtPut* and *dhtGet*. Description for these two operations follows.

- __uint64_t **dhtPut**(DHT *dht, uint64_t key, __uint128_t value): *dhtPut* inserts the pair of $\langle key, value \rangle$ in the concurrent hash-table. In case that there is already some other pair of values $\langle key', value' \rangle$, where $key = key'$, the old pair of values is replaced by the new one. In case of success, zero is returned. Otherwise, an error code is returned.
- __uint128_t **dhtGet**(DHT *dht, uint64_t key): *dhtPut* searches for a pair of values $\langle k, v \rangle$, such that $key = k$ and returns v. In case that such a pair does not exist, *dhtGet* returns ⊥, which is an invalid value.

2.5.2 DHT implementation

The DHT concurrent data structure that we present in this deliverable consists of two level, i.e. the first and the second level of hashing. The first level of hashing consists of an array of $FIRST\_HASH\_SIZE$[2] elements, each element of the first level array stores 256 bits of information consisting either a <*key, value*> pair or a pointer to a container of the second level of hashing (see Figure 13). In an effort to enforce different requests to be served by memory areas that reside in different nodes, the array of first level hashing is stored across all the system's nodes. More specifically, the first $FIRST\_HASH\_SIZE/N$ elements reside on node 0, the second $FIRST\_HASH\_SIZE/N$ reside on system's second node, etc. This and the use of a hash function that uniformly distributes keys on system's nodes guarantee good load balance and high performance.

A container of the second level of hashing consists also of array of elements similar to that of first level hashing, but its size is much smaller. For performance reasons, each container is of size 4096 bytes matching the page size and the whole container data reside at a single node.

[2] $FIRST\_HASH\_SIZE$ is initialized with a value that is big enough to map the whole system's memory.

Given that each element of a container is of size 256 bits, 128 entries are available for storing $\langle key, value \rangle$ pairs.

There are two different hash functions, one for each level of hashing. Denote by $h_1(k)$ the hash function of level-one of hashing and by $h_2(k)$ the hash function of level-two hashing, where $0 \leq k \leq 2^{\wedge}64 - 1$ is a key. $h_1(k)$ returns a value $0 \leq h \leq FIRST\_HASH\_SIZE$, while $h_2(k)$ returns a value $0 \leq h' \leq 128$.

We first describe the *dhtGet* operations. Let req be any *dhtGet* operation executed by some thread t that requests the value of some key k. At first, req computes the level-one hash value $h = h_1(k)$. h is just a pointer to some element of the first-level array. Afterwards, it atomically reads the h-th entry of the first level array by executing a *READ256* operation (remind that each entry of the array of first-level hashing is of size 256 bits). Depending on the value of $indirect$ and $lock$ fields, there are three cases.

- In cast that $lock$ is equal to $true$, then req spins until the $inderect$ field becomes equal to $false$. Whenever $indirect$ becomes equal to $false$, req follows the code-path of one of two remaining cases depending on the value of the $indirect$ field.
- In case that the indirect field of the entry that was read is equal to false, this entry does not point to a second level container. In this case, t simply compares k with the contents of the *key* field, and returns the value stored in the $value$ field of the entry. Otherwise, t returns $\perp$ (i.e. invalid value).
- In case $inderect$ is equal to $true$, it follows that the entry of the first-level array of hashing points to a container of DHT nodes of the second-level of hashing. In this case, req calculates the hash value of the second-level of hashing, let it be $h' = h_2(k)$. Afterwards, req follows the pointer to the container and reads the h' element of the container by executing a *READ256* operation. In case that the key of this entry is equal to k, req returns the value stored in the $value$ field. Otherwise, req returns a value equal to $\perp$, which is an invalid value.

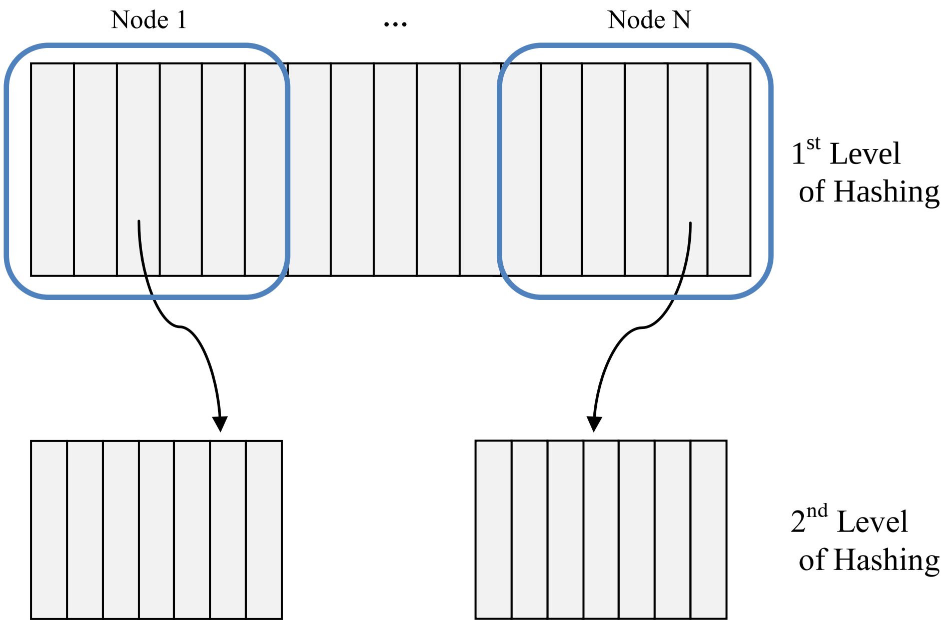

Figure 13. Basic data structures for the DHT application.

Each *DHTNode* structure consists of 5 fields (see Figure 14), i.e. a) the field that points out if the node is indirect or inner node, b) the field that is used as a lock, c) the field that stores a version number of the node, d) the field for the key of the <*key*, *value*> pair, and e) the field that stores the value of the <*key*, *value*> pair. The *key* field is of size of 64 bits and the *value* field is of size of 128 bits.

The role of the indirect/inner field is to show if a node of the first level of hashing is a simple entry of $\langle key, value \rangle$ pair or is a pointer to a container of the second level of hashing. Whenever a *dhtGet* or *dhtPut* operation takes place, it checks if the node in the first level of hashing is indirect or not. In case that the node is direct, it checks the key of the direct node and returns. Otherwise, the *dhtGet* or *dhtPut* operation reads key which is actually in this case a pointer, and starts searching to a container of the 2nd level of hashing pointed by this key. The version field of the *DHTNode* records the version of the node; at each node update, the version number is increased by one.

0	16	32	64 127
indirect	lock	version	key or pointer in case of indirection
Value			

Figure 14. Description of the structure of a *DHTNode*.

2.5.3 DHT performance evaluation

We evaluated the performance of the DHT implementation on the Trenz prototype in a setup very similar to that presented in Section 2.4. In the experiment of Figure 15, we measured the performance of the *dhtGet* operations of our DHT implementation. Specifically, we measured the average throughput of *dhtGet* operations for different numbers of nodes and different numbers of threads per node. DHT implementation was initialized to contain 10^6 different keys by using the *dhtPut* functionality. Similarly to the experiments for the GSAS environment (Section 2.4), the benchmark was executed 10 times and averages were taken. Figure 15 shows that the throughput of *dhtGet* operations increases with the number of nodes and/or threads per node.

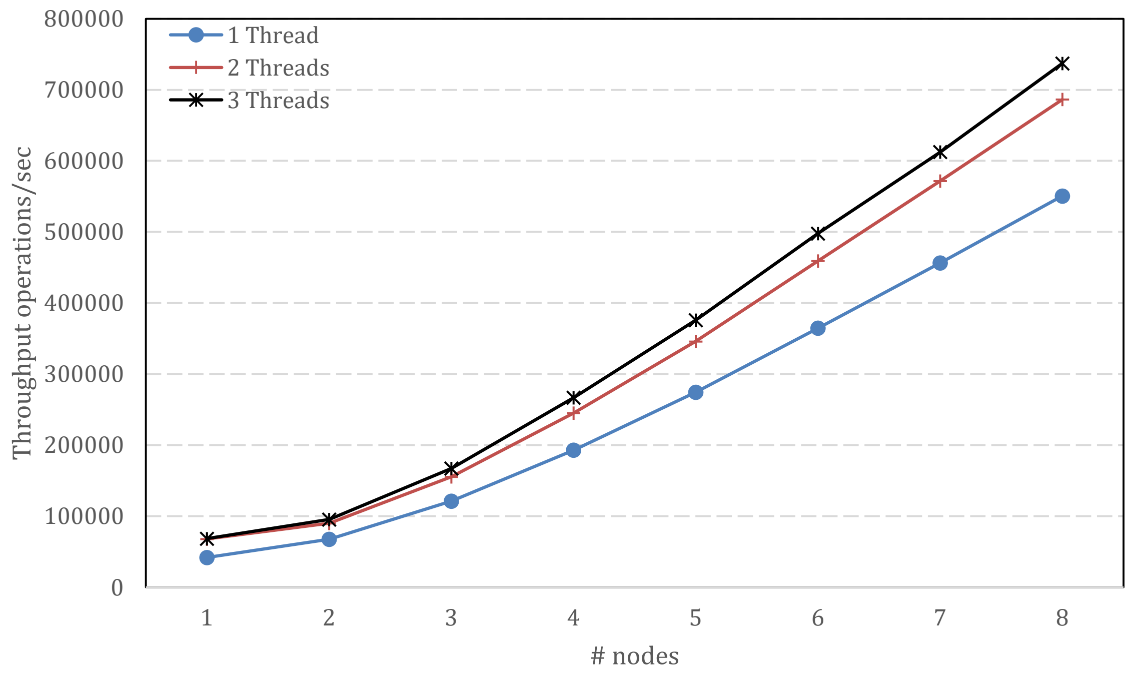

Figure 15. Throughput of dhtGet operations with different numbers of nodes and threads per node.

More specifically, the maximum achieved throughput is more than 700k operations per second, and it is achieved by the use of 3 threads. By increasing the number of nodes that issue *dhtGet* operations, the throughput increases almost linearly for more than two nodes. This shows the system's capability to increase its performance while increasing the degree of parallelism.

Figure 16 shows the performance of the *dhtPut* operations on our DHT implementation. In this experiment, the DHT was initialized to contain zero keys and the throughput was measured for fulfilling the DHT implementation with 10^5 keys. The benchmark was executed 10 times and averages were taken. Similarly to Figure 15, throughput for different number of nodes and different number of threads per node is presented. In the case of *dhtPut* operations, the throughput is an order of magnitude slower than the case of *dhtGet* operations. This is attributed to the fact that the algorithm that implements the *dhtPut* functionality is more complex since it uses a few complex synchronization primitives, i.e. locks, etc. However, the experiment clearly shows the system's capability to increase its performance while increasing the degree of parallelism.

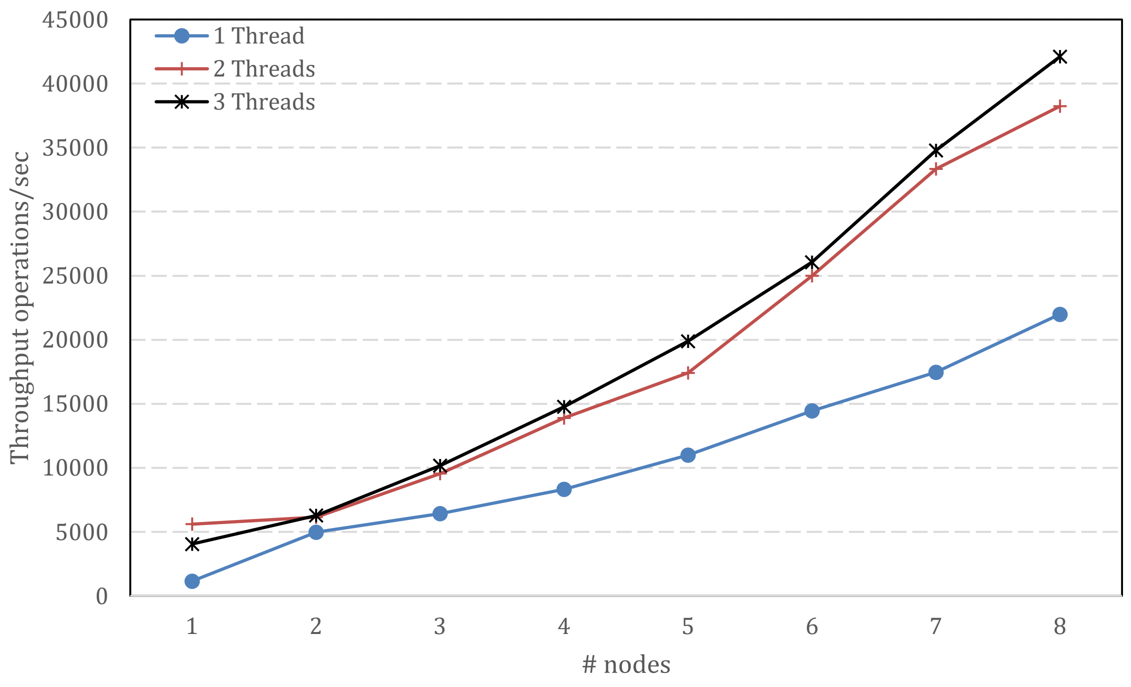

Figure 16. Throughput of dhtPut operations with different numbers of nodes and threads per node.

Figure 17 and Figure 18 show the performance of our DHT implementation when threads execute workloads that perform both *dhtPut* and *dhtGet* operations. In the experiment of Figure 17, the DHT was initialized to contain zero keys and the throughput was measured for performing 10^5 operations, where 80% of them are *dhtPut* and 20% are *dhtGet*. The benchmark was executed 10 times and averages were taken. It is shown that the throughput increases with the number of nodes and/or threads per node in an almost linear manner. More specifically, the maximum achieved throughput is about 200k operations per second, and it is achieved by using 3 threads. By increasing the number of nodes that issue *dhtGet* operations, the throughput increases almost linearly for more than two nodes.

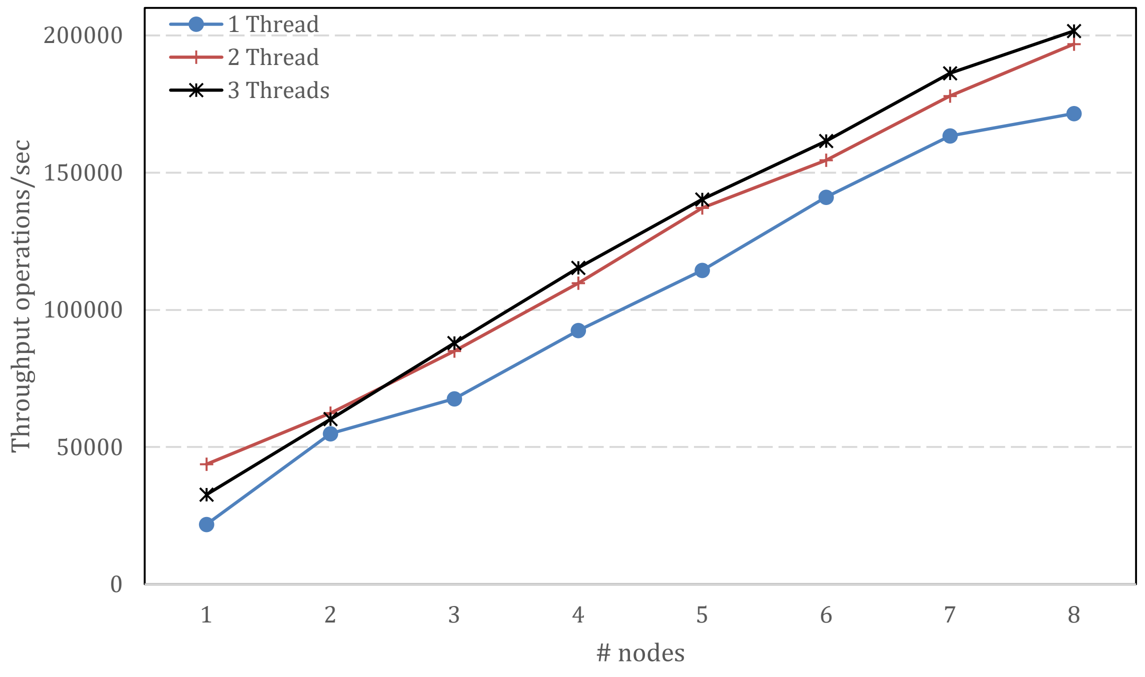

Figure 17. Throughput of dhtPut and dhtGet operations (80% dhtPut and 20% dhtGet) with different numbers of nodes and threads per node.

Similarly to Figure 17, Figure 18 shows that the throughput for performing 10^5 operations (50% *dhtPut* and 50% *dhtGet*). In this case throughput is lower than the case of Figure 17, and it is attributed to the fact that the algorithm that implements the *dhtPut* functionality is slower due to the synchronization overheads. However, the experiment clearly shows the system's capability to increase its performance while increasing the degree of parallelism an almost linear fashion.

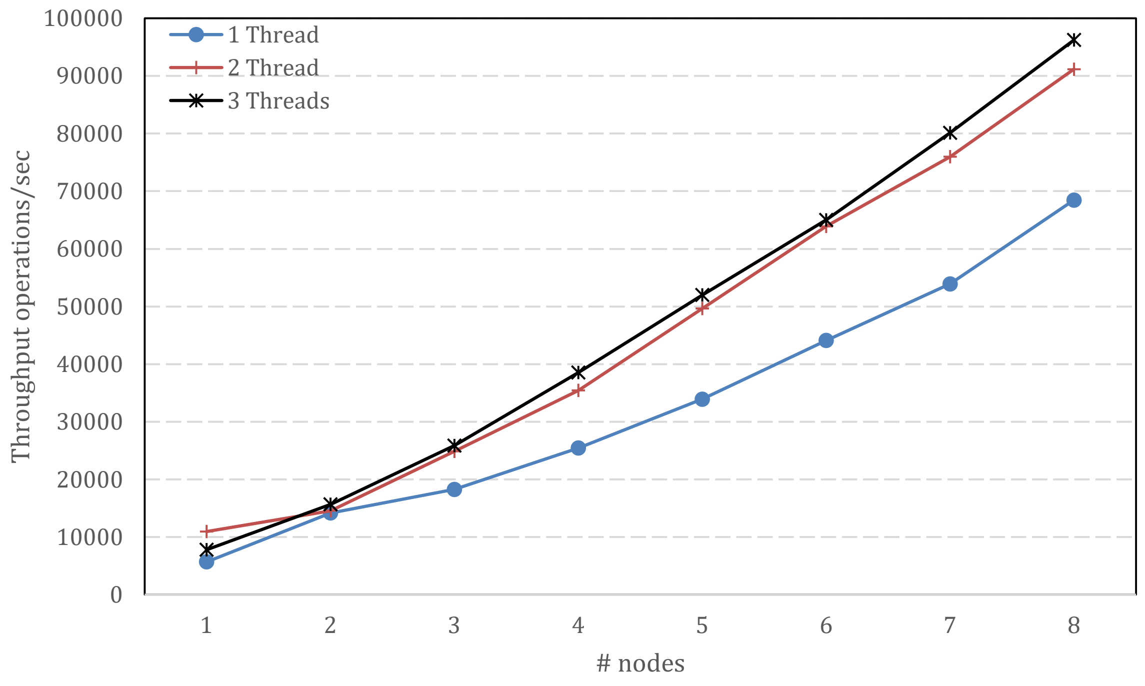

Figure 18. Throughput of dhtPut and dhtGet operations (50% dhtPut and 50% dhtGet) with different numbers of nodes and threads per node.

2.5.4 The kram module

We now describe, the kram-module, which is the most basic tool for creating and using a swap device. Whenever the module is loaded into the kernel (i.e. by using the functionality of *insmod*), the user determines how many swap devices are needed and the size of each device. The user should also determine the transfer mode, which is either a simple *memcpy* or a transfer performed by the dma engine. An example command for loading the module to the kernel is shown below.

```
> sudo insmod unimem_kram.ko ninstances=2 size=32,64 transfermode=1
```

The above operation loads the module, and asks for two swap instances of size 32MB and 64MB. Since the *transfermode* is set to 1, the dma engine will perform the transfer requests. During its initialization phase, kram-module starts a high resolution timer, which is triggered in every second, approximately. Whenever this timer is triggered, for any request for swap space creation, the driver sends a message using mailbox to every remote node, checking for remote memory availability. By sending that first message to some remote node, an intercommunication starts between the driver and the daemon of the particular node. In case that the driver has ensured that some of its neighbours has enough free memory and it is willing to spare, the driver will create the appropriate block device for swapping.

2.5.5 The kram daemon

The purpose of the kram daemon is to monitor the memory availability of the local node and to share efficiently this memory to remote nodes. The daemon uses a static array of structures, where every element represents a chunk of memory. Whenever the kram driver of a remote node requests some memory, the daemon searches for continuous chunks of memory that meet the size requested. If the search was successful, the kram-module allocates the requested memory. In the case where a driver uses some memory of the daemon's node for swapping, the daemon bookkeeps the information in the array.

The daemon keeps information of events in the log file:

```
/var/log/kram-daemon.log
```

2.5.6 Mailbox communication

Since the system has multiple boards, where everyone should be able to borrow and share memory with each other, a communication protocol between them is mandatory. In this section, we describe this communication protocol.

2.5.7 Message format

The mailbox driver allows one node to send a 64-bit message to any other remote node. The first 4 most-significant bits are used as a message opcode. The 60 remaining bits are used as a message payload. We now describe the payload format of each Mailbox message (Figure 19).

- Message identity (bits 0-3): These bits define the purpose of the message that was received. The following values are valid:
  - Driver message - 0x1 (in hex): The kram module requests memory to the daemon.
  - Driver message - 0x2 (in hex): The kram module confirms that it will allocate requested memory from the daemon.
  - Driver message - 0x4 (in hex): The kram module declines the memory allocation that was earlier requested by it.

  - Driver message - 0x8 (in hex): The kram module informs the daemon that it will free the allocated memory.
  - Daemon message - 0x1 (in hex): The kram daemon informs the driver that it has free memory for allocation.

- Node id (bits 4-7): The id of the node that sends the message.
- Swap instance id (bits 8-11): The id of the swap instance that the driver wants to allocate.
- Memory chunk id (bits 12-19): The index of the first chunk of memory involved.
- Number of chunks (bits 20-27): The number of chunks of memory involved.

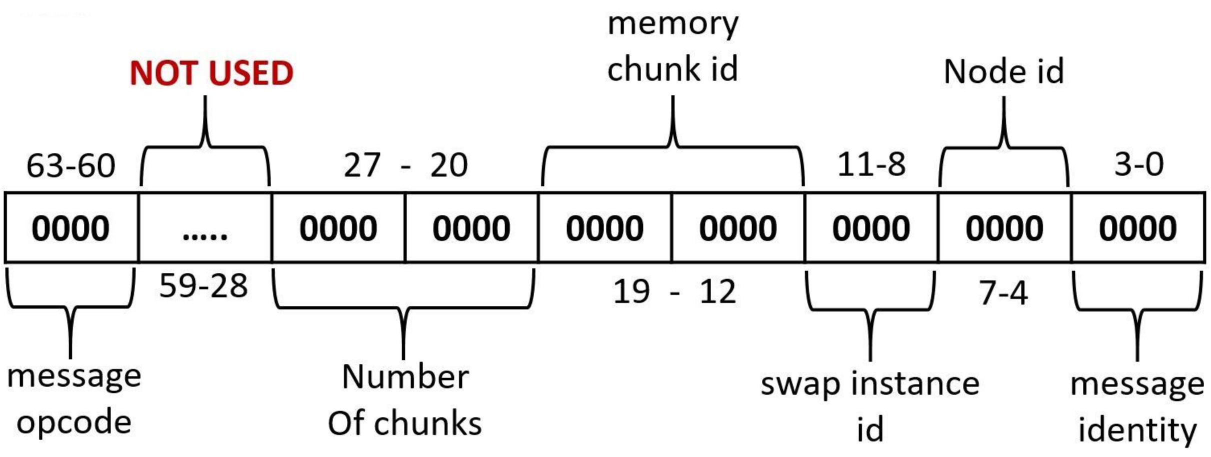

Figure 19. Message format of the KRAM communication protocol.

2.5.8 Communication protocol

Whenever, a user requests some remote memory to be used as swap space, kram-module sends a first message to the daemon, requesting remote memory by following the steps below:

1. The driver will request memory to the daemon by setting the least significant bit of the message to 1. Along with the message id, it passes the node id, the swap id, and the size of the requested portion of memory in MBs.
2. The daemon receives the message and searches for memory availability. In case that there is not enough memory available, it simply ignores the message and it does not reply. In case that a continuous chunk of memory is available, it sets the least significant bit of the message to 1, indicating that there is enough memory. It also adds to the message, its node id, the id of the swap instance received, and the index to the first available chunk of memory.
3. In case that the driver receives a message from the daemon, it checks if the swap instance id is already serviced:
   a. If the instance is not serviced, the driver will send a message to confirm the allocation. The confirmation is indicated by setting the first lowest significant bit to 1.
   b. If the instance is complete, the driver will send a message to decline the allocation by setting the second lowest significant bit to 1.
4. Afterwards, the driver adds its node id, and re-adds the chunk index, and size in MBs.
5. As a last step, when the driver no longer needs the swap device, it sends a message informing the daemon to free the allocated memory. Freeing the allocated memory is in-

dicated by setting the third lowest significant bit to 1. The kram-module driver also adds its node id to the message.

Figure 20 shows a simple schematic of the communication process.

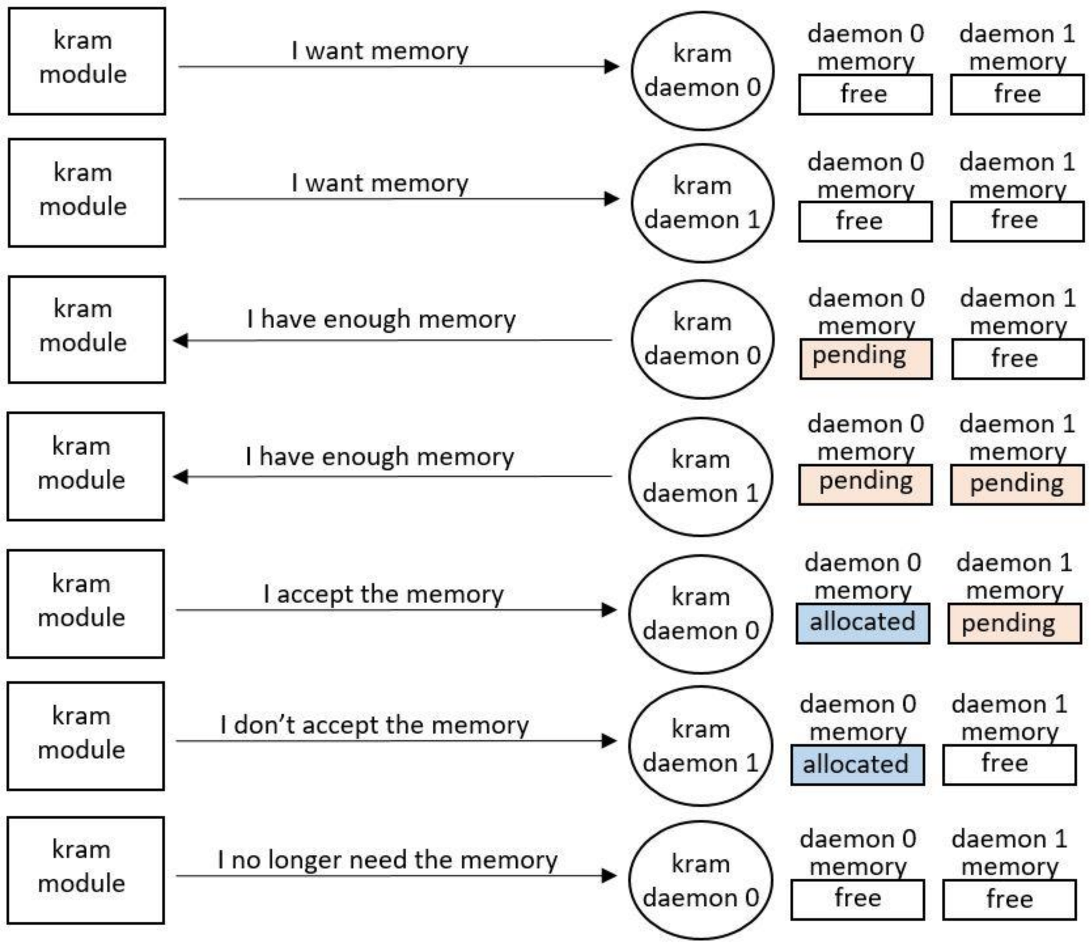

Figure 20. The KRAM communication protocol.

2.6 Swap device management

The kram-module driver handles entirely the process for creating and removing the swap devices. Some operations are executed during the module initiation, in order to improve performance while the driver runs. The other operations are executed at runtime, since they require information communicated by the daemon. When the module is removed, the swap devices are no longer required, therefore they are deleted.

2.6.1 Operations at module initiation

During their loading phase, the modules handle all initializations of daemons and allocations of descriptors that do not require information from a remote daemon. That way the modules can achieve higher performance at runtime, since it will perform the minimum required work.

A high resolution timer is used to give kram-module the ability to continuously check for incomplete instances. The timer is set to launch approximately every a second.

Using a workqueue is one of the most important aspects of kram-driver, since there is need for performing memory allocations at runtime, thus creating block devices. Since workqueues do not run in interrupt context, they are appropriate for this task. The module initiates a workqueue and it adds works to the queue that are equal to the amount of swap instances that are going to be created.

At runtime, the kram-module driver needs to add the physical addresses obtained by the daemon, to page descriptors, which will be later passed to cdma when swapping is required. For performance reasons, when the driver is running, it is more efficient to allocate the page descriptors during the initiation.

2.6.2 Operations at runtime

While the module is operating, it will continuously communicate with the daemons from the remote nodes, until all of its instances are complete. When the module receives a positive response from a memory request to a daemon, it will initiate a series of actions to create the required swap instance.

One second after of the timer initiation, the timer function will run and check for incomplete instances. When it finds an instance that needs to be created, it will send messages to daemons requesting memory.

When a daemon sends a message to the module, the callback function which is registered to the message opcode of the message received, will queue the work dependent to the instance specified along with the physical address obtained by the daemon message. After that, the work function will run, in a thread context, when the kernel decides. In the work function, the physical address range will be assigned to the page descriptors. Furtherly, the daemon will create a block request queue which is connected to a function that will handle the data transfers. There are two functions, one that uses a simple memory copy and one that utilizes the cdma. The function used depends on the *transfermode* parameter, specified at the module insertion. The last step is to create the disk device. When creating a disk device, the module assigns to the responsible structure, the correct file operations, the major and minor numbers, the request queue created above, a seemly name, and the capacity specified by the page descriptors.

2.6.3 Operations at exit

Removing the driver suggests that the swap devices cease to be mandatory. Initially, utilities like workqueues and the timer are deactivated. After that, for each instance, the disk device is deleted, the request queue is cleaned up, and the block device tied to the disk devices is unregistered.

2.7 Other use-cases for memory on remote nodes

In continuation of the work described in the D3.6 deliverable (i.e. from the 1st year of effort), we have been maintaining the various systems software modules and services to reflect changes in the underlying custom hardware blocks. This effort will continue with porting all this functionality to the final testbed of the ExaNoDe project. We have also started work on a unified framework for managing and utilizing remote memory, based on the infrastructure that we have already developed for the swap-device use-case described earlier in this section. This unified framework will combine support for swap space with NUMA-aware dynamic memory allocation policies.

Furthermore, we have worked on the following functionality areas:

- RDMA and mailbox services: We have worked towards a 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 have added support for kernel-space interception of socket-related functionality, so that we can transparently support not only user-space applications and services but also kernel-space network-based services (such as network file-systems). We currently support kernel-space socket initiation/teardown, blocking-mode reads/writes to sockets, and polling for socket-related events (e.g. availability of data to receive from a socket connection and readiness to accept new data to transmit), as per the specifications of the *epoll* system call.

3 KRAM: Remote SWAP Space

Memory-intensive applications suffer large performance loss when their working sets do not fully fit in system's memory (i.e. DRAM) due to paging-out. Yet, they cannot leverage otherwise unused remote memory, which leads to large imbalance in memory utilizations across a cluster. Existing proposals for memory disaggregation call for new architectures, new hardware designs, and/or new programming models, making them not easily applicable.

In this section, we present the KRAM remote swap space. The *KRAM* device is a DMA-enabled remote memory ram-disk. More specifically, it uses the (unused) memory of some remote node in order to create a ram disk. Each node consists of a user-space daemon, which is able to communicate with a kernel-space module of another node to accomplish the task. For memory transfer to/from remote memory, the *Xilinx Central DMA engine* and/or *memcpy* are used.

3.1 KRAM architecture

The main software components for KRAM in each of the system's node are the following:

- A kernel-space module called kram-module that handles the creation of swap devices for the operating system, and the allocation of remote memory from neighbouring nodes.
- A user-space daemon called kram-daemon, which is responsible for monitoring kram allocation request from remote nodes. This user-space daemon stores information about the allocated memory of its system from other nodes.

The kram-module and kram-daemon software blocks use the communication primitives provided by the Mailbox hardware-block. More specifically, the Mailbox uses are the following:

- The kram-module:
    - Discovers free memory through every neighbour.
    - Confirms or Declines accepted daemon responses, after a request.
- The daemon:
    - Is aware of whom handles its memory.
    - Accepts or ignores memory requests.

3.2 Swap instance creation

When kram-module is loaded, it will try to create swap instances according to the parameters given to it. After communicating with the daemons of its remote neighbour nodes, the module initiates operations that create swap instances in the form of block devices. Those devices point to the RAM of the remote nodes specified, so that when the local node runs out of RAM, it will start swapping memory, using these block devices. For this operation, the kram module:

- During the initialization of the module:
    - Initializes a high resolution timer.

  - Creates a workqueue with additional works for every instance that needs to be created.
  - Pre-allocates the page descriptors that will hold the physical addresses of the swap instances.
- At runtime for every swap instance:
  - Assigns the physical addresses acquired from the daemons, to the page descriptors.
  - Creates a request queue to handle the transfers.
  - Assigns values to Disk structure and creates the block device.

The KRAM architecture is briefly presented in Figure 21. This figure also presents an example of memory request issued by node 0 to node 1.

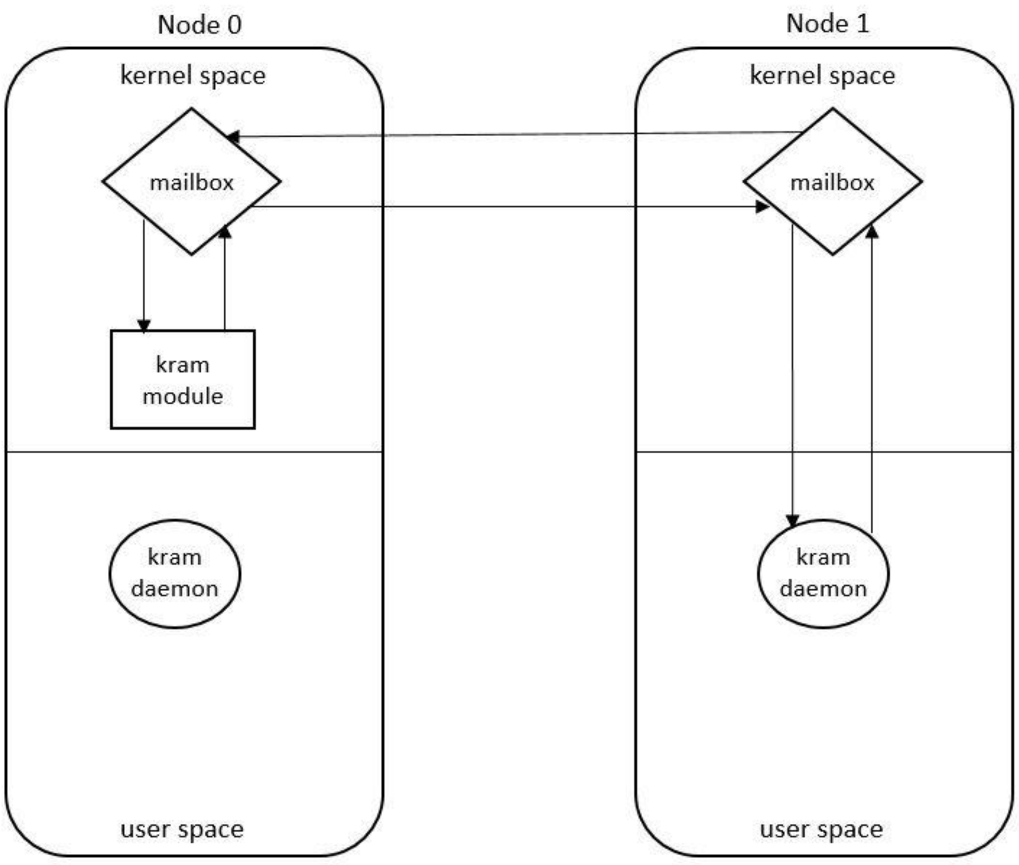

Figure 21. KRAM architecture (node 0 requests memory from node 1).

3.3 KRAM dependencies

The kram and the kram-daemon have certain system dependencies in order to be initialized, and function properly. Those dependencies are kernel modules that are loaded by initializing the Unimem environment. More specifically:

- The dependencies of kram-module are:
  - The cdma module for using the cdma engine to transfer data from/to the swap device.

    - The driver for the Mailbox hardware block, for communicating with the daemons from the other nodes.
    - The eusrvmem module, which is necessary for accessing the addresses of remote nodes.
- The dependencies of the kram-daemon are:
    - The mailbox driver for communicating with the kram modules of the remote nodes.
    - The eusrvmem, as the mailbox is dependent on it.

4 NUMA support in the operating system

In this section we report on our effort towards providing NUMA-awareness in the Linux kernel for the Unimem memory architecture. This enables applications that use the standard *libNUMA* API to use remote memory regions in our prototype. Our OS adaptations aim 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.

In a traditional SMP (Symmetric Multiprocessing) system, the machine has a single memory controller. As a result, when multiple processes try to access memory in the same time, the bus is overloaded. The quality of the services that the system provides drops and the impact is noticeable in the run-time of the applications, since memory input and output already consume a significant amount of cycles. In order to address this issue, modern multicore computers deployed as servers use multiple memory controllers with each one of them controlling a portion of the memory of a board. This technology is known as Non-Uniform Memory Access (abr. NUMA). Non-uniform memory access means that it will take longer to access certain memory regions than others. Access to the local memory of a processor is much faster than access to remote memory, but remote memory is still useful for memory bound applications. An important advantage of NUMA systems is scalability. As we already mentioned, for a computer server, a big number of applications run at any time and as a result, the memory bus is under heavy contention. For an application that its main performance constrain is memory bandwidth, it may slow down 2-3 times when most of the memory accesses are remote instead of local.

A working NUMA implementation was first made available in the Linux kernel mainline in 2002. Version 2.5 of the Linux kernel already contained basic NUMA support for the x86 architecture (specifically, for the AMD Opteron processor family). Together with the NUMA support in Linux kernel, a library called *libnuma* and a utility called *numactl* were developed by the open-source community. NUMA support for the ARM processors only came much later (starting in 2015) with the introduction of the Cavium ThunderX processor (48 cores, running at 2.5 GHz, consisting of up two NUMA nodes per system).

In this section, we describe our effort towards NUMA support for the ExaNoDe testbed based on the ARM Juno development boards (8). In this prototype, described in detail in deliverable D5.1, system nodes are interconnected via high-speed serial links, which in turn are available to each node via a FPGA connected to the node's PCI-Express system interconnect.

4.1 Implementation concepts for NUMA support in the Linux kernel

The Linux kernel manages the policy for processes or specific memory mappings, and offers the following system calls to user-space applications:

- *mbind*: select binding for the specified memory pages to nodes.
- *set_mempolicy*, *get_mempolicy*: set/get the default binding policy.
- *migrate_pages*: migrate all pages of a process on a given set of nodes to a different set of nodes.
- *move_pages*: Move selected pages to a given node or request node information about pages.

Besides the system calls described above, Linux provides the *libNUMA* library (9) as a way to issue these system calls in a more programmer-friendly and abstract manner.

NUMA policy is concerned with putting memory allocations on specific nodes to let programs access them as quickly as possible. The primary way to do this is to allocate memory for a thread on its local node and keep the thread running there (node affinity). This gives the best latency for memory and minimizes traffic over the global interconnect. However, the scheduler in the operating system cannot always optimize purely for node affinity. The problem is that not using a CPU in the system would be even worse than a process using remote memory and seeing higher memory latency. In cases where the memory performance is more important than the utilization of system's processing cores, the application or the system administrator can override the default decisions of the operating system. This allows system to be better optimized for specific workloads. Linux traditionally had system calls to bind threads to specific processing cores (using the *sched_set_affinity* system call and *schedutils*). The *libNUMA* library extends this, giving programs the ability to specify on which node memory should be allocated. To make it easier for user space programs to optimize for NUMA configurations, the *libNUMA* API also exposes topology information and allows user specification of processor and memory resources to use.

The Linux kernel supports the following modes of allocating memory[3]:

- Interleaved: allocate memory (at page granularity) in an interleaved manner over a set of nodes, i.e. successive allocation requests will reserve memory pages on different nodes. If memory is exhausted at a node, then an automatic fall-back is applied to provide memory from another node.
- Bind: Memory is allocated on the node in the set with sufficient free memory that is closest to the node where the allocation takes place.
- Preferred: The allocation is attempted from the single node specified in the allocation policy. If that allocation fails, the kernel will search from other nodes, in order of increasing distance from the preferred node. The distance information is made available by the platform firmware.

A key concept in the Linux kernel for describing the memory architecture of a system is the notion of *distance* between the node where a memory region is physically located and the node where the actual use of the memory contents is needed. A system is characterized by its memory *topology* (10), which essentially describes the distance between each processing core and each of the available memory regions. A memory node is typically the set of memory regions accessible via a specific memory controller. A NUMA system includes more than one memory controller, which results in variations of distance between its processing cores and each of the available memory nodes. A crucial part of our work has been to introduce these notions of distance and topology for the 64-bit ARM architecture.

4.2 Implementing *libNUMA* infrastructure on ARM-based systems

Whenever an x86-based system is booting-up, the system firmware retrieves the NUMA configuration of the system by using ACPI (Advanced Configuration & Power Interface) tables. ACPI support for the ARM64 architecture is currently under development, and a significant

[3] Source: Notes on Linux Memory Policy, available the Documentation/ directory of the Linux source-tree: https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt

part of the ACPI specification is not supported yet. The essential information for describing a NUMA topology is contained on the SLIT and SRAT tables:

- The ACPI Static Resource Affinity Table (SRAT) stores the topology information for all the processors and memory.
- The ACPI System Locality Information Table (SLIT) describes the relative access times between processors, memory subsystems, and I/O subsystems.

Therefore, we had to find a way to capture this essential information in our ARM-based test-bed. ARM-based systems typically rely on the device-tree data structure for obtaining a machine-readable description of platform resources and properties at boot time. The Linux kernel obtains this data structure by reading a binary file. For the purpose of identification, each NUMA node is associated with a unique token (node ID). In order to describe the connectivity and relative cost of communication between NUMA nodes, we introduced to the device tree the distance matrix. This matrix is represented as a list of node pairs and their relative distance.

In the ExaNoDe testbed based on ARM Juno systems, the implementation of NUMA support for PCI memory address space is contained within a Linux kernel module that manages the mapping of the physical PCI memory address space to the virtual memory address space of the process. This is achieved by using existing functionality of the Linux memory management subsystem, i.e., *remap_pfn_range* function. Furthermore, this module is responsible for managing the allocation policy for a specified PCI memory address space, as needed by the API of *libNUMA*. Main features include the migration and movement of memory pages to/from remote and local memory regions.

Our kernel module can also parse the appropriately formatted entries under the /proc pseudo-filesystem for obtaining system configuration information. Specifically, via the /proc filesystem we can describe remote memory regions that will become available for use via the *libNUMA* API (at the current system node), and obtain a summary of the current memory node topology (in particular, the available memory address regions).

4.3 Enablement of different service levels for libNUMA

We have implemented alternative memory allocation policies that take into account the user and group identifiers of the process that is using the *libNUMA* API. By allowing the *libNUMA* infrastructure in the kernel to identify processes that issue *libNUMA* API calls, we have created a simple control for setting the ratio of remote vs. local memory pages allocated to each process. This ratio significantly affects the user-observed performance of the calling processes, i.e. enabling the kernel to distinguish between different service levels. By exposing a control for this crucial parameter, we are looking forward to future efforts towards the development of dynamic memory allocation policies that would also consider other criteria for distinguishing among applications.

So far, we have implemented two policies:

- Allocation based on process group ID: Processes are distinguished by their process group ID, which is interpreted as a credit score. Processes with a higher credit score are given higher priority in reserving memory at their closest memory node (in terms of the NUMA distance metric). The default policy for handling allocation requests is that memory is allocated from the node that currently has the highest amount of available memory. In case of memory pressure, an allocation may fail to give priority to conflicting allocation requests by processes with a higher credit score.

- Allocation with potential for migration: Any process can obtain memory pages from any of the available memory nodes. However, a later allocation request by a process with a higher credit score may "evict" memory pages, i.e. trigger an automatic migration of memory pages to another, more distant memory nodes.

Table 2 lists the *libNUMA* API calls that we have implemented and validated on the ExaNoDe testbed. API calls are grouped based on their functionality.

Functionality area	***libNUMA* API calls implemented on testbed built with ARM Juno development systems**
Memory page allocation	*numa_alloc_onnode*, *numa_alloc_interleaved*
Movement and migration of memory pages	*numa_move_pages*, *numa_migrate_pages*
Memory policy	*numa_bitmask_alloc*, *numa_bitmask_setbit*

Table 2. libNUMA API calls on ExaNoDe Testbed.

5 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 presented our enhancements on the global shared address space (GSAS) and its communication mechanisms on the Trenz prototype. Moreover, we presented in detail the protection model applied on the GSAS environment. We described the implementation of a key-value concurrent data structure on the GSAS environment. This key-value data structure provides basic functionality for storing and retrieving pairs of keys and values and it is based on the synchronization and remote access mechanisms provided by the GSAS environment. We also presented the KRAM remote swap space, which is a DMA enabled remote memory ram disk. The KRAM environment leverages the (unused) memory of remote nodes giving the ability to augment the memory space of the local applications.

Starting from M6, FORTH is offering a remote access facility to the Juno multi-board prototype, via a web-based reservation system (implemented using the Apache Virtual Computing Lab platform). The same web-based reservation system is also used for offering remote access to the Trenz multi-board prototype.

Partners of ExaNoDe WP3 (namely: BSC, FHG, VOSYS and University of Manchester) 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 on providing 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 enhance the performance of the GSAS environment. More specifically, we plan to improve the performance of atomic operations performed in a local node. Furthermore, we plan to extend the API of the GSAS environment in order to provide more advanced mechanisms for remote memory allocation.
- DHT over GSAS: We plan to better optimize the concurrent data structure used in the DHT implementation in order to enhance its performance. We also plan to provide new, more advanced mechanisms for storing and accessing data on the DHT implementation, and conduct a comprehensive performance evaluation.
- *libNUMA* support: We plan to port *libNUMA* support to the upcoming ExaNoDe testbed, and conduct a comprehensive performance evaluation. We expect this functionality to be incorporated within a unified framework for managing and utilizing remote memory.

6 References

1. *The EUROSERVER project. European Exascale System Interconnect and Storage. GA-610456. http://www.euroserver-project.eu.*

2. *The ExaNest project. European Exascale System Interconnect and Storage. GA-671553. 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. Trenz Development Platform Technical Reference Manual - TE0808-03. *http://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/TE0808/REV03/Documents/TRM-TE0808-03.pdf.* May 2017.

6. Kallimanis, Nikolaos D., et al. Design of the ExaNoDe Firmware. *Deliverable D3.6. ExaNoDe: European Exascale Processor Memory Node Design.*

7. *Revisiting the combining synchronization technique.* Fatourou, Panagiota and Kallimanis, Nikolaos D. 2012. In Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2012). pp. 257-266.

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

9. *A NUMA API for Linux. Technical Linux Whitepaper, Novell, available at http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.* Kleen, Andy. April 2005.

10. Lameter, Christoph. Numa (non-uniform memory access): An overview. *ACM Queue* . 2013, Vol. 11, 7.