A Virtual Platform for High Speed Message-Passing-Hardware Research

Diplomarbeit im Studiengang Informatik

Fakultät für Informatik
Technische Universität Dortmund

vorgelegt von Gregor Rebel am 5. März 2009

Gutachter: Prof. Dr. Olaf Spinczyk
Dr. Klaus Danne

Diese Arbeit entstand in Kooperation mit der Firma Intel GmbH

EIDESSTATTLICHE VERSICHERUNG

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig
und ohne fremde Hilfe angefertigt habe. Die verwendete Literatur ist vollständig angegeben.

Unterschrift _____

Datum _____

Acknowledgements

At this place I would like to send many thanks to the folks at the Germany Research Center for giving me the opportunity to do research in bleeding edge technology. During my 3,5 years at Intel, I have learned much more about what drives the whole semiconductor industry from lots of talks with colleagues than during my studies before.

My direct thanks go to Klaus Danne, Thomas Lehnig and Michael Kauschke who never stopped to criticize me and to confront me with new ideas.

Abstract

The proposal of this work is to emulate a non coherent, shared memory based hardware message passing interface of a theoretical, future chip multiprocessor (CMP) by making use of virtualization techniques on todays SMP machines. The individual processing nodes of the system are represented by virtual domains which each run on one core of a multi core CPU with a coherent shared memory system.

In addition we describe the implementation of a virtual network interface card using this emulated hardware message passing interface. Together with some control scripts, this environment implements the virtual research platform. It allows to compare various configurations of message passing hardware. The exemplary implementation of a special routing protocol extends the design space of this platform.

The research platform allows to port the virtual network interface card and other software implemented and tested on this platform onto a real many core chip. It can provide simulation results for different configurations of message passing hardware much faster than traditional approaches.

Table of Content

Chapter 1
Introduction

The aim of this thesis is to enable architecture research for many core CPUs that use message passing rather than a coherent shared memory communication scheme.

1.1 Motivation for Message Passing Hardware Research

The semiconductor business is basically driven by the so called Moore's Law. In 1965 Gordon E. Moore, a co founder of Intel Corp. forecasted that the number of transistors that can be placed inexpensively on a chip will double about every two years [Moo65]. This exponential growth has led to today's server CPUs being assembled of two billion transistors (e.g. Itanium 2 quad core CPU in Q1 2008).

For future chips with even more transistors, engineers face the problem of how to turn the enormous transistor count into real world performance. One big problem they face is known as Pollack's rule [Pol99]. It states that for the same technology, single threaded integer performance scales only by the square root of transistor count.

Shekar Borkar uses this rule in his article [Bor07] as an argument why the chip industry has to move from single and multi core CPUs towards many core CPUs. When a chip designer has one billion transistors to spend on a CPU, then it can bring more performance to invest the transistors into 100 CPUs with 10 million transistors each than into 10 CPUs of 100 million transistors each.

Today one main argument for multi and many core is to increase the performance per watt ratio. With twice the number of transistors on a chip every two years, the designer has to use them wisely. The company that is able to make best use of them will lead the industry. According to Shekar Borkar, multi core CPUs provide more performance per watt than single core CPUs using the same number of transistors for more functional units and bigger caches in a big single core. Many cores can be seen as a step further where hundreds of small cores work together on one chip.

With increasing number of cores, inter core communication needs grow polynomially. Buses and point-to-point connections do not scale well over the number of nodes. Todays multiprocessors implement symmetric multiprocessing (*SMP*) designs with coherent shared memories and bus snooping caches. According to [Sten90] p.18, the limiting factor of such designs for higher number of cores is the traffic that is produced by the cache coherency protocol. For connecting more than 100 cores, Stenström claims the that use of multi stage networks is required. Already in 2007 an 80 core prototype chip was built which implements a network on chip [VHR+07]. This shows that *NoCs* may find their way into consumer products in the near future.

Such networks use non coherent distributed memory. Coherency is implemented in software using message passing. The main difference between coherent memories and message passing is that in the first approach, every change in a shared memory is distributed to all other nodes. With message passing, the software developer is in charge of maintaining required coherency. This means he or she has to determine which updates are necessary. This reduces the overall traffic in many core CPUs. A reduced traffic allows for better scalability. Therefore research on message passing is highly attractive in order to make use of the potential performance of future many core CPUs.

1.2 Convenient Approach to Research Message Passing

The two traditional ways to do research on computer hardware without building it are simulation and emulation technologies. Normally hardware emulation provides higher emulation speed than simulation at reduced precision and less flexibility. For many core architectures with more than 100 cores, simulation is too slow and hardware emulation is too expensive to run a real operating system interactively. E.g in 2008 [Iko08] announced the commercial IKOS VLE emulator which is able to emulate designs at $ 0.37 per ASIC gate, resulting in approximately $ 540 million for a two billion transistor chip. At the time of writing, IKOS VLE is a state of the art emulator.
Another problem with hardware emulation is that it always requires a synthesizable RTL model of the device. Thus incuring big efforts before the first emulation can take place.

A new alternative to the traditional ways of research can be the use of existing SMP hardware in combination with virtualized operating systems. Operating systems for many core CPUs with non uniform memory architectures (*NUMA*) can be divided into two categories:

- **Single NUMA aware operating system**

  A single OS knows about the memory architecture of the current many core CPU. The OS schedules processes on the different cores and migrates context data of each process between the cores. Building such an OS is a complex task because plain operating systems like e.g. Linux highly depend on coherent memory architectures. Large parts of these operating systems would have to be rewritten.

- **Multiple Instances of the same operating system**

  In this case a vanilla operating system is replicated for every core. Each replicate OS (further referred to as a node) runs completely in memory that is local to its core. Communication between the nodes is enabled by a special message passing interface which acts like a normal network interface.

  This approach is easy to implement because an existing OS can be used. The implementation of a virtual network interface allows to run standard network applications like MPI benchmark software for performance evaluation.

To run multiple instances of the same operating system on one many core CPU is similar to what virtualization does. Virtualization allows to run several instances of the same operating system on a *SMP* machine. Each instance is called a domain and may run on its own CPU core. Todays SMP machines provide up to 16 CPU cores on one SMP machine. To run 16 instances of the same operating system on such a machine would allow to emulate the behavior of a 16 core many core CPU.

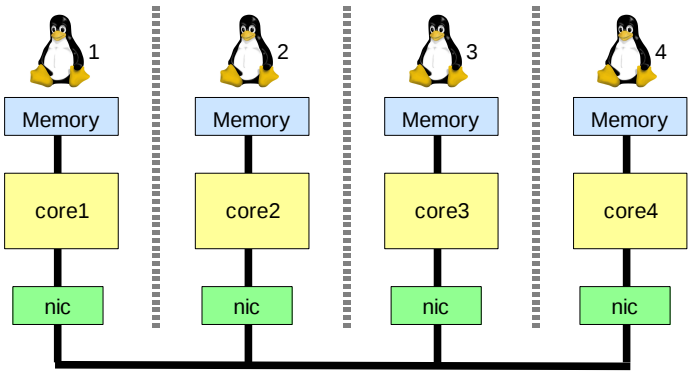

Figure 1: Many Core CPU with distributed memory

Figure 1 shows the use of a network interface to communicate among the cores of a many core CPU with distributed memory. Each core runs one single operating system instance. Such class of architecture an be emulated using a virtualization approach.

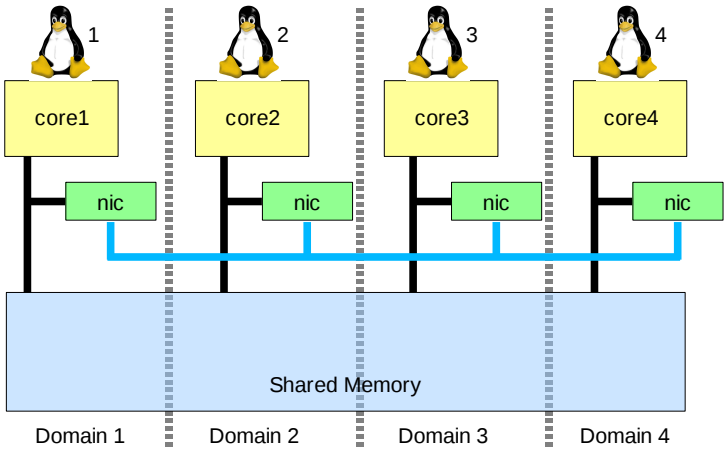

Figure 2: One virtualized OS per core

Figure 2 shows a virtualized setup corresponding to Figure 1. Each domain runs one virtualized operating system on one core. The domains communicate with each other via A virtual network interface that uses the shared memory.

The virtualization approach is followed within this thesis.

1.3 Structure of this Thesis

Chapter 2 of this thesis gives some background information in the field of todays many core research and hardware message passing. It also provides a brief summary of virtualization technology and a special routing protocol.

Chapter 3 describes how our approach is used to simulate message passing. A high level description of the software components visualizes the main concepts of the proposed research platform.

Chapter 4 goes into details of the software implementation. This Chapter covers the whole implementation range from installing and configuring the Linux OS to internals of the three kernel modules which build the core of virtualized message passing.

Chapter 5 shows a performance evaluation on a set of benchmarks. It shows the parameters of message passing that can be configured on this research platform.

Chapter 6 summarizes the main results of this thesis and gives an outlook of some thinkable enhancements.

Chapter 2
Background and related Work

This Chapter describes related research as well as the technologies used in this thesis.

2.1 Differentiation Between Multi- and Many Core

The main difference between multi and many core CPUs is the way how to turn a certain number of transistors into performance. Many core CPUs can provide a much higher theoretical performance than multi core CPUs when built from the same transistor budget.

In a multi core design, each core is designed for maximum single thread performance. The performance of cores does scale sub linear over their transistor count.

Shekar Borkar states in [Bor07] that a many core design implements smaller cores with a better performance per watt ratio than in multi core designs. Reducing the number of transistors per core allows to build more cores within the same transistor budget. Many core CPUs are often seen as a simple technological extension of todays multi core CPUs. According to Shekar, a many core CPU is not 25 instances of a 4 core multi core CPU. Instead many core means a different usage of transistors which are available from todays chip fabrication processes. Based on Moore's law [Moo65], the exponential growth of transistor count has lead to up to two billion transistors on a single die in 2008. In the past, the increased transistor count has been used to implement bigger single core CPUs by adding more functional units and larger caches. Calculation done by Fred Pollack [Pol99] shows that the performance of integer units scale only by square root of their transistor count. One reason is that adding more functional units to a CPU makes it harder to keep them busy because the instruction level parallelism is almost exploited by current CPUs. Hence, increased transistor count cannot be turned into performance easily.

Many cores promise to provide more performance per transistor. By shrinking the individual cores to an optimal size, much more of these small cores can be built with the same budget of transistors. The theoretical maximum performance then is simply the number of cores multiplied by the performance of each single core. In practice, this goal is hard to reach for software algorithms which imply lots of dependencies between calculations. If calculations are spread over multiple cores, then such dependencies require one core to communicate its results to other cores. As the number of cores grows, the overall communication needs increase super linear.

If fixed function units are seen as primitive cores, then 3D graphics accelerators can be seen as many core CPUs. The first commercially successful accelerators from the company 3DFx emerged in 1997 [Val01]. Since then, the processing nodes in these chips increased their complexity in each generation. Lots of approaches were made over the time to use these graphic processing units for more general purposes (the company NVIDIA calls this GPGPU) [gpg09]. In 2007 NVIDIA released a software development kit called Complete Unified Device Architecture (CUDA) to provide the first unified general purpose programming model for 3D graphics accelerators.

2.2 Concept of Message Passing

Message passing is a common concept of communication where a set of processes communicate with each other by sending messages. The processes may run on the same or different CPUs on one machine or on a cluster of machines. Each process typically differenciates between local and remote memory. While local memory can be accessed fast and directly, accesses to remote memory are

implemented by sending messages to other processes.

The latency of the message passing interface is crucial for overall system performance as messages are often used like function calls [RSV87].

When it comes to many core chips, message passing is an alternative to a globally shared coherent memory system. The problem of systems with coherent shared memory is that they scale badly over the number of cores, as already described in Chapter 1.1 .

On many core architectures, message passing can be implemented e.g. via a small shared memory block which is dedicated to communication only. This is the approach which is favored in this thesis. The underlying implementation of this shared memory is not part of this thesis. Several hardware implementations may be compatible to the target class of message passing architecture as described in Chapter 3.1 .

On the software side, the message passing interface (MPI) is defined as a standard [MPI95]. Some important goals are:

- Provide an application programming interface (API) to software developers
- Parallelize computation and communications by using zero-copy strategies and offloading communication from the CPU
- Take into consideration the requirements to run in heterogeneous environments
- Flexible to be implemented on many different hardware platforms
- Declare semantics independent of a certain computer language

Several different implementations of MPI exist like MPICH, LAM-MPI, Open MPI and MVAPICH. The benchmarks for this thesis have been run with MPICH version 2.

2.3 Networks on Chip (*NoC*)

Computer networks have been known for a long time. The basis for the Ethernet standard has been set by Xerox Palo Alto Research Center (PARC) in 1973-75. In 1980 the three companies Digital, Intel and Xerox defined a standard for 10Mbps Ethernet [Dig80]. In this standard, the NIC only implemented the lower two layers (physical and data link) of the seven layer ISO network protocol model [Com98] p.205. Over the time network interface cards (NICs) have evolved into very complex devices implementing even a complete TCP/IP stack [Che07].

2.4 Virtualization

The history of virtualization is well described in [CB06]. Virtualization began with IBM System/360 in 1972 as an extrapolation of time-sharing systems: These systems have been invented to share one expensive computer among many users. This sharing raised the need to isolate system resources between individual users. The new idea was that processes belong to users. User and supervisor modes were added to most commercially relevant processors. So called "privileged" instructions are only usable by programs running in supervisor mode. Only the operating system software runs in supervisor mode. All other software runs in user mode. This scheme protects the operating system against unauthorized access from user programs.

For this sharing approach, no virtualization was needed until different applications required to run different operations systems on the same physical machine. Only virtualization is able to run different operating systems on the same hardware.

Virtualization means to insert a layer of abstraction between an operating system and its underlying

hardware. This extra layer is often called hypervisor of virtual machine monitor (VMM). Virtualized operating systems are called guests or domains. Each guest has the illusion of having real access to the underlying hardware-platform. The hardware shown to guests can be different to real hardware. Each guest may contain a separate operating system.

The hypervisor has the following responsibilities:

- Run multiple operating system images and their applications
- Share the underlying hardware resources among all guest OSs
- Let each guest OS believe that it has direct access to real hardware

Though this extra layer introduces some performance overhead, it has several advantages:

- **More than one operating system can be run on the same hardware.**

  This feature is mainly used in server farms to consolidate several machines into one to save space in the computing center and to reduce operating costs.

- **A virtualized operating system sees only virtual hardware.**

  When a new hardware is developed, driver software has to be written for several operating systems. Developing a driver for many operating systems can be very expensive for a company. With virtualization, a driver is only required for the operating system which manages the virtualization. All virtualized operating systems only have to support a virtual hardware of common type for which drivers already exist.

- **The current state of a running operating system can be saved**

  Virtualization allows to save and restore the complete state of a running operating system and its applications. Complete prepackaged containers of life applications can be used to replace traditional software distribution techniques which are known to be error prone. When a certain hardware is able to run the required virtualization, then it is automatically able to run such a prepackaged software, regardless of its underlying hardware.

2.4.1 Virtualization on x86 hardware

Virtualization on x86 hardware is relatively new when compared to the IBM System/360. The main reason for this is that the x86 architecture was not designed with virtualization in mind [AA06].

On x86 hardware, four privilege levels exist which are named ring 0 to ring 3. Ring 0 has the most privileges and is able to execute especially those instructions that are required to manage an operating system. Most OSs today make use of only two of these rings (e.g. ring 0 and 3 for Linux). Application software runs in ring 3. Whenever an exception or an interrupt occurs, the CPU switches to ring 0 and enters the interrupt handling function. The basic idea of virtualization on x86 is to shift ring 0 of a virtualized OS to ring 1. A new software called hypervisor now runs in ring 0 and manages exceptions and interrupts which are generated by all virtualized operating systems.

Though x86 provides several privilege rings, its instructions do not reflect this concept consistently. E.g. some instructions are allowed to be issued in different rings but they behave differently. Such a behavior is problematic for virtualization. When an operating system, which is intended to run in ring 0, gets instanced by the hypervisor in ring 1, these problematic instructions do not cause exceptions but behave unexpectedly. The hypervisor then has to replace these instructions with code that emulates the original behavior during runtime. This technique is called binary translation. The dynamic modification of assembler code during runtime can have a major performance impact on

the virtualized operating system. This impact was the reason for the development of other types of virtualization techniques.

2.4.2 Main Types of Virtualization

The existing types of virtualization differ in the way how they handle the execution of privileged instructions or accesses to restricted memory addresses. The critical areas are mainly memory management and IO space access. No guest OS is allowed to access hardware directly or to manage physical memory on its own. These accesses need to be rerouted to the hypervisor.

Four main types of virtualization do exist:

- **Trap-n-Emulate**

  This is the traditional concept of virtualization with the highest impact on guest performance. Whenever a guest OS executes a privileged instruction, the processor raises an exception and jumps to the corresponding exception handler. The exception handler then has to find out what caused the exception and how to emulate its intended behavior. Because every write access to the page table directory causes an exception, the performance impact of this type of virtualization to the memory management is very high.

- **Binary Translation (BT)**

  Binary translation tries to avoid Trap-n-Emulate situations as much as possible by dynamic compilation.
  With binary translation, every piece of code is first analyzed by software before it comes to execution. All instructions which can cause an exception or which are known to be problematic are replaced by calls to functions which emulate the original behavior. The additional translation delays the execution of each new code but it can avoid big performance impacts from privileged instructions inside loops. To reduce the overhead of binary translation, current implementations hold a cache of already translated code.

- **Hardware Assisted Virtualization (HAV)**

  Special hardware assistance for virtualization on x86 has been developed by the companies AMD (AMD-V) and Intel (Intel VT-x) and is built into most current CPUs. For Intel VT-x, this technique is described in [Int07c] volume 3B: System Programming Guide Part 2 Chapter 26. It makes use of a virtual translation look aside buffer (TLB) to allow guest OSs full write access to virtual memory configuration structures. Exceptions are only raised under some rare conditions. The processor itself takes care of changed entries in the virtual memory management structures by setting certain flags. This hardware assisted approach can reduce the number of required exceptions significantly. Though HAV can reduce the number of raised exceptions, the authors of [AA06] claim that it can be slower than BT under certain conditions, This is because with BT frequently used code is translated only once and then put in a memory cache. The cached translations generate less exceptions than with HAV technology.

- **Paravirtualization**

  This type of virtualization can avoid Trap-n-Emulate situations without hardware support or binary translation.
  Instead of handling exceptions from privileged instructions, paravirtualization avoids these instructions by modifying the operating system that is to be virtualized. Paravirtualization therefore is also called a cooperative virtualization. Whenever a guest OS wants to access the memory management structures, it issues a special hypercall function. This hypercall function is provided with all informations required to identify which action should be taken. The function

itself raises an exception to enter the hypervisor and to carry out the intended operation. One advantage of this approach is that several operations can be bundled together into one hypercall, thus reducing the number of raised exceptions. Another effect of this cooperative approach is that the hypervisor can directly issue the correct handling function. No time is spend to find out which operation was intended by the virtualized guest OS.
The biggest disadvantage of paravirtualization is that it requires special, modified operating systems. Paravirtualization is supported by several virtualization products. One of these is XEN virtualization which is used as the basic technology in this thesis [XEN?].

2.5 Valiant Load Balance Routing (VLB)

In 1981, L.G. Valiant and G.J. Brebner described a universal scheme for parallel communication [VB81]. The essence of their work is to describe a distributed routing strategy. This strategy allows to efficiently balance the traffic in a network via multiple parallel connections. This approach avoids overloading of individual network links and is able to automatically circumvent broken links. As it is a distributed strategy, no extra control instances are required to balance the load between the nodes of the network.

VLB routing is used as a possible extension of the design space of this research platform. The following sections summarize those aspects of their paper that are relevant for this thesis.

2.5.1 Two Models of Parallel Computation

The authors first describe the concept of an idealistic model for parallel computation. This idealistic model mainly consists of N general purpose processors. The processors have access to a shared memory system with almost arbitrary parallel access. The only restriction for memory accesses is that the implemented algorithms guarantee to avoid simultaneous write accesses to the same memory location. The embargo of concurrent read accesses from the same location is formulated as an optional restriction. The model is called an idealistic model, because the authors could not see the possibility to implement it with foreseeable technologies. And even 27 years later, no implementations of coherent SMP machines with more than 100 cores are practical because of the inherent problems which have been described in Chapter 1.1 .

A more relaxed model is then described as a realistic parallel computer. This computer consists of N processors which are connected among each other by directed edges. These edges are further called links. At most d links enter and at most d links leave each of the processors. The authors postulate that d grows much slower than N like a logarithmic function because of physical limitations. The precise reasons for these physical limitation are not mentioned in the paper. But it is thinkable that these limitations arise from the problem of routing network cables in a cabinet, conductive paths on a printed circuit board or metal wires on a silicon die within a limited space. Each processor in this model owns a local memory.

With these two models in mind, each many core CPU can be seen as a realistic model of a parallel computer. When using networks on chip, many core designs can be able to provide a virtual 1 to N connectivity for every processing core by routing packets through the network [VHR+07]. This allows them to provide a better overall connectivity than the realistic model.

2.5.2 Valiant's Proposal of a Parallel Computer

The parallel computer that is proposed by Valiant and Brebner starts with an initial scenario. At start time, each node has some amount of packets of information. The task of the parallel computer is then to transport all packets from all nodes to their intended targets. At every time, each link may

transport up to one packet. All packets should reach their target node as soon as possible. In this model, different bandwidths between two nodes can be expressed by connecting them together with more than one direct links for each direction.

2.5.3 Introducing a Randomized Routing Algorithm

The main part of this scheme describes a „randomized routing algorithm“, which implements a highly distributed routing algorithm. In order to distribute the routing decisions among all nodes it is required that each packet carries some extra book keeping data with it like its destination address and routing data. According to Valiant and Brebner, the main advantage of their algorithm is that it distributes the problem of computing permutations. These permutations are required to be calculated for every packet on each node.

The routing algorithm itself is divided into two sequential phases:

a) Each packet is sent to a randomly chosen node of the network.

b) Packets are sent to their intended destination.

It is obvious that if phase 1 is run completely and then followed by phase 2, all packets will be delivered to their destination correctly. As Valiant and Brebner designed a routing strategy for physical networks, their algorithm can also handle multi hop forwarding. If in phase 2, the target of a packet cannot be reached via a direct link, then this packet will be forwarded into the correct direction.

The routing algorithm was designed to route packets in a n-cube where all nodes are logically arranged in a n dimensional hypercube which is also known as n-cube. Each node is given a unique n dimensional binary vector. A one in bit j indicates that a point belongs to dimension j. This correlates to placing the nodes on the edges of the n-dimensional hypercube. Every two neighboring nodes have hamming distance 1.

Routing in a n-cube is easier than in other network topologies, because the coordinates of neighboring nodes differ in only one bit. The protocol then directly can determine the next hop to which a packet has to be forwarded by choosing one of its neighbor nodes with lowest hamming distance to the target node.

2.6 Interrupt Response Time

For message passing interfaces, the performance impact of interrupt handling is a major concern. In normal operation mode, a message passing interface generates one interrupt request per incoming message packet. High bandwidth interfaces allow to receive more than one packet per interrupt request. The drawback of such an optimization is that it increases the transmit latency of individual packets because the first received packet is only delivered after a certain amount of packets has arrived or a timer has expired. Therefore the average time required to service an interrupt request remains critical for message passing interfaces.
Unfortunately, it is hard to find exact timing specifications for interrupt handling in current x86 CPUs. According to [Int98], the time between occurrence of an interrupt request and execution of the first instruction of the interrupt service routine is called interrupt response time. It consists of the interrupt latency and the time the OS needs to initialize the interrupt routine by saving CPU states. The interrupt latency on 386SX is lower than 63 clock cycles.
In Linux on IA32 initialization of the interrupt routine this is done in *linux/arch/x86/kernel/entry_32.S*. According to [Dan00] the average interrupt response time for fast interrupt handlers on optimized real time Linux OSs is lower than 1ms. But this time can increase a lot during heavy interrupt request loads. While servicing an interrupt request, no other interrupts

with the same interrupt number are accepted by x86 CPUs. In such scenarios of high network traffic, bandwidth optimizations can decrease average transmission latency even of single packets by receiving more than one packet per interrupt.

Chapter 3
Virtualization of Message Passing Hardware

After providing background information of the underlying technology, this Chapter describes our approach to use virtualization to simulate a certain class of many core CPUs.

3.1 Target Class of Message Passing Architecture

Our proposed research platform considers many core architectures with a message passing architecture that provides these two key features:

- A small region of shared memory used as message buffers
- A lightweight signaling mechanism used to send events between nodes

Whenever a certain message passing architecture can provide these features, its behavior can be correlated to this research platform. Such architectures are further referred to as message passing architecture (MPA).

3.1.1 Small Region of Shared Memory

Cache coherent memories of SMP CPUs share the whole memory range among all cores. Every core is able to share any page frame with other cores.

A MPA does only require a small amount of its total memory to be shared among all cores. This MPA even does not have to provide full coherency for this shared memory. Instead the shared memory can be divided into coherent fragments. Each fragment must only support coherency between two dedicated cores.

The message passing hardware can further be optimized by assuming that accesses to a fragment are mostly unidirectional. For N cores, each core requires N-1 coherent memory fragments further referred to as receive buffers. Accesses to a receive buffer should be local and fast for its dedicated core like a local cache.

Each receive buffer is associated with a sender node. Only this sender node is allowed to write to this buffer. The buffer in which a node receives data is further referred to as a receive buffer (or *RxBuffer*). From the perspective of a sender node, the receive buffer of a target node is further referred to as a target buffer (or *TxBuffer*). Using this convention, depending on the perspective each buffer can be a *RxBuffer* or a *TxBuffer*.

Figure 3 shows the different coherent memory fragments which are required for a three node setup. Each node reads from two *RxBuffers* and writes to two *TxBuffers*. In terms of coherency, the only assumption made for these buffers is that each of them is writable from one certain node and that the data can be read by another certain node.

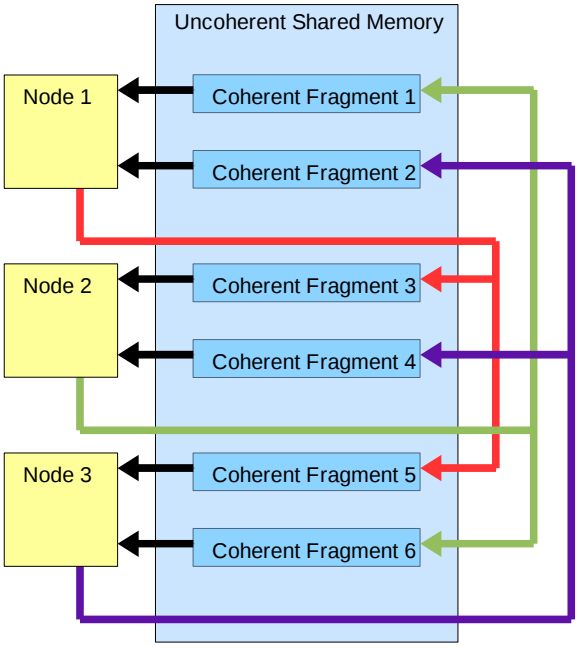

Figure 3: Unidirectional Message Buffers

3.1.2 Lightweight Signaling Mechanism

In a message passing architecture, each node must be able to signal other nodes when it wants to initiate a communication with the remote node. Basically two different signaling types can be differentiated:

- **Event polling**

  With event polling, each node periodically checks for incoming messages. When node 1 wants to send a message to node 2, it sets a signal flag in the receive buffer of node 2. The next time when node 2 checks this flag, it reads the corresponding receive buffer and processes it.

  This type of signaling has these two main disadvantages:

  1) Every node has to check all signal flags of all receive buffers. This affects compute performance even when no communication takes place

  2) When a signal flag is set, then the latency until the receive buffer gets processed can be much higher than with interrupt requests.
  In best case, node 2 reads the flag right after it has been set. In worst case, node 2 has read the flag right before it has been set by node 1. The time between best and worst case depends on the rate at which signal flags are checked. Higher rates mean lower latencies and higher impact on computing performance.

  The main advantage of event polling is that it does not require context switches to take place in the CPU.

- **Interrupt Requests**

  To request an interrupt is the natural way of signaling a CPU. Interrupts can be generated from hardware and software sources. A hardware source could be a slow device like a printer which signals to stop sending data because its receive buffer is full. Software can request a software interrupt or raise an exception.

The main advantages of Interrupts over event polling are these:

1) Nodes which do not communicate can use their full compute performance.

2) The response latency between sending a signal and the begin of its processing is smaller than for low rate event polling.

The main disadvantage of interrupt requests is their implicit context switch as described in Chapter 2.6 .

In practice, both techniques can be mixed together to combine their advantages. When an interrupt is requested, its service routine can actively poll for further events for a while. Often more than one message is received in a short period. Then this technique can reduce the amount of interrupts.

3.1.3 1 to N vs. Local Communication

Message passing architectures can be divided according to their connectivity into 1 to N and local communication architectures.

1 to N communication means that every node is able to directly send messages to all other nodes. Though this type of communication makes routing decisions easy, it may have the drawback of limited scalability.

A MPA with local communication allows every node to directly send messages to neighboring nodes only. To reach distant nodes, packets have to be forwarded by intermediate nodes.

This research platform is able to simulate both architectures.

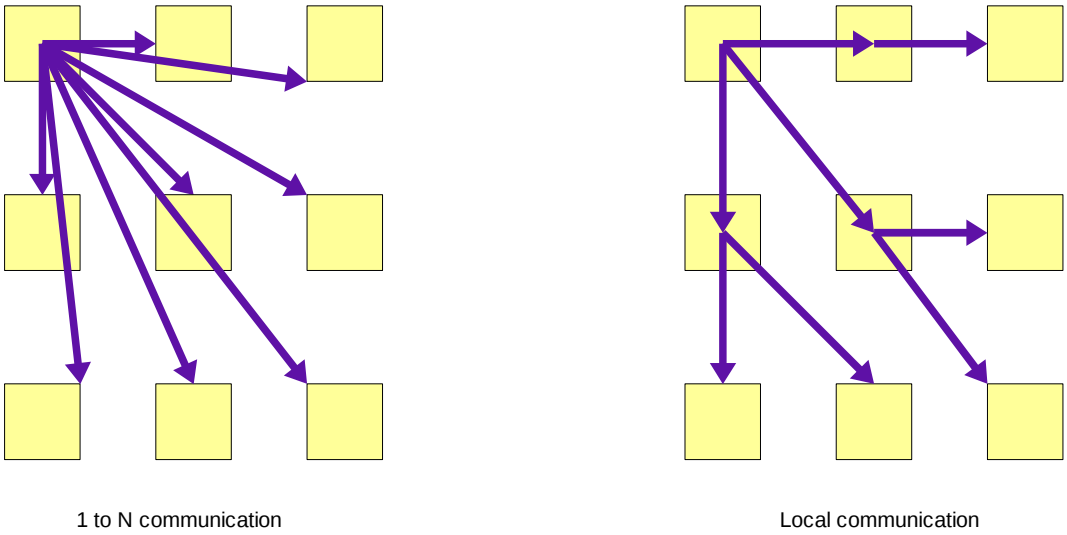

Figure 4: 1 to N versus Local Communication

Figure 4 shows the different paths of packets send from the upper left node to all other nodes on architectures supporting 1 to N communication or local communication.

3.1.4 Using Shared Memory for Message Passing

As lightweight signals do not carry message data, shared memory is used to buffer data being in transit. When the amount of available shared memory for each node is limited, then the designer has to choose among two main alternative usage models for a MPA consisting of N nodes:

- **Define less than N-1 shared buffers for each node**

  This means that each node owns less buffers than its possible connections to the N-1 other nodes. An arbitration protocol has to be implemented to allocate a buffer before a communication between two nodes can take place. This protocol requires a three way handshake to set up a communication channel:

  1) Sender sends a request to Receiver
  2) Receiver grants usage of a buffer to Sender
  3) Sender copies message into buffer

  After the message has been processed by the receiver, the buffer is available to be allocated again. This protocol incurs that the sender has to wait until the receiver grants a certain buffer to be used. For on-die message passing, the response latency of the receiver can be much higher than the time the sender node needs to send its data. This latency increases when the receiver node receives messages at a high rate.
  Another problem of this protocol is to guarantee a fair allocation of buffers to all sender nodes under high network loads.
  Several different optimizations have been developed for such protocols but for the rest of this thesis, we will concentrate on another type of message passing implementation.

- **Define N-1 dedicated buffers for each node**

  For this protocol, each node owns exactly as many receive buffers as there exist other nodes which may send a message. Whenever a sender wants to send a message it only has to check if its corresponding buffer at the receiver node is available. If the receive buffer is available, then the sender can immediately copy the data into the buffer and send a signal to the receiver.
  The big advantage is, that the sender does not have to wait for the receiver to answer. Messages are send asynchronously to the receiver. The receiver too sends its status update for the receive buffer asynchronously. Both nodes do not have to wait for each other for single messages.

  And even under high workloads, every node has the same chance to send a message to the receiver as all N-2 other nodes.
  One disadvantage of this protocol is, that it cannot dynamically adapt the communication bandwidth to the current traffic. As described later, a special routing protocol can provide this dynamical behavior without the need of three way handshakes.

  When the receiver node cannot process its receive buffers as fast as the sender nodes generate new messages, then the senders have to queue their outstanding messages for later delivery. This queuing is basically the same as for shared buffers. One exception can be that instead of one central wait queue, N-1 wait queues can be used. Each queue is dedicated to one target node. Whenever the queues get processed, up to one message can be send to the N-1 possible target nodes.

  Figures 5 to 7 show how this protocol would transport five messages from node 1 to nodes 2 and 3. During step 1, two packets have been delivered immediately to their targets. Three packets remained in the two output FIFO queues.

  After nodes 2 and 3 have processed and freed their receive buffers and during step 2, two additional messages can be delivered. Thus only one outstanding message remains in the FIFO queues of node 1.

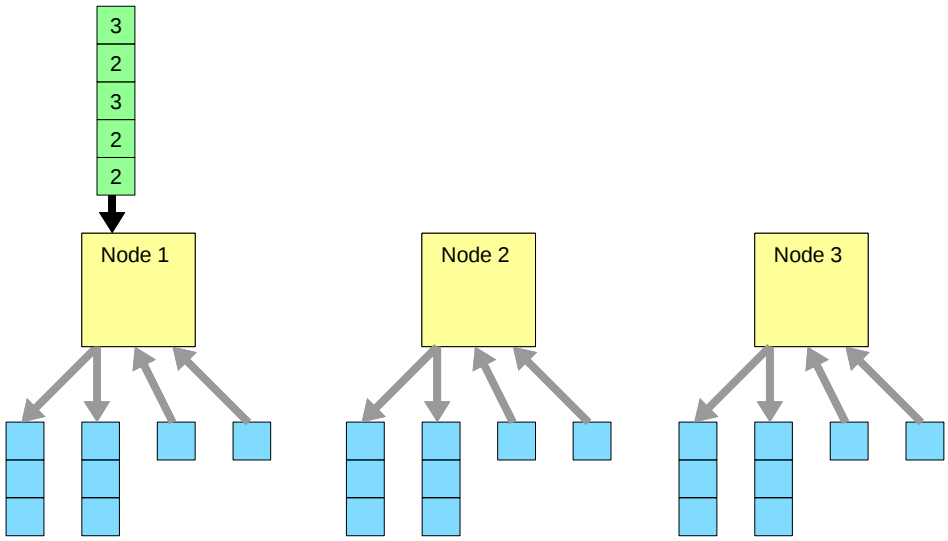

Figure 5: Multiple Output FIFO Queues per Node - Step 1

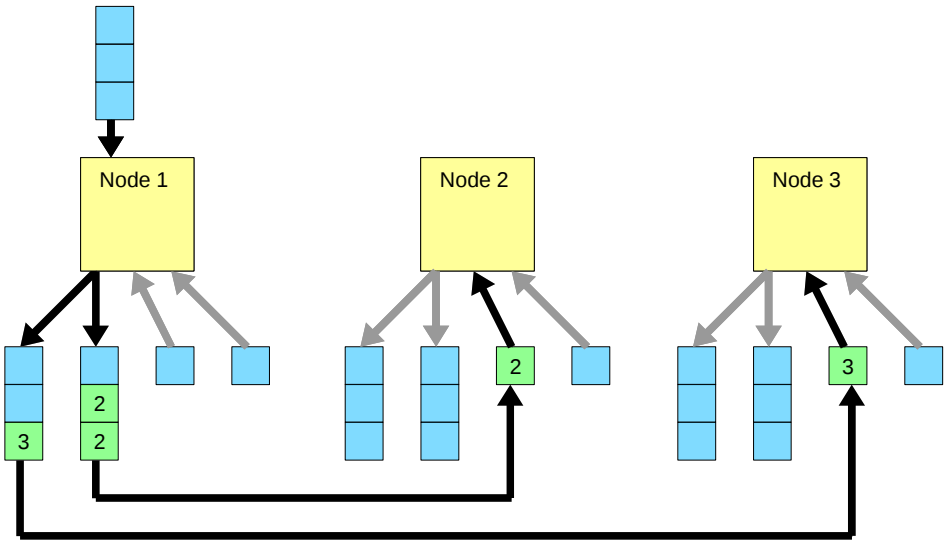

Figure 6: Multiple Output FIFO Queues per Node - Step 2

In this example, each node provides one dedicated receive buffer per other domain. Other implementations with more than one dedicated buffer per other domain are thinkable. The efforts in transistor count, that are required to share a certain amount of memory among hundreds of cores is enormous. Therefore it can be assumed that the amount of shared memory in a many core CPU is very small when compared to the total address range of the CPU. By splitting the available memory into more receive buffers could incur fragmentation of message packets and increase communication overhead. Because of this we will provide only one dedicated receive buffer per each sender-receiver pair.

As the FIFO queues can be placed in local memory of each node, their size is not critical for memory considerations.

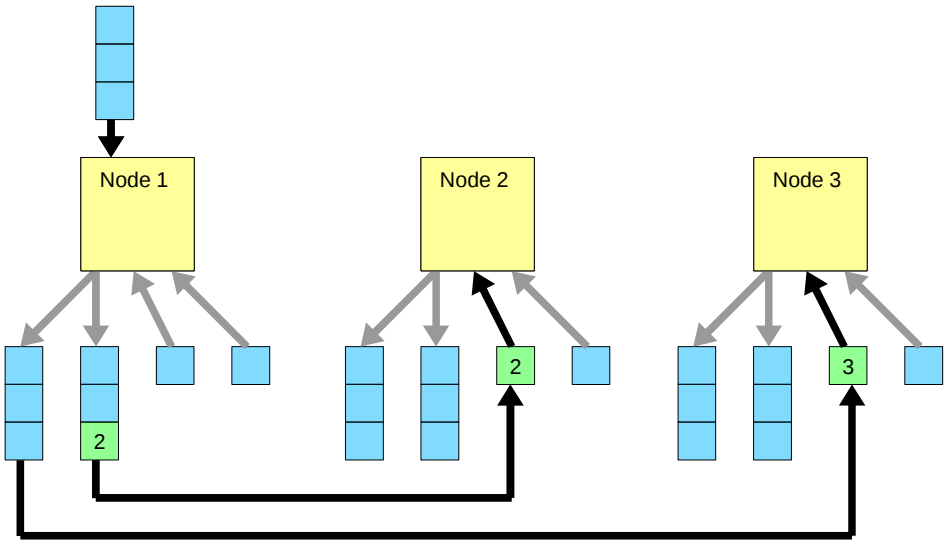

Figure 7: A Multiple Output FIFO Queues per Node - Step 3

3.2 Analogy Between Many Cores and Virtualized Operating Systems

This thesis assumes that the target many core architecture is able to run a duplicate operating system on every core. Such an operating system can be lightweight as long as it supports to run standard message passing applications such as an implementation of MPI version 1 or 2.

The basis for the analogy between a MPA and a virtualized operating system are these three requirements for the MPA:

- Small region of shared memory
- Lightweight signaling mechanism
- Each core can run its own operating system

If these requirements are met, then a set of virtualized operating systems on a multi core SMP machine can be seen as a generalized MPA. As benchmarks show, to be be comparable, it is vital that each virtualized operating system gets its own core. If more than one domain share the same core, then deadlocks can occur easily. This is because the virtualization technique applied here does not switch a domain which has entered interrupt mode.

3.3 A Virtual Network Interface for Inter Domain Communication

3.3.1 An Additional Network Interface

This approach provides message passing by loading an additional virtual network interface in every node. This network interface allows communication for most TCP/IP based software. The main difference to a real Ethernet device is the lack of broadcast capability.

The advantage of a fully functional network interface over a proprietary shared memory communication is that it supports the full range of existing benchmark software and applications. Though a low level buffer send interface does exist too.

3.3.2 Inter Domain Communication

The usual way to communicate among virtual domains is to implement a zero copy mechanism. This means that the sender domain declares a memory buffer as being shared with a receiver domain. The receiver then hands the start address of this buffer to its receiving application. Up to the application only references in the memory management are changed and no copy operations took place.

In order to simulate the behavior of real message passing hardware, this approach does actively copy bytes of data to and from data buffers. The aim of this thesis is not to build a better virtual network interface than that which is provided by the virtualization itself. Instead the copy approach allows to configure buffer sizes in a more flexible way.

3.3.3 Status of Implementation

The current implementation of this research platform allows to run any number of virtual domains and let them communicate over the provided shared memory network interface. Communication over the native virtual network interface is also possible as a comparison. A set of PERL scripts provide to start an set of domains and to run a MPICH v2 benchmark suite to compare the performance of different configurations.

3.3.4 Outlining the design space

The implemented software model of the targeted shared memory architecture is rather generic. E.g. routing costs between each two nodes are assumed to be constant and equal between each tow nodes.

In order to simulate the behavior of a certain message passing architecture, the source code has to be modified by inserting delays according to the specific delay penalties of the intended hardware architecture.

The network interface allows to change the MTU to simulate message buffers of different sizes. Every packet that is larger than the currently set MTU will be dropped and a rate limited message is printed to the logfile.

Benchmarking the implementation with different number of domains shows the scalability of a certain software model.

To test such different software models requires changes to the source code. Researchers are welcome to work on the code as the basic idea of this work is to enable research via virtualization rather than to do all possible research in advance.

3.4 Extending The Design Space With an Alternative Routing Protocol

In order to provide more configuration options for the network interface and to be able to simulate even architectures with local communication, a routing protocol has been implemented as an extension in the network interface. When activated, each packet carries an extra header with routing information. For a fixed size of message buffers, this decreases the payload size. The routing protocol offers a range of different configuration options allowing to extend the design space onto completely different message passing architectures.

For a deeper discussion on the pros and cons of Valiant Load Balance routing, we start with two definitions:

Definition: An "*evade node*" is a node or domain which accepts indirect data packets to forward them towards their intended target domain.

Definition: "*Foreign packets*" are received packets which are not targeted at the current node.

Packets which cannot be send directly may be send indirectly via evade nodes. The hop count for each packet is the number of evade nodes that the packet has visited until it reached its target node. The maximum allowed hop count is a configuration parameter.

3.4.1 Advantages of VLB Routing

- **More than one packet transmitted per context switch**

  One can think of architectures which can either compute or communicate. For such an architecture, to wait for the target to process its receive buffer could mean a major performance impact. Without VLB, transmitting P packets to the same target node would require P context switches on the sender node. This is because whenever more than one packet has to be sent to the same target, then only the first packet may be sent directly. Each other packet has to be send later, requiring an additional context switch.
  By use of VLB among N nodes, the sender can send up to N-1 packets to evade nodes at once before returning from the transmit function.

- **More than one packet received per interrupt**

  The implementation described in our approach uses N-1 fixed receive buffers for each of the N nodes. In best case, a sender can send one packet directly and N-2 packets indirectly to the target node. This means that the receiver can receive up to N-1 interrupt requests at once. Because of the nature of level triggered interrupts, while servicing an interrupt request no other request is recognized. So the receiver will miss most of these requests when receiving more than one packet at once. The software anticipates this by checking all local receive buffers for incoming data on every interrupt request. This strategy allows to receive up to N-1 packets within one interrupt request.

- **Can make use of whole network bandwidth for single transmission**

  NoCs typically provide many parallel communication buses because they have to provide several data transmissions on the whole chip at once. If all communication would have to pass a single concentration point, then the scalability would be poor. By using VLB, each node can send packets via every other node to the target. In best case all of these packets are transmitted in parallel by the *NoC*.

- **Does not increase latency of single packets**

  Even with activated VLB, whenever the remote receive buffer of the target domain is free, the packet gets transmitted directly.

- **Only local routing decisions**

  Several other bandwidth optimizations do exist for message passing interfaces. One optimization is to establish links between the source and target nodes before transmitting data. This technique always requires at least a two way handshake. One message has to be send to the target to request the link. The target domain has to enter its interrupt service routine and to send a message back to

the sender to grant the link. Instead of directly deciding on the current packet as down with VLB, such a protocol would add two interrupt response times (one at the target and one at the sender node) to transmit the first packet of a series or single packets. Additionally, if all links to a certain node are occupied, other transmissions to this node may be blocked completely.

- **Can automatically circumvent broken links**

  Whenever one node cannot send a packet to its target directly, the nodes will automatically send it indirectly. If only one link is broken, then overall performance should not be affected because packets will arrive after one evade node at their target. If a broken link is detected, then the protocol could dynamically increase the VLB_TimeToLive value of all packets which otherwise would have to be sent over a broken link.
  In contrast to most other routing protocols, this self healing takes place dynamically and does not need a central error detection.

3.4.2 Disadvantages of VLB Routing

- **Certain routing configurations can abort communication**

  When configured to send any number of packets indirectly to the target, then it can happen that the sender sends out packets at a much higher rate than the receiver can accept them. In such a case, the network gets flooded with outgoing packets, but only very few of these packets get acknowledged. This could confuse the TCP/IP stack because it can only handle a limited number of unacknowledged packets. The evade nodes get flooded with indirect packets. Many of these packets are retransmitted by the sender node. Therefore the current implementation of *sm_nic*_vlb provides several command line arguments to limit the relative amount of indirect packets.

- **Packet evading impacts performance of all other nodes**

  In a *NoC* consisting of N nodes, node A transmits P packets to node B with activated VLB. In order to keep mathematics simple, we first make two assumptions:

  - No other nodes are transmitting data at this time.
  - Packets may evade over one hop maximum (VLB_TimeToLive=1)

  Later we will generalize the results by superimposing the evade traffic of all nodes.

  From the perspective of node A, there is the target node and N-2 evade nodes. The first interesting parameter is the ratio of indirect (I) to direct (D) packets R_{max}. The value of R_{max} gives the maximum fraction of all packets from A to B which get routed via evade nodes:

  $$R_{max} = \frac{I}{D} = \frac{N-2}{1}$$

  The total receive capacity at the target domain T_{rx} is the sum of indirect and direct packets:

  $$T_{rx} = I + D = (N-2) + 1 = N-1$$

  If the maximum hop count is set to 1, which means that every evade node must transmit foreign packets directly, then the maximum number of indirect packets related to the total number of packets $I_{rel,max}$ describes the maximum ratio between indirect and direct packets. Therefore $I_{rel,max}$ is a measure for the maximum performance impact on the N-2 evade nodes from packets traveling A → B.

  $$I_{rel,max} = R_{max} / T_{rx} = \frac{N-2}{N-1}$$

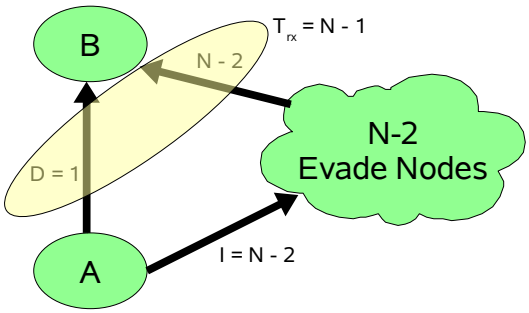

Figure 8: Direct and Indirect Capacities

Figure 8 shows the transmit and receive capacities at sender A and receiver B.

The maximum average performance impact of this transmission on a single evade node is $F_{max,avg}$. It shows the impact of a transmission A→B on every other node, assuming that the impact is the same on all nodes. This assumption is based on a VLB implementation which evades packets to random nodes.

$$F_{max,avg} = \frac{I_{rel,max} * P}{N-2} = \frac{P}{N-1}$$

The following table shows the performance impact of a transmission A→B on the evade nodes. Values are normalized to the number of transmitted packets P.

	N	10	100	1000	Total number of nodes
1)	$I_{rel,max}$	89.89%	98.99%	99.9%	Maximum indirect packets / total packets
2)	$F_{max,avg}/P$	11.11%	1.01%	0.10%	Amount of foreign packets on each evade node
3)	$N \cdot F_{max,avg}/P$	111.1%	101%	100%	Superimposed amount of foreign packets in case that all nodes communicate at once
4)	$\frac{1}{2} N \cdot F_{max,avg}/P$	55.55%	50.5%	50%	Superimposed amount of foreign packets in case that all nodes communicate 50% of the time

Though with increasing N nearly all packets are delivered indirectly, the impact on each evade node is diminishing.

In the case that all other nodes transmit data in parallel, it is assumed that the amount of foreign packets on each evade node $F_{max,avg}/P$ superimpose on all nodes as listed in the last two rows.

The calculation shows that for N=1000, a VLB routing strategy with VLB_TimeToLive=1 produces at maximum exactly twice the amount of traffic than without VLB. When all nodes communicate 50% of their time, then the other half of their bandwidth is occupied by foreign packets as shown in the last row.

If all nodes are communicating all the time, then the indirect packets add another 100% to the traffic of each node. This gives two times the traffic than without VLB routing which is not astonishing because with VLB_TimeToLive=1, in worst case, all packets travel over two links instead of one.

With increasing size of the NoC, the superimposed evade traffic seems to drop from 111.11% to 100% at 1000 nodes. This drop is based on the assumption that 1000 nodes can communicate with each other at the same constant latency as 10 nodes can do. This assumption is obviously

wrong. For actual numbers, this research platform can provide simulation results when implemented with real latency penalties.

Increasing the value of VLB_TimeToLive greater than one would increase the maximum bandwidth requirements further without providing real performance improvements. Such settings would only be useful, if a target node is not directly reachable from one evade node.

- **VLB routing doubles transmit latency**

  As seen for high bandwidth transmissions between two nodes in large networks, nearly all packets are transmitted indirectly. When packets are allowed to travel over maximum one evade node, this means that most of the packets travel over two node: Source Node → Evade Node → Target Node. So the packet latency is double the amount as for direct packets. This overhead is the only fixed overhead of VLB routing.

- **Packets travel over more than one hop**

  The VLB protocol basically allows each evade node to further evade foreign packets when it cannot immediately deliver them directly. Multiple evading may be required for certain NoC configurations where a node may not connect directly to all other nodes. To simulate such configurations, *sm_nic* allows to limit the allowed maximum hop count.

Chapter 4
Implementation based on XEN 3.0

After we have described our approach to a virtualized message passing system, this Chapter goes into details of the current implementation.

It covers the following topics:

- Specialties of XEN virtualization
- Installation and configuration of XEN
- Installation of a MPI benchmark suite
- Implementation of the three new kernel modules
- The scripts, that provide the user interface
- Known bugs in the current implementation

4.1 XEN as Virtualization Technique

XEN was chosen as virtualization technology, because it is 100% open source while still commercially usable. Though not needed for the current version of the platform, this would allow to modify the hypervisor to provide extra features if needed. This could for example be a changed event signaling mechanism which comes closer to level triggered interrupts as seen on real hardware.

When using Linux as the OS, XEN allows to run the same kernel for privileged as well as for unprivileged domains. This makes driver development easier because the source codes only needs to be compiled against one kernel source-tree for both domain types.
XEN v3.0 supports *SMP* with up to 32 CPU cores in total.

4.1.1 XEN Paravirtualization

Paravirtualized XEN introduces less compute overhead when compared to *trap-n-emulate* virtualizations (see Chapter 2.4.2). A guest OS must issue hyper-calls instead of executing privileged instructions. This means that no unprepared, proprietary OS can be run paravirtualized under XEN. Today several Linux distributions come with preconfigured XEN installations.

4.1.2 XEN Domain Concept

According to [Cam05a], each XEN system consists of multiple layers. The lowest layer contains the hypervisor only. The hypervisor runs in CPU ring 0 and has most privileges. The hypervisor may create several instances of guest operating systems which are called domains. Each domain is a virtual machine. The first domain is created automatically at system boot and is named domain 0 (*Dom0*).

Dom0 has special privileges and has the following responsibilities:

- Creating and starting other domains
- managing virtual devices of other domains
- suspending, resuming and migrating virtual machines
- providing memory to domains

Figure 9 shows the domains and privilege layers in a XEN virtualization. The hypervisor runs in CPU ring 0 and has the most privileges. *Dom0* runs in the same ring as all other domains, but it owns and manages the physical memory for all *DomUs*. *Dom0* also provides the device drivers for physical hardware in its backend modules.

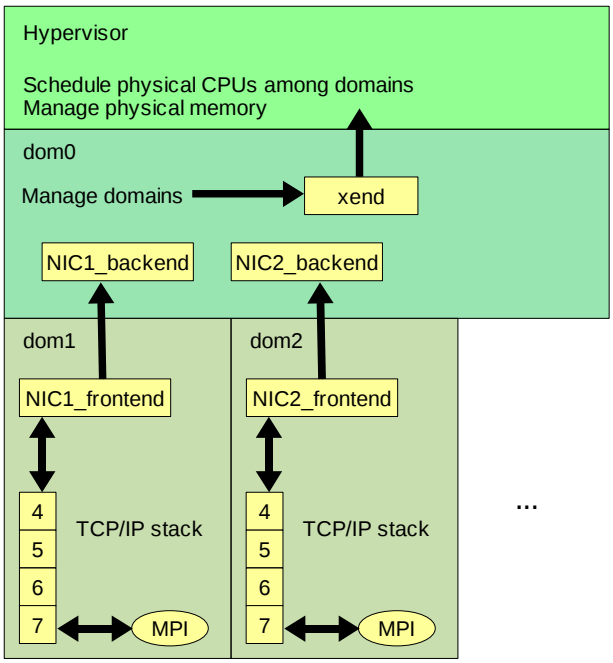

Figure 9: XEN Domain Concept

In order to manage domains, a special process called *xend* runs in *Dom0.* This process provides a HTML interface for the various actions.
All other domains are called user domains (*DomU*). Traditionally each *DomU* runs unprivileged versions of device drivers mainly provide only an interface to their counterpart drivers (called backend) which are loaded in *Dom0.*

4.1.3 Memory Management under XEN

XEN allows domains to share individual page frames from their memory contingent to be accessed by other domains. On IA-32, all bytes of one page frame share the same offset between their virtual and physical address. According to [Kra07], each process owns a pointer to its private page directory. That is mainly an array of pointers to page frames owned by the process. On IA-32, the page directory is changed by loading its pointer into the cr3 register. Whenever an address translation fails, the pageFault() function of Linux jumps in and loads the corresponding page from the swap space into main memory and corrects the page directory accordingly.

Because physical memory is allocated and freed on a page granularity, there is no guarantee that an application gets a desired amount of memory in a contiguous physical block. In order to simulate contiguous memory, XEN maintains a global table which translates guest-physical addresses to machine-physical addresses. Each domain additionally owns a local table to translate machine-physical addresses back to guest-physical addresses.

In the normal operating mode, XEN does not allow guest domains to write to the page tables. As the kernel code of user domains runs in a lower privileged ring than intended, each write access to page tables would cause a processor exception. The exception handler then emulates the instruction that caused the exception. This is called *trap-n-emulate*. XEN does support this technique, but it is much slower than *paravirtualization*. This is why this work makes use of *paravirtualization* only.

In the paravirtualized operating mode, each change to the page tables must be done via a special hypervisor call. This hypervisor call first validates the change request before its execution.

Though this special memory interface is incompatible with proprietary operating systems, it has a

big advantage over classical virtualization techniques. When an operating system can be modified to be compatible with this interface, then page table operations can be handled much faster than with classical trap and emulate techniques. This is because the hypervisor does not have to implement heuristics to find out the intention of a certain page table modification. Instead this interface even allows to bundle several page table modifications into one hypercall. This reduces the number of context switches for entering the hypervisor.

Another advantage of a cooperative guest OS is that this allows fast virtualization even on IA-32 processors without support for hardware virtualization.

4.1.4 Shared Memory with XEN

Sharing a page frame among several guest domains, requires two steps to be taken:

1. Grant the page to other domains

   The owner of a page frame issues a *gnttab_grant_foreign_access()* call for each domain for which access should be allowed. It directly modifies the global grant table. Each call returns a unique grant reference number. This number must then somehow handed over to the domain to which this page is granted. The handover can e.g. take place via a configuration filed on a shared file system. The current choice is to use a NFS folder which is shared among *Dom0* and all D*omUs*.

2. Map the granted page into the guest address space

   Each domain which should gain access to a shared memory page has to issue a hypercall carrying the XEN ID of the granting domain and the grant reference number.

   The official documentation [Cam05b] does only provide a brief overview of how to map a foreign page frame. A more detailed view of how single pages get mapped can only be derived from the kernel sources themself [Tor08]. Figure 10 shows the flow of such a map operation. The function HYPERVISOR_grant_table_op() which is defined inside hypercall_32.h directly jumps into the hypervisor. This is because only the hypervisor has the rights to create a new guest-physical to machine-physical mapping via create_grant_va_mapping() inside mm.c.

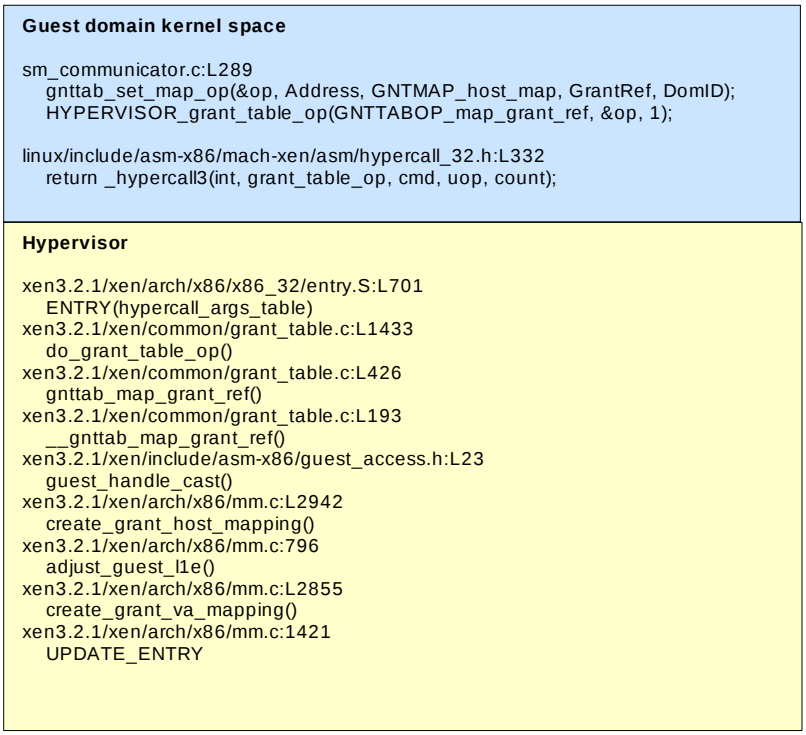

Figure 10: Granttable Hypercall

4.1.5 Inter Domain Communication Via Signals

XEN provides a lightweight signaling mechanism (see Chapter 5 of [Cam05b]). Signals only carry the id number of their originating domain. In order to be able to send a signal from Dom1 to Dom2, an event channel in reverse direction from Dom2 to Dom1 has to be setup before. The setup of an event channel is similar to page granting. One domain creates an event slot to be connected by another domain. The creation of this event slot generates a port index number. This number has to be transferred to the connecting domain like grant reference numbers for page frames. Only this one domain is allowed to connect to this event slot. After a foreign Dom2 has connected to a local event slot, a signal can be send to Dom2. Therefore to send signals between N domains, it is required for each domain to create N-1 event slots.

Setting up an inter domain communication is done in four steps:

- Step 1:

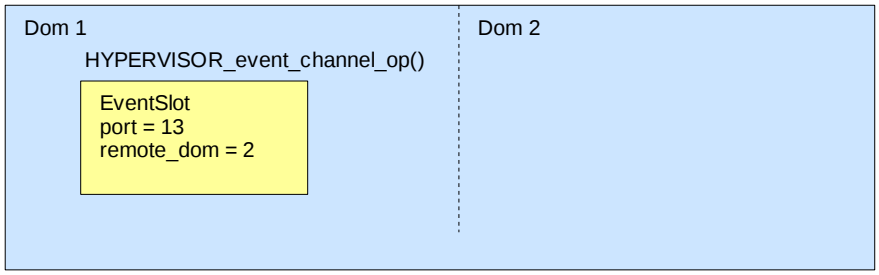

Figure 11: Allocate unbound event slot

Domain 1 issues a hypervisor call to create a new, unbound event-slot. The hypercall returns a new port number.

- Step 2:

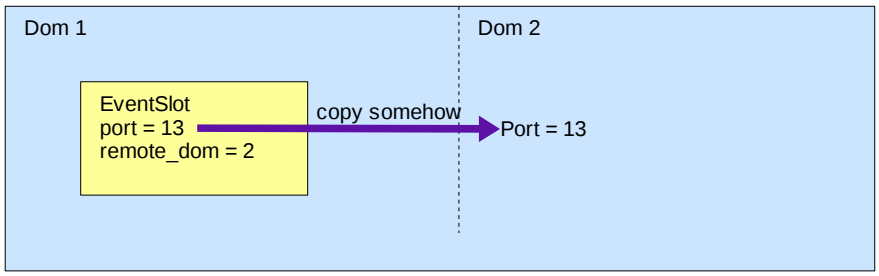

Figure 12: Send port number to other domain

The port number must now somehow travel to Dom2. This can e.g. be done via writing them to files which are shared on a NFS server for all domains.

- Step 3:

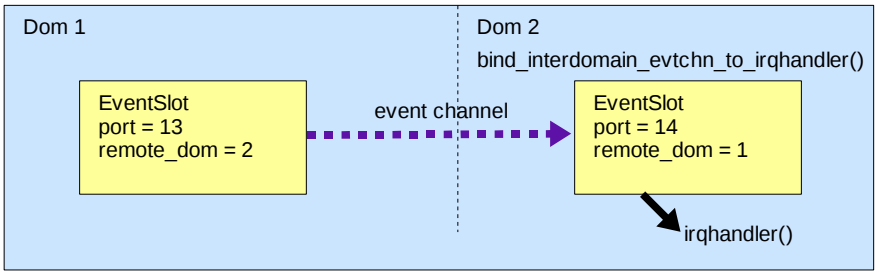

Figure 13: Create inter domain event-channel

Dom2 creates an event channel to Port 13 of Dom1. This automatically creates a new event slot and binds the provided function pointer as a callback function.

- Step 4:

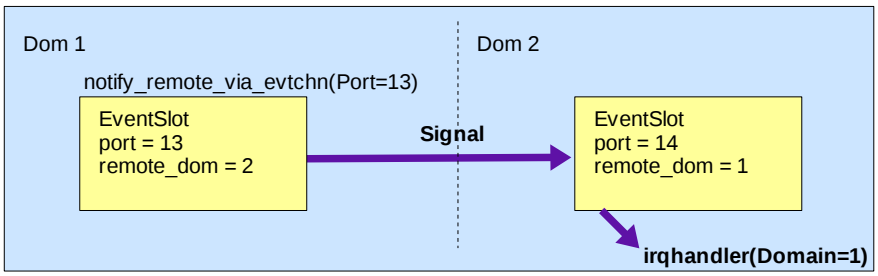

Figure 14: Send signal Dom 1 → Dom 2

Dom1 sends a signal via its local port 13 over the associated event channel. In Dom2, the registered function gets called inside an interrupt context to handle the signal. The domain ID of the caller is passed as argument to function irqhandler().

4.2 Installing the Virtual Research Platform

This Chapter describes how to install and configure a XEN virtualization with a current openSuSE Linux distribution.

4.2.1 Choosing Linux distribution

Many Linux distributions exist which come bundled with XEN like … ToDo more

4.2.2 Installing openSuSE v11

This research platform has been realized on openSuSE 11.0 with XEN 3.2.1.

Basically every Linux distribution can be used which provides this XEN version. Older versions of XEN are not compatible with *sm_communicator*.

When partitioning the hard drive, a partition of at least 50GB should be create for Linux.

During configuration of software packages, these packages are required:

- DHCP and DNS Server
- XEN Virtual Machine Host Server
- C/C++ Development
- RPM Build Environment
- Linux Kernel Development
- Perl Development

After issuing an online update, the machine can be rebooted with a XEN enabled kernel.

4.2.3 Obtaining the XEN Sources

It is preferred to use a distribution which comes with prepackaged XEN. XEN changed their distribution strategy since v3.2. The company no longer provides patches for vanilla Linux kernels. Instead only a package of one kernel (v2.6.18 at time of writing) and corresponding patches can be downloaded. The author was not able to boot a hypervisor that has been compiled from this package.
The XEN versions that come with distributions are heavily patched in order to work with the individual kernel. Therefore it is recommended to obtain a corresponding .src.rpm package and extract the source code from it. A script is provided to automatically download and extract the XEN sources in /XEN/install/. It may be required to update the URLs in this script.

4.2.4 Installing the Research Platform

An archive of this platform can be downloaded from http://rechner-architektur.de/mpi-research

After unpacking the with "*tar xf*", the script *install.sh* will unpack all directories into folder */XEN/*. If for some reason, the files should be installed in another location then a symbolic link has to point from */XEN/* to the installation folder prior to starting install.sh.

Figure 15 shows an example how to unpack the research platform v0.1.

```
mkdir /Data/XEN
ln -s /Data/XEN /XEN
tar xf mpi-research_0.1.tar
cd mpi-research
./install.sh
```

Figure 15: Unpack And Install The Research Platform

4.2.5 Installing the Required Software Packages

For openSuSE 11, several install scripts can be found inside */XEN/install/*. These are the most important scripts:

- ***install_XEN.sh***

  This script automatically downloads, compiles and installs all essential software packages. It is mandatory to install the applications listed in this script.

- ***install_NfsShares.sh***

  This script installs and configures NFS server as required. Two shares are added to the configuration file */etc/exports* automatically:

  - */XEN/share* used for data exchange between domains
  - */suse* should contain openSuSE 11.0 installation repository

- ***install_PBZIP2.sh***

  PBZIP2 provides an alternative implementation of bzip2 which makes use of all available CPU cores. After installing this compressor, *smx_BackUpCreate.pl* should run much faster on multi core machines.

4.2.6 Configuring a DHCP server

The guest domains require access to a DHCP server to obtain an IP address for their XEN network interface. The software setup described above should already have installed a DHCP server. On openSuSE, the DHCP server can be configured with yast or yast2 which uses a graphical user interface. The DHCP Server Wizard can be found in the network services section.

The DHCP server has to be configured to provide a range of IP addresses for all guest domains. This server does not require to run automatically at boot time. The script *smx_Control.pl* will automatically start its service if required.

Figure 16 shows an example configuration of a DHCP server that allots IP addresses in the range of 172.28.248.221 to 172.28.248.236.

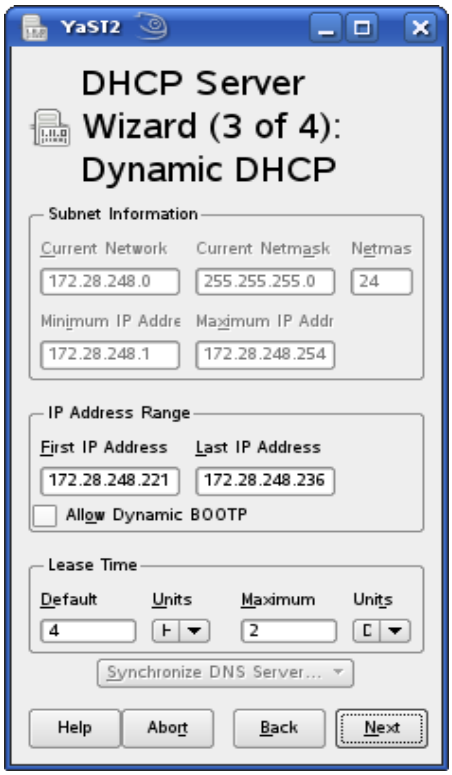

Figure 16: DHCP Server Configuration

4.3 Setting up Master and Guest Domains

Two OS image files come with the research platform, one for a so called "*master*" domain and the other for the so called "*guest*" domains. The *master* domain is used to configure all *guest* domains. . Only the *guest* domains are later used to run the research platform.

Note: The term master domain used in this chapter has nothing to do with the guest domain that loads a master instance of a kernel module as described in Chapter 4.9 .

4.3.1 Creating Duplicates of Guest OS Image files

The software package provides two image files of root filesystems in the folder */XEN/images/*:

- ***root_master.img***

  This root filesystem contains a service OS which automatically configures all guest image files.

- ***root_guest1.img***

  Each guest OS requires its own root filesystem. The initial software packages provides one image file which needs to be duplicated and configured for each guest OS. The script spreadGuestImagesFrom1.pl creates any number of duplicates from *root_guest1.img*.

This example shows the duplication of image files for 15 domains:

```
cd /XEN/images/
./spreadGuestImagesFrom1.pl 15
```

4.3.2 Creating The Master Domain

The *master* domain boots the image file */XEN/images/root_master.img*. Its purpose is to mount and configure all *guest* image files.

XEN domains can be created with the graphical tool *virt-manager*. This tool can be launched directly from the console or from the graphical YaST2 application.

Image files of all *guest* domains need to be added as writable discs to the *master* domain. The memory size and number of virtual CPUs is not critical and can be left at their defaults.

After creating the XEN master domain, it has to boot once to configure all domains.

Figure 17 shows an example configuration of the *master* domain for fifteen *guest* domains.

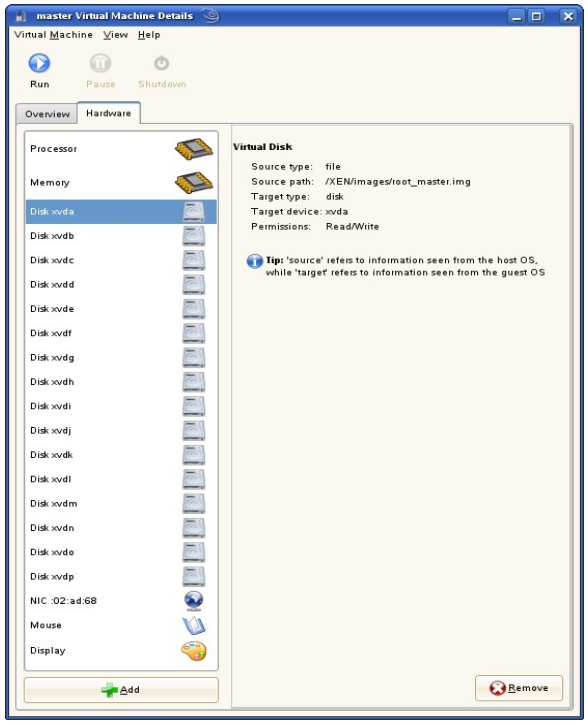

Figure 17: Master Domain Configuration

During boot up, the script */XEN/xen_StartUpGuest.pl* gets invoked automatically. This script mounts every disc device starting at */dev/xvdb* as */root_guestN,* N a running number. It then configures all *guest* OSs according to the configuration file */XEN/XendDomain.cfg* inside the master OS.

The configuration file itself provides comments that describe all available configuration options. Most of the default settings can be left unchanged.

Two important settings will have to be changed:

- ***XenShareServerIP***

  This option defines the IP address of a NFS server which exports the folder */XEN/share/* and */suse/*. If the NFS server runs on the local machine in *Dom0*, then the IP address of eth0 in *Dom0* has to be provided here.

- ***SshPubKey***

  Several external SSH public keys can be configured. Each key is copied to /root/.ssh/authorized_keys of each guest image file. This allows to connect to each user domain from the outside without the need to type in passwords.

  It is required to define at least the public SSH key of user root in *Dom0* here.

  A pair of private and public SSH keys can be generated with *ssh-keygen*. This creates two new files in *~/.ssh/*: *id_rsa* and *id_rsa.pub*. The content of *id_rsa.pub* then has to be pasted after the keyword *SshPubKey*. The keyfile can be copied into a running master domain via SSH. The default root password for all domains is root.

  SSH public keys for all guest domains are created and distributed automatically.

After saving the configuration file, the script */XEN/xen_StartUpGuest.pl* will automatically configure all *guest* image files and create *guest* configurations in the folder */XEN/*. These *guest* configuration files are read during boot of each *guest* domain later.

After successfully configuring all guest image files, the master domain can be shut down.

4.3.3 Creating All Guest Domains

Virt-manager with XEN 3.2 uses a configuration file scheme which makes it difficult to automate the configuration of a large number of guest domains. Therefore the recommended way to set up e.g. 15 domains implies to iterate one setup 15 times.

Each guest domain should own 1 virtual CPU and up to 512 MB of RAM. Each domain has to mount one guest image filesystem with write access and the master image as read only. The domain name has to follow the scheme xncN, where N is a free running number.

Figure 18 shows a sample setup of one guest domain.

Figure 18: Guest Domain Configuration

This table shows an example setup of four guest domains:

`Name: xnc1`
`Memory: 384-384 MB`
`Discs: xvda /XEN/images/root_guest1.img rw`
`xvdb /XEN/images/root_master.img ro`
	`Name: xnc2`
	`Memory: 384-384 MB`
	`Discs: xvda /XEN/images/root_guest2.img rw`
	`xvdb /XEN/images/root_master.img ro`

`Name: xnc3`
`Memory: 384-384 MB`
`Discs: xvda /XEN/images/root_guest3.img rw`
`xvdb /XEN/images/root_master.img ro`
	`Name: xnc4`
	`Memory: 384-384 MB`
	`Discs: xvda /XEN/images/root_guest4.img rw`
	`xvdb /XEN/images/root_master.img ro`

4.4 Testing the Research Platform

After creating and configuring all *guest* domains, the platform can be started for the first time.

All scripts which control the research platform from *Dom0* are located inside the folder */XEN/bin/*. Their filenames start with the prefix *smx_*. Adding this directory to the search path allows to omit the */XEN/bin/* prefix on every call.

The detailed synopsis and description of all arguments of all scripts described here can be found in the Appendix A.2 .

4.4.1 Starting And Stopping All Domains

The central script to start or stop all domains which name is of type xncN (N a number) is /XEN/bin/*smx_Control.pl*. The following command starts all domains:

```
smx_Control.pl start
```

Figure 19 shows the startup of fifteen domains in the tool virt-manager.

After all domains have been started one should log in as root via SSH from *Dom0* to every domain. The domain names and their IP addresses should have been updated in /etc/hosts of *Dom0*. This allows to connect to xnc1, xnc2, .. instead of to their IP addresses like shown below:

```
ssh xnc1
```

This should log into xnc1 automatically without asking for a password. During first login a security question may have to be answered.

Note: *Dom0* must be able to log into all *DomU*s via SSH without being asked for a password. If the SSH server in a domain asks for a password then the MPI library won't be able to send its jobs to this domain. Refer to Chapter 4.3.2 in case it is not possible to automatically log into every guest domain via SSH.

4.4.2 Launching Shared Memory Network Interfaces

In order to allow guest domains to communicate over the shared memory network interface *smn*, a set of kernel modules needs to be loaded in a so called master and two or more slave domains. The master grants memory pages and prints out a list of reference numbers. These numbers have to be provided as command line arguments during loading the module in the slave domains. After all slaves have loaded the module *sm_communicator.ko*, the basic functionality for shared memory is provided and the network interfaces can be installed.

This example shows the order in which modules get loaded in three domains:

1. master: insmod *resource_tracker.ko*
   insmod *sm_communicator.ko*
2. slave1: insmod *resource_tracker.ko*
   insmod *sm_communicator.ko*
3. slave2: insmod *resource_tracker.ko*
   insmod *sm_communicator.ko*
4. slave1: insmod *sm_nic.ko*
5. slave1: insmod *sm_nic.ko*

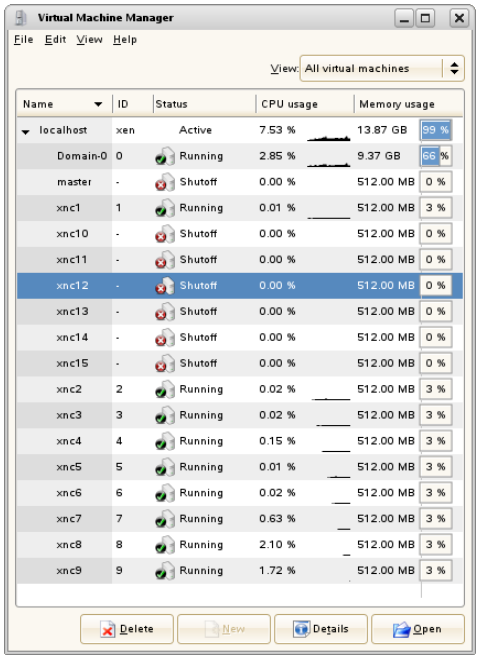

Figure 19: Startup of all Domains

A script called *smx_LoadModules.pl* is provided by this platform to load all required modules in the correct order and with all required arguments for any number of domains. The script takes up to three arguments. Beside the start or stop keyword, the name of the module to load is required. By default, the compile script */XEN/source/_compile.sh* builds two versions of the network interface: *sm_nic.ko* and *sm_nic_vlb.ko*.

The difference between the two modules is that the vlb version implements the VLB routing protocol, which is discussed in Chapter 4.10.3 in more detail. With VLB being compiled in, each packet carries an extra VLB header. Thus introducing an extra protocol overhead.

Example for loading kernel modules in all domains:

/XEN/bin/smx_LoadModules.pl start *sm_nic.ko*

4.4.3 Testing Connectivity of Slave Domains

When all modules have been loaded successfully in all domains, then the domains *xnc2*, *xnc3*, …, *xncN* now show an additional network interface named *smn*. These domains should now also be reachable as *smn2*, *smn3*, … , *smnN* for any network application like SSH or ping.

The special script *smx_TestSSH.pl* logs into all domains and tries to open a SSH connection to all other domains. To test connectivity among domains *xnc2*, ..., *xnc5*, one may issue the following:

smx_TestSSH.pl smn xnc2 xnc3 xnc4 xnc5

The first user domain which has loaded the module *sm_communicator.ko* as master does not provide a *smn* network interface.

4.4.4 Running Benchmarks

Two scripts allow to run an individual benchmark or a series of parameterized benchmarks:

- /XEN/bin/smx_Benchmark.pl runs a single benchmark
- /XEN/bin/smx_RunBenchmarks.pl issues a series of benchmark runs

These scripts are described in more detail in Chapter 5 and in the Appendix A.2 .

4.4.5 Troubleshooting

The table below shows an overview of the steps that are required to take to successfully pass Chapter 4.4.3 together with some hints of what may go wrong in each stage:

Steps	**Hints**
Choose Distribution	Linux distributions other than openSuSE 11 should also be able to run the research platform running. Though this has not been tested. It took a while to collect the list of all applications required to install. The install scripts in /XEN/install/ will only run on openSuSE. This distribution has been chosen because it is widespread, well known and provides a reliable environment with practically every development tool.
DHCP Server	Ensure that the server is running by issuing “rcdhcpd status” Open a virtual console of a running guest domain and check that it has assigned

	an IP address to eth0 by issuing “ifconfig eth0”. The root password is root by default in all *DomU*s.
NFS Server	Check that NFS server is running in *Dom0* by issuing “rcnfsserver status” Watch /var/log/messages in *Dom0* for error messages from NFS server. Try to mount the NFS shares from inside *Dom0* by issuing “mkdir /v; mount <IP_OF_*Dom0*>:/XEN/share/” (Replace <IP_OF_*Dom0*> by IP address of *Dom0*!).
Configure Domains	Check that you can log into every guest *DomU* from *Dom0* via SSH without being asked for a password. If you get asked for a password then login with password “root” and check if the correct SSH public key has been added to /root/.ssh/authorized_keys. Check that /etc/fstab in each *DomU* lists the correct IP address for /XEN/share/. If the guest root filesystems is misconfigured then refer to Chapter 4.3.2 .
Create Guest Domains	Check that each domain is assigned the correct filesystem images and has the correct name as described in Chapter 4.3.3 .
Start Domains	Several log files take error messages from various stages in different domains: *Dom0*: /XEN/share/xncN.log log module loading in all *DomU*s /var/log/messages logs messages of local servers in *Dom0* *DomU*s: /var/log/messages logs module loading in each *DomU*
Load Modules	If smx_LoadModules.pl gets stuck during step 6, then a deadlock may have occurred during setup of event channels inside sm_communicator.ko. Check the output to /var/log/messages of one of the slave domains. If the same status message is repeated without progress then the modules got stuck. A solution would be to kill all /usr/bin/perl processes and to restart the whole platform via smx_Control.pl.
Benchmarking	All log files regarding benchmarking are created in */XEN/share/Benchmarks/*. The script *smx_RunBenchmarks.pl* creates one file with filename prefix “run.”. Each individual benchmark creates its own report file with prefix “log.”.

4.5 Directory Structure

All files of the research platform reside in the central folder */XEN/* of *Dom0*. Though it is possible to replace this folder in the root directory by a symbolic link to a different location, this may lead to problems with the NFS server. The NFS servers of current Linux distributions do not allow to follow symbolic links on the server. Therefore if */XEN/* is a symbolic link, it may be required to change the scripts in /XEN/share/bin/ and /XEN/*xen_StartUpGuest.pl* inside the master image file to mount the link's destination instead of */XEN/share/*.

All folders are accessible from *Dom0*. A subset inside */XEN/share/* is only visible via a NFS share from each *DomU*.

4.5.1 Important Folders in *Dom0*

- **/XEN/backup/**

  Backups of the current state of the whole research platform can be created or restored via smx_BackUpCreate.pl and smx_BackUpDelete.pl.
  The script smx_CreatePackage.pl creates a single container package from one backup in this folder.

- **/XEN/bin/**

  The scripts in this folder form the text user interface of this platform. It is helpful to add it to the PATH environment variable.

- **/XEN/images/**

  The image files of virtual hard drives for the master and all guest domains are stored here. A ll guest image files are duplicated from the file */XEN/images/root_guest1.img*.

- **/XEN/install/**

  Several scripts in this folder allow to automatically install the required software packages on openSuSE v11. Other distributions are not supported directly but the install scripts should provide a documentation of required packages.

- **/XEN/source/**

  The source code of the three new kernel modules and a compile script are located here. In order to compile, the Linux kernel sources from /XEN/backup/common need to be extracted to this folder.

- **/XEN/share/Benchmarks/**

  Every benchmark run creates one log file here.
  Two scripts get created automatically during each run:

currentLog.sh	displays the log file output of a currently running benchmark
resumeBenchmarkRun.sh	starts a set of runs beginning where a former run crashed

4.5.2 Important Folders in *DomU*

- **/XEN/share/**

  Each *DomU* must be able to mount this folder from a NFS server and allowed write access to it. The folder serves as a way of communication between the text user interface scripts and the scripts that are running inside the *DomUs*.

- **/XEN/share/MPI/**

  A complete copy of the MPICH2 installation of each *DomU* is saved here. This folder is also used to copy the current Intel MPI Benchmark suite onto each *DomU* prior to each benchmark run.

- **/XEN/share/bin/**

  Scripts which are executed from inside each *DomU* are to be found in this folder.

- **/XEN/share/modules/**

  This folder stores the latest compiled kernel modules. Each *DomU* loads its modules from here.

4.6 Coding Style

The source code of the kernel modules described below is written in native C language style according to ISO C90 syntax and obeys to some coding styles in order to improve its readability and to reflect a continuous style. Mainly three style paradigms are implemented: Text folding, naming schemes and continuous comments.

4.6.1 Text Folding

The author believes that text folding can improve the readability of source code. The source code of the three new kernel modules and all SHELL and PERL scripts have been optimized for folding capable text editors. This is the reason why text folding is described in detail here.

Folding is a feature of some text editors which allows to collapse a range of text between two special fold marks into one single line. This single line is mostly the very first line of this range. The fold marks can be defined as any strings. The setting used here are curly brackets { } as fold-start and -end marks.

The example source in Figure 20 shows an excerpt from the source code of *sm_communicator*.c as an example of how folding can look like.

In this example, every opening curly bracket except in line 1 hides several lines of code. This view allows to see the basic control flow of module initialization of this module.
Some text editors like e.g. jEdit even detect fold marks inside comments. This allows to create additional logical blocks which can be folded too like in line 3 or 21. Here the only reason for placing curly brackets is to enable folding for a range of code. A decent comment describes what is implemented inside.

Text folding is supported by several text editors like jEdit or Kate.

```
1   static int m_init(void) {
2
3      //{ local variables..
13     if (!is_running_on_xen())  return -ENODEV;
14
15     if (debug_level > 0) {     // print startup message
19     rt_OwnerID = rt_create_owner("sm_communicator");
20     tidy_up();                 // prepare memory-buffers
21     //{ allocate dynamic buffers
70     if (master) {              // a master creates page-grants
117    else {                     // a slave maps pages to local memory
118
119    if (ErrorStatus) {         // tidy up + return error
135    return 0;                  // module loaded successfully
136 } //m_init()
```

Figure 20: Example of Text Folding

4.6.2 How To Activate Folding With jEdit

We will show how to activate folding with the text editor jEdit as an example. This editor has been *chosen* because it is free ware, runs on many platforms and it is easy to configure and extend.

The editor can be downloaded from jedit.org. The user may choose among a Windows binary or a platform independent jar installation archive. The jar archive can be started from the command line via „java -jar <FILE.JAR>“, <FILE.JAR> being the file name of the downloaded file. The installation should take place almost automatically. The editor can be installed by user root for all users or by an individual user in its home folder. The installation automatically detects which install

folder is writable and fills out the text fields accordingly.

To enable folding with this editor, a plugin has to be installed. To install plugins in jEdit, the plugin manager has to be opened. The plugin to be installed is called Configurable Fold Handler. After clicking on install, the module should be downloaded from the Internet and installed automatically. A restart of the text editor may be required to enable the new plugin. Figure 21 shows the plugin manager of jEdit with a list of already installed plugins.

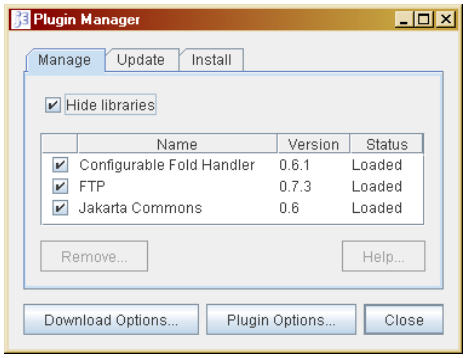

Figure 21: jEdit - Plugin Manager

If the plugin manager cannot download the plugin or even the list of plugins then the list of download mirrors may have to be updated or the proxy settings for the local network be configured. These settings can be find behind the button „Download settings“ in the plugin manager. Figure 22 shows the dialog that allows to update the mirror list and to choose the current download mirror.

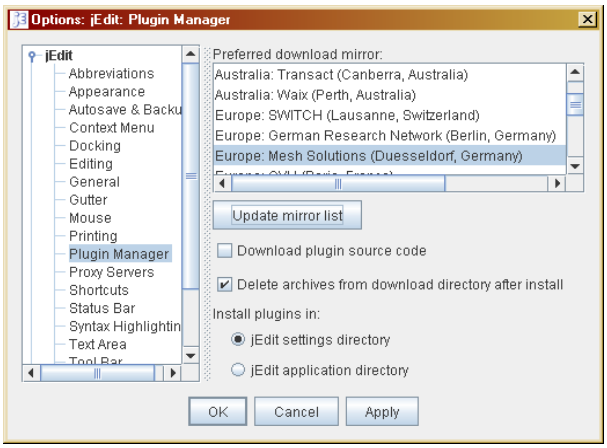

Figure 22: jEdit - Download Mirrors

4.6.3 Naming Schemes

All function names obey standard Linux kernel naming scheme as described in [Lin?]. Function names contain only lowercased characters, numbers and the underscore. Global and local variables have self explaining unabbreviated names (E.g. SmallestSlaveDomainID instead of SmSID).

4.6.4 Continuous Comments

All comments start in the same column whenever possible. This allows to read only the comments or the source code very easily as it forms a visual two column effect of the text. Only fold marks enclosing a logical block of code (like in line 21 above) start at the current code indent.

4.7 Implementation of the Three New Kernel Modules

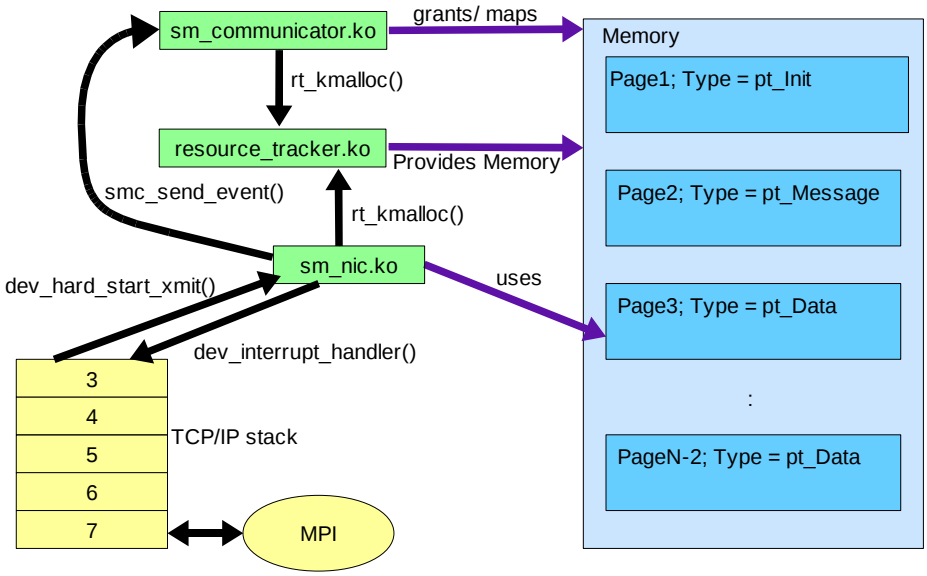

Figure 23: Software Layout of One Domain

Figure 23 shows the basic layout of the shared memory communication loaded in each domain.

The three colors have different meanings:

- **Yellow**

  The existing software parts are represented by a network application (e.g. an MPI application) and the TCP/IP stack. The network application directly communicates with the TCP/IP stack via a socket buffer interface. The TCP/IP stack then directly communicates with *sm_nic* via a kernel module interface for network card drivers as described in [CRK05] Chapter 17.

- **Green**

  The kernel modules that have been developed for this thesis consist of *resource_tracker*, *sm_communicator* and *sm_nic*. From the perspective of the TCP/IP stack, only *sm_nic* is visible. This module implements a fully functional network card interface. It makes use of shared memory pages and a basic signaling functionality which are provided by *sm_communicator*.

- **Blue**

  The shared memory page frames build the basis to to realize a network communication . Sharing page frames is implemented in *sm_communicator*. Module *sm_nic* uses this shared memory to realize a message passing interface over shared memory receive buffers.

The next three sections describe the functionality of each kernel module in more detail.

4.8 Universal Kernel Resource Tracker *(*module *resource_tracker)*

The resource tracker is a universal module which has been developed to make tracking of resource allocations easier. Whenever a kernel module allocates dynamic memory via *rt_kmalloc()*, *rt_alloc_pages()* or similar functions, this module stores this allocation data allowing to deallocate the resources automatically at any time later. The module can be seen as a simple garbage collector.

Features provided by the resource tracker:

- Allocate kernel pages and plain memory
- Keep track of all allocated pages and memories automatically
- Handle resources of different client-modules
- Check for valid arguments and print corresponding error-messages
- One call frees all resources of one client-module

Each module, that wants to use the resource tracker, needs to acquire an unique owner identification number. This is done by calling a registration function which generates a new number. This owner identification has to be provided with each call one of the functions described in the next section.

This module has been written during this thesis, but it can be helpful when developing any other kernel module as it makes resource tracking easier and less error prone.

4.8.1 Functions Being Exported by *resource_tracker*

All functions listed here are described in more detail in Appendix A.1

- *rt_create_owner*() registers a string as new memory-owner
- *rt_pages_allocate*() allocates a number of memory pages
- *rt_kmalloc()* allocates an amount of memory via *kmalloc()*
- *rt_kmalloc0*() allocates an amount of memory via *kmalloc()* and fills it with zeros
- *rt_kfree*() frees one memory block being allocated by *rt_kmalloc()* or *rt_kmalloc0*()
- *rt_free_resources*() frees all kernel resources whether pages or memories of one owner

4.8.2 Typical Usage Scenario of *resource_tracker*

Figure 24 shows an example code that uses the *resource_tracker* module to allocate memory.

When a user or a script issues an *insmod* on a kernel module, after some basic initializations, its init function gets called automatically. For further details on how to program kernel modules in detail see [CRK05]. In this case, m_init in line 14 serves as an init function to the module and is called upon a module load. The first thing to do is to create a new owner id from a given identifier string. This string must be different from those of all other modules which might use the resource tracker concurrently. The current module name suits perfectly here.

Note: If a module can be loaded several times, then each of these instances has to register its own unique identifier string!

After creating an ID number, a block of zeroed memory is allocated in line 16. Instead of returning the start address like *malloc()*, the preferred paradigm with resource tracker is "*call by reference*". The provided pointer &A is stored inside *resource_tracker*. When an individual resource is freed later, then A is reset automatically. Therefore it is important to use a global or local static variable to hold the start address. This behavior is specially interesting when it comes to local buffers inside functions. It is difficult to keep track of all of these distributed dynamic allocations. One could store these start addresses inside a global array and free them at program exit inside a for loop. To do so would mean to start to implement something like resource tracker itself. It gets even more complicated when these buffers should be resizable. When a function decides to resize its dynamic buffer, it basically has to free the old one and to delete all pointers to it. A *rt_kfree*() call on a start

address frees the memory block and resets the registered pointer to it.

```
 1 #include "resource_tracker.h"      // inside header file
 2
 3 unsigned int rt_OwnerID = 0;
 4 unsigned int *A = NULL;           // important to initialize with NULL
 5
 6 static void privateA(void) {      // private function which may get called
 7   static char *Buffer = NULL;
 8   if (!Buffer)  rt_kmalloc0( rt_OwnerID, sizeof(char) * 10,
 9                              GFP_KERNEL, (void**) &Buffer,
10                              "privateA() Buffer" );
11   sprintf(Buffer, „Hello World\n“);
12 }
13
14 static int m_init(void) {          // module initialization
15   rt_OwnerID = rt_create_owner("MY_MODULENAME");
16   rt_kmalloc0(rt_OwnerID, sizeof(int) * 100, GFP_KERNEL, (void**) &A, "A");
17   A[42] = 1;                        // use dynamic memory
18
19   return 0;                         // module loaded successfully
20 }
21
22 static void m_exit(void) {         // module gets unloaded
23     rt_free_resources(rt_OwnerID); // frees A[] and Buffer[] (if allocated)
24 }
25
```

Figure 24: Example Usage of Resource Tracker

It is also possible to use *rt_kmalloc()* and *rt_kmalloc0*() like normal *malloc()* since the buffer start address is returned too. In order to disable the automatic reset feature, a NULL value can be provided as fourth argument.

4.9 Basic Shared Memory and Signaling (module *sm_communicator)*

The shared memory communicator provides basic features to enable communication between individual domains:

- **Shared memory page frames**

  It provides a configurable number of consecutive pages of shared memory. After *sm_communicator*s have successfully initialized in all participant domains, the list of shared pages can be queried via a function call.

- **Lightweight event signals**

  After loading the modules in all configured domains, signals can be send from every slave domain to any other slave domain. Each signal carries only the XEN ID number of its originating domain. To send data from one domain to another, shared memory must be used in addition to signals.

In order to let N domains communicate with each other, the *sm_communicator* has to be loaded in N+1 domains. At first, one domain loads this module in master mode. The remaining N domains load *sm_communicator* in slave mode.

The master allocates and grants a given number of memory pages for all given slave domains. It is possible to load a master in *Dom0* or any other *DomU*. Loading a master in a *DomU* blocks this

domain from loading an instance of *sm_nic*. To reserve an extra domain for the master instance gives and advantage during module development. If a modified version of *sm_nic* produces a *kernel oops*, then only its *DomU* is affected and needs to be restarted. In many cases, a *kernel oops* in a slave domain gets logged in the master domain before the slave gets stuck.

Three different types of pages can be granted by *sm_communicator*:

- **pt_Init**

  Pages of this type are used to setup one event-channel between each two domains. These event-channels allow to send signal events from any domain to any other domain which has loaded *sm_communicator*. The number of pages to be allocated of this type depends on the number of connected slaves and is calculated during module initialization.

- **pt_Message**

  A single message page is used by *smc_printk*() to transmit text messages from the slave instances to the master instance of *sm_communicator*. One page is dedicated for this purpose because this makes *smc_printk*() independent from other transport mechanisms being implemented inside *sm_nic*. Thus allowing to debug a broken network interface.

- **pt_Data**

  Pages of this type are free to be used by other modules like *sm_nic*. A list of these pages can be obtained by calling *smc_get_shared_pages*(). The number of pages of type pt_Data can be set via command line argument order_grant_pages. After handed over by *smc_get_shared_pages*(), pages of this type may be completely overwritten.

The type of a page is stored inside a struct *GrantedPage_h* located at the beginning of each memory page. This struct is provided for all page types. Pages of type pt_Data may be completely overwritten once the module has finished its loading.

The master instance produces one grant-reference number for each page for each slave domain. These numbers are printed to /var/log/messages inside the master domain. The numbers have to be parsed from this log file and have to be passed as command line arguments whenever the module *sm_communicator* gets loaded as a slave.

We now show an example of how to load all modules for four domains (1 master + 3 slaves) in order to share 2 data pages among all domains.

1. Start master in domain 0:

```
0> insmod sm_communicator.ko master=1 current_domain_id=0 slave_domain_ids=1,2,3
                             order_grant_pages=1
```

The module will print out grant reference numbers during its initialization to /var/log/messages. We assume that these are the numbers 10, 11,12 and 13 for domain 1, numbers 14, 15, 16 and 17 for domain 2 and numbers 18, 19, 20 and 21 for domain 3.

2. Start slaves in domains 1, 2 and 3 with corresponding reference numbers:

```
1> insmod sm_communicator.ko current_domain_id=1 grant_refs=10,11,12,13
                             slave_domain_ids=1,2,3
2> insmod sm_communicator.ko current_domain_id=2 grant_refs=14,15,16,17
                             slave_domain_ids=1,2,3
3> insmod sm_communicator.ko current_domain_id=3 grant_refs=18,19,20,21
```

```
slave_domain_ids=1,2,3
```

Figure 25 shows this example setup.

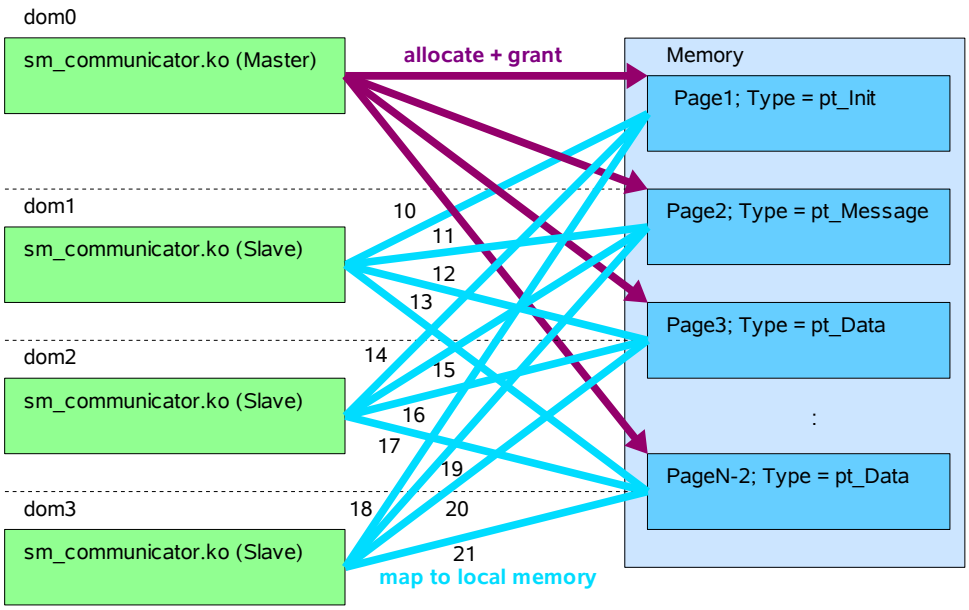

Figure 25: Allocate and grant pages

After all *sm_communicator* modules have been loaded successfully, the user may now load *sm_nic* modules in the slave domains 1,2 and 3.

4.9.1 Command Line Arguments of Module *sm_communicator*

All arguments that are known by *sm_communicator* can be categorized into one of three groups: common, master-only and slave-only according to the current value of the argument master.

Common arguments

- **master**
  =1: load module as master
  =0: load module as slave
- **current_domain_id**
  This gives the identification number of XEN domain which is loading this module (0 ≙ *Dom0*, value greater 0 ≙ *DomU*).
- **slave_domain_ids[]**
  The XEN ID numbers of other slave instances have to be given as a comma separated list.
- **debug_level**
  =0: no debug-, =1: some debug-, =2: more debug-messages
- **test_irqs**
  =1: This option activates sending of periodic sample events to all other slaves.
- **blocking**
  =1: Module-load blocks until event-channels have been established among all slaves.
  =0: Module-load returns immediately. Event-channels get established by a parallel background-thread.

Master-only arguments

- **order_grant_pages**
  The number of pages to grant to all slaves is calculated as $2^{order\ grant\ pages}$. Therefore a value of 0 means to grant one page to all slave domains. Every page is granted to all slave domains. Every grant to one domain produces one grant-reference number.
- **central_debugs**
  =1: This activates to transmit all debug messages from slaves to master *DomU*. Activating this feature slows down debug outputs but it can make debugging easier. This is because all messages arrive in one big log file in their time correct order. Even if a slave domain crashes, its debug messages can be examined here.

Slave-only arguments

- **master_domain_id**
  Provides the XEN domain identification number of domain which granted memory-pages to current domain.
- **grant_refs[]**
  Reference numbers of pages to be mapped to local memory from other domain. For the same set of memory pages, each domain gets its own set of reference numbers. When the provided scripts are used to load all modules, then the corresponding numbers can be found inside /XEN/share/GRANTS.<X>, <X> means the individual XEN domain ID.
- **grab_console**
  =1: A new console device will be registered. This console redirects all kernel oops messages via *smc_printk*(). When central_debugs has been *chosen* on the master, then even the latest *kernel oops* before a domain crashes gets logged in the master domain.

4.9.1 Functions Being Exported by *sm_communicator*

All functions listed here are described in more detail in Appendix A.1 on p.76.

- *smc_get_domains*() returns identification-numbers of all connected XEN domains
- *smc_get_shared_pages*() returns list of available shared memory-pages
- *smc_register_handler*() registers a function as event-handler for incoming events
- *smc_unregister_handler*() removes registration of given function as an event-handler
- *smc_send_event*() sends a lightweight signal to a certain domain
- *smc_printk*() immediately sends a string as debug-message to master instance
- *smc_memcpy_io*() *memcpy()* replacement suitable for shared memories

4.10 Shared Memory Network Interface (module *sm_nic*)

The virtual shared memory network interface implements a fully functional virtual network interface. This interface is always named “smn” and can be used like other network interfaces which drive physical hardware. An IP address can be assigned via the command line tool *ifconfig*.

The setup of this interface is done automatically when the research platform is started using the provided scripts as described in Chapter 5.3 .

4.10.1 Implementation of Message Passing over Shared Memory

Figure 26 shows how *sm_nic* realizes message passing via dedicated receive buffers in shared memory as described in Chapter 3.1.4 . Each of the N domains owns N-1 local *RxBuffer*s and has write access to N-1 remote *RxBuffers*. From the perspective of a domain, a remote *RxBuffer* can also be referred to as a *TxBuffer*. E.g. domain 1 has local *RxBuffer*s 1/2 and 1/3. The remote *RxBuffer*s of domain 1 are *RxBuffer*s 2/1 and 3/1. Each domain is allowed to read from its local *RxBuffer*s and to write to its remote T*xBuffer*s. The only exception is the status word which is allowed to be read and written by both associated domains in different stages of the transmission.

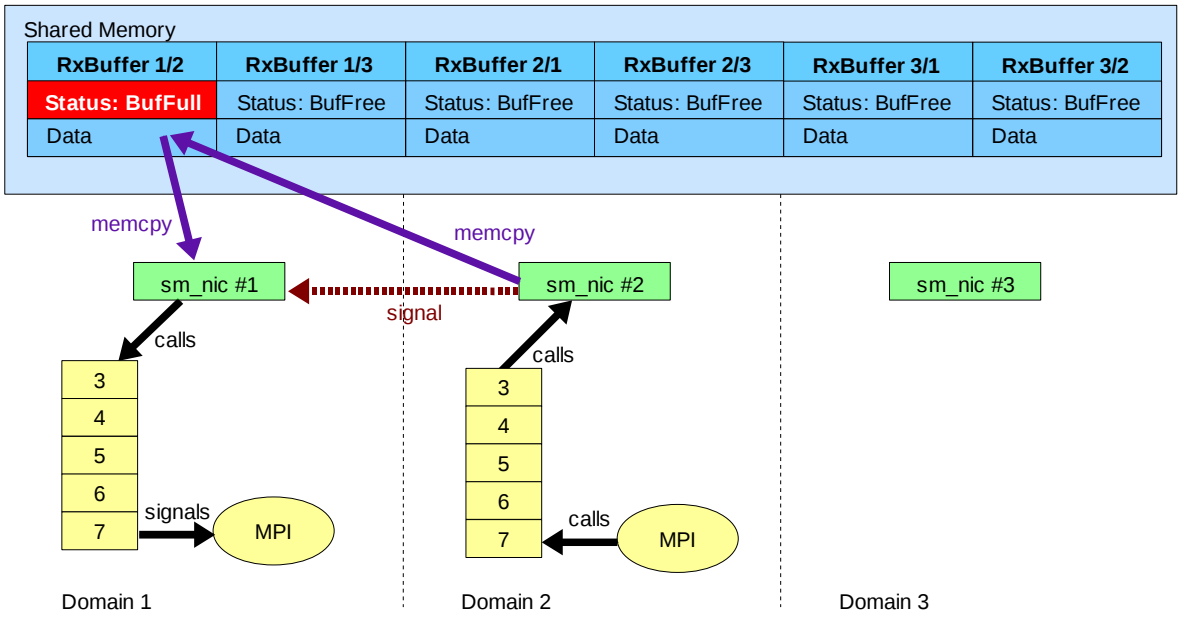

Figure 26: sm_nic: Domain 2 sends a message over Shared Memory

The numbering scheme for receive buffers is *RxBuffer* TARGET/SOURCE. Where TARGET and SOURCE mean the target and source XEN domain id of the two associated domains. On a real *NoC*, an *RxBuffer* is assumed to be local to its target and to be remote to its source domain. This means, that a read or write access to a remote buffer are established by sending packets over the network. Accesses to a local buffer can be seen as local memory accesses.

In in Figure 27, the network application (MPI) in domain 2 calls a function of the TCP/IP stack to send a data packet to domain 1. Level 3 of the TCP/IP stack calls a function of *sm_nic* #2 to transmit the provided socket buffer to its destination. This module copies all bytes from the socket buffer plus an extra header into *RxBuffer* 1/2 and sets its status to *BufFull*. It then sends a signal to interrupt *sm_nic* #1 in domain 1.

The interrupt handler in domain 1 checks all local *RxBuffer*s 1/2 and 1/3 for their status. The data of each buffer with status *BufFull* is copied into a socket buffer and handed to the local TCP/IP stack which passes it to its network application (MPI). After processing the *RxBuffer*, domain 1 resets the status of *RxBuffer* 1/2. Next time, when domain 2 wants to send a packet to domain 1 again, it will read the new status.

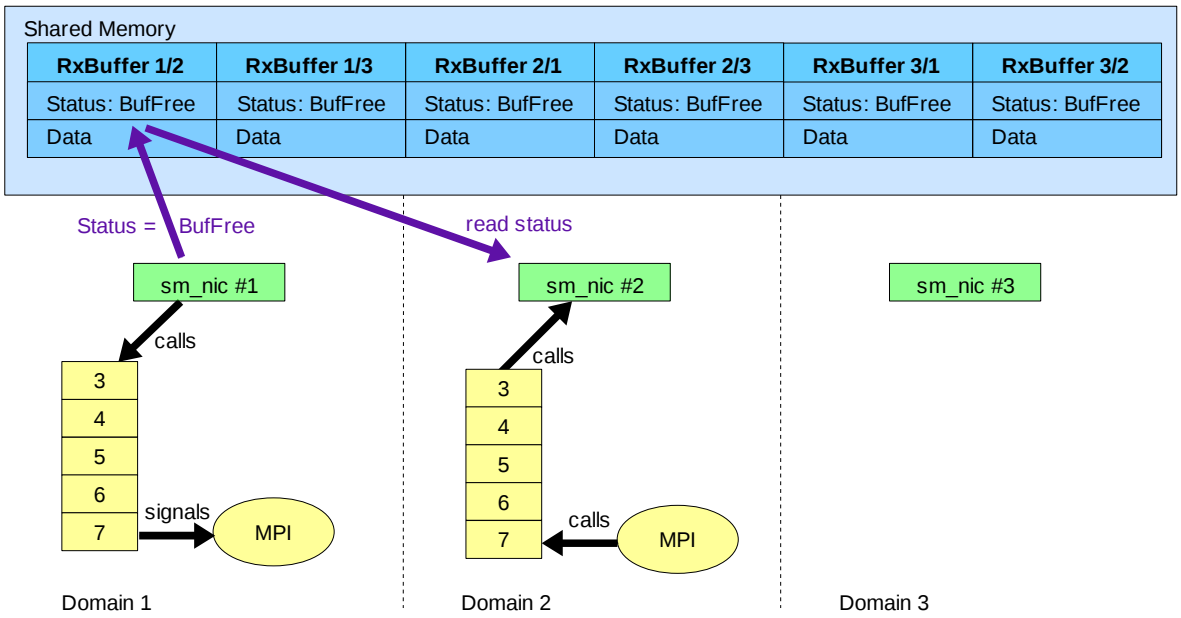

Figure 27: Domain 1 processes RxBuffer and resets Status

This bidirectional use of message buffers may not perform optimal on certain architectures. In an outlook in Chapter 6.2.4 we discuss how real unidirectional buffers can be implemented.

4.10.2 Configuration of Basic Shared Memory Network Interface (*smn*)

When the module *sm_nic.ko* gets loaded, certain configuration options can be provided as command line arguments. If the module gets loaded without the required number of arguments, a help page with current default values is printed to the file */var/log/messages*.

All arguments are case sensitive and can be passed to the module as space separated KEY=VALUE pairs. Lists of values have to be provided comma separated without any spaces.

E.g.: insmod *sm_nic.ko* debug_level=1 current_domain_id=2 master_domain_id=1

The arguments known by *sm_nic.ko* are listed below:

- **debug_level**

  Sets the level of debug outputs. All debug messages are printed to the file /var/log/messages in the local domain or in the master domain. Three different levels are available:

  • Level0: Only some messages during loading and unloading of modules

  • Level1: More messages at loading and unloading and during operations

  • Level2: Detailed messages regarding internal structures and address calculations

- **debug_packets**

  If set to 1 then the contents of every transmitted packets and their headers are printed to the log.

- **queue_size**

  The size of the output FIFO queue can be defined as the maximum allowed number of packets which can be stored for each target domain.

- **current_domain_id**

  The module cannot find out the XEN ID of the current *DomU* on its own.

- **master_domain_id**

  This argument provides the XEN ID of the domain which has loaded the *sm_communicator* module as master.

- **mtu**

  This option sets the initial maximum transfer unit (MTU) of the network interface. That is the maximum allowed packet size that this network interface accepts to send. The default MTU value for Ethernet like interfaces is 1500 bytes. If the number of shared memory pages does not allow to set the desired MTU, then the actual MTU will be as large as possible.

- **max_tx_retries**

  The network interface maintains one output FIFO queue (called TxFIFO) per other domain. A parallel running kernel thread periodically tries to send the packets from these TxFIFOs to their target domains in a loop. Whenever this loop exits without being able to send any of the remaining packets, then a counter called *Retries* is decreased. When this counter reaches zero, all packets from all queues are dropped.

  The value of the argument max_tx_retries provides the start value of this counter. If the perfomance value *tx_dropped* shows a high value, then increasing max_tx_retries may help to avoid dropping packets too early.

  Performance values are discussed in more detail in Chapter 5.1 .

- **extra_signal_every**

  Whenever the counter *Retries* (as described for option max_tx_retries) gets decreased, then a second counter called *NextExtraSignalIn* gets decreased too. If this second counter reaches zero then a signal is send to the target domain of the current packet. The intention of this extra signal is to wake up a domain in case that it has missed to serve an interrupt. The number of extra signals being sent is counted in the performance value *ExtraSignalsSent*. If this value shows a high number of extra signals, then lots of extra interrupt request are generated which may have an impact on network performance.

  Decreasing the value of extra_signal_every may decrease the number of extra generated signals. A higher value can lead to longer delays in case a domain misses interrupts more frequently.

  Performance values are discussed in more detail in Chapter 5.1 .

4.10.3 Valiant Load Balance Routing Protocol As an Extension

As described in Chapter 2.6 , interrupt requests can have a major impact on system performance. Another reason to integrate a routing protocol into a network interface can be to make better use of parallel communication hardware on a many core chip. Nodes which currently to no communicate actively can help other nodes by forwarding their packets. This can increase on die communication bandwith dynamically.

The module *sm_nic* implements a form of Valiant Load Balance routing (VLB) as a compile option. A detailed description of the original VLB algorithm is discussed in Chapter 2.5 . This routing protocol can make use of otherwise unused parallel communication bandwidth.

The implementation of VLB being described here allows to send more than one packet for the same

target immediately instead of having to wait for the target domain to react on every packet. Additionally, on a real many core CPU with level-triggered interrupts, more than one packet can be received per interrupt by target nodes.

Whenever a domain has more than one outstanding packet for the same target domain waiting in its transmit buffer then normally only the first packet is transmitted directly. This applies whenever active waiting for the receiver to process its RxBuffer would incur a major performance impact to the sender. This is true under XEN virtualization and maybe some many core chips in which the cores can only either compute or communicate. Under XEN virtualization signals are only transmitted when the originating domain issues an interrupt request on its own. This does not happen from inside an interrupt context. As the transmit function of *sm_nic* gets called from the TCP/IP stack in interrupt context, an event being generated during the function call is transmitted after the call returns. Therefore, while inside the function call, a domain cannot wait until the first packet gets processed by its target domain.

The traditional solution is to append the outstanding packets to a FIFO queue for later delivery. The network interface implemented here does provide N-1 of such *TxFIFOs*, one for each other domains. Whenever a packet cannot be send immediately, it is pushed into the corresponding queue. For simplicity, the Figures 28 to 30 show only one common *TxFIFO*.

With activated Valiant Load Balancing, if a direct *RxBuffer* is full, then some amount of outstanding packets can be send immediately as shown in the example in these Figures.

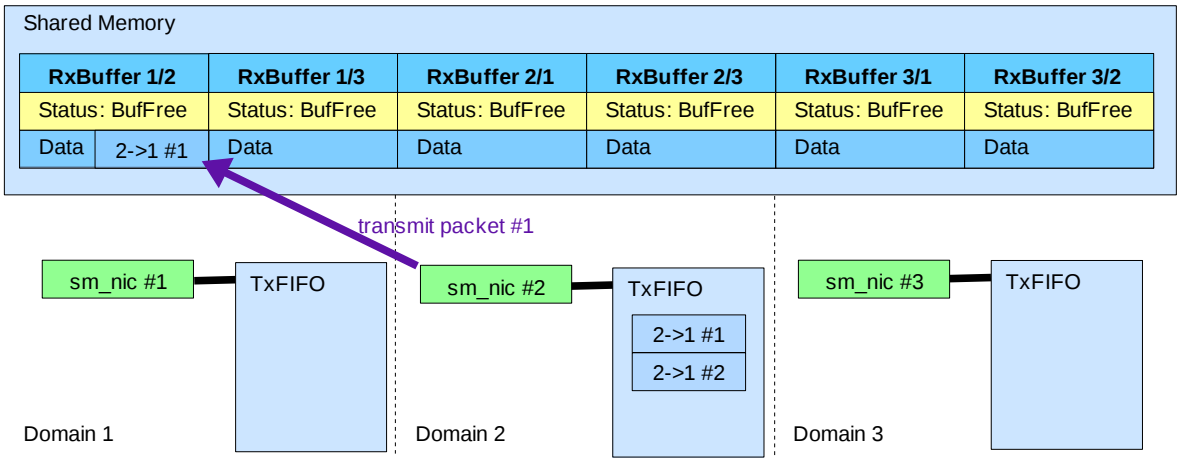

Figure 28: VLB: Step 1 - Transmit packet directly

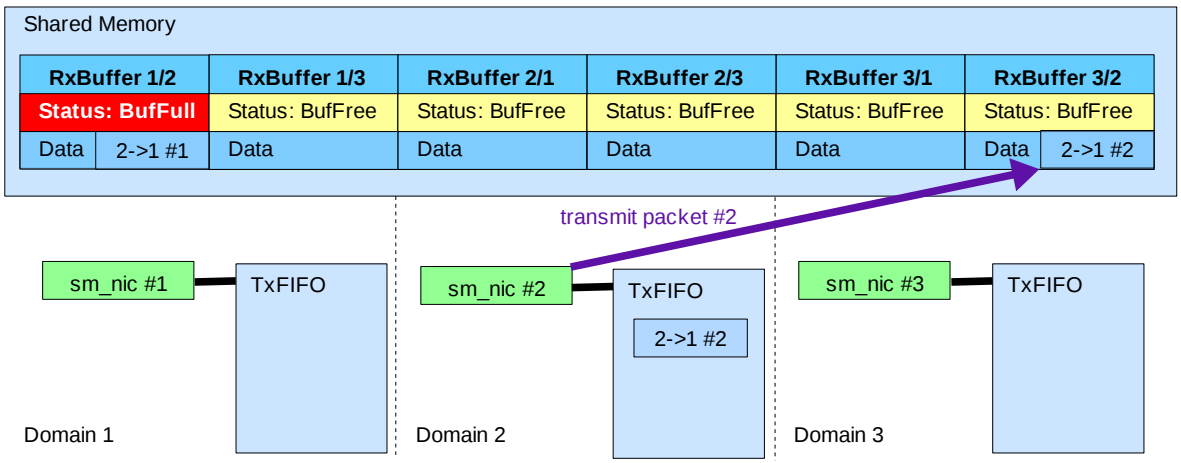

Figure 29: Step 2: Evade packet to other domain

In Figure 28, domain 2 wants to send two packets to domain 1. All *RxBuffer*s are free and no other domain wants to communicate. Domain 2 transmits packet #1 directly to domain 1.

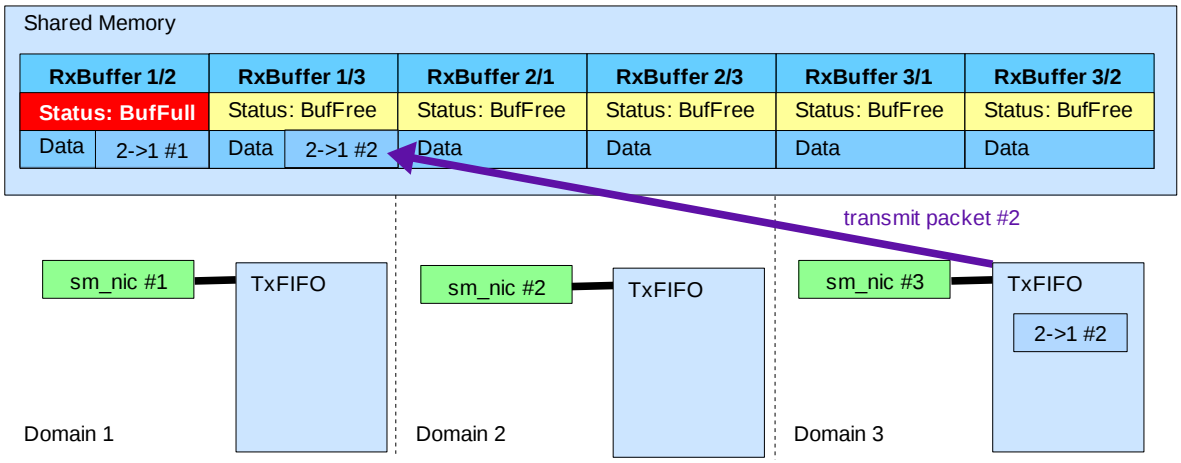

Figure 30: Step 3: Evade domain transmits to target

Now domain 3 wants to get rid of its foreign packet. The *RxBuffer* 1/3 is empty and domain 3 can send packet #2 directly to domain 1. Now all packets have been delivered to domain 1.

Domain 1 can now process both packets with just one context switch.

4.10.4 Configuration of VLB Routing Protocol

The kernel module *sm_nic_vlb.ko* provides several options to configure the VLB routing protocol. As benchmarks have shown, a misconfiguration has a major effect on network performance. The options listed below add to the basic module options as listed for *sm_nic.ko*.

- **vlb_time_to_live**

  The initial value of the time to live field of each packet defines the maximum number of evade hops that it is allowed to visit. The value is decreased on each packet being received. If the value has reached zero, then the interface will transmit the packet directly to its target domain.

- **vlb_randomize**

  In non randomized mode, if the interface has decided to send a packet indirectly, the list of evade domains is simply scanned in a linear manner from first to last for the next free TxBufffer. If randomized VLB is activated, then the list of evade nodes is permuted randomly before the scan is started.

- **vlb_max_evade_ratio**

  This argument defines a maximum allowed ratio between indirect and direct packets that a domain is allowed to generate for each target domain. An evade ratio of E means that while the TxBuffer of a destination domain is busy, up to E packets may be send indirectly over evade domains. The counter of a domain resets when the next packet can be transmitted directly to it. Therefore this argument limits the amount of indirect traffic generated by a single domain.

- **vlb_max_foreigners**

  The maximum allowed number of foreign packets that a domain accepts from each other domain can be limited with this option. If the limit is reached, then the RxBuffer over which an indirect packet was received is set to a special state. This special state forbids the associated sender

domain to pass any further indirect traffic over this link. The lock is released when the number of foreigners from this sender drops below the given limit. Thus this arguments limits incoming indirect traffic on a single domain that is generated by other domains.

- **vlb_reduce_interrupts**

  Activating this switch will suppress interrupt request generation for all indirect packets on the target domain. Everytime when an evade domain is about to send a foreign packet directly to the intended target domain, then no signal will be send to inform the target domain. The switch does not affect the generation of interrupt requests for packets being sent to evade nodes.
  The reason to not signal the target domain for indirectly transported packets is to reduce stress from indirect packets on the target domain. If e.g. domain A sends packets to domain B, then most of these packets will travel to B via evade domains. The ability of domain A to send multiple packets per send function call increases the output bandwidth of A by a factor of up to N-2, N the total number of available domains. If every indirect packet would generate a single interrupt request in B, then with XEN virtualization, the average number of packets being received per interrupt cannot be greater than 1 as the XEN hypervisor serializes and delivers all requests to B.

  As long as A continues to send packets to B, aproximately every N-1 packet is transmitted directly. Each of these direct packets is accompanied by an interrupt as with vlb_reduce_interrupts=0. During this interrupt, the direct packet and all indirect packets in the receive buffers get processed by domain B.

  Note: In order to significantly increase the performance value VLB_AveragePacketsPerIRQ past 1.0, the module argument vlb_max_evade_ratio needs to be set to a value greater than 1. This is because the module will send a signal to the target domain whenever the TxFIFOs run empty and if an amount of indirect packets has been sent to a target since the last direct packet.

- **vlb_next_hops[]**

  By defining a list of domains to which indirect traffic is allowed to be sent, the network interface allows to shape the indirect traffic according to an intended network architecture.

- **vlb_debug**

  If set to 1 then individual routing decisions of the VLB protocol get logged.

- **log_source**

  This option allows to define the ID number of a source domain to log. All packets that originate from this domain get logged in detail.

- **log_target**

  This option allows to define the ID number of a target domain to log. All packets that are targeted at this domain get logged in detail.

4.10.5 Functions Exported by *sm_nic*

All functions listed here are described in more detail in Chapter A.1

- *smn_receive_packets*() Low level function to receive one or more packets.
- *smn_transmit_buffer*() Transmits given buffer to target domain.
- *smn_send_packet*() Sends content of socket buffer to target domain.

4.11 Specification of an Unidirectional Message Buffers Implementation

Chapter 3.1.1 postulated the use of unidirectional message buffers as a performance optimization. The current implementation of *sm_nic* uses message buffers for unidirectional traffic except to the status word. By clearing the status word, a node signals that it has processed a certain buffer as described in Chapter 4.10.1 . The sender node has to read the status word from a foreign memory buffer to check if the receive buffer is free. Such remote read accesses can be very slow if generated frequently.

This can be solved easily by placing a duplicate of the StatusWord of *RxBuffer*A/B in *RxBuffer*B/A. All writes to the StatusWord can then be redirected to the remote buffer and all reads to the local buffer. This section describes a possible implementation of unidirectional message buffers.

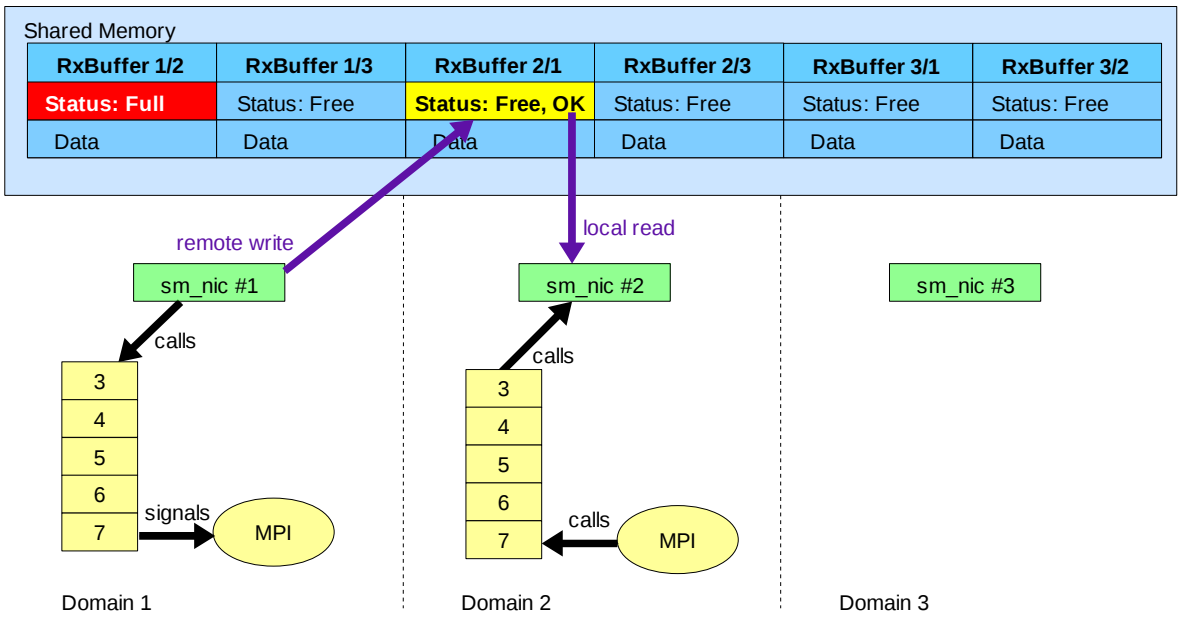

Figure 31: Unidirectional Message Acknowledgment

Figure 31 shows how to acknowledge a received packet with unidirectional buffers. An extra field in the Status field of *RxBuffer* 2/1 signals that domain 1 has processed a message from domain 2.

Two main strategies may be used to reset the OK flag:

1. Domain 2 clears the OK flag after reading it.
   This would imply to place the OK flag in its own cache line because otherwise a local write to this flag in *RxBuffer* 2/1 from domain 2 could overwrite a concurrent remote write from domain 1 to the Status field.
2. Domain 1 toggles the OK flag after processing its *RxBuffer*.

   Before setting Status=Full in *RxBuffer* 1/2, domain 2 memorizes the current state of the OK flag in *RxBuffer* 2/1. No writes to *RxBuffer* 2/1 would be required by domain 2.

4.12 Command Line User Interface

In order to start all virtual domains, to load kernel modules and for other purposes, a set of scripts has been written for this thesis. A more detailed description of all scripts can be found in the Appendix A.2

The scripts can be divided into two categories according to their location in the filesystem:

- **Scripts in /XEN/bin/**

  These scripts build the command line user interface and allow to control the whole research platform. The path /XEN/bin/ may be added to the current PATH environment variable for convenience. All scripts in this directory start with the prefix “smx_” which avoids naming conflicts with existing commands.

  - *smx_BackUpCreate.pl* Creates a backup of complete current platform..
  - *smx_BackUpRestore.pl* Restores a previous backup.
  - *smx_Benchmark.pl* *Runs MPI benchmark suite on given set of domains.*
  - *smx_Control.sh* Starts, stops or restarts all configured user domains
  - *smx_DomainIP.sh* Returns current IP address of a certain domain.
  - *smx_DomainNames.pl* Prints out list of all currently configured user domains.
  - *smx_LoadModules.pl* Loads, unloads or reloads required kernel modules.
  - *smx_TOP.pl* prints list of running processes in all domains.
  - *smx_TestSSH.pl* Tests SSH connectivity between a set of user domains.
  - *smx_UpdateIPs.sh* Updates IP addresses of all user domains.
  - *smx_UpdateXenIDs.pl* Updates current id numbers of running XEN domains.

- **Scripts in /XEN/share/bin/**

  The scripts in this folder are required by running user domains to acquire all data to establish communication over shared memory.

  - *getXenID.sh* Prints the XEN ID number of the currently running user domain.
  - *load_master.sh* Loads or unloads master instance of *sm_communicator*.
  - *load_slave.sh* Loads or unloads *sm_communicator* and *sm_nic*
  - *updateGrantRefs.pl* Collects grant references of memory pages to be shared.

Chapter 5
Performance Evaluation

After describing details of implementation, this Chapter compares the performance of the traditional virtual XEN network card (further called *xnc*) vs the new shared memory network interface (further referred to as *smn*). We will show that the current implementation of our virtual network interface basically provides the same performance as the XEN provided interface.
This chapter does not talk about VLB routing as the fine tuning of this extension is out of the scope of this thesis.

5.1 Benchmarking

For benchmarking all available module arguments of *sm_nic.ko* and *sm_nic*_vlb.ko being described in Chapters 4.10.2 and 4.10.4 can be used to parameterize the configuration of the research platform.

Additional parameterizable configuration options are as follows:

- **The number of slave domains**

  This is the number of domains which provide a *sm_nic* network interface. For configuration which do not make use of the VLB routing protocol extension, the instantiation of unused slave domains will not improve network performance. In a VLB enabled configuration, unused domains can help all other domains by forwarding their indirect traffic.

- **The number of domains on which the benchmark runs**

  To vary the amount of domains being actively used to compute and communicate during a benchmark is the natural way to compare the scalability of different configurations of message passing hardware.

- **The Maximum Transfer Unit (MTU)**

  Most Ethernet like network interfaces provide a MTU value of 1500 bytes, thus allowing to send up to 1500 bytes per individual packet. By decreasing the MTU, the platform allows to simulate the effect of small memory buffers. Smaller MTU values force the TCP/IP stack to create more packets for the same amount of data, thus increasing the protocol overhead. On a real many core CPU, the amount of shared memory available for message passing may be limited so that it might not be able to provide a 1500 byte MTU.

5.1.1 MPI Library MPICH2

The benchmarks described below have been run on version 1.0.8 of MPICH2. Earlier tests with MPICH1 showed a bug in the MPICH library. For MTU sizes of less than 1500 bytes, still packets of 1500 bytes were handed to the network interface for transmit. Also some system crashes appeared with the IMB suite. After upgrading to MPICH2, the MTU bug disappeared and the benchmarks ran more stable.

The guest OS images in the folder /XEN/images/ already provide v1.0.8 of MPICH2 preinstalled.

5.2 Intel MPI Benchmark suite (IMB)

One part of the workloads used for benchmarking is provided by running a well known set of MPI benchmarks called Intel® MPI Benchmarks, formerly known as Pallas MPI Benchmarks, on varying sets of domains.

According to its documentation, the individual tests included in the IMB are categorized into three benchmark classifications:

- **Single Transfer**

  This classification contains benchmarks in which signals are send between two processes.

- **Parallel Transfer**

  The benchmarks under this classification simulate computation and communication on all configured nodes.

- **Collective Benchmarks**

  Benchmarks in this classification do not perform computations but only communication. These tests are mainly targeted at the underlying message passing performance of the system.

Chapters 5.2.1 to 5.2.3 list the different benchmarks which have been run on this research platform for different message passing configurations. The individual benchmarks are described in detail in the users guide document of the IMB in the folder */XEN/share/MPI/IntelMPI_Benchmark32/doc/*. A current version of this benchmark suite can be downloaded from the Intel website.

5.2.1 Single Transfer Benchmarks

- PingPing One process constantly sends packets to another process.
- PingPong Like PingPing but each packet produces an answer

5.2.2 Parallel Transfer Benchmarks

- Sendrecv All Processes form a periodic communication chain. Messages are passed from the left to the right.

5.2.3 Collective Benchmarks

- Bcast One process periodically sends a message to all other processes.
- Allgather Every process provides data and receives the gathered data of all processes.
- Allgatherv Variable version of Allgather which may incur more overhead.
- Gather A root process receives data from all other processes.
- Gatherv A root process receives data from a varying number of other processes.
- Scatter A root process sends data to all other processes.
- Scatterv A root process sends data to a varying number of other processes.
- Alltoall Every process provides data and receives the sum of data from all processes.
- Alltoallv Like Alltoall but for a varying number of processes.
- Reduce Reduces a vector of float items to their sum.

5.3 Manual Benchmark Run

When the *smn* network interface is successfully instantiated in all slave domains, then any kind of network load can be put on it like pinging other domains, copying to NFS shares or special network benchmark loads.

The preinstalled IMB benchmark suite which has been discussed in Chapter 5.2 can be issued to run on a given subset of all currently running domains via the script */XEN/bin/smx_Benchmark.pl*. This script loads all required modules on all given domains and then runs the complete IMB suite once. The output of the whole benchmark run is logged into a file inside the folder */XEN/share/Benchmarks/*.

Additional to the timing and throughput measures of individual benchmarks, the *smn_nic.ko* and *smn_nic_vlb.ko* modules maintain a set of performance measures. These measures are further referred to as "*Perfomance Values*" and described in more detail below. The performance values of a benchmark run are automatically appended to each log file when using the provided script.

5.3.1 Performance Values of Module *sm_nic.ko*

Performance values will be printed to the log file /var/log/messages by the module *sm_nic.ko* or *sm_nic_vlb.ko* whenever the corresponding network interface is shut down.

In order to shutdown and restart the interface in all slave domains, two scripts have to be called:

1. /XEN/bin/smx_RestartAllInterfaces.pl RESTART
2. /XEN/bin/smx_ExtractPerformanceValues.pl <DOMAIN>

Here <DOMAIN> means the name or IP address of a slave domain from which the latest performance values should be extracted.

Figure 32 lists all performance values that are tracked by *sm_nic.ko* and *sm_nic_vlb.ko.*

5.4 Benchmark Automation

The script */XEN/bin/smx_RunBenchmarks.pl* provides a way to automatically run the same benchmark on different configurations of the *smn* network interface. The different configurations have to be defined in the source code of this PERL script.

The automation of benchmarking different configurations, requires to run a set of nested for loops. Each loop runs over one configuration parameter. The provided script already implements a configurable multi level loop iterator. A whole set of benchmark runs just has to be configured and given to a run function. The run function then iterates over all provided parameters and issues a benchmark run for each different configuration. In most cases, the complete design space of all parameters would take too long to be simulated practically. For such cases, the run function allows to define a series of configuration sets. Each set may vary only one parameter.

A detailed description of how to define configuration sets can be found in Appendix A.2.21 .

Performance Value	Description
AllocatedPackets_Max	highest number of packets managed at once
AveragePacketSize	average size of packets sent
MinimumPacketSize	size of smallest packets sent
MaximumPacketSize	size of largest packet sent
MaximumTxRetries	maximum occured retries before at least one packet could be send
ExtraSignalsSent	number of signals sent to wake up other domains after a timeout
QueuedPackets_Maximum	highest number of packets queued in all TxFIFOs
ImmediatePacketsTX	number of packets sent immediately without being queued
rx_packets	total number of packets received since last module reset
tx_packets	total number of packets sent since last module reset
rx_bytes	total number of bytes received since last module reset
tx_bytes	total number of bytes sent since last module reset
rx_errors	number of receive errors occured since last module reset
tx_errors	number of send errors occured since last module reset
rx_dropped	number of received packets which have been dropped
tx_dropped	number of packets to send which have been dropped instead
VLB_LongestRoute	largest hop count of all received packets
VLB_RX_Indirect	number of indirectly received packets
VLB_RX_Direct	number of directly received packets
VLB_MaxSeenForeigners	maximum number of foreign packets seen since last module reset
VLB_AveragePacketsPerIRQ	average amount of packets received on each interrupt
VLB_TX_Indirect	number of indirectly sent packets
VLB_TX_Direct	number of directly sent packets
VLB_TX_LimitedBySender	number of packets forced as direct packets because vlb_max_evade_ratio was reached
VLB_TX_LimitedByReceiver	number of packets forced as direct packets because vlb_max_foreigners was reached

Figure 32: List of Performance Values

5.5 Hardware Used for Benchmarking

The benchmarks were run on a 16 core Caneland server class machine named S7000FC4UR. The 16 cores are provided by four Xeon X7350 MP CPUs clocked at 2.93 GHz. Each CPU basically consists of two dual core chips of the Core2 architecture. Each chip has 4MB L2 cache. The machine was equipped with 16GB of DDR3 RAM running at 266 MHz FSB frequency quad pumped. Specifications of this machine can be found on the Intel website.

The operating system used was the open source Linux distribution openSuSE v11.0 from Novell with physical address extension (PAE) and integrated XEN v3.2. The required software packages are described in Chapter 4.2.5 .

5.6 Performance Results

This chapter shows an excerpt from the database of all measures. The database consists of several log files, one for each benchmark run, in the folder */XEN/share/Benchmarks/Evaluations/*. A PERL script in the same folder scans through all log files. For each output graph, an individual filter function has been implemented inside this script. Each filter function extracts a certain subset of data from each log file and creates a comma separated value (CSV) file. The graphs below have been created from these CSV files.

The graphs compare the performance results of benchmarks that have been run via the native XEN network interface (marked as *xnc*) and via the *sm_nic* that has been written for this thesis (marked as *smn*). Most graphs show real data points and cubical interpolated curves. Some graphs contain so much data points that the curve got hidden. For these graphs, the display of data-points and interpolation has been disabled.

5.6.1 Single Transfer Benchmarks

In each benchmarks under this classification two threads communicate with each other. Both tests do not scale with the total number of domains used.

The *PingPing* benchmark constantly sends messages in one direction from one thread to another. The comparison of three different packet sizes shows three horizontal bars in the graphs of Figure 33. Except for a maximum transfer unit (MTU) of 350 bytes, the graphs show the same bandwidths and latencies for *smn* and *xnc*. The right graph of Figure 33 shows an interesting effect for MTU=1500 where for both benchmarks, the latency drops from 100µs for MTU=375 down to 85µs. The reason may be that the TCP/IP stack is optimized for MTU sizes of 1500 bytes.

In addition to the unidirectional test, the *PingPong* benchmark sends packets in both directions. Therefore the graphs in Figure 34 show basically the same scenario. For messages of size 1kB, the MTU increase from 750 to 1500 directly shows has effect on the round trip latency. Instead of sending two packets to transport 1kB, a MTU of 1500 bytes allows to send the message in one packet. Thus giving half the turnaround latency than for 750 bytes. The left graph also shows that the bandwidth scales nearly linear over the message size from 230 kB/sec for messages of 16 bytes up to 102 MB/sec for messages of 16 kB each.

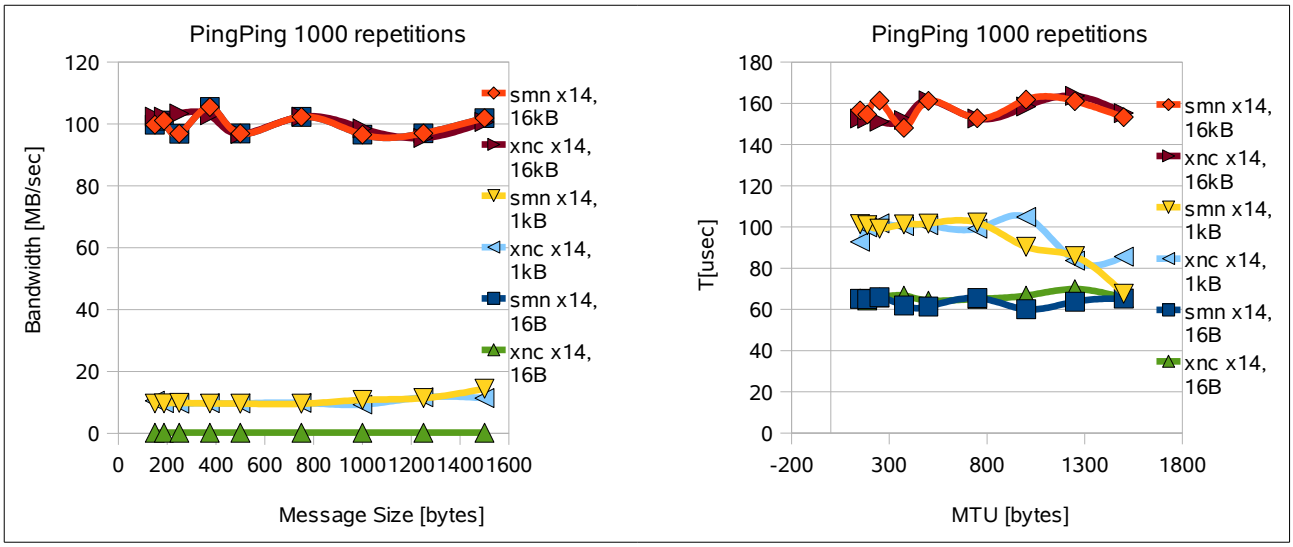

Figure 33: Benchmark Results for PingPing

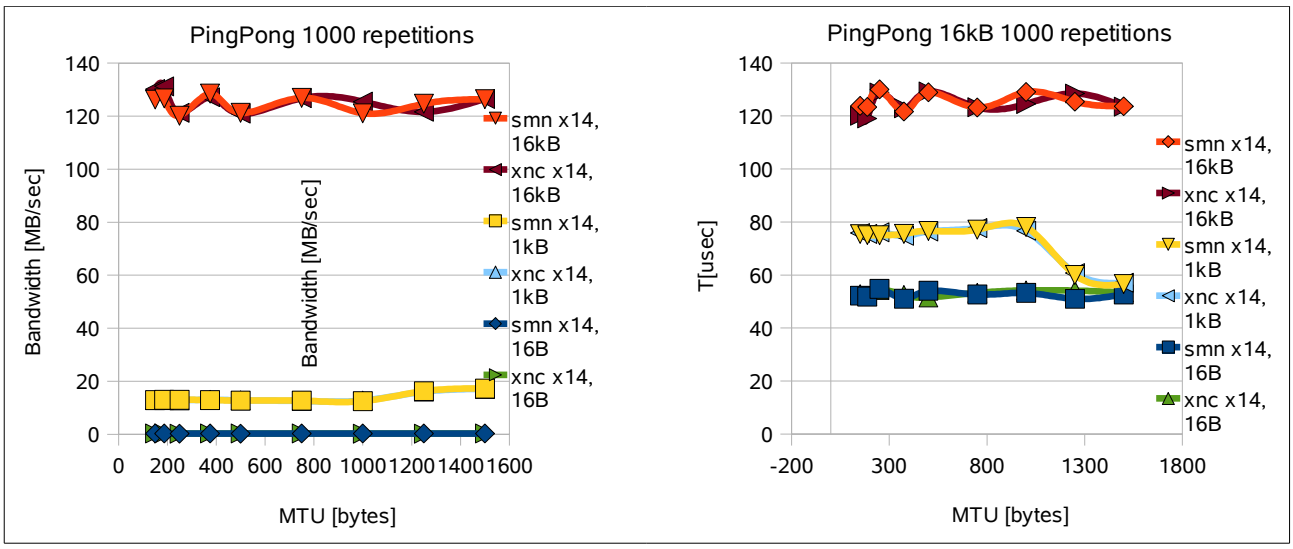

Figure 34: Benchmark Results for PingPong

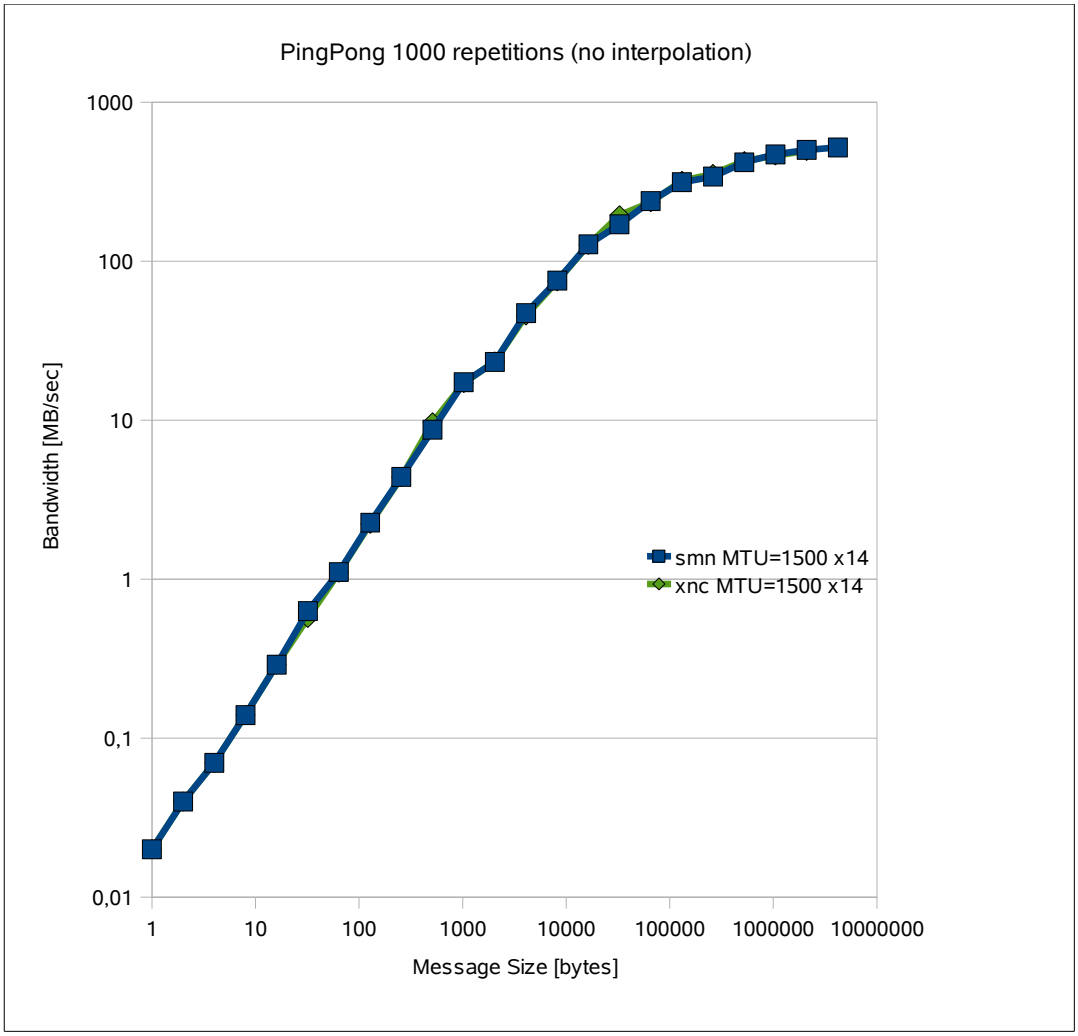

Figure 35: Scalability of PingPong Benchmark

Figure 35 shows the scalability of both network interfaces over the size of each message. The saturation is reached at approximately 100kB. The limit may be a result of the window size of the

TCP/IP stack. If an application constantly sends out messages, then the TCP/IP stack will buffer a certain amount of packets which are not yet acknowledged by the receiver. As the packet size is a constant value, only a constant amount of data can be buffered.

5.6.2 Parallel Transfer Benchmarks

The benchmark in this classification simulates global activity on the network by spawning several processes and assemble them in a periodic communication chain. Therefore the graphs show throughput and latency under heavy network load.

The left graph of Figure 36 shows the same hard bandwidth limit for message sizes greater than 100kB. The values are nearly the same regardless of the MTU value. Therefore this graph shows how effective the fragmentation engine in the TCP/IP of current Linux kernels is implemented. The graph on the right directly shows the near constant ratio between bandwidth and MTU size in this benchmark.

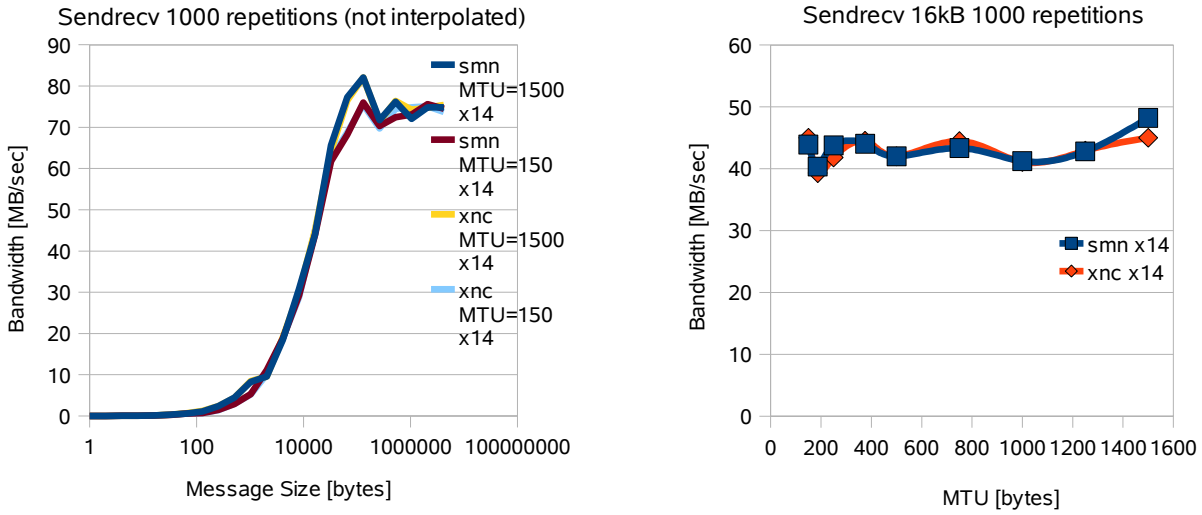

Figure 36: Benchmark Results for Sendrecv

5.6.3 Collective Benchmarks

All benchmarks under this classification transmit messages of varying length. Therefore we would expect not to find a preferred MTU size as for the *PingPing* and *PingPong* tests.

The results of the Bcast benchmark are shown in Figure 37. During this test messages are constantly being sent from one node to all other nodes. One would expect that the XEN network implementation can benefit from its zero copy strategy. For a broadcast, the same memory page can be granted to multiple domains at the same time. Such an implementation should provide broadcasts which scale over the number of domains. But as the graph on the left shows, both networks provide no scalability at all. The *xnc* and *smn* network do scale well over the message size as shown in the graph on the right.

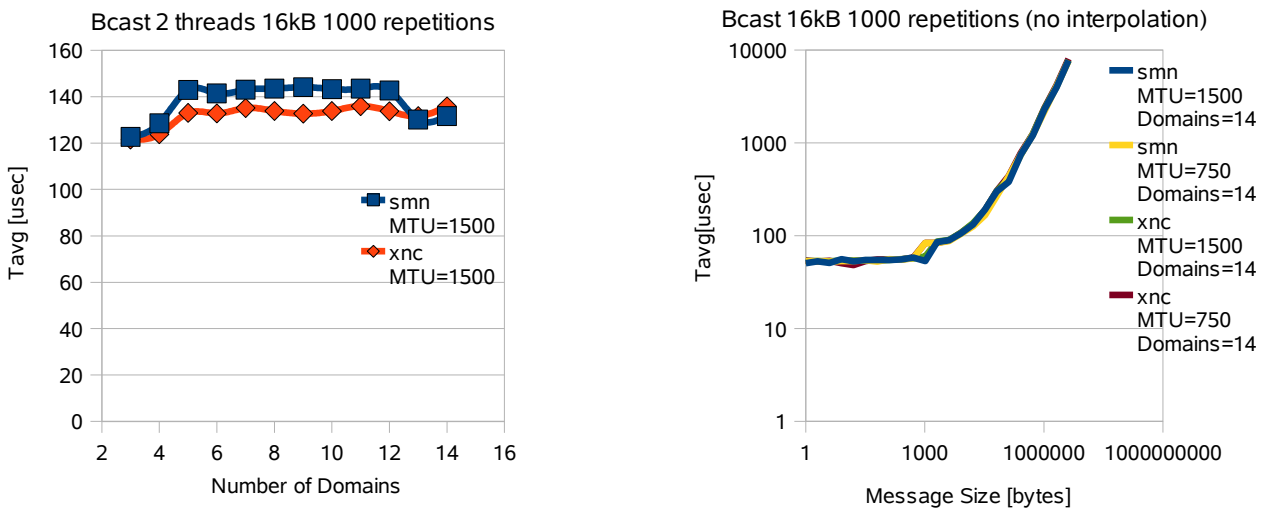

Figure 37: Benchmark Results from Sendrecv

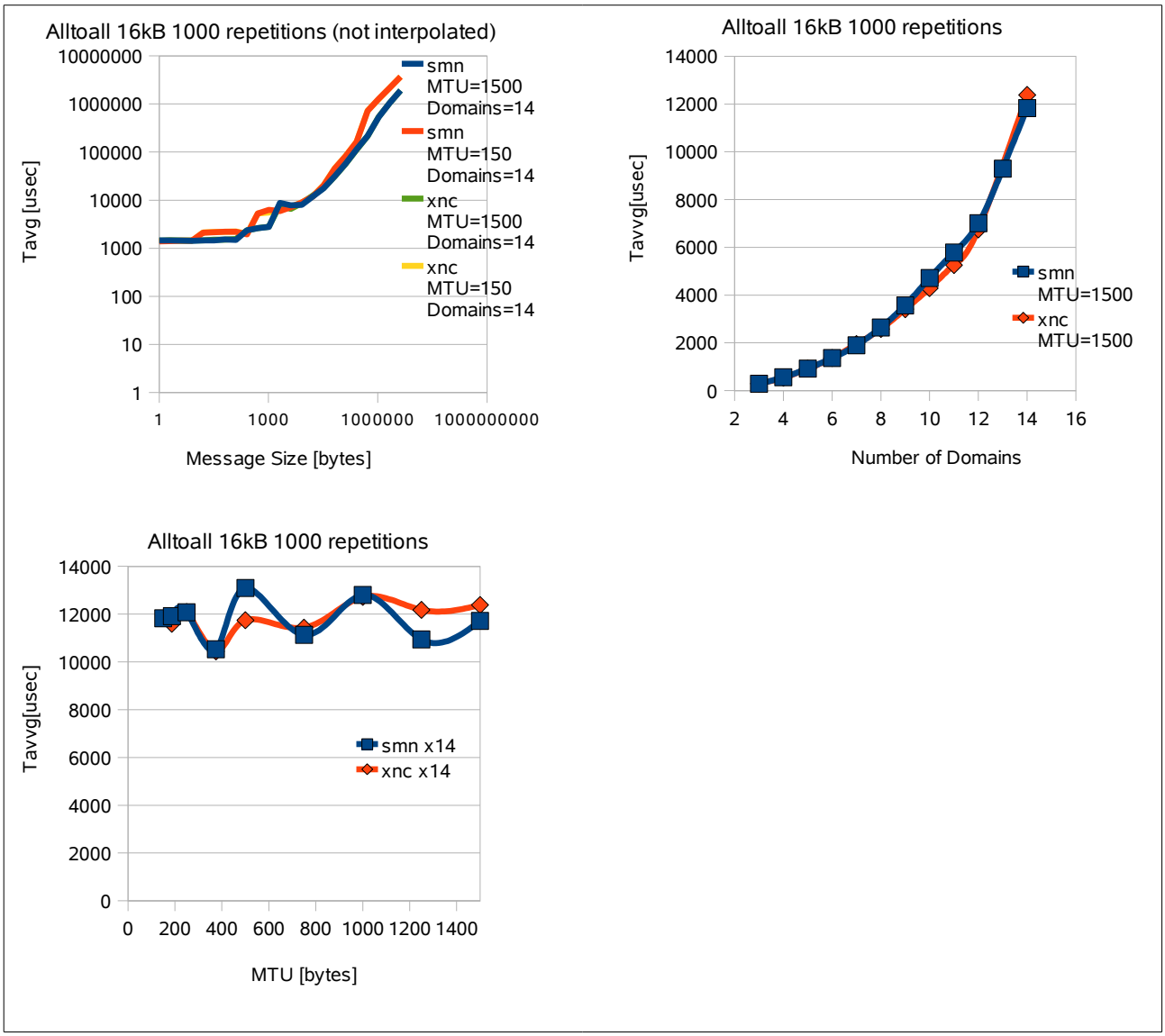

Figure 38: Benchmark Results from Alltoall

The *Alltoall* benchmark simulates a scenario where all domains communicate all the time. This test puts the most stress in terms of interrupt requests on the message passing interface. The two upper graphs of Figure 35 show a linear increase of latency over the message size while scaling by the power of two over the number of participating domains. The *smn* network provides lower latencies in some configurations. The reason for this may be the direct signaling technique between domains. Signals which are generated by the XEN network interface do not travel directly to their destination domain. Instead the interrupt request first is delivered to the corresponding backend driver in dom0. The *xnc* network instead benefits from its zero-copy strategy. While the sm_nic always copies data into remote memory buffers, its XEN pendant simply hands the affected memory page to the receiver domain. The lower left graph of Figure 38 shows that both network interfaces have their own optimal MTU sizes. The overall trend shows no advantage of one of the two implementations.

Figure 39 shows the results from the *Reduce* benchmark scenario for different MTU sizes and numbers of domains. The benchmark simulates the inverse operation of the *Bcast* scenario. Messages are send from a group of domains to one so called "*root*" domain. This test puts the individual *root* domain under heavy stress regarding incoming interrupt requests. For a message size of 16kB, the MTU is critical for the latency of each message. Therefore, the smallest MTU sizes of 150 and 187 bytes greatly increase the latency. When benchmarked against the number of sender domains, the latency scales rather non linear. Some tests show a lower latency for the XEN network but a real trend cannot be seen in the graph on the right.

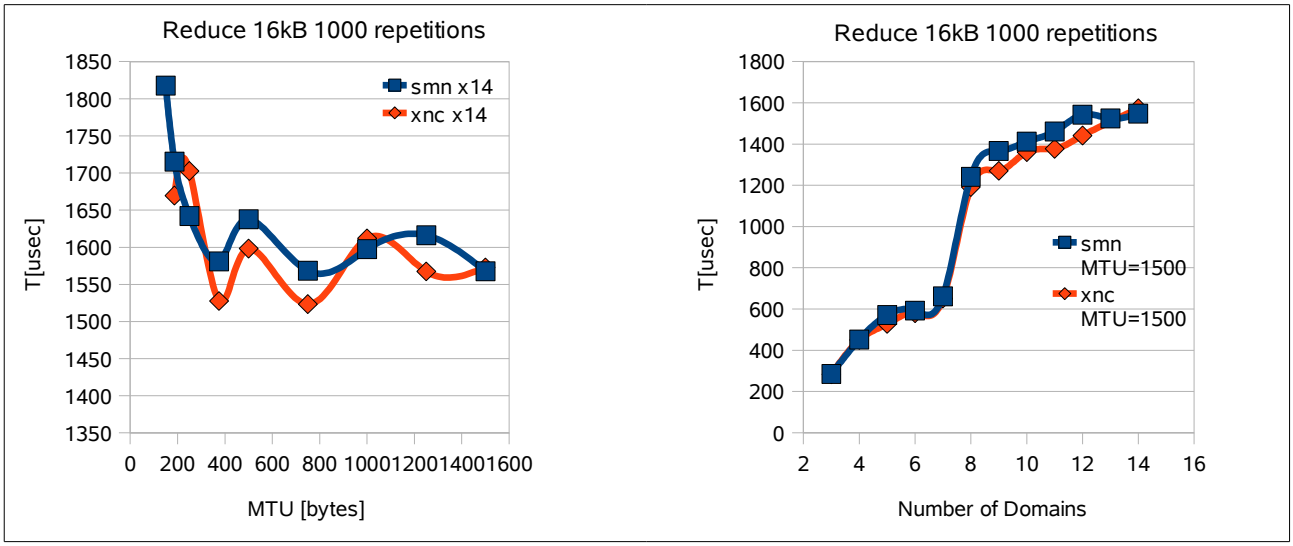

Figure 39: Benchmark Results from Reduce

5.7 Summary of Performance Evaluation

115 benchmarks have been run on the research platform on a 16 core machine as described in Chapter 5.5 . These benchmarks have shown that the new network interface, based on emulated hardware message passing, can provide similar network performance as the original XEN network interface. It can be concluded that the implementation being documented in this thesis has no major design flaws. All graphs show nearly the same asymptotic behavior for both types of network interface.

Chapter 6
Conclusion and Outlook

After describing functionality, implementation and benchmarking of the research platform, this Chapter gives an overall summary of the goals of this approach that were described in Chapter 1.2 and checks if they were met.

6.1 Conclusion

During this thesis, we have presented a software only research vehicle that enables research in the field of hardware based message passing interfaces on todays PC hardware. The research platform allows to retrieve results much faster than with the traditional approaches of hardware simulation and emulation. One disadvantage of the approach is that it is basically limited in the number of available cores and the different signaling mechanism of the XEN virtualization. Though it is possible to start more virtual domains than physical processing cores in the machine, no real parallel computations can take place. This would decrease the correlation of performance results to a real many core CPU further.
The signaling mechanism provided by XEN mainly differs from hardware based signals in two ways:

- Signals under XEN are edge triggered.
  The XEN hypervisor puts all generated signals into a FIFO queue and guarantees their delivery in order. On a real CPU, incoming interrupts are level triggered, while the interrupt is being serviced, other interrupt requests with the same number are ignored. This behavior leads to packet loss and requires special treatment by driver software.
- XEN delivers signals only outside of interrupt context.
  The transmit function of a network interface under Linux gets called in a interrupt context. In order to send the given packet to its destination, a signal has to be sent to the target domain to inform it about the new data. This signal is not transported by the XEN hypervisor before the next interrupt in the sender domain takes place. XEN delays these signals in order to minimize overall context switches. For a network interface, this behavior increases latencies for individual packets. This implies that the transmit function cannot busy wait for the receiving domain even if more than one packet is waiting in the TxFIFO.

As the performance evaluation has proven, the implementation shows basically the same performance under different network loads as the one provided by the XEN virtualization. Because the new network interface makes use of an emulated hardware message passing interface, it can be configured to behave like a network on chip of a real many core CPU as defined in chapter 2.1 . The prove of applicability currently lacks the availability of such a CPU. Therefore it has to be given by further research. This thesis is an enabler for this kind of research.

The vast design space of all configuration options of the VLB routing protocol cannot be evaluated within this thesis. With activated VLB routing, small transmissions see a performance improvement while many benchmark runs simply do not complete. This is because the routing protocol quickly generates more packets being in transit than the current TCP/IP stack implementation can handle. This incurs retransmissions of packets and therefore big performance drops and deadlock situations. Therefore VLB has a similar problem as with todays high speed fiber optic network interfaces which can have hundreds of packets in transit within an optical link. The TCP/IP stacks identifies these packets as being lost and starts a retransmission. The evaluation of VLB will require an evaluation of the current TCP/IP stack implementation inside the Linux kernel too.

6.2 Outlook

This section lists some ideas for extensions and improvements of the research platform presented in this thesis.

6.2.1 Non Uniform Routing Costs

The current implementation of *sm_nic* can send packets to every other node at the same latency. These uniform routing costs may not reflect an intended *NoC* implementation. Therefore, non uniform routing costs could add latency penalties for each packet being sent to distant nodes. A static function could take XEN ID numbers of source and destination domain and calculate a millisecond penalty. Instead of directly sending packets, *smn_send_packet*() would add every packet to TX queue for later delivery via *work_send_packets()*. The function *work_send_packets()* would then calculate the routing penalty for each packet and issue a *ndelay(Penalty)* before trying to send it. In a more precise model, one domain could simulate the current state of the network and calculate packet latencies on the fly.

6.2.2 Simulation of Broken Interconnects Between Certain Nodes

Future many core chips may be imperfect in terms that some network links or processing cores might be broken. These defects can occur during manufacturing or even during operation. Such scenarios are already known today when it comes to hard drive storage in RAID systems. These systems already anticipate the failure of individual drives by providing redundant capacities.

To simulate broken interconnects, a module argument can be added to *sm_nic* which takes a list of XEN ID pairs for which *smn_transmit_buffer*() will always assume that the target receive buffer is busy. For non VLB operation, outgoing packets for this broken domain would accumulate in the corresponding TxFIFO until its size reaches queue_size packets. With activated VLB routing, all packets would be automatically delivered indirectly via evade nodes. A performance evaluation of such a defect could be very interesting.

6.2.3 VLB Optimization: Evade Packets To Free Slot Neighbors

With activated VLB routing and whenever a packet cannot be delivered directly, the packet may be evaded via a third domain. Currently, one of two strategies may be applied:

- Evade to next free node

  Remote receive buffers of all other evade domains are checked in order of their XEN ID number. The packet is evaded to the first node which receive buffer is empty.

- Evade to random node

  The list of evade domains is first permuted randomly before the current packet is evaded to the next free node.

If the intended *NoC* implementation allows every node to read receive buffers of all other nodes, then a sending node could check which evade node currently has an empty receive buffer at the packets destination node. The packet is then evaded to this node if possible.

In order to be able to read from receive buffers of other nodes, the shared memory configuration has to be calculated from the perspective of every other domain. The current implementation calculates only the memory addresses of remote and local receive buffers which belong to the current domain. As the whole shared memory configuration is stored in a global variable of type SM_Configuration_t, it is possible to calculate an array of configurations for each domain.

6.2.4 Unidirectional Message Buffers

The principle of local receive buffers with faster access than remote buffers does apply for the postulated type of future many core CPUs and in some way for todays multi core architectures. In a todays SMP machine with 16 cores, different cache levels do exist. Such a system is typically divided into several CPU sockets. Each socket carries one CPU that contains up to two identical dies. Each of the dies provide two or more processing cores. Each core has its own local cache and each die provides its own cache. From the perspective of one core, accesses to its L1 cache are faster than those to the L2 cache and much faster than accesses to data that is currently stored in L2 or L3 caches of other sockets.

Therefore the unidirectional use of message buffers that is described in Chapter 4.11 can provide a speedup even on todays multi core CPUs.

6.2.5 Port Network Interface Onto Many Core CPU

The sm_nic.ko kernel module has been designed with a small footprint. The implementation details of shared memory setup and event signaling are capsulated inside the sm_communicator.ko module.

Therefore to port the *smn* network to a real many core CPU, the following requirements have to be fulfilled:

- The many core CPU fits into the targeted class as described in Chapter 3.1 .
- A new implementation of sm_communicator.ko provides all functions being exported by this module as described in Appendix A.1.7 to A.1.13 for the individual architecture.

  The current implementation of this module has to deal with a variable list of memory pages for a variable number of domains. An implementation for a fixed size many core CPU would allow to define one constant memory range as being shared. Therefore such an implementation should be simpler than the current one working with XEN virtualization.
- Some compile settings need to be changed inside sm_nic.h.

  Mainly only the compile switch "*RUNNING_ON_XEN*" has to be commented out.

6.3 Wrap Up

The implementation of the enhancements described above can build a more precise model of emulated hardware message passing and increase the correlation of benchmark results from this research platform to real many core CPUs.

References

Moo65: Gordon Moore, Cramming more components onto integrated circuits, Electronics, Volume 38, Number 8,, 1965

Pol99: Fred Pollack, Keynote to MICRO-32 International Symposium on Microarchitecture, Intel Corp., 1999

Bor07: Shekar Borkar, Thousand Core Chips—A Technology Perspective, Intel Corp., 2007

Sten90: Per Stenström, A Survey of Cache Coherence Schemes for Multiprocessors, COMPUTER, Volume: 23, Issue: 6, 1990

VHR+07: Sriram Vangali, Jason Howard, Gregory Ruhi, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskotel, Nitin Borkar, An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS, Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International, 2007

Iko08: IKOS Systems, Inc., IKOS' VLE Emulator for verifying multiple operating systems for MIPS Technologies 64-Bit processor Core, IKOS Systems, Inc., 2008

Val01: Michal Valient, 3D Engines in games - Introduction, , 2001

gpg09: Webpage of General Purpose GPU association, http://www.gpgpu.org/, 2009

RSV87: U. Ramachandran, M. Solomon, M. Vernon, HARDWARE SUPPORT FOR INTERPROCESS COMMUNICATION, ACM New York, NY, USA, 1987

MPI95: Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, Message Passing Interface Forum, 1995

Dig80: Digital Equipment Corp, Intel Corporation, Xerox Corporation, The ethernet: a local area network: data link layer and physical layer specifications, ACM SIGCOMM Computer Communication Review, 1980

Com98: Douglas Comer, Computernetzwerke und Internets, Prentice Hall, 1998

Che07: Chelsio Communications Corp., 10GbE Storage Accelerator, IBM Corp., 2007

CB06: Simon Crosby, David Brown, The Virtualization Reality, Association for Computing Machinery, New York, NY, USA, 2006

AA06: Keith Adams, Ole Agesen, A comparison of software and hardware techniques for x86 virtualization, Association for Computing Machinery, New York, NY, USA, 2006

Int07c: Intel Corp., Intel 64 and IA-32 Architectures, Intel Corp., 2007

XEN?: XenSource, Inc., Palo Alto, California, Xen—the Art of Virtualization - A XenSource White Paper, XenSource, Inc., Palo Alto, California, ?

VB81: L.G. Valiant, G.J. Brebner, Universal schemes for parallel communication, Edinburgh University, 1981

Int98: Intel Corp., Interrupt Latency in 80386EX Based System, Intel Corp., 1998

Dan00: Kevin Dankwardt, Comparing real-time Linux alternatives, Ziff Davis Enterprise Holdings Inc., 2000

Cam05a: XEN developers group, XEN Users' Manual v3.0, University of Cambridge, 2005

Kra07: Andres Krapf, XEN Memory Management (Intel IA-32), INRIA Sophia Antipolis - Méditerranée Research Centre, 2007

Cam05b: XEN Team, XEN Interface Manual - XEN v3.0 x86, University of Cambridge, UK, 2005

Tor08: Linus Torvalds, Linux Kernel Sources 2.6.25.16-0.1-default (SL110_BRANCH), , 2008

Lin?: Linus Torvalds, Linux Kernel Coding Style, ,

CRK05: Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, Linux Device Drivers, 3rd Edition, O'Reilly Media, Inc., 2005

Wik09: Symmetric multiprocessing, Wikipedia, 2009

Appendix

A.1 Function Index

This section describes all functions being exported by the three new kernel modules. The module a function belongs to can be identified by its prefix:

- rt_ functions belong to resource_tracer.ko
- smc_ functions belong to *sm_communicator.ko*
- smn_ functions belong to *sm_nic.ko*

A.1.1 rt_create_owner()

```
Prototype: unsigned int rt_create_owner(char* Owner)
@Owner:    Unique name identifying registrator.
           Each module using resource_tracker.ko is seen as one owner.
@return:   Each Owner-string gets its own unique identification number
           to be used for all further function-calls.
```

This function registers a given string as new memory-owner and returns the corresponding owner-ID.

A.1.2 rt_pages_allocate()

```
Prototype: unsigned long rt_pages_allocate(unsigned int Owner, unsigned int Order,
                                           unsigned char* Purpose)
@Owner:   Owner-id as returned by rt_create_owner() earlier.
@Order:   Power of two of number of pages to allocate.
@Purpose: Used in debug messages to identify individual memory allocations.
@return:  Start address of first allocated page.
```

This function allocates a given number of memory pages. It behaves like pages_allocate(). All pages are allocated consecutively in order.

A.1.3 rt_kmalloc()

```
Prototype: unsigned long rt_kmalloc(unsigned int Owner, size_t Size, int Flags, void**
                                    Pointer, unsigned char* Purpose)
@Owner:   Owner-id as returned by rt_create_owner() earlier.
@Size:    Number of bytes to allocate.
@Flags:   kmalloc flags like GFP_KERNEL, GFP_ATOMIC, GFP_USER, …
@Pointer: !=NULL: Variable will be loaded with start address of allocated memory
                  (NULL in case of error)
                  Pointer will be stored and reset to NULL automatically by
                  rt_free_resources().
                  This allows to allocate memory in static function variables without
                  having to worry about deallocation or even reallocation any more.
          ==NULL: Feature disabled
@Purpose: Used in debug messages to identify individual memory allocations
@return:  Start address of allocated memory
          This return value is provided mainly for compatibility reasons.
          Using the Pointer argument is the prefered way to use tghis function.
```

This function allocates a given amount of memory via *kmalloc()* and returns its kernel logical start address. This functions can replace *kmalloc()* in every case.

A.1.4 rt_kmalloc0()

```
Prototype: unsigned long rt_kmalloc0(unsigned int Owner, size_t Size, int Flags, void**
                                     Pointer, unsigned char* Purpose)
@Owner:   Owner-id as returned by rt_create_owner() earlier
@Size:    Number of bytes to allocate
@Flags:   kmalloc flags like GFP_KERNEL, GFP_ATOMIC, GFP_USER, …
@Pointer: !=NULL: Variable will be loaded with start address of allocated memory
                  (NULL in case of error)
                  The pointer will be stored and reset to NULL automatically by
                  rt_free_resources().
                  This allows to allocate memory in static function variables without
                  having to worry about deallocation or even reallocation any more.
          ==NULL: Feature disabled
@Purpose: Used in debug messages to identify individual memory allocations
@return:  Start address of allocated memory
          This return value is provided mainly for compatibility reasons.
          Using the Pointer argument is the prefered way to use tghis function.
```

This function allocates a given amount of memory via *kmalloc()* and fills it with zeros. See *rt_kmalloc()* for further details.

A.1.5 rt_kfree()

```
Prototype: void rt_kfree(unsigned int Owner, unsigned long Address)
```

This function frees one memory block being allocated by *rt_kmalloc()* or *rt_kmalloc0*().

Note: Memory blocks that have been allocated via *rt_kmalloc()* or *rt_kmalloc0*() must always be freed by *rt_kfree*() or *rt_free_resources*(). Otherwise *rt_free_resources*() will free an already freed memory block and cause a kernel warning.

A.1.6 rt_free_resources()

```
Prototype: void rt_free_resources(unsigned int Owner)
@Owner:  != 0: Unique identification number of owner of memories to be freed as
               returned by rt_create_owner().
         == 0: Means to free all resources of all owners
```

This function frees all kernel resources whether pages or memories that have been allocated by given owner. It is safe to call this function if no resources have been allocated so far. Owner may immediately allocate resources again.

A.1.7 smc_get_domains()

```
Prototype: unsigned int smc_get_domains(unsigned int** DomainIDs)
@DomainIDs: Will be filled with pointer to read-only array of XEN IDs of all connected
slave domains (inclusive current domain)
@return:    Number of connected slave domains (inclusive current domain)
```

This function returns a list of identification-numbers of all connected XEN domains. The list does not contain the XEN ID of master instance of *sm_communicator*.

Note: This function will block until all event-channels have been successfully established with other domains.

A.1.8 smc_get_shared_pages()

```
Prototype: unsigned int smc_get_shared_pages(MappedPage_t*** PageInfosPtr)
@PageInfos:   Will be filled with pointer to read-only array of pointers
              to MappedPage_t structs.
@return:      >0: number of entries in (*PageInfos)[]
              =0: Error occurred
```

This function returns list of all mapped memory-pages usable for inter-domain communication

Note: This function will block until all event-channels have been successfully established with other domains.

A.1.9 smc_register_handler()

```
Prototype: int smc_register_handler(void* InterruptHandler,
                                    unsigned long HandlerArgument)
@InterruptHandler:  Prototype: void InterruptHandler(unsigned long HandlerArgument,
                                                     unsigned int DomainID)
@HandlerArgument:   Provided as constant argument to InterruptHandler() on each call.
@return:            =0: success, error-code otherwise.
```

This function registers the given function as an event-handler for incoming events.

InterruptHandler() will be called on every incoming event and provided with HandlerArgument and the XEN domain id number of the originator of this event. All events are handled in IRQ context which forbids to make calls to functions which offer the cpu to another process, like *schedule()* or *msleep()*, inside InterruptHandler().

Note: This function will block until all event-channels have been successfully established with other domains.

A.1.10 smc_unregister_handler()

```
Prototype: void smc_unregister_handler(void* InterruptHandler)
@InterruptHandler:  prototype: void ec_interrupt_handler(unsigned long HandlerArgument,
unsigned int DomainID)
```

This function clears the registration of the given function as an event-handler. The handler function registered before will not be called anymore.

A.1.11 smc_send_event()

```
Prototype: int smc_send_event(unsigned int TargetID)
@TargetID:  XEN domain id number of target domain
@return:    0 on success, error code otherwise
```

This function sends an event to one given target domain. Signals are lightweight under XEN. This means that signals do not carry any extra information than the domain id number of their originator. The target domains interrupt handler must be able to gather more knowledge about a signal on its own. This is implemented by using a dedicated shared memory range to exchange more data.

Note: This function will block until all event-channels have been successfully established with other domains.

A.1.12 smc_printk()

```
Prototype: void smc_printk(const char *Format, …)
```

This function immediately sends the given string as a debug-message to the master instance of *sm_communicator*.

It behaves similar to normal printk() but it prints the resulting string into a shared memory buffer and transmits it to master instance of *sm_communicator*. If no master instance is available or central_debugs is unset, then the message will be passed to local printk().
This function will block until message has been successfully sent. Though this is very slow, it has the lowest possible latency. When this function returns the message has definitive arrived. This *behavior* can be very helpful when it comes to debug a kernel oops inside *sm_communicator* or *sm_nic*.

It is even possible to create a new console which reports directly to *smc_printk*(). This allows to redirect every oops message from kernel to master. See module parameter grab_console for details.

Note: *smc_printk*() is interrupt safe and may be called from IRQ context.

A.1.13 smc_memcpy_io()

```
Prototype: volatile char* smc_memcpy_io(volatile char *Dest, volatile char *Source,
                                        unsigned long Size)
@Dest:   start address of destination buffer
@Source: start address if source buffer
@Size:   number of bytes to copy
```

This function copies a number of bytes from Source to Dest suitable for shared memories. It serves as a central place to try several implementations of memory copying. *sm_nic* uses this function to copy data to and from input buffers.

A.1.14 smn_receive_packets()

```
Prototype: int smn_receive_packets(unsigned int TxID, unsigned long* OctetsCount,
unsigned int* PacketCount, struct sk_buff** ReceivedPackets)
@SenderID:        >0: XEN ID number whose corresponding receive buffer is to be checked
                  =0: will check all receive-buffers for incoming packets (takes longer)
@OctetsCount:     loaded with number of octets received in total
@PacketCount:     value is increased for each packet being received
@ReceivedPackets: Array that stores pointers to all received packets
                  This array must be able to store one packet pointer per receive-
                  buffer.
                  Note: The returned sk_buff blocks do only store the retrieved data.
                        smn_receive_packets() does not know about any packet-types or
                        linux devices.
@return:           1: successfully received at least one packet
                   0: no incoming data received
                  <0: general error occured during receive
```

This low-level receive function will check only the given receive-buffer or all receive buffers for incoming packets from other XEN domains.

The function does not use the TCP/IP stack at all. It is the fastest way to receive data from other domains. This function is normally called from an interrupt handler to grab data from the receive buffers on an incomming interrupt or signal event.

Note: As the returned packets have been allocated dyamically, the caller of this function has to call packet_free() on each of them in order to return them to the memory pool.

A.1.15 smn_transmit_buffer()

```
Prototype: int smn_transmit_buffer(unsigned int TargetID, unsigned char* Buffer,
                                   unsigned int Length)
@TargetID:  XEN domain id number of domain to send data to
@Buffer:    Binary data to send
@Length:    Number of octets (bytes) to send from Buffer[]
@return:    =1: buffer send successfully + target domain signalled
                There is no guarantee that target domain has processed its buffer.
            =0: Buffer has not been sent because target buffer was not free and no
                signal was sent to target domain
            <0: Error-code
```

This function transmits a given buffer over shared memory to the named target domain. It implements a low-level send function.

The TCP/IP stack is not incorporated at all. Therefore it provides the fastest way to send data to other domains. The delivery is not guaranteed.

This function is not protected against concurrency. Especially when the *smn* device is in use by the TCP/IP stack, then lots of concurrent calls may occur.

In such cases, the caller has to ensure that interrupts are disabled to avoid concurrent calls while this function is busy transmitting a buffer!

For concurrency safe transport use *smn_send_packet*() instead!

A.1.16 smn_send_packet()

```
Prototype: int smn_send_packet(int TargetID, struct sk_buff *Packet)
@TargetID: XEN domain id number of domain to send data to.
@Packet:   Socket buffer which stores data to send.
@return:   =0: Packet has been queued for later delivery.
           =1: Packet has been sent immediately and *Packet was freed automatically.
           <0: Error code
               (Packet has NOT been freed and caller has to free it!)
           -E_TOO_MUCH_PACKETS: Packet has been freed before being processed.
```

This function sends the content of a given socket buffer to the given target domain. It will send the given buffer immediately if the corresponding target receive buffer is free. A send job in a work queue is scheduled for delayed transmission otherwise.

The socket buffer if freed automatically after a successful transmission.

This function protects itself from concurrent calls. If *smn_send_packet*() is called concurrently, then the packet will be added to the corresponding wait queue instead of sending it. The delivery will then be initiated by a parallel worker thread later. This worker thread will call *smn_send_packet*() outside of IRQ context.

A.2 Scripts Providing the Text User Interface

A.2.1 /XEN/bin/smx_BackUpCreate.pl

```
Synopsis: smx_BackUpCreate.pl COMMENT
```

This script creates a backup of all directories inside /XEN/ except /XEN/backup/. It automatically shuts down all running user domains before starting the backup. The argument COMMENT serves as a short description of the state of the current system to back up.

A.2.2 /XEN/bin/smx_BackUpRestore.pl

```
Synopsis: smx_BackUpRestore.pl TIMESTAMP
```

Backups from the folder */XEN/backup/* can be restored with this script. Before the restoration ist started, all old folders inside the folder */XEN/* are moved into */XEN/alt/*.

The individual backup to be restored is identified via its time stamp. If the argument TIMESTAMP is omitted, then a list of all valid back up time stamps is displayed.

A.2.3 /XEN/bin/smx_Benchmark.pl

```
Synopsis: smx_Benchmark.pl IDENTIFIER NIC_PREFIX COMPUTE_HOST1 COMPUTE_HOST2 …
```

This script runs the pre installed MPI benchmark suite on the given set of domains via the currently running shared memory or the default XEN network.

IDENTIFIER string which is used as filename prefix for log file of this run

NIC_PREFIX = xnc: communicate via XEN network

= smn: communicate via *smn* devices

COMPUTE_HOSTx: valid entry from /etc/hosts of domain to include in benchmark

Examples:

```
smx_Benchmark.pl xnc xnc2 xnc3 xnc4 xnc5 xnc6 xnc7 xnc8
smx_Benchmark.pl smn xnc2 xnc3 xnc4 xnc5 xnc6 xnc7 xnc8
```

The log files of all benchmark runs are stored in /XEN/share/Benchmarks/. The current benchmark can be observed by starting the shell script */XEN/share/Benchmarks/currentLog.sh* from a text console.

A.2.4 /XEN/bin/smx_BenchmarkFileCopy.pl

```
Synopsis: smx_Benchmark.pl IDENTIFIER NIC_PREFIX COMPUTE_HOST1 COMPUTE_HOST2 …
```

This script implement another benchmark by copying a file of fixed size from one domain to another. The time of this copy process is taken and printed to a log file.

A.2.5 /XEN/bin/smx_Control.sh

```
Synopsis: smx_Control.sh start|stop|restart
```

This script allows to start, stop or restart all currently configured user domains. It automatically obtains a list of domain names from the script *smx_DomainNames.pl.*

During the stop action, domains which do not shut down properly after some delay will be destroyed automatically.

A.2.6 /XEN/bin/smx_DomainIP.sh

```
Synopsis: smx_DomainIP.sh DomainName
```

When called, this script returns the current IP address of one user domain.

A.2.7 /XEN/bin/smx_DomainNames.pl

```
Synopsis: smx_DomainNames.pl
```

This script prints out a list of all currently configured user domains. The list of all domains is obtained from the xm command line tool that is provided by the XEN tools. Only those entries are returned which name start with the string “xnc”.

This script is called by all other scripts to determine the current set of XEN domains to operate on. If less than the full number of configured xnc<N>, <N> a number, should be used then a space separated list of domain names has to be provided in the file */XEN/share/cfg.Domains*.

A.2.8 /XEN/bin/smx_ExtractPerformanceValues.pl

```
Synopsis: smx_ExtractPerformanceValues.pl DOMAIN
```

Every time when *sm_nic.ko* gets unloaded, it prints out values of internal performance counters. This script logs into domain DOMAIN and extracts the performance values from /var/log/messages.

A.2.9 /XEN/bin/smx_LoadModules.pl

```
Synopsis: smx_LoadModules.pl start|stop|reload||load_smn|unload_smn|load_smn [RESETLOG]
```

This script loads, unloads or reloads all kernel modules that are required to establish the message passing network. It is also possible to load, unload or reload only the *sm_nic* module in the slave domains. This allows to reuse the already shared memory pages.

It calls */XEN/share/bin/load_master.sh* and */XEN/share/bin/load_slave.sh* to load the individual modules.

A.2.10 /XEN/bin/smx_RestartAllInterfaces.pl

```
Synopsis: smx_RestartAllInterfaces.pl RESTART
```

During interface shutdown, the kernel module frees all outstanding packets in its *TxFIFOs*. When the interface is brought up again, its buffers are empty and the module starts in a defined state. This script shuts down the smn interface in all domains and restarts it afterwards.

A.2.11 /XEN/bin/smx_RunBenchmarks.pl

```
Synopsis: smx_runBenchmarks.pl COMMENT|STOP|SIMULATE [START_INDEX]
```

COMMENT	filename prefix for log files of all benchmark runs in current set
STOP	stops a currently running set of benchmarks after finishing current run
SIMULATE	prints out configurations of each run without issuing a real run
START_INDEX	allows to resume at a certain benchmark run

This script runs individual sets of benchmark runs. It is intended to be duplicated and modified according to individual needs. The original version, that is provided with this thesis, implements the set of benchmark runs which where used to create the benchmark results in Chapter 6.

The log files from all benchmark runs are created in the folder */XEN/share/Benchmarks/*.

A.2.12 /XEN/bin/smx_TOP.pl

```
Synopsis: smx_TOP.pl
```

This script connects to each running user domain and prints a summarized list of all processes which currently consume CPU time.

A.2.13 /XEN/bin/smx_TestSSH.pl

```
Synopsis: smx_TestSSH.pl NIC_PREFIX ALL|HOST1,[HOST2, ..]
```

This script allows to test SSH connectivity between all named user domains. It tests if it is possible to automatically login via SSH between each two domains. The printed table shows the overall connectivity between each two domains.

In order to successfully run MPI benchmarks, SSH connectivity is required at least from first domain to every other. This script can be used to detect broken connectivity automatically.

NIC_PREFIX = xnc: communicate via XEN network
= smn: communicate via *smn* devices

Examples:

```
/XEN/bin/smx_TestSSH.pl smn ALL
/XEN/bin/smx_TestSSH.pl xnc xnc1 xnc2 xnc3 xnc4
```

A.2.14 /XEN/bin/smx_UpdateIPs.sh

```
Synopsis: smx_UpdateIPs.sh
```

This script logs into all configured user domains and collects the current IP addresses of XEN and *smn* network interfaces. The file /etc/hosts of *Dom0* and all user domains gets updated accordingly. After issuing this script, the network interfaces of all domains can be accessed by their XEN names instead of only their I addresses.

A.2.15 /XEN/bin/smx_UpdateXenIDs.pl

```
Synopsis: smx_UpdateXenIDs.pl
```

This script calls the xm tool to obtain current XEN ID numbers of all running user domains. Each time when the domains are restarted via smx_Control.sh, all domains get new id numbers assigned.

It creates a file named “/XEN/share/XenID.<NAME>”, <NAME> is the corresponding domain name. Each file contains the current domain ID number.

A.2.16 /XEN/bin/smx_Watchdog.pl

```
Synopsis: smx_Watchdog.pl
```

Whenever a benchmark run finishes successfully then a special file is created which acts as a flag. The script smx_Watchdog.pl deletes this file and checks if it reappears within one hour. If not, it is assumed that the benchmark has crashed. All domains will be shut down and the last benchmark gets restarted. This script gets invoked by smx_RunBenchmarks.pl automatically.

A.2.17 /XEN/share/bin/getXenID.sh

```
Synopsis: getXenID.sh
```

Thi script prints out the XEN ID number of the currently running user domain. For this it reads the files which have been created by smx_UpdateXenIDs.pl.

A.2.18 /XEN/share/bin/load_master.sh

```
Synopsis: load_master.sh start|stop
```

This script loads or unloads a master instance of *sm_communicator* in the current domain. A master instance may be loaded in *Dom0* or any user domain. To Load *sm_communicator* as master in a user domain has the advantage that a kernel oops, caused by this module, will not likely crash the whole machine. The script updateGrantRefs.pl is used to extract grant reference numbers from the file */var/log/messages* and to update configuration files in the folder */XEN/share/*.

A.2.19 /XEN/share/bin/load_slave.sh

```
Synopsis: load_slave.sh start|stop
```

This script loads or unloads the modules *sm_communicator* and *sm_nic* in the current *DomU*. It reads in reference numbers that have been extracted by load_master.sh and passes them to the slave instance of *sm_communicator.*

A.2.20 /XEN/share/bin/updateGrantRefs.pl

```
Synopsis: updateGrantRefs.pl
```

This script extracts all generated page grant reference numbers from /var/log/messages and writes them into files of format “/XEN/share/GRANTS.<ID>”. Where <ID> is the XEN ID number of each user domain.

A.2.21 Defining Configuration Sets for Benchmarking

The scirp /XEN/bin/smx_RunBenchmarks.pl allows to define several sets of configurations for the research platform to use for benchmarking. Rhe current configuration is applied to the whole platform before each benchmark run. This section describes how the individual configuration sets can be defined inside the script.

The sets have to be defined as nested arrays in PERL syntax. Nested arrays are supported in PERL by placing references to arrays inside arrays. Such array references may be defined directly with their content by enclosing individual scalar values by square brackets.

E.g.: “[[1,2,3], [4,5,6]]” in PERL syntax defines an array that contains two array, each of which containing three scalar values.

For the run script, each parameter of a configuration has to be given as reference to an array which contains these two elements:

- A reference to an array containing all possible values
- A reference to a sub routine that modifies the current configuration

  The individual value is passed as $_[0] to this subroutine. It may directly change the values in the two Hashes %CFG and %VLB. A more detailed description of these data structures can be found inside the PERL script.

PERL allows to define a so called “unnamed” subroutine and return a reference to it. This can be done by use of the keyword “sub” which is followed by a block that is enclosed by curly brackets.

The example below shows the definition of one configuration parameter called MTU:

```
[ [1500, 750, 375, 187, 150],  sub { $CFG{MTU} = $_[0]; } ]
```

When one or more of these configuration parameters are put into one array, then this forms one set of configurations. Any number of configuration sets may be handed to the run function as a comma separated list at once.

The example in Figure 40 shows how to run two sets of benchmarks. One set varies only the MTU value, while the second set varies the network interface type and the number of domains.

All individual benchmark runs are enumerated. If the machine crashes during a run, then it can be continued at any time later.

```
1 RunBenchmarkSets(
2     [
3       [ [1500, 750, 375, 187, 150],  sub { $CFG{MTU}           = $_[0]; } ],
4     ],
5     [
6       [ ['xnc', 'smn'],              sub { $CFG{InterfaceType} = $_[0]; } ],
7       [ \@BenchmarkDomains,          sub { $CFG{Domains}       = $_[0]; } ],
8     ],
9 );
```

Figure 40: Example Definition of Two Configuration Sets

A.3 Figures

A.4 Glossary

DomU / Dom0	Under XEN virtualization, each running instance of an operating system is called domain. The very first domain is special in that is has no privilege restrictions on accessing hardware resources. This domain is called *Dom0*. All other domains are called user domains or *DomU*. → p. 33
Evade Node	Node which accepts indirect packets to forward them to their intended target. → p. 28
Foreign Packets	Data packets which have been received by a node and which are destined at another node. → p. 28
insmod	Command under Linux which loads a kernel module into memory and calls its initialization function().
kmalloc()	Kernel function which allocates a range of memory.
Kernel Oops	Special error message generated by Linux kernel in case of a severe error. Prints out CPU registers, caller stack and some other usefull debug data. In most cases, the occurrence an a kernel oops is accompanied by a freeze of the operating system.
malloc()	User space function which allocates a range of memory.
memcpy()	Kernel function that copies a block of memory to another address.
MPA	Message Passing Architecture → p. 22
NoC	Network on Chip. Reuse of standard network technology to transmit data among several blocks on the same die. NoCs provide better scalability than buses for higher number of nodes.
Node	Used as synonym for domain in context of this thesis. In a NoC, each individual participant is often called a node. This research platform uses XEN domains to emulate the nodes of a shared memory based message passing network. → p. 12
NUMA	Non uniform memory access. Type of memory architecture for multi processor architectures. From perspective of each CPU memory if divided into local and remote memory with different latencies and bandwidth.
paravirtualization	Type of virtualization by which the virtualized operating system knows about its virtualization and cooperates with the hypervisor. Paravirtualization is faster and does not require hardware support for virtualization but it requires a modified guest OS.
Perfomance Values	Performance measures that are tracked inside the kernel modules *sm_nic.ko* and *sm_nic_vlb.ko*. → p. 65
resource_tracker	Kernel module providing basic functionality to other kernel modules. → p. 49
RxBuffer	Receive Buffer. Shared Memory which I situated local to the node which reads data from this buffer. On a SMP machine, such a locality may not exist. → p. 55
sm_communicator	Kernel module implementing basic features required for message passing over shared memory. → p. 51

sm_nic	Kernel module implementing a virtual Shared Memory Network Interface Card. → p. 54
SMP	Symmetric Multiprocessing. Two or more identical processors can access the same coherent shared memory [Wik09].
TLB	Translation Lookaside Buffer. On x86 architectures, a dedicated buffer caches a certain amount of translations from virtual to physical addresses. This buffer gets flushed under certain conditions. Each flush can have a great performance impact, because further memory require new translations to be computed.
trap-n-emulate	Basic technique of hardware based virtualization. Whenever a guest domain executes a privileged instruction and exception is generated by the CPU. This is because the operating system runs in a lower privileged ring than its intended ring (ring 0 for kernel code). This is called a trap. The exception handler then emulates the behavior of real hardware.
TxBuffer	Target Buffer. Buffer being remote for the sender node. → p.22