2009 Eighth International Conference on Networks

Efficient Virtualization of High-Performance Network Interfaces

Holger Fröning, Heiner Litz and Ulrich Brüning
Computer Architecture Group, Institute for Computer Engineering
University of Heidelberg
Mannheim, Germany
{holger.froening, heiner.litz, ulrich.bruening}@ziti.uni-heidelberg.de

The architecture of modern computing systems is getting more and more parallel, in order to exploit more of the offered parallelism by applications and to increase the system's overall performance. This includes multiple cores per processor module, multi-threading techniques and the resurgence of interest in virtual machines. In spite of this amount of parallelism the network interface is typically available only once. If the network interface is not able to exploit the offered parallelism, it becomes a bottleneck limiting the system's overall performance. To overcome this situation a new virtualization method for network interface is proposed, relying on a speculative mechanism to enqueue work requests. It offers unconstrained and parallel access without any involvement of software instances, allowing to exploit any available parallelism. For an unrestricted use the I/O interface is not modified. Modern parallel computing systems and Virtual Machine environments can be significantly improved with this new virtualization technique.

Keywords-Device Virtualization, High Performance Networking, Virtual Machine Environment, Speculative Enqueue

I. INTRODUCTION

It is commonly accepted that the requirements for computing power are steadily increasing. Still the most cost-effective solution for this requirement is cluster computing. This is also shown by the TOP500 Supercomputer Sites List [1]. In the current list (November 2008) 410 cluster based supercomputers are listed, which is 82% of the 500 fastest supercomputers.

Cluster computing is most effective regarding the cost-performance ratio, if commodity-of-the-shelf components are used. The only exception is the network, the bottleneck of cluster computing. Sophisticated networks designed as cluster interconnects are for example Infiniband specified by the Infiniband Trade Association [2], Quadrics [3] and Myrinet [4], trying to keep pace with the processor architecture developments. Comparable to the developments in the processor architecture, the architecture of interconnection networks must be steadily improved.

The paper presented here focuses on the improvement of the Network Interface (NI) architecture. Goal is to virtualize a NI in a sense, that any number of client processes is supported with a very high efficiency and low overhead. The virtualization is performed completely in hardware, shifting multiplexing and scheduling overhead from software layers to more efficient hardware. Furthermore it is completely transparent to the processes. From their point of view, each has exclusive access to the network interface. A scalable queue-based interface with synchronization support between processes and the NI allows very efficient exploitation of the available parallelism. The new architecture presented here is not limited to NIs, it can be applied to any kind of (high performance) devices.

The main contribution of this work is a speculative mechanism to enqueue work requests on the device. It allows an efficient, scalable and safe virtualization of devices. The key features of the device virtualization are the following:

- Complete hardware based virtualization
- Highly efficient use of the limiting I/O interface
- Scalable with number of accessing processes
- Secure process identification to prevent misuse
- Cost-efficient by minimal on-device resources

The remaining sections of this paper are organized as follows. In the next section the background of this work is presented. The architecture of the virtualized network interface is described in section 3. In section 4 the impact of this work on two example applications is shown, which is a cluster interconnect and a Virtual Machine environment. Section 5 provides basic performance data which is obtained from real world experiments. We conclude in the last section and provide a short outlook.

II. BACKGROUND

Traditionally, the operating system (O/S) multiplexes the accesses to network interface (or any kind of device) to support more than one client. Here the O/S can supervise the client processes regarding correctness of the operations. But the required system calls introduce additional overhead which leads to performance degradation. The solution for this problem is the principle of User-Level Communication [5]. Here, a process can directly open the device for exclusive usage. Overhead due to system calls is completely avoided, but the task of supervision has to be done by the

978-0-7695-3552-4/09 $25.00 © 2009 IEEE
DOI 10.1109/ICN.2009.23

device itself. Additionally only one process can open the device. A promising technique to overcome this situation is the context switching. It supports several processes accessing the device. Each access is followed by a context switch. In the context process specific configuration information are stored. Typically the switching takes place in a time-sliced manner, so the available parallelism generated by several processes is not exploited.

The Infiniband specification defines support for up to 216 user interfaces, which are implemented as queue pairs (QP). Each QP can be assigned to a different process. The Infiniband hardware implementations from Mellanox [6] or Voltaire [7] which support this large number of processes use the context-switching based approach. Best to our knowledge the implementation details are not specified and the methods developed by the manufacturers like are not published. A modification of the firmware of a Mellanox Host Channel Adapter [8] also allows the use in Virtual Machine environments.

One major reason for the need of network interface virtualization is the increasing number of processes running on a system as a result of the increasing parallelism of computer systems (SMT, CMP, CMT). Applications are increasingly divided into multiple threads to exploit the available parallelism. One of the most recent developments in modern processor architecture is the virtualization of the CPU [9] [10]. This virtualization technology enables simplification of Virtual Machine Monitor (VMM) software. VMMs like Xen [11] [12] allow multiple operating systems (guests) to execute concurrently on commodity x86 hardware. Devices are either used exclusively by one guest or shared with the help of the VMM. For shared devices the data transfer is replaced by asynchronous I/O rings and interrupts by event notifications. User-Level Communication is not possible. A recent approach is to implement the virtual device abstraction on the device itself by using an embedded network processor with 8 Ethernet communication cores [13]. The supported limit is 64 virtual machines and due to the targeted application User-Level Communication is not possible.

A very recent approach is to leverage the flow control of the HyperTransport (HT) [14] protocol to implement a virtualized network interface [15], resulting in a minimal overhead and high efficiency. Obviously this approach is only compatible with HT based I/O interfaces, while the goal here is to develop a virtualization method compatible with any kind of I/O system.

III. Architecture of a Virtualized Network Interface

The architecture proposed here is enabled by context switching of hardware threads running on the device. It allows a large amount of processes/threads to access a virtualized network interface (VNI) simultaneously. A VNI provides several virtual ports, each identified by a Virtual Port Identifier (VPID). Each client process is assigned to such a virtual port. The clients issue work requests using Virtual Communication Instructions (VCI).

A. Basic architecture

Instead of executing the VCIs in a timed-sliced manner, the architecture should be able to exploit the available parallelism. A very efficient approach is to use superscalar functional units (FUs) together with multi-threading, known as SMT [17]. SMT allows each FU to be assigned to another working thread. More sophisticated implementations can even pipeline the FUs and allow each pipeline stage of a FU to be assigned to another thread. Due to the absence of partitioning SMT exploits thread-level parallelism (TLP) or coarse-grain parallelism as well as instruction-level parallelism (ILP) or fine-grain parallelism [18]. It perfectly fits to the situation when only one process is issuing several VCIs simultaneously to the VNI, as well as several processes simultaneously using the VNI with one or more VCIs.

B. Recognizing processes

The most important requirement regarding virtualization is that a VNI has to distinguish between accesses from different processes. Accesses must be tagged with the VPID to allow the VNI to recognize the different processes. The VNI stores the information relevant to one VPID in a context. A context contains in particular all information about the interface between process and VNI (e.g. queue offset, read and write pointers), the current state of the virtual port, configuration information and permissions. Fast context switches occur between operations of different processes, configuring the execution unit for this specific user process.

Here process recognition is realized by providing a set of pages in the peripheral address space. Each process maps a page and communicates directly with the VNI using read/write operations. The address associated with the operation is used to distinguish the different processes. This method is the most efficient way to signal a unique identifier to the VNI when using commodity hardware. The virtualization is transparent to a process, which is in particular important for security reasons. The main processor's Memory Management Unit (MMU) prevents processes from accessing pages from other processes.

C. Interface between processes and VNI

Compared to the main memory of the system, on-device memory is expensive. When scaling the number of processes the on-device memory may also become a bottleneck. To keep the requirements for on-device memory as small as possible, queues in main memory are very suitable as interface from processes to VNI and vice versa. Pinned and contiguous memory regions are used, allowing access using either virtual or physical addresses without doing an address translation for each page. For the direction from process to VNI these queues contain instructions (work queues), for the opposite direction notifications (notification queues). We follow the standard approach to separate work requests into a descriptor part and a payload part.

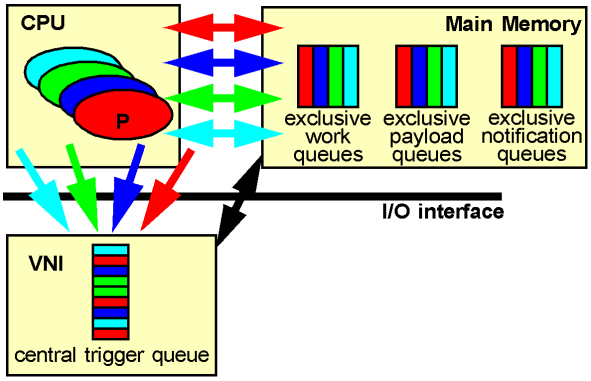

Figure 1. Interface to a Virtualized NI

Each insertion of a new element is at least dependant on a free entry in the queue: Before each insertion, the available space must be checked. The cycle of checking and insertion is a critical region and may not be interrupted by other processes. Instead of checking for available space, another solution is to ensure at least one free entry in the queue, e.g. done by Mellanox's Infiniband adapter. But this solution introduces a lot of overhead, and all insertion operations must be directed to the device. The device then stores the queue elements somewhere in main memory. Best to our knowledge, the exact working principle is not published.

Instead of using shared queues we propose to assign exclusive queues in main memory for every process (figure 1). This scales very well with the process count, furthermore only for used VPIDs queues have to be allocated. The required memory on the VNI is minimized, which significantly reduces the costs [19]. Because of the missing cache coherence scheme for I/O interconnects, the VNI cannot recognize changes in main memory. Polling continuously on the queues is not a solution due to the waste of I/O bandwidth; additionally it does not scale with the number of queues. The best solution is to notify the VNI of a new VCI using a central queue located in the VNI. Changes to this central trigger queue are immediately visible to the VNI. Only the trigger information (i.e. the VPID, here 16 bit) is stored in this queue. To maintain scalability, this central queue is shared among all client processes. The corresponding context points to the work queue in main memory which contains the VCI itself.

D. Instruction Issue

So a VNI must be explicitly informed by inserting the VPID into the central shared trigger queue. The instruction issue (or trigger operation) as an enqueue operation is normally divided into two steps, the check for available space and the insertion of a new element. Because this is a shared queue, these two steps are a critical region and may not be interrupted. Using an atomic operation is a solution for this problem. The read part is the check for available space and the write part is the insertion of the VPID. Unfortunately, atomic operations are not supported on common I/O interfaces and introduce additional overhead.

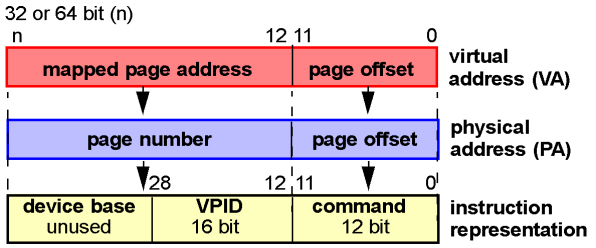

Figure 2. Interpretation of addresses

Our new approach is a kind of speculative execution of the enqueue operation. Information about the success of the operation must be provided back to the producer. The very small payload of a queue element (only the VPID, here 16 bit) allows using a single read operation. The read address is used as immediate value, containing the VPID and a command. The read result returns the information about the result of the enqueue operation back to the producer. If more work requests are issued to the VNI than can be processed, the queue fills. In the case of a full queue the enqueue must be retried.

The processes use virtual addresses to access the VNI. The MMU automatically translates the virtual address into a physical one (figure 2). This address translation only changes page information, the lower part of the address remains unchanged. Thus the lower part (page offset) is used as command section given by the calling process and the upper part (page number) contains the VPID.

During enqueue the VNI observes a read operation to a physical address. If a queue entry is free, it converts the address to a VPID/command pair, which is then stored in the central trigger queue. Otherwise the enqueue operation has failed. In both cases the process is informed about the result of the enqueue operation by including it in the response of the read operation. If the instruction is accepted, the VNI returns True, otherwise False. Of course more sophisticated return results are possible. If the I/O interconnect supports split-phase transactions, this mechanism is even improved. Several issue operations can be outstanding, and the VNI is able to favor certain ones.

By coding information in the address a larger amount of address space is required. The paging principle limits address mapping to pages, so every process opens at least one page, named the Triggerpage. Applying an access granularity of 64bit, 512 different read addresses are possible, allowing the trigger operation to be tagged with 512 different commands.

The required hardware for this access scheme is only a queue. The storage in the queue is dependant on several conditions, so the queue is called Conditional Store Buffer (CSB) [20]. An insertion into this CSB is conditional regarding available space, access rights and other restrictions. The capability of the VNI to accept or reject instructions can be used e.g. for prioritization of instructions. The complete issue operation from user process to VNI is shown in figure 3.

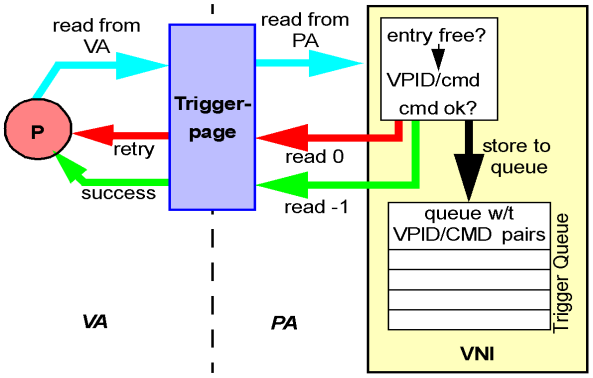

Figure 3. Issue operation

User applications are considered to be insecure, e.g. due to programming errors or due to harmful software. The Triggerpage is an efficient solution for this problem. The page mapping mechanism used here prevents processes from accessing data/control structures from other processes, because the VPID is contained in the page number of the read address, which is only visible after address translation. Further, the complete mechanism does not involve other kernel- or user-level processes, enabling User-Level Communication without additional overhead for security.

IV. APPLICATIONS

The previous introduced architecture is now evaluated for two example applications. The first example is the use in a cluster system. The parallelism within a cluster node can be fully exploited by a virtualized network interface. The other example is a Virtual Machine environment. Because the virtualization mechanism here requires no software overhead, it is perfectly suited for Virtual Machine environments.

A. *Cluster Interconnect*

Cluster computing implies a lot of parallelism, and modern system architectures allow running an increasing number of processes on one node. In a typical MPI environment, each core is assigned one process. All processes can now access the local networking interface for communication, rendering dedicated communication processes unnecessary.

All accesses from the host side to the VNI are fully virtualized by the Triggerpage and the CSB. Another access is possible, coming from the network side. As opposed to the software access, no security mechanisms are required here, all traffic being received is generated by other network interfaces. These are expected to generate only valid network packets. Hence the only requirement for network access is to have all network packets tagged with a context identifier. For the support of 216 processes, this will only add 16 bits to the header section of each packet.

During packet injection by the sending NI, the target VPID is well-known as the end-point of communication. Hence it can be simply integrated into the packet header. Upon receipt of a packet, the target NI fetches the appropriate context to configure one of its FUs, where this packet is then processed.

B. *Improving Virtual Machine environments*

VMMs like Xen allow multiple operating systems (guest systems) to execute concurrently on commodity x86 hardware. Each guest system runs in another domain. The VMM is part of the privileged domain 0 where also one guest system is running. The VMM virtualizes the underlying hardware, e.g. CPU, memory and devices. A device is shared by redirecting all I/O traffic to the VMM, which is responsible to schedule the accesses.

Virtual Machine environments benefit a lot from hardware based virtualization. It allows every guest operating system to open any number of virtual ports of the VNI. Processes from different guest systems can concurrently and directly access the VNI without scheduling by the VMM. The VMM is only responsible to manage the VNI. Each process (independent from which guest system, if kernel- or user-level) can directly access the VNI over a Triggerpage. The VNI itself is responsible for the scheduling of work requests from the different processes. From the VNI's point of view, it is completely irrelevant if these processes are part of one or different guest systems.

Figure 4 shows a typical situation in a Virtual Machine environment with a VNI. The VNI provides one management page and replicated Triggerpages. All Triggerpages in the physical address space map to the same hardware resource. Configuration and management of the VNI is done by the VMM, which runs in domain 0. There are no restrictions regarding the domain of a process or the total number of domains. All Triggerpages can be mapped into one domain or each into another domain.

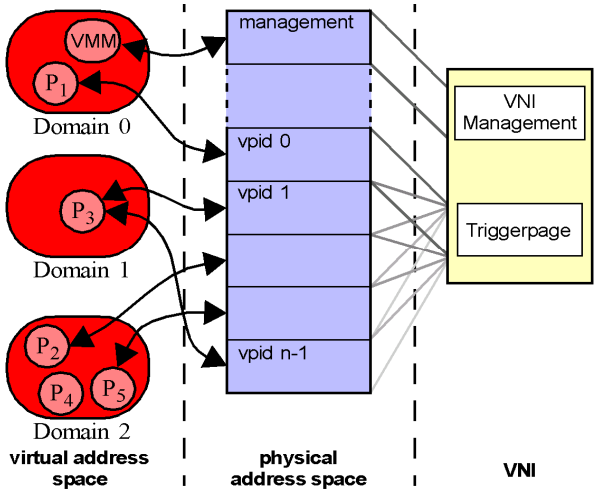

Figure 4. Virtual Machine environment

V. EXPERIMENTAL RESULTS

The speculative mechanism to enqueue work requests as the most important part of the virtualization has been tested successfully using an FPGA device. This FPGA device is connected to the host using a PCI interface running at 33 MHz. The host runs under Linux 2.6 and is equipped with a 2.8 GHz Intel Xeon (single core) and 1 GB of RAM. The tests include a single user process issuing several instructions, as well as several user processes issuing several instructions (see Table 1).

The first test shows that a work request enqueue does not last longer than a normal PCI read operation (which is 630ns for a PCI-33MHz device). This in particular shows the excellent performance of the speculative enqueue mechanism.

Only one process is used in this test to prevent O/S scheduling effects, but several different VPIDs are used to test the process identification. Hence the device itself sees accesses from different processes.

The next two tests increase the number of client processes. Compared to the first test now multiple processes are competing for the CPUs, but the device sees again accesses from different processes. The required scheduling by O/S dramatically increases the average time for a one work request issue. This is in particular shown by the non-linear increase from the second to the third test. While the number of processes is increased by a factor of 8, the measured average latency is increased by a factor of approximately 14. This is even better visualized by the total time required, which is 6.9ms for the second test, and 775ms for the third test (factor of approximately 112).

But these two tests show the basic functionality of the speculative work issue when accessed by a large number of client processes. For a better measurement of the performance a multi-core system is required. At the time of measuring this was not available; hence a single-core machine had to be used. In spite of this, the excellent performance of the work request issue to shown by the first test. This test avoids O/S scheduling effects, although it uses different VPIDs.

TABLE I. EXPERIMENTAL RESULTS

Client processes	Work request issues per process	Average time for one work request issue
1	1000 (using different VPIDs)	630 ns
8	8	107 µs
64	8	1500 µs

VI. CONCLUSION AND OUTLOOK

This paper presents a new and efficient mechanism to virtualize network interfaces. As computing systems become more parallel, unrestricted and efficient simultaneous access from user processes to the network interface becomes more and more important. In a Virtual Machine environment the virtualization can be used to bypass the VMM, minimizing the overall overhead.

In order to provide access to a non-virtualized device for any number of processes, O/S involvement is mandatory. This additional overhead is avoided by the virtualization presented here. It allows virtually any number of client processes to access directly the network interface. User-level Communication is still possible to minimize communication overhead.

Building blocks for the virtualization are a speculative mechanism to enqueue work requests on the device and an optimized set of queues. The queue set is separated into exclusive queues in main memory and one shared queue on the device. With main memory as cost-effective resource the exclusive queues scale with the number of processes. Shared is only the central trigger queue with minimal entry size. The speculative enqueue mechanism is implemented as a single read operation. The payload is included in the read address, information about the success in the read result. Security is guaranteed by the Triggerpage which is mapped into the user space.

Two example applications for a VNI are shown. In a cluster environment the available parallelism can be perfectly exploited by the network interface architecture, allowing each process to communicate directly and without any O/S involvement with other processes. Virtual Machine environments can also be improved a lot by the virtualization concept. Each guest O/S can directly access the network interface without involvement of the VMM or other supervising software instances.

An FPGA based experimental test system is used to measure the basic performance of the speculative work request issue mechanism. The performance data in terms of issue latency shows the excellent performance of the proposed virtualization method.

In the future a virtualized network interface is going to be developed on a HyperTransport connected rapid prototyping station [21], which can contain more logic than the FPGA used in the experiments here. This rapid prototyping station can be used to implement a complete network interface for a use in real world environments.

REFERENCES

[1] TOP500 Supercomputer Sites, http://www.top500.org.

[2] Infiniband Trade Association, InfiniBand Architecture Specification Release 1.2, 2004, available from http://www.infinibandta.org.

[3] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg, "The Quadrics Network: High-Performance Clustering Technology", IEEE Micro, 22(1):46-57, 2002.

[4] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su, "Myrinet: A Gigabit-persecond Local Area Network", IEEE Micro, 15(1):29-36, 1995.

[5] E.W. Felten, R.D. Alpert, A. Bilas, M.A. Blumrich, D.W. Clark, S. Damianakis, C. Dubnicki, L. Iftode, and K. Li, "Early experience with message-passing on the shrimp multicomputer", Proc. of the 23rd International Symposium on Computer Architecture (ISCA23), 1996.

[6] Mellanox Technologies, Inc., InfiniHost III Product Family. Available from http://www.mellanox.com.

[7] Voltaire, Inc., Voltaire HCA 4X0 Product Family. Available from http://www.voltaire.com.

[8] J. Liu, W. Huang, B. Abali and D. K. Panda, "High Performance VMM-Bypass I/O in Virtual Machines", Proc. of USENIX Annual Technical Conference, 2006.

[9] Advanced Micro Devices, Inc., AMD I/O Virtualization Technology (IOMMU) Specification. Available from http://developer.amd.com.

[10] Intel, Vanderpool technology. Available from http://www.intel.com/technology/platform-technology/virtualization/index.htm.

[11] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, „Xen and the art of virtualization", Proc. of the 19th ACM Symposium on Operating Systems Principles, 2003.

[12] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson. „Safe Hardware Access with the Xen Virtual Machine Monitor", Proc. of the 2004 ACM OASIS Workshop, 2004.

[13] H. Raj, K. Schwan, "High Performance and Scalable I/O Virtualization via Self-Virtualized Devices", Proc. of IEEE International Symposium on High-Performance Distributed Computing (HPDC), Monterey, California, USA, 2007.

[14] Hypertransport Consortium. 2008. HyperTransport™ I/O Link Specification Revision 3.10. http://www.hypertransport.org.

[15] H. Litz , H. Fröning, M. Nüssle and U. Brüning, "VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers", Proc. of the 37th International Conference on Parallel Processing (ICPP-08), Portland, Oregon, USA, 2008.

[16] PCI SIG, PCI Express 2.0 Base Specification, 2007.

[17] D. M. Tullsen, S. J. Eggers, H. M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism", Proc. of the 22nd International Symposium on Computer Architecture (ISCA), pages 392-403, 1995.

[18] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, D. M. Tullsen, "Simultaneous Multithreading: A Platform for Next-Generation Processors", IEEE Micro, 17(5):12-19, 1997.

[19] S. Sur, A. Vishnu, H.-W. Jin, W. Huang, D. K. Panda, "Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems?", Proc. of the 13th Annual IEEE Symposium on High-Performance Interconnects (HOTI), 2005.

[20] L. Schaelicke, A. Davis, "Improving I/O performance with a conditional store buffer", Proc. of the 31st annual ACM/IEEE International Symposium on Microarchitecture, Dallas, Texas, USA, Pages: 160 - 169, 1998.

[21] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, U. Brüning, "The HTX-Board: A Rapid Prototyping Station", Proc. of the 3rd annual FPGAworld Conference, Stockholm, Sweden, 2006.