Architectural Improvements of Interconnection Network Interfaces

Inauguraldissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

der Universität Mannheim

vorgelegt von

Holger Fröning
(Diplom-Informatiker der Technischen Informatik)

aus Braunschweig

Mannheim, 2007

Dekan:	Professor Dr. M. Krause, Universität Mannheim
Referent:	Professor Dr. U. Brüning, Universität Mannheim
Korreferent:	Professor Dr. R. Männer, Universität Mannheim

Tag der mündlichen Prüfung: 09. Juli 2007

Für Eva

Abstract

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. The recent trend for single systems is to include multiple cores in one processor module. Another trend is the introduction of virtual machines, which allows to run several operating systems (O/S) independently on one physical node. If the performance of one single system is not sufficient to meet the requirements, multiple single systems are combined in one parallel distributed system. Clusters are parallel distributed systems based on commodity computing parts and an interconnection network. They have an excellent cost-effectiveness and a high efficiency. This is substantiated by their increasing use in high performance computing. But they rely on a highly efficient interconnection network; otherwise computation is limited by the communication overhead.

While the computing nodes and systems become more and more parallel due to architectural improvements, virtual machines and parallel programming paradigms, 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.

Goal of this work is to overcome this situation and to develop a network interface architecture which offers unconstrained and parallel access by multiple processes. Any available parallelism should be exploited without limitations, which is in particular true for virtual machine environments. Beside the network interface architecture a set of communication and synchronization methods is developed, which allow a close coupling of the computing nodes. In particular for fine grain communication such a tight coupling is inevitable.

The developed network interface architecture has many similarities with the architecture of modern processors, but also introduces new techniques. It is based on Simultaneous Multi-Threading (SMT) and a memory hierarchy including an on-device Translation Look-aside Buffer. The SMT approach removes any partitioning, which allows to exploit any type of parallelism without constraints. While the SMT architecture for main processors relies on the O/S for context switching, the architecture here is self-switching. Upon an issue of a work request an available resource is switched to one of 2^16 contexts, each storing the configuration of the calling process.

The main contribution of this work is the introduction of a new technique to enqueue work requests by multiple producers into a central shared work queue. The processes are the producers, which issue work by enqueuing requests. The scheduler of the network interface is the consumer, which forwards the work requests to available functional units. This enqueue technique is essential for an efficient virtualization of the network interface. It allows almost any number of processes to issue simultaneously work requests, is completely transparent to the processes and provides security by separation. The key component to achieve a high efficiency is to avoid explicit mutual exclusion. This is achieved by integrating the complete issue process into a single operation, including flow control to inform the process of the success or failure of the enqueue operation.

Another key component of the network interface architecture is the Ultra Low Latency Transmission (ULTRA) unit. The goal here is to reduce the communication latency to a minimum, providing a tight coupling between nodes. In particular fine grain communication schemes rely on a close coupling. ULTRA uses techniques called pre-initialization and pre-completion to achieve lowest latencies. A single write access is then sufficient to inject a message into the network and a single read access can retrieve it. Instead of collecting small data structures into large bulk messages, they can now be sent out independently.

The new developed techniques like virtualization and ULTRA are successfully tested and evaluated. Both have been implemented on FPGA-based prototyping stations. It is noteworthy that both can be used with any I/O interface, because they do not require any special functionality. But in particular ULTRA can benefit a lot from a closer coupling between peripheral device and main processor than traditional I/O standards like PCI or PCIe can offer. For this purpose a new rapid prototyping station is developed, called HTX-Board. It is based on an FPGA and connects to the main system over a HyperTransport (HT) interface. The HT interface avoids any intermediate bridges between device and main processor, providing an excellent coupling. With the HTX-Board the full potential of ULTRA's communication technique is shown, resulting in yet unmatched latencies for commodity systems.

The most recent development in high performance computing is an increasing use of coprocessors for acceleration. A high-performance network interface is also a kind of coprocessor, providing CPU-offloading and acceleration. The insights gained during the development of the virtualization technique and ULTRA are used to analyze the requirements for a generic coprocessor interface. An instruction set extension is proposed to achieve a tight coupling between main processor and coprocessor. Key component is the tagging of load and store operations, which allows to include additional information. The efficiency and performance of the virtualization and ULTRA can be even improved by using the proposed instructions of this extension.

Zusammenfassung

Die Architektur moderner Computersysteme wird immer paralleler, um mehr der in Applikationen enthaltenen Parallelität zu nutzen und die Gesamtleistung des Systems zu erhöhen. Eine der neueren Entwicklungen ist mehrere Kerne in einem Prozessormodule zu integrieren. Eine andere Entwicklung ist wiedererwachtes Interesse in virtuellen Maschinen, welche es erlauben mehr als ein Betriebssystem auf einem Knoten auszuführen. Wenn aber die Leistung eines einzelnen Systems nicht ausreicht um die Anforderungen zu erfüllen, werden mehrere einzelne Systeme zu einem parallelen verteilten System zusammengeschlossen. Cluster sind solche parallele verteilte Systeme und basieren auf Standardkomponenten und einem spezialisiertem Verbindungsnetzwerk. Cluster sind sehr kostengünstig und bieten eine hohe Leistungseffizienz. Dies wird auch durch die zunehmende Nutzung im Hochleistungsrechnen belegt. Allerdings benötigen Cluster hochperformante Verbindungsnetzwerke, ansonsten ist die Rechenleistung durch den Kommunikationsoverhead beschränkt.

Während Rechenknoten und -system durch architektonische Verbesserungen, virtuelle Maschinen und paralleles Programmieren immer paralleler werden, ist die Netzwerkschnittstelle üblicherweise nur einmal pro Knoten vorhanden. Wenn die Netzwerkschnittstelle nicht in der Lage ist, die vorhandene Parallelität auszunutzen wird sie zu einem Flaschenhals und limitiert die Gesamtleistung des Systems. Ziel dieser Arbeit ist es diese Beschränkung zu überwinden und eine Netzwerkschnittstelle zu entwickeln, welche simultanen und parallelen Zugriff von mehreren Prozessen ermöglicht. Jegliche Art von Parallelität sollte ohne Einschränkungen nutzbar sein, was insbesondere für Virtuelle Maschinen gilt. Zusätzlich zu der Netzwerkschnittstelle wird ein Satz von Kommunikations- und Synchronisationsmethoden entwickelt, welche eine möglichst enge Kopplung der Knoten erlauben. Insbesondere für feingranulare Kommunikation ist eine enge Kopplung notwendig.

Die entwickelte Netzwerkschnittstelle hat viele Ähnlichkeiten mit der Architektur von modernen Prozessoren, nutzt aber auch in dieser Arbeit neu entwickelte Techniken. Sie basiert auf dem Prinzip des Simultaneous Multi-Threading (SMT) und einer Speicherhierarchie mit on-device Translation Look-aside Buffer. Der SMT Ansatz entfernt jegliche Partitionierung der Ressourcen, wodurch jede Art von Parallelität ohne Einschränkung genutzt werden kann. Währen die SMT Architektur von CPUs das Betriebssystem zum Kontextwechsel benötigt, ist die hier entwickelte Architektur in der Lage die Kontexte ohne Unterstützung und vollautomatisch zu wechseln. Wenn eine Arbeitsanforderung abgesetzt wird, wird bei einer freien Ressource zu einem der 2^{16} Kontexte gewechselt um die Arbeitsanforderung dort auszuführen.

ABSTRACT

Der Hauptbeitrag dieser Arbeit ist die Entwicklung einer neuartigen Methode zur Einreihung von Arbeitsanforderungen mehrerer Prozesse in eine zentrale geteilte Warteschlange. Die Prozesse agieren als Produzenten, welche Arbeit an die Netzwerkschnittstelle ausgeben. Der Scheduler auf der Netzwerkschnittstelle ist der Abnehmer, welcher dann die Arbeitsanforderungen an eine verfügbare Ressource ausgibt. Für eine effiziente Virtualisierung der Netzwerkschnittstelle ist diese Methode grundlegend. Die Anzahl Prozesse, welche gleichzeitig Arbeitsanforderungen ausgeben, kann nahezu beliebig hoch sein, die Virtualisierung ist transparent für die Prozesse und stellt Sicherheit und Abgrenzung der Prozesse sicher. Die wichtigste Aspekt für eine hohe Effizienz ist die Vermeidung von gegenseitigen Ausschluß. Dies wird erreicht indem der komplette Ausgabevorgang in einer einzigen Operation integriert wird. Dies beinhaltet auch eine Rückmeldung an den Prozeß über den Erfolg der Operation.

Eine weitere wichtige Komponente der Netzwerkschnittstelle ist das Ultra Low Latency Transmission (ULTRA) Modul. Das Ziel hier ist die Kommunikationslatenz auf ein Minimum zu reduzieren, um eine möglichst enge Kopplung der Knoten zu erreichen. Insbesondere feingranulare Kommunikationsschematas sind auf eine solche enge Kopplung angewiesen. ULTRA erreicht niedrigste Latenzen durch die Nutzung von pre-initialization und pre-completion Techniken. Ein einziger Schreibzugriff ist ausreichend um eine Nachricht zu generieren, und ein einziger Lesezugriff um die Nachricht zu empfangen. Statt kleine Datenmengen in großen Strukturen zu sammeln, kann man mit ULTRA diese auch sofort versenden.

Die neu entwickelten Techniken wie die Virtualisierung und ULTRA wurden erfolgreich getestet und evaluiert. Beide wurden als Prototyp auf einem FPGA implementiert. Erwähnenswert ist daß beide mit jeder I/O-Schnittstelle nutzbar sind, da keine spezielle Funktionalität gefordert wird. Allerdings kann insbesondere ULTRA von einer engeren Kopplung zwischen peripherem Gerät und CPU profitieren, als sie in einem traditionellen System mit PCI oder PCIe gegeben ist. Daher wurde ein neues FPGA-basiertes Prototypensystem entwickelt, genannt HTX-Board. Es nutzt als Verbindung zum Hauptsystem HyperTransport (HT). Eine HT-basierte Schnittstelle vermeidet Protokollkonvertierungen und bietet somit eine exzellente Kopplung. Mit dem HTX-Board kann das ganze Potential von ULTRAs Kommunikationstechnik gezeigt werden, was durch bisher unerreichte Kommunikationslatenzen für Standardsysteme sichtbar wird.

Der neueste Trend im Hochleistungsrechnen ist eine zunehmende Nutzung von Koprozessoren zur Beschleunigung. Eine Netzwerkschnittstelle kann auch als Koprozessor gesehen werden, da die CPU entlastet wird und die Operationen beschleunigt werden. Die Erkenntnisse die während der Entwicklung der Virtualisierung und ULTRA gewonnen wurden werden nun genutzt um ein die Anforderungen an eine generische Koprozessorschnittstelle zu analysieren. Eine Instruktionssatzerweiterung wird vorgeschlagen um eine möglichst enge Kopplung zwischen CPU und Koprozessor zu erreichen. Schlüsselkomponente ist die Kennzeichnung von Lade- und Speicheroperationen, welche es erlaubt zusätzliche Informationen in die Operation zu integrieren. Durch diese Erweiterung kann die Effizienz und die Leistung der Virtualisierung und der ULTRA sogar noch verbessert werden.

Abstract

Abstract

Contents

CHAPTER 1 INTRODUCTION

The architecture of modern computers is improving steadily. This is in particular true for the ability to process in parallel, achieved by architectural techniques like pipelining and replication. The most recent trend is to include an increasing number of computing cores in one processor module. Today processors with two cores are even available for mobile computing. Server processors with up to four cores are already announced by the leading processor manufacturers, while research projects already show processors consisting of up to 80 cores. The goal of the multi-core trend together with multi-threading techniques and multiple processors per system is to exploit as much of the offered parallelism from applications as possible. An increased exploitation of parallelism also increases the throughput of the system and the overall performance rises.

Another recent development is the resurgence of interest in Virtual Machine environments, where one physical computing machine is used to host more than one operating system. This allows to consolidate the workloads from multiple physical machines to a single one. Then the needs of the various applications regarding operating system type can still be fulfilled, while the administration costs are reduced. These Virtual Machines also increase the parallelism offered by a single system.

If the available performance of a single system is no longer sufficient to meet the requirements, several ones are combined to form parallel distributed systems. Each formerly single system is now a node in the parallel system. These parallel computing systems are based on interconnection networks to connect all nodes for communication and synchronization purposes. Clusters are parallel and distributed systems based on commodity computing parts and interconnection networks. The list of the 500 fastest supercomputers in the world shows that cluster computing is increasingly used to achieve highest performances. In the most recent list (November 2006, [1]) more than 70% of the fastest systems are clusters, substantiating the efficiency and cost-effectiveness of parallel distributed systems. The interconnection networks used for clusters are widely available, but differ significantly from commodity networks like Ethernet. They offer a much higher performance to allow a close coupling of the computing nodes. Otherwise the interconnect turns into the main bottleneck of the parallel system, limiting it's performance and scalability dramatically.

While the computing nodes become more and more parallel due to the architectural improvements, the network interface is typically only available once in a node. Unconstrained, efficient and simultaneous access to the network interface becomes inevitable. Only then the parallelism offered by the computing resources can also be fully exploited by the communication and synchronization resource. One of the major goals of this work is to develop a network interface architecture for an improved and unconstrained exploitation of any available parallelism. This architecture has many similarities with modern processor architectures. It can be seen that many techniques used for processors also apply for network interfaces. In order to offer as much parallelism as possible to the network interface architecture, a virtualization technique allows almost any number of processes to simultaneously access the network interface. The virtualization is not limited to network interfaces, almost any high performance device can benefit a lot from it. Key component of this virtualization technique is a highly efficient access scheme from user level. This new and unique method allows multiple processes to access a centralized shared queue without constraints. A single access is used to enqueue elements, which is in particular used to issue new work requests from processes to the device.

Virtualization of network interfaces is not only usable in the area of high performance computing, also computational data centers offering utility computing can benefit from it [2]. They differ significantly in their mode of operation from clusters. The virtualization of network interfaces is required to form logical sub-networks, resulting in an increased flexibility and availability.

Beside the architectural improvements, a sophisticated set of communication and synchronization methods is developed to allow a close coupling in a parallel distributed system. This close coupling is in particular essential for fine grain computation and communication.

The most recent development during this work is an increasing interest in coprocessing for acceleration of certain tasks. In particular the virtualization technique can also be applied for coprocessors, which are then accessible with less restrictions. The benefits of virtualization in parallel environments also apply for coprocessors. For an even improved virtualization with less required resources an instruction set extension is developed and presented. This extension is only based on three new instructions, which take into account the different application requirements.

Overview of remaining parts. This work starts with an introduction into the most important topics of parallel computing, including parallel systems, interconnection networks and network interfaces. The introduction is completed with a summarization of goals developed during this first chapter.

The *second chapter* focusses on communication and synchronization. It starts with a short introduction, followed by the development of communication and synchronization methods for high performance computing. The basic functionality of communication is explained and a set of communication instructions is developed.

These insights are applied in the *third chapter*, where the network interface architecture itself is developed. For a comprehensive understanding, this chapter again starts with an introduction into the basic topics of network interfaces. Then the architecture itself is developed, suitable to exploit any kind of parallelism. This together with a scalable queue design leads to the development of the virtualization. The virtualization allows to offer enough parallelism to the architecture. The third chapter is completed with an architecture and communication method suitable for fine grain communication. This method allows lowest communication latencies, which are yet not achieved with commodity parts. The summary of the developments in this chapter forms a network interface suitable for almost all use cases where high performance is inevitable.

The developments of the previous two chapters are agglomerated in *chapter four* and a specification of the required data structures, communication instructions and the architecture is provided. Key components of this work are implemented and evaluated to prove their efficiency and performance.

Chapter five takes into account the recent interest in coprocessing for acceleration. In particular the developed virtualization technique is also suitable for coprocessors, but the close coupling to the main processors allows even improvements. An instruction set extension is proposed for a close coupling of coprocessor and main processors. Only three new instructions are introduced. This efficient and performant interface allows the virtualization of the coprocessor with less required resources. Also other applications like fine grain communication can benefit a lot from this extension.

This work concludes in *chapter six*, showing what is achieved and provides an outlook on future developments. Key developments of this work are an architecture exploiting any kind of parallelism, device virtualization based on a highly efficient issue method, support for fine grain communication and an instruction set extension for a close coupling of coprocessors.

1.1 PARALLEL SYSTEMS

Parallel systems target to increase the performance of a computing system. If a non-parallel system does not meet the requirements regarding throughput, parallel techniques can increase the overall performance of the now parallel system. Two basic techniques exist, which are pipelining and replication.

The *pipelining* approach is based on forming an assembly line or pipeline by a number of functional units. Each functional unit performs a certain *stage* of the computing. For a single computation a pipeline does not provide any parallelism. But if several computations are executed they are overlapped by the functional units. If the time required to fill the pipeline is negligible compared to the amount of time when the pipeline is full, the speed-up equals the pipeline depth. Another important property of pipelining is the efficiency. Pipelining does not increase the required resources linearly, instead only a small overhead is required for each pipeline stage. Pipelining is typically applied inside a processor, when a functional unit is divided into several pipeline stages to increase the throughput. For instance, a *functional decomposition* of the execution leads to one stage for instruction fetch, instruction decode, register fetch, execute and write back.

The other approach is simply based on *replication* of units. These units can be complete nodes, processors or functional units of a processor. Each replicated unit adds the same amount of resources and in summary the resource requirement scales linearly. Hence the replication is less efficient than pipelining. But while pipelining suffers from the overhead to fill a pipeline, replication can be used without constraints. If enough parallelism is included in the workload and the required resources are available, replication fully exploits the available parallelism.

1.1.1 Design space diagrams

With the classification of parallel systems also the *design space* analysis is introduced. To depict a design space, representations based on [3] and [4] are used.

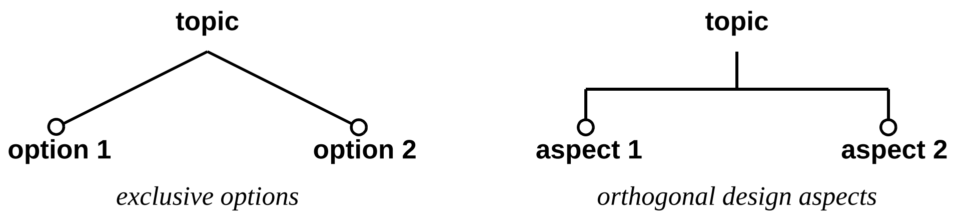


Figure 1.1 Design space diagrams

In figure 1.1 the two different design space diagrams used in this work are shown. These two kinds of diagrams are building blocks and can be concatenated to represent more complex design spaces.

The diagram on the left side shows a design space with several implementation options. Only two are shown, if more possibilities exist the number of branches increases. The different options of a topic can only be used exclusively, combinations are not possible. Options are always depicted using straight lines.

The diagram on the right side shows the design space for orthogonal design aspects. In this case the branches are completely independent of each other. Comparable to the first diagram, this diagram can also be extended by more branches. Design aspects are always depicted using orthogonal lines.

1.1.2 Classification

The first classification is based on the kind of parallelism used. Figure 1.2 shows that the design space of parallel architectures can be divided into data-parallel and function-parallel architectures [3]. While there are no major representatives for data-parallel architectures today, the function-parallel architectures are widely used. For this work data-parallel architectures are not relevant, hence the following focusses on the analysis of function-parallel architectures.

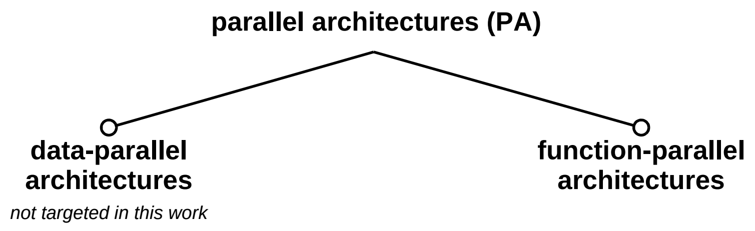

parallel architectures (PA)
data-parallel architectures
not targeted in this work
function-parallel architectures

Figure 1.2 Design space of parallel architectures

After the classification based on the kind of parallelism, in figure 1.3 the function-parallel architectures are now divided according to the granularity of the parallelism they utilize. The different types of granularity are instructions, threads and processes. The resulting parallelism is also referred to as fine, medium and coarse grain parallelism. The large amount of dependencies in fine grain parallelism requires a close coupling of the parallel units. This can usually only be achieved within one computational unit, for instance a *Central Processing Unit (CPU).*

The granularity of process-level parallel architectures allows to replicate complete computational units, which can be a multi-processor system or a network of computing nodes. These two examples already show the next classification, which takes the memory access scheme into account.

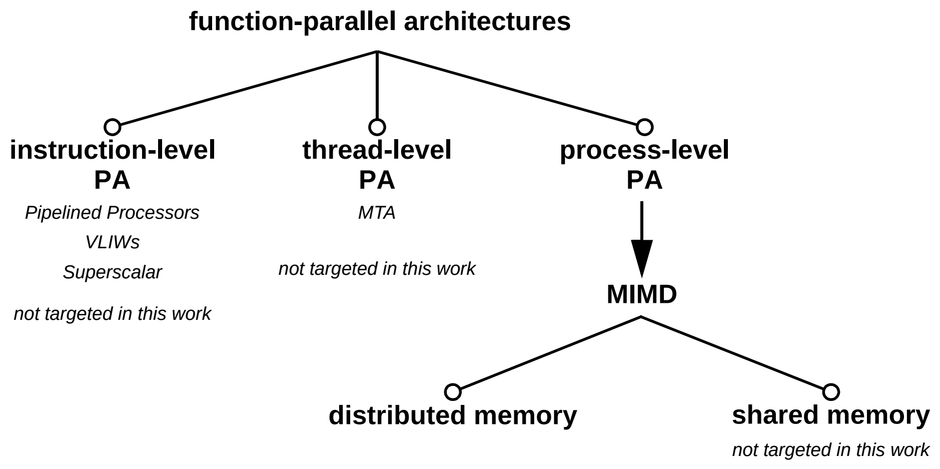


Figure 1.3 Design space of function-parallel architectures

Process-parallel systems can be classified into shared or distributed memory systems. In a *shared memory* system each computational unit can directly access every memory location without involvement of other units, resulting in a single address space. The shared memory limits the scalability of such systems, because for instance the overhead for *cache coherency* increases dramatically.

In a *distributed memory system* access to *remote* locations involves other units. This leads to multiple address spaces and the memory is divided into local and remote regions. Multiple address spaces allow to build up a parallel system by replicating complete *nodes*, which include processor, memory, I/O and peripherals. These nodes communicate and synchronize using an *interconnection network*. A very popular type of distributed memory architectures are *clusters*. The building block of a cluster is a more or less standard workstation. The use of clusters has steadily increased during the past years, which is also shown in the bi-annual TOP500 list [1].

The scalability of clusters is limited by the applied workload and, more important, by the used interconnection network. Widely available commodity products like Ethernet are not suitable, because they do not provide sufficient performance for a close coupling of the computing nodes. Specialized interconnects especially designed for cluster computing provide a higher performance. But comparable to the advances in computer architecture, these interconnects must be steadily improved to keep pace with the processor performance increase.

1.1.3 Communication and Synchronization

Communication and synchronization is the working basis for a parallel architecture. Computational units (e.g. computing nodes) communicate in order to transfer data and synchronize to notify each other of data availability, which is required to continue processing.

Analogue to the parallelism types the communication can be classified into fine, medium and coarse grain communication. While *coarse grain communication* is based on bulk transfers of large data structures, in *fine grain communication* small data elements are exchanged among the computational units. Hence for fine grain communication much less overhead can be tolerated compared to medium and in particular coarse grain communication. But for a close coupling of the computing units support for fine grain communication is inevitable, otherwise the system does not scale.

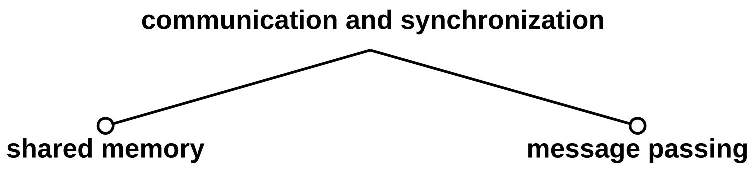


Figure 1.4 Design space of communication and synchronization

Figure 1.4 shows the design space of communication and synchronization, which can be classified into message passing and shared memory. Communication is useless without synchronization and vice versa. Otherwise data is transferred, but the counterpart is not informed of new available data. Hence the communication and synchronization scheme must always be seen together.

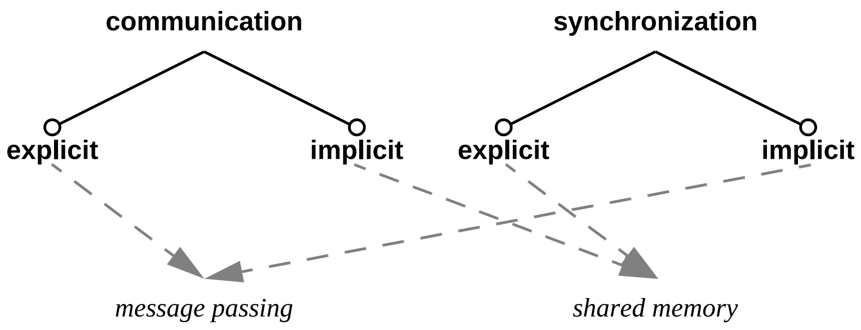


Figure 1.5 Explicit and implicit communication/synchronization

In *message passing* data is transferred between two units using messages. A message includes the data as payload. To identify the received data and associate it’s purpose, additionally a source and destination identification and a tag is included to describe it’s payload. Because the transfer of messages involves both the source and the destination in the communication, this mechanism also provides *implicit synchronization.*

Shared memory is based on memory regions which are accessible by all communication partners. For communication data is stored in a shared memory region, and then fetched from there. Sharing a region leads to multiple writers accessing the same location. If these accesses are performed simultaneously the correct behavior cannot be ensured. *Mutual exclusion* is required to ensure that a certain location is only accessed by one writer per point of time. Mutual exclusion requires synchronization, which can be achieved using *semaphores*.

While the *implicit communication* provided by shared memory is more intuitive than the *explicit communication* for message passing, shared memory requires *explicit synchronization*. In message passing provides *implicit synchronization* for each communication. The required synchronization complexity for both methods is equal.

Network Interface Controller. Message passing systems rely on a *Network Interface Controller (NIC)* (or network device), which is explicitly accessed for sending and receiving of messages (see figure 1.6). This NIC does not only controls the access to the network, more sophisticated ones also off-load certain tasks required for message passing from the CPU.

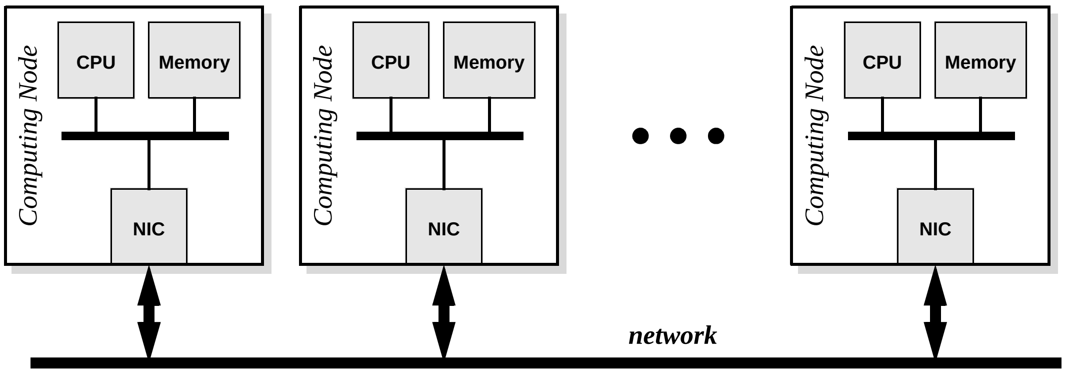


Figure 1.6 Network Interface Controller (NIC)

Shared Memory Mapper. For communication and synchronization based on shared memory no explicit access to the network is required. Either all memory is accessible directly from the main system interconnect, or a device called *Shared Memory Mapper* is present in the system. The first applies for all single systems, where the processors and the memory are directly interconnected (see figure 1.7). Shared Memory Mappers are only required in systems composed of multiple computing nodes (see figure 1.8), where the access to remote memory is otherwise not possible.

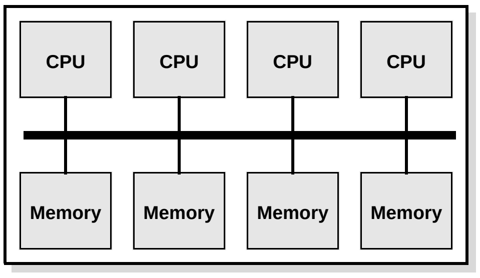


Figure 1.7 Single shared memory system

Shared Memory Mappers provide a set of pages in their I/O space. These pages are mapped to remote memory locations. Each access to such a page results in a communication to the appropriate node, where the data is either fetched or stored. A Shared Memory Mapper allows to combine a parallel architecture with distributed memory and a shared memory communication and synchronization scheme. But unpredictable latencies for memory accesses limit the scalability of such systems. A user process cannot distinguish between a local access and a remote access. Because cache coherency protocols are based on collective communication they are limited in scalability. A directory-based cache protocol can diminish this effect, but for larger distributed memory architectures it is not possible to cache remote accesses.

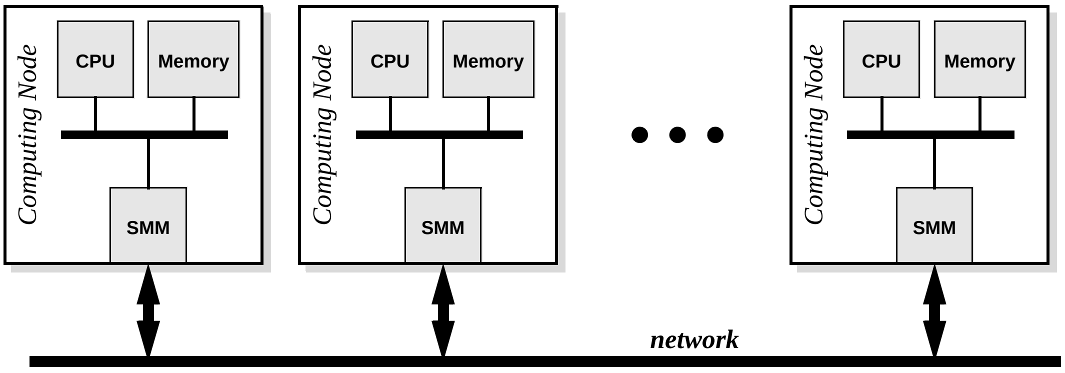


Figure 1.8 Shared Memory Mappers (SMM)

Because shared memory systems and in particular Shared Memory Mappers are limited in their scalability, this work focusses on improvements of network interfaces for message passing. The message passing approach is much more suitable for distributed systems, where the access to remote locations suffers from a latency several orders of magnitude larger than a local access.

1.1.4 Messaging layers and communication functions

Message passing applications rely on the use of messaging layers which provide an *Application Programming Interface (API)*. These software layers are typically implemented as libraries. The API includes functions for communication and synchronization, which are used by the application to communicate and synchronize.

Each messaging layer includes a certain software overhead. This overhead increases with the gap between requirements from the application and functionality offered by the network [5]. For instance, if the network does not provide reliability, the messaging layer must ensure this. Another example is in-order delivery of messages. If the network cannot ensure that messages are delivered in-order, the messaging layer must re-sort the messages.

These layers also abstract the underlying hardware. The application can use generic communication functions, which are independent of the hardware implementation. This allows to exchange the interconnection network, while the application remains unchanged.

Typical communication libraries are based on the specification of the *Message-Passing Interface 1 (MPI-1)* [6][7] and *Message-Passing Interface 2 (MPI-2)* [8][9]. Two example implementations are MPICH [10] and OpenMPI [11]. They are all similar in their basic functionality, but differ in the details.

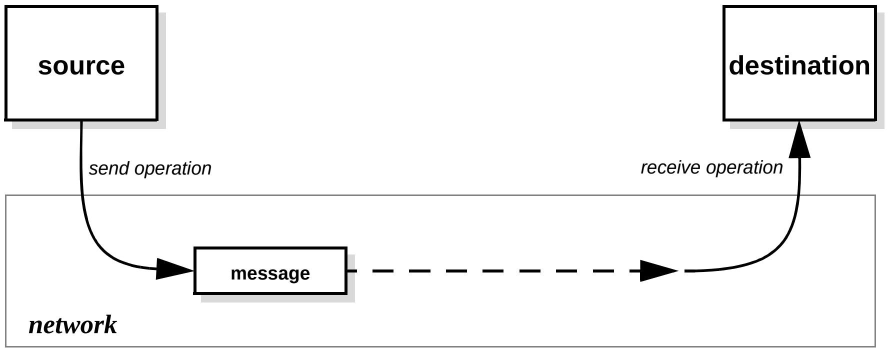


Figure 1.9 Send/Receive communication scheme

Obviously message passing requires functions to send and receive messages. This is in the following referred to as *Send/Receive scheme* (see figure 1.9). For such a scheme both the source and the destination are involved in the communication.

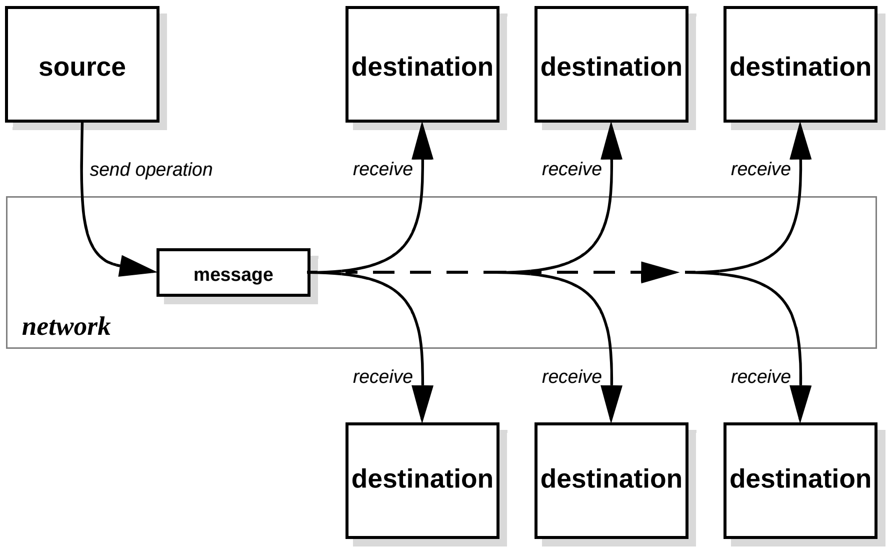


Figure 1.10 Collective communication scheme

Collective communication schemes (see figure 1.10) involve more than one destination in the communication. Such a communication function either distributes data to several destinations by a single send and multiple receives, or it collects data by multiple sends and a single receive. Support for collective operations a typical task for a communication library, where it is performed by multiple send/receive operations. Some basic collective operations like *broadcasts, multicasts* or *barriers* are sometimes supported directly by the network interface.

Another communication scheme is *Remote Memory Access (RMA)* (see figure 1.11). While a send/receive function cannot directly access the remote memory, some network interfaces provide support for RMA. To write the payload in a remote memory an *RMA Put operation* is used, while an *RMA Get operation* fetches data from remote locations. RMA operations do not involve the destination process in the communication, thus here no synchronization is possible.

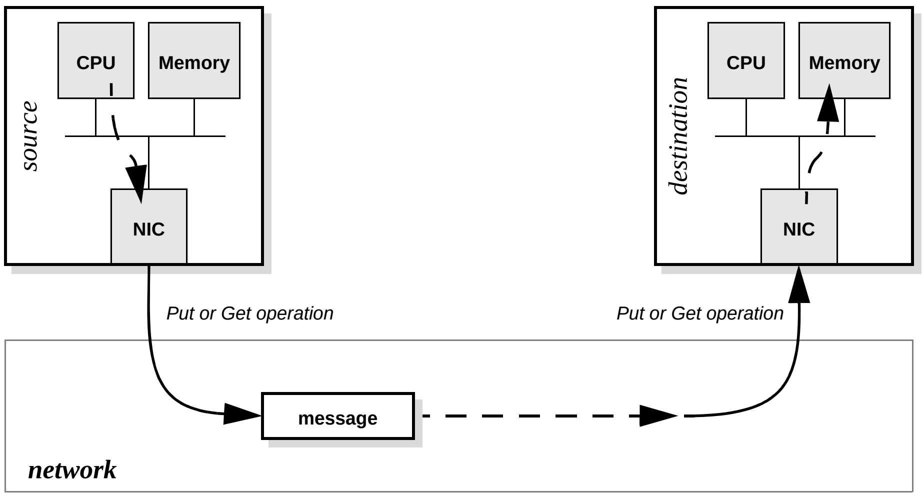


Figure 1.11 Remote Memory Access (RMA) communication scheme

1.1.5 Node architecture

In order to increase the performance of modern computing systems more and more, one goal is an increased exploitation of available parallelism. This is achieved by the parallel techniques introduced above, which are pipelining and replication. The replication technique results in multi-processor computing systems (*Symmetric Multi Processor, SMP*). Each of the processors is composed of several computing cores (*multi core*). Furthermore each core (or single processor with only one core) is *superscalar* with multiple functional units, perhaps even multi-threaded which allows to execute multiple threads simultaneously on time shared resources (*Simultaneous Multi-Threading* [12][13] or HyperThreading [14][15]).

The high access costs to main memory are diminished by a *memory hierarchy*, which introduces caching structures and register files. The caches in an SMP system are kept consistent and coherent using a *cache coherency protocol*, which ensures that data copies in caches are not outdated. Either these copies are invalidated or updated, depending on the policy (*write-invalidate policy* or *write-update policy*).

The I/O sub-system is separated from the main system by an I/O bridge. *Peripheral devices* are located in the I/O sub-system. The peripheral interconnect is usually a standardized one, allowing to use peripheral devices in different architectures. The peripheral protocol differs significantly from the main system protocol, hence bridges are necessary to perform protocol conversions. While modern main system interconnects are always cache coherent, no modern peripheral interconnect offers this. A typical I/O device is thus not cache coherent.

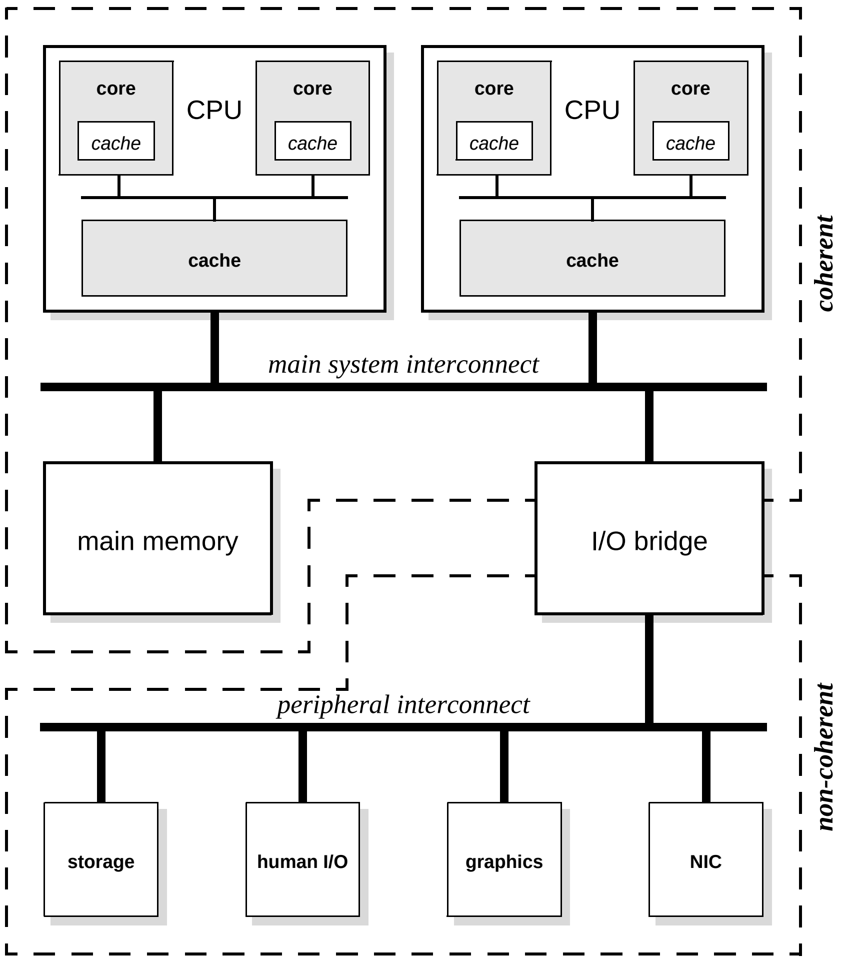


Figure 1.12 Node architecture

1.1.6 *Virtual Machine Environments*

The performance potential housed in a modern computing node recently lead to a resurgence of interest in *Virtual Machine (VM) environments* [16]. On a normal computing node only one *Operating System (O/S)* is running.

VM environments offer the ability of several simultaneously running O/S by abstracting the underlying hardware. Each VM is a complete set of resources, which are required for computing. This includes in particular the CPU, the memory and I/O. A privileged software layer above each O/S virtualizes the physical machine and provides several virtual machines. By running on a VM each O/S has the illusion of a physical machine for it's exclusive use. VM environments are perfectly suited for workload consolidation, when several computing systems with different requirements regarding the O/S are now executed on one single physical machine.

1.2 INTERCONNECTION NETWORKS

The goal of this sub-chapter is to shortly introduce *Interconnection Networks (INs)*. It starts with the key functionality, followed by basic performance metrics and finally some interconnect examples are shown.

A full and in-depth analysis is not required here. More details about the large area of interconnection networks can be found in [17] and [18].

1.2.1 Basics

Figure 1.13 shows the design space of interconnection networks. All these design aspects influence significantly the overall performance of the network.

An IN is composed of a number of *nodes* which are connected using *switches*. The switches forward incoming packets to the appropriate outputs. The *source* node injects *packets* into the network which are routed over one or several *hops* until they finally reach their *destination* node. Packets are composed of a *header* containing in particular the routing information, the *payload* and a *tail* as delimiter.

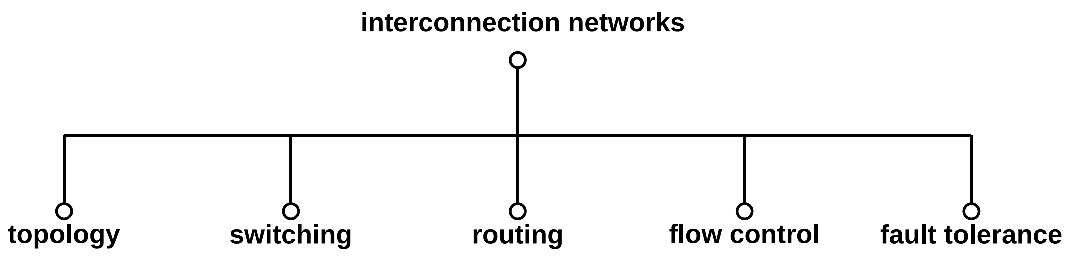


Figure 1.13 Design space of interconnection networks

Topology. The topology of INs describes the connections among the nodes of an IN. It can be furthermore classified into direct and indirect topologies. In a *direct topology* each node is directly connected to it’s neighbors, while in an *indirect topology* intermediate switches are used. These switches connect several nodes with each other. Hence in a direct topology the switching resources are distributed over the network and in an indirect topology centralized switching resources are used. The *node degree*, which is the number of links per node, can be only one for an indirect topology, while a direct topology requires more than one link per node.

Furthermore the topology can be *regular* or *irregular*. A regular topology usually results in simplifications of the routing. It is important to be mentioned, that a single fault of an IN component turns almost every regular topology into an irregular one. Hence it is not desirable to rely on a regular topology if fault tolerance is important.

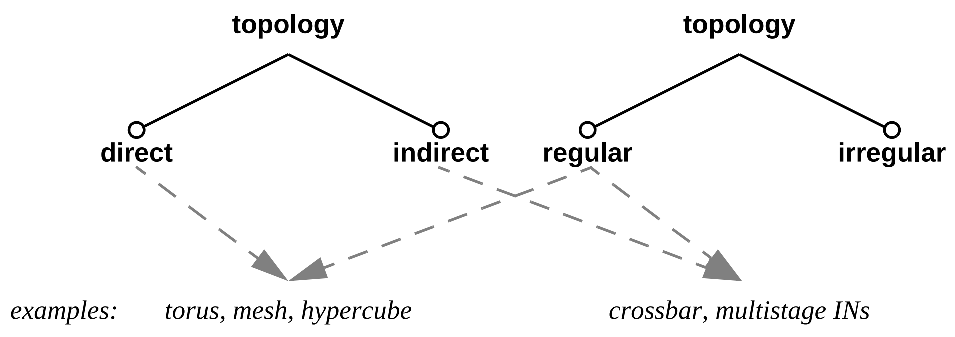


Figure 1.14 Design space of interconnection network topologies

Switching. The switching describes the method to forward packets on their way from source to destination. The first switching method is *store-and-forward*. Here an incoming packet is completely stored in the switch, before it's next hop is calculated. Optimizing this method leads to *virtual cut-through*, which immediately starts to calculate the next hop if the sufficient header of a packet has arrived in a switch. This reduces the required latency to forward a packet. The need to buffer complete packets in a switch can be diminished by *wormhole* switching. Here a packet is pipelined through the network by dividing it into smaller units (*flow control units, flits*). Each switch only provides buffer space to store one flit. Only the first flit of a packet contains the routing information, all subsequent flits are following the route of the first flit. If the first flit is blocked, immediately all following flits are also blocked.

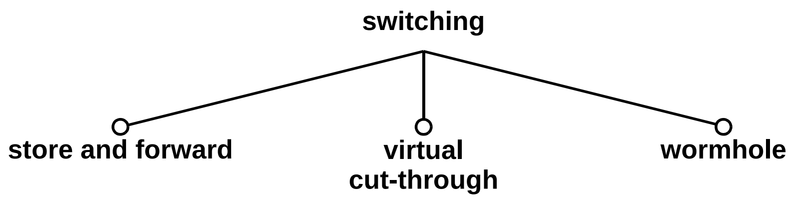


Figure 1.15 Design space of interconnection network switching methods

Routing. The routing describes the method to find the way over several hops from source to destination. It can be classified into deterministic and adaptive routing. In *deterministic routing* the route is pre-calculated and not changed during transmission. *Adaptive routing* allows to change the path from source to destination if the need arises. Reasons to change the route can be faults, blockings or congestions.

Two typical examples for routing techniques are source path routing and table-based routing. *Source path routing* is a deterministic routing where each hop of the route is included as routing string in the header of the packet. Each hop from source to destination consumes one entry of the routing string. This leads to very low switching latencies because no complex calculations are necessary. Using *table-based routing* only the destination identifier is included in the packet header. On each hop this identifier is used to index a table where the next output link is stored. This technique can be both deterministic and adaptive, but the required table lookup leads to higher switching latencies compared to source path routing.

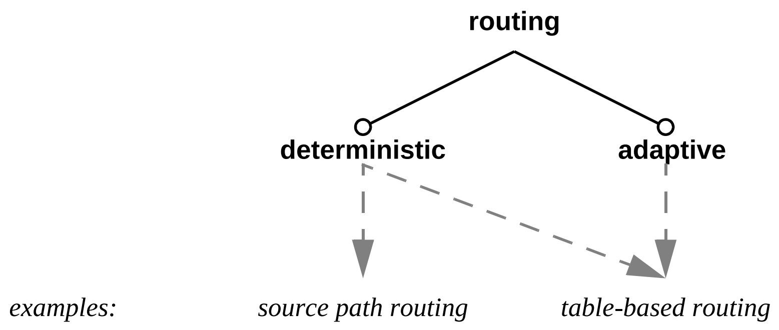


Figure 1.16 Design space of interconnection network routing methods

Flow Control. In an IN *flow control* is required to ensure that the next hop has sufficient buffer space to store an incoming packet. The unit under flow control (flit) is then granted transmission or not. A flit can be a complete packet or only a part of it. The latter is in particular true for wormhole switched networks, where the buffers are kept very small and can only store a part of the packet. The simplest method to implement flow control is a *stop/continue scheme*. If the buffer space falls below a certain limit a stop is signalled back. The transmission can only continue if the buffer space rises above another limit and a continue is signalled. The limits must take into account that signalling back takes a certain amount of time. In this time the previous hop is still sending packets.

In the *credit-based flow control* scheme the sending hop has a certain number of credits, which matches the number of flits that can be stored at the next hop. Transmission is only possible if enough credits are available. The credit number is decreased with each transmission. If the receiving hop can free some of it’s buffers by forwarding the stored flits to the next hops, it sends credits back for these freed buffers. Then the amount of credits at the sending hop increases again and matches the number of buffer space at the receiving node. The credit-based flow control scheme is independent of the propagation delay for signalling and the sending hop has precise information about available buffers at the receiving hop.

Fault Tolerance. In particular for a network which consists of a large number of distributed components fault tolerance is inevitable. Otherwise a single failure of one of the components can disable the complete network. Fault tolerance has many aspects, but at least few should be mentioned. Fundamental for a high speed interconnect are *reliable transmissions,* guaranteeing that a packet once injected reaches it's destination within finite time. Otherwise the messaging layer must guarantee the delivery, which requires to store a copy until it is ensured that the packet has reached it's destination. To ensure reliable transmission on link level, typically a *Cyclic Redundancy Check (CRC)* is included in the tail of the packet. Using this it can be checked if the packet was altered during transmission. Such failures are detected and solved by retransmission at link level.

Reliable transmission at link level is not sufficient to guarantee fault tolerance for the complete network. Furthermore it must be possible to tolerate failures of links, switches, nodes or any component involved in the network. A complete fault tolerance is obviously not possible, considering the case that the source or destination node breaks down. But toleration the fault of intermediate nodes, links or switches is desirable. This can be achieved in networks with alternate routes to the destination by adaptive routing. Adaptive routing allows to modify the route of a packet during transmission. This imposes several other problems, for instance regarding in-order delivery of packets. Further details about adaptive routing techniques can be found in [17].

1.2.2 Performance metrics

The two key metrics typically used to describe the performance of an interconnection network is the peak bandwidth and start-up latency. The *peak bandwidth* is the achievable throughput measured in bytes per second over one link. Usually the bandwidth increases with the packet size, because then the impact of overhead per packet is diminished. The *start-up latency* is the shortest time to send a packet from a user-level to user-level, including at least one hop. The start-up latency is typically achieved with the smallest possible packet sizes.

Using these two metrics only a basic performance analysis of the IN is possible, hence various other metrics do exist. For instance the *n/2 number* describes the required packet size to achieve half of the peak bandwidth. The *bisection bandwidth* metric cuts the network into two equal parts and accumulates the bandwidth of all links passing the border.

1.2.3 Examples

Implementations of interconnection networks can be classified by their interface to the host system. Either the used interface is a special solution or standardized (see figure 1.17). Specialized interfaces allow a much closer coupling of computing resources and network which results in higher performance. But their usage is restricted to custom systems. It is not possible to use such an IN with commodity parts.

INs based on standardized interfaces primary target an unrestricted use. Here the I/O interface is standardized (for instance PCI [19] or PCI-Express [20][21]) which allows to use this IN in every commodity system which provides such extension slots. But the standardized I/O interfaces also limit the coupling between network and computing resource [22]. This looser coupling reduces the achievable performance for these networks. This is also substantiated by the last TOP500 list (November 2006) [1], showing that the 4 fastest supercomputers are based on special solutions.

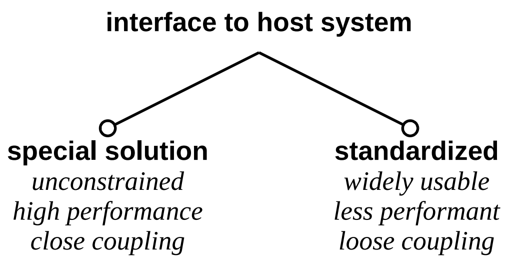


Figure 1.17 Design space of interfaces of interconnection network

Examples for INs based on special solutions are the Transputer [23], iWarp [24], IBM's Blue Gene [25][26][27] or almost any Cray supercomputer like the XD1 [28]. Compared to INs based on standardized I/O interfaces these special solutions are rather seldom. The specialized interface usually leads to the fact that these INs are not only networks but complete systems.

In contradiction to this an IN based on a standardized interface is usually only a network, which is combined with the computing resources. Examples for commercial high performance INs are Quadrics's QsNet2 [29][30], Infiniband [31] (with implementations from Mellanox [32] or Voltaire [33]), InfiniPath [34][35] by QLogic (formerly PathScale), Myrinet [36] by Myricom [37] or 10Gig Ethernet. Examples for research INs from academia are the ATOLL network [38][39] or DimmNet [40].

1.3 NETWORK INTERFACE ARCHITECTURE

This sub-chapter provides an overview of the most important properties of network interfaces. A network interface is typically a high performance device and especially in distributed systems a potential bottleneck limiting the overall performance.

A recent trend is to off-load more and more functionality from the CPU to the device, resulting in more complex hardware structures. This is in particular true when the O/S is bypassed for user level access. But the reduced CPU load levels this additional complexity and the system’s overall performance benefits from the off-loading.

1.3.1 Network interface locations

In [41] a comprehensive overview of possible locations for *Network Interfaces (NI)* is provided. Regarding performance a close coupling between CPU and NI is desirable, but this also limits the use. In figure 1.18 the various possible locations for NIs are depicted.

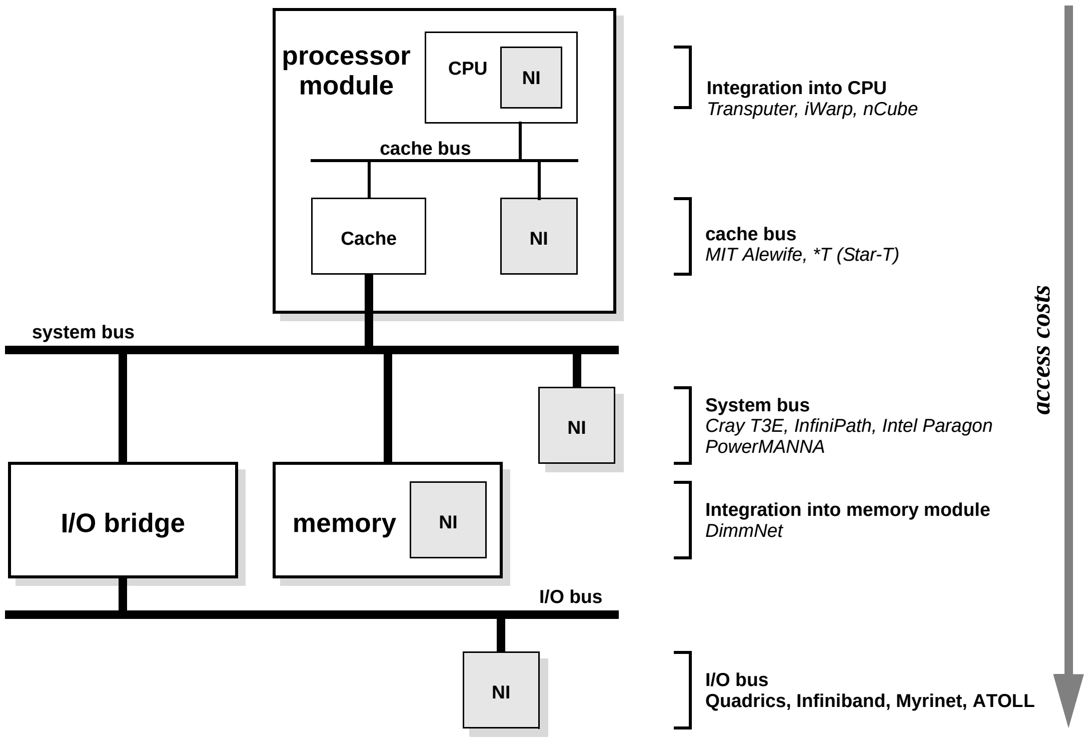


Figure 1.18 Network Interface (NI) locations

An integration of the NI into the CPU is the best solution for highest performance and close coupling between computation and communication. The more intermediate components are in between the NI and the CPU, the looser the coupling gets. Each intermediate module increases the access costs and latency.

The drawback of locations with closer coupling is that the interfaces to adjacent modules are likely not standardized or even unpublished, that a complete redesign of the housing module is required and that the NI is constrained to this special system. The most unconstrained location for an NI is the I/O sub-system, where standardized I/O interfaces allow a wide use. But in particular the I/O bridge introduces significant latency and prevents cache coherent devices.

1.3.2 User-Level Communication

If O/S involvement is required for communication the latency increases remarkable. Not only because of the system calls, furthermore the message is typically copied from user-level into system-level. In figure 1.19 the overview of communication with O/S involvement is depicted.

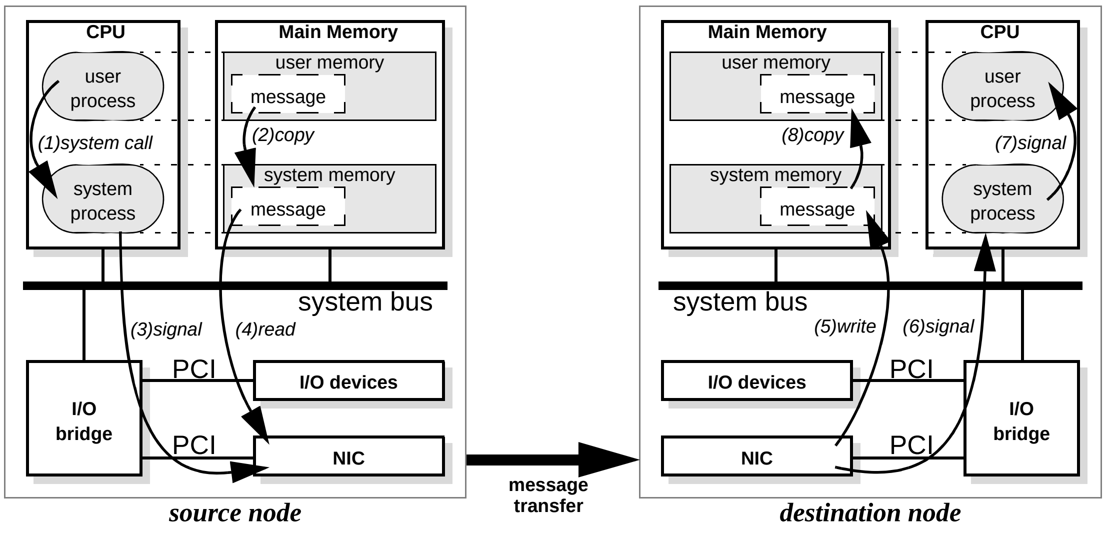


Figure 1.19 Non User-Level Communication

The advantages of this approach are that the O/S can supervise the accesses from user processes to the device and ensure that no hardware modules are disabled due to prohibited use. Furthermore the O/S can multiplex the access from several processes to one device in a time-sliced manner. This allows to use one resource by multiple clients.

But the disadvantages lead to the development of *User-Level Communication* [42]. Here the O/S is not involved for all accesses beside configuration and management. The latter ones require certain privileges which are usually only granted to trusted processes from system-level. But these configuration and management accesses are not or very seldom required during normal operation. All other accesses bypass the O/S and no overhead due to system calls arises here. The user process can directly access the device.

The O/S can now no longer supervise the user process, hence this functionality must be shifted from O/S level to device level. This increases the complexity of the device. Furthermore only one process can open the device for access, simultaneous access from multiple processes is not possible because now the multiplexing functionality of the O/S is missing.

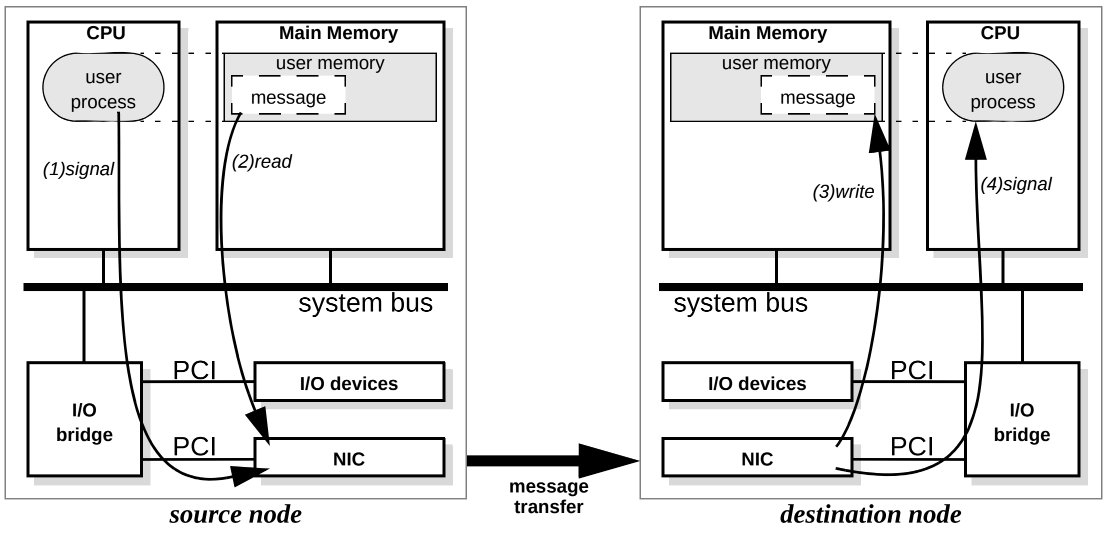


Figure 1.20 User-Level Communication

In order to allow simultaneous access from multiple processes the multiplexing functionality can also be shifted to device level. One of the goals of this work here is to develop sophisticated and efficient methods for multiple simultaneous accesses.

1.3.3 I/O interface

One of the most important limitations of an I/O device is the I/O interconnect. Access to the main system is only possible using this I/O interface. A typical I/O protocol differs from the main system protocol, hence intermediate bridges are required for protocol conversion. These bridges introduce additional latency for all accesses. Furthermore an I/O protocol is usually neither cache coherent nor providing support for virtual addresses.

All these limitations of the I/O interface result in a looser coupling of device and main system, in particular the CPU(s) and the main memory.

In spite of all these facts and disadvantages above, the I/O sub-system is typically the only available standardized interface in a system. For an unconstrained use of the device in various systems the I/O location is still the best solution. But in particular an NI as a high performance device with large performance requirements from the applications can benefit a lot from closer coupling to the main system.

1.4 THE ATOLL NETWORK

The *ATomic Low Latency* (ATOLL) interconnection network was developed by the Computer Architecture Group of the University of Mannheim as a research project. The results of the ATOLL project are summarized in a performance evaluation [39], including in particular the achieved latency and bandwidth.

The success of ATOLL lead to a new research project, of which the work presented here is part of. The insights gained with ATOLL resulted in some ideas for improvements, which are combined with new ideas and presented here. In order to provide a basic understanding of ATOLL's functionality it is now shortly presented.

1.4.1 Introduction

ATOLL is a complete network on a chip. This chip is connected over PCI-X to the host system. It provides four links towards the network side and four network interfaces towards the host side. These elements are connected using an 8x8 fully-pipelined synchronous crossbar. Figure 1.21 shows the top-level block diagram of an ATOLL card in a computing node.

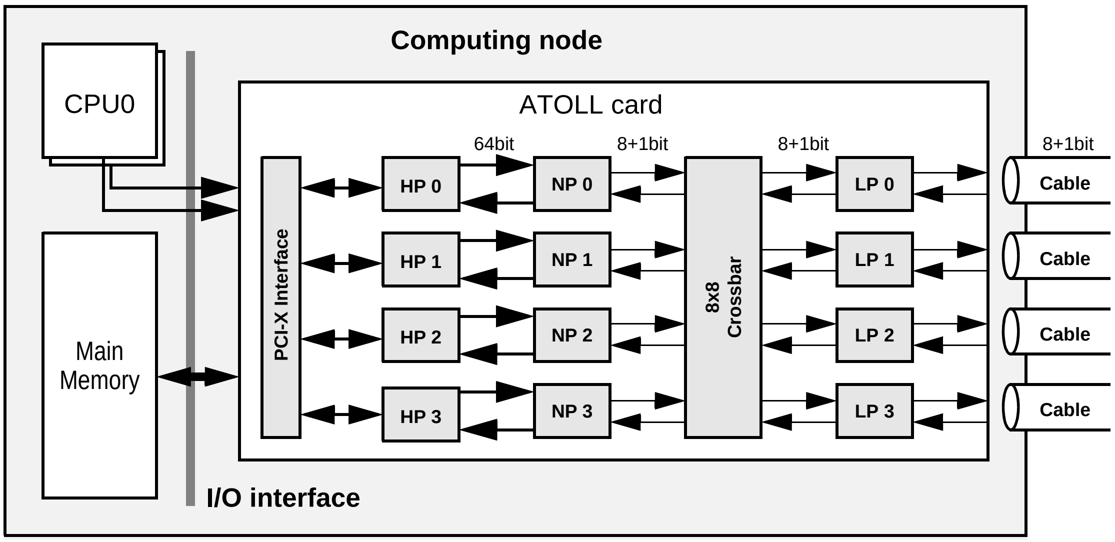


Figure 1.21 ATOLL top-level block diagram

Each of the four host ports provides an independent network device for the host system. PCI-X is used as interface towards host side. The architecture and design decisions for the host ports will be analysed in the next section in more detail. The network ports convert the data format of the host ports (64bit wide) to the ATOLL network protocol (8bit with one control bit). Control characters are added and the packet is divided into frames. The CRC is calculated for outgoing packets. The crossbar connects the network ports to the link ports. Eight input ports (*InPort*) can be switched without blocking to eight output ports (*OutPort*). The link ports contain buffers to store flits of packets. For packets coming in from the network side the CRC is checked. Retransmission of faulty flits ensures faultless transmissions. Based on the fill level of the buffers flow control characters are sent out. Flow control is based on a Stop/Continue scheme.

1.4.2 Topology and routing

In a typical ATOLL network all nodes are equipped with one ATOLL NIC card[1], which results in a node degree of four. Hence direct topologies with a dimension of two are most suitable, for instance a 2D mesh or 2D torus (shown in Figure 1.22). Another example topology is a 4D Hypercube. Each node in a graph represents a host system equipped with an ATOLL NIC. There are no centralized switching resources in the network, instead the required switching resources are distributed over all nodes. One advantage of distributed switching is the scalability. The network can be scaled by just adding more nodes to the network. Each node already houses the required switching capability. Scaling topologies with dedicated switches requires additional switches.

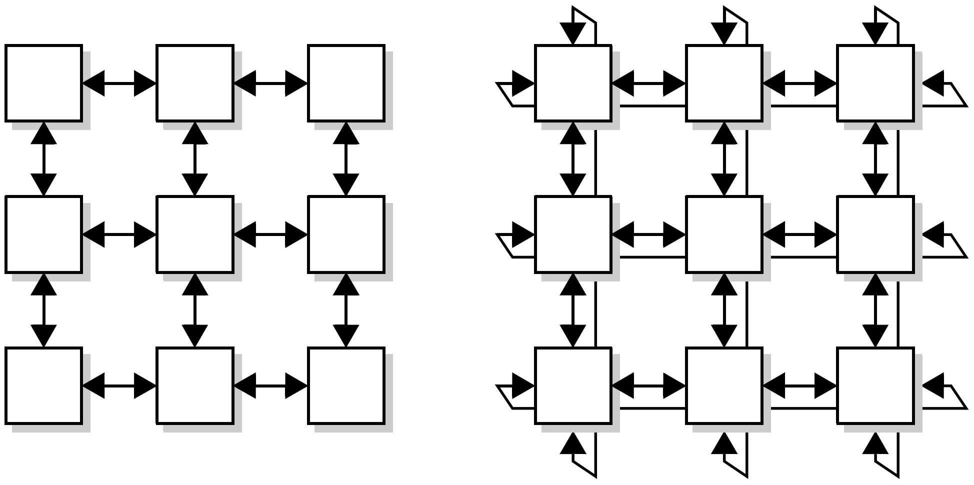


Figure 1.22 ATOLL example topologies

Because the network diameter raises with network size, a major aspect regarding the scalability of the system is the hop latency. The hop latency is the time required to forward a packet. The packet comes in over a link port and is switched by the crossbar to come out

1. It is possible to have more than one card per node, then one of the four links is used to connect the cards. The node degree increments accordingly and allows topologies with higher dimensions to be built up.

of another link port. The time also includes the routing interpretation and arbitration of an OutPort of the crossbar. ATOLL achieves a hop latency of 30 cycles, which is about 90ns for the maximal operation frequency of 330 MHz. This low latency is mainly achieved by applying source-path routing and an optimized crossbar design with a fall-through latency of only 3 cycles. Most of the remaining cycles of the hop latency are used for synchronization between the different clock domains.

The ATOLL network uses source-path routing instead of table-based routing for several reasons. The most important one is already mentioned, the impact on hop latency. The pre-calculated route simplifies the design of the router, no table look-ups are necessary. Additionally no memory for data structures is required, while for larger networks the routing tables can become a significant size and typically result in a high demand for on-device memory.

In the ATOLL network the ports of the crossbar are serially numbered from 0 to 7. Ports 0 to 3 are connected to the four host ports and the ports 4 to 7 to the four link ports. A routing string consists of a series of routing characters. Each character contains a port number and a parity bit. Each hop the first character is consumed for interpretation and removed from the routing string. The routing string shrinks with each hop. Upon arrival in a host port the remaining route is discarded. A parity bit per character is used instead of a CRC protection, so no re-calculation of the CRC is required. Advanced techniques like multicast or adaptive routing are not possible. Software layers are responsible to perform these tasks.

1.4.3 Impact of distributed integrated switches

In opposition to many other networks, the ATOLL network integrates the switch into the NIC. This close coupling allows manifold control, service and management functions [43]. Management layers can directly access the crossbar over status and control registers, while centralized switches can only be managed using packets sent over the network from a controlling host to the switch. In the case here, the notification of events from switch to management layer and resulting actions can be performed much faster.

Deadlock condition for InPort i and j (i!=j) and OutPort o

req(i,o) = request from InPort i to OutPort o

gnt(i,o) = grant from OutPort o to InPort i

stop(i,o) = stop from OutPort o to InPort i

dl_cond(o)=req(i,o) & gnt(i,o) & stop(i,o) & req(j,o)

Figure 1.23 Necessary deadlock condition for ATOLL

One resulting service is the deadlock recovery scheme of ATOLL. The arbitration logic of each crossbar is able to detect a certain deadlock condition which is necessary for a deadlock. *Figure 1.23* shows this condition. Basically a deadlock can only occur if one InPort is requesting an OutPort and another InPort has been granted this OutPort and the flow control has stopped this connection. Only when this condition becomes true this switch can be part of one or more deadlocks. Without this condition it is guaranteed that this switch is not part of any deadlock.

If a deadlock condition is detected a counter loaded with a pre-selectable value starts counting down. As soon as it equals zero an interrupt is thrown to notify the management process of the host system. This process can either communicate over an auxiliary network (like Ethernet) with other involved nodes to check if there is really a deadlock, or skip this step and immediately retrieve the packet from the network. In the first optional step the protocol ensures that there is really a deadlock, and furthermore that only one packet is retrieved from this deadlock, which is sufficient. This deadlock notification and recovery scheme is only feasible with a very short reaction time after detecting the deadlock condition. Otherwise deadlocks result in congestion and generally raise the deadlock possibility, decreasing the network's overall performance.

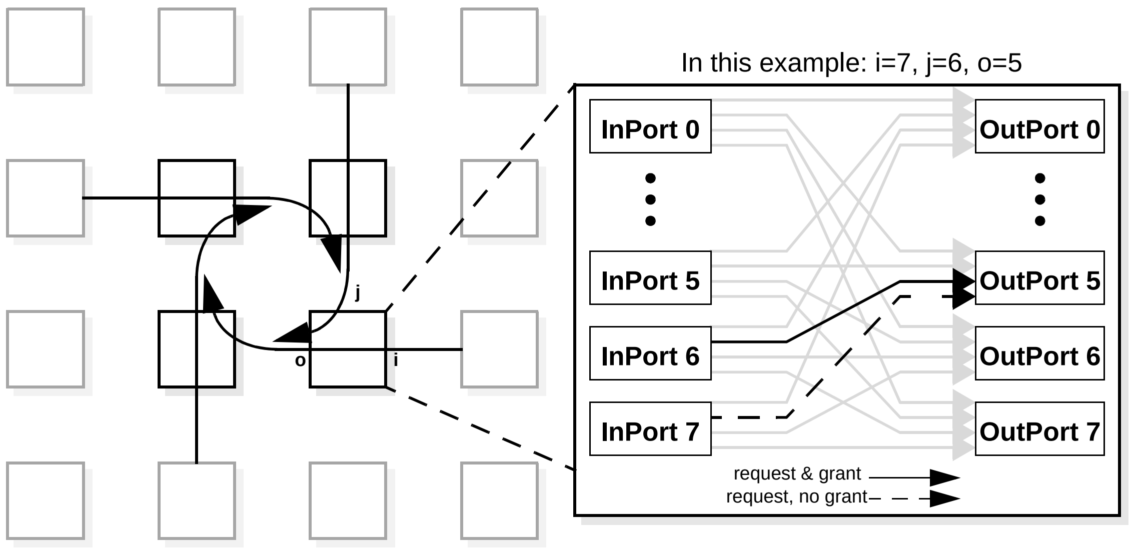


Figure 1.24 Simple deadlock example for a 2D mesh

Figure 1.24 shows a simple deadlock situation for a mesh topology. Four nodes are here directly involved in the deadlock circle. In each node one OutPort is granted and stopped by flow control, and another InPort requests the same OutPort. A snapshot of the crossbar switching is shown on the right. To solve the deadlock, it is sufficient that one packet of participating in the deadlock is retrieved from the network. The packet can be retrieved from the network by overwriting the current request and re-directing the packet to the host port associated with the management process (in this example: 0). Then all three remaining packets can proceed. The retrieved packet can be immediately re-injected. If more than one packet, e.g. all packets, are removed from the network the deadlock is also solved, but with higher overhead. To not split up a packet into fragments only packets without grants are retrieved.

Especially fault-tolerance services benefit from the close coupling between switch and management software. Beside deadlock recovery services the other main eventualities - link and node failure - can be recovered. In the case of a node failure data loss cannot be avoided, so an additional end-to-end acknowledge protocol is required to recover from this failure.

1.4.4 Network interface

Each ATOLL card not only houses one interface to the host side, instead four replicated host ports are available. Each host port can be mapped by an user process for direct access using User-Level Communication. With the four replicated ports ATOLL is perfectly suited for SMP nodes with multiple CPUs (up to four). On each of the CPUs one process can be running and accessing ATOLL.

The communication function offered by ATOLL is a Send/Receive scheme, requiring to copy the payload into dedicated buffers on both the source and target side. Two methods are available, the most appropriate is dependent of the payload size.

The processes can inject packets using several *Programmed I/O (PIO)* writes to special registers which are part of each host port. The host port combines the data provided with the write accesses into a packet. Packet reception is based on PIO reads. This method is only suitable for small messages (for instance up to 512 byte).

For larger messages a Direct Memory Access (DMA) scheme is available, which is based on a set of data structures in main memory. To send a packet, a send descriptor is generated describing the packet. This descriptor is stored in a descriptor queue. The payload is only included as a reference in the descriptor. It is stored in a send data region. On the receive side two similar working data structures are used, again located in main memory. The receiving host port stores the payload in the receive data region and generates a receive descriptor where the packet is described.

Because only main memory is used to store the data structures, on-device memory is not required. This reduces the total costs of an ATOLL card significantly.

1.4.5 Fault tolerance

Fault tolerance in the context of an interconnection network like ATOLL means that the transfer of packets is reliable regarding several conditions:

1. A packet sent over the network is not altered, it reaches it’s destination unmodified. In other terms, the sender can discard the payload after sending the packet out.
2. Once injected into the network, it is ensured that the packet reaches it’s destination within finite time.

This leads to the following requirements for the ATOLL network. Faults based on transmission errors, dead- and livelocks, link faults and node faults must be either avoided or tolerated by appropriate hardware or software support.

To preserve performance and retain cost-optimization certain trade-offs are made. For instance, transmissions on-chip and over the peripheral interconnect are expected to be safe. If the source or target of a transmission is no longer available the transfer can apparently not be ensured. Hence this case is not covered by ATOLL's fault tolerance.

To avoid transmission errors, each flit is protected with a CRC. A CRC failure results in a re-transmission of the faulty flit. Livelocks can only occur when adaptive routing is applied, which is not the case for the ATOLL network.

The deadlock handling is explained in detail in section 1.4.3 on page 25. It is based on a deadlock recovery scheme, capitalizing the integrated crossbar for a close coupling of switch resource and management process. Another possibility is to choose a routing-topology combination which is deadlock-free. Then deadlocks are avoided.

Livelocks only occur when adaptive routing is applied, which is not the case here. Hence this fault case must not be considered.

An in-depth explanation of the handling of link and node faults can be found in [43]. In a condensed form they are based on a link detection and additionally a software acknowledge protocol for the node failure case. One of the insights gained during the ATOLL project is that hardware support for end-to-end acknowledges significantly reduces the software overhead. Hence this is one of the goals of the work here.

1.5 Improvement Proposals

In order to summarize the introduction and to show what requirements for a next-generation high-performance network interface architecture exist, this sub-chapter provides an overview of the key facts shown above. This starts with the lessons learned during the ATOLL project, continues with required communication functions and finally leads to the network interface architecture itself. Finally, the goals of this work are shortly summarized.

1.5.1 Lessons learned from ATOLL

The biggest and important insight from the ATOLL project is that a direct network is an excellent solution. It’s distributed switching resources are scalable and provide redundant paths which increase the fault tolerance.

ATOLL allows User-Level Communication for up to four processes by replicating it’s Host Ports. Processes communicate to the device using queues in main memory which reduces the required on-device memory to a minimum. These queues are very efficient and reduce the overall costs of an ATOLL card. User-Level Communication is inevitable for low latency message passing. Missing is support for more than four processes. Furthermore the replication results in a partitioning of the resources, preventing a dynamic utilization of resources.

Regarding communication operations an RMA scheme is missing. Furthermore the available set of operations can be made more versatile in order to optimize certain operations for different message sizes. The PIO mode of ATOLL is not optimal, further improvements should significantly reduce the start-up latency.

Regarding fault tolerance hardware end-to-end acknowledge support is missing, which is currently achieved in software layers. Including this in hardware would further reduce software overhead.

Last, if software layers rely on a sequence number this must currently be included in the message tag. This limits the available tag size for user applications. Hence the idea is to introduce a dedicated API-tag besides the user tag.

1.5.2 Sophisticated communication methods

A larger set of communication operations is already slightly touched. To concrete this idea, not only two communication methods (for instance based on PIO and DMA) should be supported. In particular support for fine grain communication is missing. For this a lowest latency communication method is required, whose small overhead allows to efficiently transfer even smallest data structures. Beside this, for (almost) each message size highly optimized operations should be supported.

Furthermore the extensions of MPI2 should be supported, which is in particular true for the RMA operations. The window scheme of MPI-2 can be improved by a more generic one. This leads to the development of the idea to combine synchronization and communication for RMA operations, which otherwise cannot provide synchronization. RMA operations also benefit a lot from support for virtual addressing in the network interface, hence a suitable mechanism is desirable.

It should be possible to ensure the ordered delivery of messages, which reduces the required overhead in software messaging layers. Because not all applications require this, it should be selectable.

Summarized, various communication functions should be supported, each optimized for certain circumstances. This set is comparable to the manifold instruction set of modern *Complex Instruction Set Computer (CISC)* architectures.

1.5.3 Network interface architecture

Improving the network interface architecture is the major goal of this work. In particular in multi process environments the architecture should support unrestricted and dynamic utilization of it’s resources. Only then the parallelism offered by such a system can be efficiently exploited. The exploitation also requires simultaneous access from multiple user processes to the network interface, which is achieved by the virtualization. The virtualization must allow access from user-level and without O/S involvement. The virtualization is also suitable for VM environments with it’s concurrently running O/S guests.

For a virtualized device a large amount of data structures is required. The most cost efficient memory resource of a system is the main memory, hence it is desired to use this for the data structures. This allows a most scalable and cost efficient design.

A memory-less design results in a large amount of accesses to main memory. Caching structures can diminish this effect. But the efficiency of all accesses over the I/O interface must be kept in mind, which is in particular true for the simultaneous accesses from user process to device. Theses ones must be highly sophisticated to use the limited I/O resource as efficiently as possible.

1.5.4 Goal summary

Communication Instruction Set. Development of a set of communication and synchronization instructions providing support for all kinds of communication. Targeted is a message passing system with explicit communication and synchronization. The communication set includes ultra low latency communication (*fine grain communication*), RMA with window protection and sophisticated send/receive functions optimized for different message sizes. The in-order delivery should be possible but due to resulting performance restrictions selectable.

Network Interface Architecture. Development of a dedicated and specialized architecture for a high-speed network interface. The architecture should be suitable for as many purposes as possible and support the communication instruction set above in an optimal way. Other key features are a memory-less design in combination with an on-device memory hierarchy and support for fine grain communication by ultra low latency message passing. Generally spoken, the goal is to combine the versatile usability of Ethernet with the performance of high speed interconnection networks.

Utilization of Resources. Unrestricted use of the hardware resources for the process clients. This implies a high utilization of the hardware resources independent of the process client count. One process should be able to use all available resources, while a large number of simultaneously accessing processes share the resources uniformly.

Virtualization. The developed architecture should be suitable for simple system architectures with only one CPU, as well as high performance cluster nodes with several multi-core and possibly multi-threaded CPUs. The recently emerging virtualization of complete computing systems using Virtual Machines should be backed by the virtualization of the network interface in hardware. The Virtual Machine Monitor should be bypassed for all time critical operations. Uncritical operations like management functions can include the Virtual Machine Monitor. Optimally, from the architecture’s point of view it should be independent if there are several operating systems running or only one and to which O/S a process belongs. Summarized, the architecture should allow multiple processes to simultaneously access the device without the involvement of any software layer.

CHAPTER 2 COMMUNICATION AND SYNCHRONIZATION

This chapter starts with an introduction to the basic communication and synchronization schemes, followed by a classification in order to allow an improved understanding. The advantages and drawbacks of in-order and out-of-order delivery are analyzed.

This is followed by the integration of the network interface architecture in the interconnection network, showing the requirements and constraints for this work. This new interconnection network is a new research project based on the ATOLL network.

After these introductions the work flow for the supported communication schemes is analyzed. This leads to the development of several types of functional units, which process the steps required for communication and synchronization. Due to the distributed work processing, the functional units are located both on the source and the destination node.

Finally a set of communication methods is developed. For all payload sizes sophisticated methods exist to transfer the data from source to destination. Beside traditional Send/Receive schemes, RMA operations are also supported. A method to combine RMA operations with synchronization is shown.

The insights gained in this chapter allow a comprehensive understanding of the requirements for the network interface architecture, which is developed in the next chapter.

2.1 INTRODUCTION

Goal of this chapter is to provide a short introduction in communication and synchronization schemes, including their requirements. The in-order and out-of-order delivery is examined together with the impact of acknowledges.

2.1.1 Messages and packets

From an application's point of view the communication and synchronization is based on the transfer of *messages* between source and destination. A message is composed of it's payload and a tag to identify it's content. Applications rely on the help of libraries and APIs to send and receive messages or perform more sophisticated operations.

From an network's point of view, a message can be segmented into multiple *packets* due to maximum transfer units. Beside the message tag, the network (or API) requires more information about the packet. This includes the source identification, a destination identification or a route to the destination, sequence numbers and a packet type.

The work here starts with the API level, hence in the following the term packet is used for all elements transferred in the network. If messages are segmented into multiple packets or not is not relevant for the work here. This is subject to upper higher software instances.

2.1.2 Classification of Communication and Synchronization

Communication can be classified based on two schemes. The first classification takes into account the number of communication partners involved in the communication. The result is a classification into *one-, two- or n-sided communication*. N-sided communication only applies for collective operations, which are beyond the scope of this work. As communication partners only user processes are considered. Thus if at the destination only the network device is part of a communication and not the user process, this is considered as one-sided communication.

The second classification is based on the number of copies required for packet transmission. This only includes copies made by software processes. Hardware copies are not considered here. Due to the limitation to up to two communication partners, the communication can be classified into *zero-, one- and two copy schemes. Table 2.1* classifies the communication methods into these two schemes.

Synchronization only applies if more than one side is involved in the communication. Hence synchronization is limited to the two-sided and *collective communication*. A typical example for a multiple-sided synchronization is a *barrier*. Because collective communication is not covered here, this case is not further considered. Handshake protocols based on two-sided communication can be used for synchronization.

Unmodified one-sided communication is not suitable for synchronization, because the destination (process) of the operation is not involved in the communication. Synchronization can be included in one-sided communication by notifying the destination of an RMA operation.

Table 2.1: Classification of Communication

Required copies	two-sided communication	one-sided communication
two-copy	Traditional Send/Receive scheme, where the payload is copied into dedicated buffers on both source and destination side.	Basically not used. Some designs implement RMA operations based on the help of the O/S. But this prevents User-Level Communication.
one-copy	A Send/Receive scheme, where one copy can be omitted. This applies for instance when a *posted receive* is used.	
zero-copy	Send/Receive scheme, where the payload is directly fetched from and written to user space.	An RMA Put or Get operation, where the payload is directly fetched from and written to user space.

2.1.3 Two-sided communication methods

These methods include both the source and the destination *actively* in the communication. To be more concrete, the user process on the source node and on the destination node are involved in the communication.

The typical communication scheme here is based on *Send* and *Receive*. The source process sends a packet out and the destination receives the packet. Send/Receive methods can be separated into *blocking* and *non-blocking* operations.

A *non-blocking send* operation only inserts a send request in a work queue and immediately returns, independent of the current transmission status. The status of the send must be checked later. Dependent on the implementation, a *blocking send* either returns when the packet has left the source buffer, arrived at the destination buffer or at the destination process.

A *blocking receive* only returns if a new packet is present. If no packet is present, it waits until one arrives before passing control back to the calling process. A *non-blocking receive* returns immediately, independent if in the specified buffer a new packet is stored or not. Again, a later check must be done to know the status of a non-blocking receive.

Furthermore a Send/Receive scheme can be *synchronous* or *asynchronous*. Asynchronous schemes immediately send out the packet without ensuring that enough buffer space is available at the destination. In opposite, synchronous schemes rely on handshake protocols prior to the data transfer to ensure this. Smaller payloads are typically transferred using asynchronous schemes, while larger ones rely on synchronous transmission.

2.1.4 One-sided communication methods

One-sided communication methods only require the source to participate actively in the communication. Typical RMA operations are one-sided, because they read or write a remote memory location without the involvement of the user process at the destination.

RMA operations access directly user memory regions described by *Virtual Addresses (VA)*. Network interfaces located behind an I/O interface cannot use VA, only *Physical Addresses (PA)*.

Several solutions exist to overcome this situation. In [44] a survey of these methods is presented. Some methods are based on handshake protocols prior to the RMA operation to pin down the referenced memory regions and translate the addresses [45]. Only after this handshake the RMA operation can take place, using the previously calculated PAs. The most sophisticated method uses an on-device *Translation Look-aside Buffer (TLB)* [46] to translate VAs into PAs. This reduces the overhead by eliminating the handshake protocol.

A special case is a remote atomic operations. Here an atomic operation is initiated at the source on a remote address. These operations rely on support for atomic operations of the I/O interface. Otherwise the target address is not locked and simultaneously the remote CPU can access this location. This may result in unwanted behavior. A possible solution to overcome this situation is to use only remote atomic operations on shared locations, even if this location is local. Then all accesses to the semaphore are performed by the network interface, which can ensure that no atomic operations execute simultaneously.

2.1.5 Acknowledges

An end-to-end acknowledge is usually required to inform the source that a packet has reached it’s destination. This can optionally even include that the user application at the destination has received the payload.

Even for reliable interconnects, severe failures (for instance of a complete node) can result in data loss. In theses cases end-to-end acknowledges are required for reliable transmission of data.

The acknowledge scheme used here is an end-to-end acknowledge. It does not include involvement of the user application at the destination, it is only ensured that the packet has correctly arrived in the destination packet buffer. The use of the acknowledge is selectable to support different requirements for fault tolerance.

2.1.6 In-order and out-of-order delivery

Some applications require in-order delivery of messages, other not. If an application relies on in-order delivery and the network cannot ensure this, the incoming packets must be reordered in messaging layers to deliver the messages in-order to the applications. This can result in additional buffering delays.

There are several reasons that packets which are injected in-order arrive out-of-order at their destination [17]. Adaptive routing allows each packet to take another route to the destination. Consecutive packets can then use separate routes to the target and ordering constraints inside a switch element no longer apply. Some methods combine adaptive routing with in-order delivery, but this introduces a certain overhead [47]. Another reason are virtual channels or lanes. Even for deterministic routing this allows consecutive packets to overtake each other, because the arbiter in the switch element might not recognize the order of the packets waiting for a grant.

Without in-order delivery guaranteed by the network, the packets must include sequence numbers. Only then the destination is able to re-sort the incoming packets. But the reassemble overhead at destination degrades the overall performance.

If messages are segmented into multiple packets, the ordering of these packets is also important. For out-of-order delivery of packets it is then not sufficient to check for the last packet in a series, because other packets might still be outstanding. Furthermore the incoming packets must be reassembled to provide the message correctly to the application.

If the network (including the network interface) ensures the in-order delivery, several advantages and drawbacks result. Reordering in software layers is not required. For messages segmented into multiple packets it is sufficient to wait for the arrival of the last packet. The ordering ensures that the other packets have already arrived and no reassembly must be done. The drawbacks are that adaptive routing is not possible. Furthermore if all packets are processed in order, the parallelism is reduced. This can be improved by the introduction of an *ordering key*. This key limits the ordering to certain cases, for instance only if the packets have the same destination. If applications do not require in-order delivery but the network ensures this, the ordering overhead is unnecessary.

The other case to be considered is if the network delivers out-of-order but the application rely on in-order delivery. Then software layers must re-sort the incoming packets. For messages segmented into multiple packets each packet must be checked at the destination, before the message can be reassembled. On the other hand, adaptive routing is possible and ordering constraints decreasing the network’s overall performance are removed. An increased exploitation of parallelism is possible because no packet ordering constraints exist. Last, no hardware support for ordering is required.

Due to unpredictable requirements of applications regarding ordering, support for both in order and out-of-order delivery must be included in the network interface architecture. Applications can then choose if they want their messages to be delivered in-order or not.

2.2 INTEGRATION INTO THE INTERCONNECTION NETWORK

The network interface architecture developed with results of this work will be used in an interconnection network optimized for cluster environments. A short introduction into the basic functionality of this IN is required for a comprehensive understanding of the requirements and constraints for the network interface architecture and the supported communication and synchronization functionality.

This sub-chapter shortly introduces this IN. Focus is set on the interface to the network, while the detailed functionality of the other components of the network, like switches, fault tolerance or link protocol is omitted.

2.2.1 Overview

The new IN is based on the previous research project ATOLL (see sub-chapter *“1.4 The ATOLL Network” on page 23*). Again it is a direct network, integrating the switch into the network interface. The network interface provides 6 links towards the network. This allows to build up three-dimensional mesh- or torus-based topologies.

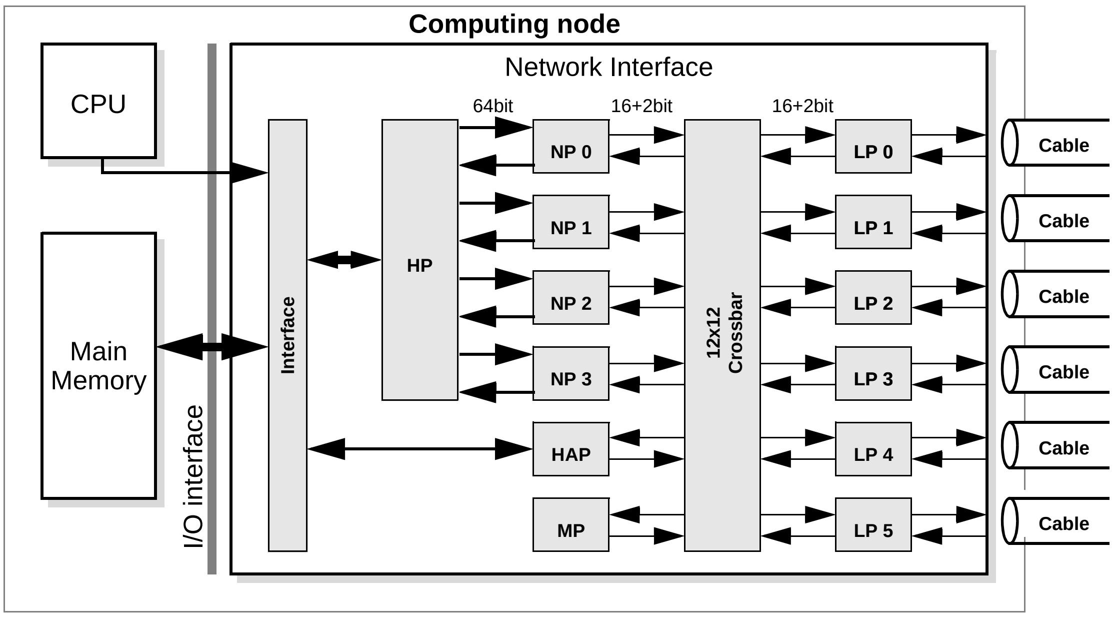


Figure 2.1 Top-level block diagram of the Interconnection Network

This work completely focuses on the development of a suitable architecture for the *Host Port (HP)*. The HP is the interface between network and host system, responsible for the injection and retrieval of packets. It is connected with up to four *Network Ports (NP)* towards the crossbar of the network interface.

The integrated crossbar in each network interface builds up the required switching resources for this direct network. The crossbars are interconnected with six links, allowing any kind of topology up to a node degree of six. The two other modules connected to the crossbar are the *Multicast Port (MP)* and the *High Availability Port (HAP)*. The task of the MP is to replicate multicast packets and thereby building up a multicast tree. The HAP is responsible to retrieve faulty packets from the network, ensuring fault tolerance.

The links together with the crossbar are considered reliable, i.e. it is guaranteed that an injected packet will reach it’s destination. This is ensured by a reliable link transmission based on a link-level acknowledge protocol and a credit-based flow control.

The network can be configured to keep the ordering of packets by including a hash value of the destination in each packet’s header. This forces the arbiter of each crossbar to forward incoming packets only in order. The links always transmit packets in order, hence no special scheme is required here.

2.2.2 Routing

A source-path routing scheme is used for this interconnect. Each packet contains a routing string. Upon each hop from source to destination the crossbar arbiter interprets the first element of the routing string. Here the direction (or link port number) for the next hop is directly included. Table lookups are not necessary.

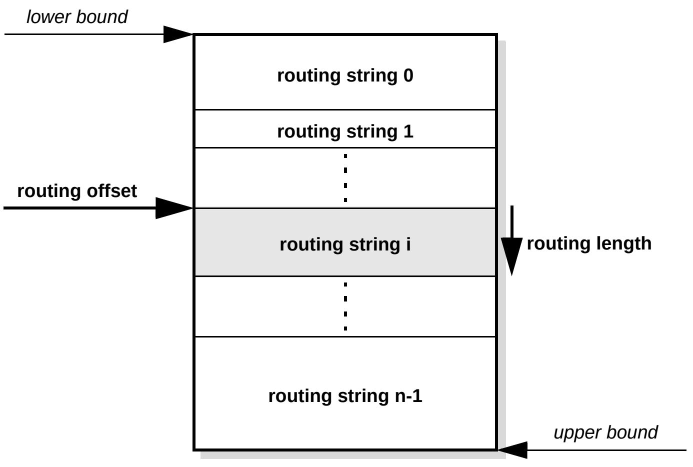


Figure 2.2 Routing space

A run-length encoding scheme allows to compress the size of the routing string if multiple consecutive routing elements contain the same direction. In this delta routing scheme each element contains beside the direction also the number of hops for this direction.

Source-path routing requires the source of a packet to provide the routing string. A *routing space* is used to store all the pre-calculated routing strings. For each destination the corresponding routing string can be determined using an offset (*routing offset*) into this table together with a length information (*routing length*). These two parameters are included in the communication operation. The use of routing offset and length also allows to vary the length of each routing entry. The variable entry size also lead to the name *routing space* and not routing table.

If the packet has to be transmitted both from source to destination and vice versa[1], the routing string is composed of two parts, a forward and a return path. The two parts are separated using a special routing element. Hence a routing lookup is only once necessary for packet transmission, even if it has to be transmitted back again.

2.2.3 Packets

Beside the routing information the packets consist of a command frame and the payload. The command frame is used to distinguish different communication operations and contains parameters for the communication. This includes the source identification, packet tag, sequence number, a remote address for RMA operations or acknowledge requests.

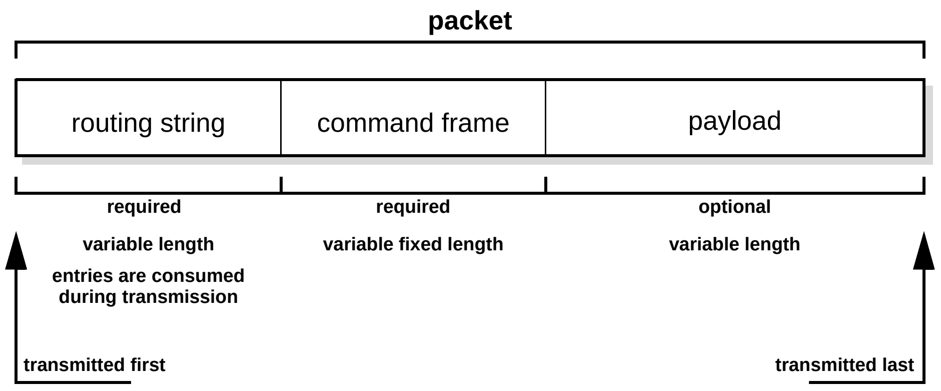


Figure 2.3 Packet format

In a packet the routing frame is always first, because this part must be interpreted by the intermediate hops. At the destination the command frame is required prior to the payload to identify the communication operation. Hence the payload is always last.

1. For instance an RMA Get operation includes a request packet sent to the destination, and a response packet containing the result sent back to the source (split-phase transaction). Another example is an end-to-end acknowledge scheme.

While the routing string is an entry in the pre-calculated routing space and the payload is provided by the user application, the command frame must be generated by the Host Port.

2.2.4 Host Port

The communication and synchronization mechanisms together with the network interface architecture developed in this work is integrated in the *Host Port*. It is the interface between application and network.

A communication set is developed in this chapter supporting both one- and two-sided communication schemes. A large set of communication operations is available, allowing to choose the most suitable operation for a payload size.

The network interface architecture is the main topic of the next chapter. Here architectures are analyzed in order to find a suitable one which exploits any available parallelism. Dedicated hardware modules are developed to perform the communication operations. One key component of this architecture is the virtualization of the Host Port, allowing almost any number of applications (or processes in general) to access the Host Port simultaneously. While in ATOLL several replicated Host Ports are necessary to allow simultaneous access from several processes the device, the virtualization supports almost any number of processes. Now one single Host Port is sufficient, which resources can be dynamically shared by all accessing processes without any partitioning.

The following work is not restricted to be used in the context of this interconnect. Due to this, the term Host Port is avoided and (network) device used instead. This also represents the process’s view of the Host Port.

2.3 COMMUNICATION ARCHITECTURE

In this sub-chapter the work flows for different communication schemes are analyzed, resulting in a communication architecture including several types of functional units on the origin and target node. Circular buffers are proposed as most suitable data structure for the queues between processes and device. Different notification methods for circular buffers are analyzed and for the different use cases the most appropriate are shown.

2.3.1 Work flow

Each communication operation[1] is initiated by an *origin node*. The destination or communication partner is called *target node*. The origin is always actively involved in a communication because here an application starts a communication. If a *two-sided communication* takes place the target participates also actively. But for a *one-sided communication* the target is only passively involved, because actions of the user application are not required. At the target the operation can be completely processed without involvement of user-level software layers.

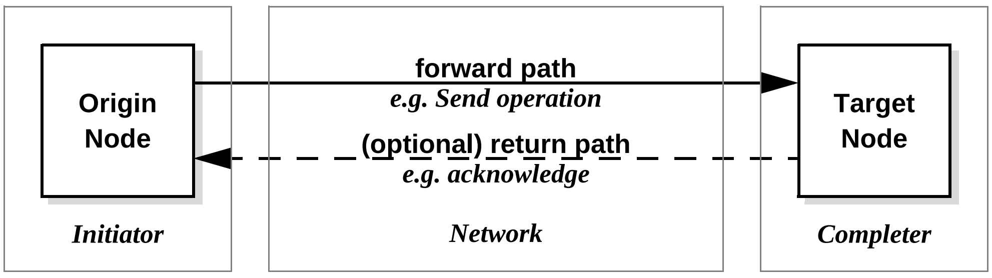


Figure 2.4 Two-staged work flow

In the simplest case the work flow is only *two-staged* (see figure 2.4). Here an operation is initiated by the origin node, hence called *Initiator.* The operation is sent over the network to the target node using the *forward path*. At the target it is completed. In the two-staged case the target is called *Completer.*

An optional *acknowledge* can inform the origin of the completion of the operation. The acknowledge is a positive in the case of a successful performed operation. If the operation

1. The last part of this sub-chapter will show that a communication operation like a Send, Receive, Put or Get operation is better described as a communication instruction. This improves the understanding from the hardware's point of view. But for now the name 'communication operation' is kept.

has failed at the target a negative acknowledge is sent back. The acknowledge uses the *return path* back to the origin.

One-sided communication can require a response in the work flow. This leads to a *three-staged work flow* (see figure 2.5). The three-staged work flow is never used for two-sided communication. For a three-staged work flow the operation is initiated at the origin and sent using the forward path to the target where a response is generated. Here the target acts as a *Responder*. The response is sent back over the network using the *return path*. The return path is not optional in the three-staged work flow. The origin then completes the operation. In the case here it acts both as an Initiator and a Completer.

The processing of all non-cumulative operations is always bounded to the origin and target node. The origin is always the Initiator of an operation. The target is either the Completer or the Responder. If the target acts as a Responder, always the Origin will finalize the operation as a Completer.

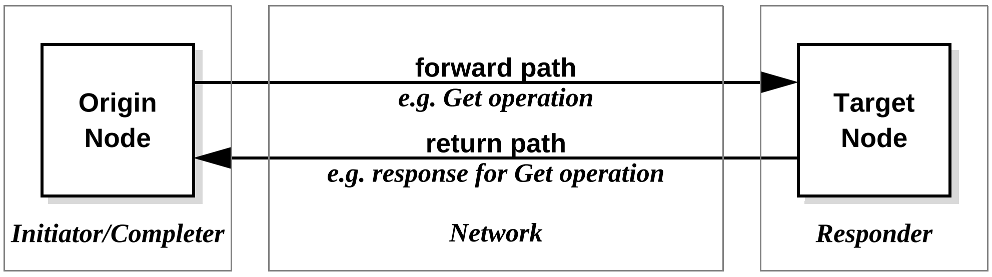


Figure 2.5 Three-staged work flow

In a three-staged work flow an acknowledge is unnecessary, because a response is already included in the work flow. The response informs the Initiator of the result of the processing at the target node.

The approach of separating the work flows in a two-staged and a three-staged helps to classify all supported operations. All two-sided operations use the two-staged work flow. Because an RMA Put operation does not require a response it is also two-staged. Only the RMA operations requiring a result, e.g. all kinds of Get operations, are three-staged.

2.3.2 Functional unit types

The work is processed by *Functional Units (FU)* which are located both on the origin and the target node. Corresponding to the nodes acting as Initiator, Responder or Completer, the units can be separated into three types. The *requester unit* is always part of an Initiator. It generates a request packet which is sent over the network to the target. Here either a *responder unit* or a *completer unit* receives the packet, depending on the type of work. Both unit types interpret the incoming request. The responder unit answers with a response sent back over the network, while the completer unit finalizes the work. Incoming responses are also processed by completer units, which is the case for the three-staged work flow. For the two-staged work flow the completer unit can optionally send an acknowledge back to the origin. Responses and thus responder units are only required for one-sided communication.

Figure 2.4 show in detail the two-staged work flow. On the origin node a process P_{Origin} initiates the work by issuing a work request to the requester unit. The requester unit processes the desired work. This results in a request packet sent out to the target node. All the information from the origin that is required to process the work request at the target side is included in this packet. With this packet, the receiving completer unit can further process the work. After finishing it, the completer unit delivers the work results to the corresponding process on the target (P_{Target}). Optionally an acknowledge can be sent back to the origin node to inform P_{Origin} that the work was successfully completed. In the case of a failure during the processing on the target side the acknowledge notifies the origin of the unsuccessful processing.

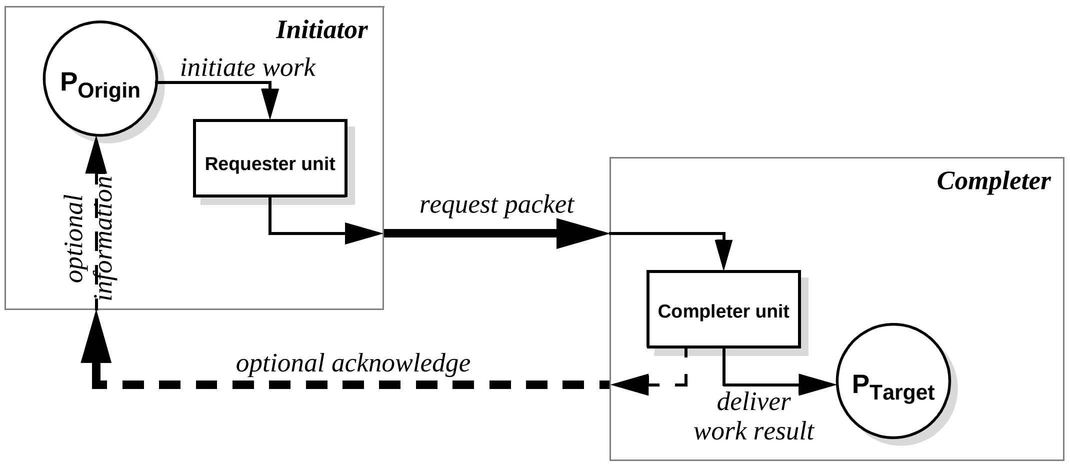


Figure 2.6 Functional unit types in the two-staged work flow

The processing of the two-staged work flow is extended to the three-staged in figure 2.5. This work flow always includes a response. In detail, the process P_{Origin} on the origin node initiates the work. The requester unit sends out a response which is received at the target side by a responder unit. With the information contained in the request packet and the local configuration this unit processes it's part of the work. At the end the response unit sends back a response packet containing the result of it's work. Optionally it can notify the process P_{Target} that an RMA operation[1] was processed. The response packet sent back is received at the origin node by a completer unit. Here the work is finally finished. At the end the completer unit delivers the result of the work to the process.

1. Note that the three-staged work flow only applies for some RMA operations. For RMA operations user-level processes are not involved in the processing.

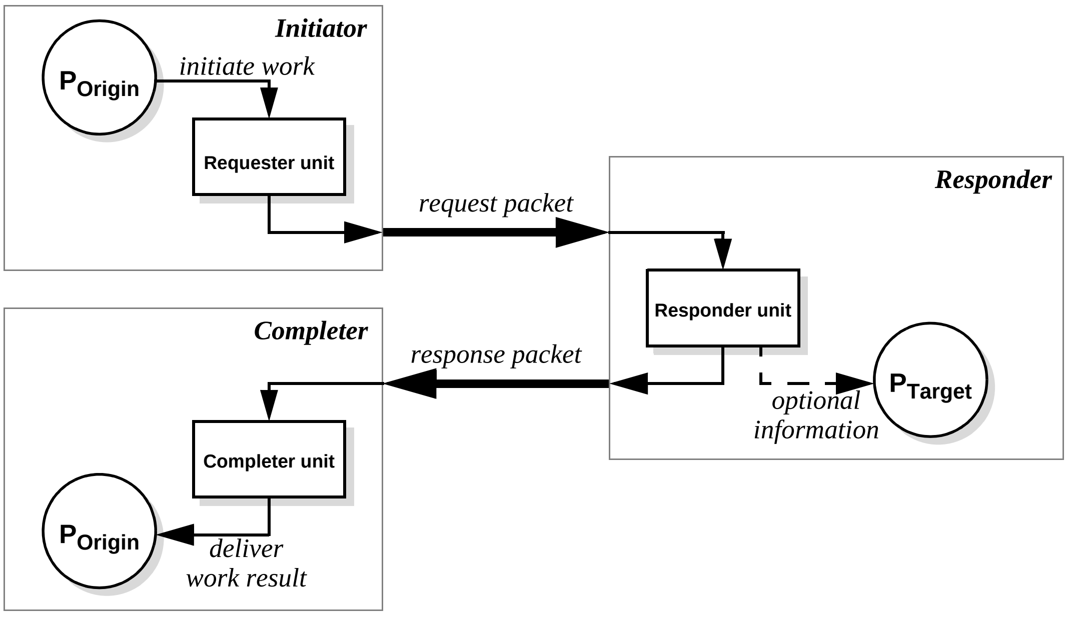


Figure 2.7 Functional unit types in the three-staged work flow

2.3.3 Send/Receive communication scheme

The supported Send/Receive scheme is a two-sided communication. Both origin and target actively participate in the communication. The origin by explicitly sending a message and the target by receiving the message. This allows to use this communication scheme also for synchronization. But communication without synchronization is also possible.

The Send/Receive scheme is a two-copy scheme. The payload is temporally stored in dedicated buffers on both origin and target side. This allows the network interface device to directly access the buffers using physical addresses. The drawback is that the additional copies introduce overhead. This scheme is most suitable for smaller messages. For large bulk transfers zero-copy schemes like RMA perform better.

The work flow for the Send/Receive scheme can be seen in figure 2.6. The work initialization matches the issue of a send request by P_{Origin} to the requester unit. The request packet sent over the network contains the payload of the message. The completer unit on the target node receives the packet and interprets the packet header. It recognizes the packet as a request for a send operation. It finds free buffer space to store the payload. After that it notifies the target process of a received message. This also includes a description where the received data can be found together with informations about sender, message tag and size.

The Send/Receive scheme is asynchronous, no handshake protocol prior to the payload transfer takes place. If no buffer is free at the target to store the packet, it will block in the network. Depending on the application’s requirements, upper software layers can easily implement a synchronous Send/Receive by first sending a short message notifying the target of a large data transfer. The target checks for an appropriate buffer and sends back a message. Only then the message with the large bulk of data is injected.

Fast operations. A Send operation can be initiated in several ways. Either the payload is included in the work request or the work request contains a reference to another data structure where the payload is located. The second solution introduces additional overhead because another data structure must be accessed. So it is only desirable if the payload does not fit in the limited size of a work request. Similar constraints apply for message receiving. If the payload can be included in the notification the lowest overhead is required. If not other data structures must be used and the notification includes a reference where to find the payload.

If the payload can be included in the work request or the notification, the operation is a *fast operation*. This distinguishes it from normal operations. The payload size determines the use of normal or fast operations.

If the network interface's location is in the I/O subsystem the costs for an additional memory access can be quite high and the use of fast operations significantly increase the overall performance. It is desirable to store as much payload in the work requests and notifications. Otherwise an unnecessary large work request or notification also wastes resources if their payload section is not enough utilized.

Notification. The receiving process can either poll for changes on the appropriate data structure for the notifications or register itself for an interrupt-based notification. The first method is time consuming and increases the load of the CPU. But it has the lowest notification latency and also allows User-Level Communication. The interrupt-based method cannot be combined with User-Level Communication because O/S involvement is required to process the interrupt. But using interrupts the CPU load is minimal and the process can even be scheduled away in favor of other processes. If the process is not waiting for this message, it can also check for new messages at any point of time later. This might be the case if the process is busy with calculations.

Posted Receives. The payload is stored in dedicated data structures accessible both from network device and user-level process. This leads to a two-copy communication scheme. A one-copy communication can be implemented using posted receives to directly receive the payload in user-level data structures. These are only accessible using virtual addresses and introduce a lot of overhead. A one-copy communication scheme is already implemented using RMA operations, hence it is not supported for Send/Receive communication.

Using posted receives the user application has to register available buffers prior to a message reception. The completer unit consumes buffers by storing the payload in them. A posted receive renders the copy from receive buffer to the target process's data structures unnecessary. But the previous registration of buffers imposes several problems. A message must fit in a buffer, otherwise it cannot be stored. Hence the target process should know which message will be received next. If no message identification takes place another incoming message might use this buffer. The posted receive scheme requires a deep analysis of it's advantages in the specific application. Generally, it is supported in the communication architecture here but considered optional. An RMA operation might be a more suitable replacement for it.

2.3.4 Remote Memory Access (RMA) communication scheme

The RMA scheme is a one-sided and zero-copy communication scheme. The payload will be fetched from and written to user-level space using virtual addresses. Only the origin has to actively participate in the communication, the target can be passive. Because of this synchronization takes not place. Additional methods are required to allow synchronizing RMA operations.

The major advantage of zero copy is that the buffer to be sent over the network is not copied by processes, reducing the total overhead. To be more concrete, no process has to copy the payload in dedicated buffers. While a typical one- or two-copy scheme requires dedicated buffer to temporally store the payload, here it is directly read out from and written back to the user process's memory region.

The network interface as a peripheral device must be able to access the user process's memory region. With an RMA operation the process provides a reference to the payload. This reference is based on virtual addresses. Hence the device must be capable to translate virtual to physical addresses in order to fetch and write back the payload.

Memory Windows.

Memory windows [8] describe a certain region in user address space and are used in the RMA scheme for several purposes. Using windows, not the complete address space of a process is accessible by remote processes. Furthermore the windows are used to synchronize accesses and ensure mutual exclusion of multiple writers. Last, windows can be exclusively assigned to one or a group of remote processes.

The first item is achieved by specifying in each RMA operation the window on the target side. The window is described by a start VA, a length and read/write rights. All RMA operations must specify a window hence the remote access is limited to these regions. Furthermore the read/write rights must match the RMA operation.

Windows can be temporally locked to accomplish the second item. A locked window can only be accessed by the process who owns the lock. All other accesses are rejected, including accesses on a local window. Hence each RMA operation must additionally specify a local window. Only then both local and remote accesses are prevented to a locked window, and mutual exclusion is ensured. Otherwise race conditions may occur resulting in unwanted behavior.

Last, each windows can be assigned a capability [48]. Only processes with the matching capability are granted access to this window. Using this scheme a policy can be set up limiting the access to this window to a group of processes. Exclusive windows can either be implemented using capabilities or permanently locking.

Put operation. The work flow of a Put operation is two-staged and can be seen in figure 2.6. It starts with the origin process issuing a work request to the requester unit, including a window and a reference to memory. It is checked that the window is not locked by another RMA operation, which might currently be using this window. The requester unit directly fetches the payload using virtual addresses translated to physical ones. The payload together with the target window and address is included in a request packet sent out to the completer unit on the target node. Here the target window is checked. Only if it is not locked the payload is written back to memory, again using virtual to physical address translation. Optionally an acknowledge is sent back informing the origin process about the success of the operation.

Get Operation. While the Put operation is two-staged, the Get operation always includes a response. The resulting three-staged work flow can be seen in figure 2.7. It starts with the origin process issuing a Get operation to the requester unit. It includes a remote or target window and reference to be fetched and a local or origin window and reference where the data is stored. The requester unit sends out a request contain all this information. On the target node the responder unit checks the target window for locking and matching capability. The data is fetched from memory using virtual addresses translated to physical ones. The data is returned in a response packet including the origin reference. On the origin node the completer unit checks the origin window and writes the received payload data directly into the memory buffer described by the virtual address.

Fast operations. Comparable to the fast operations for the Send/Receive scheme, here it is also possible to avoid references by including the payload directly in the operation. For a fast Put the work request does not contain a local reference, instead the payload. For a fast Get the remote data read is included as immediate value in the notification.

This approach avoids local window checks, address translations and memory accesses. But the usage is limited due to the restricted size of a payload included in a work request or notification[1].

Atomic operations. Another kind of fast operation are atomic operations. Examples for atomic operations are *fetch-and-add* or *compare-and-swap*. They cannot be interrupted to ensure mutual exclusion while a data location is changed. They are implemented similar to the fast operations by including a remote reference, a compare or modify value and a place holder for the returned data. The use of atomic operations is dependant on support of the I/O interface on the target node. Otherwise, local processes on the target node may change the data location while an atomic operation is performed by the network device. If this is not possible all atomic operations, both local and remote, must be performed using the network device.

Synchronizing RMA operations. Because RMA operations are based on a one-sided communication scheme, the target can be passive. In other terms a target process is not

1. In sub-chapter *“4.2.3 Communication instruction descriptors” on page 149* the operations are specified in detail together with the resulting supported payload size.

involved in the communication. Because it is sometimes desirable to combine the zero-copy scheme of RMA with the synchronization of two-sided scheme like Send/Receive, it is optionally possible to notify the target process of an RMA operation. The target process is still not actively involved, only a notification is created informing it of a remote Put or Get operation.

Some existing systems poll on a data location for changes in order to get notified of an RMA operation. Usually the last data word within a region is used for this, assuming that the I/O sub-system writes back the data in-order. This is highly dependant on the system's architecture and implementation and hence not always true. A synchronizing RMA operation improves this method significantly and is independent of the underlying system.

Virtual addresses. For an I/O network interface it is not possible to access main memory using *virtual addresses (VA)*. Only *physical addresses (PA)* are supported. Because RMA operations rely on the use of VA, the VAs must be translated into PAs. This is typically performed by the *Memory Management Unit (MMU)* of the CPU. If a device wants to translate an address, it must call the O/S for support. There the address is translated and returned. Furthermore the corresponding pages are pinned to prevent swapping. The device can now access the region using the PAs. Upon completion of the access it allows the O/S to remove the pinning of the pages.

An improvement of this expensive scheme is an on-device *Translation Look-aside Buffer (TLB)*. It stores the most recent address translations. The pinning of pages can be improved by including a *coherency scheme* in the O/S [49], that checks the on-device TLB if the page to be swapped out is currently used. If the corresponding translation entry is not used it can be removed and the page is swapped out. In best case this allows the network device to use VAs without any involvement of the O/S. The worst case where the TLB contains no matching translations is not improved, but for the average case the overhead required for address translations is diminished[1].

2.3.5 Circular buffers

Queues in main memory are required for work requests. If the payload of a send operation is not included directly in the work request but instead by a reference, the referred data structure is also organized as a queue. The same applies for the receive operation, where the received payload is stored in a receive queue. RMA operations are a zero-copy scheme where no dedicated buffer is required.

So several queues are required in this design. Queues in main memory are typically not implemented as First-In First-Out data structures. The overhead to shift all entries one step is much too high. Instead circular buffers are used. The start and the end address of the queue inside this circular buffer is described using pointers (see figure 2.8). Inserting new queue elements result in incrementing the end pointer, while the removal of entries increments the start pointer. So the first enqueued entry is always consumed first and the ordering in the queue is guaranteed.

1. This is also substantiated by recent developments [50], in which memory translations and TLBs are also possible within the I/O sub-system.

The location of a queue is described with the lower and upper bound. Alternatively instead of the upper bound the length of the queue can be used. The start and end pointer must not be incremented beyond the upper bound. If the upper bound is exceeded, they are re-set to the lower bound, resulting in a circular buffer.

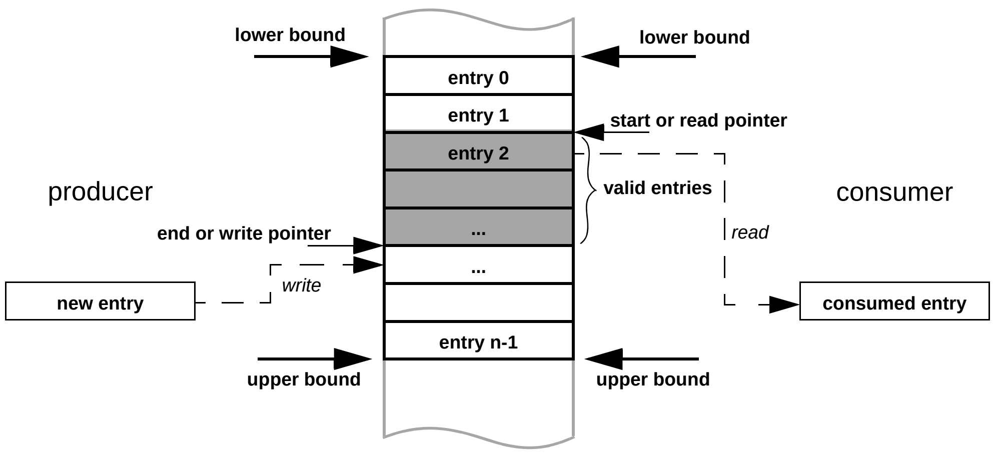


Figure 2.8 Circular buffer

Entries of a queue are inserted by the *producer*. The *consumer* of queue entries reads out and removes these entries. The start and end pointer can also be interpreted as read and write pointers. The consumer always increases the read pointer (by reading out entries), while the producer increases the write pointer (by writing in entries). A consumer never modifies the read pointer and vice versa, hence no mutual exclusion is required. The naming scheme of read and write pointer is more intuitive, hence this is used in the following.

Notification of changes. Several cases require notification when a queue is used. Either the producer is waiting for a queue entry to become free or the consumer is waiting because the queue is empty. In these cases both must be notified of a change in the queue so they can continue their work.

The design space in figure 2.9 shows that notification can either be based on interrupts or on polling. Interrupts require O/S support and prevent User-Level Communication. Polling is very time-consuming and increases the CPU load. While interrupt based schemes are well known and for User-Level Communication not suitable, the polling based scheme is now examined in detail.

The first approach is to poll on the read or write pointer for changes. A change of a pointer notifies the counterpart of a new or removed entry. The checks for an empty and full queue are also performed with the use of these pointers. Hence both producer and consumer must have access to both pointers.

If this is not wanted due to certain restrictions, for instance if the access costs to the appropriate pointer are too high, the alternative is to use a part of a queue entry as valid identifier. For instance all non-zero values indicate a valid entry, and all invalid entries have their valid field set to zero. For this approach the queue processing must be in-order, otherwise multiple entries must be checked for changes.

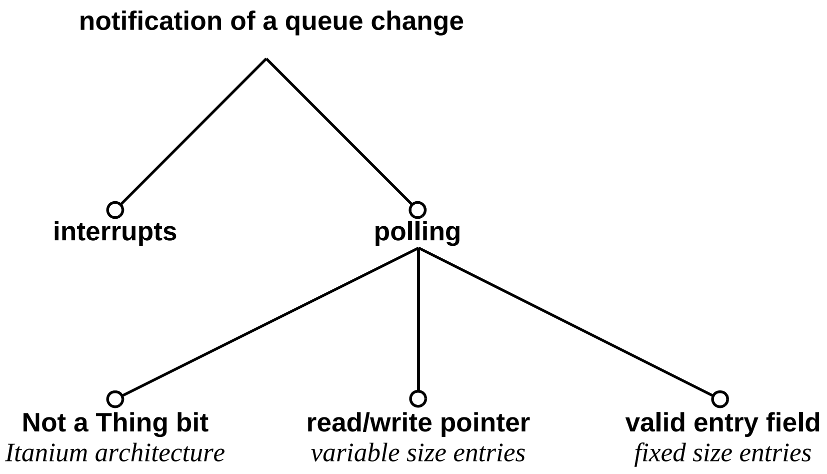


Figure 2.9 Design space of notifications

The *valid field* is always located at the end of an entry, because this is written last. Typical modern computing systems do not guarantee a write access to take place in one cycle, hence the write access can be split up in several parts. If the part of the entry containing the valid field is already written but the remaining part is still missing, the counterpart is already notified. Corrupt data is transferred, resulting in unwanted behavior. Hence the usage of valid fields requires that the entry size must be fixed. Only then the exact location of the valid field is known by both the producer and the consumer.

If a valid field is used the producer or consumer only requires access to the write respectively read pointer. Prior to an enqueue of a new entry the producer checks if the entry referenced by the write pointer is invalid. Only then the entry can be enqueued. Comparable the consumer checks the entry referenced by the read. Only if it is valid it can be consumed. Both the insert and remove operation require also to toggle the valid field.

A complete different approach requiring hardware support is based on an additional tag beside each data word. This tag indicates if the corresponding data word is valid or not. For instance the *Itanium* architecture implements such a tag, here called *Not a Thing (NAT) bit* [51]. The use of such a tag is restricted to the specialized architecture, which is not desirable here.

2.3.6 Communication context and data structures of a process

The communication *context* of a process includes it's complete configuration required to process work. The configuration describes all used queues and data structures, including base pointers and length (or lower and upper bound) and read/write pointers for the queues. Beside this a control field can enable or disable the work processing and a status field include information about the current state.

The routing space is shared among all processes, hence no information about the location of this region is required in the context. This decision is made in order to reduce the size of the context. In addition dedicated routing spaces for each context would require much more space, because a route to one destination must be stored in the space of each process using this route. For a shared space the route is only stored once.

In the following the required data structures for the Send/Receive and RMA communication scheme are shown. The data structures are derived from the analyzed work flow in sub-chapter *“2.3.3 Send/Receive communication scheme” on page 45* and *“2.3.4 Remote Memory Access (RMA) communication scheme” on page 47*.

Both schemes require at least two queues, one from process to device and the other vice versa. The first is a *Work Queue (WQ)* containing work requests to be processed, the second is a *Notification Queue (NQ)*.

Both queues are implemented as circular buffers with fixed size entries. Read/write pointers are used to control them. Due to the location of the network interface in an I/O subsystem the WQ requires a special notification scheme which is introduced in sub-chapter *“3.4 Virtualization” on page 107*. The NQ supports notification based on a valid field. If polling is not required the process can register itself to an interrupt notification scheme.

Send/Receive communication scheme. This scheme requires a dedicated buffer for sending (*Send Data Region, SDR*) and another dedicated buffer for receiving (*Receive Data Region, RDR*). Data in these buffers is referenced by the work requests respectively the notifications.

Both regions are implemented as circular buffers. They are controlled using read/write pointers. For them no notification is required because they are always referenced by entries in the WQ respectively NQ. These queues already include a notification scheme.

Each context has it’s own SDR and RDR. Hence in each context a reference to the start of the SDR and the RDR is included, together with their lengths. If the space available for a context is limited the length of all SDRs and RDRs can be equal, allowing to remove the length informations from the context.

RMA scheme. This zero-copy scheme does not require dedicated buffers. Instead the normal working address space of the process is used. Work requests and notifications include references to virtual addresses of the process’s address space.

The windows are implemented using a *Window Descriptor Table (WDT)* [48]. This table is indexed by a *Window Identification (Win-ID)*. Each entry in the table is a descriptor, where the window is described using base address, length, access rights, capability and a locked field. Each context has it’s exclusive WDT hence the context must include a reference to the start and the length of the WDT. Comparable to the SDR/RDR length, insufficient context space can lead to store the length of the WDT centrally. Then the WDTs of all context must have the same size.

2.3.7 Interface between process and network interface

The main interface between process and network interface is in one direction the work queue for issuing. In the other direction the notification queue return the result of issued work and other informations about ongoing work or the status of the network interface.

For both queues each entry is a descriptor. Figure 2.10 shows the design space of descriptors. In this communication architecture all types of descriptors shown in the figure are used. Fast operations (see sub-chapter *"2.3.3 Send/Receive communication scheme" on page 45*) use immediate values in descriptors, while normal operations typically rely on including references to other data structures. While the RMA scheme uses virtual addresses, the references used in the Send/Receive are physical addresses.

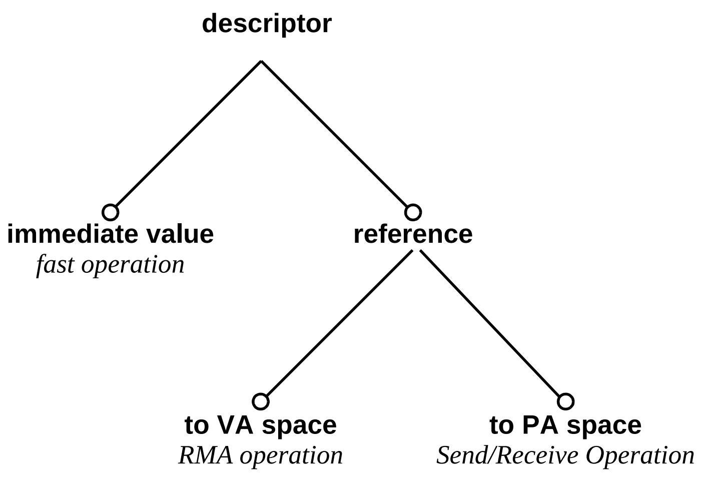


Figure 2.10 Design space of descriptors

The size of a descriptor is dependant on the number of parameters. But it should be based on the granularity of *cache lines*, because they are the smallest unit under control of the cache coherency protocol. If one cache line contains several descriptors, *false sharing* may occur. This results in multiple transfers of single descriptor, because the corresponding cache line of this descriptor is invalidated and updates upon each change of another descriptor in this cache line.

Hence a descriptor should consists of one single cache line. If a single cache line is not sufficient multiple padded cache lines must be used. The same applies for nearly all elements in the data structures of the communication architecture, in particular the *context* and the *window descriptors*. Only the buffers for payloads can be excluded from this constraint, because of their variable size.

2.3.8 Communication instructions and operations

An instruction can be seen as a set of operations. For instance an instruction is issued by a process to a CPU, where this instruction is performed using micro-coded operations. Each operation performs only a part of the work.

Comparable to this, all *operations* of the Send/Receive and RMA schemes are called *communication instruction* in the following. The next sub-chapter *"2.4 Communication Instructions" on page 56* provides the specification of the complete communication instruction set. To process a communication instruction several operations are required.

For instance an RMA Put instruction starts with the issue operation. Here the work request is passed to the network interface, where the processing starts. For a non-blocking issue operation a later check for completion is required. This can be a status query or a waiting for the appropriate notification entry. If this notification entry is present, the process consumes it. The device must be informed of the consumed entry by a pointer update operation.

The set of operations required to process the available communication instructions are now introduced and explained. Not all operations apply for user processes, some are restricted to be used only by privileged processes. But the complete set of operations must be supported by the network interface architecture.

Issue operation. The *issue operation* is the enqueuing of a new work request in the work queue including the notification of the device. Every communication instruction starts with an issue operation. The issue operation is non-blocking. If the process wants to synchronize to the completion of the instruction, it can either rely on an interrupt-based notification or poll continuously on the notification queue.

A non-blocking issue operation allows independent work flows on the main CPU and the device. Work request issued to the device take typically much more time to be processed than CPU instructions. The CPU can perform outstanding tasks instead of waiting for the work to be completed. This may not be desirable for small work requests. Then a blocking status query operation following the issue operation allows to block until completion.

Status query and control operation. The *status query operation* reads out the current state of the network interface context. It is not part of a typical work flow for a communication instruction but available for user processes. In contradiction the *status control operation* is typically restricted to privileged processes, which are considered reliable and hence do not disable any required hardware modules.

Pointer update operation. A *pointer update operation* is required to notify the producer of a queue of a removed entry and vice versa the consumer of a new entry. Hence this operation only applies for queue-based data structures which use a read/write pointer notification scheme. In the communication architecture here, in particular the notification queue read pointer must be updated by the process using this operation.

Cache operations. Special operations are required to control the on-device cache structures and the TLB. The caches are not participating in the system's *cache coherency protocol* due to the limitations of the I/O interface, hence the consistency and coherency must be ensured manually. The TLB contains the most recent address translations. Support for inserting new translations or removing outdated ones is required. Because both the caches and the TLB must be kept manually consistent, the following operations both apply for the caches and the TLB.

Depending on the update policy, several operations are required. Inevitable is the *cache flush operation* invalidating the complete cache and a *cache insert operation* for new entries. For a *write-invalidate cache policy* basically only the cache flush operation is required. A *cache remove operation* avoid to invalidate the complete cache, instead only one element is removed. A large number of further operations are possible, which can be found with a detailed description in [52]. For instance, bulk update of multiple entries likely improves the overall performance if a typical RMA packet transfer exceeds the size of one page.

2.4 COMMUNICATION INSTRUCTIONS

In the previous sub-chapter the work flows for the Send/Receive and RMA communication scheme are explained (see *“2.3.3 Send/Receive communication scheme” on page 45* and *“2.3.4 Remote Memory Access (RMA) communication scheme” on page 47*). Still missing is a detailed description of each available *communication instruction* in these schemes.

The separation into multi-staged work flows allows to execute different commands in each stage. The first command of a work flow is contained in the communication instruction, issued by a process to the requester. The requester injects a packet into the network with a command for the next stage, which is either a responder or a completer. A responder answers with a packet including a command for the completer. The completer is always the final stage of the work processing (except for the optional acknowledge). It generates a notification queue entry containing a command which informs the local process (which can be both origin or target) of the result of the work.

2.4.1 Commands for Requester

The origin process initiates the communication by issuing an instruction to the requester. The instruction issue also triggers the requester to start the work processing. When the requester finishes it’s part of the work, it sends out a request packet over the network. The instruction determines if the work is two- or three-staged. For a two-staged work flow the destination is a completer, for the three-staged a responder. Figure 2.11 shows the work flow for the requester.

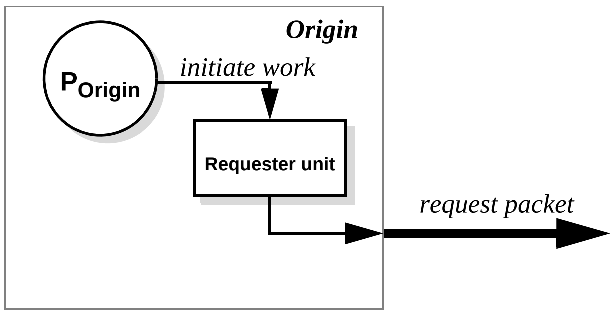


Figure 2.11 Work flow for Requester

Hence the requester receives instructions from application processes. The command section of the instruction determines the type of work to be processed. In *Table 2.2* the supported commands are listed, together with a short description.

Table 2.2: Commands for Requester

Command	Description
Send	Part of the two-sided communication scheme. Payload is located in dedicated buffers, and the instruction includes a reference. Size of the payload is only limited by the buffer size.
Fast_Send	Similar to Send instruction, but the payload is included in the instruction as an immediate value. Hence the size of the payload is restricted.
Put	Part of the one-sided communication scheme. Payload is directly fetched from user space and at the target written into user space. Payload size is only limited by the used windows.
Fast_Put	Similar to Put instruction, but the payload is included as an immediate value in the instruction, hence it’s size is limited.
Misaligned_Put	Similar to Fast_Put instruction, but only parts of the immediate value are marked valid using a byte enable. This allows to over-write only parts of a data word at the target.
Get	Part of the one-sided communication scheme. Payload is fetched directly from the target’s user space and stored at the origin into user space. Payload size is only limited by the used windows.
Fast_Get	Similar to Get instruction fetching the payload directly from the target’s user space. But here at the origin the payload is included as immediate value in the notification.
Misaligned_Get	Counterpart for Misaligned_Put instruction. Because data is only read and not written, this instruction is optional.
Fetch_And_Add	Atomic operation and part of the one-sided communication scheme. The instruction includes a target address and an addend. The addend is added to the data located at the target address and the result is both stored and returned. All payload is included as immediate value.
Compare_And_Swap	Comparable to Fetch_And_Add, but here instead of an addend a compare and swap value is included. Only if the compare value matches to the target address, the swap value is stored instead of the original value.

2.4.2 Commands generated by Requester

Depending on the instruction issued to the requester, it sends out a packet to either a responder or a completer. The commands generated by the requester are equal to those in *Table 2.2*. Using them the target can decide whether the incoming packet is processed by a responder or completer.

If no acknowledge is requested for this instruction, the requester immediately informs the origin process that the instruction is completed. This is done by inserting a new entry in the process’s notification queue.

Table 2.3: Notification Commands generated by Requester

Command	Description
Notifiy_Complete	Notifies the origin process that the instruction is processed.

If the instruction includes an acknowledge, the completer on the target node sends back an acknowledge, which is received by the completer on the origin node. The origin completer then inserts a notification queue entry.

2.4.3 Flow for Requester

In the following the requester part of the work flow is explained in detail. It starts with triggering the requester of a new entry in the work queue. Dependent on the instruction, some steps can be skipped.

1. Work request descriptor load
2. Routing fetch
3. Routing output to network
4. Command frame generation
5. Command frame output to network
6. Window descriptor load
   - 6-a. This implies a short check if the window is locked.
   - 6-b. For a locked window the processing is aborted, the descriptor is marked erroneous and written back as a notification.
7. Repeat for each page of the payload:
   - 7-a. Translate page offset
   - 7-b. Fetch data for this page
   - 7-c. Output data to network
8. Mark descriptor completed and write-back as a notification

The packet injected to the network is composed of the routing, the command frame and optionally the payload. It is send through the network to the target. Here it is either processed by a responder or a requester.

2.4.4 Commands for Responder

A responder only receives commands by a requester, but not all requester commands are processed by the responder. In figure 2.12 the work flow is depicted and the following table shows the command set for the responder.

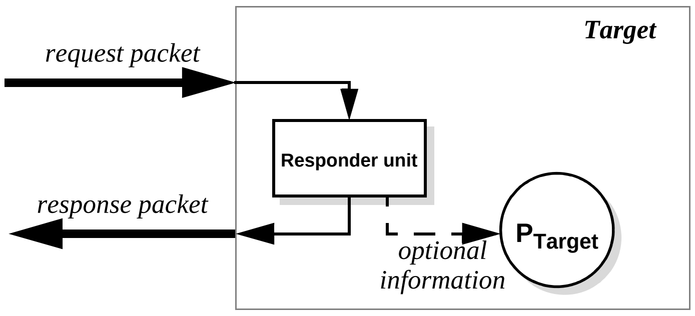


Figure 2.12 Work flow for Responder

Table 2.4: Commands for Responder

Command	Description
Get	Fetch payload directly from user-level space and return it in a response packet.
Fast_Get	
Misaligned_Get	Equals the Get command. The byte enable mask can be applied either on the target or on the origin node.
Fetch_And_Add	Perform the requested atomic operation and return the result in a response packet.
Compare_And_Swap	

2.4.5 Commands generated by Responder

Commands generated by a responder always target a completer. To distinguish the packets from requester and completer, additional commands are required. These commands are based on the incoming commands in *Table 2.4,* but mark this packet as a response.

Table 2.5: Packet Commands generated by Responder

Command	Description
Get_Response	This is the answer for a Get instruction, including the payload fetches from the target address.
Fast_Get_Response	
Misaligned_Get_Response	Similar to Get_Response, but parts of the payload can be selected using a byte enable.
Fetch_And_Add_Response	Includes the result of the atomic operation performed at the target.
Compare_And_Swap_Response	

The responder can optionally notify the corresponding target process of a performed RMA instruction. This is done by inserting a new notification queue entry in the process's notification queue.

Table 2.6: Notification Commands generated by Responder

Command	Description
Notifiy_RMA	Notifies the target process that an RMA was performed.

2.4.6 *Flow for Responder*

In the following the responder part of the work flow is explained in detail. It starts with an incoming packet from the requester. The last two steps are optional and only required for the notification of the passive target process.

1. Window descriptor load
2. Window check
   - 2-a. If the window check fails, the processing is aborted. A negative acknowledgement is sent back to inform the origin of the failure. Furthermore a notification descriptor is generated informing the target of a failed RMA operation.
3. Reverse routing output to network
4. Command frame generation with response command
5. Command frame output to network
6. Repeat for each page of the payload:
   - 6-a. Translate address for each page
   - 6-b. Fetch data for this page
   - 6-c. Output data

7. Notification descriptor generation
8. Notification descriptor store

The outgoing packet is composed of routing, command section and payload. The next step in the work flow is always a completer.

2.4.7 Commands for Completer

The completer is either located on the origin or the target node. It processes packets coming in from a requester or a responder. Except the optional acknowledge, a completer is always the final stage of a work flow. Here ends the processing of the payload.

If an acknowledge is requested, the target completer sends back an acknowledge to the origin completer. The origin completer then generates an appropriate notification queue entry. For the three-staged work flow acknowledges are not required.

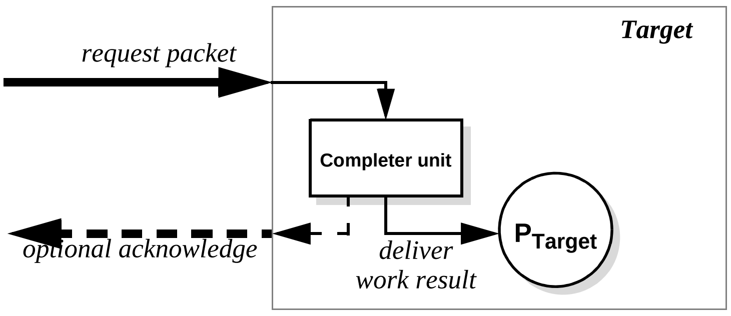


Figure 2.13 Work flow for target completer

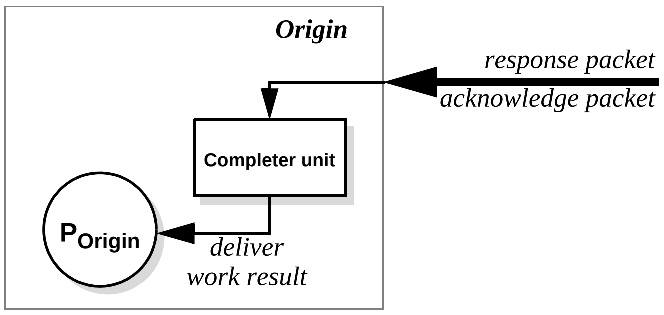


Figure 2.14 Work flow for origin completer

The following table shows the commands for the completer, derived from the commands generated by the requester and the responder. Additionally, the acknowledge is included as a command generated by a completer.

Table 2.7: Commands for Completer

Command	Description
Send	Store the payload in dedicated buffers.
Fast_Send	Generate a notification including the payload as immediate value.
Put	Store the payload directly in user space buffers.
Fast_Put	
Misaligned_Put	Similar to Put instruction, but only the parts of the payload enabled by a mask are stored in user space buffers.
Get_Response	Store the payload directly in user space buffers.
Fast_Get_Response	Generate a notification including the payload as immediate value.
Misaligned_Get_Response	
Fetch_And_Add_Response	
Compare_And_Swap_Response	

2.4.8 Commands generated by Completer

The completer generates notification queue entries which include a command. The completion of all operations listed in *Table 2.7* is notified to the corresponding process. The appropriate entry then contains the same command, allowing to identify the completed instruction.

Additionally, for a two-staged work flow a (target) completer can send back an acknowledge. The command for an acknowledge packet is shown in *Table 2.8*.

Table 2.8: Packet Commands generated by Completer

Command	Description
Acknowledge	Acknowledge packet sent back from target to origin.

The origin completer receiving an acknowledge packet generates a notification queue entry. The only available command is already shown in *Table 2.3*.

2.4.9 Flow for Completer

In the following the completer part of the work flow is explained in detail. It starts with an incoming packet. As long as no acknowledge is required the completer is the final stage of the work processing.

1. Window descriptor load
2. Window check
   - 2-a. If the window check fails, the processing is aborted. If the current node is the target a negative acknowledgement is sent back to inform the origin of the failure. Furthermore a notification descriptor is generated informing the local system (origin or target) of a failed RMA operation.
3. Optional acknowledge generation and returning:
   - 3-a. Reverse routing output to network
   - 3-b. Command frame generation with acknowledge command
   - 3-c. Command frame output to network
4. Repeat for each page of the payload:
   - 4-a. Translate address for each page
   - 4-b. Fetch data for this page
   - 4-c. Output data
5. Notification descriptor generation
6. Notification descriptor store

Chapter 3 Network Interface Architecture

As cluster nodes become more and more parallel [53], *efficient and simultaneous access* to the network device from multiple processes becomes even more important. The architecture of a network interface (or a device in general) should provide support for a large number of processes accessing directly the device using *User-Level Communication*. Another goal is a high utilization of the hardware resources independent of the number of processes.

An *efficient interface* to the device is the key for a *scalable architecture*. The interface is a bottleneck, which is in particular true when the number of client processes simultaneously accessing the device is scaled. An optimized access scheme allows to share this interface among a larger number of processes. Only then parallelism is available to be exploited by the network interface architecture.

This chapter starts with a short introduction into the most important working principles and techniques for network interfaces and devices in modern computing systems. This will form a basis for the following sub-chapters, where these existent techniques together with new ones are used to develop a most suitable architecture for a network interface. Most of the results are not limited to network interfaces, other kinds of devices may also benefit of them.

In the following *Chapter 4 ”Specification and Evaluation”* the results of this chapter are collected and combined with those from *Chapter 2 ”Communication and Synchronization”*. The proposed top-level architecture of the network interface is presented and allows a comprehensive view of this development and it’s integration into the system.

3.1 INTRODUCTION

This sub-chapter will shortly introduce common methods and techniques used in modern computing systems. It focusses on Network Interfaces and their integration into the whole system. This background is essential for an in-depth understanding of the following sub-chapters.

3.1.1 Device access methods

Inevitable for devices in modern computing systems is to allow access from several processes to a single device. Due to the required separation this can be either performed by the *operating system* or directly in hardware. Hardware solutions allow to bypass the O/S and avoid costly system calls. Figure 3.1 shows the design space diagram for device access methods. These methods are now explained in detail.

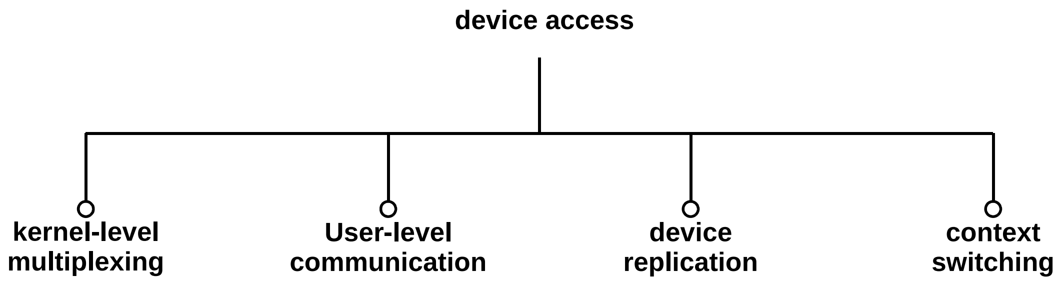


Figure 3.1 Design space of device access methods

Traditionally, the access to peripheral devices is multiplexed by O/S to support more than one process (*Kernel-level multiplexing*, *Figure 3.2a*). 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 logic complexity of the device can be kept quite low because the O/S can overtake a lot of required actions. Typically, those devices do not off-load tasks from the CPU.

A solution to avoid the performance degradation is the principle of *User-Level Communication* [42] (*Figure 3.2b*). By address space mapping of I/O pages into user-level a process directly opens the device for exclusive usage. Overhead due to system calls is completely avoided, but the O/S can no longer supervise the processes. The task of supervision has to be done by the device itself and off-loads (*CPU-offloading*) tasks from the CPU. This increases the amount of required logic for the device and so the complexity of hardware. Another disadvantage is that only one process can open the device at a time.

Techniques to support multiple processes together with User-Level Communication are *device replication* (*Figure 3.2c*) and *context switching* (*Figure 3.2d*). Due to the supported User-Level Communication both techniques perform off-loading of supervision tasks and lower the load of the CPU.

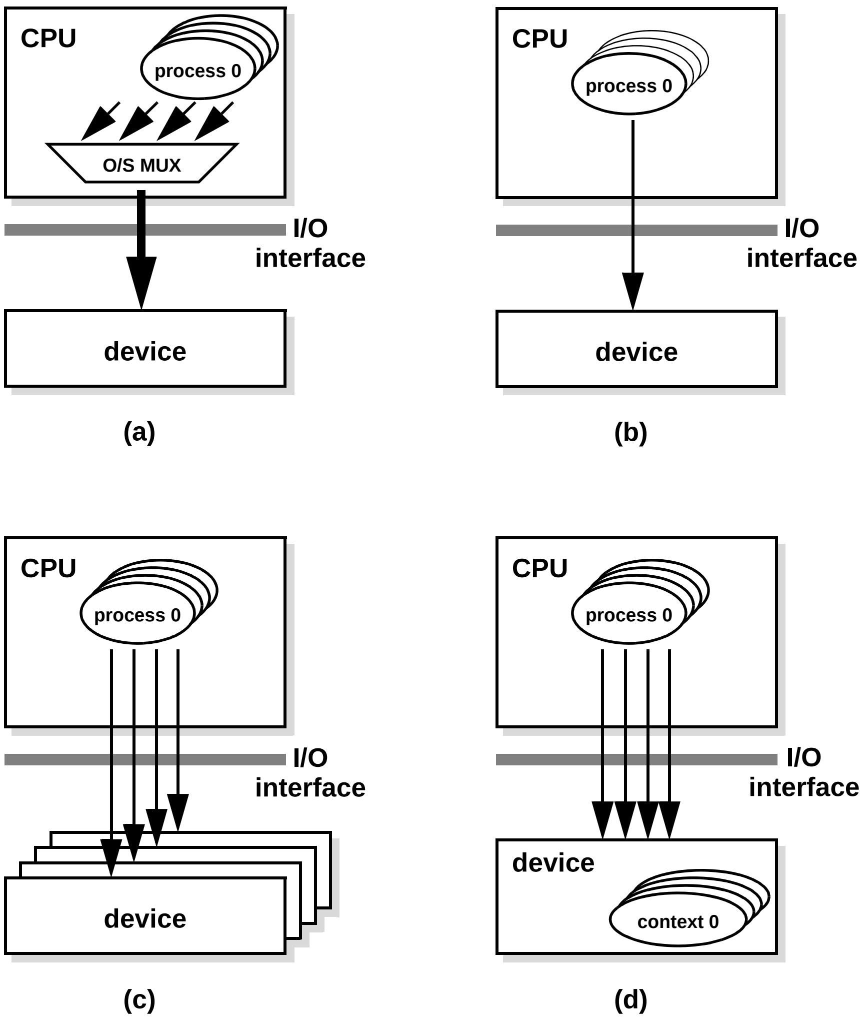


Figure 3.2 Overview of device access methods

Device replication [39] can be achieved by replicating the appropriate hardware modules. From the user process point of view, each replicated hardware module looks like an independent device. Due to the replication of the hardware the supported number of processes is fixed. This solution does not scale: every new interface requires a complete new hardware module. The scalability is limited by the on-chip network which has to connect all modules to the I/O interface. Another constraint is the required area of the design. Additionally this method is not very efficient because unassigned modules cannot be used to improve the performance by distributing the work of one process over several modules.

Context switching supports several processes accessing the device, too. Each access is followed by a device context switch. In a device context, process specific configuration informations are stored. The context switch is typically supervised by an on-device processor. This processor recognizes the calling process and configures the appropriate hardware modules. Normally the switching takes place in a time-sliced manner. The utilization of the resources and support for the exploitation of parallelism is dependent of the architecture of the embedded processor. A combination of device replication with this approach leads to a processor with several (communication) cores. Then a limited number of concurrent running threads is supported.

Context switching is the most promising. The concepts of context switching developed for CPUs is adapted to devices. In a CPU, the O/S is responsible to organize the scheduling of the user processes. For a device the counterpart can be a processor, but dedicated hardware modules are also possible and offer a better exploitation of the available parallelism. The usage of contexts allows a scalable design if the size of a context is kept small and the overhead for context switches is very small. Then this method allows the device to store the information of a large number of processes. If one process accesses the device, a free module is configured for this process using the context informations and the work is processed.

Table 3.1: Summarization of device access methods

	Kernel-level multiplexing	User-Level Communication	Device replication	Context switching
Supported number of clients	unrestricted **+**	1 **-**	limited **0**	unrestricted[1] **+**
User-Level Communication	no **-**	yes **+**	yes **+**	yes **+**
CPU off-loading	no **-**	yes **+**	yes **+**	yes **+**
Dynamic resource utilization	O/S level scheduling **0**	no **-**	no **-**	yes **+**
Logic complexity	low **+**	moderate **0**	moderate **0**	high **-**

1. Under certain circumstances, see accompanying text for requirements. Beside this, every hardware introduces limitation, but they can be set so high that they are in the end not effective.

Table 3.1 summarizes the properties of the different access methods and rates their advantages and disadvantages. The most promising approach to fulfill the goals of this work is context switching.

3.1.2 Data transfer methods

Two different approaches are possible to transport data from and to the device. In the *PIO mode* the CPU reads and writes all data from and to the device using the PIO mode. The device itself is not accessing main memory. In the *DMA mode* the CPU specifies the location of data structures in main memory. The device directly accesses the main memory and fetches the data. This unloads the CPU from this task.

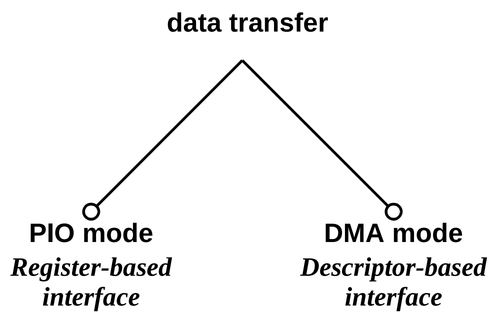


Figure 3.3 Design space of data transfer methods

The PIO mode is used for *register-based interfaces* [18]. Various versions exist, but all are based on composing a message in registers, either specialized processor registers or device registers. One register (set) is dedicated for message sending and one for message reception. The disadvantage of this approach is a large overhead for the processor, wasting computing resources while serving as a DMA engine. *CPU-offloading* is not possible. Because the sequence of message composition and retrieval may not be interrupted by other processes some additional problems arise.

1. The scheduler of the system may not switch the running process during such a sequence. Either such a sequence is uninterruptible or a single atomic operation is used.
2. It must be ensured that a sequence is always completed within finite time, otherwise the resources are blocked.
3. The desired support for simultaneously accessing processes leads again to the problem of the mutual exclusion during the sequences. This may also result in scalability problems.
4. The work can only be processed in order.

A solution to avoid the problems above is a *descriptor-based interface* [18]. It is based on the DMA mode. In a descriptor a message is described regarding destination, tag, other header information and payload. For larger data structures with variant sizes (like the payload) usually only a reference is included (using pointers), but for smaller payloads immediate values are also possible. These descriptors can be pre-assembled by the processor and issued in one block to the device.

3.1.3 Work requests and notifications

A *work request* of any kind can be described by a descriptor. Such a work request may contain a message to be sent, status query, pointer updates or anything else that is required. Work requests are sent from a process to the device. In the other direction the information is usually passed using *notifications*. Notifications may contain information of received messages, frequent status or pointer updates. For notifications are also descriptors used. To recognize a new notification, a process can either poll continuously on a known memory location for changes, or it registers itself to an interrupt notification scheme. Then the device generates an interrupt for every new notification.

A work request can also be treated as an instruction which leads to a procedure call processed by the device. It has many similarities to an instruction of a CISC architecture, including complex commands with immediate or in-direct data values. The device-level procedure call unloads the CPU from processing this task. The procedure call starts with issuing a work request to the device. Then the device processes this work. An asynchronous check for completion is performed later and the CPU is informed that the procedure call is finished.

Each work request requires a certain latency to be processed. Due to the work granularity the required time to process a work request is much larger than the processing time of a CPU instruction. By performing other tasks (e.g. computation tasks) this latency can be hidden. To allow efficient latency hiding support for several outstanding work requests is inevitable. It must be possible to issue one (or preferable more) work requests to the device. During the accumulated time to process them the CPU can perform other tasks instead of waiting for the work completion. In other terms support for several outstanding work requests allows to decouple the CPU and device processing. A *queue-based interface* is a solution for this problem. Work requests are enqueued by the process and dequeued by the device. The queue can be located either in main memory or in on-device memory, dependent on the requirements and constraints. For the opposite direction from device to process the same applies, but here the device enqueues notifications and the processes dequeue them.

3.1.4 Simultaneous device access

Computer system are getting more and more parallel due to several reasons (see figure 3.4). The processor architecture improves more and more by techniques like multi-threading or core replication [54][55]. One physical processor can run several processes or threads simultaneously. Typical modern systems house not only one but several processors. Applications tend to be more and more divided into threads. An optimal and efficient exploitation of the available parallelism is essential for a performant system. While the current processor designs already take this requirement into account, the peripheral devices and the I/O interfaces still provide no adequate support for this.

One of the goals of this work is to support multiple processes accessing the device simultaneously. All these processes issue their work request concurrently to the device. The interface to the device will be the limiting factor of the design, because it has to be shared among all processes. Furthermore the interface is used for all main memory accesses from the device.

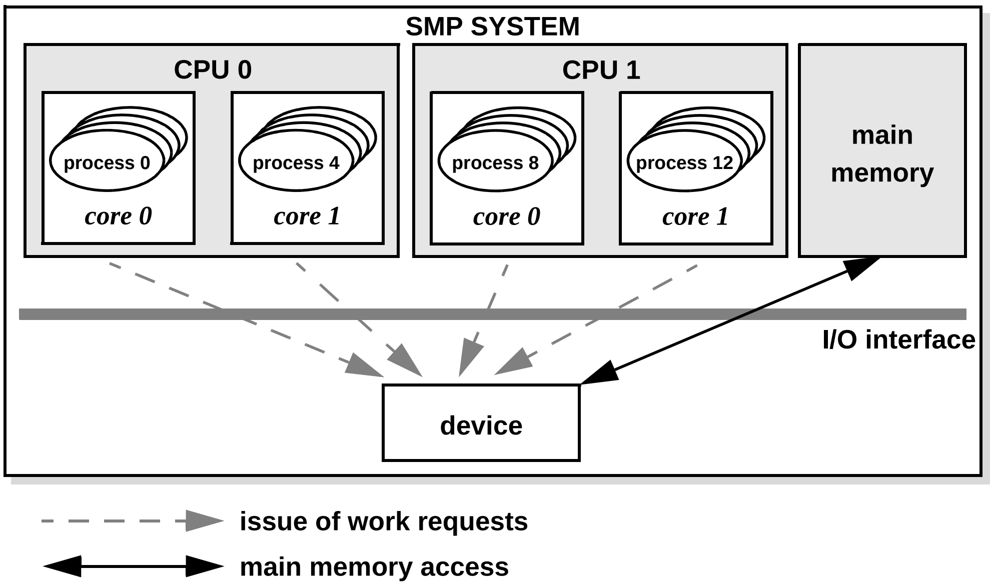


Figure 3.4 Multiple processes accessing a device

3.1.5 On-device memory

Each modern CPU has a *memory hierarchy*, from the registers over the different cache levels to the main memory (see figure 3.5). The *demand paging* principle can also be included in this hierarchy, extending it to mass storage devices. The memory hierarchy has proven to be very efficient for cost-performance optimization. Starting with the registers as most performant but also most expensive, a hierarchy component's size increases while it's cost decreases. The last component, the mass storage devices, have the lowest performance but also the lowest costs per bit.

Some parts are visible to user-level processes, others not. The registers have to be loaded explicitly while all cache components are transparent and reduce access costs to underlying components automatically. A *cache coherency protocol* ensures the correctness of all cache entries. For a user-level process the swapping of pages to mass storage devices is also not visible. The O/S with the help of the MMU takes care of this. The *working set* of a process contains all referenced resources within a given time interval. Typically a working set is bounded because applications likely work on limited resources, at least within small time intervals. This is the reason for the excellent performance gain of caches. A cache stores copies of the most recently used resources and reduces the access costs.

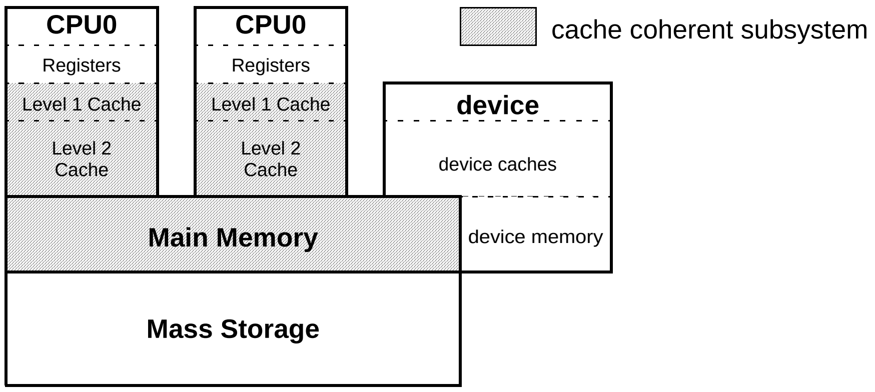


Figure 3.5 Memory hierarchy of a system

From a device's point of view there are two kinds of memories: *on-device memory* and main memory. Compared to main memory the on-device memory is more expensive and more limited regarding size. To reduce the overall costs of a design, a goal must be to reduce the amount of on-device memory. A design without on-device memory at all is also called *memory-less design* [56]. Here all data structures are stored in main memory.

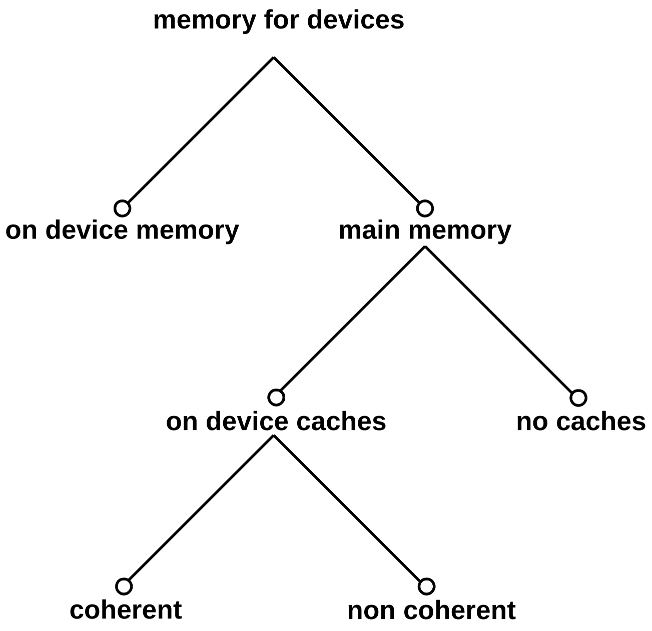


Figure 3.6 Design space of memory for devices

The largest drawback of main memory usage for a device are the access costs and a missing cache coherency. By not taking part in the coherency scheme is device is not informed of any changes in main memory. The access costs can be limited by applying a memory hierarchy to the device design, resulting in *on-device cache* structures. Comparable to CPU processes, each device thread has a working set which allows the use of cache structures. For caches much less on-device memory is required and the caches reduce the average memory access costs. But the non-coherent interface requires additional complexity to ensure coherency and consistency of the on-device caches.

In spite of this a memory-less design with on-device caches is the most scalable solution, when more memory is required. The bad performance of main memory accesses is improved by the caches.

Although the architecture would benefit from a coherent interface, one design goal is a non-coherent interface. Otherwise the design would be limited to seldom available systems with coherent interfaces.

3.1.6 Virtual Machine environments

Modern computing systems are powerful enough to host not only one operating system, it is feasible to run several operating systems on a single physical platform. This leads to a resurgence of *Virtual Machines (VM)*, a technique that is well known for a long time but was always limited to specialized high-performance mainframes and servers with dedicated support for virtualization [16]. The virtualization of a complete system, including processors, memory and I/O, makes it possible to run several operating systems on one physical machine.

There are several reasons for virtualization. Utilization, reliability and maintainability increase for VM environments. For different O/S requirements it is easy to share a virtualized server with different types of O/S running on it. Each O/S can be chosen dependant on the user’s requirements. Failures or intrusions in one O/S instance are isolated and cannot affect other instances. O/S upgrades are easily possible and minimize downtime. Regarding the workload management there are also several advantages: the various workloads in heterogeneous environments can be consolidated onto one virtualized machine. Another feature is the workload migration, which allows to shift a workload from one virtualized server to another. This can be achieved by encapsulating the state of a guest. Workload isolation allows to separate applications. If several applications share one O/S (one physical machine) they can be isolated by virtualizing the server and assigning each application it’s own O/S, again providing increased isolation.

Virtual Machine Monitors. On a non-virtualized platform one single O/S has direct control over all physical resources, they are used exclusively by this O/S. For a VM environment a new software layer is introduced, the *Virtual Machine Monitor (VMM)* or *Hypervisor.* The VMM is located on top of the operating systems. The operating systems are now guests of the VMM and each guest runs in a domain. No longer an O/S controls the physical resources, instead the VMM multiplexes the accesses from all guests to the physical resources. No O/S can directly access any physical resource. The set of virtualized resources provided by the VMM for one guest is called VM.

Resources can be virtualized by a VMM in different ways. One approach is the partitioning of resources. Here the resource is divided into subsets. Each subset can be assigned to another guest. This is only possible for resources that can be split. A typical example is the physical memory. In a typical VM environment, the total amount of memory is divided into subsets of equal size. Each guest gets only one subset. Resources that are non divisible are typically shared in a time-sliced manner between the competing guests. Examples for this are the CPUs, and I/O devices like network interfaces, storage controllers or any other peripheral device. The VMM schedules the guests in a round-robin fashion (or any other suitable) to the shared resources. Each guest gets only one time slice to use the resources, after this they are scheduled away in favour of other guests.

Very important for VM environments is that a VMM ensures isolation between guests. No guest may interfere with another guest. This separation must cover performance, security and safety. Only then the isolation can ensure that no guest affects the performance of other guests and that no guest can alter or use resources of other guests. Beside these requirements it is essential for the system's overall performance that the overhead of the VMM is as small as possible.

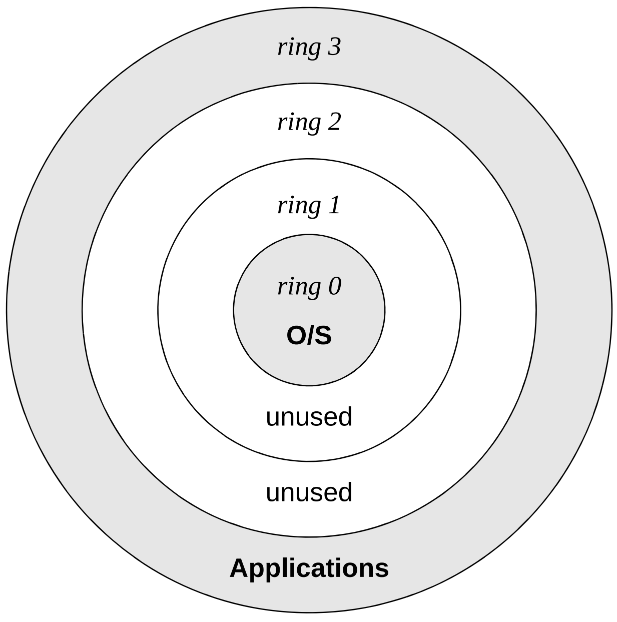


Figure 3.7 Protection scheme for non-virtualized environments

The general idea of VM environments is a new layer, the VMM, which is located on top of all guest systems. The VMM layer must be protected from all underlying layers. Specialized servers and mainframes already include support for this, but the typical x86 architecture has no built-in virtualization support. The x86 architecture only provides four privilege levels (encoded with 2 bits). Level 0 is used for most-privileged layers, level 3 for least-privileged. The levels are also called rings. Privileged instructions can only execute in certain privileged rings. The ring based protection scheme ensures in non-virtualized systems that applications have less rights than the O/S and that the O/S is thus able to supervise the applications. Most O/S today only use two rings: ring 0 for the O/S and ring 3 for applications.[1] Figure 3.7 shows the ring protection scheme for a non-virtualized environment.

The challenge is to insert the VMM layer on top of the O/S, which runs already in the most-privileged ring. The x86 architecture provides no support for this situation. Several software techniques offer a solution for this problem: the *full virtualization*, the *paravirtualization* [57] and *virtualization in O/S level*. A completely different approach is to add hardware support for virtualization to the x86 architecture. Figure 3.8 shows the design space of these techniques.

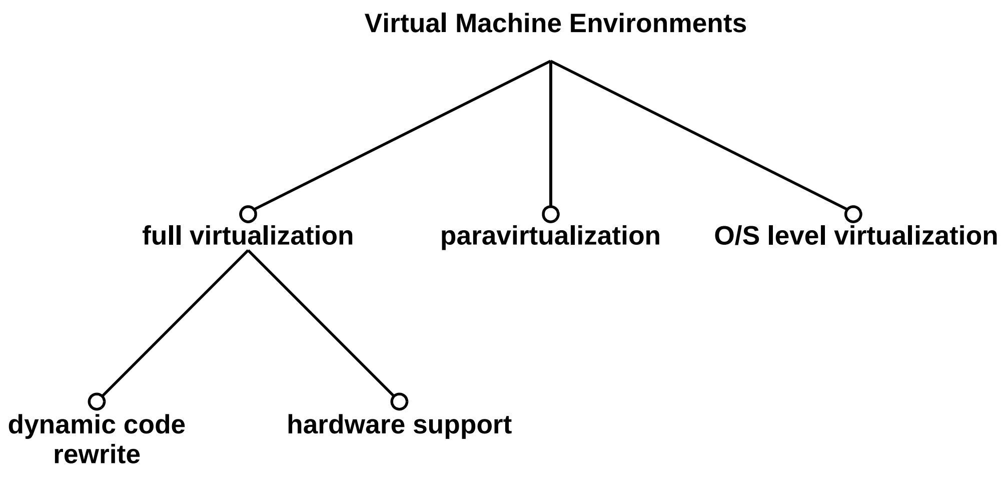


Figure 3.8 Design space of virtual machine environments

Full virtualization on x86 architectures. Full virtualization allows the hosting of unmodified operating systems. The challenge is that O/S already use the most-privileged ring 0 and the x86 architecture does not provide any hardware support. For a correct behavior certain instructions must be handled by the VMM and not by the O/S. A solution for this problem is to dynamically rewrite the code. This binary translation technique inserts traps where instructions must be handled by the VMM and not by any other underlying layer. This introduces a significant amount of overhead and is not an optimal solution.

Paravirtualization. The paravirtualization approach [57] avoids dynamic re-coding of instructions and does not require architectural hardware support. Here the virtualization is achieved by shifting the O/S from ring 0 to ring 1. Hence the paravirtualization is also called ring deprivileging. This allows the VMM to be executed in ring 0. However, this

1. The last well-known O/S for the x86 architecture that made use of ring 1 and 2 was OS/2.

approach requires modifications to the guest O/S to execute them in ring 1. Applications can still run without modifications because the *Application Binary Interface (ABI)* of the O/S is not changed.

Figure 3.9 shows the modified ring protection scheme. To improve the further reading, the depicting is changed from rings to layers. The O/S is now executed in ring 1 while the VMM can operate in the most-privileged ring 0.

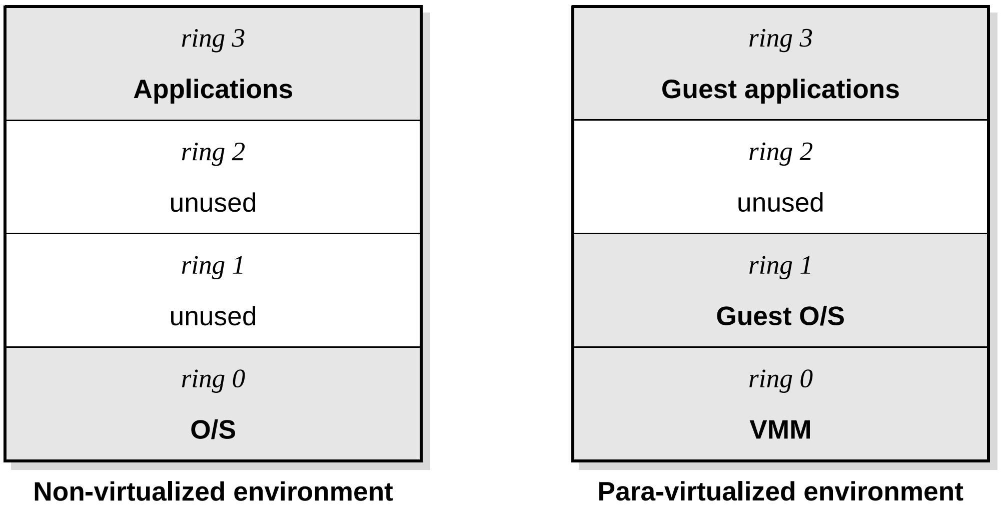


Figure 3.9 Protection scheme for para-virtualized environments

Virtualization in O/S level. If all guest systems are of the same O/S type (e.g. Linux) then virtualization can be achieved completely in O/S level. All modern O/S are designed to isolate and secure different application to prevent them from interaction. These principles are extended and allow applications to be run in guest environments. Applications as guests view their environment as a stand-alone system. While the paravirtualization and full virtualization approach run multiple kernels, the approach here executes only one kernel. The single-kernel approach introduces a much smaller overhead, but as already mentioned it only allows guest systems of the same type.

Full virtualization with architecture extensions. Hardware support for virtualization extends the x86 architecture by adding two new forms of CPU operation: the root operation and the non-root operation [58]. Both operations support all four protection rings. Guest O/S and their applications can occupy all four rings. Instructions executed in non-root operation mode are deprivileged compared to the root operation mode. Hence the VMM is supposed to be executed in root operation mode while a guest runs in non-root mode. The transition from non-root to root mode is called *VM exit* and the opposite way *VM entry.* A control structure manages VM entries and exits in non-root operation mode. In this structure the causes for VM exits can be specified, like privileged instructions or events. This allows to implement several virtualization techniques.

The most important benefit of hardware virtualization support is that paravirtualization or dynamic translation techniques are no longer required. Each O/S can be run on a VM without any modification, neither static or dynamic ones. The VMM can execute in a separate operation mode. Figure 3.10 shows the protection scheme for a virtualized environment with hardware support. Here an example guest O/S which occupies only ring 0 and 3 of the non-root mode, but all four rings can be used by a guest O/S.

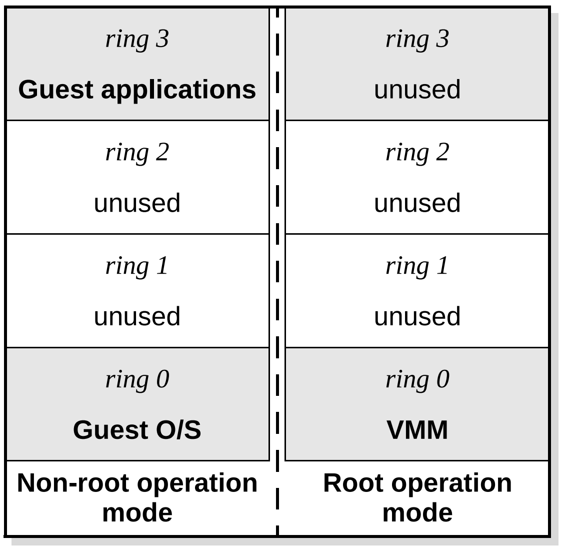


Figure 3.10 Protection scheme for a virtualized environment with hardware support

Examples. There are several VMMs available which use different techniques to virtualize a physical machine. Representatives for the paravirtualization approach are e.g. *Xen* [59] or *Denali* [60]. Examples for the full virtualization without hardware support are e.g. *VMware's ESX Server* [61][62]. The *OpenVZ* project is an open-source representative of the O/S level virtualization technique [63]. The commercial product *Virtuozzo* for Linux is based on OpenVZ. Hardware virtualization techniques are e.g. *AMD's Pacifica Technology* [64][50] or *Intel's Virtualization Technology* [58] (formerly known as *Vanderpool Technology*).

3.1.7 Virtualized devices

According to [59], "virtual devices are elegant and simple to access". This is true as long as the access to the virtual devices involves the VMM. In this case the data between guest and VMM is transferred using asynchronous I/O rings and an event mechanism replaces the hardware interrupts. The VMM schedules the accesses from all guests to the device. This introduces an additional amount of management overhead. User-Level

Communication is only possible for non-virtual devices which can only be used exclusively by one guest. Device sharing combined with User-Level Communication is not possible.

Typical approaches target to virtualize a common Ethernet controller so that all guests can access it. For Xen the approach is documented in [65]. Here the device driver runs in a *Isolated Driver Domain* (IDD) or in other terms a driver-specific VM (see figure 3.12). Driver faults are then limited to this VM and cannot effect other ones. The performance regarding bandwidth is acceptable: the driver achieves 100% for transmit and about 70% for receive compared to an unmodified Linux implementation. Several techniques are proposed in [66] to improve the receive performance. In [67] the approach to virtualize devices for *VMware Workstation* is documented. The approach is similar, special drivers multiplex the accesses from guests to the device. The required process context switches lead to a degraded performance, increasing latency and CPU utilization.

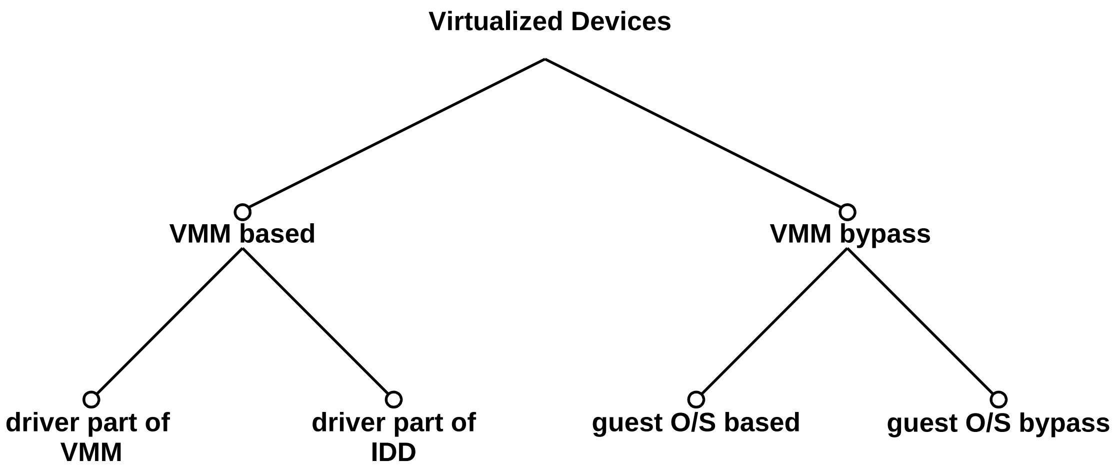


Figure 3.11 Design space of virtualized devices

Several approaches already target to virtualize a device with the VMM bypassed and not included in the communication between guest user-level or guest kernel-level processes and the device. For this approach hardware support is required and the bypassed VMM reduces the overhead for device accesses. In the following two solutions are presented together with the results regarding performance. The performance increase shows the benefit of off-loading virtualization complexity into hardware and bypassing any software layer from a guest's user-level process to the virtualized device.

VIA and Infiniband. The *Virtual Channel Interface* (VIA) [68] already specifies a kind of virtualization. Here the interface between processes and the device are Q*ueue Pairs* (QP). The VIA specification states that up to 2^{16} queue pairs per device are possible. Under certain conditions QPs are limited to single destinations, but in general one QP is sufficient for a process to communicate. So this specification enables up to 2^{16} processes to use the device. Beside the number of QPs no methods regarding implementation are specified, and no hardware was implemented supporting this large number of QPs. Successor of VIA is

Infiniband [31]. IB also specifies up to 2^{16} QPs. For IB there are hardware implementations supporting this large number of QPs. The implementation details are again not specified, and the methods developed by the manufacturers like Mellanox or Voltaire [32][33] are not published.

The development of a VMM-bypass I/O technique for Infiniband adapters shows the potential of virtualized devices [69][70]. This approach relies on the large number of QPs offered by this Infiniband adapter. Such a virtualized device allows direct access from several guest systems, including kernel- and user-level processes. The design is based on the idea of paravirtualization, but here the hardware interfaces of existing devices are not preserved. Instead *guest modules* are used to handle the privileged accesses to the device. These guest modules are part of the guest O/S. A *backend module* in the VMM provides the access for different guest modules. For non-privileged accesses the guest application can then directly access the device (see figure 3.12). The bypassed VMM allows User-Level Communication and this results in almost no performance degradation regarding bandwidth and latency.

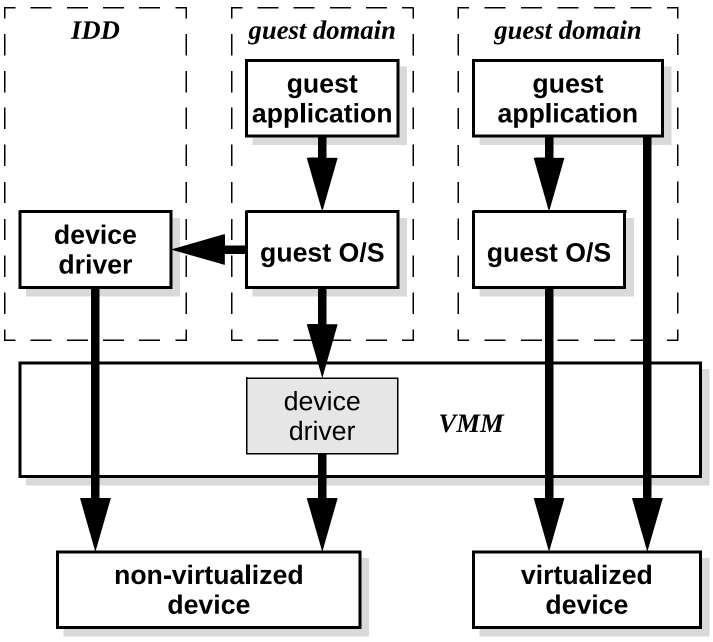


Figure 3.12 Device access in VM environments

Virtualized Ethernet network interface. Another approach to off-load the virtualization functionality to hardware is a device based on the Intel IXP2400 network processor with multiple specialized internal communication cores [71]. There are 8 of these cores and each core can hold 8 contexts. The communication cores serve as Ethernet network interfaces. Up to 64 guests can directly access the device over Virtual Interfaces (VIF) to exploit the internal concurrency. Since the functionality of the cores is Ethernet based this work does not include support for User-Level Communication. But a guest O/S can access the device directly without VMM involvement (see figure 3.12). The results compare latency and bandwidth with and without the self-virtualization technique. The latency is approximately cut in half and the bandwidth increases about 50%, too. This shows again the impact of off-loading virtualization complexity from VMM to hardware.

The virtualized Ethernet device developed in [71] is called *self-virtualizing device*. This term is most suitable to describe the desired functionality. For a self-virtualizing device no VMM support is necessary to allow access from multiple guests to the device, the VMM can be bypassed. Instead the device offers several replicated ports to the host system. For the host, these ports are devices.

The goal of the device replication presented here matched exactly the functionality of a self-virtualizing device. The only but major difference to the development in [71] is that for the device virtualization User-Level Communication is an essential goal.

3.2 ARCHITECTURE

One of the goals of this work is to develop an network interface architecture that supports simultaneous access by a large number of user-level processes. This leads to a virtualization of the device by offering a large number of ports towards the host side. Each port presents the illusion of a complete exclusive device for a process. Only the device driver is aware of the virtualized device. The large number of simultaneously accessing processes lead to a high degree of Thread Level Parallelism. Several work requests issued by one process increase the amount of Instruction Level Parallelism. Both kinds of parallelism should be perfectly exploited by the underlying hardware architecture without limiting it to one of them.

The architecture is one of the key elements of the complete design. Comparable to modern processor architectures, it must be chosen carefully with all possible circumstances in mind. Goal is to find an architecture fulfilling all goals and still providing an excellent performance for all use cases. In the following the requirements for this architecture are shown, derived from previous considerations and the goals of sub-chapter *“1.5.4 Goal summary” on page 30*:

- Support for simultaneous accesses
- Exploit the available parallelism
- High and unconstrained utilization of resources, independent of number of processes
- Support for Virtual Machine Environments, bypassing VMM layers
- Access method based on fast context switching
- Queue-based interface with descriptors for work requests and notifica-tions.
- Mem-less design with on-device cache structures

In order to find and develop a most suitable architecture, the first sub-chapter will present a recapitulation of modern processor architectures. The kinds of parallelism are introduced together with parallel architectures that try to exploit as much of the offered parallelism. The most suitable architecture found is then used as framework to develop a network interface.

3.2.1 Modern processor architectures

One of the most important goals of modern processor architectures is to raise the number of *Instructions Per Cycle (IPC)*. This number tells how many instructions retire per cycle. It is an indicator how many instructions can be processed in parallel and simultaneously by an architecture. The recent architecture developments try to boost performance by raising this number. With a higher IPC number an architecture can exploit more available parallelism of a workload. There are different kinds of parallelism and some architectures are limited to certain kinds of parallelism.

A classical *von-Neumann architecture* can only exploit less than one IPC, because several sequential steps are required to perform an instruction [72]. The minimal steps are instruction fetch, fetch of operands, instruction interpretation and instruction execution. The IPC can be improved by the introduction of pipeline stages. Here the different steps of an instruction are processed in parallel in a pipelined manner. This leads to a peak IPC of one. An IPC of one includes no parallelism. To exploit parallelism and further raise the IPC number, more sophisticated techniques have to be analyzed. But prior to this, the different kinds of parallelism are shortly introduced.

Parallelism can be separated into two main classes. The classes are *Instruction Level Parallelism (ILP)* and *Thread Level Parallelism (TLP)*. ILP is also known as *Fine Grain Parallelism*, while TLP is known as *Coarse Grain Parallelism*. TLP occurs when several processes (or threads) issue instructions to a resource. These instructions have no data dependencies, because they belong to different processes. Hence TLP is typically only limited by the number of run-ready processes. While TLP is the parallelism of several processes, ILP is the amount of parallelism comprised in one process (or thread). Here the parallelism is typically bounded by data dependencies. Only instructions which are independent of each other and have no data dependencies can be processed in parallel. Otherwise hazards might occur which lead to misbehavior.

The *superscalar architecture* extends a pipelined design by including several (possibly different) *Functional Units (FUs)* which can process instructions independently. A superscalar architecture allows the dynamic scheduling of instructions, which is performed at run time. ILP can be exploited and allows to raise the IPC above one. The IPC is limited by the available amount of ILP and the number of FUs. TLP cannot be exploited with this design, because only one process can use the resources. Several processes are executed in a time-sliced manner. A superscalar architecture has several prerequisites. It must be possible to fetch, dispatch and issue enough instructions to utilize the FUs. The same applies for instruction completion. The instructions must be analyzed in run-time for dependencies. A scoreboard can be used to find dependencies and stall instructions which are dependent of prior instructions. Independent instructions are processed out of order.

Essential for the exploitation of parallelism, independent of ILP or TLP type, is the possibility to issue several instructions per cycle. A popular visualization of scheduling for such *multi-issue architectures* is shown in *Figure 3.13 on page 84*. Each issue slot represents the possibility to issue an instruction per cycle. The instructions to be issued in parallel can be either detected and scheduled dynamically in run-time or statically by the compiler. In the past the dynamic scheduling has proved to be more successful due to it's flexibility[1], hence the following will focus on this. For completeness, a typical representative of static scheduling is the *Very Long Instruction Word (VLIW)* architecture. Here an instruction word is composed by several independent instructions. The compiler

1. For instance it is very difficult to foresee the circumstances of the instruction processing. Cache effects and code branches are two of the main reasons why dynamic scheduling typically performs better.

has to find suitable instructions. Typically limitations do exist regarding the instruction types which can be processed in parallel. One of the most popular VLIW architectures of today is the Intel Itanium family [73][74][75][76]. The Itanium relies on an excellent speculative execution support to increase the number of parallel executable instructions.

As mentioned before, ILP is very limited due to dependencies inside the single instruction stream. TLP is less restricted and exists in an increasing number of applications. A multi-programmed workload may consist of several independent programs. Another example is a parallel program, which is divided into several processes or threads. Because superscalar architectures can only exploit the limited ILP, other approaches are required to exploit the TLP.

A representative to exploit TLP is the *Multi-Threaded Architecture (MTA)* [77]. An MTA can hold several contexts of multiple threads in register sets. The primary goal of MTAs is to hide the latency of operations. The multiple register sets allow the MTA to switch to another thread if a long-latency operation occurs. MTAs can be classified in coarse grain and fine grain multi-threading. For the first the time required to switch a context is about 4-6 cycles, which is too long to hide the latencies of brand mispredictions and on-chip cache misses. Fine grain MTAs achieve a context switching latency of one cycle by using sophisticated hardware modules. MTAs can also make use of multiple FUs to exploit ILP. The difference between a superscalar and a multi-threaded architecture is that the first requires O/S support to switch between processes. The latter can hold several thread contexts and switch autonomously between them. O/S involvement results in switching times in the range of microseconds.

A recently on the consumer market emerging architecture is *Chip Multi-Processing (CMP)* [78], also known as *multi-core processors*. Here a complete superscalar architecture forms a building block or core. These cores are replicated on die. A single die now contains several superscalar cores which are seen by the O/S as multiple processors, comparable to SMP. ILP is explored by the superscalar architecture of each core, while TLP is exploited by the replicated cores. The replication has the advantage to re-use modules, which does not increase the design complexity.

An approach resulting in the combination of superscalar and multi-threaded architectures is *Simultaneous Multi-Threading (SMT)* [79]. Like MTAs it allows to hold several thread contexts to switch efficiently between them. But while MTAs and CMPs partition the available resources between the threads, SMT does not. All resources are shared, including register file, caches and FUs. This dynamic resource sharing outperforms the static resource partitioning by increasing the utilization of the resources.

For a closer analysis of the different properties of the presented architectures, the *partitioning* of the issue slots [80] of the different FUs is analyzed. Issue slots represent an available resource to process an instruction. The number of issue slots is typically equal to the number of FUs. The following figure compares the four architectures introduced above by their issue slot partitioning. In the example in *Figure 3.13* it is assumed that there are 4 FUs. So 4 issue slots are available per time slot, each represented by a square. Empty squares indicate that this issue slot is not used. Otherwise the color of the filled square shows the thread which occupies this issue slot.

For an improved comprehension of *Figure 3.13* two properties have to be introduced, the horizontal and vertical waste. The *horizontal waste* is the number of unused slots per cycle. The *vertical waste* is the consecutive order of unused slots. Horizontal waste is typically a result of resource (or issue slot) partitioning. Only one thread can occupy the issue slots in one time cycle. Time partitioning results in vertical waste. Such a temporally partitioning assigns the slots of each time cycle to another thread.

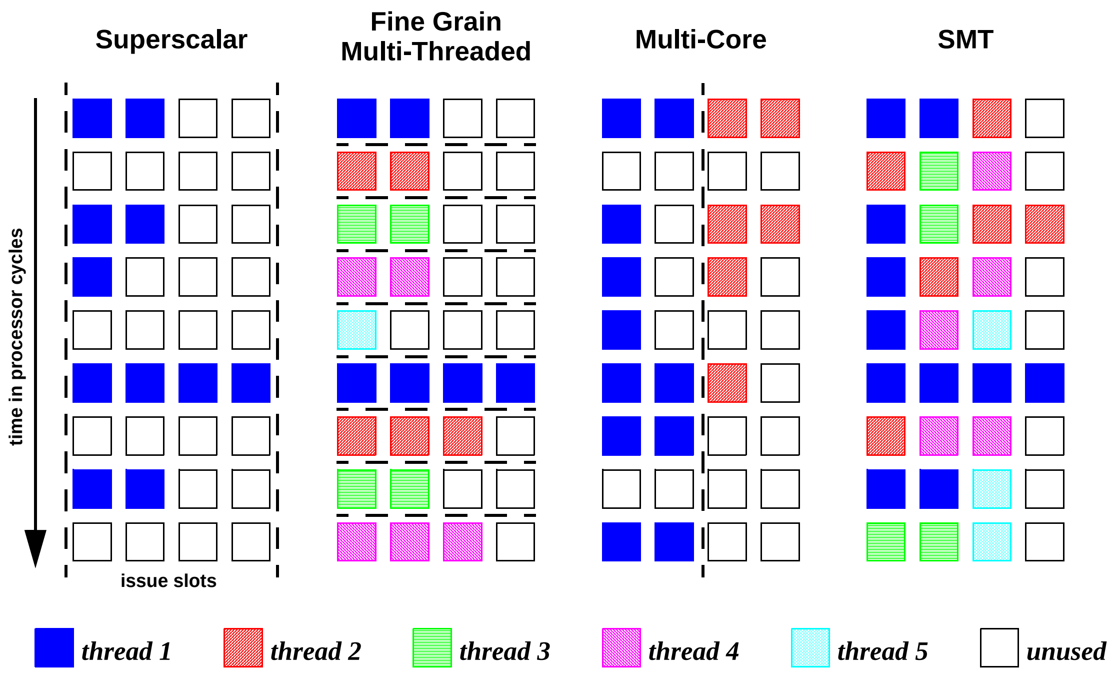


Figure 3.13 Issue slot utilization

In *Figure 3.13* is can be seen that all kinds of partitioning result in lower utilization. Partitioning is indicated by the dashed lines. The superscalar architecture suffers from a large horizontal and vertical waste by limiting all issue slots for a large number of consecutive time cycles to a single thread. Only ILP can be exploited. ILP is limited by instruction dependencies and this results in a large horizontal and vertical waste. Not shown in this figure is the switching to another thread. Thread switching for the superscalar architecture is extremely expensive. O/S is involved and the amount of required cycles cannot be visualized in this analysis. Compared to the superscalar architecture the MTA reduces the partitioning. The temporally partitioning is no longer based on blocks of time cycles. The threads are switched every time cycle, hence a lot of TLP is exploited. In this example the first issue slot is always occupied and the vertical waste is significantly reduced. Because MTAs allow no resource partitioning all issue slots of a time slot have to be occupied by the same process. If the amount of ILP is not sufficient, not all issue slots can be occupied which results in horizontal waste. The different approach of the CMP architecture reduces the horizontal waste by it’s replicated cores. ILP is exploited by the underlying superscalar architecture of the cores and the TLP by the replicated cores. Again there is resource and temporally partitioning which leads to unnecessary restrictions. The costs for thread switching are comparable to the superscalar architecture and involve O/S traps. In contrast to these architectures the SMT architecture does not partition resources at all. It allows the dynamic scheduling of threads to different resources. The impact of the removed partitioning can be seen in the figure. Each time slot threads can occupy different resources which leads to a very high resource utilization. Both ILP and TLP is exploited without restrictions.

All architectures expect SMT unnecessarily differentiate between ILP and TLP. The temporally and resource partitioning restricts the exploitation of the parallelism, resulting in sub-optimal resource utilization. Only SMT removes all partitioning and does not differentiate between ILP and TLP. The result is the highest resource utilization. The disadvantage of SMT are very high design costs. All resources have to be tagged to identify the corresponding thread. Replication of modules to reduce design costs is very limited. The thread competition for resources might also result in unwanted contention and interference. In [80] several software techniques are shown to limit these effects.

The architecture of the SUN Niagara [81] processor combines the advantages of the SMT and the CMP architecture. The architecture is called *Chip Multi-Threading (CMT)* [82][83]. The cores of this processor are no longer only superscalar, instead they are based on the SMT architecture. Four threads can concurrently run on each core. The processor houses up to eight cores in total, in total a 32-way threaded architecture. The successor Niagara-2 will house 64 threads in total and remove some of the restrictions of Niagara-1. The idea behind CMT is quite straight-ahead. SMT has the excellent advantage of unrestricted instruction issue from different threads. Scaling an SMT architecture is difficult due to the high design costs and the tagging of all resources. Because ILP is typically limited it is also not essential for the utilization. Each SMT core of a CMT architecture can exploit the ILP of the workload. The TLP is exploited by the replicated cores. On the one hand this introduces partitioning again, but the ILP characteristics limit the negative effects. Scaling the design is easier because the cores can be used as building blocks.

3.2.2 Basic architecture

The result of the previous analysis of modern processor architecture showed that the SMT architecture supports both ILP and TLP. It perfectly fits to the situation when only one process is using the device with several simultaneous instructions, as well as several processes simultaneously using the device with one or more instructions. Furthermore it does not differentiate ILP and TLP. So the SMT architecture is proposed as the optimal architecture for a virtualized peripheral device. Other research of network processor architectures show similar results, but the work focuses on handling IP traffic [84].

If the final implementation of this design shows that there are still unused resources in terms of silicon area, it might be considered to replicate the SMT architecture. This replication of modules is similar to CMP and leads to an CMT architecture. Care has to be taken on the on-chip network connecting the modules and on the cache design. The caches can be shared among all modules or replicated together with the modules, which leads to exclusive use.

The alternative to a custom SMT architecture with application-specific FUs is a device based on a commodity network processor. Typical network processors offer FUs specialized for networking application and are configured using microcode. A lot of networking processors are available, but they do not offer enough parallelism to fully exploit all available parallelism without limitations. Additionally a complete application-specific design result in an increased performance so the decision is made in favour of a custom design.

Functional units. The FUs are one of the key components in this architecture. They process the work requests of the processes. Comparable to the communication and synchronization work flow in sub-chapter *“2.3.1 Work flow” on page 42* there are three types of FUs:

1. *Requester unit*: Work requests from processes running on this node (origin) are dispatched to requester units. A requester unit sends out requests to remote nodes (target). The type of a request determines if it is processed by a responder unit or a completer unit at the target.
2. *Responder unit*: A responder unit responds to a request coming in from the network side. A response is always sent back to the origin of the request. A response is processed at the origin by a completer unit.
3. *Completer unit*: A completer unit is the final stage of a work request, independent if requests or responses are coming in. For both cases the work is completed and no further action between target and origin takes place[1].

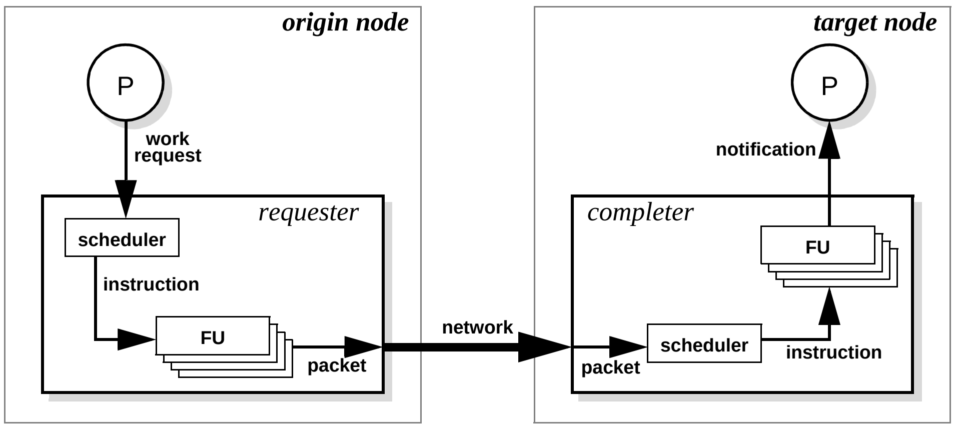


Figure 3.14 FU scheduling for a Requester/Completer scheme

1. Except the optional acknowledge.

From the architecture's point of view the most important difference between these three types of FUs is the scheduling. The scheduler of the requester units distributes the work requests of processes. The scheduler of the responder and completer units only dispatches requests and responds coming in from the network side. In the following all FU types process *instructions*. For the requester an instruction is a work request, for the two other an incoming packet. Beside the source of the instruction the remaining properties are identical.

Figure 3.14 depicts the work flow for a Requester/Completer scheme. On the local node (*origin node*) the work request is dispatched as an instruction to a FU. The FU generates a packet which is sent over the network to the remote node (*target node*). The command section of the packet is interpreted as an instruction and again dispatched to a FU. Here the work is finalized and a notification is generated to inform the target process.

The work flow including a Responder is shown in figure 3.15. Compared to the Requester/Completer the same steps are performed, but now the origin process is the final stage. The Responder receives a packet from the Requester, interprets the command section as an instruction, dispatches the work to an FU where a response packet is generated. This is sent back to the origin node.

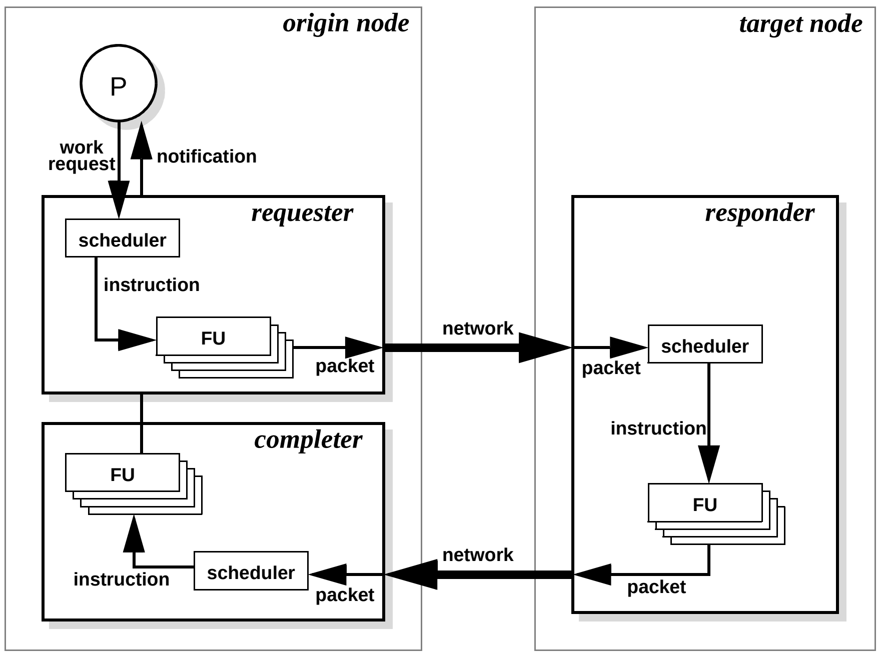


Figure 3.15 FU scheduling for a Requester/Responder/Completer scheme

A most unrestricted usage of an FU for all communication functions (Send/Receive, RMA Put and Get) is only possible if the FU is not specialized for one function. Hence each of the three FU types must be suitable for all communication functions. This unified design also reduces the design costs because modules can be re-used by replication. Specific FUs are also limited to one kind of communication. This partitioning of FUs results in a reduced scheduling flexibility.

A pipelined design of the FUs has several advantages. The partitioning into pipeline stages reduces design complexity, similar to the divide-and-conquer approach. The throughput is increased because several instructions can be processed in parallel: while one instruction is in a later stage, the next instruction can already be prepared in the previous stages. A naturally approach to divide a module into pipeline stages is a *functional decomposition*. By doing this all three FU types are pipelined to increase the throughput.

3.2.3 Context

The SMT architecture requires a multi-threaded design. A multi-threaded design holds several contexts. On this note a *hardware thread* is a loaded context. A SMT architecture allows multiple loaded contexts and thereby parallel processing. The hardware does not differentiate between software processes and software threads. If the application is threaded it must also ensure thread-safety.

The configuration set of a hardware thread is completely stored in one context. If an FU is scheduled to process work for a process, a fast context switch for only this FU occurs and configures it for this specific thread. This way each FU can process work from different processes. The analysis in sub-chapter *"3.1.1 Device access methods" on page 66* also showed that *context switching* is most promising. The fine granularity of the small sized contexts also allows to easily store them in main memory and fetch them only if needed. A cache can store the most frequently used contexts to reduce the average penalty for context switches. This is inline with the decision for a mem-less design with on-device caches as a most cost-effective solution, which was shown in sub-chapter *"3.1.5 On-device memory" on page 71*.

In a context all informations required to process the work of a process must be stored. This includes capabilities, a status word and all informations about the data structures of this process. A data structure is described with it's location (either main memory or on-device memory), offset and length. If a read/write pointer scheme is used to identify valid entries it must also be part of the context.

For an efficient implementation the context should fit into one or more *cache lines*. The size of a cache line is 64 byte for almost all modern processors. A cache line is the smallest unit for coherency and therefore the smallest unit which can be accessed. A context should fill this structure as efficiently as possible.

3.2.4 Memory hierarchy

The decision for a mem-less design (see *"3.1.5 On-device memory" on page 71*) also includes to store all data structures in main memory. Pinned and contiguous memory regions must be used for these data structures. A device as well as a user-level process can directly access the structures without O/S involvement. To ensure security a process is not allowed to access any data structure except it's own, which is in particular true for the data structure containing the contexts.

Caches are used to reduce the average access costs to the data structures in main memory. Replacement strategies are required due to the limited size of a cache. Using a sophisticated replacement strategy the hit rate of a cache is increased by keeping the most recently accessed entries in the cache. The same strategies as used in modern processor architectures are also suitable here.

The device can only access physical addresses, virtual addresses are not possible. This occurs when processing work requests with RMA operations. One possibility is to perform the address translation by the device driver. The device can notify the device driver using an interrupt to translate a virtual address into a physical one. This must be done for every single page, because the virtual address space is not contiguous in physical. The resulting overhead is not desirable, but it can be diminished. Comparable to modern processor architectures, an on-device *TLB* is used to store the most recently used address translations. For a CPU it reduces the amount of system calls for translations, here the amount of driver calls.

3.2.5 Functional units

The FUs can be classified into three types. The *requester unit* processes work requests from processes and sends out requests to remote nodes. The type of a request determines if it is processed by a responder unit or a completer unit at the target. A *responder unit* responds to a request coming in from the network side. The response is always sent back to the origin of the request, where it is processed by a completer unit. Hence a *completer unit* processes both requests and responses coming in from the network. It is the final processing stage where the work is completed and no further action between target and origin takes place.

Requester unit. In sub-chapter *“2.4.1 Commands for Requester” on page 56* the commands are listed which have to be processed by a requester unit. The following steps are required in a request unit to process an instruction. These are basically the same steps as in sub-chapter *“2.4.3 Flow for Requester” on page 58*, but two steps are added. One for the context switch at the beginning and one to write-back the modified context. The procedure starts with the scheduler issuing a context identifier to the FU. This selects a context where all further informations are stored. The next instruction to be processed is identified using pointers stored in the context.

1. Context load
2. Descriptor load
3. Routing fetch
4. Routing output to network
5. Command frame generation
6. Command frame output to network
7. Window descriptor load

    7-a. This implies a short check if the window is locked.

   7-b. For a locked window the processing is aborted, the descriptor is marked erroneous and together with the context written-back.

8. Repeat for each page of the payload:
   8-a. Translate page offset
   8-b. Fetch data for this page
   8-c. Output data
9. Mark descriptor completed and write-back
10. Context write-back

Some of the steps above can be processed in parallel which is desirable to reduce processing latency. Furthermore not all steps are required for all instructions. An RMA operation uses all the steps above but for a Send operation the window fetch and address translation can be skipped. Generally, the RMA Put and Get operations are the most complex ones. They are considered for the pipeline structures. The other operations can also be processed, but for them certain stages are skipped. This is also inline with the design goal of one general purpose request unit. The pipeline structure of a requester unit is shown in figure 3.16.

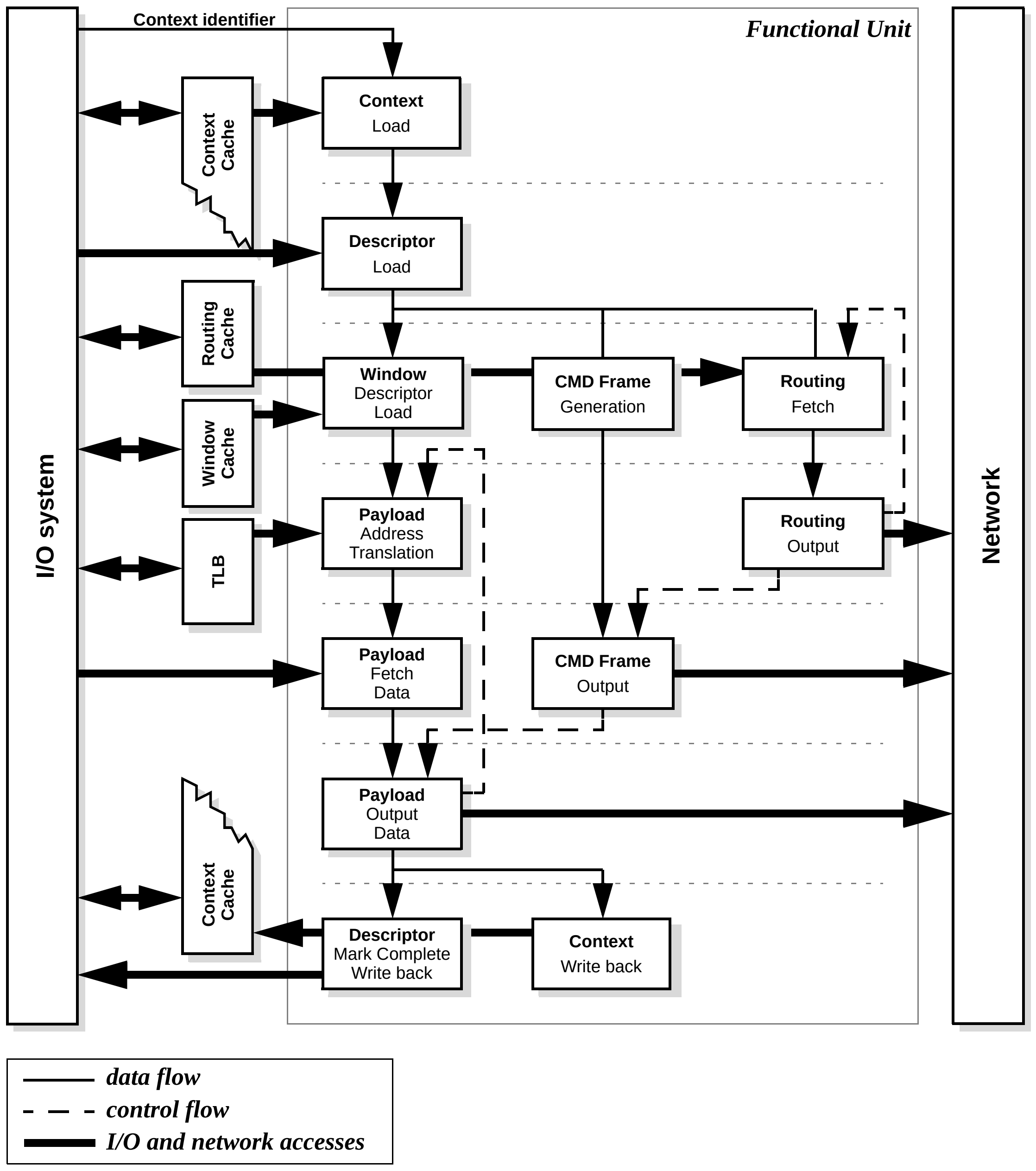


Figure 3.16 Pipeline structure of a requester unit

Responder unit. The command set for a responder and a requester is identical (see subchapter *“2.4.4 Commands for Responder” on page 59*). But for a responder the instructions come from the network side, for a requester they are issued by processes on the same node.

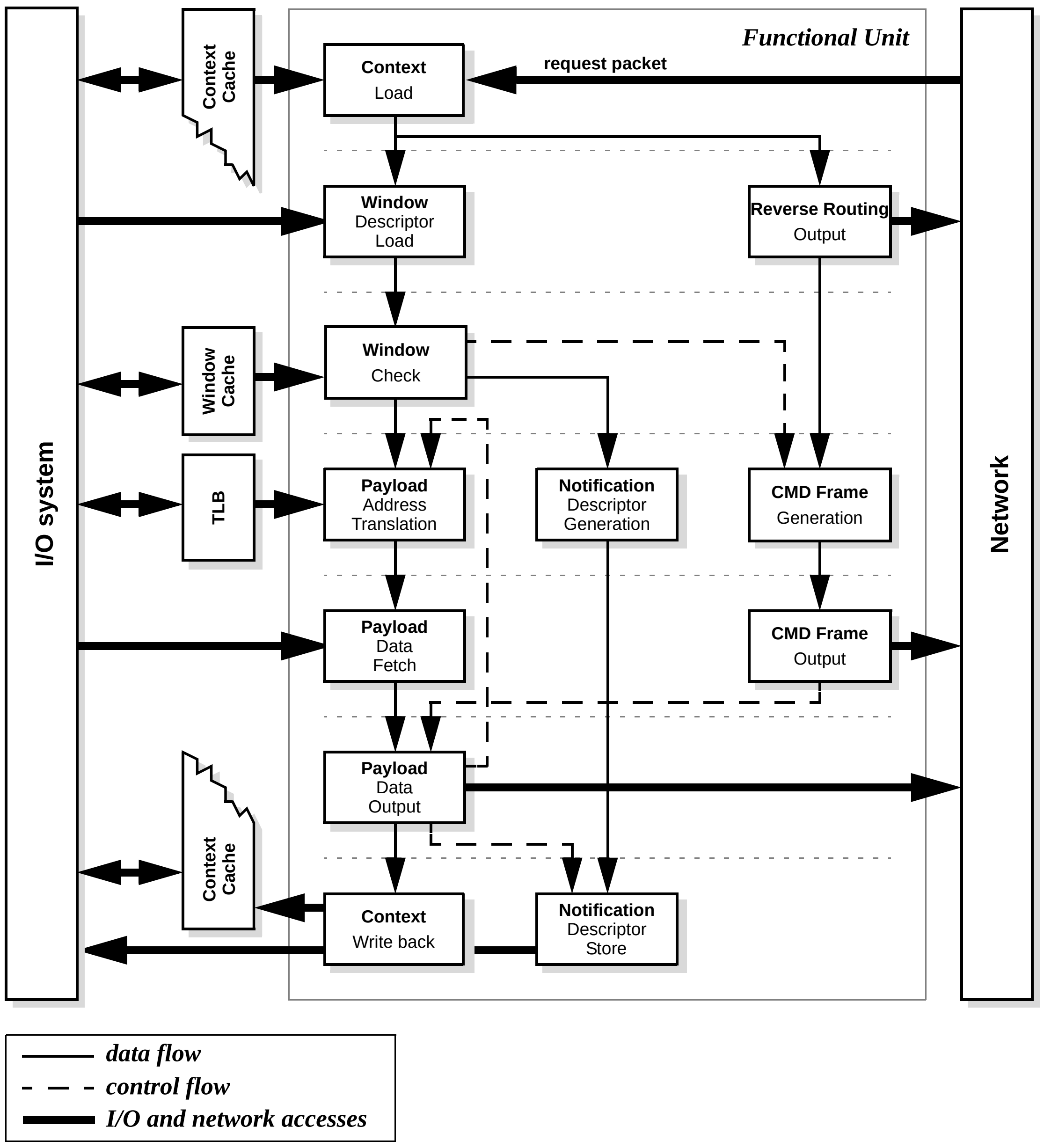


Figure 3.17 Pipeline structure of a responder unit

The required steps for a responder are shown in sub-chapter *"2.4.6 Flow for Responder" on page 60*, but again two stages are added for context load and context write-back. It starts with a packet coming in from the network which is scheduled by to a FU. The command frame of the packet contains informations about target context, window and requested data. Then the response FU starts with:

1. Context load
2. Window descriptor load
3. Window check
    - 3-a. If the window check fails, the processing is aborted. A negative acknowledgement is sent back to inform the origin of the failure. Furthermore a notification descriptor is generated informing the target of a failed RMA operation.
4. Reverse routing output to network
5. Command frame generation with response command
6. Command frame output to network
7. Repeat for each page of the payload:
    - 7-a. Translate address for each page
    - 7-b. Fetch data for this page
    - 7-c. Output data for this page
8. Notification descriptor generation
9. Notification descriptor store
10. Context write-back

An analysis of possible overlapping steps leads to the pipelined design of a responder unit like shown in figure 3.17.

Completer unit. Like shown in sub-chapter *“2.4.7 Commands for Completer” on page 61* the completer processes commands generated by requester and responder units, coming in as packets from the network side. Again a scheduler assigns packets to a free unit, where the work is finalized. Only an optional acknowledge is inserted again into the network.

The steps from *“2.4.9 Flow for Completer” on page 63* are combined with stages for context load and write-back, and in figure 3.18 the resulting pipeline structure is shown.

1. Context load
2. Window descriptor load
3. Window check
    - 3-a. If the window check fails, the processing is aborted. If the current node is the target a negative acknowledgement is sent back to inform the origin of the failure. Furthermore a notification descriptor is generated informing the local system (origin or target) of a failed RMA operation.
4. Optional acknowledge generation and returning:
    - 4-a. Reverse routing output to network
    - 4-b. Command frame generation with acknowledge command
    - 4-c. Command frame output to network
5. Repeat for each page of the payload:

5-a. Translate address for each page
5-b. Receive data for this page
5-c. Store data for this page

6. Notification descriptor generation
7. Notification descriptor store
8. Context write-back

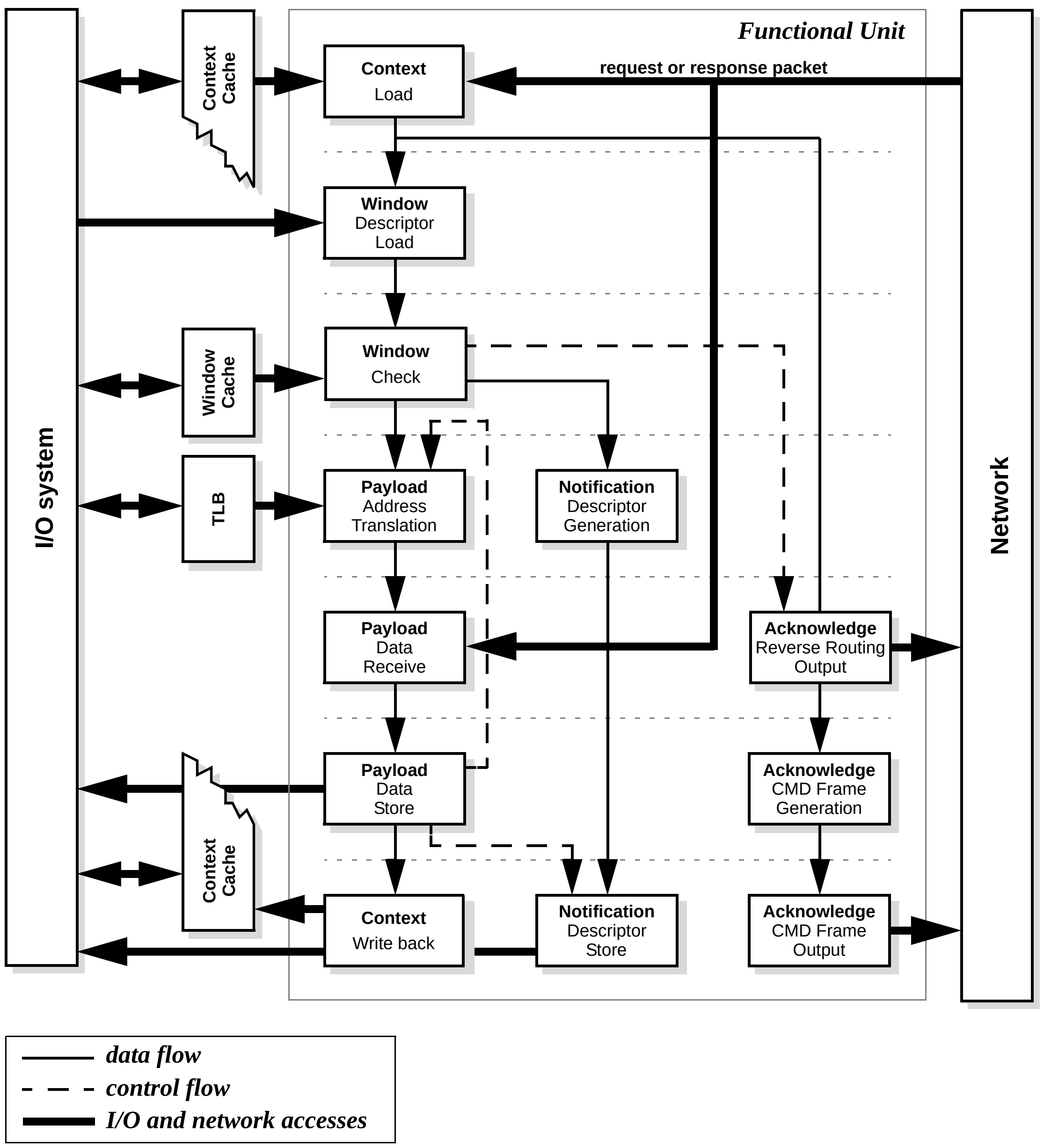


Figure 3.18 Pipeline structure of a completer unit

3.2.6 Scoreboard

In sub-chapter *“2.1.6 In-order and out-of-order delivery” on page 36* the requirements for in order and out-of-order processing of work requests are analyzed. The decision is to support both. While out-of-order processing requires nothing because all dependencies are removed, in order processing requires hardware support. The ordering module can be disabled to allow out-of-order processing.

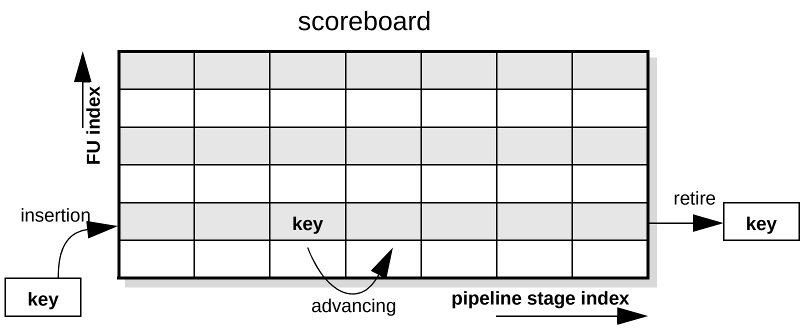


Figure 3.19 Scoreboard

Scoreboard. The problem has many similarities to modern processor architecture, where the dependencies of instructions processed in parallel must be solved. A scoreboard [46] is a solution for this problem. It keeps track of occupied FUs and ensures that dispatched work does not violate the ordering requirements.

If the combination of context, destination and type of several work requests is equal, then ordering is required. This combination is also referred to as *ordering key.* Only work requests with equal keys must be considered to ensure ordering.

The architecture presented here can hold a different context in each pipeline stage of it’s FUs. The scoreboard must supervise all of them. For each pipeline stage of each FU the key is tracked in the scoreboard. The key is inserted upon the dispatch of the work to an FU and advances with each pipeline stage. After completion of the work the key is removed from the scoreboard. Figure 3.19 shows a scoreboard with a key inserted, advancing and finally removed.

A new work request can only be scheduled if it does not violate the ordering. Thus only when no matching key is present in the scoreboard the work can be processed. An optimization leads to issuing a work request with a dependency to the same FU as the matching work is using. Inside an FU no out-of-order execution takes place, hence the second work cannot overtake the first one and the ordering is ensured. This allows to keep the context for this FU and also to schedule the work before the previous is completed.

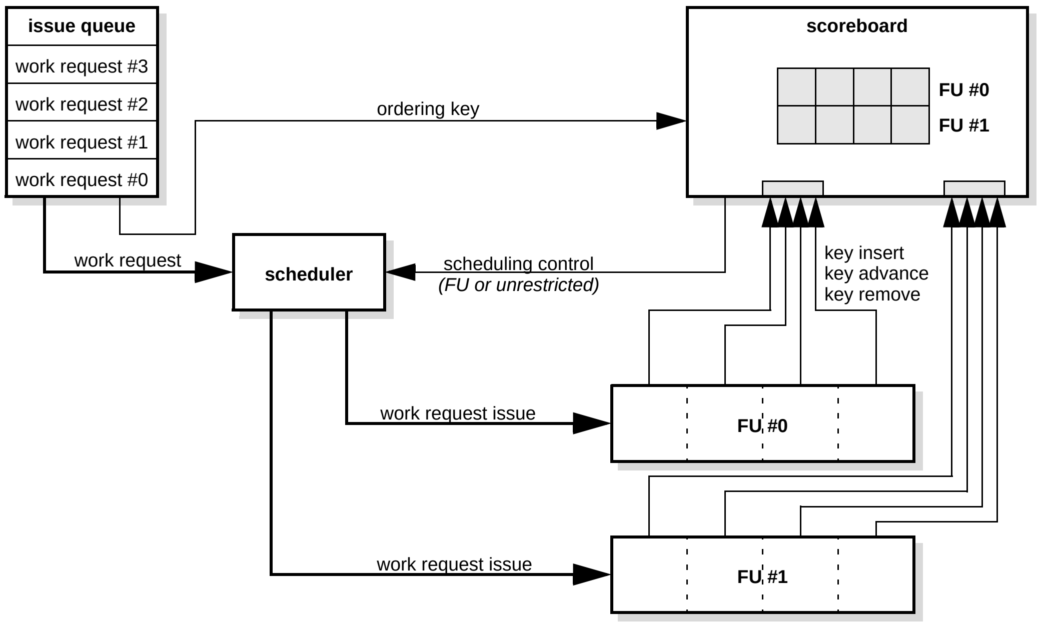


Figure 3.20 Single issue scheduling with ordering control using scoreboard

Figure 3.20 shows the scheduling if a scoreboard is used to ensure ordering. The issue queue is filled with work requests. The next to be issued is work request #0, while work request #3 is the last one inserted. The scoreboard controls the scheduling of the work requests to the FUs by providing restrictions regarding issue. Either work requests can be issued without restrictions or a matching work request is currently being processed in one of the FUs. Then an identification of this FU (for instance, the unique number) is provided by the scoreboard. Using this scheme, work requests with ordering dependencies are scheduled to the same FU.

The complexity of a scoreboard is dependant on the number of FUs and their pipeline stages. To limit the complexity either the number of FUs or the pipeline depth must be reduced. Each pipeline stage of each FU requires an entry in the scoreboard. In the example above, the 2 FUs with each 4 pipeline stages result in a scoreboard with 8 entries.

Reservation Stations. Another approach from modern processor architectures to detect hazards and solve dependencies and conflicts is to use reservation stations [46]. A reservation station is located in front of each FU. Instructions are issued to free reservation stations. Here the work request is waiting for it's resources to become available (in other terms the dependencies are solved).

The goal of a scoreboard and a reservation station for a modern processor is similar. But while the scoreboard is a suitable solution to ensure ordering of work in the architecture here, a reservation station cannot fulfill the requirements. The reservation station working principle is based on solving data dependencies, which inherently ensures the correct ordering of instructions. Here no data dependencies are given, hence this method cannot be applied.

But another ability of reservation stations is the reduction of *Head-of-Line (HOL) blocking*. For a modern processor instructions can be issue to free reservation stations without dependencies. There they wait until all conflicts are solved. The unrestricted issue does not result in blocking of instructions with dependencies in the schedule queue, instead they are immediately issued to appropriate FUs (i.e. to the reservation station of the FU).

The separation of instruction issuing and dependency solving allows to distribute the previously centralized queue to each FU. This can also be applied to the architecture here by introducing wait queues in front of the FUs. But the ordering conflicts must be solved before the issue to the wait queues. Two work requests under ordering control (identified by their ordering keys) must be scheduled to the same FU, otherwise the ordering cannot be guaranteed.

In the example in figure 3.20 a reservation station (not drawn) in front of the FUs allows to perform the issue operation also with the FU occupied. Otherwise the work request is blocked at the head of the issue queue, preventing to dispatch other work requests.

While the approach in figure 3.20 shows a single-issue approach, a multi-issue approach is more sophisticated (see design space in figure 3.21). For a multi-issue approach not only one work request can be issued per time slot, instead multiple ones. This increases the scoreboard complexity because in addition all issue candidates per time slot must be checked for ordering constraints in parallel. The advantage is a reduced HOL blocking, because one blocked issue queue entry does not block the complete queue any longer. Only if as many entries as issue slots are available are blocked, no further issue is possible and the queue blocks again.

Both the single-issue and the multi-issue can be combined with the distributed wait queues derived from the reservation stations. For the single-issue it reduces the HOL blocking significantly. Another approach for single-issue is the use of a look-ahead scheduling. If here the first queue entry blocks, the next is considered for scheduling. This can be repeated and results in a certain look-ahead depth. Comparable to multi-issue, all the following work requests in the queue must be included in the ordering check. This results in a similar complexity of the scoreboard, hence the multi-issue approach is favored because of it’s fever restrictions.

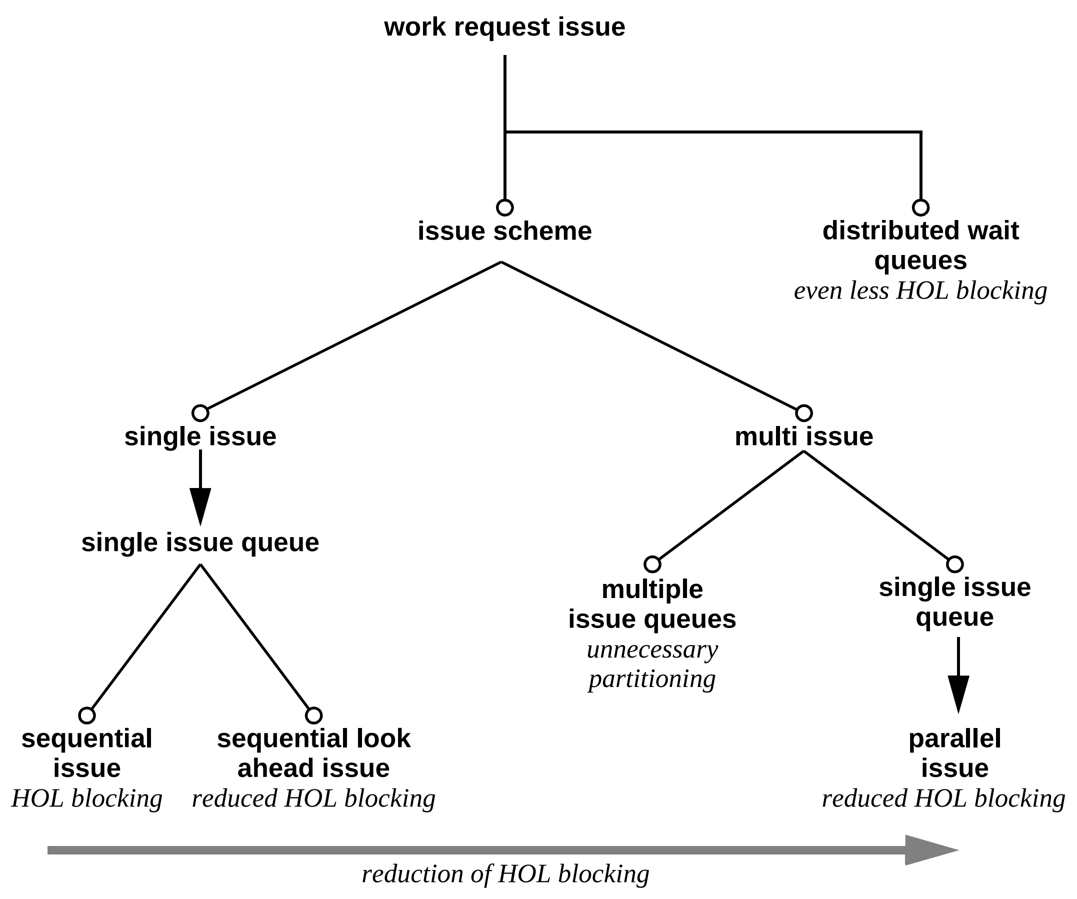


Figure 3.21 Design space of work request issue

While the multi-issue scheme with a single queue does not suffer from single source HOL blocking, the combination with the wait queues diminishes the HOL blocking even more. For unconstrained exploitation the available parallelism, the multi-issue approach with distributed wait queues is favored.

3.3 Queue Design

Queues are used to decouple the processing of the main CPU and the device. In sub-chapter *“3.1.2 Data transfer methods” on page 69* it is shown that the DMA mode off-loads the CPU from the work of transferring data from and to the device. The DMA mode is most efficient when a descriptor-based interface between process and device is used. A work request to be processed is specified by a descriptor (see sub-chapter *“3.1.3 Work requests and notifications” on page 70*). Queues as interface between processes and device allow several outstanding work requests and hence decouple CPU-level and device-level processing. A major design decision is if the queues are located on the device or in main memory (see sub-chapter *“3.1.5 On-device memory” on page 71*).

Previous considerations and the goals of sub-chapter *“1.5.4 Goal summary” on page 30* show that the queue design must include also support for simultaneous accesses. A single queue with multiple producers requires synchronization among them to ensure correct behavior. An efficient solution for the multiple producer problem is inevitable for a scalable solution when the number of producers is scaled.

3.3.1 On-device queues

The solution to allow several outstanding work requests is to introduce queues as interface from CPU to device. For each direction there is at least one queue. The work queue contains the work requests and the notification queue status information, for instance information about the completion of work requests. The queues are organized as *First-in First-out (FIFO)* data structures. Hence the work requests are typically issued in order. Advanced techniques like *reservation stations* or *out of order issue* can reduce the *HOL* blocking and allow multiple instruction issue. An out of order issue does not necessarily leads to out of order execution. A *scoreboard* (see sub-chapter *“3.2.6 Scoreboard” on page 95*) can ensure in-order work processing, without such a control unit the superscalar FUs process their work independently and hence out of order.

A fixed size of a queue entry together with the FIFO structure allows a very simple and highly efficient queue management. Entries of variable size introduce additional overhead in the queue management with leads to more complex logic. If several sizes of entries are needed, the queue design should choose the largest one as granularity. The drawback is a waste of queue space, but the advantage of a more efficient queue management prevails in this case.

Using queues it is possible to have several outstanding work requests. Now the CPU can issue a bunch of work, knowing that the accumulated latency to process all of them is large enough to perform another task. Then the latency of the work requests can be hided. The check for completion can also be done more infrequently.

In general a queue’s structure is not dependant of it’s direction, so in the following the work queue is described. Everything also applies for the notification queue with it’s inverse direction. For the work queue the CPU or a process is the *producer* and the device is the *consumer,* and vice versa for the notification queue.

The work to be processed by the device is described by a *descriptor.* In the case of many parameters a descriptor-based interface has the advantage that the work request can be composed prior to issuing it to the device. A descriptor contains all required information to process a work request, either directly or by referring to other data structures. Typical elements of a descriptor are command fields, sequence numbers, immediate values or pointers referring to blocks of memory. If the data associated with a work request does not fit in a descriptor it is stored in a separate data region, known by both producer and consumer. A pointer to the data and the data length are stored in the descriptor to identify the associated payload within the data region. The consumer uses this information to fetch the data and processes the work request.

The location of the queues is a major design decision. The first approach is to treat the queue-based approach similar to the register-based approach. This means that there are still registers on the device for work request issue and completion. But behind these registers are queues to store the appropriate requests. This requires on-device memory, implemented either as dynamic or static RAM. But the device sees all accesses to the registers and is informed about the insertion or removal of queue entries. A single write is sufficient to insert a new work request. A read can be used to consume an entry of the notification queue. By counting the outstanding work requests the CPU is also informed about the number of free entries. No check for available space in a queue is required prior to an issue operation. The complete scheme for a simple on-device queue design is shown in *Figure 3.22*.

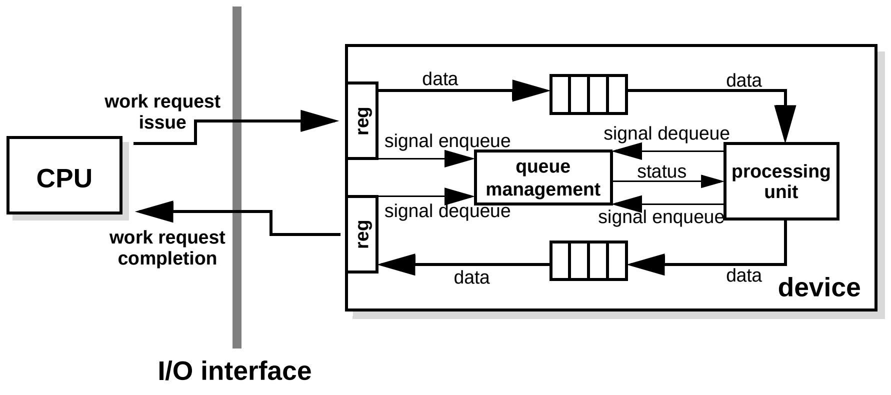


Figure 3.22 On-device queue design

On-device queues are very limited in size. On-device memory is expensive compared to the main memory of the system. Some techniques exist to off-load queue entries into main memory but this introduces significant overhead. To off-load on-device queues each queue entry has to be transmitted in total three time over the I/O interface. This increases the traffic on this bottleneck significantly. Additionally queue off-loading introduces a lot of organization overhead due to the different storage places.

3.3.2 Off-device queues

Main memory has a much better price/performance ratio than on device memory and is hence typically also much larger in size. The costs to access main memory from a device are higher compared to on-device memory structures. Another drawback is that the I/O bottleneck has to be used to access main memory. But the design gets more cost effective and it is possible to implement larger queues. Beside the price/performance ratio of main memory another advantage is the flexibility. Typically main memory can easily be expanded if the need arises. Furthermore it is possible to configure the amount of memory reserved for the queues, at least statically during boot time. Sophisticated systems can even allow to add or remove queue memory. Hence main memory is an alternative to on-device memory.

Several problems immediately arise when moving the queues from on-device memory to main memory. First, the device is not informed anymore about the insertion or removal of entries. A write cycle from CPU to main memory is not visible to a peripheral device. Cache coherency is a solution for this problem, but the device architecture presented here should not be limited to cache coherent I/O interconnects (see sub-chapter *“3.1.5 On-device memory” on page 71*). The device has to be explicitly informed upon every modification of queue entries. This can be as easy as writing a *doorbell* register [68]. For each queue there is a dedicated doorbell so the device immediately knows which queue is addressed.

The second problem is the addressing method. Peripheral devices can only access main memory using *physical addresses*, while a user-level process of the CPU can only use *virtual addresses*. Furthermore a contiguous VA region is not required to be contiguous in the PA space. An address translation is required for every used page to ensure correctness. This is possible but prevents linear accesses from device to memory. While contiguous VA regions are not contiguous in PA space, it is possible to ensure that contiguous PA regions are also contiguous in VA space. This approach requires only an address translation for the first page. The appropriate address together with the length of the region allows linear accesses for both the CPU and the device.

Next problem is the *demand paging* principle, which allows physical pages to be swapped out. Then the VA is still valid but does not have an corresponding PA anymore. At the PA location can be a different page which has just been swapped in. The device itself is not notified of this change. The solution is to prevent all pages of the memory region from being swapped out by pinning them in main memory. Then they are no longer replacement candidates for pages to be swapped.

To summarize the above, the best approach to implement queues in main memory is to use *pinned and contiguous memory regions*.

In main memory a queue is typically not organized as a FIFO structure. The overhead to shift entries is much too high. Instead an organization as a *circular or ring buffer* has proven to be very efficient. Then read and write pointers both for the producer and the consumer clearly indicate the current state of the queue, and the entries are still consumed in order. An upper bound of the queue prevents the pointers from an overflow. Instead they wrap around in this case. The producer compares the read and write pointer to check the queue is *not full*. Only then it can insert a new entry and the write pointer is incremented. The consumer also compares read and write pointer to check that the queue is *not empty*. Only then an entry of the queue can be consumed and the read pointer is incremented to mark this queue entry as free. The write pointer is only updated by the consumer and the read pointer only by the consumer. The producer and the consumer must have access to both pointers. But one pointer is never modified by both, therefore no synchronization is required.

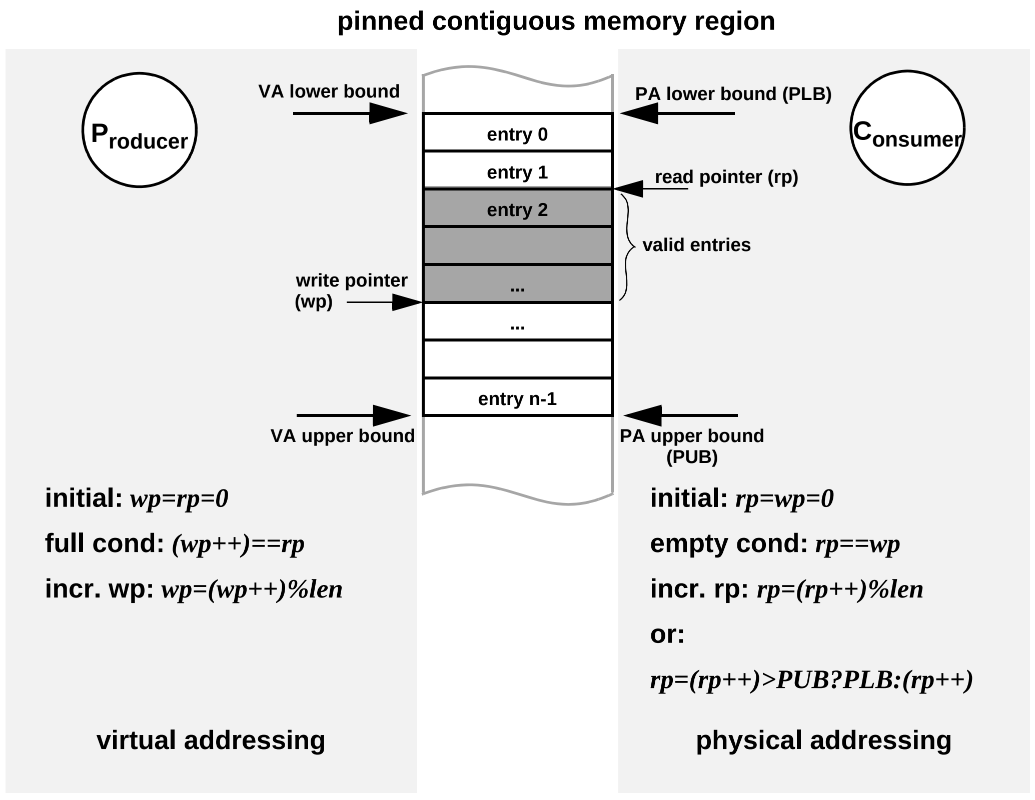


Figure 3.23 Efficient pointer scheme for circular buffers

The complete pointer scheme is explained in detail in *Figure 3.23*. It has the advantage of a very low overhead. The full and empty condition can easily be calculated, as the increment of a read or write pointer. Because the modulo operation is quite expensive[1] to be implemented in hardware an upper-/lower-bound calculation is proposed for the consumer. Then only an add operation followed by a larger-than comparison is needed. For the producer running on the CPU a modulo operation is uncritical.

1. To be more concrete, a modulo operation on the base of 2 is very easy to implement in hardware. But all other bases result in complex and expensive hardware logic. To not limit queue sizes to powers of 2, other solutions are analyzed.

Regarding the location of these pointers it is most effective to store both of them in device registers instead in main memory. The most important advantage is that the device is automatically notified of write pointer changes. If the write pointer is stored in main memory, another device register access would be required for the notification of the change. Polling a main memory location for a change is also not an efficient solution because a lot of I/O bandwidth is wasted. So at least one register access is required and this can be combined with the write pointer update. The producer can hold a copy of the write pointer and only has to write updates of the pointer to the device. Read operations can be avoided. Regarding the read pointer the producer has to fetch it from the device. A lazy pointer scheme is possible, reducing the update frequency of the read pointer copy. For instance, it is possible to update the read pointer copy only when the full condition of the circular buffer occurs (based on the comparison of the current write pointer with an out-dated read pointer copy). An updated read pointer might show that the circular buffer actually is not full.

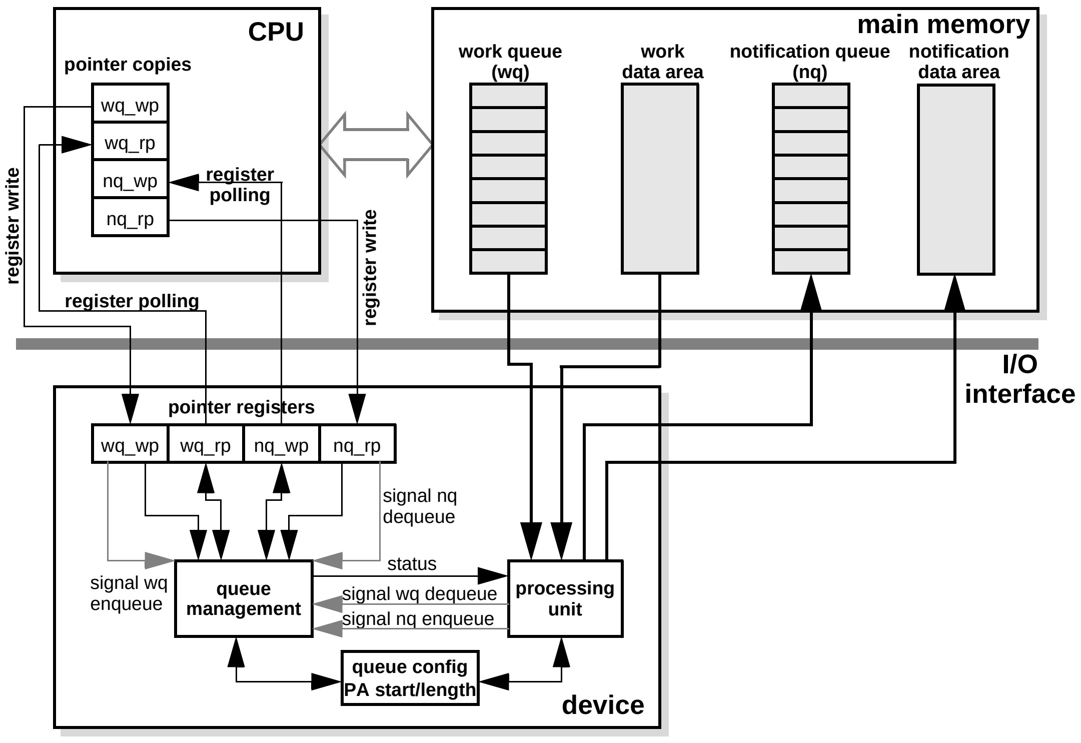


Figure 3.24 Main memory queue design

Figure 3.24 summarizes the complete architecture for off-device queues stored in main memory. All four memory regions (work queue, notification queue and their data areas) are pinned contiguous memory regions. The VAs and PAs of their lower and upper bounds are known to the producer (user-level process on the CPU) respectively the consumer (device). The read and write pointers are stored in device registers. The CPU holds copies of these pointers. Copies of pointers modified by the device are frequently updated by the CPU.

Pointers modified by the CPU are written back to the device registers so the device is notified of the change.

One data area shown is assigned to the work queue. A work queue entry (work request) contains pointers to the data area where it’s associated payload can be found. The other data area is assigned to the notification queue and works identical. The organization of these data areas should be dependant on the organization of the work/notification queue. These are organized as circular buffers. If the data areas are not organized in this way fragmentation may occur, because in-order processing of queue entries can result in out-of-order processing of buffer in the data area. Fragmentation introduces overhead to find contiguous buffer space. If consecutive entries in the queues refer to the data areas in order, the processing of the queues will also consume buffers in the data areas in order.

The CPU polls on the *nq_wp* and *wq_rp* pointer to recognize new or removed entries. This polling has only to take place if the CPU cannot continue it’s work because the work queue is full or the notification queue is empty. Thus only when it has to wait for a time-critical event to occur. For a wait situation which is not time-critical, for instance when the application is multi-threaded and only one thread waits for completion, a notification using interrupts is more suitable. Then no CPU time is consumed for polling and the higher latency of interrupts compared to polling is not important. As a trade-off between these two solutions, a lazy pointer scheme can reduce the amount of accesses to the device registers.

To summarize the above, queues in main memory are an excellent alternative to on-device queues. They have a much better scalability and are more cost-effective than on-device queues. But they have the disadvantage of higher access costs. Instead of one copy onto the device the data is first copied from normal user memory into a dedicated memory region which has to be pinned and contiguous. Then they are copied from this intermediate location onto the device for processing which introduces one additional copy. Another disadvantage that is less important but has to be mentioned is that off-device queues require memory-pinned pages. This reduces the amount of available victim pages for swapping. Furthermore the amount of memory available for applications is reduced. So designs with off-device queues have lower production costs, but part of these cost re-emerge when the complete system is set up. But the costs in total are still less compared to designs with on-device queues.

3.3.3 Support for multiple producers

Like explained in sub-chapter *“3.1.4 Simultaneous device access” on page 70* the computing nodes are getting more and more parallel and simultaneous access to NIs is inevitable. One of the goals of this work (see sub-chapter *“1.5.4 Goal summary” on page 30*) is to allow a large number of user-level processes to access a NI simultaneously. In the scope of queues these processes are producers, hence supporting only one queue is not sufficient. The optimal solution is to assign each process it’s own queue. For the queuing systems above all data structures like queues and pointers are always accessed by only one writer. Either the producer writes and the consumers reads (queue entries and write pointer) or vice versa (read pointer). The multiple-writer problem is completely avoided, hence no synchronization mechanisms are required.

Basically there are two solutions for the problem of multiple producers. One possibility is the usage of one centralized queue which is shared among all producers. The producers have to synchronize using mutual exclusion to ensure correctness. Otherwise a queue entry might be written by several producers and only the last write is stored in the queue. The alternative to a centralized shared queue are distributed queues. Each of the distributed queues is assigned to one producer and is used exclusively. No mutual exclusion is required anymore.

In the sub-chapter *"3.3.1 On-device queues" on page 99* it is shown that on-device queues are very limited regarding scalability. The available memory for queues cannot be easily or even dynamically increased. This scalability problem is the major reason why on-device queues are not a suitable solution when the amount of producers is increased. Techniques like *queue off-loading* into main memory do exist but they introduce a lot of additional overhead. The excellent scalability of off-device queues advises them to be used as building block for multiple producers. In *Figure 3.24 on page 103* the queue set in main memory is shown for one producer. For multiple producers this queue set has to be replicated for each producer. If the host's O/S supports adding of pinned and contiguous main memory regions during normal operation, these queue sets can be dynamically added or removed.

Changes in queues have to result in triggering the device. The device is not implicitly notified of a data value change in main memory, so usually a doorbell functionality is used. For a single producer a doorbell and a read/writer pointer comparison is sufficient because only one queue must be considered. A non-scalable solution for multiple producers is to simply replicate the pointer register set. Each producer is assigned to a dedicated register set. Basically, this is the approach of *device replication*. In *"3.2 Architecture" on page 81* a detailed description of a suitable architecture for multiple producers is shown, and a scalable solution for the doorbell signalling is presented in *"3.4 Virtualization" on page 107*.

3.3.4 Allocating memory for queues

The possibility to obtain physical contiguous and pinned memory region depends on the operating system. Here the Linux O/S in it's current version[1] will be examined as an example. In other O/S other methods may be available but an in-depth analysis of several O/S is beyond the scope of this work.

Basically there are two possibilities in Linux to obtain such regions. The first method is to pass a boot argument to the to booted kernel. The boot argument "*mem=x*" will force the kernel to use the given amount of memory, independent if it is more or less than installed [86]. If more is specified the system will crash sooner or later. If less is specified a part of the memory is unused. The kernel uses the higher part of memory, hence the unused region is located before the kernel memory. This region is not used by the memory management, thus it won't be swapped out and no pinning is required. Nevertheless it is possible to map this region or parts of it into user-level space. The mapping into user-level space is linear.

1. At the time of writing, the Linux kernel version 2.6 is widely used. The latest stable version of the Linux kernel is 2.6.19.2 [85].

So the region is contiguous in both physical and virtual address space. Major drawback is that the amount of memory reserved for this has to be fixed at boot time. A later expansion is not possible. An advantage is that any amount of memory can be reserved in this way.

The second possibility is to use the kernel function "*kmalloc*" or "*__get_free_pages*" to obtain a physical contiguous memory region [87]. The physical memory of the system is only available in page-sized chunks. To allow allocation of memory regions which exceed the size of a page, the kernel uses a special technique. Memory objects are sorted by their size into pool sets. Allocation requests are passed into the pool that holds large enough objects. More details can be found in [87]. But important is that the kernel can only allocate objects of certain fixed sizes. Hence it is possible that more memory is returned than requested, up to twice as much. Furthermore the maximum size of objects is limited by the set of pools principle. It depends on the kernel configuration, but usual is a limitation of 2MB. According to [87] completely portable code should not use more than 128kB. Last, it may happen that no chunk of the requested size is available due to memory fragmentation. It increases with system run time, so it's most likely to succeed directly after system boot. The following code fragment gives an example how to allocate a physical contiguous memory region:

```
#include <linux/slab.h>
void * kmalloc(size_t size, int flags);

struct struct_type *s;
s = kmalloc(sizeof (struct struct_type), GFP_KERNEL);
if (!s) /* requested amount of memory not available */
    return -ENOMEM;
/* do something with the physical contiguous memory region */
kfree(s);   /* free memory region */
```

The main difference between "*kmalloc*" or "*__get_free_pages*" is that the size for the second is limited to whole pages (to be more concrete, the page count must be a power of 2), for the first one any size is possible (at least 32 or 64 bytes, depending on the page size of the system's architecture).

The contiguous memory region has still to be pinned down to prevent it from being swapped out. This is done by the function "*SetPageReserved*", which has to be called for every page of the region. To allow the device to access this region the PA must be known. The function "*virt_to_page*" returns the physical start address. The length of the region is equal in VA and PA space. Last, the region is mapped into user space. There is no need that the device knows the VA and vice versa.

Physical contiguous memory is either allocated on demand or one large region exists which is not used by the memory management of the operating system. In the second case the device driver has to manage the available memory and allocate the different queues inside it. For the first case it is preferable to have as small chunks as possible, i.e. by allocating every queue independently.

3.4 VIRTUALIZATION

The *device virtualization* method is proposed to overcome the restrictions of traditional devices and to improve the overall performance of sophisticated specialized devices, allowing a large amount of processes/threads to access a device simultaneously using *User-Level Communication*. The architecture proposed here is enabled by *context switching*. This method allows a large amount of processes/threads to access a device simultaneously. A virtualized device provides several virtual ports. Each process accessing a virtualized device is assigned to such a virtual port. A port is identified by a unique *Virtual Port Identifier (VPID)*.

The goal of the virtualization is to allow a large and nearly unlimited number of user-level processes to access a device simultaneously to issue work requests. Multiplexing in O/S level is unwanted because this will increase the latency remarkable due to the required overhead for system calls. The processes should be able to directly access a device without any multiplexing intermediate software layer.

The device architecture proposed in *“3.2 Architecture” on page 81* allows to exploit both ILP and TLP. The number of FUs limits the amount of parallelism that can be exploited. If more processes issue work requests than FUs are available the FUs are scheduled between the competing processes in a fair scheme. In this case the work requests are stalled and kept the in the queues until an FU becomes idle.

If several processes are accessing one resource it must be ensured that the processes cannot interact with each other. The separation can be improved by transparency. A transparent virtualization is not visible to a process. Each process sees an exclusive device for it’s own. It has no information about other processes. The safety of the system increases and prevents interaction, either due to harmful intention or programming errors. Another important requirement is the identification of a process towards the device. The device must be able to identify a process in a safe way, which means that a process must not be able to fake it’s identification.

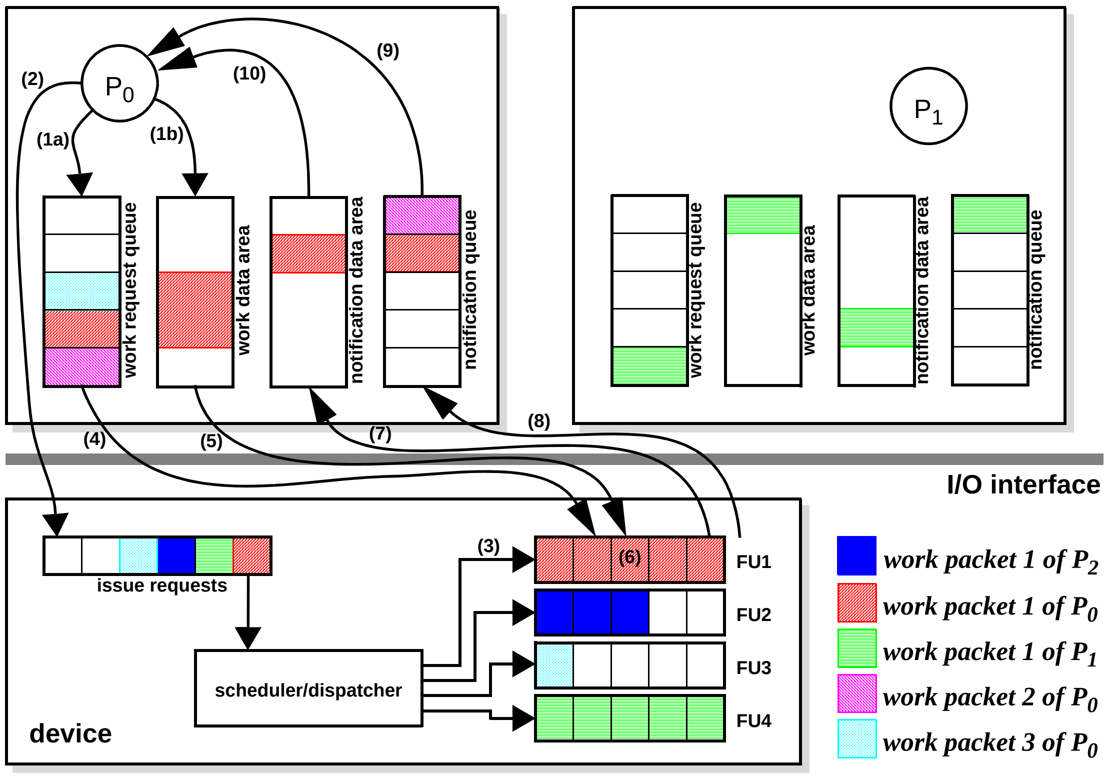
Each process is assigned a context storing all required informations to process work requests. This includes informations about the queues for this process, access rights and process-dependent configuration. The right context is found using the process identification. For each work request the assigned FU will switch to the appropriate context of the origin process.

3.4.1 Processing overview

In general a process accesses a device because it has work to be processed. The work is described using work requests. The device processes the work request and returns the result back to the process using a notification. A notification can be as small as an information that the work is finished. Additionally a notification can also contain a result of the work. In *“3.1.2 Data transfer methods” on page 69* queues are proposed for work requests and notifications, which allow to decouple the CPU-level and device-level processing. The separation into CPU-level and device-level processing has some similarities to the *Decoupled Access/Execute Architecture* described in [88].

This can be seen as a non-blocking device-level procedure call. A process issues a work request (or an instruction) to the device to be processed. Immediately after issuing the process can perform other tasks. The device is triggered with the issue and starts processing the work request (CPU off-loading). After finishing the work the process is informed. Either the process waits for the notification, or it is still performing other tasks and will check for completion later. With the check for completion it synchronizes with the device-level processing.

A work request is not necessarily only processed by the local device. In a networking environment the device is the network interface. For instance an RMA work request is at least partly processed at the target, which is a remote node in the network. Hence device-level procedure calls can be further separated into local device-level procedure calls and remote device-level procedure calls.



Legend:
- 🟦 *work packet 1 of P_2*
- 🟥 *work packet 1 of P_0*
- 🟩 *work packet 1 of P_1*
- 🟪 *work packet 2 of P_0*
- ▨ *work packet 3 of P_0*

Figure 3.25 Work flow for a virtualized device

To allow an unconstrained scheduling of the work request of any process to the several FUs, a work request can be dispatched to any FU. In the case that only one process accesses the device it can occupy all FUs. If several processes issue work, the FUs are shared among them. If more work requests are outstanding than FUs are available the FUs are scheduled in a time-sliced manner.

Figure 3.25 shows an overview of the work flow for a virtualized device. Processes create work requests in the main memory queue and then issue them to the device. There the issue requests are collected and dispatched to the FUs. Each issue contains a process identification to switch the context of an FU and thereby configuring it for the correct set of queues. Only for the first step (here work request issue) the calling process must be identified. Then this information is stored through the complete work processing. Once an FU is configured for a context the remaining steps can be performed. The FU processes the work request, probably fetches additional data from other queues in the set and finally creates a new notification entry to notify the process of the completed work request. The notification queue entry can optionally contain the work results, either as immediate value or by referring to a special notification data area.

The example shows a device with four FUs. Each FU has five work stages: one for fetching the work request (including the context switch of this FU), one to fetch the optional work data, one for processing the work, one to write back the optional notification data and the last to insert a notification into the appropriate queue. Here three processes issue work to the device (only two are shown). Each process has it's own set of queues in main memory, but the central trigger queue with issue requests is shared. The FUs are dynamically shared between the processes.

Only for work packet 1 of P_0 the flow of the work is shown to maintain readability. In this example the flow for this work packet includes 10 steps:

1. Process P_0 generates the work request and places the associated data into the work data area.
2. Process P_0 triggers the device which inserts an issue request into the central trigger queue.
3. This issue request is scheduled to an available and suitable FU (here FU1) by the scheduling/dispatching unit of the device. If the work must be processed in order, this unit also takes care that the dependencies are solved before processing.
4. FU1 switches it's context to P_0. Now the location and status of Process P_1 queues are known and the FU can fetch the work request.
5. The work request includes information about an optional payload. In this case the FU fetches the payload from the work data area. Otherwise this step is skipped.
6. The FU has all required data and processes the work packet.
7. After finishing processing the FU writes back the results of the work packet. This is optional but used in this example case. The notification data area are will contain this result.
8. The FU's last step is to insert a notification entry in the notification queue. If the process P_0 is polling on the queue waiting for a change it is immedi-

ately notified, hence this step must be the FU's last one. Otherwise race conditions may occur.

9. Process P_0 is notified of a change in it's notification queue, either due to polling or by an interrupt. In both cases it fetches the entry and checks the result of the work packet.
10. In this case the result contains additional payload in the notification data area, so the process retrieves the data.

For a complete understanding of the work flow, the status of the other work packets is as follows:

- Work packet 1 of P_2: Currently processing in FU2
- Work packet 1 of P_1: Processing in FU4 completed, currently inserting a notification queue entry.
- Work packet 2 of P_0: Complete processing finished, but entries in work request and notification queue are still present.
- Work packet 3 of P_0: Dispatched to FU3, currently context switching/work request fetching takes place.

3.4.2 Recognizing processes

The most important requirement regarding *device virtualization* is that a virtual device has to distinguish between accesses from different processes. Thus all accesses must be tagged with the VPID to allow the device to recognize the different processes. The device stores the information relevant to one VPID in a context. A context contains in particular all informations about the interface between process and device (e.g. queue offset, read and write pointers), the current state of the virtual port, configuration informations and permissions. Fast context switches occur between operations of different processes, configuring the execution unit for this specific user process. No multiplexing kernel-level processes or IPC are required.

The design space in figure 3.26 shows that processes can be directly or indirectly identified. Using A *direct process identifier* the CPU explicitly tags the access to the device with an unique identifier. The identifier must be inserted by the CPU in a secure way. In particular this must prevent the process from modifying or inserting wrong identifiers. Only then the desired separation and security among the processes can be ensured. Using a trusted process to insert this identifier as data value into an access is not possible because this would prevent User-Level Communication.

An *indirect process identifier* can be realized using address space. Then a set of pages in the peripheral address space is provided. Each page of this set can be mapped by one user-level process. The address associated with each page of the set is used to identify the process. By read/write operations on these mapped pages the processes communicate directly with the device, comparable to User-Level-Communication. The address associated with the operation is used to distinguish the different calling processes. Because the MMU manages the address translations a process cannot modify the address of an access to the device and the security and separation is guaranteed.

An indirect process identifier is the most efficient way to signal an unique identifier to the device when using standard processors. The direct method requires modifications to a CPU which is not desirable for a general usable device virtualization. Because of this the device virtualization relies on indirect process identification. Figure 3.27 shows the mapping of device I/O space into the address space of user-level processes.

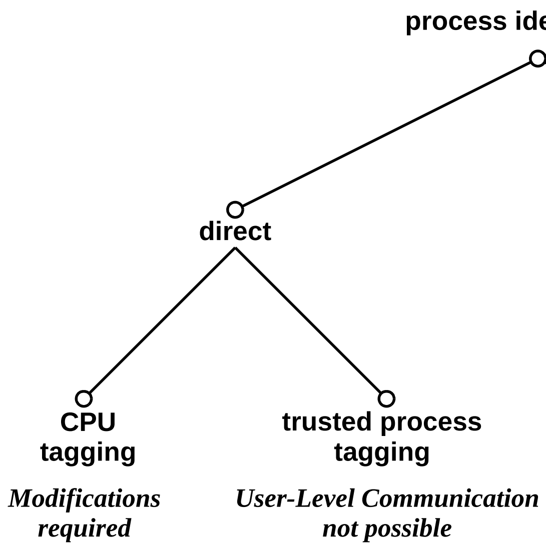


Figure 3.26 Design space of process identification

In the address space of the device one page is used for management purposes and should only be mapped by trusted processes. A set of pages is available to be mapped by user processes. Each page of the set is called *Triggerpage*. Because all work queues are located in main memory these pages are only used for triggering the device, which leads to the naming. Each Triggerpage can be mapped by only one process. But one process can map more than one Triggerpage to separate it's accesses.

The device driver manages the virtualization page set. Each time a process opens the device, one of these pages is mapped into the address space of the process. The device driver takes care that no other process can map this page. The device management is accessible over a separate page which is not part of the virtualization. This management page is mapped by the device driver or a privileged process to perform device configuration, status query and management functions. Normal processes access the device only over the virtualization page set.

The virtualization is transparent to a process, so it only sees a device for its own exclusive usage. The transparency has also a security aspect. It prevents processes to interact with other processes. Independent if the interaction is due to programming errors or due to harmful intention, the processes are completely separated. Because of the paging principle of modern architectures, transparency can only be implemented using pages. Every process accesses the device using pages, and the MMU prevents the processes from accessing data or control structures of other processes. Further, the complete mechanism does not involve other kernel- or user-level processes, enabling User-Level Communication without additional overhead for security. It is also usable in VM environments where with this technique the VMM can be bypassed and is no longer involved in device accesses.

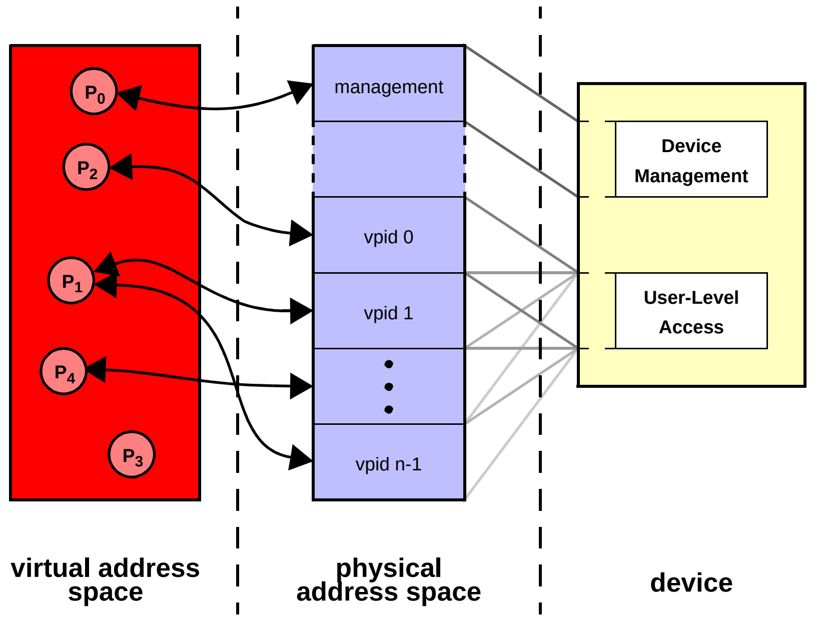


Figure 3.27 Access over mapped pages

3.4.3 Queue design

In *"3.3.3 Support for multiple producers" on page 104* it is shown that replicated dedicated queues in main memory are an excellent solution for multiple processes accessing the device. For each process a set of queues is created which are only used by this process. Access from several processes (as producer/consumer) to one queue is not allowed. These dedicated queues avoid synchronization, which is inevitable for shared queues.

The targeted NI here is a peripheral device with a non-cache coherent link to the system. Because of the non-coherent link the device is not informed about data changes in main memory. The architecture is not ought to be limited to coherent interfaces, so another solution must be found to solve this problem. Having the device polling on the queues for changes wastes valuable I/O bandwidth. Additionally, this solution does not scale with the number of processes because more and more queues must be checked. The check if a change occurred also requires to store a compare value which requires a storage place, preferable on the device. Again, this limits the scalability.

The most efficient solution is to let the processes notify the device of the change. An on-device queue can store the trigger information of the different processes. Several problems arise with this approach of a *shared central trigger queue*:

1. The on-device queue is shared between all processes which insert their trigger information. For a shared queue the multiple writer problem arises again and synchronization by mutual exclusion is required to avoid race conditions.
2. Because on-device space is limited due to the decision for a memory-less architecture and as much different triggers as possible fit into the queue, one trigger entry should be kept as small as possible.
3. Because the queue is limited in size, a trigger operation might fail. Then the process must be informed to retry.
4. The trigger operations consume limited I/O bandwidth. The trigger payload should be as small as possible to reduce the required bandwidth.

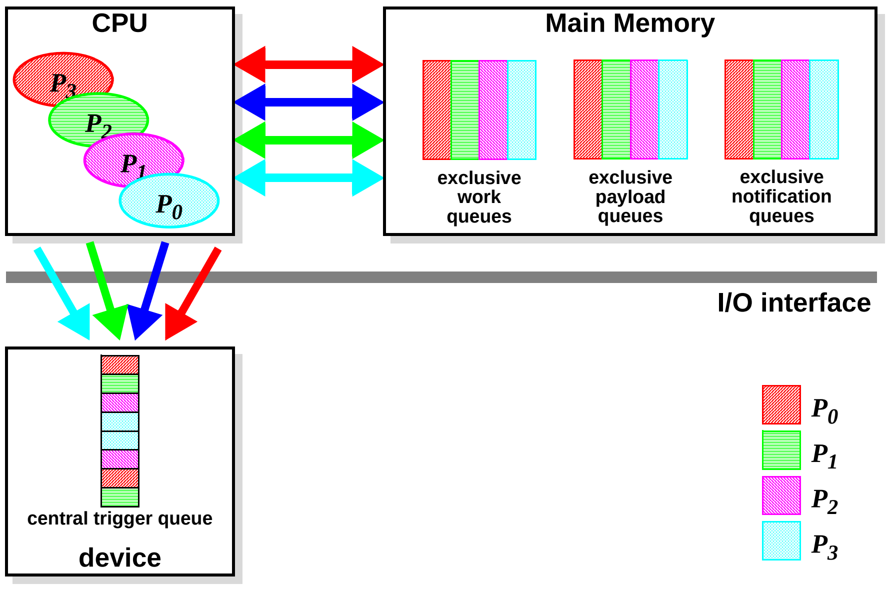


Figure 3.28 Interface to a shared central trigger queue

In *Figure 3.28* the complete process queue set for a virtualized device is shown. This follows the approach presented in *Figure 3.24 on page 103*. Several processes are running on the system, either on one or several CPUs. Each process accessing the device has an exclusive queue set in main memory, consisting of work, payload and notification queues. To recognize a change in one of the queues in main memory, the processes have to explicitly notify or trigger the device. These trigger informations are stored in the central trigger queue to be processed by the device.

3.4.4 Triggering operation

A virtualized device must be informed of new work queue entries by inserting the VPID into the *central trigger queue*. This *instruction issue* or *trigger operation* is divided into two steps, the check for available space in the queue and the insertion of a new queue element. Multiple producers are enqueuing entries and the two steps are part of a critical region. An enqueue operation must not be interrupted. Figure 3.29 shows an overview of the design space.

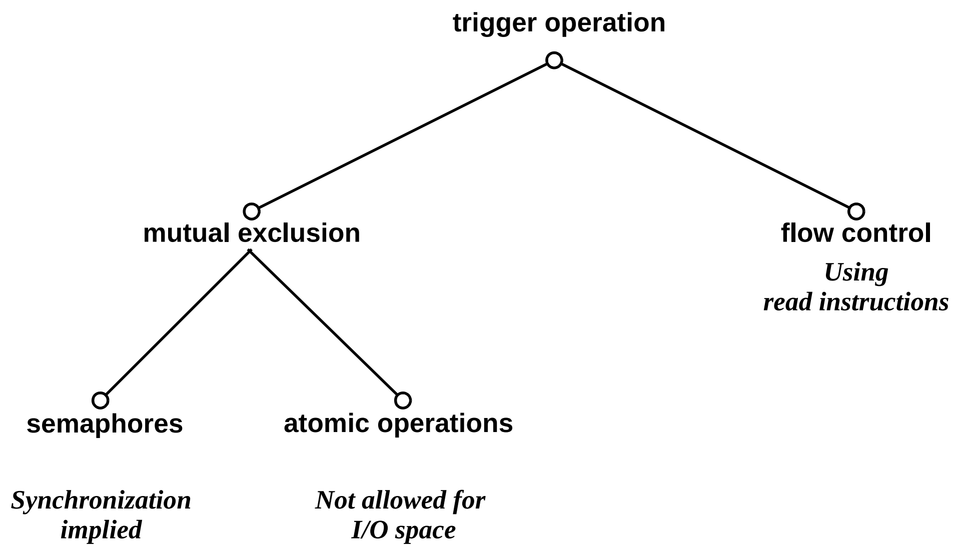


Figure 3.29 Design space of trigger operations

Several solutions exist to ensure correctness in critical regions. The first is based on mutual exclusion. A *semaphore* can ensure that only one enqueue operation is performed at a point in time, no operation can take place simultaneously. The usage of a semaphore leads to synchronization among the producers and the overhead is quite high.

An *atomic operation* with a *Read-Modify-Write (RMW)* scheme is another solution. An atomic operation is uninterruptible and implicitly prevents simultaneous accesses to the same location. The read part of the RMW operation is the check for available space and the write part is the insertion of the VPID. RMW operations or atomic operations in general are usually not allowed in the I/O address space. A limitation to seldom available interfaces supporting them is unwanted. Both solutions, the atomic operation and mutual exclusion also prevent several outstanding operations, because in both schemes the enqueue operations are serialized.

A completely different and new approach is to include flow control in the trigger operation. The flow control contains information about the success of the operation. If the queue is full the operation fails and the flow control returns an error to the producer. The producer has to try again to insert his VPID. If the queue is not full the flow control informs the producer of the enqueued VPID. In other terms support is required for failed instructions. The advantage of this approach is that less overhead is required. Furthermore, if several operations are outstanding one of them can be favoured.

Challenging with this approach is that no instruction set of modern processor architectures has support for failed instructions or flow control. Only the read (or load) instruction includes an answer in it’s processing. So this instruction deserves a closer analysis.

Read instruction. The parameter of a read instruction is the address to be read. The result of a read instruction returns the value read. To use it as an enqueue operation the VPID must be included as a parameter. The result can contain the information of success or failure of the enqueuing. A read instruction contains only one step and is hence atomic by default. If the VPID can be included in the read address the read instruction is very suitable for the enqueue operation. But the producers are untrusted and so the VPID must be included in the address in a secure way, preventing a producer from falsifying or modifying the VPID.

A process can only be prevented from modifying an address with the help of the MMU. The MMU converts the virtual addresses into physical ones (figure 3.30). The address translation only changes page information, the lower part of the address remains unchanged. Thus the VPID cannot be included in the page offset, instead the page address must be used. Typically a VPID is limited in size[1], which allows to use a part of the page address to contain the VPID.

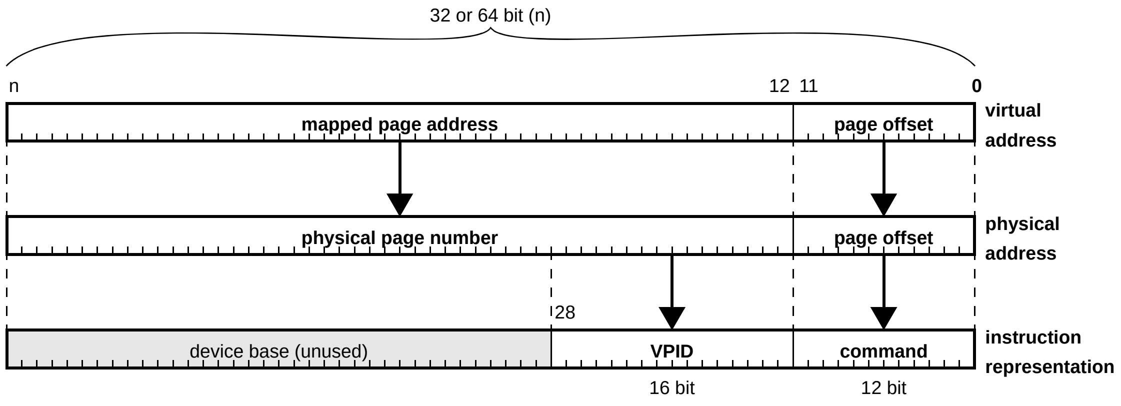


Figure 3.30 Semantic of Triggerpage addresses

The producer is a process and hence uses VAs. The device only sees the corresponding PA. A part of the PA is used to include the VPID. With the help of the PA the device can identify the producer. A process has no knowledge about the corresponding PA and for the producer the VA contains no information about even it’s own VPID. The included VPID is completely transparent to the producer which further increases the safety.

Thus the lower part (page offset) is not modified by the MMU and can be used as a command or tag section. By using different page offsets a producer can signal different

1. For instance a VPID with a size of 16bit allows to differentiate up to 65536 processes. This amount is expected to be large enough to provide support for almost any type of application.

commands to the device as well as providing hints about the type of the work to be processed.

The device observes a read operation to a physical address. It converts the address of the read operation to a VPID/command pair. This pair is stored in the central work queue. The producer is informed if the issued instruction is enqueued or if it is abandoned because missing queue space. This information is returned as result of the read operation. If the instruction is accepted, the device returns *True* as result of the read access. Otherwise *False* is returned. Of course more sophisticated return results are possible. For instance, this additional information can be a proposed wait time if the queue is full, or the currently free entries or any useful status information of the device.

For the MMU the approach using read instructions is a normal address translation without modifications. The TLB can still be used. It stores the most used address translations and reduces the average translation time. If the TLB does not contains the appropriate entry a page table walk must be done by the MMU.

Applying an access granularity of 64bit, 512 different read addresses are possible, allowing the read operation to be tagged with 512 different commands (see figure 3.30). This allows not only to issue work requests to the device, other operations are also possible.

Conditional Store Buffer. Each trigger operations results in a inserting a new entry in the trigger queue. The storage in the trigger queue is dependant on several conditions, so the queue is also called *Conditional Store Buffer (CSB)* [89]. An insertion into this *CSB* is conditional regarding available space, access rights and other restrictions. The capability of the device to accept or reject instructions can be used e.g. for priorization of instructions. It stores the trigger informations, which are:

- *VPID*: used to identify the calling process
- *Command*: used to specify the command to be processed in more detail.
  This information might be redundant with the information stored in the
  corresponding work request, which is stored in a queue in main memory.
  The scheduler behind the queue uses this command tag to ensure the
  optional ordering.

The new trigger method is very efficient because it is only a read operation. Compared to mutual exclusion or atomic operations no additional overhead is introduced. It fulfils all requirements to issue instructions to the central work queue of a virtualized device. The number of processes issuing VCIs is not limited by this mechanism. Only the available I/O address space is an upper bound for the process count. If the I/O interconnect supports split-phase transactions, this mechanism is even improved. Several issue operations can be outstanding, and the device is able to favour certain issue operations.

Figure 3.31 shows the procedure to trigger a virtualized device in detail. A user-level process has mapped a Triggerpage. The trigger operation includes the following steps:

1. To trigger it performs a read operation to a certain address inside the Triggerpage, associated with the wanted command (for instance work request issue).
2. The MMU of the host system converts the VA of the read operation to a PA.
3. The device sees a read operation to a PA.
4. From the PA it can extract the VPID and the command.
5. The device validates the VPID/CMD pair, checks for available space in the CSB and performs secondary checks including access rights, priorization or more sophisticated features.
6. If all checks have passed successfully the VPID/CMD pair is inserted into the CSB
7. The process is notified of the successful trigger operation by returning *True*.
   If one of the checks fail, a *non-True* value is returned. This value can optionally include informations about the reason. In this case the process will sooner or later re-try the trigger operation.
8. To process a command the VPID/CMD pair is dequeued, scheduled to an appropriate FU and there processed.

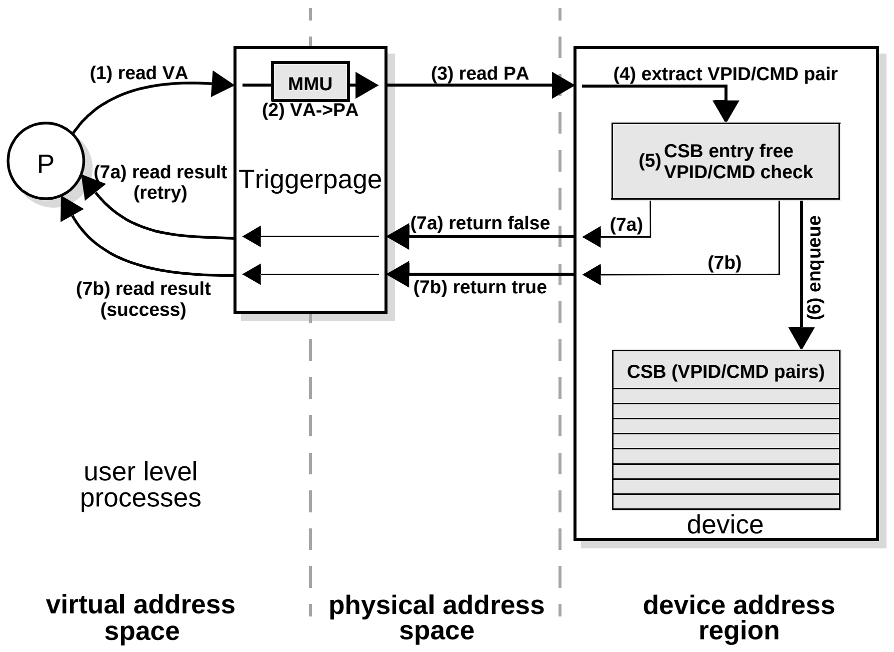


Figure 3.31 Issue operation using the Triggerpage

3.4.5 Virtual Machine Environments

Because no software overhead is required for the device virtualization, Virtual Machine environments and the device virtualization perfectly fit together.

VMMs like Xen allow multiple operating systems (guests) 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. Devices are typically shared in a time-sliced manner [90] between several guest systems by redirecting all I/O traffic to the VMM, which is responsible to schedule the accesses. Another possibility is the exclusive use of a device by only one guest system.

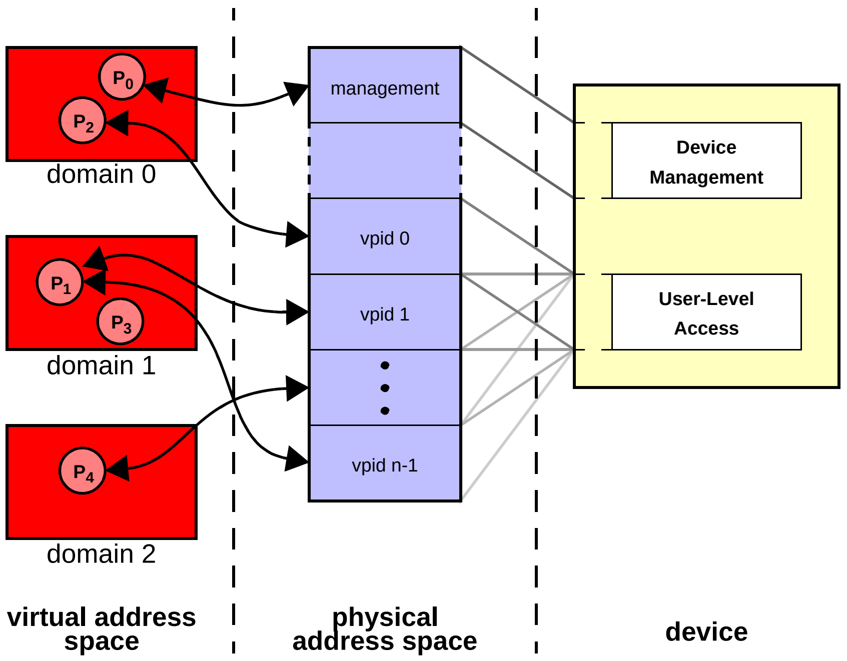


Figure 3.32 Device virtualization in a Virtual Machine environment

Virtual Machine environments can benefit a lot from device virtualization. The device virtualization allows every guest operating system to open any number of virtual ports of the device. Processes from different guest systems can concurrently and directly access the device without scheduling by the VMM. The VMM is only responsible to manage the device. For instance open operations from the guest operating system are handled by the VMM, which returns a page to be mapped. The VMM takes care that no one other maps the same page. From now on, the process (independent from which guest system, if kernel- or user-level) can directly access the device using this page as a *Triggerpage*. The device itself is responsible for the scheduling of work requests from the different processes. From the device’s point of view it is completely irrelevant if these processes are part of one or different guest systems.

Figure 3.32 shows a typical situation in a Virtual Machine environment where a device is virtualized using the methods presented here. The device 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 device is done by the VMM, which is running in domain 0. If a process wants to open the device it maps a Triggerpage. All access to the device is done over this Triggerpage. A process can also map several Triggerpages. The device recognizes the accessing processes by the address of the Triggerpage. The operations are enqueued in the CSB and scheduled to free functional units.

There are no restrictions regarding the domain of a process or the total number of domains. The devices provides a number of Triggerpages. All Triggerpages can be mapped into one domain or each into another domain. The device recognizes the processes only over the VPID, which is a result of the process’s Triggerpage address.

3.5 THE ULTRA ARCHITECTURE

The architecture developed in the previous chapters targets an unconstrained use. It exploits any available parallelism and the communication set is composed of various operations for different purposes. The virtualization allows almost any number of processes to access the device simultaneously. The hardware resources can be dynamically scheduled among the competing processes.

What is not achieved yet is support for a lowest latency communication scheme. The virtualized device architecture relies on contexts and on work request issue using the Triggerpage. This is an excellent approach to use the limiting I/O interface very efficiently, but at least one memory access is required to inject a packet into the network. This memory access fetches the work request from the process's work queue in main memory. Caching effects or references to other data structures may increase the number of required memory accesses.

While this is necessary for a scalable architecture and furthermore not a problem for normal message passing applications, dedicated support for *fine grain communication* is still missing. To meet the goals of this work (see sub-chapter *"1.5.4 Goal summary" on page 30*) now a low-latency communication method is presented.

The low-latency communication method presented here is called *Ultra Low Latency Transmission (ULTRA).* It allows a shift from coarse-grained to fine-grained communication and the system becomes closer coupled. Instead of collecting many small data structures into large bulk messages, small elements can be sent out independently. *Active messages* [91] (which are sent out to trigger certain operations) are an application where ULTRA fits perfectly. Also fine-grained *Global Address Space (GAS)* applications can be improved a lot, rendering software assistance unnecessary [92].

If the latency of a message transfer is examined in more detail, it can be seen that the main component originates from the I/O interface [93]. The remaining parts like network device and switch fall-through latencies are much smaller. For instance the fall-through latency for an ATOLL switch is about 90ns [39], while a PCI I/O cycle requires about 500ns. Hence the work presented here focusses on an optimized use of the I/O interface. Goal is to develop a message injection and retrieval scheme which requires as few I/O cycles as possible, and the inevitable ones must be highly efficient. Only then the latency can be reduced to the minimum. Fine-grained communication schemes do not require high payloads, so it is for instance sufficient to support message sizes up to the size of a cache line.

Comparable to a modern CISC architecture with it's manifold instruction set, the send/receive operation offered by ULTRA is only one in a communication instruction set. Compilers or (communication) libraries are responsible for choosing the most appropriate instruction or communication method. ULTRA's focus is to reduce the communication latency to it's minimum. If constraints are introduced, they can be tolerated because other communication functions from the set can be used in this case. Compared to the previous work this is a complete different approach to inject and retrieve packets from the network.

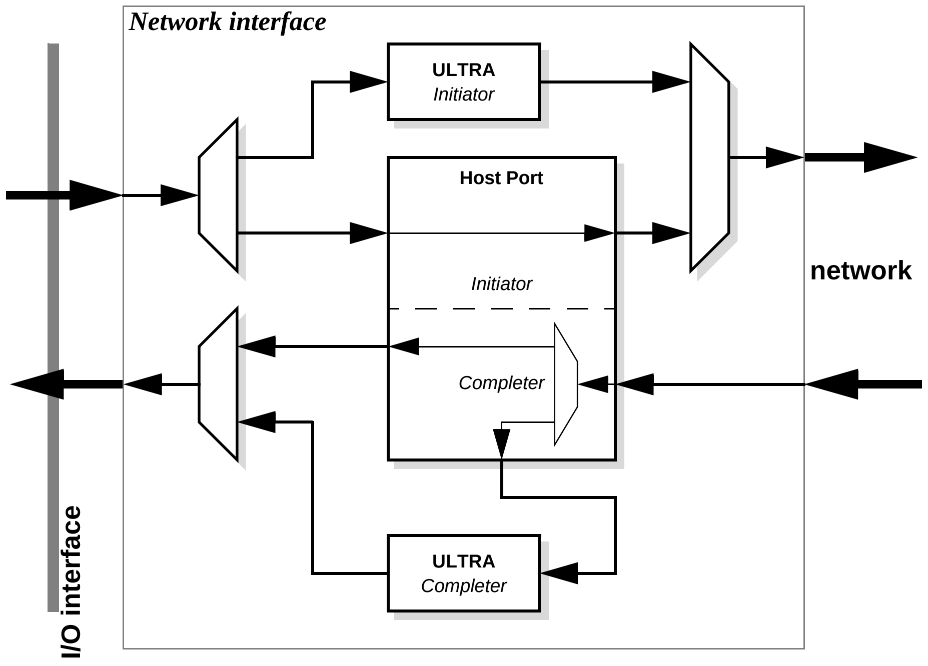


Figure 3.33 Integration of ULTRA in the network interface architecture

Figure 3.33 shows an overview of the network interface. The previous work, in particular the multi-threaded context-based architecture, the memory-less scalable queue design and the virtualization are combined in the *Host Port*. ULTRA is a separate module beside the Host Port. Both modules can directly be accessed by processes and share the access to the network. If an incoming packet requires to be handled by ULTRA, the scheduler in the Host Port recognizes that this incoming packets is for ULTRA. Then it is not forwarded to one of the Host Port’s FUs but to the ULTRA module.

3.5.1 Related work

There are several other approaches trying to reduce the latency of a message transfer. Basically they differ regarding the location of the network interface (see sub-chapter *“1.3.1 Network interface locations” on page 19*) and if specialized or standardized interfaces are used (see sub-chapter *“1.3.3 I/O interface” on page 21*).

Some systems integrate the communication modules into the main processor, which is certainly the best approach to minimize latency. But the use is restricted to the used processor type. Examples for such systems are the Transputer [23] or the iWarp [24]. In both systems data is sent out by writing to special registers, and data is received by reading special registers. This approach requires a new processor design or the modification of an existing one, which is not the goal of this work.

A different approach is DimmNet-1 [40], which is based on a network interface plugged into a DIMM socket of the mainboard. These sockets allow much faster accesses than I/O sockets, but the use is again very restricted: DIMM devices cannot signal any events to the processor or invalidate cache lines to enforce a re-read.

An example for a specialized solution is the RapidArray fabric used in the XD1 system from Cray [28]. It's use is limited to this system. Few details are known, but the network interface embeds processors to off-load network functions. Memory copies are used to transfer small messages onto the NIC. A special solution is not targeted with ULTRA, instead it should be usable in widely available systems and not be limited to a certain special system type.

Examples for systems plugged into standard I/O interfaces are Quadric's STEN unit, InfiniPath by QLogic (formerly PathScale) [35] or the research project ATOLL [38]. While the exact functionality of the commercial solutions is best to our knowledge not published, the ATOLL approach is well documented.

ATOLL provides two mechanism for message transfer, the PIO and the DMA mode [38]. Because of the overhead to initiate a DMA transfer this mode is most suitable for large messages. Small messages (for instance with a size below 512 Bytes) are optimally transferred using the PIO mode. PIO means the processor copies data word for word into the NIC. The number of I/O cycles required to inject a message is the main component of the total latency. For ATOLL, several PIO writes to different registers are required, bursts are not possible. The ATOLL approach can still be significantly improved and the basic principle of it's PIO mode is used here.

3.5.2 Basic architecture

Focus of the development is to reduce the required I/O cycles to inject and retrieve packets to it's minimum. For a peripheral device the minimum number of I/O cycles required for packet sending and receiving is respectively one. Only with one cycle minimum latencies are achievable. The communication function offered by ULTRA is designed to be only a part of a complete communication set. Other communication functions are available if ULTRA cannot be used. The use of ULTRA is restricted, for instance regarding packet size and the number of destinations.

The basic idea of ULTRA is to separate the packet into a variable (dynamic) and static part. The static part is stored in the ULTRA unit by configuration. The variable part is provided with each access cycle. For minimal latencies only one cycle can be used, but a single burst cycle allows to include multiple data words.

A packet is composed of the following components: type, route, tag and payload[1]. The payload is considered variable, because the overhead required for re-configuration upon

1. The error correction is calculated and checked by the hardware modules and hence not required to be provided by the user process.

each change would render the advantages of this approach unnecessary. Similar applies for the tag, which usually contains a sequence number. But the type is obviously fixed because only ULTRA packets are supported by the modules here. The route is also considered static. The resulting limitation of an ULTRA unit to one destination can be overcome by replication. In particular for source-path routing including the routing string in each access is definitely not an option[1].

To achieve lowest latencies *User-Level Communication* [42] is inevitable, otherwise the required system calls would dramatically increase the latency of a packet. To support multiple processes using ULTRA with User-Level Communication, several *ports* are offered to the host side. Each port can be used by a different process to access ULTRA.

ULTRA is a part of the complete network interface architecture. Here also applies the separation into Initiator and Completer respectively origin and target node. On the sending origin node an ULTRA *requester unit* resides and on the receiving or target node an ULTRA *completer unit*. ULTRA does not require a responder unit. Figure 3.34 shows an overview of the communication scheme. The *ULTRA path* starts with the origin process initiating a packet transfer, continues with the requester and completer unit and ends with the target process receiving the packet. In particular the part from origin process to requester unit and from completer unit to target process are highly optimized regarding latency.

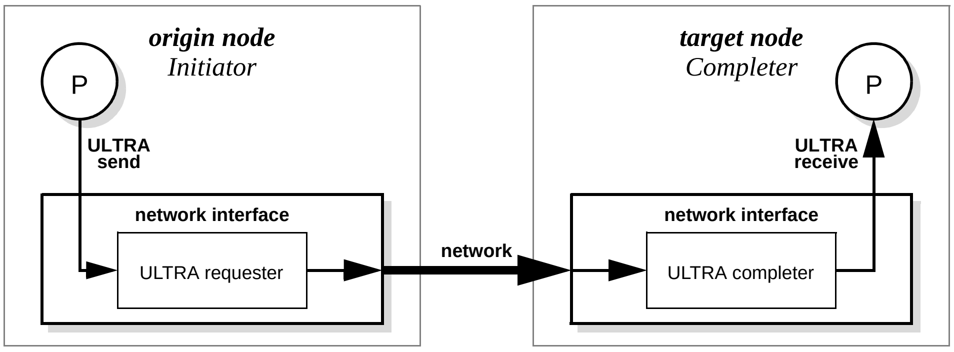


Figure 3.34 Packet transfers using the ULTRA communication scheme

Constraints. The intend of ULTRA is not to introduce a second network in parallel to the existing one, instead only one is used for both normal and ULTRA packets. This implies that the same packet format is used.

If an acknowledge of an ULTRA packet transfer is requested the existing acknowledge transport mechanism is used. The existing mechanism is suitable because acknowledging a

1. Porting ULTRA to an interconnect which uses table-based routing may result in including the destination identifier in each access cycle.

successful completion is not part of the time-critical *ULTRA path*. In addition, software layers can rely on a single notification queue where all informations are gathered.

An open design issue is if ULTRA packets must be in order with other packets. The optional ordering scheme used here (see sub-chapter *“2.1.6 In-order and out-of-order delivery” on page 36*) requires that packets with the same source, destination and communication type are kept in order. ULTRA is a two-sided communication scheme and hence part of the Send/Receive communication set. Only using the Triggerpage the in-order delivery of ULTRA instructions together with other Send instructions can be ensured. For an in-order delivery they must be issued over the Triggerpage. Then the work requests are kept in main memory. This renders the advantages of ULTRA compared to a Fast Send instruction unnecessary. Hence ULTRA instructions are only kept in-order among themselves.

Pre-configured ports. The basic idea is to use pre-configured ports for packet injection and retrieval. The configuration of a port includes as many parameters as possible. Only frequently changed parameters are not included in the configuration. They are passed to ULTRA as dynamic data, while the configuration is considered static[1].

At least one access to the requester unit is required to trigger the packet injection. The payload as a frequently changing parameter of the packet can be combined with this access. Otherwise every time the payload changes a re-configuration of an ULTRA port would be required, which would outweigh the advantages of pre-configuration. The tag is used to store packet sequence numbers (which changes with every packet), so it is also considered to be a dynamic parameter. This results in only including the source identification and the routing information in the *pre-initialization* of a requester unit. To inject a packet a single burst write passes the tags and the payload to the requester unit. The payload size is automatically recognized using the start address of the burst write. It is stored together with the tag and the source identification in the packet header.

Comparable to the pre-initialization of the requester unit shown above, for the completer unit a *pre-completion* is applied. This only includes the source of a packet. The completer unit is configured to receive only packets from one source. Then only the tag and payload must be stored in buffers while the source of the packet is static and thus known. This approach is in-line with the definition of an ULTRA path from origin to target.

The number of available destinations per requester unit is limited by the pre-initialization to one. The same applies for the completer unit, but regarding the source. To allow multiple destinations and sources with an ULTRA unit, replicated ports are available to support multiple destinations and sources. The port replication is achieved using contexts. A process maps one requester port for every destination it uses, respectively one completer unit for every source. Because User-Level Communication is applied one port can only be used by one process.

1. The configuration can only be changed by a privileged software instance. It cannot be part of a typical operation, because the required system call results in a large amount of overhead. The typical operation must be completely performed in user-level.

To support bidirectional communication always pairs of requester and completer units are implemented. Each port of a unit has an unique *ULTRA Port Identification (UPID)* number. The UPID is comparable to the *VPID* introduced in sub-chapter *“3.4 Virtualization” on page 107*. Each packet has an *Origin UPID (OUPID)* to identify it’s source and a *Target UPID (TUPID)* to determine the destination.

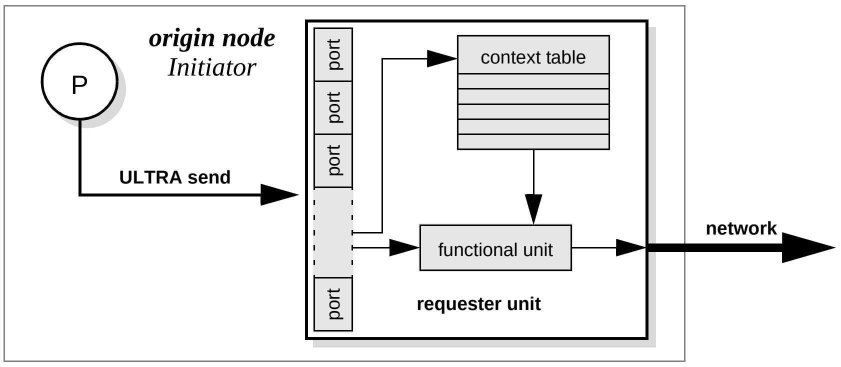


Figure 3.35 Basic functionality of an ULTRA requester unit

So the requester unit provides a set of ports to support both multiple destinations and multiple accessing processes. The hardware module itself is not replicated, instead the access to a certain port results in a fast context-switch. This configures the unit for this process. Figure 3.35 depicts the basic functionality of a requester unit.

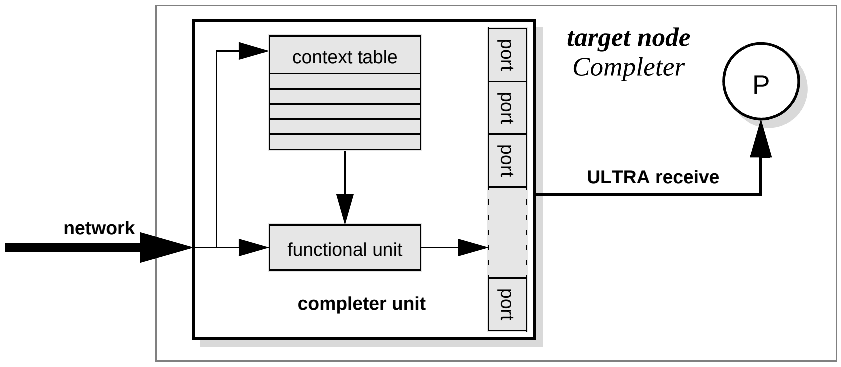


Figure 3.36 Basic functionality of an ULTRA completer unit

While the requester unit has to recognize the used port for an access from host side, the completer unit must identify the target port of an incoming packet. The port is described by the TUPID, which has to be part of the packet header. Then the packet can be stored in the appropriate buffer and fetched by the target process. The basic functionality is shown in figure 3.36.

Limitations. The pre-initialization and pre-completion presented above result in some limitations regarding the packet transfer. Each requester port is limited to a single destination and each completer port to a single source. To allow multiple sources and destinations ports are introduced. Nevertheless the total amount of available destinations and sources is restricted, in particular when the ports are shared by several processes.

Another limitation is the supported packet size. An ULTRA packet is temporally stored in the requester unit. The used on-device memory structure is limited and results in a maximum packet length.

Because the routing string is also stored in a requester unit as part of the pre-initialization the routing length is also restricted. Sophisticated run-length encodings of the routing string can diminish the limitation of available destinations. Applying table-based routing schemes even overcomes this limitation.

3.5.3 Address space mapping

The ULTRA ports have to be accessed from user-level to allow User-Level Communication. Compared to sub-chapter *"3.4 Virtualization" on page 107* separation and security must be ensured because several applications can access ULTRA over the replicated ports. This is achieved using the paging principle and mapping of I/O space into user-level.

ULTRA presents a set of pages to the host. One page is privileged and used for management and configuration purposes. Furthermore the set consists of one page for each requester port and one for each completer port. The requester and completer pages are used by user-level processes to inject and retrieve ULTRA packets. Figure 3.37 shows an overview of ULTRA's address space. One process can map several pages to obtain multiple OUPIDs respectively TUPIDs. Each OUPID is configured for one destination, while each TUPID is assigned to one source.

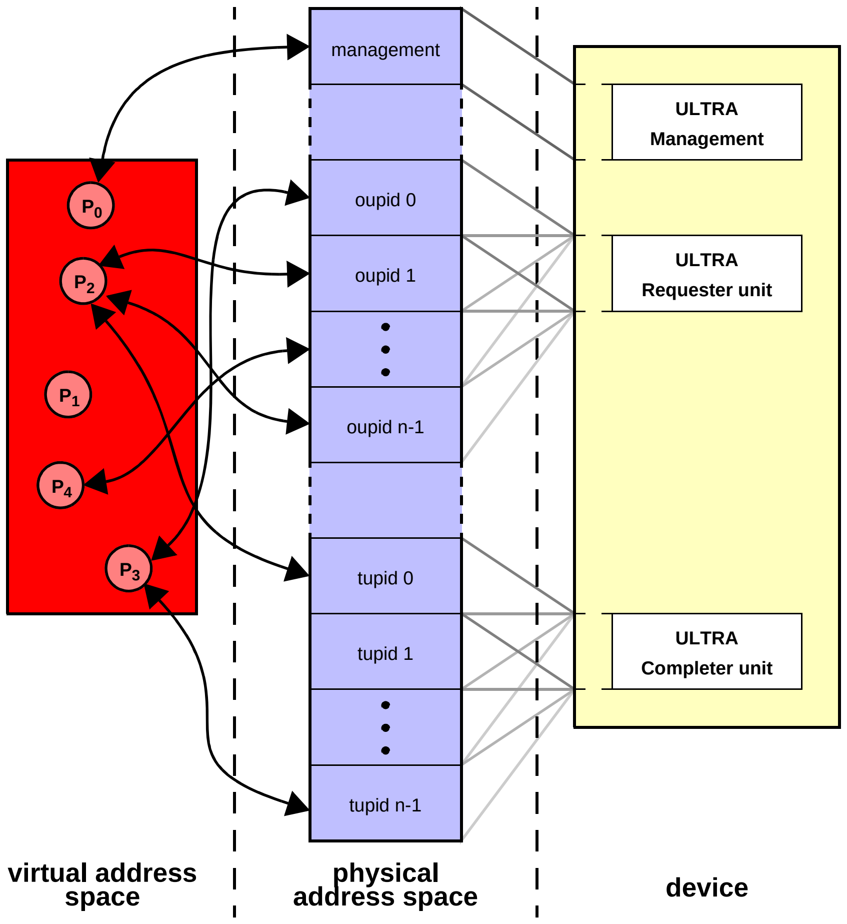


Figure 3.37 ULTRA address space overview

Figure 3.38 shows the address interpretation when user pages are accessed. The MMU only translates the upper part of the address while the page offset is kept. Comparable to sub-chapter *“3.4 Virtualization” on page 107* the upper part is used for identification of the process while the page offset contains the command. The identification is secure because processes cannot change the page number. In contradiction the page offset is available for a process to include more information about the operation to be performed.

The ID field represents the UPID which is accessed (up to 11 bits, dependant on the number of ports implemented) and with the mode field (1 bit) user- or driver-level access can be separated. While in user mode only packet injection and retrieval (including pointer updates) is possible, the driver mode is used for the (pre-) configuration of the appropriate unit.

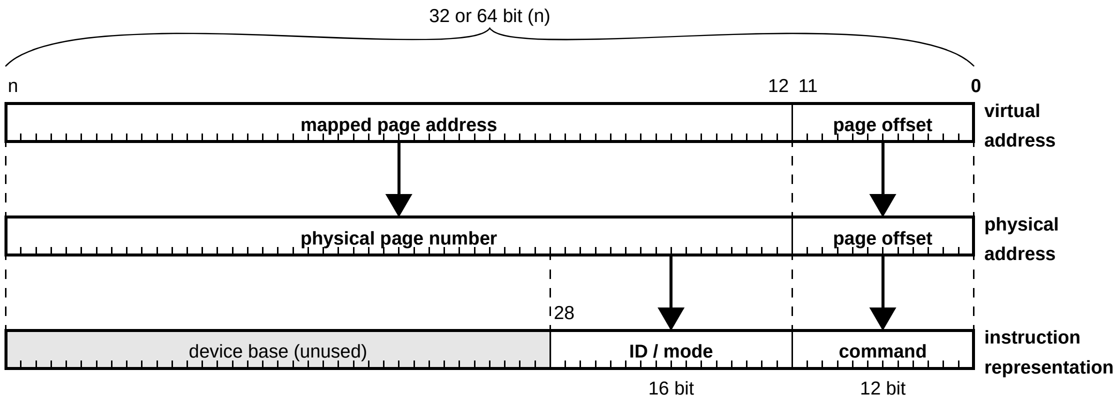


Figure 3.38 Interpretation of ULTRA addresses

3.5.4 Packet injection

The prerequisite for packet injection is a pre-initialized requester port. For each access from user process the requester unit is then configured using the appropriate port context. With the access the process only provides the packet tags together with the payload data and size. Figure 3.39 shows the command coding for the requester unit. The addresses are aligned to 64 bit, hence the lowest three bits cannot be used.

For each packet the notification method for the completion of the operation can be selected. This is achieved with the three bits. The *IRQ* bit results in an interrupt for notification, here polling is not required. This is the preferable method if the completion is not time critical. If the notification is based on polling, the *NOTIFY* bit is set. Polling is used for time critical completion, but the CPU load increases. With the *ACK* bit an acknowledge is requested. Then the target node will send back an acknowledge containing the result of the packet transfer and the completion notification includes that the packet has arrived at it's destination. Otherwise the notification only includes a successful injection of the packet into the network.

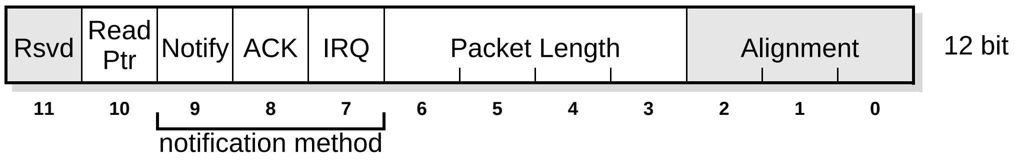


Figure 3.39 ULTRA requester unit command coding

Beside the notification selection it is also possible to read out the status of the appropriate requester port. The only status information required by a user-level process is the current read pointer of the packet buffer. A new packet is only accepted if enough buffer space is available. Hence the most recent read pointer can be fetched by the process setting the *RP* bit to one and performing a read operation. This is the only read operation supported for user-level access on the requester unit, all other are writes.

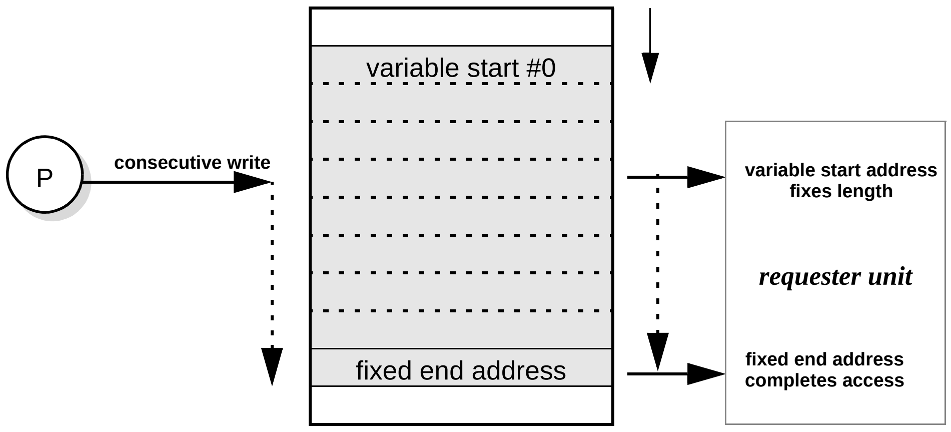

variable start #0
fixed end address
P
consecutive write
variable start address fixes length
requester unit
fixed end address completes access

Figure 3.40 ULTRA requester unit access

Payload size detection. Beside the payload data and the tag the process must also include the payload size in the access. But this additional data can be avoided. The first approach uses a single burst access starting with a fixed address. The first data word of the burst is interpreted as tag, the remaining ones as payload. The requester unit detects the burst and sets the size accordingly. But burst accesses cannot be always ensured, because the host system can split one burst cycle into several smaller cycles. Beside this, if the burst accidentally uses too high addresses, the IRQ bit is set which results in unwanted behavior.

The solution for this is not to use a fixed start address for the burst. Instead a fixed end address is used, resulting in a variable start address. Then the start address varies with the desired payload size. Bursts are not required to recognize the packet length and the IRQ bit cannot be accidentally overwritten.

Figure 3.40 shows an example packet send access. The start address is always chosen in a way that the last word of the payload is written to the fixed end address. Using the start address the requester unit immediately knows the packet length and can decide if it fits in the buffer. An access to the fixed end address terminates the send operation. In the example above the packet length is 6 words (each word 64 bit).

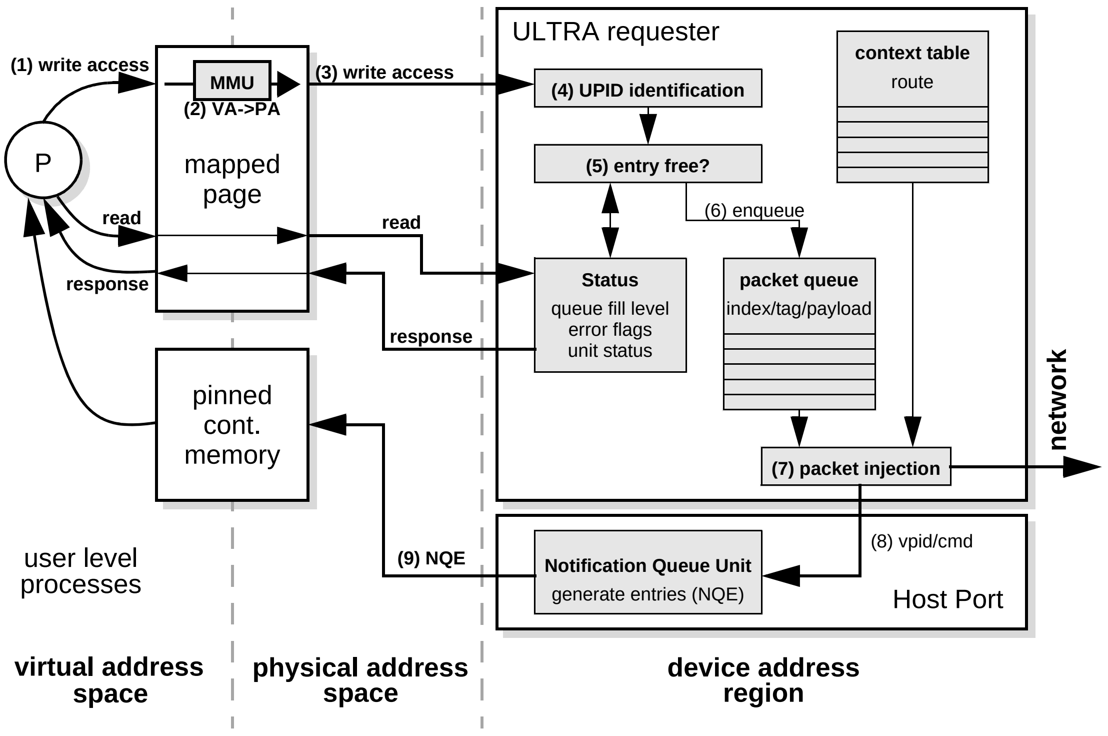


Figure 3.41 ULTRA packet injection

Work flow. Figure 3.41 provides a complete overview of the packet injection procedure. It starts with a process accessing the mapped page to write the packet's tags and payload to the ULTRA requester unit. The used address defines the length of the packet and the selected notification scheme.

1. Process writes to mapped page.
2. The MMU translates the VA into a PA. Only the page number is changed, now containing the port identification. The lower page offset containing the command is unchanged.
3. The requester unit sees a write access to one of it's ports.
4. The UPID of the port is calculated out of the address and the appropriate context is selected.
5. With the pointers stored in the context it is checked if the space in the packet queue is sufficient to store the packet.
   - 5-a. If space is missing the write access is ignored and a notification generated informing the process of the failure.
6. Enough space results in enqueuing the packet.

7. This context is scheduled to inject packets. The route is fetched from the context table and the tags and payload from the packet queue. The packet is assembled and injected into the network.
8. The Host Port's notification unit is used to inform the process. A VPID/command pair is issued to the Host Port, which completes this operation for the requester unit.
9. A notification for this VPID is generated. The command includes the tag and that this notification origins to the ULTRA requester port. The included tag allows to identify the packet.

Beside the access in order to inject packets the process can also read a certain address of the mapped page to obtain informations about the status of it's requester port. Currently this is only used to obtain the fill level of the port's packet queue.

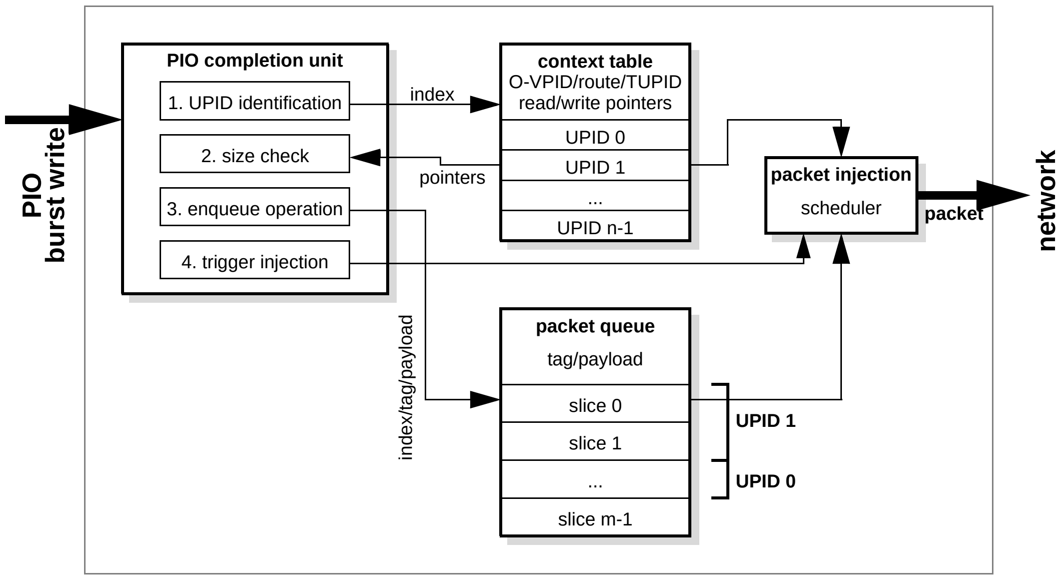


Figure 3.42 ULTRA requester unit

Requester unit. While figure 3.41 provides an overview of the complete packet injection procedure, in figure 3.42 the requester port is depicted in detail. Basically it consists of a PIO completion unit, a context table, a packet queue and a packet injection scheduler.

The *packet queue* is divided into slices. The slice of each port can vary in size. This allows to assign the complete queue to one port if no other is used. For each number of used ports the queue can be optimally utilized. The assignment of slices to ports is a privileged operation and only possible for the management process. For each port a separate data structure which is not shown contains the lower and upper bounds in the packet queue.

The *context table* is indexed with the UPID of the port. For each UPID a corresponding VPID for the origin and the TUPID is stored. The origin VPID is used for the notification on the origin node. The TUPID is required to identify the destination port on the target node. The route to the target node is also part of a context. Last, one read and one write pointer are used to identify occupied and free areas in the appropriate slice of the packet queue.

The *PIO completion unit* determines the UPID, selects the context and performs the check for sufficient queue space. If it is passed successful it enqueues the packet in the port's packet queue slice. Last it triggers the packet injection scheduler of an outstanding packet to be processed.

The *packet injection* includes a scheduler to select a context to be processed. The read and write pointer of this context show if there is outstanding work for this context. If yes, a new packet is generated, starting with fetching the route from the context. The packet is completed with the tags and the payloads from the packet queue slice and finally injected into the network. The read pointer is updated to mark this packet as consumed.

Not shown is the last step, the notification of the process. For this the notification unit of the Host Port is used to reduce the overall complexity. Furthermore the process can rely on a single notification scheme and does not have to check several sources. The packet scheduler sends the corresponding origin VPID together with the tag and a command to the notification unit of the Host Port. There a notification queue entry is enqueued in the VPID's queue. If selected, an interrupt is generated to inform the process.

3.5.5 Packet retrieval

To retrieve an ULTRA packet from the completer unit two different methods are possible. Either the packet is stored within the completer unit and the target process fetches it using a PIO burst read, or the completer unit writes the packets into a data structure in main memory. Then the target process can fetch the packet from main memory. For both methods the I/O interface is used. The more interesting issue for the completer unit is the target process notification of new packets. Interrupts require involvement of the O/S which introduces a large amount of overhead. The other possibility is polling. The target process can either poll on a device register or on a main memory location for changes. A change notifies it of a new packet. The polling on device registers wastes I/O bandwidth so main memory polling seems to be the better choice. Both methods are implemented in the completer unit, but the following explanation and analysis focusses on the main memory queue implementation and polling scheme.

Comparable to the packet injection, the prerequisite for the packet retrieval is the pre-completion of a completer unit. The pre-completion includes to configure a port for a certain source (or origin process). The origin is identified by it's OUPID. Only incoming packets with this OUPID are accepted, others are discarded. The source is now known for all packet retrieved from the completer unit. Only the tags and the payload have to be stored in queues. For queues in main memory furthermore the base address and the length of the queue must be stored in the context of the port.

While a packet retrieval from an on-device queue automatically notifies the completer unit of a consumed packet, an access to a main memory queue is not seen by the completer unit. Here the process must explicitly inform the device of consumed entries by updating it's read pointer. The read pointer together with a write pointer is used to identify free and occupied areas in the queue.

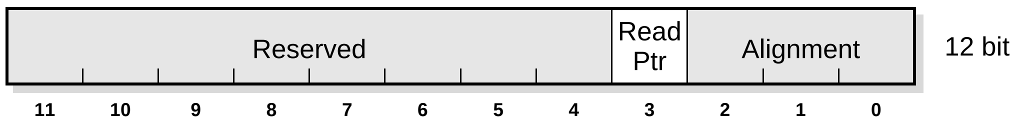


Figure 3.43 ULTRA completer unit command coding

The read pointer is updated using a memory-mapped page. The target process writes the new read pointer to a certain address. The read pointer update is the only supported user-level command and shown in figure 3.43.

Main memory queue. For the receive queue of a completer port a *pinned contiguous main memory region* is used. This region is divided into slices of equal size. Each slice can store one packet. The size of a slice limits the packet size, but this is already limited by the requester unit. Hence the slice size is set according to the requester part. Beside the packet tag and payload, a slice contains a status word. This status word includes a valid identifier and the packet size. The source of the packet is known due to the pre-completion of the completer unit.

The partitioning allows the target process to poll on the status word of a slice for changes. The next slice that will become valid is known because they are written in order. If now an entry becomes valid the status word changes to a non-zero value. Packets are written into the queue using single burst cycles. The status word is located at the end of a slice and hence written last. This ensures that the notification only happens if the packet is already stored.

The partitioning has the disadvantage of wasting memory space if packets not fully utilize the slices. But the possibility to poll a memory location instead of a device register outweighs this. Furthermore, the write pointer for this queue must only be known by the completer unit. The target process does not need access to it, because valid entries are marked by the status word in each slice.

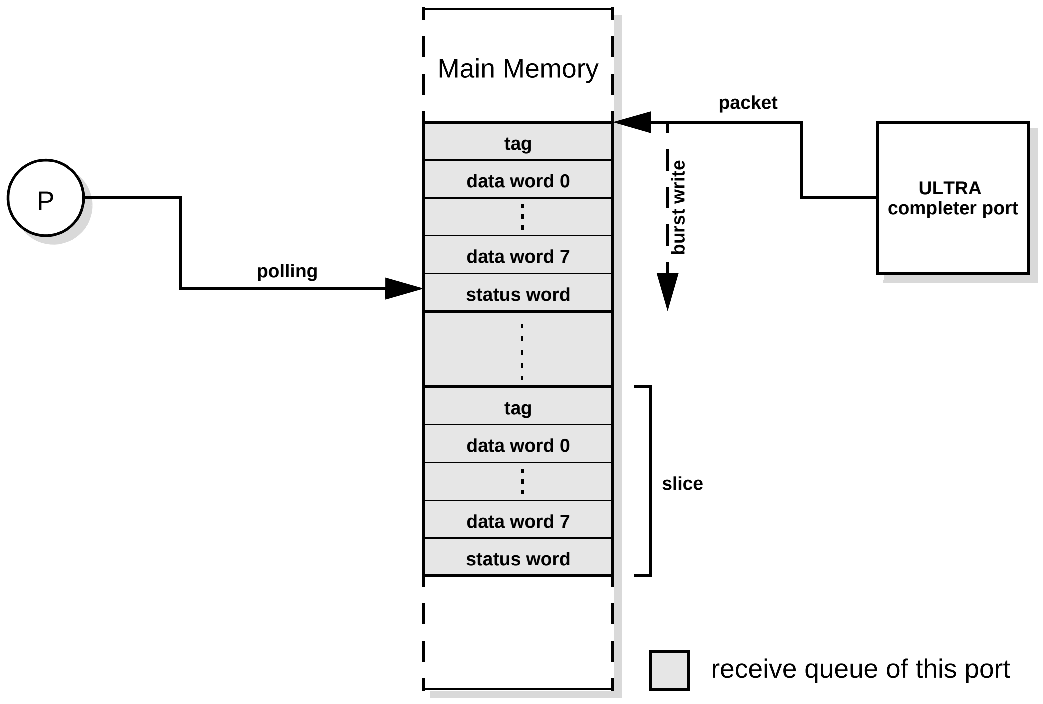


Figure 3.44 Partitioning of the ULTRA receive queue

Work flow. The work flow for the packet retrieval is depicted in figure 3.45. It starts with a packet coming in from the network side.

The detailed procedure to retrieve a packet is composed of the following steps.

1. The packet is retrieved from the network.
2. The TUPID is determined using the packet header. It is used to select the appropriate context.
3. The context contains the matching OUPID for this port and the queue status, described by the read and write pointer. The packet is checked if is comes from this OUPID. The queue pointers show if a receive queue entry is free to store the packet.
4. Only if both checks are successful, the packet is passed to the store unit.
5. Using the queue base address and the write pointer the packet is stored in main memory. The write pointer is updated and written back to the context.
6. If requested an acknowledge about the successful processing is sent back to the origin. For this the acknowledge unit of the Host Port is used. The acknowledge includes the origin VPID to identify the corresponding notification queue at the origin side and the packet tag as identification.

7. The process is continuously polling on the status word of the next free queue slice. As soon as the status word of the queue slice is modified by the store unit, the process is notified of the new entry. It interprets the status word to obtain the packet length.
8. The packet is fetched from the receive queue. The process has to overwrite the status word with zeros in order to recognize a change for the next time this slice is used.
9. The process writes an updated receive queue read pointer back to the device.
10. The MMU of the CPU converts the VA into a PA. The PA now contains as page number the UPID of the ULRA port.
11. The completer unit sees the write access to the address of one of it’s ports. The appropriate context is updated with the new read pointer, marking the queue slice(s) as consumed.
    For the steps 9 to 11 a lazy pointer update scheme is possible reducing the update frequency.

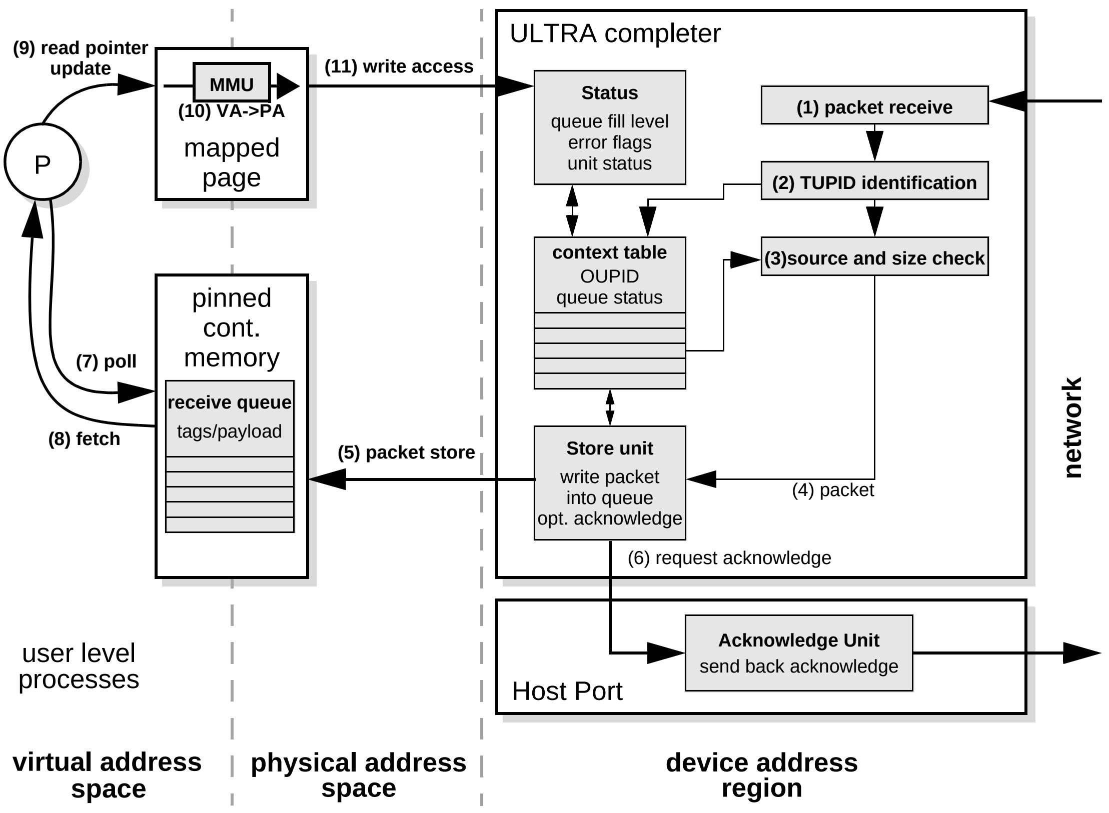


Figure 3.45 ULTRA packet retrieval

The store unit can optionally notify the target process using an *interrupt*. This includes involvement of the ULTRA driver and the required O/S trap is not desirable regarding latency. The interrupt notification should only be used to avoid polling.

The polling on a main memory location has several advantages compared to device register polling. First, the valuable I/O bandwidth is not wasted. In addition, the main memory region containing the receive queue is under control of the *cache coherency protocol*. The cache coherency ensures that the process only accesses the main memory if there was a change. The main memory is only accessed if the first read of the status word results in a cache miss. The data read is also stored in the processor's cache. The next times the status word is read only the cache is accessed, until the main memory location is changed by the completer unit. Then the *write-invalidate policy* of the cache coherency protocol invalidates the cache copy and the following read results again in a cache miss. Now the main memory location is read, returning the most recent value of the status word. The cache coherency protocol thus minimizes the main memory accesses required for polling.

An even more sophisticated method is possible using a *write-update cache policy* instead of the write-invalidate policy applied above. Here the cache line is not invalidated, instead it is updated with the most recent status word. But the cache policy is part of the CPU cache coherency protocol and cannot be influenced by ULTRA.

3.5.6 Initialization of ULTRA paths

Beside the user-level access for packet injection and retrieval by applications, ULTRA units are also accessed by privileged software instances, e.g. drivers or trusted processes.

The privileged accesses are separated from the user accesses by different page addresses. One bit decides if the access address is in user or driver mode (see figure 3.38).

An ULTRA path or connection is established by configuring a requester unit at the origin and a completer unit at the target appropriately. This is also referenced as pre-initialization and pre-completion of ULTRA and can be performed only in driver mode. The driver mode is also used for the partitioning of the packet queue in the requester unit and the configuration of the completer's receive queue (queue base address, length, read and write pointer). The driver is also responsible to keep track of assigned ports, preventing access from multiple processes to one port.

To initialize an ULTRA path and pre-initialize a requester port and a completer port the protocol shown in figure 3.46 is used. The protocol always starts at the target side and consists of six steps. If any of these steps failed an ULTRA path cannot be established due to missing resources.

1. On the target node an unused and available completer port is assigned.
   The TUPID of this port is stored. The completer port is still not enabled.
2. An initialization request is sent to the origin node including the TUPID.
3. Upon the arrival of the request at the origin node one unused and available requester port is opened. The requester port is configured to send packets only to the given TUPID, but not yet enabled.

3-a. If no port is available a failure is sent back to the target node, where the already opened completer port is freed.

4. An initialization response is sent back to the target node, including the OUPID of the opened requester port.
5. On the target node the completer port is configured to accept only packets from the received OUPID and then enabled.
6. The target node signals initialization complete to the origin node.
7. The origin node enables the requester port.

The ULTRA path is now initialized and user-level access is possible to send and receive packets.

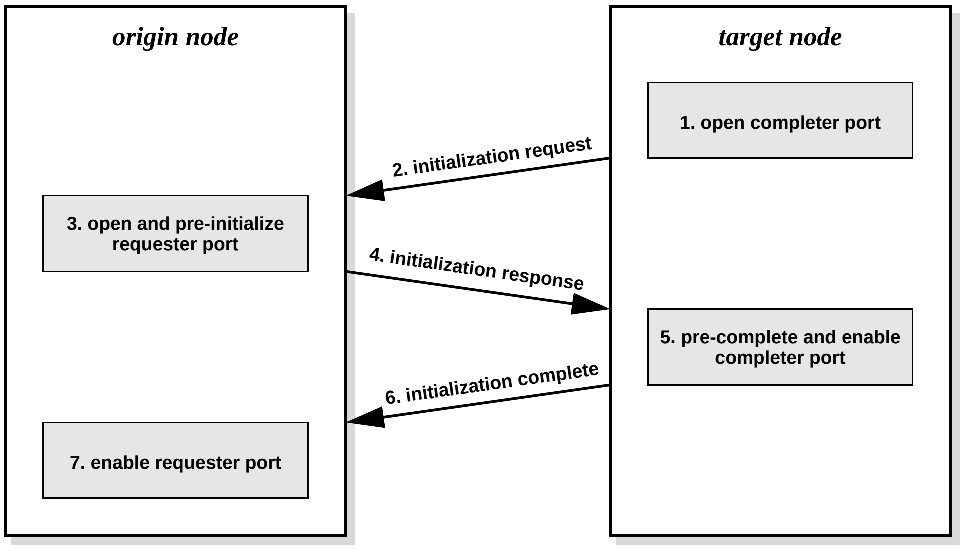

origin node
target node
1. open completer port
2. initialization request
3. open and pre-initialize requester port
4. initialization response
5. pre-complete and enable completer port
6. initialization complete
7. enable requester port

Figure 3.46 Initialization of an ULTRA path

CHAPTER 4 SPECIFICATION AND EVALUATION

This chapter starts with a detailed specification of the design developed in the previous chapters. In particular, the communication instruction set developed in *Chapter 2 ”Communication and Synchronization”* and the network interface architecture (also named Host Port) from *Chapter 3 ”Network Interface Architecture”* is specified.

After the specification follows an evaluation. An FPGA-based board with a PCI interface is used for prototyping. The first tests with this board soon showed that it’s resources are too limited. Because a suitable FPGA board fulfilling all needs was not available, the decision was made to develop a custom one. The developed *HTX-Board* is introduced in sub-chapter *“4.4.1 HTX-Board” on page 168*.

The evaluation itself covers two components and starts with the Triggerpage. It is the building block for the virtualization of the network interface and deserves an evaluation. When the Triggerpage was tested the HTX-Board was not yet available, thus the PCI-based FPGA board is used here. But this shows in particular that even an old I/O interface is sufficient for the Triggerpage. Nevertheless the virtualization will benefit a lot from a more sophisticated and better performing modern I/O interface. After this follows a performance evaluation of the ULTRA unit together with a first real-world measurement[1].

1. The implementation details of ULTRA on the HTX-Board exceeds the scope of this work. The first measurement is only provided to show the performance achievable with the ULTRA architecture.

4.1 SPECIFICATION OF DATA STRUCTURES

In this sub-chapter all required data structures for the supported one- and two-sided communication scheme are collected. A context is defined as basis for the virtualization of the Host Port.

In figure 4.1 an overview of all data structures used for the implementation is shown. All data structures are located in physical contiguous and pinned memory regions.

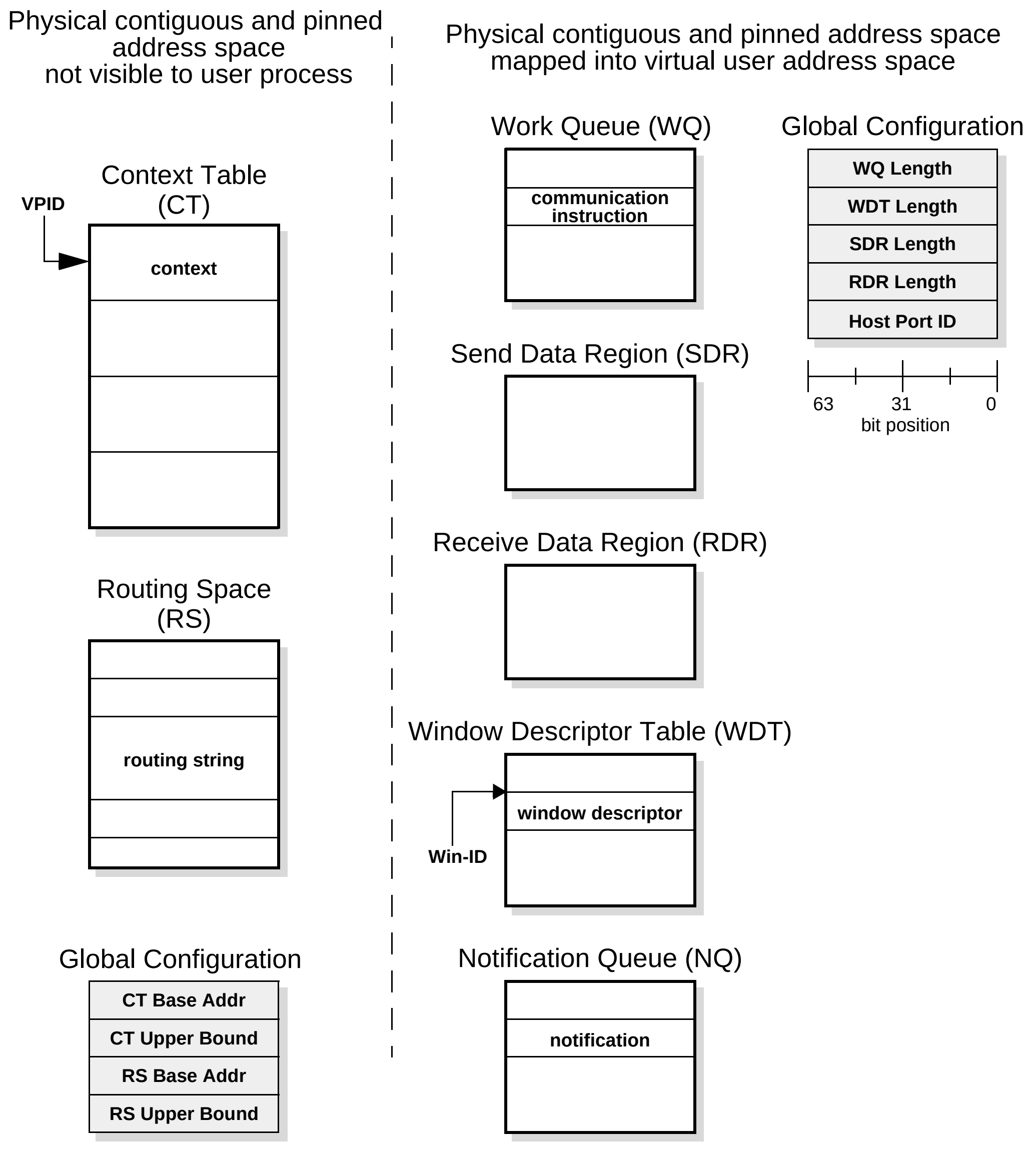


Figure 4.1 Data Structure Overview

Some of the data structures are protected from user access, other can be accessed by user processes. This also applies for the global configuration registers. The location of the context table and the routing space is protected from user access. Other configuration registers like those containing data structure sizes or the Host Port ID are accessible from user space.

Like already explained in sub-chapter *“2.3.7 Interface between process and network interface” on page 53,* the size of all descriptors, table and queue entries is based on the granularity of a cache line. For the x86 architecture the size of a cache line is 64 byte.

4.1.1 Context Table

The *Context Table (CT)* is protected from user space access. For each VPID a context is stored in this table. The table is indexed using the VPID. A loaded context configures the appropriate hardware module for this VPID. Hence references to all other data structures are included a context. The only exception is the routing space, which is shared by all processes. Figure 4.2 shows an entry in detail.

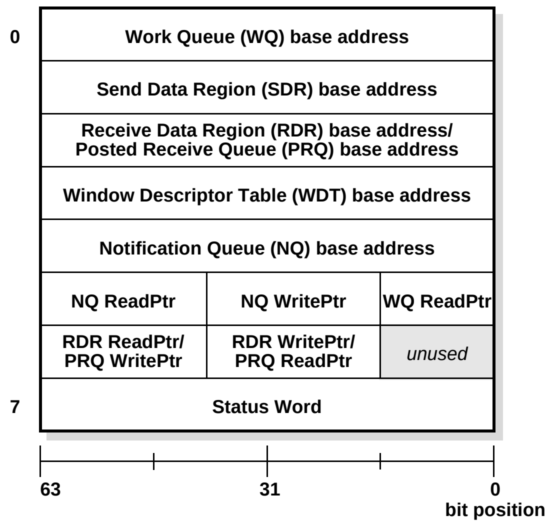

0
Work Queue (WQ) base address
Send Data Region (SDR) base address
Receive Data Region (RDR) base address/
Posted Receive Queue (PRQ) base address
Window Descriptor Table (WDT) base address
Notification Queue (NQ) base address
NQ ReadPtr
NQ WritePtr
WQ ReadPtr
RDR ReadPtr/
PRQ WritePtr
RDR WritePtr/
PRQ ReadPtr
unused
7
Status Word
63
31
0
bit position

Figure 4.2 Context of a VPID

In order to save space in the context, the length of the data structures in a set is equal for all VPIDs. This allows to use global configuration registers to store the size of these regions.

The *Status Word* is used to control a context. It contains in particular a valid bit to activate this context. For unused contexts this bit is not set, preventing accidental accesses to this VPID. Beside this, the Posted Receive scheme can be enabled and additionally configured for physical or virtual addresses (see sub-chapter *“4.1.7 Posted Receive Queue” on page 143*). The use of windows for the RMA schemes can be selected (see sub-chapter *“4.1.6 Window Descriptor Table” on page 142*).

4.1.2 Routing Space

The *Routing Space (RS)* is not directly accessible from user level. In communication instructions a reference to the routing space is included, describing a specific routing string which precedes the packet. Beside these references no pointers are required to control the routing space. A description of the routing functionality can be found in sub-chapter *"2.2.2 Routing" on page 39*.

4.1.3 Work Queue

Each VPID has it's own *Work Queue (WQ)*. The base address of the WQ is stored in the context. All WQs have the same size, configured by a global configuration register.

A work queue entry is a communication instruction. These are specified in detail in the sub-chapter *"4.2 Specification of Communication Instructions" on page 145*.

New work queue entries are signaled to the Host Port using the Triggerpage (see sub-chapter *"4.2.7 Triggerpage" on page 161*). Hence no write pointer is required by the hardware modules. Upon signalling the *WQ read pointer* stored in the context allows them to fetch the next entry.

4.1.4 Send Data Region

The *Send Data Region (SDR)* is also part of the set replicated for each user process. The base address is stored in the context while it's size is globally defined.

A controlling pointer scheme is not required, because this region is only referenced by communication instructions. They contain all required informations to find the corresponding payload in this region. The completion notification of an instruction tells the user process that the referenced payload area in the SDR can be free again.

4.1.5 Receive Data Region

The *Receive Data Region (RDR)* is the counterpart to the SDR. Here the payload for incoming packets based on Send/Receive is stored. This region is controlled by a read/write pointer scheme. The write pointer is updated by the Host Port, while the user process updates the read pointer. This pointer scheme allows the hardware modules to easily find free space in this region.

The read and write pointers are stored in the context. Their size of 24 bit together with a granularity of 64 bit allows an RDR size of 128MB. The write pointer is used internally in the Host Port and additionally included in notifications of received packets.

The user process has to frequently update the read pointer to mark payloads in the RDR as consumed.

4.1.6 Window Descriptor Table

The *Window Descriptor Table (WDT)* is also replicated for each user process. The base address is stored in the context and it's size is globally defined. The windows are used to protect the user space for RMA. Windows can be opened for exclusively one remote process, a group of processes or all. Furthermore a window can be temporally locked to ensure mutual exclusion. All this is achieved using a *rights flag* (or *capability*) [48] in the window descriptor. The RMA instruction must include a *Window Identification (Win-ID)* and the matching capability to be processed.

The window descriptor is shown in figure 4.3. It consists of a *window start address* (virtual address), the *length*, the rights flag and two pointers to the previous and next window in a set.

If a set of windows is opened on the same user space region, the *previous and next pointer* are used to form a linked list. This allows software layers to easily walk through all window descriptors, for instance in the case that all windows except one must be locked.

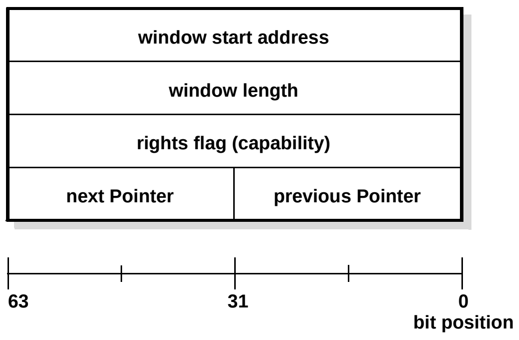


Figure 4.3 Window Descriptor

4.1.7 Posted Receive Queue

In order to receive packets using a Posted Receive scheme, descriptors are required which specify regions in user space to be used. The payload is directly written into user space with the help of the on-device TLB. The translation requires a VPID, a Win-ID and an address inside this window.

The *Posted Receive Descriptors (PRD)* are stored in the *Posted Receive Queue (PRQ)*. Because either the normal Receive scheme or the Posted Receive scheme is used, the RDR can be used as PRQ. The configuration is set in the context. The RDR read/write pointers are then used as PRQ read/write pointers. Now the user process inserts PRQs and the Host Port consumes them. The RDR read pointer is now the PRQ write pointer (updated by the user process) and the RDR write pointer becomes the PRQ read pointer (updated by Host Port). The PRD shown in figure 4.4.

Optionally, the Posted Receive scheme can be configured to write back data using physical addresses instead of virtual ones. Then the Win-ID is omitted in the PRD above and the address is a physical one.

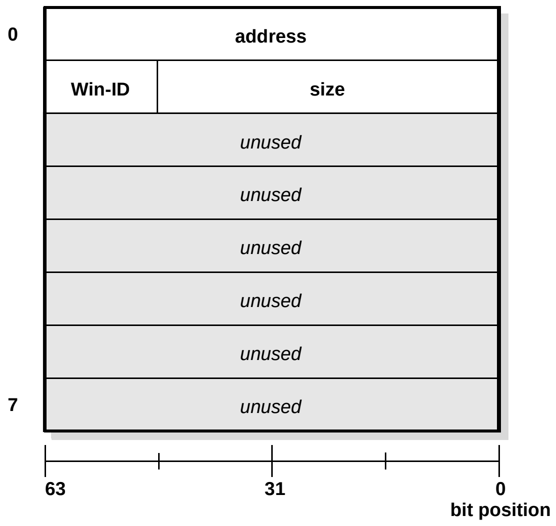


Figure 4.4 Posted Receive Descriptor (PRD)

Future developments based on this work may result in more sophisticated PRDs, for instance matching the origin or the tag of an incoming packet.

4.1.8 Notification Queue

The *Notification Queue (NQ)* contains notifications for all events that occur and must be signaled to the user process. In contradiction to the WQ the entries are enqueued by the Host Port and consumed by the user process. Each context has it's own NQ, hence the offset of this queue is stored in the context. The size is globally defined in a configuration register.

The context also contains the read and write pointers for this queue. The user process as a consumer increments the read pointer, while the Host Port updates the write pointer. Each entry in the queue additionally includes a valid identifier, allowing the user process to poll on the next queue entry to become valid.

The entries of this queue are notification descriptors. They are explained in detail starting with sub-chapter *"4.2.4 Overview of notification descriptors" on page 154*.

4.2 Specification of Communication Instructions

Communication instructions are passed from user processes to the Host Port in order to process work. The WQ is used to store these instructions as work request entries. The Host Port processes these instructions and signals their completion back to the user process using notifications. Notifications are entries in the NQ.

This sub-chapter starts with the specification of the supported communication instruction set, followed by the notifications. After this, the use of the Triggerpage is shown, including it's layout and the supported operations (for instance instruction issue or pointer update).

4.2.1 Overview of communication instructions

Only instructions for the Host Port are passed using descriptors. This sub-chapter includes these communication instructions. These instructions are issued using the Triggerpage. The instructions for the ULTRA unit require a special send and receive scheme and are not part of this sub-chapter.

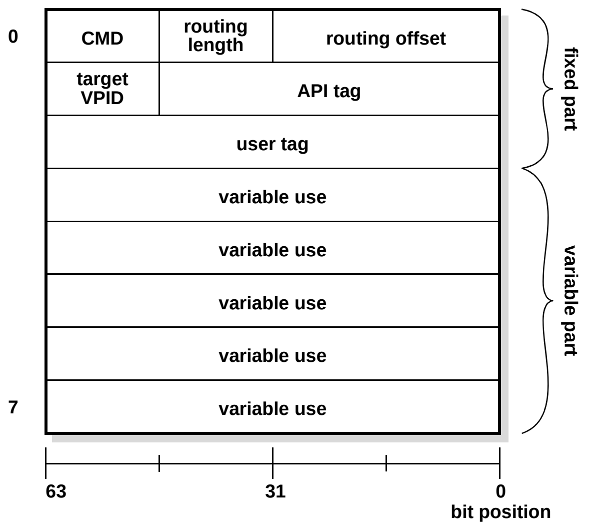


Figure 4.5 Virtual Communication Instruction (VCI) descriptor

The communication instructions are passed from the process to the Host Port. They are stored as descriptors in work queues located in main memory. The virtualization of the Host Port leads to naming these instructions *Virtual Communication Instructions (VCI).*

The size of a VCI is fixed and matches the size of a cache line (64 bytes) for the x86 architecture (see also sub-chapter *“2.3.7 Interface between process and network interface” on page 53*). Figure 4.5 shows that a VCI consists of a fixed part and a variable part. The fixed part is the same for all VCI types, while the use of the variable part is dependent on the type.

The fixed part of a VCI contains the *command (CMD)* field, the routing information and the packet tags. In the processing of a VCI the command must be interpreted first, hence it is located at the beginning.

The next entries contain the routing information. The *routing offset* and *routing length* are used to fetch the appropriate routing string from the routing space (see sub-chapter *“2.2.2 Routing” on page 39*). The fetched routing string is extended to include the *target VPID* as last element[1].

From the point of view of an API or application, the *routing handle* to determine the desired destination is composed of the routing offset, routing length and target VPID.

The next entry in the VCI descriptor are the packet tags. Two tags are available to describe the packet. The normal *user tag* with a size of 64 bit can be used by applications. The *API tag* is dedicated for middle-ware software layers or APIs. In this tag additional packet informations like sequence numbers can be stored.

4.2.2 Command coding

The first entry of a VCI descriptor contains the command. In the following figure 4.6 the command coding overview is shown. The command is separated into three parts: the error code, the command group and the (sub-)command including the parameters.

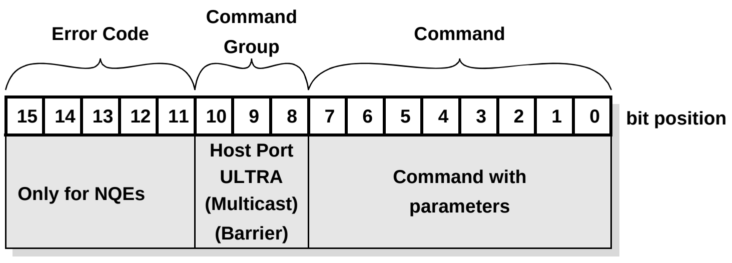


Figure 4.6 Command coding overview

Error Code. In a VCI descriptor the error code is always set to zero. After processing this VCI a *Notification Queue Entry (NQE)* is generated to inform the process of the completion. This notification queue entry contains a copy of the command provided with

1. The target VPID could alternatively be included in the routing space instead of the VCI. But this requires an routing entry for each VPID at the target node, which significantly increases the required size for the routing space. Because of this the target VPID is included in the VCI.

the VCI, but the error code is then used to include information about success or failure. The error codes are described in detail in the sub-chapter *“4.2.6 Notification error codes” on page 160*.

Command Group. The Host Port is only a part of the interconnection network introduced in sub-chapter *“2.2 Integration into the Interconnection Network” on page 38*. Depending on the corresponding module, the command group allows to differentiate the commands. This work here only covers commands for the Host Port and the ULTRA unit. The other communication functions are a multicast and a barrier, additionally in the future others might be developed. The coding of the command group is shown in figure 4.7.

Not all commands are issued using the Triggerpage, but for all commands are entries in the same notification queue generated. Because of this the commands are strictly separated into different groups.

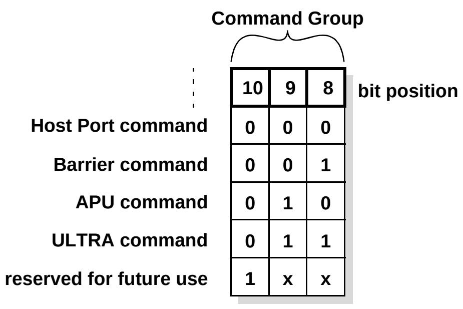


Figure 4.7 Command group

Command. The last part contains the command itself. If the command requires parameters, they are included in the command coding. The applied scheme is shown in figure 4.8. If applicable the parameters are always located at the same bit position.

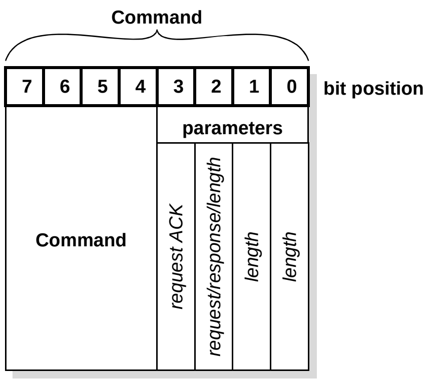


Figure 4.8 Command coding scheme

The resulting coding of commands is shown in *Table 4.1*. The three most right columns show for which data structures the appropriate table row applies. The work queue only includes VCI commands. The next column shows the commands possible in the command frame of a packet, and the third column shows available commands for the notification queue.

Table 4.1: Command Coding [1] [2]

Bit Position								***Command***	***Additional Notes***	***Applies for***		
7	***6***	***5***	***4***	***3***	***2***	***1***	***0***			***VCI***	***CMD Frame***	***NQ***
0	0	0	1	A	L	L	L	FAST_SEND	LLL: # Direct Data Words	X	X	X
0	0	1	0	A	x	L	L	FAST_PUT	LL: # Direct Data Words	X	X	X
0	0	1	1	x	0	L	L	FAST_GET	LL: # Direct Data Words	X	X	X
0	0	1	1	x	1	L	L	FAST_GET_RESPONSE			X	
0	1	0	0	A	x	0	0	MISALIGNED_PUT		X	X	X
0	1	1	0	x	0	1	0	FAA	Fetch-And-Add Atomic Operation	X	X	X
0	1	1	0	x	1	1	0	FAA_RESPONSE			X	
0	1	1	1	x	0	1	1	CAS	Compare-And-Swap Atomic Operation	X	X	X
0	1	1	1	x	1	1	1	CAS_RESPONSE			X	
1	0	0	1	A	x	x	x	SEND		X	X	X
1	0	1	0	A	x	x	x	PUT		X	X	X
1	0	1	1	x	0	x	x	GET		X	X	X
1	0	1	1	x	1	x	x	GET_RESPONSE			X	
0	0	0	0	0	0	0	0	NQE_INVALID	Invalid NQEs			X
1	1	1	1	0	0	0	0	NOTIFY_CC	Notification of Com-mand Completion			X
				0	0	0	1	NOTIFY_RMA	Notification of RMA			X
				0	0	1	0	NOTIFY_FAST_RECV	Notification of FAST_RECEIVE			X
				0	0	1	1	NOTIFY_RECV	Notification of RECEIVE			X
1	1	1	1	1	1	1	1	ACK	Acknowledge		X	

1. x: Don't care value
2. grey: main CMD, green 'x': don't care, yellow: request/response pair, red 'L': length of the immediate value, blue 'A': Acknowledge required or not (1=Acknowledge, 0=no Acknowledge.)

4.2.3 Communication instruction descriptors

The fixed part of a VCI descriptor is the same for all types of communication instructions, while the variable part is dependent on the type. In the *Table 4.1* already provides a list of available communication instructions (marked with VCI column set). The descriptor formats for these VCIs are now explained in detail.

Send. The Send instruction requires the payload to be stored in a dedicated buffer. Hence the descriptor for this instruction includes a reference to the payload, specifying it’s exact location and length. The lower and upper bound for this buffer are stored in the VPID context.

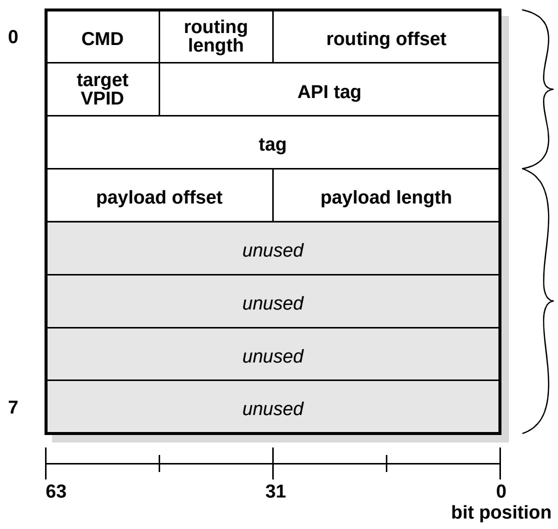


Figure 4.9 Send VCI descriptor

The size of the payload is only limited by the size of the buffer. The buffer sizes on origin and target side must be considered.

This two-sided communication method involves both the origin and the target in the communication. The target decides which receive scheme is most appropriate. The origin cannot influence this.

Fast Send. A Fast Send instruction includes the payload as immediate value in the descriptor, rather than referring to other data structures. This saves one additional memory access compared to the Send instruction. But the size of the payload is limited by the available space in the descriptor. The command coding includes the length of the payload.

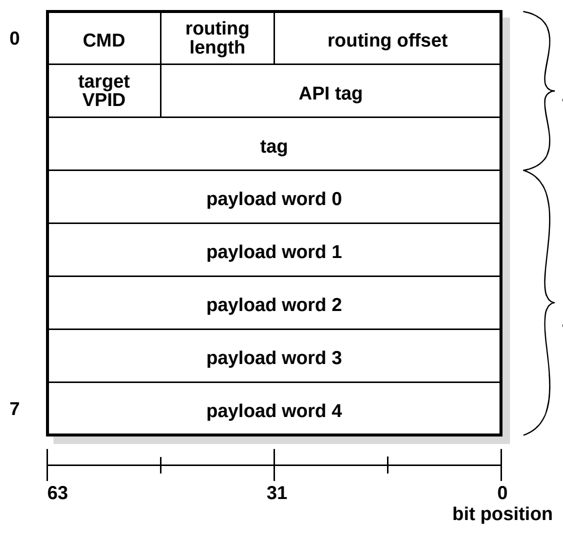


Figure 4.10 Fast Send VCI descriptor

Figure 4.10 shows that up to 5 words of each 64 bit can be transferred using a Fast Send operation.

This two-sided communication method involves both the origin and the target in the communication. Again the target decides which receive scheme is most appropriate.

Put and Get. At the origin side the Put instruction directly fetches the payload from user space. No dedicated buffers are required. At the target side the payload is directly written back into user space without the involvement of the target process. A Get instruction is similar but reads the payload from the target and writes it back at the origin. These two instructions only differ by the direction of the data transfer.

To protect the user space memory regions from unwanted access and to allow mutual exclusion, windows are used on both the origin and target side. The location of the payload is relative to the window offset. A Put or Get instruction includes an o*rigin and target window identifier (Win-ID)*, an origin and target address for the payload and the payload size. While the origin window is only checked if it is locked, the access to the target window must be verified with a *capability* (which is basically a key).

The size of the payload is only limited by the size of the window. The window sizes on origin and target side must be considered.

This one-sided communication method involves only the origin actively in the communication. The target process can optionally register itself to be notified of an access, but it does not participate in the data transfer.

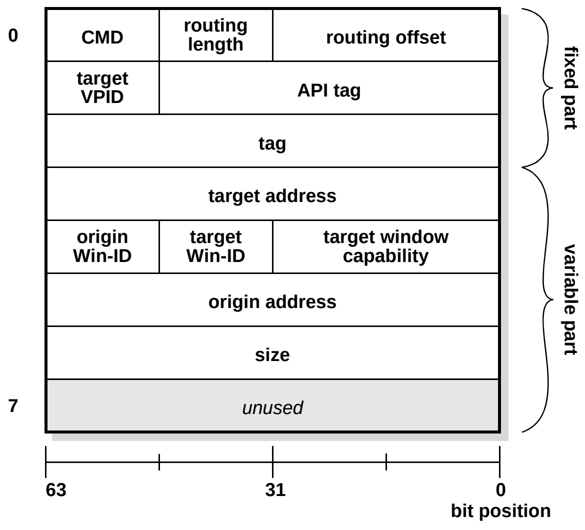


Figure 4.11 Put/Get VCI descriptor

Fast Put and Fast Get. Instead of including a reference to fetch the payload from user space, the Fast Put instruction includes the payload as an immediate value. This saves one memory access. At the target the payload is written into user space. Because no origin window must be provided the origin Win-ID is omitted. The length of the payload is included in the command coding for this instruction.

Similar applies for the Fast Get instruction. The payload is fetched at the target from user space. Back at the origin the data is not written into user space, instead it is included in the notification queue entry which informs the origin process of the completion.

Figure 4.12 shows that the space in the descriptor is sufficient for up to three data words as payload.

This one-sided communication method involves again only the origin actively in the communication. The target process can optionally register itself to be notified of an access, but it does not participate in the data transfer.

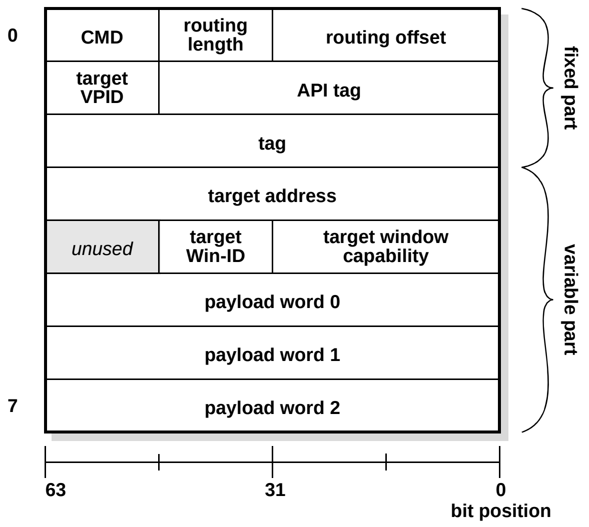


Figure 4.12 Fast Put/Fast Get VCI descriptor

Misaligned Put. While all previous instructions always transfer data based on the granularity of 64 bits, the misaligned instruction here include a *Byte Enable (BE)* entry. The BE allows to mask the payload word. The granularity of the BE is 8 bits. For a word size of 64 bit the resulting size of the BE entry is 8 bit. Each bit of the BE enables or disables one byte of the data word.

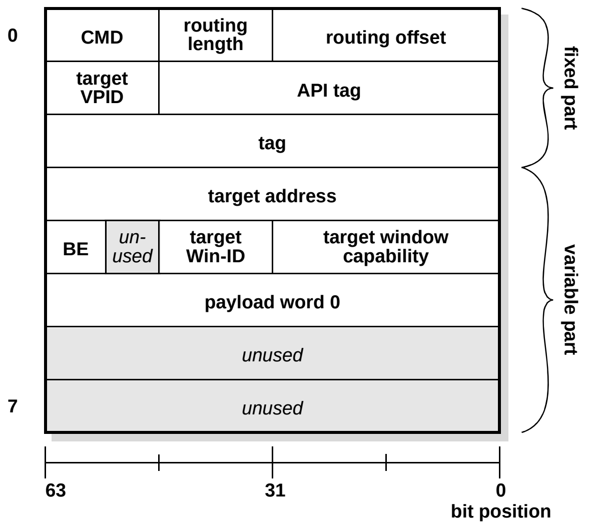


Figure 4.13 Misaligned Put VCI descriptor

This instruction only has to be used if upper software layers are based on a data word granularity smaller than 64 bit. To allow remote put operation without overwriting unwanted parts of the destination, this instruction must be used.

While a Misaligned Put is inevitable for correct behavior of the above mentioned software layers, the Misaligned Get can be omitted. For a Get the data is only read. If only parts of the payload word are requested, the origin process can easily extract them from the delivered payload.

The target process can still optionally register itself to be notified of an access, but it does not participate in the data transfer. Only the origin actively participates in the communication.

Fetch and Add. The Fetch and Add instruction is the first of two supported atomic operations. The target address is read, added with the *addend* included in the instruction and written back. This *result* is also returned and included in the NQE.

If the I/O interface of the target node supports atomic operations, they should be used. Otherwise the method described in sub-chapter *“2.3.4 Remote Memory Access (RMA) communication scheme” on page 47* can be used to overcome this limitation.

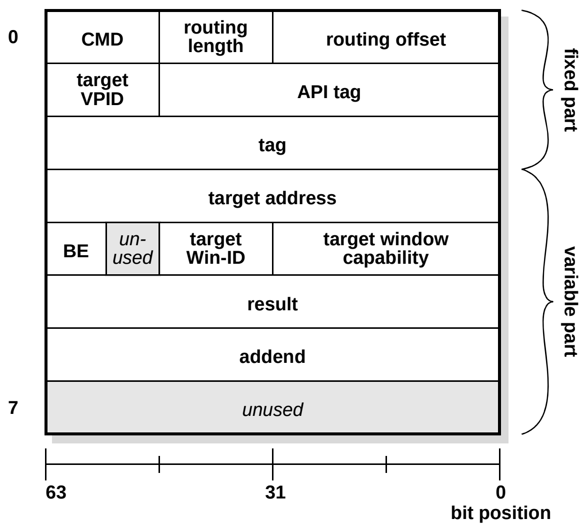


Figure 4.14 Fetch and Add VCI descriptor

Again only the origin actively participates. The target process can optionally register itself to be notified.

Compare and Swap. The second atomic operation differs from the previous only by the atomic read/modify/write scheme. The target address is read and compared to the provided *compare value*. Only if they are equal the *swap value* is written to the target address. Then the swap value is returned as *result,* otherwise the original value.

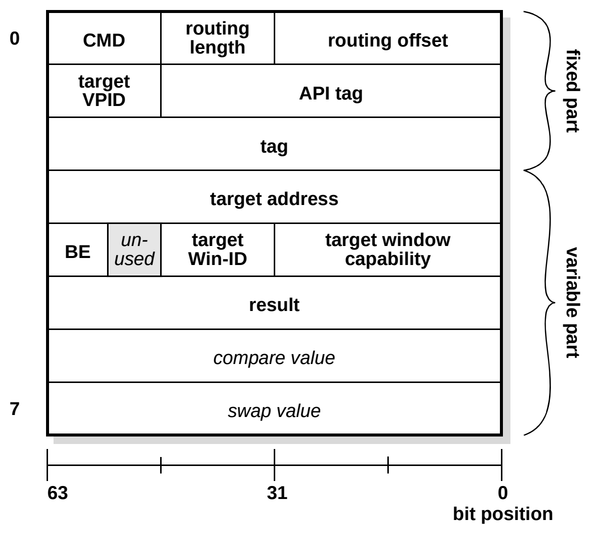


Figure 4.15 Compare and Swap VCI descriptor

4.2.4 *Overview of notification descriptors*

Notification descriptors are entries in the NQ. NQEs contain a command, which allows the receiving process to recognize which event has happened. The coding is already shown in *Table 4.1*.

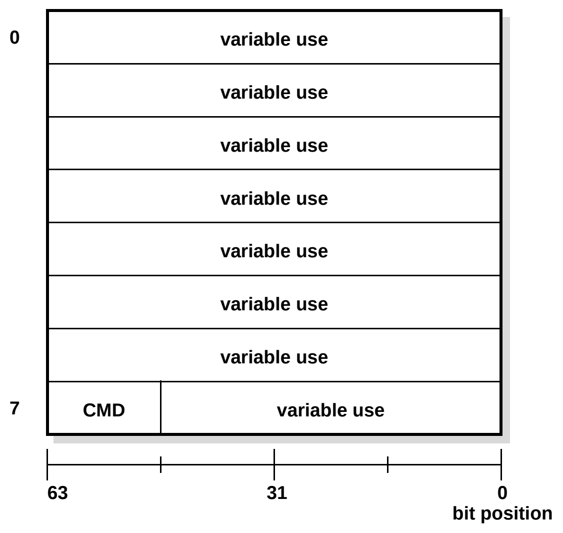


Figure 4.16 Notification descriptor

The size of an NQE is again one cache line (64 bytes). It can be separated into a fixed and a variable part. The fixed part only contains the CMD entry. Parts of the CMD are used as error code. Figure 4.16 shows the descriptor for an NQE.

The CMD is located at the end of the descriptor. This allows the process to poll on this location for a change, instead of polling on the write pointer of this queue. Then during normal operation the process does not need to access the write pointer.

4.2.5 Notification descriptors

Now follows a description of all notification queue entries. Their command coding is already shown in *Table 4.1*. Each NQE is based on Figure 4.16.

There are several reasons why an event is notified to a process. Either a communication instruction is completed, a remote access occurred, a packet was received or the status of the corresponding context was requested and is now returned in the NQ.

Completion Notification. If a work request is completely processed the origin process is notified. If the work provides a result, it is included in the notification as an immediate value or as a reference to another data structure. If the work cannot be processed due to certain errors, these errors are also included. Figure 4.17 shows the descriptor format when no immediate values are included and figure 4.18 the descriptor format with included immediate values.

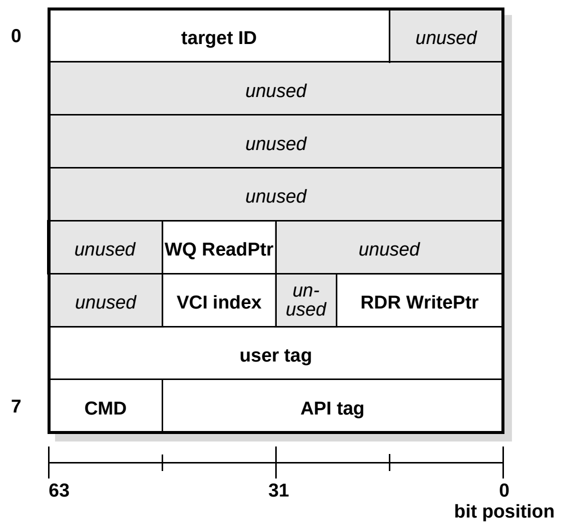


Figure 4.17 Completion notification descriptor without immediate values

The notification includes some informations from the corresponding command instruction, allowing to identify the original instruction. Beside the index of the communication instruction (*VCI index*) these are the *user tag* and the *API tag*. The routing handle is replaced with the *target ID*. This target identification consists of the unique target node identification (Host Port ID, 32 bit) and the target VPID (16 bit). This allows to identify the counterpart for this operation.

Beside this, a snapshot of the most recent pointer values is included. Only the pointers controlled by the Host Port (*WQ ReadPtr* and *RDR WritePtr*) are necessary. This snapshot is only used as optimization, reducing the number of status requests issued by the process.

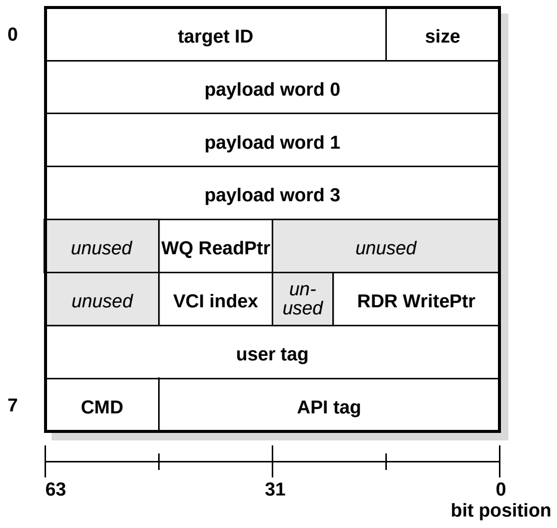


Figure 4.18 Completion notification descriptor with immediate values

Some RMA instructions like Fast Get and atomic operations include the result as immediate value. The corresponding descriptor is shown in figure 4.18. In the *size* field the number of valid payload words is given, and the *payload words* contain the immediate data.

Remote Access Notification. A passive target of an RMA instruction can register itself to be notified when such an RMA takes place. This allows to use RMA instructions also for synchronization.

Figure 4.19 shows the descriptor layout. It contains the counterpart identification (*origin ID*), the tags of the remote access (*user and API tag*), the local window identification (*target Win-ID*) and the *address* and the *size* of the access.

Again for optimization, the pointer snapshot is included. The VCI index is omitted because this information is only meaningful for the origin process.

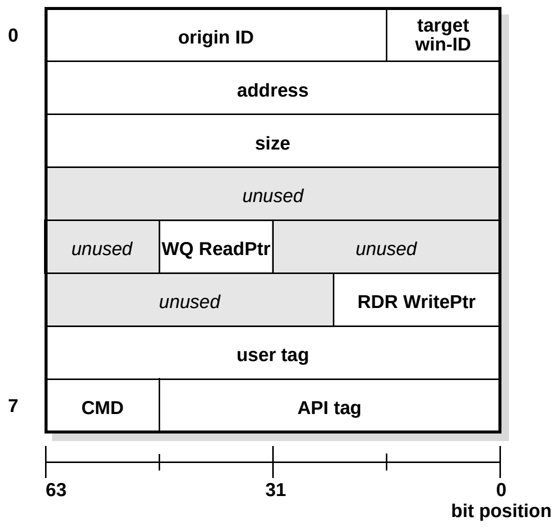


Figure 4.19 Remote access notification descriptor

Receive Notification. The receiving of packets takes place automatically. The target Host Port receives packets up to a size of 5 words as Fast Receive. Depending on the configuration of the Host Port, larger packets are either stored in the RDR or received as Posted Receives and stored directly in user space.

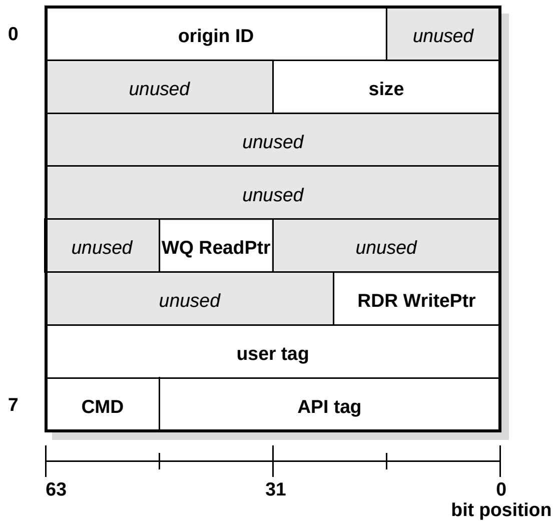


Figure 4.20 Receive notification descriptor

No involvement of the target process is required expect ensuring that enough buffer space for incoming packets is available. The completer unit stores the incoming packet in the appropriate buffer and notifies the target process that a new packet is available. This is done by inserting an NQE in the queue.

Figure 4.20 shows the Receive descriptor. It includes the origin of the packet (*origin ID*), the *size* (it is limited by the corresponding Send instruction) and the tags (*user and API tag*). This time the pointer snapshot is not only for optimization, because the process requires the updated *RDR write pointer* to know the exact location of the payload within this buffer.

Fast Receive Notification. If the payload size of an incoming packet is 5 words or less it fits as immediate value in the notification descriptor. The pointer snapshot is omitted in this kind of notification in order to store more payload data.

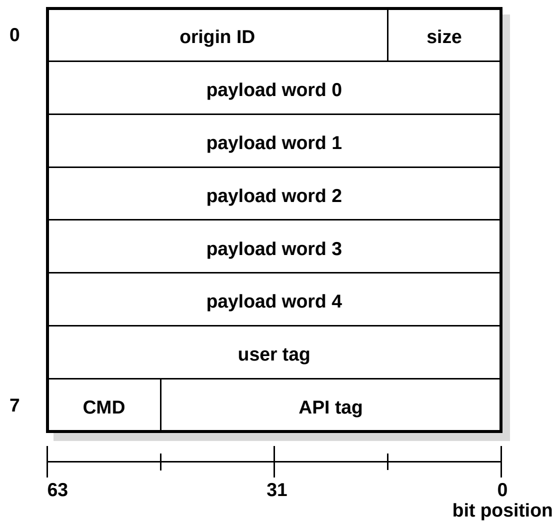


Figure 4.21 Fast Receive notification descriptor

Posted Receive Notification. The Host Port can either be configured to receive packets using the dedicated buffer (RDR) or to use Posted Receives. The first case is already explained in figure 4.20.

If Posted Receives are used, the RDR is used as a PRQ. The process inserts entries and the Host Port consumes them. Each descriptor specifies regions in user space where incoming packets can be stored. These queue entries are consumed in order. If the size of an incoming packet exceeds the size of a Posted Receive descriptor, an error occurs. Otherwise it is stored in user space and a notification shown in figure 4.22 is generated.

The notification now contains a *PRQ read pointer.* This shows the last consumed descriptor which was used to store the described packet.

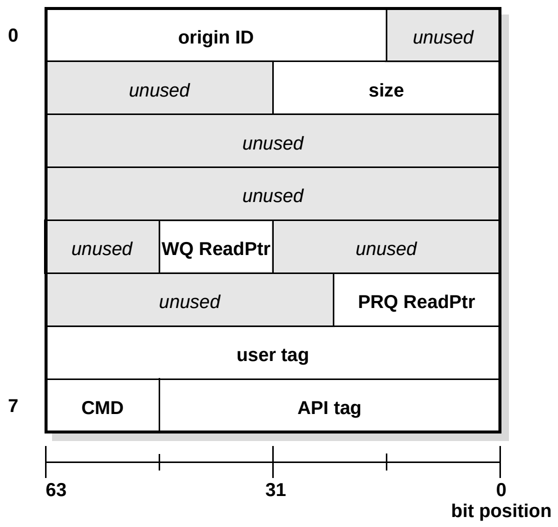


Figure 4.22 Posted Receive notification descriptor

Status Notification. The process can request a status snapshot containing the pointers of the data structures. The update of a process controlled pointer also results in such a notification. The generated notification descriptor always contains all pointers of this context, both these controlled by the process and by the Host Port.

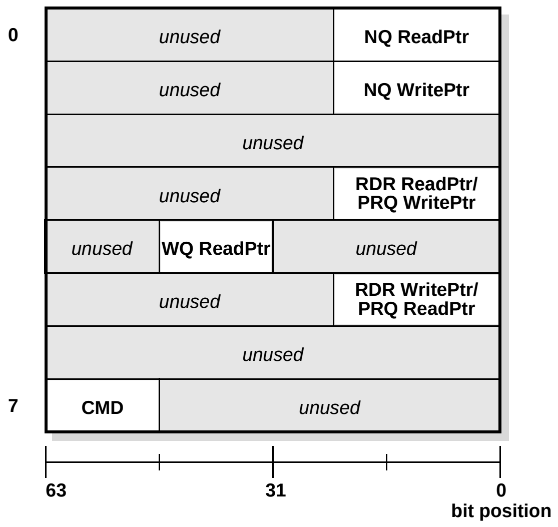


Figure 4.23 Status notification descriptor

In the case that Posted Receives are used the RDR is used as PRQ. The read and write pointer is exchanged. This is indicated in the figure above by naming both possibilities.

4.2.6 Notification error codes

The command field in the notification descriptors above contains also the error code for the corresponding instruction. The following table shows the coding of these errors. A detailed description can be found in [4].

Table 4.2: Notification Error Codes

No.	*Code*	*Name*	*Description*
0	00000	ERR_NOERR	Obviously no error occurred
1	00001	ERR_CMD_INV	Invalid command
2	00010	ERR_OVPID_INV	Invalid origin VPID
3	00011	ERR_ROUTE_INV	Route exceeding upper bound / routing length=0
4	00100	ERR_OWINID_INV	Invalid origin Win-ID (disabled or exceeding upper bound)
5	00101	ERR_OWINID	Segmentation fault on origin Win-ID
6	00110	ERR_OOFFSET	Misaligned origin offset
7	00111	ERR_OLENGTH	Misaligned origin length
8	01000	ERR_TVPID_INV	Invalid target VPID
9	01001	ERR_TWINID_INV	Invalid target Win-ID (disabled or exceeding upper bound)
10	01010	ERR_TWINID_CAPA	Invalid target Win-ID capability
11	01011	ERR_TWINID	Segmentation fault on target Win-ID
12	01100	ERR_TOFFSET	Misaligned target offset
13	01101	ERR_TLENGTH	Misaligned target length
14	01110	ERR_ROUTE_BROKEN	Destination tag does not match or destination cannot be reached
15	01111	ERR_POSTED_PA	Posted Receive: PA not allowed
16	10000	ERR_POSTED_PA_INV	Posted Receive: Invalid PA, offset or length not aligned
17	10001	ERR_POSTED_TRUNC	Posted Receive: Message truncated

4.2.7 Triggerpage

The Triggerpage is a new method developed in this work which allows multiple producers to enqueue elements in a central shared queue. Each access to the Triggerpage includes flow control which informs the producer of the result of the enqueue operation. Hence prior checking for available space is not required, because the producer is informed of insufficient space by the returned result of the operation. For the application here the Triggerpage is used for all accesses from (user) processes to the virtualized device except ULTRA.

A user process needs to access the Host Port for several reasons, but all are related to issuing instructions. Hence several issue commands are available for a Triggerpage access.

The enqueuing of new VCIs in the WQ must be signalled. The consumption of NQEs and payloads retrieved from the RDR require to update the corresponding pointer stored in the context. If the Posted Receive scheme is used, the process must provide new PRDs describing regions in user space to store incoming packets. The insertion of new PRDs is also signalled by pointer updates to the Host Port. The context is not directly accessible by the process, all accesses are secured using the Triggerpage.

Triggerpage layout. In the last chapter the Triggerpage is introduced. A Triggerpage access is always a read, never a write. *Figure 3.30 on page 115* already shows that the address of a read access to the Triggerpage is used to encode parameters. Figure 4.24 now shows the address layout in detail. The lowest three bits cannot be used due to an alignment to 64 bit units.

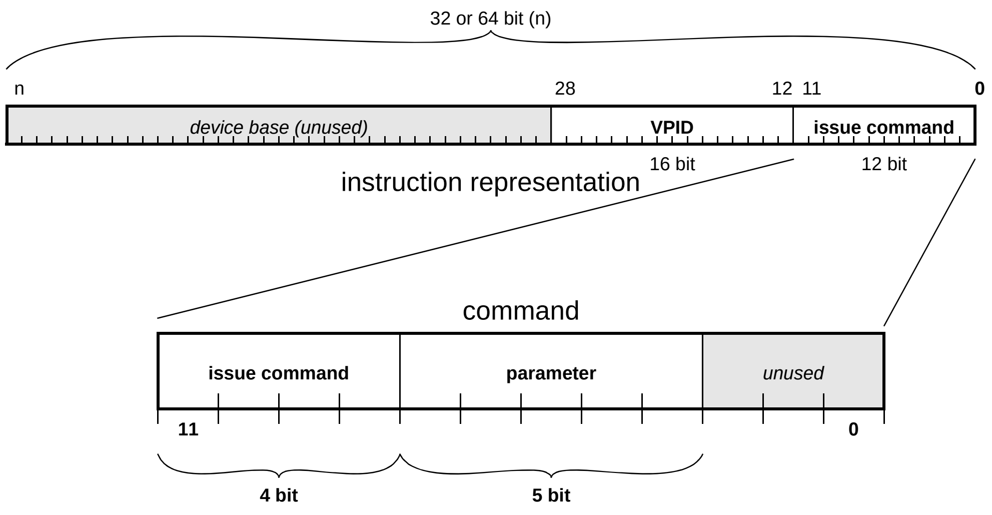


Figure 4.24 Triggerpage address Layout

The VPID of each access is used to identify the corresponding context. Using the command part, divided into the issue command itself and a parameter, the desired operation is specified.

Triggerpage Issue Commands. Using the issue command several operations can be specified. The following table provides an overview of the available operations for the Triggerpage.

The first operation (*Issue VCI*) issues a given number of new VCIs to the Host Port. These VCI have previously been inserted in the WQ and now the Host Port is notified of these new entries. The Host Port may decide not to accept all VCIs, for instance when the CSB behind does not provide enough free entries. The return value always contains the number of accepted VCIs.

Using the next operation (*Request Pointer Snapshot*) the process can get a snapshot of the pointer set stored in the context. No parameter is used here.

Table 4.3: Triggerpage Issue Command Overview

Issue Command	Coding	Parameter	Description
Issue VCI	0000	Count	Issue *count* numbers of VCIs
Request Pointer Snapshot	0001		Request an NQE with current pointer/status set
Increment NQ RP	0010	Count	Increment NQ read pointer by *count* units
Increment RDR RP	0011	Count	Increment RDR read pointer by *count* units
Barrier Reached	0100	barrier_id	Notify that barrier with ID *barrier_id* is reached

The next two operations (*Increment NQ RP* and *Increment RDR RP*) are used to update the appropriate read pointers by incrementing them. The parameter is used to provide the number of units, which are added to the current read pointer.

The last operation is used to signal *Barrier Reached* to the Host Port. The exact functionality of the implemented barrier is beyond the scope of this work. Because several barriers are available, the parameter encodes an identification of the barrier.

The result of the operations above is returned in the read answer. Figure 4.25 shows the return value.

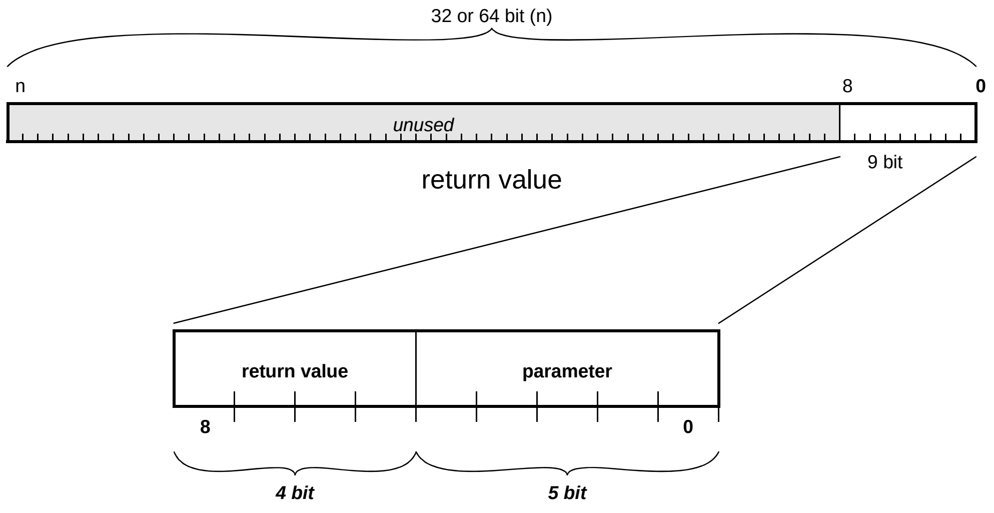


Figure 4.25 Triggerpage return value

The following table describes the possible return values. A detailed specification of the behavior can be found in [94].

Table 4.4: Triggerpage Return Values

Error	Coding	Parameter	Description
No Error	0000	count	Operation succeeded, *count* VCIs issued
Partially Executed	0001	count	Operation succeeded, but only *count* VCIs issued
CSB full	0010		Operation failed due to full CSB, come back later
Invalid	0011		Operation failed due to invalid command
Burst Error	0100		Operation failed due to burst read access
Forbidden	0101		Operation failed due to missing permissions
Disabled	0110		Operation failed due to disabled device

4.3 SPECIFICATION OF THE HOST PORT ARCHITECTURE

The insights gained and developed in *Chapter 2 ”Communication and Synchronization”* and *Chapter 3 ”Network Interface Architecture”* here are now combined to form the final network interface. The architecture is combined with the queue design, the virtualization technology and the ULTRA unit to support all communication and synchronization requirements of typical cluster applications.

Focus is set on a complete view of the architecture. The details of modules, building blocks, data structures and working principles are explained in the appropriate sub-chapters before.

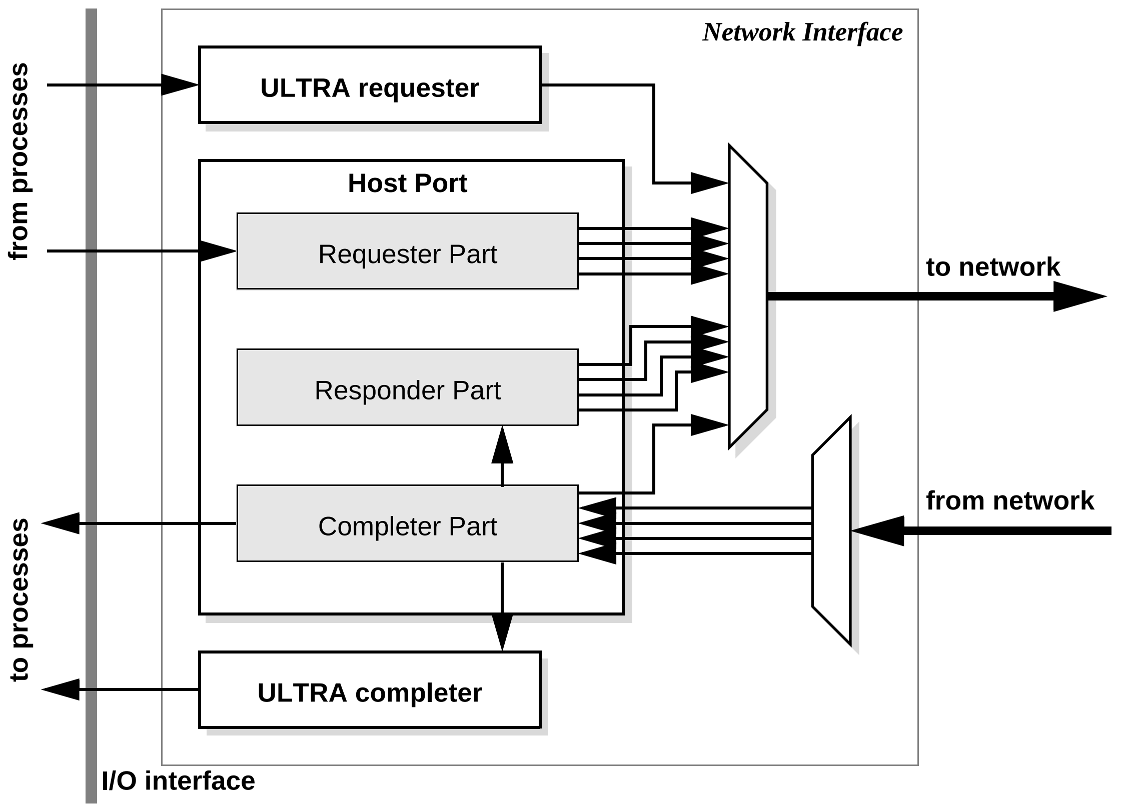


Figure 4.26 Host Port top-level block diagram

In figure 4.26 the top-level block diagram of the Host Port is depicted together with the integration into the network interface. The Host Port basically consists of three parts. Each part houses the appropriate function unit types, derived from separating the work flow into stages (see sub-chapter *“2.3 Communication Architecture” on page 42*).

The exact implementation of the ULTRA unit is not covered in this work. A detailed description with a large design space exploration can be found in [94].

Not shown in this figure are the used caches. Caches to hold copies of frequently used contexts, windows and routing strings and a TLB for address translations are integrated in the final architecture. The TLB as the most complex unit is developed in [52], including a sophisticated control unit and an implementation for a *Field Programmable Gate Array (FPGA)* and an *Application Specific Integrated Circuit (ASIC)* technology. The required *Ternary Content Addressable Memory (TCAM)* for the TLB is developed in [95].

4.3.1 Requester Part

The Requester Part is shown in detail in figure 4.27 and contains the requester units and other modules required to inject request packets. The Triggerpage with the CSB is also included in this part. A VCI fetch unit stores the VCIs in a queue. From this queue the VCIs are issued to FUs using a scheduling unit, which keeps track of occupied FUs. The scheduling unit with the help of a scoreboard is also responsible to ensure the optional in-order delivery of packets.

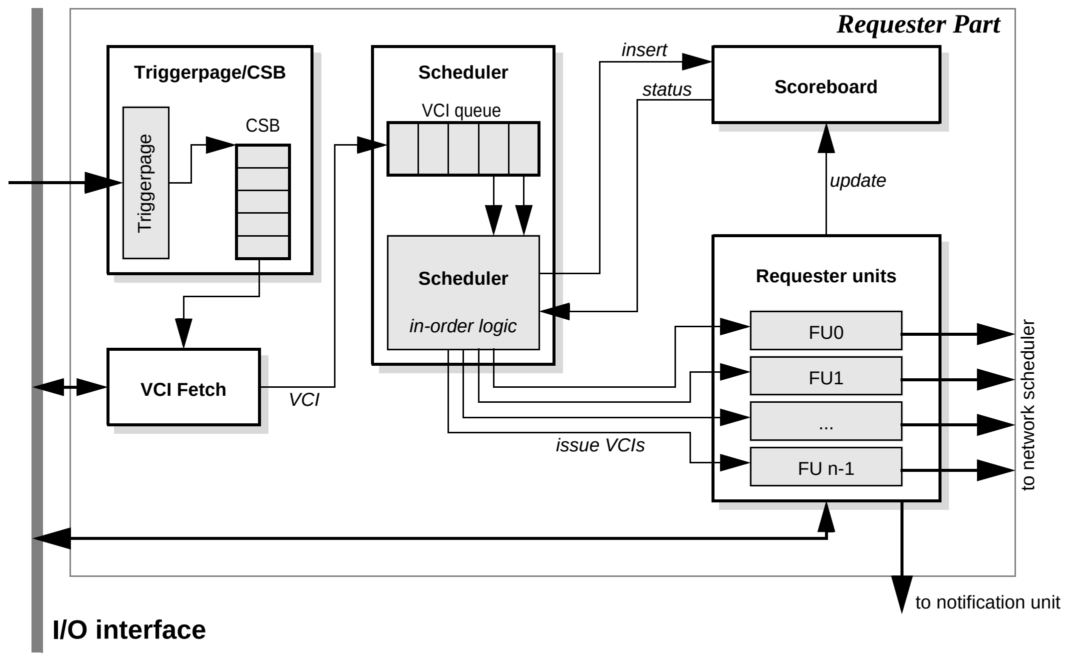


Figure 4.27 Host Port Requester Part

Notifications can be generated when a VCI is completely processed. The notification unit is located in the Completer Part, hence the request to generate a notification is forwarded there.

4.3.2 Completer Part

The Completer Part (see figure 4.28) includes the completer units and a scheduling unit to ensure in-order delivery. If incoming packets are not addressed to a completer unit, the scheduler forwards them to the Responder Part or the ULTRA Completer. The notification unit is also located here. If an incoming packet requests an acknowledge, the acknowledge unit generates an appropriate packet which is sent back to the origin.

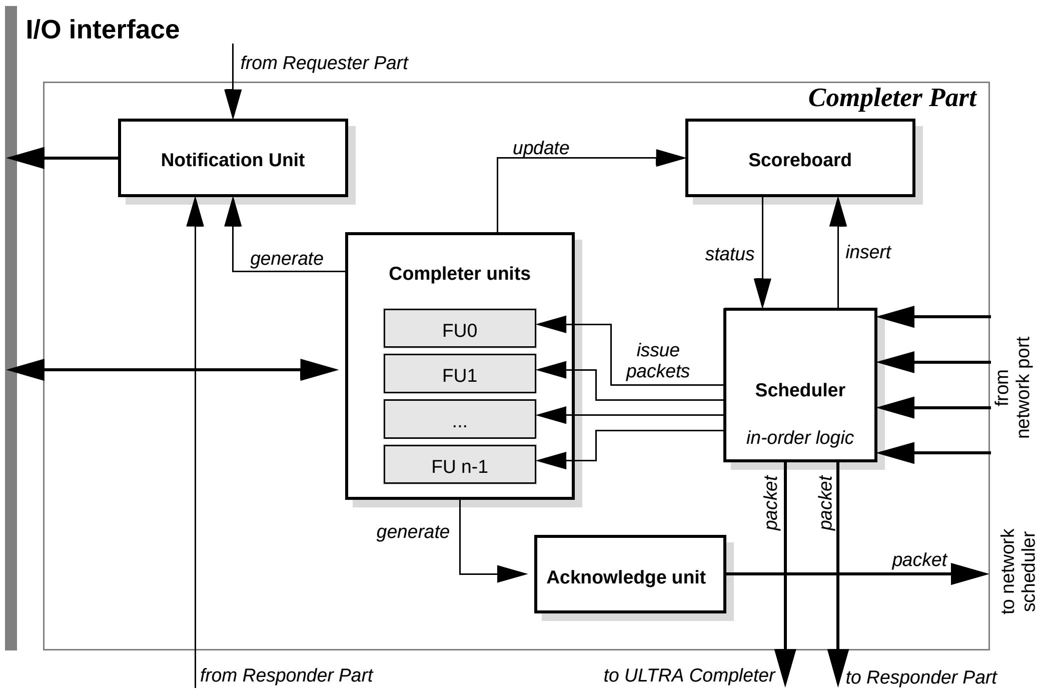


Figure 4.28 Host Port Completer Part

4.3.3 Responder Part

The Responder Part integrates the responder units (see figure 4.29). Packets are incoming from the scheduler located in the Completer Part. In this part the optional in-order delivery is again ensured using a scheduler and a scoreboard.

Synchronizing RMA instructions can optionally include a notification at the target, hence appropriate requests are sent to the notification unit in the Completer Part.

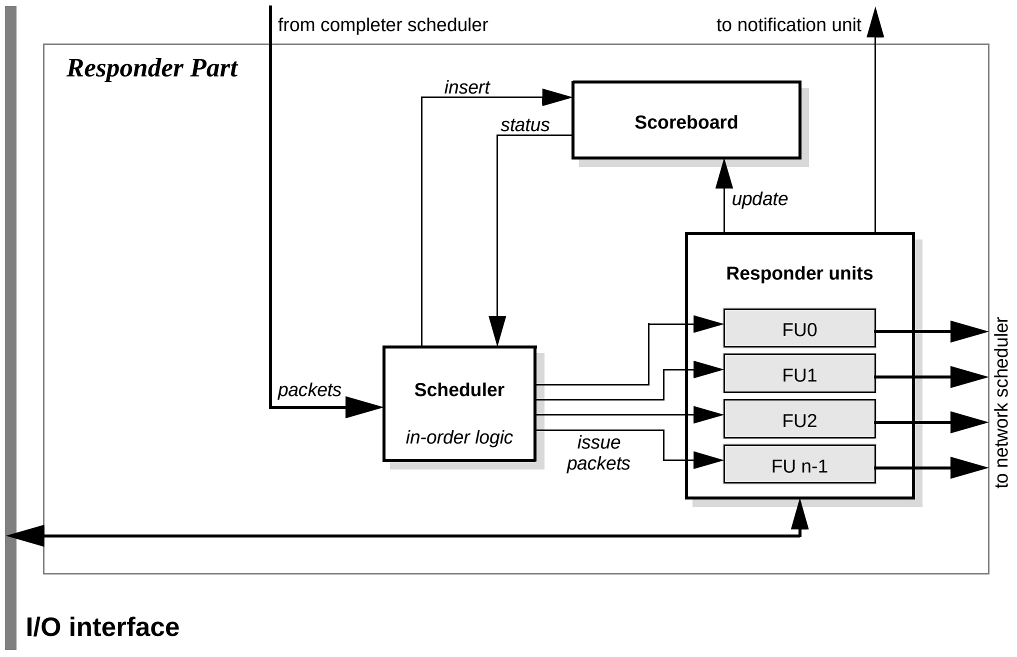


Figure 4.29 Host Port Responder Part

4.4 EVALUATION

Two key components of this work are evaluated, in order to prove that they behave correctly and performant. An evaluation of the complete design developed in this work is much too complex to be covered here.

The first component is the Triggerpage. The test will show that the Triggerpage allows simultaneous access from multiple user-level processes to the device and prove the correct functionality. A performance evaluation shows that the latency of a Triggerpage access is dominated by the included PCI read access, while the latency part caused by the Triggerpage is minimal.

The second component to be evaluated is the ULTRA unit. A performance evaluation is provided with first real world measurements. The achieved latency is compared to commercial available interconnects, which shows ULTRA's excellent performance regarding latency.

The environment used for evaluation is FPGA-based. The FPGA device is a Xilinx Virtex2 and connected to the host using a PCI interface. The PCI interface is capable of 33 and 66 MHz as operating speed. The host runs under Linux 2.6 and houses two Intel Xeon CPUs. This prototyping station soon showed it's limitations and the decision was made to develop are more suitable one, aggregating a high-performance FPGA with a HyperTransport connection to the system.

This HTX-Board is introduced in the next sub-chapter. Then follows the evaluation of the Triggerpage, which is done using the PCI-based FPGA board. After this the ULTRA unit is evaluated. The implementation of ULTRA on the HTX-Board is currently in progress. Because of this the implementation is not part of the work presented here, only a short outlook of the performance achieved is provided.

4.4.1 HTX-Board

The *HyperTransport (HT)* technology [96][97][98] is a point-to-point link interconnect designed for chip-to-chip or board-to-board communication. This technology provides high bandwidth together with very low latencies, making this technology suitable for almost any application from embedded systems over PCs to high-performance computing systems. Nearly the complete CPU portfolio of AMD already integrates at least one HT link [99][100]. One of the most important recent developments in the HT context is the introduction of the HTX connector [101][102]. In a very short time period the HTX technology was accepted by industry and lead to the launch of new main boards from different vendors. The first add-in card designed for HTX is InfiniPath by PathScale [35], meantime several others are available. Still missing is a rapid prototyping station, which is mandatory for fast developments of HTX devices. The HTX-Board presented here is exactly designed for this purpose. It's main component is a high-performance FPGA, which is closely coupled to the main CPU over the HTX connector. The HyperTransport

Consortium [103] already included the HTX-Board as a Reference Design in their portfolio, which shows the demand for such a HyperTransport-based prototyping engine.

The HTX-Board is an unique design, integrating an FPGA suitable for rapid prototyping and an HT connection to the system. The HT connection replaces the traditional I/O interface for peripheral devices (for instance PCI or PCI-Express). Now the device is directly connected to the system interface and intermediate bridges for protocol conversion are no longer required. This close coupling result in very low latencies for accesses from CPU to device.

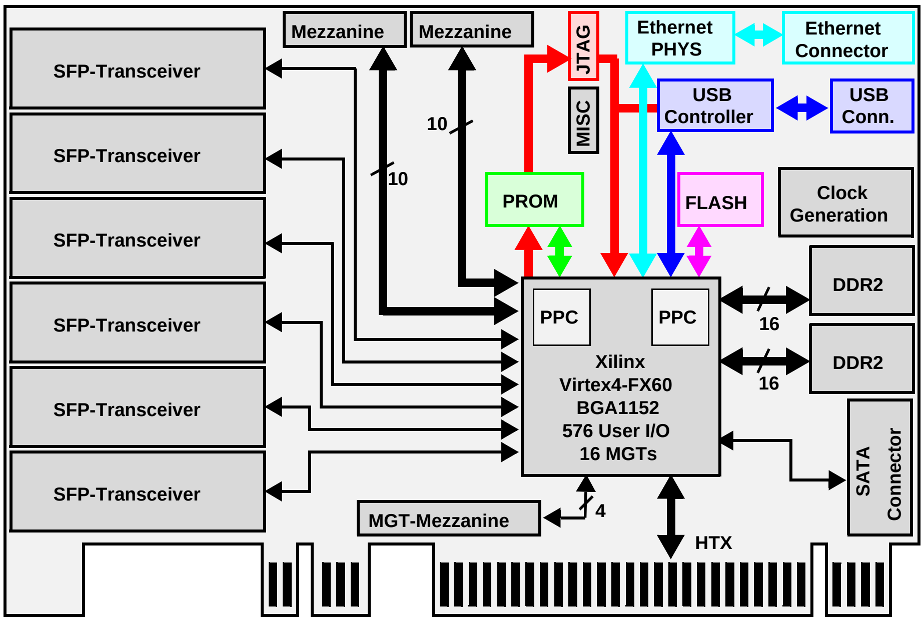


Figure 4.30 Block Diagram of the HTX-Board

The other key component of the HTX-Board is the FPGA, which is connected to the various components like communication devices, dynamic and flash memory or for auxiliary functions. A Xilinx Virtex4-FX [104][105] was chosen because it already contains CPU cores and a large number of high speed serial transceivers. Other features like support for dynamic reconfiguration, differential I/O with up to 1GHz or the embedded Ethernet MAC cores are not mandatory, but fit very well in the architecture of the design and are not left unused.

The FPGA is directly connected to the HTX interface with differential links. This connection is 16bit wide in each direction. Wider HT connection are not supported over HTX connectors. The complete power supply is also provided by the HTX connector, no external power is required.

Beside the HTX interface, the most important feature are the Small Form Factor Pluggable (SFP) Transceivers. Six of these SFPs are placed on the board, connected to the high speed serial links of the FPGA. These links can run speeds from 622 Mbit/s up to 6.25 GBit/s. One advantage of SFP is that the transceivers are pluggable. By exchanging the transceivers every kind of transmission is possible, electrical or optical over various connector types. Running all SFP transceivers at full speed, the bidirectional bandwidth is 60 GBit/s[1]. All six transceivers are accessible at the front panel of the board. The intend is to build up direct interconnection networks with a 3D-topology, for instance tori or meshes (or any other topology with a node degree [106] of not more than six).

The embedded CPU cores of the FPGA use a data width of 32bit, which is sufficient for the targeted applications. For unconstrained usage of the CPU cores, additional memory is required on the device. Furthermore, flash memory and an Ethernet interface is useful for loading the bootstrap of the CPU cores. Thus the FPGA is connected to auxiliary components like a DDR2-SDRAM, a flash memory and an Ethernet device. Regarding the data width of the DDR2-SDRAM interface it is optimal to match the data width of the CPU core, which is 32bit. Because the highest data width of available DDR2 devices is 16bit, two of them are placed on the board to match the data width.

The top-level block diagram is shown in figure 4.30 with all important components, mainly the FPGA, the HTX connector and the SFP array. These components are already shortly introduced. A deeper explanation can be found in [107].

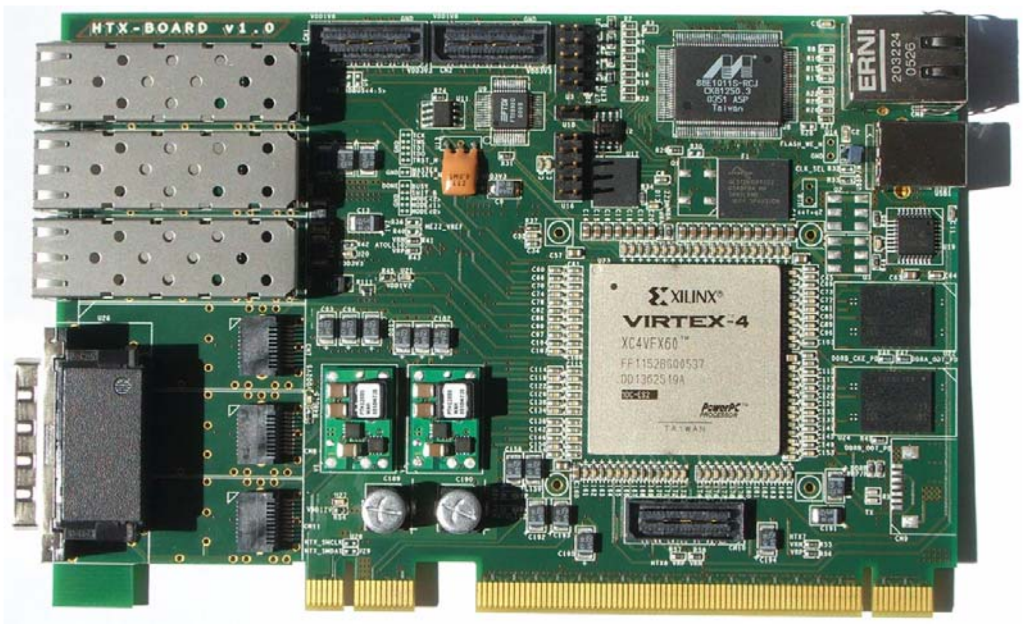


Figure 4.31 Photo of the HTX-Board

In figure 4.31 a photo of the HTX-Board is shown. The FPGA and the HTX interface can easily be recognized.

The HTX-Board in combination with the HT-Core [108] shows the impact of the close coupling to the system and the absence of intermediate bridges. First measurements results

1. Assuming an 8b/10b code and all six transceivers running at 6.25GBit/s.

are presented in [109]. The latency of a read access starts with 320 ns for a 4 byte transfer. In this configuration the HT-Core is only running with 100 MHz and a data width of 8 bit, hence there this latency can still be optimized in the future. Currently a 16 bit wide HT-Core running with 200 MHz is under development, which will further reduce the latency significantly. An ASIC implementation of the HT-Core can achieve even higher operation frequencies.

4.4.2 Triggerpage evaluation

The Triggerpage is a new method to efficiently issue new work requests to a virtualized device. It’s use is not limited to network interface. Any high-performance device that is capable of User-Level Communication and virtualized in hardware can benefit from the Triggerpage.

The only hard limit for the number of processes simultaneously accessing the device is the available address space for an device. The tests here use a PCI peripheral device, which address space is limited by the PCI specification [19] to 256MB or 2^{16} 4kByte pages. Hence up to 2^{16} processes can concurrently open and access the device.

In order to test and evaluate the Triggerpage it is implemented using an FPGA technology [94]. Because of the limited resources in the FPGA the Triggerpage could only be implemented with a PCI interface running at 33MHz (PCI33).

In the first test one user process performs 1000 operations on the Triggerpage. The measurement shows that one single operation is performed in 630ns (see *Table 4.5*). This value is compared to the latency of a normal read of an on-device register which is 630ns for PCI33. This comparison obviously shows that the latency of a Triggerpage operation is completely dominated by the included PCI read access.

Table 4.5: Triggerpage Performance Evaluation

number of simultaneously accessing threads	number of operations per thread	average latency per operation
1	1000	630 ns
8	8	107 us
64	8	1513 us

The next test creates multiple threads which all access the Triggerpage. The number of threads and the number of operations per thread are varied (see *Table 4.5*). The measurements show the average latency of 1000 iterations.

Both cases performed successfully and showed that the Triggerpage allows a large number of processes to simultaneously access the device. But the considerable latency increase deserves a closer evaluation.

The non-linear increase of latency per operation in *Table 4.5* is most likely caused by the thread management system which is part of the O/S. The Triggerpage cannot influence this. This is also proved by the first measurement, where the latency increases linearly (with a factor of one) with the number of operations performed.

To conclude the test of the Triggerpage, it is shown that the Triggerpage allows a large number of user processes to simultaneously access the device. The operations are performed and interpreted correctly. Regarding the performance evaluation, a host system capable of more simultaneously running processes is required together with a more performant I/O subsystem. Additionally an FPGA with more resources is needed for an in-depth evaluation. Only then possible performance limitations of the Triggerpage can be unveiled.

4.4.3 ULTRA evaluation

The other evaluation performed within the scope of this work is a performance analysis of the ULTRA unit. The goal of ULTRA is to allow communication between user-level processes with lowest latencies possible. In order to achieve this goal, the message injection and retrieval scheme requires only a minimum number of I/O cycles. This is made possible by pre-configuration of the requester and completer units. Then the static part of a message is already stored in the device. Only the message tag and payload must be transferred using a single burst access on each side.

For the evaluation basic PCI-66 measurements are combined with simulation results. The resulting end-to-end latency between two nodes (including the crossbar and cabling) is only 1198ns. In [94] a break-down of this latency is provided, showing it's components in detail. In summary, the major part of the latency origins to the I/O interface. To inject the message one PCI burst write is required, which costs 471 ns. The message is received using a queue located in main memory and the required access from device costs 260 ns. The latency of the ULTRA requester unit is 13 clock cycles, the ULTRA completer requires 21 clock cycles. Applying a clock speed of 100 MHz (which is more than feasible for an FPGA technology) the resulting accumulated latency is 250ns. Applying a typical operating frequency for an ASIC technology like 500MHz, the latency of the two units is reduced to 50 ns.

Only few cluster interconnects achieve such low latencies. The following interconnects have been bench marked during the tutorial of the International Supercomputer Conference (ISC) 2005. Only commercial products are included here. For comparison, the start-up latency of Gigabit Ethernet is 37.45 us. Myrinet reaches 2.78 us (based on PCI-X), Mellanox's Infiniband based on PCI-Express 2.76 us and Quadric's QsNet2 based on PCI-X 1.71 us. A interconnect which is not based on PCI or PCI-Express is InfiniPath by PathScale. It is connected towards the host using the HyperTransport technology. The resulting latency is as low as 1.35 us, and shows again the impact when replacing a traditional I/O interface. These values were measured within the scope of the tutorial and all companies agreed to publish the results [110]. For comparison, ATOLL achieves a latency of 3.4 us [39].

This analysis together with the results of the tutorial show the possible impact when replacing the PCI(-Express) I/O interface with a high performant one. An implementation on the HTX-Board connected over HyperTransport to the host is currently in progress. First basic measurements show that the performance increase is tremendous, resulting in latencies around 750 ns [109] for a packet transfer.

Chapter 5 Instruction Set Extension for Coprocessing

In the previous chapters several new techniques have been developed, specified and evaluated to improve the overall performance of network interfaces. In particular the virtualization is not limited to be used only for network interfaces. It is a generic approach which allows to virtualize almost any high-performance device. The ULTRA architecture targets lowest latency packet transfers and is currently limited by the I/O interface.

Both the virtualization and the ULTRA architecture can be improved by a closer coupling between CPU and device. The virtualization is based on a large set of pages for process identification and a read operation for triggering. Both methods are work arounds to overcome the limitations of I/O interfaces. This also applies for the sophisticated queue synchronization schemes used for ULTRA, where the synchronization overhead is reduced as much as possible.

A modified protocol between CPU and device can significantly improve the performance of the two applications above. This results in not only modifying the CPU, additionally all involved intermediate components and protocols require changes. A direct connection between CPU and device limits this number of changes and allows a close coupling.

One of the most recent developments is a resurgence of interest in coprocessing. This is substantiated by the AMD’s Torrenza project and the use of *Graphic Processing Units (GPUs)* for acceleration. This development started when both virtualization and ULTRA methods were already developed and evaluated. Because especially the virtualization can be used also for coprocessors, the work here is extended in order to show how an instruction set extension can improve coprocessing.

A close coupling between coprocessor and CPU is required for a most efficient acceleration. This can be achieved by special coprocessing instructions. Dedicated coprocessing instructions allow to most efficiently use the limiting interconnect. Using them the typical load and store operations can be extended for a more sophisticated use. This instruction set extension is comparable to many previous extensions, for instance the MMX instructions. Special coprocessing instructions are also in line with the working principle of modern *Complex Instruction Set Computers (CISC)*.

The goal is to limit the required changes for the CPU to the instruction set. Modifications of the architecture, for instance the register file, functional units or internal organization are not desired because then a re-design of the hardware is required. An instruction set extension should be possible by only modifying the micro code of the CPU.

5.1 Introduction to Coprocessing

In [111] a definition of coprocessing is provided, stating that *coprocessors cannot be used alone and they cannot handle regular instructions or I/O operations*. While this is in particular true, some deeper explanations might provide more insights. The central unit of a computing system is always the CPU. A *Coprocessor (COP)* is always dependent on work issued by the CPU.

The goal of coprocessing is the efficient acceleration of specific tasks. A CPU, which is designed for unconstrained and general purpose use, is capable to process almost every task. A COP is a specialized and optimized resources for a specific task, and this task can be processed by a COP much more efficiently and faster than by a CPU. Hence coprocessing always implies CPU off-loading by delegation work from the CPU to the COP.

Typical example applications for coprocessing include (but are not limited to) storage and network, media acceleration (e.g. GPUs), security processing, scientific computing or financial data processing.

Coprocessing can be distinguished in *Traditional Coprocessing* and *Modern Coprocessing*. Both kinds are now introduced and their main difference is shown.

5.1.1 Traditional coprocessing

In the traditional coprocessing the COP is directly connected to the CPU over a dedicated and specialized interface (see figure 5.1). Beside the exclusive connection to the CPU, it can directly access the main memory to fetch and store data.

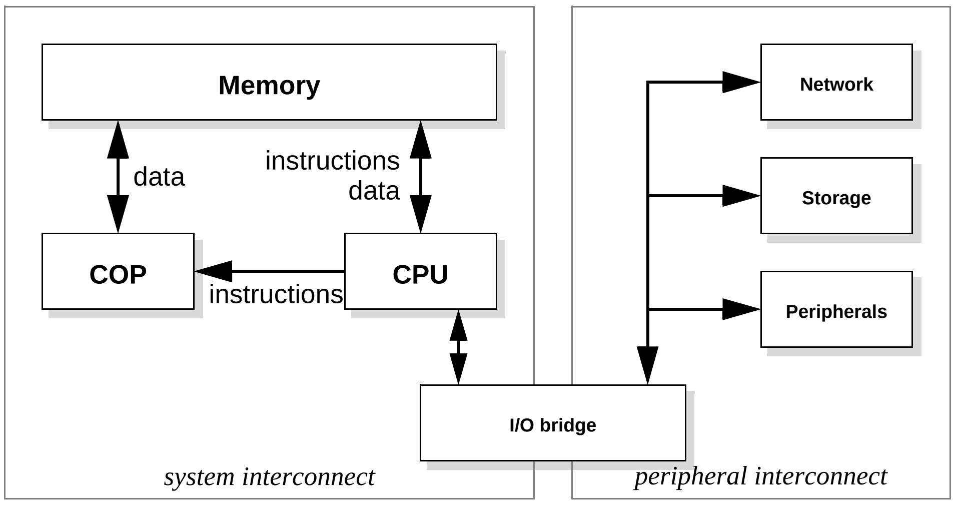


Figure 5.1 Traditional coprocessing

Special coprocessing instructions are available in the CPU’s instruction set. If a COP is present in the system they are forwarded to the COP. If no COP is present an exception is raised. If possible the CPU performs the task by emulation which is obviously much slower. If this is not possible, the application is terminated. These special instructions and the dedicated interface allow a very close coupling.

Examples for such COPs are floating point accelerators, vector processors, digital signal processors and media accelerators. Over the time, the most important COPs were directly integrated into the CPU. For instance, the floating point instructions are again a typical representative.

5.1.2 *Modern coprocessing*

Modern coprocessing differs from traditional coprocessing by the changed location of the COP. The COP is no longer connected over a dedicated and specialized interface to the CPU. Instead a standardized interconnect is used, depending on the location either a standard system or peripheral interconnect. Today a typical location for a COP is the peripheral sub-system (see figure 5.2).

The COP no longer has exclusive access to the main memory and specialized instructions in the CPU’s instruction set are not available. Compared to traditional coprocessing the CPU and the COP are much less coupled.

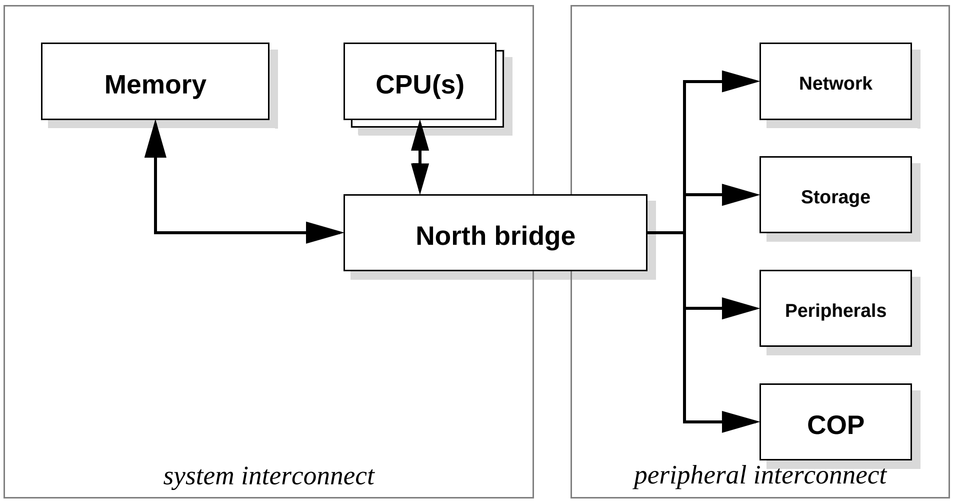


Figure 5.2 Modern coprocessing with peripheral COP

A typical task for a modern COP is to provide hardware acceleration of software algorithms. If this algorithm includes enough parallelism, the hardware is usually more suitable to exploit this parallelism. More specialized modern COPs are for instance network controllers, GPUs or any other sophisticated high performance device which adopts tasks from the CPU.

The most recent trends in modern coprocessing are FPGA- and GPU-based COPs. The ability of reconfiguration respectively programming is used to perform application specific tasks. The COP is once programmed with a certain algorithm, and the CPU can accelerate appropriate work using the COP.

A current trend to standardize the interface between CPU and COP is AMD's Torrenza initiative. The goal of this initiative is to set up a common basis for developing COPs. The development of application specific COPs is then simplified by relying on this common basis.

5.1.3 Locations of modern coprocessors

The most important difference for a modern coprocessor is it's location within the system. Either it is located in the peripheral sub-system (see figure 5.2) and connected over a standardized technology like PCI, PCI-X or PCI-Express to the rest of the system. This approach suffers from the absence of cache coherency, an intermediate bridge which is required for protocol conversion and the resulting high access costs from CPU to COP and from COP to main memory. Nevertheless the standardized interface allows a broad use of the COP.

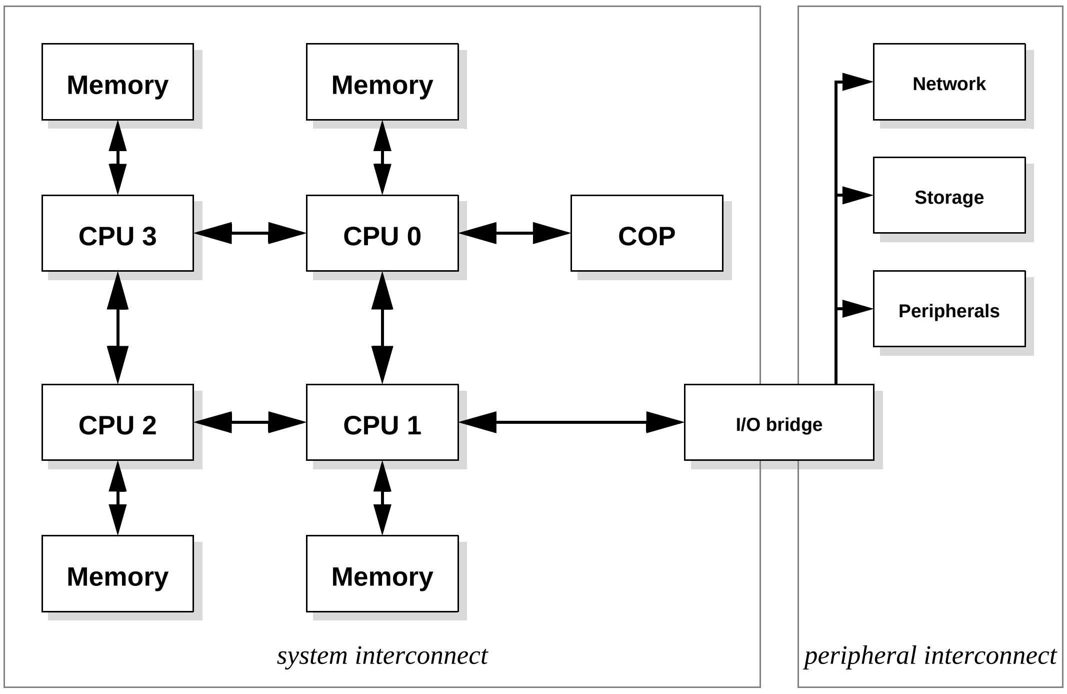


Figure 5.3 Modern Coprocessing with directly connected COP

To overcome the limitations introduced by the intermediate bridge a COP can also be connected directly to the system interconnect (see figure 5.3). Then protocol conversions are no longer required and the access costs are dramatically reduced. A direct connection allows a much closer coupling due to the reduced overhead for accesses. The drawback is that most system interconnect protocols are either not published or the main board provides no connection for a COP. A recent development is the HTX expansion connector, which provides a standardized interface and allows a direct connection to the system interconnect.

The configuration shown in figure 5.3 is not limited to four way CPU systems. For a dual CPU configuration, CPU 2 and 3 can be left away together with their memories. If the CPU 0 as boot master provides enough links for a direct connection of both the COP and the I/O bridge, even a single CPU configuration is possible.

A similar approach is to use a multi-processor system and replace one CPU (except the boot master) with a COP. For the configuration shown in figure 5.3 this even allows to attach memory directly to the COP. The drawback here are non-standardized and frequently changing CPU sockets, or again unpublished system protocols.

5.1.4 Examples of modern coprocessors

Now some typical representatives of modern processors are shown, starting with COPs located in the peripheral sub-system. *ClearSpeed's Accelerator Board "Advance"* [112] is a typical accelerator connected over PCI-X to the main system. The goal is hardware acceleration of software algorithms.

The *RCHTX high performance computing board* from *Celoxica* [113] targets the same application, but this accelerator uses an HTX connector to the system. This is one of the coprocessors which are directly connected to the system, but are designed as expansion card. While ClearSpeed's board is based on ASICs, the accelerator from Celoxica is based on an FPGA by Xilinx, allowing FPGA-based co-acceleration of software algorithms.

Typical examples for COPs replacing a CPU are the *Reconfigurable Processor Unit RPU100* or *RPU110* by *DRC Computing* [114] and the *XD1000 FPGA Coprocessor Module* by *XtremeData* [115]. Both fit in an AMD Opteron compatible CPU socket (Socket 940) and are based on FPGAs. While DRC Computing's unit houses a Xilinx Virtex device, the module by XtremeData uses an Altera Stratix FPGA.

One of the most recent trends is to use GPUs for coprocessing. A modern GPU houses a large amount of computing power which can also be used for application specific acceleration. The first example is the *ATI Stream Computing* (by AMD) which is based on ATI Graphics Cards [116]. The other example is NVidia's *Compute Unified Device Architecture (CUDA)*, which is based on NVidia GPUs with up to 128 cores [117].

5.2 Interface from CPU to Coprocessor

A coprocessor directly connected to the main system interconnect provides the best performance. Additionally the direct connection does not require any intermediate bridges and it is possible for the COP to participate in the cache coherency protocol. Such a location is most suitable for the proposed instruction set extension. Beside the instruction set only the protocol of the system interconnect must be slightly modified. Then the changes can be limited to the main processor’s micro code, a small extension of the protocol and the coprocessor itself.

A location of the coprocessor in the I/O subsystem not only provides less performance, it also results in much more changes. In addition to above a custom I/O bridge and a changed I/O protocol is required. Hence the first solution with a direct connection is targeted for the proposed extension here.

The HT technology is a suitable interface. The HTX-Board introduced in sub-chapter *“4.4.1 HTX-Board” on page 168* is perfectly suited to act as a coprocessor for a first test of the extensions proposed here. It can easily be reconfigured to execute the new instructions.

5.2.1 General requirements

A typical environment for a coprocessor is a multi-processor computing system. Each CPU additionally consists of several cores (see figure 5.4). It is expected that the recent multi-core trend continues which will lead to even more cores per CPU. Other techniques like multi-threaded architectures or VM environments additionally increase the parallelism.

The coprocessor is a highly specialized component of the system and typically only available once. The coprocessor can be virtualized to grant all processes user-level access. The architecture proposed for the device virtualization is also most suitable here. The SMT approach allows to exploit any amount of parallelism without restricting the utilization by partitioning the resources.

Each CPU has an instruction pointer, which leads to an autonomous instruction stream. The CPU as a master issues work requests to the coprocessor (which is the slave). The instruction stream of the coprocessor is dependant on the work issued by the CPUs, hence no instruction pointer is present.

One of the most important requirements when issuing work to a coprocessor, independently if it is for off-loading or acceleration purposes, is a highly efficient interface. Only then the required overhead for *work issue* and *wait for work completion* is low enough to allow a fine grain work issue. A highly efficient interface is also the key component for a scalable design, when the number of processes simultaneously accessing the COP is scaled.

Both work issue and wait for work completion are the only two synchronization points when processing work on a COP. The work issue must be non-blocking to allow independent work flows. Then the CPU can continue processing other tasks, while the COP executes the issued work. The wait can either be blocking if the CPU cannot continue with other tasks, or non-blocking if this is only a check for completion and other tasks are outstanding.

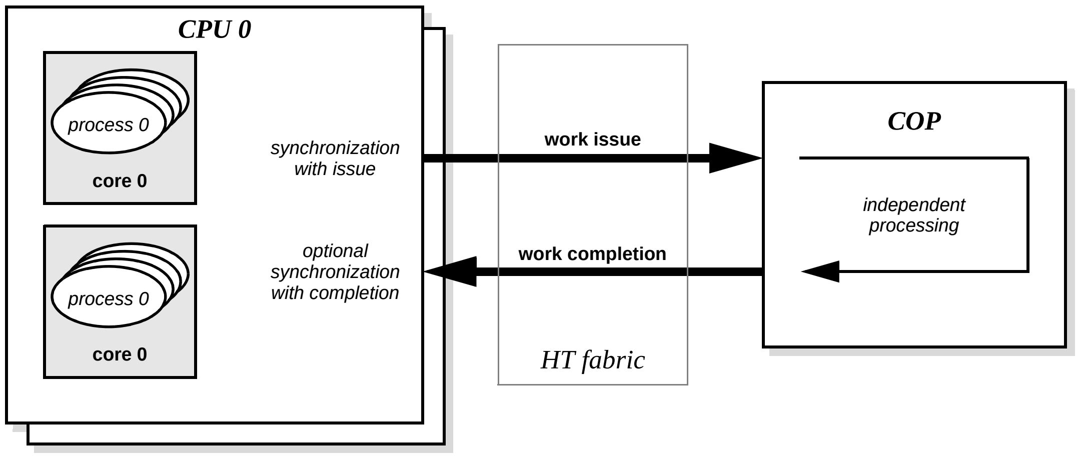


Figure 5.4 Typical environment for a virtualized coprocessor

This results in tight coupling between CPU and COP at the synchronization points. But the independent work flows still allow to decouple the processing, which leads to a high degree of concurrency and does not introduce unnecessary constraints.

5.2.2 Analysis of device virtualization

The device virtualization allows a large number of processes to simultaneously access the device. The identification of processes is secure, preventing accidental or intended access to data structures or resources belonging to other processes. The process identification allows the device to load the corresponding context and hereby configure the appropriate hardware modules for this process.

Device virtualization can also be applied for coprocessors. Comparable to a virtualized device, a virtualized coprocessor is most suitable for multi process environments. In such an environment a large number of processes is simultaneously running and each of them can be granted access to the coprocessor. The SMT architecture proposed for the device virtualization is also suitable for a virtualized coprocessor, allowing it to exploit a large amount of the offered parallelism.

Requirements. Essential for the virtualization is a secure process recognition. This is currently achieved using a set of pages in the I/O space. Because of the shared trigger queue a read operation is used for triggering. The read result includes information about the success of the operation, which is for instance depending on a free queue entry.

The address space of the Triggerpages is configurable but static. For a large number of processes it has a remarkable size. The large number of pages may also lead to TLB misses. The access to a virtualized device is only efficient if the TLB of the CPU contains the appropriate address translation entry. Otherwise the MMU is involved in the translation which results in a large amount of overhead required for a page table walk.

Furthermore the required read operation does not allow to include any immediate data in the issue access. This is currently solved by referring to a separate work queue. For each trigger operation the device has to fetch the instruction from this queue. This memory reference results in additional overhead. If the instruction is included in the issue operation as an immediate value the additional memory access is unnecessary and the overhead reduced.

Proposed extension. The solution to improve the two problems above is a store operation which is tagged with a process identifier. This identifier must be included in a secure way in the access, preventing it to be modified or faked by a user level process. Only then the access is safe and the separation of different processes is guaranteed.

Using the tag the virtualized coprocessor can recognize the calling process. A set of pages in the I/O space is no longer required. The work request previously stored in separate work queues can now be included as payload in the *tagged store*. This saves one memory access which is otherwise required to fetch the work request. The central trigger queue turns into a work queue.

For a shared queue the multiple writer problem remains. Several processes are simultaneously accessing the queue to insert entries. Mutual exclusion is required to ensure correct behavior. The cycle of checking for available space and enqueue of the entry may not be interrupted.

This can be achieved efficiently by a *tagged conditional store*, where the condition is a free queue entry. Only if this is true the new entry is inserted into the queue. The tag is still used to identify the calling process in a secure way to the coprocessor. This instruction must include a response which return value informs the calling process of the success or failure of the instruction.

5.2.3 Analysis of ULTRA

The goal of ULTRA is to allow fine grain communication and synchronization between processes on different nodes. The approach is to initiate a communication by writing to a special address, and to receive the transmitted data by reading a special address. In the previous chapters ULTRA is developed to be part of an I/O device. While this approach does not imposes any restrictions to inject a packet, the packet retrieval is only possible with a status word indicating if the data is already valid. If not, following accesses to the address are required to poll until the data becomes valid.

Requirements. ULTRA can be improved by a more sophisticated receive scheme. If the load operation takes into account if the data at the target address is valid or not, the calling thread can be blocked. Only if the data becomes valid the process can continue to execute on the CPU. This makes polling obsolete and reduces the CPU load.

A multi-threaded architecture even improves this scheme. The blocking is then only thread-blocking, not CPU-blocking. If one thread is blocked due to a missing resource (load to an invalid address), a fast context switch occurs and another run-ready thread is allowed to execute on the CPU. As soon as the data becomes valid, the blocked thread is re-scheduled as run-ready. This thread scheduling allows to hide the receive latency. Because modern CPUs based on HT are typically not multi-threaded and such an architectural change is far beyond the intend of this chapter the following analysis focusses on load operations including a valid identifier.

Proposed extension. The proposed extension is a *tagged load*, comparable to the previous tagged store. Now the response not only contains the data, additionally the tag is used as valid identifier. The process can interpret this tag to know if the read value is valid or not. Dependent on this it either retries the tagged load or continues processing.

For a *Multi-Threaded Architecture (MTA)* with support for fast context switches the tag can also be used to determine if the context has to be switched or not. It is most likely that an invalid load will be retried, which results in polling. The wasted CPU time can be used more efficiently if another thread is run-ready.

The proposed tagged load is used as a kind of wait for completion. Only if the work is finished the data becomes valid, which allows the process to continue it's execution using this data. Similar to ULTRA, the tagged load is also suitable as wait for completion operation for the virtualization, rendering the valid identifier used in the notification descriptors unnecessary.

The tagged load can also be used as issue operation for the virtualization based on a shared trigger queue. A secure tag allows the identification of the process, and the response informs the process of the success or failure of the issue operation. In opposition to the tagged store the work request cannot be included as immediate value. But while the tagged store alone is not suitable for a shared queue, the tagged load is sufficient.

5.3 PROPOSED INSTRUCTION SET EXTENSION

The two applications in the previous sub-chapter show the various possible use for *tagged instructions*. It also leads to the result that support for secure and user accessible tags is required. A generic approach will take both kinds of tags into account.

This sub-chapter examines the proposed new instructions in more detail. Because the focus of this chapter is to extend the instruction set and not to modify the architecture of the CPU, the proposals for architectural modifications (like thread switching upon blocking) are no longer analyzed. The goal is to allow an efficient coprocessor interface using small modifications of the instruction set, preferable those who are possible to implement using micro code patches. Modifications to the coprocessor are not crucial, because they are typically based on custom designs.

To summarize the last sub-chapter, tagged instructions can be used to improve the virtualization and the ULTRA architecture. In particular, support for *tagged store* and *tagged load* instructions is required. A *tagged conditional store* provides most efficient support for the virtualization, including all required accesses to issue new work requests in a single instruction.

5.3.1 Tagged store instructions

The *tagged store* (see figure 5.5) is proposed to overcome the required set of pages in the I/O space required for virtualization. The tag of the store is used to identify the process towards the virtualized coprocessor. This identification must be secure, hence only privileged instances are able to modify it.

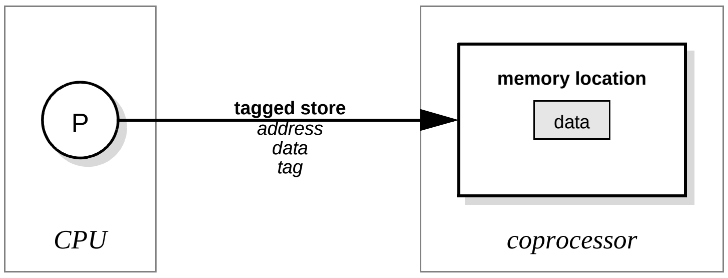


Figure 5.5 Instruction set extension: Tagged Store

This can be achieved as part of a context switch. The O/S sets a special register with the identification of the process. This register cannot be modified by an unprivileged instruction. The value of this register is used as tag for each tagged instruction the user process performs.

A generic approach leads to two kinds of tagged store instructions. One where the tag is protected like proposed above and another one where the user process can modify the tag for each tagged instruction. For the second *user tagged store* any register can be used as tag.

5.3.2 Tagged load instructions

The *tagged load* (see figure 5.6) is used to replace the dedicated valid identifiers stored with data values. This is achieved by tagging the returned value. Dependent on the tag a status bit of the CPU is set. After the tagged load the process executes a conditional branch, in order to repeat the procedure if necessary. This ensures that the execution is only continued if the data is valid.

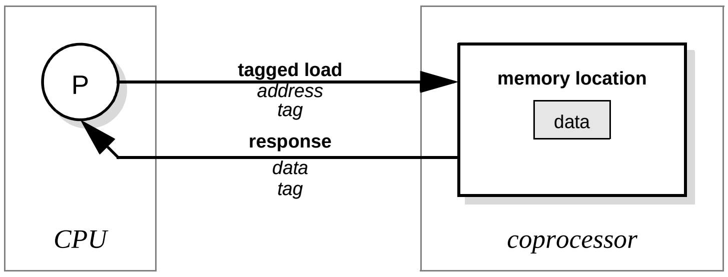


Figure 5.6 Instruction set extension: Tagged Load

The response tag is not accessed directly by a user process. It is set by the coprocessor and the process only uses the tag indirectly over a status bit. Beside the response the request part of a load can also be tagged. Comparable to the tagged store this can be used to allow the coprocessor to identify the calling process. In this case, the tag must be secured from user access, using the same approach as already described for the tagged store. The two tags from request and response are strictly separated and do not directly influence each other.

In the two example applications here (Virtualization and ULTRA), there is no need for a user-accessible request tag. Although it is proposed to include, both for completeness and a most generic approach.

5.3.3 Tagged conditional store instructions

The *tagged conditional store* (see figure 5.7) instruction is the most sophisticated in the proposed extension. The trigger operation for the virtualization can be completely performed using this instruction.

The tagged conditional store differs from the tagged store in two ways. First, the store itself is dependent on a condition. Only if the condition is met, the store is executed.

Furthermore the conditional store includes a response to the process. This is used to inform the process if the condition is met or not, or in other terms if the store was executed or not.

The condition check and the store may not be interrupted by other stores. This mutual exclusion is required for all shared data structures, where multiple writers access a shared location. Otherwise correct behavior cannot be ensured.

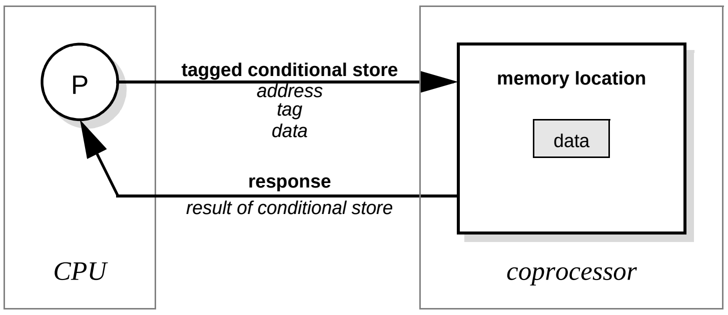


Figure 5.7 Instruction set extension: Tagged conditional store

Comparable to the tagged store, the instruction here is again tagged in a secure way to allow process identification. Beside the address and the data to be stored it furthermore includes a condition. This condition can be of almost any type, for instance a check if a bit is set or not.

Chapter 6 Conclusion and Outlook

The major goal of this work is to improve the architecture of network interfaces. The motivation is manifold.

The computer architecture and in particular the processor architecture is steadily improved and the parallelism offered by the host system increases. At the moment of writing, mobile computing with dual-core processors is pervasive. Quad-core server processors are announced by the leading manufacturers or already available. A current processor design combining multi-core with multi-threading techniques allows to run up to 32 threads in parallel. Most recent research projects show processors with up to 80 cores, indicating the multi-core trend will continue on. Another recent development is a resurgence of interest in Virtual Machine environments, which in addition increase the parallelism of the system by hosting multiple O/S on one physical machine.

If such a system is a node of a parallel distributed computing system, the performance of communication and synchronization is essential. Only if the amount of overhead for communication and synchronization is minimal, a close coupling of the nodes is possible. A close coupling is also inevitable for fine grain communication. In addition, a close coupling is the key for a scalable system, with a performance increase analog to the size of the parallel system.

Communication and synchronization functionality is provided by the network interface. While a computing node exploits an increasing amount of parallelism, a network interface is typically only available once in a node. In a distributed system the exploited parallelism is offered as communication and synchronization work to the network interface. With the increasing parallelism of the computing node, *parallel, unconstrained and simultaneous access to the network interface* becomes inevitable.

Virtualization. Support for simultaneous access is achieved by device virtualization using the *Triggerpage*, which is the one of the key contributions developed in this work. It allows unconstrained access from almost any number of processes[1] to the device using *User-Level Communication*. In particular the I/O interface is used very efficiently, which otherwise turns into a limiting bottleneck.

The Triggerpage with it's *issue operation* is the interface for all accesses from user process to the network interface. By a single load instruction (seen by the device as a read cycle) any user process can issue new work requests to the device. This highly efficient issue operation uses addresses as immediate values, and in the other direction the read response for flow control. With this issue operation, central shared queues can be accessed simultaneously by any number of user processes. This innovation is the enabling key of unrestricted simultaneous access to devices and it's high efficiency is inevitable for a *scalable virtualization*.

The Triggerpage also ensures *security* and *separation* among the accessing processes. User processes are considered to be insecure, but no process is able to interfere with other ones, preventing both failures due to erroneous or harmful software. Furthermore each accessing process sees an exclusive device for it's own. The complete virtualization technique is completely transparent to a process, increasing the safety of the design.

The virtualization relies on a set of data structures in main memory. No additional on-device memory is needed, reducing the overall costs of the device. Furthermore the main memory as most cost-efficient memory resource is most suitable when the number of processes is scaled. This data structure set is also one of the enabling techniques for scalable virtualization.

The complete virtualization does not require any modification of the I/O interface. This allows to use this technique with any available I/O interface. In particular *PCI, PCI-X, PCI-Express and HyperTransport* are analyzed, but any modern I/O system should be suitable.

The Triggerpage with the issue operation as key component of the virtualization are implemented and evaluated. The results substantiate the expected efficiency and performance, showing that an instruction issue over the Triggerpage does not last longer than a normal read cycle on the I/O interface.

Architecture. The virtualization above is only a part of the network interface, and offers a large amount of parallelism to the execution units. For maximum performance the network interface architecture must be able to exploit as much of the offered parallelism as possible. The virtualization allows a large number of processes to issue work simultaneously, hence the offered parallelism includes beside instruction level parallelism also thread level parallelism. A suitable architecture for an unconstraint exploitation of both types of parallelism is developed, based on the insights gained by an analysis of modern processor

1. The specification provided in chapter four allows up to 2^{16} processes, expecting this large number to be sufficient for almost all applications. The virtualization method itself is not restricting this number, currently only the PCI specification limits it to 2^{16}.

architectures. It has many similarities with *Simultaneous Multi-Threading*, removing any partitioning of the superscalar functional units.

The *absence of partitioning* not only exploits any parallelism without restrictions, it also allows a dynamic use of the available resources, which leads to a high utilization of the superscalar functional units. In particular all resources can be occupied with work of a single process. In the case of several competing processes the resources are shared equally.

To fulfill the special requirements of network interfaces, the architecture includes support for in-order work processing. This is achieved by supervising the superscalar functional units with a scoreboard, ensuring that work requests with dependencies are not processed in parallel. A potential increase of issue queue blocking is solved by a multi-issue technique in combination with distributed wait queues. These wait queues are derived from reservation stations, which are well-known in modern processor architecture.

The device virtualization and the architecture are not limited to network interfaces. Any high performance device can benefit a lot from these techniques.

Communication and synchronization. The network interface architecture is combined with a set of communication and synchronization instructions. This set is composed by several sophisticated communication methods, each optimized for a certain range of message sizes. Goal is to minimize the communication latency by reducing the overhead. Dependant on the message size, the most suitable communication method is chosen. This manifold set is comparable to the instruction set of a modern CISC architecture.

The set includes communication methods based on Send/Receive and RMA schemes. For both, several different instructions are available. The architecture includes an TLB to optimally support RMA with virtual addresses. Furthermore well-known RMA operations are extended by including support for synchronization, which otherwise is not possible for one-sided communication. Memory windows are used to protect the user address space from unwanted access, and to ensure possibly required mutual exclusion.

All the communication and synchronization instructions in this set are issued to the network interface using the Triggerpage, allowing an unrestricted use.

ULTRA. The costs of virtualization are minimal but not negligible. To provide support for *fine grain communication* an additional communication method called ULTRA is developed. The goal of ULTRA is to achieve lowest latency communication, which is the key for fine grain communication. Efficient transfer of small data structures is only possible with minimal overhead, which in particular requires a low latency.

ULTRA is based on a highly efficient packet injection and retrieval scheme. A pre-initialization and pre-completion of packets is used to minimize the overhead for each packet. One single write cycle is sufficient to inject a packet into the network. Similar to this, the packet retrieval is also based on a single I/O cycle.

The major part of the overall latency is introduced by the I/O interface. The efficient injection and retrieval scheme minimizes the corresponding latency component, resulting in an unmatched latency[1]. With a first prototype a latency below one microsecond is achieved.

The restrictions of this prototype still leave room for improvements, thus even lower latencies are possible with ULTRA.

Coprocessing. The recent interest in coprocessing lead to an analysis if the virtualization technique is also applicable for coprocessors. It can be used without constraints, but the analysis pointed out a method to even improve the coprocessor virtualization.

In particular the resources required by the Triggerpage can be reduced by the introduction of an instruction set extension. This extension includes *tagged load and store* instructions and in particular a *tagged conditional store*. The last one allows to reduce the requirements regarding address space of the Triggerpage to the minimum. Unconstrained simultaneous access to the coprocessor is still possible. These tagged instructions are *tightly coupling* the coprocessor to the main processor, and especially fine grain communication benefits also a lot from this.

Summary. The methods and techniques developed in this work tried to fulfill several conflictive goals, mainly support for simultaneous access and fine grain communication.

The first one lead to the *virtualization* technique in combination with an architecture suitable to exploit the available parallelism without restrictions. The *key component* for the scalable and efficient virtualization is a trigger operation, also referred to as issue operation or as doorbell. It is issued by the CPU as a single load instruction and seen by the device as a read cycle, combining *security, atomicity and flow control*. This new and unique method is the main contribution of this work to the research community.

The goal to support *fine grain communication* lead to the development of *ULTRA*, a message passing method with minimal overhead. These two developments in combination achieve the given goals. The instruction set extension to improve coprocessing takes into account the insights gained during these developments. It is proposed to overcome the existing limitations, and both developments can be even improved using these specialized instructions.

Outlook. The network interface architecture including virtualization will be used in the future in a next generation system area interconnect, which is a research project of the Computer Architecture Group at the University of Mannheim. Furthermore the virtualization will be implemented on the HTX-Board to test and verify it in an HyperTransport environment. The virtualization should benefit a lot from the close coupling to main processor and memory. Beside this, the development and implementation of the on-device TLB is pushed, together with the other caching structures. Similar developments of I/O MMUs and I/O TLBs from industry show the relevance of this topic.

A modified successor of ULTRA is used in a collaboration with Sun Microsystems. First results are very promising, but the work is not finished yet. Beside this, an analysis of

1. This is based on a preliminary measurement, which is compared to public available measurement results of other interconnects based on standardized I/O interfaces. Interconnects based on specialized I/O interfaces can easily achieve lower latencies, but their use is significantly restricted.

the impact of ULTRA in combination with the proposed instruction set extension for coprocessing is also planned.

In particular this instruction set extension will be further analyzed and developed. The HTX-Board is perfectly suited as a prototype coprocessor for this case. A close collaboration between AMD and the Computer Architecture Group already exists, and a first discussion showed significant interest. The plans for the future sound very promising and hopefully the proposal here will be used to improve coprocessing.

APPENDIX A

Bibliography

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

[2] O. Lysne, S.-A. Reinemo, T. Skeie, Å. G. Solheim, T. Sødring, L. P. Huse, B. D. Johnsen. *The Interconnection Network - Architectural Challenges for Utility Computing Data Centers*, Research Report 2007-02, Simula Research Laboratory, Olso, Norway.

[3] D. Sima, T. Fountain, P. Kacsuk. *Advanced Computer Achitectures: A Design Space Approach*, Addison Wesley, 1997.

[4] Sven Stork. *Design of an efficient Software Environment for a RDMA Network Interface Controller*. Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2006.

[5] V. Karamcheti, A. A. Chien. Software Overhead in Messaging Layers: Where does the time go?. In *Proceedings of ASPLOS-IV*, San Jose, California, U.S., 1994.

[6] W. Gropp, E. Lusk, A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.

[7] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Available from http://www.mpi-forum.org/docs, 1995.

[8] W. Gropp, E. Lusk, R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*, MIT Press, 1999.

[9] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. Available from http://www.mpi-forum.org/docs, 1997.

[10] W. Gropp, E. Lusk. *User's Guide for mpich, a Portable Implementation of MPI*. Argonne National Laboratory, 1994.

[11] E. Gabriel et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings of Euro PVM/MPI 2004*, Budapest, Hungary, 2004.

[12] 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. In *IEEE Micro, 17(5):12-19, 1997*.

[13] R. Kalla, B. Sinharoy, J. Tendler. Simultaneous Multithreading Implementation in POWER5-IBM's Next Generation POWER Microprocessor. In *Proceedings of Hot Chips 15*, August 2003.

[14] D. T. Marr et al. Hyper-Threading Technology Architecture and Microarchitecture. In *Intel Technology Journal, 6(1):4-15*, February 2002.

[15] D. Koufaty, D. T. Marr. Hyper-threading Technology in the Netburst Microarchitecture. In *IEEE Micro, 23(2):56-65*, 2003.

[16] R.P. Goldberg, Survey of Virtual Machine Research. In *Computer*, pp. 34-45, June 1974.

[17] J. Duato, S. Yalamanchili, L. Ni. *Interconnection Networks*, Revised Edition, Morgan Kaufmann, 2002.

[18] W.J. Dally, B. Towles. *Principles and Practices of Interconnection Networks*, Morgan Kaufman, San Fransisco, U.S. 2004.

[19] E. Solari, G. Solari, W. Solari. *PCI & PCI-X Hardware and Software: Architecture and Design*, 5th Edition, Annabooks, San Diego, U.S., 2001.

[20] PCI-SIG. *PCI Express base specification 1.0a*, 2002.

[21] D. Anderson, R. Budruk, T. Shanley. *PCI Express System Architecture*, 1st Edition, Addison-Wesley Professional, 2003.

[22] P. Sassone. *Commercial trends in off-chip communication*. Technical Report, Georgia Institute of Technology, May 2003.

[23] C. Whitby-Strevens. The Transputer. In *Proceedings of the 12th Annual international Symposium on Computer Architecture (ISCA85)*, Boston, Massachusetts, U.S., June 1985.

[24] T. Gross, D. R. O'Hallaron. *IWarp. Anatomy of a Parallel Computing System*, MIT Press, Cambridge, Massachusetts, U.S., 1998.

[25] F. Allen et al. Blue Gene: a vision for protein science using a petaflop supercomputer. In *IBM Systems Journal 40:2(310-327)*, February 2001.

[26] A. Gara et al. Overview of the Blue Gene/L system architecture. In *IBM Journal of Research and Development 49:2/3*, 2005.

[27] N. R. Adiga et al. Blue Gene/L torus interconnection network. In *IBM Journal of Research and Development 49:2/3*, 2005.

[28] R. Brightwell, D. Doerfler, K. D. Underwood. A preliminary analysis of the InfiniPath and XD1 network interfaces. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS06)*, 2006.

[29] Quadrics Ltd., http://www.quadrics.com.

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

[31] Infiniband Trade Association. *InfiniBand Architecture Specification Release 1.2.* October 2004.

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

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

[34] L. Dickman, *An Introduction to Pathscale InfiniPath.* Available from http://www.pathscale.com.

[35] L. Dickman, G. Lindahl, D. Olson, J. Rubin, J. Broughton. PathScale InfiniPath: A First Look. *In Proceedings of the 13th Symposium on High Performance Interconnects (HOTI)*, 2005.

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

[37] Myricom, Inc., http://www.myri.com.

[38] U. Brüning, L. Schaelicke. ATOLL: A High- Performance Communication Device for Parallel Systems. In *Proceedings of 1997 Conference on Advances in Parallel and Distributed Computing*, Shanghai, China, March 1997.

[39] H. Fröning, M. Nüssle, D. Slogsnat, P. R. Haspel, U. Brüning. Performance Evaluation of the ATOLL Interconnect. In *IASTED Conference, Parallel and Distributed Computing and Networks (PDCN)*, Innsbruck, Austria, February 2005.

[40] N. Tanabe, J. Yamamoto, H. Nishi, T. Kudoh, Y. Hamada, H. Nakajo, H. Amano. Low Latency High Bandwidth Message Transfer Mechanisms for a Network Interface Plugged into a Memory Slot. In *Proceedings of Cluster Computing 5(1):7-17,* Jan. 2002.

[41] U. Brüning, W. Giloi. Future Building Blocks for Parallel Architectures. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP04)*, Montreal, Canada, 2004.

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

[43] Mondrian Nüssle. *Design and Implementation of a distributed management system for the ATOLL high-performance network.* Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2003.

[44] C. Bell, D. Bonachea. A New DMA Registration Strategy for Pinning-Based High Performance Networks. In *Proceedings of 17th IEEE International Parallel and Distributed Processing Symposium (IPDPS03)*, 2003.

[45] S. Sur, H. Jin, L. Chai, D. K. Panda. RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2006.

[46] J.L. Hennessy, D.A. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufmann, San Francisco, U.S. 1996.

[47] J. C. Martinez, J. Flich, A. Robles, P. Lopez, J. Duato, M. Koibuchi. In-Order Packet Delivery in Interconnection Networks using Adaptive Routing. In *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS05)*, 2005.

[48] Dirk Franger. *A multi-context Engine for Remote Memory Access to improve System Area Networking.* Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2004.

[49] Thomas Schlichter. *Exploration of hard- and software requirements for one-sided, zero copy user level communication and its implementation.* Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2003.

[50] Advanced Micro Devices, Inc. *AMD I/O Virtualization Technology (IOMMU) Specification.* Revision 1.00, publication #34434, February 2006.

[51] D. Mosberger, S. Eranian. *IA-64 Linux Kernel: Design and Implementation*, Prentice Hall, 2002.

[52] Felix Rembor. *Exploration, Development and Implementation of different TLB Functions and Mechanism*. Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2006.

[53] T. Agerwala, S. Chatterjee. Computer Architecture: Challenges and Opportunities for the Next Decade. In *IEEE Micro, 25(3):58-69*, 2005.

[54] M. J. Flynn, P. Hung. Microprocessor Design Issues: Thoughts on the Road Ahead. In *IEEE Micro, 25(3):16-31*, 2005.

[55] M. Kistler, M. Perrone, F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. In *IEEE Micro, 26(3):10-23*, 2006.

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

[57] A. Whitaker, M. Shaw, S. Gribble. *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*. Technical Report 02-02-01, University of Washington, U.S., 2002.

[58] R. Uhig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kägi, F. Leung, L. Smith. Intel Virtualization Technology. In *IEEE Computer, 38(5):48-56*, 2005.

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

[60] A. Whitaker, M. Shaw, S. Gribble. Scale and Performance in the Denali Isolation Kernel. In Proceedings of *5th Symp. of Operating Systems Design and Implementation*, USENIX, 2002.

[61] S. Devine, E. Bugnion, M. Rosenblum. *Virtualization system including a virtual machine monitor for a computer with segmented architecture*. US Patent, 6397242, Oct. 1998.

[62] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In Proceedings of *5th Symp. of Operating Systems Design and Implementation*, USENIX, 2002.

[63] S. J. Vaughan-Nichols. New Approach to Virtualization Is a Lightweight. In *IEEE Computer, 39(11):12-14*, 2006.

[64] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Revision 3.12, publication #24593, 2006.

[65] K. Fraser, S. Hand, R. Neugebauer, I. Pratt. Safe Hardware Access with the Xen Virtual Machine Monitor. In *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004.

[66] A. Menon, A. Cox, W. Zwaenepoel. Optimizing Network Virtualization in Xen. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC)*, Boston, Massachusetts, U.S., 2006.

[67] J. Sugerman, G. Venkitachalam, B.-H. Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC)*, Boston, Massachusetts, U.S., 2001.

[68] D. Cameron, G. Regnier. *The Virtual Interface Architecture*, 1st Edition, Intel Press, 2002.

[69] J. Liu, W. Huang, B. Abali, D. K. Panda. High Performance VMM-bypass I/O in Virtual Machines. In *Proceedings of the 2006 USENIX Annual Technical Conference (ATC),* Boston, Massachusetts, U.S., 2006.

[70] W. Huang, J. Liu, B. Abali, D. K. Panda. A case for high performance computing with virtual machines. In *Proceedings of the 20th Annual International Conference on Supercomputing (ICS),* Cairns, Queensland, Australia, 2006.

[71] H. Raj, I. Ganev, K. Schwan, J. Xenidis. Self-Virtualized I/O: High Performance, Scalable I/O Virtualization in Multi-core Systems. *Center for Experimental Research in Computer Systems (CERCS) Technical Reports 2006* (GIT-CERCS-06-02), Georgia Institute of Technology, U.S., 2006.

[72] U. Brüning. *Lecture Notes for Computer Architecture II*. University of Mannheim, Germany, 2006.

[73] J. Crawfold. Introducing the Itanium Processors. In *IEEE Micro, 20(5)*, 2000.

[74] H. Sharangpani, K. Arora. Itanium Processor Microarchitecture. In *IEEE Micro, 20(5):24-43*, 2000.

[75] C. McNairy, D. Soltis. Itanium 2 Processor Microarchitecture. In *IEEE Micro, 23(2):44-55*, 2003.

[76] C. McNairy, R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. In *IEEE Micro, 25(2):10-20*, 2005.

[77] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. In *Proceedings of SPIE Real Time Signal Processing IV*, 1981.

[78] K. Olukotun et al. The Case for a Single Chip Multiprocessor. In *Proceedings of the 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, ACM Press, New York, U.S. 1996.

[79] D. M. Tullsen, S. J. Eggers, H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA95)*, June 1995.

[80] J. L. Lo. *Exploiting Thread-Level Parallelism on Simultaneous Multithreaded Processors*. Doctoral Thesis, 1998.

[81] P. Kongetira, K. Aingaran, K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. In *IEEE Micro, 25(2):21-29*, 2005.

[82] L. Spracklen, S. G. Abraham. Chip Multithreading: Opportunities and Challenges. In *Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11)*, 2005.

[83] S. Chaudry, P. Caprioli, S. Yip, M. Tremblay. High-Performance Throughput Computing. In *IEEE Micro, 25(3):32-47*, 2005.

[84] P. Crowley, M. Fiuczynski, J.-L. Baer, B. Bershad. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, NM, U.S., May 2000.

[85] www.kernel.org,
last visited 15.01.2007.

[86] http://www.faqs.org/docs/Linux-HOWTO/BootPrompt-HOWTO.html, last visited 15.01.2007

[87] J. Corbet, A. Rubini, G. Kroah-Hartman. *Linux Device Drivers*, 3rd Edition, O'Reilly, 2005.

[88] J. E. Smith. Decoupled access/execute computer architectures. In *ACM Transactions on Computer Systems (TOCS), 2(4):289-308,* November 1984.

[89] L. Schaelicke, A. Davis. Improving I/O performance with a conditional store buffer. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture,* Dallas, Texas, U.S., 1998.

[90] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 2004 ACM OASIS Workshop,* 2004.

[91] T. von Eicken, D. E. Culler, S. C. Goldstein, K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual international Symposium on Computer Architecture (ISCA92),* Queensland, Australia, May 1992.

[92] W. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *Proceedings of the 14th international Conference on Parallel Architectures and Compilation Techniques (PACT),* IEEE Computer Society, Washington, DC, U.S., 2005

[93] J. Beecroft, D. Addison, F. Petrini, Moray McLaren. Quadrics QsNetII: A Network for Supercomputing Applications. In *Proceedings of Hot Chips 15,* Palo Alto, California, U.S., 2003.

[94] Heiner Litz. *Advanced Hardware Communication Techniques*. Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2005.

[95] Florian Giesen. *Design of a Translation Look Aside Buffer using Content Adressable Memory for a 0.18um Technology*. Diploma thesis presented to the Department of Computer Engineering, University of Mannheim, Germany, 2004.

[96] HyperTransport Consortium. *HyperTransport Technology I/O Link - White Paper*. Available from http://www.hypertransport.org, July 2001.

[97] Hypertransport Technology Consortium, *Hypertransport I/O Link Specification Revision 3.00*. Document #HTC20051222-0046-0008, 2006.

[98] J. Trodden, D. Anderson, *HyperTransport System Architecture*, 1st Edition, Addison-Wesley Professional, 2003.

[99] Advanced Micro Devices (AMD). *AMD Athlon64 Product Data Sheet*. Publication #24659, 2006.

[100] Advanced Micro Devices (AMD). *AMD Opteron Product Data Sheet*. Publication #23932, 2004.

[101] HyperTransport Consortium. *The Future of High Performance Computing: Direct Low Latency Peripheral-to-CPU Connections*. Available from http://www.hypertransport.org, November 2005.

[102] Hypertransport Consortium, *HyperTransport EATX Motherboard/Daughtercard Specification*. Document #HTC2004105-0040-0006, 2005.

[103] HyperTransport Consortium, http://www.hypertransport.org.

[104] Xilinx Corporation. *Virtex-4 Family Overview*. Document ds112 v1.5. Available from http://www.xilinx.com, 2006.

[105] Xilinx Corporation, *Virtex-4 User Guide*. Document ug070 v1.5. Available from http://www.xilinx.com, 2006.

[106] K. Hwang, Z. Xu, *Scalable Parallel Computing*, 1st Edition, McGraw-Hill, 1998.

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

[108] D. Slogsnat, U. Brüning, A. Giese. A versatile, low latency HyperTransport core. In *Proceedings of Fifteenth ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA07)*, Monterey, CA, U.S., February 2007.

[109] M. Nüssle, H. Fröning, A. Giese, H. Litz, D. Slogsnat, U. Brüning. A Hypertransport based low-latency reconfigurable testbed for message-passing developments. In *2. Workshop Kommunikation in Clusterrechnern und Clusterverbundsystemen (KiCC07)*, Chemnitz, Germany, February 2007.

[110] ISC2005 tutorial “Interconnect Architectures in Practice”, results of hands-on exercises. Available from http://www.ra.informatik.uni-mannheim.de, 2006.

[111] K. Hwang, Z. Xu. *Scalable Parallel Computing*, McGraw-Hill, 1998.

[112] ClearSpeed Technology Inc., http://www.clearspeed.com.

[113] Celoxica Limited,
http://www.celoxica.com

[114] DRC Computing Corporation,
http://www.drccomputer.com

[115] XtremeData, Inc.,
http://www.xtremedatainc.com

[116] Advanced Micro Devices, Inc. *ATI Stream Computing.* Available from http://ati.amd.com/companyinfo/events/StreamComputing/index.html, 2006.

[117] Nvidia Corporation. *Compute Unified Device Architecture (CUDA).* Available from http://developer.nvidia.com/object/cuda.html, 2006.

Appendix B

Acronyms

ABI	Application Binary Interface
ACK	Acknowledge
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ATOLL	ATomic Low Latency
BE	Byte Enable
CISC	Complex Instruction Set Computer
CMD	Command
CMP	Chip Multi-Processing
CMT	Chip Multi-Threading
COP	Coprocessor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSB	Conditional Store Buffer
CT	Context Table
DMA	Direct Memory Access
FIFO	First-in First-out
FPGA	Field Programmable Gate Array
FU	Functional Unit
GAS	Global Address Space
GPU	Graphic Processing Unit
HAP	High Availability Port
HOL	Head-of-Line
HP	Host Port
HT	HyperTransport
IDD	Isolated Driver Domain
ILP	Instruction Level Parallelism
IN	Interconnection Network
IPC	Instructions Per Cycle
IRQ	Interrupt
MMU	Memory Management Unit
MP	Multicast Port
MPI-1	Message-Passing Interface 1
MPI-2	Message-Passing Interface 2
MTA	Multi-Threaded Architecture
NI	Network Interface
NIC	Network Interface Controller
NP	Network Port
NQ	Notification Queue
NQE	Notification Queue Entry
O/S	Operating System
OUPID	Origin UPID
P	Process
PA	Physical Address
PIO	Programmed I/O
PRD	Posted Receive Descriptor
PRQ	Posted Receive Queue
QP	Queue Pairs

RDR	Receive Data Region
RMA	Remote Memory Access
RMW	Read-Modify-Write
RS	Routing Space
SDR	Send Data Region
SFP	Small Form Factor Pluggable
SMM	Shared Memory Mapper
SMP	Symmetric Multi-Processor
SMT	Simultaneous Multi-Threading
TCAM	Ternary Content Addressable Memory
TLB	Translation Look-aside Buffer
TLP	Thread Level Parallelism
TUPID	Target UPID
ULTRA	Ultra Low Latency Transmission
UPID	ULTRA Port Identification
VA	Virtual Address
VCI	Virtual Communication Instructions
VIA	Virtual Channel Interface
VLIW	Very Long Instruction Word
VM	Virtual Machine
VMM	Virtual Machine Monitor
VPID	Virtual Port Identifier
WDT	Window Descriptor Table
Win-ID	Window Identification
WQ	Work Queue

APPENDIX C

List of Figures

APPENDIX D
Index