ACCELERATION OF THE HARDWARE-SOFTWARE INTERFACE OF A COMMUNICATION DEVICE FOR PARALLEL SYSTEMS

Inauguraldissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

der Universität Mannheim

vorgelegt von

Mondrian Benediktus Nüßle
(Diplom-Informatiker der Technischen Informatik)

aus Mannheim

Mannheim, 2008

Dekan:	Professor Dr. F. Freiling, Universität Mannheim
Referent:	Professor Dr. U. Brüning, Universität Heidelberg
Korreferent:	Professor Dr. R. Männer, Universität Heidelberg

Tag der mündlichen Prüfung: 18.02.2009

Für Carola

Abstract

Abstract

During the last decades the ever growing need for computational power fostered the development of parallel computer architectures. Applications need to be parallelized and optimized to be able to exploit modern system architectures. Today, scalability of applications is more and more limited both by development resources, as programming of complex parallel applications becomes increasingly demanding, and by the fundamental scalability issues introduced by the cost of communication in distributed memory systems. Lowering the latency of communication is mandatory to increase scalability and serves as an enabling technology for programming of distributed memory systems at a higher abstraction layer using higher degrees of compiler driven automation. At the same time it can increase performance of such systems in general.

In this work, the software/hardware interface and the network interface controller functions of the *EXTOLL* network architecture, which is specifically designed to satisfy the needs of low-latency networking for high-performance computing, is presented. Several new architectural contributions are made in this thesis, namely a new efficient method for virtual-to-physical address-translation named *ATU* and a novel method to issue operations to a virtual device in an optimal way which has been termed *Transactional I/O*. This new method needs changes in the architecture of the host CPU the device is connected to. Two additional methods that emulate most of the characteristics of *Transactional I/O* are developed and employed in the development of the *EXTOLL* hardware to facilitate usage together with contemporary CPUs. These new methods heavily leverage properties of the HyperTransport interface used to connect the device to the CPU. Finally, this thesis also introduces an optimized remote-memory-access architecture for efficient split-phase transactions and atomic operations.

The complete architecture has been prototyped using FPGA technology enabling a more precise analysis and verification than is possible using simulation alone. The resulting design utilizes 95 % of a 90 nm FPGA device and reaches speeds of 200 MHz and 156 MHz in the different clock domains of the design. The *EXTOLL* software stack is developed and a performance evaluation of the software using the *EXTOLL* hardware is performed.

The performance evaluation shows an excellent start-up latency value of *1.3 μs*, which competes with the most advanced networks available, in spite of the technological performance handicap encountered by FPGA technology. The resulting network is, to the best of the knowledge of the author, the fastest FPGA-based interconnection network for commodity processors ever built.

Zusammenfassung

Der immer weiter steigende Bedarf nach Rechenkapazität führt zu einer fortschreitenden Parallelisierung im Bereich der Rechnerarchitektur. Anwendungen müssen parallelisiert und optimiert werden, um die Möglichkeiten moderner Architekturen ausnutzen zu können. Die Skalierbarkeit von Anwendungen ist heute immer häufiger limitiert durch die mangelnde Verfügbarkeit von Entwicklern, da das Programmieren von immer komplexeren parallelen Anwendungen eine sehr fordernde Aufgabe darstellt, und durch die fundamentalen Probleme der Skalierbarkeit, die durch die Kosten von Kommunikation in verteilten parallelen Systemen entstehen, wird dieses Problem weiter verschärft. Es ist unbedingt notwendig die Latenz der Kommunikation zu verringen, um dadurch die Skalierbarkeit zu steigern. Niedrige Latenz ist dabei eine technologische Voraussetzung, um verteilte System leichter, mit höherer Abstraktion und vermehrter compilerbasierte Automation zu programmieren. Gleichzeitig kann sie die Leistung der Systeme im Allgemeinen erhöhen.

In dieser Arbeit werden die Software/Hardware Schnittstelle und die Funktionen des Netzwerkcontrollers der EXTOLL Netzwerkarchitektur vorgestellt, welche speziell entwickelt wurde, um die notwendigen Bedingungen eines Netzwerks mit niedriger Latenz für das parallele Rechnen zu erfüllen. Mehrere neue Beiträge im Bereich der Rechnerarchitektur werden vorgestellt, insbesondere eine Methode zur effizienten Übersetzung von virtuellen in physikalische Adressen durch ein Netzwerkgerät, welche *ATU* genannt wird, und ein neues Verfahren, um Befehle an ein virtualisiertes Gerät abzusetzten, welches *Transactional I/O* heißt. Dieses neue Verfahren setzt allerdings Änderungen an der CPU und dem Verbindungsnetzwerk zwischen CPU und Gerät voraus. Um Systeme mit heutigen Prozessoren zu ermöglichen, werden zwei weitere neue Methoden vorgestellt, die *Transactional I/O* emulieren und die Haupteigenschaften von *Transactional I/O* aufweisen. Diese Verfahren, welche in starkem Maße Funktionen der HyperTransport-Schnittstelle einsetzen, werden für EXTOLL umgesetzt. Schließlich wird eine optimierte Remote-Memory-Access Architektur eingeführt, die sehr effiziente Kommunikation sowie atomare Operationen ermöglicht.

Die komplette EXTOLL Architektur wird auf einem FPGA als Prototyp implementiert. Auf diese Weise wird eine detailliertere Analyse und Verifikation der Architektur ermöglicht, als sie durch die Verwendung von Simulation allein erreicht werden könnte. Im Ergebnis werden 95 % der Ressourcen eines 90 nm FPGAs verwendet und das Design erreicht 200 MHz respektive 156 MHz in den verschiedenen Clock-Domains. Die Software für EXTOLL wird vorgestellt und eine Evaluation der erreichbaren Leistung durchgeführt.

Die Ergebnisse zeigen, dass EXTOLL trotz den Leistungsnachteilen, die durch eine FPGA Umsetzung entstehen, höchste Leistungen erreicht und mit einer Kommunikationslatenz von *1.3 μs* mit den schnellsten heute verfügbaren Netzwerktechnologie mithalten kann. Nach bestem Wissen des Autors ist EXTOLL damit das schnellste FPGA-basierte Netzwerk, das jemals zur Verbindung von handelsüblichen Computern gebaut wurde.

Contents

TOC

Contents I

1 Introduction 1

1.1 State of the Art 6

1.2 Outline 13

2 The ATOLL Software Environment 15

2.1 The ATOLL-Project 15

2.1.1 ATOLL Software Environment - Overview 16

2.2 PALMS 18

2.2.1 Memory Layout of an ATOLL Hostport 19

2.2.2 PALMS Design 21

2.3 Managing ATOLL 23

2.3.1 AtollManager 27

2.3.2 Additional Tools 28

2.4 MPICH2 for ATOLL 30

2.5 Debugging the ATOLL ASIC 33

2.6 Performance of ATOLL 36

2.6.1 Microbenchmarks 37

2.6.2 Application Level Benchmarks 39

2.6.3 High-Performance LinPACK 39

2.7 Evaluation of Larger Networks 41

2.8 Zero-Copy and ATOLL 48

2.9 Lessons Learned from ATOLL 49

3 EXTOLL System Environment 51

4 Communication Paradigms 57

4.1 Two-sided Communication 57

4.2 Remote Load/Store 59

4.3 Introduction to One-Sided Communication 61

4.4 Important Communication APIs 63

4.4.1 Sockets 63
4.4.2 MPI-1 65
4.4.3 MPI-2 67
4.4.4 PGAS 72

4.5 Conclusions for EXTOLL 77

5 The Virtual Address Space Barrier 79

5.1 State of the Art 80

5.1.1 X86-64 Processor MMU 80
5.1.2 Classical Devices and the Linux DMA API 82
5.1.3 Mellanox Infiniband HCA 83
5.1.4 iWARP Verbs Memory Management 86
5.1.5 Quadrics 87
5.1.6 Myrinet MX 87
5.1.7 SciCortex 88
5.1.8 Graphics Aperture Remapping Table 88
5.1.9 IBM Calgary IOMMU 90
5.1.10 AMD IOMMU and Intel VT-d 91
5.1.11 PCI Express and HT3 Address Translation Services 95
5.1.12 Virtual Memory Hooks in the Linux Kernel 97

5.2 Design Space of the EXTOLL Address Translation 101

5.2.1 Interrupt Driven Software-Only Approach 103
5.2.2 Software Pre-translation 103
5.2.3 Managed TLB 104
5.2.4 Autonomous TLB 105
5.2.5 Full Hardware Table-Walk 105
5.2.6 Reduced-Depth Hardware-Table Walk 106
5.2.7 Registration Based versus Kernel-Hook Based Designs 106
5.2.8 VPID Handling 106
5.2.9 On-Device ATS 108
5.2.10 Conclusion 109

5.3 The EXTOLL Address Translation Unit 109

5.4 ATU Microarchitecture 116

5.5 ATU Verification and Implementation 117

5.6 Performance Analysis 118
5.7 Future Extensions 122

6 Transactional I/O 123

6.1 EXTOLL Requirements 123
6.2 The Classical Approach 124
6.3 Hardware Replication 126
6.4 Self Virtualized Devices 127
6.4.1 Triggerpage Study 128
6.4.2 I/O Transactions 130
6.4.3 Central-Flow-Controlled Queue 132
6.4.4 Central-Flow-Controlled Queue with Direct Data Insertion 134
6.4.5 OS Synchronized Queue 135
6.5 Completion Notification 136
6.6 RX Virtualization 138
6.7 Conclusion 139

7 The EXTOLL Hardware 141

7.1 HT-Core and HTAX 142
7.2 Registerfile 143
7.3 EXTOLL Network Layer 144
7.4 EXTOLL VELO Engine 146
7.5 EXTOLL RMA Engine 150
7.5.1 RMA Instructions 151
7.5.2 PUT Instructions 153
7.5.3 GET Instructions 153
7.5.4 RMA Remote Lock Instruction 153
7.5.5 Physical vs. Virtual Addressing 160
7.5.6 RMA Microarchitecture 160
7.5.7 RMA Registers 162
7.6 EXTOLL URMAA engine 163
7.7 EXTOLL FPGA Implementation 163
7.8 EXTOLL ASIC 166

8 The EXTOLL Software 169

8.1 The EXTOLL Kernel Space Software 169
8.1.1 Base EXTOLL Driver 169
8.1.2 EXTOLL Registerfile Driver 170
8.1.3 VELO Driver 171
8.1.4 RMA Driver 171
8.1.5 ATU Driver 171
8.1.6 Summary of Drivers 171
8.2 Routing and Management 172
8.3 The VELO Stack 172
8.4 The EXTOLL RMA Software Stack 173
8.5 EXTOLL Kernel-Level Communication 173
8.6 EXTOLL MPI - Protocols 174
8.7 EXTOLL GASNET - Protocols 180
8.8 Software Summary 181

9 Results and Performance Evaluation 183

9.1 Microbenchmark Results 183
9.2 RMA one-sided MPI-2 Prototype 188
9.3 MPI-1 Protocols 188
9.4 Summary of Results 191

10 Conclusion and Outlook 193

A Graphical Representation and Methods 197

B Acronyms 199

C List of Figures 203

D List of Tables 207

R References 209

Introduction

Chapter 1

The need for efficient interconnection networks in parallel and distributed computing is ever growing. The last decade saw the rise of cluster machines in the TOP500 list [1] of the fastest super-computers of the world. Multi-core CPUs, multi-CPU SMP (symmetric multi-processing) systems and clusters of SMPs demand highly parallel algorithms to use their computing power efficiently. In the same time period the race for higher clock speeds has diminished, mainly due to the excessive power dissipation of extremely high-clocked designs. Instead, many-core architectures bring parallel computing increasingly to the main-stream and consumer domains.

Efficient exploit of parallel architectures is a difficult and a demanding task to programmers. Algorithms only scale by a certain amount; a typical application speed-up curve is shown in figure 1-1. When adding more and more processors beyond a certain number of processors, the execution time of the application does not improve any more or even degrades. Reasons for this behavior are the costs of communication and synchronization which can only be overcome by either reducing the cost for these operations or hiding these costs through clever algorithm and application design. At the same time it is crucial that parallel application performance is improved; but without enhanced systems as base, the programmer is left alone to realize the necessary improvement. The necessity to extremely optimize and tune an application to reach the computational performance expected does further increase the difficulty level of parallel programming.

Even more difficult than parallel programming in general, is designing applications for distributed memory systems. Programmers that truly understand distributed memory programming and that are qualified to produce efficient code are a scarce resource today. It is for these reasons that clusters are sometimes used as throughput systems, meaning they are used as a pile of individual servers, each processing one job. While the time for each job is not improved, at least a higher number of jobs can be executed. But the goal for the future must be to improve the usage of parallel machines on all levels.

The necessity to address the problem of programming parallel systems has been accepted by academia and the industry equally. For many-core shared-memory applications, new development tools have appeared on the market by major vendors including the CPU manufacturers itself [3], who have also recognized the problem of programmability, as well as numerous projects from research initiatives. In the area of distributed-memory-systems better programmability is addressed by current research trends centered on *Partitioned-Global-Address-Space* (PGAS) languages [4]. It is also one of the goals of the DARPA High Pro-

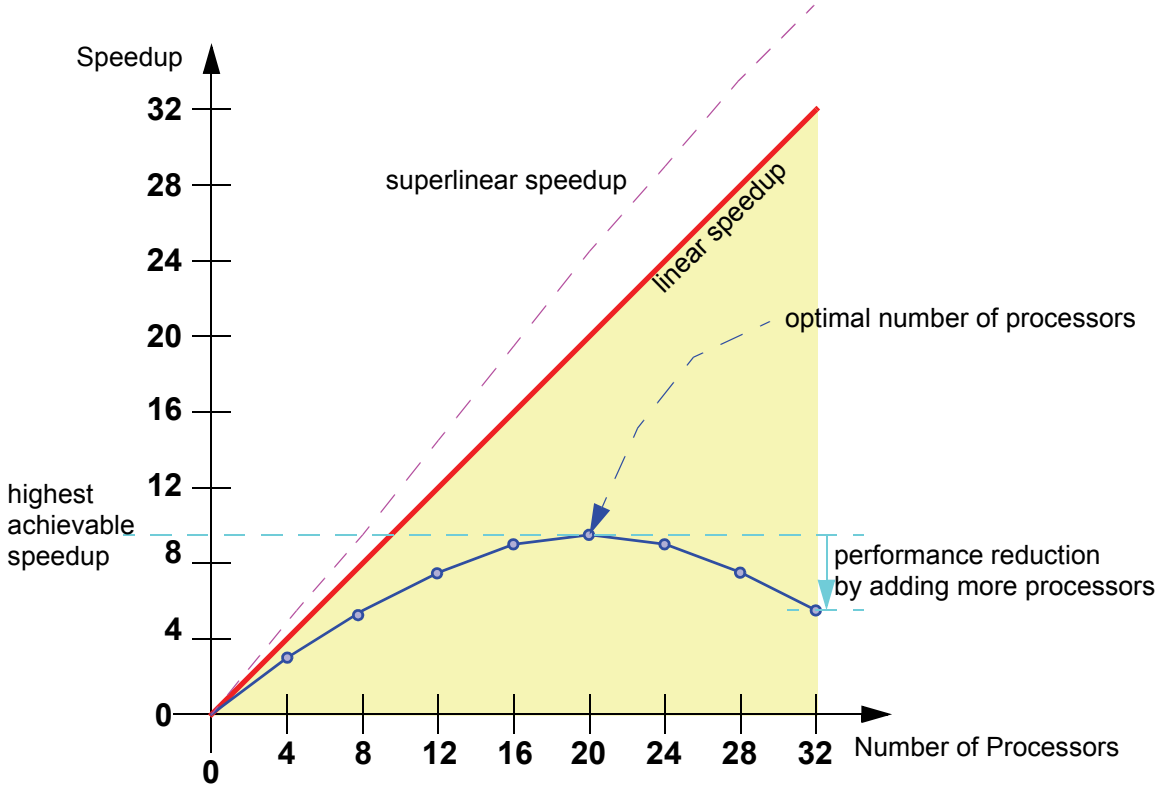


Figure 1-1: Limited Speed-up of Parallel Applications [2]

ductivity Computing Systems Program [5] to foster the development of tools for better programmability of high-performance, distributed-memory systems. One of the primary goals of all these initiatives is to increase the amount of automations when developing parallel applications, i.e. raise the level of abstraction in the specification of the program. This can help programmers in many ways and accelerate development times by removing difficult machine-dependent decisions form their shoulders, but at the same time it creates new challenges for the runtime and development software to produce efficient applications. Now, it is the compiler that needs to take into account the placement of threads and data on the machine, and to insert the appropriate communication operations in such a way that performance is optimized.

Consequently, an important component of parallel systems that influences the reachable performance levels both in the case of careful manual tuning as well as automatically machine-generated code is the interconnection network of the system. It was already mentioned, that the costs of communication and synchronization are probably the most important factors limiting scalability of parallel applications. Interconnection networks have been characterized and studied using different models, but two performance parameters have always been identified as most important: latency and bandwidth.

Bandwidth of commodity interconnection networks has expanded exponentially from 1998 to 2008, from a hundred megabits to in excess of 20 gigabits per second. However, at the same time the latency to transport a message from sender to receiver has not seen such dramatic improvements. Due to this discrepancy Patterson states “Latency lags bandwidth” [6] and there are many reasons for the lack of reduction of latency. One particularly trivial reason is the marketing strength of bandwidth - bandwidth sells, latency is more difficult to convey [6]. It is also fundamentally easier to increase bandwidth than latency - doubling of the number of parallel wires doubles the bandwidth whereas to half the latency no such simple solution exists. However, today latency becomes the truly limiting system property for parallel systems.

Low-latency systems drive future application scalability and low-latency interconnects are a key factor for low-latency systems. This can considerably ease programming for such a system, because less effort has to be spent to hide latency and optimize the application to cope with the deficiencies of the system. The productivity of individual programmers is increased and compiler and runtime for future PGAS systems can benefit tremendously from low-latency communication in the system. Thus, optimized latency can become an *enabling technology* for future systems. But even today, many markets that need high computational power and rely on low- to medium-processor-count clusters can benefit from reduced latency interconnection networks. In the enterprise market, applications are often inherently latency sensitive: clustered and distributed databases need low-latency messaging as a basis; on-line transaction processing (OLTP) benefits from it, and finally for many financial and analysis applications (for example quantitative analysis systems) low-latency communication and processing is absolutely mandatory. In [7] and [8] the implications for the scalability of distributed OLTP databases are mentioned, albeit rather marketing oriented. In a report from the 2*007 Linux Kernel Summit Customer Panel*, an information technology manager from *Credit Suisse* allegedly said that *Credit Suisse* “would like to get full bandwidth out of high-performance network adapters while sending large numbers of small (64 byte) packets” [9] for the in-house Linux based banking applications. Finally, in the more traditional high-performance computing (HPC) markets, it is well known that technical and scientific codes benefit from low latency: new exploitations of parallelism become possible, algorithms scale to larger node counts with a better overall performance, and, as already mentioned, application developer productivity can be increased, which is a key cost factor in times of ever growing application code sizes and complex systems.

To effectively lower system latency, the whole system architecture needs to be considered. In the case of an interconnection network this also involves the host architecture, the interface between host and the *network interface controller* (NIC), the NIC architecture itself, the network layer and last but not least the software components of the system. On the software side, standards have emerged to enable low-latency communication from application to application. Besides the well known *Message Passing Interface-1* standard (MPI-1), one-sided communication as defined in MPI-2 and the PGAS model are noteworthy. While these software standards describe a model to enable efficient communication with low latency, many current implementations considerably lack performance in this area. This is both due to non optimal software implementations and interconnection network structures.

It will be shown in this thesis that the architectural improvements described do not suffer from these limitations, but support a very efficient implementation of one-sided and two-sided communication primitives effectively accelerating communication operations.

The *EXTOLL* (Extended ATOLL) project has the declared goal to bring node to node latency down into the sub-microsecond area and help to build more efficient systems. In this work, this new network interface architecture is presented. A holistic approach is needed to decrease system communication latency and accelerate host-device interaction. Improving system bandwidth can be obtained in a straight-forward manner, but only careful minimization and optimization of latency will bring the necessary scalability and performance in many areas of computing of tomorrow. Particular emphasis is put on acceleration and improvement of the hardware-software interface, so both hardware structures on the NIC and the corresponding software function of the host are considered. Coming from the experiences with the development of the software environment of the ATOLL (ATOmic Low Latency) [10][11][12] project presented in chapter 2, the design space for a complete new network interface architecture is thoroughly analyzed, especially the problems associated with host-device interaction. This covers the paradigm of communication between host and NIC, the software necessary to implement this on the CPU side, and the problems of virtualization, posting, and completion of operations, where several new contributions are made. A very efficient method for virtual-to-physical address translation is presented, an essential building block to enable true zero-copy and Remote-Memory-Access (RMA) NIC architectures where the NIC is programmed directly from user-space. The architecture and the virtualization features also heavily leverage features of the HyperTransport [13] interconnect which is used to connect the NIC to the host. The design space analysis in these key areas is then used to develop the host-device interface for the EXTOLL implementation.

Two completely new communication engines are presented in this thesis. *VELO* (Virtualized Engine for Low Overhead) is a thoroughly optimized architecture to send and receive messages of a size of up to one cacheline (64 byte). It uses a novel, virtualized programmed Input/Output (PIO) method to post operations and completes them with the least possible overhead using the minimum of DMA (direct memory access) operations on the receiving side. This engine is able to reach an unprecedented low latency close to one microsecond which is presented in the software and evaluation part of the thesis.

The second communication function presented is called *Remote Memory Architecture* (RMA) engine. It is a function that is optimized to efficiently handle one-sided and larger two-sided communication patterns. RMA is optimized for low latency and overlap of communication and computation, a point that many interconnection networks (INs) with a low microbenchmark latency lack. To facilitate these features, RMA takes advantage of the novel and efficient virtualized command handling method presented in chapter 6. A minimum, yet ample command set is developed to cover the necessary operation conditions for the RMA unit. To implement optimal computation/communication overlap a novel notification system for operation completion signaling and a locking architecture is designed which enables very efficient split-phase transaction synchronization over the network. On the soft-

ware side, all the protocols needed to make efficient use of the EXTOLL hardware architecture are presented, including two efficient two-sided protocols on top of the RMA engine for medium and large message sizes.

The performance of the system architecture of EXTOLL is not only based on the new, improved host-device interaction, though, but also relies on other parts being optimized, including the HyperTransport interface to connect the NIC functions to the host, and an extremely efficient network layer. Both parts are out of the scope of this work and more information about them can be found in [14][15][16][17][18] and [19].

With the growing complexity, hardware research has often focused on simulation since the costs, development time and risk to develop and manufacture a custom chip, for example in ASIC (application specific integrated circuit) technology are becoming more and more prohibitive. System simulation may introduce errors in terms of system evaluation and, more important, is limited in the amount of verification and information about correctness that it can offer. Therefore, the implementation of a prototype of the EXTOLL architecture was undertaken and after careful analysis of the technological options a full system implementation in FPGA (field-programmable gate arrays) technology was chosen.

The advent of large FPGA devices enables this kind of complete system prototypes of new hardware without having to produce custom chips. FPGA devices have continuously been improved both in capacity and performance in recent years and can increasingly support such complex designs. Of course, an FPGA implementation of a given hardware architecture will never be able to reach the performance levels that can be obtained using ASIC technology for the same architecture, but today it is possible to implement designs that, using superior architectural features, can compete against existing well-optimized ASIC devices, as the results of this work will show. FPGAs also offer further advantages, because of their reprogrammability features. It becomes possible to move hardware design closer to the way software is developed so that a faster innovation cycle is possible. Furthermore, for completely new architectures, the possibility to rapidly iterate over gradually improving architectures and analyzing them in system is a priceless advantage. In the area of special purpose architectures, FPGAs can also play their strength, since hardware devices with low quantities of installed devices become possible and so special purpose variants of an architecture which reuse parts of a design become profitable; in fact components of EXTOLL are already actively being used for other research projects.

As a result, the FPGA implementation of the EXTOLL architecture together with the necessary software layers are presented here and the whole system is evaluated using microbenchmarks and several specialized tests to cover the special features of the EXTOLL network proving the great potential of the architecture. The full system prototype enables detailed studies of the resulting architecture, provides support for software development and gives the security that the resulting hardware actually co-operates with the different components of the overall system, most importantly with the CPUs. Additionally, the prototype functions as a baseline implementation for further research in the area of networking and computer architecture.

In short, the main contributions of this work are:

- a complete software environment for the ATOLL network,
- a novel address translation unit for virtual-address space handling in virtualized devices,
- design, analysis, benchmarking and implementation of new virtualization techniques,
- a new architecture for low-latency communication which also efficiently supports real-world problems,
- analysis and implementation of the software to drive low latency from innovative hardware architecture to complete system software, and
- a holistic, system-view driven approach to optimize the whole system from hardware micro architecture to software middleware packages.

The evaluation of the resulting prototype is presented in chapter 9, which shows the impressive potential and performance of the EXTOLL architecture. To better understand the contributions of the EXTOLL architecture, the state of the art of networking for parallel, distributed memory architectures, is outlined in the next section.

1.1 State of the Art

To implement a low-latency system, it is not enough to address the problem of accelerating the host-device and hardware-software interface, which is the main topic of this work, but the complete network and host system needs to contribute to this goal. Current networking solutions generally lack some features necessary for the envisioned performance and characteristics that the EXTOLL system will offer. The following overview highlights the advantages and shortcomings of modern networks for parallel computing.

Probably the most widespread networking hardware is Ethernet and its many variants. In the high-performance sector, both Gigabit Ethernet (GE) and also 10-Gigabit Ethernet (10-GE) are used. Ethernet NICs are still mostly traditional devices which are tied very closely to the use of the TCP/IP stack. The use of interrupt driven reception and the overhead of the TCP/IP stack in the kernel are one reason, why Ethernet is not a low-latency network and generally also not the best choice for parallel systems (exceptions exist of course). The effects of these factors are concisely presented in [20]. Another point against Ethernet, and in a way the reason why the TCP/IP stack is necessary, is the lack of efficient hardware retransmission and flow-control features in the network. While there have been efforts to integrate some flow-control into newer Ethernet standards, these solutions are far from the effective flow-control needed for high-performance networks. This shortcoming causes the need for elaborate end-to-end protocols such as TCP to enable reliable communication between end-points. Another major problem is the waste of memory bandwidth by the classic stack. Usually each and every packet is copied by the CPU at least once when sending and once when receiving, sometimes even more often. At 1 GB/s link rate this can amount to several gigabytes of memory bandwidth wasted for the copying of packets in main memory. This is one major fact why *zero-copy-protocols* and RMA or *remote-direct memory-*

access (RDMA) architectures are becoming more and more popular. Recent improvements in classical Ethernet adapters include the support of multiple send and receive queues for a more efficient use of multi-core and virtualized servers [21].

The next class of network devices contains the *enriched* Ethernet devices. The CPU load that is caused by the higher-speed Ethernet variants is significant; TCP/IP processing alone can completely utilize single processors, which is not desirable of course. To counter this problem, some vendors have developed adapters that feature their own processors or processors and memories which offload the TCP/IP processing completely or for the most part from the host CPU. Such adapters are often called TOEs (TCP Offload Engines). TOEs do not enable significantly lower latency than well tuned systems using standard Ethernet adapters; this is apparent, since the embedded processors are still required to process the TCP/IP protocol. An often quoted critique on TOEs is also, that they are prone to errors, security flaws and incompatibilities since they are not as well supported as the TCP/IP stack software in use by the major operating systems. Nevertheless they succeed in freeing CPU resources for other work than network packet processing.

Ethernet networks and adapters are nearly always accessed from applications using the BSD Sockets API (Application Programming Interface) or one of its variants (see also section 4.4.1). The Sockets API is closely coupled with the POSIX standard library, and thus communication can be caused by calls to the *read()* or *write()* system call. A host of other functions exists, too. In the TCP case, connection management is clients/server oriented and generally, all communication calls pass through the kernel.

In recent years a number of new networking devices appeared which promised to remedy all the problems of Ethernet. These devices are currently represented by the Infiniband (IB) and the iWARP host-channel-adapters and their respective networks. All of these adapters use a variant of an API called *Verbs* [22]. *Verbs* as well as many other features of these networks were first invented for the Virtual Interface Architecture (VIA). Infiniband introduces significantly lower latencies than Ethernet and also promises the use of RDMA. VIA and then Infiniband are also examples for networks that offer *virtualized* hardware access. The current many-core trend of more and more CPU cores in one system forces NIC architectures to become *virtualized* to support the ever growing number of end-points in a system. In this work, *end-point* references the software abstraction which is used by one process, thread or OS (Operating System) to communicate with a network device. A typical SMP server only five years ago featured two to four CPUs and thus two to four concurrent MPI process in a typical numerical computation environment; today, systems with 8-16 cores and the same number of MPI processes are common and it is expected that this number will increase considerably over the next years. This increase in cores causes the need to support enough end-points and to move from a dedicated or replicated device structure to a virtualized structure. The virtualization of the device can help to accelerate the hardware-software interface to reach low latency in two main aspects: it can be leveraged to implement complete user-space access to the device and eliminate data copy operations. Virtualized devices also offer interesting new possibilities in the area of virtual machines [23].

The VIA specification from 1997 [24] was a predecessor to the Infiniband standard and lead to many of the design decisions of IB. VIA used the architecture depicted in figure 1-2. The

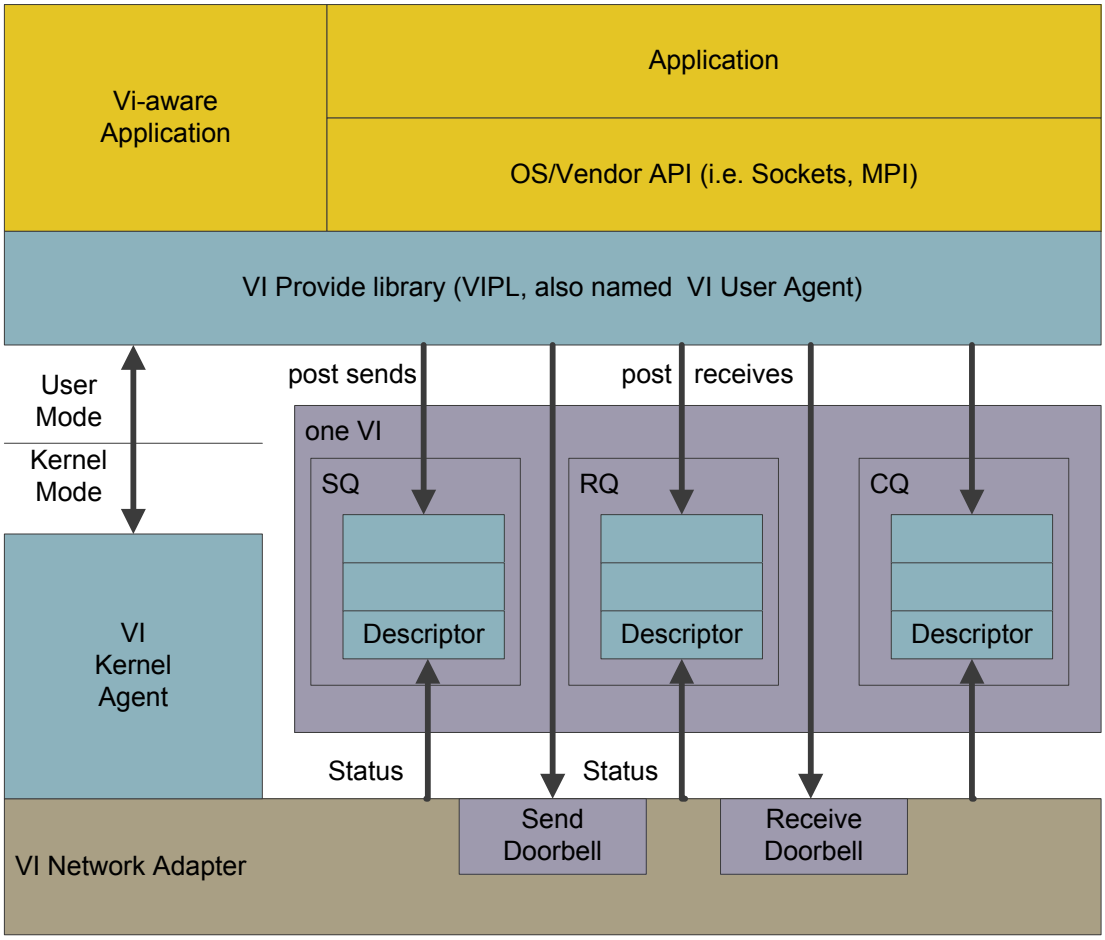


Figure 1-2: Virtual Interface Architecture

interface to the network adapter is one Virtual Interface (VI). An most important idea of VIA was, that a VI is virtual in the sense, that the actual hardware can provide a VI to literally hundreds or thousands of client processes on a single node, i.e. this is an implementation of a virtualized device (see also section 6.2). The user-level API of VIA is known as *VIPL* (VI provider library). The important communication resources of the user interface are the q*ueue-pairs* of a *send queue* (SQ) and a *receive queue* (RQ) and, as a third important resource, the *completion queue* (CQ). To start a transaction, a descriptor is built and inserted into the SQ and the send doorbell is rung (conceptually this could be a write to a register in the network adapter updating a write-pointer). The operation is then processed by the VI network adapter asynchronously to the application and once the operation finishes a completion descriptor is inserted into the CQ, on which the application can poll or wait. For two-sided operations, *receives* are posted into the RQ by the application and the receive doorbell is rung. The actual communication is processed very similarly to IB and described

in more detail further on in this section. Connection management is a basic client/server model. The server calls *VipConnectWait()* to wait for incoming connection requests, while the client calls *VipConnectRequest()* to initiate a request to a given server. In peer-to-peer applications, like MPI, processes can also act both as server and clients.

In VIA all memory to be used in data transfer operations must be registered prior to being used. The memory registration is normally handled by the VI kernel agent (i.e. the kernel driver of the VI NIC) and typically locks the physical pages, enters the pages to be registered in an internal data structure, which typically resides on the NIC, and returns a handle for the newly registered memory region. The VIA specification states that the NIC is responsible for the virtual to physical translation; memory is always referenced as a tuple of *{handle, virtual address}*, where the handle is an opaque handle returned by the memory registration function. Protection is handled by protection tags and memory protection attributes (RDMA enable/disable, write access, read access, and per VI access) associated with individual memory regions.

RDMA write (also known as *put*) operations are mandatory in the specification but *RDMA read* operations (also known as *get*) are optional. Not many hardware implementations of VIA were actually designed and built due to the relatively high hardware effort (for the 1990s) necessary to support the complete specification. One hardware design is described in [25]; an open-source academic software-only implementation is M-VIA [26].

Today, VIA has been superseded by IB. The IB specification [22] specifies a software interface to clients, like VIA did. This interface is specified as a semantic interface, so it describes what *must be provided* and what it should do, but it does not specify *how* that should actually work and it does not specify a concrete API, i.e. exact functions with parameters. This concept is called *Verbs*. In essence the *Verbs* specification defines an abstraction above a concrete API and is similar in many respects to VIA's VIPL. Several IB APIs (*Verbs*) are available that are sometimes incompatible. Two of the most prominent *Verbs* are *Mellanox Verbs*, the *Verbs* implementation for Mellanox HCAs (Host Channel Adapter, the IB jargon equivalent of NIC), and the *Verbs* that is part of the *OpenIB* distribution, which is now part of the *Open Enterprise Fabric Distribution* (OFED) [27]. The IB *Verbs* specification is heavily influenced by the VIA specification which can be seen as a predecessor of the IB specification.

But IB also specifies the other parts of a complete network. A number of different transport layer services are defined which can be used via the API; namely these are reliable connection, reliable datagram, unreliable datagram and unreliable connection. The end-point of an IB communication on the API side is again called a Queue-Pair (QP), and at the creation time of such a QP the transport service can be chosen. The chosen service then has an influence on the communication semantics and on the availability of certain *Verbs* respectively parameters to *Verbs*. Connection management is performed on the QP layer. All memory used in IB *Verbs* as source or destination of an operation must be registered (see section 5.1.3 for more details). The notion of protection domains (i.e. memory management con-

texts), memory regions (areas of registered memories), and memory windows (sub-regions attached to individual QPs) can be used to control the access to registered process memory which is necessary to support typical client-server datacenter use-cases.

An HCA generally provides a limited amount of resources; these can be queried using the respective *Verb*. For example the number of contexts, QPs, the maximum size of a memory region, the maximum number of memory regions, the maximum number of completion queues and the maximum queue sizes can be queried. A Mellanox Connect X HCA provides 64k QPs and a maximum of 128k memory registrations which are shared by up to 510 user contexts (i.e. end-points).

IB Verbs supports a number of different communication transfer operations. The *post send request* work-request (WR) causes the *send* part of a two-sided operation. The source buffer has to reside in registered memory and is sent using a zero-copy *send* protocol. *Send* transactions are completed using a *receive* request. When a *send* request reaches the receiving HCA, the request is matched to the receiving QP and the top most posted *receive* WR is removed from the receive queue to complete the send request. From the HCA's point of view this constitutes a zero-copy completion. This scheme poses several problems to higher level protocol developers. First, it is necessary to always post enough receive WRs to the receive queue of each and every QP of the application; the queue is never allowed to be drained, since severe performance penalties or data loss can be the consequence. Also, each posted receive must have an attached buffer which must have the size of the maximum transfer unit (MTU) and must reside in registered memory. The large number of receive buffers that need to be posted at all times leads to a high memory consumption of IB communication.

Later revisions of IB introduced the concept of the shared receive queue (SRQ). If using a SRQ, receives are posted to the SRQ instead of one of the RQs. Multiple queue pairs can be associated with one SRQ. Whenever a *send* transaction arrives for any of the QP, which is associated with the SRQ, the top-most *receive* WR from the SRQ is consumed and the *send* transaction is completed using this WR. The completion notification is delivered to the CQ of the QP and the completion entry is logically filled with values, as if the completion would have been performed with a WR from the dedicated RQ of the receiving QP. This scheme can be used to somewhat reduce memory footage.

Another down-side for the two-sided IB communication primitives is, that they do not allow zero-copy two-sided *receives* natively. Since receive buffers are pre-posted and only matched in posting order, it is very difficult to make sure data is placed in the correct application buffers; an intermediate copy from the receive buffer to the application buffer will often be necessary. One common idea to work around this is, to only send short messages using send WRs and implement large send operations using a combination of a message carrying matching information and source address of the send operation. The receiver matches this information with its view of application side buffers and completes the two sided operation through the use of a one-sided *get* WR which directly places the original data into the application buffer. The added latency of one round-trip can often be tolerated

for large bulk transfers, since the advantages of not having to copy the received payload and replenishing the (S)RQ all the time. This idea actually can be used generally for RDMA/ RMA enabled NICs and is also a basic use-case idea for the EXTOLL RMA unit.

One sided WR requests are *RDMA write* and *RDMA read* operations, which have to operate on registered memory on both sides. Local memory can be passed to the API in form of a scatter-gather list. The HCA can optionally support atomic operations (*compare and swap*, *fetch and add*). WRs can generate completions, which in turn generate entries in the CQ, on which the application can poll (or sleep). Note that there is no remote notification, i.e. the one-sided operations are truly one-sided and there is no direct way for a process to know that it was the target of a RMDA operation.

In 2008 Mellanox has introduced its newest generation of IB HCAs called Connect X. This adapters feature native PCI Express (PCIe) 2.0 support and provide very good latencies in modern systems equipped with PCIe 2.0 slots. As low as 1.2 µs were measured on the MPI level according to the authors of MVAPICH [28].

The host interface part of Infiniband does support some of the necessary operations for a modern high-performance network. Unfortunately, it does lack some elegant features to support upper-level protocols, such as remote notification of operation completion or atomic operations that can be used to implement all of the variants of *MPI_Lock*. Also, the tested HCAs feature a high overhead for memory registration and deregistration which is prohibitive for highly dynamic buffer and memory usage (see also section 5.1.3). The memory registration latency for a single page is more than ten fold the time to send and receive a message. The underlying network is also specified in the Infiniband specification, and while it allows for flexible topologies, high-bandwidth and low-latency switches, it also has some drawbacks, the most prominent being the lack of link based retransmission, so that error checking and re-transmission must be completely accomplished by the end-points. Since reliable communication in IB has to be combined with connected communications, each connection to a peer needs at least some memory for management purposes. While SRQs have diminished memory usage, it remains a problem that is often cited when scaling to large systems. In one study, the amount of memory that has to be dedicated for communication management in a 4096 node cluster was stated to be 1 GB on each node [29].

A special case of an Infiniband HCA is the Qlogic Infinipath HTX adapter (formerly Pathscale Infinipath) [30]. This adapter connects to a standard Infiniband fabric, but implements a different host interface than defined in the specification. The Infinipath adapter does not support standard *Verbs* operations but is optimized for send/receive MPI traffic. It is based on the concept of *on-loading*, the opposite concept of *off-loading*. Not only is protocol processing and matching performed on the host CPU, but all sending has to be performed based on PIO, as Infinipath has no TX DMA engines. It leverages a fast HyperTransport interface and achieves very good end-to-end latencies - 1.14 µs is the start-up latency using the adapter back-to-back. Around 1.3 µs are achievable using an intermediate switch. These numbers were measured at the MPI level. Critics of the adapter point out that the performance is paid for by high CPU load, because sending is PIO based and it does not to employ zero-copy techniques.

Closely related to IB is the Internet Wide Area RDMA Protocol (iWARP), a standard specifying RDMA services over TCP (it is actually based on the RDMA over TCP standard from the RDMA Consortium). The iWARP standard is managed by the Internet Engineering Task Force (IETF). In order to reduce the performance bottleneck caused by the TCP/IP stack in kernel, especially in 10-GE network applications, TCP offload engines have been built, which offload the TCP stack onto the NIC hardware. Still, the actual payload has to be copied from the user send buffer to a DMA send buffer on the NIC, and vice versa in opposite direction. Often even more copies are necessary.

To further increase performance, iWARP introduces a zero-copy RDMA protocol between the host-software and the NIC. iWarp builds strongly on the IB decisions, at least in terms of the user level API. It employs the OFED stack; most API calls can remain the same whether the underlying hardware is actually an IB HCA (Host Channel Adapter) or an iWARP RNIC (Remote DMA NIC - iWARP term for an Ethernet NIC supporting the iWARP protocol stack). Many of the terms and definitions of iWarp are directly taken from the IB specification On the NIC the messages are packed into the Data Direct Protocol (DDP), and then using a specially tweaked TCP are transmitted, usually over 10-GE Ethernet. Like for IB, several other protocols are also available to run on top of iWARP, mainly Sockets Direct Protocol (SDP), iSCSI and SCSI RDMA Protocol (SRP).

The actual RDMA interface of the suite defined in the RDMA Protocol Verbs Specification [31] is a *Verbs* definition very close to the VIA or IB specification. An end-point generally employs one or more QPs and one CQ to communicate with the RNIC. Memory Management necessary for RDMA access is presented in section 5.1.4. The start-up latency reported for an iWARP adapter is 9.78 µs [32].

Another network with a long history in high-performance computing is Myrinet [33]. The current hardware is called Myrinet 10G and supports 10 gigabit link bandwidth both using the Myrinet protocol and the 10-GE protocol. The preferred API to access Myrinet is called Myrinet eXpress (MX) and offers start-up latencies over MPI of about 3.6 µs [32]. Myrinet employs a processor on its adapters as well as additional SRAM chips to run the network protocols and the protocol towards the host. MX does not offer RMA capabilities at the moment but goes to great lengths to support send/receive efficiently including minimizing the host CPU load. The Myrinet network and switch are good examples for a mature high-performance network, but Myrinet is not able to reach really low latencies. Also the adapter architecture is not slim but uses a processor and additional memory chips. Similar in many ways is the current generation of Quadrics hardware [34] except that Quadrics also supports RMA.

The optimal network for a high-performance system combines excellent performance values with a balanced design. A typical characteristic that is often unbalanced in network adapters is the concept of on-loading or off-loading. On-loading means to let all of the protocols be handled by the host CPU, whereas off-loading means to let the NIC handle the complete protocol. To this purpose off-loading NICs are often equipped not only with local processors but also with a significant amount of (S)RAM chips. The EXTOLL architecture has been designed to be a balanced design, offloading protocol parts that can efficiently be

Network	Latency	Bandwidth	NIC Feature	Network Features
10-GE	> 10 µs	1000 MB/s	no offloading, heavy CPU load, no zero copy	no retransmission, poor flow control
Infiniband Mellanox Infinihost III	3-4 µs	1500 MB/s	virtualization, RDMA support, lacking some features	no retransmission
Infiniband Mellanox Connect X	1.2 - 1.8 µs	> 1500 MB/s	same as above IB	same as above IB
iWARP	10 µs	1000 MB/s	virtualization and RDMA support, high latency	same as 10-GE
Myrinet MX	3.7 µs	1000 MB/s	CPU offloading, no RDMA, not optimal latency, complex NIC	retransmission, flow control
Infinipath	1.3 µs	1000 MB/s	not much offloading	same as Infiniband

Tabelle 1-1: Overview of Networks

handled in hardware, and on-loading parts that can more efficiently be handled in software on the host CPU. The architecture is also designed to deal with many of the problems mentioned above. It supports zero-copy protocols, a very low latency, hardware features for retransmission and flow-control, and even the time for memory registration has been optimized. To summarize the discussion of the different networks table 1-1 gives the key characteristics of the individual networks.

1.2 Outline

The remaining work is organized as follows: The next chapter covers the hardware/software interaction and the software design of the ATOLL project, as the lessons learned from this project were fundamental for the EXTOLL project. Chapter 3 analyzes the implications of modern systems on the design of communication devices. This chapter also introduces basic performance numbers to asses design choices. The next chapter covers communication par-

adigms (software) and how they can be mapped to (hardware) devices. Chapter 5 then addresses the problem of address translation. After an analysis of related work and a design space analysis, the novel ATU (Address Translation Unit) architecture is introduced. Chapter 6 describes the design space for device virtualization including a number of studies for a detailed assessment of the different methods of device virtualization considered. After covering these fundamental problems, the actual hardware and implementation of EXTOLL are presented. In chapter 8 the software stack for EXTOLL is described. An evaluation of system performance follows. The results gained and an outlook for future developments of EXTOLL are discussed in the final chapter.

Chapter 2
The ATOLL Software Environment

The ATOLL network constitutes the predecessor of the EXTOLL project. The development of a complete network environment lead to valuable experiences which guided many of the design choices of the EXTOLL architecture. This chapter presents the results from the ATOLL project, especially the software environment which formed the final work-package of the project. The last section of this chapter summarizes the lessons learned for the EXTOLL network.

2.1 The ATOLL-Project

ATOLL was a research project aimed to implement a high-performance interconnect, which can be used as System Area Network (SAN) to build clusters of PCs or workstations. Most notable, ATOLL was a true *Network on a Chip*: four independent network interfaces called Hostports, an 8x8 crossbar switch and four link interfaces were integrated into one single application specific integrated circuit (ASIC). The block diagram of the ATOLL chip is shown in figure 2-1. There is no need for external switching hardware with ATOLL; the four links directly enable 2-d grid topologies of interconnected SMP systems to form a high-performance cluster.

The ATOLL chip features about 5 million transistors running at more than 250 MHz. The ASIC was implemented using a 0.18 µm CMOS (complementary metal–oxide–semiconductor) technology.

Connection to the host system is provided by a PCI-X (Peripheral Control Interface eXtended) interface. Four replicated Hostports allow user-space communication with four distinct processes. From the software side this is the equivalent of a NIC with a total of four simultaneously usable contexts. To copy data from and to main memory each Hostport features TX (transmit) and RX (receive) DMA engines. See [11] for an overview of the ATOLL hardware; [35] and the ATOLL Hardware Reference manual [36] describe the hardware architecture in greater detail. The ATOLL NIC is complemented with a complete software suite to enable efficient usage in cluster environments, which is described in this chapter.

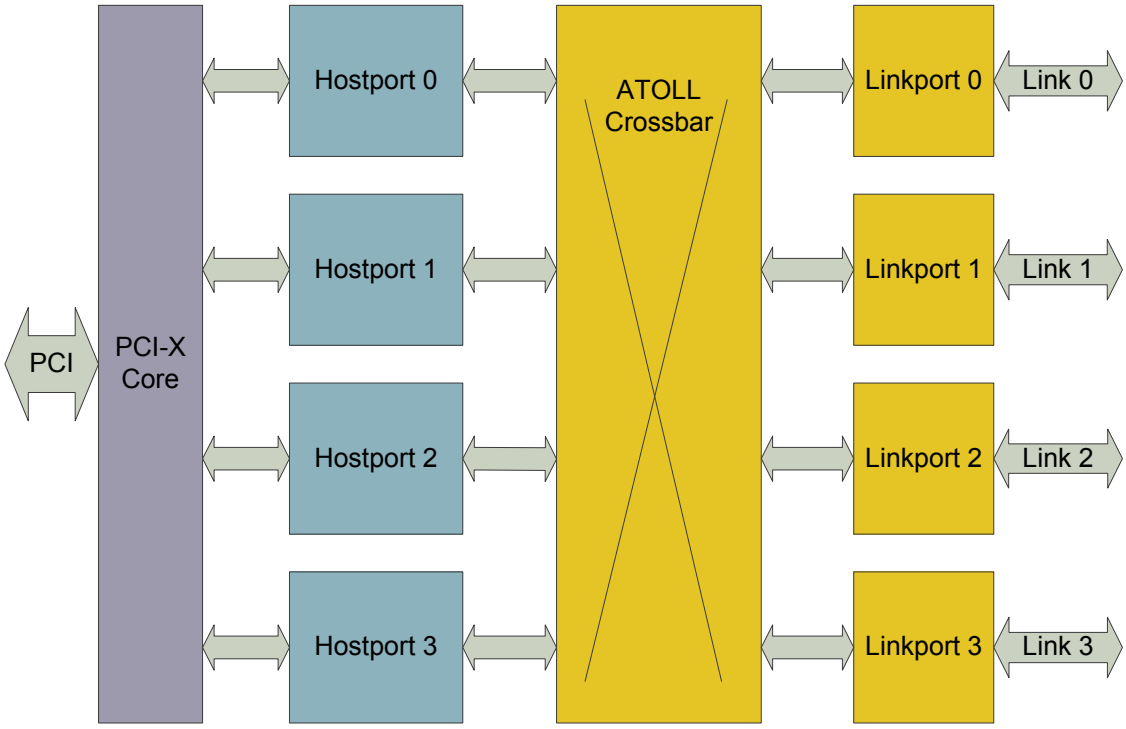


Figure 2-1: ATOLL Block-Diagram

2.1.1 ATOLL Software Environment - Overview

Three core components make up the ATOLL software stack: the user API (application programming interface) library *PALMS*, the Linux kernel driver *atoll_drv* and the ATOLL management daemon (*AtollD)*. In addition, an implementation of the MPI standard (MPICH2 for ATOLL), the *AtollManager* GUI (graphical user interface) management front end and a number of utility, test and benchmarking applications are available. Figure 2-2 shows a graphical overview of the ATOLL software environment. PALMS is a user-space API library that enables applications (or libraries on top of PALMS) to directly interact with the ATOLL network hardware for sending and receiving messages. System resources such as Hostports, DMA memory, register file etc. are managed by the atoll_drv Linux kernel driver. The ATOLL Management daemon implements network management, analysis, topology management, routing, debug functionality, as well as connection management. A typical user-application only links to PALMS or possibly an additional middleware library. The PALMS library interfaces to the kernel driver to allocate resources when first started. It connects with the ATOLL daemon to request virtual communications and trough memory mapped I/O (Input/Output) and DMA memory regions directly with the hardware to insert messages, retrieve messages, poll, update status, and control registers of the designated Hostport.

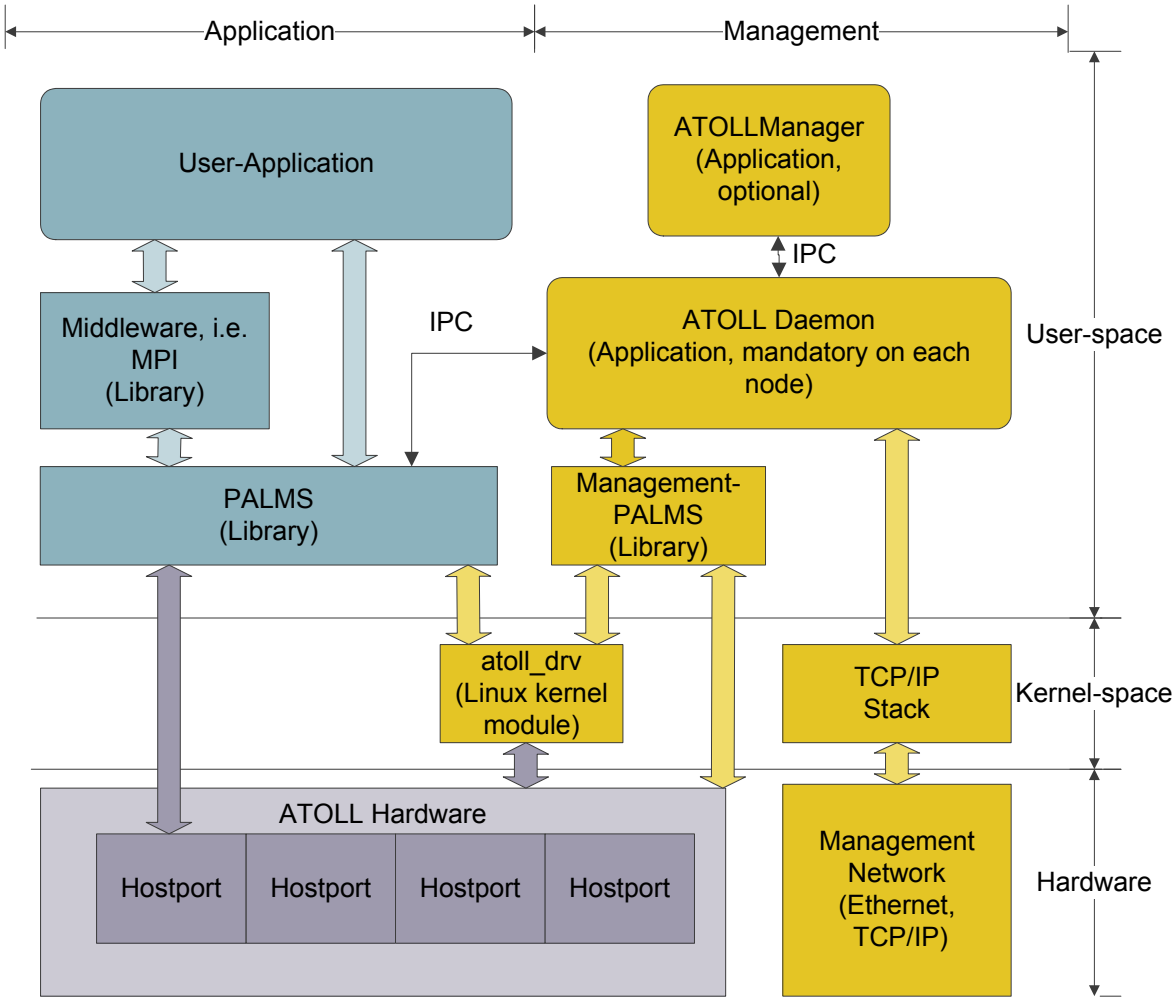


Figure 2-2: ATOLL Software Environment

The AtollManager application attaches to all ATOLL daemon instances of a cluster to monitor and administrate the network. MPICH for ATOLL provides MPI services over ATOLL and is the most commonly used method for message passing applications.

Great care was taken to program all of the software to be portable across different machine architectures. In particular the x86, x86-64 and IA64 processor architectures are supported. Thus the software runs on both 32-bit and 64-bit processor architectures. Besides careful programming the consistent use of the GNU autotools suite [37] enables portable and well-functioning build services across different machine types. All of the software packets also use the *doxygen* [38] packet to provide source-code level documentation.

2.2 PALMS

The PALMS ATOLL Library and Management Software implements a user space API to access the ATOLL network. Two major revisions of the API have been released, 1.4 and 2.1. PALMS offers point-to-point as well as basic multicast services to applications. The communication paradigm used employs *virtual connections*. Connection oriented communication means, that prior to be able to send a message to a peer, a connection to this peer must be established. In PALMS, virtual connections mean, that the connection itself does not directly involve hardware resources of the Hostport such as queues etc. It is purely a handle for subsequent communication operations that identify the peer. On the hardware level *virtual connections* refer to locations of routing strings within the ATOLL routing area. Establishment of a virtual connection always involves the ATOLL daemon to manage the necessary routing string to reach the peer.

The PALMS 1.x implements basic user space send and receive features. PALMS 2 adds the possibility to fill descriptors and messages in place, i.e. splitting the basic send operation into multiple smaller units. Also, completion of messages with the associated freeing of resources is decoupled from initiating communications. Thus, PALMS 2 implements a more asynchronous mode of message transmission than was previously possible with the 1.x versions. Freeing of resources in the context of interaction with an ATOLL Hostport involves access to control registers (read- respectively write-pointer registers), which is a relatively costly operation. The new mechanism allows the aggregation of several free operations into one. This scheme is a perfect example for a *lazy pointer update* algorithm.

These new abilities are especially useful to implement MPI semantics on top of PALMS. Basically, it becomes possible to allocate a descriptor, allocate the necessary DMA buffer for a MPI send request and then fill the different locations one by one. Actually this proved to be rather effective in both simplifying the adaptation of MPICH to ATOLL as well as improving performance.

All in all, several software optimizations have been established to increase the performance of ATOLL message passing from PALMS 1.x to 2. In particular these were:

- Optimized copy routines: Since ATOLL message passing uses 2-copy semantics copy performance is crucial for efficient communication.
- Aggressive use of in-lining: This saves some time by eliminating function calls, for example eliminating the overhead of assembling the atoll_send function from several of the building-block send functions (see below).
- Introduction of a maximum transfer unit (MTU) to facilitate pipelining of message transfers: The ATOLL network hardware has an upper limit of transferable message size between two peers which is equal to the minimum of the send DMA size of the sending Hostport and the receive DMA size of the receiving Hostport. It is advantageous to use a smaller MTU though. With smaller MTUs it becomes possible for the sending process to commit the first part of a large message to the hardware. While the CPU sets the second part up and copies the data to the send buffer, the ATOLL network hardware already transports the first message fragment to the receiver. On the receiving side it is also possible to overlap receiving DMA of the hardware with CPU activity. The result is a sub-

stantial net-gain in bandwidth. Additionally, well behaved applications can send messages of arbitrary sizes since the PALMS layer performs the fragmentation into smaller message parts.
- Optional use of building-block functions: This feature enables more efficient middle ware libraries on top of ATOLL. This feature proved very helpful with the implementation of MPICH on top of PALMS.
- Introduction of fast send and receive functions for very small messages: These messages can for example be used to implement barriers and the like. They are also very useful for the ubiquitous 0-byte latency of benchmarks.

Starting with PALMS 2, the library also became portable and supported 32-bit Intel x86, 64-bit x86-64 (also known as AMD64) and 64-bit Itanium architectures.

In [39] some experimental additions to PALMS are described which add support for communication with end-to-end significance (using an acknowledge based protocol), PIO extensions and advanced multicast support.

2.2.1 Memory Layout of an ATOLL Hostport

To fully understand the functioning of the ATOLL hardware together with the PALMS API, it is important to understand the memory layout of an ATOLL Hostport as it is the fundamental resource and interface used by the software environment.

The Hostport memory footprint, as shown in figure 2-3, is made up of two fundamentally different types of memory: uncacheable mapping of device registers into user application address space and physically contiguous main memory regions. The registers can be divided into four sections, each managing one of the four memory sections of the second type.

Theses regions are allocated in kernel space and are mapped cacheable into the user applications address space. The physically contiguous allocation makes it possible to manage the regions using base-offset addressing, both from the device and from the applications. The physical base registers are set by privileged software; the read- and write-pointer for the regions are accessible by user software. All regions are managed as ring-buffers. The buffer is empty when the pointers have the same value.

For the send DMA area and the send descriptor-table, the write-pointer is incremented by software and the read-pointer is incremented by hardware. For the two receive-regions the ownership is inverted.

To send a message, the payload is copied into the DMA send area first. Software then fills the next entry in the send descriptor-table with values describing the message. By incrementing the send descriptor-table write-pointer, the hardware is triggered to start processing of the message.

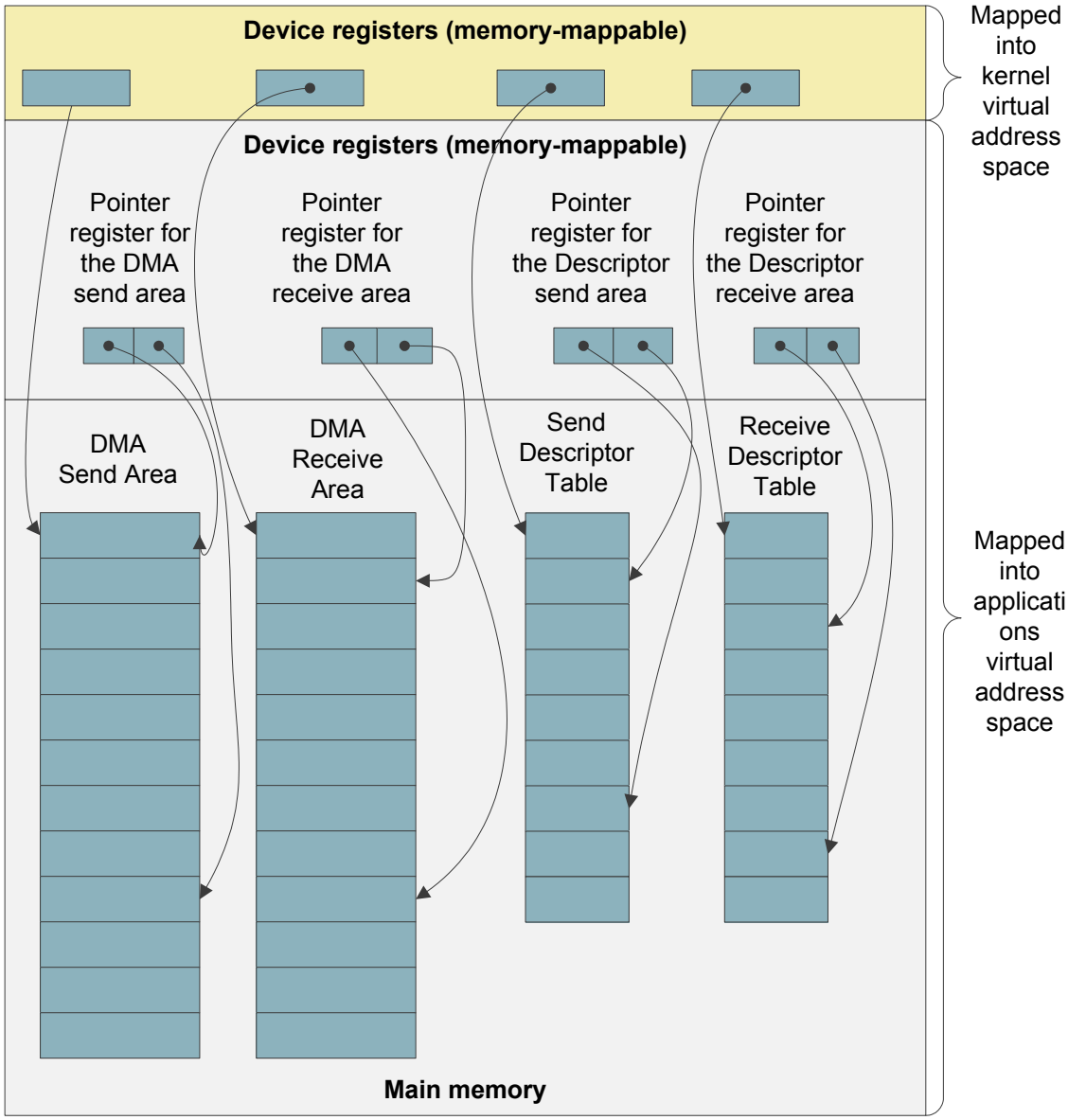


Figure 2-3: ATOLL Hostport Memory Layout

On the receive side, the hardware copies the message payload into the receive buffer using the current write-pointer as starting point. Hardware then inserts a descriptor into the receive descriptor-table and increments the two write-pointers accordingly. Software detects the arrival of a new message by the changed descriptor write-pointer and can start to consume the message.

2.2.2 PALMS Design

Much effort was put into a simple yet efficient communication software interface with PALMS. The complete library can be sub-divided into five different modules. Each module deals with one of the key aspects of communication:

- port Open/Close functionality,
- message receiving functions,
- message sending functions,
- descriptor manipulation functions, and
- utility functions.

Messages are always send in 64-bit words across the network, therefore message sizes are always multiples of 64-bit. Note that this is a hardware restriction. All API functions can also be called with unaligned sizes, i.e. it is possible to send a 3-byte message and receive it into a 3-byte buffer.

However, all functions on the receiver side will return a 64-bit aligned value for the size of data/header segment, so user applications or additional middle ware libraries (such as MPI) have to ensure that the correct size is received, if unaligned sizes are to be supported in the user's environment.

En lieu of execution speed almost no function of PALMS performs tests on pointers specified. It is therefore necessary that user applications ensure that pointers passed on to PALMS are valid.

Internal memory is always allocated and de-allocated by PALMS without user-application intervention. In normal operation it is necessary to provide memory for an *Atoll_Port* variable (which in essence is a pointer variable), for an *Atoll_Handle* variable for every virtual connection to be used, and of course for buffer space. On the receive side it is necessary to have memory space available for tag and source ID (of type *uint32_t* respectively *Atoll_Id*).

To use ATOLL, a Hostport must be reserved for the application and mapped into the application virtual address space. This functionality is performed by the *atoll_open* function. Typically the last thing an application does is to give the Hostport back to the system using the *atoll_close* functions. Both functions directly interact with the ATOLL kernel level driver, and thus are considered slow functions in the sense that they contain a system call and may put the calling process or thread to sleep.

After having allocated a Hostport, an application must initiate virtual connections to all peers it wants to communicate with. The *atoll_connect* function communicates with the ATOLL daemon on the local node to request the connection. The daemon makes sure that the peer is available for communication, and that a valid routing path to the peer is available in the ATOLL routing table, and then returns a suitable routing string offset to the application. PALMS uses the routing string offset as an opaque handle for the virtual connection with this peer. The *atoll_disconnect* function closes a connection.

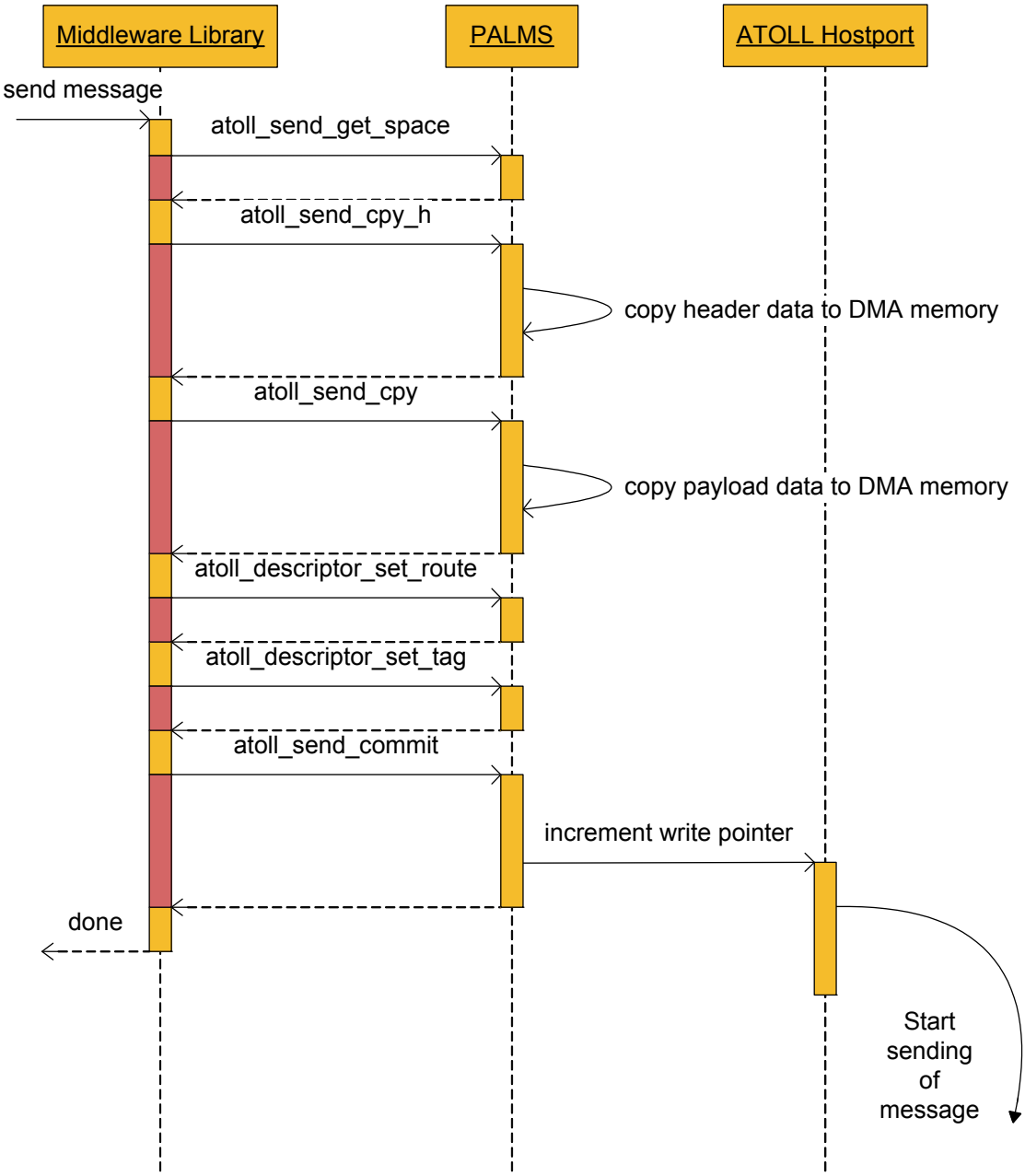


Figure 2-4: ATOLL Send Sequence Diagram

To send messages PALMS provides several functions that can be further subdivided into building-block functions and complete send functions. With building-block send functions sending of a message is performed using a sequence of functions. This allows users or middle ware libraries a great flexibility in the implementation of their send functionality. An example for a common send sequence is provided in figure 2-4. First, space for a send descriptor slot and enough DMA memory for the message are allocated using *atoll_send_malloc*. Using the *atoll_copy_h* message header data is copied into the message

header section. The *atoll_copy* or *atoll_cpyv* (gather copy) function is then called to collect the data to be sent in the payload section of the message. The different ATOLL descriptor manipulating functions are then used to fill in the fields of the ATOLL send descriptor for example the destination or tag. The *atoll_send_commit* function is used to trigger sending of the prepared message.

Complete send functions implement common sequences of building-block functions to simplify programming. Internally they also call the building-block code. Specifically, *atoll_send*, *atoll_multicast* and *atoll_fast_send* are of interest; *atoll_fast_send* sends a very small message using the tag field of the descriptor. Only port handle, peer handle and one 32-bit operand are passed to this function.

On the receive side, again two function groups can be distinguished, building-block and complete receive functions. Building block receive functions include support for such functions as matching of descriptors, for example searching the next message received from a specified peer, copying data or header section to a user buffer, and finally freeing the resource associated with the message within the receive DMA queue. Again, ATOLL message descriptor functions can be used to manipulate the ATOLL descriptor further, for example getting the tag of a received message and the like. Figure 2-5 depicts a typical receive flow. The receive function group includes functions to receive a message blocking, non-blocking or fast. Fast receive is the counterpart to the fast-send operation described above.

As mentioned, PALMS provides a number of functions to manipulate all fields of ATOLL message descriptors, which makes the structure of the ATOLL descriptor completely transparent to the user, yet retains the greatest possible flexibility. Actually, ATOLL descriptor manipulation routines very much resemble the getter/setter methods in object-oriented languages. Finally, PALMS also implements a number of utility functions such as checking available resources, translating error numbers into human-readable strings, setting watermarks for the lazy-pointer update policy employed by PALMS, and querying the ATOLL ID of the local node. Another interesting feature for some applications is the possibility to query the ATOLL IDs of the neighboring nodes. This is especially useful to implement nearest-neighbor or 2-d grid based parallel applications.

2.3 Managing ATOLL

The ATOLL daemon together with several other tools enables smooth operation of an ATOLL network. The ATOLL daemon automatically enables and disables network links (Linkports) if cables are plugged or unplugged. It also automatically recognizes the topology of an ATOLL network and installs routing tables in each node. Additionally the network is constantly monitored in terms of performance counters and to recognize possible errors including message deadlocks which can occur if links fail permanently. The ATOLL daemon has been described in more detail in [40]. The following sections describe the different features of the ATOLL daemon in the context of the complete software environment.

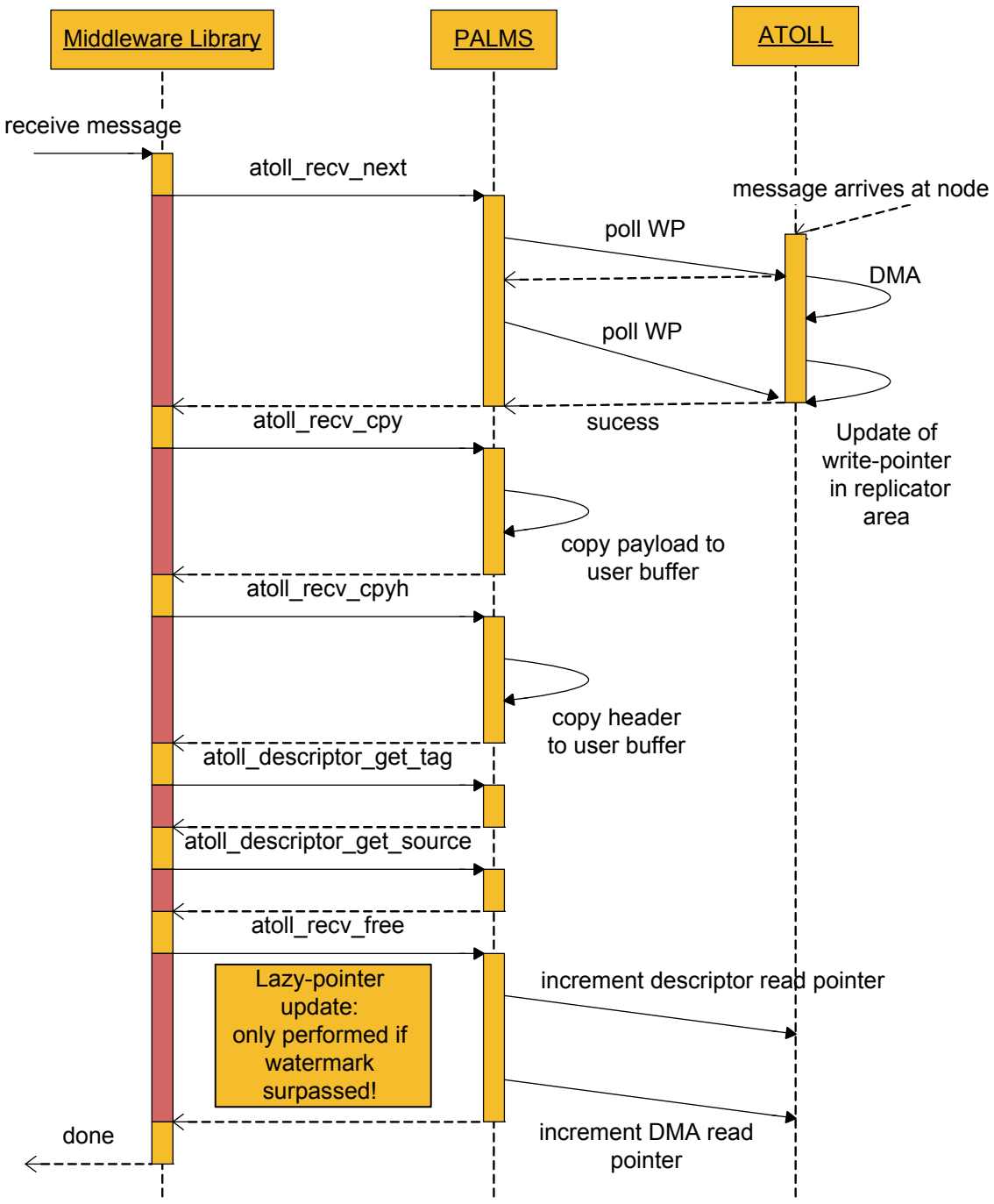


Figure 2-5: ATOLL Receive Sequence Diagram

Automatic topology recognition

The automatic topology recognition algorithm consists of two distinct components. The first is the Linkport management. A software finite-state-machine (FSM) implements a protocol which detects if a link is enabled, active and an active peer on the other side of the link

exists. Within the protocol the ATOLL IDs of the two neighbors are exchanged when the link is brought-up. The new neighbor is inserted into the topology table. If it is torn down again, the entry is removed from the local topology table.

The second component implements the distribution of topology information to the complete cluster. The Linkport protocol enables the daemon to learn about its immediate neighbors. When a daemon connects to a new neighbor, it transmits its complete topology table to it and vice-versa. The tables are then merged on the node, and the resulting table is forwarded to the other neighbors of the node. If a node receives the same update a second time, it discards the update. So, the topology is broadcast throughout the complete cluster and a short time after a link has joined or left the network, the topology information of all nodes is updated. This system provides for a good robustness and fault tolerance since everything is distributed and no single master node is necessary. On the down side, the protocol generates quite a bit of traffic when a cluster is first initialized and many nodes join the cluster in a short period of time.

Routing Table Management

Based on the current topology and the configured routing strategy, the ATOLL daemon calculates a routing table for the ATOLL cluster. This table features one entry for each possible destination; and each entry specifies the source-path from the local node to the destination.

Several routing strategies or algorithms have been implemented for the ATOLL daemon. These are dimension order, Dijkstra's shortest path, West-First and Up*/Down*. Of these dimension-order is the preferred method for mesh setups. Tori are discouraged since they can not be used efficiently without at least two virtual channels. Custom topologies or irregular topologies can be used. In this case it is recommended to either use Up*/Down* routing which is always deadlock free but sub-optimal in terms of average path length and diameter or, if possible, the shortest path algorithm. For hypercubes the shortest path or dimension order can be used.

Performance Monitoring

Atolld monitors the load on all links of the local node via the performance counter registers available in the supervisor register set of ATOLL. One register is updated every clock cycle and is used as a reference time base for the link load measurement. Other registers count the characters send/received on the respective link. The measurement is performed periodically, by default once every second. The numbers read from the registers are then used to calculate relative load of the link and absolute used bandwidth.

Routing Service for Applications

PALMS applications request virtual connections from the ATOLL daemon via IPC (interprocess communication) using UNIX domain sockets. The ATOLL daemon finds the routing entry that specifies the path to the requested destination host. The routing string is inserted into the system wide routing area. This routing area is physically contiguous memory. The base address of this memory region is known to the ATOLL device via a supervisor register. When ATOLL sends a message source path routing is specified within the message

descriptor by a routing offset into this memory area. After inserting the string into the routing area, the daemon hands out the offset of the routing string to the PALMS application. If the topology changes the routing table is also updated. Existing handed-out routings that need to be changed are invalidated. User-applications receive an error with their next send call and need to re-request a routing to the destination node. If an application exits or calls the disconnect function the routing string is freed from the routing area. Notice that the size of the system wide routing area limits the number of virtual connections that can be active at any given time. This is the reason why the area is shared by all Hostports.

Monitoring Interface

The ATOLL daemon exports a management and monitoring interface to client-management-applications via a TCP socket. Applications can connect to this socket. An ASCII based protocol is used to query information from the ATOLL daemon. Additionally the protocol supports logfile forwarding and a management mode enabling remote manipulation of ATOLL features.

Debugging Services

The monitoring interface can also be used to request debug services from an ATOLL daemon. This includes images of system memory regions including Hostport DMA areas, descriptor ring buffers and the routing area.

The Linkports and the crossbar are also continuously monitored. Ports that are currently unavailable are masked in the crossbar requester, effectively blocking the port. This applies both to Host- and Linkports. If a message requests a blocked port, an interrupt is raised which triggers the ATOLL daemon to pull the message out of the crossbar and pushing it into an internal message queue. The error is logged and, via log forwarding, forwarded to monitoring clients. This feature proved to be very effective for software development. Prior to this, an error in an application could easily lock the whole network. For example if one of the peers of a distributed application crashed due to a segmentation fault, the other peer would still send messages to this application which would eventually block network resources without ever freeing them. The blocking feature together with the routing services also enables synchronization between peers of a distributed application in the sense that one peer can only send a message to another peer after the receiving Hostport has been set-up correctly again helping to make applications behave correctly. Finally, the pulling of blocked or broken messages together with the ability to dump them to the log enables analysis of possible network failures and proved valuable during ATOLL bring-up and validation (see also section 2.5).

Sensor and Serial EEPROM

The ATOLL adapter card also features an EEPROM (Electrically Erasable Programmable Read-Only Memory) device for serial number storage etc. which is accessible through an I^2C (Inter-Integrated Circuit) bus connected to two of the general purpose I/O pins (GPIOs) of the ATOLL IC. Additionally a thermal sensor is accessible over this bus which can be used to monitor the temperature of the ATOLL NIC. In [41] a software I^2C stack suitable

for the ATOLL GPIO (general purpose I/O) is described. This I^2C stack has been integrated into ATOLL daemon and is used to access the thermal sensor periodically. If an over-temperature is detected, a warning can be emitted.

2.3.1 AtollManager

For easy and efficient usage, a GUI is used to present the user the most relevant information about the ATOLL network, including:

- a list of hosts,
- link connections per host,
- the topology (with graphical representation),
- routing information (with graphical representation),
- network load indicators,
- host (CPU-) load indicators, and
- Hostports in use.

At startup, the user must provide the hostname of one of the nodes of the cluster. The AtollManager then connects to the management port of this node via the TCP management interface of the ATOLL Daemon. The user must also provide a username and password to authenticate to the ATOLL daemon. The management application then automatically retrieves the members of the cluster and its topology from the initially connected node. More detailed state information including CPU and network load information of individual nodes is retrieved by connecting directly to the respective node and requesting the necessary information.

To control and debug the network, redirection of messages from the logs of all of the daemons of the cluster is possible, a feature called *log aggregation* which is a very powerful tool for administrators to obtain an overview of the status of the whole cluster. Within a single window, the user sees all relevant information about the state of the ATOLL network for example the topology with the connected links is displayed. The administrator can interact with the display to re-arrange the individual nodes and select one node for detailed state information in the left-hand properties pane (figure 2-6).

If interaction of the user is required for example in the case of failures, the user can request to enter maintenance mode. This mode enables direct control of advanced hardware features. In maintenance mode it is possible to trigger resets of individual parts like Hostports, Linkports or the crossbar of ATOLL NICs remotely. It is also possible to reset the ATOLL daemons and force re-initialization and shutdown of a part of or the whole network. To this purpose it is also possible to activate or deactivate selected links. Again, all of these activities are exported from the ATOLL daemon via its TCP based management interface

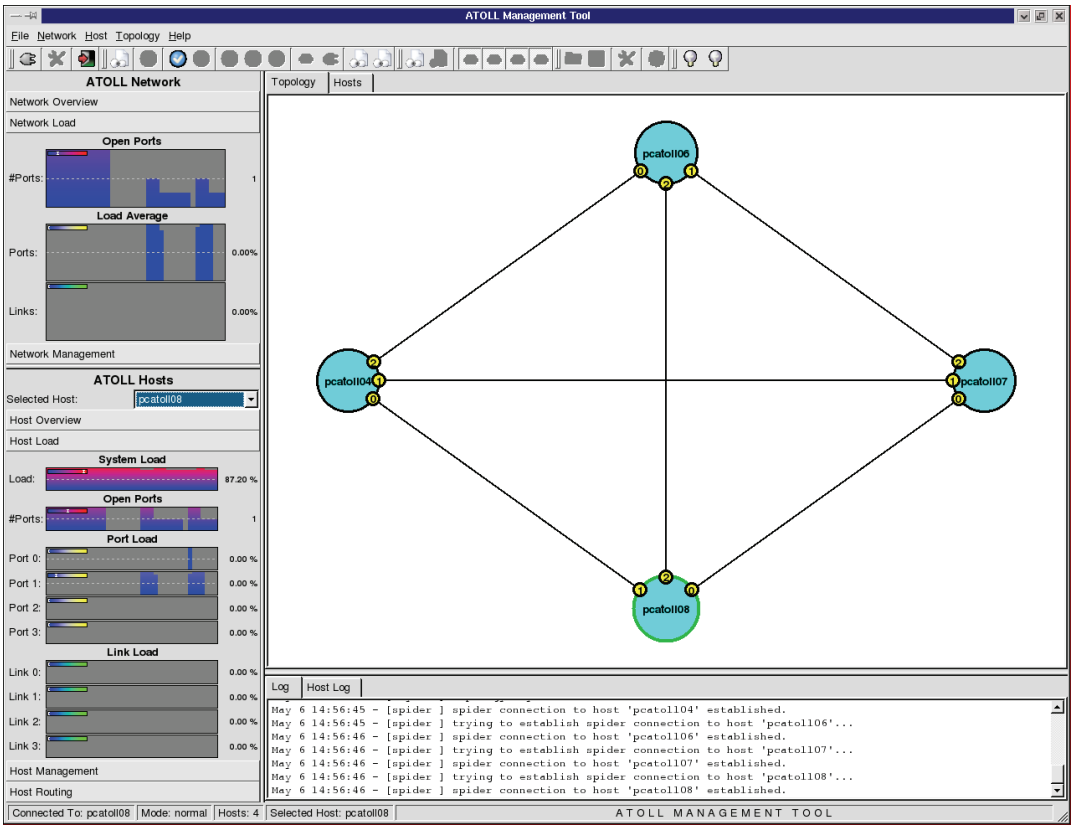


Figure 2-6: AtollManager

2.3.2 Additional Tools

To complete the basic software environment several additional tools have been developed. There is a whole suite of command line driven management tools available, as well as different network test programs and the *atollrun* distributed execution environment.

Command Line Management Tools

A number of command line management tools are available to interact with the ATOLL daemon using the same TCP based management interface of the ATOLL daemon as the graphical ATOLLManager application. Using one tool, it is possible to request re-initialization of the different hardware parts of an ATOLL chip. This is especially useful when developing/debugging, since it allows the developer to put the hardware in a known state. Other tools are available to activate or deactivate certain links, restart the ATOLL daemon, print status information or dump the topology and the routing table.

Command Line Test Tools

A number of command line test utilities have been developed to test and benchmark ping-pong latency, bandwidth, and multicast traffic. A sink tool is available, which receives all incoming messages and discards them. Some of these tools were used to verify the PALMS level; others serve as microbenchmarks on the PALMS level.

Atollrun

The *atollrun* utility allows remote invocation of distributed applications on an ATOLL cluster, similar to the *mpirun* or *mpiexec* tools of MPI. The tool is written in Python. Via SSH (Secure Shell) connection, PALMS based SPMD (single program - multiple data) applications can be invoked on remote hosts. A list of machines that should each execute a copy of the application is passed to *atollrun* (like a MPI *machinefile*). The standard output and error streams of the applications are redirected via SSH to the starting *atollrun* instance enabling easy monitoring of jobs (output aggregation). Signals like SIGTERM (commonly caused by pressing Ctrl-C on the console) are forwarded to the remote application, so that all of the processes of the invocation behave similarly to a single job. Basically, atollrun completely fills the role of mpirun for applications based directly on PALMS.

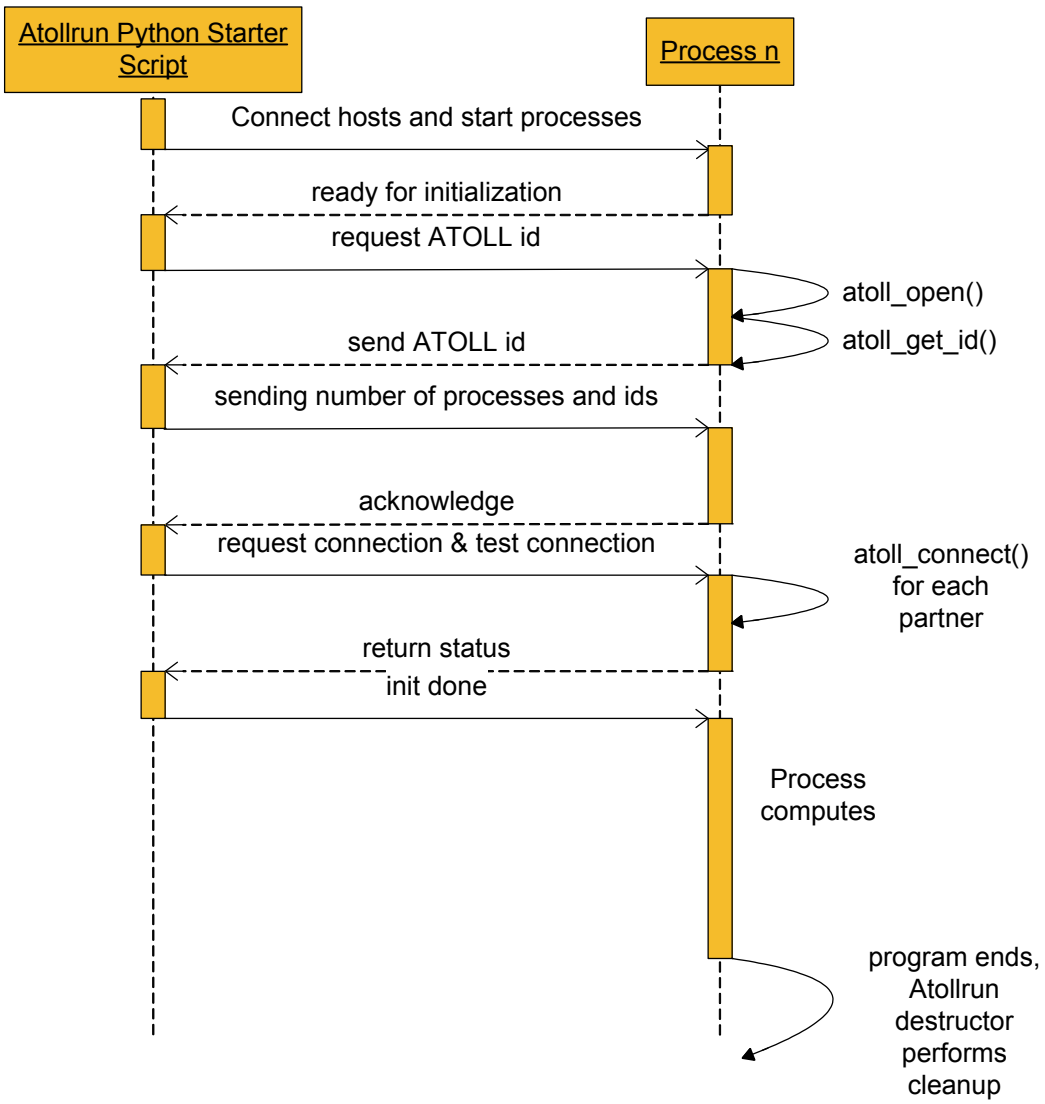


Figure 2-7: Atollrun Initialization Protocol

The atollrun package consists of a Python script used to run the SPMD application and the atollrun library that is linked to the applications. Within the application, the libraries provides a C++ *Atollrun* class respectively C function wrapper for it, which are used to initialize the SPMD application. This includes synchronization at startup, exchange of IDs, connection establishment, and communication of rank and size of the application. This mimics the behavior that MPI implementations typically perform in their *MPI_init()*, *MPI_rank()* and *MPI_comm_rank()* functions. The Python script and the application communicate with each other using a simple ASCII based protocol. Figure 2-7 shows the typical initialization phase for a 2-process SPMD application using atollrun. From the API side, the Atollrun class provides three methods to configure atollrun prior to the actual initialization. It is possible to specify if ATOLL connections are established automatically and if a communication test is performed internally. The initialization function gets the commandline passed and performs the above described initialization protocol. The utility functions *getCommRank()*, *getCommSize()*, *getPort()*, and *getHandle()* have to be used by the application to query these parameters which are not directly accessible anymore since the PALMS function calls that actually returns these parameters are already performed within the Atollrun initialization. The destructor of the Atollrun class performs the clean-up, the job typically performed by *MPI_finalize()* in MPI. For programs not written in C++ a C wrapper is available that makes all the functionality available in plain C.

The actual *atollrun* Python script accepts several parameters on the commandline. First, a machinefile defines the set of machines available to the application. The *-p* option defines the number of processes started on these machines. The groupfile option defines sets of machines having a single login for the user. The user has to provide his credentials (i.e. username and login for remote access) only once for each group of machines. Instead of connecting via SSH, it is also possible to connect using the (unsafe) RSH (Remote Shell) program. The last parameters on an *atollrun* commandline are the name of the PALMS application to be executed and potentially additional commandline parameters destined for this application.

2.4 MPICH2 for ATOLL

MPICH2 [42], the successor of MPICH, is a very popular and free implementation of the MPI standard. MPICH2-PALMS is the adaptation of MPICH2, which interfaces to the PALMS software and thus allows MPI programs to communicate via the ATOLL network. This implementation adds an ATOLL *ch3* (channel device) interface to the MPICH2 library. MPICH2 consists of three main layers: *MPI*, *ADI* (abstract device interface) and *ch3*. The MPI layer is very thin. It implements MPI parameter checks and polls on requests and triggers the process engine. The ADI layer is located beneath the MPI layer. It manages the different data types and protocols used by MPICH2. For the actual communication, the *ch3* layer is used. The channel devices are then responsible for the communication including connection set-up and shutdown.

The *ch3* interface describes a non-blocking, point-to-point communication semantics. It consists of a number of functions and macros, which can be grouped into send/recv functionality, request handling, the progress engine, and initialize/finalize functionality. The ATOLL *ch3*-device is implemented on the basis of the generic shared memory device. The ATOLL *ch3* device then uses sub-devices depending on the destination of a message. Messages on the same node are transported using the shared memory sub-device, messages to remote processes are transported using the ATOLL sub-device which uses PALMS to interact with the ATOLL hardware. Figure 2-8 shows the layers of ATOLL MPICH2. The blocks with yellow background denote ATOLL *ch3* specific code.

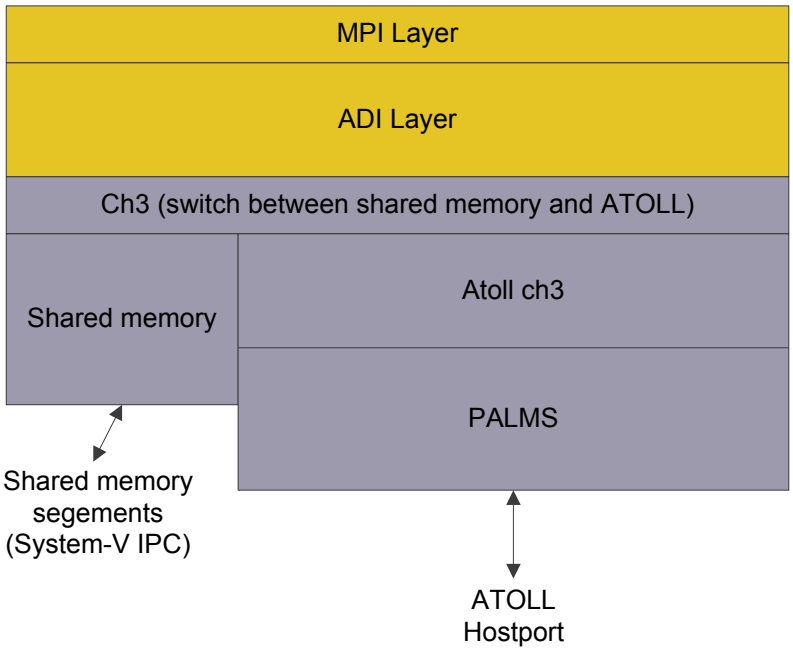


Figure 2-8: MPICH2 *CH3* Device with Sub-devices

Intra-node communication is performed using the original shared memory code thus avoiding the rollback path via PCI-ATOLL-PCI and improving local performance. The shmem code already features a general structure that is very close to the structure necessary for PALMS integration. The buffer layout for the shared memory protocol as well as the message format used by MPICH2 is shown in figure 2-9.The structure was also defined so that it is possible to add another sub-device later for example to support RDMA functionality of a NIC. The top-level of *ch3* thus is implemented as a switch which calls the corresponding function of the actual sub-device that is responsible.

The intra-node shared memory device is implemented using standard System-V shared memory segments. The basic idea is that each process allocates one shared buffer for each communication partner. These buffers are used as receive queue. The corresponding partner maps the receive queue into its virtual address space and uses it as send queue. The sending process updates the write-pointer and the receiving process updates the read-pointer. The

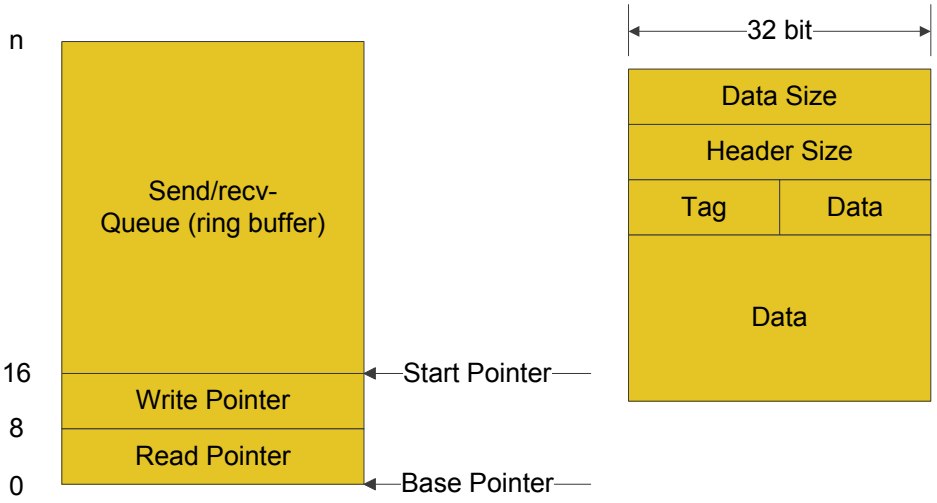


Figure 2-9: MPICH2 Shared Memory Buffer and Message Format

protocol is thus based on a classic ring-buffer and involves two memory copies, one to place the actual message data into the shared buffer, and the other to move the data into the application buffer on the receive side.

The ATOLL sub device uses the send area of a Hostport as sending queue and the receive area as receive queue, effectively changing just enough of the buffer structure to efficiently support the physical NIC. The message structure itself is maintained as in the physical shared memory sub-device.

MPICH2 uses virtual connections to describe the path between two end-points in the network. A channel device implementation can extend this structure to reflect network specific, additional data and state for a connection between two end-points. Unfortunately, MPICH2 copies the complete structure when a new MPI communicator is created, which can cause problems in the channel device which is not informed of such a (high-level) operation. So, the ATOLL implementation holds a local structure describing the low level data and the state associated with a virtual connection. Only a pointer to this structure is added to the MPICH2 virtual connection structure. The virtual connection of a message is coded within in the header of a PALMS message and can thus be decoded at the receiving end-point.

One other very important data structure used in the *ch3* device is the MPI request, which describes the state a high-level MPI operation is in at a given time. The progress engine tries to make progress on currently outstanding requests. MPICH2 generally attempts to send or receive messages immediately if certain constraints can be met, i.e. the message is smaller than the MTU and enough space is available in the send buffer region. If it is not possible to process an MPI operation immediately, often called *eager*, a MPI request is generated and inserted into the corresponding request queues.

The progress function is called from the upper layer to make progress in receiving messages from the ATOLL device. The function checks for the next ATOLL message and calls the respective receive function if new data is available. The send progress function, advances the progress of MPI send requests. While sending of small messages does not really need a progress function (the message is sent *eager*), larger send requests need to be fragmented into smaller ATOLL level messages and these fragments are sent sequentially. The *ch3* progress functions make intensive use of the new PALMS 2 building block functions to selectively first receive the header of a message, then copy the data part to a scatter-gather list of buffers and finally free the DMA memory.

As a future enhancement, the *ch3* design could also support a hypothetical RDMA enabled ATOLL. Based on the structures already in place an RDMA sub-device can be implemented. Instead of mapping the buffer into the sending processes address space and copying the data from the application buffer into the shared queue as by the shared memory sub-device, a RDMA *put* operation of the NIC would be used. This is done to implement send/receive semantics on top of a shared memory/RDMA hardware. While the usage of MPICH2 for EXTOLL designs has been abandoned in favor of OpenMPI [43], the basic idea of implementing send/receive functions on this principle has remained the same.

The ATOLL MPICH2 implementation also has some limitations in regard to the full MPI-2 specification. Specifically, dynamic process management is not supported and true RDMA support is missing. Implementing for MPICH2 has been partly challenging because the interfaces were not as clear and good documented as it would be desirable. While it is said that the design of MPICH2 is a huge improvement over MPICH1 it still suffers from extensive usage of C macros, obscure functions and data structure dependencies. The complete channel device is about 7500 lines of C code (without include-files but including comments) and thus of moderate complexity.

2.5 Debugging the ATOLL ASIC

Bring-up and debugging of a complex hardware/software system is always challenging. In particular, hardware bugs are sometimes difficult to identify and sometimes impossible to remedy without a re-run of the ASIC production. In addition, software bugs or deficiencies can interact with a hardware phenomenon producing completely unexpected results. Using DMA directly from user-space and kernel-level device access can easily lead to machine crashes and lockups making error analysis even more difficult since no access to the machine is possible anymore neither through a debugger nor the console or a logfile.

To bring up and verify the ATOLL chip in real systems a number of software components and test methods were developed and utilized. These methods also lead to the discovery of several bugs in the ATOLL silicon.

Relatively early on it was discovered that operation of ATOLL in PCI-X 100 MHz mode lead to frequent lock-ups of the complete machine. In PCI-X 66 MHz mode lock-ups were less frequent; if only one Hostport is in use they could be completely avoided. Using an advanced Tektronix logic state analyzer with PCI analysis software together with carefully

tuned test software on the host systems finally revealed a problem with the PCI-X flow-control feature. When the PCI-X host bridge cannot accept any more data during a long burst transfer, it signals a “stop at next address boundary (ADB)” condition to the device. The device has to stop transmission at the next 128-byte aligned address. If this condition happens very close to the next ADB, the PCI-X IP core employed in ATOLL ignores the stop signal for the next ADB. The bridge receives data although it is not able to accept it and the machine crashes. After this behavior was discovered the bug could also be provoked within the ATOLL HDL (hardware description language) simulation environment. So finally it was discovered that if the stop occurs to close to an ADB the core wrongfully accounts this stop for the *next* ADB, but the bridge wants the stop for *this* ADB. A patch to fix the bug in the PCI-X core was prepared and inserted into the ATOLL HDL model.

Another problem emerged when running in PCI mode. In this case the observed performance of ATOLL was very low. The reason for this behavior was a wrong PCI command used for DMA read transfers. Instead of issuing a read burst, the DMA read transfer was executed as a sequence of individual reads. This bug was also fixed in the HDL.

Once a larger cluster with ATOLL became available (Karibik Cluster), a link-layer bug was found. Initially when running applications involving more than 4 nodes, sometimes a lock-up of the network was detected. It was in some cases possible to again use the network if all relevant Linkports and crossbars where reset. After further analysis a test program could be developed that only involved two active nodes (i.e. hosts which are actively involved) as well as at least one intermediate node. This program was able to cause the error with a relatively simple message pattern. The path *between* those active nodes had to be multi-hop, though. The network lock-up could be deterministically reproduced with this setup. The debugging features of the ATOLL crossbar were used to analyze the raw data that arrived at the destination node. A special command-line tool was written to dump the raw link-layer phits (physical units) into a text file on disk.

The dumped file was textually analyzed and a protocol violation was found: under certain conditions the reverse flow-control misbehaved. It can happen that a stop character, a continue character and a second stop character are requested to be transmitted. In this case due to the priority logic in the Linkport the order of the second stop character and the continue character is exchanged. On the link the sequence “STOP,...,STOP,...,CONTINUE,...,CONTINUE,...” can be seen instead of “STOP,...,CONTINUE,...,STOP,..., CONTINUE”. The result is that the link resumes transmission of data after the first continue, although logically, it should do so only after the second continue is received. Thus, buffers are partly overwritten and data is corrupted.

Within the ATOLL HDL code it was discovered that the hysteresis for the stop/continue generating circuit had been removed at some point in the design. This can cause a cycle-to-cycle occurrence of STOP/CONTINUE characters enabling the occurrence of the above problem.

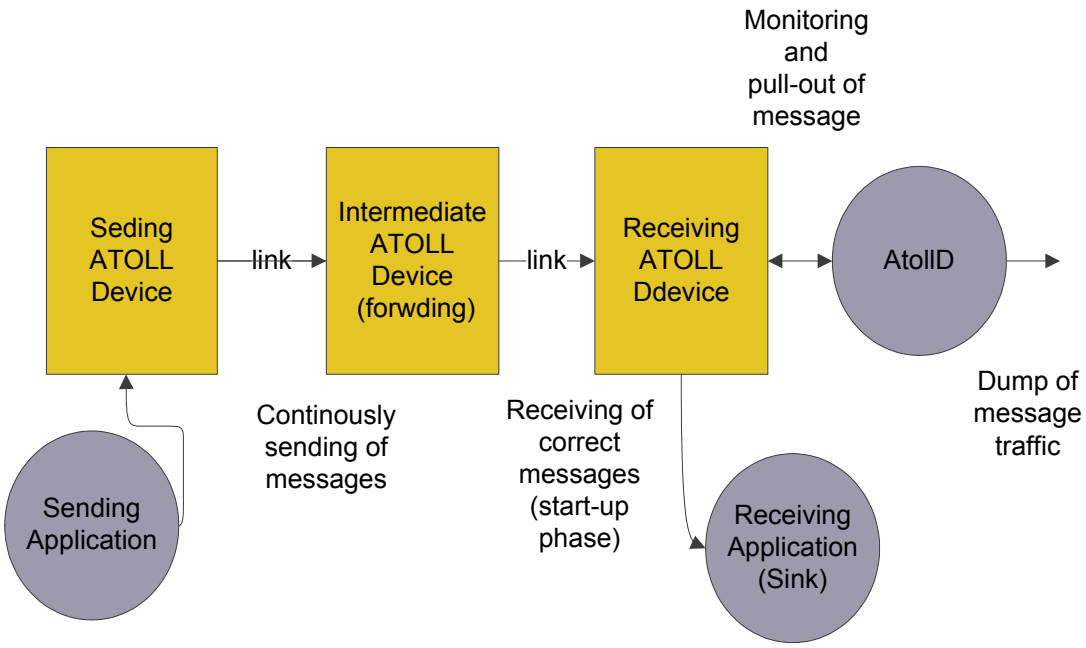


Figure 2-10: ATOLL Link Test Scenario

This problem can be fixed with an HDL change. During the ATOLL test and analysis phase an external solution was also tested involving a small FPGA board. This board was also used to test an SFP Small Form-factor Pluggable (SFP) based optical link layer for ATOLL and the bring-up of the OASE chip [44].

The link-cable FPGA board as depicted in figure 2-11 plugs into an ATOLL link connector on the one side, on the other side a standard ATOLL link cable connects to another node. All traffic is passed through the Xilinx Virtex II Pro FPGA where the stream is analyzed and the protocol error from the above described bug is removed using a small state machine.

The link-correction logic within the FPGA runs at full link speed (250 MHz) and only adds 4 pipeline stages to the link path. The FSM uses five states and is shown in figure 2-12.

The XC2VP4 FPGA also features multi-gigabit transceivers called RocketIO which can be used to serialize/deserialize data streams. As such the cable FPGA board is a prototype for an ATOLL serializer/deserializer module. The OASE chip is an ASIC solution that was planned to fill this role. An SFP module was placed on the FPGA board to enable direct transmission and reception of serial data. An FPGA design called Optical-test-bed was designed which enabled testing of both SFP based serial communication as well as testing of the OASE chip. For testing the OASE board is connected to the FPGA board using an ATOLL cable. Over this cable a parallel data-stream, either already 8b/10b coded or not, is transmitted to the OASE chip which then operates as a serializer and outputs the serial stream to the SFP connector on the OASE board, or directly drives a laser diode. The serial data stream can then be received with the SFP on the FPGA board and the correctness of the

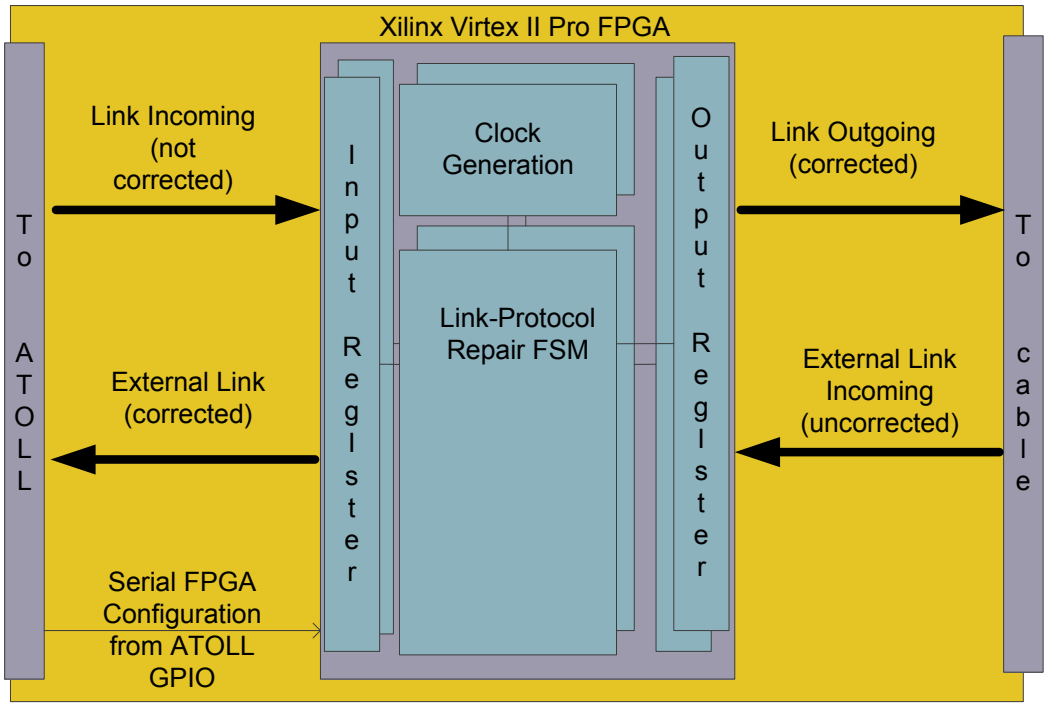


Figure 2-11: ATOLL Link-Cable FPGA Board

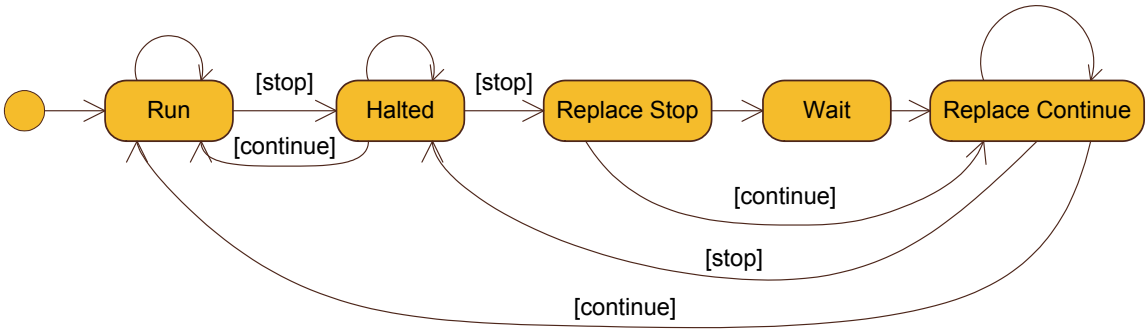


Figure 2-12: Link Correction FSM

transmission is checked. Reception using OASE was not tested since the OASE3 revision had several errata preventing successful use as a receiver and deserializer device. A newer OASE revision will be tested with the Virtex4 based HTX-Board.

2.6 Performance of ATOLL

The performance of the ATOLL network was analyzed in a number of ways. First, several microbenchmarks were used to characterize the basic communication parameters as latency and bandwidth. Microbenchmark results were also reported in [12]. Then, standard application level benchmarks were used to show the performance of the ATOLL network when executing real parallel applications. The performance data presented here were obtained on

the first revision of ATOLL and a new revision of ATOLL should show higher performance data because of a fixed PCI-X block. At the International Super Computer Conference 2004 an eight node ATOLL cluster was shown running the UG application [45]. In addition to real-world analysis, the SWORDFISH (Simple Wormhole Routing and Fault Injection on Simulated Hardware) simulator [46] was developed to enable modeling and analysis of medium to large ATOLL networks.

2.6.1 Microbenchmarks

Microbenchmarks have been performed to measure latency and bandwidth of the ATOLL communication system. Latency is measured as half-round-trip latency using ping-pong tests, both proprietary code based on PALMS, as well as NetPIPE [47]. To this end, NetPIPE was used on top of MPICH2. Additionally, the NetPIPE communication routines were ported to work directly on top of PALMS. In the ping-pong test, one node, the master node, sent a message to the slave node. Once the slave node has received the message it answers with a message to the master node. The master node then receives this message and the iteration of the ping-pong test is completed. Time is only measured on the master node. The half-round-trip latency is defined as half of the time measured. On low-latency systems such as ATOLL, the timing of the used time measurement facility of the operating system (*gettimeofday*) is not accurate enough to measure the latency of a single ping-pong iteration. Additionally, all time measurements are subject to system noise, meaning time variation due to periodic system book keeping tasks (triggered by the timer interrupt), scheduling, and other hardware interrupts. So, it is necessary to repeat the ping-pong experiment appropriately depending on latency to measure valid results. NetPIPE performs a dynamic algorithm to use a large enough trial set. The proprietary ping-pong method uses a trial set that is sufficiently large, larger than NetPIPE's set actually. This gives accurate results, but benchmark runtimes are adversely affected.

The second microbenchmark method used is called streaming. In this benchmark, the master node sends a continuous stream of messages of a given size to the receiver. After the network has ramped up, the measurement takes place. The time is measured on the master node for the transmission of a fixed number of messages. The number of the transmitted messages multiplied by message size gives the streaming bandwidth or point-to-point bandwidth. Actually, if the measurement is performed correctly this is the maximum payload bandwidth of the network in steady state which is also called effective point-to-point bandwidth. The point-to-point measurements have also been repeated with several intermediate nodes, which only served as switches, thus making it possible to quantify the effects of several stages of ATOLL switches onto the measured parameters.

In addition the Intel MPI benchmarks [48] and the Special Karlsruher MPI Benchmark (SKaMPI) [49] were also used. These are MPI based microbenchmarks that test most of the MPI provided functions. The individual benchmark results are omitted here, since they do not give additional insight. But it is worth to note that these applications proved useful in testing and debugging the MPICH implementation.

The measurement results were performed on Pentium 4 Xeon machines. The CPUs run at 2.8 GHz clock rate and used a Serverworks GC-LE chipset and 1 GB of RAM. The ATOLL cards used were plugged into a dedicated PCI-X-100 slot. SuSE Linux version 9.1 with a standard Linux kernel version 2.4.25 was used. The ATOLL was programmed to run at 300 MHz core frequency for these measurements.

The startup latency of ATOLL is 3.3 µs. Figure 2-13 shows the ping-pong latencies for different message sizes up to 0.5 MB. The small graph in the upper half shows a zoomed-in version for small message sizes. In addition to the average latency, the minimum and maximum latencies are also plotted showing the maximum deviation. The last curve shows the impact of traversing 7 hops instead of just one. With additional intermediate hops, a start-up latency of 4 µs could be reached. With seven hops all nodes in a 64 node ATOLL cluster can be reached.

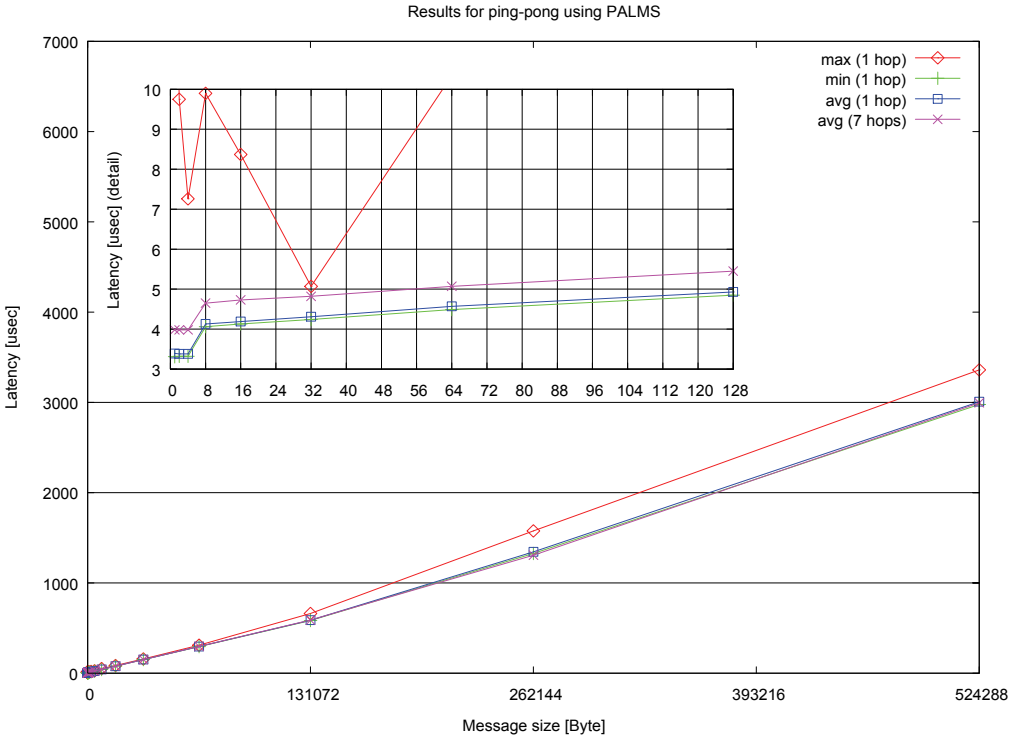


Figure 2-13: ATOLL Ping-Pong Latencies [12]

The streaming bandwidth test resulted in a measured maximum of 269 MB/s. The peak bandwidth is reached at transfer sizes of 32 kB. It is especially noteworthy that the $n1/2$ bandwidth is already reached at 512 byte-sized transfers and 90 % of the peak bandwidth are available with 4 kB transfers. The complete streaming bandwidth results are depicted in Figure 2-14.

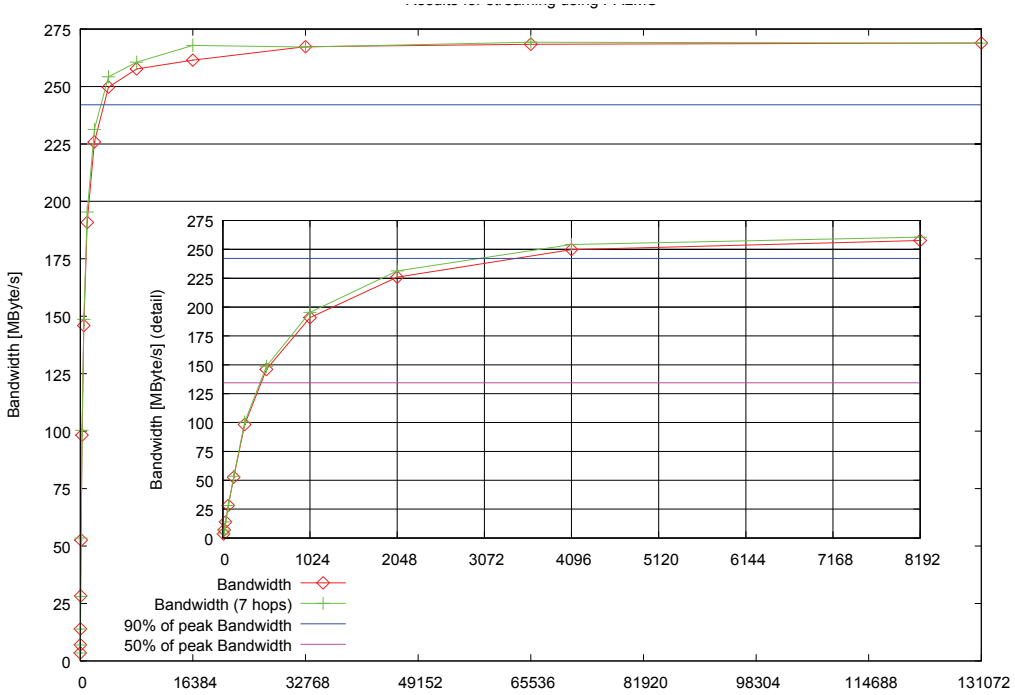


Figure 2-14: ATOLL Streaming Bandwidth [12]

2.6.2 Application Level Benchmarks

To analyze the performance of ATOLL in addition to the pure synthetical microbenchmarks presented above, the High-Performance LinPACK (HPL) benchmark [50] was used. HPL is the benchmark from which the semiannual TOP500 list of the fastest super computers of the world is derived.

2.6.3 High-Performance LinPACK

HPL is an implementation of the LinPACK benchmark for distributed memory computers. The LinPACK benchmarks solves a dense linear matrix problem $Ax=b$ using 64-bit floating point (double precision) arithmetic. The problem size is given by the dimension of the (quadratic) matrix, i.e. problem size 1000 corresponds with a matrix size of 1000^2 which uses approximately 8 MB of memory. The algorithm used to solve the linear equation system is based on LU decomposition with partial pivoting. The result of the benchmark is given in floating-point operations per second. The HPL package is written in C and uses BLAS (Basic Linear Algebra Subprograms) and MPI for linear math respectively communication. The BLAS routines from the AMD Math library were used and again MPICH2 served as communication library. The benchmark was run with ATOLL and Gigabit-Ethernet as communication device on the Karibik cluster[1].

1. 8 dual-processor Opteron 248 (2.2 GHz), 4 GB RAM.

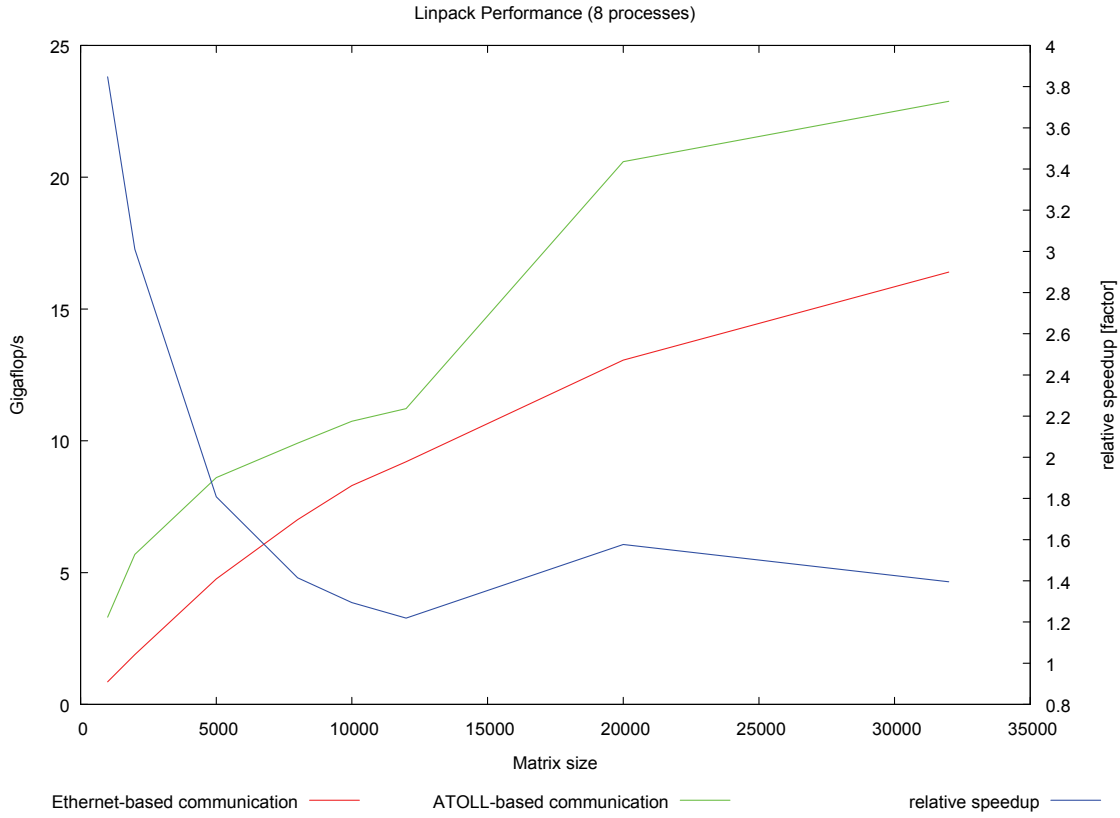
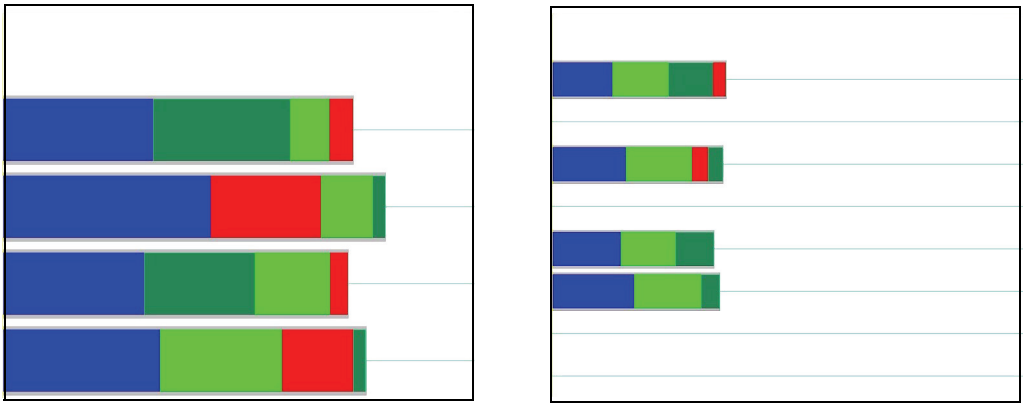


Figure 2-15: High-Performance LinPACK

Figure 2-15 shows the result of the measurements for different matrix sizes up to 32.000^2 (8 GB matrix size). In addition to the Gigaflop/s reached, the blue curve plots the relative speedup gained by using ATOLL. This was performed with the first version of ATOLL, and thus ATOLL was only operating in PCI mode and register access was not optimal. With a completely fixed ATOLL version performance may have increased.

At the largest matrix size measured, ATOLL performed around 30 % faster better than GE (Gigabit Ethernet). This is quite a difference, since HPL does not measure primarily communication performance but also (and to a large extent) the performance of the CPU and the memory subsystem.

To gain some more insights into the reasons for the performance advantage of ATOLL, MPICH2 was instructed to instrument the binary and log all MPI function calls and timing into a file. After completion of the run the logfile was analyzed using the Jumpshot visualization tool. Using a timeline and histogram plot of the MPI logfile it becomes quickly clear that the MPI library spent a lot less time in the relevant MPI functions used, namely *send*, *receive* and *sendrecv,* when ATOLL was used. Both, the higher bandwidth of ATOLL and the by far lesser latency causes this effect.



(a) Gigabit Ethernet (b) ATOLL

Figure 2-16: Histogram of MPICH Function Time Use

The different times spent within the MPICH functions can bee seen in a histogram plot (Figure 2-16). The left histogram shows the result from a Gigabit Ethernet run, on the right side an ATOLL run is shown. Each horizontal plot corresponds to one process. The plot on the right has four unused process slots. In the vertical direction the time the process spent for different functions is plotted: blue is *send*, dark green is *sendrecv*, light green is *recv*, red is I/O (i.e. kernel time) and the dark-green on the far right of the GE plot is *iprobe*. The rest of each timeline represents the relative time the process spent inside the application code, i.e. for actual computation.

The theoretical peak LinPACK performance of one Opteron processor is the product of the number of floating-point-operations per cycle finished by the floating-point pipeline and the cycle rate. The Opteron processor can retire up to 2 floating point operations per cycle. At 2.2 GHz one processor of the karibik cluster can perform at a theoretical maximum of 4.4 GFlop/s. Thus 8 processors (as used for the measurements for Figure 2-15) provide up to 35.2 GFlop/s. The GE implementation with a measured peak execution rate of 16.4 GFlop/s thus shows an efficiency of about 47 % whereas ATOLL showed an efficiency of 65 %. The TOP500 list of the 500 fastest supercomputers of the world is also compiled from LinPACK benchmarks. Contributors are not required to use the portable HPL implementation used here; instead optimized variants can be used (for details see also [50]). Typical GE based clusters in the June 2008 TOP500 [1] show efficiencies from 23 % up to 64 % (showing that some installations go to great length to optimize the benchmark) with an average of 52 %. Infiniband based clusters show an average efficiency of 67 % and BlueGene machines of 79 %.

2.7 Evaluation of Larger Networks

With Karibik an 8-node test cluster with ATOLL was available for software development, tuning and system evaluation. To analyze the performance of ATOLL for larger networks of up to thousands of nodes, other methods have been used. In particular a network simulator

called SWORDFISH [46] has been developed. The design goals for this simulator included support for large networks, handling of deadlocks, injection of network faults, flexibility in terms of scenarios, routing options etc. Another important point was a powerful mechanism to interpret the simulation results.

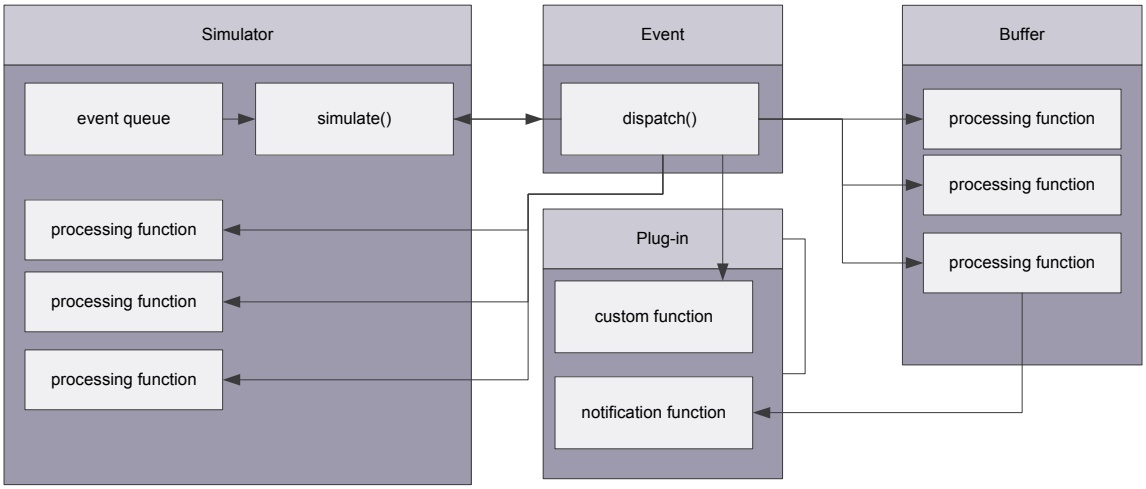


Figure 2-17: SWORDFISH Schematic Diagram [46]

SWORDFISH uses a discrete-event model to simulate complete, large-scale ATOLL-like networks. The simulator is component- or plug-in-oriented and written in C++. This means that different models for major components of the network can be selected at runtime and new models can easily be added to the simulation framework. Figure 2-17 shows a picture of the simulation flow of SWORDFISH. A central event queue holds the events to be dispatched to elements of the simulation environment. The central *simulate* function removes events one at a time from the event queue and updates the global time when no new events are available for the current time. The *simulate* function than calls the *dispatch* method of the event object, a classic example for object oriented polymorphism. The *dispatch* method for an event is different based on the type of event. Some events trigger operations in buffer objects, one of the central models in a SWORDFISH simulation environment, others trigger custom functions in arbitrary plug-ins. A complete simulation run is configured in a scenario XML (Extensible Markup Language) file. Here, all used components, the topology, routing strategy, traffic pattern and configurable options are set-up.

SWORDFISH version one specifically supported ATOLL like networks, version two added some new features. Figure 2-18 shows the hardware model of SWORDFISH which is closely modeled after the ATOLL hardware. For some of the hardware components several alternative models exist, most notably there is a choice of FIFO and round-robin arbiters which is the standard choice for hardware. The different propagation delays (denoted as forward and switch delay in figure 2-18) can be configured in the scenario XML file. To actually feed messages to the network the traffic controller component loads scenario applets. Such an applet is a user written program that uses the SWORDFISH messaging API to send

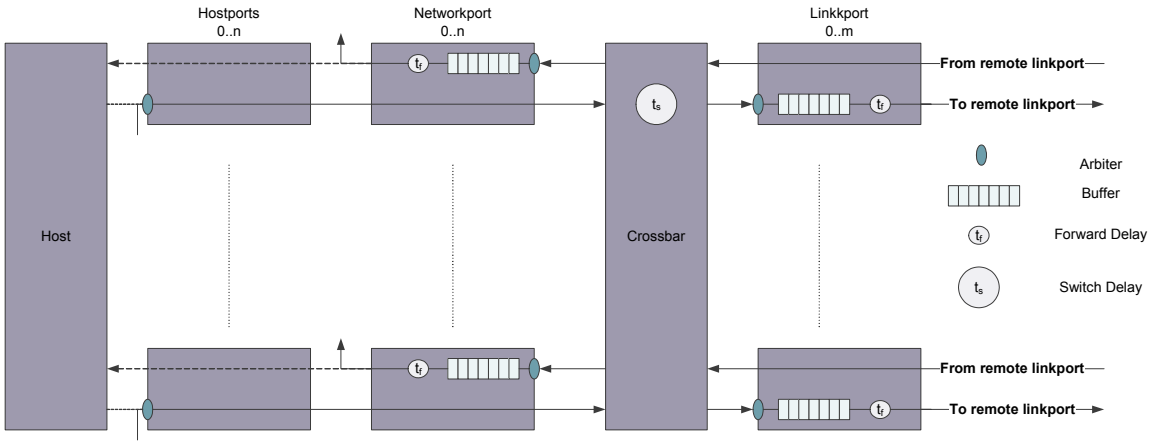


Figure 2-18: SWORDFISH Hardware Model of One Node [46]

and receive packets. One thread executing one copy of the applet is started per node, providing a MPI like environment for simulation. The functions of the API together with a classification in function groups are given in table 1.

Function class	**Functions (actual function name start with SWORDFISH_ prefix)**	**Description**
management functions	SWORDFISH_Init	initializes and finalizes system
	SWORDFISH_Finish	
	SWORDFISH_Size	returns number of processors/ returns number of own processor
	SWORDFISH_Rank	
time related functions	SWORDFISH_Time	returns current simulation time/ model computational time
	SWORDFISH_ConsumeTime	
sending of messages	SWORDFISH_Send	sends a message/packet
receiving of messages	SWORDFISH_Recv	receives a message/packet
	SWORDFISH_RecvFrom	
	SWORDFISH_RecvTag	
probing for messages	SWORDFISH_Probe	checks for new message(s)
	SWORDFISH_ProbeFrom	
	SWORDFISH_ProbeTag	

Table 1: SWORDFISH API

For routing, several routing strategies are supported including the well known dimension-order, shortest-path-first, west-first and Up*/Down*. For port arbitration there are a choice of a FIFO arbiter and a classic round-robin arbiter. To control the simulation, a command line component or a GUI component, depicted in figure 2-19 (a), can be chosen. Basic simulation statistics are always printed to the console additional output is governed by the choice of the statistics plug-in. A dummy or null plug-in and a plug-in rendering HTML pages are available. Figure 2-19 shows an example output of the HTML statistics plug-in.

Using point to point microbenchmarks it was tested that SWORDFISH actually models the ATOLL hardware accurately enough to analyze network behavior including switching events, blocking of messages and deadlocks.

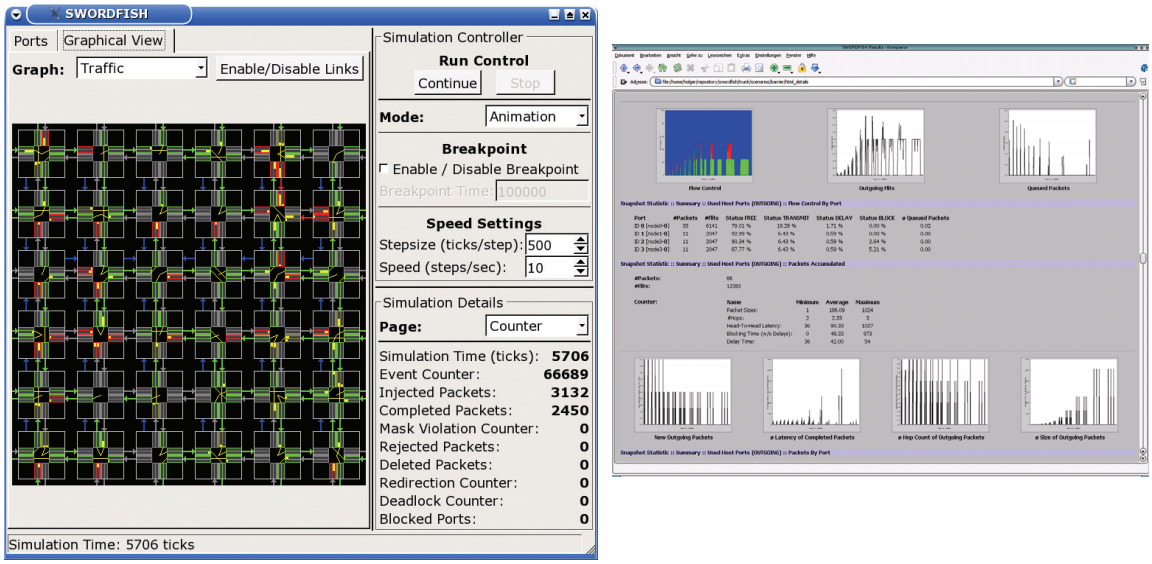


Figure 2-19: SWORDFISH (a) GUI and (b) Report [46]

Simple scenarios of SWORDFISH with appropriate parameters haven been benchmarked against ATOLL and a reasonable confidentiality was established. Simulation with networks of 256+ nodes showed that the deadlock rate, using not deadlock-free routing strategies as shortest path, is so high that the deadlock recovery abilities of ATOLL do not suffice to reach competitive performance levels. For an ATOLL network dimension-order routing was deemed to be most effective. One problem that arises from this choice is that it is not possible to optimally configure an ATOLL network as a torus, since ATOLL does not support virtual channels for deadlock free routing even in the presence of wrap-around channels (for further information see also [51]). It is possible to use a modified dimension-order routing which does not yield minimum routing distances for all paths, though. In this variant, the wrap-around channels are counted for as two additional dimensions. Results from a scenario featuring traffic with a different *locality* (x-axis) are summarized in figure 2-20.

Figure 2-20: SWORDFISH Blocking Rate in a 16x16 Mesh [52]

SWORDFISH uses a deadlock-resolving algorithm, so routing deadlocks are actually resolved and result in long blocking and high latency of messages. In the diagram it can be seen that the (unconstrained) Dijkstra routing algorithm causes routing deadlocks in the network. Dimension order routing, Up*/Down* and West-First routing are all deadlock free in meshes and thus exhibit low blocking rates. The problem is shown explicitly in figure 2-21 where the deadlock probability and the deadlock cycle length for Dijkstra's algorithm in a mesh are shown.

Now, if the same experiment is repeated on a torus topology the results are somewhat different. Dimension order as it is used in ATOLL is not deadlock-free any more and West-First is no longer applicable. To illustrate this, figure 2-22 shows the deadlock figures for the same scenario as above but using a torus topology.

SWORDFISH has also been used in turn to analyze the design space for the EXTOLL network layer, including choices at the switch or crossbar level and the link level. An important question that was analyzed using the simulator was the dimension of buffers and number of virtual lanes implemented for the EXTOLL network layer [53].

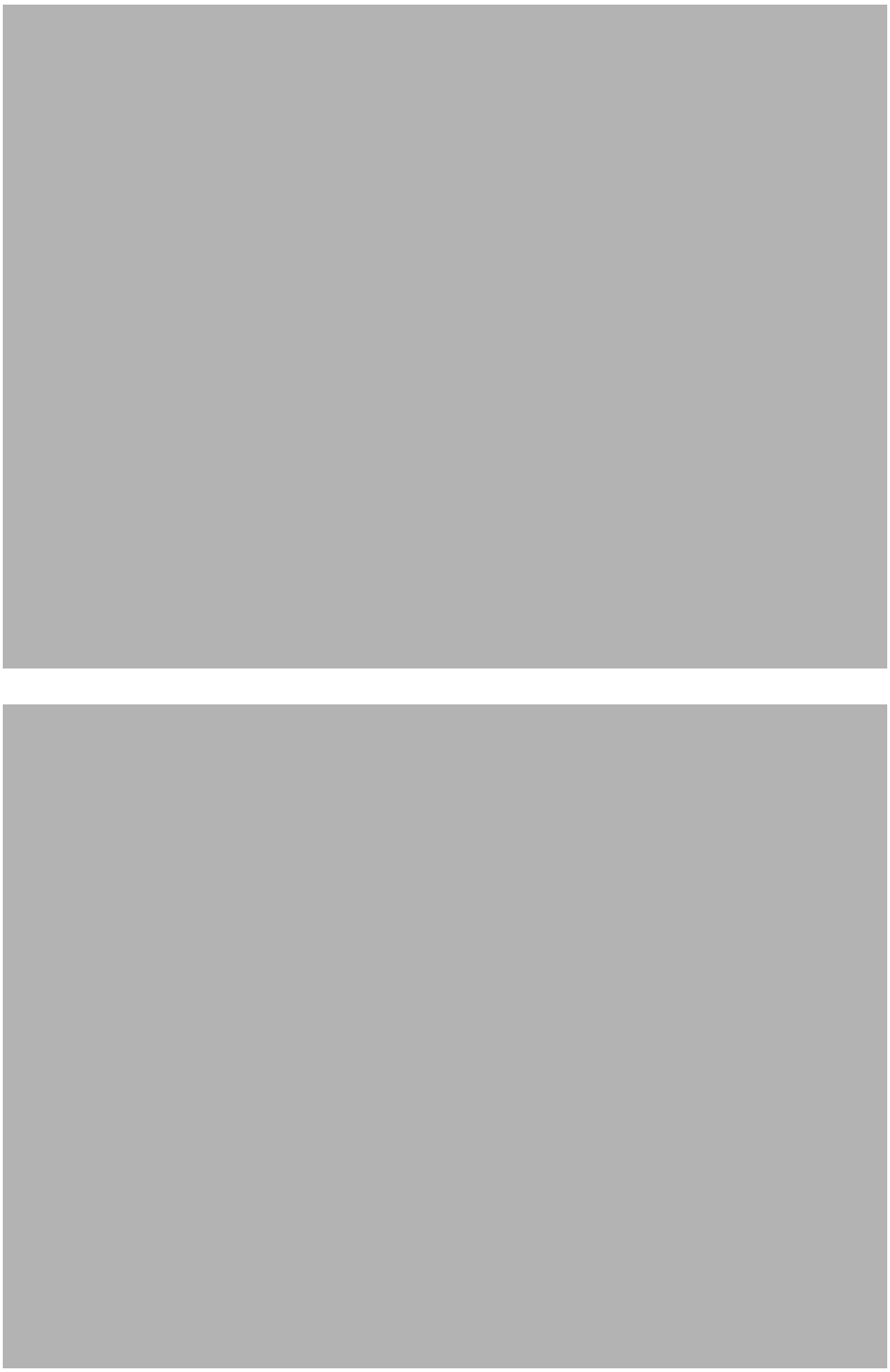


Figure 2-21: Deadlock Diagrams for a 16x16-Mesh (Locality) [52]

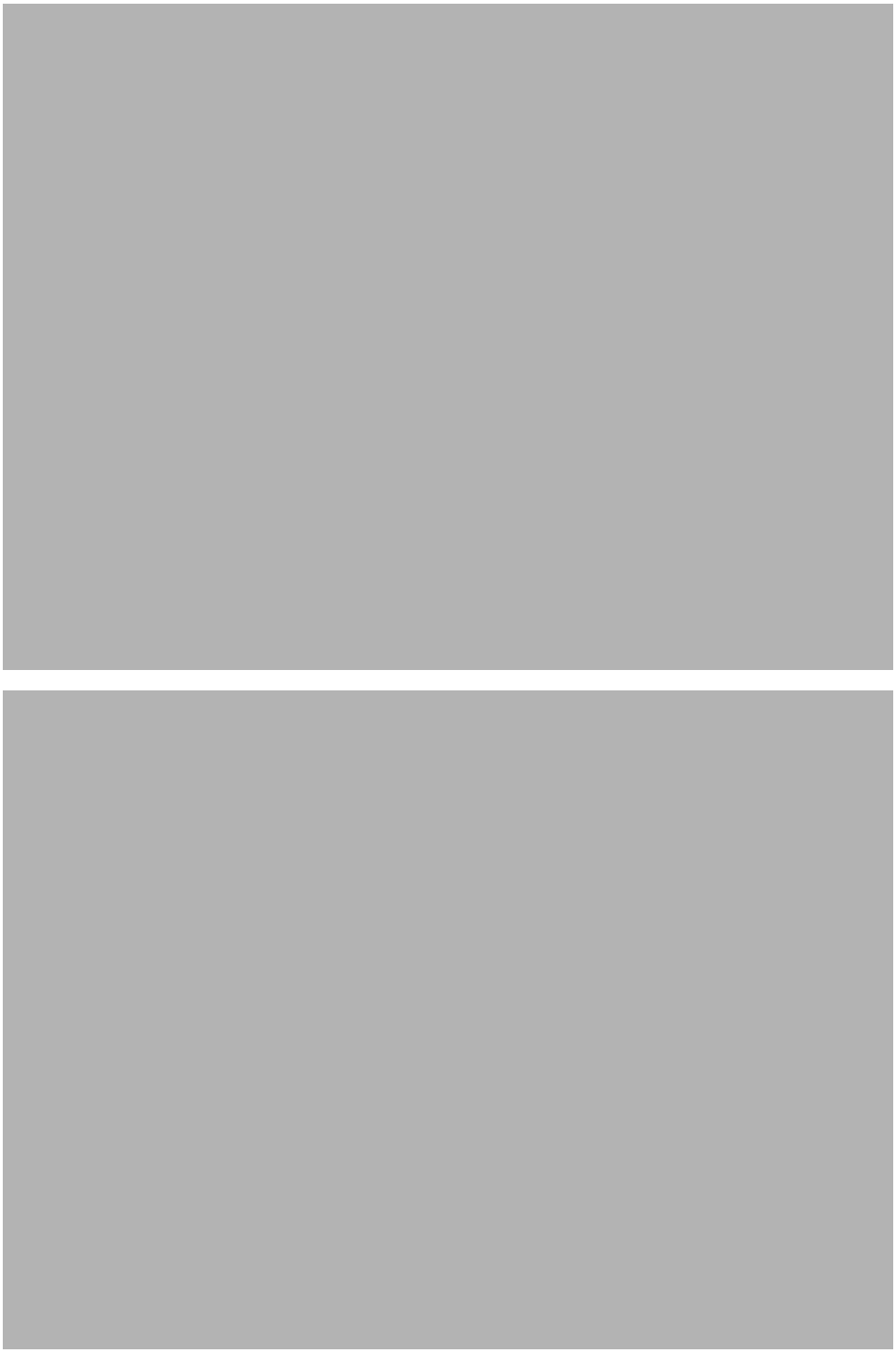


Figure 2-22: Deadlock Diagrams for a 16x16-Torus (Locality) [52]

2.8 Zero-Copy and ATOLL

The complete software environment for ATOLL described in this chapter relied on a two-copy scheme. So, while other means to optimize network performance and reduce CPU load like true user-level communication are implemented and used, every message has to be copied from the user/application buffer to the send queue (send DMA area) by the CPU on the sending node. Furthermore, the message has to be copied by the CPU a second time on the receiving node. One worthwhile optimization would be the use of a zero-copy approach. In a perfect world software could notify the hardware in user-space of a user/application buffer to be sent and the NIC hardware would perform all necessary tasks. On the receiver side, the NIC would perform some sort of *receive packet matching* and directly place received packets in application buffers. If it is possible to implement both sides of a zero-copy protocol, still a one-copy protocol could be interesting to explore.

While the ATOLL hardware was actually designed to support two-copy operations very efficiently, some efforts were made to analyze possible zero-copy applications. One proposal was to manipulate the kernel page tables in such a way, that memory that gets registered as a send buffer by the application is actually placed within a physical send DMA area. At the point of the memory registration, the content of the registered memory is copied into the page within the DMA area. Now, it is possible to send data from this registered memory without an additional memory-to-memory copy. The problems that arise here are:

- ATOLL DMA areas have to be physically continuous memory, a scarce resource.
- As large regions of continuous memory can only be allocated at system start-up, it is nearly impossible to allocate more than a few MB of physically continuous memory on a fully booted Linux.
- Even worse, afore mentioned problems make it necessary to allocate this memory statically, while the needed size is not known.
- An additional problem arises from the problem of memory registration. For send/recv style MPI, there is no such concept as buffer registration. Applications can request that any memory region should be sent. One possible solution could be to move a specific page into the send region for future zero-copy send, if it is detected to be source of repeated sends. This idea follows a cache like approach.

With the availability of Graphics Aperture Remapping Table (GART) resources in each Opterons Northbridge (see section 5.1.8) it would be possible to overcome the above problems by dynamically remapping pages using the GART which is restricted in size but at least allows mapping of up to 2 GB of physical RAM.

Zero-copy receiving in a send/receive model is even more complex than sending of messages. One problem often encountered is the matching of the incoming message with an appropriate receive buffer. Naive implementation could try to implement a posted receive scheme, where buffers get posted to a receive queue, and the receive DMA engine copies incoming messages into these buffers as they come in. Unfortunately it is generally unknown in what order messages are received, for example because multiple nodes are transmitting, and the size of the message to be received is also a priori unknown. So, for a true zero copy it is necessary to *match* the incoming message with the most appropriate

receive buffer. If the receive has not been posted early enough the message must be received in a general receive queue, the unexpected queue, and software copies the message to the application buffer, once it becomes posted. Since ATOLL does not implement any matching hardware it is for all practical purposes not possible to implement a zero-copy receive protocol.

In [54] a relatively simple extension to the ATOLL hardware is proposed which enables zero-copy RDMA operation. The RDMA extension together with a rendezvous protocol can be used to implement a true zero-copy message transfer protocol (at least for messages of a certain length; smaller message are more efficiently transferred in an eager fashion). The necessary virtual-to-physical address translation is straight forward, and based on a per-Hostport translation buffer where memory is pinned and then inserted by the kernel driver. A high-level software emulation of this method has been implemented.

2.9 Lessons Learned from ATOLL

With the end of the ATOLL project a résumé of the lessons learned from ATOLL can be given. These lessons were especially interesting for the subsequent EXTOLL project.

While ATOLL showed impressive possibilities, some general short-comings for a next generation interconnection network were seen. Those were:

- ATOLL's design is based around classic ring-based DMA based transfer methods. This has two drawback: It is not possible to implement zero-copy protocols, a prerequisite for near-DRAM bandwidth devices. Furthermore it is necessary to have continuous physical memory available for each queue (at least two per end-point).
- While ATOLL is a true user-space enabled communication device and with its four Hostports enables four user processes access to the hardware, a true virtualization layer is desirable. This would enable nearly arbitrary number of user level processes to access the device concurrently while preserving user-level communication and keeping hardware resources in reasonable bounds. With the advent of multi-core and many-core CPUs this becomes more and more important. The current trend of virtualization on OS level increases the interest for a virtualized device for the intra-datacenter transport in more classic datacenter application environments.
- ATOLL does not handle virtual addresses at all. It is not necessary for ATOLL, since all communication with the user space is performed via pinned, continuous physical memory. For true zero-copy protocols and virtualization it is necessary to solve the problem of physical-to-virtual address translation and vice-versa.
- The network layer of ATOLL offers high performance, but as the experiments with SWORDFISH have shown the lack of any provisioning against head-of-line blocking in the network layer is a problem when scaling to larger networks.
- With a node degree of four, ATOLL offers the possibility to implement different topologies. A 2-d torus seems to be especially interesting. Unfortunately the absence of a second virtual channel prevents an efficient minimal routing in a 2-d torus. The dead-lock recovery technology in ATOLL is very promising to solve problems that arise if a network component fails and for a short period of time routing has not been updated in the whole

network. In this case deadlocks can happen in a system where deadlocks are prevented during normal operation. Deadlock recovery features can help to make the system work relatively smooth, albeit with a performance penalty shortly after a failure, even in case of such network problems. The deadlock recovery algorithm is not an efficient way to enable minimal routing in a torus, though, since the deadlock probability is too high and the costs of redirecting in terms of software overhead and time are prohibitive. For a next generation network to support tori, it is absolutely necessary to support at least two virtual channels for the purpose of deadlock-free routing.

- The original ATOLL design featured a method to send small messages with very low latency: PIO based sending and receiving. Unfortunately, PIO sending/receiving did not work very well with ATOLL for two reasons. First, the buggy PCI interface proved to be a problem. Secondly the ATOLL design did not take the host CPU to device latency into account in a realistic way. The next generation system has to improve significantly in this direction.
- ATOLL suffers from a lack of support of multicast and barrier operation.
- Last but not least, a hardware design that works very well in simulation may still have numerous problems in a real system. Some situations are just not covered by the test vectors, other problems are not even available because of missing features in models for external IP. Rapid prototyping and verification in a real system environment has been identified as a major improvement.

These lessons have contributed heavily to the design and the implementation of the EXTOLL network as described in the rest of this thesis.

EXTOLL System Environment

Chapter 3

This chapter introduces the hardware system environment of the EXTOLL project. The prototype platform and performance numbers of system operations are introduced. From the performance numbers it is apparent why a HyperTransport and Opteron based system is the best choice for the EXTOLL prototype implementation. Most importantly the system offers a host-device access latency that is five times lower than possible with other competing technologies.

In the previous chapter ATOLL was introduced. ATOLL is connected to the system using a PCI-X Bus which alone contributed significantly to the overall system latency. Recent improvements in the architecture of commodity machines enable lower latencies previously not possible. System architecture has reduced memory system latencies as well as I/O to host latencies. Figure 3-1 shows the architecture that was common until recently. Traffic from the CPU passes several bridge chips, an I/O link and the front-side bus before reaching the device. Also, memory transactions between the device and the main memory have to travel over these links. The INs used are usually buses. For high-speed networking applications, buses exhibit significant disadvantages. They are unidirectional in the sense that a transaction at one time may only travel in one direction, so incoming and outgoing transactions from the same device may block each other. Even worse, buses are shared between all end-points, thus blocking between different devices may occur.

The modern architecture found in AMD Opteron based machines since 2003 is shown in figure 3-2. A similar architecture is being introduced by Intel with its codename “Nehalem” processors. Here, the memory controller is integrated into the CPU together with a crossbar switch that replaces the classic northbridge of the chipset. This architecture is sometimes called direct connect architecture (DCA) because no additional chips are necessary to built a multiprocessor machine. This new architecture enables devices to move closer to the CPU cores and main memory, thus enabling lower latency transaction. Additionally, the link used between these components is based on a modern bi-directional point-to-point link improving the situation further in contrast to traditional bus based systems. The same interconnect structure is used to connect several CPUs together to form a *non-uniform memory access* (NUMA) multiprocessor machine. The AMD family of processors uses the HyperTransport [13] standard as system interconnect. The future will see high performance I/O devices to move even closer to the CPU core and to the memory controllers of the system, i.e. they will also be integrated into the CPU chip. Devices that do not require high-speed are still connected using one or more bridges (southbridge) as well as traditional I/O

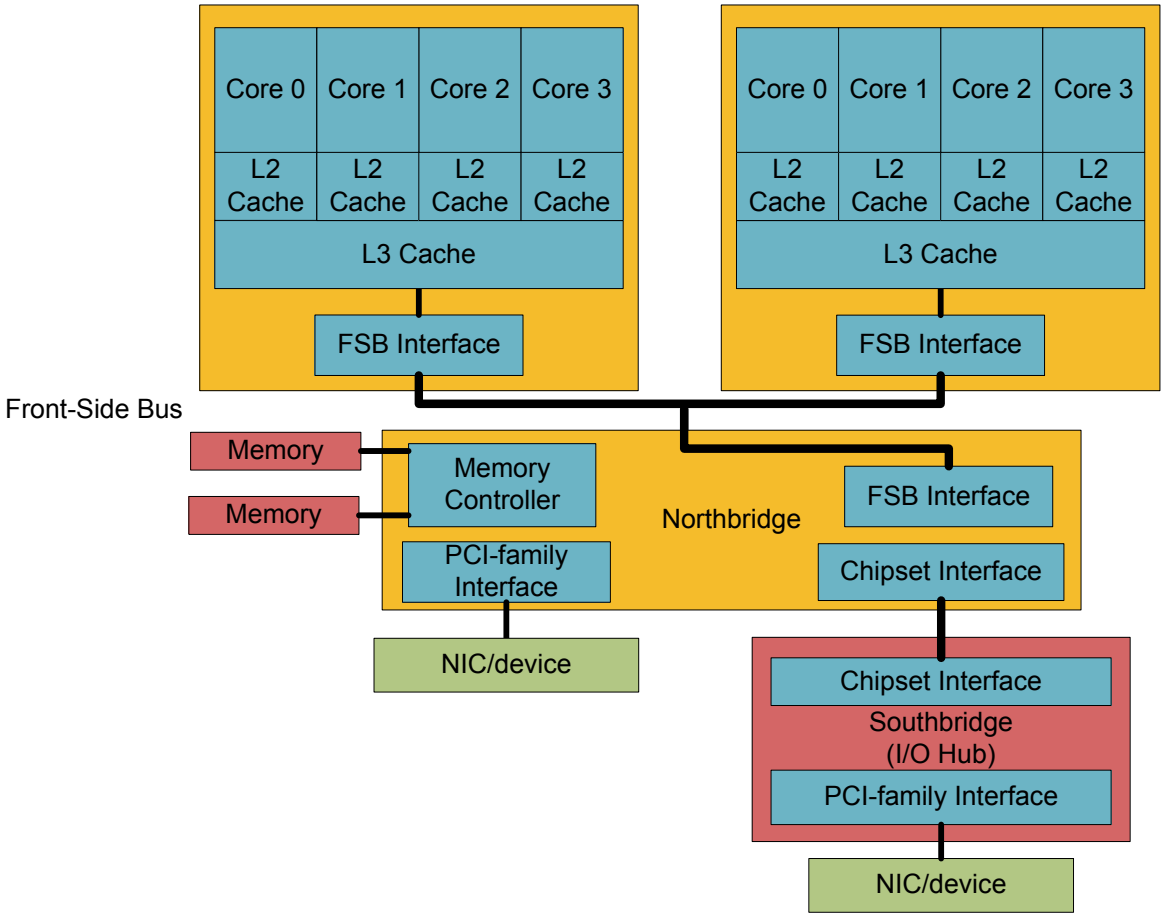


Figure 3-1: Traditional System Architecture

interconnect standards based on PCI technology. Such an architecture also enables the use of coherent devices, i.e. devices that participate in the cache coherence protocol of a multi-processor machine [55].

Another recent trend is the transition from parallel, bus based PCI to serial, bi-directional, packet based PCI-Express (PCIe) for many devices. PCIe-based devices can be found in both architectures introduced above. PCIe devices are still connected to the CPU using a bridge, typically using several bridges. PCIe needs the use of serializer technology on the physical layer. FPGA prototypes for PCIe unfortunately suffer from excessive latency, since the serial transceivers in current FPGAs are not optimized for low latency.

For EXTOLL, HyperTransport (HT) was chosen because it offers by far the lowest latency for the host-device communication. The hardware was implemented on the HTX-FPGA Board [56] (figure 3-3) which was designed by the Computer Architecture Group (CAG) at the University of Mannheim (now University of Heidelberg). The HTX standard allows adapter cards to communicate via HyperTransport with the CPU. The open-source HT-Core intellectual property core (IP-core) [57] also developed at the CAG was used to implement

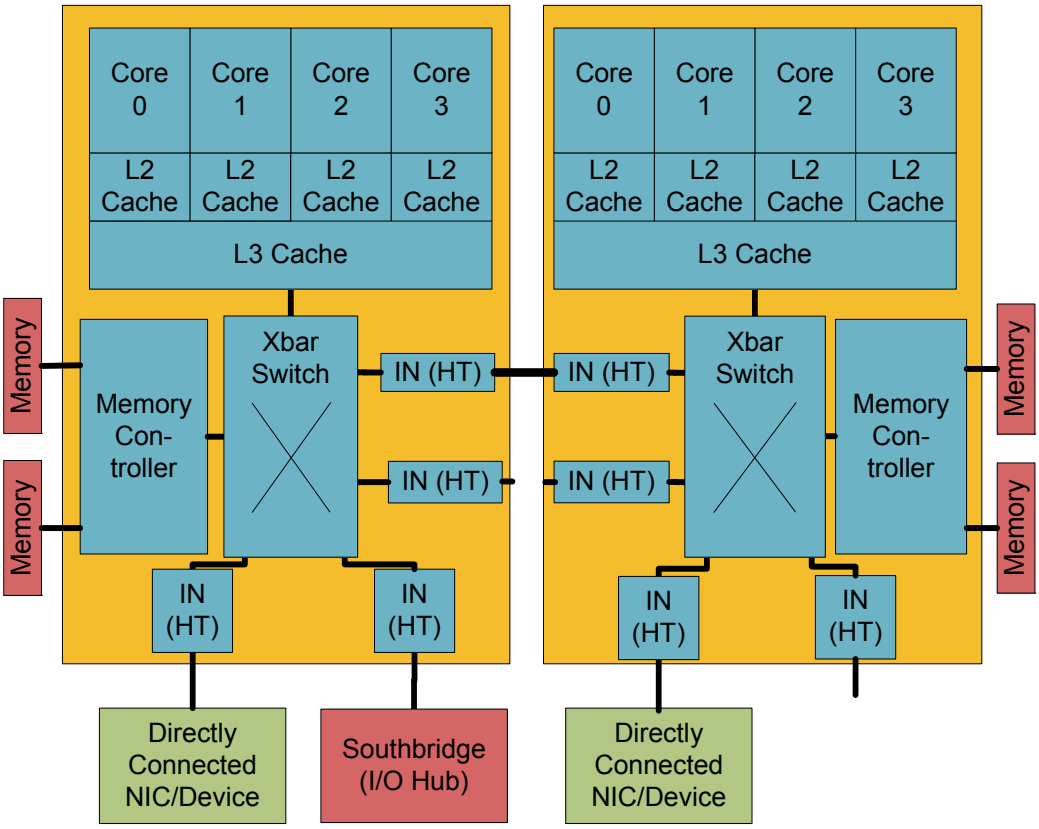


Figure 3-2: Modern System Architecture

the HyperTransport interface on the FPGA. The HT-Core is specifically designed for low-latency communication between the host and the device by minimizing the number of pipeline stages.

For the analysis, literature and measurements of existing systems were taken as guides. Most hardware design choices were modeled in terms of latency and other important parameters using a set of system performance parameters summarized in table 3-1. This table also gives the values for the parameters as measured on the prototype platform (the HT400/16-bit column) and the extrapolated values expected for a platform where the FPGA is replaced by an ASIC running with 3-5 times the clock frequency of the FPGA [58].

The performance numbers were measured using the HT-Example design, which is part of the HT-Core package and which was also used to characterize the HT-Core performance in [57]. They assume a device that is located immediately behind the HT-Core with no additional on-chip networking, buffering or clock domain crossing. The EXTOLL architecture as it is implemented does not fulfill these requirements, so that the device access latencies are somewhat higher. This fact was ignored for the design space analysis, since it is only a minor issue and applies to all analyzed solutions. So the parameters given in table 3-1 must be seen as a lower bound for the true latencies. Table 3-2 shows the differing parameters for

Parameter	Abbr.	Value (16-bit, HT400)	Value (16-bit, HT1000)	Notes
CPU cache access	t_{cc}	6 ns	6 ns	access of data in L1/2 cache, not dependent on HT frequency
CPU main memory access	t_{cm}	70 ns	70 ns	not dependent on HT frequency
CPU device access via HT	t_{cd}	100 ns	~50	half round-trip latency; partly dependent on HT, partly on CPU frequency
device to main memory access (via HT)	t_{dm}	280 ns	~180 ns	round-trip; partly dependent on memory latency
interrupt latency	t_i	1 µs	< 1 µs	latency from the device starting an MSI and the actual interrupt handler on the CPU being invoked
system call	t_s	170 ns	170 ns	average time it takes to enter the kernel from a user space program
CPU tablewalk	t_{trans}	180 - 300ns	180 - 300	translation time includes pinning, but not the system call
unpinning	t_{unpin}	135 ns	135 ns	
device cycle time	t_{cyc}	6.4 ns	2 ns	The cycle time for the HT400 case reflects the actual clock speed of the current EXTOLL FPGA implementation (156 MHz). For the HT1000 a clock speed matching to the HT core is assumed (500MHz).

Tabelle 3-1: Performance Parameters

a PCIe based system. The FPGA device access latency was measured using devices and corresponding PCIe IP cores from two FPGA vendors. In both cases the read latency was slightly higher than 1 µs, thus the write latency is estimated as a half-round trip with 500 to 600 ns. For an ASIC implementation the latency is expected to be cut in half, because of better serial transceivers and a more optimized IP core for the upper layers of PCIe (link and transport layer). Nevertheless, the ASIC performance given here is only an estimate and must be taken with the necessary skepticism. In the future, FPGA technology will probably also improve in terms of PCIe performance. A major step forward for this is the incorporation of *hard-IP blocks* for PCIe, i.e. fixed function ASIC technology blocks in the FPGA device for PCIe access.

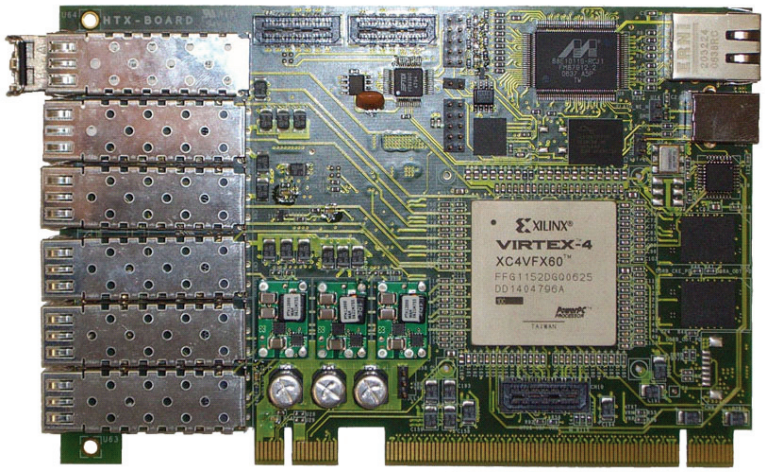


Figure 3-3: CAG HTX-Board

Parameter	Abbr.	value PCIe, x4 (10 Gb/s)	Notes
CPU device access via I/O interconnect	t_{cd}	>500 ns	PCI Express; FPGA PCIe devices
CPU device access via I/O interconnect	t_{cd}	<250 ns	PCI Express; estimated ASIC

Tabelle 3-2: Performance Parameters (PCIe)

The AMD64 architecture of the system together with the HyperTransport interconnection offers advantages besides a five times lower latency over using other processors together with PCIe that are not immediately apparent, though. For example, modern processors feature write-combining buffers which can be used to bundle several individual I/O write operations into one operation, increasing PIO efficiency considerably. Opteron processors and HyperTransport links can handle arbitrary packet sizes from one quadword (8-byte) to one cacheline (64 byte). Other processors only handle 8-byte or full cacheline transfers. Thus, the chosen architecture offers a greater capability when transferring messages from the CPU to the device.

Communication Paradigms

Chapter 4

In parallel, distributed memory system programming several paradigms have been used and are being used for inter-process communication. These communication paradigms and their implementations are an important factor for the actual programming of user applications since they influence the architecture of the applications. In the past and present, applications often take advantage of NIC implementations by preferring a special subset of a communication interface for best performance. The different programming paradigms and the implications of the different concepts for NIC designs are introduced first. The selection of the standards is based both on the importance of the interfaces, their expected future importance and their representativeness of software interfaces to networking. This is followed by an introduction to current and influential network application programming interfaces (APIs) and their architectures. It is important to understand these models, their characteristics, up- and down sides and use-cases in order to apply a correct design space analysis for the communication functions that EXTOLL should support. It will be shown that, while two-sided communication is still the most prevalent communication paradigm in scientific HPC (High-Performance Computing), RDMA (in the IB Verbs sense) may become more and more important both in enterprise computing and scientific computing and Global Address Space systems promise to be very effective in solving new scientific problems. Requirements to efficiently support these paradigms will be pointed out.

Thus, it is important to perform the right choices for the communication functions of EXTOLL to enable the NIC to work efficiently with a wide range of potential software and applications. The design space analysis at the end of this chapter will show how the software requirements extracted from the analysis of the software and API layers drive the design of the hardware and which requirements have to be taken into account when designing a new, modern NIC like EXTOLL. The results show that a low-latency architecture that efficiently supports synchronization, locking, small two-sided and efficient one-sided communication fits the software requirements. An important factor is also the possibility of high computation/communication overlap.

4.1 Two-sided Communication

The classic approach to distributed memory parallel systems programming is the message passing model, where messages are exchanged using matched pairs of send and receive function calls from the sender respectively receiver thread. The basic sequence of commu-

nication of this model is shown in figure 4-1[1].This well-known principle has gotten especially wide-spread with the use of MPI-1[59] in most parallel scientific codes. One important concept in two-sided communications is the notion of matching. Matching means, that a receive call does not deliver just the next message available from the transport layer, but a message that matches certain criteria. Typical criteria implemented are *sender ID*, *size* or the user specified *tag*. The intricate matching semantics of libraries like MPI make it also difficult to implement this matching in hard-wired circuits. NICs that provide advanced matching in hardware are nearly always processor based. It also regularly introduces a host of problems from handling of unexpected messages (i.e. messages for which no matching receive was posted by any receiver on the machine) to the exact semantics of *closest* match and the handling of possible race conditions between arriving messages and receives that are being posted. If a NIC performs receive matching, it can also provide for zero-copy receives, because the message can be delivered right into the application buffer given in the receive call. A design and prototype implementation for MPI matching using associative data structures in hardware is described in [60]. The prototype for one queue used ~15.000 slices of a Virtex2 Pro FPGA. This provides matching capacity for one queue (i.e. the posted queue) with a match width of 42 bit, a tag of 16 bit and a queue depth of 256 entries. A real NIC would need one hardware unit for the posted and the unexpected queue per *end-point* supported.

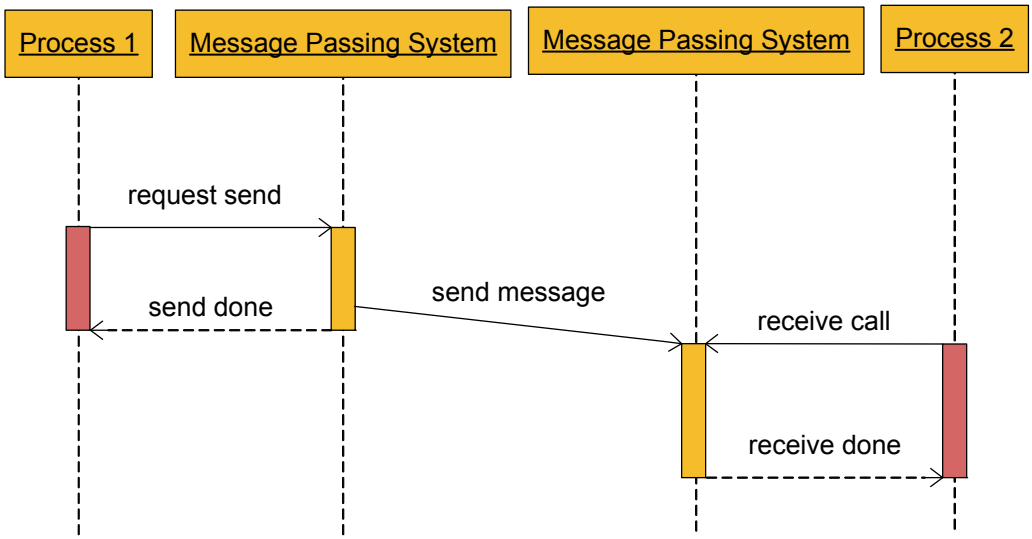


Figure 4-1: Basic Send/Receive Sequence

If an implementation chooses not to implement matching on the NIC, the matching has to take place in the host CPU. The normal process here is to place the message into the host main memory using DMA. The host CPU inspects the header of the message and performs the necessary matching; using a *memcopy()* operation the reception of the message can be completed. There are also proposals where the last *memcopy* is again offloaded from the

1. See appendix A for an explanation of sequence diagrams.

CPU [61]. The possible solution to only deliver the header to the CPU for inspection and matching while keeping the payload in an internal buffer of the NIC suffers mainly from excessive buffer use on the NIC and the latency introduced by the additional device-CPU-device roundtrip in signaling. Since large memory copy operations are expensive on standard CPUs it is generally agreed that zero-copy receive techniques are helpful and should be employed whenever possible. It will later be shown, that RMA techniques allow to emulate zero-copy matching receives to a certain extent at the cost of an additional network roundtrip.

Up until now the discussion has focused on the receiving side, since the sending side is much less problematic. Actually, zero-copy sends need to solve the address translation problem (see chapter 5) but the rest is straight forward copying of the payload to the NIC using DMA since no special matching has to take place. Often discussions of two-sided protocols differentiate several transaction sizes that are handled considerably different. Usually this involves small, large and sometimes also medium transactions. The background is, that it is often effective to handle small messages with the least complicated protocol and to use the CPU for matching and copying, while large messages can profit from additional offloading of tasks. The message sizes that serve as threshold between the different protocols are most often gained empirically.

4.2 Remote Load/Store

In the remote load/store paradigm, a process can issue a communication operation by just performing a load or a store operation to a special address. A load operations triggers a network request and the response to it carries the requested memory cell which is then used by the NIC to complete the load operation. A remote store operation functions analog, though no response is necessary (figure 4-2). This model requires only a simple NIC, actually more of a *bridge* function. A simple way to implement such a system could be to use PCIe switches which feature a non-transparent port [62].

While the remote load/store method looks tempting to many programmers and architects it suffers from fundamental drawbacks for parallel computing, that are intensified by the characteristics of today's computing architectures. First, CPU load and store operations normally only support relatively small data movement granularities. So, without modifications to the basic protocol, transactions are always in the register size region (32-128 bits) limiting the network efficiency. Secondly, long load latencies will stall CPU cores, since there is only a limited amount of outstanding memory transactions. Transactions to I/O space are normally subject to tighter ordering rules than cacheable memory intensifying the problem since transaction are executed strictly sequentially. Stalling a CPU for just 1 µs equals several thousand wasted cycles. Also, remote operations can only target memory locations; it is not possible to directly access CPU registers from the outside, which could increase the efficiency. Because of the high access latency and the CPU core stalling it is thus necessary to copy a remote variable to local memory. There, work on the variable can be performed efficiently, and consequently the result can be copied back. This model implicates that the programmer or runtime system handles coherency issues of an explicit caching model. The

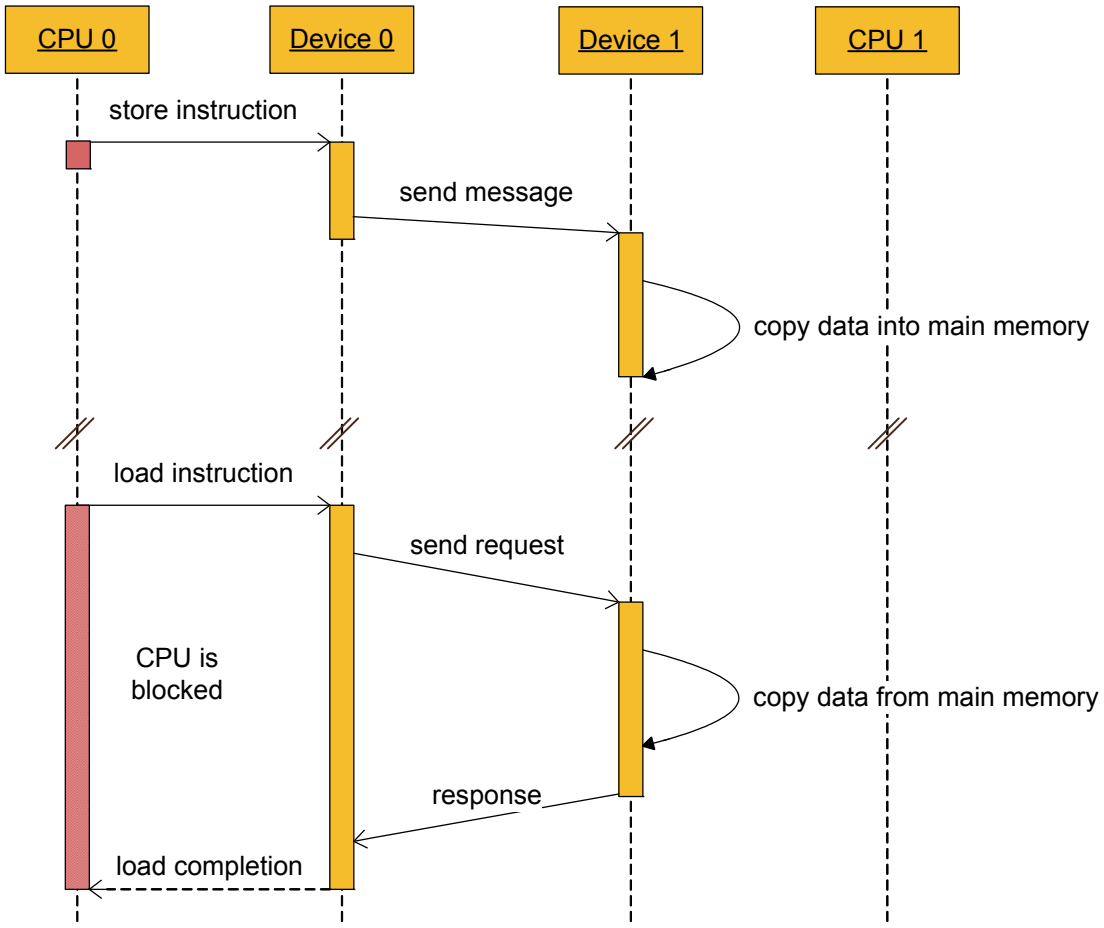


Figure 4-2: Remote Load and Store Operations

resulting method actually looks exactly as the RMA one-sided model. Actually, an efficient RMA NIC offers many advantages over a pure remote/load store model, for example CPU offloading for copy operations, no CPU stalling for remote accesses, and asynchronous access with completions.

With the necessary access to main memory and the cache-coherence protocol consequences the effective communication latencies of remote load/store implementations do not outperform competing small-sized send/receive functions. One idea to at least improve the possible bandwidth is to transport whole cachelines and provide a remotely cached version of the original memory in the remote NIC. This involves a cache-coherent NIC and the resulting complications for the implementation [55]. The ultimate result of such considerations would be an adapter that actually provides for a true distributed shared memory (DSM) system with full cache coherency [63] which introduces a complete different set of problems. DSM system architecture exceeds the scope of this discussion and the reader is kindly referred to [64] for more information in this regard.

4.3 Introduction to One-Sided Communication

So called one-sided communication is an ever more present trend in modern interconnection network designs. In the one-sided communication paradigm, also called RDMA, RMA or *put/get*-model, conceptually only one process is actively involved in a communication transaction. In the classical send/receive paradigm, both processes, the source or initiator and the target or destination process, participate in the transaction. An RDMA transaction features always an active and a passive partner. The *put* operation also called *RDMA write* operation writes the content of a local buffer of the initiator process into a specified buffer of the target process. Notice that the initiator needs to know the address of the remote buffer beforehand.

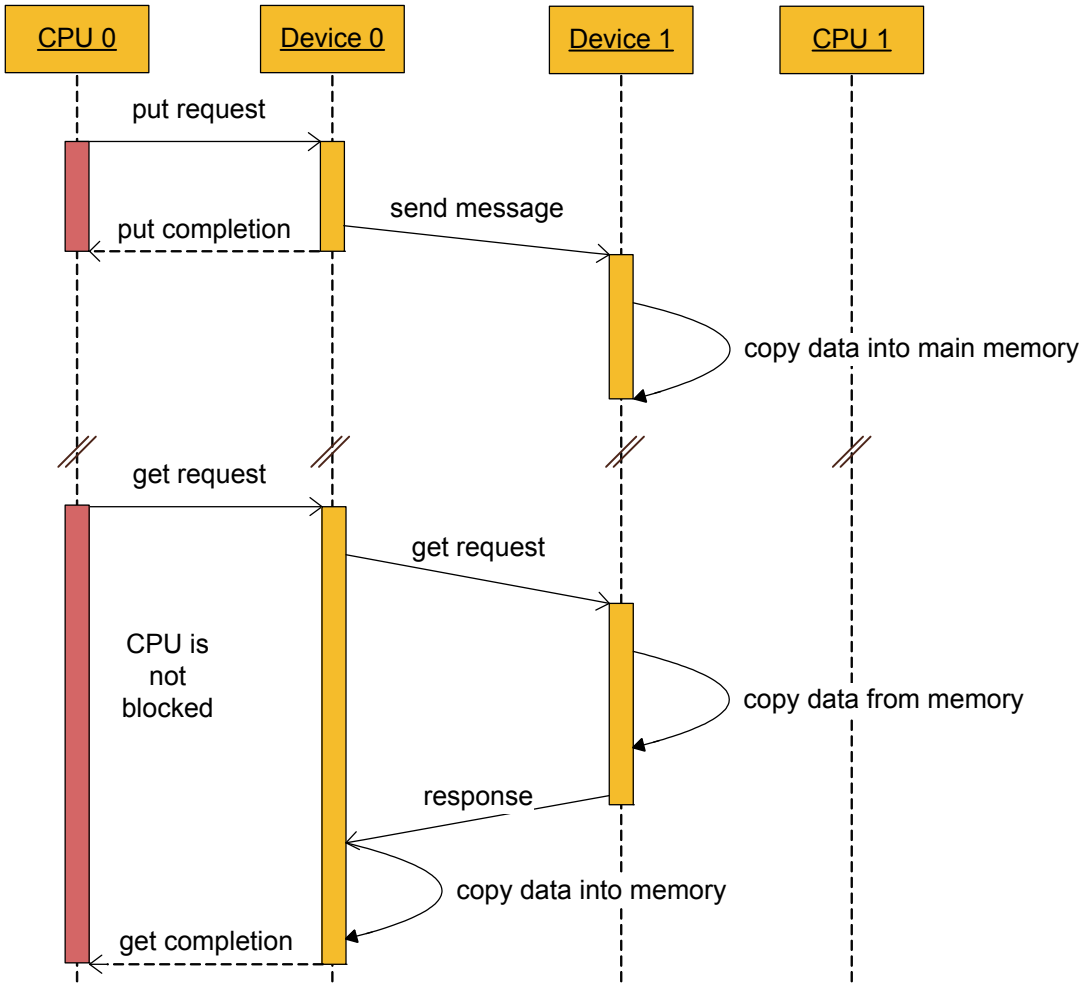


Figure 4-3: RDMA Operations

The *get* or *RDMA read* operation is the opposite; the initiator request the contents of a remote buffer to be copied into the specified local buffer.

In a sense, the one sided-communication paradigm is influenced by retaining a small part of the programming model of shared memory systems for distributed memory systems. RDMA operations can also be used to effectively implement zero-copy receive strategies, mostly for large messages. Consider the protocol shown in figure 4-4:

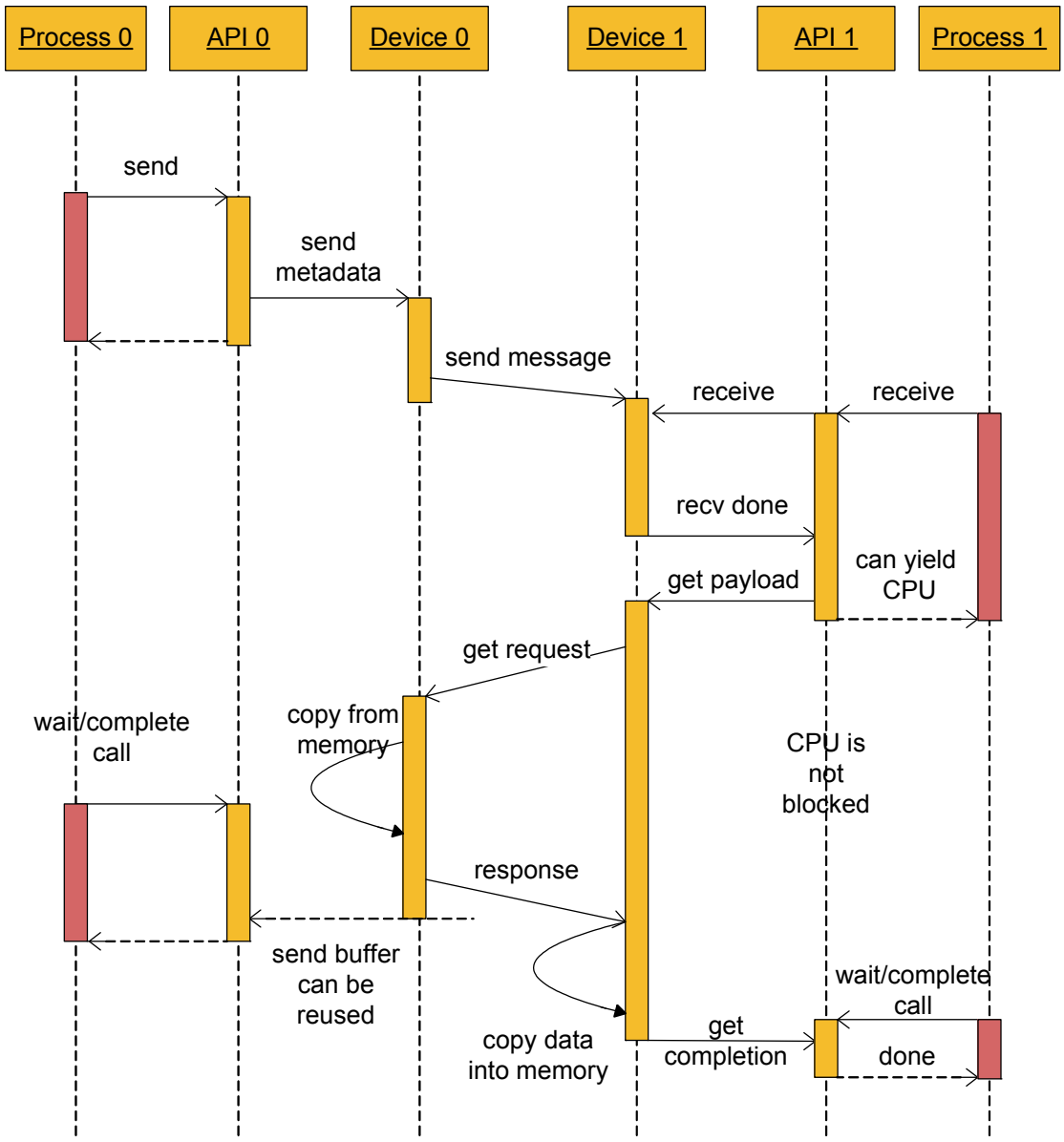


Figure 4-4: Zero-Copy Send/Recv using RDMA

- Source sends match request to receiver including source address.
- Target receives request using normal receive semantics (w/o zero copy).

- Receiver CPU performs message matching. If no matching receive buffer is available yet further processing of the request will be postponed until such a request is available. A possible variant gets the data into a system buffer defeating the original zero-copy though, to accelerate later reception.
- The receiver initiates an RDMA *get* (or read) to directly transfer the message data from the source buffer into the destination application buffer.
- After the *get* has been completed, the receiving process can be informed about the completed receive and operate on the data.
- The original sender must also be informed that the buffer can now be reused. Often this involves another network transaction to signal the completion to the sender.
- If the *get* operations can signal its completion at the remote part, this feature can be used to signal completion to the sender.

The above protocol is compelling because it can leverage the flexibility and speed of general purpose processors for the actual matching, while still providing true zero-copy. But the protocol suffers from network latency, since at least one additional round-trip is necessary. Low-latency service is thus beneficial for this protocol.

Another recent use-case for RDMA protocols evolves around distributed storage systems. Here in-kernel protocols are used to let RDMA-enabled networks run improved versions of well-known storage protocols such as NFS [65], iSCSI [66], and DAFS [67][68].

Currently quite a number of different one-sided implementations, standards and protocols have been proposed and implemented. With the advent of ever higher I/O bandwidths, zero-copy techniques become essential and one-sided communication techniques offer an elegant model to handle a large number of distinct problems elegantly.

In the following sections, the current state of the art in communication models and APIs is presented.

4.4 Important Communication APIs

Currently, both two-sided, one-sided, hybrid APIs and APIs supporting several models are in use. The selection of protocols presented here is based on the prevalence in modern systems and to showcase important design options.

4.4.1 Sockets

The Berkeley Sockets API [69] is probably the most widely-used API for networking and was first introduced with the release of 4.2 BSD in 1983. The goal of the Sockets API was a generic interface for accessing computer networking resources. A large number of different protocols, also on different protocol levels, are supported. Today, the most often used sockets are bound to the TCP/IP, UDP/IP and the UNIX Domain sockets protocol.

The Sockets API is based on the concept of a socket being the abstraction of the logical connection between two end-points, thus it is generally formed by a tuple of end-points. The addressing within the Sockets API is protocol dependent and abstracted using different address families, for example AF_INET for IP based protocols.

Since sockets based communication is a generally well understood, and because of its wide spread use, very well documented communication API, a very short overview is provided here. If the used protocol is connection oriented as in the case of TCP, sockets generally follow a client-server paradigm, where the server opens a socket and listens for incoming client connect requests which are then accepted forming a new socket at the server side and connecting the socket at the client side. In UDP (User Datagram Protocol) sockets, this step is omitted. Often data is sent using the common POSIX standard library functions *write* (respectively the corresponding systemcall) and received using the *read* function. There are also a number of other functions available to optimize the communication, for example *sendfile* to send the content of a file directly over a socket. This can be exploited to reduce the number of copies in the sender path as the content of a file usually is already present in a kernel level buffer. A typical sequence for a simple communication is shown in figure 4-5. The two copy operations that are minimally performed by typical implementations are shown. Each transition from the client or server to the API causes a transition into the OS, each arrow from the API leaves the OS again.

The sockets API which is often but falsely synonymously identified with TCP/IP networking suffers generally from high communication latencies. These are introduced because the classical socket implementations are OS based (thus every operation includes a system call), have to pass through a relatively deep and complex protocol stack including several memory copies and finally because of suboptimal latency characteristics of the NIC and network hardware. An instructive reading to understand latency characteristics of sockets, TCP/IP, and Ethernet is [20]. The authors show that currently the minimum latency for the stack is at least > 10 µs. This paper gives a detailed breakdown of the latencies of the sending and the receiving path. The actual sockets interface (i.e. from application to the end of the TCP layer, including the system call) is given with 950 ns.

There are also more optimized implementations available. For example in [70] a user space implementation of a socket library is presented which communicates using Myrinet. Such a user-space implementation with reduced copies promises good performance but also introduces numerous new problems as correct systemcall interference, process fork behavior, limited network end-point handling, and missing support of newer, optimized system calls such as *sendfile*. The Socket Direct Protocol (SDP) [71] is OS based but uses Infiniband to provide a better performance. Similar solutions exist for other high-speed networks.

So, the support of sockets is generally deemed necessary because of the high count of application codes that use the API. Since the API has a strong usage and long history it is necessary to support all aspects correctly, since otherwise problems with some applications will occur sooner or later. The most compatible solution is still a kernel level solution and the

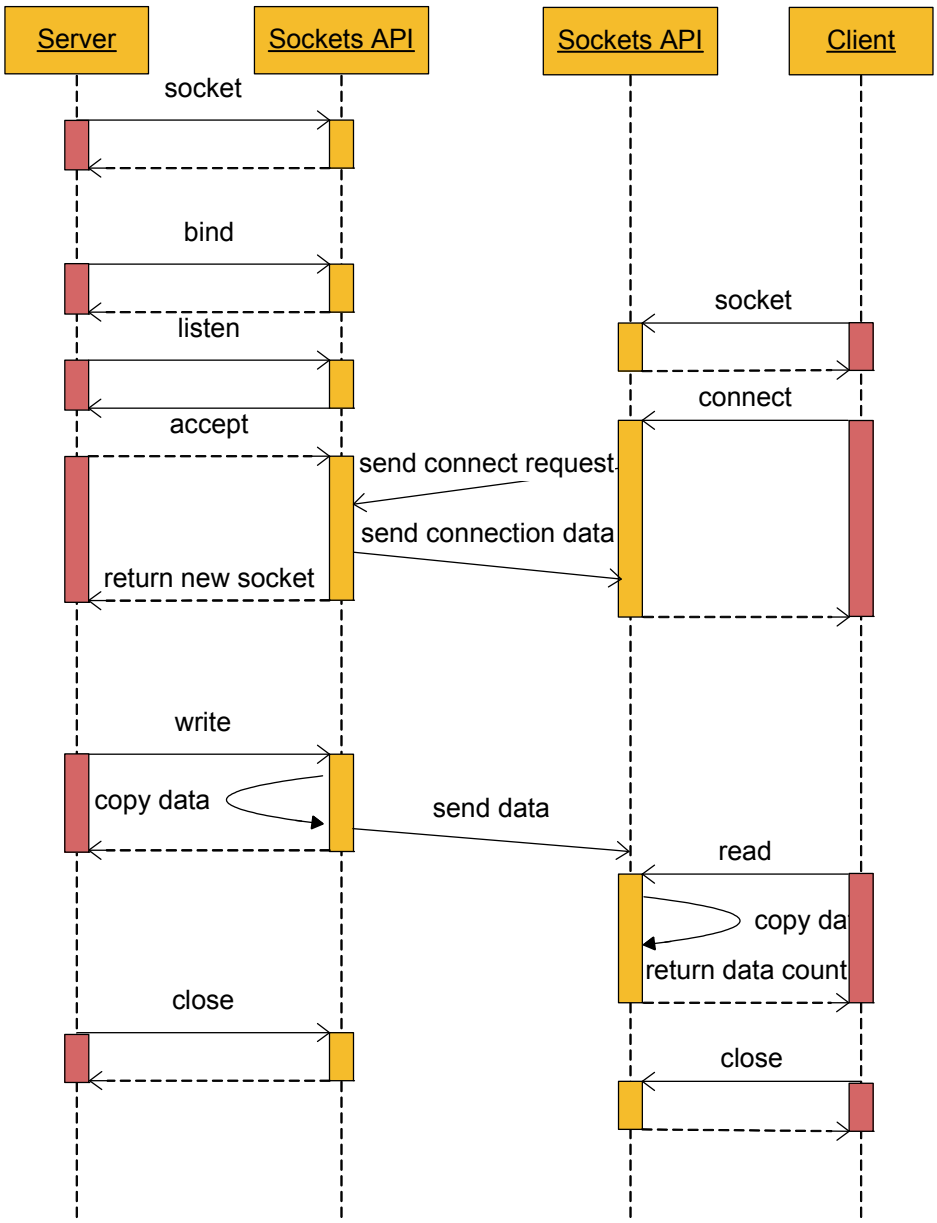


Figure 4-5: Sockets Sequence

features of modern high-performance NICs are very difficult to be exploited by Socket based applications. This is also one of the reasons why standards such as Infiniband and iWARP (RDMA over TCP) have occurred on the market.

4.4.2 MPI-1

MPI was originally standardized in 1994 [59] to form a consistent standard for the programming of different distributed memory computers. Point-to-point communication operations in different flavors form the basis for the MPI API. Additionally, different communication

groups, a large variety of collective operations, topology related operations and a number of a other functions are part of the standard. Here, the emphasis is on the point-to-point operations. MPI distinguishes between buffered and unbuffered send operations, blocking and unblocking operations. Probably the most important feature of the API which complicates implementations are the matching rules. A message arriving at a process must first be compared to already posted receive operations, i.e. receive operations that have been posted by a previous function call. If the message matches, the closest match according to the MPI matching rules must be chosen and the receive completes. It is possible that no matching receive is found; then the MPI implementation puts the send operation into the so called *unexpected queue*. Whenever the local application posts a new receive operation, the receive operation is first matched against pending entries in the unexpected queue before it is forwarded to the posted receive queue, ready to be matched to newly incoming messages. Messages have to be matched according to source (including wild-cards), size and tag (a user provided integer).

Receive matching is the key problem to build an extremely efficient MPI-1 solution, because only if one can somehow solve the matching problem, zero-copy receive-side protocols become possible. Receive matching has been implemented in a variety of ways. The most simple way, and the only possible solution for many architectures, is to move the complete matching into the host CPU and perform a *memcopy* operation after the match succeeded to copy the buffered message into the final application buffer. This protocol is straight forward to implement and still performs relatively good if the rest of the system is designed well. Examples that employ this approach are MPICH for ATOLL as well as all TCP or UDP based MPI implementations. Even implementations with more elaborate solutions may choose to implement this strategy for a subset of messages. Especially for small messages the high-computation speed of current host processors enables fast matching.

There are NICs that perform the matching or parts of it. These NICs without exception use an on-board or on-chip processor to implement this. While this solution offloads the host CPU from the matching it introduces new problems and bottlenecks. Generally, the NIC processors offer one order of magnitude less single thread performance than the hosts CPUs adding to the latency of operations. The matching information has also always to be transported to the NIC and the processor on the NIC often has only a very limited amount of memory available for message and descriptor queues. If a received message does not match any of those posted to the queue on the NIC, the message is put into main memory (the unexpected queue) and the host CPU must ultimately match and deliver the message. Performance of such solutions may degrade dramatically, for example if the number of necessary posted receives exceeds the NICs resources, or receives are constantly posted too late to allow for timely matching. Note also that there is a potential race condition when a receive is posted to the NIC, since the matching message may just be copied into main memory by the NIC at the time the receive is posted to it.

One completely different implementation class often chosen to avoid the above mentioned problems is the class of rendezvous based protocols. Rendezvous-protocols add additional round-trips but offer the chance for true zero-copy which makes them especially attractive for large messages. The RDMA based zero-copy protocol shown in figure 4-4 is one exam-

ple for a rendezvous-protocol. It has been shown, that implementations using rendezvous-based zero-copy-protocols exhibit high bandwidth for large messages with low CPU utilization. At the same time is often necessary to use a buffered protocol for small messages to keep latency reasonably low.

There have also been thoughts on how to accelerate the actual matching of messages. Processors generally perform poor if associative searches have to be performed. Usage of advanced data structures for the MPI queues helps. Good handling of large queues becomes absolutely necessary for large parallel machines where thousands of communication peers are present. In [60] a hardware architecture for message matching is described. This work uses a hardware structure similar to a ternary CAM (content addressable memory). The authors claim a reduction in latency but of course, the solution is resource constrained since associative memory elements must be present for each slot of the matching queue and the unexpected queue. The resource problem is aggravated by the fact that matching hardware resources must be added for each end-point supported. If multiple end-points need to be supported this solution does not scale.

In addition to the described point-to-point operation, MPI defines a number of collective operations including synchronization calls like *MPI_Barrier* and collective calls transporting data like *MPI_Broadcast*, *MPI_Alltoall*, *MPI_Gather*, and *MPI_Scatter*. Another important group is the group of reduce operations which can be used to combine data from all processes into one location at one specified root process using an operation given in the function call. MPI even allows to use a user-specified operation. Collective operation have to be called by all participating processes and are generally blocking in nature. There are countless research projects, implementation proposals and implementations of collective operation on different architectures and interconnection networks. Collective operations play an important role in some codes and problems. Known hardware assisted implementations can follow strategies like hardware barriers, multicast communication primitives and reduce computation in the NIC.

4.4.3 MPI-2

The MPI forum started on the second revision of the MPI standard shortly after the final revision of MPI-1 was published. MPI-2 [72] introduced several new key features, namely dynamic process management, parallel I/O functions, C++ bindings and one-sided communication routines. While the additional language binding is a pure software enhancement for better interaction with the C++ language, the new dynamic process management is more of a runtime extension and allows to dynamically spawn additional processes during the runtime of an MPI job. MPI I/O introduces an API to perform efficient parallel I/O from the processes of an MPI job. This is an important feature for many large codes; from a hardware/software design perspective it does not add new requirements since all the feature can be efficiently implemented using either one-sided or two-sided communication paradigms. It would be interesting to see the effects of a dedicated, hardware assisted, distributed storage system that directly interacts with MPI I/O.

For this work, the one-sided extensions for MPI are most relevant as they introduce RMA capabilities into the MPI standard. The one-sided MPI operations are *MPI_Put*, *MPI_Get* and *MPI_Accumulate*. In addition, the standard defines memory or buffer management functions and a number of synchronization functions that can be used to implement several different communication/synchronization patterns. In order to share a region or memory, i.e. open it for remote access, the function *MPI_Win_create* must be used. This is a collective operation, which is passed the base address and the size of the region to be shared. The size is not required to be same on each process; it is even possible to pass a size of zero, which actually exposes no memory for this process. The function is collective, so that the creation of the window and its parameters can be made known implicitly to all other participating processes by the runtime system. The function returns a handle for the created window which is used in subsequent calls to reference the window. There is of course a corresponding *MPI_Win_free* function.

Once a window is created, the transfer functions can be used for data transport. The *put* function writes data to the target process, while the *get* function reads from the target. The *accumulate* function manipulates data at the target process. The parameters of the *put* and *get* function are straight forward; the target address is given through a window handle and a displacement within this window. *MPI_Accumulate* behaves in essence like a *put* where the data at the target is not replaced but combined using one of the standard reduce operations already defined by MPI-1, like *sum*, *and*, *or* etc. A purely hardware based implementation of *MPI_Accumulate* is much more challenging than one of the other two functions, since data must actually be transformed, that is there must be some form of computational resource available. One way to implement this is to use an embedded processor on the NIC. The complementary approach is to let the host CPU handle the computation by either getting the value, updating it and writing it back or let the target CPU handle the combination. Both strategies add latency to the implementation. The first is still truly one-sided but may suffer from synchronization problems.

All MPI-2 one-sided operations are non-blocking and require synchronization calls to ensure that changes are actually visible in the memory model of the involved processes (that the operation has completed). To this end *access epochs* of windows are introduced. An *access epoch* starts with a synchronization call and ends with a synchronization of the associated window. After the *epoch* has ended, changes by one-sided operations to the memory region of the window are guaranteed to be visible. MPI-2 distinguishes two basic synchronization methods, namely active target and passive target synchronization.

In active target synchronization, the target process actively participates in the communication in the sense that the process calls synchronization functions. In passive target mode the target process does not participate in synchronization calls. The simplest form of synchronization is the *MPI_Win_fence* function, which is a collective function associated with a window. It provides a barrier function for the referenced window and can act as the synchronization call between two epochs in active target mode.

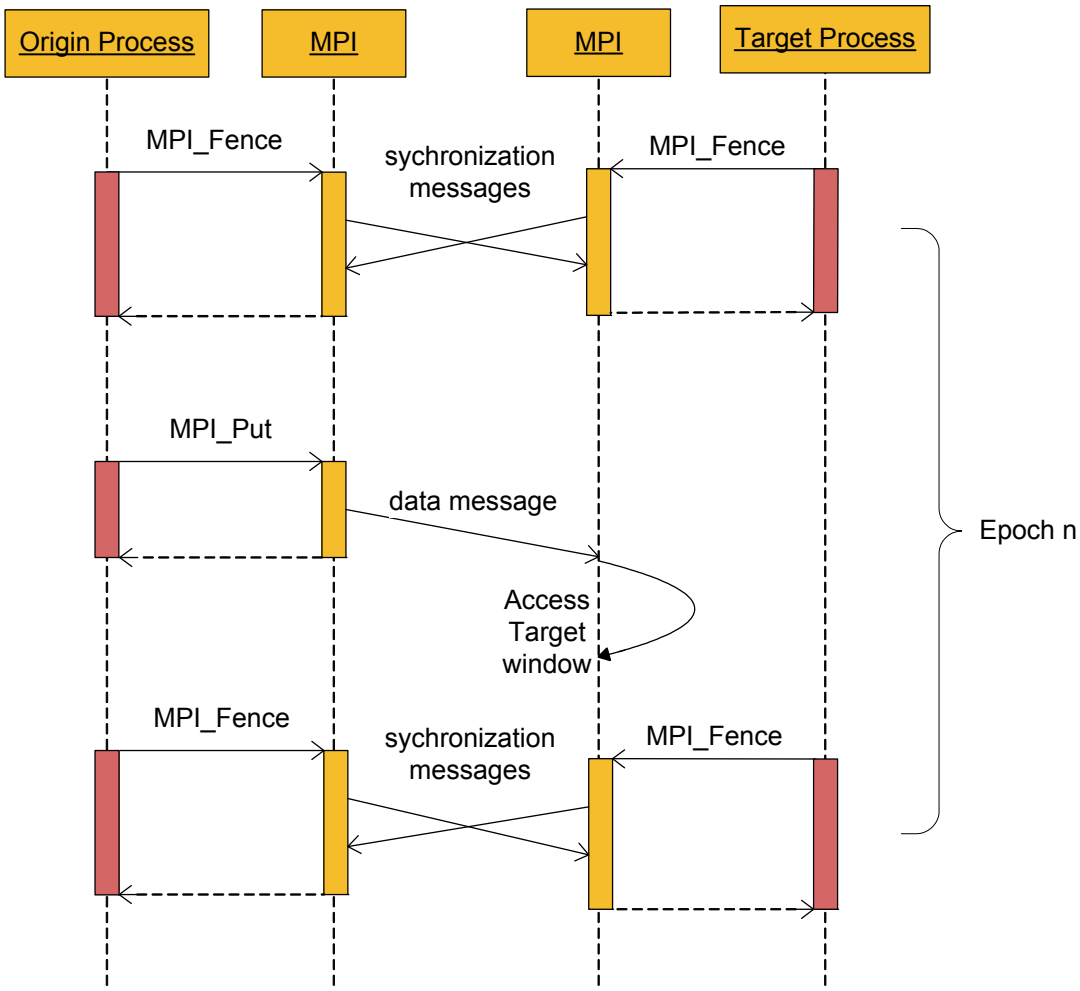


Figure 4-6: MPI One-Sided Communication with Fence

The second method, also in active target mode, uses a set of functions for finer grain synchronization. The functions for this method are *MPI_Win_start*, *MPI_Win_complete*, *MPI_Win_post* and *MPI_Win_wait*. At the origin side(s), i.e. the process(es) initiating one-sided operations, the epoch is started using a call to *MPI_Win_start* and after a number of RMA calls closed again using the *MPI_Win_complete* function. On the target process(es), the epoch is started using a call to *MPI_Win_post* and the epoch is closed again by *MPI_Win_wait*. In a sense *MPI_Win_start/MPI_Win_post* and *MPI_Win_complete/ MPI_Win_wait* are pairs of functions that are collective synchronization operations to a given group of processes only, namely the ones involved in the operation. These functions are allowed to be non-blocking, i.e. the start function returns, although the post function has not been called by the peer process and the actual communication operation has to be delayed until that point in time.

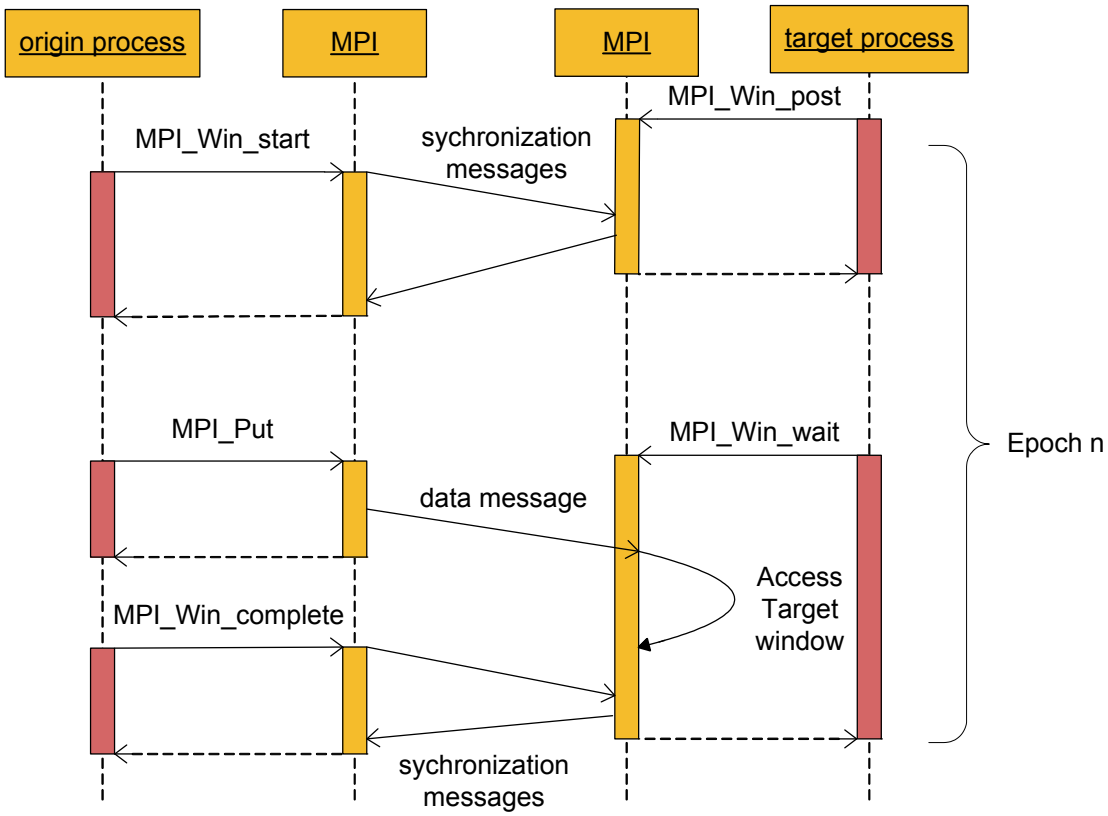


Figure 4-7: MPI Start/Complete/Post/Wait Synchronization

Also, an implementor can exploit the non-blocking nature of the one-sided operations to implement one-sided functions on point-to-point two-sided primitives in active target mode. This is sometimes called one-sided emulation and is used for MPI implementations that run on standard Sockets based TCP networks. Conceptually a *put* can be implemented using a *send* which is initiated through the *MPI_put* call. The matching *receive* which completes the function by copying the data to the final destination is started by the synchronization call of the target at the end of the corresponding epoch.

Finally, passive target synchronization is enabled using *MPI_Win_lock* and *MPI_Win_unlock*. These functions are only called at the origin (since the target has to remain passive) and form an access epoch. While a process holds an exclusive lock on a target window, no other process is allowed to access this window. If processes hold a shared lock, no processes requiring an exclusive lock can access the window during the time the shared lock is held. No assumptions about ordering of the lock/unlock operations of different origins are made, i.e. lock/unlock operations are not collective. The passive mode is much more difficult to implement; one constraint exists, though. Lock/unlock can only be used with windows whose backing memory has been allocated using *MPI_Alloc_mem* thus allowing the run-time system to provide for special arrangements to handle locking and unlocking correctly. Trivially, locking and unlocking always need an actual network transfer on distributed memory systems and are probably most efficiently implemented using an

atomic compare-and-swap or similar operation. Unfortunately many system do not provide this operation over the network. In such a case an implementation may have to resort to an asynchronous thread handling the actual locking/unlocking.,

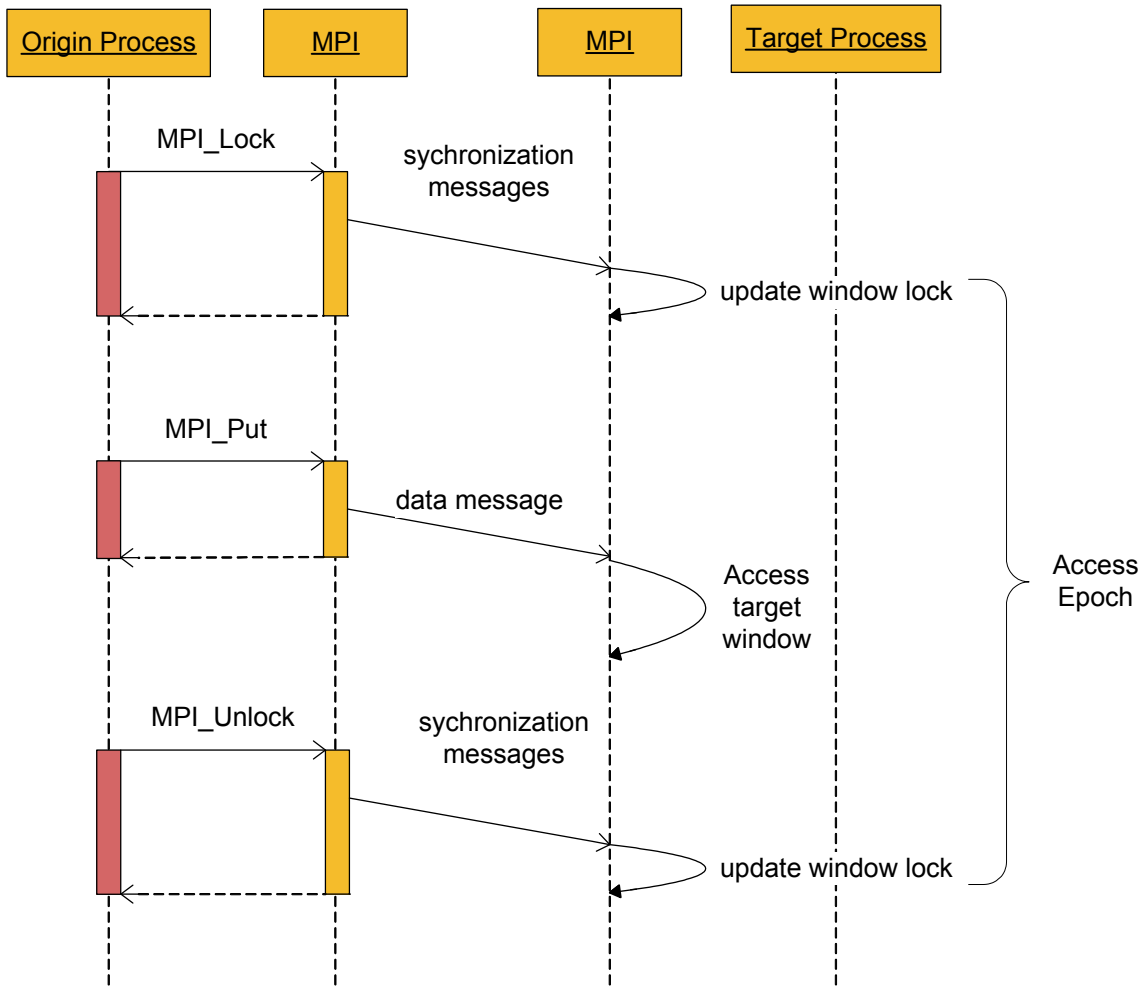


Figure 4-8: MPI Passive Target Synchronization

From an implementors side, the active target synchronization mode is straight forward to implement correctly even on low-level APIs and NICs that do not provide RMA operations. The passive target synchronization may expose more overhead. It was originally devised for shared memory operations. One way to implement it is based on an asynchronous agent, for example a thread, on the target side, that periodically handles incoming locking requests which is not the most elegant solution. In [73] the authors show, how exclusive locks are implemented using IB's atomic compare-and-swap (CAS) transaction. Shared locks still require an asynchronous agent on the target, which issues a CAS transaction to set the lock in shared state. It then uses internal counters to handle the shared lock. The problem of network contention through excessive sending of IB CAS operations is not handled in [73], but the authors propose to use exponential back-off as a the method of choice. Earlier IB MPI-2

implementation used a two-sided approach to locking, where communication functions (*get* and *put*) as well as lock calls are queued locally and only at the synchronization point, i.e. the call to unlock, get propagated en-block to the target node, where they are completed. It goes without saying that this method exhibits an extremely low overlap of communication and computation defying the purpose of one-sided MPI.

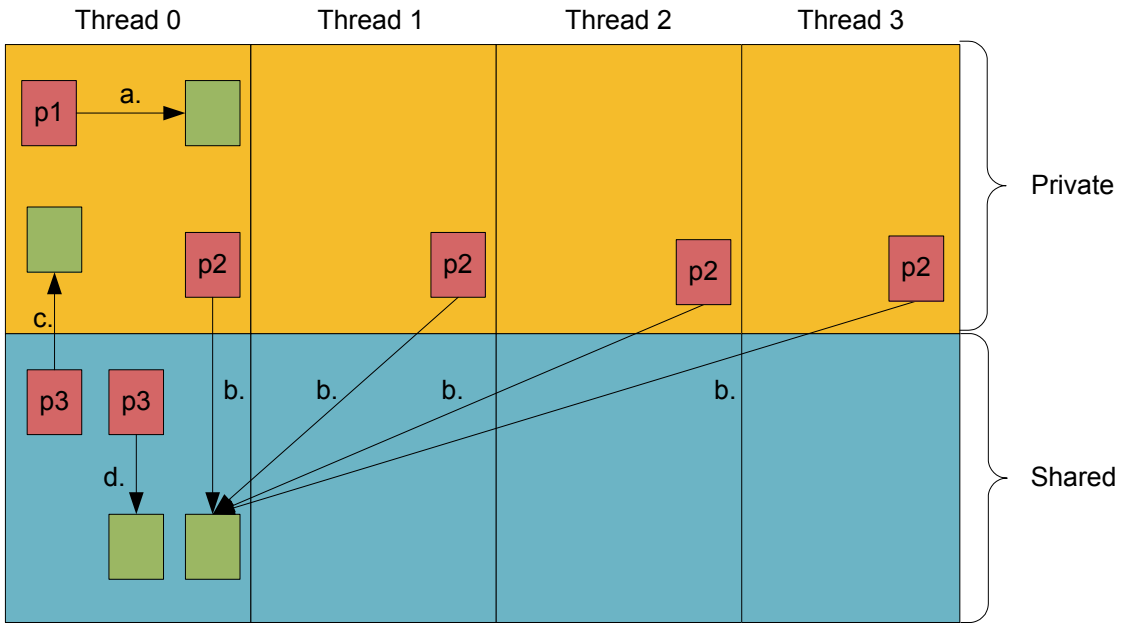
In summary, MPI-2 one-sided operations have been met critically by many parallel code developers since the performance goals were often not reached. One major reason for this is the poor overlap of MPI implementations and the poor implementation of synchronization functions both of which is paramount to reach good performance with MPI-2 one-sided codes. In [74] a synthetic benchmark is studied that simulates a program that employs nearest-neighbor ghostcell exchanges using one-sided communications. The authors found that the code runs slower using one-sided communication than if implemented with two-sided communication, regardless of the type of synchronization used on many MPI implementations. Even on a shared memory machine tested with SUN MPI, the point-to-point version outperformed one-sided operations for all tested message sizes; for short messages point-to-point was about six times faster than one-sided MPI-2. Similar results have been reported from other researchers.

An excellent implementation of MPI-2 is difficult to develop and many efforts are currently under way [73] which may prove in the future that one-sided communication is a better communication style than two-sided communication.

4.4.4 PGAS

Partitioned global address space (PGAS) systems describe a class of systems where a global address space is partitioned between the different participating threads of a parallel application. Often, the global address space is divided using two orthogonal criteria: address space can be shared or private, and address space belongs to one thread. This division is schematically shown in figure 4-9.The pointer a. is a pointer residing in the local, private space of thread 0 and points to a location in the same region. This is the equivalent of a standard C pointer. The pointer b. is shared pointer, that is every process has a copy of the pointer which points to the same shared variable. The third version is not recommended, it is a pointer that resides in shared space but points to a local variable. Although the pointer is shared, only the owning thread can dereference it. The last pointer d. is a shared pointer that points to a shared variable. In Universal Parallel C (UPC) [75] such a pointer is allocated at thread 0.

PGAS maps both to shared memory and distributed memory architectures. In any case, access to the local part of the address space is cheap for a thread regardless if the it references shared or the private memory. Access to a remote private region is forbidden and referencing a remote shared region may incur a higher cost even on shared memory systems, for example because of ccNUMA characteristics. On distributed memory architectures such an access is mapped to network access(es). To make this model attractive and provide a higher abstraction layer than message passing, it is used with a suitable programming language. Well known PGAS programming languages are UPC [75], a parallel C dialect, Tita-



a. int *p1;

c. int *shared p3;

b. shared int *p2;

d. share int * shared p4;

Figure 4-9: PGAS Address Space

nium [76], a compiled, Java based language, Co-array Fortran [77] and recently Fortress [78] and X10 [79]. All of these languages can be implemented directly on shared memory systems but require a network runtime environment for distributed memory systems. The compiler decides where to place data objects: in the address space of which thread, in the local or shared address space. The programmer can usually influence this if no at least implicitly. The programming languages have primitives to express parallelism, parallel loops etc. to implement SIMD (Single Instruction, Multiple Data) or SPMD style programs. The actual programming in a PGAS language is not the topic of this analysis, though. The necessities for the underlying runtime are of interest for the definition of a modern NIC which optimally supports the PGAS model.

Access to local data objects can be compiled to normal load/store machine instructions; remote accesses are handled by the compiler. Several suggestions exist on how to implement this. One suggestion uses remote load/store, possibly enriched by additional features such as a local cache to accelerate some operations, while others are based on two-sided and one-sided low-level operations. As an example for an existing, proven and open implementation, the GASnet library will be examined in the next sub-section. After that a short introduction follows on how memory-mapper devices with remote load/store semantics can be used to implement PGAS.

GASnet

GASnet [80] is a communication library which delivers low level services for PGAS runtime environments. It is open-source and is used by the Berkeley UPC, Titanium and Co-Array Fortran compilers and runtime environments. The GASnet API consists of two parts: The core and the extended API. An implementation for a network must provide the complete core API. For the extended API, one can use the reference implementation which is implemented in terms of the core API, or, to reach better performance, it is possible to support the extended API directly.

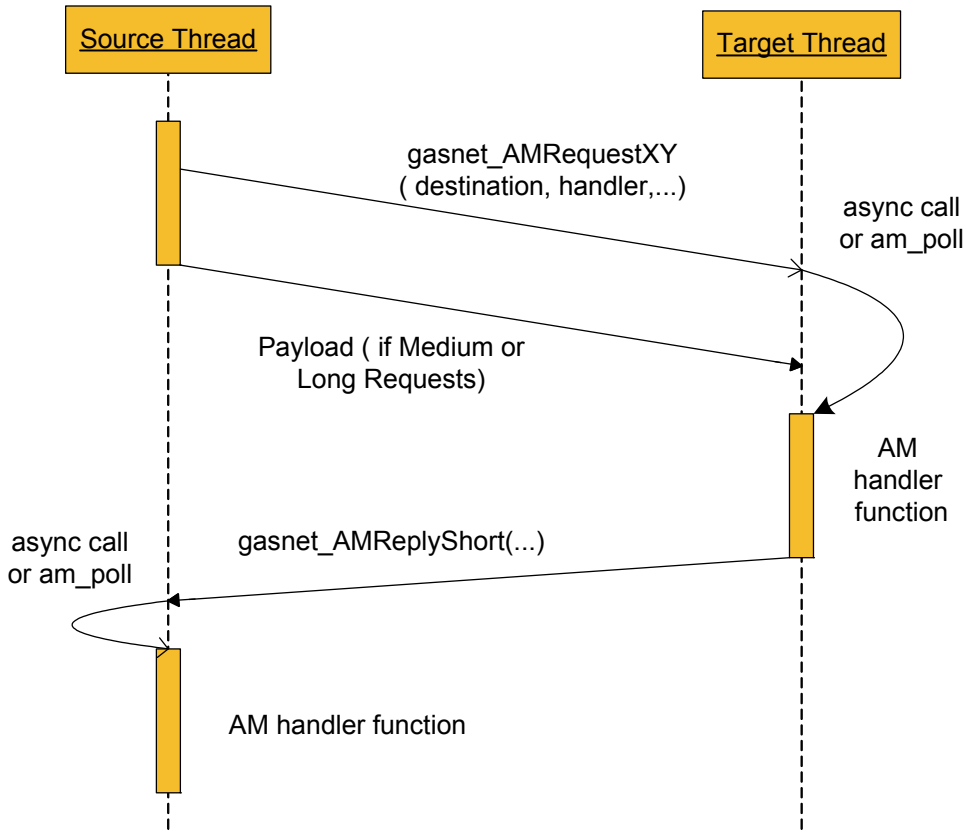


Figure 4-10: PGAS: GASnet AM-style Messaging

The initialization routine of the core API *gas_attach* exhibits barrier semantics and initializes a shared segment on all participating processes. The shared segment can be used for zero-copy transactions by later API calls without further actions. Several parameters are available to control how large the shared segment can get. In terms of the PGAS languages, a segment incorporating the whole virtual address space of the process would be best. The core communication operations are heavily influenced by the original Active Message (AM) [81] specification. The current GASnet specification (version 1.8) is most closely related to revision 1.2 of AM. GASnet uses requests that are sent to receiving threads. At the receiver the request triggers a request-specific handler, which gets a (small) number of arguments passed. The request handler may initiate a response to the sender which will trig-

ger a corresponding handler at the sender. The different handler and their respective IDs are registered at GASnet initialization time, they are thus static and consistent over the whole application. It is not allowed to reply more than once and also a reply handler is not allowed to reply again. GASnet discriminates three types of messages: short, medium and long messages. Short requests only carry the handler ID and the small number of (integral) parameter values. Medium messages can carry an amount of payload. The maximum payload can be set by the implementation but must be at least 512 bytes. Finally, long requests carry a payload which is to be delivered to a sender specified address within the shared segment of the receiver. Long requests are thus often implemented using RMA functionality, if available. The core API is complemented with support for atomic operations for handlers and some utility functions.

The extended API adds a number of *put/get* operations and synchronization operations. The RMA functions defined by the extended API may use any address on the local side (i.e. also addresses that are not within the shared segment). To implement this feature three options are pointed out by the specification authors: either use the reference implementation in terms of the core API, use a RMA enabled network which can map the complete virtual address space of participating processes or use a registration strategy such as the one described in [82] to enable good performance. Implementations are also required to support 2^{16} -1 outstanding non-blocking operations. There are a number of synchronization functions, which are used to ensure the completion of the RMA functions. Finally, the extended API defines a split-phase barrier operation.

Note, that GASnet does not specify the exact address layout of actual applications generated by PGAS language compilers. The API is also explicitly not targeted to be used by programmers directly, but to be used by machine generated code.

Memory-mapper Device for PGAS

A PGAS system using a memory-mapper device allows non-coherent access to remote address space using accesses via the device. The shared memory of all processes is mapped into the address space of the participating processes. A physical mapping possibility is enough. The physical address space of each node is exported via the device and mapped to a global physical address space. All processes can then *mmap* parts of this global physical space via memory-mapping of device address space (BAR space) into its local, virtual address space. The concept is shown in figure 4-11. Thread 0 exports a part of its virtual address space; this becomes shared space. The operating system makes these pages known to the device. The same happens on all other nodes. An access to the address space of the device is then forwarded to the appropriate address at the target node. In its elementary form no mapping tables are necessary, since all nodes export their complete physical RAM. More complicated variants need to add mapping tables. The runtime software takes care that only device addresses that correspond to physical address that are accessible to the job, can be mapped into the corresponding application address space.

In its simplest form, remote accesses become remote load/stores. It is possible to add DMA engines though, for CPU offloading and implementing of one-sided operations. This system supposedly allows PGAS applications to run with the lowest possible overhead; it is diffi-

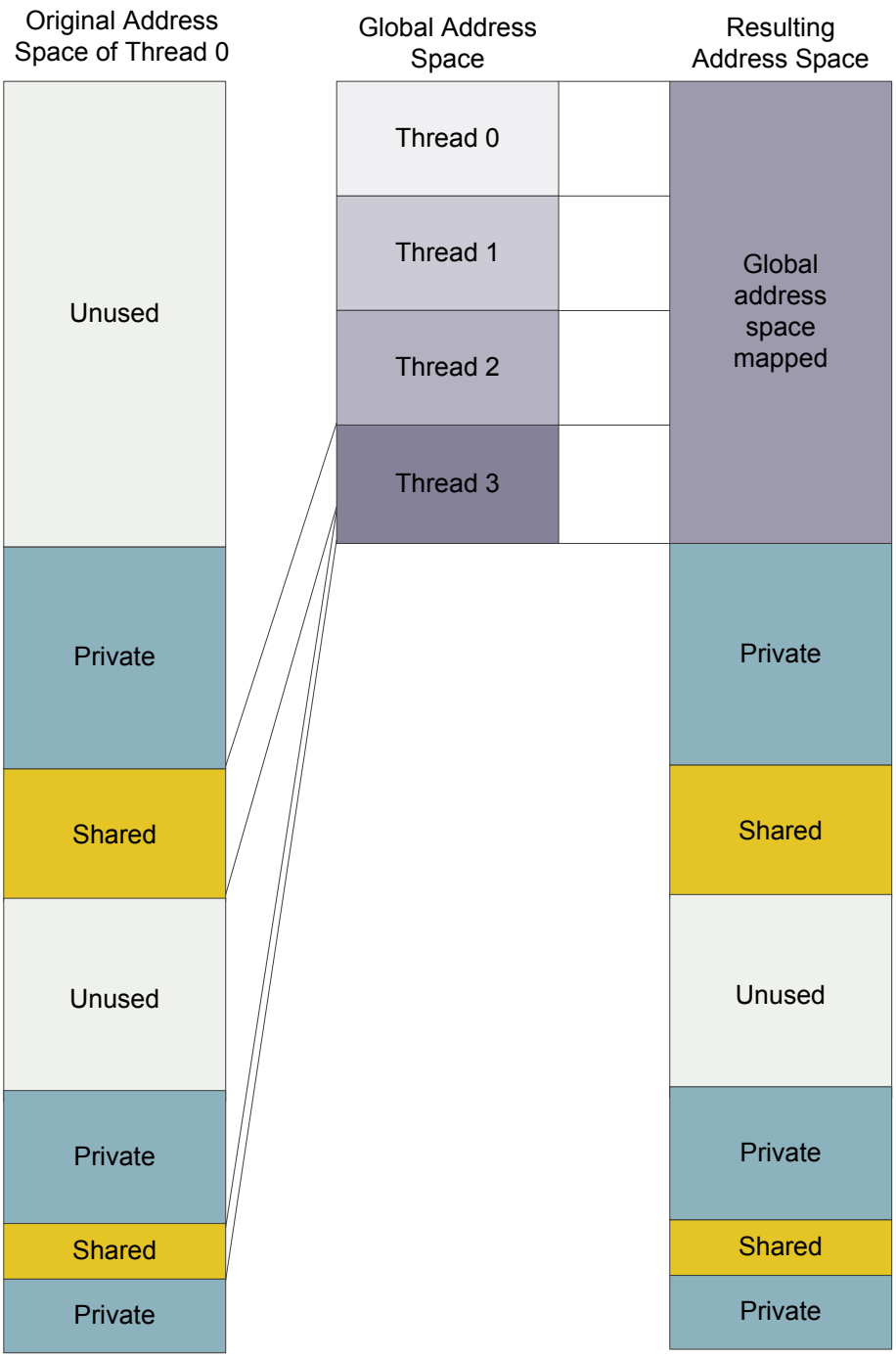


Figure 4-11: Memory-Mapper Device Address Space

cult to see though, why combined, efficient two-sided/one-sided low-latency implementations should not provide the same or better performance. Also, the inherent split-phase characteristics of more traditional approaches can be beneficial for system performance.

4.5 Conclusions for EXTOLL

From the above introduction of communication paradigms in parallel software requirements for the EXTOLL NIC can be derived. Efficient support for two-sided communication is still important, since this communication pattern is widely used. One-sided communication is also deemed an important feature. The difficulties for MPI-1 implementations, mainly because of matching and zero-copy, have been shown. Several approaches to implement one-sided messaging have been taken into account; the design space explored is shown in figure 4-12.[1]

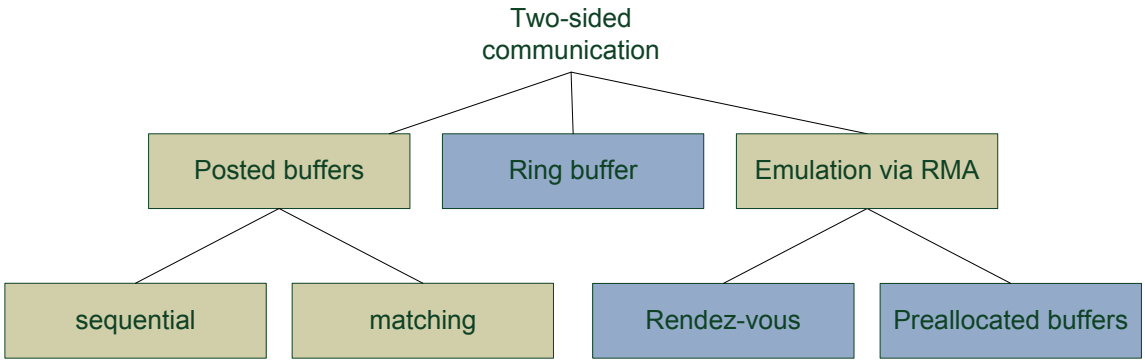


Figure 4-12: EXTOLL Two-Sided Communication Design Space

Posted receive buffers, as for example defined by the VERBS API family have been dropped for two reasons: the buffers have to have maximum message size, regardless what they actually will hold, zero copy *receive* is not possible in the general case, MPI *matching* is difficult. The second option, to use posted buffers together with an elaborate matching is discarded because of the complexity of matching algorithms that would be needed to be implemented in hardware (see section 4.4.2 for details).

Single ringbuffer implementations, as used by ATOLL, do not allow for zero-copy communication. They are very efficient for small messages though, were hardware latency is more important then DMA offloading and zero-copy protocols.

The final possibility taken into account is two-sided emulation using RMA. This solution brings a great flexibility, but also needs efficient algorithms on the software side to implement the different strategies. With the emulation strategy, medium copy protocols as well as receiver side matching, efficient memory usage and true zero-copy are possible. Also, the same hardware needed for one-sided communication can be reused for medium to large two-sided communication, increasing hardware efficiency.

1. See appendix A for an explanation of design space diagrams.

For two-sided communication, a two-way approach was chosen: efficient ring-buffer style communication for messages of limited size. This enables low-latency communication for small messages with shared queue semantics. A dedicated hardware function is used for this protocol. Larger messages have to be transported using RMA. In the MPI-1 case, medium sized messages use an explicit sender-side queue protocol without zero-copy while large messages use an optimized RMA rendezvous protocol is used. The protocol for large messages has already been described in section 4.3. Details of the protocols are covered in section 8.6.

For the one-sided communication it is required, that data transport operations can completely overlap with communication. To enable this, efficient synchronization protocols must be employed. For active-target style synchronization, efficient small-message service functions can be reused. Passive-target operations require an optimized atomic network transaction which can efficiently implement the needed lock semantics which existing designs can generally not achieve. A novel transaction protocol supporting concurrent shared and exclusive MPI-2 locking will be introduced in this work (section 7.5.4). All of the transactions are split phase and non-blocking; an orthogonal set of different completion events (called *notifications* in the context of EXTOLL) can be generated, often reducing the needed synchronization between processes, and simplifying two-sided and other higher level protocols. To support zero-copy and virtual addresses (OS-bypass), an effective address translation scheme must be developed. While GASnet can benefit from the sharing of the complete virtual address space of a participating process, this is not strictly necessary, advanced protocols may also be enough. For MPI-2, full virtual-address space exposure is not necessary. All of the units need to be carefully optimized for lowest latency including posting of operations and completion operations.

This analysis lead to the design of two functional units for communication; one for the small message service and one for RMA. First, the problems of address translation (see chapter 5), virtualization and posting of operations (chapter 6) are covered, though.

Chapter 5

The Virtual Address Space Barrier

A difficult set of problems for efficient I/O operations on current standard machines is memory addressing. User space applications address memory using virtual addresses (VA). These are translated by the memory-management unit of the CPU into physical addresses (PA). These PAs are then used to access main memory or I/O resources from the CPU. I/O devices access main memory via DMA also using PAs. Without a memory management unit (MMU) a device can not use VAs at all. For user-space I/O, PA usage of devices poses a serious problem since all addresses an application knows are VAs and it does not know the corresponding PAs. It is also not allowed to know them for obvious security reasons. They are only known to the operation system kernel. On some architectures devices do not use physical addresses but bus addresses which are translated by the I/O bridge into physical addresses when accessing main memory. An example for such an architecture were several SPARC architecture machines [83]. But even with these machines the problem essentially remains the same, only now it is bus addresses and not physical addresses.

For networking applications at least three questions arise, namely:

- How to most efficiently communicate the address of a (local) source buffer to the network hardware?
- How to handle remote addresses respectively buffers?
- And how-to handle unmapping/freeing of memory previously allocated to an application?

In [83] a design space analysis for the problems was performed. While most of the performance data is completely outdated by now, most of the requirements still hold true today so that for EXTOLL the following requirements have been identified:

- The design should be as simple as possible.
- The translation must enable access to all of the physical memory of a node.
- The design should be high-performance, specifically it should exhibit
  - low-latency translation using a *translation look-aside buffer* (TLB),
  - low TLB miss-latency for a design with limited TLB performance (FPGA implementation), which also is useful to give relatively good worst-case performance for workloads that exhibit poor locality, and
  - low memory registration/deregistration times to support efficient dynamic memory usage patterns.
- Support for self-virtualized device with the possibility to support thousands of contexts and user-space access.
- Flexibility in the integration of different types and number of TLBs.

- Streamlined design to support efficient implementation on different target architectures as well as be able to reach high clock frequencies.
- On-device protocol between address translation unit and the functional units (FUs) for translation and invalidation requests and responses.

To thoroughly understand and address the problem, the current state of address translation is studied carefully. This leads to deeper understanding of the existing solutions, their advantages, possible pitfalls and their short-comings. It is also necessary to gain a system level view on the problem; so both software, CPU based and device oriented techniques are considered. This detailed analysis also reveals why current solutions fall short of providing a design which meets the requirements for a high-performance virtualized device, as specified above. With the statement of requirements and the insight from current systems, the innovative, streamlined design described later in the chapter is developed.

The remaining chapter is organized as follows: First, the current state of the art in handling of physical versus virtual addressing will be thoroughly analyzed, giving the reader an understanding and insight of current implementations, trends and their implications on system performance and capabilities. This part is followed by a design space analysis for address translation for EXTOLL. The chapter closes with the presentation of the ATU architecture, its implementation and the achieved performance.

5.1 State of the Art

5.1.1 X86-64 Processor MMU

AMD64 processors like the Opteron family as well as Intel X86-64 processors all implement a hardware MMU to translate memory accesses of applications into physical addresses. All modern processors also implement some form of a TLB to accelerate translation from VAs to PAs. The methods used for CPU MMUs and TLBs are a well known field within the computer architecture, details can be found for example in [64].

X86-64 processors use a 4-level page walk to translate a VA. Figure 5-1 illustrates the process schematically and shows the four main memory accesses necessary to perform the translation. This diagram is for long-mode (64-bit mode) of a K10 architecture AMD processor (codename *Barcelona*) with a 52-bit physical address width. Other members of the AMD64 CPU family may implement a smaller physical address width (for example 40 bits on the original K8 Opteron cores). Different translations are used, if the processor is operating in legacy mode (32-bit). Also, the translation shown is for a page size of 4 kB, the most common scenario. AMD64 processors also support other page sizes, notably 2 MB and 1 GB. For 2-MB pages, the page offset is 21 bits wide and the page-directory table becomes the last translation level effectively removing one level; for 1-GB pages, the page offset becomes 30 bits and the page-directory-pointer table becomes the last translation level leaving a two level page walk.

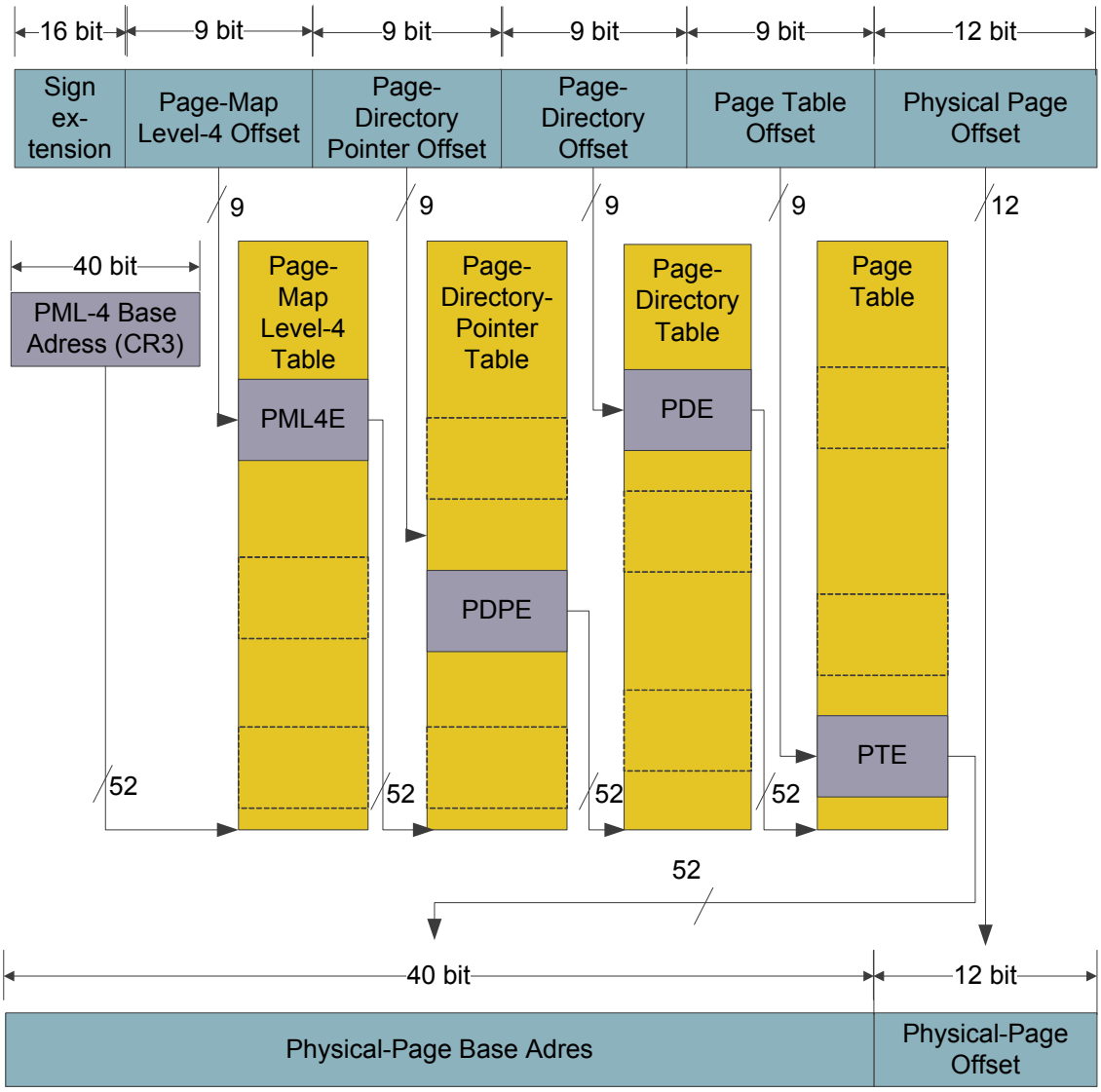


Figure 5-1: X86-64 Page-Table Walk

Since main memory accesses are expensive, TLBs were introduced into CPU designs to reduce the number of table walks. Also, translation tables are typically cached, which can also help to accelerate address translation. Since address translation can have a significant impact on overall system performance, the CPU's TLBs have been improved and increased in size over time. A K10 AMD CPU features a fully associative first level instruction TLB with 32 entries for 4-kB pages and 16 entries for 2-MB pages. The second level instruction TLB is 4-way set-associative and features 512 entries for 4-kB pages only. On the data side, the first level TLB is fully associative and can hold 48 entries (either 4 kB or 2 MB). The second level data TLB has a capacity of 512 standard page entries, again 4-way set-associative. There is an additional second level two-way set associative TLB for 2-MB pages with a capacity of 128 entries [84].

For the future, TLBs may become ever more important. The advent of virtualized computing in standard PCs and servers caused the Hypervisors to perform a large part of the memory translation management in software to hide the real, physical memory from the guest operating systems (*shadow paging*). Hardware virtualization of processors now introduced the concept of *nested page tables* [85], where the hardware can actually understand the notion of different OS instances and the table walks are organized accordingly by augmenting the table walk hardware. In essence this means, the *guest virtual address* is translated to a *guest physical address*. The guest OS has full control over the guest page tables. The resulting *guest physical address* is then translated into a *real physical address* using a second table walk using *nested page tables*. The complete process of translating a guest virtual address into a real physical address is also called 2-d page walk in [85], because each reference to one the 4 table levels or the guest translation spawn a nested page walk. Such a table walk can cause up to 24 main memory accesses with an according effect on performance. Still, the CPU vendor hopes to increase the performance in respect to the software solution.

5.1.2 Classical Devices and the Linux DMA API

Traditionally, devices are controlled from kernel-space. In kernel-space it is possible to translate virtual into physical addresses using software. The PAs can the be transferred to the device. The Linux kernel provides a rather sophisticated library of functions to manage the addressing of memory in a portable way when interacting with devices in the traditional method. These functions are collectively known as the DMA API or DMA-mapping API [86].

The API enables completely portable device drivers. The device driver neither needs to worry if the architecture uses explicit bus addresses or only physical addresses, nor if there are restrictions on the memory that can be addressed by devices[1].

The API generally distinguishes two basic cases: using large DMA buffers and using small DMA buffers. To use a large, consistent DMA buffer, the function *dma_alloc_coherent()* is used. This function allocates a physically contiguous memory region of the given size and returns both the kernel virtual address to reference this memory from software and a parameter named *dma_handle* of type *dma_addr_t*. The usage of a *dma_addr_t* instead of a pointer, makes it possible to handle different architectures transparently to kernel drivers: irrespective if the system uses physical addresses or an IOMMU (I/O memory management unit), the API calls to allocate a DMA buffer are always the same. It is also guaranteed, that the content of the *dma_handle* can be directly passed to the device in order to set-up a DMA transfer[2]. The minimum size for such an allocation is often one page. The function *dma_free_coherent()* frees such a DMA buffer again, after it has been used. Handing out kernel virtual addresses and DMA address at the same time, the time of allocation of the

1. A very common problem here are devices, that can address memory only with 32-bit wide addresses, but are used on modern systems with more than 2-4 GB of main memory.
2. Coherent memory means, that the memory takes part in the cache coherency protocol. There are also non-consistent versions of the DMA API functions available; the Linux developers discourage their usage other than in special, often legacy cases, though.

buffer, saves later page table translations. It used to be the case, that the physical address of a page could be very simply calculated from the linear kernel address by just subtracting a constant; this is no longer the case since the kernel often does not even map the complete physical address space into kernel virtual address space.

For drivers that use many DMA buffers of small size, the kernel offers the DMA pool API. Using the function *dma_pool_create()*, a pool of memory for smaller buffers is created. Using the function *dma_pool_alloc()*, a small buffer from the pool (i.e. usually smaller than one page) can now be allocated from the pool and returned to the pool using *dma_pool_free()*. The advantage is that allocating from the pool can potentially perform much better, and the memory usage can improve, since one large memory region (the pool) can serve multiple buffers.

One other important feature of the DMA API, which is essential for portability, is the support for different DMA address limitations. Using these functions it is possible to make sure, that *dma_alloc_coherent* allocates only memory that is actually accessible from the device.

There are also functions that can map an existing Linux kernel virtual mapping onto a *dma_addr_t*. Using these functions makes it necessary for the driver programmer to worry about alignment, DMA accessibility (the DMA mask), and physical contiguity of the memory. Additional functions exist to map single pages (*dma_map_page()*) or whole scatter lists (*dma_map_sg()*). The scatter-list support is smart in the sense that it can potentially combine physically adjacent entries into one segment. Scatter-lists are especially important and used extensively by block devices and also for some network devices.

To map user application memory for I/O, Linux offers the *get_user_pages()* function. It returns an array of *struct page*s which can then be used to get the physical address. The pages are also implicitly locked in memory. By using *put_page()* a page can be unlocked again. This interface is used by many drivers for zero-copy I/O. There is another interface for zero-copy block-device I/O (*bio_map_user()*).

This API is reasonably simple and covers most cases for traditional I/O while preserving full portability.

5.1.3 Mellanox Infiniband HCA

To support user-space initiated, zero-copy *put/get* and *send/receive* operations the Mellanox HCA supports address translation. An HCA uses tables to store the context information. Each context, which can be used by one process to communicate using the HCA, includes queue-pair states, memory information, and firmware state. To manage memory a table of registered memory regions (pages) called memory translation table (MTT) is used. The HCAs implement on-chip caching of contexts and memory translations. Older HCA implementations from Mellanox used a dedicated DRAM on the card to store the context information whereas the work queue and all data buffers were located in main memory. Mellanox HCAs support millions of Queue Pairs, but a more limited amount of contexts. A maximum of 510 possible user contexts was measured[1].

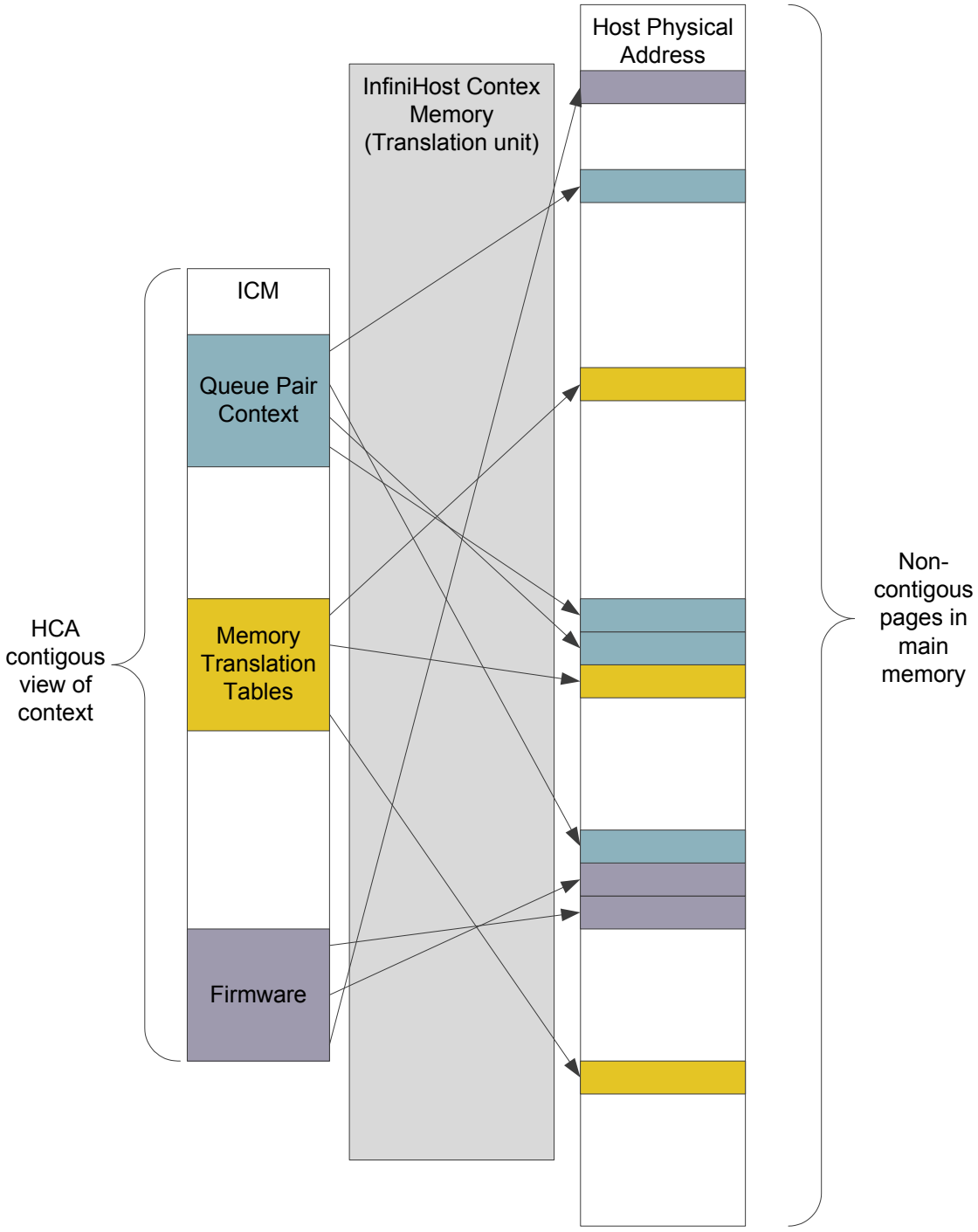


Figure 5-2: Mellanox Context and Translation Architecture [87]

1. Mellanox ConnectX SDR HCA, firmware version 2.3, Linux 2.6.24, OpenFabrics 1.3, 4x Quad-core Opteron system (2.2GHz)

So the amount of DRAM on the adaptor card limited the number of contexts respectively their size, for example the size of the MTT. Later, so called “memfree” HCAs move the complete context information to host memory. The HCAs also support context information in physically non-contiguous pages. The necessary logic to support this is called InfiniHost Context Memory (ICM) and works similar to a Graphics Aperture Remapping Table (GART, section 5.1.8), i.e. remaps contiguous addresses to non-contiguous physical addresses. This layer seems to replace the memory controller logic on non-memfree HCAs. The actual ICM table also resides in main memory. Thus, if no information is currently cached on the device, an IB transaction involves quite a number of main memory accesses:

- fetch WQE (Work Queue Entry) from main memory,
- fetch context from memory (potentially cached),
- perform translation using the MTT (potentially cached),
- and fetch actual data from main memory.

In [88] the effects of caching on the performance of a “memfree” HCA are described. While standard ping-pong test do not suffer, it is possible to implement cache trashing tests, so that the context needed for the next transaction is never loaded into the on-chip context cache. [87] claims that the *RDMA write* based latency only increases from about 9 µs to 11 µs when moving to a memfree design in the case contexts are not cached, using a PCIe memfree InfiniHost III HCA. The presentation also shows that the half-roundtrip latency is increased by 4-6 µs by cache misses.

The kernel driver manages these tables and it is necessary to register/deregister memory as it is needed. Memory registration is an expensive operation since it involves at least a system call to enter kernel mode. An analysis of the relevant part of the Infiniband low-level driver in the Linux kernel[1] shows that registration of user memory uses the *get_user_pages()/put_page()* interface to pin and translate virtual user addresses in one step. The resulting translations are then assembled in a scatter/gather list. The command to load and register MTT entries is written to the HCA using memory mapped I/O. The HCA reads in the translation entries using DMA and finally completes the action, which the driver polls for.

In [89] the memory registration costs of previous generation PCI-X and PCIe based Infiniband HCAs are analyzed in detail. The IB stack performs the usual three sequential operations to register memory (note that this uses an older kernel API based on a different set of functions):

- pin the requested pages using *mlock()*, fault in the pages if not present
- translate the virtual address
- transmit them to the HCA

1. Linux kernel version 2.6.24

Interestingly enough, the most time consuming factor was the address transfer to the HCA with more than 100 µs for a single page[1]! In contrast, the *mlock()* function only took ~2µs and the actual translation ~2µs, too[2]. Beyond buffer sizes of 256 kB the run time became dominated by pinning the pages. The high communication runtimes are caused by the HCA only reacting after this amount of time.

The duration of the *ibv_reg_mr()* function was benchmarked on a current Mellanox Connect X HCA[3]. In figure 5-13 the results are collected together with the results from the EXTOLL ATU unit. Notice, that constantly the minimum registration time, i.e. for one page, starts beyond 50 µs indicating that even with the newest HCA generation and software revisions, pinning of memory is a very expensive operation on Mellanox hardware. The time to deregister memory (*ibv_dereg_mr()*) was also benchmarked; the results are shown in figure 5-14. Again it must be noted that deregistration times start beyond 50µs. Repeated pinning of smaller sizes does not exhibit the peaks visible in the two graphs. The reason for these peaks is unknown.

These very high memory (de-)registration times explain the large number of papers concerned with optimizing the way memory is managed for IB based middlewares and applications [90][91][92]; an analysis of the impact of buffer re-use can be found in [93] and [32].

5.1.4 iWARP Verbs Memory Management

iWARP [94], as the specification of RDMA over TCP/IP is also known as, uses a Verbs API closely related to VIA or IB [31]. The memory management employs the notion of memory regions and memory windows.

Memory regions are identified by a steering tag (STag), a base tagged offset (base TO) and a length. The base TO is the first virtual address of the region. Each memory region is associated with a physical buffer list (PBL) through the STag. Memory regions are registered with the kernel level driver, which translates the virtual address to a physical address, builds the physical buffer list by translating involved pages, and hands the list to the NIC while also associating it with an STag. The maximum size of the PBL is NIC specific. The STag is composed of an 8-bit consumer key which can be used arbitrarily by the user and a 24 bit STag index, basically enabling up to 2^{24} different memory regions. There is a special STag of zero which can be used by kernel-level clients like for example filesystem applications to directly post physical addresses in work requests to the NIC.

1. Measurement System: memfree PCIe Infinihost III Adapter, Opteron 1.8 GHz, Linux 2.6.10, Mellanox InfiniBand Gold Edition Package Software
2. Actually, the graph for the memfree HCA reports a translation latency of 22 µs for single page. This seems a little too high and is probably a measurement error. This thesis is backed by the results for the non-memfree HCA of 2 µs. For this reason, the lower number has been reported here.
3. Mellanox ConnectX SDR HCA, firmware version 2.3, Linux 2.6.24, OpenFabrics 1.3, 4x Quad-core Opteron system (2.2GHz), 16 GB RAM

iWARP Verbs also uses memory windows to enable more dynamic access control to memory regions. Memory windows are associated with Queue Pairs (i.e. connections). This feature allows to use one memory region for receive buffers and partition them between different clients, so that one client cannot interfere with other clients. In all work requests, memory is identified to the NIC using the STag (to identify the memory region or memory window), a tagged offset to specify the offset of the starting address within the region/window and a length. All of the iWARP specifications is, by design, relatively general to support different NIC implementations.

5.1.5 Quadrics

The Quadrics network adapters feature an embedded processor and SRAM on the card. Some installations used part of the SRAM as an IOTLB and managed the contents from host kernel software. This is a patched Linux kernel which promotes unmapping of pages to the SRAM on the adaptor. The patches to the kernel to support to transparent page translation by the adapter have been rejected from mainline kernel so far (see also section 5.1.12).

The Quadrics website [95] states that there are actually two variants being used, one based on callbacks/notifier functions being called whenever the mapping of an application changes so that Quadrics driver software can update the translation structures on the adapter; the other variant uses memory registration techniques like IB does, too, and does not require a patched kernel.

5.1.6 Myrinet MX

The Myrinet eXpress (MX) [96] suite is a combined library, kernel driver, and firmware approach, where the firmware part runs on the LanAI RISC processor located on Myrinet adapter boards. Myrinet boards traditionally feature on-board SRAM which is accessible to both the LanAI and the host processor. With the current MX software, Myrinet uses direct, zero-copy DMA for large messages (> 32 kB). Smaller messages use copy protocols.

Closer examination of MX reveals that MX employs an approach where “DMA windows” are implicitly registered when needed. This means the kernel pins the pages associated with a communication operation, and then transfers the resulting translations to the adapter using several PIO transactions. The translations are associated with an ID, which is passed to the user application. It can then initiate as many operations on this buffer as it wants; it just passes the window ID to the adapter which can look up the translation in on-board SRAM. Since the on-board SRAM is quite limited (2 MB are reported), the number of DMA windows is limited. The registration can be freed after the operation is finished. MX can also employ a sort of registration cache, where windows are only removed if they become invalid or the resources are needed for other windows. In registration cache mode only straight matches (address and length match exactly) can reuse the same registration (this may change in future revisions, since this seems to be a software issue). Using GLIBC hooks, MX tries to invalidate registrations if the user space memory mapping changes. So, in case of a registration cache hit, large send operations can proceed as pure user-space

operations, else, the kernel is invoked to first provide a DMA window for the buffer. MX does not provide complete RDMA, the above described scheme is only used for a zero-copy two-sided protocol.

In [97] a software solution to emulate the MX API on standard Ethernet hardware is presented. This is notable because it employs kernel-based zero-copy techniques at the sender side. The developer reports a latency of 200 ns for the necessary call to *get_user_pages()* on an Intel Xeon E5345.

5.1.7 SciCortex

SciCortex uses a proprietary network for their family of MIPS processor based supercomputers [98]. The NIC, called *DMA Engine*, supports RDMA operations via a microcoded engine. Access to the application's VA from user-level communication is implemented via an address translation table called *Buffer Descriptor Table* (BDT). Applications can register memory, i.e. buffers, with the *scdma* Linux kernel driver. The driver then pins the memory and inserts an entry (a Buffer Descriptor or BD) into the BDT for each page referenced by the memory region. The white paper states that BDs are invalidated when the kernel unmaps a virtual page. It is unclear how or if this is actually performed. The BDT is only accessible from kernel space. The memory registration returns an index into the BDT which is used by subsequent commands to the DMA engine.

5.1.8 Graphics Aperture Remapping Table

The Graphics Aperture Remapping Table (GART) is a method to translate discontinuous physical address space into continuous physical address space. It was first introduced for graphics applications with the Accelerated Graphics Port [99]. In essence this creates an alias of a physical address within a window of physical address space called the *aperture*. Such translation tables can be found in Intel integrated graphics chipsets, many other chipsets and PEG/AGP bridges and, most important within this context, in all AMD64 northbridges. In [100] the programming interface for the AMD GART is described which uses four PCI configuration space registers of PCI function 3 of the northbridge. They are listed and described in table 5-1.

The size of the aperture can either be 32, 64, 128, 256, 512, 1024 or 2048 MB. The translation table is a flat table in contiguous physical main memory holding one 32-bit entry for each 4-kB page frame of the aperture. Each entry specifies a valid bit, a coherent bit (if the entry is to be held coherent with CPU caches) and the physical address, which backs this part of the aperture. Only main memory addresses below one terabyte (i.e. lower 40 address bits) can be remapped even on systems with larger physical address space (K10 and newer Opterons).

Figure 5-3 shows an example for a GART operation. The upper half of the displayed physical address space is used for memory mapped I/O mappings, while the lower part is actually used by physical DRAM, the main memory. Using the *GART Aperture Base register* and the *GART Aperture Control Register*, an aperture is established in the memory mapped I/O

Register Offset	Name	Description
0x90	GART Aperture Control Register	enable/disable the GART and set aperture size
0x94	GART Aperture Base Register	set the physical base address of the aperture
0x98	GART Table Base Register	set the physical base address of the translation table
0x9C	GART Cache Control Register	trigger GART TLB flushing and also for translation error reporting

Tabelle 5-1: AMD GART Registers

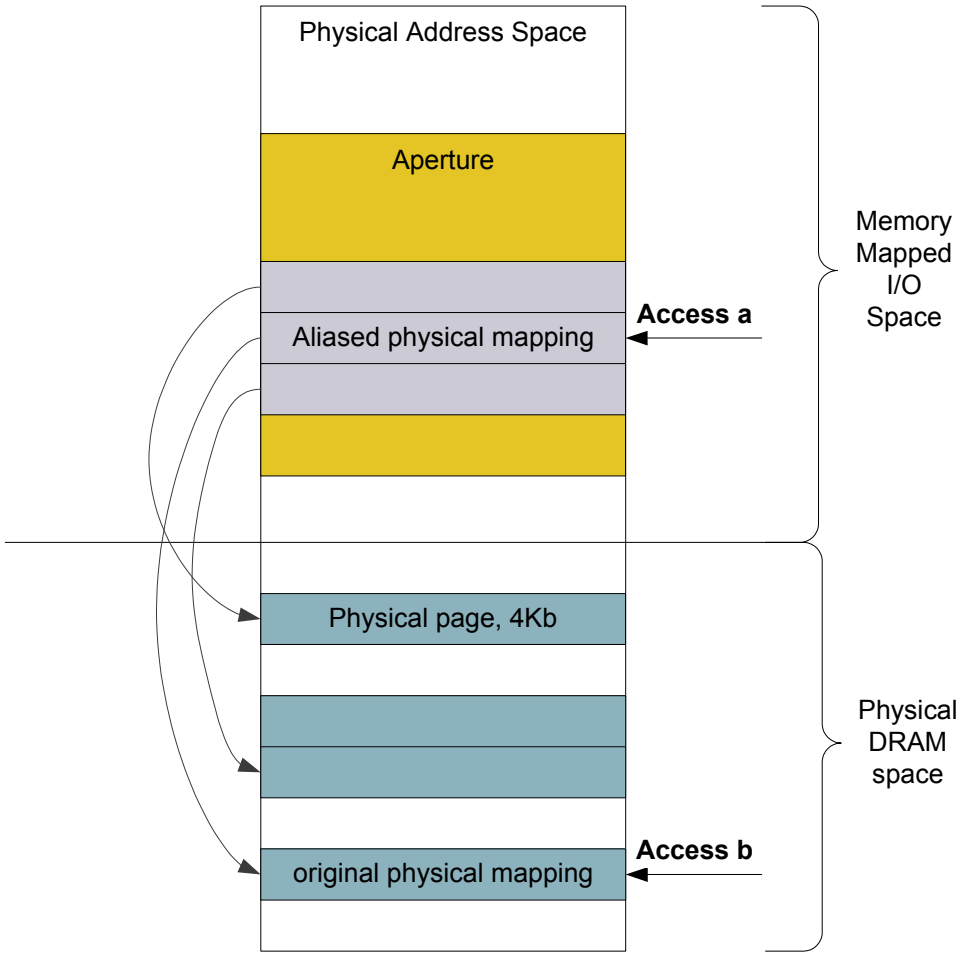


Figure 5-3: GART Remapping

space. For each 4 kB of address space in the aperture, one entry is used in the GART table. The example shows the mapping of three pages from their respective mapping in the GART to their backing, physical memory location. The two memory references shown, *access a* and *access b* actually address the same physical memory location.

The GART is widely used to enable physically discontinuous buffers for graphics processing. User space buffers (virtually contiguous) can be mapped so that they are contiguous for the DMA engine of the graphics controller. The user can then send commands to the GPU (graphics processing unit) using only offsets into the region. The offsets are the same, whether the application accesses the memory using its virtual base address plus offset or if the device accesses the memory using the aperture base address and the offset. The GART can also be used in other applications, for example Cray uses it on their XD-1 range of machines to make more then 2 MB of DMA buffer available for applications using the integrated FPGA [101].

Generally, the GART can be used whenever it is necessary or useful to access large chunks of physically contiguous memory from a device, while it is not feasible to allocate the memory in this way on the machine. While a system is up and running it becomes increasingly difficult to acquire large chunks of contiguous memory; on Linux one is limited to 2 MB in one allocation call even right at boot time. So, this mechanism can be used to increase the feasible DMA area of device functions like the ATOLL Hostport (see also section 2.2.1) or the VELO mailbox (see section 7.4). For ATOLL it actually would enable a clever zero-copy send concept.

5.1.9 IBM Calgary IOMMU

The IBM *Calgary* IOMMU is used on x86-64 (X-series with *Hurricane* chipset) machines from IBM as well as some p-series machines. Its primary use is to enable 32-bit devices to efficiently function in 64-bit environments and to provide DMA isolation which is especially useful in a virtualized environment.

As can be seen in figure 5-4, Calgary uses a table in main memory which contains *Translation Control Entries* (TCE). Each entry holds the host physical address as well as permission bits for the device. A TLB caches TCEs on chip for faster operations. Whenever a device behind the IOMMU, which is located in a PCI-X bridge, accesses the system the given address is used to address a TCE and the resulting translated address is used for the up-stream transaction if all protection checks succeed.

The Calgary IOMMU is fully supported on Linux and the driver has been merged in version 2.6.21 into the main kernel tree. The driver code interacts with the DMA API to enable transparent operations. A driver for a device behind the IOMMU calls a DMA API function to allocate a DMA buffer, for example. Since the device is behind a Calgary IOMMU, a TCE is allocated for the new buffer. In this case, the *dma_handle_t* returned by the DMA API functions represents not a physical address but the one used on the subordinate bus to address the correct TCE, i.e. an index into the TCE table. The address space accessible with the Calgary IOMMU is a full 64-bit address space. The table size can be configured in eight

steps from 64 kB up to 8 MB in powers of two. Thus the minimum sized table provides for 8192 entries, while a Calgary IOMMU with a maximum sized table can handle at most 2^{20} entries corresponding with 2^{20} x 2^{12} bytes = 2^{32} bytes = 4 GB of DMA addressable memory at one time.

In [102] some performance aspects of this IOMMU are discussed. One major drawback is the lack of the availability of a command to invalidate a single entry of the IOTLB. Instead there is only a command available that flushes the complete TLB. This means deregistration of memory becomes an expensive operations; even worse, other subsequent DMA operations are adversely influenced.

The COC925 also known as U3 northbridge that was used in Apple G5 machines and is used on IBM JS20/21 blades features an IOMMU called *DMA Address Relocation Table* (DART) which is similar to the Calgary TCE table. The main difference is that only validity of entries and no access rights are tracked. The maximum address space that can be translated is 36-bit.

5.1.10 AMD IOMMU and Intel VT-d

The AMD IOMMU specification [103] defines a full IOMMU to be used mainly in server systems. AMD identifies DMA security, support for direct device access by virtualized operating systems, and substituting of the GART as main usage models for its future implementation. Figure 5-5 shows a block diagram of a system with one CPU, two southbridges, an IOMMU and several peripheral devices. Southbridge 1 implements an IOMMU with on chip TLB. All DMA requests from downstream devices will be translated by this IOMMU. Requests initiating in integrated peripherals will also be translated since they are logically downstream. Requests initiating from southbridge 2 or its down-stream devices will not be translated at all, since no IOMMU is located on the way to the CPU hostbridge. This also implies that any directly connected device (i.e. a HyperTransport device which is connected directly to a CPU) cannot use the service of an IOMMU. The IOMMU specification also arranges to be compatible with PCIe Address Translation Services (ATS), which is necessary if a down-stream device implements an IOTLB (device 3 in the diagram).

A full table walk first accesses the device table followed by up to six levels of page tables accounting for a total of up to seven main memory accesses for a single translation. IOMMU page walks are a generalization of the AMD64 page walk. If certain rules are followed, the IOMMU and CPU MMU can share the same set of page tables. An IOMMU can use a full 64-bit address space (other than the CPU). The introduction of the next level field into the page table entries allows individual levels to be skipped. Furthermore, the size of the page, which an entry references, can be individually given.

Requesters (i.e. devices) are identified by a requester ID. One set of translation tables exists per requester ID. For PCIe packets, the requester ID is formed by the *Bus:Device:Function* (BDF) triple of the PCI device. This gives a 16-bit ID, albeit if not misused, only up to 8 contexts per device are possible since the function field has a width of 3 bits. A virtualized device with more contexts would have to allocate the according number of device IDs and,

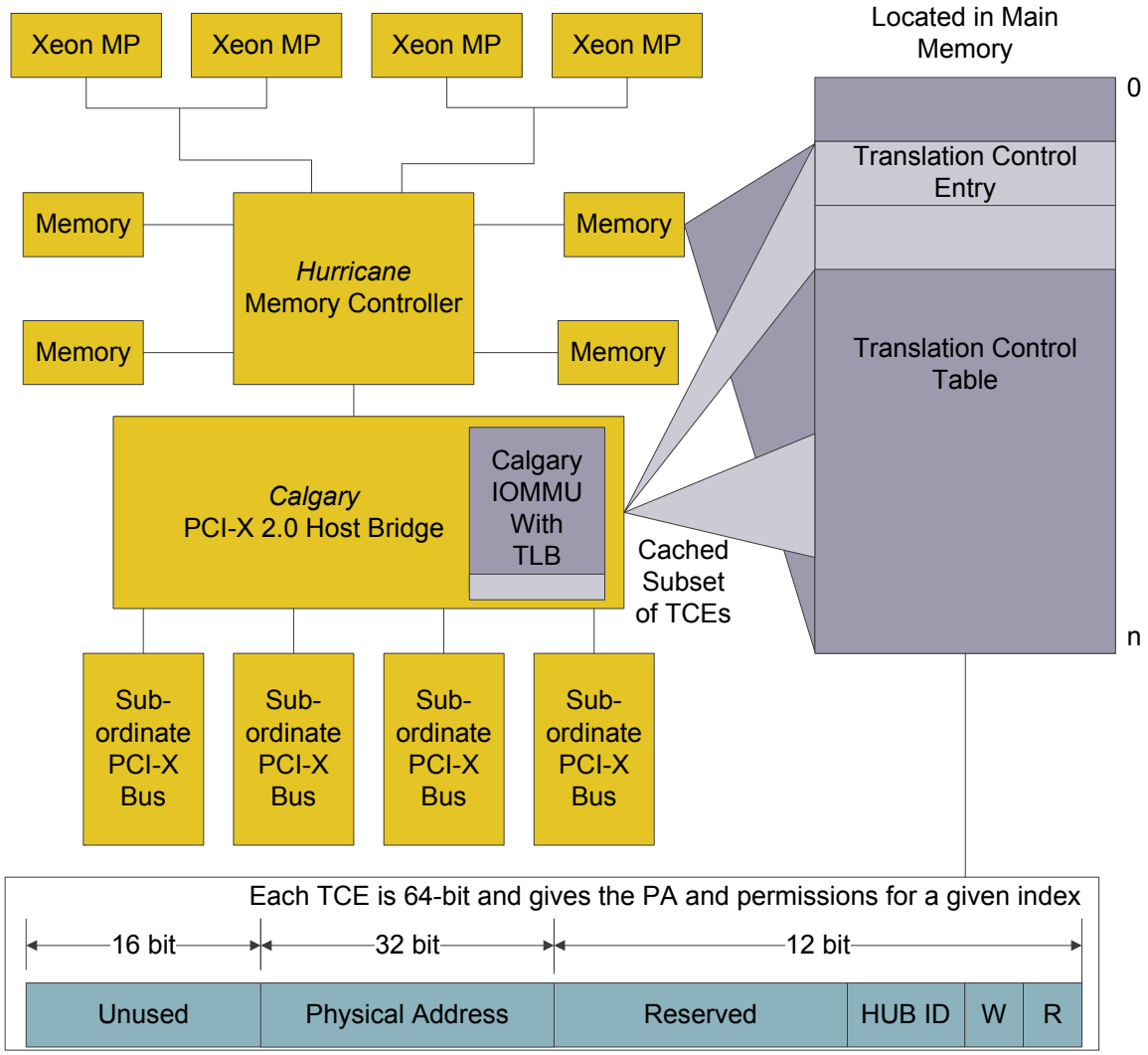


Figure 5-4: IBM Calgary IOMMU Architecture

subsequently, would be seen as a number of devices by the whole system. In a HyperTransport fabric, the requester ID is formed by the 5 bit UnitID and the 8 bit bus number of the device, thus further reducing the context/device address space. Logically, a requester ID is now used instead of a process ID for the table walk. The necessary data structures, which are stored in main memory (figure 5-6), are:

- A device table. The device table entry is indexed by the requester ID of the request to be translated. Each device table entry has a size of 256 bits and (besides other bits) encodes the base address of the top level translation table for the page tables of this device and also, the base address for an interrupt remapping table.
- Up to 6 levels of page tables. The number of page table levels is not fixed. Each page table uses one 4-kB page and features 512 entries of 64 bit size. The 3 bit next-level-field of an entry specifies if another level follows (encoded as 000 or 111) or if this is the last table in the sequence (all other encodings). The next-level-field also selects the corresponding 9 bit block from the device address to be used as index for the next table. If levels are skipped,

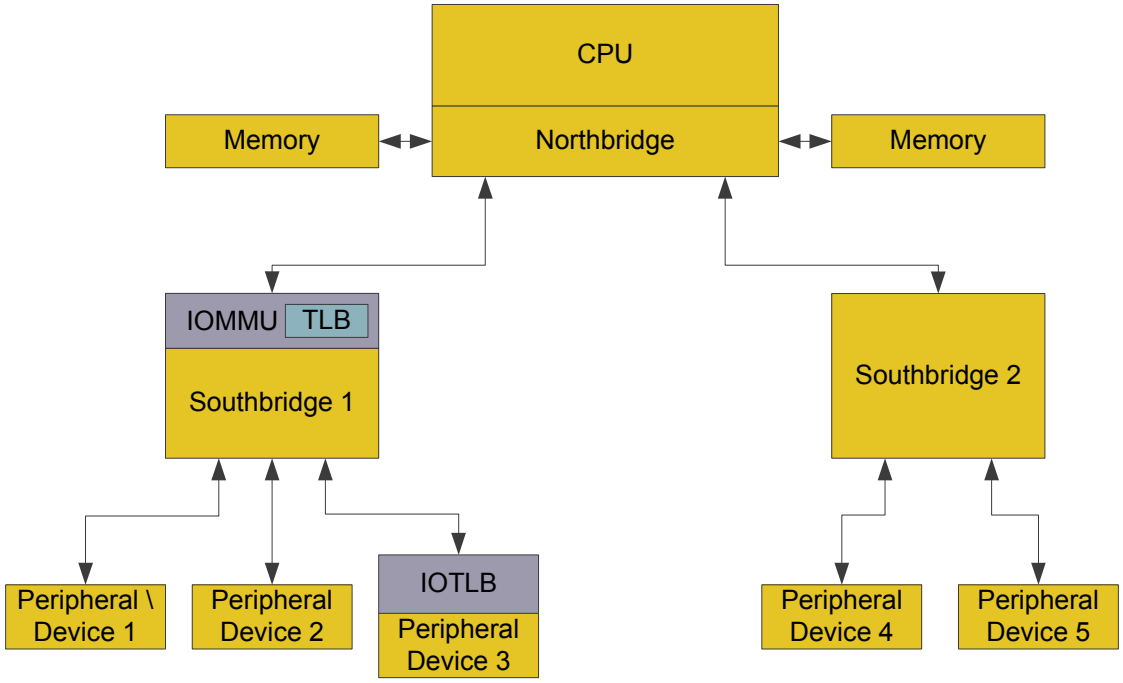


Figure 5-5: AMD IOMMU System Architecture

the skipped 9 bit device address blocks must be zeroed. The IOMMU specification assigns a default page size per level (which is given by the remaining lower bits of the device address, i.e. 2^{12} bytes for 1st-level, $2^{9+12}=2^{21}$ bytes for 2nd-level etc.). Additionally, the default page size can be overridden by an base address encoding like it is used by the PCI address translation services (see section 5.1.11).

- A command queue. Implemented as a circular buffer, into which commands are inserted by system software and from which commands are consumed by the IOMMU. The buffer is managed using registers in the MMIO PCI space of the IOMMU.
- An event log queue. Implemented as a circular buffer, into which event entries are inserted by the IOMMU and from which entries are consumed by system software. The buffer is managed using registers in the MMIO PCI space of the IOMMU.
- Interrupt remapping tables. An interrupt remapping table holds entries to remap or translate the interrupts originating from a device to another interrupt.

All IOMMU commands are 128-bits in size with an 4-bit op-code and two 60- respectively 64-bit sized operands. Commands are fetched in order from the command queue, but the specification allows the IOMMU to execute several of them in parallel; the COMPLETION_WAIT command is specified for synchronization purposes. The other commands are rather straight forward: INVALIDATE_DEVTAB_ENTRY invalidates a device table entry upon modification, INVALIDATE_IOMMU_PAGES invalidates translation table, INVALIDATE_IOTLB_PAGES causes an invalidation ATS request to downstream IOTLB devices, and INVALIDATE_INTERRUPT_TABLE invalidates interrupt remapping table entries.

The IOMMU reports errors using the second circular buffer, the event log queue. It is possible to configure the IOMMU to trigger an interrupt upon adding an entry to the queue. Reported errors include illegal table entries, illegal accesses by down-stream devices, ATS timeouts (down-stream devices not responding in time to an invalidation request). If the queue ever overflows event logging is automatically disabled.

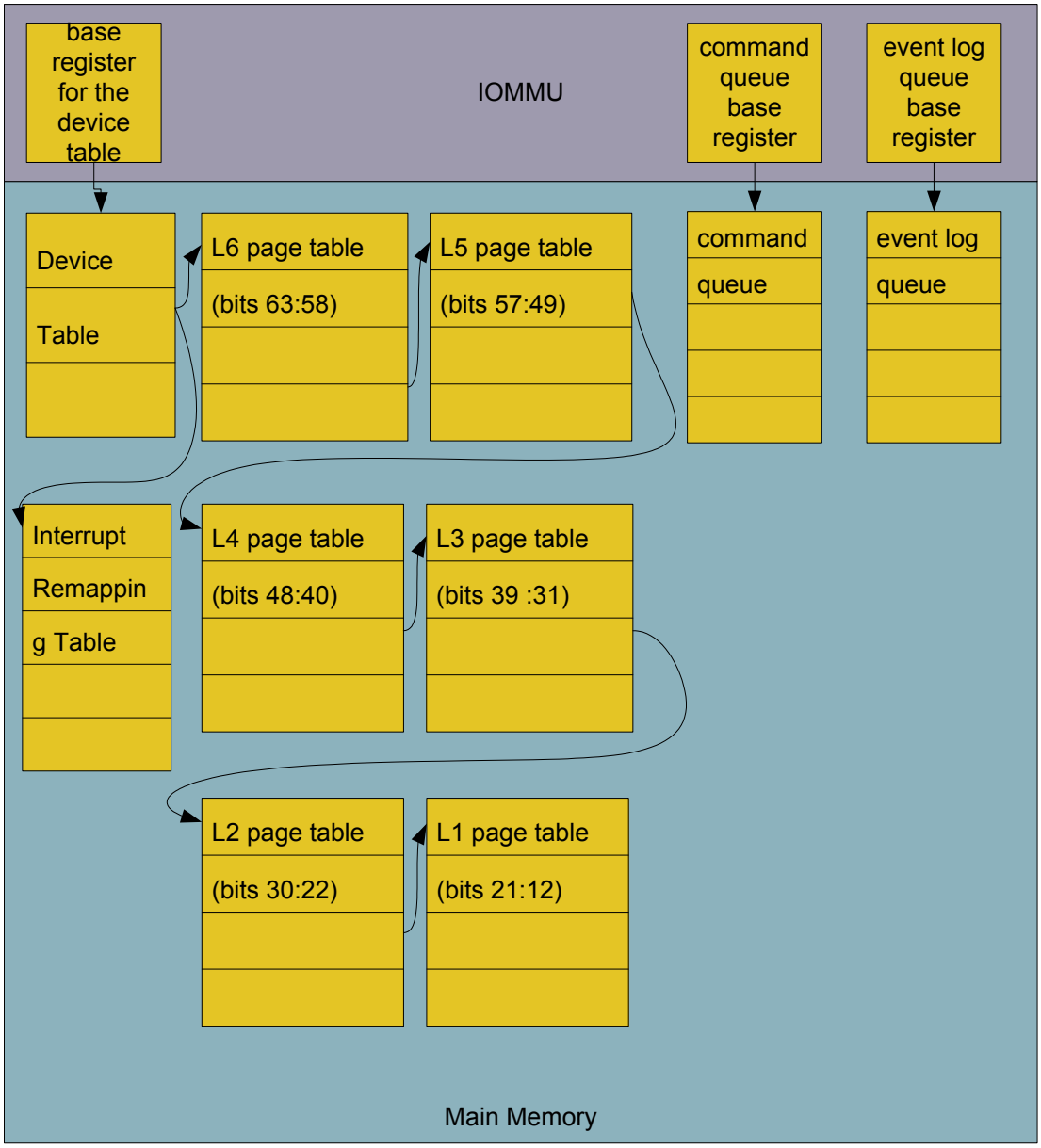


Figure 5-6: IOMMU Data Structures

The performance impact when an IOMMU is introduced into the systems is not only dependent by the IOMMU implementation but also dependent on system software, since the choice of different translation strategies can have a dramatic effect on the number of main memory accesses necessary for a single translation. The IOMMU proposal does not scale

very well to self-virtualizing devices or functions as EXTOLL, since the requesting context is a device ID in the underlying peripheral interconnect. Also, the IOMMU functions as a kind of bridge between the link to the northbridge/CPU complex and the devices south of it. For directly connected devices, which communicate directly with the CPUs host-bridge without intermediate bridges, it is not possible to leverage future system chipset implementations of the specification.

The Intel Virtualization Technology for Directed I/O (VT-d) [104] specifies a similar IOMMU for the Intel architecture. Since Intel systems feature discrete northbridges, VT-d is located in the northbridge of the system. VT-d covers largely the same issues as the AMD IOMMU, however some small differences exist. One example is the protection domain table (known as device table in the AMD specification) which feature two levels of lookup, one based on the bus number, and the second level based on the device and the function number.

5.1.11 PCI Express and HT3 Address Translation Services

To enable faster address translation for DMA accesses initiated by a device, a device may employ an IOTLB, as it is for example described in the AMD IOMMU specification. The PCI SIG (Special Interest Group) has recognized the necessity to support such an architecture and has recently specified the PCIe ATS [105]. In the PCIe jargon, the IOTLB is dubbed Address Translation Cache (ATC), an up-stream IOMMU is called address translation agent (TA). The PCI SIG states, that typical purposes of DMA address translation are *sandboxing* of DMA I/O functions, providing scatter/gather support, redirecting interrupts, converting from 32-bit to 64-bit address space and supporting OS virtualization. Figure 5-5 shows the system architecture with an IOMMU which also is an example for a system where ATS must be employed between the IOTLB of device 3 and the IOMMU in southbridge 1.

The PCIe ATS protocol involves the following changes to the original PCIe specification:

- Memory requests are extended by a 2-bit AT field (previously reserved). These two bits encode, if the device wishes the address associated with the request to be translated or, if the request contains an already translated address (via an ATC/IOTLB). If the request is a translation request a third coding must be used. If the request is initiated by a device that is not allowed to access main memory with physical addresses, this is an error.
- A new PCIe request called *translation request* which is initiated by an ATC and destined to a TA. A translation requests has no data attached, and is used by a device with ATC to request the translation of an address from the TA. An PCIe address *translation request* for a 64-bit address has a size of 4 quadwords (i.e. 32 bytes). By specifying a larger length value, the ATC can request multiple translations for a contiguous amount of virtual addresses.
- A new PCIe completion called *translation completion*. This is the completion packet sent by a TA to the ATC in response to a translation request. A t*ranslation completion* has a header of 32-bytes (64-bit addressing) and, if the translation was successful, the translated addresses are returned in the data payload of the completion packages using 8 bytes each. A tricky encoding allows the TA to specify the length of this translation to be between

4 kB and 4 GB. If one special bit is set in the translation, the translation applies to blocks larger than 4 kB and the receiver has to check the address bits starting from bit 12 (the lowest address within a translation). If bit 12 is set, the translation applies to a block size larger than 8192 bytes, if bit 13 is set, the block is larger than 16 kB. Thus, the least significant zero within the address gives the size of the block (note that for larger blocks the lower address bits are not necessary, since blocks must be naturally aligned). While this can be rather efficient in terms of number of translations, it is more difficult to handle in hardware, both in the control logic and TLBs, than a fixed format.

- A new PCIe request called *invalidate request*, which flows from TA to ATC. An *invalidate request* causes an ATC to invalidate a number of translations which have been changed in upstream entities. The format includes a 32-byte header plus 8-byte payload containing the untranslated address of a block that is to be invalidated. Encoding of block size is the same as with translation completions. Invalidate Requests are broadcast in the fabric, since an TA does not know, which device actually uses ATS, respectively if a device has a cached copy of the translation.
- A new PCIe completion called *invalidate completion*. An ATC responds to an *invalidate request* with an *invalid completion* to indicate that it is safe to proceed with an updated translation. Before it can issue the completion, the ATC must be sure that the translation is removed from the cache and that no transactions currently in execution by the device are using this translation. If a device does not understand an invalidate request, it issues an error response which can be handled accordingly by the TA.
- A PCI configuration space capability to enumerate and configure the ATS capabilities of devices.

Since revision 3.0 HyperTransport [13] defines packets to support the PCIe ATS protocol. This becomes necessary since ATS is designed to operate between an IOMMU and IOTLBs on devices, and the IOMMU in AMD64 systems may well reside upstream on a HyperTransport chain. Since the EXTOLL design is closely coupled with HyperTransport, a description of the packet format of the HyperTransport ATS protocol follows. The PCIe packet formats are very similar; indeed, the HT ATS protocol is defined in Appendix B.7 of the HT specification which is located in Appendix group B called *Ordering Rules and Mapping of Other I/O Protocols*.

In table 5-2, the packet format for the HT3 ATS Request is shown. It has a length of 96 bit. The translation response (table 5-3) features a 32-bit control packet followed by 8-byte payload for every page returned as translation. Up to 8 translations may thus be completed with a single packet, which is a difference from PCIe that supports more translation in the response packet.

A translation invalidation request consists of a 96-bit control packet followed by a 64-bit payload and is transported using the posted channel. The format of the packet is shown in table 5-4. Note bit 2 of bit-time 7 which is shaded in the table. This bit, called *invalidate response* is the only bit to distinguish ATS invalidation *request* packets from invalidation *response* packets.

Bit-Time	CTL	7	6	5	4	3	2	1	0
0	1	2'b10		Cmd[5:0]=6'b111110					
1	1	Device [4:0]					Function[2:0]		
2	1	Bus[7:0]							
3	1	Reserved						Address Type[1:0]=2'b01	
4	1	SeqID[3:2]		Cmd[5:0]=01x1x0					
5	1	PassPW	SeqId[1:0]		UnitID[4:0]				
6	1	Count[1:0]		Cmp=0	SrcTag[4:0]				
7	1	Ignored						Count[3:2]	
8	1	Virtual Address [15:12]				Ignored			
9	1	Virtual Address [23:16]							
10	1	Virtual Address [31:24]							
11	1	Virtual Address [39:32]							

Tabelle 5-2: HT3 ATS Translation Request

The translation invalidation response (table 5-5) also travels in the posted channel (despite of its name) and uses a 96-bit control format followed by a 32-bit one-hot coded response payload. Here, bit 2 of bit-time 7 (again shaded) holds a 1 to state that this is an *Invalidation Response* packet. Theoretically up to 32 invalidation can be completed with one invalidation response packet. There is one response required for each traffic class in PCIe respectively for the normal and isochronous channels of a HT link; the completion count field is used to indicate the number of responses sent for the invalidation.

ATS complements techniques like the AMD IOMMU or Intel's Directed I/O and enables complex systems with distributed IOTLBs. Still, the problems for close-coupled, self-virtualized devices are not solved.

5.1.12 Virtual Memory Hooks in the Linux Kernel

The different NICs mentioned in the previous sections use memory registration with pinned pages, a notable exception being Quadrics Elan network adapters. Locking (or pinning) ensures that the memory-mapping is valid during the time the device accesses main memory. If the mapping ever becomes invalid while the device still accesses the memory undefined behavior can result, most likely data corruption. One point that is often raised against

Bit-Time	CTL	7	6	5	4	3	2	1	0
0	1	Isoc	Rsv	Cmd[5:0]=6'b110000					
1	1	PassPW	Bridge	Rsv	UnitID[4:0]				
2	1	Count[1:0]		Error 0	srcTag[4:0]				
3	1	Rsv/RqUID		Error 1	Rsv/RspVCSet			Count[3:2]	
4	0	Reserved					U	W	R
5	0	Physical Address 1 [23:16]				Size	Coh	Reserved	
...									
11	0	Physical Address 1 [63:56]							
12-...	0	up to 7 more 8 byte translations							

Note: U=untranslated flag, W=write permissions, R=read permission, coh=coherent access required to this region, Isoc=isochronous, Rsv=reserved for future use

Tabelle 5-3: HT3 ATS Translation Response

pinned-down registration based memory management is the stress the method puts on the virtual memory subsystem and the over-subscription of memory resources by applications using the device since such memory resources can not be swapped out. Another point against memory pinning is the cost of memory registration and pinning which can be quite expensive (see section 5.1.3) thus causing applications to map as much space as possible and never returning it to the system, aggravating the first problem.

But sometimes it is argued, that a system that is designed for high-performance must accommodate enough physical memory to handle this kind of load. If the target memory of a user-level initiated RDMA request is swapped out, the low latency of the communication is by far outweigh by the cost of swapping.

There are quite a few further problems, some of which are discussed in an LWN.net article of 2005 [106]:

- *get_user_pages()* is not designed for this use-case.
- Forking can pose a problem since normally copy-on-write is employed.
- Freeing of all resources when a process exits must be assured
- The special case of overlapping registrations must be handled.

An alternative approach to pinned memory registration is to add virtual memory system hooks to the operating system. Whenever a page gets mapped or unmapped, functions that are registered with the respective hook get called and can perform appropriate actions. This

Bit-Time	CTL	7	6	5	4	3	2	1	0
0	1	2’b10		Cmd[5:0]=6’b111110					
1	1	Device [4:0]					Function[2:0]		
2	1	Bus[7:0]							
3	1	Reserved						Address Type[1:0]=2’b00	
4	1	SeqID[3:2]		Cmd[5:0]=6’b1011x0					
5	1	PassPW	SeqId[1:0]		UnitID[4:0]				
6	1	Count[1:0]		Cmp=0	Error	1’b0	Reserved		
7	1	Reserved					1’b0	Count[3:2]	
8	1	Target Device [4:0]					Target Function [2:0]		
9	1	Target Bus [7:0]							
10	1	Addr[31:24]=8’hFB							
11	1	Addr[39:32]=8’hFD							
12	0	Invalidation Tag [4:0]					Reserved		
13	0	Virtual Addr[15:12]				Size	Reserved		
14	0	Virtual Address [23:16]							
15	0	Virtual Address [31:24]							
16	0	Virtual Address [39:32]							
17	0	Virtual Address [47:40]							
18	0	Virtual Address [53:48]							
19	0	Virtual Address [63:54]							

Tabelle 5-4: HT3 ATS Translation Invalidation Request

is exactly the method Quadrics uses to enable full virtual memory system integration. Unfortunately, patching the operating system kernel, especially in such an intricate and important part as the virtual memory system raises reliability and correctness concerns.

There is quite a history of attempts that have been made to establish such a hook within the mainline Linux kernel. In 2003 Thomas Schlichter [107] proposed a patch called *TLB hooks* which enabled attaching to the TLB flushing within the kernel. In 2005 David Addison from Quadrics proposed in a post to the Linux kernel mailing list a patch dubbed

Bit-Time	CTL	7	6	5	4	3	2	1	0
0	1	2’b10		Cmd[5:0]=6’b111110					
1	1	Device [4:0]					Function[2:0]		
2	1	Bus[7:0]							
3	1	Reserved						Address Type[1:0]=2’b00	
4	1	SeqID[3:2]		Cmd[5:0]=6’b1011x0					
5	1	PassPW	SeqId[1:0]		UnitID[4:0]				
6	1	Count[1:0]		Cmp=0	Error	1’b0	Reserved		
7	1	Reserved		Completion Count [2:0]			1’b1	Count[3:2]	
8	1	Target Device [4:0]					Target Function [2:0]		
9	1	Target Bus [7:0]							
10	1	Addr[31:24]=8’hFB							
11	1	Addr[39:32]=8’hFD							
12	0	Invalidation Tag Bitmap [7:0]							
13	0	Invalidation Tag Bitmap [15:8]							
14	0	Invalidation Tag Bitmap [23:16]							
15	0	Invalidation Tag Bitmap [31:24]							

Tabelle 5-5: HT3 ATS Translation Invalidation Response

ioproc_ops, for *I/O Processor Operations*, the name owing to the fact that devices using virtual memory resemble processors in their own right. The patch basically proposes something very similar to the patch from Schlichter but adding hooks to all important functions in the virtual memory area. The patch was not taken over into the mainline kernel but remains available from the Quadrics Website [95]. One of the key disadvantages of out-of-kernel patches can be seen here, since only a small number of kernels is supported, and usually these are relatively old. Of course it is often possible to apply the patch to a newer kernel, but it is untested and may or may not work without problems.

Then, SGI’s Jack Steiner tried to get a driver into the mainline kernel for their GRU hardware. Apparently GRU is an offload engine within the system chipset for *memcopy* like operations. The engine is fed with commands directly from user-space and, must access virtual addresses to copy data on behalf of the user. This GRU hardware also contains a TLB. GRU seem to be the next generation of DMA hardware used in large NUMA systems by

SGI for shared memory communication. Lameter [108] states that the next generation, Xeon based NUMA machines from SGI actually have to run several instances of Linux running on the same shared memory machine, since the amount of memory in the machine exceeds the addressing limit of the processor (16 TB). Thus, the GRU may be of use to support fast message passing between different Linux instances. The GRU driver uses another virtual memory system hook patch, the *mmu notifier* patch from Andrea Arcangeli which has been accepted into the mainline kernel tree starting with Linux version 2.6.27 and basically provides the same features as the earlier *ioproc_ops* patch. Another very important driving force behind the acceptance of the *mmu notifier* patch are OS virtualization techniques. Today, Hypervisors have to manage shadow page tables, table structures mimicking a real table structure for guest operating systems, in addition to the real table structures (see also section 5.1.1). To keep everything consistent the current KVM implementation has to pin all pages that are currently used by a guest system, and therefore puts a high pressure on the virtual memory system. The *mmu notifier* patches are expected to help tremendously in improving KVM memory behavior.

The rational from all of this is that new RDMA-enabled hardware or other hardware that wishes to directly operate on user virtual memory should be designed to work with both registration based and hook based memory management.

5.2 Design Space of the EXTOLL Address Translation

For the EXTOLL RMA units address translation is necessary. The unit that will perform this function is called Address Translation Unit (ATU). After the analysis of the previous paragraphs about different current, state-of-the-art memory management methods, this section presents a design space analysis for the EXTOLL ATU. The requirements have already been stated in the introduction to this chapter.

All of the different aspects of the design space have been summarized in figure 5-7. The blue shaded entries are analyzed in more depth in the following sections. One decision is if hardware- or software-based translation should be used. Software based mechanisms include the interrupt-driven approach, i.e. always triggering a CPU interrupt and continuing when the result has been returned by the CPU, and the pre-translated approach. Pre-translation means that actually only PAs are given to the device, so translation in the device in unnecessary. Pre-translation also implies kernel-based virtualization (see section 6.2). For hardware based translation there is the choice of direct mapping or inverted mapping. Inverted mapping is discarded since it does not map well to the problem at hand, since the methods available either perform badly or are not possible to implement. In the direct mapping domain, it can be differentiated between designs implementing a full system table-walk or a reduced table-walk. TLBs can help speeding up the process of translation, for the design space TLB-less designs, software managed and hardware managed designs are analyzed. With a software managed TLB all operations are completely controlled from CPU software, hardware only performs lookups and interrupts the CPU in case of misses. A

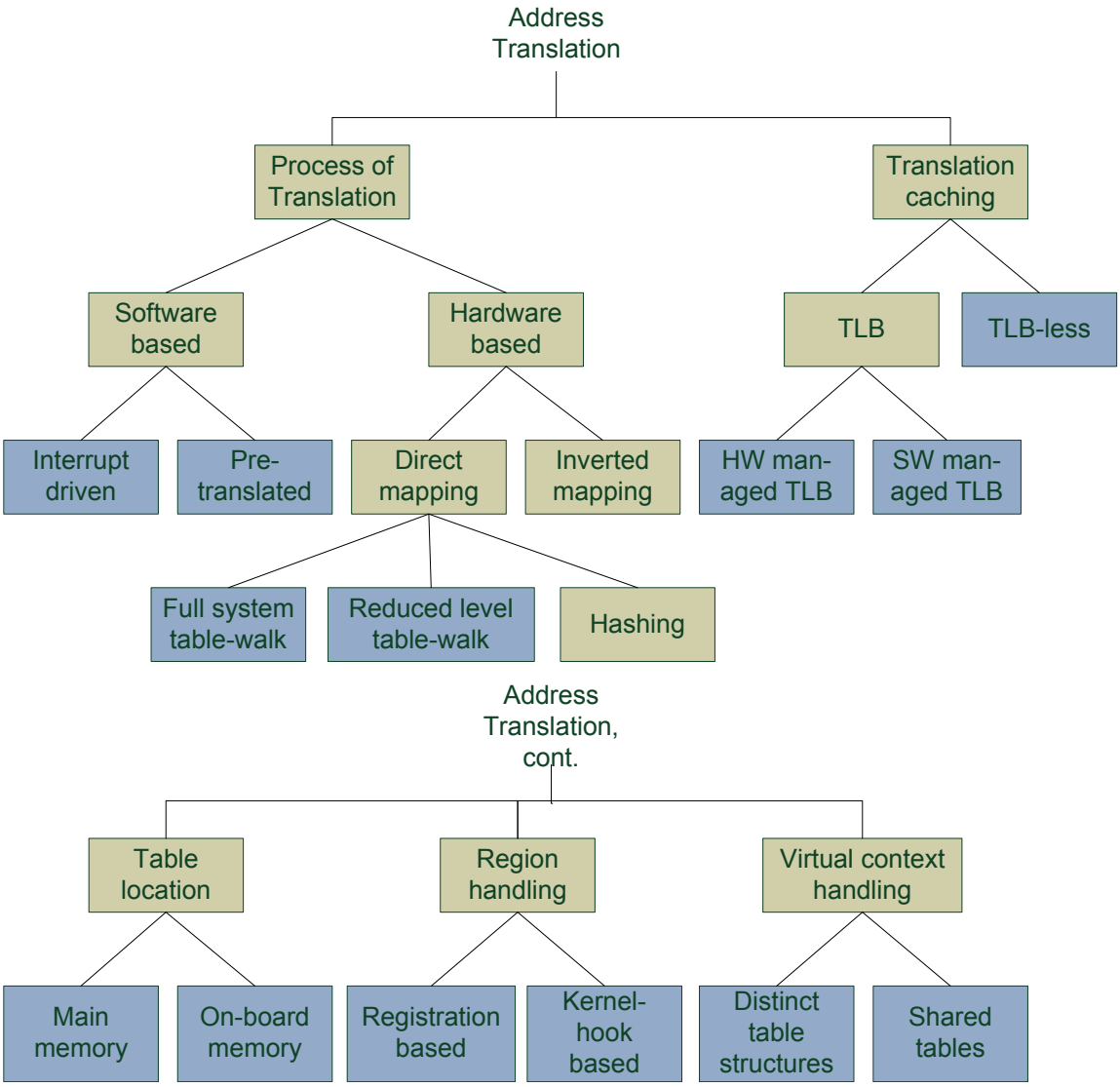


Figure 5-7: EXTOLL Address Translation Design Space Diagram

hardware managed TLB functions largely autonomously, much like the TLB in modern CPUs. Table-walk-less designs with a TLB need to interrupt the CPU on every miss. Table-walk-less designs without a TLB are the same as the above mentioned software based, interrupt-driven design. Full-depth designs require up to 5 or 6 levels of table walks, like the AMD IOMMU. Reduced depth table-walk engines are restricted to a smaller amount of table walk depth, an example being the IBM Calgary engine. Reduced depth lowers the needed number of memory accesses to perform the table walk and thus the time it takes to complete one translation.

Additional choices are shown in the lower part of the diagram. Translation tables can be located in main-memory, on-board memory (i.e. dedicated memory devices directly connected to the EXTOLL device) or on-chip. Memory can either be registered or managed using kernel hooks. Different end-points or process contexts can either be handled using

distinct tables or using a shared table method. A choice largely unrelated to the other decisions concerns the address translation service protocol used on chip (not shown in the design diagram).

The following discussion focuses on main memory based designs for the following reasons: On-chip tables are prohibitive because of the sheer size of the tables. Off-chip, dedicated memory is a choice - it can always be substituted for main memory if available. It does increase costs, though, also the I/O-pin count of the device goes up. The following paragraphs describe the resulting design choices resulting from reasonable combinations of the blue marked leafs in the design space diagram. All of the performance estimations below are based on the model presented in chapter 3 (see table 3-2). The numbers for the HT400 link are used to provide a baseline, lower-bound of the performance numbers. Of course, faster HT links increase performance, while a PCIe connection will slow the performance down.

To estimate the expected performance of a given design the following metrics (where applicable) are used:

- $t_{lat:}$ the latency for a single translation
- $t_{miss:}$ the latency incurred if a translation misses the TLB
- t_{reg} and $t_{dereg:}$ the latencies to register respectively deregister a memory region. This latency is especially important if large memory spaces are to be mapped or the memory used by the device is highly dynamic as in PGAS applications.

5.2.1 Interrupt Driven Software-Only Approach

This approach, one of the most basic ones, interrupts the CPU for each request, both at the initiator and the completer side. The interrupt causes the CPU to search for the appropriate translation and passes it back to the NIC which can then actually start the memory transfer.

Since this approach does not employ a TLB, the latency is always the same and can be approximated by

$$t_{lat} = t_i + t_{trans} + t_{cd}$$

resulting in a latency of > 1.5 µs on each side. The method is exhibits thus a prohibitive aggregated latency of more than 3.0 µs per complete communication operation and is thus discarded.

5.2.2 Software Pre-translation

In this design the translation for both, source and destination address, is performed at the sender in kernel space prior to initiating the operation. The actual initiation and passing of the PAs to the device must also occur in kernel space to enforce basic security rules.

The latency for such an operation is approximated by

$$t_{lat} = t_s + 2 \cdot t_{trans}$$

This would amount to ~ 1 µs. Unfortunately, the translation for the destination (i.e. remote) address must be known to the kernel; this introduces significant complexity and overhead into the overall scheme. If only registered memory is supported, t_{trans} must be substituted by the time it takes to lookup the memory region in a flat table which is in the order of t_{cm}. The complete translation latency becomes then

$$t_{lat} = t_s + 2 \cdot t_{cm}$$

which evaluates to 310 ns. Unfortunately, this does not include all overheads. Nevertheless, since this scheme is also one possibility for the device virtualization, it was implemented as one of two address translation solutions for the EXTOLL RMA unit. The results, also in comparison to the hardware ATU, are discussed in section 7.5.5.

The time to (de-) register memory is

$$t_{reg} = t_s + t_{trans} + t_{cm}$$

and

$$t_{unreg} = t_s + t_{unpin} + t_{cm}.$$

This evaluates to 470 ns respectively 370 ns.

5.2.3 Managed TLB

This design involves an on chip TLB which is used to translate addresses. Whenever there is a TLB miss, an interrupt request is sent to the CPU. The interrupt handler on the CPU retrieves the address, translates the address into a PA, chooses a TLB entry and writes the new entry into the TLB. Now, the functional unit can continue with the PA and perform its memory access. This design enables a very flexible architecture which is mostly defined by software, since the TLB is completely managed by software, i.e. which entry actually gets replaced is completely left to the software. This also means the method involves a relatively simple hardware, depending on the TLB architecture of course. It turns out that the design has several flaws, though, which makes it ultimately not the best solution for the above stated requirements: while latency is very good when the TLB is hit (~60 ns), it is catastrophic in case of a TLB miss. Consider the time it takes to service a TLB miss:

$$t_{miss} = t_i + t_{trans} + T_{manage} + t_{cd}$$

where t_i is approximately 1 µs for modern systems in the best case. The translation time depends on the tables used by the software and involves from one to 6 memory accesses corresponding with the depth of the translation tables used. So, t_{trans} can be estimated to be between a few nanoseconds (one level of translation, cache hit) up to more than 500 ns (4 levels of translation, cache misses). Actually, the translation time may be higher than given in table 3-2, since the translation is triggered from an interrupt context and the right process context must first be searched and loaded. The management time (i.e. choose an entry to be replaced with the new entry) is largely dependent on the associativity of the TLB, which defines the amount of different choices, and uses aging methods possibly involving addi-

tional device access. So an estimation runs from a few nanoseconds to > 200 ns if additional device access is necessary. To insert the entry into the TLB it is mandatory to at least perform a single write transaction to the device which takes in the order of 100-200 ns.

In the best case, the interrupt latency, a number of accesses to cached data-structures and the writing of the resulting entry back into the device must be considered, leaving t_{miss} to be about 1.6 µs. The average case, including interrupt context related overheads is more likely to be around 2 µs.

Again the design can be somewhat optimized by using registered memory. The interrupt handler thus has only to lookup the PA in a flat table of registered memory which also removes the necessity to find out the right page table base address. t_{miss} can be expressed as

$$t_{miss} = t_i + t_{cm} + T_{manage} + t_{cd}$$

and evaluates to 1.3 µs in the best-case and to 1.6 µs average-case latency. The time to (de-)register pages is the same as in the previous section.

5.2.4 Autonomous TLB

The only difference to section 5.2.4 is to let the TLB mange itself. Advantages are that the TLB can keep better information about usage patterns to manage entries more efficiently, also the $t_{miss} = t_i + t_{trans} + T_{manage} + t_{cd}$ is slightly reduced since the CPU has not to search for an appropriate entry and the TLB can search for the entry while the CPU is performing the actual translation. This design is expected to perform slightly better than the one described above with a more hardware-intensive TLB implementation. In [109] a design and an implementation of such a TLB are presented. The design uses several stages of a CAM to implement a fully associative TLB. The complete design features eleven pipeline stages. A mapping to a Virtex4 FPGA with a number of limitations was performed, yielding 100 MHz clock frequency with 16 entries using 48 % of the slices of a Virtex4 FX60. The design was later improved mainly in terms of resource utilization [110]. The usage of CAM modules on FPGA architectures like the Virtex4 shows prohibitive resource consumption due to the necessity to emulate associative matching with standard dual-port SRAM blocks. For many applications the addition of CAM blocks to FPGA architectures would be a very promising features which the (somewhat specialized) CSwitch architecture already features [111].

5.2.5 Full Hardware Table-Walk

Full tablewalk in this context means, that the ATU performs a full X86 table walk, either directly on the processor page tables (shared page tables) or using dedicated page tables that show the same or a similar structure as the processor page tables. So, this approach exhibits similarities with the AMD IOMMU specification from section 5.1.10. One principal difference of course is the support of the virtualized EXTOLL units with its VPIDs (Virtual Process IDs), which identify one end-point, instead of the use of RIDs (bus:device:function requester ids) as protection domain identifiers.

The TLB miss latency can be calculated by the following formula:

$$t_{miss} = \left(\sum_{0}^{m} t_{dm} \right) + x \cdot t_{cyc}$$

where *m* is the number of page table levels. The number of levels depends somewhat on the exact scheme chosen. It is reasonable though to assume a maximum of five levels, four of them for the normal page tables and the top level to manage different VPIDs. Each level costs one read transaction to main memory accounting for 280 ns. Additionally several clock cycles are needed to check the result and insert the result it into the TLB. So, in summary this approach gives an approximate maximum T_{miss} of ~1.4 µs. This is better then the previous two possibilities analyzed. Another advantage is that the translation is off-loaded from the CPU. On the downside it can be said, that the latency is still relatively high and the method only plays its strengths if it is used together with a *virtual memory hook* system to enable consistent, shared used of page tables.

5.2.6 Reduced-Depth Hardware-Table Walk

This design chooses to use a reduced-depth set of page tables for the device. These tables must be set up, managed and kept coherent with host tables by kernel software. It offers the possibility to reduce the miss latency considerably and free the CPU from translations. The use of TLBs is possible. With a single level translation the miss latency calculates to ~280 ns. When applied twice for an RDMA transactions it accounts for ~0.5 µs. Also, hardware complexity is simpler and it is not necessary to use the host table format which increases portability.

5.2.7 Registration Based versus Kernel-Hook Based Designs

As discussed in section 5.1.12, the design chosen should be able to support both methods efficiently. The most promising methods so far, pre-translation and reduced-depth table walk fulfill this requirement. Kernel-hook-based software designs offer the possibility to reduce the load on the virtual memory system of the OS, since fewer pages have to be locked at one time.

5.2.8 VPID Handling

Two possible approaches were identified. The first employs a different set of page tables for each protection domain, which is a Virtual Process Identifier (VPID) for EXTOLL. This is the traditional approach which is used by CPUs and the AMD IOMMU specification. It must be noted though, that some parts of a page-table-set may be shared with the page-table-set of a different protection domain. The whole approach has the disadvantage for devices like EXTOLL that it makes one additional level of indirection necessary just to

fetch the context. This is the device table in the AMD IOMMU specification. One way to address this problem would be to hold the context completely on chip or employ a large enough context cache.

Method	Advantages	Disadvantages
VPID private tables	• Smaller amounts of physically contiguous memory necessary	• Base address table necessary: either one indirection via main-memory or on-device table for each context • Entry format bits are not optimally used
Single shared context table	• Just one base address in device register • Uses bits in entry format efficiently	• One large table for all • Large amounts of physically contiguous memory necessary
Multiple shared context tables	• No restrictions on physical placement of table pages • Incremental growth of tables is easily implemented • Uses bits in entry format efficiently	• At least 2 levels of tables necessary. Top level either via main memory or on-device table

Tabelle 5-6: Strategies of Context Handling in Page Tables

This approach which is called *VPID private tables* in table 5-6, does not use the bits of an individual entry of the lowest level of the translation table optimally. The entry must carry the physical address (which should be 40 bits equaling support for 52 bits of physical address space) plus access and present bits. So, with a 64-bit-entry there is quite a number of reserved bits. A physical address space of 52 bits seems reasonable because the next generation of processors will first introduce this amount of physical address space. Notice that this is the addressable memory. The *mappable* memory may be different and is based on the number of entries in these tables.

A novel approach uses one set of page tables for all contexts, called *shared context page tables*. Here, each entry holds the context ID it belongs to next to the actual translated address and the typical status bits. When the MMU fetches the entry, its ID is compared against the context ID which is associated with the translation request. Only if this comparison succeeds the translated address is returned, otherwise a fault is returned. This com-

pletely eliminates the context indirection level and it leads to the new situation, that all contexts see one virtual address space, called *Network Logical Addresses* (NLA), but only some of these addresses are reachable for the context. This approach can easily support the mapping of all of the physical memory present in a machine. The amount of memory spent to manage the tables is in the same order as the traditional approach. In fact, it is even somewhat less. The individual entry uses the available space better. Taking 40 bits for the physical page number, 3 bits for access and management and 16 bits to store the owning VPID, still 5 bits in a 64-bit entry can remain reserved for future extensions. There are two variants, one is using a single table and the other is using a scatter-gather like implementation. The scatter-implementation uses a number of chip registers to address more than one physical region for the translation table. In essence this creates a two-level translation with a narrow first level which fits completely on-chip. Table 5-6 summarizes the advantages and disadvantages of the two designs called *single shared context table* and *multiple shared context tables*. Note that the system is secure, because user space software cannot access memory that is not registered for its VPID.

5.2.9 On-Device ATS

The EXTOLL devices use an on-chip communication network. Currently this is the HTAX [15]. The on-chip protocol is closely related to HyperTransport. It is necessary to transport address translation service messages back and forth between the translation agent and the individual FUs using this service.

The PCIe ATS protocol was shortly introduced in section 5.1.11. HyperTransport 3.0 features a very similar protocol (see Appendix B.7, Address Translation Packets [13]). This protocol is designed to support multiple translations in one request and also to implement different sizes of translation. This is especially useful if the latency for ATS is high, and to be as general as possible. Both of the reasons are not true for the EXTOLL system. The packet format in standardized ATS protocols uses at least 96 bits for a request. The actual information carried, though, can be encoded in 64-bit.

The HTAX interconnect features multiple Virtual Channels to support different packet traffic classes. Three virtual channels are used by the standard traffic types of posted requests, non-posted requests and responses. For the HTAX ATS, a fourth VC can be used with virtually no additional cost, which is in turn dubbed ATS VC. Over this VC, ATU ATS packets are routed from client FUs to the ATU and back. Routing is done purely on port numbers. There is no possibility to pack more than one request/translation in one packet, but since each packet is 64 bit in size, it only takes one cycle plus the arbitration latency to transfer it over the HTAX. The HTAX supports an maximum of one 64-bit-transfer every other cycle, so the ATS protocol sets a theoretical limit of $f_{clk}/2$ requests per second. A 156 MHz HTAX can thus sustain up to ~78 million translations per second.

5.2.10 Conclusion

From the design space analysis it becomes clear, that a full table walk engine must be rejected since it accounts for too much latency. Designs that employ CPU managed TLBs and CPU based table-walks are also limited by high miss latencies and can only perform competitively through the use of very large on-chip TLB structures.

A case in point is to study the overhead of CPU translation and kernel space transitions which offers an interesting option to implement a physical address space only device, and tunnel all requests through kernel space. One could say this is a *minimal kernel involvement* device. Short message service actually does not need address translation, it is always faster to perform a copy instead of performing a translation. This design choice has been incorporated in the VELO unit (see section 7.4).

To implement translation for RMA in EXTOLL a novel shared context page table design is used. This new design minimizes t_{miss} through minimization of table depth. At the same time it does not impose a high memory overhead on the system. A TLB should be implemented for highest performance; but the design should also perform well in case of a small or no TLB. This is especially important for EXTOLL implementations mapped on FPGA technology. The standard ATS protocol was also analyzed and it was shown that a much more streamlined protocol should be used for EXTOLL. A further advantage of the design are low t_{reg} and t_{dereg} latencies.

The following sections introduce the detailed architecture, implementation, verification and results of the ATU.

5.3 The EXTOLL Address Translation Unit

The requirements for the EXTOLL ATU have been formulated at the beginning of this chapter and the results of the design space analysis have been summarized in the previous section. Now, the actually chosen architecture which incorporates several novel features and ideas is presented. The resulting table structure is visualized in figure 5-8.

There are two levels of translation. The first is performed on chip. The high-order bits of the NLA are used to select one of the *Global Address Table* (GAT) base registers. The selected GAT base address is then concatenated with the low-order NLA bits to get the main memory address of the *Global Address Table Entry* (GATe). A 64-bit value is fetched from memory, which forms the translation. The high-order bits of the translation contain the VPID, the lowest-order bits contain management information (read/write access). All of them are checked, and if access to the page is granted for the requesting VPID, the translation is successfully concluded and the PA returned.

The supported physical address space has been chosen to be 52 bits, which is the maximum size for any main stream x86 server processor for the next 3 years to come and will probably be enough for more years to come. Very important is also the *mappable* physical address space since the design should be able to map all of the physically available RAM of the

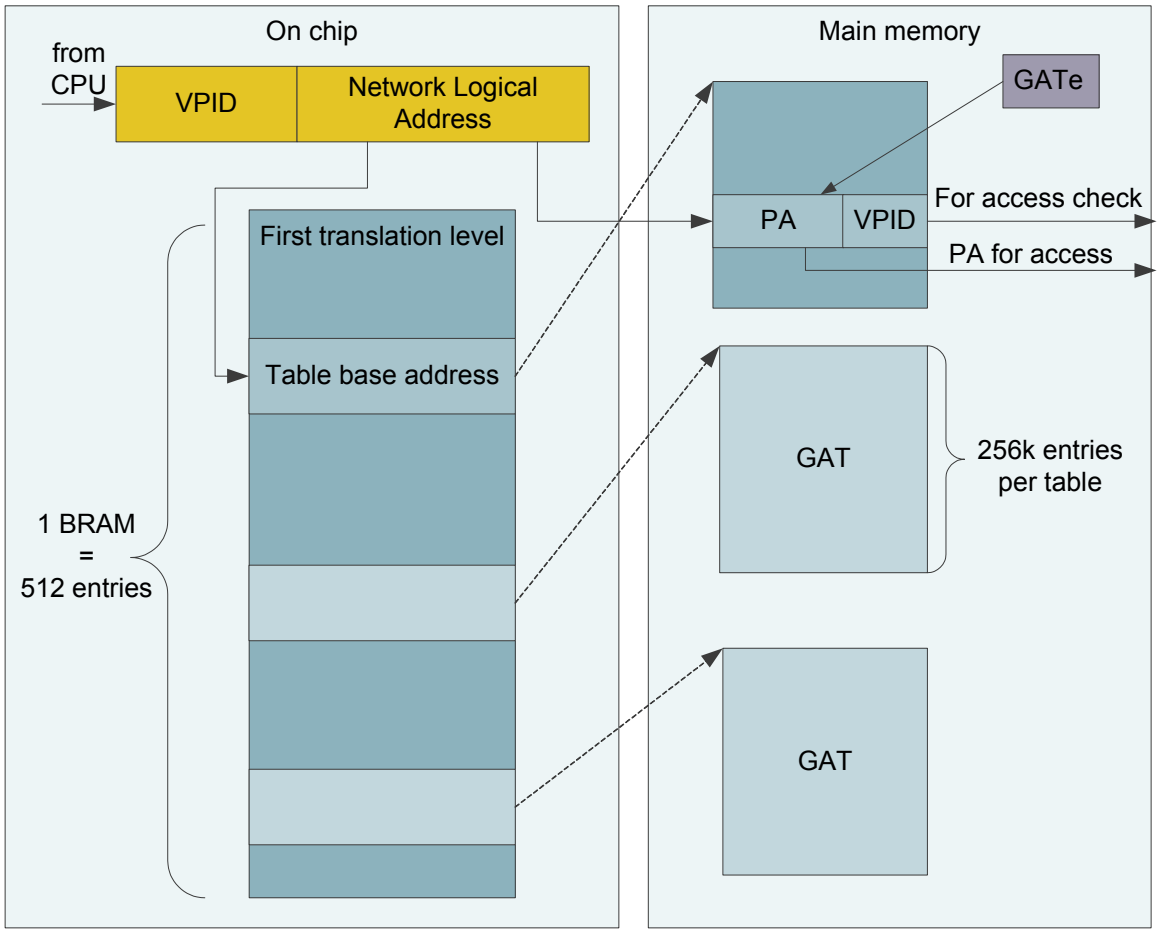


Figure 5-8: ATU Data Structures

machine. The on-chip translation level is implemented in an SRAM block. The width of the RAM block is 32 bit and the depth is 2^9. Thus the on-chip translation for 512 GATs fits into one standard 18-kBit block RAM.

The second level translation tables resides in main memory with 2 MB of size each. This size matches good into existing OS and CPU virtual memory design, since it supports a choice of either implementing a table in a single 2-MB page (*hugetlb* in Linux) or supporting it through a number of physically contiguous 4-kB pages. And for the second case, 2 MB also mark the limit of memory that can be relatively easily allocated contiguously by the Linux kernel. For larger chunks it becomes increasingly impossible to allocate the amount of contiguous memory.

One GATe is 8 bytes in length and handles one page of physical memory. So, to be able to map 2^x bytes of physical memory, 2^{x-12} entries with a total size of 2^{x-9} must be allocated with a minimum mappable area of one page (4096 bytes). To put it differently, a single 2 MB GAT filled with 64-bit entries, hosts translations for 256k pages equaling 1 GB (2^{30} bytes) of mappable address space. The offset within one of the GATs is 18 bits wide. The final page offset is 12 bit. So, to be able to map all of the physical RAM in one machine, one

needs to allocate one GAT for every GB of mappable main memory. With one block RAM (9 bits), the total mappable address space is thus 39 bits, or 512 GB. This makes sure the first requirement is met, to be able to map all of the physical RAM of a machine. For the future, larger RAM sizes are envisioned. The short- to mid-term path is to implement a larger GAT base address table, using up to 10 block RAMs and thus supporting up to 5 TB of mappable main memory. For the long term, the base address table can be mapped to main memory and the on-chip memory becomes a cache of the first level table. The main-memory instantiation uses 2 MB of memory and thus supports 2^{19} entries increasing the mappable address space to 49 bits (0.5 Exabyte).

Practically, this scheme means, that the overhead of being able to map all of the physical memory of a server amounts to 2 MB of tables for each GB of main memory, so a typical high-end 2-socket server with 32 GB of memory uses up to 64 MB of translation tables.

The architecture maps very well to FPGAs as well as ASICS and provides for a low worst case latency overhead of ~ 500 ns for a single RMA operation. To further improve performance, even a simple TLB can help. The architecture accounts for the inclusion of a TLB. The choice of the actual TLB is technology dependent and for the FPGA implementation, a simple direct-mapped TLB was chosen, since large associativity increases resource usage dramatically on today's FPGAs. If special associative matching hardware is available, either by the use of an external IC or by the use of on-chip macro blocks, a more sophisticated TLB can be used. Actually, the architecture does not dictate the position of the translation tables.

The ATU design can be used either with software that supports registering/deregistering of memory or with a design taking advantage of kernel features enabling a tighter integration into the kernel virtual memory system (e.g. the *mmu notifier* patch). In either case, an entry is inserted into a free entry of a GAT, the concatenated GAT base index plus the GAT entry index is returned to the user application and called NLA. It can then be used in user-level commands to a RMA unit.

The complete process of registering and using a memory region is shown in figure 5-9. The individual steps are:

1. The user process calls the API function to register memory.
2. Kernel code allocates an empty entry in GAT in main memory. The entry is written to main memory, this includes the physical address of the user page to be registered and the VPID as well as some other access restriction bits.
3. The index of the new GATe into the table is returned to the user application.
4. The user application can now use this GATe index instead of a physical or virtual address to signal the correct memory location to the device. Note that this implies the usage of some sort of translation data structure and algorithm in the application.
5. The user application sets up a command descriptor involving a reference to user memory. It passes the NLA to the device. The corresponding VPID is passed through the device by means of the physical address space used to issue the command (see section 6.4.3).
6. The request with the untranslated GATe index arrives at a functional unit (FU) of the device.

7. The index along with the VPID the request originates from is sent to the on-chip ATU.
8. ATU first performs a TLB lookup operation. During the TLB lookup, the base address of the GAT to be used in case of a TLB miss can be calculated using the high-order bits of the NLA as index into the GAT base table.
9. The TLB performs a lookup using the NLA. It also tests VPID and access permissions. If the TLB returns a match this entry can be used.
10. In case of a TLB miss, a single 64-bit sized read from a main-memory location yields the correct GATe. The entry is checked if it belongs to the correct VPID and all access restrictions apply.
11. If all checks succeed, the translation can be returned to the requesting FU and the FU can continue with request processing.

The process is very similar if the translation is triggered from a network packet arriving at a FU. In this case, the EXTOLL network packet carries a tuple of *{VPID, NLA}* which is then handled as described above.

In the case a mapping of a page has to be revoked, either because the process is terminated or the respective memory is freed, system software is invoked, again either explicitly because of a deregistration call or implicitly because of a kernel callback function. It removes the translation from the in-memory tables (i.e. overwrites the GATe with zeroes) and then notifies the ATU of the flushing of the NLA (*ATS flush request*). This is performed via control register writes. The ATU then notifies all FUs that the NLA is to be flushed from their registers and caches. At the same time the entry is removed from the ATU TLB. Once the FU(s) have answered with an *ATS response*, the corresponding status register is set. System software can check the register and the memory is free to be used for other purposes. FUs have to answer with a flush response as soon as they can be sure, that the address is not stored in internal state and not currently used for a DMA transfer. For example, the RMA unit checks against the transfers currently under way, and as soon as the NLA in question is not used, the response is sent back. If a new request for this NLA arrives immediately after sending the flush response, the mapping is already gone, and the transfer fails, which is the correct behavior in this case and should be handled by system software correctly, for example as the remote equivalent of a segmentation fault.

The ATS protocol of the EXTOLL ATU extensively uses the features of the EXTOLL on-chip HTAX interconnection network. All ATS transactions travel across a dedicated ATS virtual channel. There are four transactions defined, all of them with a size of 64 bit, summarized in figure 5-10. The first pair of transactions is the *translation request* and its *response* initiated by FUs needing an NLA to be translated. The second set is the *flush request* and *flush response*, initiated by ATU to inform FUs of NLAs being flushed and collect their responses. The fixed size and format of the transactions together with the ATS VC account for a streamlined protocol implementation that also exhibits a very good performance and latency characteristic.

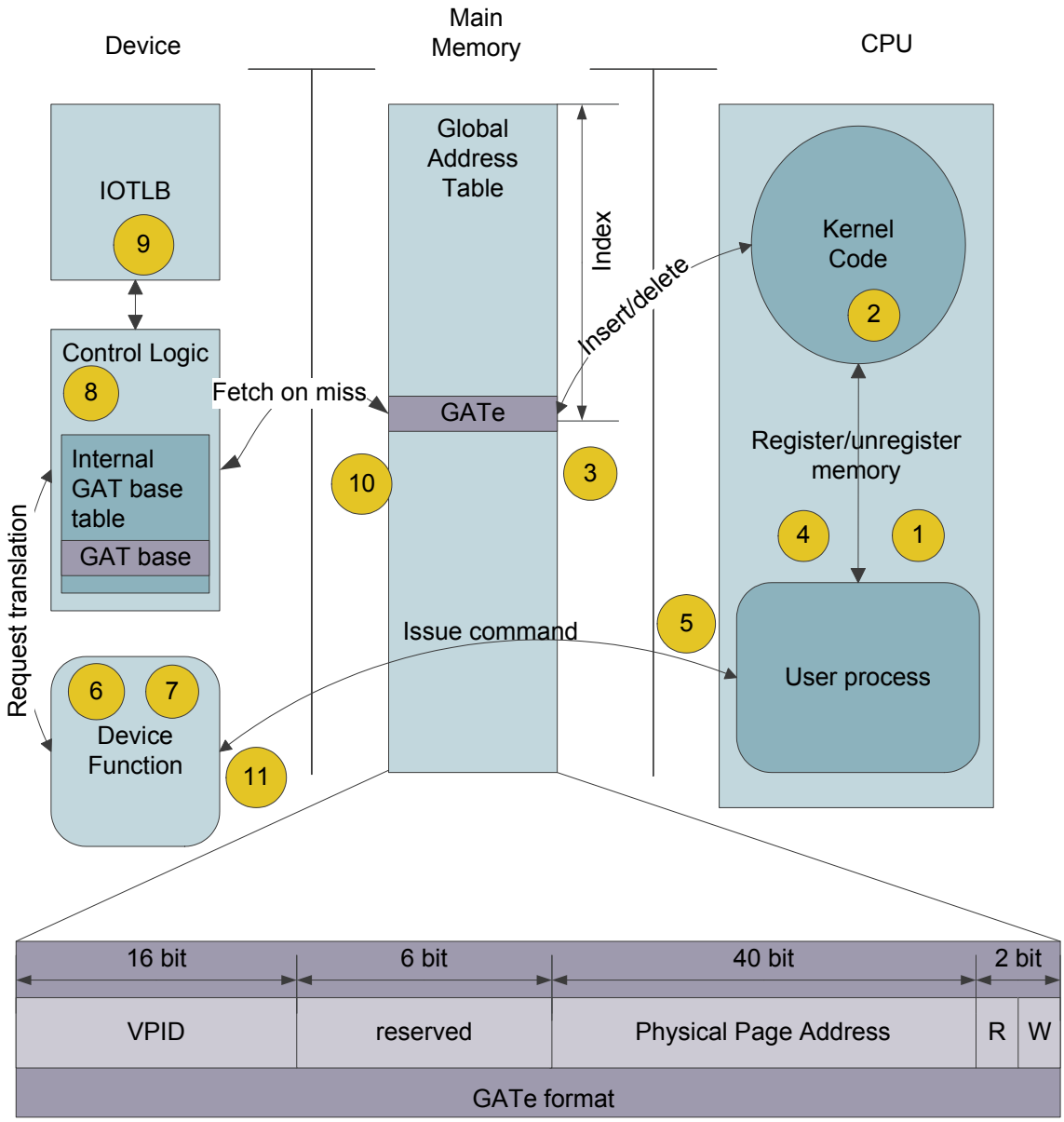
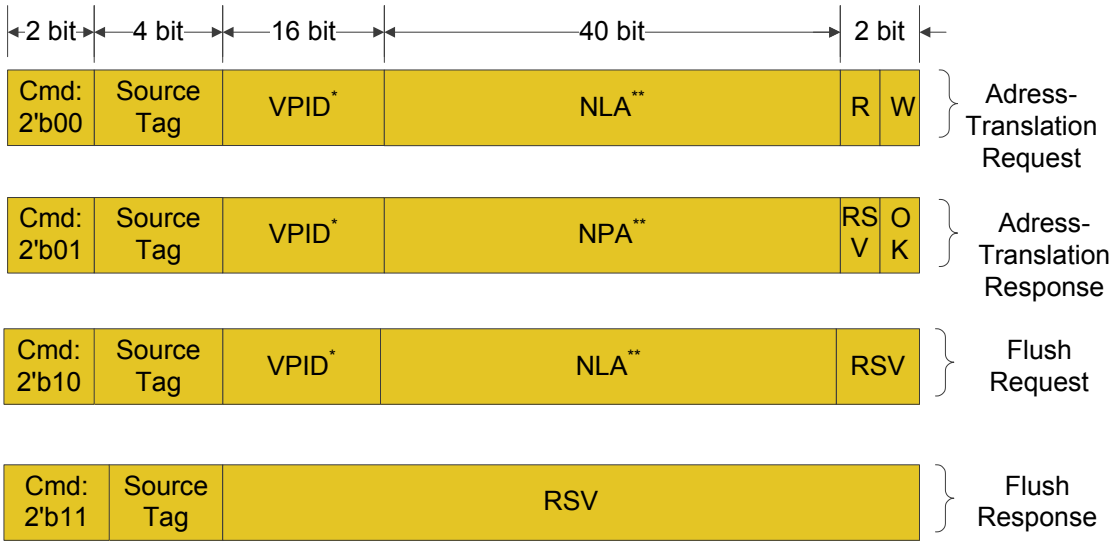


Figure 5-9: ATU Address Translation Process

For system software, the programming interface of the ATU consists of the GAT and GATe data structures, already defined in figure 5-8 and the register level interface summarized in figure 5-11. To enter a new translation, no register has to be manipulated. To enter a new GAT, the physical base address of the table has to be entered into the corresponding GAT base address register involving a single store operation.

Flush operations need to first invalidate the GATe in main memory and then access the ATU Flush Request register. Software can either request to invalidate an TLB entry, perform an *ATS flush request* or both. Subsequently software can poll on ATU count registers or the ATU Flush Request register to monitor the state of pending flush operations. With the cur-



All traffic runs on the dedicated ATUVC of the HTAX. In current Implementations that is VC 3.

Coding	Command
2'b00	Translation Request
2'b01	Translation Response
2'b10	Flush Request
2'b11	Flush Response

Glossary	
Cmd	Command
VPID	Virtual Process ID
NLA	Network logical address
NPA	Network physical address
OK	1 if access is ok, 0 otherwise
R/W	Read/Write access
RSV	Reserved

* Implementation notice: Only the least significant 5 bits are currently implemented yielding up to 32 different VPIDs

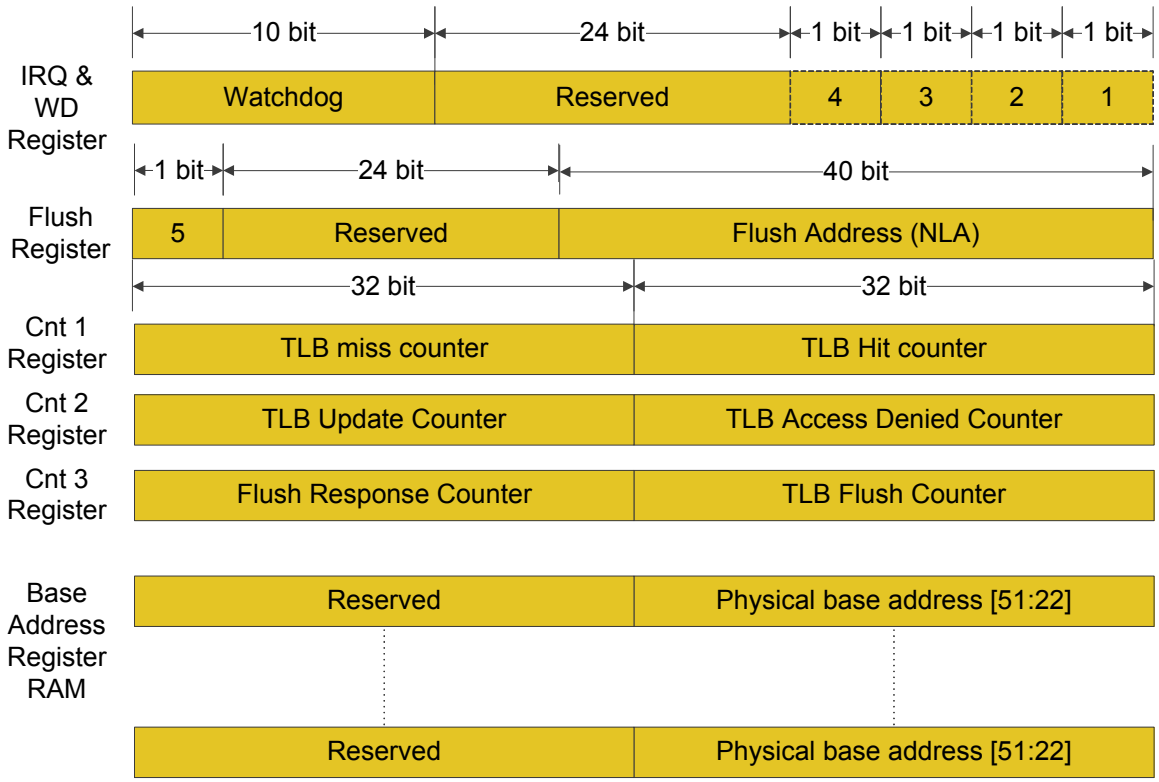
** Implementation notice: Only the least significant 32 bits are currently implemented yielding up to 2^{32} addressable 4kB pages (=16 Tbyte)

Figure 5-10: EXTOLL ATS Protocol Overview

rent RMA application, an ATS request spawns 3 ATS requests which are eventually answered by the RMA sub-modules. The current RMA implementation guarantees, that any ATS flush requests flushes all internal buffers.

The remaining registers are all event counters and are useful to keep track of the performance of the ATU and for debug reasons. The events that can be monitored are TLB hits, TLB misses, TLB flushes, and ATU flushes completed.

One last aspect of the architecture remains to be explained, the aspect of memory management by system and user level software components. System software must insert translations into free GATes upon request of client applications. Usually it is a good idea to set a



1) Enable bit for IRQ messaging of completed Flush requests
2) Flush IRQ bit (clear on read), set after an IRQ was triggered
3) Enable the watchdog timer & IRQ (which is started, when a Flush request initially arrives)
4) Watchdog IRQ bit (clear on read), set after an IRQ was triggered
5) The Valid bit set by software when writing a new flush request to the register. It is cleared by Hardware, when the new request has been accepted.

Figure 5-11: ATU Register Interface

GAT aside for single page requests and try to serve multi-page requests en-block. While user-level software is required to handle non-contiguous NLAs, it may still be more efficient for system software to try to reduce fragmentation to ease search for free entries.

The process of memory registration eventually returns with a vector of NLAs to user software. The user API is recommended to treat such a list as the representation of a *memory region*. An opaque data structure or handle is used to represent the registered memory region in the program. So, to send from a registered memory region, the handle to the region is passed on to the *send* API function, which can then, based on the offset within the region, select the right NLA(s) to issue the actual command to the device.

The notion of handles for registered memory regions actually blends in with most existing APIs and software architectures in the realm of communication and parallel computation. For example, one-sided MPI primitives feature a window argument which can be used to reference the memory region. In GasNET, core operations always reference the remote access memory segment; extended API operations are more difficult to support [80]. Relative straight-forward solutions to this problem involve either to register all memory actually mapped by a process (limiting memory resources of parallel applications to the physical memory size) or using an algorithm similar to the Firehose algorithm [82]. One particular exception, which unfortunately may be an important one, can be found in two-sided MPI communications. Here, the mapping from virtual address to NLA must be performed by user-space software. For 2-sided MPI communications, an often quoted method is to use a 2-copy mode for small to medium messages and switch to a RMA and rendezvous based zero-copy protocol only for large messages. The aggregated overhead for large bulk transfers allows to register/deregister memory on the fly in this case. Details of the integration of the ATU into the EXTOLL software stack are presented in section 8.1.

This concludes the description of the ATU architecture. The next sections provide a description of the microarchitecture for the ATU implementation and some remarks about ATU verification and performance.

5.4 ATU Microarchitecture

The ATU has been implemented for the EXTOLL prototype. The complete code of the unit covers ~1800 lines of Verilog HDL code including the code for four FSMs that were generated with the FSMDesigner tool [112] but excluding the register interface. This demonstrates the lean and efficient design.

Figure 5-12 shows the block level architecture of the ATU. The location of FSMs is marked with circles inside its respective top-level block. The *HTAX Inport* and *Outport* handle the communication with the on-chip system interconnect. The *Request FIFO* buffers incoming Translation requests from the FUs which are then read-out by the *Requester* unit. The *Requester* unit starts a TLB request and, simultaneously, prepares to read a GATe using the right GAT base address from the register file. When the TLB answers, the *Request* unit either forwards the GATe read request information to the HTAX outport (in case of a TLB miss) or forwards the necessary information to the *Responder* unit (in case of a TLB hit). The *Reorder* buffer accepts incoming GATe responses and reorders them in request order (read responses may arrive out-of request order in a HyperTransport system, as in most modern system interconnects). The reordered responses are then forwarded to the *Responder* unit. The *Responder* gets ATS response requests either from the *Reorder* buffer as result of a GATe read request or jointly from the TLB and *Requester* if it the result of a TLB hit. The *Responder* builds an ATS response and forwards it to the *HTAX Outport* which transmits the packet across the HTAX to the requesting FU. Flush requests originate at the registerfile and are passed through the same units. The flush responses are collected at the *Responder* unit and the result written to the register file. Flush commands also trigger a corresponding flush of the TLB entry through the TLB connection to the register file.

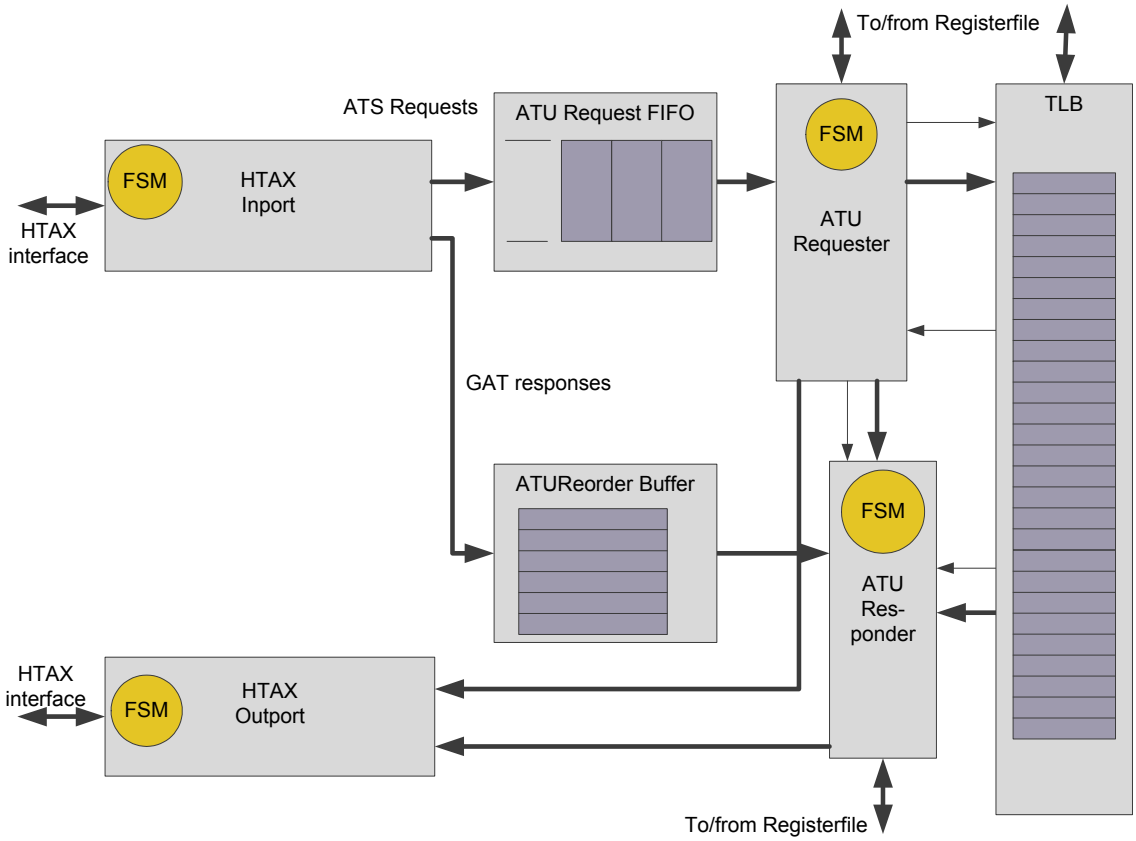


Figure 5-12: ATU Architecture: Block Diagram

Not shown in figure 5-12 are the actual control and status registers, which reside in the EXTOLL register file. The EXTOLL register file is specified and generated using a unique flow described in section 7.2.

5.5 ATU Verification and Implementation

The ATU was verified on three different levels. First the individual modules were verified using custom testbenches to establish the specified functionality of these modules. Next the ATU core, that is all modules except the HTAX ports were tested using a testbench. Different requests were simulated. Lastly, the unit was verified within the full EXTOLL system simulation including all of the other EXTOLL modules and the HyperTransport Bus functional model [113] to drive the HT core of EXTOLL. Testing in this stage was performed closely coupled with the RMA unit since this is the client FU of the ATU.

The ATU was implemented on a Virtex4 FPGA, within the current EXTOLL implementation platform. The result in resources if the unit is synthesized, placed and routed alone are summarized in table 5-7.

Resource type	absolute	relative
Flip-Flops	829 of 84.352	~ 1 % of FPGA
Slices	936 of 42.176	~ 2 % of FPGA
18 kb block rams	8 of 376	~ 2 % of FPGA

Tabelle 5-7: ATU Resource Usage

For this test the unit was synthesized using a 1024-entry direct mapped TLB, a 32 entry reorder buffer and a 64 entry request FIFO using Xilinx ISE version 10.1 with a Virtex 4 FX100-11 FPGA as target device. Within the EXTOLL implementation the ATU uses about 2 % of the resources of the complete design (see also section 7.7).The ATU alone reaches ~ 250 MHz of clock frequency on the FPGA, so the EXTOLL core clock frequency of 156 MHz is easily reached. These results shows the excellent performance and resource utilization that is reachable through the optimized architecture and implementation of ATU.

5.6 Performance Analysis

The first performance numbers that are of interest are the microarchitecture related results that are gained using cycle accurate simulation of the Verilog HDL specification of the module.

Path		cycles	absolute timing in EXTOLL prototype
TLB Hit		9	~58 ns
TLB Miss	complete	20 + main memory access	~128 ns + main memory access
	from ATS request to GAT read	11	~71 ns
	from GAT response to ATS response	9	~58 ns
Invalidation	complete	7 + variable time in FU	~ 45 ns + variable time
	from RF to ATS request	4	~26 ns
	from ATS response to RF	3	~19.2 ns

Tabelle 5-8: ATU Latencies

The raw hardware latencies are summarized in Table 5-8. So the minimal translation latency is nine cycles and in the case of a TLB miss 20 cycles plus a main memory access have to be accounted for. In the real system a translation latency between 50 and 330 ns can be expected. For the invalidation, there is the time spent within ATU and then, in addition the FUs can delay the completion of the invalidation by a variable amount of time. It is possible to flush TLB entries every second cycle.

ATU was benchmarked in the same system as the Mellanox HCA[1] to enable a direct comparison. If operations caused only TLB hits, in excess of 10 million operations were measured on the prototype hardware running at 156 MHz. With TLB misses the throughput diminishes, of course. Only about 1.2 million operations have been measured in this case. So, depending on the communication pattern, a throughput of between 1 to 10 million translations can be performed by ATU in the prototype system.

The same measurements regarding memory registration that have been performed using the Mellanox Connect X IB HCA were repeated with the ATU. The results of the registration latencies for different registrations sizes are plotted in figure 5-13. The results from the Connect X HCA are also plotted for reference. Memory registration latency starts below 2 µs and then increases mostly linearly with the number of pages that need to be registered. The plot is not completely smooth, but has no very high peaks as the Mellanox measurement features. While the difference is not as important for large sizes, the plot shows that ATU outperforms Connect X at all sizes

Figure 5-14 shows the plot of the deregistration latencies. Deregistration latency starts also at about 2 µs and then linearly increases. As with the registration latency, ATU outperforms Connect X at all sizes. For the deregistration the advantage in latency is a little more pronounced than in the registration case.

Finally, the distribution of registration and deregistration latency of a multitude of runs of 16 kB size each was measured. This measurement gives an insight how stable the registration/deregistration operations are in terms of consumed time. The results are shown in figure 5-15 and figure 5-16. The plot clearly shows the high stability and predictability of the ATU registration and deregistration processes.

The goal of an efficient address translation unit for the EXTOLL project was reached with ATU. The analysis showed, that ATU is very economically in terms of resources and at the same time delivers very good performance. Besides the good translation performance, which will become even more apparent when analyzing the performance of EXTOLL RMA, ATU also provides for an excellent registration/deregistration performance. This can help in developing middlewares and applications with highly dynamic memory access patterns and increases the flexibility of ATU and EXTOLL by a great deal. ATU is a novel contribution featuring a highly compact, very high-performance architecture for address translation in devices. The approach of tables that are shared by the same context both simplifies and accelerates the design and makes it possible to reach performance rates that compete very well even if the ATU implementation is only done in an FPGA.

1. Linux 2.6.24, 4x Quadcore Opteron system (2.2GHz), 16 GB RAM

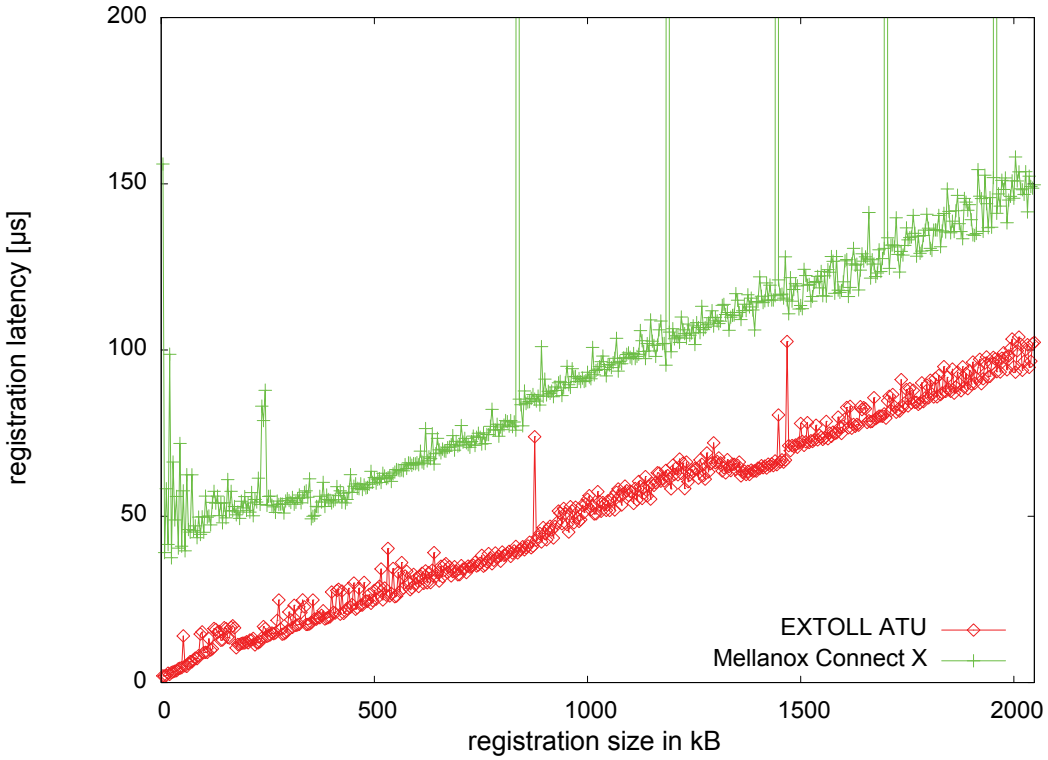


Figure 5-13: ATU/Connect X Registration Latency

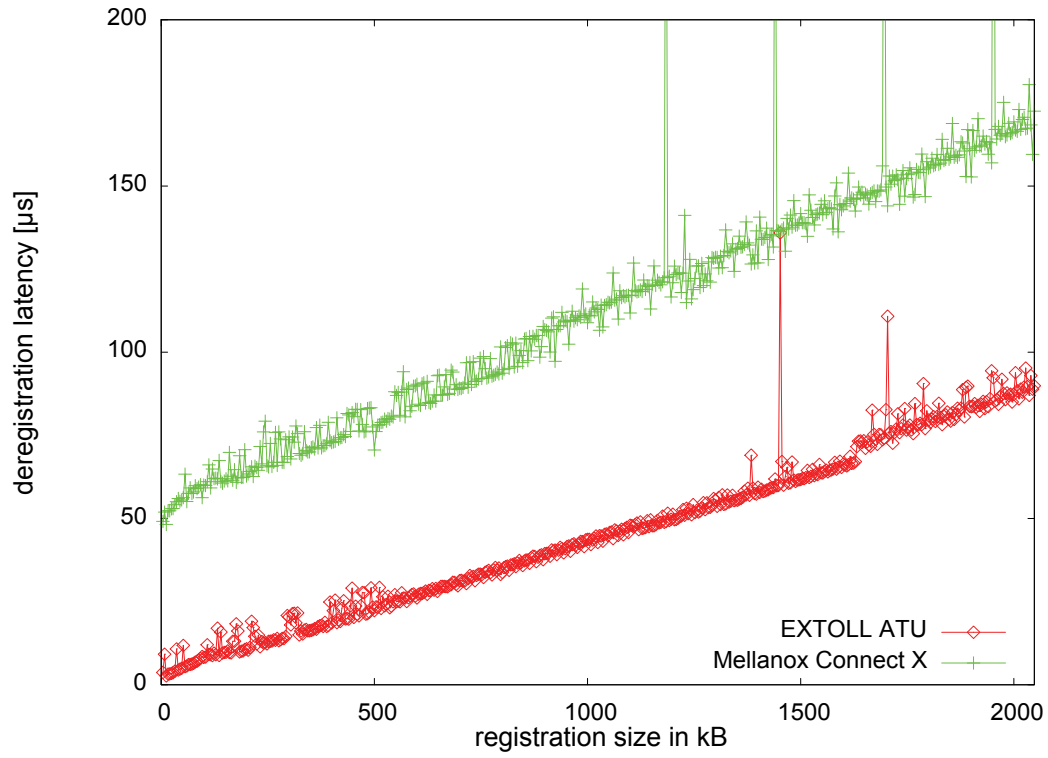


Figure 5-14: ATU/Connect X Deregistration Latency

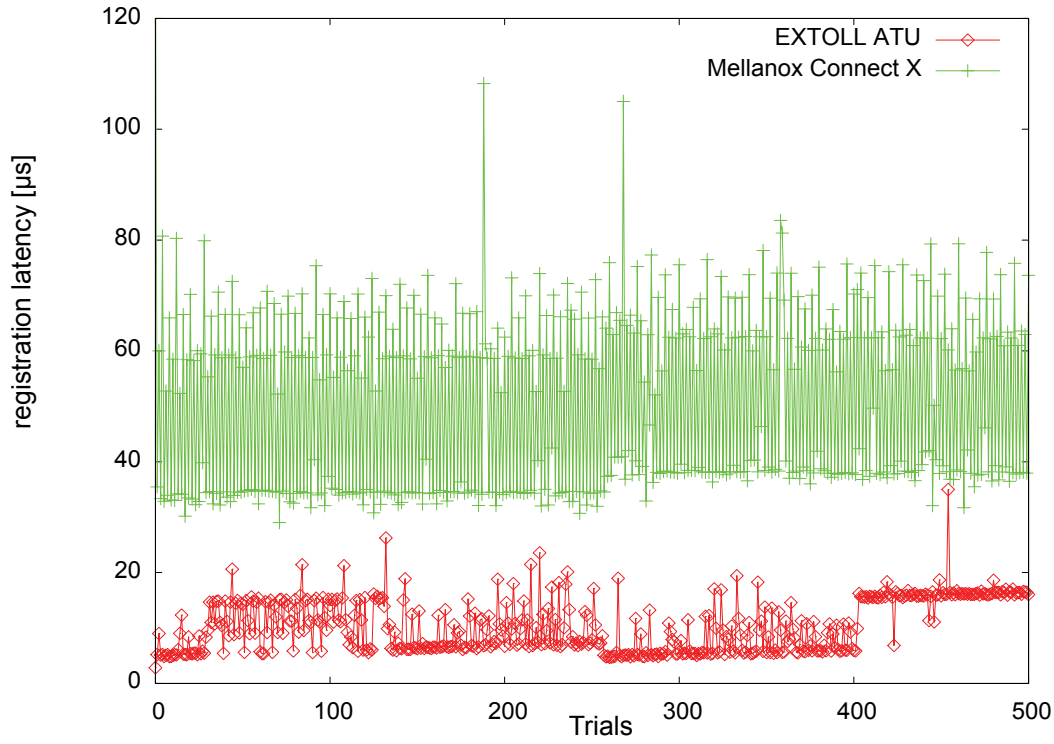


Figure 5-15: Distribution of 16 kB Registration Latency

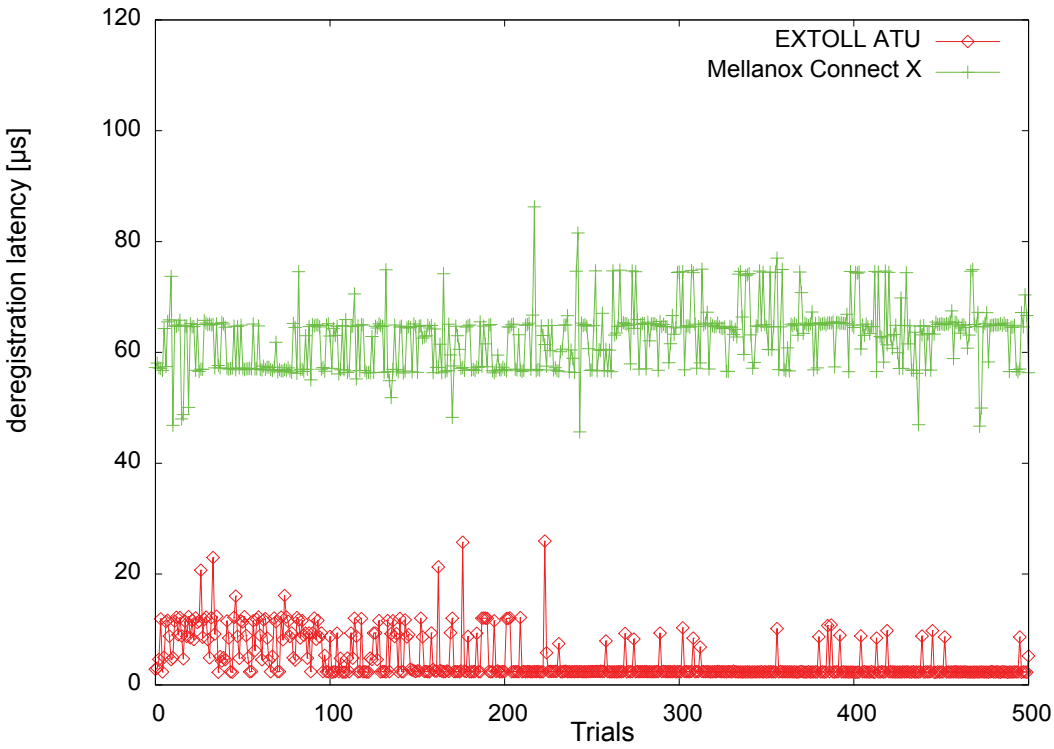


Figure 5-16: Distribution of 16 kB Deregistration Latency

5.7 Future Extensions

There are a number of extensions that can be developed for the ATU. One modification is to move the GAT from host main memory to memory connected directly to the device. Depending on the memory technology this may improve translation latency in the miss case. For example, the HTX Board features 256 MB DDR2 SDRAM which can be used to hold the GAT tables. The latency to access this memory was measured to be 75 ns at 200 MHz; the necessary DDR2 IP core uses about 906 Flip-Flops, 1090 LUTs and 32 BRAMs on the Xilinx chip. Conceptually, the architecture is not changed from this extension. The extension can reduce t_{miss} down to ~200 ns. An implementation using high-speed external SRAM will of course run even faster, but this modification increases the costs for an adapter considerably.

The second improvement is to substitute the direct-mapped TLB for a more complex and more powerful design. One possibility is to implement a set-associative or fully-associative TLB on chip, or to use a commercially available *network search engine* chip for the associative search. These chips have been designed for IP routers and can handle 64k entries in four associative sets, providing a powerful TLB mechanism. Again, the extensions are improvements to the implementation but do not change the fundamental architecture.

The last improvement is to add a memory-mapped command interface which would allow to pass larger jobs to the ATU asynchronously. This can be used to further improve deregistration performance but increases the area utilization of ATU.

Chapter 6 Transactional I/O

This chapter analyzes the problem of issuing operations to a device in the most efficient and virtualized way. This chapter presents a design space analysis for this problem. Also, several studies have been performed and are presented here to quantify the effects of several promising solutions. Device virtualization is a problem encountered at the TX function of a NIC first. The RX side is often more straight-forward, as is the notion of operation completion (for asynchronous and split-phase transactions). All of these aspects fall under the category of device virtualization and are thus covered in this chapter. Note that device virtualization is orthogonal to the memory virtualization issues which have already been discussed in the previous chapter.

A new mechanism to access devices is introduced which addresses the problems by speculatively issuing an I/O operation with data operands. In the case the operation is interrupted or fails on the device, the state of the system before the operation was started is restored and the operation can be retried. Since these semantics are very similar to the well known method of Transactional Memory [114] which is used to access memory in an efficient, atomic way, the new method was named *Transactional I/O*. The mechanism needs special support in the CPU, though, but it will be shown how emulation of the behavior using available technology can be accomplished.

6.1 EXTOLL Requirements

Device virtualization names the process of making a single device (I/O resource) simultaneously available to multiple clients (processes or threads). Since concurrent client access has been identified as a mandatory feature of EXTOLL and the employed method can have a major impact on performance and latency of the system, this chapter analyzes the different approaches for EXTOLL.

Traditionally, access to devices has been virtualized by the OS; more recently, high-performance applications have driven the need to access devices from user-space. The growing increase of the number of CPU cores in systems and the more and more pervasive nature of HPC to enterprise work-loads drives the need for low-latency device access with simultaneous device virtualization to serve all these clients. Device virtualization can roughly be classified into the traditional approach, hardware replication and self-virtualizing devices.

The requirements for device virtualization for the EXTOLL project can be summarized as follows:

- EXTOLL should support an arbitrary number of contexts (in the architecture; there may be implementational limits imposed).
- It is necessary to achieve low latency and high bandwidth.
- The method should strive to minimize the amount of state information and if possible eliminate state altogether.
- Operations may fail due to resource constraints, but if they do, this needs to be gracefully and software needs to be able to retry them. Finally, this needs to lead to a successful termination of the operation. The access to the device needs to be fair.
- At all times, standard security rules need to be obeyed (one process may not interact with another process unauthorized etc.).

The third point actually is a corollary of the first two points, since statelessness (or at least state minimization) makes it easier for the architecture and a particular implementation to support a large number of contexts. At the same time, less state information means that less access to state information is necessary to perform an operation. This causes a minimization (or elimination) of the latency cost of device virtualization. Usually bandwidth is no problem with all device virtualization approaches considered, so it will be omitted from the discussion. Following from the points above, three additional requirements can be identified:

- The device must be virtualized.
- It must be possible to insert commands including some payload directly to facilitate low latency communication.
- Self-virtualized devices demand atomic insertion of work queue elements.

The following sections briefly introduce the three main virtualization classes. The class of self-virtualized devices is analyzed in more depth. Several studies have been performed to better quantify the performance metrics for self-virtualized devices. Next, an ideal solution to the virtualization problem is presented called *Transactional I/O*. Unfortunately, this method can only be implemented with changes in both the CPU and the CPU-device interconnection network. Thus, the solution chosen for EXTOLL emulates some of the *Transactional I/O* behavior using facilities available in today's system. This method is based on *Central-Flow-Controlled Queues* (CFCQ).

The last sections are dedicated to the problem of operation completion and receive side contexts. The chapter closes with the presentation of the actual virtualization architecture employed by EXTOLL.

6.2 The Classical Approach

The classic approach for device virtualization is software based. Typically, the operating systems multiplexes and also abstracts access to the device. This involves a system call for every I/O operation to first change into the OS context. Often, this approach also involves one or more memory copies before the operation has finished. Another typical characteristic is the usage of interrupts to enable asynchronous progress and completion of operations.

Besides user space communication, the recently intensified interest in OS virtualization adds another dimension to the virtualization approach. In virtual machine environments, several trends can be identified to handle I/O:

- Actual devices are handled by one domain, other domains access the device via special virtual devices (a software construct) which communicate with the original device driver via the Hypervisor.
- The device is exclusively owned and used by one domain.
- The device is owned and handled by the Hypervisor; access to the device is possible via a special virtual device which connects domains with the Hypervisor.

Since high-performance storage and network access plays an important role in enterprise computing, the performance penalty introduced by these software virtualization techniques is relevant and active efforts are under way to reduce this.

For HPC, user-space access to I/O is generally regarded as an important measure to ensure low latency and high bandwidth for computing intensive applications. Classic devices which use the OS cannot employ user-space access and thus suffer from the increased overhead by always going through the OS. Traditional devices only have one context, so that there can only be one user. OS mitigates this constraint by performing request multiplexing/de-multiplexing. This is the actual process of device virtualization.

In terms of performance, many traditional I/O mechanisms also apply an elaborate software stack in addition to OS multiplexing before the actual data is transferred from or to the device. A good example is the TCP/IP stack for networking. To send/receive data to/from the network, applications do not only have to perform a system call to enter kernel space, but all communication is also passed up respectively down the stack involving typically at least one additional copy. See [20] for a detailed breakdown of the latencies involved in a modern, kernel-based communication system. It shows that large parts of the latency are due to software stack and interrupt related issues. In the evaluation part of this thesis, a comparison of user-space versus (stackless) OS based communication is presented quantifying the overhead of OS passing in a best case scenario. Measurements performed for this analysis have shown that the pure overhead of performing a system call is in the order of 250 - 300 ns on a 2 GHz Opteron processor. Additional software overhead has to be taken into account to manage multiplexing/demultiplexing of different clients. At the receiver side, polling or interrupt driven completion are possible. Traditionally, interrupt driven approaches have been employed. There is an important reason to do so: polling in a kernel driver usually leads to problems, from processor time accounting to not being trivially able to kill a process. Interrupt driven completion adds more overhead to the communication. Again, measurements on the test platform show an interrupt latency of at least 600 ns. It is also often necessary to check the cause or source of an interrupt and clear it which may lead to even higher latency. After an interrupt has been received, the process that is the receiver of the message can be woken up. This is the final part of the overhead of this method and the time until the process is actually scheduled to run again on the CPU may be relevant since this involves a real context switch which can easily take in the order of several micro-

seconds. So in summary, the traditional solution is not latency optimized and thus, while the method of choice for many I/O tasks, is not the right choice for a low-latency, high-performance network for parallel computing.

6.3 Hardware Replication

One way to at least give a small number of users access to the function of a device is replication. For example TX and RX DMA engines of a NIC may be replicated four times allowing four concurrent clients to use the NIC. Examples of this design are ATOLL (chapter 2) and the recently announced SUN 10GE Ethernet Adapter [21]. The block diagram of Sun's architecture is shown in figure 6-1. The adapter uses several replicated ports which support,

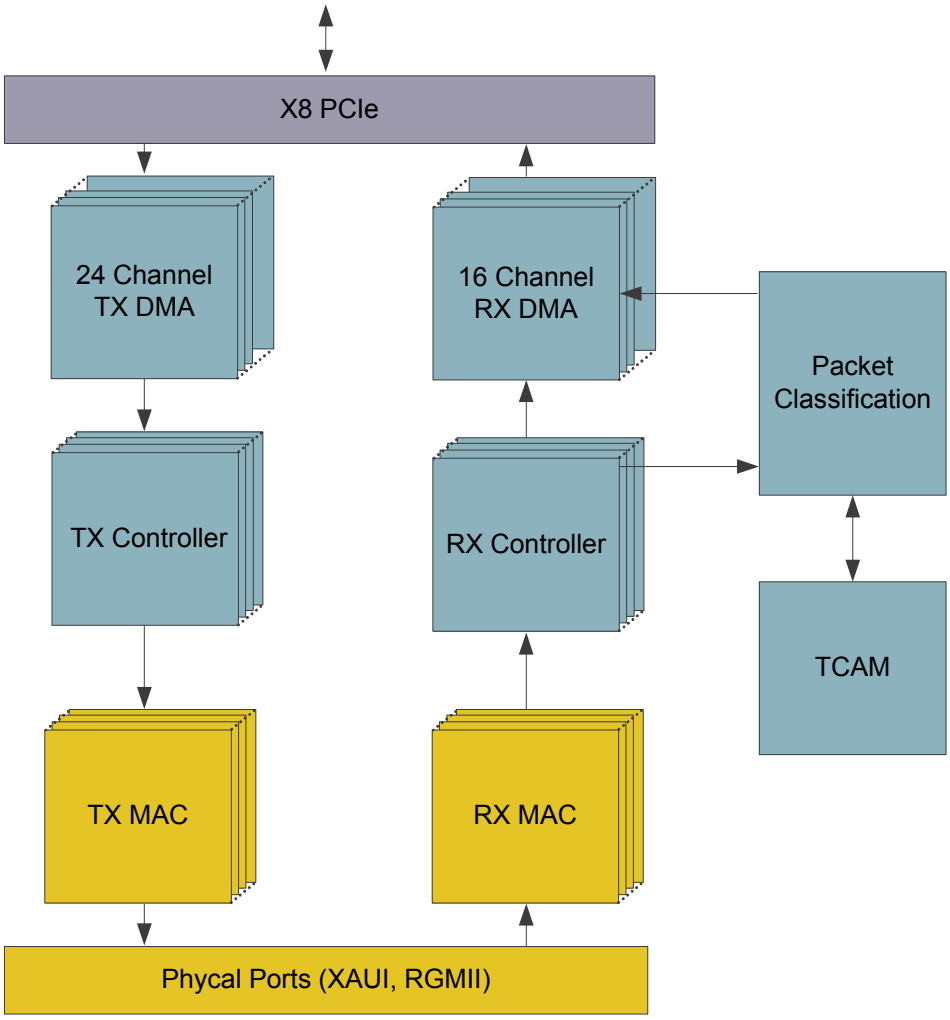


Figure 6-1: SUN 10GE Adapter

as SUN states, “efficiently virtualized environments”. The classification unit together with the associated TCAM allows the adapter to enqueue incoming packets based on their affiliation to a traffic or stream class.

As was already mentioned in section 2.9, the generally very limited amount of possible clients is a major drawback of this method. Therefore the architectural family of self-virtualizing devices was introduced [115].

6.4 Self Virtualized Devices

A number of self-virtualizing devices and architectures have been proposed. These devices often feature maximum context counts in the hundreds (512 in a recent Mellanox ConnectX HCA) and are most often firmware based, i.e. not fixed function but to a certain amount programmable. [115] first introduced a completely self virtualizing NIC architecture which enables a fixed function device to scale to many thousands of simultaneously active contexts. It is generally believed that fixed functions (i.e. FSM based) devices provide a better performance as programmed devices (micro-coded or processor based) which often reach lower clock rates.

As a general rule, self virtualizing devices can be classified by context organization and command issuing; the different design choices are shown in figure 6-2.

Context state can be held in on-chip memory, on-board memory or main memory. One design principle useful for low-latency designs is to minimize state to enable as much on-chip state as possible. For certain functions it is actually possible to eliminate context state altogether leading to stateless virtual functions, a special case off on-chip state. Early Mellanox HCAs are an example for context state held in on-board memory. Later, state moved to main-memory and the InfiniHost Context Memory (ICM) was added to the HCA as an intermediate layer which translates scattered physical main memory holding state information in a contiguous memory space suitable for the Mellanox HCA. This process is similar to the function of a GART (section 5.1.8). The virtualized device in [115] stores all of the state information in main memory but employs extensive on-chip caching to accelerate the common case.

The method of issuing commands to a virtualized device is also an important property of the device. Several possibilities have been proposed including: kernel-based triggering, memory-mapped door-bell registers, the triggerpage mechanism in conjunction with a conditional store buffer (CSB) [115] [116] and here the novel approaches of *Transactional I/O* and the *Central-Flow-Controlled Queue*. The triggerpage design incorporates a central command queue for all contexts. Commands are issued to this queue using a load instruction from the CPU. The offset within an address space window which maps to the queue is used to carry the actual command. The response to the load instruction then designates if the command issue was successful or not. The main reasons for failure include resource problems (the queue is full) or security violations. This method is very appealing and offers an elegant way to manage a central, shared command queue for a virtualized device regardless of the host-device interconnect. The downsides are very limited command size (only a few

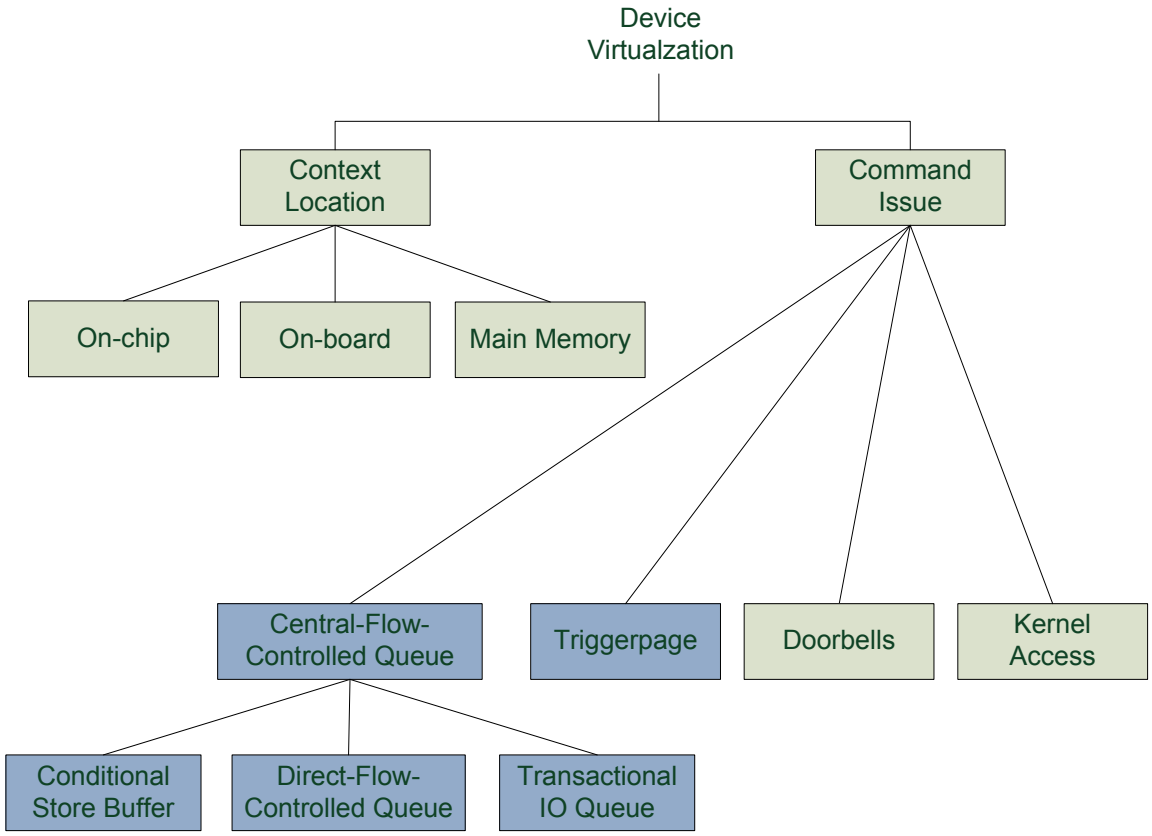


Figure 6-2: Virtualization Design Space

bits in the address are available) and the blocking semantic of an I/O read instruction in today's processors stalling the CPU unnecessarily. The first problem is remedied by the introduction of off-chip, per context command-descriptor queues. The central queue then only serves as a kind of a doorbell to notify the device of new entries and is one of the exclusive context-associated main-memory queues. This of course leads to additional latency due to main memory access(es) by the device.

6.4.1 Triggerpage Study

This method has been tested and analyzed on the HTX-Board. An RTL (Register Transfer Level) model of the triggerpage was written in Verilog HDL. Together with the HT-Core an FPGA design was completed which was implemented on the HTX Board. Figure 6-3 shows a diagram of the system and a block-level diagram of the implemented hardware.

The second part of the this study involved to write the necessary test software to actually use the triggerpage hardware. To this end two programs were written: one client program and one master program. Client programs simulate user threads accessing the triggerpage from user-space, i.e. issuing commands to the hardware. The master program controls the triggerpage design and removes entries from the CSB. The removal of entries from the master program as well as the insertion of new commands from the clients can be caused from

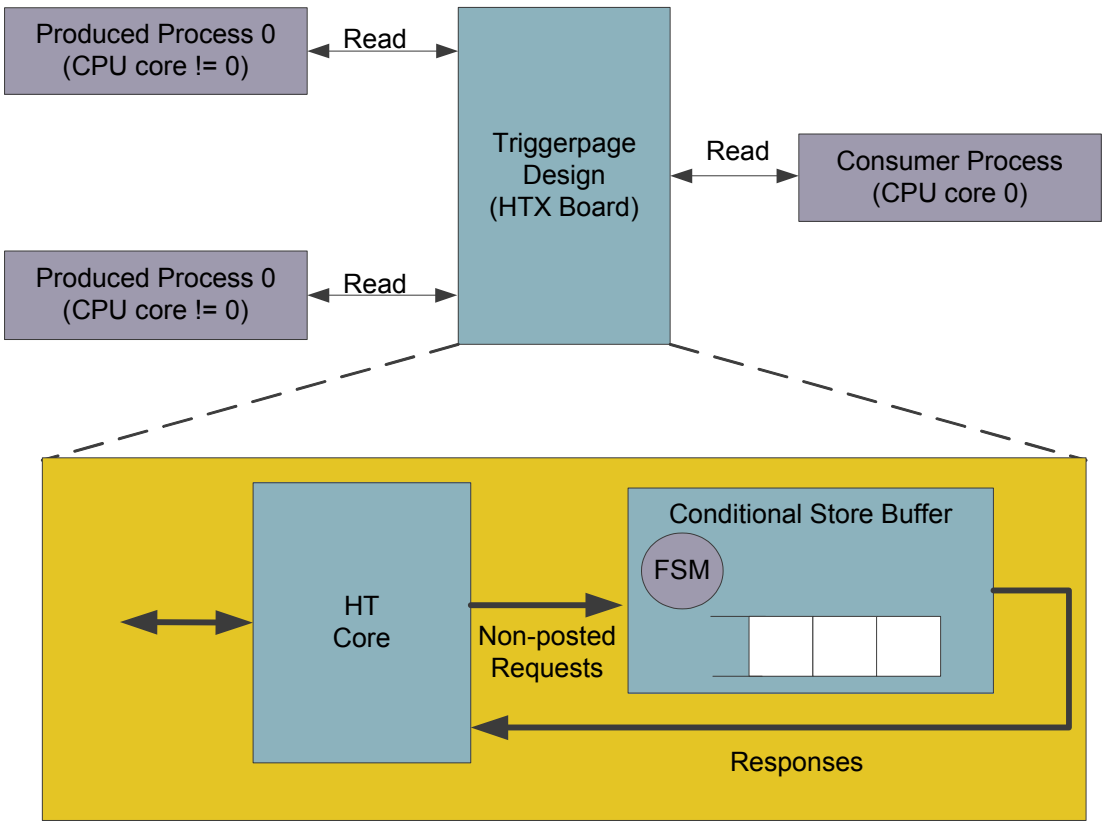


Figure 6-3: Triggerpage Experiment

pseudo-random number generators. Both programs directly access the hardware from user space. To this end, the *PCI base address region* (BAR) of the hardware design was memory-mapped into the applications virtual address space. The page offset of the mapped region (i.e. the address bits beyond bit 11) decide which context an access is assigned to. Page 0 is the master context and used to remove entries from the FIFO. Page 1 to N are then used to insert commands originating from context 0 to N-1. The context can be given to the client test application on the commandline; so it is possible to simulate traffic from different contexts.

Two tools were used for measurements which confirmed the theoretical analysis to be true: measurement of CPU clock ticks in software and hardware measurements using Xilinx ChipScope [117].

Triggerpage Results

The most important result is that the triggerpage method performs as expected. The triggering of a command, i.e. the notification of the arrival of a new command is as fast as the first half of a read-round-trip from the CPU to the device. Generally, the device has then to issue

a DMA read command to fetch the complete command descriptor since usually the available bits that can be encoded in the trigger address are not sufficient. For the hardware to be able to start working on a new request, a minimum latency of

$$t_{start} = t_{np-req} + t_{dequeue} + t_{dma}$$

is necessary. From table 3-1, it becomes clear that on the tested system the startup latency can be given with $t_{start} > 90 + 10 + 280 = 380$ ns. On a system where the device is located at a greater distance from the CPU, for example current PCIe systems, the startup latency grows linearly with the interface latency. Current FPGA based PCIe designs feature roundtrip latencies in the order of 1 µs, resulting in a $t_{start} > 500 + 10 + 1000 \approx 1510$ ns, more than 3 times the HyperTransport startup latency.

Another important characteristic is the *command issue rate*, which forms an upper bound on the number of communication transactions that can be performed by such an architecture. The command issue rate is fixed by the read latency of the architecture in conjunction with the associated software overhead:

$$r_{issue} = \frac{1}{t_{read} + t_{overhead}}$$

So, if a $t_{overhead}$ of 0 is assumed, the tested system can sustain an $r_{issue} < 5.2 \cdot 10^6$ with a $t_{read} \approx 190$ ns. Again, since the CPU-device roundtrip latency goes into the equation, a low latency is mandatory for a good performance.

To asses the NIC transmit latency when employing the triggerpage method, it is necessary to also take the complete NIC architecture into account. For a NIC architecture as presented in [115], the transmit latency can be approximated by

$$t_{tx} = t_{start} + t_{context} + t_{vci} + t_{dma}$$

Where t_{start} is the above triggerpage latency, and $t_{context}$ is the time to fetch the necessary context. t_{vci} denominates the time to fetch the virtual communication instruction (i.e. the TX descriptor) and t_{dma} the time to actually copy the payload using DMA and pass it on to the network. As a simple lower bound, each t_{vci} and t_{dma} need to access main memory; the loading of the context needs either to access main memory (context cache miss) or on-chip resources (context cache hit). In the HTX Board case this sets the lower bound (without address translation and assumed context cache hit) for TX start-up latency at about $t_{tx} > (100 + 2 \cdot 280) = 660$ ns. Address translation adds at least 50-60 ns (using ATU timing), so the complete TX latency would be more than 700 ns on the current EXTOLL technology.

6.4.2 I/O Transactions

Ideally, the complete command with all necessary operands (i.e. the command descriptor) can be inserted into a central command queue in an atomic fashion. Since the queue can be full, the operation can fail. A new method that satisfies these requirements is named *Transactional I/O*. This methods introduces a device structure together with an instruction set

architecture (ISA) extension and an interconnect extension to support I/O much more efficiently. This extension allows true atomic transactions of up to 64 byte in size. If the transaction can either not be handled atomically or not be completed by the device, the transaction is aborted and software is informed so that the transaction can be retried.

The idea for *Transactional I/O* is very simple:

- A direct I/O operations in a virtual environment can fail, if resources are not pre-allocated which leads either to poor resource usage or considerable management overhead.
- I/O operations must be atomic, so that the device is able to efficiently distinguish operations from different sources and does not have to keep state for started but not yet finished operations [1].
- Operation start-up should be latency optimized: a single I/O transfer should suffice.

These requirements lead to the *Transactional I/O* method. A special CPU instruction called *iocommit* is introduced which transfers a number of registers (the use of wide, multimedia registers like XMM in X86-64 is suggested) in one non-posted I/O transfer directly from the CPU to the device. The operation should support the transfer of at least 64 byte en-block. The operation itself is atomic. If the process is un-scheduled right before the instruction is issued, the CPU registers are part of the architectural state of the CPU and saved by the context switching mechanism.

The *iocommit* instruction causes a non-posted transfer, meaning a response is expected. This response is generated by the device and sent back up-stream to the CPU where it is directly stored in a register available for immediate inspection by software. If the return value indicates success, the application can proceed, otherwise a retry mechanism is used. Since the response is transported to a register, a full scalar integer data type can be used to indicate additional information to software.

The whole sequence of triggering a command in the CPU is summarized in the following pseudo-code:

1. Load command descriptor and data into CPU registers.
2. Call *iocommit* instruction.
3. Check return value in general purpose register (for example in RAX in X86-64).
4. If value indicates failure, jump to step two.

On the device the following sequence of events is triggered:

1. A non-posted packet containing the command descriptor/data arrives at the host interface.
2. The device checks if it can accept the packet.
3. If the packet was accepted the necessary response packet carries a success indicator value.
4. Otherwise a response containing failure is generated.

1. While the idea of holding the state of started operations seems appealing at first, consider the case of a process that gets unscheduled during the posting of the operation. It may well be that the process is resumed milliseconds or more later. In this time potentially thousands of other operations may get started and finished. In essence, this solution would mean to hold memory space for one unfinished operation for every supported context.

The needed transaction on the interconnect is a standard non-posted write (as defined in HT or PCIe) with one additional feature: generally non-posted writes only generate a *target done* or *target abort* response. In this case a response packet analogous to a read response must be provided, carrying data.

A somewhat more restricted, but still useful implementation could also use *non-posted write* transactions and *target done* packets to signal success or failure. The error bits in the response control packet encode the operation status. This possibility needs no changes to the host-device interconnection network.

This method performs in the average case, i.e. no contention on the queue, very good. The start-up latency can be approximated by $t_{start} = t_{np-req}$, thus around 200 ns on the HTX-Board platform. Since the complete command descriptor is transferred to the queue directly, the issue rate remains the same as in the triggerpage study, since the non-posted (read) latency still causes the upper bound of this rate.

6.4.3 Central-Flow-Controlled Queue

The newly introduced *Transactional I/O* method is not applicable to current systems; so a method was developed which sacrifices a minimum of the capabilities but is implementable on a current system. This reduced but innovative solution, called the *central-flow-controlled queue (CFCQ)*, is used for the functional units in this work. It becomes possible because HyperTransport is a flow-controlled interconnect. Thus, the HyperTransport flow-control ensures that the central queue never overflows. This approach allows commands (or entries) of up to 64-bytes to be inserted into the queue at once leveraging the write-combining feature of modern CPUs. If the queue is full and a thread posts an additional command, the CPU core gets stalled until the device returns a credit. This characteristic is also one of the two problems this solution still lacks to the optimal one: the CPU gets blocked if it posts an operation when the queue is full; the second problem relates to the use of the write-combining buffer for larger transactions. Such a transaction may be interrupted and in this case the queue operation is not atomic. The design has to handle this in a well defined way.

For the CFCQ method to work, the write transaction containing the command must be atomic. This is the case for all simple store instructions on x86 CPUs[1]. The method has to be used with commands or command descriptors which are smaller or equal in size to the maximum CPU store size, so that no additional DMA is necessary to start the operation. On AMD64 the largest data type that can currently be stored in one instruction is a 128-bit vector value using the SSE2 instruction set[2]. Thus, commands with a descriptor size of 16 bytes or less can be issued directly. Some additional bits are available by encoding them in the store address of the command issue transactions. Prior to implementing an actual design using this method, a study was performed to clarify two questions:

- Is an SSE2 store truly uninterruptible?

1. This is actually implementation dependent, and there are x86 processors which may divide a 64-store to I/O into two 32-bit stores
2. Mid-term future CPU generations may support 512-bit memory accesses.

- What happens if the queue is full?

Figure 6-4 shows the block-diagram of the used test design.

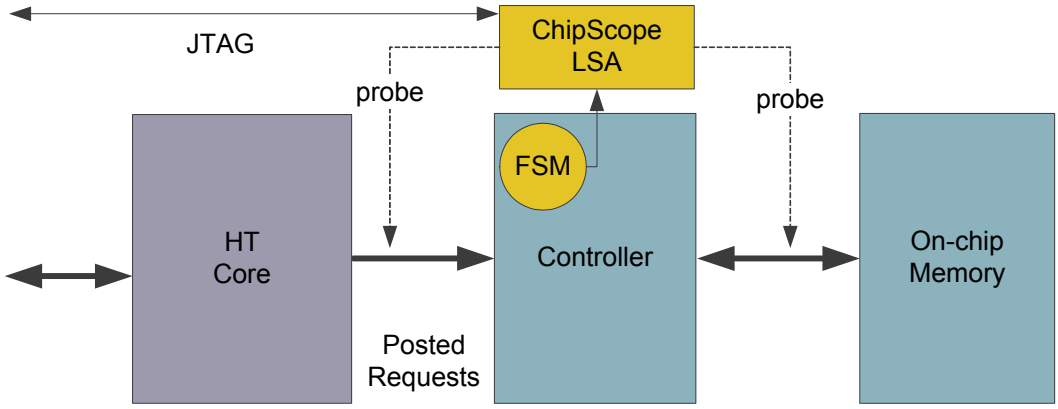


Figure 6-4: Central Flow-Controlled Queue Block Diagram

Software issues *stores* to the hardware. The controller block handles the *stores* and directs them to on-chip memory. For the study the actual values could also be discarded, but to check the correct functionality it is useful to be able to actually check the content of the individual transactions. To effectively view what is happening in the system several IP blocks from Xilinx ChipScope [117] were used. The general ChipScope core allows to instantiate an logic state analyzer in an FPGA design which was used here to monitor the timing of individual transactions. Additionally, the controller implements a number of counters, for example one counting all store transactions and one counting only transactions of a certain size or smaller. Using the Virtual I/O core of ChipScope, the counter could be read and written to from the ChipScope GUI. This design was used to prove in system, that the AMD64 CPUs really generate 128-bit stores if an *movntdq* instruction is executed. The result of this test was positive.

The next question was what happens when the central command queue runs full and the credit based flow-control of HyperTransport puts back-pressure on the load/store unit of the requesting CPU core. The result is actually, that the load-store unit of the CPU core causes the complete core to stall. This can be the cause of system deadlock so it is absolutely mandatory to ensure progress of the system.[1]

The performance of this method is excellent. The start latency is given by $t_{start} = t_{p-req}$ and evaluates thus to < 100 ns on the HTX-Board platform. The Transmit latency can thus be approximated with a lower bound of $t_{tx} = t_{start} + t_{dma}$ which equals the *Transactional I/O*. But since only a posted transfer is necessary, the issue rate is actually even higher and is $r_{issue} = \frac{1}{t_{write} + t_{overhead}} < 11.2 \cdot 10^6$ with an assumed overhead of zero and $t_{write} \approx 90$ ns.

1. It may be possible to use the Machine Check Architecture features of the CPU to detect a queue full condition. It is unclear though, what can be done to remedy the problem subsequently.

6.4.4 Central-Flow-Controlled Queue with Direct Data Insertion

This is a special case of the previous method. In addition to the command itself the complete payload of a communication operation is given to the hardware using a direct write interface to the requester to increase small-message bandwidth and throughput as well as reduce small message latency. To this end the write-combining buffers present in modern CPUs can be leveraged to transfer up to one cache-line (64 byte) of data in one direct transfer. SSE2 store instructions can be used to transfer the data in chunks of up to 16 bytes. After 64 bytes have been assembled in a write combining buffer, the buffer is flushed to actually trigger a single transaction on the I/O link containing the complete cache line. Partially filled write-combining buffers can also be flushed; the AMD64 processors cause a write transaction with the size of the fill-level of the buffer at this point.[1]

Usage of write-combining enables high bandwidth on I/O links using PIO accesses. Originally write-combining features were added to modern CPUs to accelerate interaction with graphics devices respectively video frame buffers. Write-Combining buffers get flushed, either if this is explicitly asked for using a *fence* or serializing machine instruction or if they need to be re-used. Context switching normally also causes the buffers to get flushed.

One difficulty arises, when an application writing to a device is interrupted: the write combining buffer can be flushed prematurely. Once the application is resumed it continues storing into the now again empty write-combining buffer and finally flushes the buffer. The result is a transaction to the device that is fragmented into two smaller sub-transactions. In an extreme case a transaction can be interrupted multiple times, between each instruction that is in the path of storing all of the data. Two important rationales follow these observations:

- The device needs to be able to deal with fragmented transactions.
- Minimization of fragmented transactions is desirable.

The hardware/software design from the previous section was reused to analyze the fragmentation behavior of a modern system. On the software side, three implementations of the actual function performing the transaction were considered:

- memcopy from glibc,
- a compiler intrinsics based SSE2 based implementation, and
- a hand optimized SSE2 based implementation.

The probability of an interruption is dependent on the length of the critical code path and the interrupt probability on the local CPU. To model different system environments the test software executed the transaction function both single-threaded and multi-threaded (to model a system were multiple threads/applications use a virtualized engine). Also the influence of the usage of different cores of a system was explored, since different cores may han-

1. Intel CPUs cause a sequence of individual integer size stores in this case which is of course less efficient and also not necessarily atomic on the I/O link.

dle different external interrupts. Another important factor is the *tick* frequency, i.e. the frequency of the time keeping interrupt which is also used for process scheduling. This interrupt occurs on every core, but its frequency can be configured within the Linux kernel.

Transactions of size 16 byte or smaller can always commit atomically, since they can be triggered by a single x86-64 transaction. Figure 6-5 shows the interrupt rate in ppm of 32-byte sized transactions for different parameters, including the core and the number of threads. It can be seen, that even under heavy load (high number of threads) the number of interruptions is very small. An interesting anomaly results of using only core 3 of the system, which exhibits very a very high rate of interrupted transactions. The reason for this phenomenon is that core 3 handles all of the external interrupts in the tested system including network and storage. Since the system was accessed using *ssh* a steady stream of network interrupts was generated on this core. But the interruption rate is still low in this case, less than one transaction in every 100,000 transactions is disrupted. These measurements were made using SSE2 based storing of the data. If moves with a smaller granularity are used, the interrupt rate is somewhat higher. The result is, that a hardware function implementing this method must be able to handle interrupted transactions. It may do so with a reduced efficiency, since the average case is a non-interrupted transactions. The VELO functional unit of EXTOLL which employs the CFCQ scheme with direct data as defined here, can recover from a split transaction and delivers such a transaction using two distinct network transactions. Software on the receiving node can detect the situation from bits in the message header and is responsible to reassemble the message.

The general performance parameters of this method are similar to the base CFCQ design, the difference being that the network transaction can be started without any further DMA, since the payload is already part of the command. Thus, t_x reduces to t_{start}.

6.4.5 OS Synchronized Queue

Since a transition into the privileged kernel space is relatively cheap on modern processors, It is worthwhile to study a device with a user-level software stack, but where the actual access to the device for example to trigger an action is performed in kernel space. This method offers three potential advantages: First, less resource usage is needed, for example address space, since context information can be passed to the hardware more efficiently (the OS is a trusted entity in the system). Secondly, the OS can perform additional multiplexing/ demultiplexing if hardware resources are exhausted (available end-points) thus enabling a seamless transition to pure software virtualization. Finally, this goes together with a pure software based address virtualization (section 5.2.2).

Measurements performed of the time it takes to enter and leave the OS on the same machines that were used above showed the cost to be around 170 ns. The result of this measurement is the latency given in table 3-1. So, starting of an operation from kernel-space costs at least an additional 170 ns. The results of this approach to a complete NIC application can be found in 7.5.5.

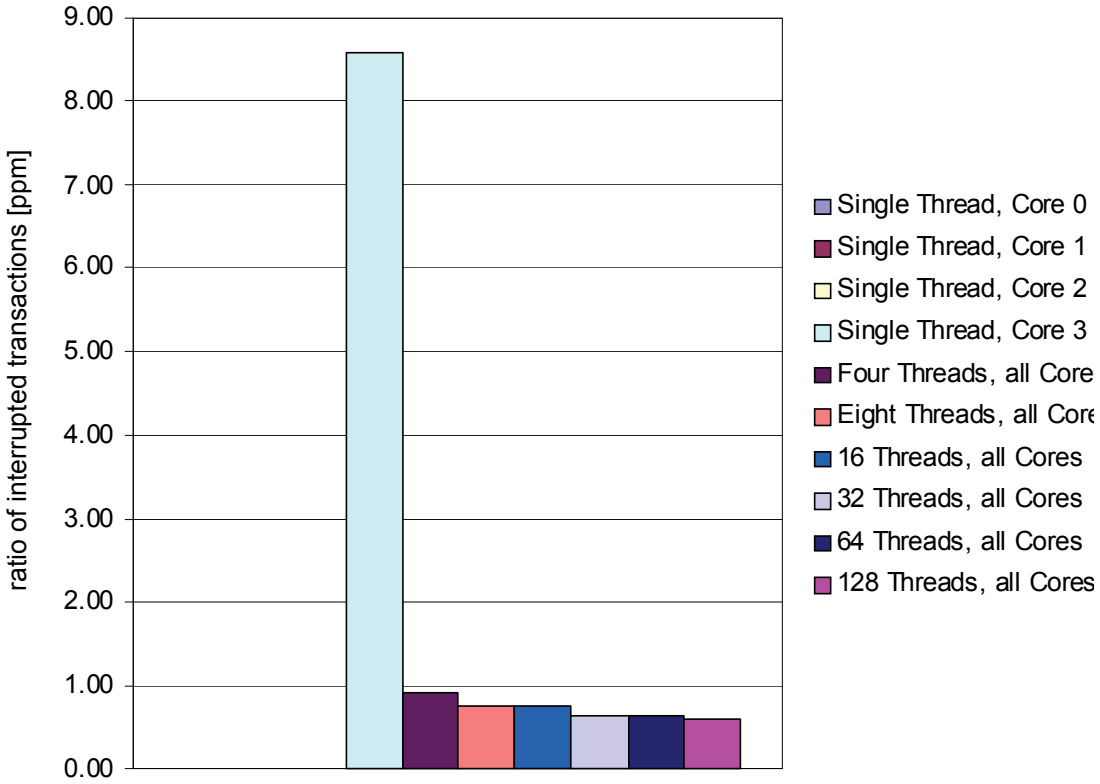


Figure 6-5: CFCQ 32-Byte Transaction Interrupt Rate

6.5 Completion Notification

In all the above designs, the completion of an operation is also an important factor. For example, when requesting a NIC to transmit some data, it is mandatory to know if and when the buffer to be transmitted can be re-used. Figure 6-6 summarizes the design space which is available for completion notifications.

The first choice is whether the CPU be informed using interrupts or if software has to actively poll some register or memory location. Interrupt driven notification enables asynchronous behavior and may remove CPU load in certain scenarios. On the other side, polling features a much lower latency. Consider the hardware interrupt latency measured using the HTX-Board of at least 700 ns. Additionally, the interrupt handler has to execute some code, possibly accessing main memory or even worse device registers. Usually it is a user-space application that is actually interested in the completion event. The interrupt handler thus, has to mark the process runnable and trigger a scheduling event in the OS kernel. In [20] approximately 5 µs are used for interrupt handling, soft IRQ handler call and application wake-up. For the EXTOLL communication units a hybrid solution was chosen: the architecture is optimized for low-latency polling solutions, but interrupt generation can be

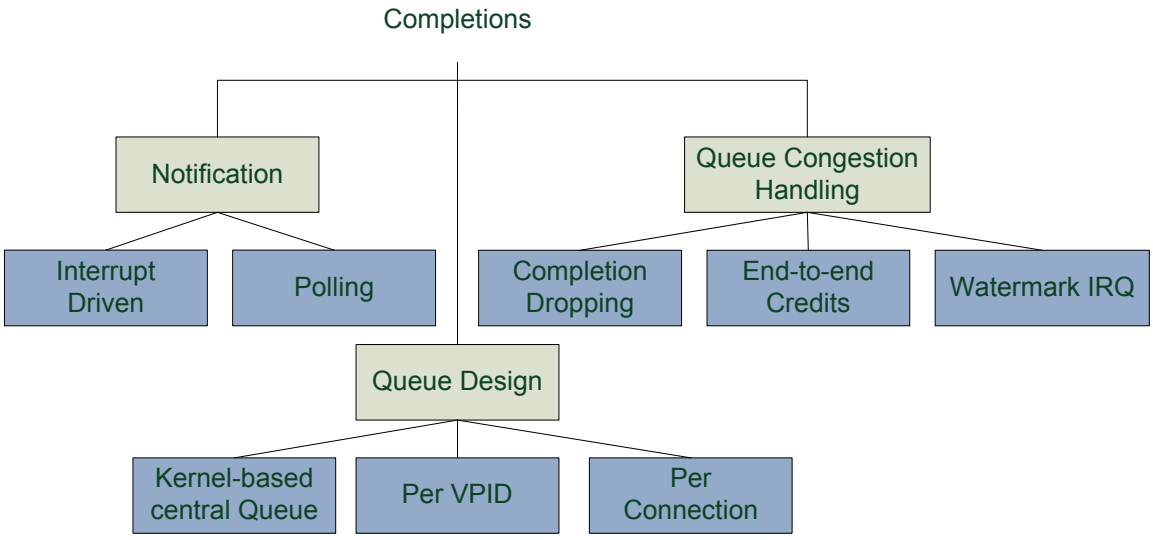


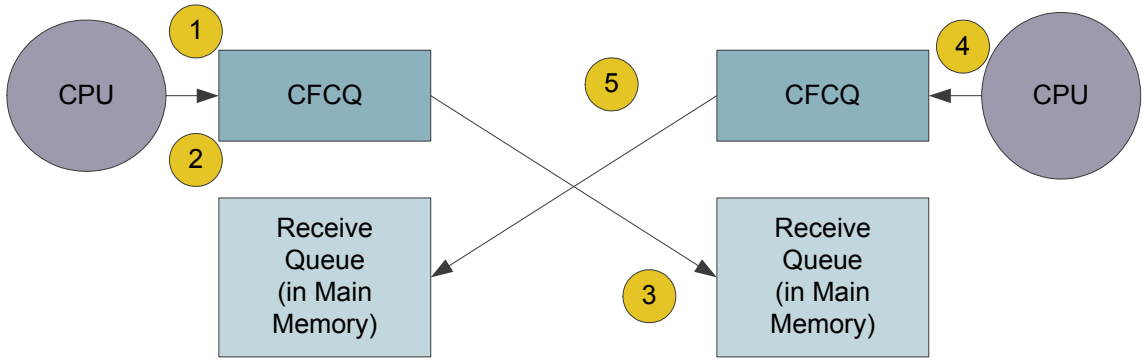
Figure 6-6: Design Space of Completion Notification

turned on. Since interrupt generation is selectable per VPID, interrupts can be used from a VPID that is used by an in kernel EXTOLL client (for example a cluster file system) while synchronous polling is used for other VPIDs in use by MPI processes.

The question of the queue design is resolved using one queue per VPID as this offers the most advantages. Solely in-kernel solutions can be dropped since it is an EXTOLL requirement to support user-space communication. The per-connection queue design has the major drawback, that a number of queues must be polled simultaneously. That means that the polling latency (i.e. the latency from the actual event until the notification arrives at the application via polling) increases linearly with the number of communication partners. Single queue designs eliminate this overhead all together. On the downside single-queue designs may cause a more complicated flow-control scheme to be necessary. For both communication units described in this thesis, a single queue design was used.

The preceding paragraph already touched the subject of completion queue congestion. If an operations causes a completion event which needs to be recorded in a queue, the question arises, what happens if that queue is full. Three possible solutions were identified: The simplest solution is to actually drop the completion. In this case it is an software error to cause a queue to become full. This solution is also adopted by other networks for example Infiniband. Watermark and interrupt based solutions inform the host CPU whenever a queue reaches a certain fill-level. In this case, the OS is responsible to clear the situation. The downside of this solution are increased overhead, as generally access to the queue can only be given through a critical section, since both OS and application may consume entries. Also, it may be difficult to estimate the right fill level at which to inform the OS. A last solution would involve a completion queue credit system, where requesters need to acquire a credit for the destination completion queue before they can initiate the operation. This can be implemented in software (and in fact is one solution for the software problem possibly arising when using the first solution) or in hardware or hardware assisted. It was estimated

that in the usage scenarios of EXTOLL, the first solution is sufficient. Note, that dropping of completion may be necessary in contrast to just waiting, since completion queue congestion plus CFCQ congestion may lead to system deadlock if a CPU tries to insert a new request that can not proceed because every resource is congested including the completion queue at the far end of the communication. Here, the local CPU tries exactly the same and is also blocked. A circular dependency has evolved. Since the HyperTransport flow-control stalls the CPU, the non-preemptive condition for a deadlock is satisfied as all of the other classical conditions. Figure 6-7 shows this situation which needs to be avoided by all means.



1 CPU 0 inserts a command into CFCQ 0

2 CFCQ0 is full and thus stalls CPU0 through the HT flow control

3 CFCQ0 is not drained because Receive Queue 1 is also full

4 CPU 1 performs the same operation as CPU 0, resulting in a similar situation

5 Both CPUs are stalled, receive queues can not be drained, deadlock results

Figure 6-7: Queue Deadlock

6.6 RX Virtualization

Virtualization on the RX side is based on the RX queue management and address management. Generally, queue based operations need to fetch the base address for the queue which applies to the receiving context. This is the reason why context-less, virtualized RX functions for two-sided functions are not possible. The same argument holds true for devices employing posted-receive queues or similar design schemes. One-sided accesses must translate the address but other than that need no context state. However, EXTOLL zero-copy functions still employ context state, since the completion of a receive operation can always be flagged to the receiving application using a completion queue.

Kernel based RX virtualization moves de-multiplexing into kernel space software, so it is necessary to inspect header data for the addressed communication end-point (queue or address), before the payload can be processed. There are several possibilities how this can be handled. Either the payload is copied into kernel buffers and the kernel copies the data then to its final destination (standard in TCP/IP stack implementations), or the payload is held back by the device and the copy is only performed by the device after the kernel has given the final physical address to the device. No current implementation of this method is known. It needs very large device buffers to handle the time between the arrival of a message and the time information about the final destination has been given by the OS.

Another path has been proposed and implemented by Intel with their *Intel I/O Acceleration Technology* [61]. This technique employs a programmable DMA engine located in the northbridge of the chipset. This engine can be programmed by the host in kernel mode (it uses physical addresses) to copy payload data from one spot to another (actually it can copy any data). This offloads the CPU of the final copy in a communication stack, for example the Linux TCP/IP stack. Yet, another special case is the kernel based translation approach as taken by the physical address RMA prototype, where the final address is already inserted into the communication by the kernel software on the TX side, so the RX side is trivial and the device can immediately deliver the payload to its final destination.

6.7 Conclusion

The above analysis show that device virtualization and operation issuing are an important point for a low-latency device. The *CFCQ* design is the most promising, implementable derivative of the true *Transactional I/O* method. Both communication functions of EXTOLL use such a queue design for commands. In the case of VELO (section 7.4), the queue is used as a combined command and data queue whereas in the case of RMA (section 7.5) the CFCQ is used as a command queue only. VELO employs state minimization and elimination: the transmit function of VELO is stateless while the RX function has to employ minimized state information per context. These are receiving queue's base address and read/write-pointers. Something similar holds true for the RMA unit architecture; all three sub-units employ extensive state minimization and only completion state has to be held per context. Both of these sub-units could benefit from true *Transactional I/O*, removing some more software overhead and the deadlock problem. Both functional units are primarily designed for polling completion on single queues but optionally also support completion notification through interrupts. The complete design of *Transactional I/O* or CFCQ can also be leveraged for other I/O devices; specifically accelerator/coprocessors applications are promising.

The EXTOLL Hardware

Chapter 7

In this chapter the hardware of EXTOLL is described. As mentioned EXTOLL consists of a number of building-block components. It is possible to adapt the architecture depending on the requirements for a given system environment. The prototype implementation covered in this thesis chooses components and a configuration which optimizes communication latency especially for small communication operations in AMD64 host systems while it is at the same time able to support a general set of communication operations and fits into an FPGA based hardware platform.

The platform EXTOLL is mapped to, the CAG HTX-Board, has been described in [56]. One goal of this development was to provide a platform for NIC prototyping [118]. In short, the main features are an HTX compliant HyperTransport interface, 6 SFP optical transceivers for the network and a Virtex-4 FX platform FPGA as central, reconfigurable logic resource.

The general EXTOLL block diagram is given in figure 7-1. The complete architecture is formed through a number of major blocks, namely the host interface, the on-chip network, the functional units, one or more Networkports, the EXTOLL crossbar and a number of Linkports. The host interface is implemented by the HT-Core [14], which was also developed with EXTOLL in mind and which was optimized for low-latency operation. Another possible option for the host interface, which was developed, is a the combination of a PCIe [119] core and the CAG PCIe-to-HTAX bridge IP [120]. Because of the interface characteristics, system topology and FPGA based SERDES implementation this option would lead to a significantly lower hardware performance on an FPGA prototype though. The on-chip interconnection network which both connects the host-interface with the various functional units and the functional units with each other is another IP block developed at the CAG named HyperTransport Advanced Crossbar (HTAX). There have a number of functional units been developed for the EXTOLL architecture including the ATU which was analyzed in detail in chapter 5, several communication units (the requirements for them were analyzed in chapter 4) and the status and control register file unit.

The network part of EXTOLL is formed by the Networkports, Linkports and the EXTOLL crossbar. A short overview is given below, but since the network layer is not topic of this thesis, the reader is kindly referred to [14][15][16][17][18][19] for more information about the network layer.

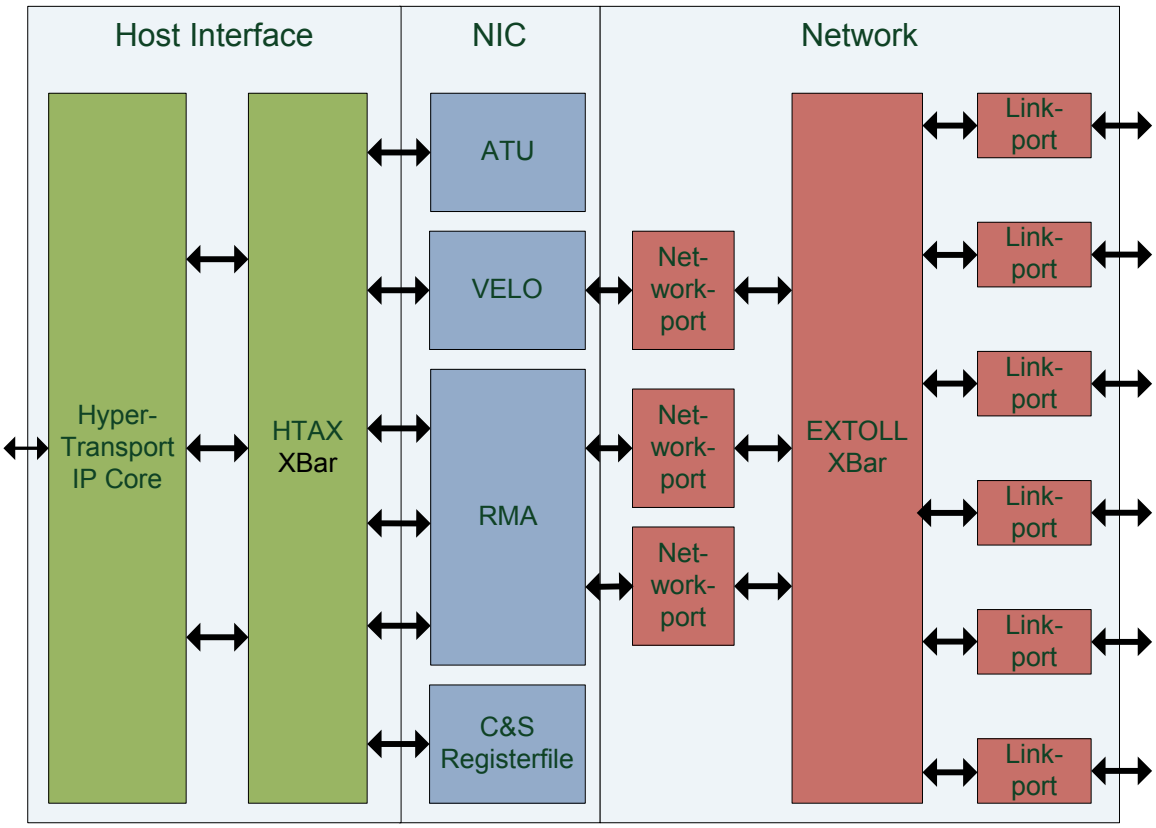


Figure 7-1: EXTOLL Block Diagram

In the next three sections, the host interface and HTAX, registerfile and the network layer components will be shortly introduced. The rest of this chapter is devoted to the architecture of the EXTOLL communication units.

7.1 HT-Core and HTAX

The HT-Core forms the preferred host interface for EXTOLL, enabling low latency and high-bandwidth. Latency of the core itself is as low as 12 respectively 6 pipeline stages from outside the chip to the application layer FIFO interfaces. With a peak bandwidth of 1.4 GB/s on a FPGA and one-way latencies below 100 ns the core forms an ideal building block for EXTOLL. To connect more than one functional unit to the host interface, the HTAX [15] forms a dynamic network-on-a-chip for EXTOLL. HTAX not only connects the functional units with the HT-Core and vice-versa, it also connects functional units with each other, for example for the ATS protocol within EXTOLL. HTAX is a completely new design which features among others virtual channels, overlapping arbitration and split-phase transaction. An adapter unit, the HTAX-to-HT bridge connects the HTAX to the HT-Core. This component also provides for source tag translation. This feature enables

EXTOLL components to virtually use the full source tag space of 5-bit for split phase transactions. The actual sharing of the source tags of all of the units onto the source tags available in the HT fabric of the system is transparently handled by the bridge. The introduction of the bridge as one component also allows the exchange of the HT-Core with another host interface IP while keeping all of the remaining architecture completely unchanged (an example being the HTAX-to-PICe bridge [120]).

The latency for the HTAX connection is 6 respectively 3 cycles for the incoming or outgoing path. The HTAX and bridge both live in the EXTOLL core clock domain, whereas the HT-Core uses its own set of clock domains. The application interface FIFOs of the core form the clock domain crossing, decoupling the EXTOLL clock domain from the host interface.

In summary, the components form a capable and very high-performance path to the host for EXTOLL. In the FPGA prototype a 16-bit, HT400 implementation is used, for an ASIC analysis suggests that the HT-Core can operate a 16-bit HT1000 link, increasing the peak bandwidth to nearly 4 GB/s per direction and also reducing latency significantly.

7.2 Registerfile

The EXTOLL register file serves as a central component for control and status functions. A central versus a distributed design was chosen to significantly ease the implementation. Within the agile development style of EXTOLL, a nearly completely automated design flow for the control and status registers proved to be very useful.

The register file design relies on the System Register Description Language (RDL) [121] and the Denali Blueprint [122] RDL compiler. In an RDL, registers and sets of registers can be easily and efficiently modeled at a much higher abstraction layer than for example an HDL language allows. The Blueprint compiler analyses RDL sources and generates an internal representation. Using different output generators, the original RDL can be translated to representations in other languages. For the EXTOLL registerfile, 5 different output generators are used concurrently: the Verilog generator creates RTL-level HDL code used for the hardware implementations, the HTML output generator creates HTML reference documentation of the registers and SGML output for usage in FrameMaker. The FrameMaker output is used for the EXTOLL reference manual [123]. The XML generator is used to build a representation of the register file used in a later step. Finally, the ANSI-C module generates header files modeling the complete registerfile using C-datatypes. Additionally, a large set of macros is generated to enable access to individual fields of registers.

This whole process reduces changes to the register space to a simple RDL patch followed by an automated process. The integrated documentation facilities are especially useful, since this ensures that documentation is up to date. RDL and blueprint are also useful, because they are highly flexible and a number of special features is used by the EXTOLL register file which would have been needed to be implemented by hand otherwise. To name just a few:

- defining and implementation of Interrupt, Interrupt Mask and Interrupt Clear registers including the needed inter-register semantics
- definition of arrays of registers as RAM blocks for the implementation
- definition of counting registers, thus making it very simple to introduce performance- or debug-counters into a functional unit. The unit only has to assert one signal to increment the counter; the rest is handled by the registerfile logic

To connect the generic bus interface of the generated hardware implementation to the HTAX protocol, a wrapper module was developed. A TCL script was written to automatically instantiate the register file, the wrapper and add RAM blocks as needed. The wrapper module also handles the translation of wire based interrupts from the registerfile to MSI style messages required for HT.

To further automate the process and reduce possible sources of errors, a tool was developed which uses the XML output of Blueprint together with the ANSI-C macros to generate a complete Linux kernel driver for such a generated registerfile. The kernel driver implements complete access to all registers using the /sys pseudo-filesystem [124] and provides for a comfortable API to access the register from other modules within the kernel.

As a last component, the source revision the register file[1] was built from is automatically set in the version register at offset 0 of the register file, making it possible for system software to check if the hardware and software revisions match. Again, this process is fully automated.

The complete flow of the generation of the registerfile and all dependant modules is shown in figure 7-2. This automatic flow has proven to be a very efficient way to handle control, status, debug, and performance analysis registers in a design which is actively developed.

7.3 EXTOLL Network Layer

The Networkports, Linkports and the EXTOLL crossbar constitute together the network block or network layer of EXTOLL[2]. The EXTOLL network protocol used by these components is built from 16+2-bit sized phits (network characters). Up to 64 phits form one *flow control unit*, flit. The flit is the unit of buffer sizes in the EXTOLL network and thus one credit of the flow control protocol means buffer place for one flit. Above the flit protocol, messages, more accurately packets, are assembled from up to three *frames*, the routing frame containing the packet routing information, the command frame for packet header information (communication function specific) and the payload frame. Each flit is protected by a CRC.

1. The EXTOLL project used the Subversion[125] version control system to manage the source tree of the project.
2. Additionally the SERDES/MGT block implementing the physical layer are also logically part of the EXTOLL network block.

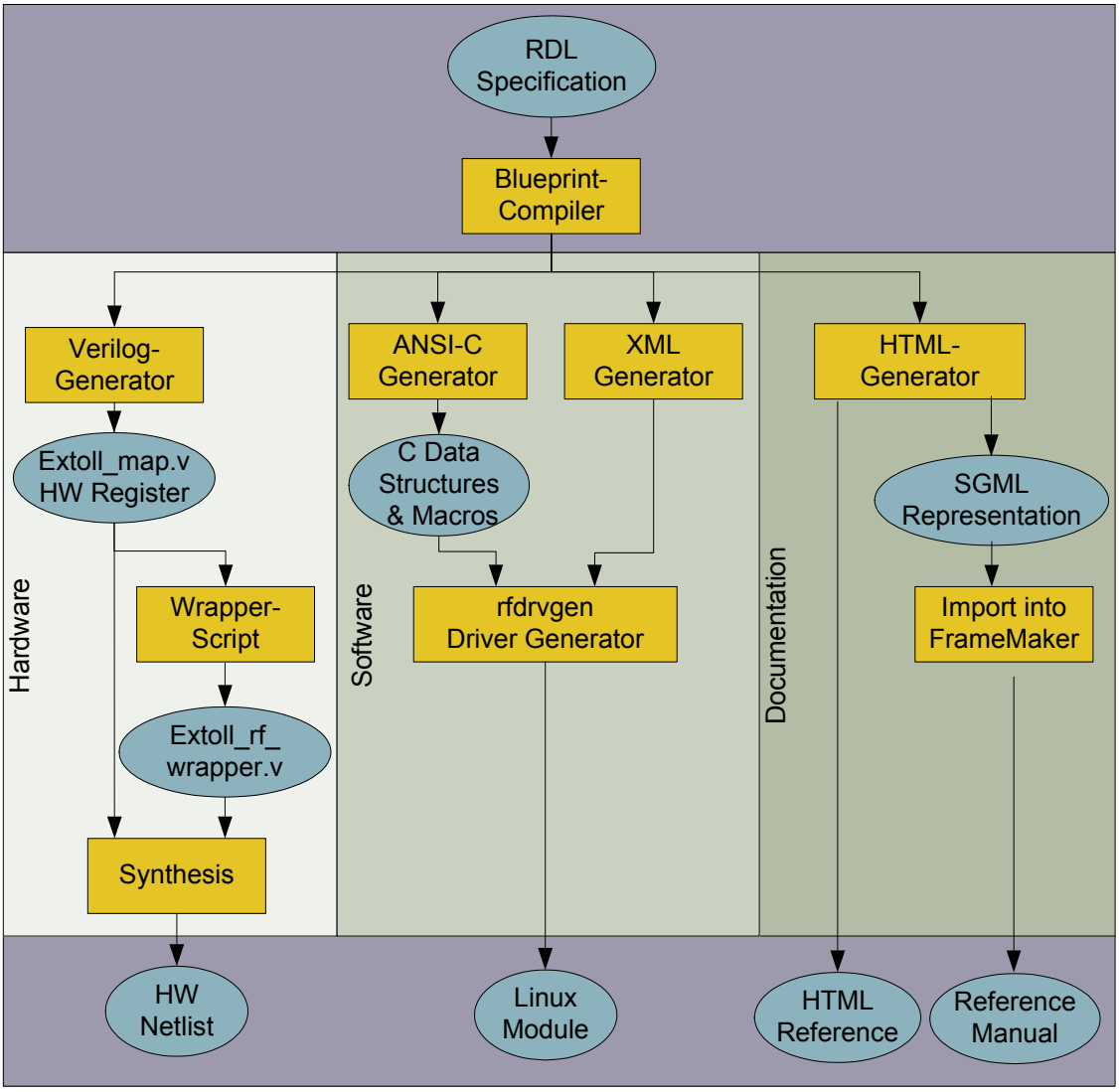


Figure 7-2: Registerfile Design Flow

The Networkport converts packets to the network layer format and also handles virtual lanes for functional units. Another notable attribute is that the Networkport serializes the 64-bit wide data format of the functional units into the 16+2 bit wider format of the EXTOLL network and back again. The Linkports convert packets to the link layer format and implement a hardware retransmission protocol to ensure reliable network operation. The EXTOLL crossbar finally implements the switching and routing resources of the EXTOLL network.

Switching in EXTOLL is cut-through and routing is based on a modified source-path/delta routing. Each routing character in the routing string determines the direction a packet should take at the next switching stage. It also determines one of two virtual channels to enable deadlock free routing in a torus topology. Finally, to optimize routing string size, a run-length encoding of the routing strings is employed. That is, if a packet has a number of sequential switches to the same output port, this is encoded in one routing character using a 6-bit counter. At each crossbar, the counter is decremented by one and only if the counter reaches zero, the routing string is consumed as in classic source-path routing. To further optimize this scheme, counters for both virtual channels are coded within one character. With this method, dimension order routing for a 3-D torus with $2^6 \cdot 2^6 \cdot 2^6 = 262.144$ nodes can be encoded in one 64-bit word.

An important trait of the network layer which will appear again in the evaluation part of this thesis is the performance characteristic. The network protocol is a tagged 16-bit wide format enabling a raw bandwidth of twice the network clock frequency in bytes. The exact latency characteristics will be shown in the evaluation of the architecture.

As an example, simulation shows that to transport a 56-byte payload VELO message, 41 cycles are needed by the link. The number of cycles would be enough to transport 82 bytes. Thus, at 56-byte package size, the EXTOLL protocol operates at ~68 % link efficiency. The link efficiency grows up to about 80 % for large RMA messages. Several proposals exist to improve the efficiency, mainly increasing the flit size and or widening the data path. This may be required to reach high bandwidth, for example to use 4x serial links. Both, larger flits and wider data paths, have not been implemented because of FPGA resource limitations. See also [126] for an in depth discussion of the link protocol and efficiency effects.

7.4 EXTOLL VELO Engine

VELO is a communication engine for small, two-sided messages. It is designed around the ideas of a CFCQ to post messages. Messages are received into one DMA ring buffer per VPID. An earlier version of the unit, missing some features, was described in [127]. Since then, support for asynchronous receive mode (i.e. interrupts), message type tags (MTT) and variable size DMA buffers has been introduced.

Figure 7-3 shows the sequence of a typical VELO message transfer. The sender copies the data from the original buffer to a *requester address*. The requester address decodes to the VELO requester unit. At the same time the address carries several pieces of information to the requester unit: destination node, destination VPID, source VPID, length of the message and a 2-bit message type tag (MTT). The first four pieces are all encoded in address bits beyond the 12th, thus enabling system software to control whether a process can communicate with another or not by mapping or not mapping the respective physical page into the virtual address space of the process. The address bits 6 to 11 are used to encode the message lengths and the MTT. The MTT can be used to differentiate software dependent four types of messages, for example user messages, flow-control messages, synchronization messages and ULP control messages. The length of a VELO message can be in between 8 and 64

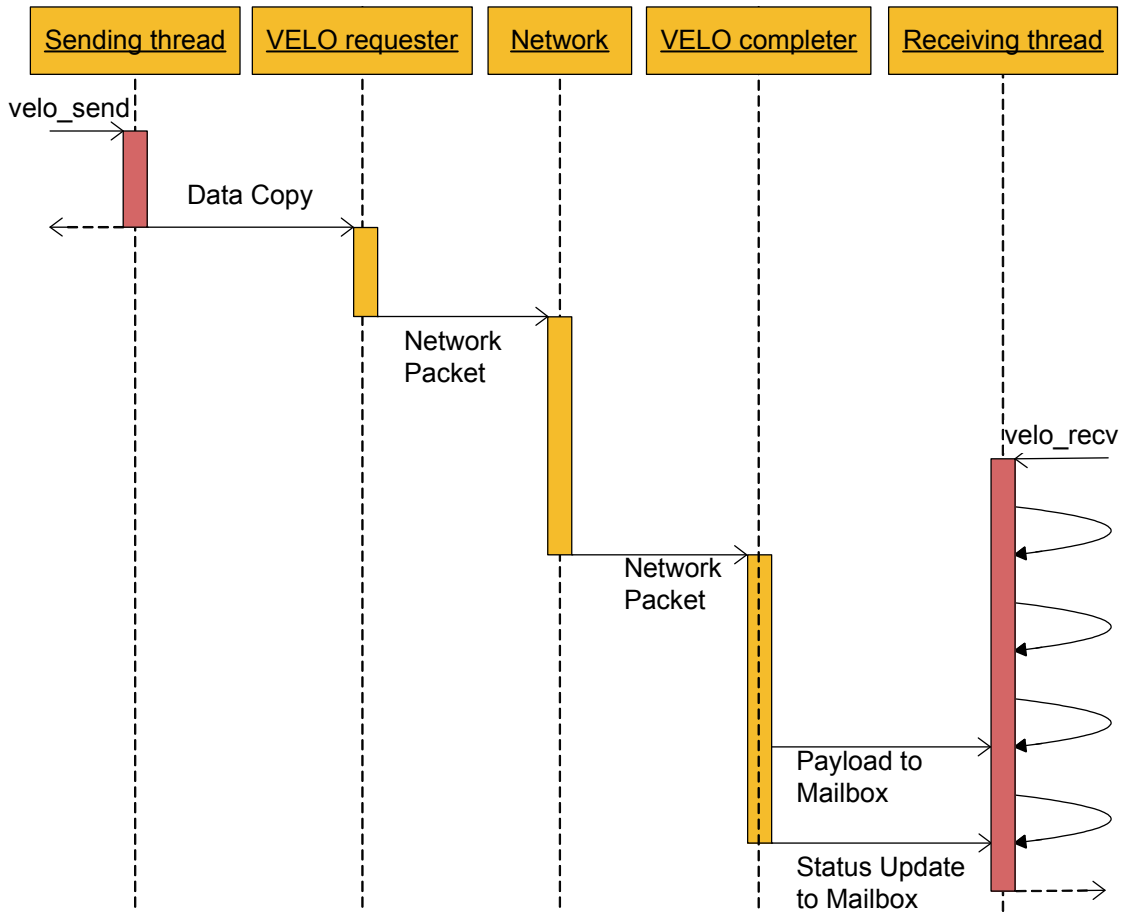


Figure 7-3: VELO Sequence

bytes (in steps of 8 bytes). The specification of the message length in the start address, together with the fact that the length of a HT packet is given in its header allows the VELO requester to differentiate if a message has been split or not. Theoretically, messages should be posted atomically. As discussed in section 6.4.4, a write-combining access may be interrupted though. VELO generates a different header for the parts of a split message to allow software on the receiver to reassemble the message again. The address layout for VELO as implemented by the EXTOLL prototype is given in figure 7-4.

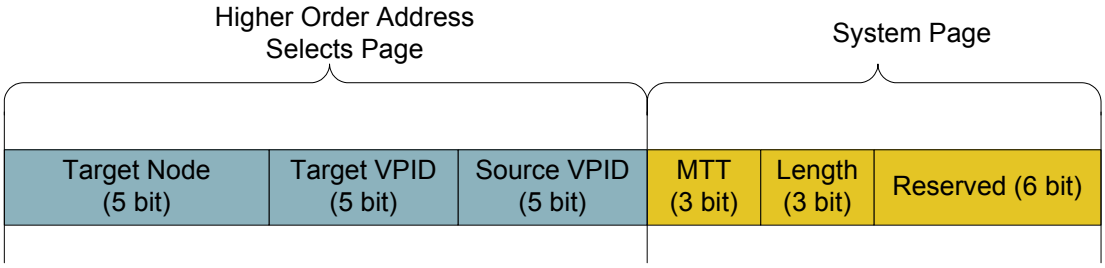


Figure 7-4: VELO Requester Address Layout

On the receiving side, the DMA ring buffer for one VPID is divided into slots of one cache-line (64 bytes). When receiving a message the completer writes an 8 byte status word including source node, source VPID, message length, length of the current fragment and MTT to the first quadword of a slot. The rest of the slot is filled up with data until the length of the message. A slot is always written using one HT packet, writing of one slot is thus atomic. The CPU can poll on the status word (which is initialized with all zeroes). When the status word content changes, a new message has arrived. Note that the status word for a received message is always unequal to zero. Software copies the payload of the data to the final buffer. The DMA ring-buffer space is freed by writing to the associated write-pointer register. Generally, software implements a lazy pointer scheme to statistically lower the impact of the register access on message latency.

A special case arises for message of 64 byte length. They do not fit in one slot and are thus divided up into two slots by the Completer. The first slot is filled with a status word and the first 56 bytes of the message. The second slot gets a second status word and the remaining bytes. Software is required to poll again on the second status word to avoid any race conditions. The whole memory layout of the completer side is depicted in figure 7-5.

VELO Implementation Constraints

The actual implementation of VELO puts a number of constraints on the design which will be briefly explained here. The number of destination nodes, destination VPIDs and source VPIDs is limited by the amount of physical address space available. The requester address space is mapped into its own BAR in the EXTOLL prototype and was decided to be limited < 256 MB to avoid problems with buggy PC firmware (BIOS) implementations which will not accept larger bars. Furthermore, the number of VPIDS and nodes supported determines the number of entries needed in the diverse registers. For the prototype 32 VPIDs and 64 nodes were deemed large enough. These constraints are already present in the following register space description.

VELO Registers

While VELO is designed around the idea of state minimization, a number of status and control registers are necessary for VELO. On the requester side, this is mainly the routing look-up table used to generate the correct EXTOLL routing string for a message. On the completer side, the different DMA regions for the VPIDs have to be managed. Additionally, interrupt control and counter registers are available. All of VELO's registers are summarized in figure 7-1, a more detailed reference can be found in [123]. All of the registers are part of the EXTOLL registerfile and are thus created from an RDL specification.

VELO Asynchronous Mode

The interrupt control register enables configuration of interrupt generation per VPID. This feature enables asynchronous notification of the CPU of incoming messages. The use-case for this feature is mainly for in-kernel communication protocols and management, or synchronization protocols in passive target applications.

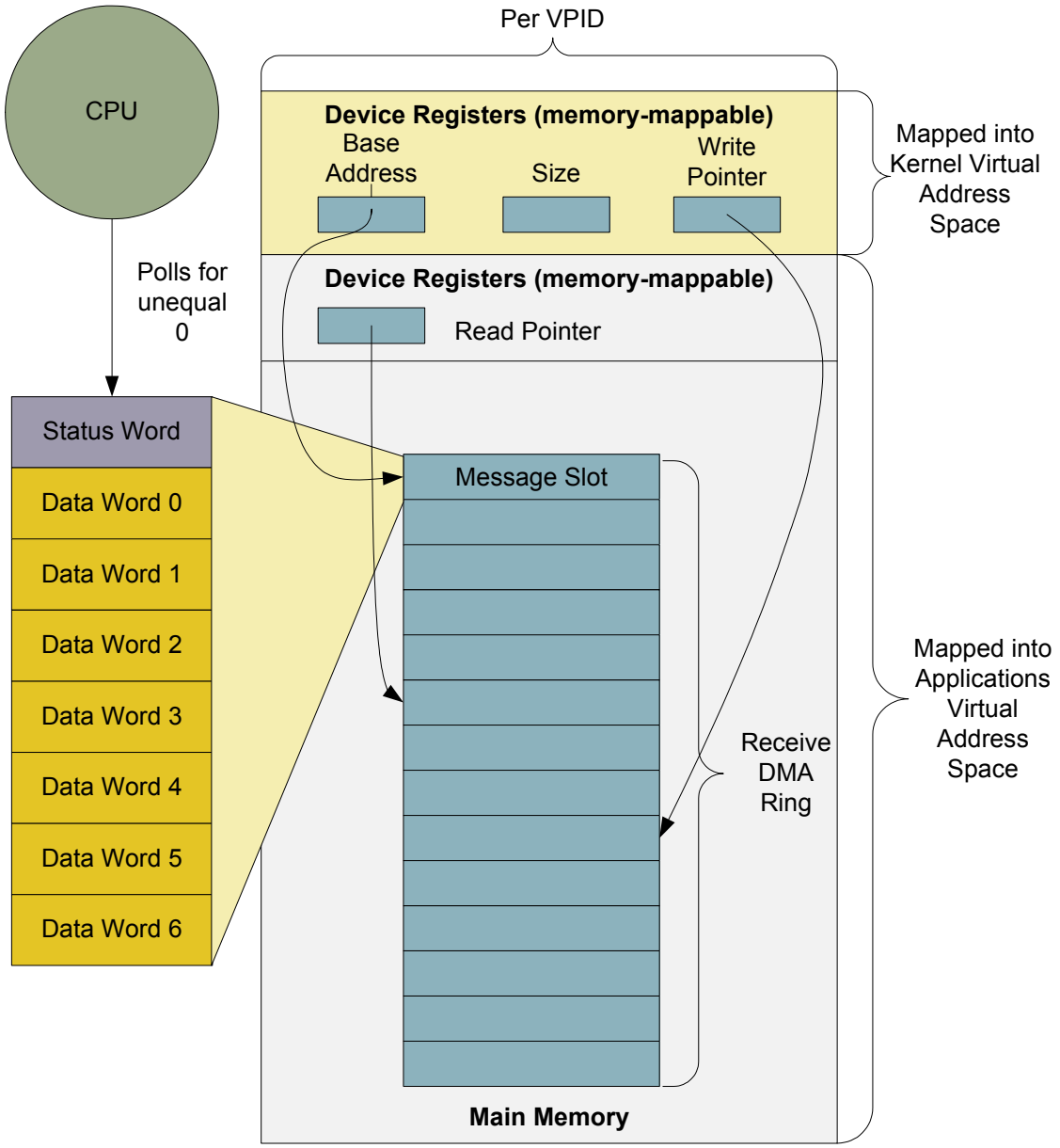


Figure 7-5: VELO Completer Memory Layout

Of course, using interrupts introduces a relatively high latency penalty, especially when compared to the extremely low-latency or normal VELO operations. Nevertheless, this constitutes a faster way to send a message asynchronously between two nodes than any other.

VELO Microarchitecture

VELO is implemented in two components, the requester and the completer unit. Each of them connects to the HTAX, albeit only unidirectional, and they actually share one bidirectional port of the HTAX crossbar. The same holds true for the Networkport interface on the other side of the units.

Name	Size	Description
Routing Lookup Table (RLT)	64 x 8 byte	Routing strings for VELO Destinations
MBox Enables (general register)	8 Byte	Holds an enable bit for each mailbox
VELO Counter Register	8 Byte	General message counters (sent, received, discarded)
Mailbox Size Lookup Table (MSLT)	32 x 8 byte	Holds the size of each mailbox, thermocode
Base Address Lookup Table (BALT)	32 x 8 byte	Physical base address of each mailbox
Write-pointer Lookup Table (WPLT)	32 x 8 byte	Write-pointer of each mailbox, updated by HW
Read-pointer Lookup Table (RPLT)	32 x 8 byte	Read-pointer of each mailbox, updated by SW. Each pointer resides on its own page in the address space to enable user-space mapping

Tabelle 7-1: VELO Registers

Within each unit, an FSM (designed using the FSMDesigner tool [112]) forms the control path, while the data-path is written in Verilog HDL. The HDL code involves about 440 lines of code for the completer and 310 lines of code for the requester not counting the FSMDesigner generated code. All of the registers and lookup tables reside in the registerfile and are thus specified in the RDL code of the register file.

7.5 EXTOLL RMA Engine

The EXTOLL RMA unit provides for *put*, *get* and *lock* network transactions. This hardware unit is designed to support medium to large data transfers with efficient CPU offloading by using virtualized user-level access, the CFCQ approach, and DMA engines. Address translation can be provided for using ATU or using the kernel. In general the ATU solution is preferred.

The RMA instruction set has been designed to be highly orthogonal enabling efficient mapping of different upper-level-protocols on top of the basic RMA service. For example the choice of notification events at remote and local end-point of a network transaction is useful to implement different rendezvous or send/receive protocols. The *lock* transaction as well as the notifications can be used to implement overlapping, extremely efficient one-sided operations for example for MPI-2.

The completion of transactions or part of transactions is flagged to software using notifications. A notification is a 128-bit value which is written into a notification queue. Each VPID has one notification queue, which is managed from the register file using read-, write-pointer and a base address. The queue is implemented as a ring-buffer. As an optimization, each read-pointer resides alone in one page allowing user-level mapping of the pointer and read-pointer updates directly from user space (like VELO). There are three classes of notifications, requester-, completer- and responder notifications named after the unit that generates them. Software can poll on a notification to detect that a transaction was completed meaning for example that a buffer can be reused, that data is valid or that a lock is now held.

7.5.1 RMA Instructions

The different instruction for the RMA unit can be categorized into four groups: *put*, *get*, *lock* and special *put* operations. *Put* and *get* operations move data to or from a remote node to the other partner. *Lock* transactions use the novel remote *fetch-compare-and-add* (FCAA) method described in section 7.5.4. The special *put* operations are immediate and notification *put*. An immediate *put* carries the payload already in the descriptor of the command. The payload is limited to 64-bit in this case, but a fast path for remote single-word-stores is provided for. The notification *put* operation only generates a notification at the remote end with a part of the remote notification descriptor being set again by the payload of the command descriptor. This command is especially useful for synchronization and management purposes. For example it allows exchanging data even if no remote address is known (yet).

Finally, *put* and *get* transactions, the real workhorses of the RMA unit are provided in two size flavors: quadword and cacheline sized. The operand size command bit decides whether the size field of the requester descriptor is to be interpreted as the number of quadwords or cachelines to be transported. This allows for a minimum transfer size of 64 bit up to 4 kB using one RMA request. Not all transfer sizes are possible, for example transfers larger than 64 byte can only occur in multiples of 64 bytes. Also, no transfer is allowed to cross a page border (4 kB). If misaligned data or data of not matching length has to be transferred, the complete transfer has to be built using a number of transfers of different sizes; the same holds true for buffers crossing page boundaries. The page boundary rule makes the address management much more efficient in hardware.

Requests are posted by writing a 128-bit descriptor to a special address within the requester memory-mapped I/O space. The address encoding is shown in figure 7-6. The descriptor format (in the most common case) is shown in figure 7-7. These 128 bits have to be written to the requester atomically using an SSE2 operation as described in section 6.4.3.

For each instruction four command modifier bits can be set: requester notification, completer notification, responder notification and operand size qualifier. As described above, the operand size qualifier sets the transfer size to be either cacheline or quadword size based. This bit is ignored for special *put* transactions and remote locks. The notification bits configure at which point a notification is written back to the respective node's main memory. All of the combinations of the notification bits and the RMA instructions are summarized in table 7-2; obviously some combinations are not supported, hardware has been designed to ignore the bit in this case.

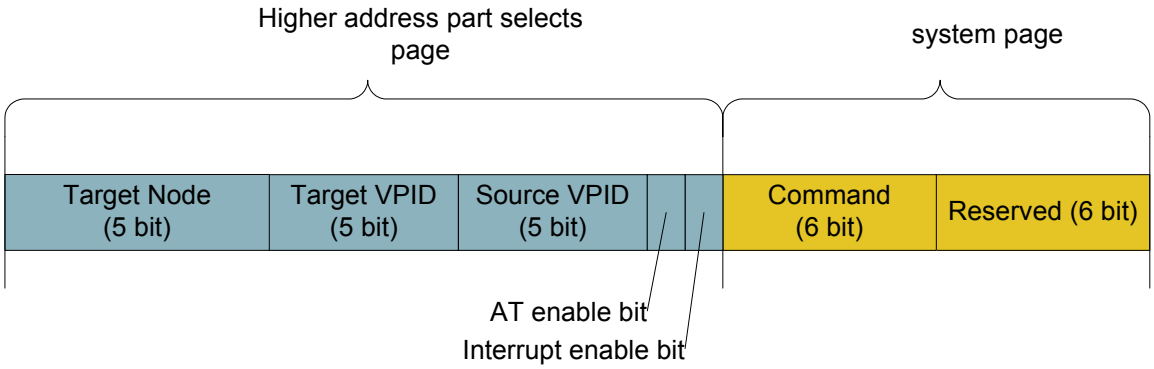


Figure 7-6: RMA Requester Address Encoding

An instruction or command is transformed to a network descriptor by an RMA unit and then forms the command frame of an EXTOLL packet. Finally, a command frame is transformed to a notification entry at a receiving RMA unit if the corresponding notification bit is set. The process is illustrated in figures 7-7 and 7-8 which show the descriptors of an RMA *put* command. The network and the notification descriptor contain the same fields and are thus shown in one illustration. Note that a *put* command can generate both a requester and a completer notification, which differ only in the notification bit that is set and the exchange of source and destination in all fields.

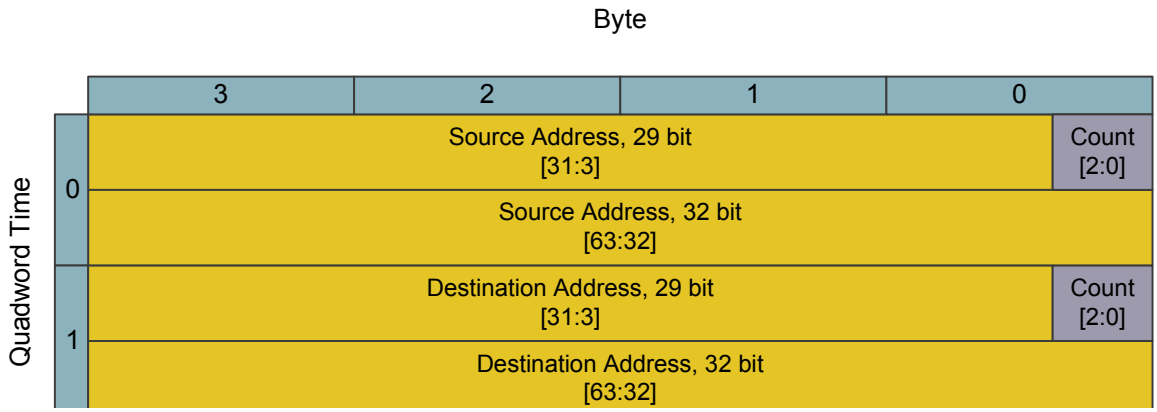


Figure 7-7: RMA Requester Command Descriptor (*Put*/*Get*) [128]

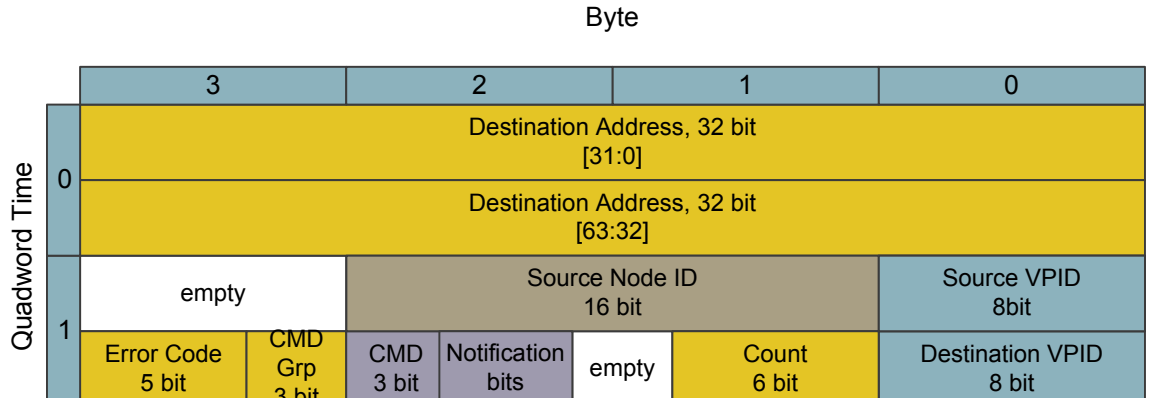


Figure 7-8: RMA *Put* Requester Notification Descriptor [128]

Finally, depending on the interrupt bit, notifications from a command can trigger an interrupt. The interrupt bit is located beyond the page border enabling supervisor software to control the usage of interrupts for user-space software by mapping the corresponding pages. Interrupt driven execution is expected to be of use especially for enterprise applications and kernel level applications like storage or sockets emulation.

7.5.2 PUT Instructions

The *put* operation is the equivalent to a remote write. The payload size ca either be given in quadwords or cachelines (64 byte). A maximum of 64 units can be transported. Thus, up to 4 kB can be *put* in one command. For the *put* command at least three combinations of the notification modifier bits are useful, for details see table 7-2.

7.5.3 GET Instructions

The *get* command represents a remote read transaction. Payload sizes are the same as for the *put* command. Again, the instruction table lists the useful notification modifier combinations (table 7-2).

7.5.4 RMA Remote Lock Instruction

The remote *lock* command requires a more comprehensive description and analysis. The goal of the remote *lock* (sometimes called atomic operation) is to enable efficient mapping of software algorithms using memory region locking as in MPI-2. A fully satisfactory solution was not found in the literature (see also section 4.4.3). In this section, a novel remote *lock* operation and the operation's semantics are introduced first, followed by the description how the remote *lock* operation is efficiently implemented for RMA and EXTOLL.

Basic command	Requester Notification	Completer Notification	Responder Notification	Description
Put	0	0	0	No notification, intermediate transfers
	1	0	0	standard one-sided *put*
	0	1	0	n.a.
	1	1	0	*put* for two-sided emulation
	X	X	1	not possible
Get	0	0	0	n o notification, intermediate *get*
	1	0	0	n.a.
	0	1	0	standard one-sided *get*
	1	1	0	n.a
	0	0	1	n.a
	0	1	1	for two-sided, rendezvous protocol
	1	0	1	n.a.
	1	1	1	n.a
Lock	0	0	0	n.a
	1	0	0	n.a
	0	1	0	normal *lock* operation
	1	1	0	n.a
	0	0	1	n.a
	0	1	1	*lock* operation, remote side informed
	1	0	1	n.a
				n.a.
Immediate Put	X	1	X	used to *put* one quad-word
Notification Put	X	1	X	synchronization, management
1 - Bit is set in command, 0 - Bit is not set in command, X -value is ignored				

Tabelle 7-2: RMA Instruction Set

The FCAA operation

We first define a compound atomic operation that is called fetch-compare-and-add (FCAA) which is complex in comparison to atomic operations available in modern CPUs and may be comparable in complexity with the CAS2 instruction introduced with the Motorola 68020. The FCAA operation has three source operands: the destination address (a memory location), a compare value and a 2-complement value designating the value to be added. It returns two values: the new value of the memory location and a success flag. In pseudo-code the operation can be described as follows:

```
FCAA (destination, compare, add):
if (*destination <= compare)
      *destination += add;
      return (*destination, true);
else
      return (*destination, false);
```

At the completer side the operation proceeds as follows:

1. The value is fetched from its memory location.
2. The value is compared with the compare operand. If the value is smaller or equal, the operation proceeds with the next step, otherwise the current value is returned together with the failed flag.
3. The third operand is added to the value. Since this is a 2-complement, it is also allowed to add a negative value and so actually subtract a value.
4. The new value is written back, returned to the initiator together with the flag set to success.

This operation can be used to directly implement a large number of atomic transactions used in parallel distributed applications. Three examples are given below. The first is a general fetch-and-add (FAA) operation, the second is an implementation for the MPI-2 passive target synchronization with shared and exclusive locking. The third example shows how the MPI-2 active target *MPI_start/post/complete/wait* semantic can be implemented using FCAA.

FCAA Examples

To implement the basic FAA operation, the compare value is set to the maximal value that can be held within the value operand. The operation will always succeed, and the addition will proceed atomically.

MPI_lock and *MPI_unlock* can also be implemented using FCAA. The specialty of these kinds of locks is, that they can be in one of three states: *unlocked*, *exclusive* and *shared*. The rational is, that it is possible to lock a memory window for multiple (remote) readers (shared state), or to lock it exclusively for one writer. Table 7-3 summarizes the operand values for the different cases. The actual lock variable has the encoding specified in table 7-4.

Operation	Compare value	Add value
Lock Exclusive	0	n+1
Lock Shared	n	1
Unlock Exclusive	MAX_INT	-(n+1)
Unlock Shared	MAX_INT	-1
n - *the number of processes in the MPI Job* MAX_INT - *maximum value that fits in the operand*		

Tabelle 7-3: FCAA: MPI_Lock/MPI_Unlock Operands

State	Lock Variable Encoding
Locked Exclusive	n+1
Lock Shared	m, m is between 0 and n
Unlock Exclusive	0
Unlock Shared	0

Tabelle 7-4: FCAA: MPI_Lock/Unlock Encoding

MPI_start/post/complete/wait is an active target synchronization method for MPI-2 one-sided communication. Since it is active target, it can also relatively efficiently be implemented using two-sided communication. Here, a method to implement the operation using FCAA is proposed. Table 7-5 shows the operand values and table 7-6 the lock variable encoding. The FCAA semantics enforce that the target only advances if its *post* operation was successful and the originating process only if its *start* operation was successful. So, the *MPI_post/start* semantics as required by the standard (see also section 4.4.3) are fulfilled. The same holds true for the *MPI_complete* and *MPI_wait* pair. Successive *MPI_post/start* operations only succeed once an *MPI_complete/wait* operations have been performed.

Operation	Compare Value	Add Value	Local/Remote
Post	3	-1	local
Start	2	-1	remote
Complete	1	-1	remote
Wait	0	3	local

Tabelle 7-5: FCAA: *MPI_Start/Post/Complete/Wait* Operands

Implementation Requirements

For the hardware implementation in a NIC the following requirements must be met:

state	lock variable encoding
Idle	3
Posted	2
Started	1
Completed	0

Tabelle 7-6: FCAA: *MPI_Start/Post/Complete/Wait Encoding*

- The NIC must no interrupt processing of a single operation (atomicity of the completer).
- The NIC may choose to queue up a request and answer at a later point in time (allowing for an optimization by postponing the completion of a failing request, until it succeeds).
- The NIC may choose to immediately answer a request.
- The local process must also use the same NIC function to manipulate a lock variable.

The FCAA network operations employ a non-blocking split-phase protocol. With above requirements it is clear, that a request may fail or be delayed until it finally succeeds. Thus, software must be prepared to retry a failing request while the hardware only guarantees a best effort to not send failing answers. This mechanism is expected to reduce network load due to lock polling considerably and to improve locking performance. Also, the need to employ an exponential back-off algorithm for lock polling is mostly eliminated.

Remote Locking Hardware Architecture

The remote *lock* operation can be elegantly integrated into the RMA engine reusing hardware components. Other possibilities considered were to add FCAA to VELO or implement a new functional unit. A new functional unit would add a complete new bidirectional HTAX port, a bidirectional Networkport and an EXTOLL crossbar port, which was rejected because of resources. VELO does not include the necessary interface to the host, since DMA read transactions are not provided for. But RMA includes already all the basic operations needed.

Using RMA, kick-off is always done at the requester. The format of the associated RMA command descriptor is shown in figure 7-9. Operand sizes for the FCAA operation have been chosen to be 32-bits. The command and notification bits necessary to compute the address where to store the command have already been given in table 7-2.

Incoming atomic requests are destined to the RMA responder unit, which usually handles incoming *get* requests. The RMA responder gets the associated value from main memory and performs all necessary computations. It then generates an answer message to the completer of the initiating process which finally writes a notification with the result of the FCAA request (figure 7-10).

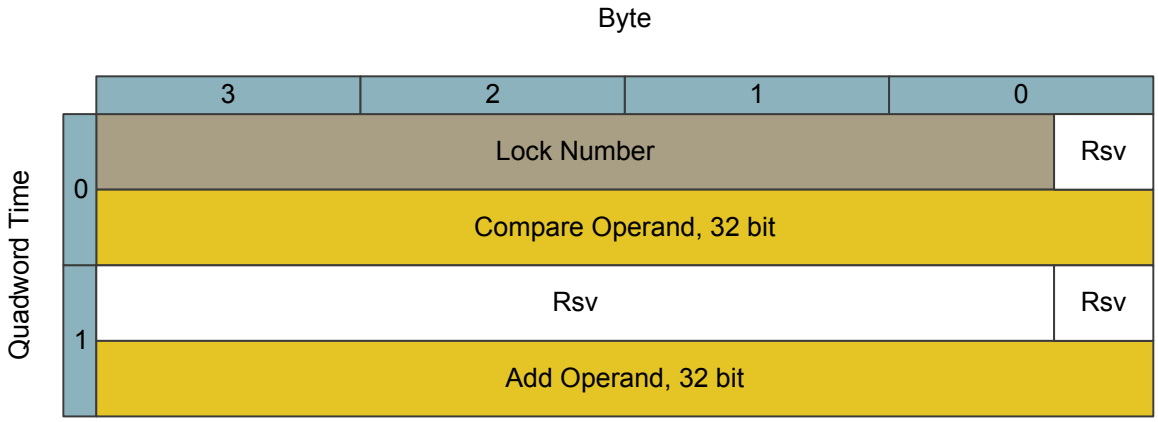


Figure 7-9: RMA *Lock* Command Descriptor

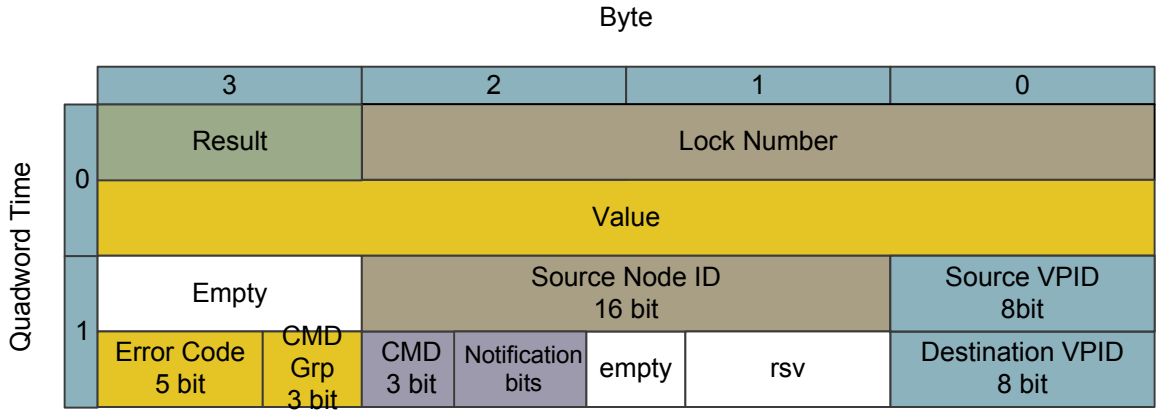


Figure 7-10: RMA *Lock* Notification Descriptor

For processes wanting to lock their local window, two possible solutions exists: either message are routed through the local crossbar, thus they appear as remote requests at the responder, or the necessary amount of special purpose communication between the units is introduced. The preferred solution is the first method using internal crossbar routing because of symmetry, simplicity and resource usage.

The lock variables can be organized in two ways. Either they are variables addressed using NLAs as used by *put/get* operations, or they reside in their own *lock address space*. The first version is more flexible and general but needs an address translation to complete a FCAA operation; the other option is somewhat less flexible but does not need an address completion, instead a base address register addresses a physically contiguous memory region. Within the region a page is selected by the VPID, the page offset by the lock address. A lock variable is a 32-bit double word. So, each VPID has 1024 locks available. The memory and

address scheme is depicted in figure 7-11. Lock 2 of VPID 1 is shown shaded darker. The base address of the lock region is stored in a register. The VPID forms the high-order bits of the lock offset and the lock number the low order bits of the lock offset. Together with the base address the lock offset directly forms the lock address. The physically contiguous memory region necessary is in size 128 kB for the EXTOLL prototype; an implementation with 512 VPIDs needs a 2 MB region.

A future extension refers to the optimization of failing lock requests. For this purpose requests that failed the compare stage of the process can be stored in an associative data structure at the completer. If a new operation is processed at the completer, the next operation stored in the failed operation buffer which targets the same lock must be re-evaluated. It is expected that good results can be obtained with a low number of outstanding operations Larger buffers are possible within the specification and architecture if resources allow for it (ASIC implementation). The simplest solution for an FPGA implementation seems to be a linear search of outstanding FCAA operations if the queue is not empty after the completion of an operation. This can simply be implemented in the corresponding responder FSM and requires a small FIFO and a number of cycles proportional to the number of outstanding operations. This optimization was not included in the prototype implementation.

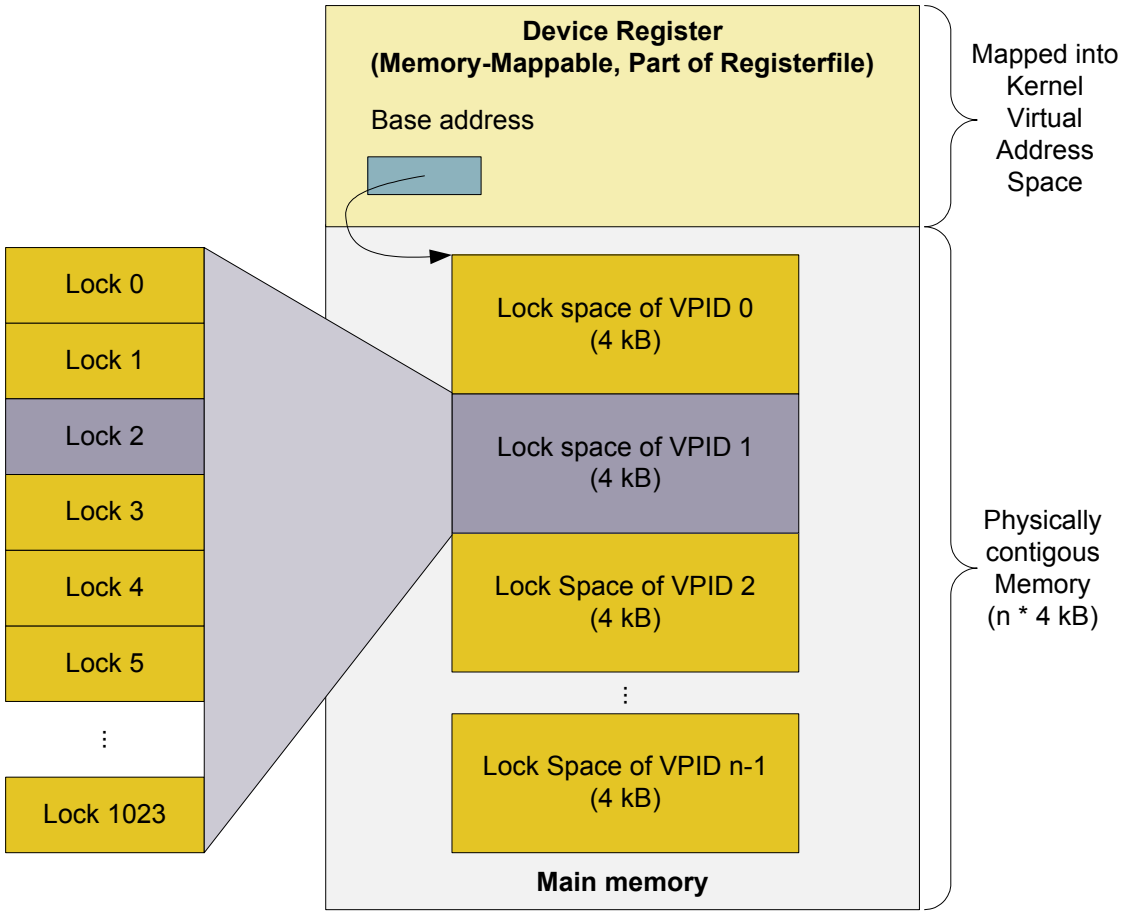


Figure 7-11: RMA *Lock* Addressing

7.5.5 Physical vs. Virtual Addressing

Devices can usually only access physical memory. To enable true user-level communication, ATU was introduced to the architecture. Furthermore, we can also use kernel-level services and communication using physical addresses. In this model, each communication request is passed through the kernel on the requester side. The driver translates all addresses; for this purpose the driver also manages a table of all remotely available windows. The descriptor is then passed as quickly as possible to the underlying hardware. All completions are also preprocessed by the kernel, again translating addresses this time from physical back to virtual addresses before they are delivered to the user space application. This scheme saves the address translation capabilities in the hardware at the expense of a much higher complexity at the kernel level and increased latency. The complexity at the kernel level mainly stems from the management of the memory tables, both local and remote, that have to be kept consistent across the network. The kernel trap is the main contributor to the additional latency.

Metric	Physical Addressing (kernel based pre-translation)	Virtual Addressing (ATU & NLAs)
min. Latency	2.59 µs	1.99 µs
max. Bandwidth	264 MB/s	275 MB/s
$n_{1/2}$ size, ping-pong	1024 bytes	576 bytes

Tabelle 7-7: RMA Virtual vs. Physical Addressing

Table 7-7 shows the resulting performance numbers of a comparison of both methods. The bandwidth does differ by a small amount, the latency is about 600 ns higher for the kernel level case. Also, the CPU load is increased by the kernel based solution.

7.5.6 RMA Microarchitecture

The RMA micro architecture is composed of three main modules, the requester, the completer and the responder. Each module is made up of one FSM and a datapath. Additionally, the requester features a command buffer to de-couple the HyperTransport interface respectively the CPU from the network. The responder and requester unit both feature a reordering buffer which is necessary to bring DMA read responses from the host back into request order.

The requester receives incoming posted packets from the CPU and stores them inside the command FIFO. Entries are removed from the FIFO and interpreted as RMA work requests. Depending on the type, data is requested from the host using DMA reads and the re-order buffer or the network packet is sent immediately (in case of an immediate *put* or a *get* operation, for example.

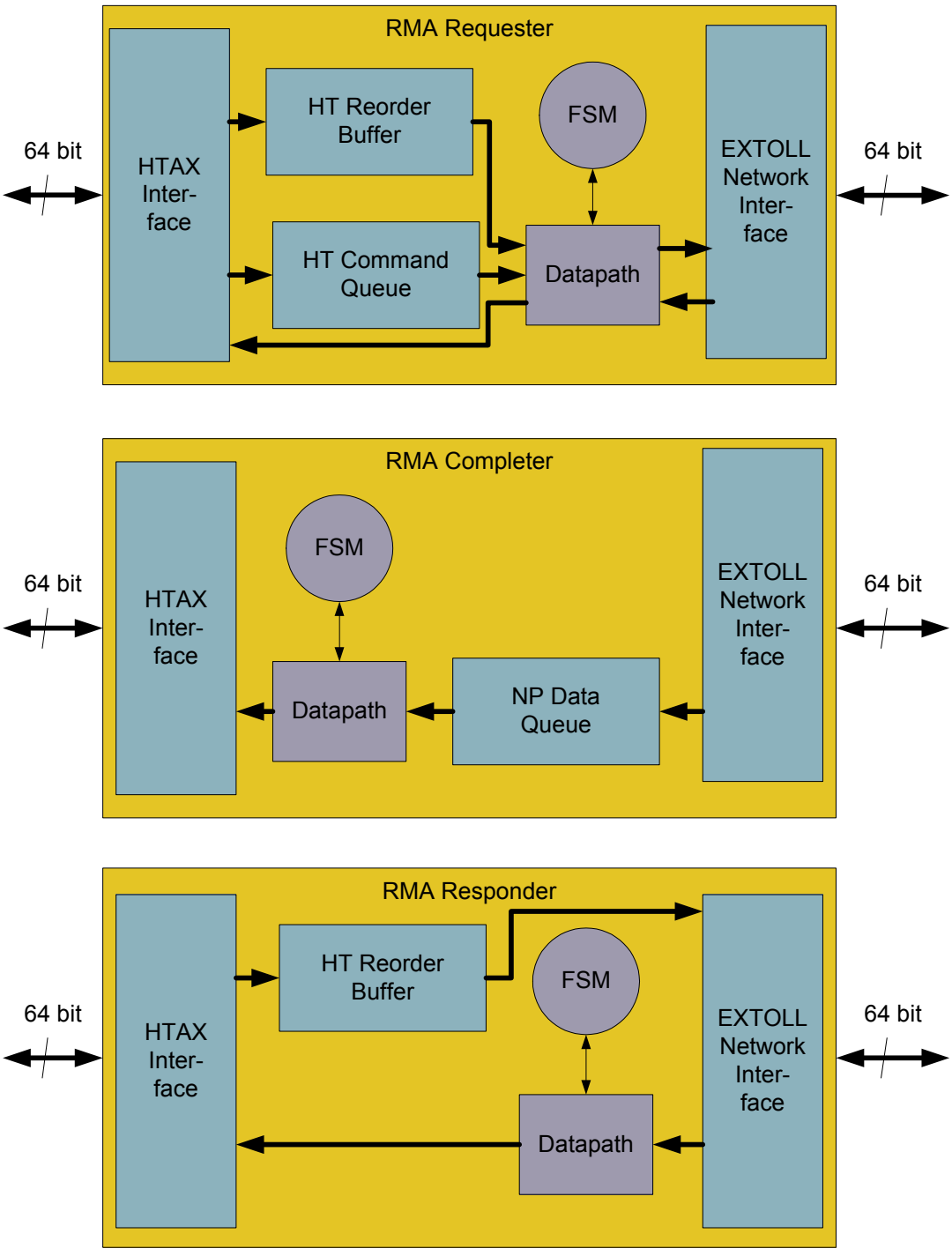


Figure 7-12: RMA Block Diagram

The completer receives *put*-packets from the network and completes them by writing them to the host memory. This unit also receives the *get*-responses which have been generated by a remote responder.

The responder handles incoming *get*-request and *lock* requests. Figure 7-12 shows the schematic block diagram of the different units.

7.5.7 RMA Registers

The RMA units feature a number of registers and look-up tables (virtualized registers). Again, they are created from a RDL specification and the complete reference can be found in [123]. A short summary is given in 7-8.

Name	Size	Description
Notification Write-Pointer	32 x 8 byte	one per notification queue
Notification Read-Pointer	32 x 8 byte	one per notification queue; each pointer resides in its own page
Notification Queue Size	8 byte	size of the notification queues
Requester PUT Command Counter	2 byte	operation counter
Requester GET Command Counter	2 byte	operation counter
Responder GET Command Counter	2 byte	operation counter
Responder PUT Command Counter	2 byte	operation counter
Completer PUT Command Counter	2 byte	operation counter
Completer GET Command Counter	2 byte	operation counter
Completer Enable	4 x 8 byte	256 bit to support 256 VPIDs; only 32 VPIDs used currently
Lock Base Register	8 byte	physical base address of the *lock* region
Lock Size Register	8 byte	size of the *lock* region

Tabelle 7-8: RMA Registers

7.6 EXTOLL URMAA engine

One other communication function that had been designed for EXTOLL and was originally envisioned to form the communication function is called *Unified Remote Memory Access Architecture* (URMAA). A Verilog HDL prototype simulation model for the URMAA was implemented [110]. The design of this unit builds heavily on the work of [115] and [129]. The URMAA engine is not completely finished because of resource constraints, since it requires more resources than the other functions described so far. Also, the URMAA is an engine which is more optimal for throughput communication than low latency with its multiple, simultaneous contexts. To reach low latency in the usual case, URMAA requires large on-chip caches which are not available for the FPGA technology. The high complexity of the URMAA design also puts requirements for verification, implementation and test very high.

7.7 EXTOLL FPGA Implementation

The EXTOLL prototype as an implementation of the EXTOLL architecture was conducted for the Xilinx FPGA Virtex-4 architectures. For the FPGA implementation parameter choices had to be made, for example the number of VPIDs was fixed to be 32 both for RMA and VELO. This was chosen because it allows a 32 simultaneous processes, a perfect match for coming 32 CPU-core systems and also of good size for 16 core systems which are expected to be the target system for the next step in the project.

Most of the design is coded using Verilog HDL. Most state machines have been developed using the FSMDesigner tool [112] which allows a graphical design of a finite state machine. The FSM can then automatically be converted into Verilog RTL code. Additionally, simulation and verification features help to verify the functionality of the design. As stated above, the register file is based on an automatic flow based on the RDL language. The crossbar components are generated from building blocks using scripts, thus they are available as parametrizable blocks with differing widths, number of ports etc. Additionally, the EXTOLL project uses a number of IP blocks to implement certain parts of the design. Namely, a CAG developed general FIFO block is used, RAM blocks, FPGA I/O blocks and clocking resources (digital clock managers, clock multiplexer etc.). The host interface is formed by the open-source HyperTransport IP developed and supported by the CAG.

The Verilog source code that is the result of the above tools is then used together with a standard Xilinx tool-flow to generate a final FPGA bitstream file. Xilinx XST is used for synthesis, ISE 10.1 map and place & route are used for the design [130]. Additionally Xilinx PlanAhead has been used for floor planning and design performance analysis for timing closure [131].

The complete simulation and verification is built on Cadence tools, notably *ncsim* and *simvision* are used. To model the host, the AMD HyperTransport Bus Functional Model 3.0 has been used [113]. The individual blocks were tested alone at first, and then in a design constituting one EXTOLL in loopback-link configuration as well as a simulation set-up where two EXTOLL instances were connected. These EXTOLL models were then used with random generated test vectors both for VELO- as well as RMA-communication. Additionally, corner cases were simulated using directed tests.

Some errors were still not found in simulation, mostly because it takes too long to simulate the system to get to this point or because of missing congruence of simulation with reality. For example the AMD Bus Functional Model does not exactly model an actual AMD CPU. Also in-depth post-place-and-route simulation was extremely difficult because of resource usage[1]. So, as a last stage the hardware was thoroughly tested in system. In addition to simulation the code base was linted.

The resulting netlist uses approximately 95 % of the slices of a Virtex-4 FX-100 FPGA, which is a very high usage rate for an FGPA design. A compilation of resource usage is given in table 7-9. Figure 7-13 shows a tree plot of the resource usage for the different blocks of the EXTOLL design as generated by the Xilinx PlanAhead tool (screenshot). From the plot one can see the distribution of resources to the different parts of EXTOLL. The largest part is the EXTOLL network layer with more than 50 % of the resources followed by the NIC. The HyperTransport core and other modules (I^2C, clock management) make up a small part. Figure 7-14 shows the design placed and routed; the different big blocks have been highlighted to show their relative usage and their placement. The EXTOLL network layer is shown in red, the NIC layer in blue, the HyperTransport core in green and the rest in grey. All blocks except the Linkports and the MGT wrapper module are automatically placed. The rectangular regions that were allocated for the Linkports and the MGT wrapper can be seen in the lower middle of the floor plan. The HT-Core clock domain runs at 200 MHz, the EXTOLL clock domains run at 156 MHz. All in all, the design employs 10 different clock domains.

The efficient HDL coding leads to high performance of the design despite the resource usage of the FPGA being at the limit.

Resource	Count	% of a VP4-FX100 device
Flip-Flops	27063	32 %
LUTs	74137	87 %
Slices	40355	95 %
RAM blocks	139	36 %

Tabelle 7-9: Design Resource Usage

1. A dual EXTOLL post-place-and-route simulation takes > 9 GB of RAM and simulates *very* slowly.

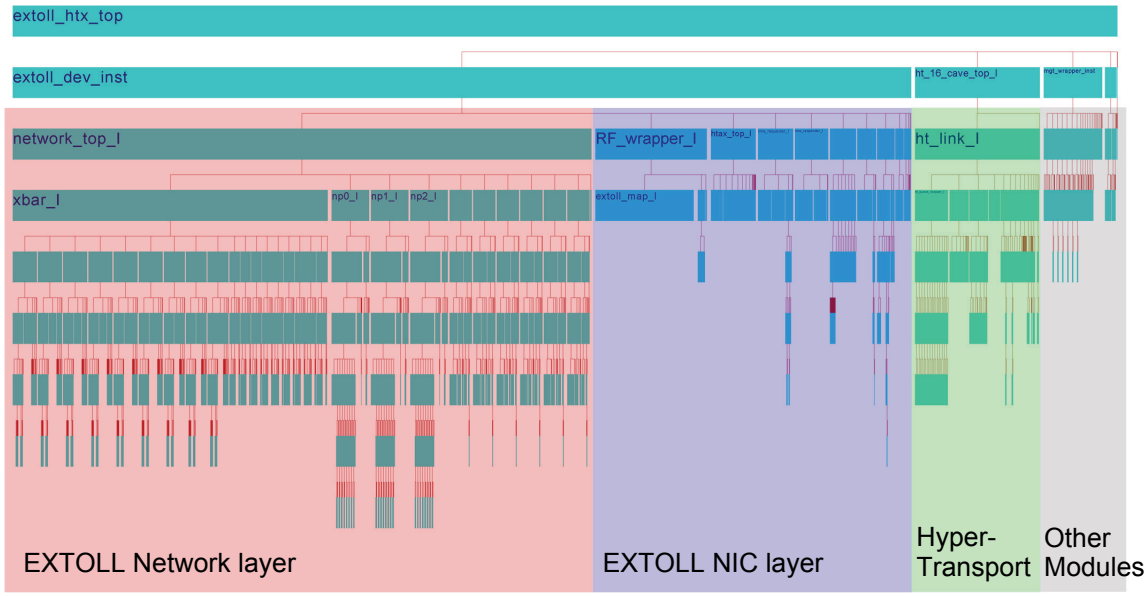


Figure 7-13: Resource Usage by Block (PlanAhead Screenshot)

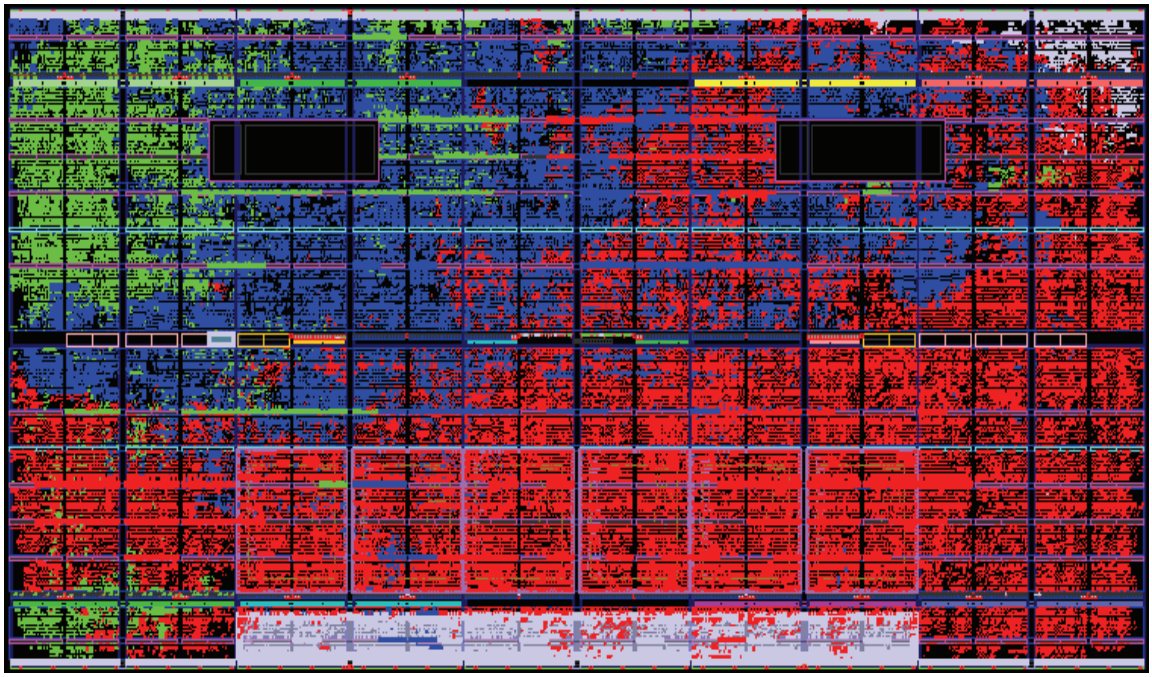


Figure 7-14: EXTOLL Placed and Routed on FX100 FPGA

7.8 EXTOLL ASIC

Preliminary experiments show, that an ASIC implementation of the same design is expected to reach 500 MHz even when implemented on a conservative process. The HyperTransport IP has been successfully mapped to a conservative standard cell implementation reaching more than 400 MHz without any optimization efforts. A design study synthesized the EXTOLL crossbar and a number of Linkports to form an EXTOLL switch. This design reached 1 GHz post synthesis on a current CMOS process technology.

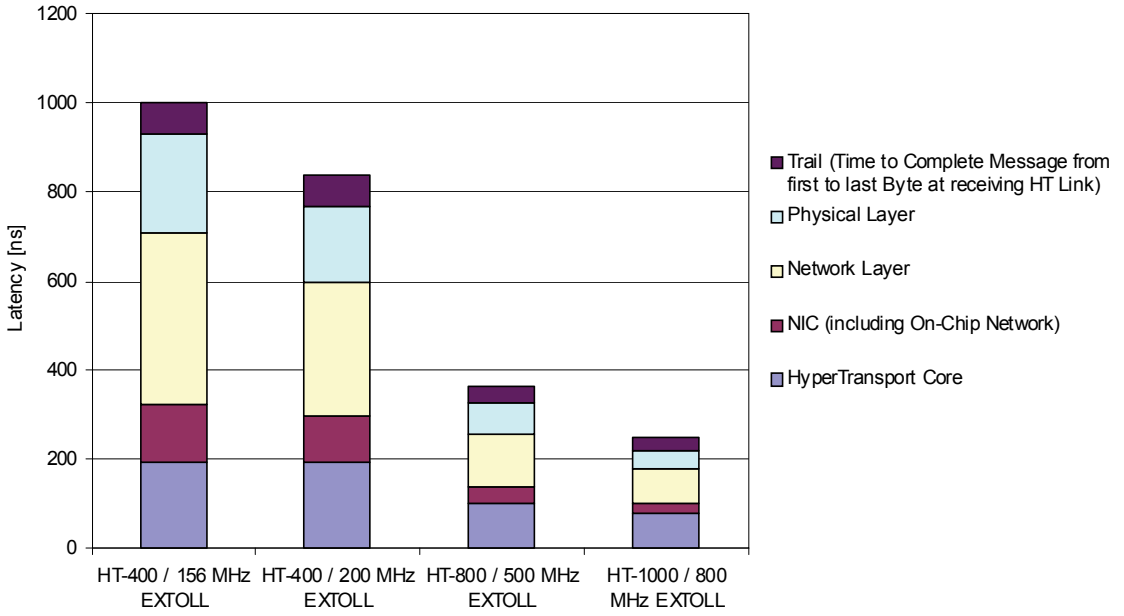


Figure 7-15: EXTOLL Latency Breakdown and Scaling

The higher clock speed of an ASIC implementation will have a dramatic effect on overall performance numbers. For example, the bandwidth for one link scales with the clock frequency, thus reaching 1 GB/s at 500 MHz. Latencies of all paths within the EXTOLL chips scale linearly with the clock frequency. Figure 7-15 shows the hardware latency of EXTOLL broken down into the latency due to the physical network layer (serializers/deserializers and synchronization), the network layer, the NIC and the HyperTransport connection. All of the latencies include the sending and the receiving side. These numbers were measured using a cycle accurate HDL simulation of two connected EXTOLL instances. The network latency also includes two switches, since the message passes at least two EXTOLL crossbars. The numbers do not include the latencies induced by the processor and the host cache-coherence and memory subsystem. This is the reason why the measured system latencies (Chapter 9) are higher. Table 7-10 summarizes the overall message latency as well as other key performance parameters for different technology scenarios. For reference, the

number for a Mellanox Connect X DDR HCA are given in the last row of the table. Startup latencies of both VELO and RMA are estimated to fall below 600 ns at 500 MHz EXTOLL clock frequency and reach 500 ns end-to-end latency at 800 MHz EXTOLL frequency. Even higher frequencies are imaginable for an EXTOLL ASIC, featuring an HT 3.x host interface and > 1 GHz of internal frequency. Higher frequencies of the HT interface also benefit the overall latency.

HT	Frequency	Start-up Latency	Bandwidth	Comment
HT-400	156 MHz	1300 ns	274 MB/s	FPGA, current prototype
HT-400	200 MHz	1000 ns	400 MB/s	FPGA, optimized
HT-800	500 MHz	800 ns	880 MB/s	ASIC
HT-1000	800 MHz	500 ns	2.8 GB/s	ASIC, 32-bit phit
n.a.	Reference: Mellanox ConnectX DDR	~1200 - 1500 ns	~1.5 GB/s	There are also ConnectX SDR and QDR adapters, their actual bandwidth may differ.

Tabelle 7-10: Effects of Technology

An ASIC implementation also opens the possibility to extend the design in certain areas to increase performance even further. Obvious candidates are the ATU unit and the *lock* operation in the RMA, which can both benefit from the availability of associative memory blocks. The ATU TLB can be changed from a direct-mapped version to a set-associative or even fully associative implementation. The higher integration possible for an ASIC also means that the number of VPIDs can be increased, for example to 128 or 512 VPIDs as well as the routing tables to accommodate for more nodes. A last improvement that promises significant advantages is to move the network layer from 16 bit to 32 bit. The actual code of the EXTOLL network layer can handle 32-bit phits without many modifications. This increase of the phit size can improve effective bandwidth (i.e. percentage of theoretical bandwidth that can be used for payload) by increasing the flit size. Also, latency is improved a little (the Networkport serializes/deserializes only 2:1 instead of 4:1). Finally, this change doubles the theoretical bandwidth in the network layer. The 800MHz design would show an aggregate bidirectional bandwidth of all 6 links of 16.8 GB/s. The NIC to network bandwidth would be in excess of 8 GB/s, and the singe link bandwidth would be > 2.8 GB/s. These high bandwidth numbers come together with the extremely low-latency characteristics of less than 500 ns of application-to-application latency.

The EXTOLL Software

Chapter 8

In this chapter the software stack for the EXTOLL environment is presented. A hardware design is futile without software that can actually use it. The schematic diagram in figure 8-1 shows an overview of the different components. A set of kernel level drivers forms the basis, complemented by a number of user-level APIs and middlewares. On the right side the network management software components are shown. Currently theses are limited to a simple tool. Not shown in the diagram is the possibility to also use kernel level services for data transport operations. This chapter describes the different software components in turn starting with the kernel level and ending with the APIs. For the middlewares (i.e. MPI and GASnet) design studies, concepts and protocols are presented complementing the analysis in chapter 4. The performance evaluation shows the capabilities of the EXTOLL system.

8.1 The EXTOLL Kernel Space Software

The EXTOLL kernel space software components are made up of five Linux kernel modules. The *extolldrv* component forms the basic EXTOLL driver. The *extoll_rf* component is responsible for the EXTOLL register file and the *sysfs* connection. *atudrv* manages ATU, while *velodrv* and *rmadrv* manage their respective functional units.

8.1.1 Base EXTOLL Driver

extolldrv detects the presence of a device, performs the basic configuration and enumeration of the device, remaps the PCI BAR spaces into the kernel virtual address space etc. In summary, extolldrv forms a PCI device driver for Linux. *extolldrv* does not expose any interface to user-space. Instead, it offers the possibility to get the physical and virtual addresses of important device resources for usage by other kernel modules. One other important feature of *extolldrv* is interrupt management. Other modules may register callback functions for different interrupt cause bits. Whenever an interrupt is triggered by EXTOLL the interrupt handler in *extolldrv* is executed, calls the registered interrupt callback functions, and clears the interrupt cause register.

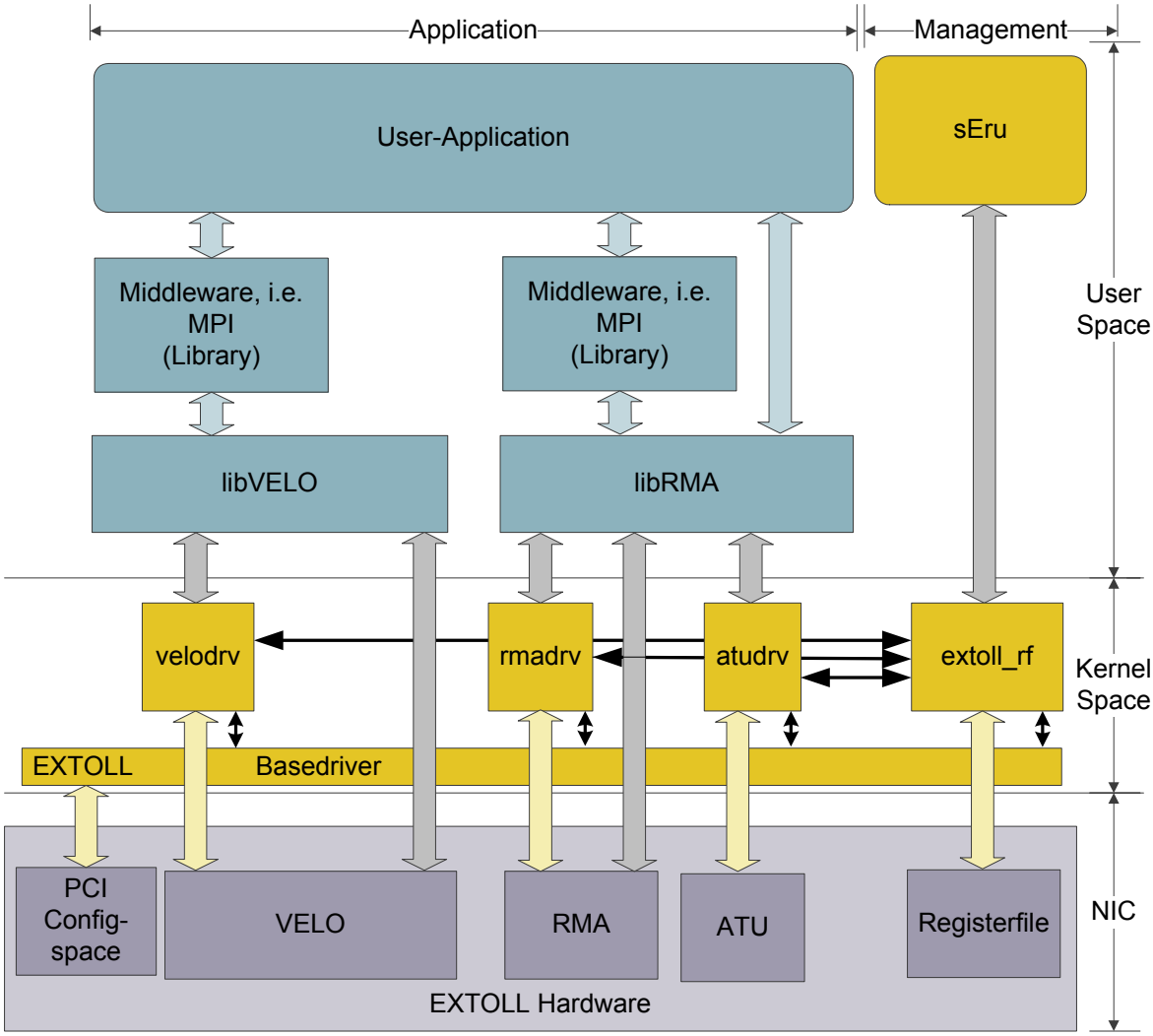


Figure 8-1: EXTOLL Software Stack

8.1.2 EXTOLL Registerfile Driver

The *extoll_rf* driver forms the interface to the EXTOLL registerfile. As described in section 7.2, this driver is generated fully automatic. It registers itself with the EXTOLL base driver and then exports a kernel-level API for convenient and hardware independent register access to the other modules. The *extoll_rf* driver also implements a *sysfs* interface for all of the registers. The *sysfs* pseudo-filesystem is a Linux feature to expose driver level configuration and status data to user-space. Within the sysfs filesystem, the EXTOLL registers are represented as a set of files in one common directory. By reading or writing from or to these files, the corresponding register is read from or written to. This interface is used by management and debugging tools to interact with the hardware. One interesting aspect of the interface is the ease of interaction with scripting languages from shell scripts to Python. Both the *sEru* management tool and the registerfile GUI application are based on this interface.

8.1.3 VELO Driver

The *velodrv* Linux module implements the complete handling of the VELO functional unit. It provides resource management including allocation of receive DMA memory areas, VPID management, opening and closing of end-points, requesting access to VELO requester pages. Additionally, the asynchronous read and write methods of the driver implement kernel level communication for user level programs. RX notifications can be based on polling or can be interrupt driven for VELO kernel communication. The interrupt driven code uses the interrupt callback service of the EXTOLL base driver. In the default operation mode of a VELO application, the driver is only invoked at startup to allocate and map needed resources and at the end of the application's lifetime to free the resources again.

8.1.4 RMA Driver

The RMA driver is analogous to *velodrv* but manages the RMA functional unit. The driver supports kernel level virtualization and full user-space communication. This requires an operational ATU. For ATU management the *rmadrv* uses services exported by *atudrv*. Again, the driver usually manages resources at startup and stop. In normal operation the registering and deregistering of memory regions are passed through *rmadrv* before using low-level routines from *atudrv*.

8.1.5 ATU Driver

The final kernel level component of the EXTOLL software stack is the *atudrv* module which manages the ATU. The driver exports an API to other kernel modules, notably the RMA driver, to allocate and free pages and ranges of pages. A best effort service to allocate contiguous *NLAs* for a given vector of virtual pages is provided. The API allows other drivers to use the ATU in different ways, particularly to implement different policies regarding NLA allocation. For example it is possible to allocate first level NLA regions to completely belong to one VPID. The API also enables the integration with a future kernel feature to allow dynamic memory unmapping (see also section 5.1.12).

8.1.6 Summary of Drivers

The EXTOLL kernel level software is complete in the sense that all hardware capabilities can be used and are exposed to APIs for applications. Communication is possible from user-space and kernel space. Kernel space communication so far has only been tested by initiating and completing requests from user applications. The integration with kernel initiated and completed communication layers remains to be implemented, for example NFS-over-EXTOLL and IP-over-EXTOLL. The current level of kernel code encompasses about 3800 lines of code plus 3000 lines which are automatically generated.

8.2 Routing and Management

Network management traditionally serves two main purposes, routing and fault handling. The current EXTOLL software stack does not include a polished network management solution as ATOLL did; rather a basic implementation was performed. To this purpose the Simple EXTOLL Routing Utility (SERU) was developed. SERU is written in Python and leverages the *sysfs* interface of the *extoll_rf* driver to communicate with the hardware. The tool supports the definition of the network topology. In initialization mode SERU reads the network topology from */etc/extoll/topology* and the node ID of the current node from */etc/extoll/nodeid* and then performs a shortest-path routing strategy for the network. This route management is sufficient for smaller networks as used for the analysis and bring-up of the prototype. A more elaborate and complete management system for EXTOLL remains to be developed. Experience with SERU and other projects suggest to implement this future EXTOLL Network Manager using the Python language.

To help debugging the network, the Registerfile GUI was developed, a C++ application using the Qt GUI framework which gives convenient access to the register values of EXTOLL.

For a future full-fledged EXTOLL Network manager, many requirements have to be considered, namely robust topology detection, routing policies, fault handling, performance monitoring and visualization of network properties.

8.3 The VELO Stack

The VELO stack is comprised of the kernel level *velodrv* already described and a low-level user-space API called *libvelo*. On top or in addition to this library several components have been developed to test and evaluate the VELO hardware.

The API, *libvelo*, provides all basic services available from the VELO unit. In particular, it is possible to open a port, the software abstraction of a completer mailbox. A port is always associated with a VPID, identifying the port on this node. Opening a port also means to remap the DMA region and the page containing the VELO read-pointer into user-space thus enabling to receive VELO messages completely independent from the kernel. Once a port is opened, an application is able to connect to other end-points identified by a *{nodeid, VPID}* tuple. Connecting actually is a virtual operation, in the same sense as this was the case with ATOLL, i.e. no actual connection is made, but the necessary resources are provided so as to be able to communicate with the specified peer. In the VELO case that means, the right requester page corresponding to the tuple *{source VPID, destination VPID, destination node}* is remapped into the user application space. Care is taken to remap the page using write-combining memory semantics.

Once a port is opened and the application has connected to its peers, messages can be sent using the *velo_send* function which accepts a source buffer of up to 64 bytes and a tag. The tag is allowed to be in the range of 0 to 7 and is translated to an MTT on the hardware side.

On the receiving side, two functions are available *velo_probe* and *velo_recv*. While *velo_recv* receives blocking in the sense that it polls until a message is available, *velo_probe* is unblocking by just checking if a message is available. Additional receive-side functions are a matching receiving function and the *velo_check* function which only checks if a message is available.

The API also includes a number of utility functions to set and query different parameters. For easier parallel programming, the *atollrun* environment was ported to the VELO environment, and then called *velorun*. *Velorun* accepts the same parameters as *atollrun* and in essence performs the same steps. It is also written in Python and allows for a parallel application to be started in parallel on a cluster of nodes. Finally, an experimental byte-transfer-layer (BTL) component for OpenMPI has been developed, which allows MPI programs to transfer messages of a size of up to 48 bytes using VELO. The remaining 16 byte of a maximum sized VELO message are used for the MPI header.

Finally, a number of evaluation applications and tools has been written for VELO and the NetPIPE [47] benchmark application has been ported to VELO. Evaluation applications perform transmission and reception of messages, ping-pong traffic patterns, bandwidth measurements as well as a multi-threaded latency measurement tool which shows the exceptional behavior of the virtualized VELO unit when used by multiple cores of a node.

8.4 The EXTOLL RMA Software Stack

The RMA user-space software consists of an API library, which provides the services the hardware offers to the software. It is possible to request and open an end-point (a VPID), register and deregister memory regions, post work requests and poll for notification events. Descriptors and notifications are represented with their own data type each and there are functions available to manipulate all of the aspects of the datatypes. The design mimics the object-oriented approach in C and it also resembles design principles that have been successfully applied to the PALMS API. Using the provided datatypes and functions, the complete power and flexibility of the RMA engine is made available to user-space applications. There are also functions available that are shortcuts for a sequence of function calls, which are expected to occur often. For example, there is a function that performs a *put* operation directly instead of having to allocate a descriptor, fill it with values and then post it to the hardware. Again this procedure is very similar to the techniques that have been used with PALMS.

8.5 EXTOLL Kernel-Level Communication

As mentioned above, both VELO and RMA also have the capability to provide for kernel level communication thus avoiding user-space communication. The main reason to include this in the system is to enable compatible behavior with legacy applications, for example sockets-based applications, and an efficient implementation of kernel-only protocols like NFS or other storage protocols. There are also people that claim user-space communication

to be a concept from the past that is superfluous with today's systems providing very high performance with traditional OS based I/O or even I/O in VMM environments using virtual software devices and software based virtualization to shared I/O devices in a single domain.

The VELO kernel communication capabilities can be used to perform a comparison of different I/O methods on a given platform presenting insight into the question what the impact of the OS is on performance of a high-performance device. It could be shown that the difference in start-up latency equals 300 ns on the sender-side and another 300 ns on the receiver-side for a total of 600 ns of overhead. The bandwidth was not further affected. Experiments using interrupt driven communication yielded again different results. Depending on the configuration of the kernel in terms of timer interrupt frequency and activated preemption start-up latency increases substantially, because the receiving thread is awoken after an interrupt occurred and the time for the thread to become scheduled on a CPU varies amongst others with the mentioned parameters. In summary it can be said that the latency was increased by approximately another microsecond through interrupts. For RMA, the first prototype implemented was based on kernel communication [128] and did not need an ATU unit for RMA. The results of a comparison between kernel-space and user-space based communication have been presented in section 7.5.5 (both use polling receives).

8.6 EXTOLL MPI - Protocols

MPI is comprised of several distinct parts, as already discussed in chapter 4. The design space for actual MPI implementations is rather large. One question is for example which MPI to base support for EXTOLL on. The candidate chosen is OpenMPI, which offers a component oriented architecture (figure 8-2) well suited for rapid integration of new features. On the downside, the component and plug-in oriented software architecture of OpenMPI causes some decrease in performance numbers. To implement point-to-point MPI functionality, i.e. MPI-1 features, at least two different components can be chosen in OpenMPI. Implementing a component for the BTL is the most general solution. In this case, higher level OpenMPI components implement request matching and completion. Another possibility is to implement a message-transfer-layer (MTL) component. In this case, more functionality has to be provided by the network specific component, but the MTL also enables to implement more network specific optimizations. An experimental VELO BTL component has been written. It is far from full production quality code, does not actually exploit the hardware features of EXTOLL and does not deliver the performance expected (see also [127]). The VELO BTL is just capable of sending and receiving messages of a restricted size. An RMA BTL could be implemented that allows to send messages of up to 4 kB using *put*-operations into previously allocated and exchanged memory regions, basically implementing a 2-copy emulation protocol for send/receive based on RMA. Requester notifications are used to manage freeing of resources when sending and completer notifications are used to detect the arrival of new messages. Notification *puts* are used to implement the needed receiver-controlled flow-control for the protocol.

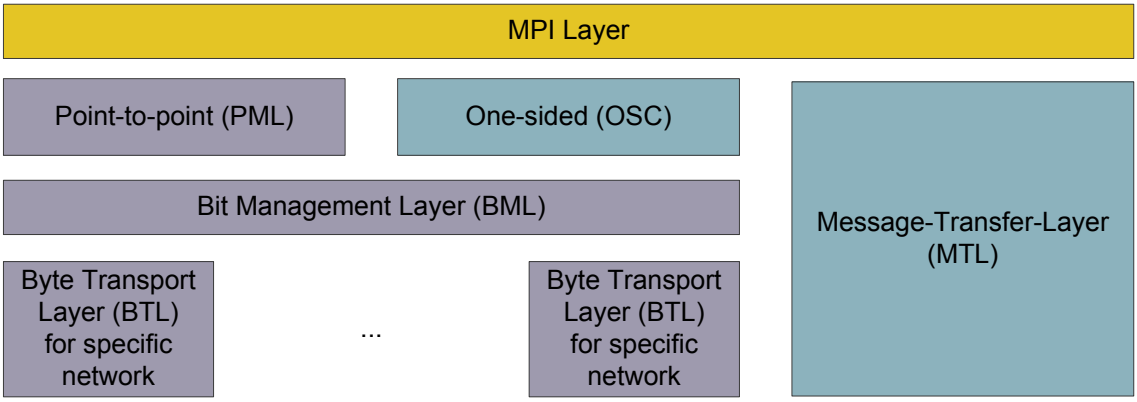


Figure 8-2: OpenMPI Architecture

Since the BTL is a low-layer component of OpenMPI, and a considerable amount of layers is located on top of it, the performance numbers for both BTL components are lower than if communicating using the low-level API directly. A method to optimize MPI behavior is to reimplement higher levels of OpenMPI, respectively provide a higher level specialized for EXTOLL. For example the Myricom MX [96] communication framework is integrated into OpenMPI at the MTL level providing a very low overhead MPI integration. The same or a similar implementation ultimately has to be provided for EXTOLL.

Such a high-level MPI integration calls for a number of protocols and algorithms to solve the most important MPI problems. For EXTOLL a small message, medium message and large message protocol are envisioned as analyzed in chapter 4. All of them are described in detail in the following paragraphs. Together with a matching function, they are expected to form an efficient, high-performance point-to-point MPI-1 implementation. The collective operations can then use the reference OpenMPI implementation which is built on top of point-to-point requests. If collective hardware features become available to the EXTOLL architecture, the collective component can also be exchanged. In this case, the functions that are hardware accelerated will be rewritten selectively while keeping the reference implementation for the rest.

For the small message protocol, a straight forward VELO implementation is the best performing choice. The messages are sent using VELO; the actual payload is prepended with a minimal header containing the necessary data for message matching which are not automatically present within the VELO status word. The user tag, communicator context ID and exact size in bytes have to be added to the message. Depending on encoding either 8 or 16 byte are used by this header. Thus the effective payload for small MPI messages is 56 or 48 bytes (figure 8-3). Matching is performed entirely in software, unexpected messages are moved from the VELO DMA area to the central *unexpected message queue*.

For messages larger than 56 bytes, a different protocol must be used. Two candidates were identified. First, it is possible to send a larger message by splitting it up into several fragments each being transported by VELO. This protocol uses the MTT feature to distinguish a multi-segment message from a normal message. A multi-segment message features an MTT

VELO Request:

Target Node (5 bit)	Target VPID (5 bit)	Source VPID (5 bit)	MTT of 0	VELO Length	Reserved (6 bit)

VELO Packet:

MPI-Tag	MPI Communicator	Size	RSV
Payload (Byte 0 – 7)			
Payload (Byte 8 – 15)			
Payload (Byte 16 – 23)			
Payload (Byte 24 – 31)			
Payload (Byte 32 – 39)			
Payload (Byte 40 – 47)			
Payload (Byte 48 – 55)			

Figure 8-3: Small-Message Layout

tag of one instead of zero. In the MPI header of the first message, the number of fragments is stored in the last remaining byte of the 8-byte MPI header area. The size field now gives the number of bytes valid in the last fragment. All previous fragments have maximum size, i.e. 56 bytes for the first segment and 64 for the following. Thus, it is possible to send up to $(2^8 - 1) \cdot 64 + 56 = 16.376$ bytes using VELO. The packet format is shown in figure 8-4.

The second candidate for medium-sized messages is based on RMA. Each process allocates a block of memory as receive buffer for each peer. This memory block can be allocated virtually contiguous, because it will be accessed using normal RMA *put* operations. The location and size of each of these blocks is sent to the corresponding peer. A process manages a write-pointer and a read-pointer for each remote buffer it knows about. Whenever the process wants to send a message, the message is sent to the location starting at the offset given by the write-pointer within the receive buffer exported from the receiving process. The actual sending is done using a one-sided *put* transaction. On the receiving side, the *put* operation causes a *remote completion* to be generated, which the receiving process polls for. Remote completions are a novel concept introduced by the RMA unit of EXTOLL, where a one-sided operation can also cause a completion to be generated on the target side. Now, the receiver can proceed just like in the ring-buffer style messaging. The receiver also manages a read-pointer to the buffer. Whenever a certain threshold is reached, the receiver sends a message to the original sender indicating how many received bytes have been consumed by it, so freeing this part of the buffer again for new messages. A typical sequence for this protocol for medium messages is shown in figure 8-5.

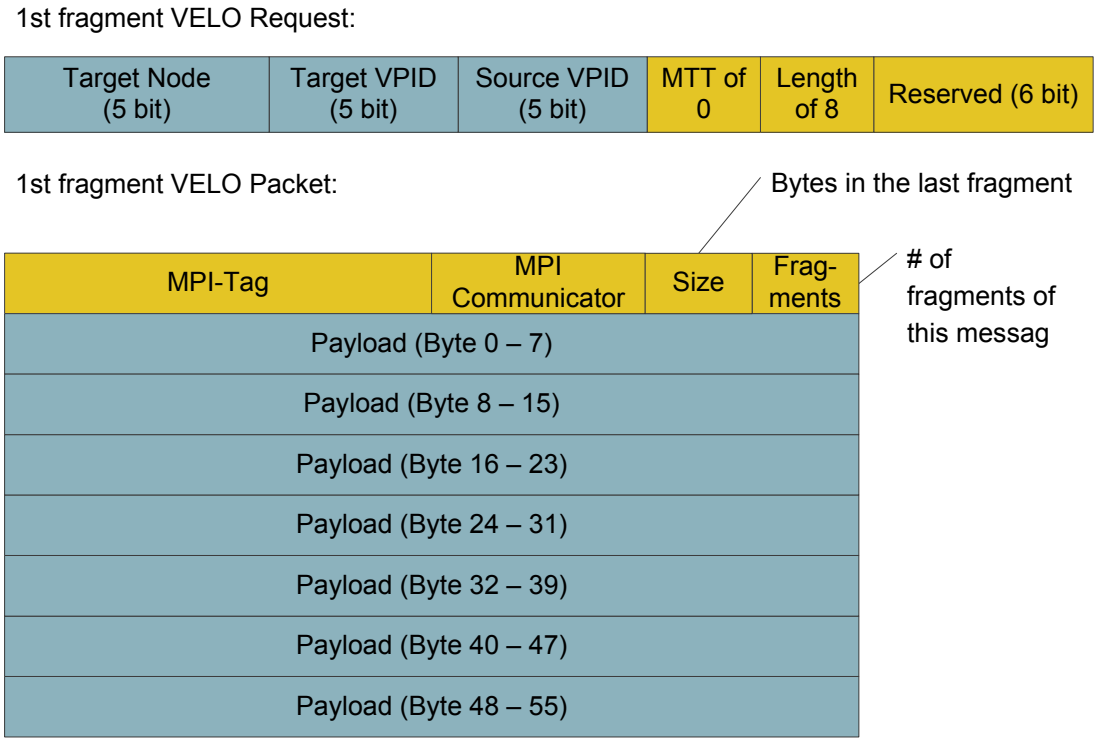


Figure 8-4: VELO-Based Medium Message Layout

Which of the two protocols is used or even if both protocols are used for different message sizes, remains to be empirically determined. This also holds true for the question when to switch to the last of the protocols, the large message protocol which is based on an RMA rendezvous protocol and is the only protocol so far that is a true zero-copy protocol. The large message protocol is based on the idea described in chapter 4.3. If a large message is to be sent, the sender posts a VELO message with a MTT reserved for this protocol. The VELO request has a size of 16 byte and the structure shown in figure 8-6.

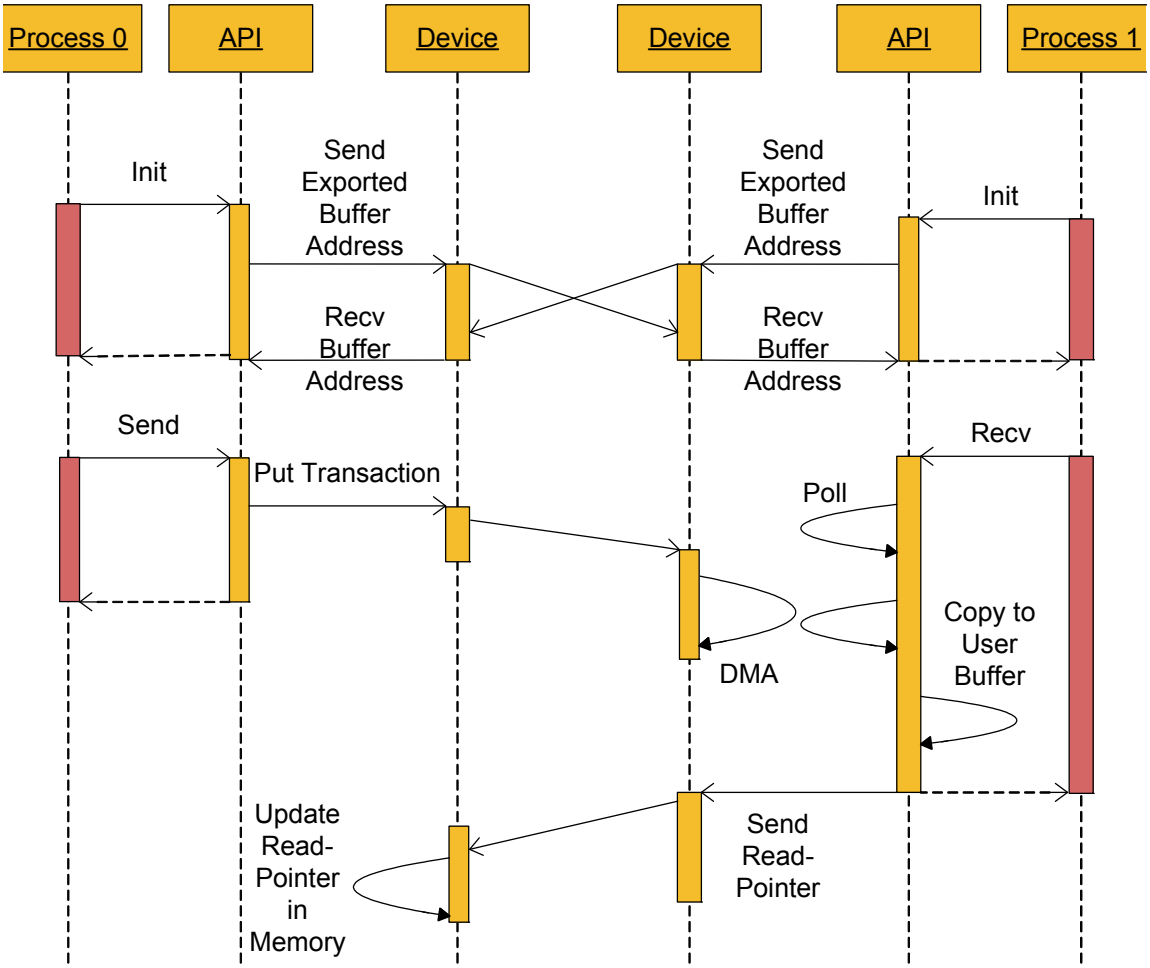


Figure 8-5: RMA-Based Medium Size Protocol

VELO Request for large messages:

Target Node (5 bit)	Target VPID (5 bit)	Source VPID (5 bit)	MTT of 2	Length 3	Reserved (6 bit)

VELO Packet for large messages:

MPI-Tag	MPI Communicator	RSV
Message Length	RSV	
NLA of the Source Buffer		

Figure 8-6: Large-Message Request Layout

Once the receiver obtains the message it can be matched against the already posted receive requests. If no match is available, the request is stored in the unexpected queue. Once the request can be matched, the second part of the protocol is executed. The receiver posts a number of RMA *get* operations to directly read the complete data payload from the sender's original buffer into the final application buffer in the receiving application. The last *get* transaction uses the *responder notification modifier* bit to generate a notification at the sender. This enables the sender to efficiently detect when the application buffer can be reused. For this last bit the completer notification bit is also set so that the receiver can detect the completion of the overall operation. This protocol implements a true zero-copy rendezvous protocol shown in figure 4-4. The solution also employs the minimum number of network roundtrips, since the sender notification can be achieved implicitly.

As a proof of concept the described protocols have been implemented outside MPI and tested. The resulting performance numbers are presented in the evaluation section of this chapter.

All of the MPI discussion in this section was focused on MPI-1 until here. As we have seen in chapter 4, MPI-2 is becoming more and more important if the performance problems that many implementations suffer from are solved. EXTOLL provides a nearly one-to-one hardware solution for MPI-2. In OpenMPI, the one-sided communication is handled by an instance of an one-sided component (OSC) (figure 8-2). In the base distribution, only one implementation is available which implements one-sided communication on top of point-to-point messaging, a non-optimal solution leaving room for significant enhancements. An OpenMPI OSC was developed which directly leverages EXTOLL hardware functionality.

The MPI-2 implementations compromises the following MPI functions: *MPI_Put*, *MPI_Get*, *MPI_Fence*, *MPI_Post/Start/Complete/Wait* and the *MPI_Window* handling functions. The *MPI_Accumulate* function has been implemented using a *get* transaction followed by the data transformation and a subsequent *put* operation to complete the accumulate function at the target. Actually, it is very difficult for any hardware to implement *MPI_Accumulate* satisfactorily and the solution chosen suffers from an additional network round-trip of all of the data. The alternative is to implement the accumulation at the target side, but this defeats the passiveness of the target.

Put and *get* operations are straight forward assembled from RMA *put* and *get* transactions using the RMA API library. Only requester side notifications are used to mimic the semantics of MPI-2 in respect of the completion of events. The *MPI_Fence* function is implemented using mutual, immediate *put* operations. The *MPI_Post/Start/Complete/Wait* operations as well as the passive target synchronization using *MPI_Lock/Unlock* has been implemented using RMA *lock* transactions. Finally, *MPI_Window*s are implemented using RMA API level memory regions.

The complete MPI-2 component puts a thin layer on top of the *librma*. The performance and also the overlap of communication and computation is presented in the evaluation section.

8.7 EXTOLL GASNET - Protocols

The GASnet specification for the support of PGAS languages has been introduced in section 4.4.4 and identified as an important target for EXTOLL based systems. The GASnet Core API maps nearly 1:1 on VELO and RMA transactions. Communication in GASnet is based on the Active Message [81] idea. Three different message types are differentiated by GASnet: short, medium and long requests. All three carry a handler and a up to 16 parameters in the AM message. Each parameter has a size of 32 bit. Unfortunately this is too much for one VELO operation; the design decision is to send one VELO message if possible and split the request into two messages if it is too big. See figure 8-7 for an illustration of the mapping of AM request parameters to VELO messages.

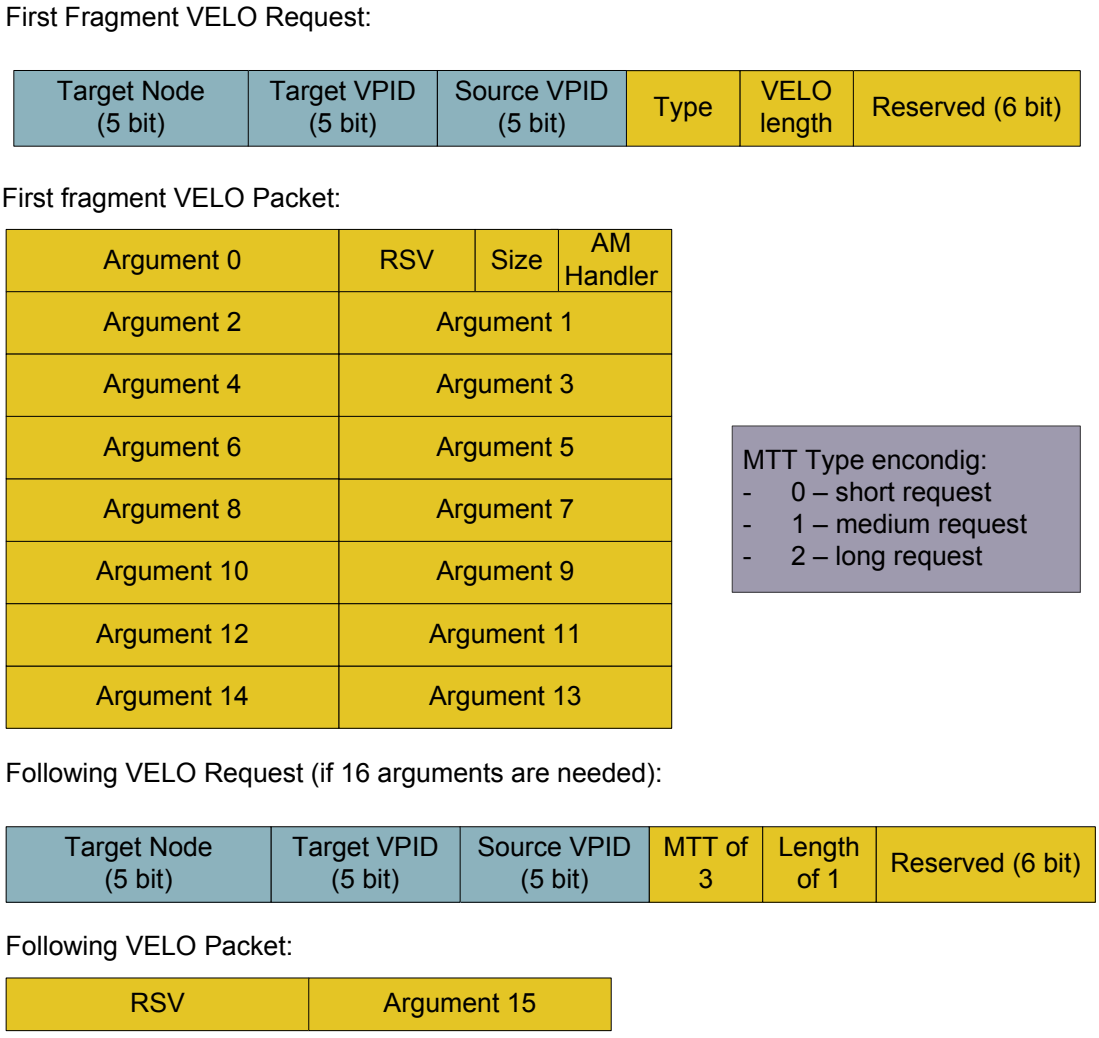


Figure 8-7: GASnet Request Layout

Replies can be handled the same way: the VELO MTT feature can be used to distinguish requests from replies quickly. Short requests do not carry additional payload, however medium and long requests carry additional payload. The medium requests use a non-zero copy protocol. Implementations are free to chose the maximum size of medium requests as long as it is bigger than 512 bytes. The payload can be transported using one of the medium size MPI protocols introduced in the previous section. So, a medium GASnet request causes one or two VELO messages transporting the actual request and additionally the RMA requests for the payload. Long requests are a combination of an AM and a one-sided *put* into the (registered) shared segment of the receiving GASnet peer. It is obvious, that long requests are implemented using again VELO messages for the request and a direct RMA *put* for the payload.

The extended API actually proves to be much more difficult to implement. A first shot may reuse the Firehose library [82] that is part of the GASnet source distribution to implement memory management. Besides memory registration the different one-sided operations can be mapped to RMA transactions straight forward. Other parts of the Extended API of GASnet can be provided by the reference implementation, which is based on an realization of the base API. Successively, more functions of the Extended API can then be replaced by more optimized variants for EXTOLL.

The actual implementation for GASnet for EXTOLL remains to be done and the resulting PGAS platform and its performance will be most promising. The basic protocols are well understood and tested for the MPI implementation.

8.8 Software Summary

The EXTOLL architecture has already proven to exhibit very interesting properties with the prototype implementation that was described in chapter 7. This chapter introduced the EXTOLL software stack for the hardware. It was shown how the hardware and software components interact and are designed to optimally work together in implementing methods for communication. Specifically, the low-level system integration, API level software and protocol design to support higher level software were shown.

The software development is still at an early stage and full fledged, production ready packages to support MPI-1, MPI-2, GASnet and middlewares to support enterprise clients are needed. In addition, the network management part has to be greatly expanded, either in the direction of the ATOLL daemon *AtollD* [40] or the IB subnet manager.

Results and Performance Evaluation

Chapter 9

This chapter describes the performance of the EXTOLL prototype system as it was benchmarked in a real two-node configuration. It proves the functionality of the architecture, the design, implementation, and the excellent performance of EXTOLL. A number of different microbenchmarks and special programs were run to give an overview of the different properties.

9.1 Microbenchmark Results

A basic ping-pong microbenchmark was executed on two nodes. The ping-pong benchmark varied the size of the transaction. Also, the different methods for sending data using EXTOLL were used. All of the latency and bandwidth plots that follow use a logarithmic scaling for the x-axis, the operation size.

The resulting plot for VELO messages, RMA *put* and RMA *get* transactions is shown in figure 9-1. The RMA *get* operation actually performs one *get* operation (round-trip). The latencies for the other two operations is plotted as half-round-trip, i.e. the measured latency for one complete round-trip is halved prior to plotting it. It can be seen, that VELO starts with a lower latency then RMA, mostly caused by the necessary additional DMA transfers. At a size of about 400 bytes, RMA already surpasses VELO and provides a lower latency.

To further the understanding of the start-up latency of EXTOLL, figure 9-2 shows the latencies for a single operation of each communication type in the EXTOLL system. The numerical values can be found in the last column of table 9-1. VELO starts with the lowest latency of 1.3 µs. It is closely followed by an RMA *notification put* transaction. Figure 9-3 shows a plot of the predicted latency per operation based on an analysis of the sub-operations that form the operation and the time to complete each sub-operation. All operations are divided into *DMA reads*, *ATU translations*, *network trips*, *DMA writes*, *PIO writes* and the *software overhead*. The estimated values for the sub-operations are listed in table 9-2.

It is clear that a VELO transaction and an RMA *notification put* operation have very similar characteristics. The additional 100 ns of an RMA *notification put* are due to additional FIFOs and pipeline stages within the RMA hardware. The next lowest latency is exhibited by the *immediate put* operation without notification. Its startup latency is a little higher than the *notification put* although they cause the same number of DMA accesses. The difference of 70 ns is caused by the ATU translation necessary at the RMA Completer. If the *immedi-*

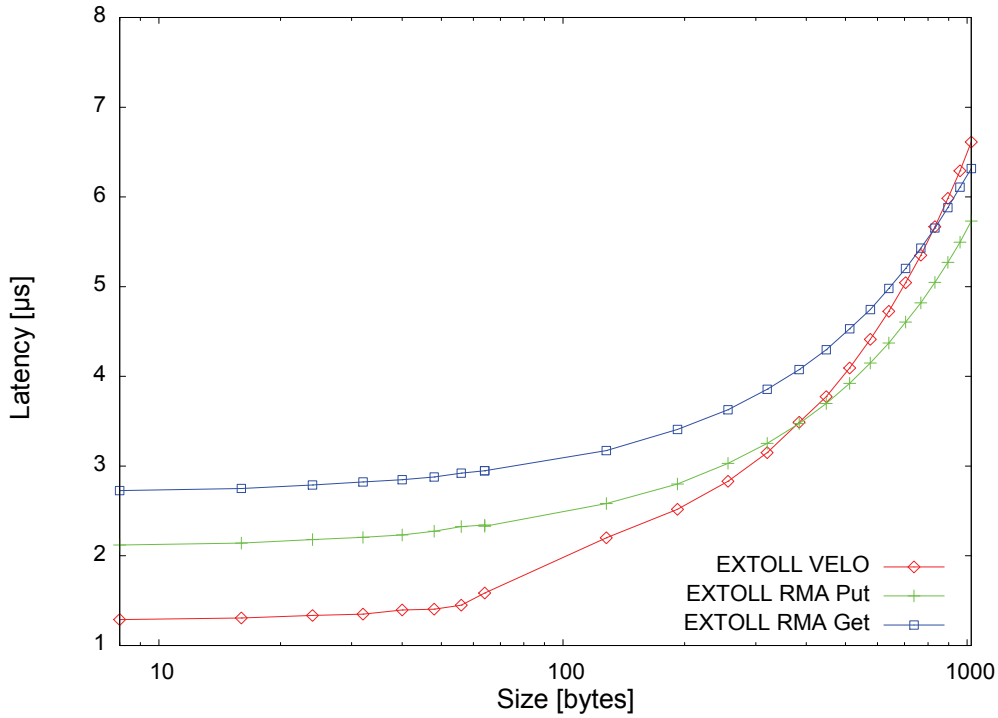


Figure 9-1: EXTOLL Ping-Pong Latencies

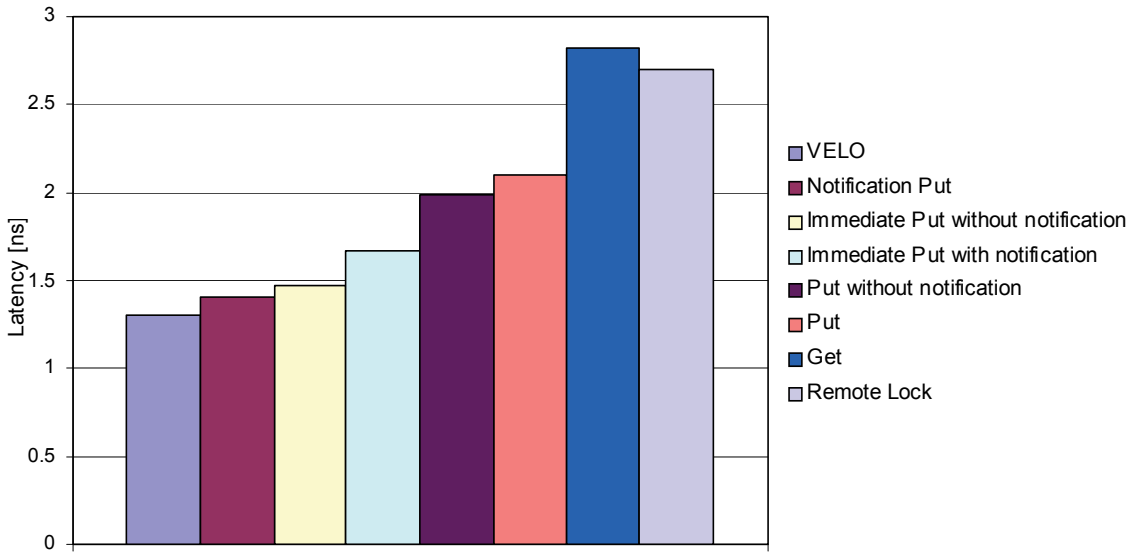


Figure 9-2: EXTOLL Start-up Latencies

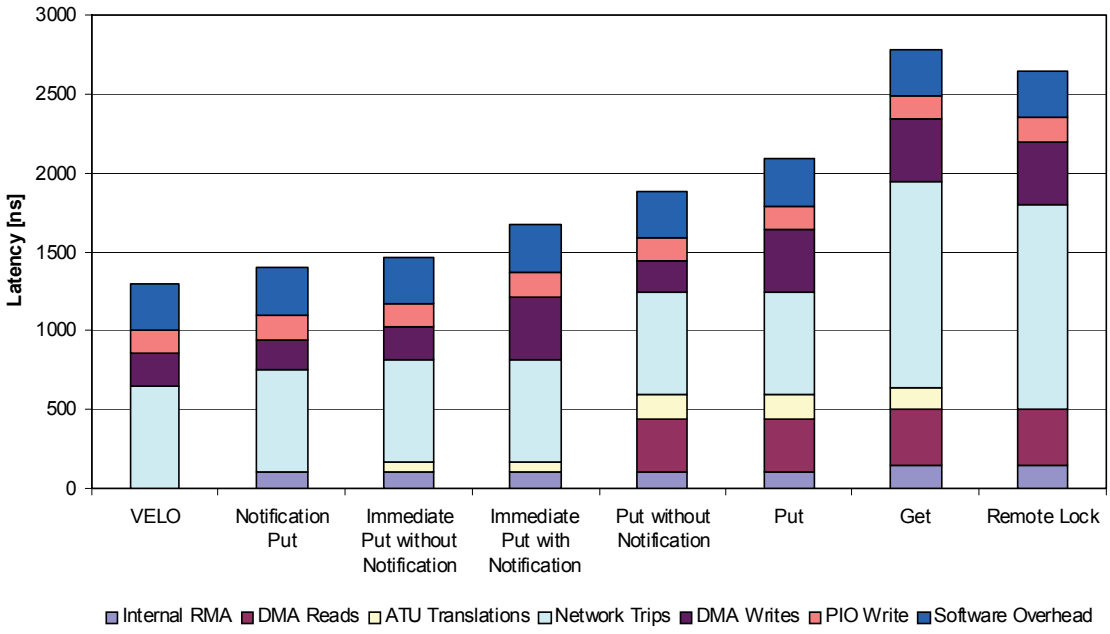


Figure 9-3: EXTOLL Latency Details

ate put operation is performed with a completer notification, 200 ns additional latency can be seen stemming from the additional DMA write operation performed. The same effect is also visible from the latencies of a normal *put* with and without completer notification. The higher latency of the *get* transaction is finally caused by the double network latency of a full roundtrip and the additional passing of the RMA responder.

The bandwidth reached using the ping-pong test is summarized in Figure 9-4. It can be seen, that VELO never reaches the maximum possible bandwidth, whereas *put* reaches it and *get* remains very close below it.

Besides the ping-pong latency, the streaming bandwidth was also tested. This microbenchmark measures the maximum bandwidth that can be transported from one peer to another for a given operation type and a given operation size. The results are shown in Figure 9-5. VELO already reaches its maximum bandwidth with operations of 64 bytes - which is also the maximum size for a single VELO operation. Both *put* and *get* bandwidths rise very steep and surpass VELO bandwidth at 384 respectively 512 bytes. The maximum bandwidth is reached by *put* and *get* with operations of ~4 kB size. Half of the maximum bandwidth (n½) is already reached at an operation size of 32 bytes (VELO operation).

One other performance measure that is closely related to the streaming bandwidth and that is sometimes cited is the message rate of a network interface. EXTOLL reaches the message rates summarized in table 9-3. VELO's number is especially impressing with more than five million messages that can be sent per second.

Operation	# of DMA Reads	# of ATU Trans.	# of Net. Trips	# of DMA Writes	# of PIO Write	# of RMA Units	Latency Prognosis [ns]	Latency Measured [ns]
VELO	0	0	1	1	1	0	1300	1300
Notif. Put	0	0	1	1	1	2	1400	1400
Imm. Put w. Notif.	0	1	1	1	1	2	1470	1470
Imm. Put w/o. Notif.	0	1	1	2	1	2	1670	1670
Put w/o. Norif.	1	2	1	1	1	2	1890	1990
Put	1	2	1	2	1	2	2090	2100
Get	1	2	2	2	1	3	2790	2820
Lock	1	0	2	2	1	3	2650	2700

Tabelle 9-1: Sub-operation Composition of EXTOLL Transactions

Sub-Operation	Time [ns]
DMA Read	350
ATU Translation (TLB Hit)	70
Network Trip	650
DMA Write	200
PIO Write	150
Software Overhead	300
RMA Units Passed	50

Tabelle 9-2: Sub-operation Timing

The difference in user-space versus kernel based communication was already mentioned. Both VELO and RMA can be driven from the kernel. If an additional transition from user-space to kernel-space is performed for an operation, this adds about 600 ns to the start-up latency of the operation, regardless if it is an RMA or VELO operation. If the operation also completes via an interrupt, the actual latency increase can be higher than that, since additional interrupt latencies have to be taken into account.

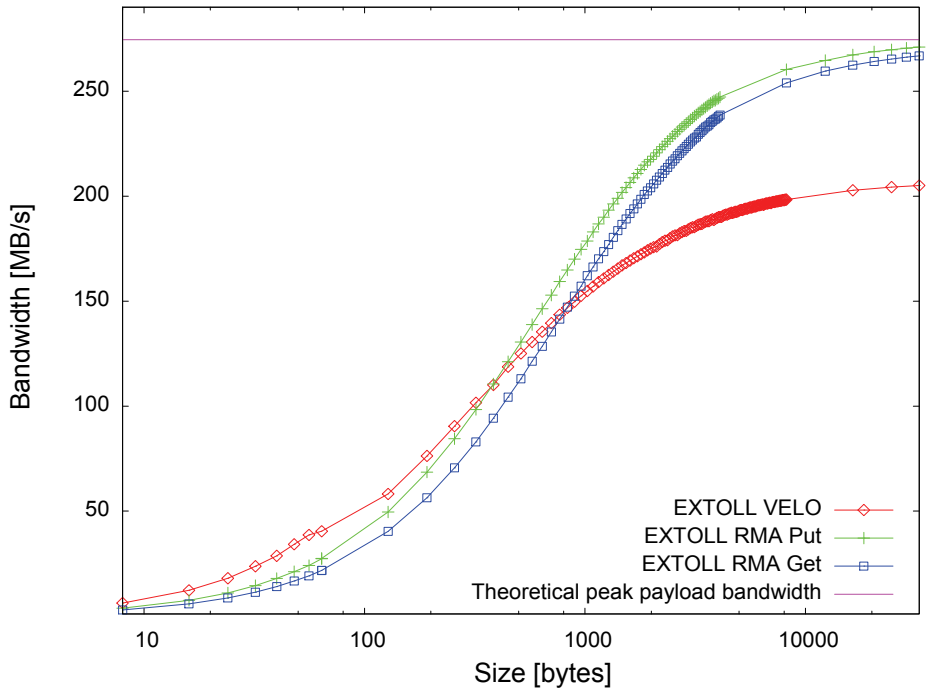


Figure 9-4: EXTOLL Ping-Pong Bandwidth

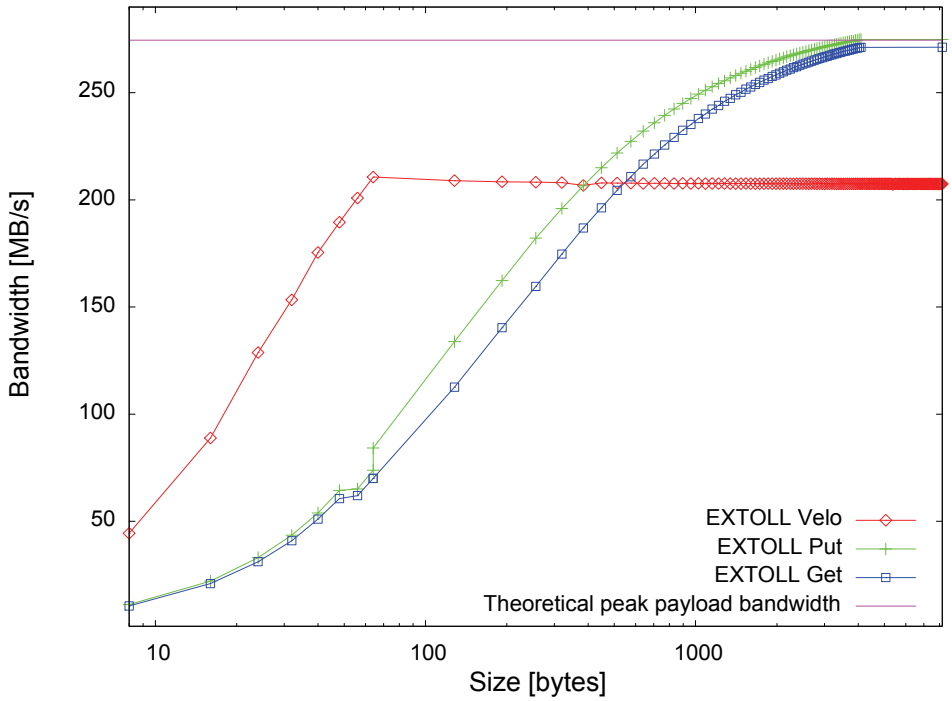


Figure 9-5: EXTOLL Streaming Bandwidth

Operation	Message Rate
VELO	$5.5 \cdot 10^6$
Notification Puts	$2.6 \cdot 10^6$
Standard Put	$1.375 \cdot 10^6$

Tabelle 9-3: EXTOLL Message Rates

9.2 RMA one-sided MPI-2 Prototype

The basic characteristics of the operations used for MPI-2 have been presented in the previous section. One additional microbenchmark is very interesting when talking about one-sided communications and that is computation-communication overlap. The benchmark used is similar to the test described in [73].

The benchmark performs synchronization, followed by a *put* operation and finishes with another synchronization operation. An increasing amount of simulated computation is inserted between the *put* operation and the synchronization that concludes the test. The percentage given is the percentage of the time measured for the communication operation without computation in relation to the time that can be spent computing without increasing the overall runtime. A high overlap is very desirable.

The normal OpenMPI one-sided communication[1] implements all one-sided communication using two-sided primitives. Not surprisingly, the measurement shows an overlap of nearly zero for all tested scenarios using OpenMPI. The MVAPICH MPI implementation for Infiniband is reported to perform better, at least in certain cases. In [73] an improved MVAPICH implementation reaches overlap rates of 10 to 70 % for different message sizes with 30 % for messages sizes of 4 kB. The EXTOLL implementation is able to reach overlap as high as 80 %. The results of the test are shown in figure 9-6. Even small message sizes such as 32 byte show significant potential for overlap. This is due to the low overhead necessary to initiate a *put* operation. These measurements have been performed with a development version of the OpenMPI one-sided component for EXTOLL.

9.3 MPI-1 Protocols

To analyze the performance and the handover point for the protocols suggested for MPI-1, test implementations of all protocols were developed and the resulting latencies for the four protocols were measured. Figure 9-7 shows the results of the test. The *Tiny* protocol uses a

1. OpenMPI version 1.2.6

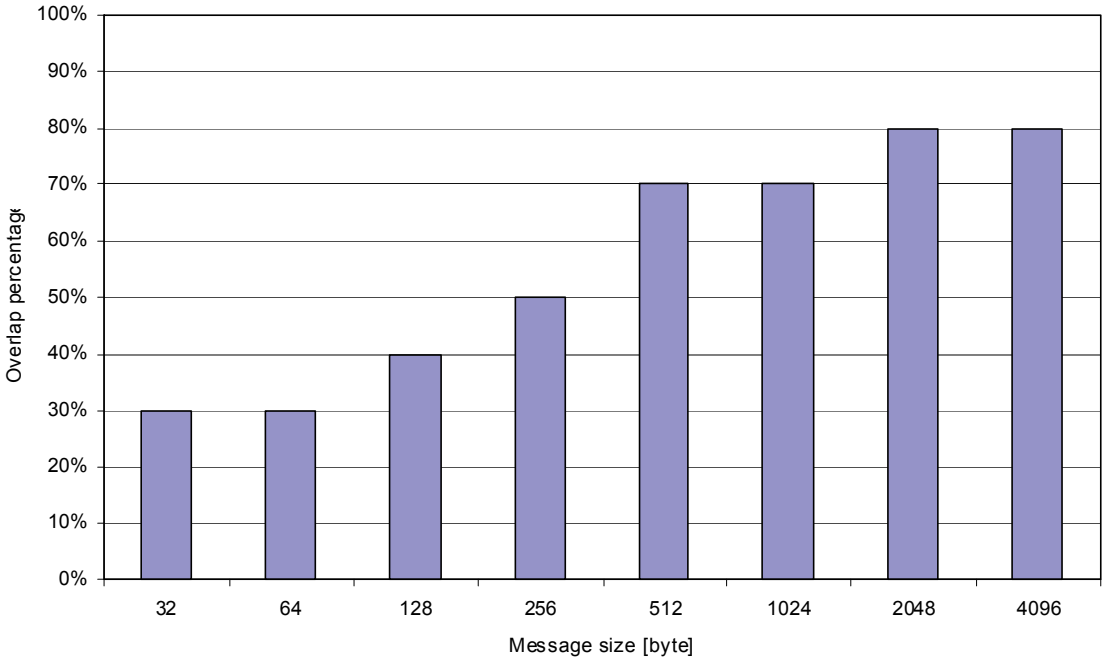


Figure 9-6: EXTOLL MPI-2 Overlap

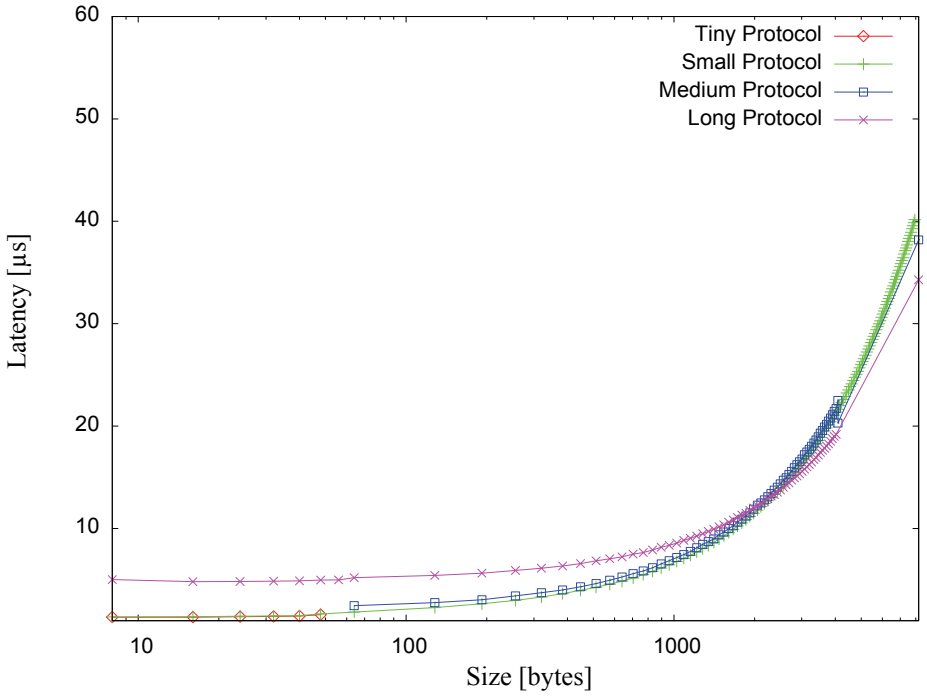


Figure 9-7: EXTOLL MPI Protocols: Ping-pong Latency

single VELO message to transport data, the *Small* protocol uses repeated VELO messaging and the *Medium* protocol is the two-copy protocol based on VELO and RMA *puts*, while the *Long* protocol uses the rendezvous zero-copy protocol. From the graph it is apparent that the *Long* protocol is slower than the other protocols for small messages, as expected due to the additional network round-trip. For very small messages, VELO in the *Tiny* or *Small* incarnation is the fastest method to communicate. As the message size grows, the *Medium* protocol tracks the *Small* protocol very closely. Even if the latency is a little bit higher, it may be advantageous to switch to the *Medium* protocol at about 512-1024 bytes, due to better buffer management, flow-control possibilities, and lower CPU load (DMA). The *Long* protocol reaches the *Medium* protocol at a size of 2 kB. A detailed view of the results acquired with messages from 1 to 3 kB is given in figure 9-8. The test does not take into account the cost for an additional registration for send and receive buffers that may be necessary. But it seems that 4 kB is a very good candidate to switch to the zero-copy protocol. While the latency may be higher sometimes, if one or both of the buffers are not registered beforehand, this will often not be the case because of buffer reuse. Additionally, the protocol removes load from the CPU and thus increases communication/computation overlap. A size of 4 kB is very small in comparison to traditional network solutions, where a zero-copy rendezvous protocol is so costly that it is often only used for message sizes of 32 kB and larger.

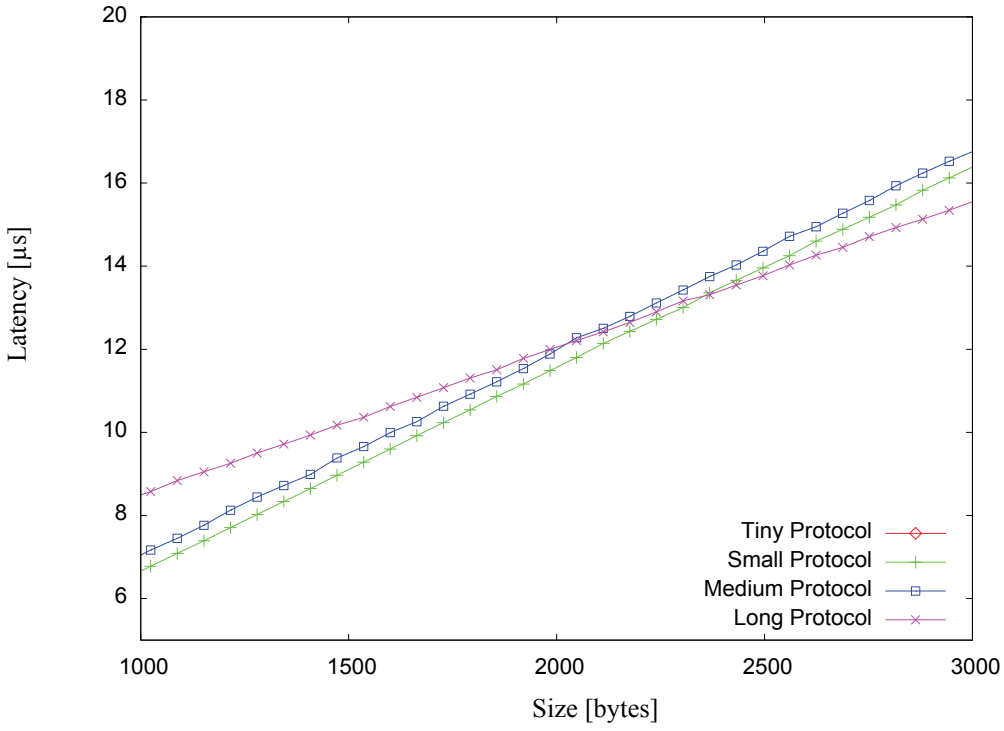


Figure 9-8: MPI-protocols: Detailed View

9.4 Summary of Results

This chapter gave a comprehensive evaluation of the EXTOLL hardware and software prototype. It could be shown that EXTOLL indeed provides an excellent communication latency across all tested operations. It was also pointed out that the architecture is able to deliver the bandwidth that it promises. Finally, benchmarking demonstrated that the architecture enables very efficient communication/computation overlap both in two-sided and one-sided applications.

In regard of other high-performance networks, the EXTOLL prototype either performs best or at least comparably in all categories except bandwidth. EXTOLL exhibits latencies that are much better than most networks and equal to the Connect X chip from Mellanox, the de-facto standard provider of IB HCA silicon and the most modern chip in the family of Mellanox HCAs. Measurements of Connect X show a ping-pong half-round trip latency of 1.3 µs on an Opteron based system equalling the EXTOLL latency exactly. The Qlogic Infinipath HTX adapter, which is well known for being the network adapter with the lowest send/receive latency, beats the start-up latency of EXTOLL with a value of 1.14 µs using ASIC technology and no switch; the EXTOLL prototype is only 160 ns slower but already includes a 6-port switching stage. Infinipath's latency including a switch is equal to VELO's and Mellanox Connect X.

The small message bandwidth of EXTOLL (8-byte) is 50 % higher than the best competitor as well as the message rate, though. EXTOLL also performs best when it comes to overlap and memory registration/deregistration. The bandwidth reachable in the EXTOLL prototype is severely restrained by area and frequency constraints. With the advent of FPGAs offering higher capacity or an ASIC, the link bandwidth of EXTOLL can easily be brought up to values of other networks (section 7.8). To the best of the knowledge of the author, EXTOLL is the best performing network for cluster computing ever constructed using FPGA technology.

Conclusion and Outlook

Chapter 10

The goal of this work was to improve and accelerate the hardware-software interface of a network interface controller to support distributed memory applications in a very efficient way. The main contributions of this work are focused around the problems of defining and implementing the necessary NIC hardware structures and host software components to facilitate minimum communication latency and high computation/communication overlap. The resulting hardware architecture was implemented using FPGA technology and the performance of the developed design was shown and evaluated in a real system. The excellent performance results gained show the acceleration of the NIC interface and prove the effectiveness of the contributions developed. An old English proverb says “the proof of the pudding is in the eating”; in computer architecture the proof is in actually implementing a new architecture in a real system, testing it for functionality and evaluating its performance. The software components plus the EXTOLL prototype hardware are the proof for the EXTOLL architecture and actually show that EXTOLL works and performs as expected. This result is much more reliable than a pure system level simulation can ever be.

The individual, architectural contributions of this work are the latency optimized, virtualized CFCQ and *Transactional I/O* methods, the novel address translation unit ATU, the EXTOLL units using these novel contributions, the software architecture to support all this hardware and finally the complete system putting all of these concepts into a complete, working prototype.

The *Transactional I/O* and CFCQ methods were introduced as means of supporting virtualized devices with a minimized latency. Methods to implement the receiver side in a virtualized fashion were also shown. The concept of state minimization as an important factor for latency minimization of a virtualized device was introduced. Using this method the VELO functional unit can reach microsecond application-to-application latencies using the FPGA based prototype.

The next contribution revolves around the question how to handle the virtual-to-physical address translation problem. It is necessary to solve this in hardware - the effects of the software approach using kernel level communication were also briefly shown in this thesis. The hardware solution has to follow design choices to minimize latency. The latency of an address translation is generally based on the number of memory accesses necessary. The developed ATU minimizes the main memory accesses necessary for an address translation. To further optimize latency a TLB was added to the architecture which lowers the communication latency in the average case significantly. The ATU unit as well as a simple TLB

were implemented for an FPGA architecture and integrated into the EXTOLL architecture to solve the address translation problem for EXTOLL. The evaluation later showed the efficiency of the ATU approach.

The EXTOLL RMA unit together with the described software protocols implements a method that combines extremely low latency and good computation/communication overlap characteristics. The careful design of the instruction set of RMA allows the use of the unit in a wide range of scenarios, from MPI-2 one-sided communication and MPI-1 point-to-point communication to PGAS-style communication. Advanced features like the FCAA remote atomic operation and fine-grain control over completion notification contribute to the effectiveness of the unit. The results also show, that the combination of RMA and ATU offers exceptionally low latency paired with good overlap and bandwidth characteristics. With its start-up latency of 1.3 µs the EXTOLL prototype performs already en-par with the best networks available and is the fastest FPGA based network for commodity clusters ever built.

On the software side, both kernel level and user level software were developed. Using this software it was possible to operate the hardware and perform the evaluation in a real system. The complete EXTOLL software written to support this work exceeds 15.000 lines of C-Code. It must be noted, that all performance numbers were gained using an FPGA implementation of the hardware model. The Verilog RTL code used is about ~85.000 lines in size. As described in section 7.8, an ASIC implementation easily increases the performance of EXTOLL by a factor of ~3 in latency and ~6 in bandwidth. Simulation based latency analysis and real system measurements (chapter 9) suggest that an EXTOLL ASIC will reach start-up latencies of less than 500 ns between two nodes which is only twice or thrice the latency of a cache miss in a modern server machine thus moving distributed memory nodes closer and closer together. An ASIC implementation also increases bandwidths both in the network and the host interface section by a factor of ~10 offering in the range of 20-30 Gb/s per link.

In summary, the goals formulated in the introduction of this thesis were all reached and a very promising architecture was developed and implemented. Still, a large amount of future work remains to be done. The implementation of an EXTOLL ASIC would enable new performance levels not reachable with traditional architectures. Full software implementations and optimizations of MPI-1, MPI-2 and GASnet are necessary on the middleware side of the software. Possibly the implementation of other middlewares and basic protocols could prove to be very interesting in terms of supporting more applications for example from the enterprise sector. Also, protocols to support remote storage and I/O operations could benefit greatly from the EXTOLL performance characteristics. Finally, a concise management software component for the EXTOLL system remains to be developed. On the architectural level several proposals for the EXTOLL network hardware architecture that have previously been discussed could not be included in the EXTOLL hardware prototype due to resource constraints of the FPGA platform. It is most interesting to add the implementation of these advanced networking features to the EXTOLL system, namely the EXTOLL Hardware Barrier, the multicast port and the High Availability Port (HAP). These features could

improve collective operations by a great deal and the HAP can be used to further increase reliability and serviceability features of EXTOLL beyond the hardware retransmission and data protection features already present in the prototype implementation.

The EXTOLL architecture can also lead the way to a more generalized architecture in the area of acceleration or coprocessing. Recent trends in computer architecture have brought back the concept of coprocessing or acceleration which can also be viewed as heterogeneous multi-processing. So, this concept is not quite the same as the one of the traditional floating point coprocessors from the 1980s; instead accelerators implement some complex function that is expensive to compute on a main processor core. Examples for accelerator hardware used today are GPUs, FPGA boards or boards featuring specialized processors like Clearspeed's [132]. In the accelerator area several problems exist: on the higher level the problem of hardware virtualization and hardware abstraction exists, on a lower level, software-hardware interfaces, hardware enumeration and configuration. There are proposals to tackle some of these problems, for example APIs to standardize the reconfiguration of an FPGA coprocessor in system [133].

The EXTOLL architecture can serve as a methodology or construction kit for a whole family of accelerators which are closely coupled to the CPU. This accelerator architecture can heavily leverage the experiences and results from the EXTOLL project. Many of the EXTOLL components can be re-used as building blocks for new accelerators and devices. The complete host interface (the HyperTransport core), the HTAX on-chip network, the EXTOLL register file methodology and the ATU unit can be used unaltered for such an accelerator. The CFCQ method for command management in the EXTOLL communication units and the notification strategy of the EXTOLL RMA units can also be included in the FUs of other devices. Besides accelerators in the network and computational area, it is also feasible to employ EXTOLL architectural concepts to other I/O accelerators, for example for storage devices.

Finally, with a generalized accelerator architecture, the proposed *Transactional I/O* can find a broader application than a single network controller. *Transactional I/O* as introduced in section 6.4.2 promises to be a very efficient technique to integrate a coprocessor into a system. In this area further research including system level simulation is necessary since then it becomes possible to integrate the new instruction and communication semantics into the main CPU of the system.

The EXTOLL prototype implementation will soon be evaluated in a greater network with more nodes enabling a better understanding of application scalability and behavior of the system. At the Computer Architecture Group of the University of Heidelberg, a 10-node cluster is being built and the Parallel Architectures Group of the Universidad Politécnica de Valencia is building a 64-node Opteron cluster featuring 1024 CPU cores and 64 HTX-Boards which will initially be shipped with an EXTOLL configuration.

Recent other developments include an HTX 3 board and HT3 IP efforts which may lead to EXTOLL implementation with vastly improved bandwidths even using FPGA technology. Other directions that could be valuable are the division of the EXTOLL host interface blocks and the network layer. The network layer can also be used in different special pur-

pose applications for example in high energy physics. The EXTOLL host interface block, main topic of this work, can also be combined with other networks to form novel NICs for that network. A prominent candidate for this approach is undoubtedly Infiniband. Finally, again, a complete EXTOLL ASIC will prove to be a veritable best-in-class network.

Graphical Representation and Methods

Appendix A

This thesis uses several graphical representation methods for visualization. These include:

- design space diagrams,
- sequence Diagrams,
- state diagrams,
- flow diagrams, and
- block diagrams.

The design space diagrams are based on [134]. Orthogonal aspects are visualized by perpendicular connections (figure A-1). Different design options use straight connectors. Additionally, items that are covered in-depth in the text are highlighted in blue (option 1 of aspect 2 in the example diagram).

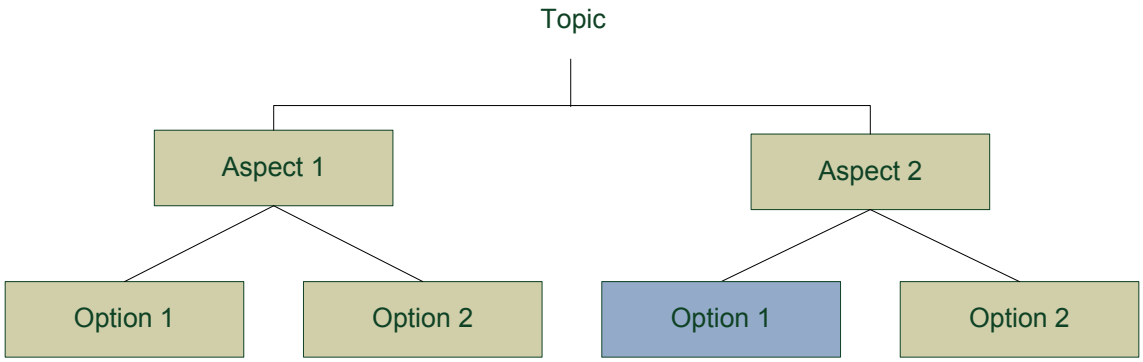


Figure A-1: Design Space Diagram

The sequence, state and flow-diagrams are closely referring to the *Unified Modeling Language* (UML) [135] standard diagrams. Sequence diagrams are used to show a protocol or sequences of events. An example diagram is shown in figure A-2. The acting entities are shown at the top, each featuring a timeline extending downwards. If one entity, for example a process or a CPU, sends a message to another entity, this is shown by a solid arrow (1). This could also be a function call, if a sequence in software is described. The return of the call or the response to a message is indicated by a dashed arrow (2). In this thesis, an arrow is sloped (3) if it exemplifies a message across a physical network, i.e. if this action involves sending of at least one network packet. Memory copy operations performed by hardware or

software entities are shown as curved arrows (4). No other active entity in the system is involved; memory is passive and was thus chosen to be not represented by an own entity. Finally, entities activated by a message and the state of being activated is depicted by vertical, yellow boxes. Generally, activation is started by an incoming message (arrow). An entity that started an action at a different location may be blocked or not blocked until the other entity responds (if ever). A typical example is a CPU that may be blocked by an access to a device or not. Entities that are blocked are shown through red boxes (5). Non-blocked entities, that may perform other tasks while another entity is active, are shown without box for this time (6).

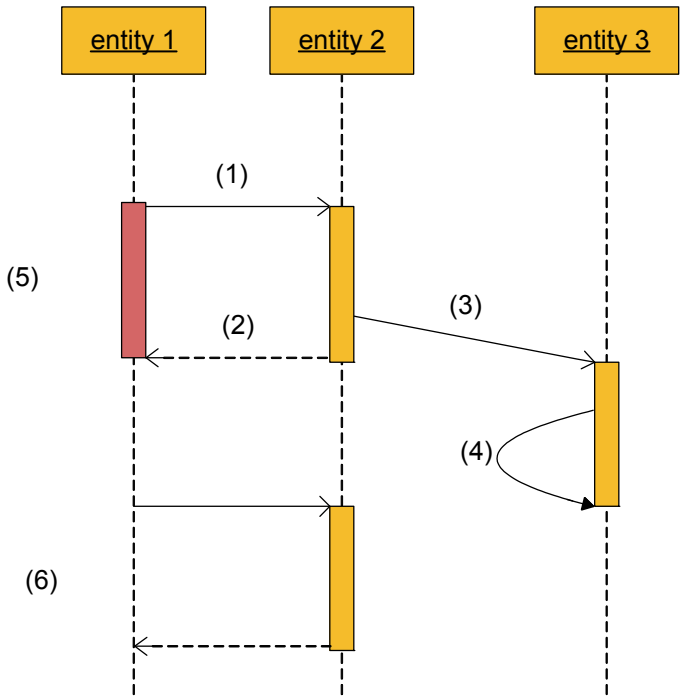


Figure A-2: Sequence Diagram

To analyze different design alternatives, three main methods were employed:

- Theoretical modeling/analysis,
- Empirical studies,
- Prototype implementation and analysis using FPGA technology.

For the analysis, literature and measurements of existing systems were taken as guides. Major hardware design choices were modeled in terms of latency and other important parameters using a set of empirically gained system performance parameters. The prototype system used for the measurements in this work was, if not mentioned otherwise, a pair of dual processor systems with dual Opteron 870 2.0 GHz processors, an IWILL DK8-HTX mainboard, 2 GB memory and SuSE 10.3 operating system. The FPGA platform used is a Xilinx part (Virtex4 FX100-11) on the CAG HTX-Board [56] (figure 3-3).

Acronyms

Appendix B

10-GE	10 Gigabit Ethernet
ADB	Allowable Disconnect Boundary (PCI-X)
ADI	Abstract Device Interface (MPICH)
AM	Active Messages
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ATC	Address Translation Cache
ATOLL	ATOmic Low Latency
ATS	Address Translation Service
ATU	Address Translation Unit
BAR	Base Address Region
BLAS	Basic Linear Algebra Subprograms
CAM	Content Addressable Memory
CAS	Compare-And-Swap
CFCQ	Central-Flow-Controlled Queues
Ch3	Channel Device (MPICH)
CMOS	Complementary Metal-Oxide-Semiconductor
CQ	Completion Queue
CSB	Conditional Store Buffer
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EXTOLL	Extended ATOLL
FAA	Fetch-And-Add

FCAA	Fetch-Compare-And-Add
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FU	Functional Unit
GART	Graphics Aperture Remapping Table
GE	Gigabit Ethernet
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HAP	High Availability Port
HCA	Host Channel Adapter
HDL	Hardware Description Language
HPC	High-Performance Computing
HPL	High-performance LinPACK
HT	HyperTransport
I^2C	Inter-Integrated Circuit
IB	Infiniband
ICM	InfiniHost Context Memory
IOMMU	I/O Memory Management Unit
IP	Internet Protocol
IPC	Interprocess Communication
MPI	Message Passing Interface
MPICH	an MPI implementation
MPICH2	an MPI-1 and 2 implementation
MTT	Memory Translation Table (IB) or Message Type Tag (VELO)
MTU	Maximum Transfer Unit
MVAPICH	MPICH derivative for Verbs based networks
NIC	Network Interface Controller
NLA	Network Logical Address
NUMA	Non-Uniform Memory Access
OFED	Open Fabrics Enterprise Distribution

OLTP	Online-Transaction Processing
OpenMPI	an MPI-1 and 2 implementation
OS	Operating System
PA	Physical Address
PCI	Peripheral Control Interface
PCI SIG	PCI Special Interest Group
PCIe	PCI Express
PCI-X	Peripheral Control Interface eXtended
PGAS	Partitioned Global Address Space
PIO	Programmed Input/Output
QP	Queue Pair
RDMA	Remote Direct Memory Access
RMA	Remote Memory Access or Remote Memory Architecture
RQ	Receive Queue
RTL	Register Transfer Level
RX	Receive
SAN	System Area Network
SDP	Sockets Direct Protocol
SFP	Small Form-factor Pluggable
SMP	Symmetric Multiprocessing
SPMD	Single Program - Multiple Data
SQ	Send Queue
SRAM	Static Random Access Memory
SRQ	Shared Receive Queue
TA	Translation Agent
TCE	Translation Control Entries
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
TOE	TCP Offload Engine
TX	Transmit

UDP	User Datagram Protocol
UPC	Universal Parallel C
VA	Virtual Address
VELO	Virtualized Engine for Low Overhead
VI	Virtual Interface
VIA	Virtual Interface Architecture
VIPL	Virtual Interface provider library
VPID	Virtual Process Identifier
WR	Work Request
XML	Extensible Markup Language

List of Figures

Appendix C

Figure 1-1: Limited Speed-up of Parallel Applications [2] 2
Figure 1-2: Virtual Interface Architecture 8
Figure 2-1: ATOLL Block-Diagram 16
Figure 2-2: ATOLL Software Environment 17
Figure 2-3: ATOLL Hostport Memory Layout 20
Figure 2-4: ATOLL Send Sequence Diagram 22
Figure 2-5: ATOLL Receive Sequence Diagram 24
Figure 2-6: AtollManager 28
Figure 2-7: Atollrun Initialization Protocol 29
Figure 2-8: MPICH2 CH3 Device with Sub-devices 31
Figure 2-9: MPICH2 Shared Memory Buffer and Message Format 32
Figure 2-10: ATOLL Link Test Scenario 35
Figure 2-11: ATOLL Link-Cable FPGA Board 36
Figure 2-12: Link Correction FSM 36
Figure 2-13: ATOLL Ping-Pong Latencies [12] 38
Figure 2-14: ATOLL Streaming Bandwidth [12] 39
Figure 2-15: High-Performance LinPACK 40
Figure 2-16: Histogram of MPICH Function Time Use 41
Figure 2-17: SWORDFISH Schematic Diagram [46] 42
Figure 2-18: SWORDFISH Hardware Model of One Node [46] 43
Figure 2-19: SWORDFISH (a) GUI and (b) Report [46] 44
Figure 2-20: SWORDFISH Blocking Rate in a 16x16 Mesh [52] 45
Figure 2-21: Deadlock Diagrams for a 16x16-Mesh (Locality) [52] 46
Figure 2-22: Deadlock Diagrams for a 16x16-Torus (Locality) [52] 47
Figure 3-1: Traditional System Architecture 52
Figure 3-2: Modern System Architecture 53
Figure 3-3: CAG HTX-Board 55
Figure 4-1: Basic Send/Receive Sequence 58
Figure 4-2: Remote Load and Store Operations 60
Figure 4-3: RDMA Operations 61
Figure 4-4: Zero-Copy Send/Recv using RDMA 62
Figure 4-5: Sockets Sequence 65
Figure 4-6: MPI One-Sided Communication with Fence 69
Figure 4-7: MPI Start/Complete/Post/Wait Synchronization 70

Figure 4-8: MPI Passive Target Synchronization 71
Figure 4-9: PGAS Address Space 73
Figure 4-10: PGAS: GASnet AM-style Messaging 74
Figure 4-11: Memory-Mapper Device Address Space 76
Figure 4-12: EXTOLL Two-Sided Communication Design Space 77
Figure 5-1: X86-64 Page-Table Walk 81
Figure 5-2: Mellanox Context and Translation Architecture [87] 84
Figure 5-3: GART Remapping 89
Figure 5-4: IBM Calgary IOMMU Architecture 92
Figure 5-5: AMD IOMMU System Architecture 93
Figure 5-6: IOMMU Data Structures 94
Figure 5-7: EXTOLL Address Translation Design Space Diagram 102
Figure 5-8: ATU Data Structures 110
Figure 5-9: ATU Address Translation Process 113
Figure 5-10: EXTOLL ATS Protocol Overview 114
Figure 5-11: ATU Register Interface 115
Figure 5-12: ATU Architecture: Block Diagram 117
Figure 5-13: ATU/Connect X Registration Latency 120
Figure 5-14: ATU/Connect X Deregistration Latency 120
Figure 5-15: Distribution of 16 kB Registration Latency 121
Figure 5-16: Distribution of 16 kB Deregistration Latency 121
Figure 6-1: SUN 10GE Adapter 126
Figure 6-2: Virtualization Design Space 128
Figure 6-3: Triggerpage Experiment 129
Figure 6-4: Central Flow-Controlled Queue Block Diagram 133
Figure 6-5: CFCQ 32-Byte Transaction Interrupt Rate 136
Figure 6-6: Design Space of Completion Notification 137
Figure 6-7: Queue Deadlock 138
Figure 7-1: EXTOLL Block Diagram 142
Figure 7-2: Registerfile Design Flow 145
Figure 7-4: VELO Requester Address Layout 147
Figure 7-3: VELO Sequence 147
Figure 7-5: VELO Completer Memory Layout 149
Figure 7-6: RMA Requester Address Encoding 152
Figure 7-7: RMA Requester Command Descriptor (Put/Get) [128] 152
Figure 7-8: RMA Put Requester Notification Descriptor [128] 153
Figure 7-9: RMA Lock Command Descriptor 158
Figure 7-10: RMA Lock Notification Descriptor 158
Figure 7-11: RMA Lock Addressing 159
Figure 7-12: RMA Block Diagram 161
Figure 7-13: Resource Usage by Block (PlanAhead Screenshot) 165
Figure 7-14: EXTOLL Placed and Routed on FX100 FPGA 165
Figure 7-15: EXTOLL Latency Breakdown and Scaling 166
Figure 8-1: EXTOLL Software Stack 170

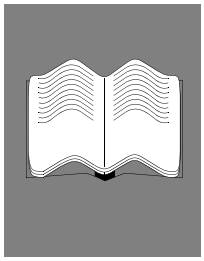
Figure 8-2: OpenMPI Architecture 175
Figure 8-3: Small-Message Layout 176
Figure 8-4: VELO-Based Medium Message Layout 177
Figure 8-5: RMA-Based Medium Size Protocol 178
Figure 8-6: Large-Message Request Layout 178
Figure 8-7: GASnet Request Layout 180
Figure 9-1: EXTOLL Ping-Pong Latencies 184
Figure 9-2: EXTOLL Start-up Latencies 184
Figure 9-3: EXTOLL Latency Details 185
Figure 9-4: EXTOLL Ping-Pong Bandwidth 187
Figure 9-5: EXTOLL Streaming Bandwidth 187
Figure 9-6: EXTOLL MPI-2 Overlap 189
Figure 9-7: EXTOLL MPI Protocols: Ping-pong Latency 189
Figure 9-8: MPI-protocols: Detailed View 190
Figure A-1: Design Space Diagram 197
Figure A-2: Sequence Diagram 198

List of Tables

Appendix D

Table 1-1: Overview of Networks 13
Table 3-1: Performance Parameters 54
Table 3-2: Performance Parameters (PCIe) 55
Table 5-1: AMD GART Registers 89
Table 5-2: HT3 ATS Translation Request 97
Table 5-3: HT3 ATS Translation Response 98
Table 5-4: HT3 ATS Translation Invalidation Request 99
Table 5-5: HT3 ATS Translation Invalidation Response 100
Table 5-6: Strategies of Context Handling in Page Tables 107
Table 5-8: ATU Latencies 118
Table 5-7: ATU Resource Usage 118
Table 7-1: VELO Registers 150
Table 7-2: RMA Instruction Set 154
Table 7-5: FCAA: MPI_Start/Post/Complete/Wait Operands 156
Table 7-3: FCAA: MPI_Lock/MPI_Unlock Operands 156
Table 7-4: FCAA: MPI_Lock/Unlock Encoding 156
Table 7-6: FCAA: MPI_Start/Post/Complete/Wait Encoding 157
Table 7-7: RMA Virtual vs. Physical Addressing 160
Table 7-8: RMA Registers 162
Table 7-9: Design Resource Usage 164
Table 7-10: Effects of Technology 167
Table 9-1: Sub-operation Composition of EXTOLL Transactions 186
Table 9-2: Sub-operation Timing 186
Table 9-3: EXTOLL Message Rates 188

References



[1] *TOP500 Supercomputer Sites*; http://www.top500.org, retrieved Dec. 2008

[2] U. Brüning; *Vorlesung Rechnerarchitektur 2*; Script, 2008 available from ra.ziti.uni-heidelberg.de; retrieved Sept. 2008

[3] *Intel Software Development Products*; http://www.intel.com/cd/software/products/asmo-na/eng/index.htm, retrieved Dec. 2008

[4] K. Yelick, D. Bonachea, W. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A Kamil, R. Nishtala, J. Su, M. Welcome, T. Wen; *Productivity and performance using partitioned global address space languages*, Proceedings of the 2007 International Workshop on Parallel Symbolic Computation, London, Canada, July 2007

[5] *DARPA High Productivity Computing Systems Program;* http://www.highproductivity.org/, retrieved Nov. 2008

[6] D. A. Patterson; *Latency lags Bandwidth*; Communications of the ACM, Vol. 47, No.10, Oct. 2004

[7] R. Srinivasan; *Achieving Mainframe-Class Performance on Intel Servers Using InfiniBand Building Blocks*; White Paper, Oracle, 2003

[8] M. Ronström, M. Ronström; *MySQL Cluster on Multi-Core Intel Xeon using Dolphin Express: Delivering 'Real-Time' Response to the database market*; White Paper, Dolphin Interconnect, 2007 available from www.dolphinics.com/solutions/wp-download.html; retrieved Sept. 2008

[9] J. Corbet, *KS2007: The customer panel*; LWN.net, http://lwn.net/Articles/248878/, retrieved July 2008

[10] U. Brüning, L. Schaelicke; A*TOLL: A High- Performance Communication Device for Parallel Systems*; In Proceedings of the 1997 Conference on Advances in Parallel and Distributed Computing, Shanghai, China, March 1997

[11] L. Rzymianowicz, U. Brüning, J. Kluge, P. Schulz, M. Waack; *ATOLL: A Network on a Chip*; Cluster Computing Technical Session (CC-TEA) of the PDPTA'99 conference, in Las Vegas, USA, June 1999

[12] 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

[13] HyperTransport Technology Forum; *HyperTransport™ I/O Link Specification;* Revision 3.00c; 2007

[14] D. Slogsnat, A. Giese, M. Nüssle, U. Brüning; *An open-source HyperTransport core*; ACM Transactions on Reconfigurable Technology; Syst. 1, 3, Article 14, September 2008

[15] H. Litz; *The HyperTransport Advanced Crossbar (HTAX)*; Internal Technical Documentation, 2008

[16] F. Ueltzhöffer; *Design and Implementation of a Virtual Channel Based Low-Latency Crossbar Switch*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2005

[17] B. Geib; *Improving and Extending a Crossbar Design for ASIC and FPGA Implementation*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2007

[18] N. Burkhardt; *Fast Hardware Barrier Synchronization for a Reliable Interconnection Network*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2007

[19] S. Schenk; *Architecture Analysis of Multi-Gigabit-Transceivers for Low Latency Communication*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2008

[20] S. Larsen, P. Sarangam, R. Huggahalli; *Architectural Breakdown of End-to-End Latency in a TCP/IP Network*; Proceedings of the 19th International Symposium on Computer Architecture and High Performance Computing, Gramado, Brazil, October, 2007

[21] Sun Inc.; *Unleashing 10 Gigabit Networks*; White Paper, March 2007

[22] Infiniband Trade Association; *InfiniBand Architecture Specification Volume 1*; Release 1.1, 2002

[23] J. Liu, W. Huang, B. Abali, D. K. Panda; *High Performance VMM-Bypass I/O in Virtual Machines*; USENIX'06, Boston, USA, 2006

[24] Intel, Compaq, Microsoft; V*irtual Interface Architecture Specification Version 1.0*; December 16, 1997, ftp://download.intel.com/design/servers/vi/VI_Arch_Specification10.pdf, retrieved August 2008

[25] W. E. Speight, H. Abdel-Shafi, J. K. Bennett; *Realizing the performance potential of the virtual interface architecture*; Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing (ICS'99), Rhodes, Greece, 1999

[26] *M-Via - A High performance Modular VIA for Linux*; http://www.nersc.gov/research/FTG/via/, retrieved August 2008

[27] *OpenFabrics Alliance*; www.openfabrics.org

[28] *MVAPICH MPI*; http://mvapich.cse.ohio-state.edu, retrieved Dec. 2008

[29] R. Brightwell, D. Doerfler, K. D. Underwood; *A Preliminary Analysis of the InfiniPath and XD1 Network Interfaces;* Parallel and Distributed Processing Symposium (IPDPS), Rhodes, Greece, 2006

[30] L. Dickman, G. Lindahl, D. Olson, J. Rubin, J. Broughton; *PathScale InfiniPath: A First Look*; Proceedings of the 13th Symposium on High Performance Interconnects, Stanford, USA, 2005

[31] J. Hilland, P. Culley, J. P, R. Recio; *RDMA Protocol Verbs Specification (Version 1.0)*; 2003, available from www.rdmaconsortium.org/home/draft-hilland-iwarp-verbs-v1.0-RDMAC.pdf, retrieved Sept. 2008

[32] M. J. Rashti, A. Afsahi; *10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G*; IEEE International Parallel and Distributed Processing Symposium, Long Beach, USA, March 2007

[33] N.J. Boden, D. Cohen, R. E. Felderman,A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W. Su; *Myrinet: A Gigabit-per second Local Area Network*; In IEEE Micro, 15(1):29-36, 1995

[34] F. Petrini, W. Feng, A. Hoisie, S. Coll, E. Frachtenberg; *The Quadrics Network: High-Performance Clustering Technology*; In IEEE Micro, Volume 22, Issue 1, 2002

[35] L. Rzymianowicz; *Designing Efficient Network Interfaces for System Area Networks*; Ph.D. Thesis, CSE, University of Mannheim, 2002

[36] *ATOLL Link Chip - Hardware Reference Manual;* Internal Documentation, Computer Architecture Group at the Department of Computer Engineering, University of Mannheim, 2003

[37] G. V. Vaughan, B. Elliston, T. Tromey; *Gnu Autoconf, Automake, and Libtool*; New Riders Publishing, 2000

[38] *Doxygen Project*; www.doxygen.org

[39] T. Hettinger; *Design and Implementation of Efficient and Reliable Network Protocols for the ATOLL System Area Network;* Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2004

[40] M. Nuessle; *Design and Implementation of a distributed management system for the ATOLL high-performance network*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2003

[41] D. Franger; *Software Implementation of the I2C Protocol on the ATOLL Network Card to access an EEPROM Device and a Temperature Sensor*; Project Report, University of Mannheim, 2004

[42] *MPICH2*; http://www.mcs.anl.gov/research/projects/mpich2/; retrieved August 2008

[43] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall; O*pen MPI: Goals, Concept, and Design of a Next Generation MPI Implementation*; Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro-PVM/MPI04), Budapest, Hungary, 2004

[44] B. Strohmeier; *Design and implementation of a high speed serializing multipurpose interconnect with improved testability as a key aspect*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2003

[45] P. Bastian, K. Birken, K. Johannsen, S. Lang, V. Reichenberger, C. Wieners, G. Wittum, C. Wrobel; *A parallel software-platform for solving problems of partial differential equations using unstructured grids and adaptive multigrid methods*; In W. Jäger and E. Krause (ed): High performance computing in science and engineering, pages 326--339. Springer, 1999.

[46] M. Nüssle, H. Fröning, U. Brüning; *SWORDFISH: A Simulator for High-Performance Networks*; IASTED Conference: Parallel and Distributed Computing and Systems (PDCS), Phoenix, USA, Nov. 2005

[47] Q. O. Snell, A. Mikler, J. L. Gustafson; *NetPIPE: A Network Protocol Independent Performance Evaluator*; Proceedings of the IASTED International Conference on Intelligent Information Management and Systems, Washington D.C., USA, 1996

[48] *Intel MPI Benchmarks (IMB)*; available from http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm; retrieved Sept. 2008

[49] R. Reussner, P. Sanders, J. L. Träff; *SKaMPI: a comprehensive benchmark for public benchmarking of MPI*; Scientific Programming Vol. 10, Issue 1, Jan. 2002

[50] J. J. Dongarra, P. Luszczek, A. Petitet; *The linpack benchmark: Past, present, and future*; Concurrency and Computation: Practice and Experience, Vol. 15, p. 820, 2003

[51] W. J. Dally, C. L. Seitz; *Deadlock-Free Message Routing in Multiprocessor Interconnection Networks*; IEEE Transactions on Computers, vol.C-36, no.5, pp.547-553, May 1987

[52] H. Sattel; *A Scalable Generic Wormhole-Routing Simulator - SWORDFISH*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2004

[53] R. Sohnius; *Creating an Executable Specification Using SystemC of a High Performance, Low Latency Multilevel Network Router*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2005

[54] T. X. Jakob; *Multilevel Optimization of Parallel Applications Utilizing a System Area Network*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2002

[55] D. Slogsnat; *Tightly-Coupled and Fault-Tolerant Communication in Parallel Systems;* Dissertation, Ph.D. Thesis, CSE, University of Mannheim, 2008

[56] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, U. Brüning; *The HTX-Board: A Rapid Prototyping Station*; 3rd annual FPGAworld Conference, Stockholm, Sweden, Nov. 2006

[57] D. Slogsnat, A. Giese, M. Nüssle, U. Brüning; *An Open-Source HyperTransport Core*; ACM Transactions on Reconfigurable Technology and Systems (TRETS), Vol. 1, Issue 3, p. 1-21, Sept. 2008

[58] I. Kuon, J. Rose; *Measuring the gap between FPGAs and ASICs*; In Proceedings of the 14th international Symposium on Field Programmable Gate Arrays, Monterey, USA, Feb. 2006

[59] Message Passing Interface Forum; *MPI: A Message-Passing Interface Standard Version 1.3*; 2008; available from http://www.mpi-forum.org/docs/docs.html; retrieved Sept. 2008

[60] K. D. Underwood, K. S. Hemmert, A. Rodrigues, R. Murphy, R. Brightwell; *A Hardware Acceleration Unit for MPI Queue Processing*; Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, Denver, USA, April 2005

[61] Intel Inc.; *Server Network I/O Acceleration*; White Paper, 2004, http://download.intel.com/technology/comms/perfnet/download/ServerNetworkIOAccel.pdf, retrieved August 2008

[62] J. Regula; *Using Non-transparent Bridging in PCI Express Systems*; White Paper, PLX Technology Inc., June 2004

[63] R. Kota; R. Oehler; *Horus: large-scale symmetric multiprocessing for Opteron systems*; IEEE Micro, Volume 25, Issue 2, March 2005

[64] J. L. Hennessy, D. A. Patterson, David Goldberg; *Computer Architecture: A Quantitative Approach, Third Edition*; Morgan Kaufmann, 2002

[65] R. Noronha, L. Chai, T. Talpey, D. K. Panda; *Designing NFS with RDMA for Security, Performance and Scalability*; International Conference on Parallel Processing, XiAn, China, Sept. 2007

[66] M. Ko, M. Chadalapaka, U. Elzur, H. Shah, P. Thaler, J. Hufferd; *iSCSI Extensions for RDMA Specification*; Internet Standard RFC 5046, available from http://www.ietf.org/rfc/rfc5046.txt. retrieved 08/2008

[67] K. Magoutis, S. Addetia, A. Fedorova, M. I. Seltzer; *Making the Most out of Direct-Access Network Attached Storage*; In Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03), San Francisco, USA, March 2003.

[68] M. Rangarajan, L. Iftode; *Building a User-level Direct Access File System over Infiniband*; Proceedings of the 4th Annual Workshop on System Area Networks (SAN-4), Madrid, Spain, Feb. 2004

[69] R. Stevens; *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*; Prentice Hall, 1998

[70] M. Fischer; *Efficient and Innovative Communication Mechanisms for System Area Networks*; Dissertation, Ph.D. Thesis, CSE, University of Mannheim, 2002

[71] D. Goldenberg, M. Kagan, R. Ravid, M. S. Tsirkin; *Zero Copy Sockets Direct Protocol over InfiniBand - Preliminary Implementation and Performance Analysis*; Proceedings of the 13th Symposium on High Performance Interconnects, Stanford, USA, Aug. 2005

[72] Message Passing Interface Forum; *MPI-2: Extensions to the Message-Passing Interface*; http://www.mpi-forum.org/docs/mpi2-report.pdf; 2003; retrieved August 2008

[73] G. Santhanaraman, S. Narravula, D. K. Panda; *Designing passive synchronization for MPI-2 one-sided communication to maximize overlap*; IEEE International Symposium on Parallel and Distributed Processing IPDPS, Miami, USA, 2008

[74] R. Thakur, W. Gropp, Brian Toonen; *Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication*; Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2004), Budapest, Hungary, Sept. 2004

[75] UPC Consortium; *UPC Language Specifications V1.2*; May 31, 2005

[76] P. Hilfinger, D. Bonachea, K. Datta, D. Gay, S. Graham, B. Liblit, G. Pike, J. Su, K. Yelick; *Titanium Language Reference Manual*; U.C. Berkeley Tech Report, UCB/EECS-2005-15, 2005

[77] R. W. Numrich, J. Reid; *Co-array Fortran for parallel programming*; SIGPLAN Fortran Forum, Volume 17, Issue 2, Aug. 1998

[78] E. Allen et al.; *The Fortress Language Specification, Version 1.0*; March 31, 2008; http://research.sun.com/projects/plrg/Publications/index.html, retrieved August 2008

[79] V. Saraswat, N. Nystrom; *Report on the Experimental Language X10, Version 1.7*; June 18, 2008; http://x10.sourceforge.net/x10doc.shtml, retrieved August 2008

[80] D. Bonachea; *GASNet specification, v1.1*; Technical Report UCB/CSD-02-1207, U.C. Berkeley, October 2002.

[81] A. M. Mainwaring, D. E. Culler; *Active Messages: Organization and Applications Programming Interface*; Technical Document, University of Berkeley, 1995; available from http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps, retrieved Sept. 2008

[82] C. Bell, D. Bonachea; *A New DMA Registration Strategy for Pinning-Based High; Performance Networks*; Workshop on Communication Architecture for Clusters (CAC'03), Nice, France, 2003

[83] I. Schoinas, M. D. Hill; *Address Translation Mechanisms in Network Interfaces*; Proceedings of the Fourth International Symposium on High-Performance Computer Architecture, Las Vegas, USA, Feb. 1998

[84] AMD Inc.; *Software Optimization Guide for AMD Family 10h Processors*; Revision 3.06, April 2008

[85] R. Bhargava, B. Serebrin, F. Spadini, S. Manne; *Accelerating two-dimensional page walks for virtualized systems*; ACM Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, UAS, Mar. 2008

[86] J. E.J. Bottomley; *Dynamic DMA mapping using the generic device interface*; Linux Kernel documentation, version 2.6.25, retrieved July 2008

[87] D. Goldstein; *MemFree Technology*; Presentation, OpenIB Workshop, Sonoma, USA, Feb. 2005

[88] Mellanox Technologies; *InfiniHost III Ex MemFree Mode Performance*; White Paper, retrieved July 2008

[89] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, W. Rehm; *Analysis of the Memory Registration Process in the Mellanox Infiniband Software Stack*; European Conference on Parallel Computing (EURO-PAR), Dresden, Germany, Aug. 2006

[90] L. Ou, X. He, J. Han; *A Fast Read/Write Process to Reduce RDMA Communication Latency*; Proceedings of the 2006 International Workshop on Networking, Architecture, and Storages, Shenyang, China, Aug. 2006

[91] T. S. Woodall, G. M. Shipman, G. Bosilca, A. B. Maccabe; *High Performance RDMA Protocols in HPC*; Proceedings of the 13th European PVM/MPI Users' Group Meeting (Euro PVM/MPI 2006), Bonn, Germany, Sept. 2006

[92] F. Mietke, R. Rex, T. Mehlan, T. Hoefler, W. Rehm; *Reducing the Impact of memory registration Infiniband*; 1. Workshop Kommunikation in Clusterrechnern und Clusterverbundsystemen (KiCC), Chemnitz, Germany, Nov. 2005

[93] D. K. Panda et al.; *Performance Comparison of MPI Implementations over Infiniband, Myrinet and Quadrics*; Proceedings of the International Conference on Supercomputing (SC'03), Phoenix, USA, 2003

[94] R. Recio, P. Culley, D. Garcia, J. Hilland; *An RDMA Protocol Specification*; 2002, www.rdmaconsortium.org, retrieved July 2008

[95] Quadrics; *Quadrics Linux Kernel Integration*; Website, http://web1.quadrics.com/downloads/ReleaseDocs/hawk/LinuxKernelIntegration.html, retrieved July 2008

[96] Myricom Inc.; *Myrinet Express (MX): A High-Performance, Low-Level Message-Passing Interface for Myrinet*; version. 1.2, 2006

[97] B. Goglin; *Design and Implementation of Open-MX: High-Performance Message Passing over generic Ethernet hardware*; IEEE International Parallel and Distributed Processing Symposium, Rome, Italy, May 008

[98] L. C. Stewart, D. Gingold; *A New Generation of Cluster Interconnect*; White Paper, SciCortex, Dec. 2006/revised Apr. 2008

[99] Intel; *AGP V3.0 Interface Specification*; Sept. 2002

[100] AMD Inc.; *AMD64 BIOS and Kernel Developer's Guide for AMD Family 10h Processors*; Revision 3.06, Sept. 2008

[101] K. D. Underwood, K. S. Hemmert, C. Ulmer; *Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications*; Proceedings of the International Conference on Supercomputing (SC'06),Tampa, USA, Nov. 2006

[102] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, L. Van Doorn; *The Price of Safety: Evaluating IOMMU Performance*; Proceedings of the Linux Symposium, Ottawa, Canada, Jun. 2007

[103] AMD Inc.; *AMD I/O Virtualization Technology (IOMMU) Specification*; version 1.2, Feb. 2007

[104] D. Abramson et al.; *Intel Virtualization Technology for Directed I/O*; Intel Technology Journal, Volume 10, Issue 3, 2006

[105] PCI-SIG; *Address Translation Services*; Revision 1.0, Mar. 8, 2007

[106] J. Corbert; *Supporting RDMA on Linux*; LWN.net, 26. Apr. 2005, retrieved July 2008

[107] T. Schlichter; *Exploration of hard- and software requirements for one-sided, zero copy user level communication and its implementation*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2003

[108] C. Lameter; *Bazillions of Pages*; Proceedings of the 2008 Ottawa Linux Symposium, Ottawa, Canada, May 2008

[109] F. Rembor; *Exploration, Development and Implementation of different TLB Functions and Mechanisms*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2006

[110] M. Kunst; *A Unified Multi Context Networking Engine*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2007

[111] *CSwitch*; www.cswitch.com, retrieved July 2008

[112] F. Lemke; *FSMDesigner4 - Development of a Tool for Interactive Design and Hardware Description Language Generation of Finite State Machines*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2006

[113] AMD Inc.; *HyperTransport™ 3.0 Bus Functional Model User Guide;* Revision 0.20, 2007

[114] M. Herlihy, J. Eliot, B. Moss; *Transactional Memory: Architectural Support For Lock-free Data Structures*; Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, USA, May 1993

[115] H. Fröning; *Architectural Improvements of Interconnection Network Interfaces*; Dissertation, Ph.D. Thesis, CSE, University of Mannheim, 2007

[116] L. Schaelicke, A. Davis; *Improving I/O performance with a conditional store buffer*; Proceedings of the 31st Annual ACM/IEEE international Symposium on Microarchitecture, Dallas, USA, 1998

[117] Xilinx Inc.; *ChipScope Pro 10.1 Software and Cores User Guide;* version 10.1, 2008

[118] M. Nüssle, H. Fröning, A. Giese, H. Litz, D. Slogsnat, U. Brüning; *A Hypertransport based low-latency reconfigurable testbed for message-passing developments*; 2.Workshop Kommunikation in Clusterrechnern und Clusterverbundsystemen (KiCC'07), Chemnitz, Germany, Feb. 2007

[119] PCI-SIG; *PCI Express Base Specification*; Revision 1.0, 2002

[120] T. Reubold; D*esign, Implementation and Verification of a PCI Express to HyperTransport Protocol Bridge*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2008

[121] Denali Software Inc.; *SystemRDL 2.6A: Standard for System Register Description Language (SytemRDL)*; 2007

[122] Denali Software Inc.; *Blueprint Compiler Users Guide;* version 3.7.2, 2008

[123] *EXTOLL Reference Manual*; Internal Documentation, CAG, University of Heidelberg, 2008

[124] P. Mochel; *The sysfs Filesystem*; Proceedings of the 2005 Linux Symposium, Ottawa, Canada, Jul. 2005

[125] *Subversion - an open-source version control system*; http://subversion.tigris.org/; retrieved Sept.2008

[126] C. Leber; *A hardware-oriented simulator for high performance interconnection network architectures*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2007

[127] H. Litz, H. Fröning, M. Nüssle, U. Brüning; *VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers*; 37th International Conference on Parallel Processing (ICPP-08), Portland, USA, Sept. 2008

[128] M. Scherer; *Implementation, Synthesis and Verification of a Remote Shared Memory Access Functional Unit*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2008

[129] D. Franger; *A Multi-Context Engine for Remote Memory Access to Improve System Area Networking*; Diploma Thesis presented to the Computer Engineering Department, University of Mannheim, 2004

[130] Xilinx Inc.; *Xilinx ISE 10.1 Design Suite Software Manuals and Help*; version 10.1, 2008

[131] Xilinx Inc.; *PlanAhead User Guide*; version 10.1, 2008

[132] Clearspeed Technology plc; *Clearspeed Advance e710 Accelerator*; available from www.clearspeed.com, retrieved Dec. 2008

[133] OpenFGPA Inc.; *OpenFPGA GenAPI version 0.4 Draft Specification*; available from http://www.openfpga.org/, retrieved Sept. 2008

[134] D. Sima, T. Fountain, P. Kacsuk; *Advanced Computer Architectures: A Design Space Approach*; Addison Wesley, 1997

[135] M. Fowler, K. Scott; *UML Distilled Second Edition A Brief Guide to the Standard Object Modelling Language*; Addison Wesley, 1999