


Hardware Support for Efficient Packet Processing



Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Dipl.-Inf. Benjamin Ulrich Geib
aus Heidelberg

Mannheim, 2012

Dekan: Prof. Dr. Heinz Jürgen Müller, Universität Mannheim
Referent: Prof. Dr. Ulrich Brüning, Universität Heidelberg
Korreferent: Prof. Dr. Reinhard Männer, Universität Heidelberg

Tag der mündlichen Prüfung: 21.03.2012

Für Sonja

Abstract

Scalability is the key ingredient to further increase the performance of today's supercomputers. As other approaches like frequency scaling reach their limits, parallelization is the only feasible way to further improve the performance. The time required for communication needs to be kept as small as possible to increase the scalability, in order to be able to further parallelize such systems.

In the first part of this thesis ways to reduce the inflicted latency in packet based interconnection networks are analyzed and several new architectural solutions are proposed to solve these issues. These solutions have been tested and proven in a field programmable gate array (FPGA) environment. In addition, a hardware (HW) structure is presented that enables low latency packet processing for financial markets.

The second part and the main contribution of this thesis is the newly designed crossbar architecture. It introduces a novel way to integrate the ability to multicast in a crossbar design. Furthermore, an efficient implementation of adaptive routing to reduce the congestion vulnerability in packet based interconnection networks is shown. The low latency of the design is demonstrated through simulation and its scalability is proven with synthesis results.

The third part concentrates on the improvements and modifications made to EXTOLL, a high performance interconnection network specifically designed for low latency and high throughput applications. Contributions are modules enabling an efficient integration of multiple host interfaces as well as the integration of the on-chip interconnect. Additionally, some of the already existing functionality has been revised and improved to reach better performance and a lower latency. Micro-benchmark results are presented to underline the contribution of the made modifications.

Zusammenfassung

Der wichtigste Faktor, um die Leistung heutiger Supercomputer weiter steigern zu können, ist Skalierbarkeit. Da andere Ansätze, wie zum Beispiel Frequenzskalierung, ihre Grenzen erreicht haben, ist Parallelisierung der einzig mögliche Weg um die Leistung solcher Systeme weiter steigern zu können. Damit ein System skalierbar ist, muss die Zeit, welche für die Kommunikation zwischen einzelnen Knoten benötigt wird, so klein wie möglich gehalten werden.

Der erste Teil dieser Dissertation analysiert Möglichkeiten und zeigt neue architektonische Lösungen, welche die Latenz reduzieren und somit die Skalierbarkeit verbessern. Die Wirksamkeit dieser Lösungen wurden in einer FPGA (Field Programmable Gate Array) Umgebung getestet und bewiesen. Zudem wird eine Hardware-Struktur vorgestellt, welche das Dekodieren von Datenpaketen, wie sie beim Wertpapierhandel an Finanzmärkten verwendet werden, mit besonders niedriger Latenz und hohem Durchsatz ermöglicht.

Den zweiten und Hauptbestandteil dieser Dissertation stellt die neu entworfene Crossbar Architektur dar. Darin wird eine Lösung vorgestellt, wie die Fähigkeit Multicasts in einem Netzwerk zu senden, effizient in einen Crossbar integriert werden kann. Des Weiteren wird eine effiziente Implementierung von adaptivem Routing dargelegt, welche die Empfindlichkeit gegenüber Blockierungen in paketbasierten Netzwerken reduziert. Die niedrige Latenz des Designs wird durch Simulation und seine Skalierbarkeit durch Syntheseergebnisse gezeigt.

Der dritte Teil konzentriert sich auf die Verbesserungen und Erweiterungen, die an Extended ATOLL (EXTOLL) gemacht wurden. EXTOLL ist ein Hochgeschwindigkeitsnetzwerk, das speziell für Anwendungen mit den Anforderungen niedrige Latenz und hoher Durchsatz entwickelt wurde. Im Rahmen dieser Arbeit wurden Module zur effizienten Integration mehrerer Host Interfaces sowie des On-Chip Netzwerks entworfen und implementiert. Zudem wurde bereits existierende Funktionalität nochmals durchleuchtet und verbessert, um höhere Performanzwerte und eine niedrigere Latenz zu erreichen. Um den Beitrag der gemachten Veränderungen zu unterstreichen, werden Micro-Benchmark Ergebnisse präsentiert.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Graphical Representations	4
1.3	Outline	5
2	Efficient Packet Processing	7
2.1	Arbiter Generator	9
2.1.1	State of the Art	10
2.1.2	Implementation	11
2.2	FIFO Implementations	12
2.2.1	Multi-Queue FIFO	14
2.2.2	Speculative FIFO	17
2.2.3	Double Shift-Out FIFO	20
2.3	Write Combining Buffer	22
2.4	Virtual Ring Buffer Handler	24
2.4.1	State of the Art	26
2.4.2	Design Space Analysis	26
2.4.3	Implementation	28
2.4.4	Performance	33
2.5	Tag Matching Unit	33
2.5.1	State of the Art	34
2.5.2	Implementation	35
2.5.3	General Operation Description	41
2.6	FAST Decoder	42
2.6.1	FAST Protocol	43
2.6.2	Related Work	46
2.6.3	Design Space Analysis	46
2.6.4	Baseline Implementation	48
2.6.5	Baseline Performance	52
2.6.6	FAST Decoder Improvements	54

3	High Performance Switching	63
3.1	Switching Elements	63
3.1.1	Time Division Switching Elements	63
3.1.2	Space Division Switching Elements	65
3.2	Crossbars as Interconnect Switches	66
3.2.1	Crossbar Buffering	66
3.2.2	Crossbar Scheduling	68
3.3	Topologies	69
3.4	Terminology	70
3.4.1	Deadlocks	70
3.4.2	Virtual Channels	71
3.4.3	Head-of-Line Blocking	72
3.4.4	Virtual Output Queuing	72
3.4.5	Credit Based Flow Control	73
3.4.6	Wormhole Switching	74
3.4.7	Virtual-Cut-Through Switching	74
3.4.8	Adaptive Routing	74
3.5	Requirements for an Interconnection Network	75
3.6	State of the Art	76
3.6.1	Cray Gemini	76
3.6.2	Tofu	78
3.6.3	TianHE-1A Interconnect	80
3.6.4	Blue Gene/Q	80
3.7	State of EXTOLL R1 Crossbar	82
3.7.1	Packet Format	85
3.7.2	Routing	86
3.7.3	Architecture	87
3.7.4	Performance	90
3.8	Implementation of R2 Crossbar	93
3.8.1	Routing	93
3.8.2	Ordering of Packets	99
3.8.3	Packet Format	101
3.8.4	Multicast	102
3.8.5	Request Allocation and Grant Generation	107
3.8.6	Fine Grain Credits	108
3.8.7	Debug-Ability and Maintainability	111
3.8.8	Early Arbitration	112
3.8.9	Further Optimizations	113
3.8.10	Multi-Queue-FIFO Optimizations	116

3.8.11	Overall Architecture	120
3.8.12	Evaluation	121
3.9	Outlook and Future Improvements	125
3.9.1	Multicast Optimization	125
3.9.2	Read Slot Lookahead	126
3.9.3	Removal of Erroneous Packets	127
3.9.4	Balance Packet Travel Time	129
3.9.5	Split Inport Into Smaller Units	130
3.10	Crossbar Summary	130
4	EXTOLL	133
4.1	HW Overview	134
4.1.1	Host Interface	135
4.1.2	HTAX	137
4.1.3	Register File	138
4.1.4	ATU	139
4.1.5	RMA	139
4.1.6	SMFU	139
4.1.7	LP	140
4.2	Crossbar	141
4.3	HTAX Bridge	141
4.3.1	Design Space Analysis	142
4.3.2	Implementation	144
4.4	PCI Express Bridge	147
4.4.1	Requirements	148
4.4.2	Implementation	149
4.5	Network Port	151
4.6	VELO	153
4.6.1	Introduction	154
4.6.2	VELO Evaluation	156
4.6.3	VELO Requester R2	158
4.6.4	VELO Completer R2	161
4.6.5	VELO Performance	166
5	Conclusion	169
	Acronyms	175
	Bibliography	179

Contents

List of Figures	191
List of Tables	195

1 Introduction

Researches always have the need for more computing power. To satisfy this thirst, a great amount of work is done trying to increase the performance of supercomputers. This performance is measured in floating point operations per second (FLOPS) using a standard benchmark called *Linpack* [1]. Today's fastest supercomputer, Japan's *K-Computer* [2], is capable of performing around 10 Peta-FLOPS, a ten with 16 zeros. The next big step for such machines is to reach the *exascale era*.

1.1 Motivation

Supercomputers need to be a hundred times faster as the *K-Computer* in order to achieve an Exa-FLOP. It is unfeasible to further increase the frequency with which the individual processors run as they hit the power wall, because the power consumption increases quadratically with increasing frequencies. Thus, the only solution to increase the overall performance is seen in parallelization.

While increasing the number of processors is a valid approach, the performance does not scale with the same rate due to limiting factors. These limiting factors are again power-, area consumption and most importantly communication overhead. It is common sense that with an increasing amount of processors, both power and area consumption increase linear. Increasing the performance of a supercomputer, equipped with currently available technology, by a factor of hundred thus increases the power and area consumption by at least the same factor. While the two problems power and area can be solved with pure force, the limiting factor communication overhead remains. When parallelizing a given problem to solve it on a large cluster of processing nodes, the problem size that each individual node has to process decreases in the same degree. Consequently the time required to calculate the result also decreases. In an ideal environment with a problem that can be partitioned into an endless number of sub-problems and without any communication overhead, the performance increases linear with the parallelization degree.

1 Introduction

However, communication overhead between the nodes increases as the problem has to be transferred to all nodes. Furthermore each node needs to synchronize their results more often with other nodes. The time required for this communication overhead unfortunately does not decrease for smaller sub-problem sizes and due to the increased amount of messages the overall time for communication increases significantly. At a certain degree of parallelization the time spent for communication is larger than the time spent for actual computation. This leads to an overall performance drop as it is illustrated in figure 1.1. This effect is known as *Amdahls Law* [3].

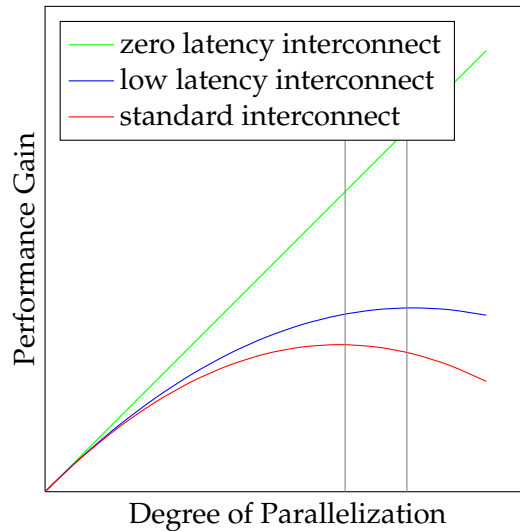


Figure 1.1: Performance Gain when Parallelizing a Computational Task [3]

To reduce the amount of time spent for communication is therefore an important research topic. There are three criteria to characterize the performance of an interconnection network. *Bandwidth* or *throughput* is the most common term to describe the performance of networks in high performance computing (HPC) and describes the amount of data that can pass through the network in a period of time. *Latency* is the time required to send a minimum sized packet from one node to another including all software (SW) and hardware (HW) overhead. The *message rate* describes the amount of messages that can be injected into or received from the network by a single node in a period of time, effectively describing how good the processing of independent messages can be overlapped. The higher the overlapping, the more messages can be processed in a certain period of time.

Latency is an important factor due to the above mentioned percental increase in time spent for communication compared to computation when parallelizing tasks. The lower the latency, the less time is spent in communication. Furthermore the scalability of the network increases as illustrated in figure 1.1. Compared to a *standard interconnect* the performance decrease due to communication of a *low latency interconnect* is at a higher

parallelization rate. The gap between the two gray lines represents the increased scalability of the network due to the lower latency. Here the performance gain decreases at a higher rate of parallelization improving the overall efficiency of the cluster.

The three fastest supercomputers according to the *Top500* list of June 2011 [4] all use a custom built interconnection network to leverage the increased processor count without decreasing the overall performance. Such extreme scale computing will always have a need for custom interconnects, like the *TOFU Interconnect* from Japan's *K Computer* [2] which ranked first in the June and November 2011 *Top500* lists or Cray's *Gemini Interconnect* [5], which is used in Cray's *XE6* and *XK6* series systems. Custom interconnects have the advantage of being able to be designed to exactly meet the customers needs and therefore remove bottlenecks that occur with commodity interconnect technology.

But not only the HPC sector strives for low latency, also financial companies require it to be able to participate in high frequency trading (HFT) or low latency trading. In low latency trading information from the exchanges are received through a common network, need to be processed and then a decision has to be made whether to buy or sell a stock based on the received information. Subsequently an order has to be sent back to the exchange. Consequently low latency is the key for making money, as only the fastest company is able to buy or sell the most profitable opportunities. For HFT however not only latency, but also *throughput* is of essence to be able to process all incoming information provided by the exchanges. If the system is not capable of processing all arriving data, packets need to be dropped. The result is an inconsistent view of the current market situation, which can then result in wrongful trades.

The latency of a HW structure is determined by two factors, first the amount of pipeline stages and second the frequency at which the module is able to operate. While a deeper pipeline, i.e. a high amount of pipeline stages, enables the designer to increase the operation frequency, it also increases the latency as each pipeline stage requires a clock cycle. A trade off between number of pipeline stages and clock frequency has to be made in order to get the best latency result for a certain module.

Therefore the aim of this thesis is to find ways to reduce latency required for communication and at the same time increase the throughput. This thesis presents HW structures that enable the reduction of latency for different communication types in particular and also shows structures that enable lower latencies and a high throughput for packet processing in general.

1.2 Graphical Representations

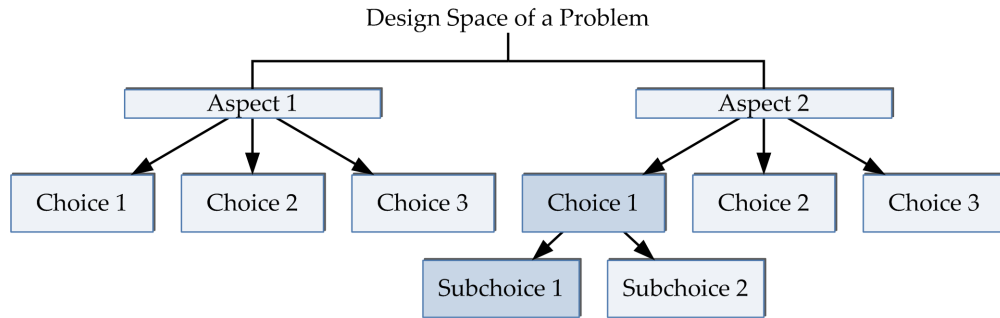


Figure 1.2: An Example Design Space Analysis

An important part of this work is to analyze options and aspects to solve a given problem. *Design space diagrams* have been introduced by [6] and are a common way to illustrate all different aspects that need to be taken into account and their design choices. Figure 1.2 shows an example diagram where a given problem is divided into multiple orthogonal aspects which are connected using also orthogonal lines. For each aspect all possible options are connected using direct arrows. A third level is used, if an option can be further divided. The option chosen for an implementation is highlighted using a darker blue tone.

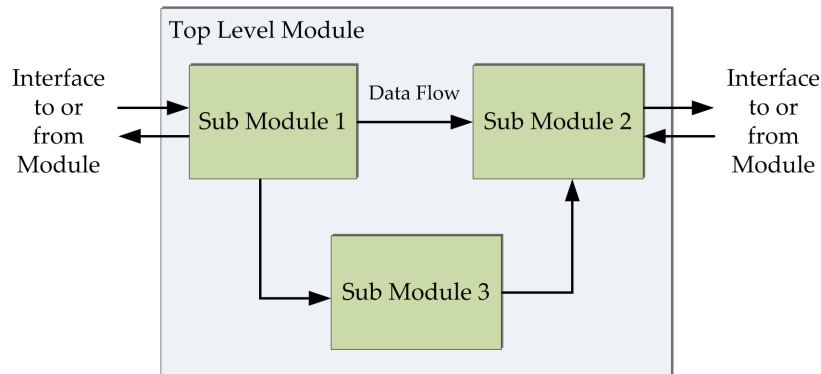


Figure 1.3: An Example Block Diagram

HW is always described in modules, which include the desired functionality. If a problem is very complex, these modules are often divided into multiple sub-modules, which are connected to each other. A *block diagram* is then used as an abstraction layer to illustrate the connectivity of these sub-modules within such a top level module. In figure 1.3 an example for such a block diagram is given. Interfaces to other modules are illustrated using arrows at the edges of modules, the direction of the internal data flow

is also illustrated using arrows. The actual wiring required between modules and sub modules is often not fully displayed, as the diagram is otherwise unclear.

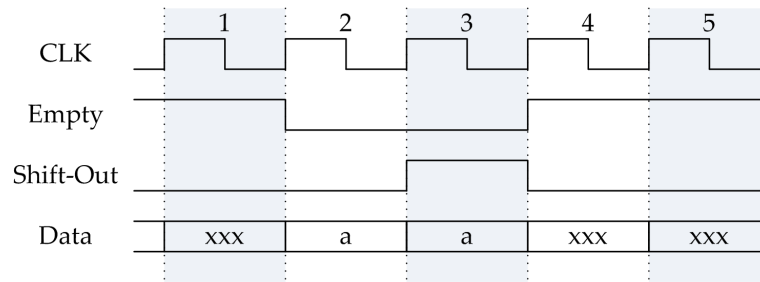


Figure 1.4: An Example Timing Diagram

The third used diagram in this thesis is a *timing diagram*. It illustrates a cycle accurate behavior of a module, unit or interface. The timing of each signal is given in dependency to a common clock. Using such diagrams enables to illustrate the latency of a module or the correct interaction between modules.

In figure 1.4 an example for such a timing diagram is given. In this example the interface between a module and a first-in-first-out (FIFO) buffer is illustrated. It shows a fall-through FIFO implementation, meaning the first data word stored inside is already visible at the output before data has actually been shifted out.

1.3 Outline

This thesis is divided into four main chapters. The first chapter introduces mechanisms and building blocks that reduce latency, enable an increase of the throughput, offload processing intensive tasks to specialized HW or ease the HW-designers work. They can be used in various environments that require high throughput and low latency processing. Additionally a low latency, high-throughput decode unit for the financial market is introduced.

The second chapter concentrates on crossbars in general and on Extended ATOLL (EXTOLL)'s crossbar design in particular. It discusses all made design decisions and explains their implementation. Performance is compared to a previous design and an outlook for future extensions is given.

The third chapter explains the functionality of EXTOLL with all its functional units (FUs) that facilitate EXTOLL achieve its outstanding performance values. All involved

1 Introduction

units are explained and those which are part of this thesis are introduced in detail. Micro-benchmark results of the complete design are given to underline the made contributions of the crossbar as well as the EXTOLL Network Interface Controller (NIC).

The final chapter closes this thesis with a reflection on the achievements and the impact of this work.

2 Efficient Packet Processing

All communication intensive computing tasks like Internet services, cloud computing, HFT or HPC require a lot of computing resources just handling incoming and outgoing packets due to the fact that the network gets faster with every new generation. However, the protocols themselves or the SW handling those networks often do not improve with the same rate. Ethernet for example is a network standard that has been introduced in 1980. More than 30 years later it is still the most often used protocol without any improvements. It requires a lot of processing power just to encode and decode the various framing layers.

The Internet is driven by Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), both protocols work on top of Ethernet and are also widely used in HFT or HPC. The standard use case is kernel level communication, which always involves the operating system (OS). Kernel-level communication means that a SW thread hands over a packet to the OS and the system then issues the packet at the network adapter. This makes it necessary to copy packets to be sent at least one time inside main memory of the host system and involves at least one thread change from the user application to the OS. The same of course also applies for the reception of packets. All this makes kernel level communication very inefficient because resources are used for protocol handling which can otherwise be used to solve the actual workload.

Latency is often an issue, especially in timing critical applications like HFT, where every spared microsecond can be turned into possibly more profit. Reducing latency also increases the scalability of a network as the time required to traverse the network decreases making it possible to build larger networks. Efficient mechanisms are therefore necessary to reduce the workload of a single machine and reduce the latency.

The latency of a packet can be reduced using two mechanisms. Either it can be avoided by minimizing the required processing steps a packet has to pass, or it can be hidden by overlapping the processing of multiple packets. Both mechanisms can of course also be combined to get the best results.

Sometimes a change to a new type of interconnection network like switching from

2 Efficient Packet Processing

Ethernet based systems to InfiniBand [7] or other high performance networks already increases the scalability by an order of magnitude. However, in some cases this might not be possible. The infrastructure at exchanges for example are a given and cannot be changed by companies actively trading there. But improving the performance in this environment is very critical for the longterm survival of companies involved in HFT on such exchanges.

One commonly employed mechanism when improving performance of a network is to bypass the kernel of the OS and therefore do everything, including communication at the user-level. I/O devices must be virtualized in order to be able to handle them on the user-level. The virtualization is required so that devices can safely be mapped into the user-level address space, otherwise multiple applications can intervene each other when accessing devices or act as another process and therefore hijack a connection. Sophisticated Ethernet adapters like the ones offered by *Solarflare* [8] or the *QLogic* [9] InfiniBand adapters can be configured to bypass the kernel. This reduces the latency significantly, the computation and handling of packets on the other hand remains in the host Central Processing Unit (CPU).

To further improve the performance of a network two approaches are used, either *onloading* and *offloading*. *Onloading* tries to keep the HW as simple, and therefore fast, as possible and do most of the computation on the host CPU. The above mentioned QLogic InfiniBand adapter for example is very lean and QLogic promotes onloading [10] as the way to improve performance in the multi-core era.

Offloading is the exact opposite. It tries to put as much logic and packet processing tasks as possible into specialized HW as it often can process those tasks faster than a general purpose CPU, even if the CPU runs a lot faster than the HW.

Both approaches have their positive and negative sides and care has to be taken which approach is more beneficial for which task. There is no general rule that *onloading* or *offloading* is always better, it depends a lot on the task and the actual implementation whether or not it really is beneficial at the end. A trade-off has therefore to be made for every problem that has to be solved in order to get the best performing HW-SW co-design.

This chapter introduces two of the most often leveraged building blocks in chip designs, *arbiters* and *FIFOs*. Subsequently more sophisticated new HW structures are introduced that help to reduce the SW effort required for packet processing. In section 2.3 on page 22 the *Write Combining Buffer* is presented, a unit capable of reducing network traffic and simplifying both SW and HW for packet processing. In section 2.4 on page 24 a new HW

structure is introduced that is capable of efficiently maintaining large virtual ring buffers in main memory.

Section 2.5 on page 33 presents a module to parallelize the search of a communication partner when using Message Passing Interface (MPI) [11].

Finally in section 2.6 on page 42 an application of HW acceleration for HFT is shown. In contrast to section 2.5 on page 33, the proposed structure leverages the advantages of HW acceleration not by parallelizing independent tasks, but decoding a protocol which is highly serial in nature, using a specialized structure to process packets consecutively.

2.1 Arbiter Generator

Every chip design has some resources that are shared by several other resources. This is done for modules that are not performance critical or too complex, i.e. too large. Replicating them requires additional or too much area on the chip or field programmable gate array (FPGA). Thus, arbiters are a common building block required in almost every design. They are utilized to equally share that single structure whenever multiple units require access to it. In this case the arbiter makes sure that only one unit has access to the unique resource at a given time, but all units receive access sometime. Another scenario where arbiters are often deployed is to equally distribute elements generated by one instance to a number of units. This can for example be utilized to parallelize work and make sure all units have similar workloads.

Consumed area and maximum achievable clock frequency are the most important criteria for such arbiters. While being fast and small, they however also need to be fair. Fairness means two things, first there must be no starvation, i.e. no requesting unit has to wait for an indefinite time for the resource. Secondly, there must be no order or priority in any way, all units must be treated the same and receive access to the resource in the same amount.

The most commonly implemented arbiter is a *round-robin* arbiter. It selects the next grant by going through all set requests either clockwise or counterclockwise starting at the last given grant.

2.1.1 State of the Art

An arbiter generator has been developed at the Computer Architecture Group of the University of Heidelberg (CAG) by [12] which can be depicted using a finite state machine (FSM). Its size grows exponentially and its performance decreases accordingly with growing input numbers as it is shown in figure 2.1.

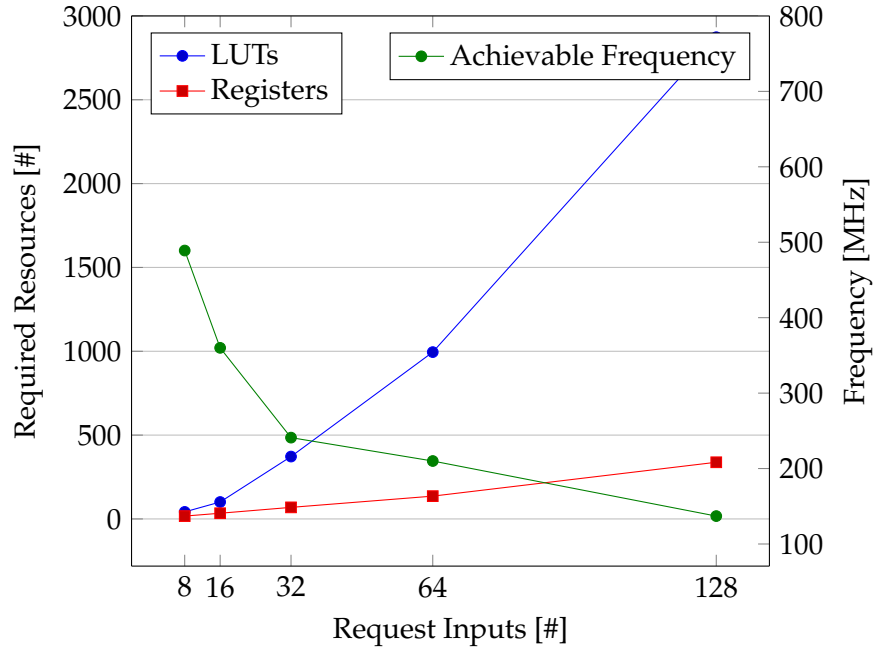


Figure 2.1: FSM based Arbiter Resource and Performance Evaluation

A good overview of other possible implementations is given in [13]. One particularly interesting is the one introduced in [14]. The arbiter proposed in this paper is based on a binary tree search (BTS) [15] to find the next request to be granted. It can be described best as two fixed priority arbiters that have been merged into a single one. A fixed priority arbiter is an arbiter which always finds the first request in the input vector. Thus, the first arbiter of these two finds the first request always starting at the very first bit position. The inputs of the other arbiter are masked by the last given grant and therefore finds the first request starting at the position of that last given grant. If the second arbiter finds a request, then its result is used, otherwise the result from the first arbiter is used.

The resulting block diagram is illustrated in figure 2.2 on the next page. Internally the arbiter leverages a thermo code to implement the arbitration algorithm. This thermo code is then looped back to be used as mask and a thermo-to-one-hot code logic generates the one-hot coded output.

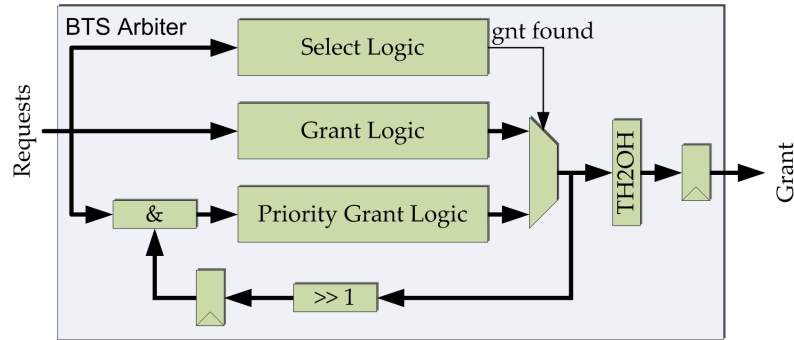


Figure 2.2: Simplified Block Diagram of the BTS Arbiter [14]

2.1.2 Implementation

To prove the claims made in [14] a *Perl* [16] based script has been implemented to be able to generate arbiters with various variables. Configurable upon generation are not only the number of desired requests, but also which output encoding is to be used. Selectable are either binary, one-hot or both encodings. Further it can be decided whether or not an *any grant* signal is to be generated and whether or not a *stop* input is to be taken into account. Additionally all outputs of the generated arbiter can be either synchronous or asynchronous to a given clock. This feature can be used if one of the outputs is required for further logic within the instantiating module. However, these asynchronous outputs should only be used if very little additional logic is to be added before registering the results, as otherwise the clock frequency at which the design can operate decreases significantly.

Finally a rich set of *SystemVerilog* verification and coverage code [17] can be generated, making it easy to verify the correct functionality of the arbiter during simulation.

The BTS based arbiter design from [14] performs well in both area as well as timing. When comparing the results given in figure 2.1 on the facing page and figure 2.3 on the next page it can be seen that it scales significantly better as the previously used FSM based arbiter and achieves higher clock rates in any of the tested configurations. Its size increases only linear with growing input numbers. The increase is therefore significantly smaller than the one from the FSM based arbiter as it can be seen on both diagrams for look-up table (LUT) as well as register consumption.

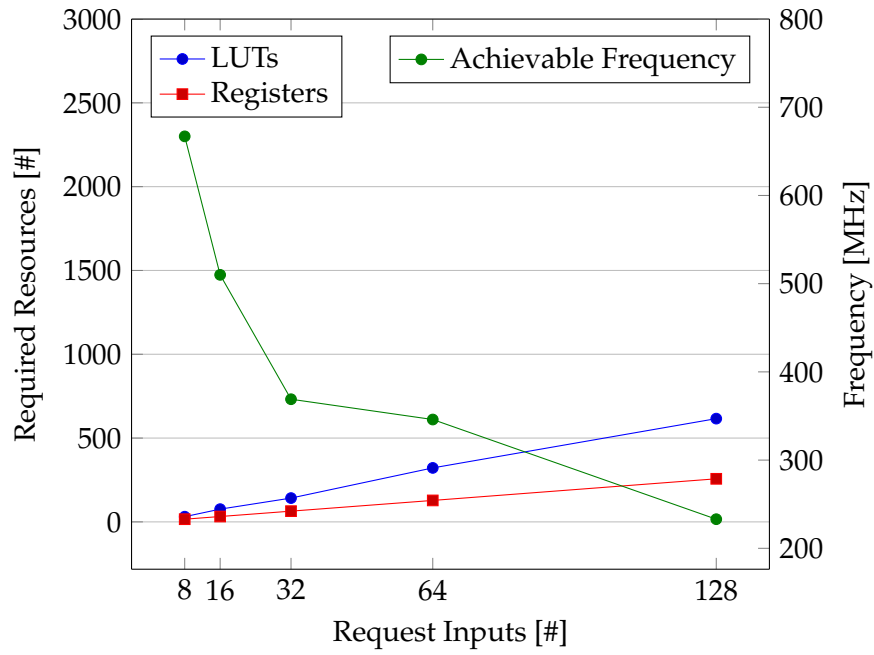


Figure 2.3: BTS based Arbiter Resource and Performance Evaluation

2.2 FIFO Implementations

One of the most common building blocks in HW designs are FIFO buffers. The applications for FIFOs are versatile. They can for example be used to synchronize data between two clock domains, store data until an event happens or to decouple two modules. Often, additional logic is needed around these FIFOs to implement the required behavior. Therefore it is beneficial to have a set of different FIFOs with additional features at hand that can improve performance, simplify or reduce the latency of a design.

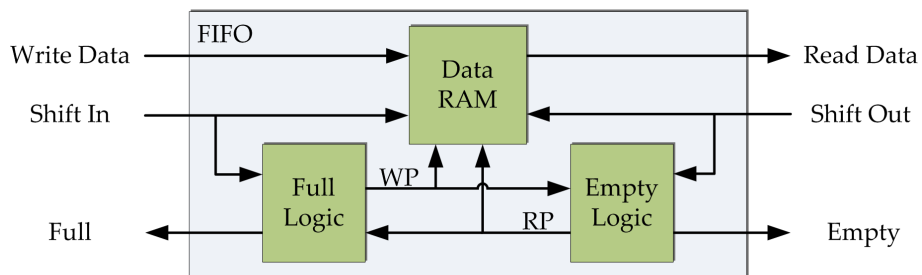


Figure 2.4: FIFO Block Diagram [18]

A FIFO is a HW storage structure where data can be pushed into or pulled from. The order of data is maintained while pushing data into or pulling data out of a FIFO. There are

two possible implementations to realize the FIFO functionality. It can either be described using a ring buffer or using shift registers.

The ring buffer approach is realized using a random access memory (RAM) or a sea of registers for data storage and a set of a write- and a read pointer. The write pointer points to the next address to write data to and is thus incremented with every *shift-in* operation. The read pointer on the other hand points to the next data value that can be read and is incremented upon every *shift-out* operation. A block diagram of such a FIFO is shown in figure 2.4 on the facing page.

$$empty = (wp == rp) \quad (2.1)$$

It can be distinguished whether a FIFO is full or empty by comparing the write- and read pointers. The buffer is considered empty if both pointers are equal as indicated in equation (2.1) and full if the write pointer plus one equals the read pointer as shown in equation (2.2). Upon initialization both pointers are zero, consequently the FIFO is empty. The FIFO always has one free element left when using the equation for the *full* condition given in equation (2.2). This has to be done in order to be able to differentiate between the *full* and *empty* condition.

$$full = ((wp + 1) == rp) \quad (2.2)$$

To completely fill a FIFO up to its last cell a more complex equation and an one bit wider register is required for the read- and the write pointer to be able to distinguish whether the pointers both wrapped around or not as indicated in equation (2.3).

$$full = ((wp[WIDTH - 1 : 0] == rp[WIDTH - 1 : 0]) \& (wp[WIDTH]! = rp[WIDTH])) \quad (2.3)$$

A shift register implementation uses a series of registers, that are connected in a line. Each register additionally has a state assigned to indicate whether it is full or not. The first empty register is used to write data into. Upon every read all data is shifted by one to the right and the most right value is the output value for the read port.

Shift register based FIFO implementations are usually used for very small buffers, as they are fast but require more resources with increasing size. RAM based FIFOs are then leveraged for larger buffers.

2.2.1 Multi-Queue FIFO

In applications where multiple threads access a resource, it is often required to have multiple FIFOs in parallel to store information for each thread independently. However, only one of these FIFOs is accessed at a time, as the HW itself is not multi-threaded. Each of these FIFOs requires a discrete RAM for the data, one set of read- and write pointer registers and all the logic required to calculate the full and empty condition. Having a lot of parallel FIFOs is not very resource efficient in both FPGA and application specific integrated circuit (ASIC) implementations as it results in the usage of a lot of small RAMs and a lot of redundant logic that is only used consecutively.

A set of small RAMs usually requires more space in an ASIC as a single RAM with the accumulated size of all small RAMs. Additionally, the design of the power grid for a chip with lots of RAMs is more difficult as each cell requires its power supply, making it complicated to find a regular structure where all cells fit into. In a FPGA environment, each small RAM is implemented using a single RAM resource of the device, resulting in a lot of unused buffer space. A *Multi-Queue FIFO* can be used to replace such a stack of FIFOs. Multiple FIFOs are merged, sharing a single RAM, thus using resources more efficiently. However, each queue remains logically separated and also keeps its order. Thus, set of pointers is required for each queue is still required. The logic to generate the full and empty condition for each queue however can be shared.

State of the Art

An US patent claims a FIFO with the capability to store data for multiple independent threads [19]. Its implementation however is very complex, as the plurality of FIFOs is realized using linked lists storing the data in a shared buffer. The free addresses of the buffer are also managed using a linked list.

Linked lists are always hard to handle in HW as it requires a start-, an end pointer as well as an intermediate pointer which points to the address of the next value until the end is reached. Whenever an item is added to the list, the end pointer and the intermediate mapping has to be updated resulting in two RAM write accesses. Removing an item

requires only a single write access as it is sufficient to update the start pointer. However, for every update, all RAMs need to read. The advantage of this approach is, that the size of the different queues can be dynamically adapted as long as buffer space is available. Another implementation has been introduced by Tamir et. al. [20] which also uses linked lists to manage the independent queues as well as a free block list.

Implementation

In contrast to [19] the here proposed approach does not use linked lists for the management, it uses a set of read- and write pointer per queue. This has the advantage of being a lot simpler and therefore resulting in less and faster logic. The complete pointer structure is stored in a single RAM making this solution scalable. The resulting block diagram is shown in figure 2.5. Additional *R-Queue Select* and *W-Queue Select* signals are added to the interface. They are used to select the queue to be accessed. Upon every write or read, a lookup of the corresponding set of pointers needs to be done before the actual data value can be written or read.

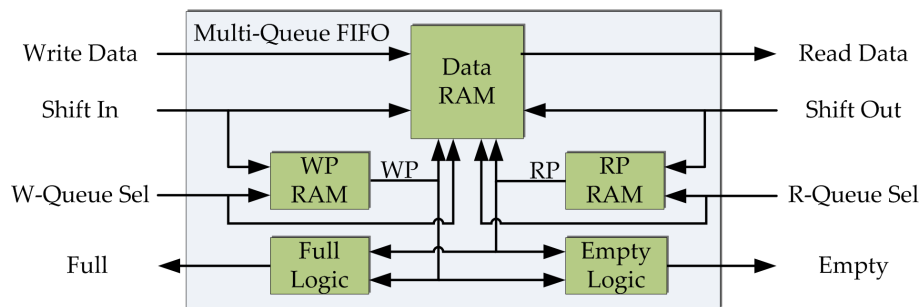


Figure 2.5: Multi-Queue-FIFO Block Diagram

The address for the *data RAM* results from the concatenation of *queue select* and the current pointer. The upper part selects the queue to write to or read from and the lower part selects the position inside this selected queue. This is beneficial as no additional logic is required to isolate the different queues. Consequently all queues have the same size.

To be able to dynamically select the size and amount of queues inside the *Multi-Queue-FIFO* an additional *mask* input has been added. *Mask* has the width of the *data ram address* and selects which part of the address is to be used as *queue length* and which as *queue select*. The partition of the address can then be changed dynamically to either less, but larger queues or more queues with less elements. However, changing the mask and therefore the size of the queues can only be done if all queues are empty, as otherwise data integrity

cannot be guaranteed. Possible values for the amount of queues are 2^n whereas $n \leq m$ so that 2^m equals maximum number of supported queues.

In figure 2.6 the size of the *write pointer* and the *queue selection* are determined by the *mask*. A complex *shift and merge* logic makes sure that the *queue selection* is correctly positioned in the resulting address, avoiding overwriting parts of the *write pointer*. The *queue selection* register is shifted by the amount of bits necessary, i.e. the amount of set bits in the *mask*.

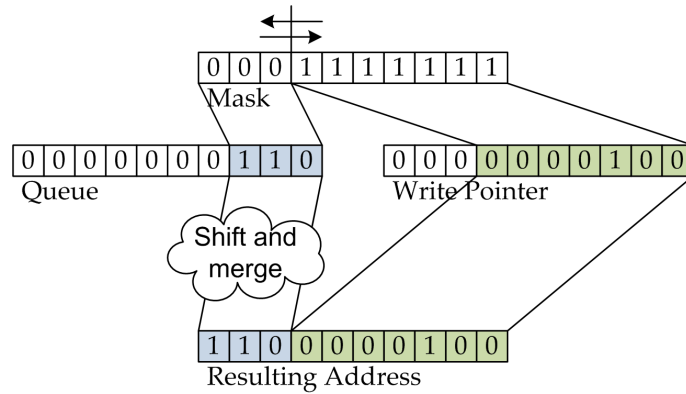


Figure 2.6: Address Generation Using a Complex Logic

In figure 2.7 the address generation is much simpler. *Queue selection* is bitwise reversed and then masked, resulting in less complex logic, as it has no longer to be determined at which position the *queue selection* has to be placed. Reversing the order of bits is just a matter of wiring without any logic requirements. The reversed order can be used safely without the risk of data corruption, as the result is also unique and thus always points to the same location inside the RAM.

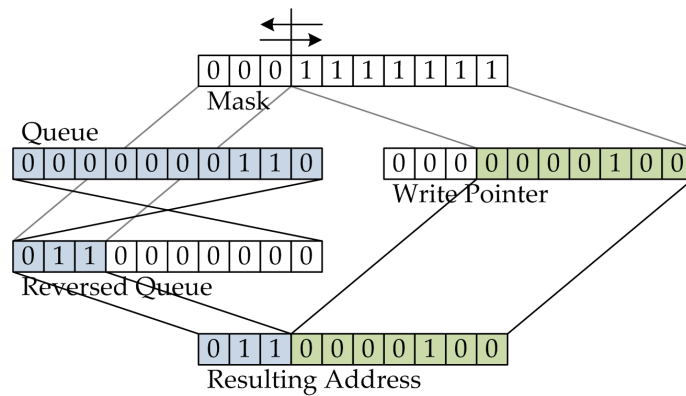


Figure 2.7: Address Generation Using a Reversed Queue Selection

Evaluation

A comparison with a conventional FIFO has been made to evaluate the resource consumption. The conventional FIFO has been instantiated multiple times to put the resource consumption into perspective with the *Multi-Queue-FIFO*. The results of the conventional FIFO can be seen in figure 2.8.

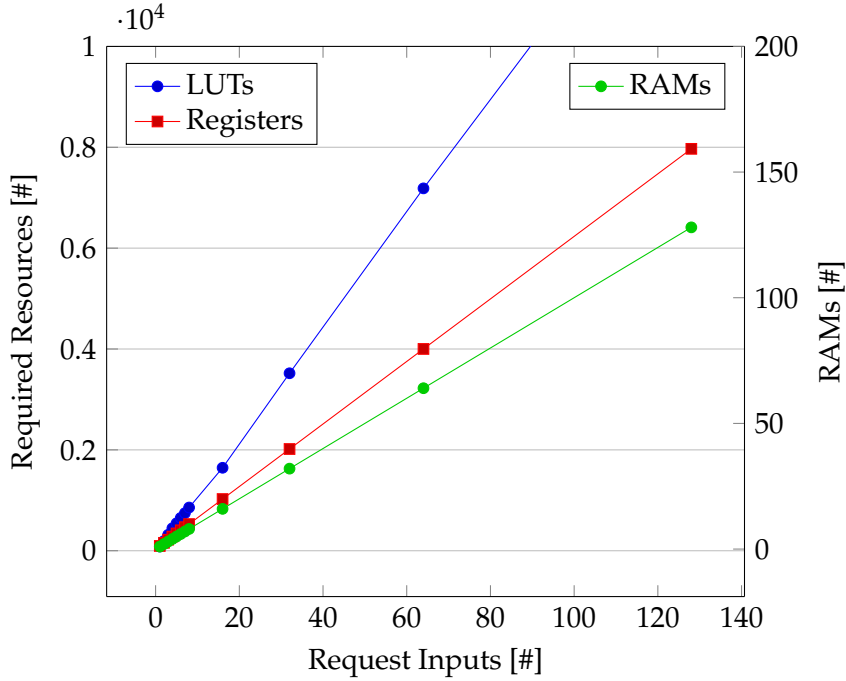


Figure 2.8: Resource Requirements of N conventional FIFOs

The *Multi-Queue-FIFO* requires only a fraction of the resources as the diagrams shown in figure 2.9 on the next page indicate. A *Multi-Queue-FIFO* with 128 queues of size M requires a comparable amount of resources as 32 discrete FIFOs.

This FIFO has successfully been integrated in the EXTOLL crossbar, described in chapter 3 on page 63, substantially supporting the scalability of the crossbar design.

2.2.2 Speculative FIFO

Packet processing often requires the checking of a cyclic redundancy check (CRC) or in the Ethernet Protocol (ETH) context a checksum. The result of such checks is only known after all data has been stored in some kind of buffer as the whole packet needs to be considered for the check. If this buffer is a conventional FIFO removing data again is a

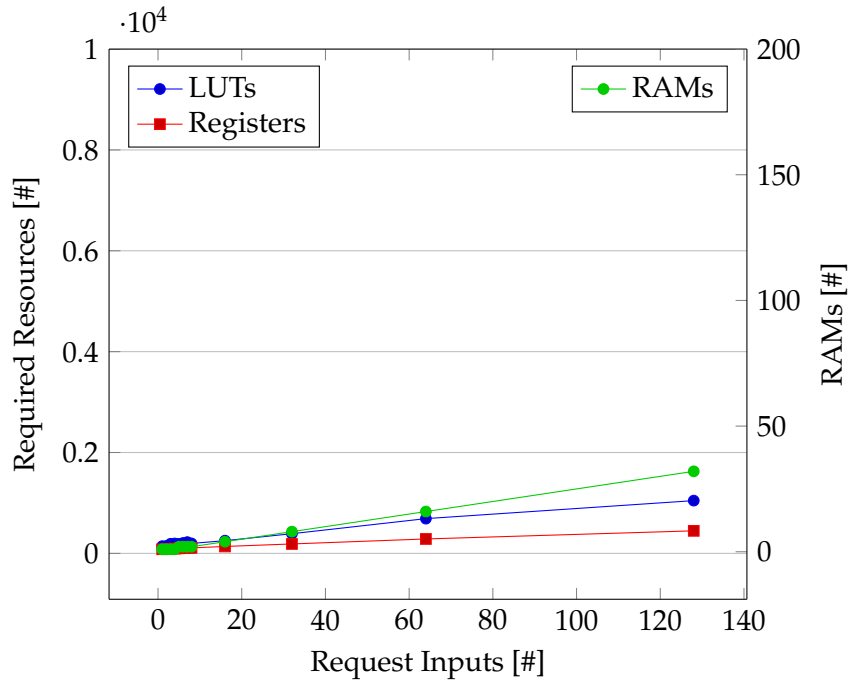


Figure 2.9: Resource Requirements of a Multi-Queue FIFO with N Queues

tedious task as all data not belonging to the erroneous packet need to be kept inside the buffer. In addition, removing an erroneous packet from a FIFO again requires N clock cycles, whereas N is the length of the packet in data words.

A solution for that problem is to add speculative writing capabilities to the FIFO. This means that data is written into it, but the read side of the buffer sees the data only if the write logic has committed them. Once an erroneous packet is detected, the already written words are revoked in a single clock cycle and the next packet can be written at that position instead.

The same mechanism can of course also be used for the read side of a FIFO. Here the feature can for example be leveraged to realize a retransmission buffer for packets. By resetting the read pointer, all data can be read again starting at the selected position without the need of an additional RAM or complex logic.

State of the Art

[21] claims in its US patent a FIFO with a *push back* function, enabling it to put the last data word back into the FIFO. The drawback in this implementation is that it is only capable of

repeating the very last word, making it only feasible for very specific applications. [19] claims to have speculative shift-in and -out logic. However it does not explain how this mechanism is realized. Another US patent describing similar functionality is [22]. It is a FIFO with a *retransmit mode*. Thus it is only capable speculatively shifting data out of the buffer, but not into it. The retransmission is realized using a shadow register which can be used to reset the actual read address. The FIFO has to be put in *retransmit mode* and back to set new retransmission positions. Consequently it requires additional clock cycles to be able to reset the read address.

Implementation

In contrast to [21] and [22] the here proposed architecture is less complex and can be applied for both, read and write side of a FIFO. For simplicity reasons only the write side will be explained, the read side works analog. It is capable of speculatively writing data to the FIFO without any idle cycles as long as there is buffer space available. Reverting or committing data takes only a single cycle, whereas committing can even be done in parallel to writing data.

The implementation uses two pointers instead of only one, a *speculative*- and a *committed* pointer. Only the committed pointer is handed over to the read logic to calculate whether data is stored in the FIFO or not. The speculative pointer on the other hand is used as write address and to compute the full condition. Consequently it can happen that the buffer is full and empty at the same time. A block diagram of that FIFO is shown in figure 2.10.

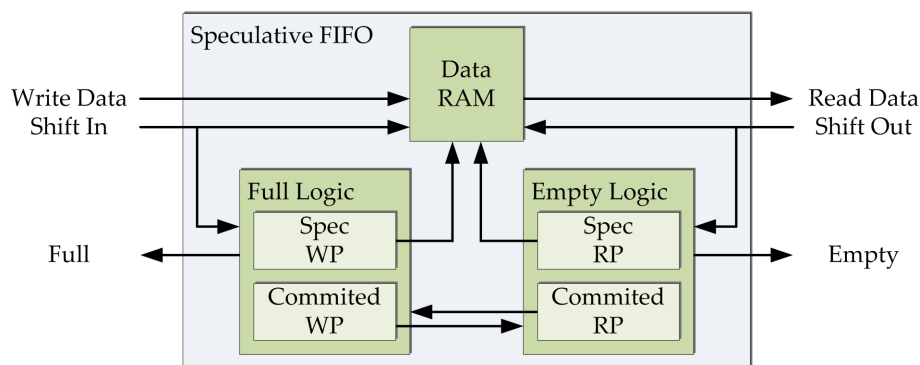


Figure 2.10: Speculative FIFO Block Diagram

Only the speculative pointer is incremented, whenever a word is written into the FIFO. A separate interface containing an *increment pointer*, a *decrement pointer* and a *value* signal realizes the pointer manipulation. *Value* gives the amount of words to commit or revert,

increment pointer indicates a commit of words and *decrement pointer* indicates that the pointer needs to be reverted.

Once *increment pointer* is pulsed the committed pointer is incremented by the value given on the *value* bus. However, it is taken care that the committed pointer can at maximum reach the speculative pointer and therefore never overtake, which otherwise causes data corruption. Whenever *decrement pointer* is pulsed the speculative pointer will be subtracted by the value given on the *value* bus. Additional logic makes sure that the speculative pointer can be decremented only to the value of the committed pointer and therefore never be smaller.

The proposed architecture results in an efficient implementation where only an additional set of pointers and some logic are necessary to feature speculative read- and write capabilities. Table 2.1 shows the synthesis results of a conventional FIFO in comparison with the speculative FIFO.

	LUTs	Registers	Cycle Time (in ns)
Conventional FIFO	197	23	2.533
Speculative FIFO	237	41	2.777

Table 2.1: Cost Comparison of a Conventional and a Speculative FIFO

As it can be seen, the amount of LUTs increases by around 20 percent for the extra logic required to manipulate the pointers and the register count almost doubles due to the required second set of pointers. Performance in respect of achievable cycle time however decreases by less than 10 percent. The required RAM resources remain the same for both implementations and are therefore not shown in table 2.1.

This FIFO implementation is leveraged in the EXTOLL's Network Port (NP) to implement an efficient store-and-forward mechanism with error checking capabilities.

2.2.3 Double Shift-Out FIFO

Usually a FIFO is only capable of shifting out a single word per clock cycle following the first-in first-out directive. In some cases it might however be beneficial to be able to increment the internal read pointer not only by one, but by two and therefore jump over one entry.

This can be needed if there are data dependencies. For example a certain condition is encoded in the first word of a two word command. After processing the first part it became clear that the second part of the command is not required. With a conventional FIFO this second part needs to be shifted out in a second clock cycle, requiring a total of two cycles. The double shift-out capability removes the necessity of the second clock cycle as both parts can be shifted out in one cycle, thus cutting the required time to process the stored values in half. These saved clock cycles reduce the latency of a FU considerably as it will be shown in section 2.6.6 on page 58.

State of the Art

An US patent that has been filed in 2004 [23] claims similar functionality. The differences to the here proposed architecture is that the patented FIFO implements two independent interfaces to either shift-in or -out one or two words. Thus either a single RAM with two independent read- and two independent write interfaces, or multiple RAMs are necessary, resulting in a complex RAM structure. No further publications have been found on this type of FIFO.

Implementation

The double shift-out feature is implemented by adding a *double shift out* control signal indicating that the next value inside the FIFO is to be skipped. Internal logic prevents that incrementing the read pointer by two does corrupt data. This can occur if only a single word is stored inside the FIFO and the connected unit does not take that information into account. Whenever only one entry is stored in the FIFO incrementing by two causes the read pointer to overtake the write pointer which has to be avoided at all times since then the FIFO appears full instead of empty. If only one entry is stored and the *double shift out* control signal is set, then the read pointer is only incremented by one instead of two. An additional control output notifies the unit leveraging this capability of this condition to avoid flow control issues.

	LUTs	Registers	Cycle Time (in ns)
Conventional FIFO	197	23	2.533
Double-shift FIFO	213	28	3.220

Table 2.2: Cost Comparison of an Conventional and a Double-Shift-FIFO

As it can be seen in table 2.2 on the previous page the FIFO with integrated double shift out feature only requires little additional resources. The amount of RAMs required for the two different implementations are the same and therefore not shown in table 2.2 on the preceding page. The improvement impact however can be significant as it is later shown in section 2.6.6 on page 58.

2.3 Write Combining Buffer

Today's host CPUs are able to split, reorder or combine memory *read response*- as well as *write* packets for performance purposes. This poses no problem for devices that work purely with addresses like memory controllers. However, devices that rely on the correct ordering of packets need to be able either reorder and combine these packets again or be able to work with the split parts directly.

Such functionalities are better encapsulated in a separate module instead of integrating it directly in each unit capable of receiving data from the host CPU. Encapsulating it and reusing the same functional block multiple times, not only simplifies the implementation of the actual FU, it also reduces the risk of failure due to wrongful implementation. This single module can be verified extensively and then integrated into FUs knowing its proper functionality.

To solve such split and reordered packet the *Write Combining Buffer* has been developed. It is capable of combining and reordering such split packets. Additionally it is capable of combining packets that are larger than a single maximum transfer unit (MTU) of the host interface to one single packet, making it possible to leverage the full potential of the EXTOLL network, which is described later. Thereafter the now combined packets are forwarded as a whole to the connected FUs.

When merging packets a buffer is required in the *Write Combining Buffer* to store the separate parts and forward the packet once it has been received completely. Each thread accessing a FU requires an exclusive part of that buffer, as split parts of different packets can also be reordered and interleave each other. As this buffer increases with the number of supported threads and supported packets per thread, a trade-off must be made to use the available real estate on an FPGA or ASIC efficiently. The minimum requirement however is that at least one network MTU fits into each part of the buffer, so that an assembly of a complete packet is possible. This minimum requirement can only achieve a throughput rate of less than 50 percent, as a packet first has to be written completely, then

the whole packet needs to be read by the FU before the reassembly of the next packet can begin.

This bottleneck can be reduced by adding a shared buffer behind the actual combining buffer, in which complete packets are streamed into as soon as it is ready to be processed, at least reducing the amount of idle cycles to a bare minimum. However, this improves the performance only to a maximum throughput rate of 50 percent, for more performance multiple slots per thread are required.

To achieve full performance double buffering is required for each thread, meaning that at least two maximum sized network packets need to fit into the reassembly buffer. For the FPGA implementation supporting multiple network MTUs per thread is unfeasible, due to RAM limitations on the device, therefore only one packet is supported. If there are less than the maximum amount of threads currently running on a host, one thread can also use two such slots to improve its performance and then leverage the full bandwidth.

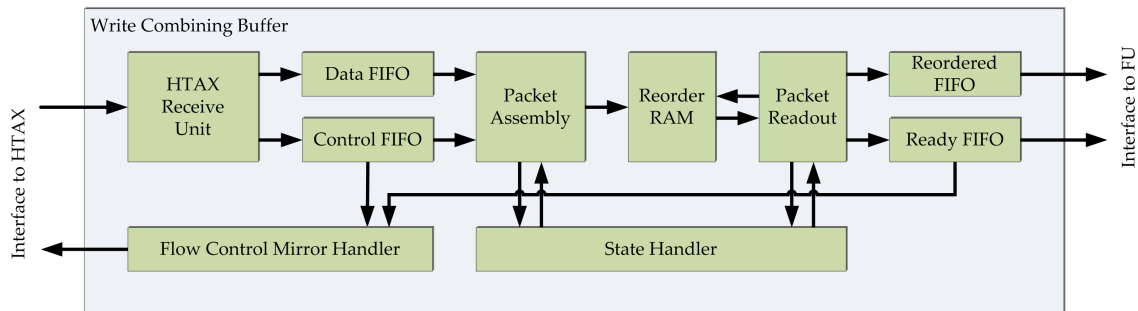


Figure 2.11: Write Combining Buffer Block Diagram

For flow-control reasons the fill-level of the shared FIFO is periodically reflected back to main-memory. The sending SW has to check that level before inserting new packets through the host interface, otherwise this interface can get blocked until progress can be made on the network. Back-pressure can occur in HyperTransport (HT) based systems if the SW does not observe the number of injected packets, effectively creating a possible deadlock and stalling the whole system. Write requests to non-coherent devices can only be made using the *posted* virtual channel. A required update to the register file then cannot be completed as the *posted* channel is occupied due to above mentioned back-pressure.

Only the first part of every message contains a header with the length of the complete message. Once this part arrives¹ the remaining words are calculated taking into account the words that have already been written and the words of the current packet. The

¹the first part does not necessarily arrive first.

position of each split part within a message can be extracted from the address offset that is transmitted with each part.

The block diagram of the resulting unit can be seen in figure 2.11 on the preceding page. The *Write Combining Buffer* is successfully integrated in the FUs of EXTOLL, namely Virtualized Engine for Low Overhead (VELO) and Remote Memory Access (RMA). Both units are described in more detail in chapter 4 on page 133.

2.4 Virtual Ring Buffer Handler

Ring buffers are a common construct to receive data or notifications whenever a SW communicates with a device like a network card, a general purpose graphics processing unit (GPGPU) or a FPGA accelerator. Essentially, a ring buffer is a FIFO, handled and maintained by SW and located in main memory. Such a ring buffer has the same *full* and *empty* criteria as a FIFO, described in section 2.2 on page 12. These ring buffers can be used to avoid kernel level communication. When using kernel level communication, the HW works on its given task independently and sends an interrupt to the OS when it is done. Thereafter the OS wakes up the user application and informs it that the HW has completed its task. This lacks efficiency due to the required OS involvement, resulting in a high latency. Moreover, a lot of jitter is introduced, as the required thread change from the kernel to the user application results in indeterministic latencies. This jitter is to be avoided at all times whenever parallelizing a task as it constrains the scalability. Jitter wastes processing time, as multiple threads are finished processing their given problem at different times and therefore need to wait for others.

A higher performance can be achieved if only user-level communication can be used. A main memory region needs to be mapped into the virtual address space of the SW thread, which can be leveraged as a ring buffer by the HW to write into and by the SW to read from it. However, in addition to a write- and read pointer, the main memory ring buffer requires a base address and the size of the memory region to be fully defined. Given these information, the HW writes into the memory with an ascending write pointer starting from the base address and once the upper bound, i.e. the size of the memory region, has been reached starts again at the base address. Instead of passively waiting to be woken up, an application can then actively poll on the next value in its ring buffer to check whether new data arrived or not.

Some FUs like VELO, which is described in section 4.6 on page 153 in more detail, also

use such a ring buffer in main memory to not only write notifications, but also write data directly in line with the notification. The advantage of writing data in line is that no second main memory access from the SW side is necessary to receive a complete message. The larger such a region is, the less frequent a SW thread has to check whether new data or notifications have been received and can use the CPU time to process its actual task as the buffer is more elastic. Additionally, the latency for credit updates² can be hidden efficiently.

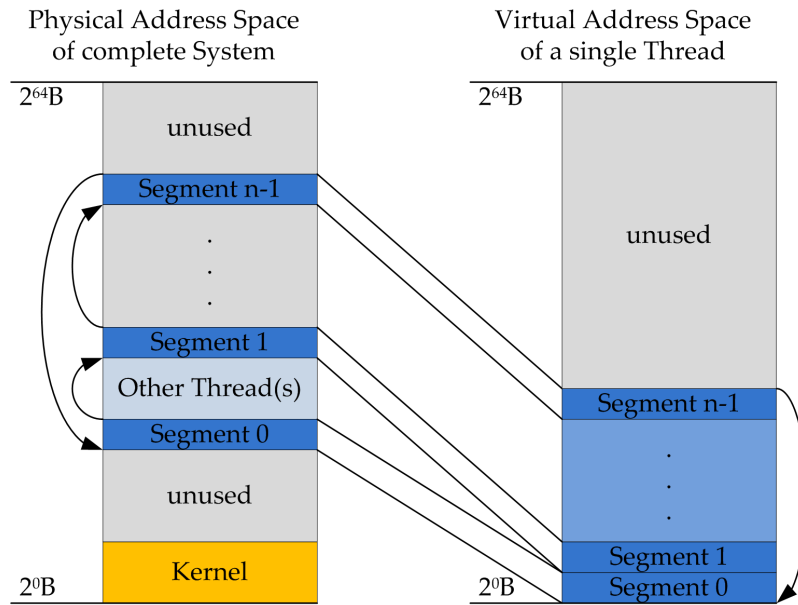


Figure 2.12: Memory Mapping of Segments to User Space

The limit for continuous memory segments that can be requested in today's Linux [24] based systems is four MegaByte (MB). This is due to the implementation of the memory allocator in the Linux kernel. The largest chunk that can be requested is the mentioned four MB, but it cannot be guaranteed that even these are granted under all conditions as the allocator separates the whole main memory in four MB slots and bisects such a slot if less than the maximum size is requested. The main memory is fragmented in a way that no maximum sized slot is available anymore, if a system does not have a large memory or is already running for a while. Larger consecutive areas of main memory are only available if they are either isolated (hidden from the Linux kernel) during the booting process of the complete machine or one can request many fragments and hope that some of them lie next to each other so that a larger chunk can be formed afterwards.

Figure 2.12 illustrates a typical memory allocation in a host system. The physical address

²update of read pointer in HW

space of the complete system is parted into segments. Each time the user application requests memory one such segment is mapped to the address space of an application.

Today's host interfaces like HT [25] [26], QuickPath Interconnect (QPI) [27] [28] or PCI Express (PCIe) [29] have a very high bandwidth well above five GigaByte (GB)/s and those four MB are therefore written in a split of a second, hence having larger segments is desired to avoid stalling the CPU due to credit problems or interrupt the working thread too often in order to free up parts of the receive buffer. To get a larger queue, restarting all machines to isolate memory is of course not feasible in large scale clusters with multiple running applications and it also cannot be guaranteed that requesting memory segments and checking whether or not they are consecutive results in the desired segment size.

Hence a unit which can utilize a plurality of memory segments and form one large virtual main memory ring buffer is desired. A building block called the Virtual Ring Buffer Handler (VHRD) has been developed which serves as an administrator for main memory ring buffers, enabling the hardware designer of FUs to focus on the development of the core functionality of the serving unit.

2.4.1 State of the Art

In 1999 an US patent [30] has been filed by IBM [31] that claims a plurality of ring buffers in a main ring buffer. Main memory was very limited in that time³, therefore this patent is targeted to synchronize a write and a read task with minimizing the need to swap to hard disk drives. A set of main read- and write-pointers point to the current sub-ring buffer and for each sub-ring a second set of pointers indicate the location within the current sub-ring. The whole buffer management for reading and writing is handled using SW as both users of the ring buffer are also SW threads.

There has been no further publications in the recent years regarding multi-segment ring buffers management in HW to the best of the writers knowledge.

2.4.2 Design Space Analysis

The VHRD needs to be capable of handling an arbitrary number of main memory segments to form one large virtual ring buffer and manage all necessary information to do so. Having this variability enables upper layer protocols (UPLs) to manage buffer space efficiently,

³around 16 MB

threads requiring large buffers can use multiple segments for their ring buffer, others can use only a single segment. In order to be able to reserve these main memory segments at any time during operation it shall not be required to have any alignment or size restrictions between different segments.

A plurality of FUs need to be supported, as well as multiple ring buffer for each FU in order to be able to fully support a virtualized device. In order to handle a large amount of memory segments and virtual ring buffers, an interface to read from main memory to reduce the required on-chip RAM is necessary as it is described in section 2.4.3 on the next page.

To reduce the on-chip memory requirements for the administration of these segments, only a few entries are to be cached on the device, the rest is stored in main memory. In order to read these administration information a main memory read interface is required.

Design decisions have to be made before developing such a building block concerning the location of the unit itself as well as the location of the memory read interface. Figure 2.13 shows a design space diagram with all possibilities regarding these two aspects.

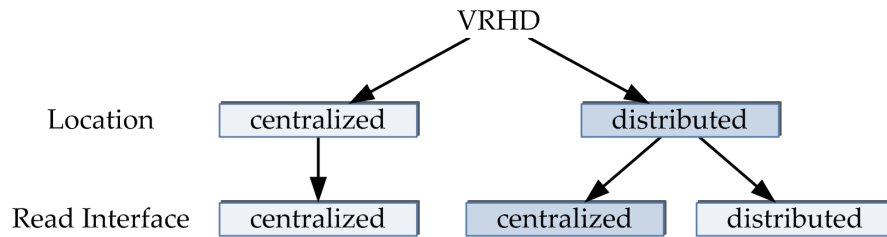


Figure 2.13: VRHD Design Space Analysis

A centralized location of the unit has the advantage that the interface to read from main memory exists only once, thus making good use of the read interface resources. The main drawback however is that only one single unit can request an address at a time, which limits the performance of the whole system, as some arbitration mechanism is necessary in case requests of different FUs overlap.

A distributed location increases the performance of the system as each FU is assigned to its own unit and can issue requests concurrently, removing the probability of overlapping requests. On the other hand, also distributing the read interface has the consequence that it will not be used very often and therefore requiring a lot poorly used resources.

To get the best performance and area results a hybrid approach with a plurality of distributed units which however share a single main memory read interface is to be

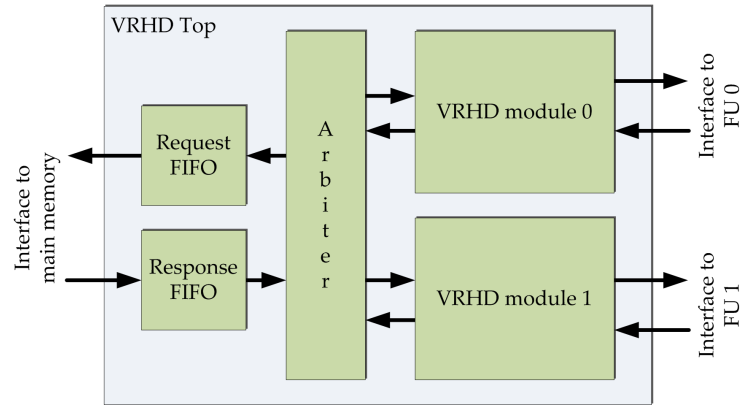


Figure 2.14: VRHD Block Diagram

avored. Each FU can request addresses with minimum latency, the not timing critical main memory read can share its interface. The necessary arbitration for reading from main memory of the different VHRD modules can be done using a round-robin arbiter. Memory read requests are not timing critical in this case. Their latency can be hidden completely, if at least two segments are locally stored.

Figure 2.14 illustrates a coarse block diagram of the VHRD as all design decisions have been made.

2.4.3 Implementation

The implementation of the unit is split into two parts. First part is the module handling all virtual ring buffers for one FU, the second part is handling a plurality of these modules.

As already stated, the VHRD module is able to handle multiple virtual ring buffers, each consisting of an arbitrary number of main memory segments. All of these segments however are separated into slots of fixed size to simplify address calculation as well as the interface to FUs. Having a fixed slot size ensures an efficient HW implementation of the VHRD module itself as well as serving FUs, due to the fact that this is the only way to ensure that a packet that will be generated for the currently requested address fits into the current segment. Thus, there are a few constraints for main memory segments. The first being its size has to be a multiple of the slot size as otherwise a support to split and span one packet over multiple memory regions is required, the second being that its base address starts aligned with the slot size. Latter constraint does not restrict the acquisition of memory segments in any way, since allocated memory space always starts at a four KiloByte (kB) border, as a page, i.e. four kB, is the minimum size of

allocatable memory. In the current implementation slot sizes of 16, 32, 64 and 128 Byte are supported. These values have been chosen since they represent commonly used message sizes for completion notifications of devices, HT or PCIe packets. There is no technological restriction to increase the number or size of supported slot sizes. However, the larger the slot size is, the higher is the chance of not using the buffer efficiently.

Each segment is described by a *descriptor element* containing its *base address*, the number of *slots* in this segment and the *slot size* itself as it is illustrated in figure 2.15. The width of the *slot size* field is two bit, as this is sufficient to encode the four supported slot sizes. *Segment Size* is 16 bit wide to be able to fully describe a maximum sized segment using 64 byte sized slots. Individual segments need to be smaller than four MB if smaller slot sizes are to be used. The last field inside a *descriptor element* is the base address of the segment. It is sufficient to only store the upper part of this address, as the lowest 12 bit are always zero due to the page alignment of such elements. The most upper 12 bit of the base address are not stored either, as today's CPUs only support physical addresses of up to 48 bit anyway, making this solution fit for the near future. As there are still some reserved bits left, the width of the *base address* field as well as the *slot size* field can be increased at later implementations. The alignment of the fields inside these elements have been chosen arbitrarily.

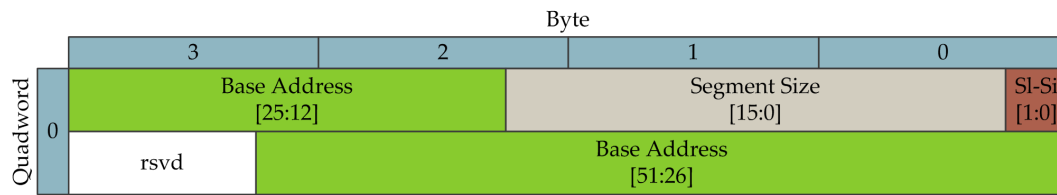


Figure 2.15: VRHD Descriptor Queue Element

These *descriptor elements* are again handled using multiple ring buffer structures, one for each served process. This structure, called the *descriptor queue*, is stored in main memory in order to minimize the on-chip RAM footprint of a single VHRD module. Only the *descriptor elements* for the currently used and the next segment of each ring buffer are cached locally to hide the latency that incurs to read the next valid segment from main memory. All address requests from the serving FU will be processed using the current segment. A new segment descriptor will be read from main memory once the end of the current segment has been reached. The locally cached next segment then becomes the new current segment and the read descriptor will be the new cached next segment once the read response arrives. *Queue full* will be returned in case a read response arrives too late and also the next segment has already been used completely. This has to be done in order to avoid packets being sent to an illegal address. In this case a second read request

2 Efficient Packet Processing

will be issued immediately and the first arriving read response will be directly used as the new current segment. FUs have to repeat requests until they get a response with a valid address. Insisting on repetition of requests makes it possible to implement a stateless VHRD module. Statelessness is in general a desired feature for modules as it simplifies HW significantly due to the fact that no contexts have to be stored and recovered.

The module itself is stateless, the queues on the other hand of course require registers to store the state of each queue. All these state registers exist for each ring buffer and are stored in a RAM to avoid a large sea of registers. To handle the descriptor queue itself it is necessary to know its *base address*, its *size* and the *read pointer* which points to the next segment to read as it is illustrated in figure 2.16. *Size* gives the number of available segments. It is more HW resource efficient to use a mask instead of a segment count for the *size*. However, this requires a power of two amount of segments and therefore loses a lot of the flexibility compared to using a count field, which can realize any number of segments. Once the descriptor queue *read pointer* reaches the number of segments it is reset to zero and the first segment will again be used, thus closing the virtual ring.

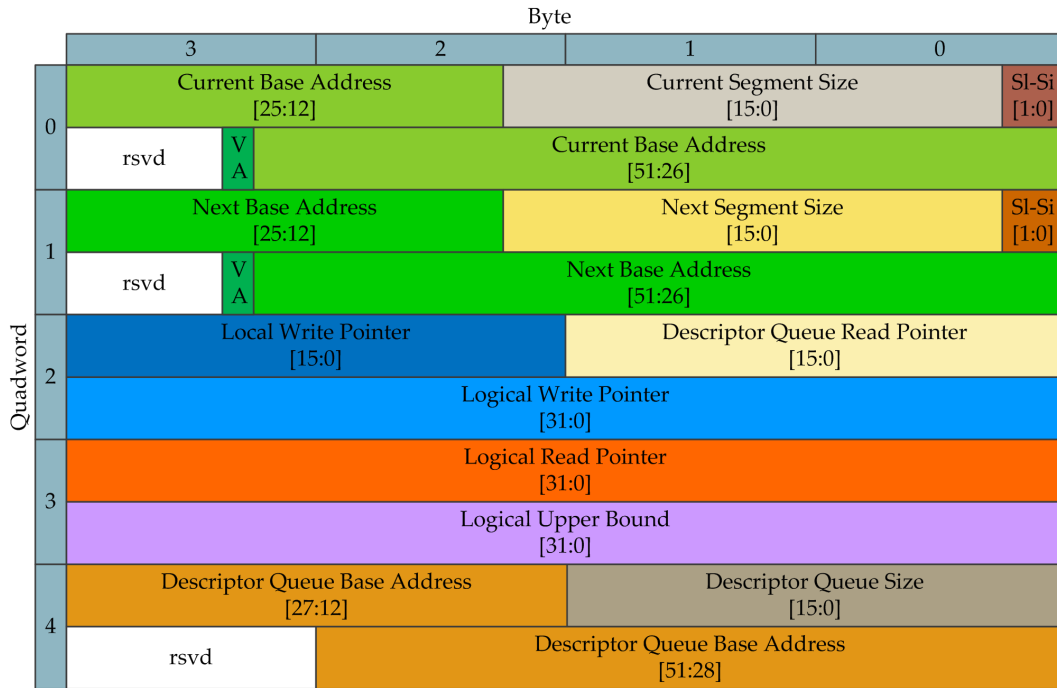


Figure 2.16: VRHD Queue State RAM

In addition to the fields required to handle the descriptor queue itself, the following information, also shown in figure 2.16, are required for each virtual ring buffer.

The *logical upper bound* field is the sum of slots over all segments. It is necessary to be

able to reset the *logical write pointer* to zero once it reached the end of the last segment. The *logical write pointer* is used to calculate the full condition of each queue, i.e. the flow control. It is incremented with each successful address request, spans over all segments and will be reset once the *logical upper bound* has been reached. The *logical read pointer* has the same functionality as the *logical write pointer* mentioned above. This pointer will only be set by SW and is used in combination with the *logical write pointer* to realize flow control, which is required to avoid a buffer overflow. The *local write pointer* points to the next free slot inside the current segment, is also incremented with every successful address request and will be reset every time a new segment is started.

The *current descriptor element* in quadword 0 describes the main memory region which is currently being used for writing. It includes all fields shown in figure 2.15 on page 29. *VA* indicates that the current descriptor element is valid and can be used. The VHRD module returns full if an address is requested and valid is not set, to avoid illegal memory writes sent from the requesting FUs. The *next descriptor element* in quadword 1 holds a pointer to the main memory region which will be used once the end of the current segment has been reached. *VA* is again an indicator that this descriptor element is valid and can be used.

As an additional feature enabling serving FUs to hide latency, it is possible to request addresses speculatively. This has the positive affect that an address can be requested as soon as a target is known and then revoke or commit the request at a later time when it is clear whether the address is required or not. To avoid data inconsistencies, every request has to be explicitly committed, otherwise it will be assumed as revoked after a timeout occurred.

A *mailbox full* indicator will be returned in case the *logical read-* and *write pointer* indicate a full condition (i.e. the SW has not read a buffer for a while) or the read responses for the segment caching has not yet arrived.

A simple interface is used to request an address for the next free slot of a specific queue/virtual process identification (VPID). The FU only has to inform its VHRD module for which VPID an address is required and later has to commit that the address has been used. The ring buffer management is fully transparent for FUs, they only need to be able to request an address again in case the virtual ring is full for the above described reasons.

The following example in figure 2.17 on the following page illustrates the usage of all fields and the functionality of a single VHRD module. It shows how the buffer is put together and what is stored in the different fields inside the *Internal State RAM*. The colors used for all fields in this example are the same as in figure 2.16 on the preceding page.

2 Efficient Packet Processing

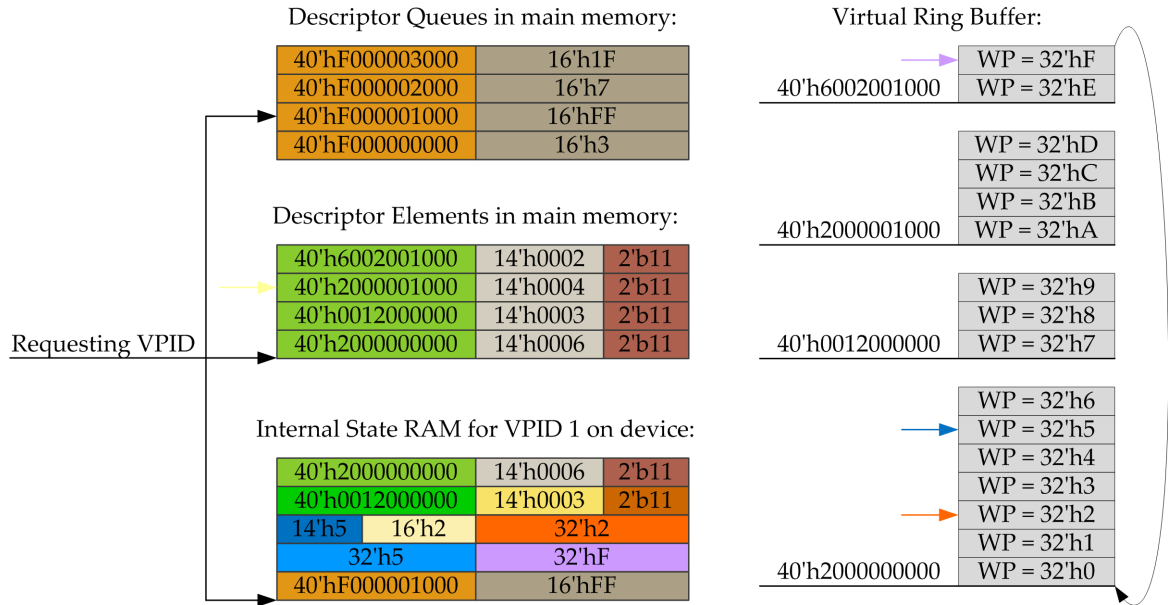


Figure 2.17: Example 1 for a Single Virtual Ring Buffer

An unit is requesting an address for *VPID 1*. The *logical read pointer*, who's value is two (in orange) and *logical write pointer*, its value is five (light blue) are more than one apart, so the virtual ring is not full at the moment. The current segment is not yet full either, since the *local write pointer* value is five (dark blue) and therefore not the same as *segment size*, which has a value of six (light brown). Consequently no new descriptor element will be requested and both, *logical*- and *local write pointer*, will be incremented after the request has successfully been responded. The returned address value for this request is *64'h2000_0000_0000_0028*.

The second example, illustrated in figure 2.18 on the facing page, shows the state of a virtual ring as an unit is again requesting an address for *VPID 1*. Only this time, the *local write pointer* has a value of *six* (dark blue) and therefore has the same value as *segment size* (light brown). A new descriptor element is required and will be requested as the current element will be completely filled after this request has been processed. The *logical write pointer* is incremented, the *local write pointer* on the other hand is reset. All *current descriptor element* fields are overwritten by the *next descriptor element* fields (first two lines in the *Internal State RAM*) and the new *next descriptor* values are written as soon as the response arrives. The new internal state is illustrated in figure 2.19 on page 34.

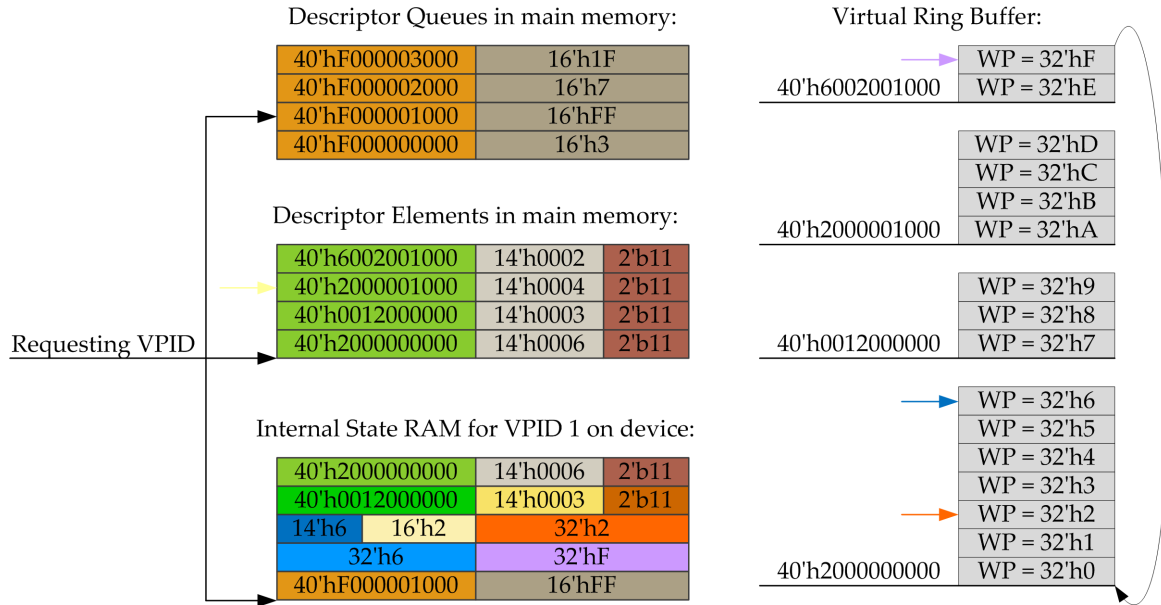


Figure 2.18: Example 2 for a Single Virtual Ring Buffer, View Before Request

2.4.4 Performance

Each VHRD module is capable of responding a new address requests every four clock cycles. Performance is limited as a complete read-modify-write access cycle is required for each address request. The second performance limiter are the RAMs themselves, as they have a read latency of one cycle, thus adding to the above mentioned four cycles.

Nevertheless, this is sufficient to fully utilize the HyperTransport Advanced Crossbar (HTAX) and completely satisfy the host interface bandwidth in the FPGA implementation of EXTOLL, even with minimum sized packets. In the ASIC implementation however, a minimum sized HTAX packet is only two clock cycles long due to the wider internally used interface at the host side. Therefore a single unit is no longer capable of fully utilizing the offered bandwidth. To resolve this issue, application SW needs to either utilize more than a single FU of EXTOLL or send larger packets.

2.5 Tag Matching Unit

Tag matching is an often required capability in high performance computing. All MPI [11] based communication requires the tagging of messages in order to be able to uniquely identify the communication partners. Until now, this task is handled using only SW.

2 Efficient Packet Processing

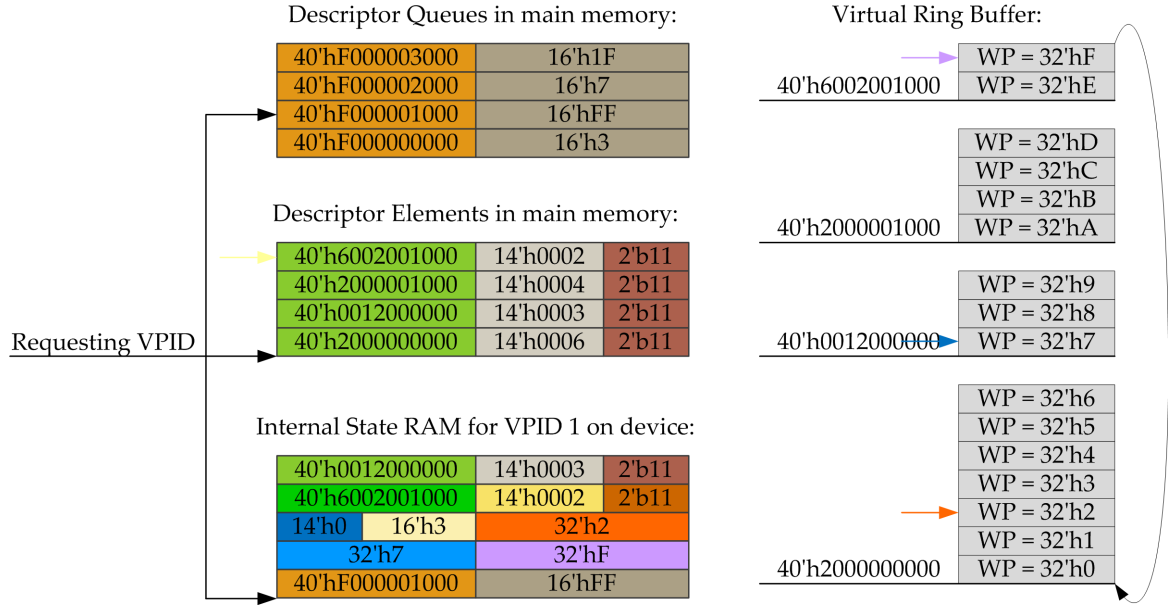


Figure 2.19: Example 2 for a Single Virtual Ring Buffer, View after new Descriptor Element has been stored

Essentially tag matching can be split in two parts, storing and maintaining a list of tags that have not yet been matched and comparing new arriving tags of messages against all stored tags. Both tasks are time consuming on a standard host CPU. Matching in SW requires either to go through all entries in a list sequentially or matching the entries in parallel. A sequential walk through the list is very time consuming and thus increases the latency of a packet significantly. The parallel approach on the other hand requires multiple CPUs, which are better utilized for solving the actual given task than work on communication tasks.

A specialized HW unit is required in order to be able to offload this task to an accelerator or into the NIC and therefore freeing the host CPU. The here proposed Tag Matching Unit (TMU) is a structure with very fast and efficient header matching capabilities in hardware, which is beneficial to improve such MPI based communication. Another advantage is the reduced amount of memory copy operations as messages can be delivered into main memory directly using a zero copy approach.

2.5.1 State of the Art

One paper that explains the problem of progress and computation/communication overlap in terms of MPI implementations for InfiniBand is [32]. In particular the problematics

of rendezvous protocols, both pipelined and true remote direct memory access (RDMA) based, in conjunction with synchronous progress are shown.

Myrinet MX implements matching on the LanAI processor on their NICs, so this is a firmware implementation [33]. Another firmware based (or at least possibly in firmware) implementation is based on the Portals interface [34] and Cray's SeaStar interconnect [35]. Quadrics used to implement matching on their NIC, it is believed that this was also firmware and embedded processor based [36]. Unclear is, how Infinipath's matching is performed [37].

On the research side, a number of papers have been published by the same group that is also responsible for Portals, which address the MPI matching problem. In [38] some interesting examination of the lengths of posted receive and unexpected queues can be found. In [39] an embedded processor implementation⁴ is presented. It is augmented by some hardware assistance for list traversing and management, which is used to implement the posted receive queue and the unexpected queue. Finally, in [40] a hardware architecture for tag matching is described that is based on an associative data structure, i.e. a modified TCAM structure. A prototype has been implemented in an FPGA. While a good performance in certain respects is shown in simulation, the hardware is very resource intensive and again geared towards the SeaStar chip.

Most current MPI implementations provide for synchronous progress (i.e. in-band polling) and host CPU based matching for networks like Ethernet, InfiniBand and shared memory transports (like MPICH [41], MVAPICH [42], OpenMPI [43]). A resource efficient, yet high performing HW structure however has not yet been published.

2.5.2 Implementation

To increase performance the proposed module is highly parallelized and implements a large number of matching elements. Arriving MPI messages can be categorized as either unexpected receive (UE) or posted send request (PR). UE messages are receive requests that have not yet been matched with a received message and PR are the exact opposite, a received message for a not yet posted receive request. By distributing the headers which are to be matched against a specific UE or PR over a large number of match units, the match latency and match rate are significantly improved compared to a software based approach. The overall structure of the TMU is shown in figure 2.20 on the following page.

⁴based on Cray's SeaStar interconnect

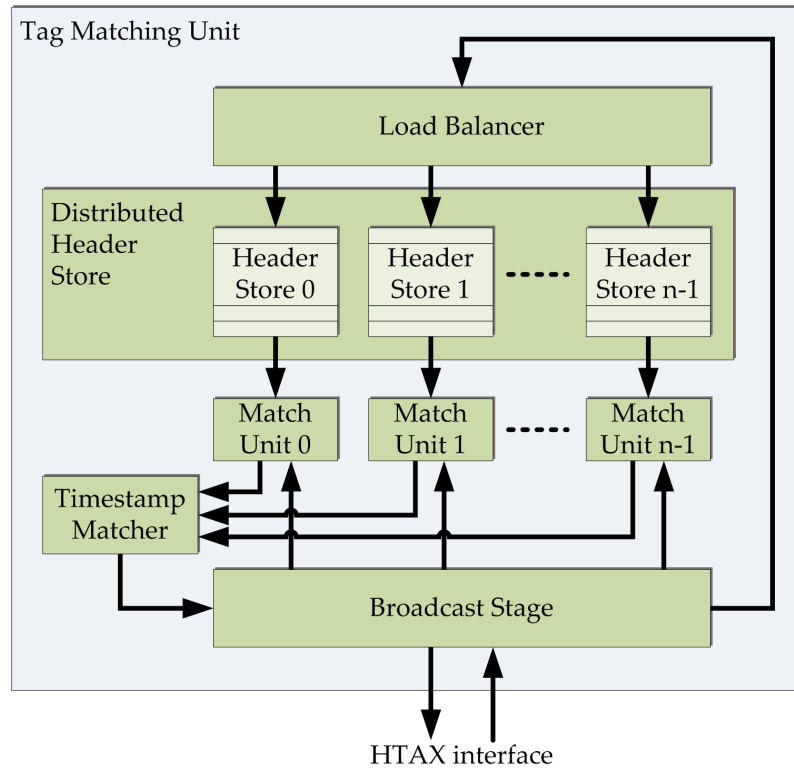


Figure 2.20: TMU Block Diagram

All incoming messages as well as outgoing matches are handled using the HTAX interface, which is described in section 4.1.2 on page 137. The functionality of each module depicted in figure 2.20 is described in the following sections.

Broadcast Stage

The *Broadcast Stage* receives all incoming headers and buffers them in a small FIFO to avoid blocking the HTAX. If there currently is no matching in process it forwards the first received message header to the *Match Units* and awaits the results. Only one match is processed at a time to avoid race conditions that otherwise can occur. Once the match is finished and a match has been found, then this match is sent back using again the HTAX interface. If there has not been a match, then the processed header is either an UE or PR and needs to be stored in the Distributed Header Store (DHS) using the *Load Balancer*. The next received header can be processed once the previously unmatched header has been stored properly. It has to be waited until the previous header has been stored to again avoid race conditions.

Distributed Header Store

A large *Header Store* is used to hold UE and PR headers. The *Header Store* is distributed over several queues implemented by a set of memory based FIFOs to improve the throughput of the *Tag Matching Unit*. Each single queue belongs to a specific *Match Unit* and each *Match Unit's* access is restricted to the headers in its dedicated *Header Store* queue. This approach avoids read and write conflicts as it does not allow several match units to access the same item of the DHS.

The number of *Match Units* and associated *Header Store Units* is easily scalable, a FPGA based implementation will probably have less queues as an ASIC implementation. The *Distributed Header Store* module itself does not contain any logic, it is just a container for all instantiated *Header Store Units*.

Header Store Unit

Each *Header Store Unit* is partitioned into two parts whereas one part stores all unmatched PRs, the Posted Send Request Queue (PRQ) and the other stores unmatched UEs, the Unexpected Receive Queue (UEQ). These parts are then further subdivided for the different processes, i.e. VPIDs. While it is easily possible to support a different number of processes and in case of fewer processes increase the queue size per process, the division in PRQ and UEQ is static as this differentiation as always to be made even for only one or two processes.

The two queues and their sub-queues are realized using a RAM, where the address is split into three parts. The first part, starting at the highest order bit, separates the two main queues, with the second part the queues per process can be distinguished and the third part is used for the entries of a single queue.

Alongside the actual header, additional information of configurable length can be stored in the queues which are required by units leveraging the TMU. The header itself contains all information required for the matching itself and is shown in figure 2.21 on the next page.

The *Tag* field is a user specified tag for the message and is compared between PRs and PRQ during matching. *Context* can be used to identify for example the used communicator type in MPI. *WC* is the wild-card field. One bit is used to mask the *Tag* field to enable matching of any tag. The second bit is used to mask the *Context* field to enable matching

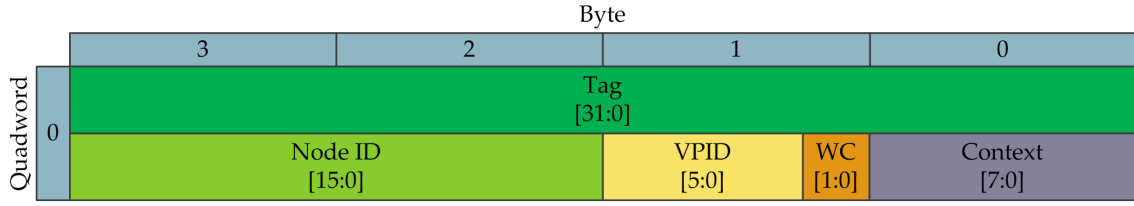


Figure 2.21: TMU Header Format

of any used context. The *VPID* field identifies the process at the sending or receiving node and *Node* field identifies said node.

As already stated, the header supports a variable amount of queues for applications that require fewer than the maximum supported number of ranks per node. When reducing the amount of supported ranks, the depth of each queue increases accordingly. The amount of needed queues can be set during configuration of the unit. Possible values are 2^n whereas $n \leq m$ so that 2^m equals maximum number of supported ranks. A change of the queue partitioning is only allowed while all queues are empty and idle, otherwise data corruption is imminent. The applied scheme for changing the size of the queues inside the RAM is the same as in section 2.2.1 on page 14.

All queues are organized as self compacting queues as it is also described in [44]. Values are added at the end of the queue in a chronological order and can be removed from anywhere inside the queue. Consequently, the first entry in a queue is always the oldest and a search can be stopped as soon as one match has been found.

To avoid gaps in the queue after a value has been removed from it, the queue is concurrently and permanently compacted. This can be realized with a dual ported memory where the values, which are inspected during a linear search within a match process, are permanently copied to the correct location via the second write port. The process of compacting the queue can be carried out in parallel while searching the queue. Compacting the queue has the advantage that there is no fragmentation and therefore the maximum amount of entries are always available. Otherwise gaps cannot be used as each queue itself has to be in correct chronological order at all times. Doing this compacting while searching ensures that there is at maximum a gap of one entry, the one that just matched. This gap of one will be removed upon the next search. Compared to [44] the self compacting mechanism is less complex, as each queue is accessed atomically. Consequently the queues are either read and compacted or written to.

During a search, all header stores are locked to avoid insertion of new headers, thus to maintain the order and avoid race conditions. While the result of the search can be

returned as soon as there is a match (which can occur at any location inside a queue), the search operations needs to look all entries to perform the compaction operation. In this course, the *age* field of each and every entry is incremented by one. The concept of *age* field is described later in section 2.5.2 on the next page.

Matching Units

Parallel matching of a single message provides several advantages. All *Match Units* operate on a single header simultaneously. By searching in parallel the search space is reduced by a factor of N whereas N is the number of *Match Units*. The search space is further reduced as only the queue corresponding to the active destination MPI rank is analyzed. The match rate or match bandwidth is equal to an approach that matches different messages in parallel. However, the match latency is improved significantly. Furthermore, memory conflicts and race conditions are avoided as the UEQ or the PRQ are only accessed by a single match unit. The crossbar in between the match units and the memory block is avoided as well and scaling the amount of match units is facilitated. Using self compacting FIFO data structures instead of linked lists allows for an easy pipelined, linear walk through the memory and fewer management logic.

For each match, all fields of the header and the communication type are taken into account. It is possible that an incoming request matches multiple headers from different *Match Units*. In this case the oldest entry is the correct match, the oldest entry being the first matched entry. Inside a *Match Unit* the order is ensured by the FIFO characteristics of the data structure.

Timestamp Matcher

The *Timestamp Matcher* compares the *sequence-* as well as *age numbers* added to each header by the *Load Balancer* of several successful matches. It determines the oldest match and notifies the corresponding match unit to remove and forward the entry to the *Broadcast Stage*. This unit then forwards the matched entry using the HTAX interface.

Load Balancer

The *Load Balancer* distributes UEs and PRs to the different match units and their corresponding header stores. To store a header the *Load Balancer* analyzes the MPI rank, i.e. the VPID of an incoming header and checks the full signals of the appropriate queues in the various header stores. The header is distributed following a round-robin like policy if multiple header stores have buffer space available.

In addition it maintains a sequence counter for each MPI rank. Each forwarded header is appended with the *sequence number* and the sequence counter is incremented afterwards. This *sequence number* allows to define a chronological order between the headers distributed over the various header stores. The sequence counter is reset whenever a match occurred, the *age* on the other hand is incremented for all entries that did not match and are still stored in the queues. As already mentioned, the *age* is directly managed by each *Matching Unit* itself. In summary, the *age* is used to maintain the order within a queue and the *sequence number* is to maintain the order across all queues. There are several scenarios that can occur when using a *sequence number*, *age number* scheme that need to be covered.

Sequence Overflow — The chronological order is not guaranteed anymore as soon as a sequence number overflow appears. Therefore, an overflow of the sequence counter has to be strictly avoided. If the sequence counter has reached its maximum value, all matching units are locked and a dummy match is run on the corresponding queues. This match operation will not return any actual result, but will increment all aging counters of the entries and the sequence counter can be safely reset to zero afterwards.

Age Overflow — When headers of one rank ever reach the maximum age, an exception occurs. This can, unfortunately not be handled gracefully by HW alone. All entries are then dumped to main memory and the host CPU is interrupted. At the same time, HW matching is disabled and handed over to SW, which may enable it again at any time.

Header Queue Overflow — A queue overflow will occur, when all header queues are full and another header arrives. Whenever this happens, further packet processing is stopped for incoming packets that target the overflowing queue and a host CPU is interrupted and informed with the overflowing queue descriptor. Incoming packets are blocked respectively forwarded to main memory. The host SW treats header queue overflows as a slow case. Hardware matching for the MPI rank is turned off automatically by HW, all headers are dumped to main memory and operation reverts to a SW header matching algorithm. This is actually the exact same mechanism as in the age overflow case. SW

may choose to enable matching again at a later time. Since the SW knows the size of the queues, it may also be possible to try to prevent this case using clever scheduling and flow-control during normal operation.

Disabled HW Matching — All headers are forwarded to a main memory queue where SW-based matching can be performed, if HW matching has been disabled for some reason, either because SW turned it off or one of the above mentioned exceptions occurred. This mode of operation is called bypass mode. The SW may choose to reinsert headers again into the queues and re-enable HW matching at any time. In this case, headers are inserted by the local host while matching is still disabled and the unit is locked. Incoming network traffic is blocked for that time to ensure that all SW headers are inserted first. After all headers have been inserted in chronological order, host SW can enable matching, and the unit resumes normal operation and the lock is reset.

When the HW dumps entries to main memory, they are not ordered as this makes the HW a lot simpler. The queues in each *Header Store* can be read one after the other without taking into account the *sequence-* and *age numbers*. The SW has to inspect all entries with respect to their *sequence-* and *age numbers* to reorder them accordingly. New entries that result from bypass mode will specify that in a special field and will not contain valid sequence/aging information any more.

In main memory, each entry requires the size of the actual header, which is eight bytes plus the size of the unit specific values. Additionally *sequence-* and *age* information are required to enable SW to reorder all entries.

2.5.3 General Operation Description

This section summarizes the general operation of the TMU. Incoming posted receives or posted send messages are forwarded to the TMU. The *Broadcast Stage* determines the type of the header and forwards it to the *Match Units*. The match of the incoming header versus all headers in the specific *Header Store Queues* is performed in parallel. Each match unit traverses its headers of the specific MPI rank and reports the first (chronological) match of that header by presenting the *age-* and *sequence numbers* of it to the *Timestamp Matcher*. After all *Match Units* have finished searching their queues the *Timestamp Matcher* determines the match with the oldest entry. The according *Match Unit* is informed and forwards its entry to the *Broadcast Stage*.

If no successful match exists the header is forwarded to the *Load Balancer* which inserts

the header into a *Header Store Queue*. During insertion no new headers are distributed by the *Broadcast Stage* to make the insertion atomic.

2.6 FAST Decoder

Another area where the reduction of latency is essential for the user is electronic trading. In today's major stock exchanges well beyond 90 % of all trades are executed electronically, while the traditional floor trading is in the meantime almost non-existent. The automation of trading services through electronic means provides significant advantages in terms of increased speed and reduced cost of transactions. Recently, not only the trading process, but also the trading decision making process has been automated in the form of algorithmic trading or HFT. According to the Aite Group, the impact of HFT on the financial markets is substantial, accounting for more than 50 % of all trades in 2010 on the US-equity market with a growth rate of 70 % in 2009 [45]. HFT describes a set of techniques within electronic trading of stocks and derivatives, where a large number of orders are injected into the market at sub-millisecond round-trip execution times [46]. HFT is particularly interesting for so-called *Market Makers* whose responsibility is to provide liquidity to the markets by providing quotes for buying and selling stocks. Thereby, they enable valuation of stocks at all times, even if there are currently no interested buyers or sellers, by providing liquidity in times of low volatility. High frequency traders aim to end the trading day 'flat' without holding any significant positions and utilize several strategies to generate revenue, by buying and selling stock at very high speed. In fact, studies show that a high frequency trader holds stocks for only 22 seconds in average [47].

In electronic trading, all interested parties receive a feed of stock related information from a so called feed handler which is usually the exchange itself. These information then need to be received and processed with minimum latency, a decision whether to buy or sell something has to be made and a trade request has to be posted back at the exchange.

To enable minimal round-trip latencies, a HFT engine needs to be optimized on all levels, beginning from the packet reception to the end, at the order transmission. The required low latency connection to the feed handler can be achieved through collocation, which allows servers to be deployed very close to the stock exchange. In addition, the feed needs to be internally distributed with minimum latency to the servers of the HFT firm.

One commonly used protocol in HFT is FIX Adapted for Streaming (FAST) [48], a standard based on the Financial Information Exchange Protocol (FIX), which has been

defined by the FIX Protocol Ltd. [49]. It is used by various exchanges worldwide to distribute stock information to all subscribing financial firms. The carrying network for FAST most commonly is *Ethernet*, which supports packet multicasting by leveraging Internet Group Management Protocol (IGMP).

FAST is very complex to decode in commodity CPUs and therefore predestined for a special purpose decoding HW. Section 2.6.1 describes the protocol itself and its pitfalls in order to be able to see why HW acceleration is beneficial.

2.6.1 FAST Protocol

FAST has been released in 2006. By then bandwidth has been very limited. Its purpose is therefore to reduce the required bandwidth of messages by dynamically reducing the size of transmitted fields using compression mechanisms and supporting some functions that are triggered after data decoding to avoid sending redundant data. Each transmitted message is encoded using predefined underlying *templates*, which have to be known both at the encoder and decoder side. These *templates* are formally defined using an Extensible Markup Language (XML) based syntax and are provided by the exchanges. *Templates* are a compilation of *fields*, *sequences* and *groups*, whereas *groups* are a compilation of *fields* that only occur once in a stream and *sequences* are a compilation of *fields* that can occur multiple times. All *fields* have a type assigned, which defines the decoding and handling of the *field*. Valid types are:

- 32 Bit wide signed integer (sint32)
- 32 Bit wide unsigned integer (uint32)
- 64 Bit wide signed integer (sint64)
- 64 Bit wide unsigned integer (uint64)
- decimal -
 - sint64 mantissa
 - sint32 exponent
- ASCII string
- unicode string
- byte vector

An example template is given in figure 2.22 on the next page. This template consists of four *fields*, whereas two of the *fields* are grouped together in a *group*.

```
<template name="This_is_a_template" id="1">
  <uint32 name="first_field"/>
  <group name="first_group" presence="optional">
    <sint64 name="second_field"/>
    <decimal name="third_field"/>
  </group>
  <string name="fourth_field" presence="optional"/>
</template>
```

Figure 2.22: FAST Example Template

Data compression is achieved by removing leading zeros or ones within each data word. For example only one byte is transmitted for a 64 bit signed integer with value 1 even if the actual value of course is 64 bit wide on the host CPU. All types except *byte vectors* use a so called stop bit encoding. There is no length information or prefix for *fields* using the stop bit encoding. The length is solely encoded using the eighth bit of each byte to mark the end of a *field*. Consequently only seven bit can be used for actual data transmission and *fields* have to be spread across multiple bytes if its value is larger than the coding space allows. Upon decoding, the stop bit needs to be removed for every byte and the remaining seven bit need to be shifted to the right. The shift operation however only applies if a *field* is larger than one byte and is not of type *string*. *Byte vectors* on the other hand are the exception. They have a length prefix and make use of the full eight bit of each byte.

Consider the following incoming binary stream: 10000111 00101010 10111111. Two *fields* are encoded in these three byte as it can be seen at the underlined stop bits. In order to decode the actual value of the first *field* it is sufficient to replace the eighth bit with a zero bit. The result is a binary value of 00111111, its hexadecimal value therefore is 0x3F. The second *field* however spans over two byte. Lets assume that this *field* is of type uint32, an unsigned integer with a length of 32 bit. To get the actual value of this integer, the stop bits need to be removed and the remaining bits of the two byte need to be shifted together and padded with two zero bits. The result is 00000011 10101010 binary or 0x03 0xAA hexadecimal. However, all stop bits only have to be replaced by zero bits in case the *field* type is a *string*, the result then is 0x07 0x2A hexadecimal.

Integer and decimal values have size restrictions by there given width definition. An *uint64* for example which is eight byte long after decoding can however span over 1 to 10 byte in FAST, thus making decoding complex. Strings on the other hand have no restrictions regarding their length.

Additionally, all *fields* also have an *operator* and a *presence* attribute. *Operators* are functions that can be applied after a field has been decoded.

The following list shows all available operators, which are used to reduce the required bandwidth by only transmitting some parts of the field and composing the actual value at the receiver using the currently and previously transmitted values.

- Constant
- Default
- Copy
- Increment
- Delta
- Tail

The *delta* operator for instance can be used to only transmit the difference between the previous and the current value for integers. Not all operators can be used in combination with all field types, explaining their purpose in more detail however is out of the scope of this work.

The *presence* attribute can either be *mandatory* or *optional*. Depending on this attribute, the field is always present in a data stream or may also be absent. The presence or absence of optional fields is described using a Presence Map (PMAP). Each transmitted message starts with such a PMAP followed by an unique identifier called the Template Identifier (TID), as it is shown in figure 2.23.

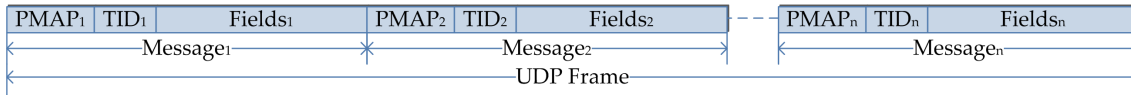


Figure 2.23: Fast Messages Encapsulated in UDP Frame

The PMAP is a mask and used to specify which of the defined *fields*, *sequences* or *groups* are actually present in the current stream. The usage of the PMAP is dynamic and depends on the combination of *operator* and *presence* attribute, therefore not every *field* occupies a bit in the PMAP. A *field* with applied *delta* operator for example never occupies a bit in the PMAP, whereas it depends on the *presence* attribute for fields with the *constant* operator.

If the *presence* attribute for a *field* has the value *optional*, then the field type also defines whether or not the transmitted value is *nullable*. This means, that a single byte with a certain value is present in the stream for this *field*, but its value is supposed to be not present. Zero is used as the *NULL* value and consequently all other values need to be decremented in order to get the actual *field* value in case the field type is a number.

Figure 2.23 on the previous page also shows that multiple FAST messages can be clumped together into a single UDP frame, which is the commonly used protocol in the transport layer. Care has to be taken as a single decoding mistake requires dropping the entire UDP frame, since there is no way of finding the start of the next message until the next frame starts.

While these techniques enable to keep up with the increasing data rates that are provided by the exchanges, it increases processing complexity significantly. To transform the compressed FAST data stream into processable data, the complete stream needs to be decoded and interpreted in real time. If at any time the processing system cannot keep up with the data rate, critical information is lost. Furthermore, by decompressing the data stream, the bandwidth that needs to be processed effectively increases. Hence, to develop a high performance fast decoder, two different goals need to be achieved. First, the decoding of the protocol need to be performed with the lowest possible latency. Second, it must be guaranteed that the arriving data rate can be processed at all times.

2.6.2 Related Work

An increasing volume of work has recently appeared in literature, though probably the largest part of work done by institutions is not published. A very good overview about the acceleration of high frequency trading is given in [50]. Furthermore, a system for accelerating *Options Price Reporting Authority (OPRA) FAST* feed decoding using Myrinet MX [51] hardware is presented. A multi- threaded *faster FAST* processing engine using multiprocessor machines is presented in [52]. While both approaches focus on accelerating FAST decoding, to the best of writers knowledge the following approach is the first one that deploys FPGA hardware for this purpose. Morris [53] presented an FPGA assisted HFT engine that accelerates UDP/IP stream handling similar to [54]. Sadoghi et. al. present an FPGA based matching engine [55], it however focuses on the latency improvement for matching algorithms, not the decoding of FAST. Finally, Pasetto et. al. also present an *OPRA FAST* decoder [56] using the IBM PowerEN Chip [57]. This is also a pure SW solution, based on [52] ported to the IBM PowerEN Chip.

2.6.3 Design Space Analysis

The FAST protocol is highly (bit-) serial in nature, as section 2.6.1 on page 43 shows. It makes bit manipulations, like masking and shifting necessary in order to decode it

properly. In addition to the complex decoding mechanisms itself, the protocol also offers a huge number of *field* combinations, thus requiring a highly flexible decoding structure.

There are a number of possibilities when designing a decoder for such complex protocols, as it is shown in figure 2.24. The general approach is to decode it using SW with its numerous options of programming languages, examples here are Java [58] or C [59]. Numerous SW solutions already exist for FAST all suffering in performance, due to the nature of the protocol.

A number of design possibilities also exist for a HW based decoder. One option is to write a full *case* decoder, which is usually very fast in respect to latency. Though, it will not reach a high clock rate, due to the complexity of the protocol. There are just too many possibilities of field combinations.

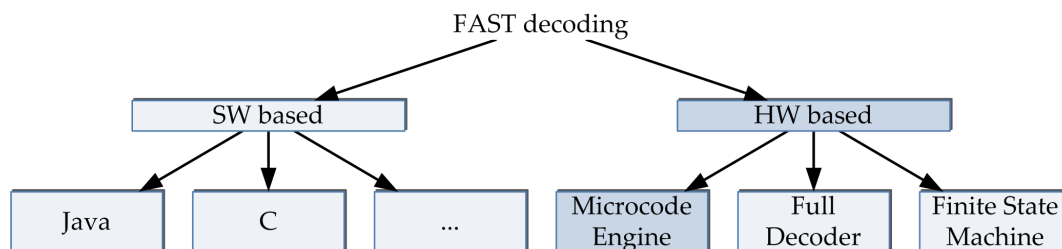


Figure 2.24: Design Space for FAST Decoding

The second option is to develop a FSM based decoder. The advantages are that FSMs can be designed using a higher abstraction layer tool like the *FSMDesigner* [60] [61], which already optimizes the register transfer level (RTL) code and checks for errors making the implementation a lot simpler and less prone to errors. Drawbacks are that, due to the complexity of the protocol, the resulting FSM requires a lot of states and transitions making it very hard to read and design. In addition it will not reach the desired performance values as well.

This leaves only one opportunity, a *microcode engine*. Microcode engines can be seen as a hybrid approach between SW and HW based decoding. It is a highly specialized and optimized CPU with only a very small set of instructions, therefore being able to decode the protocol much faster than a conventional CPU, even at lower clock rates. Compared to the first two approaches, the microcode engine requires a few more clock cycles to decode the protocol, but due to its simplicity is able to run at higher clock speeds as the other options and is more flexible.

However it also requires a program to be able to do so. These programs are called

firmware and are programmed into the engine upon initialization and are written in an higher abstraction layer protocol. Having this flexibility improves the ability to debug, as errors in the control- or data flow can easily be removed by changing only the firmware. Additionally, changes in the protocol itself, like template modifications, can be carried out in a short period of time. These template modifications actually appear twice a year in various stock exchanges.

2.6.4 Baseline Implementation

The first step has been to get a working prototype to see whether it is even possible to design an efficient enough decoding structure. This baseline implementation of the FAST decoder is split into three modules as shown in figure 2.25. These modules are decoupled using a FIFO buffer to be able to cope variable processing times, which depend on the input data.

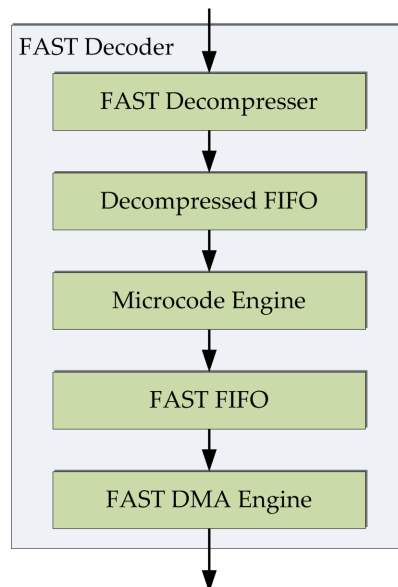


Figure 2.25: FAST Processor Block Diagram

The first module aligns all incoming *fields* to a multiple of a fixed word size. In most cases a single *field* fits into a single word, but some *fields* like *strings* require multiple such words. *Fields* are detected using the stop bits as described in section 2.6.1 on page 43. This is the first restriction for this decoder since *byte vectors* use a different encoding. However, this poses no real restriction since they are rarely used⁵. There are no information on the

⁵In Eurex, there is only one template with a byte vector defined. It is of fixed size and position and can therefore be removed before the actual decoding takes place.

type of the *field* necessary in this first step, all decoding of the *fields* themselves takes place in the microcode engine.

The *FAST Microcode Engine* uses pipelining to achieve a high throughput, i.e. high clock frequencies. The unit itself is again split into two logical paths, data- and control path. The control path uses only one pipeline stage to be able to take branches in the control flow with lowest costs. A low stage count is desired as each additional pipeline stage increases the branch penalty for such engines. Branches require two independent instructions or no operations (NOPs) after them to accommodate the program counter change and the command RAM read latency. The data path however uses four pipeline stages for an efficient decoding of the *fields* and its assigned *operators*. With the very short control path and the relatively long data path this design ensures a high issue rate of instructions and at the same time can reach a high clock rate.

Each cycle a new decoding request can be issued although it takes four clock cycles to decode a single value. In the first stage of the pipeline the type of the field, and in case it is an integer the sign of the value, is determined. The second stage removes the stop bits and pads the values with zeros or ones, depending on the sign and type of the current *field*. The third stage applies the operator using a previously stored value of the same field. Currently only operators that manipulate decimals and integers are supported⁶. If required, support to manipulate strings can easily be added in SW since all necessary information are known there as well. The last stage increments the value by one in case it is an optional, non negative integer to accommodate *nullable fields*.

Figure 2.26 illustrates an example data flow through the FAST decoder. First the UDP frame passes the *FAST Decompressor* where each data word is split in multiple words, in this example five. Thereafter, the *FAST Microcode Engine* decodes the *field* according to the microcode and the given *template*. All decoded *fields* are tagged with a unique identifier after passing the *FAST Microcode Engine* so that SW can process the data efficiently.

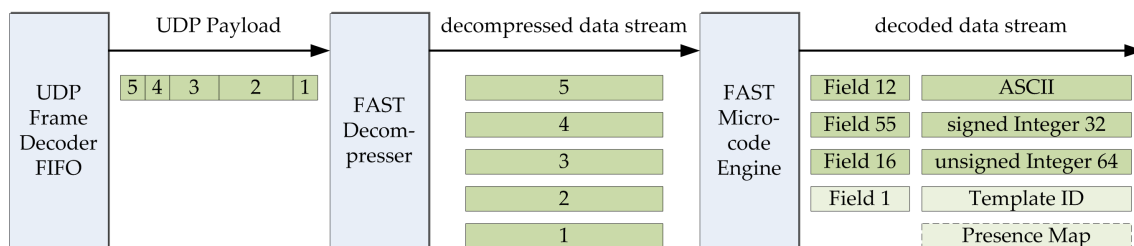


Figure 2.26: FAST Data Flow

⁶Again this poses no restrictions, as Eurex never uses *operators* on any other *field* types

2 Efficient Packet Processing

In order to process decoded data it is also necessary to write it to main memory where a SW thread can read it. This is done in the last stage of the decoder, the *direct memory access (DMA) Engine*. Reducing the latency of HW-SW synchronization is best done using user-level communication as already discussed in section 2.4 on page 24. The same mechanism, a ring buffer in the user-level address space, is used in this case. Data is written using 64 byte packets, as it is the commonly used size for a modern CPU's cache line.

Every packet written to the ring buffer consists of seven decoded words, each 64 bit wide and an eighth word containing the corresponding unique IDs of the seven words. The eighth byte of the last word is used for error detection and as an indicator whether the ring buffer has wrapped around or not. The highest order bit is inverted once the write pointer of the ring buffer reaches its maximum value. This enables SW to easily detect new data without the need to overwrite old values reducing the amount of required memory writes from the SW side significantly.

0B	Arrival Time of Packet							
	Template ID							
	Value of field 1							
	Value of field 2							
	Value of field 3							
	Value of field 4							
	Value of field 5							
	Value of field 6							
64B		6	5	4	3	2	1	FD
	Value of field 7							
	Value of field 8							
	Value of field 9							
	Value of field 9							
	Value of field A							
	Cycle Count				Parsing Time			
	empty							
128B		0	FF	A	9	9	8	7
	invalid data							
	invalid data							
	invalid data							
	invalid data							
	invalid data							
	invalid data							
	invalid data							
		IN	IN	IN	IN	IN	IN	IN

Figure 2.27: FAST Ring Buffer Usage

An example message dump is given in figure 2.27. For performance evaluation some additional values like the arrival timestamp or parsing time of the template can also be written to main memory.

The *arrival time* for example is a 40 bit clock cycle counter started after reset. It shows

the time a packet arrived at the *Ethernet Frame Decoder*. This counter will approximately overflow every 1.5 hours and has an accuracy of 6.4 nanoseconds. Parsing time is a 16 bit value which counts the clock cycles between start and end of decoding a template.

This first implementation is a fully functional FAST decoder that already increases the overall performance of FAST decoding significantly as it is shown in section 2.6.5 on the next page. Restrictions of the current design are that it does not fully support all possible *operators* and that there is no support for *byte vectors*.

Additional features have been added in section 2.6.6 on page 54 which again improve performance.

FAST Microcode Engine Firmware

In order to work properly, the *FAST Microcode Engine* requires a firmware, i.e. a sequence of commands. A small set of instructions have been specified which are sufficient to decode FAST templates and an assembler tool has been written which transforms the higher abstraction level programming language into binary code for the engine. Upon system initialization this microcode is then loaded into the program RAM of the engine. It is fairly easy to adapt the program whenever template changes are specified or decoding errors have been detected, using this dynamic programming approach.

Figure 2.28 on the next page shows an example code for the example template given in figure 2.22 on page 44. The program is divided into four columns, the first one is the command for the data path, the second column is the unique identifier each field is tagged with, the third column is the command for the control path and the fourth column is used for conditional or unconditional branch targets. Both relative or absolute targets are supported for these branch targets. Absolute branch targets can be identified using a name of choice as it has been done in the example with *START* and *TID1*. The code lines between these two identifiers is a part that is required for each template to first identify the correct sub-routine. The code lines starting from *TID1* are then the actual decoding sub-routine for the example template. Further explaining each command and its function is getting too much into the details of FAST and the decoder. The given example template however shows the dependencies that occur when decoding FAST. It has a lot of conditional branches and therefore requires a lot of NOPs. Optimizing and reducing these is part of section 2.6.6 on page 54.

1	:START			
2	SET_TIME	253	STORE_PRES	
3	SET_TID	1	STORE_TEMP	
4	NOP		INCR_PC	
5	NOP		INCR_PC	
6	NOP		JUMP_TEMP	
7	NOP		INCR_PC	
8	NOP		INCR_PC	
9	:TID1			
10	CON_U32_MAN	2	INCR_PC_DATA	
11	NOP		JUMP_PRES	+8
12	NOP		INCR_PC	
13	NOP		INCR_PC	
14	CON_S32_MAN	3	INCR_PC_DATA	
15	CON_DEXP_MAN	4	INCR_PC_DATA	
16	CON_DMAN_MAN	5	JUMP_EOFIELD_DATA	
17	NOP		INCR_PC	
18	NOP		INCR_PC	
19	NOP		JUMP_PRES	+6
20	NOP		INCR_PC	
21	NOP		INCR_PC	
22	CON_ASCII_MAN	6	JUMP_EOFIELD_DATA	
23	NOP		INCR_PC	
24	NOP		INCR_PC	
25	SET_EOT	255	JUMP_UNC	START
26	NOP		INCR_PC	
27	NOP		INCR_PC	

Figure 2.28: FAST Example Assembler

2.6.5 Baseline Performance

In order to measure the performance of the decoder efficiently a theoretical model that describes an upper bound for the data rate has been developed, taking into account the different compression mechanisms and processing overhead of the FAST protocol and its carrier protocol UDP.

$$D_{fast} = D_{feed} * \frac{1}{overhead} * comp \quad (2.4)$$

Equation (2.4) shows the sustained FAST data rate that is provided by the exchange. Note, that although peak data rates can be higher, only the sustained data rate is of interest as peak data rates can be absorbed with a buffer of sufficient size between the ETH core and the FAST decoder.

$Comp$ is the compression factor of the FAST protocol, and $overhead$ the overhead of the ETH- and UDP protocol. D_{feed} is the data rate of the incoming Ethernet data stream. Accordingly the maximum data rate that can be supported by the FAST decoder can be denoted as shown in equation (2.5).

$$D_{max} = CF * \frac{1}{CPI} * fieldsize \quad (2.5)$$

Wherein CF is the clock frequency, *cycles per instruction* (CPI) denotes the amount of cycles required to decode a single field or instruction and *field size* represent the number of bytes per field and instruction. An upper bound for D_{fast} can be given by choosing the theoretical maximum for the values D_{feed} which is currently the maximum data rate an Ethernet device can offer - 10 Gbit/s, and the maximum compression rate which is eight, as fields with a maximum width of 64 bit can be compressed to the size of a single byte. To provide a realistic estimation of the overhead an analysis of several GBs of market data feeds has been performed. While the overhead of UDP and ETH depends on the size of the packets a median overhead of about 10% for the ETH, IP, UDP headers and framing have been determined.

$$D_{fast,upper} = 10Gbit/s * \frac{1}{1.1} * 8 \approx 72Gbit/s \quad (2.6)$$

This results in a upper bound for the decompressed data stream of 72 Gbit/s as shown in equation (2.6). Measurements of real market feed data however show that current data rates for D_{feed} are closer to 2 Gbit/s and that the median compression rate is approximately four. Please note that in the baseline implementation every field is decompressed using 64 bit, even if the original data is only specified to be 32 bits wide. Hence a realistic sustained bandwidth can be calculated using equation (2.7).

$$D_{fast,realistic} = 2Gbit/s * \frac{1}{1.1} * 4 \approx 7.2Gbit/s \quad (2.7)$$

In the same way, the sustained data rate D_{max} that can be processed by the FAST decoder can be calculated. In the baseline implementation *field size* equals 64 bits and CF equals 200 MHz. One might suggest to increase one or both of these values to increase throughput. However, both appears to be difficult as FAST is a truly sequential protocol, processing of a field depends on the previous field, which renders it impossible to increase the data width by processing multiple fields in parallel. The operating frequency already represents

nearly the maximum that can be achieved with the current pipeline stage design of data- and control path and the deployed FPGA technology. While a deeper pipeline allows a further increase of the clock rate, the sequential nature of FAST ensures that there are only a very limited amount of instructions that can be executed independently. Furthermore, performance is mainly limited due to branches in the microcode. A deeper pipeline in the control path therefore is even counterproductive, as it increases the branch penalty.

The last parameter that can be analyzed is *CPI*. In particular, the analysis of the baseline architecture revealed that the FAST decoder in average requires 2.66 cycles to process a field, which leads to a maximum data rate of 4.8 Gbit/s as shown in table 2.3.

As D_{max} is smaller than $D_{fast,realistic}$ it is obvious that in periods of high volatility the proposed engine might not be able to cope the incoming data rate. As discussed, the only parameter that can be improved is *CPI*. Therefore, the following sections propose numerous different optimizations for improving this factor.

	CPI	Bandwidth (in GBit/s)
Baseline Implementation	2.66	4.8

Table 2.3: Performance of Baseline Implementation

The Cost of a performance enhancement can best be measured using the register and LUT usage of the decoder. The baseline implementation uses 2233 registers and 3695 LUTs for a single instance on the targeted Virtex-6 device as shown in table 2.4.

	Registers	LUTs
Baseline Implementation	2233	3695

Table 2.4: Costs of Baseline Implementation

2.6.6 FAST Decoder Improvements

As already mentioned, the performance analysis of the baseline implementation revealed that the proposed architecture is not capable of coping a real market stream under all conditions. A few bottlenecks have been discovered and the following subsections explain the improvements that have been implemented in order to increase data throughput and reduce the decoding latency.

Template Profiling

After analyzing the decoded data itself, it has been recognized that a large amount of the incoming FAST messages with the same TID also have the same PMAP. Figure 2.29 shows the distribution of incoming PMAPs over all messages for the reference data stream also used for the performance analysis.

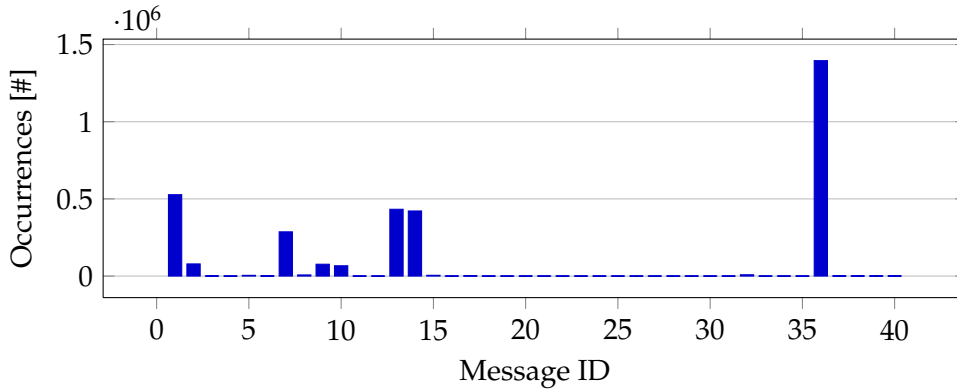


Figure 2.29: FAST Message Distribution

As it can be seen there are very few TID-PMAP tuples that occur considerably more often than others. Making the common case fast, which is the one with the most occurrences can therefore significantly improve performance. The PMAP uniquely identifies the whole path to take through a program of a template including all branches since it is a mask that shows the presence of each field. This can be accounted for when writing the firmware code. Adding a few optimized programs for certain TID-PMAP tuples and a general case for each template does not increase the required program space by much, but can improve performance. The HW has to be modified in a way that upon jumping to the template specific program, not only the TID, but also the PMAP is taken into account. If a program for the current tuple has been found, then the optimized version, otherwise the general version for this TID is used to decode the message.

The performance increased by around 13 percent by adding four of these optimized programs as it is shown in table 2.5 on the next page. With four slots for specialized templates all significant tuples can be covered, which is more than 87 percent of all messages. Adding more does not increase the percentage significantly in this use case. The fifth and largest bar on the far right in figure 2.29 is a template which cannot be optimized due to its definition without any branches or fields and has therefore not been considered in the calculation. Adding more specialized templates makes the branch condition more complex, making it difficult to meet the targeted timing constraint on a FPGA and has

therefore not been considered in the current implementation. However it is possible to add more if other exchanges have wider distributions.

	CPI	Bandwidth (in GBit/s)	Increase (in %)
Baseline Implementation	2.66	4.8	-
Template Profiling	2.35	5.4	13

Table 2.5: Performance of Template Profiling Implementation

The main drawback of this optimization is that it requires the engineers hand optimized microcode in order to work, which has to be developed for every tuple and does not work anymore if any changes in the template definition occurs.

The cost of this improvement only increases by 1.8 percent for registers and 4.2 percent for LUTs as shown in table 2.6, thus making this optimization profitable.

	Registers	Increase (in %)	LUTs	Increase (in %)
Baseline Implementation	2233	-	3695	-
Template Profiling	2275	1.8	3851	4.2

Table 2.6: Costs of Template Profiling Implementation

Pointer Prefetching

Another optimization is to reduce the time it takes to jump to the actual program by speculatively pre-fetching the next *program counter* between the first two submodules of the *FAST Decoder*. This does not improve the latency for the first message, but all following messages benefit hereof because the lookup latency can then be hidden due to the fact that the first of the two units, the *FAST Decompressor*, is able to process new fields every 1.1 cycles and is therefore faster than the microcode engine which currently requires 2.35 cycles per field.

The performance increases by another 12 percent in respect to the performance of section 2.6.6 on the preceding page as shown in table 2.7 on the next page.

Costs however increase significantly as illustrated in table 2.8 on the facing page. This large increase is not due to the rather simple pointer pre-fetching itself, but due to the combination with the template profiling optimization. This optimization requires to not

	CPI	Bandwidth (in GBit/s)	Increase (in %)
Baseline Implementation	2.66	4.8	-
Template Profiling	2.35	5.4	13
Pointer Prefetching	2.10	6.1	12

Table 2.7: Performance of Pointer Prefetching

only address a RAM every cycle, but also requires the PMAP and the TID to be present in the same clock cycle. This is required in order to be able to create the tuple and check whether one of the special templates is to be taken or not. Since the PMAP occurs in front of the TID in the data stream and there is the possibility of having a gap between the two fields⁷ as described in section 2.6.1 on page 43, it is difficult to store these two values without a complex additional logic. To avoid such complex logic between the two units, it is beneficial to modify the *FAST Decompressor* to ensure that both values can be interpreted in the same cycle at all times. Moreover, these changes can also be leveraged in the optimization described in the following section.

	Registers	Increase (in %)	LUTs	Increase (in %)
Baseline Implementation	2233	-	3695	-
Template Profiling	2275	1.8	3851	4.2
Pointer Pre-fetching	2882	26.7	4924	27.9

Table 2.8: Costs of Pointer Pre-fetching

Immediate Integer Decoding

Due to the stop bit encoding, sint64 and uint64 can span over ten byte on the wire, which is wider than the data bus width of the FAST decoder. In the current design the internal data bus width of the FAST decoder is balanced with the width of the main memory write- and the 10G-Ethernet receive data bus widths, which is eight byte. This makes it necessary to process 64 bit integers in more than one cycle. In fact at least three cycles are necessary since the end of the field has to be detected and a branch is required if the field is not finished yet, resulting in a six cycle delay for large numbers.

The third optimization targets this issue by increasing the data bus between *FAST Decompressor* and *FAST Microcode Engine* to ten byte, making it possible to decode integers

⁷in case the forward flow control issues a stop condition or the *FAST Decompressor* requires a gap

in a single cycle. The *FAST Decompressor* adds the first three byte of the next field to the current field and repeats these three byte in the next cycle in case it was not an integer with more than eight byte. Replicating these three byte has the advantage that no byte shift operations have to be done in the microcode engine in order to form the next valid field and all fields start without an offset, also making decoding less complex. This optimization still makes it necessary to remove the second word from the FIFO, however it avoids the otherwise required branch, thus reducing the three to six clock cycle penalty (branch taken or not taken) for integers to deterministic two cycles.

Since integers are the most common field in the template definitions this optimization has a great impact on the performance. Another 24 percent increase compared to the previous optimization can be achieved as shown in table 2.9.

	CPI	Bandwidth (in GBit/s)	Increase (in %)
Baseline Implementation	2.66	4.8	-
Template Profiling	2.35	5.4	13
Pointer Pre-fetching	2.10	6.1	12
Immediate Integer Decoding	1.70	7.5	24

Table 2.9: Performance of Immediate Integer Decoding

The consumed resources however only increase insignificantly as some of the required logic for this feature has already been implemented for the previous improvement. The amount of required LUTs actually dropped by 0.6 percent, as the *FAST Microcode Engine* is now less complicated, since less corner cases for the integer handling are required. The required amount of registers increases by 4.3 percent as shown in table 2.10.

	Registers	Increase (in %)	LUTs	Increase (in %)
Baseline Implementation	2233	-	3695	-
Template Profiling	2275	1.8	3851	4.2
Pointer Pre-fetching	2882	26.7	4924	27.9
Immediate Integer Decoding	3007	4.3	4894	-0.6

Table 2.10: Costs of Immediate Integer Decoding

Using a Two Shift-Out FIFO

The penalty of requiring two cycles to process integers instead of only one cycle can be avoided by using the FIFO implementation described in section 2.2.3 on page 20 in front

of the *FAST Microcode Engine*. An input that signals to increment the read pointer by two instead of the usual one makes it possible to jump over the unnecessary word inside the FIFO, thus avoiding the otherwise required second shift out cycle. This can of course only be done if the second word actually is stored in the FIFO, otherwise data corruption of the FAST message is imminent.

	CPI	Bandwidth (in GBit/s)	Increase (in %)
Baseline Implementation	2.66	4.8	-
Template Profiling	2.35	5.4	13
Pointer Pre-fetching	2.10	6.1	12
Immediate Integer Decoding	1.70	7.5	24
Two Shift-Out FIFO	1.54	8.25	10

Table 2.11: Performance of Using a Two Shift-Out FIFO

The CPI can be reduced to 1.54 resulting in a bandwidth capability of 8.25 GBit/s as shown in table 2.11. Costs however only increase by 0.5 percent for registers and actually drops for LUTs as illustrated in table 2.12, again due to less corner case handling requirements.

	Registers	Increase (in %)	LUTs	Increase (in %)
Baseline Implementation	2233	-	3695	-
Template Profiling	2275	1.8	3851	4.2
Pointer Pre-fetching	2882	26.7	4924	27.9
Immediate Integer Decoding	3007	4.3	4894	-0.6
Two Shift-Out FIFO	3025	0.5	4789	-2.1

Table 2.12: Costs of Using a Two Shift-Out FIFO

All optimizations combined achieve a performance gain of over 72 percent compared to the baseline implementation and only require around 35 percent more registers and 30 percent more LUTs. The achievable bandwidth can be increased to 8.25 GBit/s, thus achieving the required throughput for today's sustained bandwidth requirements.

A few more optimizations were thought of, but not implemented and are discussed in the following sections.

Speculative String Decoding with EOF Lookahead

Strings have no length limitation as already mentioned in section 2.6.1 on page 43. Therefore decoding them always requires a branch to detect the end of the field and repeat the current operation in case it is not completed. The example assembler code for decoding a string can be seen in figure 2.28 on page 52 in the lines 22–24. There are always two NOP commands required in order to decode strings properly. Adding a look ahead function to detect not only the end of the field in the current word, but also the end of the field in the next word and adding a command to speculatively decode the next word as a string, but be able to revoke the result in case it was not part of the string, removes one of the two NOP operations, thus reducing the required branches by a factor of two.

Adding even a three cycle lookahead and therefore removing all NOPs can improve the overall performance even more. However, it adds more complexity to the *FAST Microcode Engine* and the *FAST Decompressor*. In addition, most strings are smaller than 16 characters, hence they fit into two words and therefore make the possible gain smaller than the additional effort.

Sequence Length Pre-decoding

Analogue to the integer decoding, the additional bytes of the next field can also be used to pre-decode the length of a sequence. At the beginning of each sequence, the amount of repetitions have to be checked resulting in a branch and two NOPs afterwards to accommodate the branch penalty. By interpreting the length one cycle earlier this penalty can be reduced to one cycle giving an improvement of 33 percent for sequences.

Overall Performance

The plot given in figure 2.30 on the next page summarizes the bandwidth capabilities of the above presented FAST decoder implementation. A performance improvement of over 70 percent has been achieved compared to the results of the first implementation (baseline). The data rate $D_{fast,realistic}$ of around 7.2 Gbit/s can now be achieved, thus ensuring that even in times of high volatility, all arriving packets can be decoded properly without the risk of losing data.

To put the presented results in perspective, a comparison of the FPGA based FAST

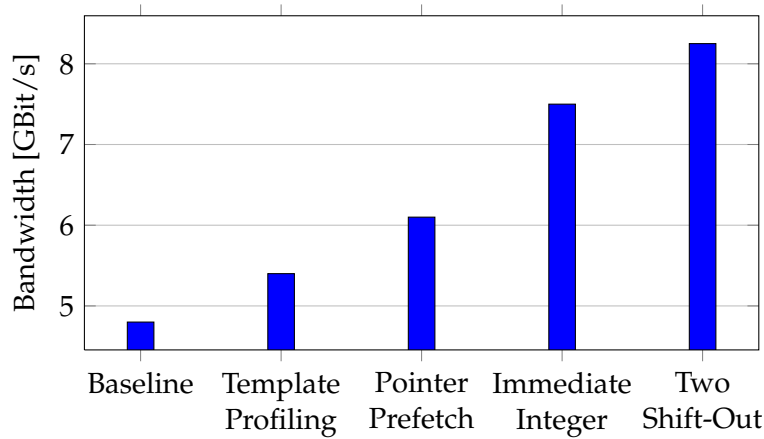


Figure 2.30: FAST Decoding Bandwidth Capabilities

decoder to conventional software based CPU approaches has been made. As reference, performance results from [53] that have been measured on an *Intel Xeon E5472* and an *IBM Power6*, as well as results from [57] measured on an *IBM PowerEN* are used. All these implementations have in common that they are software based FAST decoders. An average message size of 21 byte is assumed as it is done in [53].

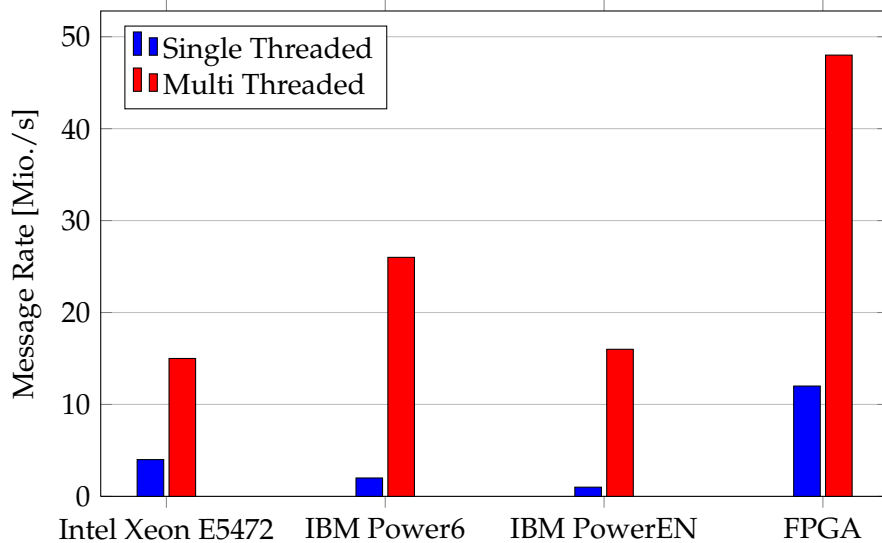


Figure 2.31: FAST Bandwidth Comparison [53][57]

Figure 2.31 shows the maximum message throughput that can be achieved on the three different architectures. The figure shows a single thread and a multi-threaded performance for all four architectures. The multi-threaded performance values for the different architectures are based on four threads on a *Intel Xeon E5472* and FPGA as well as 16 threads on a *IBM Power6* and *IBM PowerEN*.

The CPU based approaches benefit from multi threading by processing multiple, but independent FAST streams in parallel as a single stream cannot be processed in parallel. Nevertheless, the proposed FPGA based decoder outperforms the software based approaches significantly. The single thread performance is about one order of magnitude higher than the CPU approaches and is still twice as fast as the IBM implementation with 16 threads. Furthermore, the multi-core approach can also be leveraged for the FPGA design to further increase performance, by instantiating the proposed FAST decoder multiple times.

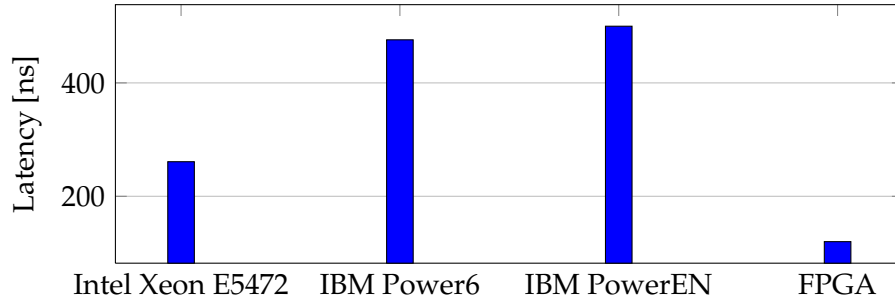


Figure 2.32: FAST Message Decoding Latency Comparison [53][57]

The limiting factors for this approach is first the size of the FPGA and the timing that can be achieved using multiple of these units. Putting four of these decoders in a single FPGA however is possible, still reaching the targeted timing of 200 MHz. In addition, the host interface bandwidth soon becomes the limiting bottleneck, since the decoded FAST stream is significantly larger than the incoming UDP stream. Nevertheless, again the FPGA is almost twice as fast using only four threads compared to the *IBM Power6* implementation leveraging 16 threads.

Latency wise the proposed architecture also is considerably faster than the other two SW based approaches as shown in figure 2.32. Decoding a single FAST message of 21 byte with an average compression rate of three takes around 120 ns in the here proposed FAST decoder compared to 261 ns on an *Intel Xeon E5472*, 476 ns on an *IBM Power6* or 500 ns on an *IBM PowerEN*.

3 High Performance Switching

An interconnection network is the key element for every supercomputer as it is required to connect multiple nodes to form the complete computer. Each interconnect requires some form of switching fabric, a module which is capable of selecting multiple inputs at each output. The fabric itself again is composed of switching elements.

3.1 Switching Elements

Switching elements are the central component for every interconnection network. Independent of the topology, protocol or application, at some point in the network a routing module, i.e. a switching element is required to connect multiple hosts, CPUs or any other type of communication engines or units. According to [62] these elements can be classified as depicted in figure 3.1.

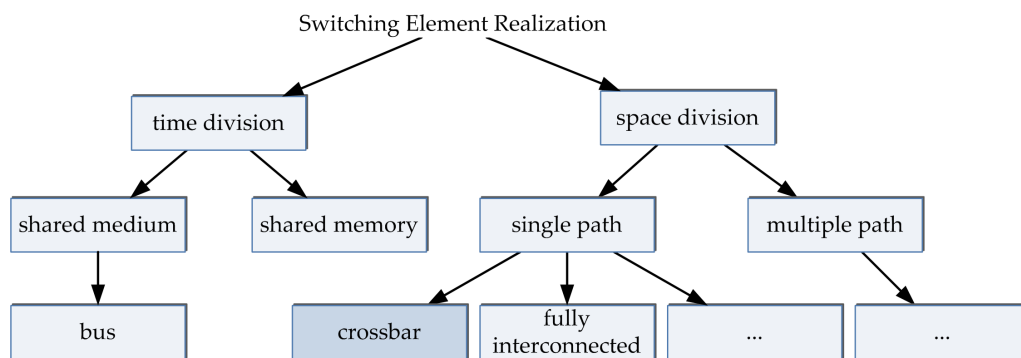


Figure 3.1: Design Space Diagram for Switching Elements [62]

3.1.1 Time Division Switching Elements

When using *time division switching elements* all units send packets on different times, but use the same resource.

3 High Performance Switching

This resource can either be a memory or a common medium. Multiple units accessing the same memory makes it necessary to either have a RAM with N ports, whereas N is the number of units or this RAM must be clocked at a N times higher clock rate to ensure that all write- and read accesses can be processed without blocking each other. Both possibilities cannot be realized using standard CMOS technology.

Bus

The most common shared medium switching element is a *bus*. Figure 3.2 depicts an example configuration for such a bus based switching element. All connected units share a common connection which is used for both sending and receiving data. Only one unit can transmit at a time and all units receive all transmitted packets. Whenever multiple units want to transmit data, either conflict avoidance or detection has to be implemented. For conflict avoidance a bus master can either assign fixed time slots to each unit where it is allowed to send packets or units can explicitly request a transmit and the bus master notifies the units when to send.

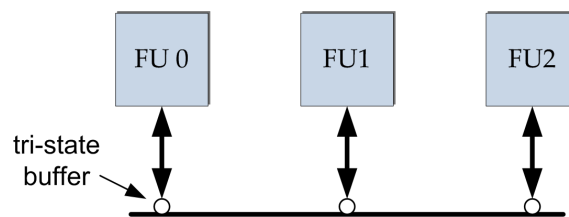


Figure 3.2: Connecting Multiple Units Using a Bus

When using conflict detection, every unit starts transmitting whenever it has packets to be sent. An initiation sequence is sent before the actual packet to detect whether the bus is currently in use or not. If a conflict occurred, the transmission is aborted and started again after a random period of time.

The advantage of a bus is the low complexity and effort required to connect a series of units. New units are simply added to the bus at one end.

The drawbacks on the other hand are that with every added unit or node the length of the bus, and with it its capacity, increases. This lowers the frequency at which the bus can be operated, thus reducing the achievable bandwidth and increasing the latency. Consequently, the bandwidth is reduced and latency is increased. In addition, the bandwidth is effectively reduced for every added unit, as it has to be divided for all connected units or nodes.

An example of an interconnection element that leverages a *bus* is the *hub*. It basically is a packet replicator where incoming packets on every inport are sent to all outports without analyzing the actual target. This leads to a lot of traffic on the connected units and consequently to an inefficient use of bandwidth. If packets cannot be transmitted due to a congested link, they are dropped and retransmitted after a timeout. *Hubs* were popular in the early days of ETH switching, but no recent interconnection network use this kind of interconnection elements anymore due to its obvious drawbacks.

3.1.2 Space Division Switching Elements

In *space division switching elements* units have multiple paths from its input to the various outputs. Therefore all internal logic of the switching element can run at the same clock rate as the unit itself.

This class of switching elements can be further divided into *single path elements*, if only one path between an input output pair exists and *multi paths elements* if multiple paths exist.

Crossbar Switching Element

One particularly interesting element in the *single path element* class is the *crossbar switching element*, also known as *crossbar fabric*.

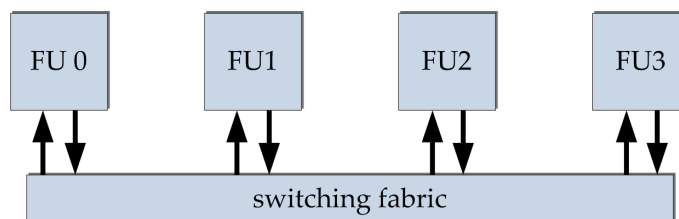


Figure 3.3: Connecting Multiple Units Using a Crossbar

Each unit has a bidirectional interface to the switching element as depicted in figure 3.3. This switching element is capable of simultaneously connecting pairwise any inport and outport of an unit under the constraint, that every port can only be connected once at any given time. Therefore multiple packets can be sent simultaneously over the crossbar fabric as long as connected units do not require transmitting packets to the same output.

The crossbar element uses multiplexing structures to establish those connections as

depicted in the *outport* part of figure 3.4. This type of switching element offers a high bandwidth that does not decrease with each added unit due to the various connections that can be established and the resulting independent packet flows and at the same time is more resource efficient than fully interconnected units.

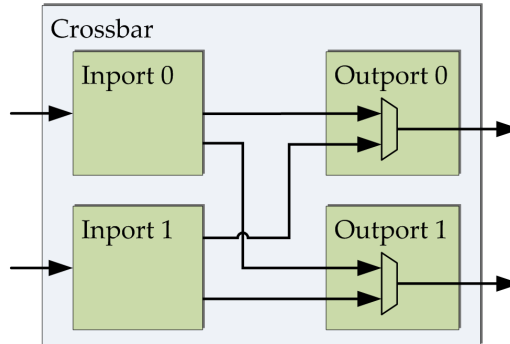


Figure 3.4: Detailed View of a Crossbar

3.2 Crossbars as Interconnect Switches

Due to its characteristics, the *crossbar switching element* lends itself to be used as a basic element for interconnection networks.

To avoid confusion as many definitions for the term *crossbar* exist [63] [62], the whole unit as depicted in figure 3.4 including its multiplexing structure is meant whenever the term crossbar is used in this thesis.

3.2.1 Crossbar Buffering

Crossbars are generally characterized by their buffering scheme and can be divided into five types as depicted in figure 3.5 on the next page.

The first possibility is to use no buffering at all. When there is no buffering in the crossbar it must be ensured that all arriving packets can be forwarded to the appropriate output under all circumstances, otherwise packet loss is inevitable. This of course cannot be ensured as all inports can send packets to the same output at the same time making it necessary to clock the outputs and the unit behind at N times the frequency, whereas N is the number of inports, in order to be able to cope the incoming packet stream. The ratio of

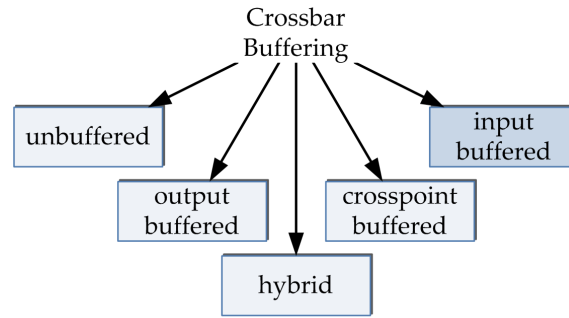


Figure 3.5: Crossbar Buffering Design Space Analysis

the required higher clock frequency in regards to the normal operation frequency is called *speedup*.

Output buffered crossbars suffer from the same problem as unbuffered crossbars where N packets can arrive simultaneously. However due to the inserted buffers at the output only the fabric itself and the buffer must be clocked faster, the connected unit itself can run at its normal clock rate. There are no examples of solely output buffered crossbars as the required internal speedup is hard to realize.

Crosspoint buffered crossbars have small buffers at each crosspoint inside the fabric. Such a fabric is depicted in figure 3.6. Instead of having only the enable logic of the multiplexer, each crosspoint has an additional buffer. This buffer avoids the necessity of an internal speedup requirement as packets can be buffered in front of each output if it is blocked.

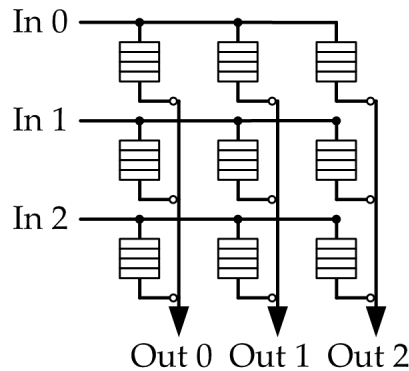


Figure 3.6: Detailed View of a Crosspoint Fabric

[64] for example implemented such a crosspoint buffered crossbar called *CICQ*. The drawback of such implementations is the amount of buffers required. N times M buffers are required for a crossbar with N inputs and M outputs.

The most common type of crossbars are input buffered crossbars like *Tiny Tera* [65]. All incoming packets are buffered at the input first and transmitted to the corresponding output only if it is idle. This has the advantage that no *speedup* is required inside the fabric and only N buffers are necessary, whereas N is again the number of inports.

Finally all the above mentioned buffering schemes can be mixed forming a hybrid type. Examples for such crossbars are [66] and [67].

3.2.2 Crossbar Scheduling

The second and third criteria crossbars are classified by is the applied scheduling algorithm and the packet length.

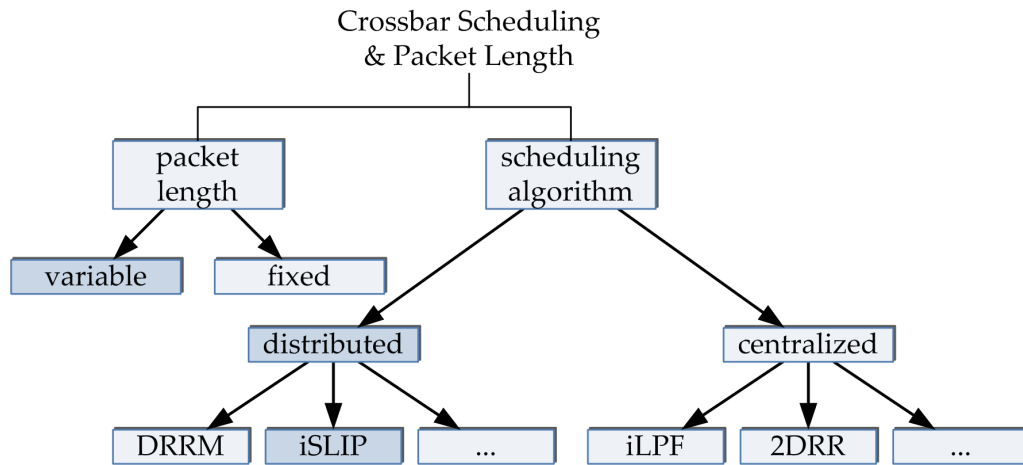


Figure 3.7: Scheduling and Packet Length Design Space Diagram

As figure 3.7 indicates, the packet length can either be of fixed or variable size. Scheduling on the other hand can either be centralized or distributed.

In centralized arbitration a single unit is responsible for matching all incoming requests to the appropriate outputs. Therefore it is only applicable if the packet size is fixed, so that arbitration can be done in fixed time slots. Otherwise the centralized arbitration gets very complicated, as it has to start every time a transmission to any output is complete, making it necessary to not only take the source and target of the upcoming transaction into account, but also the current status of all units. Consequently, centralized arbitration algorithms are more complex, as not one, but all outputs need to be taken into account [68]. Examples for a centralized algorithms are *iLPF* [69] or *2DRR* [70]

In distributed arbitration every output has its own arbiter, all working independently. Consequently this allows a fixed as well as a variable packet size. Distributed arbitration allows simpler and therefore faster logic for the individual arbiter. Examples here are *DRRM* [71] or *iSLIP* [72].

A good overview of scheduling algorithms including a complexity comparison is done in [68]. Another survey of available algorithms is given in [73].

3.3 Topologies

Only a short introduction to topologies is given, as it is not the scope of this thesis to evaluate the best suited topology for a network. To find a topology that works best for a network is highly dependent on the workload and use case of the resulting cluster. Nevertheless, a basic knowledge is required to understand some design decisions that will be made later. Figure 3.8 gives a coarse overview of available topologies. Basically two classes of topologies exist, *direct* or *indirect* connected networks.

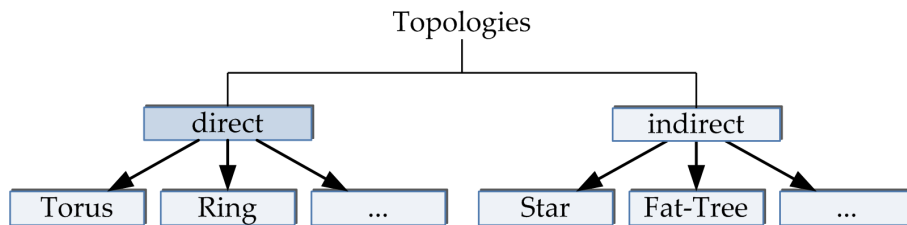


Figure 3.8: Topology Design Space Diagram

In Indirect connected networks central or multi-level switching units are present connecting each host. The *star* or *fat-tree* topology are examples for that type, illustrated in figure 3.9 on the next page.

In direct connected networks the switching element is distributed and integrated among all nodes. Examples are the *2D-torus* or a *ring* as shown in figure 3.10 on the following page.

For more information about topologies please refer to [63].

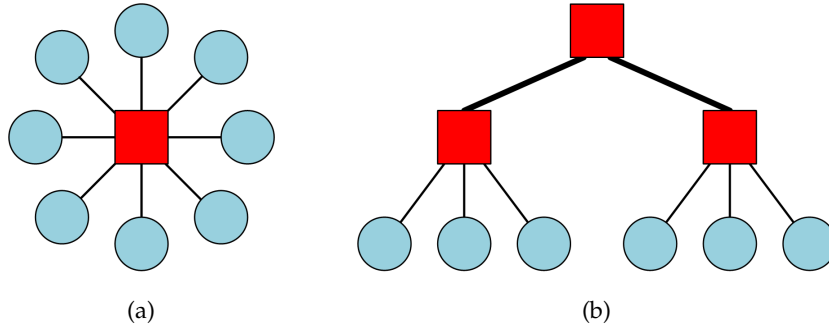


Figure 3.9: Examples for Indirect Topologies (a) Star, (b) Fat-Tree

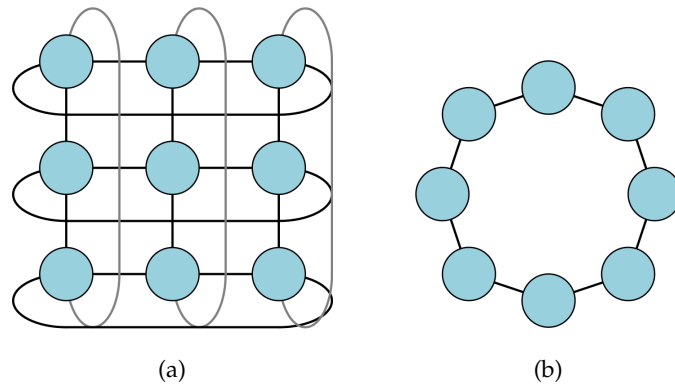


Figure 3.10: Examples for Direct Topologies (a) Ring, (b) 2D-Torus

3.4 Terminology

The following definitions and descriptions of terms are beneficial to fully understand the analysis of crossbars, as well as the design space exploration and implementation of EXTOLL crossbar done in the upcoming sections.

3.4.1 Deadlocks

A deadlock is a state in the network where two or more packets form a circular dependency where each packet waits for the others to release the physical resources. An example for such a condition is given in figure 3.11 on the next page where *packet 1, 2, 3* and *4* all wait for the availability of the shared resource, which is blocked by other packets respectively [74]. Without external interference the network itself is not capable of resolving this state.

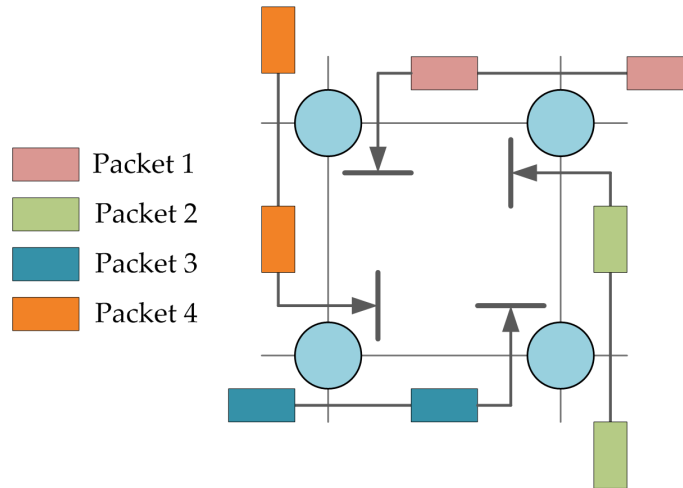


Figure 3.11: An Example for a Deadlock Condition

3.4.2 Virtual Channels

The virtual channel (VC) concept has been introduced by Dally et. al. [75] to be able to avoid deadlocks in interconnection networks. Each packet traversing the same physical channel is assigned to a specific VC. This assignment can for example be realized using a tag in the packet header. Each VC is assigned to a separate buffer, so that each channel can be read independently without having the need to read packets of other VCs first. With this tag, packets traversing the network on different VCs can share the same physical resources concurrently without blocking each other, thus using bandwidth that is otherwise unused. If one VC gets blocked, other VCs still can use the physical resources and therefore make progress eventually dissolving the blockage. As [76] and [77] depict, it is sufficient to have two VCs in order to be able to avoid deadlocks, even in tori topologies.

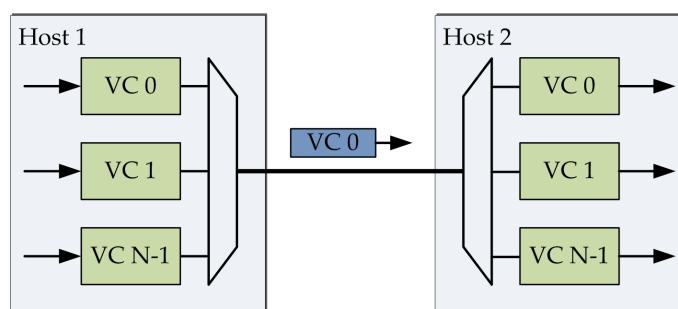


Figure 3.12: An Example for Virtual Channels

An example configuration is given in figure 3.12, where *host 1* sends packets using N VCs to *host 2* using only one physical connection.

3.4.3 Head-of-Line Blocking

head-of-line blocking (HOL) is a phenomenon that can only occur in input buffered crossbars, which use a FIFO based buffering scheme [78]. Packets are stored in the order of arrival and also handled in that same order. If the destination output of the first packet in the FIFO is blocked, then also this whole inport is blocked as well, as all following packets, which might be destined for other outputs, cannot be transmitted as well due to the blocked first packet. The VC concept reduces HOL but does not resolve it, as each added VC, and with it each added buffer, reduces the chance of blocking other packets in said buffers. Nevertheless, a chance that packets are blocked always remains.

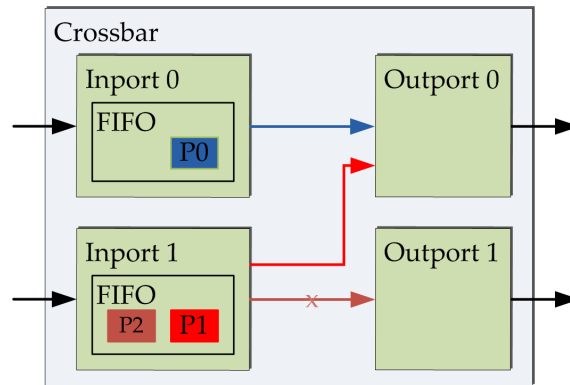


Figure 3.13: An Example for Head-of-Line Blocking

Figure 3.13 shows an example for HOL. *Inport 0* is currently transmitting packet *P0* to *output 0*. In *inport 1* are two packets stored at the same time, the first one is destined for *output 0* and the second one for *output 1*. *Output 0* is currently busy, consequently *inport 1* has to wait until it is ready again as the first packet in the FIFO cannot be transmitted. Thus, the second packet *P2* stored in *inport 1* could be processed as *output 1* is idle, but is blocked as it is buffered in the same FIFO.

3.4.4 Virtual Output Queuing

One way to completely resolve HOL is the use of virtual output queuing (VOQ) [79]. In VOQ the buffering still resides in each inport, but all packets are sorted into multiple buffers according to their destination port. Whenever an output is blocked, only packets destined for that output are blocked. All other packets can be processed without any restrictions. Figure 3.14 on the facing page illustrates the same example as in figure 3.13. However this time the two packets in *inport 1* are stored in two separate FIFOs. Thus,

packet $P2$ can be transmitted while $P1$ is waiting for the availability of the resource *outport 0*.

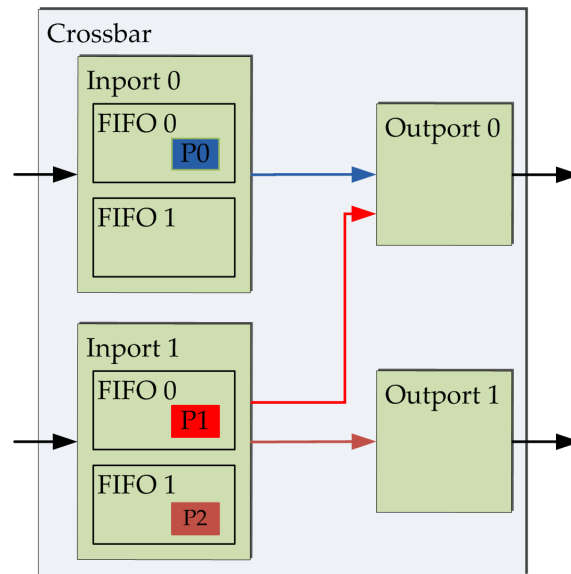


Figure 3.14: An Example for Virtual Output Queuing

The drawback of VOQ, according to [78], is that it does not scale, as the required buffer space increases quadratically with the number of ports.

3.4.5 Credit Based Flow Control

Flow control is an important issue in lossless networks. It has to be ensured that no packet is lost due to a full buffer at any point during the transmission. When using a *credit based flow control* all buffers are partitioned into slots of fixed size, each slot equates one credit. In the same way packets are rationed into flow control units (flits), equal to the slot size, whereas one flit also equates to one credit.

At power-up, each unit informs its connected unit from which packets are received about the amount of available credits. Packets can only be sent if a credit is available. Consequently, whenever a flit is sent to one of the buffers it consumes one such credit and the internal credit counter is decremented. Once the flit is read from the buffer, the credit can be freed again and sent back to the originating unit, which then increments its credit counter again.

This way the sending unit is always up-to-date on how many flits it is allowed to send before it needs to wait. If the buffer space is chosen wisely, the latency inflicted upon

sending a credit back to the sender can completely be hidden. For this calculation, the maximum length of the cable, the propagation delay of the signal in the cable and the time required to send and receive the credit on chip have to be taken into account.

3.4.6 Wormhole Switching

In *wormhole-switching* packets are split into multiple flits [63]. All flits of a packet are sent independently, however only the first flit of each packet contains the routing information. The path through all switching stages is determined with the first flit and then stored at each intersection for all following flits, creating a *wormhole* through the network. This path is closed with the last part of a packet. Wormhole-switching enables the design of low latency networks as the routing information can be interpreted as soon as the first flit arrives. The buffer requirements are relatively small, as each flit is small compared to the complete packet.

3.4.7 Virtual-Cut-Through Switching

In virtual cut-through switching (VCT) packets are not split into multiple flits, but sent as a whole. Consequently each packet is a flit. As a result there is no need to store the routing information along the way through the network. The routing information can be interpreted as soon as they arrived, thus not the complete packet needs to be received first in order to be able to start forwarding. VCT can reach comparable latency results as wormhole-switching, as packets can also be forwarded before they have been received completely. However, VCT requires more buffer space. The additional required buffer space is due to the increased flit size.

3.4.8 Adaptive Routing

Packets from one host to another always traverse a standard network using the exact same path. This keeps all packets in order. However, this can lead to congestions or hot-spots, paths that are heavily used, in the network.

Adaptive routing routing is a way to reduce these hot-spots. Instead of following a strict path, packets can be forwarded adaptively along a set of paths depending on the current load of the network. Using links with a lower utilization reduces the chance

for congestions or at least avoids that more packets are sent to the already congested area. Thus, this relaxes the heavily loaded areas of the network, improves the throughput significantly and can reduce the latency of packets. The broader the view of the network load is, the better is the adaptive routing decision that can be made.

3.5 Requirements for an Interconnection Network

After describing the most important terms it is important to understand what the requirements on a state of the art network and its switch are in today's HPC.

Low latency — As already discussed in section 1 on page 1, latency is the key to be able to scale a cluster. Therefore it is required that the network is able to send packets from one host to another with the minimum possible latency. The lower this latency is, the larger can the cluster be before the total performance decreases again according to *Amdahl's Law*[3]. In HW latency can be reduced by a combination of increasing the frequency and reducing the amount of pipeline stages. Ideally both methods are applied. However, the less pipeline stages are available, the more needs to be calculated within a single clock cycle, thus lowering the achievable frequency. A good trade-off needs to be found to get the best performance results.

High throughput — The larger a network of nodes is, the more small messages need to be passed from one host to another. As a consequence, the network needs to be able to issue these small messages efficiently. In best case, the design is capable of sending minimal sized packets back-to-back, so without any gap.

High availability — A high availability is required in order to keep the complete system operational. It is not desirable to shut down the complete system, if only a single node inside a large cluster fails. High availability can be achieved by implementing the ability to dynamically route around defective nodes.

Lossless — A packet that has been injected into the network needs to be received under all conditions at the target. Thus it is required that it is ensured that no packet is lost during its traversal of the network. This can be achieved by implementing some sort of flow control mechanism, ensuring that packet can always be buffered.

Reliability — In addition to the ensured reception of a packet, the packet also needs to be received correctly. A strong packet error detection and retransmission is therefore mandatory. The best performance can be achieved if the retransmission is done in HW

on a link level, instead of a end-to-end retransmission. The link level retransmission ensures that defects in a packet are recognized as early as possible, immediately sending the correct version.

In order delivery — The ability to receive packets in the same order as they have been received enables applications to be less complex and therefore a lot faster. Thus the network needs to be able to keep the order of packets belonging to a logical stream from each source-destination tuple. Packets belonging to other tuples may pass each other, only the sequence within a tuple is of importance.

Congestion management — Keeping the order of all packets requires the use of a deterministic route through the network. However, this route is then used by all packets, increasing the chance of a congestion. On the other hand, for some applications it is not necessary to keep the order of packets. For those applications an even distribution of the traffic, depending on the current load of the network, allows to reduce the chance for congestions. Thus it is beneficial if the network supports some sort of adaptivity in the routing process for applications that do not rely on the correct order.

Support for multicasting — The communication patterns in HPC often require the transmission of packets to multiple recipients. This communication scheme is called multicasting. Consequently it is beneficial if the network has an efficient support for multicasting, as a HW support can significantly reduce the time required to complete a multicast.

3.6 State of the Art

The following sections describe state of the art interconnection networks and their crossbar based switches. All these networks are leveraged in the top ten of the current *Top 500* [4] list. The most important features are mentioned and at the end, a comparison of all networks is given.

3.6.1 Cray Gemini

Cray's most up-to-date interconnect is implemented in their ASIC *Gemini* [5] [80]. It features two independent HT links, each connected to one NIC. Thereby two host CPUs can be connected to a single Gemini chip. These two NICs are then connected to a crossbar

using eight of the 48 in total available crossbar ports. Four of these ports are always grouped together forming a physical link to connect outside hosts, however each of the four ports is arbitrated separately while packets always only use one of these four ports. The resulting block diagram is illustrated in figure 3.15.

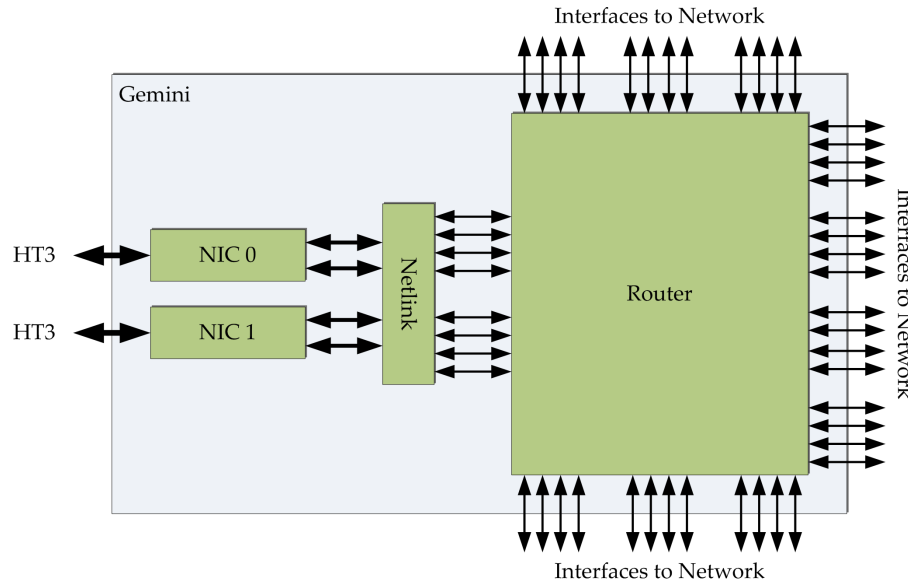


Figure 3.15: Gemini Block Diagram [5]

This results in a total of ten links, each providing an unidirectional bandwidth of 75 GBit/s. These ten links are used to form a 3D-torus with two links in each direction for the x- and y-dimension and one link per direction for the z-dimension. The crossbar itself, called the *Gemini Router*, is composed of 48 identical tiles, each containing all required logic and buffers for one input- and output port combination including a small eight by eight switching fabric. These tiles are placed in a fashion that they form a six by eight mesh on chip. This tile-based approach simplifies the chip development as it forms a regular structure. The tiles use input- and output buffers as well as row-buffers. Functionality of each tile is best described using a packet flow example. Arriving packets are first stored in the input buffer of the incoming tile. The input port determines the row of the tiles the packet needs to be forwarded when they reach the head of this buffer. Subsequently the packet is stored in the according row buffer at the input of the fabric. Once the packet reaches the head of this buffer, the routing decision to which column the packet needs to be forwarded has to be made, thus enabling the switch fabric to forward the packet to the according tile. Upon arrival, packets are stored in an output buffer associated to the originating tile and an arbiter multiplexes these buffers onto the output port. Consequently there is no centralized arbitration, but distributed arbitration is used.

Packets have a header of 24 byte and can have up 72 byte of payload, resulting in a

3 High Performance Switching

maximum bandwidth efficiency of only 75 percent. An eight byte sized packet results in an efficiency of 25 percent.

The chip is parted into two clock domains, the crossbar itself runs at 800 MHz, the NIC operates at 650 MHz. The minimum half round-trip latency that can be achieved is 0.7 μ s for a remote put. Each additional hop typically adds 105 ns to the above mentioned latency.

Switching is done using VCT, consequently all packets are composed of only one flit. Furthermore packets can be forwarded as soon as the head of the packet is available on link level. Internally wormhole-switching is used due to small internal buffers. A 16 bit unique identifier is used to specify the target chip, two additional bits are used to identify the node connected to this chip. Deterministic routing for in-order delivery of packets as well as adaptive routing for load balancing of the network is supported. For diagnostics a source-path routing algorithm can be used, otherwise a hash function reduces the 18 bit combined identifier and table-based routing is used.

Only two VCs are supported, one is used for requests, the other one for responses.

3.6.2 Tofu

Japan's *K-Computer* [2] with its 705 024 cores requires an efficient interconnect to reduce the jitter inflicted by communication and leverage the massively parallel computing resources. Therefore it uses a proprietary interconnect called *Tofu* [81] [82], which stands for *TORus FUSion*, specially developed to fit the needs of such a large system. The interconnect chip itself is called *InterConnect Controller (ICC)*. In contrast to Gemini it only offers one host interface, but four independent NICs, called *Tofu Network Interface (TNI)* as indicated in figure 3.16 on the facing page. Switching is done using the *Tofu Network Router (TNR)*. It provides ten links to connect other nodes, each offering 40 GBit/s raw unidirectional bandwidth. All three units of the ICC operate at 312.5 MHz.

The applied topology is quite unique. Two channels are used to connect four ICCs in a mesh, these four ICCs are all suited on a single mainboard. Two additional links are used to connect three of those boards, forming the so called *Tofu Unit* and the six remaining links are used to form a 3D-torus topology to connect multiple *Tofu Units*.

The network offers four VCs, two are used to separate requests and responses, the other two are used for deadlock avoidance during routing in the 3D-torus topology. To increase

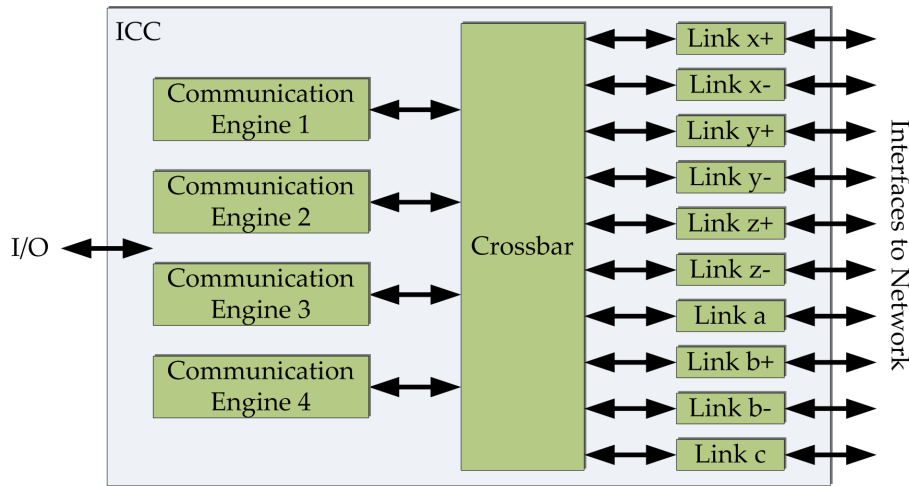


Figure 3.16: ICC Block Diagram [81]

the effective switching throughput, a VC is not only changed upon the alteration of an axis inside the 3D-torus, but already one hop prior to the change of direction. This lookahead allows packets of different targets to overtake blocked packets already in the stage before the turn, thus reducing HOL.

A three stage delta routing is used in the network, the first stage relies inside the source *Tofu Unit*, the second stage refers to the 3D-torus and the final stage again relies inside the destination *Tofu Unit*. Two alternative paths are encoded in each packet to balance the traffic inside the *Tofu Units*. This fact in combination with the fact that the four independent TNIs can be used in parallel by one thread leads to the conclusion that the network does not support in-order delivery of packets.

The crossbar or TNR itself is composed of 15 smaller five by three crossbars aligned in a five by three matrix. So again, just as in *Gemini*, a tile based approach has been chosen to ease the chip design. VCT is used inside each 5x3 crossbar to reduce the hop latency.

A single packet has a 40 byte large header and can have up to 1920 byte of payload, resulting in maximum bandwidth efficiency of 98 percent. The efficiency for a minimum sized packet of eight byte however is only 17 percent. The lowest achievable latency for a direct descriptor message, i.e. a Programmed Input/Output (PIO) transfer is 0.91 μ s.

3.6.3 TianHE-1A Interconnect

TianHE-1A is China's first supercomputer that was listed at position one of the *Top500* list in November 2010 [83]. Its interconnect [84] [85] uses a different approach as the first two mentioned interconnects. In contrast to the others, it is composed of two chips, a NIC chip and a *Network Routing Chip (NRC)*. Consequently instead of using direct connections between hosts a central switch is utilized. The topology is a hybrid fat-tree consisting of two layers. The first layer connects each node within a rack, the second layer connects all racks using a series of 11 384 port switches. The switches again are composed of a number of NRCs. The NRC itself is a 16 by 16 tile-based crossbar with 16 tiles, each implementing a four by four crossbar. Each of NRCs 16 ports offers a unidirectional bandwidth of 80 GBit/s and the internal operation frequency is 312.25 MHz. Inside the NRC wormhole switching is used to reduce the communication latency.

To avoid deadlocks, again four VCs are offered. The routing algorithm used is source-path routing. Thus, the required routing string is added to each packet by the NIC chip and only deterministic routing is available.

The maximum packet size is 128 byte of payload plus 32 byte for the packet header, therefore resulting in a minimum bandwidth efficiency of 20 percent and a maximum bandwidth efficiency of 80 percent. Using immediate RDMA writes, the *TianHE-1A* Interconnect achieves a latency of 1.57 μ s.

3.6.4 Blue Gene/Q

Blue Gene/Q [86] is the third generation of IBM's Blue Gene supercomputer family. It combines not only the NIC and the crossbar on a single chip, but also integrates the CPU and its cache memory. Having everything on a single chip provides the possibility to achieve lowest latencies for communication between two nodes. The network part supports 11 links, ten are used to build a 5D-torus topology, whereas the 11th link is only used as an I/O interconnect. Each link offers 16 GBit/s unidirectional bandwidth.

Packets have a 32 byte header already including the command of the NIC, with a size of 20 byte and an eight byte trailer including the packet CRC. The MTU is 512 byte, thus resulting in a minimum bandwidth efficiency of 50 percent and a maximum efficiency of 96 percent, taking the command into account into both calculations.

The crossbar suited in the network part of the chip is composed of receivers and senders

for the 11 links and additional FIFOs to send and receive packets to the local NIC. Each receiver has an exclusive FIFO to send data to the NIC. Multiple FIFOs also exist in the opposite directions, i.e. from the host. However these FIFOs are not exclusive for one sender, thus each FIFO can send data to every link. The block diagram is shown in figure 3.17, the internal connectivity however is not shown for clarity reasons.

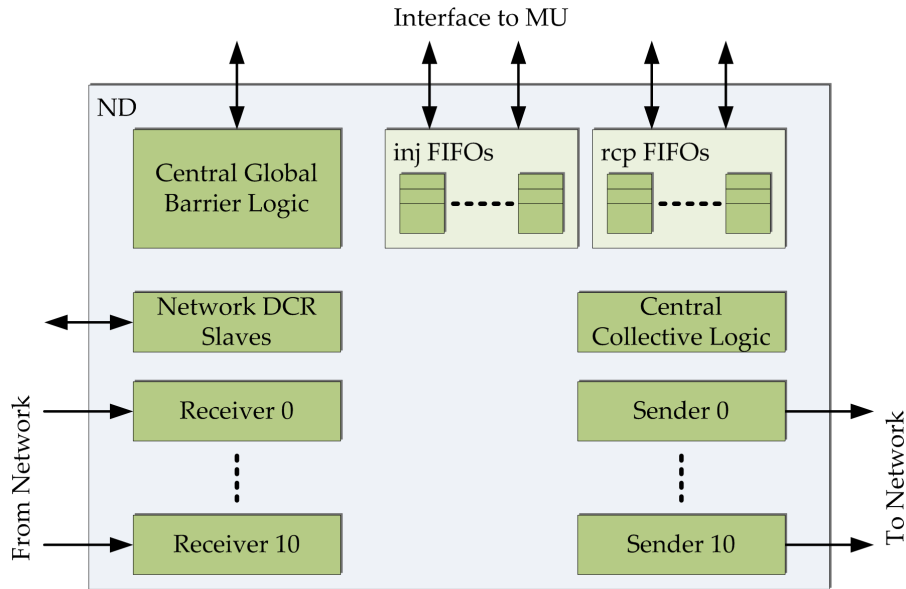


Figure 3.17: BlueGene/Q Network Device Block Diagram [86]

A total of five VCs exist to avoid deadlocks and ensure quality-of-service. HOL is avoided through VOQ, hence packets destined for different links do not block each other. VCT is used to reduce the packet latency, a packet is forwarded as soon as the routing information has been received. Routing supports both deterministic and adaptive paths and is table-based with coordinates as look-up address in the table.

Table 3.1 on page 83 summarizes the most important features of the presented interconnects. It can be seen that all crossbars share some features. Currently the most powerful systems have direct connect networks, but switch based networks are also present in the 11/2010 *Top500* list [83], starting at position four.

The last column of the table gives a preview of EXTOLL's features, a more detailed description of this interconnect is given in the following sections. Notable is that all interconnects are realized using ASIC technology, whereas EXTOLL currently uses FPGA

technology. Nevertheless EXTOLL reaches competitive performance results, as it is shown in section 4.6.5 on page 166. Most interesting are the efficiency numbers, as these numbers cannot be improved by implementing the design with newer technology. These numbers can only be improved by changing the design and the packet specification. EXTOLL offers the best efficiency when it comes to a typical payload size of a single cache line, which is 64 byte in size.

3.7 State of EXTOLL R1 Crossbar

The EXTOLL Revision 1 (R1) crossbar has been developed over various stages in [87], [88], [89] and [90].

Multiple research projects at the University of Valencia in Spain, for example *Memscale* [91], use this design successfully. The testbed in Heidelberg and Valencia is a Hyper-Transport expansion slot (HTX) based Virtex-4 [92] board, designed and manufactured at the CAG. The cluster in Valencia currently uses 64 nodes, each featuring one of said boards. An overview over the complete EXTOLL project and its realization on the above mentioned FPGA testbed is given in [93].

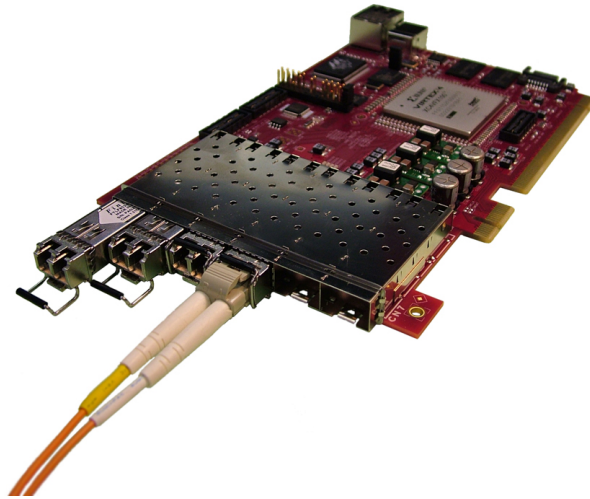


Figure 3.18: The HTX Testbed in the 4th Revision

The EXTOLL crossbar uses input buffering to avoid the necessity of an internal speedup, making it scalable and implementable on FPGA technology. In addition, it uses wormhole switching to reduce the route-through latency of packets and minimize the buffer requirements. As already mentioned, wormhole switching crossbars are prone to deadlocks. To

Interconnect	Gemini	Tofu	TianHE-1A	Blue Gene/Q	EXTOLL R2
Technology	ASIC	ASIC	ASIC	Full Custom	FPGA
# Lanes (per link)	12	8	8	4	4
raw Bandwidth [Gbit/s, per lane]	6.25	6.25	10	4	4
total Bandwidth [Gbit/s]	75	50	80	16	16
Ports (per node)	10	12	1	11	6
Latency [μ s]	0.7	0.91	1.57	0.72	0.99
Routing	Table- Based	Source- Path	Source- Path	Source- Path	Table- Based
Switching	VCT	VCT	Wormhole	VCT	VCT
Type	direct connect	direct connect	switch based	direct connect	direct connect
Topology	3D- Torus	6D- Torus	hybrid Fat-Tree	5D- Torus	arbitrary
Header Size [byte]	24	40	32	40	16
Max. Payload [byte]	72	1920	128	512	256
Max. Efficiency [%]	75	98	80	96	94
Min. Efficiency [%]	25	17	20	50	33
64 B Efficiency [%]	72	62	76	76	80

Table 3.1: Overview Over Different Interconnect Families

3 High Performance Switching

resolve the possibility of deadlocks, the EXTOLL crossbar implements the VC concept using the minimum of two channels to enable deadlock free routings [76] [77].

A problem which can occur in input buffered crossbars is HOL. As [94] states from the experiences of the EXTOLL predecessor ATOMIC Low Latency (ATOLL):

The network layer of ATOLL offers high performance, but as the experiments with *SWORDFISH* [95] have shown the lack of any provisioning against head-of-line blocking in the network layer is a problem when scaling to larger networks.

EXTOLL therefore uses VOQ on switch level, which completely solves the HOL problem by sorting all packets with different destinations into different queues [79], making it possible to not only read packets of independent VCs concurrently, but also packets for the same VC but for different outputs as destination in that node.

As an additional feature, the EXTOLL crossbar offers four different traffic class (TC) to provide a quality-of-service mechanism. The sending SW threads decide on which TC a packet shall traverse the network. This can be used to separate request packets from response packets or send storage or management packets on their own TC. Changing the TC during the transmission in the network is not possible. Further discussing this feature is out of the scope of this work, as the allocation of TCs is solely a SW matter.

The four TC together with the VC form the flow control channel (FCC), which are used to separate independent packet streams inside the network.

For a low latency communication it is desired to avoid the necessity of reordering packets at the receiver side. Reordering requires at least a second memory copy from the reorder area of the main memory to the user space memory region. Therefore the crossbar needs to maintain a logical order of all packet from one transmitter to a specific receiver. Packets from other transmitters or to different receivers do not need to be in any order in respect to other data streams. The EXTOLL crossbar enforces the order of packets by design due to the used VOQ mechanism, where packets are stored in a first-in, first-out manner.

3.7.1 Packet Format

Each packet is generated at the network layer, i.e. the FUs of EXTOLL and is not restricted regarding its size, as the units themselves are not aware of the MTU of the network. Since EXTOLL uses wormhole based switching, packets are split into multiple flits with a size restriction. This splitting is desired to reduce the buffer requirements, be able to ensure that buffer space is available and to ensure a strong and reliable error detection. The MTU is 32 physical units (phits) or 128 byte and each flit additionally has a start- and an end-control character as well as a CRC. A phit is a single word transmitted over the link and consists of 32 bit of data and four control bit, one for each byte within the 32 bit data word. These four control bit are used to determine whether the byte is a control or payload word. The total size of a flit therefore is 128 byte payload plus 12 byte for control characters. The maximum size of 32 phits has been chosen to be able to implement a simple and robust buffering scheme and still be able to fit a complete cache line of today's most commonly used CPUs into a single flit.

A greedy approach is used for packet fragmentation if a packet exceeds the MTU of one flit, which leads to the fact that each flit of a packet has the maximum length, except the last one. Figure 3.19 shows an example EXTOLL packet including its fragmentation. In this example only two flits are used to transmit the whole packet. However, the number of flits can vary.

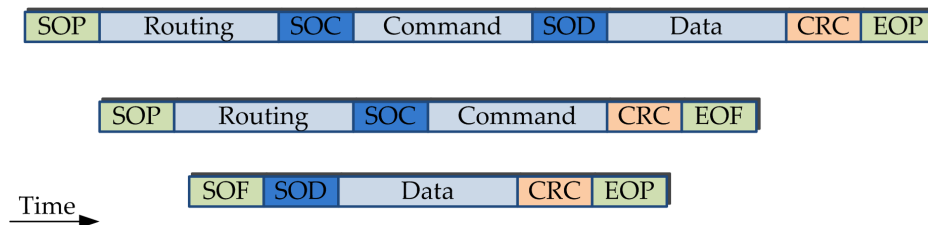


Figure 3.19: EXTOLL Crossbar R1 Packet Format

The start- and end-characters are special k-characters from the 8B/10B coding space [96] of the link layer, which have been selected to ensure the largest possible Hamming distance in order to have the highest possible link reliability [97].

As it can be seen in figure 3.19, each packet consists of three frames — *Routing*-, *Command*- and *Data* frame, whereas the data frame is optional and does not necessarily need to be appended. The routing frame is used to determine the way through the network. It is appended at the source, consequently the whole route is predetermined. The command frame holds the information on how to process the appended data and is specific for every FU. The packet closes with the data frame which includes the actual payload of the packet.

Frames are separated using again control characters from the 8B/10B coding space. The command frame always starts with the start of command (SOC) character and is also not limited in size. The same applies to the data frame, which starts with a start of data (SOD) character. As the routing frame is always required and is the first part of a packet, it has no additional control character.

For the EXTOLL crossbar itself, only the routing frame is of importance, the rest of the packet is treated as payload inside the crossbar.

3.7.2 Routing

Every routing phit contains two routing words which are again divided into four fields, as it is depicted in figure 3.20. The first field defines the direction, i.e. the crossbar outputport, in which the packet shall be transmitted. The second field defines the TC the packet belongs to.

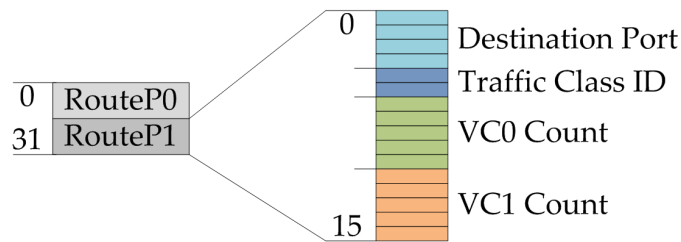


Figure 3.20: EXTOLL Crossbar Routing Phit

The last two fields define how many hops the packet shall take in that same direction, i.e. use the same outputport at following stages, one field for *VC0* and one field for *VC1*. *VC0* is always used first in case both counters are unequal to zero. Consequently a new routing word is required if a packet has to start at *VC1* and switches down to *VC0*, even if the destination remains the same. Moreover, each change of direction requires another word as the destination field then has a different value.

Every time a packet passes a crossbar stage in the network, one of the two counters is decremented and if both counters are zero the whole word is set to zero including the TC and destination field. The phit will be removed from the flit once both routing words inside that phit are zero. As a result the routing frame has been completely consumed when the packet arrives at its destination. The routing frame in EXTOLL is limited to the size of one flit minus one phit, thus 31. This is due to the fact that the crossbar design does not support the extraction of whole flits from a packet if there is no more usable data

inside. Thus at least a single phit of other data needs to remain in the first flit. However as a single flit is up to 32 phits long this is sufficient to send packets through a network with a maximum hop count of 1952 with a single VC change.

3.7.3 Architecture

The architecture of the crossbar can be divided into two main modules — *inport* and *outport*. These two modules can be used to form an N by M crossbar, whereas N is the number of inports and M the number of outports as it can be seen in figure 3.21.

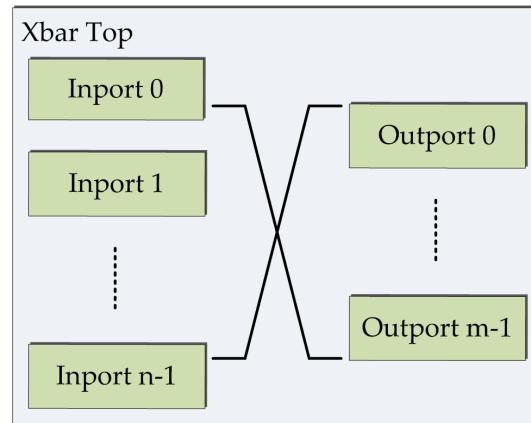


Figure 3.21: EXTOLL Crossbar Top Block Diagram

As already mentioned, VOQ is used as buffering scheme. Therefore packets are not only divided into the target outputs but also for the target FCC on this output. This results in number of outputs times number of FCCs queues per inport. To address the issue of scalability, which arises with having multiple buffers in each inport, all packets of an inport are stored in a single shared buffer and only the meta-data information are stored in separate buffers. These meta-data buffers are significantly smaller than the actual data buffer, therefore improving the scalability of the design. Meta-data is organized using a FIFO structure for each destination-FCC tuple. Since only one packet can arrive at a time, it is not required to be able to store a packet for every possible destination at the same cycle. Thus, a single RAM is sufficient to store all meta-data, further improving scalability. The *Multi-Queue-FIFO*, described in section 2.2.1 on page 14, is leveraged for that purpose.

A credit based flow control is used to avoid buffer overflows and therefore ensure a reliable transmission of flits. A request-grant-acknowledge scheme is used to arbitrate the physical crossbar fabric, where each inport requests the medium for all FCCs and outputs it has flits stored for. The arbitration is done in a distributed manner, so each output has

an own arbiter to grant its resources. A round-robin arbitration is used to avoid starvation of packets and to equally divide the resource among all sending inports. Grants are only given if a credit for the requested FCC is available and the current requesting inport has started this packet or there is no other packet pending for this FCC from another inport. This is necessary to ensure packet integrity, as packets can only be distinguished by the FCC they traverse through the network. Otherwise there is no way of separating two packets again once packets of the same FCC, but from different inports intersect each other.

This arbitration scheme has similarities with *iSlip* [72], but is not exactly the same as it is even simpler. The here used arbitration updates the priority in the grant generation logic every time a grant is given, whereas *iSlip* only updates the priority when the given grant is accepted.

Crossbar Inport

The most complex logic of the whole crossbar lies within the inport. It includes multiple modules to handle all functionality. The first function is routing interpretation, second is buffering all incoming data and finally the third is to request destination outports and forward packets accordingly.

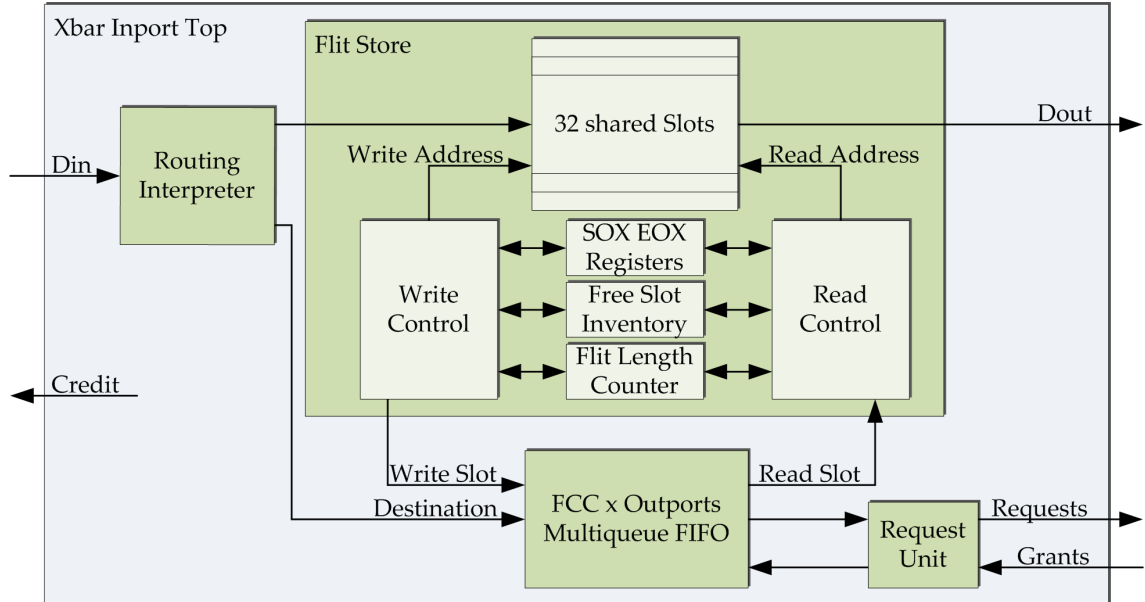


Figure 3.22: EXTOLL Crossbar R1 Inport Block Diagram

Figure 3.22 depicts the block diagram of an inport. The *Routing Interpreter* handles all

routing related tasks. A destination port is calculated whenever the first flit of a packet arrives at the crossbar and the destination is stored in a small buffer for all following flits of said packet.

All payload related storage including command- and data frame is handled in the *Flit Store*. As already mentioned, the crossbar uses a shared buffer for all incoming flits, as it is also used in the *ComCoBB* chip introduced in [79]. The buffer is divided into slots of fixed size, in a way that a maximum sized flit without start- and end characters fits into a single slot. This ensures that flits are always stored en bloc, which results in a simple buffer management, as it does not require handling fragmented flits. The CRC value is never stored, since it is invalid after the modification of the routing frame anyway and will be recalculated in the *Link Port* [98] before leaving the chip. Start- and end characters are stored in a separate buffer using an efficient coding, so that not the complete 64 bit of both characters combined need to be stored. Since both characters have static values it is sufficient to only store which kind of character occurred. The length of every flit is counted during the storage operation, in order to be able to overlap arbitration and transmission of flits, which is desired to be able to hide the arbitration latency.

In contrast to [79], the buffer slots are not managed using linked lists, but a *Multi-Queue-FIFO* stores the slot in which each flit has been written to, in order to be able to read them once a grant has been given by the targeted output. The order of packets is maintained through the nature of the FIFO. The inverted empty signals can directly be used to generate the according requests.

Last, the *Request Unit* is responsible for generating all requests and selecting one of the received grants. Requests will be made to all outputports and for all FCCs. One of the received grants is selected in case more than one has been received. Again a round-robin scheme is applied. Round-robin arbitration ensures fairness under all condition, equally distributing the available bandwidth over all outputports and FCCs. The chosen outputport is informed of its selection and the transmission is initiated. A credit is sent to the connected unit every time a flit has been sent completely.

Crossbar Output

The outputport itself is a fairly simple. Figure 3.23 on the following page depicts the block diagram of that unit. Only three modules are required to handle all outputport functionality.

Din Mux is connected to all inports, multiplexes the selected inport data stream and sets

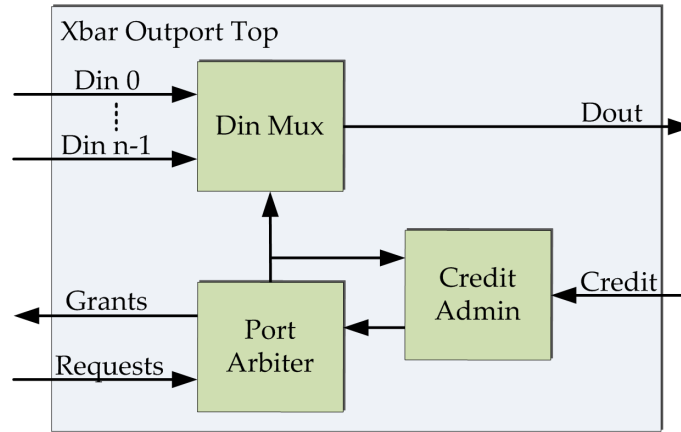


Figure 3.23: EXTOLL Crossbar Outputport Block Diagram

the data output accordingly. Essentially, this unit with its multiplexing structures forms the actual fabric of the crossbar. The *Credit Administration* keeps track of all available credits and informs the *Port Arbiter* which of the FCCs are eligible to send flits. The *Port Arbiter* as the last unit uses a round-robin arbitration scheme to grant one of the incoming requests according to the information provided by the *Credit Administration*. The arbitration cycle is repeated, if no grant has been received for a certain period of time from the currently granted inport.

3.7.4 Performance

The Performance numbers of the crossbar have been extracted from simulation and are based on the Virtex-4 design, used in the testbed shown in figure 3.18 on page 82. The complete design runs at 156.25 MHz. The interface width is 32 bit and therefore has a peak bandwidth of 540 MB/s per direction and port. The complete design is a full duplex nine by nine crossbar and thus has an aggregated peak bandwidth of 11.52 GB/s.

The crossbar is capable of sending more than 15 million messages per second for very small messages as shown in figure 3.24 on the facing page.

The bandwidth efficiency almost grows linear with the length of flits and has its peak at around 84 % as illustrated in figure 3.26 on page 92 or 540 MB/s per port as depicted in figure 3.25 on the facing page. Please note that these values are without any routing information or SOC, SOD characters and this is therefore the theoretical peak performance of the crossbar itself. A minimum sized network packet has 16 bytes of payload including routing, CRC and control characters which results in a message rate of 15.6 million

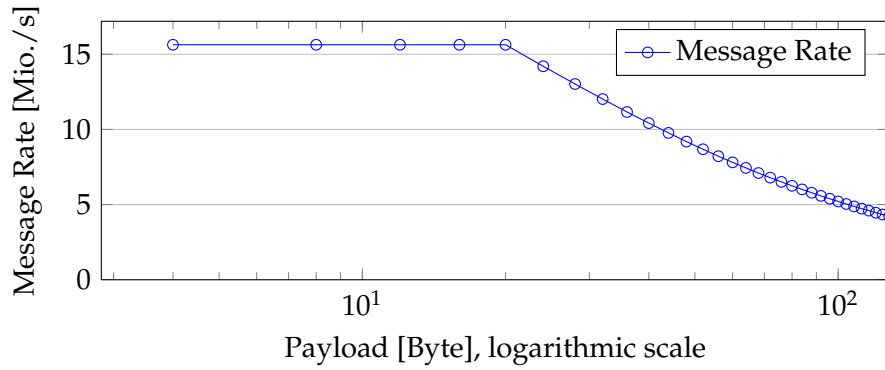


Figure 3.24: EXTOLL R1 Crossbar Message Rate

messages per second and a peak bandwidth of around 250 MB/s. Measured values may vary as the performance is also affected by surrounding units, like the host interface or the FU and is not necessarily limited by the crossbar.

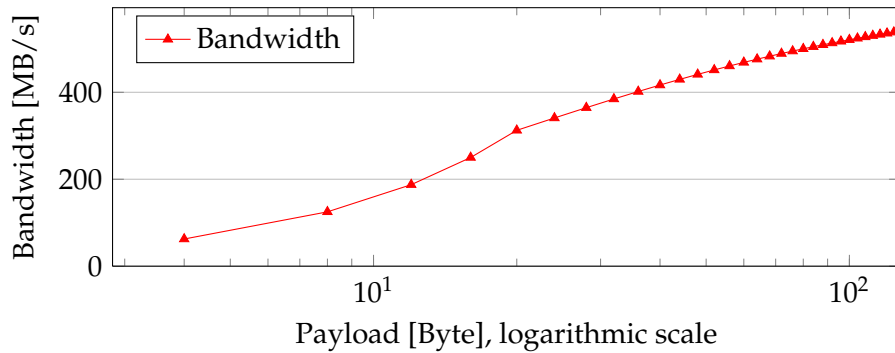


Figure 3.25: EXTOLL R1 Crossbar Bandwidth

Figure 3.27 on the next page shows the route-through latency in a quite network. Shown is a timing diagram, retrieved from simulation that indicates the pipeline stages that have to be passed from reception of a packet until the packet reached the output of the targeted output. The difference is 13 clock cycles which results in 83.2 ns at 156.25 MHz. Route-through latency can of course be higher if other flits are stored in the inport buffer.

Cost

The cost of the design can best be measured in the used resources on the FPGA of the testbed platform shown in figure 3.18 on page 82. In this case it is a Xilinx Virtex-4 FX100 [99] device. Table 3.2 on the next page shows the resources utilization. The consumed resources correspond to around 35 percent of all available logic on the device.

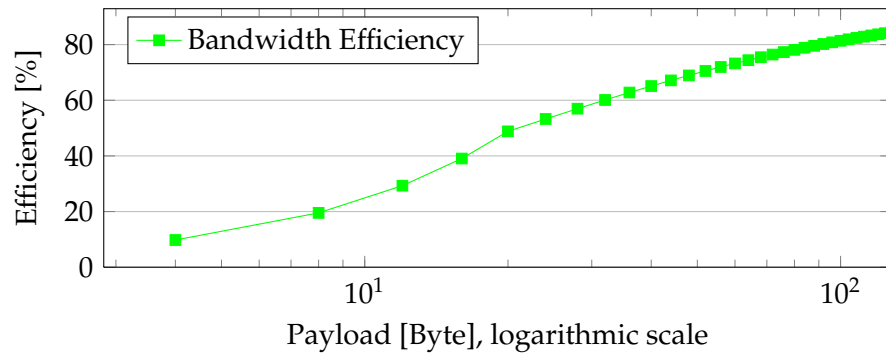


Figure 3.26: EXTOLL R1 Crossbar Bandwidth Efficiency

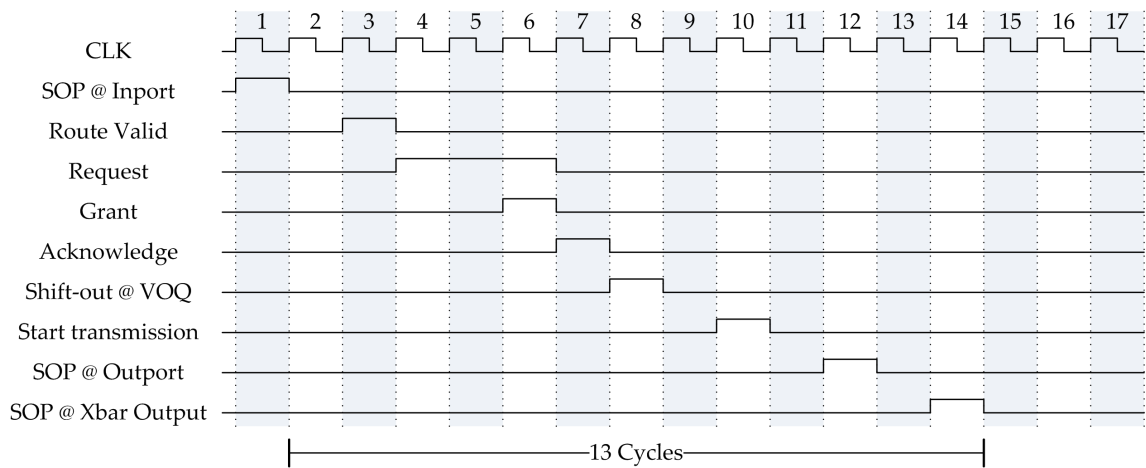


Figure 3.27: EXTOLL Crossbar R1 Route-Through Latency

For a comparison with the results of the upcoming redesign, a run on a Xilinx Virtex-6 LX240T [100] device has been made, which is the target testbed for the Revision 2 (R2) implementation. The results are also shown in table 3.2. The LUT usage has dropped around 40 percent, which can be explained by the new technology of the FPGA. Virtex-4 devices use LUTs with four input values, whereas Virtex-6 devices use LUTs with six input values. Thus more complex logical equations can be mapped into a single LUT. In addition, the RAM usage has dropped by almost exactly 50 percent. This is also due to the new FPGA technology. The RAM block size has been increased by 50 percent in the new FPGA family. The decrease in the register count can be explained by less register

	LUTs	Registers	RAMs
Virtex-4	25287	9312	63
Virtex-6	15639	9087	32

Table 3.2: EXTOLL Crossbar R1 Synthesis Result

duplication, which is a technique to improve timing on critical paths. It reduces the load on a single register by duplicating it.

3.8 Implementation of R2 Crossbar

Based on the experiences gained in earlier developments in [87], [88], [89] and [90], a completely new crossbar architecture has been developed. This development has then successfully been tested and in an FPGA implementation and an ASIC implementation is currently under development at the EXTOLL GmbH [101].

A thorough evaluation of all features and bottlenecks of the previous design has been made to improve its performance. The main criteria for this evaluation are the desire to achieve a higher bandwidth efficiency, a hundred percent throughput and maintain its good route-through latency.

3.8.1 Routing

As already mentioned, the R1 crossbar uses a source-path routing algorithm. The routing capabilities are limited due to the required on-chip look-up table for the complete path which supported only up to four phits per entry and only 64 entries in total. The amount of entries as well as each entry size is not enough to support a large scale cluster. With each routing phit a packet is capable of traversing 64 nodes in one direction, taking into account that a single VC switch is possible. Thus, with four phits a maximum of 192 hops with a total of up to three changes in direction are possible, since the last phit is required to send the packet to its destination FU.

Table 3.3 on the next page illustrates the amount of RAM required in each FU, in order to increase the number of supported target nodes as well as the routing capabilities by increasing the length of each routing string.

Having a routing table of two MB in each FU is not possible, neither in a FPGA nor in an ASIC implementation. A solution is to use a caching algorithm to only store a small part of the table on chip. However, this adds a source for non-uniform latency of packets, requires additional main memory read logic in each FU and thus adds a lot of complexity to the design. In addition, prepending every packet with 256 bit of routing even if not completely necessary, due to a very short path, adds additional latency and reduces the efficiency

Number of nodes	128-bit entries [kB]	256-bit entries [kB]
1024	16	32
2048	32	64
4096	64	128
8192	128	256
64 k	1024	2048

Table 3.3: EXTOLL R1 Source-Path Routing Table Size

as the overhead increases. This seems counter-productive since one of the design goals is to reduce latency as much as possible. The more hops are necessary, i.e. the larger the network, less bandwidth is available for the actual data transmission.

Another problem with the current source-path algorithm is that it only supports grids and tori efficiently and that it does not support arbitrary routings in case of malfunctioning links between two nodes, as the path is already determined at the source. Currently there is no support for exchanging the routing string in a packet. Thus, remote management is difficult, as it cannot be guaranteed that a response will be received. A solution to the problem with defective links in source-path routed networks is given in [90]. The resource requirements however cannot be underestimated. Due to all the above mentioned drawbacks, it has been decided to rethink the choice of routing algorithm with the goal to achieve a simple and efficient routing algorithm which supports a high node count, a low and deterministic latency and support for arbitrary topologies. Figure 3.28 shows all options that are available for routing.

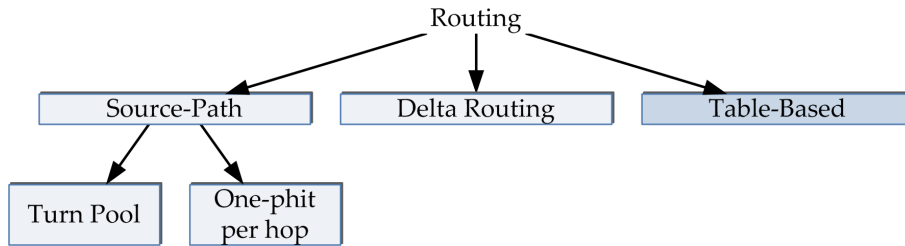


Figure 3.28: EXTOLL R2 Routing Design Space Diagram

Delta Routing

A node identifier (ID) can be seen as a tuple stating the position of the node in a three dimensional grid; X, Y, and Z. The router knows its own position in the grid and the outgoing target port can be computed from this information.

The advantages of this algorithm are that it requires no memory resources on chip and the target can be computed in a single clock cycle. Thus, it is as a low latency and deterministic algorithm. The main drawback however is that it only supports grids and tori, non k-ary n-cube topologies cannot be supported using delta routing. In addition, it is not possible to route around defective parts of a network.

Source-Path Routing

In source-path routing the path to the destination is pre-calculated at the source. It has already been introduced in section 3.7.3 on page 87. The problems that occur when using this routing scheme have been laid out. As a consequence this option renders no option.

Table-Based Routing

Table-based routing has been introduced by Kermani et. al. [102]. In table-based routing every node is assigned an unique identifier and the target of packets is determined by these IDs. Upon reception of a packet this identifier is used as lookup address inside a table, which holds all target information. One of the advantages is that the overhead inside each packet is reduced to a minimum, as only the target ID is required. The same applies when using delta routing. In contrast to delta routing however, table-based routing supports arbitrary routing algorithms. It has not to be decided on implementation which algorithm to use, but it can be chosen on initialization of the network, making table-based routing also interesting to do hands-on research on routing algorithms in a network. There are also no restrictions whatsoever regarding the topology of the network or regarding its diameter. The only restriction that applies to table-based routing is given by the size of the used ID. The width of this field determines how many nodes can be addressed in the whole network.

The possibility to reroute packets in case of defective links or nodes is easily possible, as the route is not predetermined and the method allows arbitrary routes. However, SW has to take care that no deadlocks can occur due to modified routes. Adding support for adaptive routing, which allows packets to decide from hop to hop which path to take is also possible in a simple matter. Only the information which of the output ports are valid destinations for the adaptive paths have to be added to the routing tables. Adaptive routing allows packets to travel along less busy paths and therefore reduces the possibility of congestions.

3 High Performance Switching

The drawback of table-based routing is the need to have a large table in each inport of the crossbar, in order to be able to make a routing decision, which has a number of entries equal to the number of supported nodes.

The minimum requirements for each entry is, that at least a destination port as well as the target VC is present to enable the management SW to calculate deadlock free paths. Therefore the minimum size requirements for a single table entry of an EXTOLL design with six links and a number of network ports is therefore four bit to encode the destination port and two bit to encode the VC. Table 3.4 shows the size of the required table in each inport and the resulting accumulated size of all tables for a ten by ten crossbar design.

Number of nodes	4+2 Bit Entry [kB]	Accumulated Table Size [kB]
1024	0,75	7,5
2048	1,5	15
4096	3	30
8192	6	60
64k	192	1920

Table 3.4: Minimum Size Requirements for Table-Based Routing

As it can be seen, even with a direct-mapped table, where every destination node has a single entry, the accumulated RAM requirements are lower than the ones from the previously used source-path routing in a single FU. For example, a single Virtex-6 block-RAM can support the routing table for a network with 4096 nodes as the size of such a RAM is 4.5 kB. This approach eliminates routing caching and reduces the routing interpreter to a local table look-up with the destination ID as read address to the RAM.

To support a larger amount of nodes or reduce the RAM requirements for a certain number of nodes, a hierarchical approach for the routing is also possible, as it is also done for example in Cray's Gemini interconnect [5]. Therefore the network is partitioned into several segments and the IDs are separated into a local and a global part, as it is depicted in figure 3.29 on the next page. There is no drawback due to higher frequented links if using this partitioned approach, as each inport has its own routing table and can thus choose different paths to reach another partition.

The routing interpreter then uses two tables, one for the segments, i.e. the global routing, and one for routing inside a segment, i.e. local routing. A check whether the target node of a packet lies within the same partition as the own ID or not is performed while accessing both RAMs. If the global part matches, then the routing entry of the local routing table is used, otherwise the routing entry of the global table is used. The advantage of this

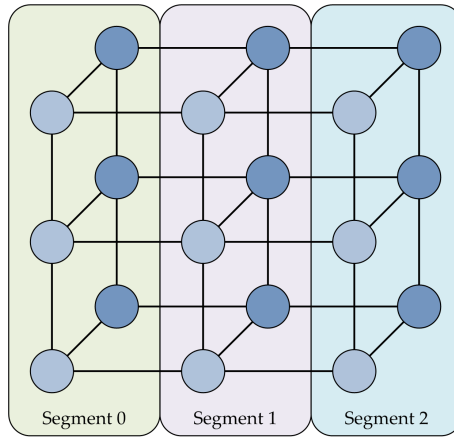


Figure 3.29: Network Partitioning

approach is, that the size of both hierarchical tables combined is smaller than a single direct-mapped table to support the same amount of nodes.

Supporting again 64k nodes and using for example a partition of four bit for the global and 12 bit for the local table, i.e. having 16 segments each hosting 4096 nodes reduces the required RAM space to $16 \times 6 + 4096 \times 6 = 3084$ byte for a single crossbar port and $3084 \text{ byte} \times 10 = 30 \text{ kB}$ for the whole crossbar with again ten outports. Comparing this value to the requirements for a direct-mapped table, the hierarchical approach reduces the RAM usage by an order of magnitude without adding latency or reducing the functionality significantly. Using other apportionments can further reduce these requirements.

To further reduce the required RAM resources it is possible to share tables among multiple inputs. However, this makes arbitration for the read accesses necessary and thus results in additional and indeterministic latencies for each access. The advantages of table-based routing outweigh its drawbacks by far. Hence, the new design leverages table-based routing with a hierarchy of two.

As figure 3.33 on page 101 depicts a 16 bit wide field has been chosen for the target node, hence supporting 64k nodes in the network. For the FPGA implementation a width of six bit for the *global node ID*, thus supporting up to 64 segments and a width of ten bit for the *local node ID*, supporting up to 1024 nodes inside a single segment, have been chosen. These values are however arbitrary and can easily be changed. Investigating a better local-global tuple is part of the routing algorithm SW and therefore out of the scope of this work.

Once a packet arrives at a crossbar inport, the routing interpreter compares the incoming

3 High Performance Switching

target node ID with the ID of the local node. If the global part does not match its own ID then the global table entry is used for the routing decision. If the global part is the same but the local part does not match, then the local table entry is used. The packet is directly forwarded to a FU if both parts are the same. Then the packet has reached its target destination. Partitioning of the network can be chosen freely, nodes do not need to be next to each other to be part of the same segment. Only the node IDs need to be arranged properly by the management SW.

As already mentioned, also an adaptive routing support is desired and thus added to the R2 design. Therefore, according to Duato et. al., a third VC is required to ensure deadlock freeness [103]. A bit-mask in each routing entry depicts to which outports packets can be forwarded to at each crossbar stage. The *almost full* signals of the VOQs are used to detect the ports with the least traffic. If a certain amount of packets are already stored in one of the queues, then this outport is not selected on this hop, thus using less populated connections. If multiple ports are eligible, then a round-robin arbitration is used to select one of them, equally distributing the traffic among all possible links. A fall-back to a deterministic route is used in case none of the selectable ports seems feasible. Packets using adaptive routing can arrive out of order at the recipient, since they have paths of unequal length and latencies. In this case receiving SW has to take care of reordering packets if necessary.

The crossbar ports of the complete design are split into two logical parts, where the lower port IDs are used to connect FUs on chip and the upper ones are used to connect the physical links to external hosts. However, the ports itself are all the same, only the addressing scheme is different. This partition avoids that packets sent from one FU can be received by a different unit type. The network ports are only reachable if the target node ID matches the ID of the current node. Using this addressing scheme removes the possibility to loop traffic back to the same host using the external links. On the other hand, this is only useful for benchmarking or debugging of the complete design, a productive system rather sends packets directly from one CPU to another without involving the NIC.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TC3					TC2					TC1										TC0											
Det. Ports		VC			Det. Ports		VC			Det. Ports		VC			Adaptive Ports					Det. Ports		VC			Adaptive Ports						

Figure 3.30: EXTOLL R2 Routing Table Entry

The crossbar supports four TC as already mentioned in section 3.7 on page 82. Each of the TC shall be able to use different paths in the network, to balance the network load. Therefore each table entry offers separate fields for each TC. To keep the RAM requirements within a limit, only two of the four TC however support adaptive and

deterministic routing, as the adaptive part of an entry is considerably larger than the deterministic part. These two TCs can i.e. be used for storage traffic or any other kind of traffic where ordering requirements are not strictly necessary.

This results in a 32 bit wide routing table entry which is used both for the local- and the global table as depicted in figure 3.30 on the facing page.

Only *TC0* and *TC1* support adaptive routing. Therefore the lowest part of the according entries offer the bit mask used to choose the allowed outputs, where only link ports are selectable, as it needs to be impossible to forward packets to FUs adaptively. The two bits of the *VC* field are used to determine the outgoing VC in case of deterministic routing and the three bit wide *Det. Port* field is used to determine the outgoing link port in case of deterministic routing. The number of network ports is added to the value written inside this field to calculate the actual crossbar port ID. The entries for *TC2* and *TC3* are identical to the deterministic part of *TC0* and *TC1*. FUs are targeted solely by the *target unit* field which is suited in the start cell of each packet.

A RAM with a size of 4.25 kB is required in each crossbar inport using the above described routing entry with a six-ten split of the *target unit ID* for the global- and local table. The resulting RAM requirements for a complete ten by ten crossbar is therefore 42.5 kB.

3.8.2 Ordering of Packets

A problem that arose during the testing of EXTOLL R1 was the possible mixture of packets. This occurred if two independent packet streams of the same TC arrived at an inport on separate VCs and one of them changed its VC due to the applied routing algorithm.

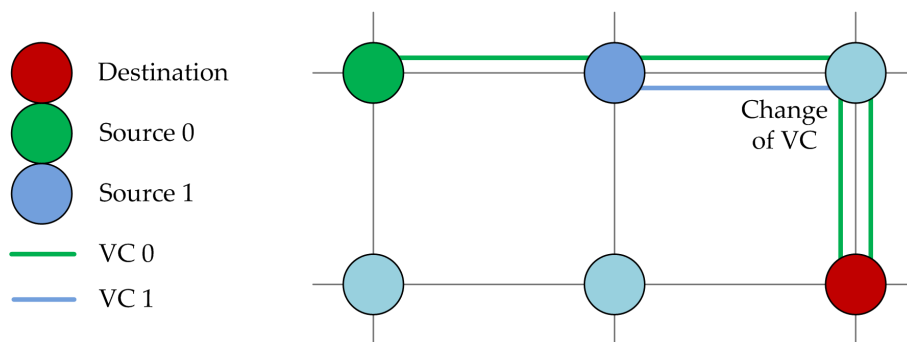


Figure 3.31: EXTOLL R1 Routing Problem (1)

Figure 3.31 on the previous page shows a section of a network with six nodes. In this example the applied routing algorithm is a simple dimension-order routing, meaning that first packets traverse along the x-axis, followed by the y-axis. *Source 0* (green) and *source 1* (blue) are both sending packets to the same destination (red). When changing directions on the top right node both packets are forced to VC 0 due to the routing algorithm.

Figure 3.32 illustrates the internals of the receiving inport of the top right node where 1 and 2 are flits from *source 0* and A and B are flits from *source 1*. Both streams will be merged in the same queue of the inport, if both streams concurrently send flits, as it will be the case due to the used round robin arbitration. This results in corrupted packets, as they cannot be kept apart anymore.

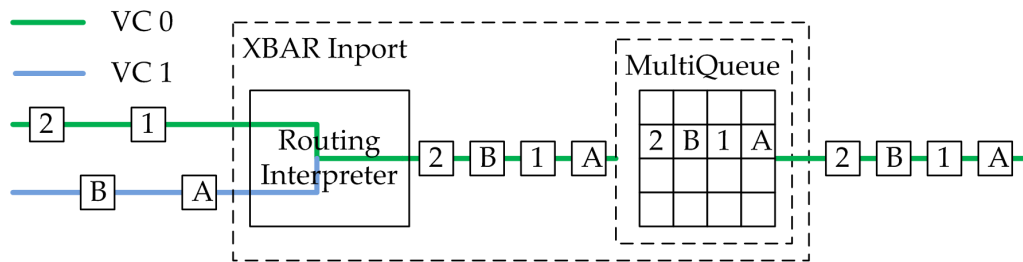


Figure 3.32: EXTOLL R1 Routing Problem (2)

There are three possible solutions to resolve this issue. First, all packets going from one inport to the same outport except the currently transmitting one can be blocked. This however increases latency for all other packets, even if they are not related to the excepted one. All packets have to wait until the current prioritized packet has been transmitted completely and this of course can result in deadlocks, as it is required to be able to send packets of different VCs independently. Blocking these packets therefore makes the concept of VCs obsolete. As a result, the network will no longer be deadlock free, which is of course not an option.

The second possibility is to add a second arbitration level, which arbitrates first an available VC and then with the second phase the actual physical medium. This however increases the route-through latency for packets which is not desired.

The third option is to move from wormhole-switching to VCT, which by design avoids the merging of packets, as each packet is only one flit long. VCT has a higher buffer requirement as packets now traverse the network as a whole. On the other hand, with the advancing technology, buffer space becomes more and more affordable for both FPGA, as well as ASIC technology. By removing the necessity of flit handling, the crossbar internals are actually less complex, increasing the potential for a better performance.

3.8.3 Packet Format

The packet format itself has not been part of this work, but has been developed in [104]. Nevertheless, a short introduction is given, as the new crossbar leverages from the new format.

Due to various changes, for example in the switching and routing method, the need for a new packet format arose. This new format is shown in figure 3.33. Packet are now defined in *cells* instead of phits, making the specification independent from the actual data bus width. These cells are the smallest unit in the specification, a single cell is a 64 bit word. The data bus width is therefore changeable in a multiple of cells, called lanes. Actual data bus width can be calculated when multiplying lanes times cell width. Each packet consequently begins with a *start cell* and ends with an *end cell*.

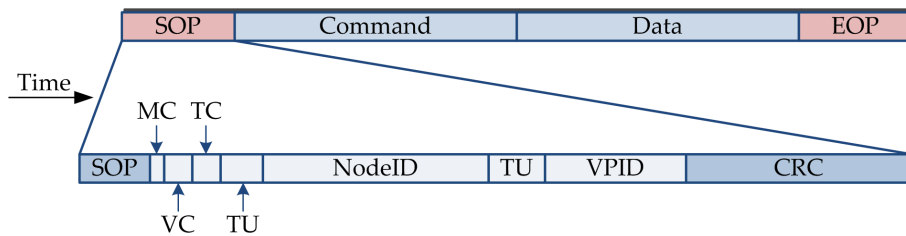


Figure 3.33: EXTOLL R2 Packet Format

Packets are now no longer split into multiple flits due to the applied VCT. Start- and end cells are therefore reduced to start of packet (SOP), end of packet (EOP) and end of erroneous packet (EOP_E) respectively. There no longer exists a routing frame inside each packet, since routing is now table based as explained in section 3.8.1 on page 93. A *target node ID*, included in the start cell, is the only required routing information, thus increasing the bandwidth efficiency.

SOC and SOD characters have also been removed, as the target FU is also encoded inside the start cell and each unit knows its command length. The concepts of TC and VC have been maintained as it proved to be beneficial. A description of the different fields inside the start cell is given in table 3.5 on the next page.

The packet CRC has been moved into the end cell. Thus a packet now consists of a single start cell, an arbitrary number of cells and an end cell. For the FPGA implementation a cell count of 32 has been chosen, as this has been the previous flit length. These modifications of the link level protocol increase the bandwidth efficiency for the FPGA implementation from 14.3 to 33.3 percent using minimum payload packets and in the best case (maximum payload) from 85.7 to 94.1 percent.

Field	Width	Description
CRC	16	A CRC-16 Checksum used to protect the SOP-Cell
VPID	10	VPID of target thread
NodeID	16	NodeID of target node
TU	3	ID of target functional unit on target node
TC	2	Traffic Class packet belongs to
VC	2	Current Virtual Channel ID
MC	1	Indicator that packet is part of a multicast
SOP	12	Indicator that this cell is a Start of Packet

Table 3.5: EXTOLL R2 SOP Field Definitions

3.8.4 Multicast

As Nüssle [94] states from the experiences of the EXTOLL predecessor ATOLL:

ATOLL suffers from a lack of support of multicast and barrier operation.

The design space and implementation for barriers and fast synchronization support is discussed in [98].

Multicasts are packets that are injected into the network only once, but can have multiple recipients. All recipients can be grouped together and multicasts are distinguished by applying a *multicast group ID*. A logical tree is spanned over the network connecting all multicast members. This is always done by SW due to its complexity and the fact that this is only necessary once during initialization and therefore not timing critical.

In general every network is capable to support multicasts using a modified driver that replicates packets and exchanges the target node ID accordingly. The same effect can be achieved by sending the packet multiple times directly from the user application. Obviously this is not very efficient, since all packets are replicated at the earliest possible stage. Thus, each replication has to pass through the host interface, the whole NIC and the complete network to its destination.

More efficiency can be achieved when replicating the packet as late as possible, at a point in the network where they do not share the same path to their target anymore, thus following a logically spanned tree. This however can only be achieved with HW support. An example of a SW based and a HW based multicast is given in figure 3.34 on the facing page.

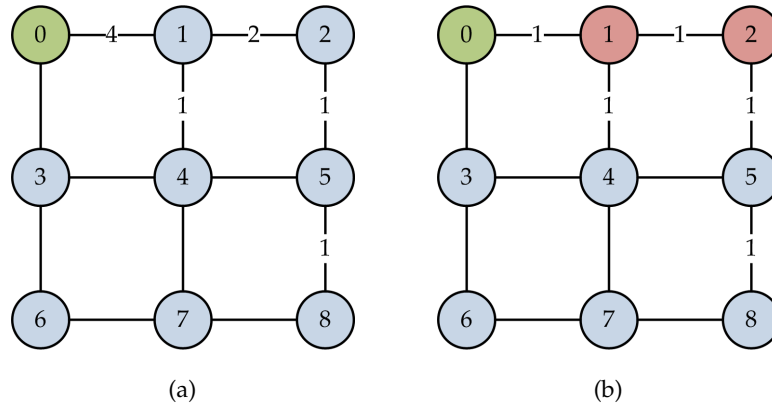


Figure 3.34: Multicast Example Using a SW- (a) and Using a HW Approach (b)

The figure depicts a network with nine nodes. In this example node 0 (green) wants to send a multicast packet to the nodes 1, 2, 4, and 8. Numbers on the links between the nodes depict the number of packets that need to traverse this link for this multicast. The accumulated number of link traversals for the SW approach is nine, the required amount for the HW approach however is only four, when spanning an ideal tree. Consequently the load in the network can be reduced and is now less than 50 % when using a HW supported multicast instead of the SW solution. The red nodes in figure 3.34 replicate and forwarded the packet multiple times. Reducing the network load also has a positive effect on the latency of packets and reduces the possibility of congestions. Hence it is desirable to also have this feature in the next revision of EXTOLL.

In general, multiple ways to support multicasts in HW are possible as illustrated in the design space diagram in figure 3.35.

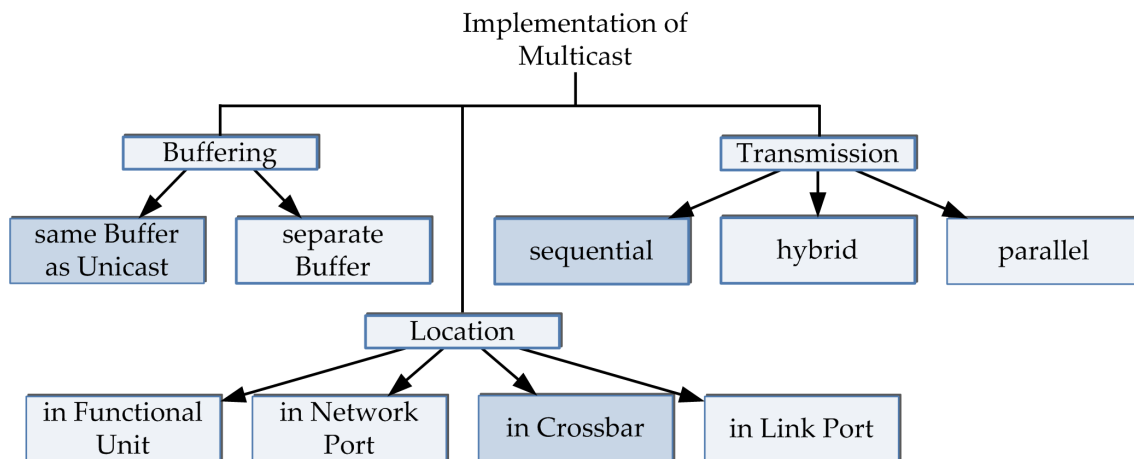


Figure 3.35: Multicast Design Space Analysis

3 High Performance Switching

Multiple aspects have to be considered when designing multicast capabilities. The first is where to add these capabilities. Adding them to the FU that injects packets or to the NP, which is the conjunction between NIC and network, is not a good solution. The replication takes place right at the beginning of the network path and therefore requires the maximum amount of buffer space and link traversals in the network. Nevertheless, this solution is still faster than a pure SW approach, as the message is replicated in the NIC instead of the host CPU.

Adding it to the Link Port (LP) already improves buffer space requirements, as the packets can be replicated at later stages. However, this is a violation of the layer paradigm, which states that the LP is only responsible for a reliable data transmission across one link. Additionally, the buffer requirements are not optimal, as packets are present multiple times in the directly connected crossbar inport buffer connected to the link. This leaves only one option left — inside the crossbar. A multicast packet is stored only once and replicated upon transmission to the next stage, therefore requiring no additional buffer space compared to an unicast packet. Hence, the multiplication of packets is done at the latest possible stage in the network, avoiding that replicas of packets traverse the same link.

The second aspect is to where store multicast packets. Two implementations are considerable, storing it in a separate buffer or reusing the existing buffer. McKeown et. al. propose the usage of separate buffers in [105]. However, this not only increases the amount of buffers per inport, but also the scheduling algorithm complexity increases as a second level of priority is required in the proposed *ESlip* algorithm to switch between unicast and multicast. Reusing the existing buffer, as it is also proposed in [106], is to be favored, due to the better buffer usage, resulting in a smaller memory footprint of the design. Multicasts are treated the same as unicasts in respect to requesting the outport, keeping the arbitration simple and the buffer space requirement low.

The last aspect is how to transmit the multicast packet from the inport to all receiving outports. The fastest way is to send the multicast in parallel to all outports. This however requires that the multicast must be first in line for transmission at all outport queues, which is quite unlikely in a network that is not idle all the time. Thus, all outports need to grant in the exact same clock cycle or already granted outports need to be blocked until all targeted outports of the multicast have been acquired. This introduces HOL which has to be avoided for an efficient bandwidth usage and a low packet latency.

Sending all replicas sequentially is the easiest way, since no changes in the arbitration algorithm is necessary and they can be sent whenever the corresponding output port is available. This solution has been favored due to its simplicity.

An hybrid approach is to send all replicas sequentially, however whenever outports of the same multicast packet have been granted in the same cycle, then send it to all of them in parallel. In the best case this results in the same performance as the parallel approach, but without the restrictions of the latter. Worst case is a complete sequential transmission, as it is with the second solution. The difficulty in this approach is that data structures have to be maintained that indicate whether a request has been issued for one or multiple multicasts or unicasts and once the grants have been received these information have to be considered when selecting one or more grants. Due to the complexity this approach has not been chosen for the first implementation, but left for future improvements. A more detailed analysis of the requirements of a hybrid approach is given in section 3.9.1 on page 125.

Implementation

Quite a few changes in the *Crossbar Inport* are necessary to realize the above made design decisions. First the *Routing Interpreter* needs to be adapted, in order to be able to make routing decisions for unicasts as well as multicasts. Thus a multicast routing table has to be added. As already mentioned, multicasts can be distinguished by their ID. This ID is best suited in the same field as *target node ID* in the start cell. A single bit separates unicasts from multicasts. Using the same field for both, unicast and multicast avoids an additional field as either one of them will be unused anyway.

Consequently the *Routing Interpreter* now has to look at this bit and depending on its value interpret either the global- and local- or the multicast routing table. For the FPGA prototype a fairly small number of 256 multicasts are supported to keep this table small, for an ASIC implementation on the other hand up to 64k IDs are possible, as this is the size of the *target node ID* field. Each entry of the multicast routing table is 26 bit long as illustrated in figure 3.36 on the following page. The field *Destination Ports* is six bit wide, one bit for every available LP of EXTOLL. If this bit is set then the multicast shall be forwarded to this port. The next two fields, namely *VC-0* and *VC-1* determine the outgoing VC for every destination port, depending on the incoming VC. The two fields *H* and *VC-H* determine whether the packet shall also be sent to the FU of this host and on which VC it shall do it. *Repetitions* is a redundant information as it counts the number of times the packet shall be forwarded using binary encoding. However this results in a

3 High Performance Switching

more efficient HW, as it will be shown later on. Finally, the field *V* is used to mark the routing table entry as valid. Packets arriving for an invalid routing entry are discarded and the used credits are freed.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rsvd.						V	Repetitions				VC-H		H	VC-1						VC-0						Destination Ports					

Figure 3.36: EXTOLL R2 Multicast Routing Table Entry

After the routing decision has been made, the *Crossbar Inport* needs to be able to store and read the contents as well as the meta data of the received multicast. Storing the actual contents of the multicast packet is simple, as multicasts only differ in the header and can therefore be handled the same way as unicasts. They are only stored once in the shared data buffer and read each time one repetition of the multicast is sent.

A modification is required in order to be able to store the meta information, as they are stored in a shared queue structure. There exists one queue for each outport-FCC tuple and only a single queue can be written at a time. Multicasts are destined for multiple outports, therefore it is required to either sequentially write into the shared queue structure or modify it in a way that it can write into multiple queues at once. It is required to store these information in each target queue as the *empty* signals from each queue are used as indicators to which outports requests need to be made.

As minimum sized packets require only four clock cycles to be sent in the FPGA implementation¹, writing sequentially is not possible due to the seven possible targets, which requires seven clock cycles. Writing in parallel to a single RAM is not possible either, as there is only a single write port available. Thus the shared queue structure needs to be split. It is however sufficient to part it by the amount of outports as each multicast can only be sent to each outport once and not to multiple FCCs at each outport. As a result now *N Multi-Queue-FIFOs* are instantiated with each *M* queues instead of one FIFO with *N* times *M* queues, whereas *N* is the number of ports and *M* the number of FCCs. The *Routing Interpreter* selects all *Multi-Queue-FIFOs* that are targeted by incoming packets and shifts the required information in all of them at once. During transmission only one of the FIFOs can be read, making a multiplexing structure on the read side necessary, adding a clock cycle to the route-through latency.

The *Routing Interpreter* also informs the following units on how often a multicast needs to be replicated, i.e. how often it shifted the meta-data into the queues. This information is required by the read logic of the shared data buffer in order to be able to free the occupied

¹and only two cycles in the planned ASIC implementation, due to the wider data bus

slot properly. An additional look-up table counting the amount of repetitions required for each slot is added to the shared buffer structure. Each time a packet is sent to one output port this counter is decremented and if it reached the value zero the slot is freed, i.e. the used credit is sent.

Three implementations are possible to realize this counting functionality. Either the *shift-in* signals from the *Multi-Queue-FIFOs* directly or a binary encoding can be used. The binary encoding again can be either computed in HW from the above mentioned signals or set by SW in the multicast routing table.

Using the one-hot encoded *shift-in* signals has the advantage that no RAM resources are required inside the routing table for redundant information. However one has to either count the amount of set bits, which results in a large multiplexing structure or directly use the one-hot encoded signals and clear one of the set bits every time a repetition has been sent. Once all bits are cleared, all repetitions have been sent and the packet can be removed from the buffer.

Using the binary encoding has the advantage that a simple decrement logic can be used to subtract by one upon each transmission. In addition this information has to be stored somewhere, again requiring a RAM. As the binary encoding is the most efficient in respect to RAM usage, the decision has been made to store the repetition count that way.

The multicast routing table is not as large as the tables for unicasts, as already shown in figure 3.36 on the preceding page. Due to the fact that only a small set of RAMs are available in the planned ASIC implementation, all having a fixed data width², these free bits can be used to encode the (redundant) information on how often a multicast has to be replicated to reduce the required logic resources. The same applies also for the FPGA design. A *Block-RAM* resource in the Xilinx device is required to implement the multicast routing table anyway. Whether it is used as a 22 or a 26 bit wide RAM does not matter, as a single *Block-RAM* resource can either be used as two 32 bit wide RAMs or a single 64 bit wide RAM [107].

3.8.5 Request Allocation and Grant Generation

Every time an inport of the crossbar has multiple packets for different output ports stored, it requests all destination ports according to the *iSlip* algorithm [72]. There are two possibilities for an inport regarding the remaining requests after a grant has been received

²to ease the chip layout

from any of the requested outputs. One can either keep not granted requests set or revoke all requests. The first approach reduces the required overlap time by one clock cycle for concurrent packets as all requests are already set once the currently transmitted packet reaches its end. On the other hand, this approach introduces a lot of false grants. Outputs grant a set request and wait for the acknowledge, despite the fact that the inport actually is busy transmitting to another inport and cannot process the given grant. These false grants introduce bubbles in the output data stream, a throughput of one hundred percent can therefore not be reached. This waste of bandwidth has to be avoided at all times. Consequently all inports have to revoke their requests as soon as they received a grant. The required additional clock cycle however requires a longer overlap period of transmission and requesting, thus requiring longer packets to fully leverage the available bandwidth.

In the R1 version of the crossbar all requests remained set until they received a grant, R2 now only sets requests if a new arbitration round is required, reducing false grants to a minimum and thus leveraging the full bandwidth capabilities at the outputs.

In addition, the R1 implementation used a request-grant-acknowledge scheme similar, but not exactly like *iSlip* [72]. The difference is that in R1 the *grant* generating arbiter, suited in the output, adjusts its priority with each given grant. However, not every given grant is selected by the inport. *iSlip* adjusts its priority only if the grant is also selected. According to [72] this desynchronizes the arbiters and thus increases throughput. As a very high throughput is one of the design goals, the *Port Arbiter* inside the crossbar output is adapted accordingly. More complex arbitration algorithms like [108] or [109] have been reviewed and discarded due to their complexity.

3.8.6 Fine Grain Credits

EXTOLL in its R1 state uses a rather coarse grained credit scheme. All available buffer slots were as large as a maximum sized packet, i.e. 128 byte. This results in a rather inefficient buffer usage as each packet consumes one credit, thus blocking 128 bytes of buffer space, not taking into account how large the packet actually is. Additionally, if a lot of small packets are traversing the network, back-pressure is applied rapidly through the credit based flow control, even if most of the buffer space is still unused. As bandwidth is increased, the available 32 slots are all consumed faster than credits can be freed again, thus inserting bubbles into the packet stream. This poses an even stronger problem in R1, as the MTU is increased to 256 byte.

To improve the buffer usage and therefore also increase the amount of packets in the network, it is beneficial to have finer grained credits. Hence, each packet is logically split into multiple smaller parts, but nevertheless transmitted as a whole. The available buffer is separated into smaller slots and a packet hence consumes more than one credit and buffer slot.

With these fine grained credit scheme it is possible that a packet consumes up to N credits depending on its length, N being the divider of the maximum length. The total buffer space however stays the same as it is when using the coarse grained credit scheme. However, now up to N times more packets can be stored in each crossbar inport before back-pressure has to be applied, therefore increasing the amount of packets in the network and improving the buffer efficiency.

Since packets do not include size information, the crossbar outputport can only grant the physical line if at least N credits are available, but can release unused credits again after the transmission of a granted packet is complete. The 32 available slots of the R1 design are now parted into $32 \times N$ slots, therefore making it possible to store up to $32 \times (N-1)$ packets. Figure 3.37 compares the buffer usage of a classic credit based buffer management where each packet requires a single credit (a) and the fine grained credit scheme (b). As it can be seen, the stored packets in R2 use less buffer space as in R1, thus the buffer is used more efficient.

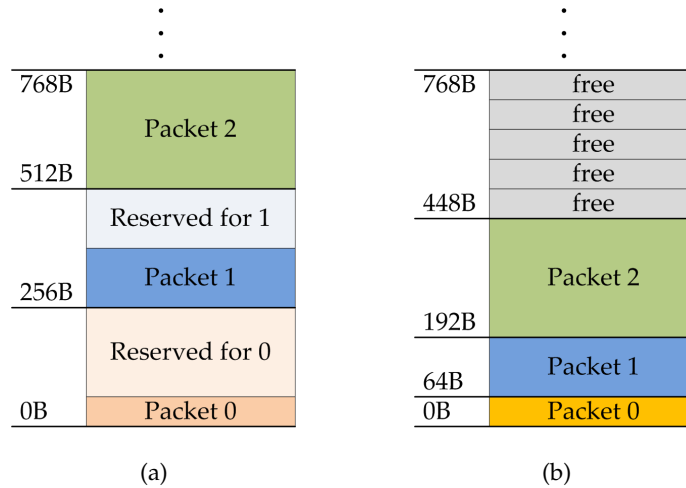


Figure 3.37: Crossbar Buffer Usage in R1 (a) and in R2 (b)

Only $32 \times (N-1)$ packets can be stored since a grant is only given if N credits are available. After $32 \times (N-1)$ minimum sized packets, only $N-1$ credits are left, therefore leaving $N-1$ slots unused, if the last packet is of minimum size.

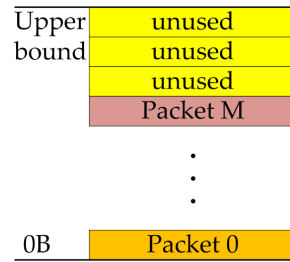


Figure 3.38: Crossbar Buffer Usage when Storing the last Packet

Figure 3.38 on the next page depicts how the last packet that fits into the buffer is arbitrated with N equals four as an example value. Before Packet M has arrived, there were four slots left inside the buffer. After Packet M has been stored, only three slots are free and remain unused until more slots become available again. The crossbar will no longer arbitrate, as it is not known how large a packet actually is in advance and the maximum packet size is four slots in this example. Not stopping the crossbar at this point can lead to a buffer overflow which has to be strictly avoided.

Since a packet can now be stored in multiple slots, it is necessary to have some sort of administration to know which slot to read next until the packet is complete. One solution is to leverage the *VOQ FIFO* for that purpose. However, this requires some sort of arbitration as multiple sources need to write and read from it at the same time. Therefore it is more efficient to use a separate administration. A small RAM is sufficient to implement a linked list where the current read slot can be used as address and the corresponding read value is the next slot to be processed.

As credits also have to be sent over a link, thus requiring bandwidth, the packet division factor may not be too high, otherwise too many credits need to be sent over the link, effectively increasing the per packet overhead.

For a FPGA implementation a factor of four is a good value, increasing the amount of packets by said factor and thus defining a credit to be 64 byte³. With that factor the linked list needs to be read every eight clock cycles to get the next slot to know where the packet continuous inside the shared buffer. The amount of in flight packets is increased from 32 to 125 and the buffer efficiency is thus increased from 3.1 to 12.2 percent for minimum sized packets with a size of eight bytes. For packets with a payload of 64 byte the efficiency is even increased from 25 to 97.6 percent.

³The maximum packet size has increased to 256 byte in the R2 implementation due to the wider data interface, $256/4 = 64$

Figure 3.39 on the facing page depicts the efficiency for packets ranging from eight byte to 256 byte. As it can be seen, the fine grain credits allows a much more efficient usage of the available buffer space, even for small packets. This scheme of fine grain credits can of

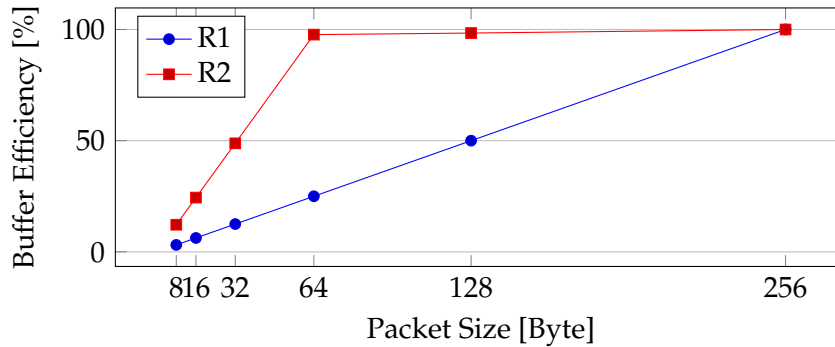


Figure 3.39: Crossbar Buffer Efficiency for a Segmentation of Four

course also be used with an even finer fragmentation, thus splitting the packet into eight, 16 or even more parts. However, the smaller the single credit is, the more complicated is the administration of the buffer itself and the required overlapped reading and arbitration, making four a good compromise between increased complexity and performance gain.

3.8.7 Debug-Ability and Maintainability

One large problem during the bring-up phase of the R2 design has been the changes that are required throughout all top-level files, whenever more debug information is to be added. These debug information need to be passed through all hierarchy levels to the *Register File*, so that it can be read during operation. Hence, the timing of these paths is always critical as they have a large routing delay in order to reach the *Register File*.

The *Register File* has been split into multiple sub-modules which are connected to the top level using registered read and write interfaces. These sub-modules are directly instantiated in each FU, including the crossbar, which not only reduces the amount of signals from or to each unit, but it also improves timing by splitting, thus pipelining the address decoder of the *Register File*. If a change in one part of the *Register File* concerning only one of connected units is necessary, then it is now no longer necessary to change all top level files, but only the unit involved. This improvement of course not only concerns the crossbar itself, but can be applied to all modules of EXTOLL. As a consequence, the SW readable debug information of the crossbar during runtime can be increased, while keeping the same timing budget. For more information on the *Register File* modifications, the reader is kindly referred to [110].

3.8.8 Early Arbitration

In R1 the crossbar detected the end of a flit solely due to the amount of data words written into the buffer. This limited the performance for small packets, as they require less clock cycles to be sent as the round trip time for a single arbitration round. This round trip time, from the detection that a new arbitration needs to be initiated, until the grant has been given and selected, is the performance limiting factor. Reducing this time increases the message rate and effectively increases the useful bandwidth for very small packets.

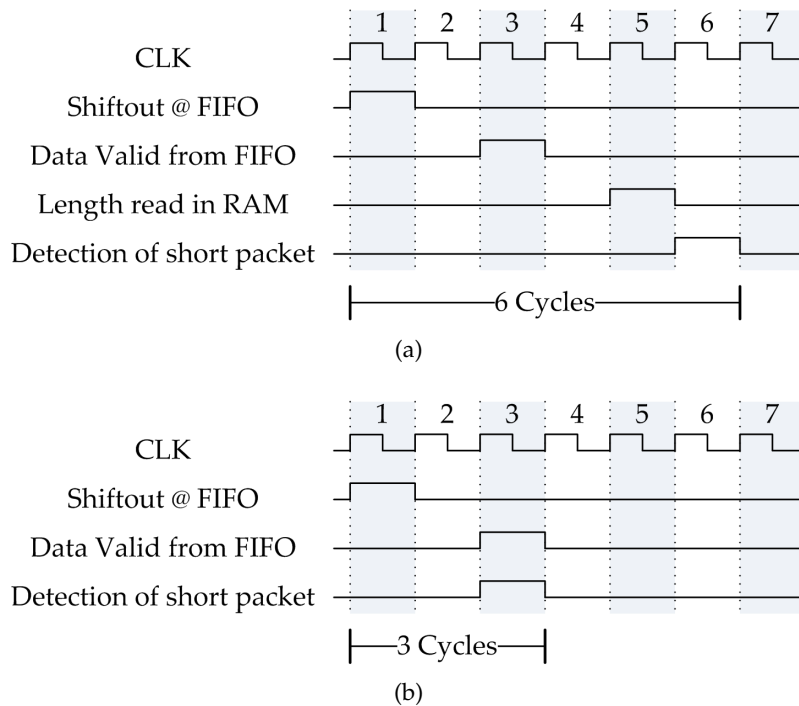


Figure 3.40: Timing Diagram without early Arbitration Detection (a) and including early Arbitration Detection (b)

The *early arbitration* mechanism targets to reduce this critical path. As already indicated, the crossbar stores data in a shared buffer and uses VOQ only on the meta-data of packets. These meta-data are written into the VOQ buffer as soon as the routing decision has been made. An additional bit has been added to these meta-data indicating that the packet has already been received completely when the routing decision has been made. This information can be leveraged to start a new arbitration round as soon as a packet has been granted by the targeted output, avoiding the need to first check the length of the packet stored in the slot of the shared data buffer. Figure 3.40 shows two timing diagrams. Figure (a) depicts all required clock cycles, from the shift out of the meta-data until the end of the packet is detected. Figure (b) shows the resulting timing diagram if an indicator that a short packet is now being processed is directly stored in the meta-data-FIFO. As it can be

seen, this mechanism reduces the arbitration round trip time from six down to three clock cycles, effectively reducing it by 50 percent. These three clock cycles allow the crossbar to fully utilize its theoretical bandwidth with packets that have a payload of only 48 bytes instead of 72 byte and increases the peak message rate from 25 to over 33 million messages per second.

3.8.9 Further Optimizations

Other than the architectural and functional changes of the crossbar, a lot of smaller improvements, both architectural and in the hardware description language (HDL) coding style have been done as well, which either increase the performance in terms of latency, achievable clock frequency, or decrease the required area of the crossbar. The following sections shortly introduce those changes.

Credit Counting Scheme

The available credits in EXTOLL are divided into small subsets that are exclusively usable by each FCC and a larger subset that is shared over all FCCs. This has been done in [89] to be able to use the whole available bandwidth with a single FCC without having the need for large buffers. Until now, the credits that are exclusively available for each FCC have been counted independently and only if the exclusive pool has been depleted, the shared credit pool has been modified. This results in a rather complicated control logic as the exclusive and shared counters need to be both interpreted and depending on its values only one of them needs to be updated.

Keeping this counting method becomes complicated, especially with the new fine grained credit scheme, as there are various possibilities the four required credits can be combined from the exclusive and the shared credit pools. In addition, the possibility that a credit is freed again at the same time has to be considered, making it unfeasible to keep this counting scheme.

	LUTs	Registers
Using a shared count	54	28
Using a total count	24	18
Percental Improvement	55 %	35 %

Table 3.6: Cost of different Credit Administration Techniques

In R2 the exclusively assigned credits still have to be counted independently of course, as it is otherwise not possible to check how many credits a particular FCC consumed. However, instead of counting only the credits of the shared pool, now a counter for all available credits is used. This new counter needs to be wider as the previous shared credits counter, as now not only the shared credits but all credits have to be counted. Though, the logic for incrementing and decrementing the total and exclusive counters is now completely independent of each other and therefore a lot simpler.

The results of this optimization can be seen in table 3.6 on the preceding page. Xilinx ISE [111] has been utilized to obtain the values with a Virtex-6 LXT [100] as a reference device. Both, the register count as well as the LUT count have dropped by almost a factor of two using a total credit count instead of a shared credit count. This effectively reduces the required area of the complete design, as the *Credit Administration* is required in each outport.

Case Coding Style

Furthermore, some code optimizations have been done improving both, area and timing of the *Credit Administration* in a Xilinx FPGA environment. The first change is to remove unnecessary *default* cases as figure 3.41 depicts. The *default* condition in the case environment does not contain any code. The Xilinx tool chain however is not able to fully detect that the *default* condition is unused and nevertheless implements an address decoder covering all cases, resulting in a larger area requirement as it is actually necessary.

<pre>case (a) 1: b <= c + 1; 2: b <= b + 2; 3: b <= b + c; default: ; endcase</pre>	<pre>case (a) 1: b <= c + 1; 2: b <= b + 2; 3: b <= b + c; endcase</pre>
(a)	(b)

Figure 3.41: Code Example where (a) Generates less Efficient Code as (b)

The performance and area improvements can be seen in table 3.7 on the facing page. The first row in this table represents the baseline implementation after the count optimizations from above, but with the full functionality of R2, meaning the support for the fine grain credits introduced in section 3.8.6 on page 108. In row two the resources have dropped by around 35 percent, only commenting out a few lines of code.

Removing the reset condition of some registers, that are set to a valid state after reset anyway, reduces the resource consumption further and also decreases the required cycle time. Not resetting registers can be done without any risk in a FPGA environment, as the architecture of a FPGA forces all registers to zero after power-up anyway, if not declared otherwise. However, not resetting registers must be avoided in an ASIC environment, as it results in an unknown state of the register. It can thus either be one or zero, which can then generate unwanted behavior of the HW.

	LUTs	Registers	Cycle Time [ns]
Baseline	476	114	3.730
no empty default cases	302	89	3.573
removed unused reset	259	81	3.261

Table 3.7: Costs of Free Slot Management Implementations

Both changes, removing unnecessary *default* statements as well as resets can of course be applied to the whole design, not only to this single unit, this unit has only been chosen as an example. The results can again be seen in table 3.7. Both, the LUT, as well as the register count dropped. The achievable clock frequency on the other hand increased, as the cycle time also decreased.

RAM Pipelining

RAMs are always a timing critical part in both, FPGA and ASIC designs. Most of the times the path to or from a RAM defines the achievable clock frequency. A register has been added at the data outputs of all RAMs, in order to make the crossbar design feasible for an ASIC implementation and improving the clock frequency for the FPGA. This however made it necessary to adapt all RAM read-out logic parts so that they can cope the additional cycle of read latency.

All units accessing a RAM inside the crossbar need to be able to hide this additional clock cycle, in order to avoid bubbles in the data stream. The implementation of these modules have therefore been adapted. Despite the additional clock cycle, all modules are capable of sending a data stream without interruption.

Free Slot Management

An efficient hardware structure has been developed in [90] that is capable of handling the management of free slots of the shared data buffer using a four clock cycle search algorithm. Due to the optimized network packet specification and increased bandwidth, the minimum packet sized dropped to one up to three phits, depending on the data bus width, making this structure unfeasible for R2.

Several choices are available to adapt this management structure. The first implementation choice that can be applied is an arbiter without any priority, resulting in a first free slot search in a single clock cycle. The arbiter required less LUTs, however the register count almost doubled compared to the search algorithm.

	LUTs	Registers
Search Algorithm	169	80
Arbiter based	143	151
FIFO based	92	59

Table 3.8: Costs of Free Slot Management Implementations

The second implementation uses a simple FIFO that is initialized with the addresses, i.e. the numbers, of all slots after each reset of the design. Results are shown in table 3.8, both values for the register as well as the LUT count dropped by using a FIFO. The results again have been obtained using Xilinx ISE [111] and are all based on the same amount of available slots, namely 32.

3.8.10 Multi-Queue-FIFO Optimizations

Until now, a generic FIFO implementation, which supports the concept of multiple queues in a single data RAM, has been utilized in the crossbar to store all meta-data. However, this generic *Multi-Queue-FIFO* offers more functionality as the crossbar inport actually requires. Removing some of these features further reduces the resource consumption and improves timing, despite the fact that it is already more efficient than having multiple discrete FIFO instantiations.

Since this module is instantiated N times in each inport, whereas N is the number of outports, even a very small optimization can have a large impact on the whole design. To get a better feeling of the required resources, all optimization steps have been evaluated

not only using synthesis, but running a complete tool flow through Xilinx ISE [111]. The results are listed in table 3.9.

	LUTs	RAMs	Registers	Cycle Time [ns]
Generic Multi-Queue-FIFO	323	0	121	3.667
Using RAMs for Pointers	151	24	76	2.890
Exclusive Read or Write	129	12	74	3.714
Using 'if' instead of in-line	122	12	71	3.621
Pipelining 'almost empty'	116	12	87	2.472

Table 3.9: Costs of Multi-Queue-FIFO Implementations

As it turned out, most of the resources were required to manage all the read- and write pointers necessary for the different queues. The first optimization therefore has the target of a more efficient pointer structure. Line one of table 3.9 depicts the starting point before the various optimization steps have been deployed. In line two, all pointers are stored using RAMs instead of registers. The amount of memory elements rises from zero to 24 and the amount of required LUTs dropped by more than 50 percent. This of course increases the read-out latency of the FIFO, as now first the pointer-RAM has to be read before the data-RAM can be read. But the smaller resource utilization and the lowered cycle time make it worth the additional clock cycle latency.

Since shifting-in and -out of the FIFO can occur at the same time, one RAM for the read- and write pointers is not sufficient, due to the fact that shifting-in and -out can be meant for different queues and therefore require different pointer sets. Each of the pointer-RAMs therefore needs to be replicated, in order to be able to read both independently in case of simultaneous shift-in and -out of different queues occurs.

This replication can be avoided, if it can be ensured that both accesses to the FIFO never happen at the same time, or if at least it can be ensured that shifting-in and -out never occurs consecutively. In the crossbar use case, it can be ensured that this is the case, as this FIFO is used for meta-data storage only and a minimum sized packet (in an ASIC implementation) is two clock cycles long. Even under ideal conditions, both read and write can therefore only occur every second clock cycle. Thus, the FIFO can be modified in a way that, whenever both events occur simultaneously, shift-in is delayed for one cycle making the access to the pointer-RAM atomically. Delaying the shift-in operation is beneficial because the time required to shift-out the meta-data is the performance critical path in the crossbar design and the crossbar is currently initiating a transmission when reading, making it possible to delay shift-in operation without any performance loss. Results can be seen in table 3.9. The required memory elements dropped by 50 percent,

due to fact that each pointer now only exist once. This also reduced the required logic elements, as compare logic can now be shared. Unfortunately the timing improvements that have been made so far are lost completely.

Further optimizations have been done to again reduce the cycle time. One change has been incredibly simple but improves timing by around 0.1 ns and also reduces the required resources. The timing improvement may not sound much, but at 200 MHz, this is a two percent improvement with almost no effort and as this unit is instantiated over a hundred times, even a very little resource consumption makes a difference. Xilinx XST, the synthesis tool of Xilinx ISE, is able to generate more efficient designs when a Boolean expression is used in an *if* clause instead of a direct assignment if the developer only wishes to set or reset a variable. Figure 3.42 depicts a code example for this optimization. The code snippet given in (a) requires more resources and results in worse timing as the code given in (b).

<pre>c <= (a < b);</pre>	<pre>if (a < b) c <= 1;</pre>
(a)	(b)

Figure 3.42: Code Example where (a) Generates less Efficient Code as (b)

The last optimization that led to an acceptable resource utilization, further draws the optimized *Multi-Queue-FIFO* away from a general purpose FIFO. The most timing critical path inside this module is the generation of the *almost empty* signal, which is required for the adaptive routing support of the crossbar. Pipelining the calculation of this signal improves timing by more than a nanosecond as shown in the last row of table 3.9 on the preceding page. The consequence of this optimization however is that in a common use case this leads to data corruption, as usually FUs depend on the correct setting and resetting of the control signals of a FIFO. In the crossbar use case however this is tolerable, as these signals are only used for the decision making of adaptive routing. The worst case scenario is that a non-optimal path is selected, but there is no possibility that packets can get lost due to an overflowing FIFO.

The results of all these improvements are, that each *Multi-Queue-FIFO* instance now requires around one third of the LUTs, around 30 register less and also has a timing improvement of around 30 percent compared to the first implementation. This is a great improvement, keeping in mind that the crossbar as it is used in EXTOLL requires one hundred of these instances.

Multicast Counter Optimizations

The *Multicast Counter* is used to count how often a packet has to be replicated before it can be removed from the shared buffer inside the crossbar inport. Essentially it is a RAM which holds a count value for each slot. This value is decremented with each transmission and once it reached zero the packet can be removed. The baseline implementation uses only registers instead of a RAM making it quite large. By rearranging the code without changing the functionality or the timing of the interface, the unit can be reduced by a factor of 30 for LUTs and a factor of almost ten for registers as it is illustrated in table 3.10 on the next page.

```
always @(posedge clk)
begin
    if (wen_a)
        mem[a] <= data_a;
    else if (wen_b)
        mem[b] <= data_b;
end
```

(a)

```
always @(*)
begin
    if (wen_a)
    begin
        address = a;
        data     = data_a;
    end
    else
    begin
        address = b;
        data     = data_b;
    end
end

always @(posedge clk)
begin
    if (wen_a || wen_b)
        mem[address] <= data;
end
```

(b)

Figure 3.43: Code Example where (a) Generates less Efficient Code as (b)

The reason that caused the inefficient result after synthesis is that Xilinx XST is not to be capable of combining a register array to a RAM if there are multiple addresses as a source, even if the write access is exclusive. The solution is to asynchronously assign the resulting address to an intermediate signal and register this signal afterwards. The code examples given in figure 3.43 illustrate this problem. The code in (a) results in a sea of registers. The code in (b) however results in a single RAM. The first *always* block in the second example asynchronously selects the address and the second block uses a clock to actually write the data into the memory.

The results of the different coding style can be seen in table 3.10 on the next page. The

	LUTs	Registers	RAMs
Synchronous Logic	693	408	0
Asynchronous Logic	23	42	2
Percental Improvement	97 %	89 %	-

Table 3.10: EXTOLL R2 Multicast Counter Code Optimization

required amount for both LUTs as well as registers dropped by around 90 percent for the cost of two additional RAMs.

3.8.11 Overall Architecture

After all design changes have been applied, it appears as if the crossbar itself did not change that much, seeing it from a top-level point of view. Nevertheless, the code basis for the crossbar changed significantly. The resulting block diagram for the crossbar import can be seen in figure 3.44.

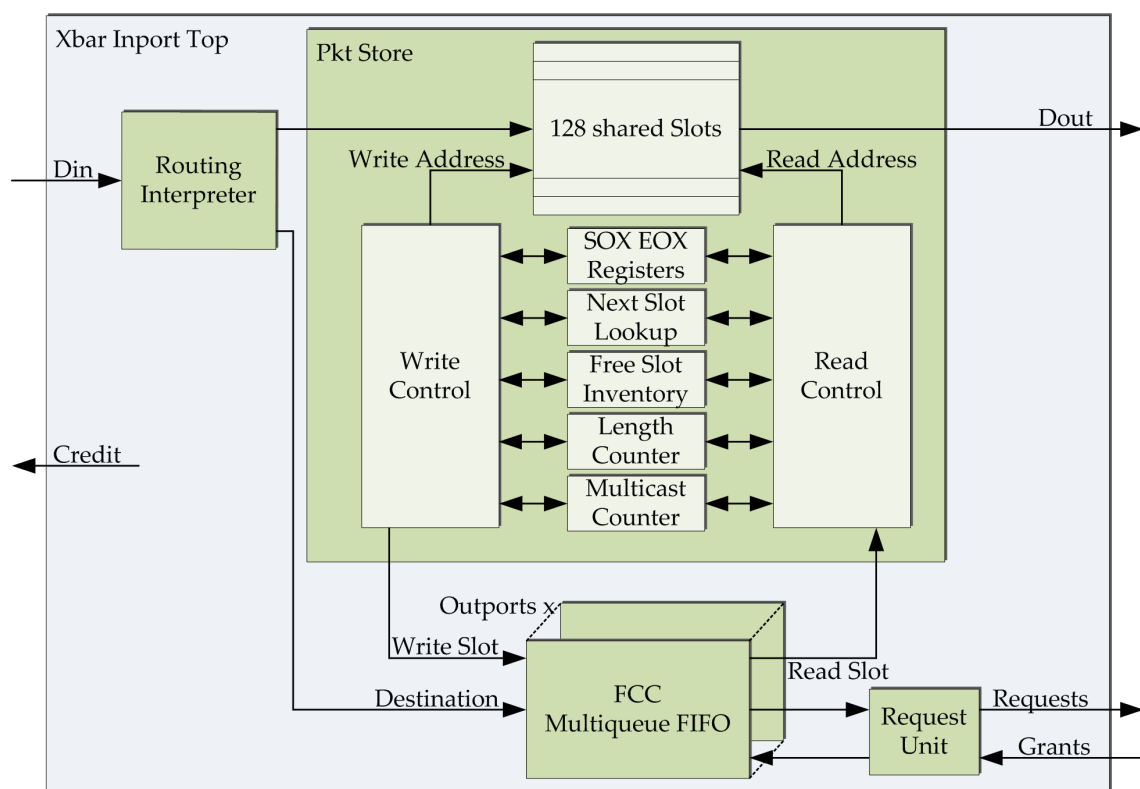


Figure 3.44: EXTOLL R2 Crossbar Import Block Diagram

The interfaces to and from the crossbar are now less complex, due to the simpler packet

format. Inside the crossbar, only a few architectural changes were necessary to add above described functionality. The *Routing Interpreter* has been changed from source-path to table-based routing, making only internal changes. Largest change involves the *Multi-Queue-FIFO*, which is now split into multiple smaller units to be able to support multicasting. The *Packet Store* now supports 128 slots instead of only 32 and features an additional *Multicast Counter* that counts how many times a multicast packet still needs to be replicated. The last modification is required for the fine-grain credit scheme. A *Next Slot Lookup* table has been added to the *Packet Store* implementing the linked list. Added *Register File* modules are not illustrated in figure 3.44 on the preceding page.

The crossbar output remained rather unchanged in its functionality, the only added feature is the increased credit counter width and the feature to internally free credits again in case transmitted packets did not consume all granted credits.

3.8.12 Evaluation

Multiple configurations have been synthesized in order to see the scalability of the crossbar. In all configurations, the register file has been neglected, to get only the results of the crossbar itself, despite the fact that the register file is partitioned and included in R2 of the crossbar.

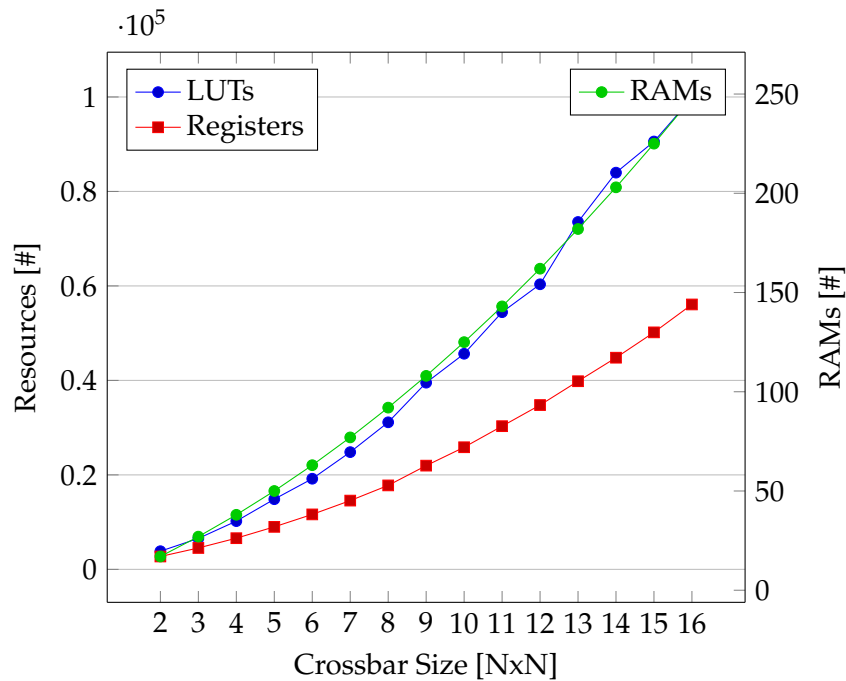


Figure 3.45: EXTOLL R2 Crossbar Resource Utilization

	LUTs	Registers	RAMs
2x2	3 846	2 726	17
3x3	6 552	4 532	27
4x4	10 211	6 599	38
5x5	14 853	8 999	50
6x6	19 187	11 636	63
7x7	24 819	14 558	77
8x8	31 134	17 775	92
9x9	39 519	21 959	108
10x10	45 642	25 852	125
11x11	54 442	30 311	143
12x12	60 351	34 785	162
13x13	73 536	39 827	182
14x14	83 981	44 807	203
15x15	90 573	50 171	225
16x16	99 776	56 078	248

Table 3.11: EXTOLL R2 Crossbar Resource Utilization

The amount of NPs have been kept the same for all configurations, so only the amount of ports reachable using adaptive routing increases. This is more resource intensive as adding multiple NPs, but shows the scalability even when increasing the logic required to calculate the destinations for adaptive routing. The results are shown in table 3.11 and figure 3.45 on the previous page. As it can be seen, the size increases slightly more than linear with growing port numbers, but not quadratically or even exponential as claimed by [78].

Performance

The Performance numbers of the crossbar design have been extracted from simulation and are based on the Virtex-6 [100] design of EXTOLL, which runs at 200 MHz. The data interface of this design is 64 bit wide and therefore has a peak bandwidth of 1.6 GB/s per direction and port. The complete design is a full duplex ten by ten crossbar and therefore offers an aggregated peak bandwidth of 32 GB/s.

As indicated in figure 3.46 on the facing page, the crossbar is capable of sending more than 33 million messages per second for small messages with a size of up to 24 byte. The efficiency almost grows linear with the length of the packets and has its peak at around 98 % as illustrated in figure 3.48 on page 124. As shown in figure 3.47 on the facing page analogue to the efficiency, also the bandwidth increases with its peak at 1505 MB/s per

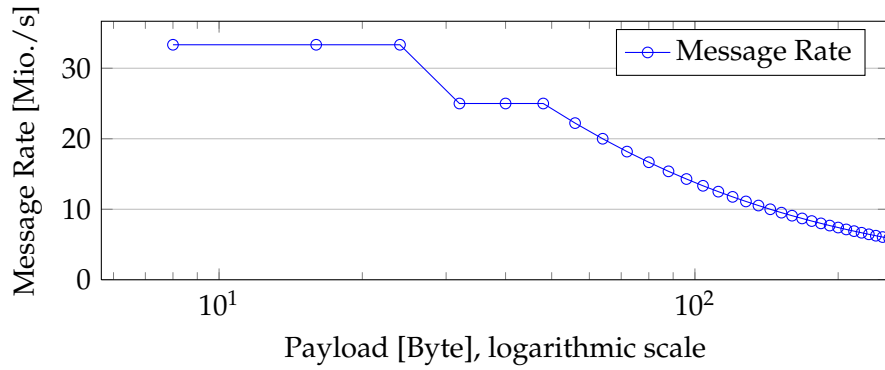


Figure 3.46: EXTOLL R2 Crossbar Message Rate

port. Measured values can vary significantly, as the performance is also affected by all surrounding units, like the host interface, the FU or the LP.

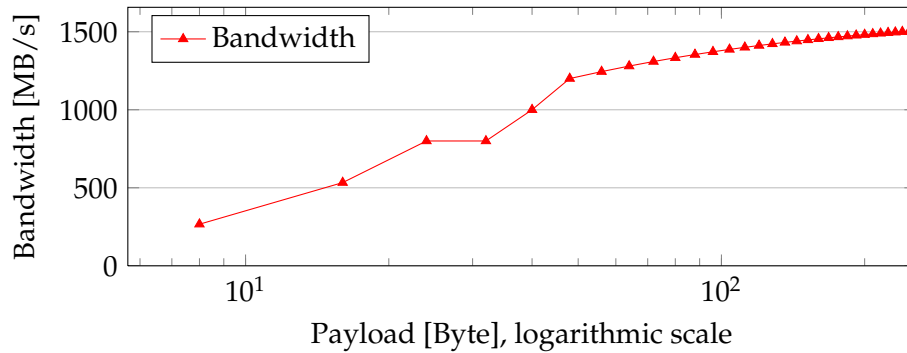


Figure 3.47: EXTOLL R2 Crossbar Bandwidth

Figure 3.49 on the next page shows the route-through latency in a quite network. Shown is a timing diagram with all required pipeline stages from reception a packet at the inport to the transmission to another outport. The difference is 16 cycles, which is 80 ns at 200 MHz. Compared to the R1 crossbar, three more cycles are necessary. However, due to the higher achievable clock frequency the route-through latency is nevertheless smaller, although only by 3.2 ns. The crossbar is targeted for 750 MHz in the ASIC design, which results in a route-through latency of only 22 ns. Adding the same amount of clock cycles to let the packet pass through the LP and serialization, results in a hop latency of less than 44 ns. Compared to the 105 ns of Cray's Gemini interconnect, this is an outstanding value. The hop latency can of course be higher if other packets are stored in the inport buffer.

However, achieved latency is three cycles more than the R1 version of the crossbar. The additional clock cycles can be explained through the changed routing algorithm, which now requires a RAM access, due to the multiplexing structure added at the VOQ and the pipelined RAM accesses. The remaining modules of the crossbar actually reduced or

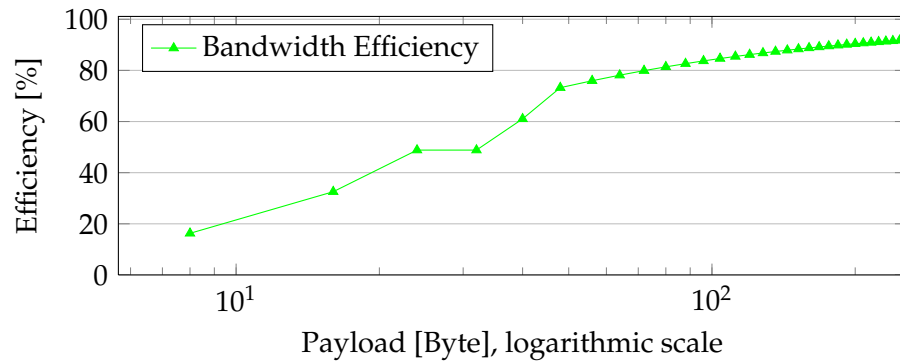


Figure 3.48: EXTOLL R2 Crossbar Bandwidth Efficiency

kept their latencies. As a result, the design achieves higher clock rates and includes more functionality than before.

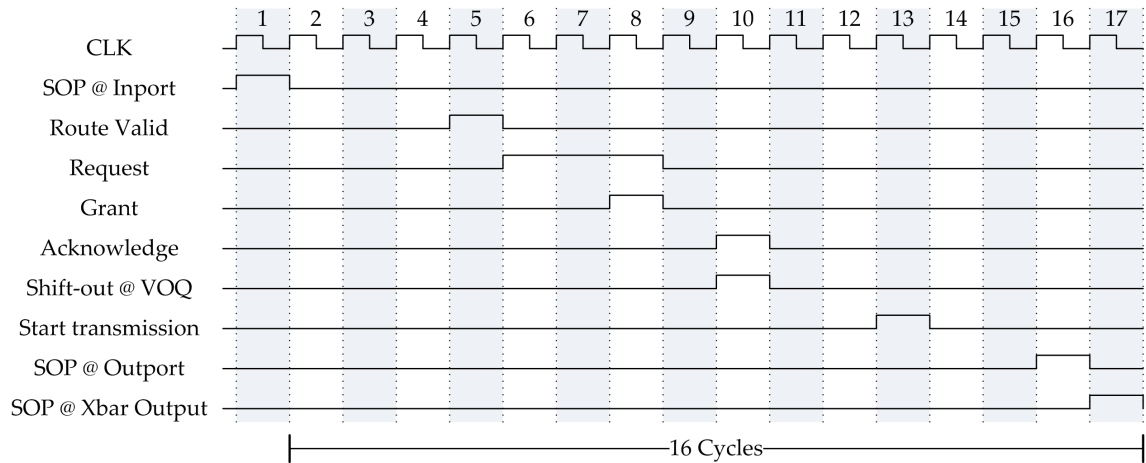


Figure 3.49: EXTOLL R2 Crossbar Route-Through Latency

Cost

The cost of the design can best be measured in the required resources on the FPGA, used in the testbed platform shown in figure 3.50 on the facing page. In this case it is a Xilinx Virtex-6 LX240T [100] device. Table 3.12 on the next page shows the resources utilization. The required resources correspond to around 34 percent of the available resources. Compared to the R1 resource utilization, which can be seen in table 3.2 on page 92, the size increased almost by a factor of three. This is mostly due the now included control and status register file, which alone consumes around 5 700 registers and 6 100 LUTs, as the difference between the first two rows in table 3.12 on the next page indicate. Without the register file modules the crossbar only consumes around 29 percent of the

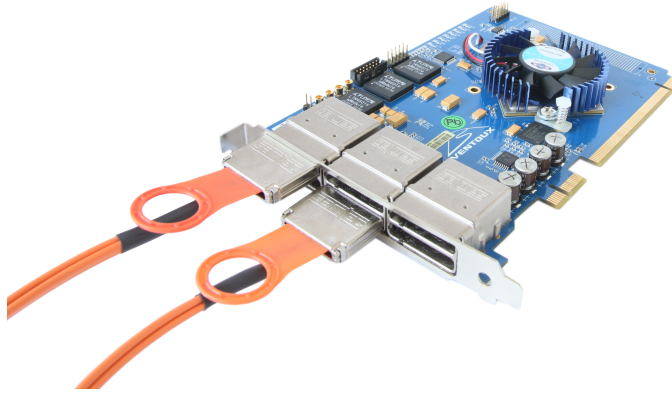


Figure 3.50: EXTOLL Ventoux

targeted FPGA. Further, the increase of the buffer space from 128 byte to 256 byte, the relatively large routing table in each inport, the added support for adaptive routing, the additional inport and outport and the added multicast feature increase its size. Compared to the previous platform utilized in R1, the resource consumption stayed rather constant in respect to the total available amount of resources.

	LUTs	Registers	RAMs
including Register File	51 415	32 018	125
without Register File	45 642	25 852	125
Difference	5 773	6 166	0

Table 3.12: Virtex-6 Synthesis Results for a 10x10 EXTOLL R2 Crossbar

3.9 Outlook and Future Improvements

The current design of the crossbar performs very well both in simulation and in EXTOLL R2, as it is shown in section 4.6.5 on page 166. Nevertheless there is still room for improvement. Most of the following improvements require a very high hardware effort and are therefore unfeasible to be tested in a FPGA. Nevertheless these optimizations are interesting in a research point of view and are therefore worth mentioning.

3.9.1 Multicast Optimization

The current implementation of multicast support is completely sequential, as described in section 3.8.4 on page 102. This is the slowest, but most simple way to integrate multicasting

in the crossbar design. Consequently this offers a lot of room for optimization. To improve the performance of multicast transmission only an hybrid approach is feasible, as the parallel transmission requires the stalling of other transmission.

The hybrid approach works as follows. Whenever multiple multicast targets are ready at the same time, transmit to all of them. However, if only one destination is available, send it only to this one. This does not require that all targeted outports of the multicast are ready to initiate a transfer at the same clock cycle and thus avoids HOL, but at the same time has the potential to speed up the transmission of the multicast. In the best case, a multicast packet is transmitted completely in parallel, in the worst case it is transmitted completely sequential.

In order to implement this properly, it is required to compare all slot numbers that currently request an outport, as the slot number uniquely identifies a packet and can therefore also be used to identify whether requests are set for the same packet or not. This is currently not possible, as the instantiated *Multi-Queue-FIFO* has only a single data output shared for all queues, which does not provide a lookahead feature for the next item in a queue. Moreover, the logic to compare all slot IDs does not scale, as there are N times M queues to compare, whereas N is the number of outports and M is the number of FCCs.

In addition, a decision has to be made which multicast to send, if there are multiple different packets requesting at the same time and a subset of them receives grants. The most efficient solution is to select the multicast which received the most grants. However, this requires even more comparison logic, as the one-hot coded grants have to be counted per multicast first and then all results of the different packets must be compared. All this is impossible to do in a single clock cycle and pipelining this decision process adds latency to the performance limiting path. Adding multiple clock cycles requires to further overlap transmission and arbitration, resulting in a lower achievable message rate and making it more difficult to fully utilize the available bandwidth.

3.9.2 Read Slot Lookahead

The performance limiting path inside the crossbar in respect to achievable message rate is from the point a request is set until the transmission starts. Essentially this path limits the message rate and achievable bandwidth for very small packets as bubbles are introduced due to delays introduced here.

One way to improve the timing at this path is to reduce the time required for slot lookup

of a granted request. This can be achieved by adding a look-ahead functionality to the *Multi-Queue-FIFO*, which is leveraged for the meta-data storage. As a shift-out at the *Multi-Queue-FIFO* first triggers a read of the current read pointer and then a read of the actual data, it has a latency of three clock cycles until the actual FIFO value can be read at the output side. This read latency can be completely hidden using a register for each queue containing the next value of the corresponding queue. Whenever this value is read, the next value will be read from the FIFO and stored again in said register, requiring again three cycles. This read latency for the next value however does not affect the performance, as the crossbar is currently busy processing the current packet.

	LUTs	Registers	BlockRams
Baseline 9x9	39519	21959	108
Lookahead FIFO	62529	31898	108
Percental Increase	58 %	45 %	0 %

Table 3.13: Read Slot Look-Ahead Resource Utilization for an Example Size

Theoretically this removes two cycles read latency which improves the message rate from 33 to 50 million messages per second and allows a bandwidth of 400 instead of only 267 MB/s for eight byte sized packets. The drawback however is that this latency improvement requires a lot of resources. This *Multi-Queue-FIFO* is instantiated N times in each inport and each instance has M queues, whereas N is the number of outports and M is the number of FCCs, therefore requiring N times M times data width registers per inport. In addition, the logic to implement the lookahead is quite complex. Table 3.13 shows the increased area requirements for a nine by nine crossbar. As it can be seen the amount of LUTs and Registers both increase by around 50 percent, which is an unacceptable increase. Thus, this improvement is not feasible for the current design. The increase of more than 50 percent is simply not acceptable, taking into account that this FIFO is required in each inport.

3.9.3 Removal of Erroneous Packets

Until now packets are forwarded to its recipient not taken into account whether the packet is tagged erroneous or not. The reason for that is that the *error tag* is located at the very end of each packet to avoid the need to store and forward the packet at each hop through the network to check and tag it accordingly. However, if a packet has been received completely at the crossbar inport before the transmission to the outport started, it is theoretically possible to drop the packet and therefore reduce the load of the network. The retransmission of the LP makes sure that an error free version of this packet will arrive

directly after the erroneous version. If a packet is dropped, several control structures need to be updated. First of all, the credits that the packet consumed need to be sent back. Otherwise the crossbar runs out of credits after some time and that port will be stalled until the management SW resolves this situation. Secondly, the meta-data structure needs to be updated and finally the occupied slots need to be marked as free again.

Basically there are three possible times when a packet can be extracted, as illustrated in figure 3.51.

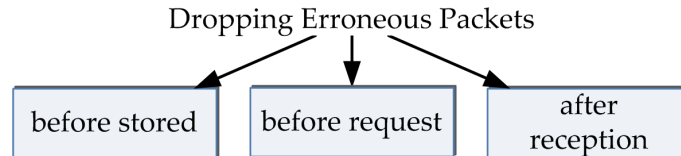


Figure 3.51: Dropping Erroneous Packets Design Space Diagram

The first option can only be applied to very short packets, where the end character arrives before the routing look-up is complete. Then the meta-data do not need to be stored, the used slot in the shared buffer however needs to be freed again and also the used credits of this packet must be sent to the connected unit to avoid losing credits.

The second possibility is to drop the packet right before the request to the output is set. This option has the advantage that, at least for unicasts, the meta-data can be accessed easily as it is the next value to be shifted out of the VOQ and for multicasts at least the transmission to the current target can be avoided. Additionally, the extraction occurs on the 'correct' side if the *Multi-Queue-FIFO*, as the request side is responsible for shifting data out of said FIFO. The problem however is that there must be some sort of structure that stores the information, whether the current transmission is for an erroneous packet or not. This structure then must be checked before a request is set without actually knowing the slot number it is stored in. Alternatively the slot can be read first, inflicting additional latency. This can be avoided with the *read slot lookahead* as introduced in section 3.9.2 on page 126. However, putting that feature in place introduces a lot of additionally required resources. Multicasts need to be specifically tagged inside the *Multi-Queue-FIFO* in order to be able to identify them as such and it has to be made sure that the control logic counting the number of required repetitions is changed properly to again avoid credit loss.

The third and last option is to remove the packet once it has been received completely, avoiding the need to have additional structures which store the packet integrity. However, currently the shift-in side of the FIFO has no support to remove words again. A support to decrement the write pointer inside the FIFO is required to support this feature, as well

as the above mentioned necessity to send the correct amount of credits. The required logic to revoke already stored data is introduced in section 2.2.2 on page 17. This third option can only be applied if the transmission of the erroneous packet to the targeted output has not yet started. Nevertheless this option is the most efficient way to reduce erroneous traffic through the crossbar. Due to its complexity however, it has not yet been integrated.

3.9.4 Balance Packet Travel Time

Balancing the time a packet requires to traverse the network has the advantage that communication jitter is reduced, making it possible for applications to better hide latency. In order to balance the travel time of packets a more sophisticated arbitration algorithm than the currently used *iSlip* algorithm in the crossbar is required.

Adaptive routing also balances the travel time of packets by using paths that are potentially least congested. Potentially because the view of the network is only local at the current node in the used implementation, whereas the network can be congested at a later stage of the path. In contrast to the already implemented adaptive routing, this mechanism keeps packets in order, avoiding the requirement of a reorder buffer in SW.

To be able to assign a priority to packets according to their age over all VOQs requires some sort of aging information. The simplest way to implement aging is a counter that is always incremented with each packet that is written to the data buffer. This however requires a corner case conditioning whenever a counter overflow is imminent. All currently stored packets require a new age, still maintaining their order. Doing this requires to go through all parallel queues and assign a new age to all values in respect to the previous age. The width of this counter decides on how often this time consuming re-aging has to be done, general functionality must be turned off in that time to maintain the order of packets. This of course has a bad influence on latency and also increases jitter again. Moreover, the administration of the age information requires a structure with a size of counter width times available credits. A RAM cannot be used, as all aging information, at least for all packets first in line at each queue, need to be available in one clock cycle in order to be able to compare them. The required real estate for that structure and the compare logic alone renders this functionality unfeasible.

3.9.5 Split Inport Into Smaller Units

To increase the performance of the crossbar it is possible to split the inport into multiple smaller units which can then work independently of one another. A splitting can be done after the *Routing Interpreter*, using not one, but N data buffers and *Port Request Units*, whereas N is the number of outports. Data has to be delayed until the routing decision has been made in order to be able to store the data into the correct buffer instead of simply storing it in a single shared buffer as it is done now. This on the other hand increases the route through latency for packets. The advantage is that the required arbiter inside the *Port Request Units* is smaller by a factor of N , increasing its potential for higher clock rates. The integration of a multicast support without any latency drawbacks, as introduced in section 3.9.1 on page 125 using the parallel approach, is then also done by design, as all sub-inports work independently.

The network can recoup faster from back-pressure, as there is a very high read bandwidth. This however is also the main drawback as write bandwidth is a factor of N smaller, it is not balanced and the available read bandwidth cannot be used most of the time. In addition, the increased resources are not to be underestimated. As credits are assigned per FCC and not per target outport, each data buffer must be large enough to hold the maximum amount of packets. Essentially this results into a N times larger buffer requirement for each inport without benefiting from it, effectively reducing the buffer efficiency by a factor of N .

As the crossbar is already a RAM centric design, requiring a high percentage of the die area solely for said RAMs, this approach will not be implementable in the near future, the resulting design does not scale.

3.10 Crossbar Summary

The new EXTOLL crossbar has been improved in many aspects compared to R1. The design is now capable of running at higher clock rates. The message rate has increased by over a hundred percent from around 15 million to 33 million messages per second. Bandwidth has been improved due to the wider data bus and increased clock rate as well. The overall bandwidth efficiency has been improved leveraging the new packet format and reducing the unused clock cycles between small packets.

The change with the most impact is the added capabilities for multicasting. It enables

EXTOLL to implement an efficient support for collective operations. The routing algorithm has been changed from source-path routing to table-based routing, now supporting arbitrary topologies and up to 64k nodes without any path length restrictions. Adaptive routing has been added to be able to distribute traffic equally over multiple paths, reducing the possibility of congestions. All this has been achieved still maintaining the small memory footprint of R1 with very limited routing capabilities. Despite all these changes the route-through latency of the crossbar has been maintained at around 80 ns for a FPGA design and an astonishing 23 ns in an ASIC implementation.

4 EXTOLL

EXTOLL [101] is an interconnection network that has been developed at the CAG for large scale clusters. During its development process, special care has been taken to achieve a very low latency, a high bandwidth, a good fault tolerance and a high availability. In large scale clusters it is important to have as little communication overhead as possible to increase its scalability. This has been achieved by developing a lean protocol and reducing the amount of protocol changes to a bare minimum. Additionally, every pipeline stage throughout the whole design has been revised to keep the time required for a single message as low as possible.

A high fault tolerance and availability is required to be able to get a cluster up and running and keep it operational. It is simply unfeasible to halt a complete cluster every time a single node has a malfunction, which occurs often for example due to failed cooling.

During the design phase of EXTOLL all available levels from the link level in HW up to the user application in SW have been analyzed in respect to these aspects. An FPGA prototype has been developed to prove that all made design decisions not only work in theory, but can actually be realized as [93] shows. With this prototype a half round trip latency has been achieved that has never been reached at the time being, even with ASIC implementations.

This work focuses on some of the module implementations of EXTOLL. However to understand the basic functionality an overview of all available units and their roles is given. For a more detailed view of other parts, the reader is kindly referred to [112], [94], [97] and [113], which explain the functionality of the other modules and the corresponding SW in greater detail.

4.1 HW Overview

The HW part of EXTOLL can be parted into three main blocks as shown in figure 4.1. The first block is the host interface which is used to communicate with the host CPU, enabling communication of EXTOLL with the rest of the node.

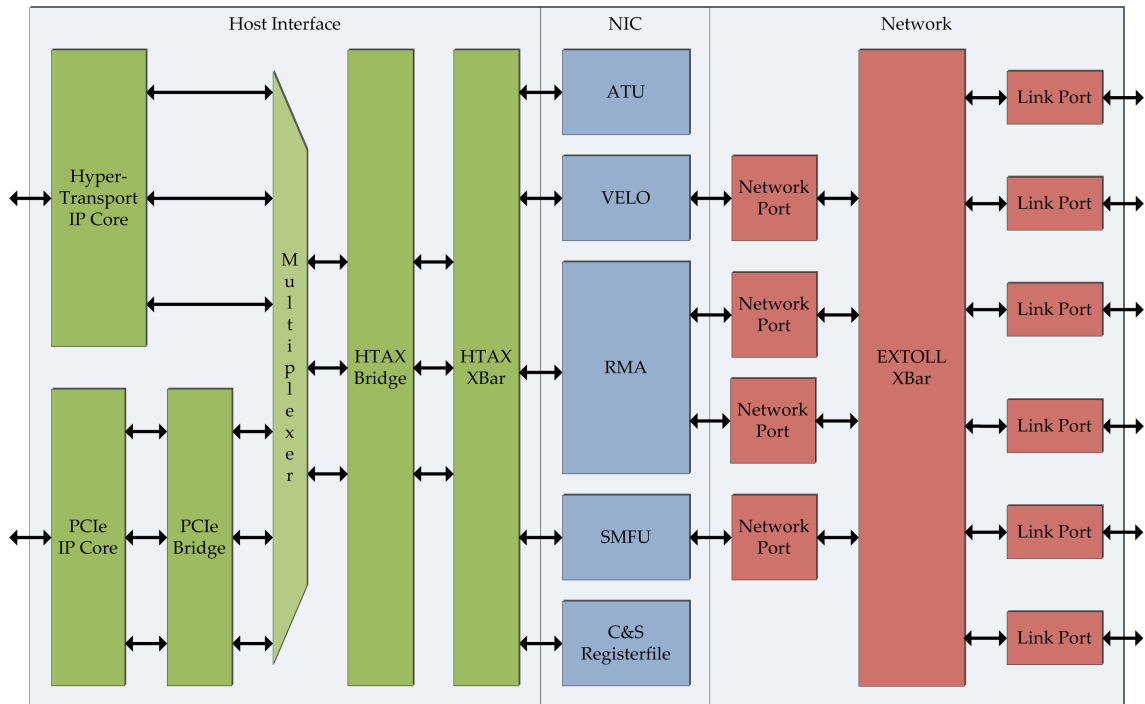


Figure 4.1: EXTOLL Block Diagram

The second block offers the actual NIC functionality. It is used to transform the command- and data packets sent through the host interface to network packets and vice versa. The NIC features the VELO unit for a low latency transfer of small messages using two sided communication, which will be described in more detail in section 4.6 on page 153. For larger block transfers the RMA unit is leveraged, featuring one sided communication. The Address Translation Unit (ATU) offers fast address translations from virtual to physical addresses for the RMA. The Shared Memory Functional Unit (SMFU) enables multiple nodes to share their main memory without special SW requirements. The last unit in this block is the *control and status register file* required to configure EXTOLL as well as acquire debug and status information.

The last block is the network itself including the crossbar described in chapter 3 on page 63. The connection between the network and the NIC is done using NPs, described

in more detail in section 4.5 on page 151. The LPs are used for the physical connection between nodes.

The following sections first describe all modules of EXTOLL that have been implemented in a team effort at the CAG and have thus not been part of this thesis. Nevertheless, as EXTOLL only works as a whole, each unit and individual contribution is important to be able to achieve the outstanding performance values.

4.1.1 Host Interface

The host interface, which is depicted green in figure 4.1 on the preceding page, is used to connect EXTOLL to the host system. Today's mainstream CPU architectures offer three different peripheral interfaces: HT [25], PCIe [29] or QPI [27]. The preferred host interface of EXTOLL is HT, as it offers a very low latency through a direct connection to *Advanced Micro Devices (AMD) Opteron* processors [114] [115] without intermediate switch- or bridge chips. In addition, it has explicitly been developed to not only connect multiple CPUs in a multi socket system, but also support peripheral connections through an expansion slot called HTX. The specification is openly available and may also be used freely for research purposes. EXTOLL uses an HT-core that has been developed at the CAG as part of a diploma thesis [26]. The core offers a high throughput and low latency interface leveraging all advantages of HT.

As alternative interface QPI is also of interest. It offers comparable performance and functionality as HT, directly connecting Intel CPUs [116] with each other. It has been developed for the same purpose as HT. The main drawback however is, that the specification is not open and freely available and it is hard to obtain a license and thus the right to use this interface. In addition, an explicit specification of an expansion slot is missing, making it necessary to adapt a CPU socket connection for that matter.

To be able to target a more wide spread environment and not be limited solely to the AMD ecosystem, the choice to not only integrate HT but also PCIe has been made. PCIe offers less performance than HT due to its protocol overhead and the fact that in most available machines PCIe is not directly connected to the host CPU, but requires either one or multiple switch- or bridge chips. The latter drawback is likely to change, since more and more CPUs offer or will offer a direct PCIe interface, thus reducing the latency penalty to a minimum. Examples for such processors are Intel's *MIC (Many Integrated Cores)* architecture [116] [117] or their *Ivy Bridge* generation of server CPUs.

PCIe is the quasi standard for adapter cards and therefore it is required to also have PCIe as host interface in EXTOLL, in order to be fully compatible with now available and future systems. Therefore it has been added as a second option in EXTOLL.

Other host interfaces are no longer existent in today's ecosystems and therefore not discussed here. Examples are Industry Standard Architecture (ISA), Peripheral Component Interconnect (PCI) or PCI-eXtended (PCI-X). All these interfaces no longer offer a competitive latency or bandwidth for any application.

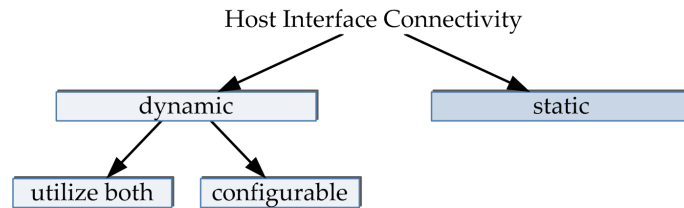


Figure 4.2: EXTOLL Host Interface Connectivity Design Space

The difficulty when designing a chip with multiple and incompatible host interfaces is the question how to connect these with the internal FUs in an efficient way. Figure 4.2 depicts the design decisions that have to be made in that case. Essentially there are two options, the first option is to use a dynamic approach. Dynamic means in that case that the interfaces can either be used concurrently or that SW can switch between them during runtime. However, changing the interfaces dynamically during runtime causes various problems. One is that the order of packets sent from EXTOLL to the host cannot be maintained during the transition, as the interfaces have different latencies and no synchronization mechanism. This can cause problems for some applications.

Another and more serious problem are outstanding read requests which can occur in both directions. It has to be made sure that the receiving part can complete transactions if an interface switch between two or more requests is done. Both problems can be avoided when it is made sure that EXTOLL is idle during the switching process. A physical problem however occurs when connecting two or more host interfaces to a chip. The connectivity required to and from the chip, due to the high bandwidth offered by modern interfaces, is tremendous. This can be resolved when designing a new mainboard for a system and directly placing the chip on that printed circuit board (PCB). This however is very cost intensive and has to be repeated with every processor generation or family. When using EXTOLL on an add-in card, that problem cannot be resolved as PCIe and HT have interfering physical constraints for their add-in cards, making it impossible to comply to both constraint sets at the same time.

The second option is to statically switch between the interfaces. This means that a multiplexer is used to switch between the interfaces at a point where both have a common interface. That multiplexer can only be changed either on power-up, i.e. before the system has booted or is fixed for all time. Consequently only one of these two interfaces can be used during operation. This keeps the HW simpler, as no support for multiple interfaces is required in the FUs.

As HT does not support hot-plugging¹ and due to the fact that designing a new PCB for a host system is too expensive, the static approach is to be favored for the first version of EXTOLL. The concept of using two interfaces however remains interesting for future implementations, once the bandwidth offered on the network side exceeds the host interface capabilities. In this case it might be interesting to use HT only for latency sensitive applications and PCIe for other communication types, like storage or control applications. Another possibility is of course to implement the same host interface twice and leverage the increased bandwidth in the same way.

As EXTOLL started as a development solely for HT, the internally used HyperTransport on-chip Protocol (HToC) packet specification has been developed to reduce the packet translation to a minimum when translating the packet from HT to HToC. This translation is required to add information needed for the routing of packets through the on-chip crossbar HTAX described in section 4.1.2.

Thus, a bridge unit is necessary to map the PCIe protocol to HToC, this unit is described in detail in section 4.4 on page 147.

4.1.2 HTAX

Some sort of switching is required to be able to connect multiple FUs to a single host interface. In EXTOLL, this role is fulfilled by the HTAX. It connects all FUs present in the NIC part of EXTOLL to the host interface. It is a protocol agnostic on-chip network specified and developed by Dr. Heiner Litz [113] [118]. It is capable to send packets as short as two phits back to back, ensuring the highest possible bandwidth one can achieve on chip. In contrast to an off-chip network, the focus of this crossbar design is to be able to send small packets, based on the host interface MTU very efficiently and to avoid buffering. Additionally, the design does not require a complex flow control or features

¹according to the specification hot-plugging is supported, but has never actually been implemented in the AMD Opteron processors, thus making it unusable.

like multicasting. Thus, the arbitration needs to be faster than in off-chip networks, to be able to efficiently use the available bandwidth.

The HToC protocol is utilized on top of the HTAX by all FUs. It has also been developed by Dr. Heiner Litz [119] as part of an internal work paper. It is kept close to the HT specification to avoid latency inflicting protocol translation layers between the host interface and the FUs.

4.1.3 Register File

The Register File (RF) is used to control all functionality offered by EXTOLL. Additionally, all debug information are collected here and can be read by SW. The RF is automatically generated using the Register File Surrogate (RFS) tool chain. RFS is also an internal development at the CAG [110].

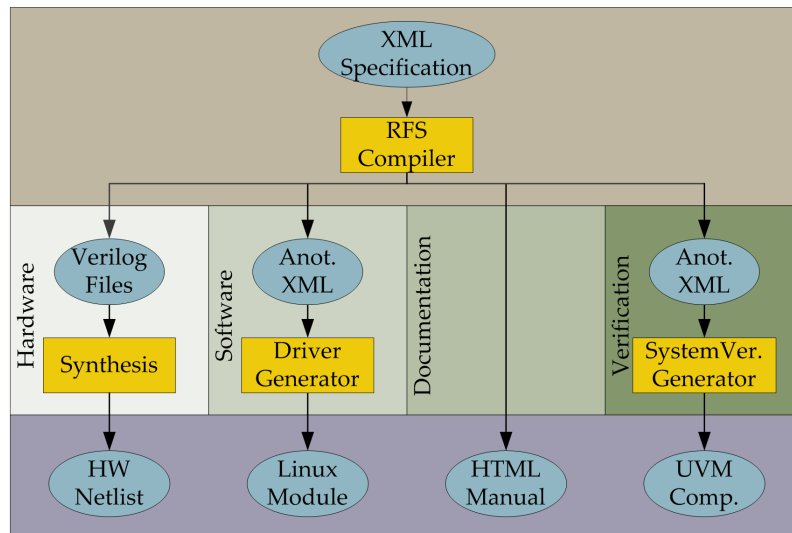


Figure 4.3: EXTOLL Register File Tool Flow

This tool chain reads a XML based description, describing all required registers defined by the designers of each module, and generates the whole environment required for a successful implementation, from the RTL code over the Linux kernel driver, the verification environment to the documentation as illustrated in figure 4.3. Using RFS ensures the integrity of all four important blocks at all time, reducing the risk of inconsistencies between one of these blocks.

4.1.4 ATU

The ATU supports the RMA unit with the translation of virtual addresses from a thread to physical addresses [120]. Usually this task requires the infliction of the OS, resulting in a scheduling of processing threads on the host CPUs and therefore adding a lot of latency. With the help of ATU however this time can be significantly reduced, without losing the needed separation of threads and security. Essentially it can be seen as a Memory Management Unit (MMU) featuring a Translation Look-Aside Buffer (TLB). For more information about the ATU, the reader is kindly referred to [120].

4.1.5 RMA

The RMA unit has been developed to directly access main memory of remote nodes using DMA methods and *put* and *get* style communication [120]. It is specialized to write or read larger chunks of data without interfering the application during the whole operation. The SW only has to write a descriptor to the unit with either a virtual- or physical address from where to where data has to be written or read. The RMA then works independently on this task and notifies the application once it has finished the operation, thus offering complete asynchronous communication.

An address translation is necessary in case virtual addresses are used to communicate, as RMA needs to apprehend the physical addresses to read from or write to. This required address translation is done leveraging the supporting unit ATU, described in section 4.1.4.

Compared to VELO, the start-up latency of the RMA is higher. As the message size increases, the latency gap between these two units decreases and at some point it is more effective to use RMA instead of VELO, as illustrated in figure 4.4 on the next page. The given figure shows results of the first revision of EXTOLL, where both units were implemented less pipelined and have now been redesigned completely, but the relation between the two will remain roughly the same. [120] and [94] offer more detailed information on the RMA unit.

4.1.6 SMFU

The SMFU is the third sending and receiving unit in the NIC part of EXTOLL. It provides a way to forward load and store operations to remote memory locations without SW

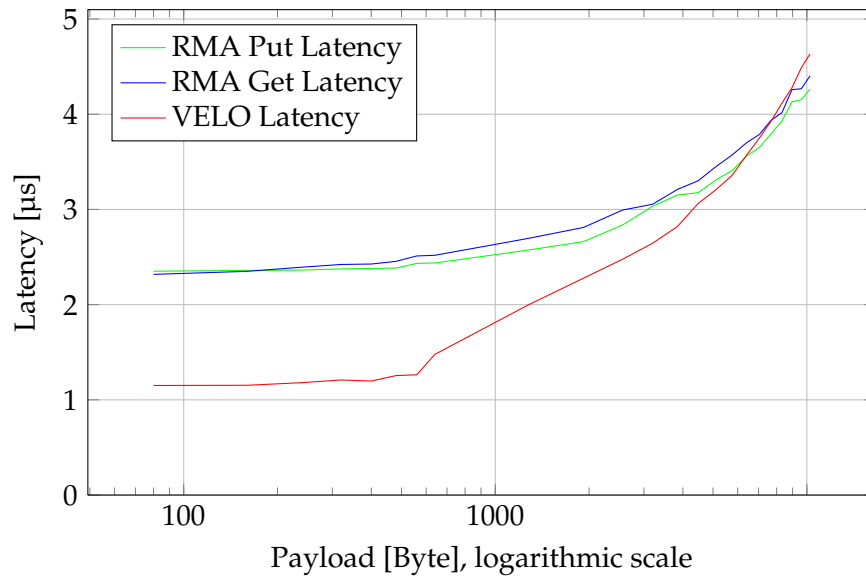


Figure 4.4: Ping-Pong Latency Results for RMA and VELO in EXTOLL R1 [94]

interference by directly forwarding HT or PCIe transactions over the EXTOLL network [121]. It therefore offers an efficient HW support for the Partitioned Global Address Space (PGAS) programming paradigm [122]. It transparently forwards packets from one node to another. Applications can thus access remote memory in the exact same way as local memory.

4.1.7 LP

The LP ensures an error free transmission of packets through the EXTOLL network [98]. Every packet is protected using a strong CRC before it is sent on the physical connection between two systems. Upon reception, the integrity of each packet is checked and in case an error is detected it is marked as erroneous. Additionally a retransmission is triggered on the link level and the erroneous packet and all following are retransmitted, ensuring the packet order and integrity with the lowest possible latency. This is done fully transparent to all other modules inside EXTOLL. The link level retransmission is faster and results in lower latencies compared to an end-to-end correction approach, due to the fact that errors are already detected at the physical medium and packets are immediately corrected at the earliest stage, avoiding a complete round-trip.

Figure 4.5 on the facing page again depicts the block diagram of EXTOLL. All modules that have been introduced so far, where developed by other team members, as such a large design cannot be implemented by a single person, but requires a team effort These units

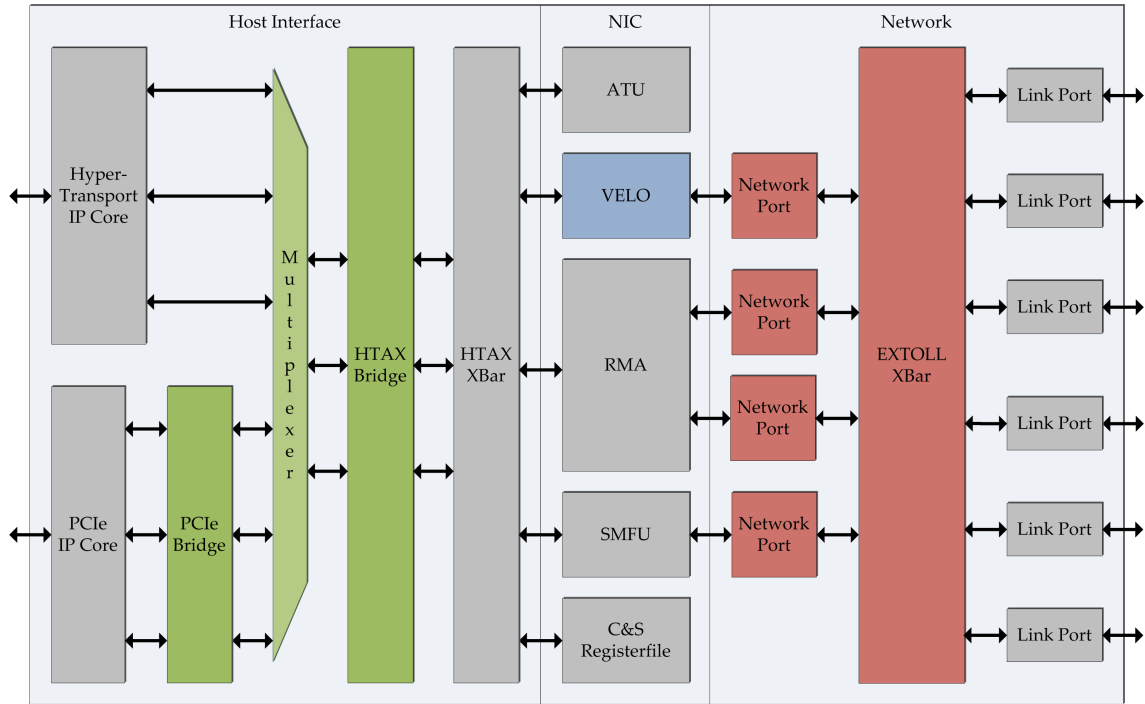


Figure 4.5: EXTOLL R2 Block Diagram with highlighted Modules

are marked gray. The other modules were developed during the course of this thesis and are now described in depth in the following sections.

4.2 Crossbar

The crossbar forms the backbone of the EXTOLL network. It is introduced in depth in section 3.8 on page 93.

4.3 HTAX Bridge

The *HT-Core* as well as *PCIe-Cores* offer only a single, FIFO-based interface to connect a single module. These FIFOs are required to handle the clock domain crossing, as these cores can operate on different clock speeds depending on the host CPU and the *bios* implementation. However, EXTOLL requires the connection of multiple units to these interfaces, which is why the HTAX has been integrated. The interfaces on the host side and the HTAX are incompatible. Hence a bridging module is required that adapts the FIFO-

based interface of the host side to the HTAX interface. The following sections analyze the required functionality and describe the implementation of this bridging module.

4.3.1 Design Space Analysis

To avoid HOL and deadlocks, both phenomena are described in section 3.4 on page 70, the *HT-Core* offers not one, but three independent FIFO interfaces, one for each VC that is defined in the HT specification for non-coherent devices. These three VCs are: *Posted*, for all write commands that do not require a completion notification, *nonposted*, for read requests and write commands that do require a completion and *response*, for read responses and write completion notifications.

EXTOLL includes the on-chip crossbar HTAX with its protocol HToC, as already discussed in section 4.1.2 on page 137, to not only enable multiple FUs to communicate with each other, but also to connect them to the host interface. In addition to the protocol conversion from HT to HToC, some sort of routing is required for packets from the host side, to be able to address the correct FU inside EXTOLL. Finally a *source tag* administration is required.

Source tags are used in both HT and PCIe to uniquely match packets from the *response* channel to the originating request from the *nonposted* channel. For that purpose, HT offers 32 tags. It has to be ensured that each tag is only used once at a time, to be able to do the matching uniquely upon reception of responses.

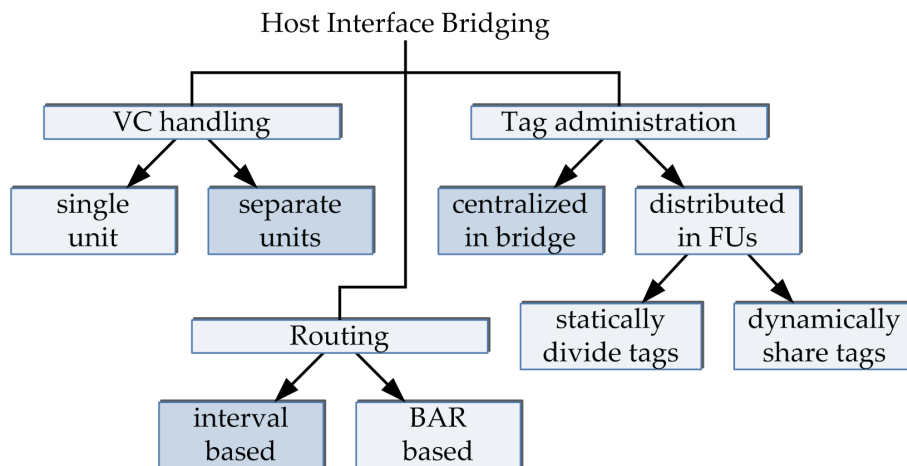


Figure 4.6: HTAX Bridge Design Space Diagram

Figure 4.6 summarizes the different aspects of design decisions and requirements that

have to be made, when designing a bridging module from the HT packet format to the HToC format. The first requirement is to handle the VCs properly, so that no deadlocks can occur. This can be done either by a single unit that is able to handle multiple streams independently without blocking other packets once one VC is blocked, or by using multiple independent units, one for each VC. The second option is deadlock free by design, as each unit is connected to a single FIFO interface. Since the HTAX specification only allows a single outputport to be requested at a time, it is not possible to implement a bridging module that connects all three FIFO interfaces with a single HTAX port. The specification only allows to request multiple VCs to be requested to the same outputport, but not multiple outputports at the same time. This has been done to avoid a more complex and time consuming request-grant-acknowledge scheme to be used in the on-chip crossbar in favor for a more simple and faster request-grant scheme. A scenario can easily be generated where a FU is waiting for a packet which cannot be delivered due to another packet from a different VC that blocks the crossbar, if only one module of the bridge is responsible for all VCs. Consequently, deadlocks and HOL can only be avoided, if discrete ports for each VC are utilized.

The second task that needs to be handled is the management of the available *source tags*. Since there are multiple units connected to the host, either a centralized administration of these tags is necessary or the available tags need to be distributed among all units. The statical distribution of tags is not very efficient, as there are phases where a single unit requires a lot of tags in order to be able to achieve its full capabilities and there are phases where another unit requires more tags. This leaves only the dynamic distribution, the question where this administration resides remains. An exclusive unit that is responsible solely for tag arbitration between all FUs requires a lot of connectivity across the chip as all FUs need to be connected to it. In addition, each FU then requires an extra interface to request a tag, which adds additional latency. The most efficient way is to implement the tag administration, somewhere along the path from the FU to the *HT-Core* — in the *HTAX Bridge*. This unit is the central and single connection between the host and HTAX. Each and every packet has to traverse it anyway, making it an ideal candidate for the job. Available tags can however only be granted in a first-come-first-serve manner, as the HTAX interface does not reveal the sender of a request in advance.

It is not only beneficial to do this in the bridge, but in order to be able to route responses back to its originating unit, some sort of mapping is required anyway, as responses to *nonposted* packets do not contain any target information. They are routed using said *source tags*, making it necessary to store the information to which port a response needs to be forwarded anyway.

As just explained, responses from host to EXTOLL are routed through an unique *source tag* assigned and administered by the *HTAX Bridge*. However read- and write requests from the host side also need to be forwarded to their target FU. Hence some sort of routing is required. The only usable field within the packet header is the address. The PCIe configuration space, which is also used for HT, allows in theory the setup of up to six 32 bit address ranges or up to three 64 bit address ranges, the so called base address registers (BARs). These can be used to route packets to different targets, as they are pre-decoded by the host interface cores, if six small or any combination of small and large BARs are sufficient. In reality however, it is often not possible to set up multiple large BARs, due to an erroneous BIOS implementation of vendors. This restriction can change with every BIOS version or vendor. In order to be less dependent on the functionality of the BARs it is desired to have some alternative at hand. As one can only use the address of packets, interval routing is the only alternative that can be implemented to enable routing capabilities.

4.3.2 Implementation

The overall block diagram of the *HTAX Bridge* can be seen in figure 4.7 on the next page. To avoid deadlocks and HOL the direction towards the crossbar is split into three units, one for each VC. Each of these units is connected to one of the FIFO interfaces of the *HT-Core* and works independently.

The *ID Map* in the center of the block diagram is a combination of a RAM, used for the mapping of tags and additional information, and a FIFO, which holds all available *source tags*. This FIFO is initialized after reset with all available tags. One tag is removed every time a request occurs. A tag is added again when a response returns. Consequently, there are no tags left and no further *nonposted* packets can be processed when this FIFO is empty.

As it has been decided to use a centralized tag administration in the bridge, each FU uses and manages its own tag domain. It has to ensure that each tag of its own domain is only used once at a time when sending requests. HTToC *nonposted* packet therefore do not only contain a *source tag* field, but also a field with an ID for the originating unit, making it possible to separate the domains.

A mapping is done in the bridge for each *nonposted* packet traversing the unit from HTAX to the host. The tag assigned by the FU is replaced with a *source tag* from the host domain and the originating FU as well as the original tag are stored in a look-up-table. The originating unit is required to be able to route the response upon arrival, the tag on

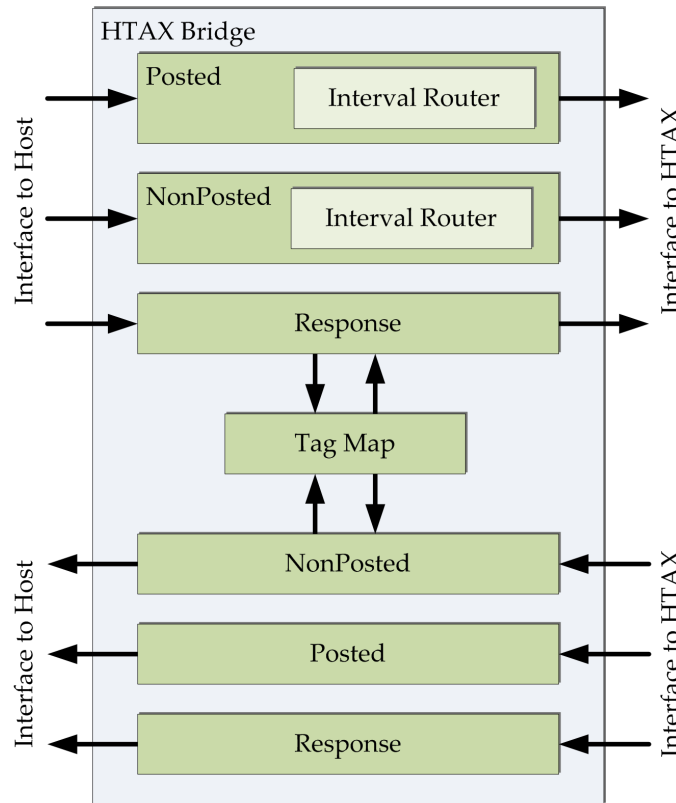


Figure 4.7: HTAX Bridge Block Diagram

the other hand is required by the targeted unit to be able to match the response to its own requests. Once the response from the host arrives, this tag is freed again and the response can be forwarded to the correct unit. Figure 4.8 on the following page shows an example where *functional unit 0* sets a request in the *EXTOLL tag domain* to main memory with tag *B*. This request get mapped to tag *1* in the *Host tag domain* and once the response is received, mapped back to its original value *B* and forwarded to the originating unit.

Units for *posted* and *nonposted* VC in the direction towards the HTAX are equipped with an interval router to be able to target multiple ports. It compares in parallel the incoming address with upper- and lower bound registers, one set for each interval. These intervals are initialized by the management SW. Each interval can be set up to an arbitrary target outport and also supports an arbitrary interval size. Additionally, the allowed VCs for that outport can be set using a mask to avoid unsupported traffic at a FU. SW has to take care that these intervals do not overlap. The appropriate target port will be requested if an incoming address is larger than the lower- but smaller than the upper bound of an interval. Packets not matching any of the intervals are forwarded to the *register file* and tagged as erroneous. There, they are discarded and an error response is generated in case

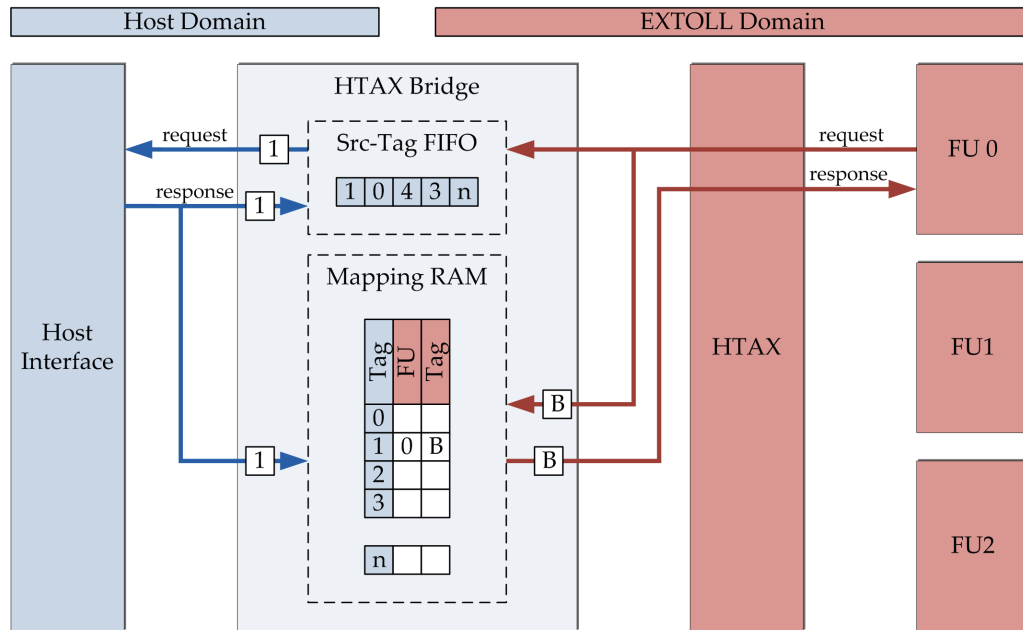


Figure 4.8: HTAX Bridge Tag Mapping Example

of *nonposted* packets. This ensures that no requests stay pending, occupying *source tags* and at some point stalling the whole system, due to run out tags. The *Register File* has been chosen for dropping and responding erroneous packets instead of the *HTAX-Bridge*, as the required logic and paths for response generation already exists there, making it more resource efficient, although these packets of course consume some bandwidth on the HTAX. This case however is considered to be occurring rarely, thus the bandwidth consumption can be neglected.

The path from HTAX to HT is also split in three independent units as it is done in the opposing direction. This is not really a requirement in this direction, as the HTAX offers the possibility to grant multiple VCs to avoid deadlocks, but since the HTAX ports are available and required for the path towards the crossbar, there is no drawback to also doing the separation in this direction. In contrast, as the *HT-Core* might run at a higher clock rate as the rest of the chip, the additional bandwidth offered through the multiple ports can be leveraged to balance the bandwidths capabilities of both clock domains.

The resulting design requires only two clock cycles to interpret packets and their route and requesting the corresponding target port at the HTAX interface, making it a true low latency design. In order to be able to send packets back-to-back, the routing interpretation and target requesting is decoupled and can thus be overlapped.

4.4 PCI Express Bridge

Using HT as host interface in EXTOLL for Intel [116] based systems is not possible, as Intel CPUs only support QPI or PCIe. PCIe lends itself as an addition for HT, as it is currently very difficult to obtain a license for QPI and the fact that PCIe is also available in AMD [114] based systems². In order to be able to use the same FUs in EXTOLL for PCIe and HT a bridging module is required, as all units use the HTtoC packet specification derived from HT. Without such a bridging module a redesign of all units connected to the host interface is required, doubling the implementation and especially the verification effort to ensure a working design also for PCIe. Consequently, the costs for an ASIC implementation increase as two independent chips have to be produced, one for HT and one for PCIe. Such an error prone, time as well as money consuming second chip is to be avoided, thus making a bridging module desirable.

PCIe and HT use different terms for the same matter. To avoid confusions only the nomenclature of HT is used in this work. Table 4.1 gives an overview of the most common terms used and their corresponding substitute in both specifications.

HT	PCIe
Packet	Transaction Layer Packet (TLP)
Read Response	Completion
Write Packet	Write Request Packet

Table 4.1: HT and corresponding PCIe Nomenclatures

Although both interfaces share some common features, they differ in others. Both interfaces use a credit based flow control and use packets to transmit data, the flow control however is completely handled in the cores, transparent for the user. Additionally both use VCs to avoid deadlocks, PCIe however only uses two, one for requests and one for responses, whereas HT has three and distinguishes not only writes from reads, but also writes or reads that require a response (*nonposted* channel) and writes that do not require a response (*posted* channel).

The second difference that has to be taken into account is the MTU. HT is capable of sending 64 byte of payload in a single packet, whereas PCIe can be configured to support a payload of up to 256 byte.

This increased payload capability can be leveraged to increase the bandwidth efficiency, as the amount of required headers can be reduced in comparison to the payload. Making

²The future of HT as a peripheral interconnect is unclear for upcoming AMD CPUs

	HT 3.1	PCIe 2.0
VCs	3	2
MTU	64 byte	up to 256 byte
Source Tags	32	up to 256
max. Bandwidth	51.2 GB/s	15 GB/s
Read Latency	147 ns	240 ns
Number of Lanes	8 or 16	1, 4, 8 or 16

Table 4.2: HT and PCIe Comparison [123]

use of this larger MTU for packets towards the host however cannot be done solely in the *PCIe Bridge*, but must already be supported in the FUs that generate the transmitted packets. In the opposite direction, from the host to EXTOLL, it is also not required to check the length as the SW driver can either make sure that no packet sizes are sent that the receiving FU cannot handle or the *PCIe-Core* can be configured to only support the same MTU as HT offers.

All packets that require a response are again tagged with a *source tag*, in order to be able to map received responses back to their originating read requests. HT offers 32 such tags whereas PCIe can again be configured to support up to 256 tags.

There are some more differences concerning the physical specification, these however are transparent to the user and therefore not explicitly mentioned. [123] gives a good overview of these differences, which are also summarized in table 4.2. Noteable is the read latency of both interfaces.

4.4.1 Requirements

The target for this implementation is not to implement a fully compatible bridging module that is capable of mapping all possible packet types in both directions, but to implement a fast and lean protocol converter that is specialized for the EXTOLL environment.

The bridge itself must be designed in a way that it is either an add-in replacement for the *HT-Core* or can be instantiated in a way that both I/O interfaces, PCIe and HT can be used with the capability to switch between these two at power-up or with a fuse upon assembly. Therefore it is beneficial to have the same interface on the host side as the *HT-Core* to ease the connection to the *HTAX Bridge*.

As it is not the target to implement a fully functional *PCIe-Core*, the bridge must be

as vendor unspecific as possible to ease the adaption to any new *PCIe Core* vendor. A mapping of *source tags*, which has until now been handled in the *HTAX Bridge*, described in section 4.3 on page 141, needs to be handled in the *PCIe Bridge* when leveraging PCIe, in order to be able to leverage the possibility of more outstanding read requests.

During the implementation of the *HTAX Bridge* the decision to use a centralized source tag management and implement separate paths for the different VCs has been made. The design space for the *PCIe Bridge* looks exactly the same, without the routing decision aspect, as this is still done in the *HTAX Bridge*. The resulting design space diagram is illustrated in figure 4.9.

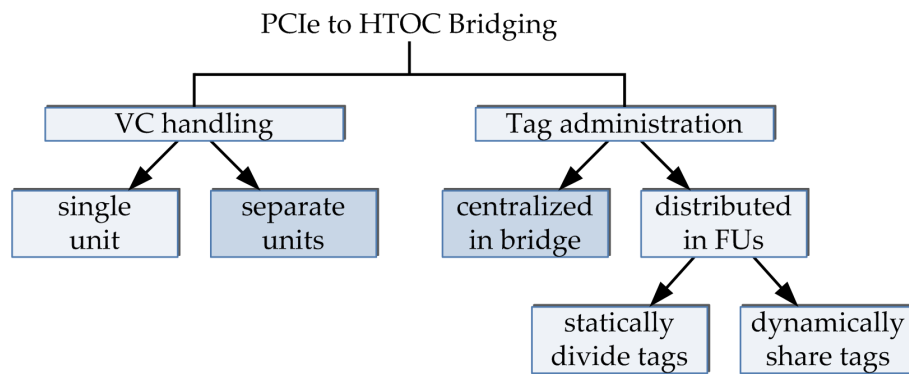


Figure 4.9: PCIe Bridge Design Space Diagram

Due to the decisions made earlier, the same design choices are also applied for the *PCIe Bridge*. Furthermore, it must be made sure that unsupported packet types are not dropped silently, but handled in a way that the sending unit (or host) remains intact and the normal operation is not disturbed.

4.4.2 Implementation

In order to be able to use one of two host interfaces in the same device a multiplexer-demultiplexer structure is located between the host interfaces and the *HTAX Bridge* as figure 4.1 on page 134 depicts, making it possible to switch between the two interfaces.

The bridge can be divided into the two directions packets can pass through and within each direction split into three independent sub-modules, one for each VC of the HT domain as depicted in figure 4.10 on page 151. The same internal layout has been chosen to leverage the high bandwidth capabilities of the *HTAX Bridge* and to have a common interface for both, PCIe and HT in the direction towards EXTOLL. The block diagram

therefore looks almost identical as the one from the *HTAX Bridge* with an additional *Vendor Layer* and a second *Tag Map*.

The *Vendor Layer* inside the bridge has been added in order to be as independent as possible from the *PCIe-Core* vendor specific interface. Only this module has to be adapted in case a different FPGA vendor with its own *PCIe-Core* or another ASIC implementation is to be chosen in future designs. This layer is also responsible for splitting up the two VCs of PCIe to the three VCs used in EXTOLL and HT. This splitting is done according to the arriving packet type and the HT VC usage. *Memory writes, atomic operations and messages* are forwarded to the *posted* channel, *memory reads, I/O reads, I/O writes, config reads* and *config writes* are all forwarded to the *nonposted* channel and *responses* are forwarded to the *response* channel.

In contrast to the *HTAX Bridge* a tag mapping is now required in both directions. In HT a mapping is only required for *nonposted* packets leaving EXTOLL in order to be able to forward the response properly to its originating HTAX port. Other than that, only the tag itself needs to be replaced as described in the example given in figure 4.8 on page 146. The exact same functionality is also required in the PCIe case. In addition, a second *Tag Map* is required for read requests arriving from the host CPU, as PCIe response packets require additional information that are suited in the corresponding read request and cannot be mapped to any of the available HToC fields. Storing these fields in the *Tag Map* makes it possible to retrieve them again once the response needs to be mapped back to PCIe.

Packets arriving at the bridge in direction towards the *PCIe Core* can be mapped without any error handling requirements, as all packets coming from any EXTOLL FU are known and well defined and can therefore be mapped to a corresponding PCIe packet. These Packets only differ in the header encoding itself and the already mentioned *source tag*. The increased amount of tags can be leveraged to effectively increase the amount of pending read request by doing the mapping of the *source tags* in the *PCIe Bridge* instead in the *HTAX Bridge* in the PCIe case.

The opposite direction, from PCIe to HToC however has more complexity as not only packets that can be generated by EXTOLL have to be taken into account, but all specified packet types need to be handled somehow. PCIe has multiple different packet types; *PCIe messages, memory write- and read requests, I/O write- and read requests, configuration write- and read requests* and *atomic operations*. Whereas *memory-, I/O and configuration* request packets can be uniquely mapped to apposite HToC packets, it is more difficult to map *messages* and *atomic operations*. *Messages* can contain anything, as this is a container format for proprietary packets. All arriving *messages* are discarded or forwarded to a special

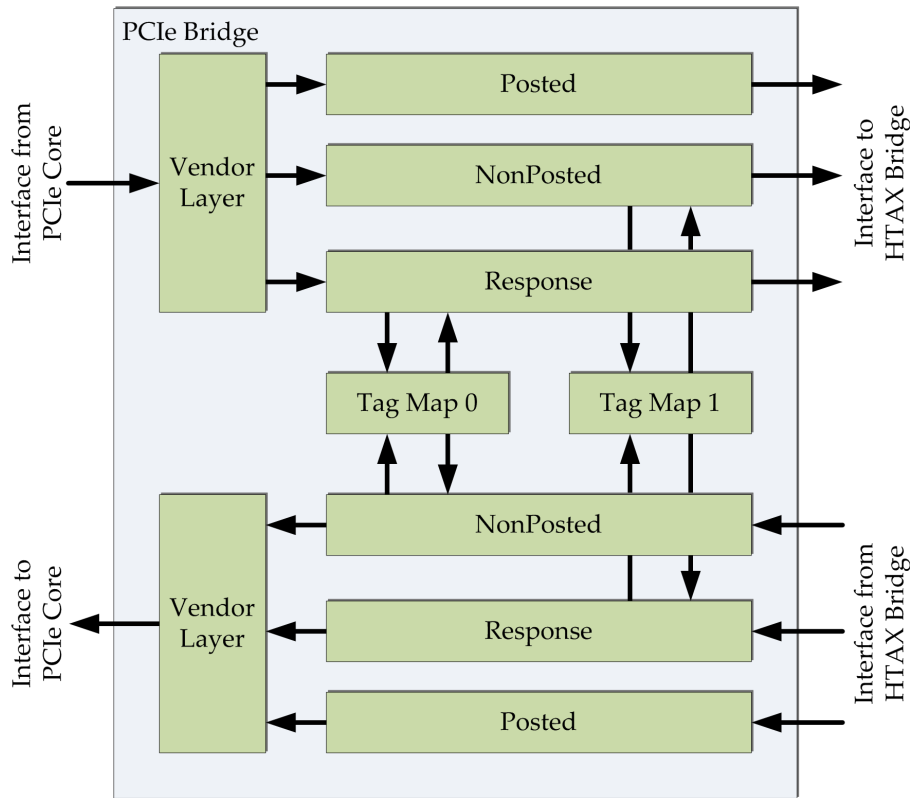


Figure 4.10: PCIe Bridge Block Diagram

debugging interface, as EXTOLL cannot decode such packets and the HToC specification does not support proprietary packets. The same applies for *atomic operations*, they are also either discarded or extracted from the packet stream to the debugging interface. Although HT supports atomic operations, not all atomics can be mapped uniquely, as there are more operations specified in PCIe as in HT. In addition, EXTOLL currently does not support atomics, so forwarding them makes no sense for the time being.

The above described implementation results in an efficient bridging module with a very low latency of only a few cycles. The exact amount of cycles depends on the *vendor layer*, the translation itself requires only three clock cycles.

4.5 Network Port

The NP is the entry and exit point to and from the network part of EXTOLL. Hence, its main purpose is to enable communication between the network crossbar and the FUs. This communication can be seen as two independent data streams, which is best implemented

also using two separate units as shown in the block diagram, illustrated in figure 4.11. The *Network Port Sender* handles all communication from the FU to the crossbar. The *Network Port Receiver* handles all traffic from the crossbar to the FU. As already mentioned in section 3.8.3 on page 101, the target FU of a packet is encoded in the header and is addressable through the network only if packets reached their target node. Consequently, each FU has an exclusive connection to a single NP.

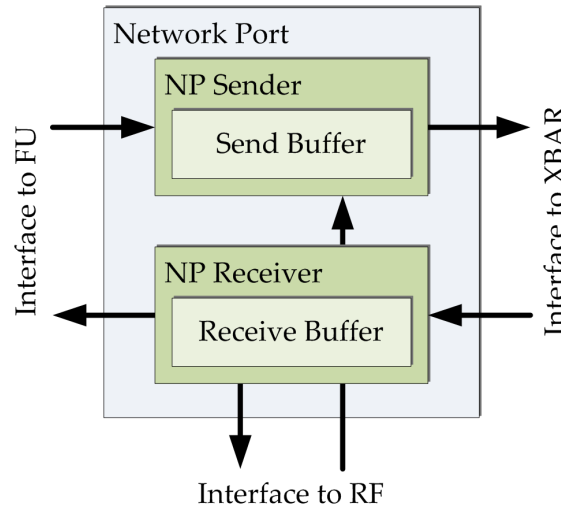


Figure 4.11: NP Block Diagram

FUs in EXTOLL are part of the NIC, which is separated from the actual network. Consequently these units are not aware of the credit based flow control, the concept of FCCs and the packet framing used in the network. The concept of FCCs is described in section 3.7 on page 82. The NP ensures that flow control and framing are maintained and assigns all packets to an outgoing FCC.

The *Network Port Sender* ensures that enough credits are available before injecting packets into the network and frames all data according to the EXTOLL network specification. In order to be able to do that it has to check the SOP cell, that has already been added by the FU³, for a correct encoding and pads the payload of the packet with an EOP cell where the packet CRC is suited. This CRC will be added by the LP at a later stage. A small send buffer is used to store outgoing packets in case no credits are available and to buffer data while streaming to compensate the required additional cycle for EOP padding.

The *Network Port Receiver* retrieves packets from the network. It is equipped with a buffer, as it has to be able to accept as many packets as credits are available. Framing information that are not required anymore are removed during packet reception and the

³the SOP cell includes the target node and VPID as well as the TC of the packet and has therefore to be added by the FU

interface is adapted to the requirements of the connected FU. Credits are freed again after the FU has read the part of a packet that occupied a single credit to ensure a seamless packet stream. This way, always the maximum amount of credits are available in the network.

As it is ensured by the specification that packets arrive en bloc from the network, a *cut-through* approach is used in the *Network Port Receiver*, enabling the lowest possible latency for packet reception. However, as packets can be corrupted during transmission, packet errors are signaled, but forwarded to the FU and have to be resolved there. If a FU is not capable of handling such errors, the NP also offers a *store-and-forward* mode. In this mode packets are buffered until it is known whether a transmission error occurred or not. Once the faultless reception of the packet has been detected, it is forwarded to the FU. A FIFO with speculative shift-in capabilities, as it has been introduced in section 2.2.2 on page 17, is leveraged to realize this feature. Once an erroneous packet has been detected, the write pointer of the FIFO is set back to the value it had before the packet started. Enabling *store-and-forward* of course adds additional latency to the packet transmission, but simplifies the logic of FUs, as no error handling has to be implemented there. As a result, the traffic on the host-interface can be reduced, as only validated packets are written to main memory, freeing a valuable resource.

Finally a point to inject and extract packets from the network for debugging purposes is preferred. This enables the SW engineer to inject packets byte by byte into the network, making it possible to, for example, force error conditions and check the resulting behavior in a real system instead of only in simulation. The extraction of packets can be helpful in case the connected FU is for some reason blocked and consequently no progress is made. Packets can then safely be extracted without a loss of packets, ensuring integrity of all messages. For this purpose, the NP offers an interface to the *register file*, making it possible to shift-in and -out single quadwords. As every shift-in requires a RF-write, shift-outs a RF-read respectively, this interface does not offer a high-performance. But as it is only used in faulty states or for debugging purposes, it forms no performance bottleneck for everyday workloads.

4.6 VELO

VELO is one of the three key FUs enabling communication in EXTOLL. First an introduction of the unit itself and its status is given, followed by a section describing the redesign towards the R2 implementation.

4.6.1 Introduction

VELO has specifically been designed to enable the transmission of small messages with a size of up to one cache line, i.e. 64 byte, with the lowest possible latency [124], including not only the transmission through the network, but also taking the SW overhead into account. Therefore it combines the advantages of PIO and DMA data transmission. A sender writes the command and data in-line to VELO using PIO in a single write access to the device, which has less overhead compared to DMA upon message start-up. The reception of messages however is handled using a circular buffer in main memory, called mailbox, where applications can use polling to check for new messages. Polling blocks the host CPU as it continuously reads a memory address, but for low latency applications there is no mechanism to receive data faster. Traditional message reception involves the OS and uses interrupts, which requires at least two context switches in the host CPU, thus adding latency.

Meanwhile, the unit is stateless and fully virtualized, allowing secure user-level communication without kernel involvement for multiple threads. The upper bound of 64 byte has been chosen as it is the largest data chunk that can be written atomically using PIO.

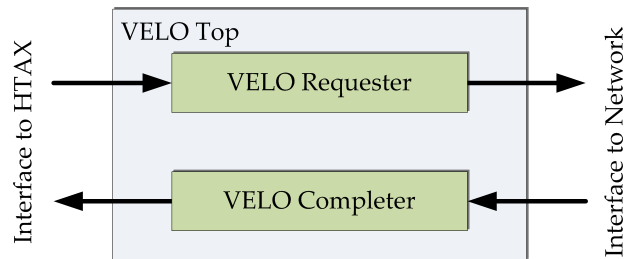


Figure 4.12: VELO R1 Block Diagram

The two tasks of message transmission and reception are completely independent of each other. VELO is therefore split in two units, as it is depicted in figure 4.12. Both units share a single NP- and HTAX port respectively. However, they require only the receive- or send- part of each port, as each part of the unit itself is only unidirectional. In addition, both units are single stage based and are thus only able to sequentially process packets.

The *VELO Requester* is responsible for receiving messages from the host and injecting these into the network. Messages are received over the HTAX, prepend with routing information and a status word before they are forwarded to the NP.

The status word encodes all information required to be able to write the message in the main memory on the target side. Figure 4.13 on the next page illustrates the layout of this

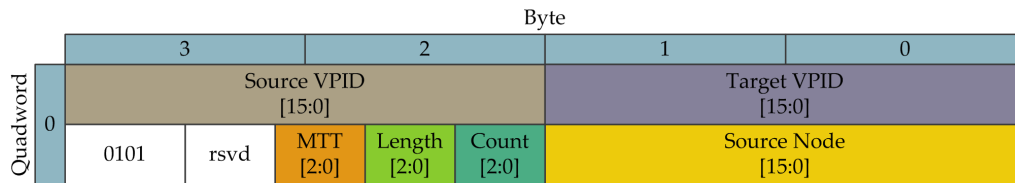


Figure 4.13: VELO R1 Status Word

status word. This word is formed using information provided in the address of incoming HToC packets, which is illustrated in figure 4.14. The *source node ID* information is a static information provided by the RF. The function of *target*-, *source VPID* as well as *source node* fields are quite obvious. *Length* field gives the amount of quadwords of the complete message. *Count* on the other hand is used by the receiving SW to detect whether the current message is part of a split message or not, the problem of split messages has been laid out in section 2.3 on page 22. The *Message Type Tag (MTT)* can be used by applications to distinguish different types of messages, this field is just passed through the unit without triggering any functionality. The *target node* field in the address is used to look up the routing string that is added to the network packet required to reach this node.

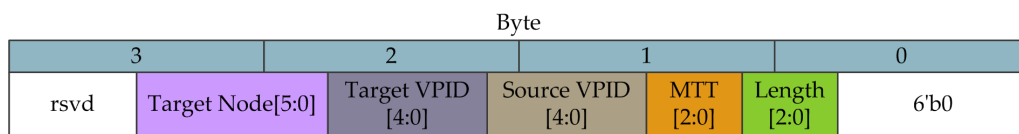


Figure 4.14: VELO R1 Address Encoding

The *VELO Completer* on the other side receives messages from the network and writes them into the hosts main memory. All necessary steps from receiving the network packet to writing into the main memory circular buffer are processed sequentially. There is a small segment of continuously allocated and pinned main memory for each process which can be mapped in the user address space, so that no kernel-level communication is necessary for data reception. This memory segment is used as a circular buffer with a fixed slot size of one cache line, which is 64 byte. The first quadword written into this slot is always the above mentioned status word, therefore only 56 byte can be used for actual payload in each slot. This way however, the receiving SW always knows whether the next slot contains valid data or not. Once it has read the slot completely, at least the status word needs to be reset to zero in order not to read it as a valid message upon the next iteration of the circular buffer. A second slot is used for the remaining bytes in case a message is larger than 56 byte. In this case the status word is repeated in the next slot.

An example of the slot usage is given in figure 4.15 on the following page. As it can be

0B	Status Word 0
	Data 0-0
	Data 0-1
	Data 0-2
	Data 0-3
	Data 0-4
	Data 0-5
	Data 0-6
64B	Status Word 0
	Data 0-7
	Invalid data
	Invalid data
	Invalid data
	Invalid data
	Invalid data
	Invalid data
128B	Status Word 1
	Data 1-0
	Data 1-1
	Data 1-2
	Invalid data
	Invalid data
	Invalid data
	Invalid data
192B	Status Word 2
	Data 2-0
	Data 2-1
	Data 2-2
	Data 2-3
	Invalid data
	Invalid data
	Invalid data

Figure 4.15: VELO R1 Ring Buffer Usage

seen the first message spans over two slots, whereas the second and third message fit into a single slot.

4.6.2 VELO Evaluation

The VELO R1 implementation works and performs well as shown in [124]. Nevertheless, there are a number of shortcomings that can be improved to achieve a higher packet throughput and increase the overall usability. The following sections explain the shortcomings of the R1 implementation, followed by an description on how these restrictions have been resolved.

First of all, VELO depends on a certain behavior of the write combining buffer. Larger messages can be split into multiple smaller parts by the host CPU. Current Intel CPUs generally never combine writes unless they fill a complete cache line. AMD CPUs may also split combining buffers. Both leads to, more or less serious problems, as introduced in section 2.3 on page 22.

The current implementation treats these split messages as independent messages on

the *VELO Requester* side and resolves this issue in SW upon reception, thus reducing bandwidth efficiency of both, the network and the host interface by adding more overhead and increasing the SW overhead. In addition, this behavior is likely to change with new CPUs generations or host interconnects and can thus become a problem.

While it is acceptable that the size of messages is limited, a payload that is larger than 64 bytes can be extremely useful. For example, 64 byte are not enough to carry a maximum sized *Active Message* for GASnet [125]. 64 byte are very small if a MPI header including some payload has to be transported. A maximum size of two to 16 cache lines can be considered useful, which results in a message size of 128 to 1024 byte.

All size and destination information required for a packet are encoded via address mapping due to the possibility of split packets. This causes a few problems, the first problem is in relation to the usage of address space itself. A large address space is required for this encoding, since every destination node, source VPID and target VPID combination requires its own page to ensure that VPIDs can only reach permitted destinations. This may be a problem if a platform firmware or HW does not support large address mappings, which is very probably the case with commercial off the shelf bridges and firmwares. Obviously supporting more than 64 nodes or 32 sending VPIDs increases the required address space as it can be seen in figure 4.14 on page 155. All required bits cannot be mapped into the available address bits once any of the fields grows to large. The encoding space in the *status word* on the other hand is large enough to cover a large number of target nodes and processes.

Secondly, certain CPUs may not even have a large enough physical address space to scale to large networks, i.e. current AMD based systems only provide a 48 bit address space, whereas the address space of Intel based systems is even smaller, only providing 32 bit.

The large address space requirements also have an effect in relation to TLB misses. Every transmission of a messages causes a virtual to physical address translation to occur in the host CPU. If every destination has its own page, sending to a number of different destination causes the same number of translations. This may either be slow, i.e. a TLB miss, or waste TLB entries which are also direly needed for actual computation. A design that works with only a few pages per source VPID is therefore beneficial.

In addition, it is important to be able to set the TC when sending a message in order to be able to separate different message flows for I/O, inter process communication (IPC),

et cetera, avoiding deadlocks through the network on the application level. Thus, the allowed TCs for each VPID must be selectable by supervisor software only.

On the *VELO Completer* side tests have shown that the unit is not able to send HToC packets utilizing all of the available bandwidth due to its sequential design. Hence, this part forms the bottleneck when using VELO as communication engine. By pipelining the design more bandwidth can be achieved.

The current implementation also lacks an efficient support of interrupts, which enable better asynchronous protocols. Additionally it is necessary to support large enough receive buffers. In the current implementation only a single continuous main memory area is used as reception mailbox, limiting its size tremendously.

Another problem is that messages that are blocked at the *VELO Completer* due to a full mailbox are currently dropped after a timeout without any notification mechanism. Dropping of packets however requires some sort of notification generation to inform the supervisor SW.

All these considerations lead to a redesign of the *VELO Requester* and *-Completer*.

4.6.3 VELO Requester R2

The following shortcomings have been identified in section 4.6.2 on page 156 on the *VELO Requester* side. First the possibility of split messages, second the lack of support for message sizes larger than 64 byte, third the required large address space and fourth the lack of security. While the *VELO Requester* is a rather simple unit and is already capable of sending network packets at full line rate in R1, most of these shortcomings address the usability of VELO. The new R2 design is targeted to resolve these issues.

The introduction of a *Write Combining Buffer* within the device is used to remedy most of the problems at the cost of an additional RAM buffer and some additional cycles of latency. The main function of this buffer is to collect split messages, store them until all parts have been received and only then forward them to the *VELO Requester*. This unit is described in more detail in section 2.3 on page 22.

When combining host packets in front of VELO the amount of network packets that are required to send through the network can be reduced. Each split part of a host packet is transmitted as a single network packet in R1 of VELO. Thus, removing all split parts

from the network and only transmit the packet as a whole makes combining of packets on the receiver side obsolete, making the reception SW simpler and therefore faster, thus reducing latency as well as the memory footprint.

Adding the *Write Combining Buffer* also allows a flow control to be added, which ensures that a host CPU cannot be stalled due to back-pressure of HToC packets, which as well has been an issue in R1 of EXTOLL. In R1 it has been possible that a read pointer update cannot not be completed, which is necessary to be able to make progress due to other outstanding messages and therefore causing a deadlock.

In addition, the buffer also reduces the required address space considerably. Since it can now be ensured that a message arrives as a whole at the *VELO Requester*, it is no longer necessary to encode all information in the address of the HToC packet, which is transmitted with every part of the message. Only the source VPID and the TC need to be encoded in the address for security reasons, as only this way it can be ensured that the transmitting VPID has the privilege to actually use VELO. Instead a status word, as it is also transmitted through the network, can be used on chip, thus reducing the required address space considerably. It is now no longer necessary to reserve a page for every target node and encode the size of packets inside the address.

Figure 4.16 shows the new address encoding. As it can be seen, the amount of fields is reduced to two and the required address space is smaller nonetheless more VPIDs are now supported. This solution therefore solves the address space pollution as well as the TLB problem, as now only one page per process is used compared to the the requirement of one page per process and target node in R1.

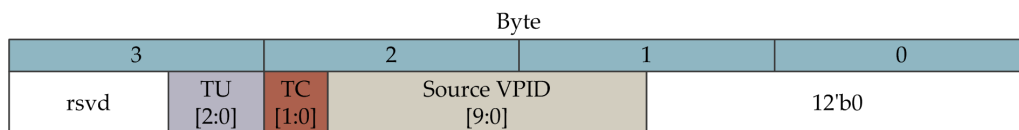


Figure 4.16: VELO R2 Address Encoding

In contrast to the status word used in R1, this new word has wider fields and offers more functionality. It is illustrated in figure 4.17 on the following page. The *target node* field is now 16 bit, thus matching the capabilities of the EXTOLL network. Additionally this field can be leveraged to encode not the target node, but also a multicast group, enabling VELO to make use of an EXTOLL crossbar feature, described in section 3.8.4 on page 102. A single bit is sufficient to distinguish between unicast and multicast addressing. The *target node* field is interpreted as multicast ID if this bit is set. The width of the *target VPID* has also been increased to ten bit supporting up to 1024 threads on each host making VELO

fit for the foreseeable future. The *Message Type Tag* field has been reduced to two bits, but an additional *User Tag* is introduced, which more than compensates this restriction. Interrupts can now be requested on the *VELO Completer* side with an additional interrupt bit. An interrupt will be generated once the whole message has been written into main memory if this bit is set.

Finally, a single field is now used for the length of the message. The value of this field no longer counts the quadwords of the message, instead a byte count is used and the size is increased to support messages with a length of up to 2048 byte. Counting bytes has the advantage that arbitrary messages sizes are possible simplifying the reception SW.

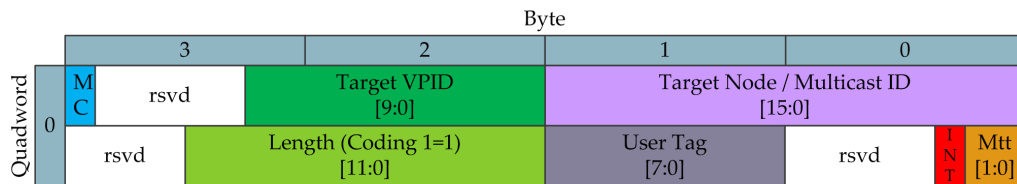


Figure 4.17: VELO R2 Status Word as the Requester receives it from SW

Due to the move of the *target node ID* field from the address to the status word it is now no longer possible for system SW to guarantee safety and security in regard to the destination of a message, as this is now completely handled in the user-level. Therefore some sort of mechanism has to be put in place to ensure that only privileged processes are actually writing data to a main memory region through VELO.

A table in the RF with one entry per VPID contains a Protection Domain Identifier (PDID), which only supervisor SW can change. This PDID is transmitted only through the network and does not leave the chip to the host side, ensuring that the user-level has no chance to obtain or change it in the network packet. The *VELO Completer* checks the transmitted PDID with its own local table and forwards the message to main memory only if the source VPID is member of the same *Protection Domain* as the target VPID.

This PDID is added to the status word in the *VELO Requester*, therefore this status word used in the network is different from the one the sending application has issued. The status word as it is transmitted on the wire is illustrated in figure 4.18 on the facing page. The PDID has been chosen to be 16 bit wide, to ensure a certain safety and security against outside attacks. Its width is a trade-off between addressing space in the status word and security, because choosing it to large requires reducing other functionalities or increasing the status word, thus increasing overhead.

As it can be seen, the *length* field is now one bit smaller as it is in the status word sent

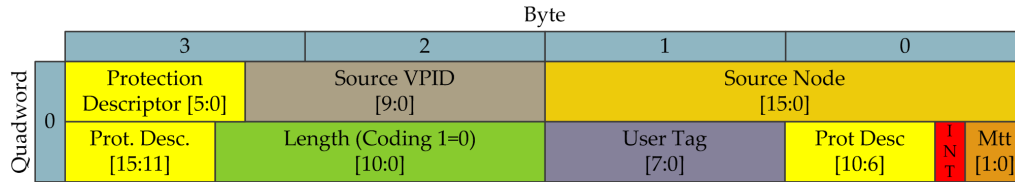


Figure 4.18: VELO R2 Status Word as it is Transmitted Inside the Network

from the application, illustrated in figure 4.17 on the preceding page. This is achieved without a loss of information by using a different encoding for the length. The value 0 now means that the message is actually one byte long. Due to this recoding it is not possible to send messages with zero length, but this restriction is not of importance as a message of zero length only contains the status word without any usable data and will never be actually transmitted. This recoding is done as there is otherwise not enough bit space available to encode all fields. Both transformations of this value are done in HW to avoid the need for the application to do the subtraction upon transmission and addition upon reception of each message. In HW the clock cycles required for this recoding can be completely hidden, whereas in SW these cycles are always in the critical path.

4.6.4 VELO Completer R2

In order to improve the bandwidth utilization, it is necessary to split the *VELO Completer* into multiple units, i.e. pipelining it, so that the processing of the different sub-tasks can overlap each other.

As the two main tasks of the *VELO Completer* is to first fetch and check incoming network packets and then write them into main memory, leveraging the HTAX, it renders obvious to realize these two functions in separate units. To ease the flow control and be able to let both units process messages independently, it is beneficial to decouple these two units using FIFO buffers, as the HTAX can sometimes stall VELO due to other transmitting units or an otherwise blocked host interface.

The resulting block diagram is illustrated in figure 4.19 on the following page. The first unit, called *Fetch Unit*, is responsible for reading the data from the NP, checking whether the content of the network packet is a valid VELO message and storing addresses to where the message has to be written to. There are multiple checks a message has to pass in order to be forwarded to the next unit. The first one is that, in case of an unicast, the *target node ID* field must match the own host ID. In case of multicasts this check is not possible, since the same field that is used for the *target node ID* is also used to encode the *multicast ID*. The

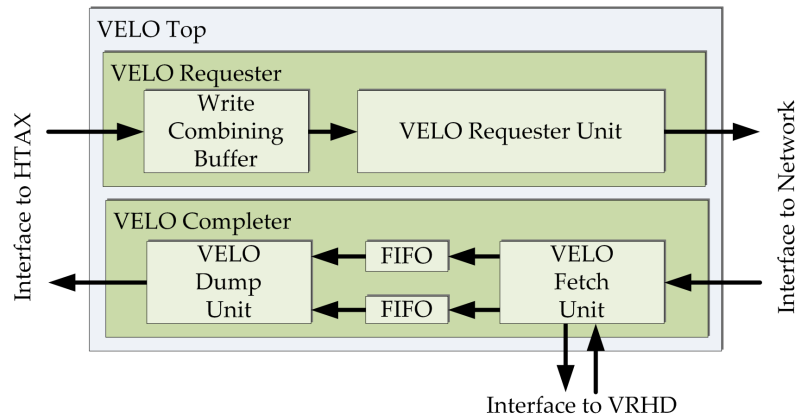


Figure 4.19: VELO R2 Block Diagram

next check is that the *target unit* field must also match the ID of VELO. As a third check, the *protection descriptor* introduced in section 4.6.3 on page 158 needs to match the value set in the local table. The final check is that the targeted VPID needs to be enabled, which means it is initialized and ready to receive data. If one of those checks fail, the message is discarded and an interrupt is generated, informing the supervisor SW that an error occurred.

One of the desired optimizations is to be able to have a larger receive buffer per VPID. The spliced buffer implementation presented in section 2.4 on page 24 can be leveraged to achieve this. *VELO Completer* requests a new addresses by using a dedicated interface to the VHRD. The first address is speculatively requested as soon as a new packet is available in the NP receive buffer to hide latency, since the VHRD requires a few cycles to respond with a new address. These cycles are used to perform the above mentioned checks. The request can then be revoked in case one of the above mentioned checks fails, otherwise the request is committed and the next request is set as soon as one slot in the mailbox is filled and the current message requires another slot. Additionally to the larger mailbox support and exploit of the VHRD, VELO is now able to use not only slots with a size of 64 byte byte, but also 128 byte, leveraging the larger MTU of PCIe. However this cannot be changed dynamically from message to message, but can only be enabled upon initialization, as not only VELO, but also the VHRD and the reception SW needs to be properly initialized.

Valid addresses are stored in a separate FIFO next to the data FIFO, as it is illustrated in figure 4.19. A timeout option can be enabled if in any case no valid address is returned for a period of time, again dropping the packet and informing supervisor SW.

The next unit, called *Dump Unit*, starts generating HToC packets as soon as the first address is available in the address FIFO. Due to the latency of the VHRD it can be ensured

that enough data is present in the data FIFO once an address is present in the address FIFO.

The interrupt bit in the status word on the wire tells the *Dump Unit* whether to generate an interrupt or not, once it finished sending a message to main memory. Additionally a local bitmap disables the possibility to generate an interrupt on a VPID basis.

Due to the support for larger message sizes, messages can now span not only over two, but over several slots in a queue. To increase the performance of message reception, this part has also to be revised. Figure 4.20 illustrates the design space concerning the main memory circular buffer usage. As already described, the previous implementation repeated the status word, which is also the indicator of the arrival of new data in each slot, thus wasting one eighth of the available buffer space and bandwidth, but ensuring data integrity under all conditions.

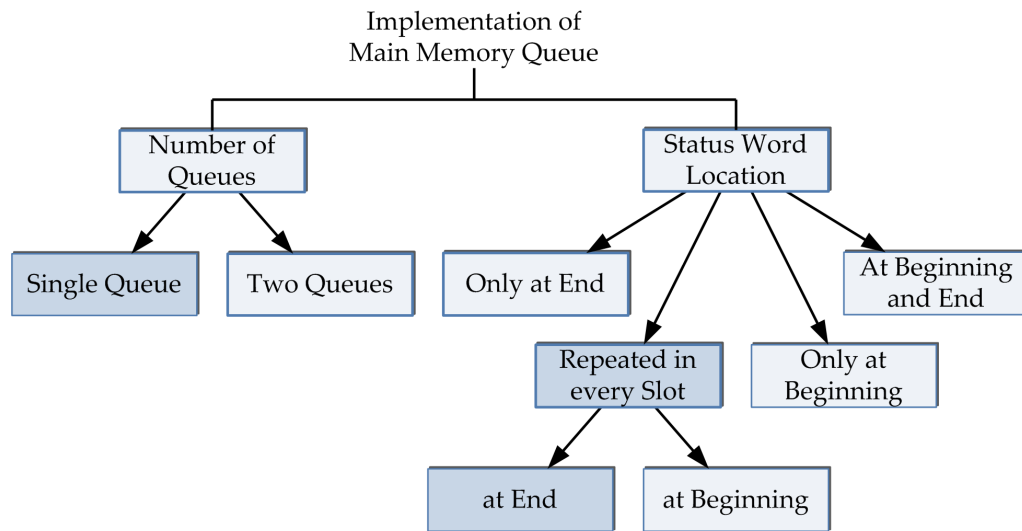


Figure 4.20: Main Memory Queue Design Space Diagram

The first alternative approach is to use two queues per receiving VPID, one possibly larger queue for the payload of messages, which is 64 byte aligned, and one for the status words, which is only 64 bit aligned, the exact size of a single status word. The advantages of this approach are that there is no need to do a realignment of the data, as all data is written continuously to the data queue and that the SW only needs to invalidate the status queue. Disadvantages however are that the receiving SW always has to read at least two cache lines, which increases the latency for small messages. In addition, the HW has to monitor not only one, but two queues for their full status. Due to the two queues, read pointer updates are now required for both queues as well, doubling the RAM usage for pointer structures and doubling the amount of required RF accesses.

The above mentioned approach can be improved concerning the latency for small messages by also aligning the status queue to 64 byte and writing messages with a payload smaller than 56 byte in-line with the status word to the status queue. This reduces the latency of the first approach and still keeping its advantages. However it doubles the main memory usage in respect to the original approach, still has all disadvantages of the first approach and adds even more complexity to the HW, as it now has to decide where to store messages. In addition, there is no clean separation of the purpose for the two queues.

Using only one queue as it is already done in R1 and writing the status word in-line keeps the memory footprint lower compared to the previously mentioned methods. The buffer space usage can be improved by writing the status word only in the first slot and writing an end marker or repeating the status word only in the last slot of the message. However this requires to read the first slot, interpret the size of the message, then read the last slot until the end marker is seen and only then can the rest of the message be read, otherwise it cannot be guaranteed that all data has already been written properly. This jumping around in the memory thrashes the cache, making an efficient use impossible and requiring a lot of memory reads.

Another possibility is to keep the scheme of repeating the status word in every slot of the ring buffer, thus reusing a well known and proven to be working approach. However, the status word can be moved from the first to the last word of the slot. This has the advantage that only this way it can be ensured that the slot has been completely written before SW starts reading it, in case an intermediate bridging device or the host CPU splits the packets sent from VELO to the main memory. This can be the case when using PCIe as host interface. Additionally, erroneous packets can now be tagged easily, which has not been possible in R1, since the error indicator in a network packet resides in the very last phit, in the EOP cell.

To ensure the minimum amount and size of packets on the host interface, smaller messages than the slot size minus the status word are written in a way that they do not start at the beginning of the slot, but at a point that makes it possible to write the data and the status word in a single HToC packet without having to send a maximum sized packet just to align the status word to the end of the slot.

Due to its simplicity and the fact that it proved to be working, this last option has been chosen, keeping a well working scheme, nevertheless improving it considerably. Figure 4.21 on the next page shows the slot usage of the optimized *VELO Completer*. The figure also shows that it is now possible to write into slots with a size of 64 or 128 byte

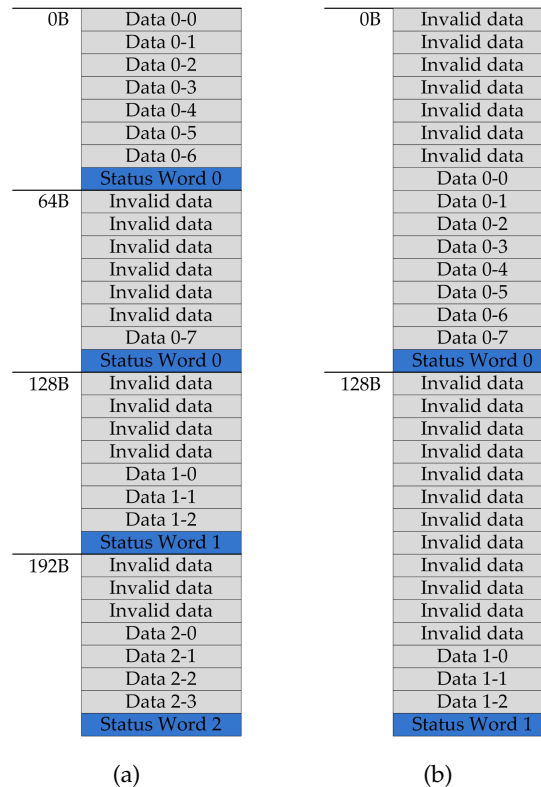


Figure 4.21: VELO Memory Slot Usage for 64 (a) and 128 Byte (b) Slots

leveraging the possibility of larger main memory writes when using PCIe. 64 byte slots are advantageous when writing a lot of small messages due to the fact that the buffer is then used more efficiently, as there are not too many unused quadwords between two messages. However, 128 byte slots are advantageous for larger messages as the need to repeat the status word can be reduced.

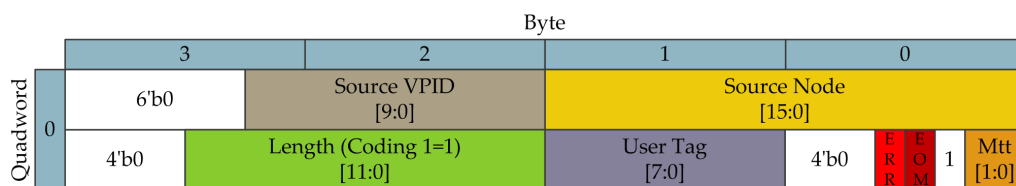


Figure 4.22: VELO R2 Status Word as the Completer writes it into Main Memory

Figure 4.22 shows the status word as it is written into main memory. It is again different from the other two words used on the wire or sent to the *VELO Requester*. *Target node* and *target VPID* fields have been removed, as they are obvious to the receiving SW and therefore redundant. Instead, the according sources have been added to enable the receiving SW identifying its communication partner without sending additional

information in the payload section of messages. The *protection descriptor* has also been removed, to ensure that malicious SW cannot use it to inject unapproved traffic. *Length*, *MTT* and *user tag* are the same as in the *VELO Requester* status word shown in figure 4.17 on page 160. The field *EOM* is used to indicate that the current slot is the last part of a message, *EOR* indicates an error during the transmission through the network. Messages tagged with *EOR* can be ignored as the link level retransmission of the LP ensures that the correct version of this message is received in the next slot. Note that the *1* next the *MTT* is used to indicate a valid status word, as it is possible that all fields inside the status word have a value of zero.

4.6.5 VELO Performance

The performance of VELO can best be measured using micro-benchmarking. For these tests the FPGA testbed shown in figure 3.50 on page 125 has been utilized. The design runs at 200 MHz in the NIC and network clock domain and at 300 MHz in the host interface clock domain. The used hosts are a set of Hewlett-Packard DL-165 machines, each equipped with two quad-core 2380 Opterons, running at 2.5 GHz, and 8 GB of main memory.

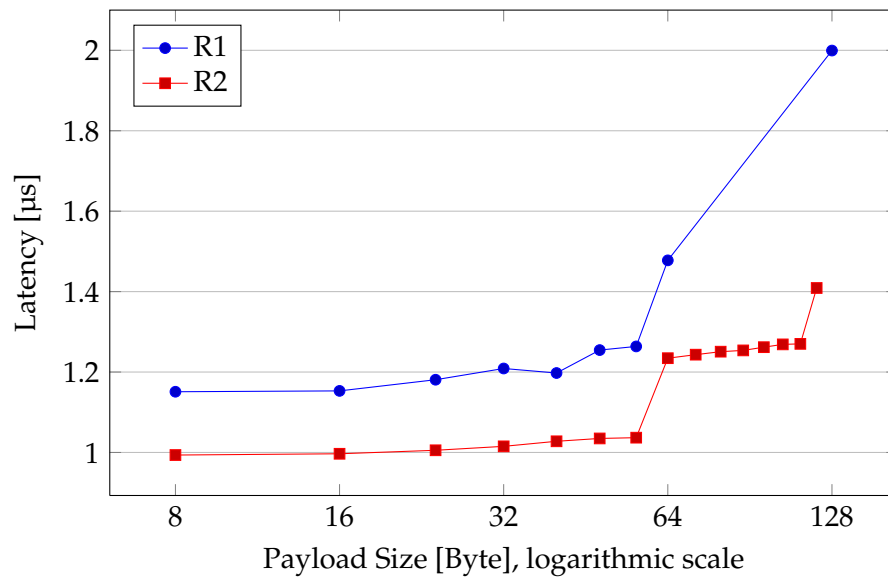


Figure 4.23: EXTOLL VELO Latency Comparison

For the first test two hosts are utilized where each one is initialized with a single thread. These two threads send messages to each other in a ping-pong manner. This test effectively shows the latency that can be achieved using VELO and the EXTOLL network. Figure 4.23 shows the results, the numbers show the time required for a half-round-trip, including all

required SW layers. Compared to R1 the latency is smaller in every case. The minimum latency that can be achieved for a message with a payload of up to eight byte is $0.99\ \mu\text{s}$ compared to $1.15\ \mu\text{s}$ in R1, a 16 percent improvement. In addition, the gradient of the curve is smaller as in R1, especially for larger packets.

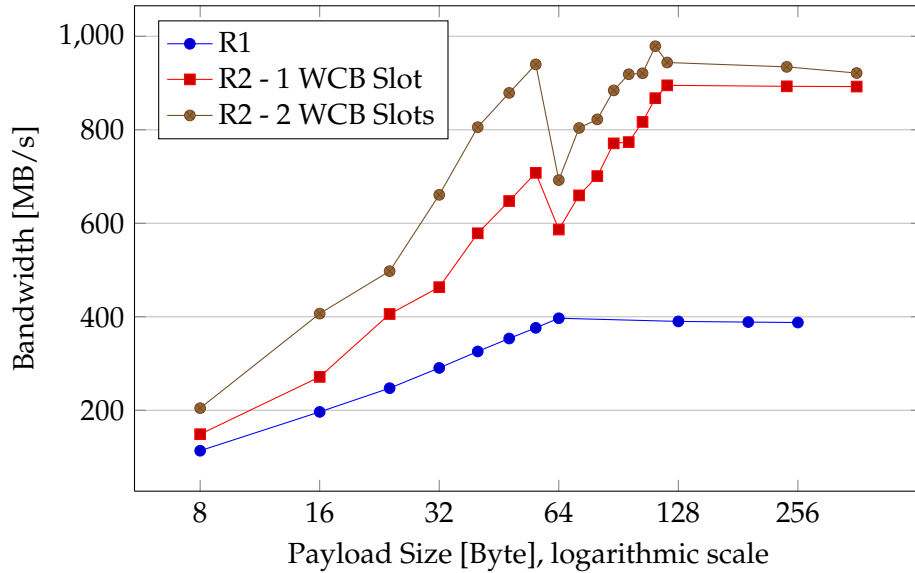


Figure 4.24: EXTOLL VELO Bandwidth Comparison

In a second test one thread is initialized solely as a sender, another one as a receiver, streaming messages from one host to the other. Figure 4.24 shows the achievable bandwidth in that configuration. The *Write Combining Buffer* forms the performance limiting factor due to its single slot per VPID implementation, as explained in section 2.3 on page 22. The achievable bandwidth is 891 MB/s. To remove this bottleneck the sender thread can be set up to utilize two VPIDs of the *Write Combining Buffer*, thus using two slots, effectively removing said bottleneck. This improves the bandwidth performance to 921 MB/s. Compared to R1 this is an improvement of 259 percent. This can be explained with the wider network data interface on the one side, but also the pipelining of the *VELO Completer* plays an important role in that matter.

The two performance drops in the R2 plot of both, figure 4.23 on the facing page and figure 4.24, can be explained through the cache line usage. Exceeding a payload size of 56 byte requires a second cache line on both, the sender and the receiver, requiring two write- and two read accesses. When more than 112 byte are transmitted a third cache line is required, again adding another write- and read access.

Table 4.3 on the following page summarizes the collected results and compares them with results obtained from an InfiniBand QDR adapter [126]. Already the FPGA imple-

	Latency [μ s]	Message Rate [mio. messages/s]	Bandwidth [MB/s]
EXTOLL R1	1.15	5	355
EXTOLL R2	0.99	25	921
InfiniBand QDR	1.0	9	3400
EXTOLL ASIC (estimated)	0.6	100	10000

Table 4.3: EXTOLL R2 VELO Performance

mentation of EXTOLL R2 can compete with current ASIC InfiniBand solutions in respect to latency and message rate. The bandwidth results are limited due to the relatively low clock speeds of the FPGA. Estimated values for an ASIC implementation, targeted to be running at 750 MHz, are even competitive at bandwidth tests.

5 Conclusion

The goal of this thesis has been to design and develop *Hardware Support for Efficient Packet Processing*, thus showing ways to improve the overall performance of packet processing tasks by improving the key issues, throughput and latency, in today's packet processing. These goals have been achieved by identifying bottlenecks and describing solutions to remove or at least reduce them by using hardware (HW) structures.

All introduced HW structures have been implemented using synthesizable *Verilog* code and tested in real world applications using field programmable gate array (FPGA) technology. These tests are far more reliable as synthetic system level simulations. The results proof that the proposed architectures are capable of delivering performance improvements while keeping the resource consumptions in line.

Individual contributions in the first part of this thesis are the arbiter generator and the first-in-first-outs (FIFOs) implementations with additional features. These implementations can be leveraged by many applications introduced in chapter 2 on page 7 as well as in Extended ATOLL (EXTOLL). Their additional capabilities enables the designer for example to efficiently implement a packet store-and-forward mechanism, as it is done in EXTOLL's *Network Port*.

The introduced Virtual Ring Buffer Handler (VHRD) enables the realization of a plurality of large main memory ring buffers that can be leveraged by multiple HW modules. The four MegaByte (MB) restriction imposed by the Linux operating system (OS) for memory allocation can be circumvented efficiently. HW modules leveraging the VHRD are offered with a simple address request interface and no longer need to handle address generation and flow control issues, thus simplifying all modules requiring main memory write access.

The *Tag Matching Unit* offers an efficient way to accelerate Message Passing Interface (MPI)-based communication, by offloading the time consuming matching task to specialized hardware. The HW is capable of distributing the search for a tag to an arbitrary number of *matching engines* using a round-robin scheme, significantly reducing the latency required to find a match. A patent is pending, covering the above described functionality.

5 Conclusion

The *FIX Adapted for Streaming (FAST) Decoder* introduced in section 2.6 on page 42 delivers outstanding decoding and filtering performance. FAST is a protocol widely deployed in stock exchanges worldwide. Even with the limitations inflicted when using FPGA technology, the proposed architecture is capable of outperforming all currently known software (SW) alternatives. The FAST decoder requires only 1.54 cycles to decode a single field inside a data stream. The time consuming task of decoding the highly serial protocol is efficiently solved in a specialized HW structure, freeing viable Central Processing Unit (CPU) cycles for actual trading algorithms. In addition, its capabilities to write normalized data directly to the user-level address space, enables kernel bypass streaming with the lowest possible latency.

In the EXTOLL context the main contributions are the two bridging modules. The *PCI Express (PCIe)-Bridge* is capable of translating PCIe packets to HyperTransport on-chip Protocol (HToC) packets and vice versa with minimal latency. This enables EXTOLL to leverage both, PCIe and HyperTransport (HT) as a host interface without the need to modify any unit of the Network Interface Controller (NIC) part of EXTOLL.

After translating packets from the two independent host interfaces to a common packet specification, the *HyperTransport Advanced Crossbar (HTAX)-Bridge* can be leveraged to forward arriving packets to their targeted functional unit (FU) with a high flexibility and again a very low latency of only two clock cycles. In the opposing direction the unit is capable of receiving multiple independent packet streams from the HTAX and forwarding them to the according host interface.

The optimizations applied to the Virtualized Engine for Low Overhead (VELO) unit enable EXTOLL to leverage its low latency host interface with larger message- and cluster sizes. Pipelining the design made it capable of fully leveraging the offered bandwidth on both, the network as well as the host side, again keeping the latency at a minimum. Additionally, the receive buffers have been increased significantly deploying the VHRD unit. The cache usage inside the host CPU has been improved by reducing the required address space to communicate using VELO, effectively reducing cache trashing, despite the fact that now up to 1024 virtual process identifications (VPIDs) are supported. As a result of the restructuring, VELO now supports two orders of magnitude more target nodes and 32 times more VPIDs as the previous version. The maximum message size on the network side has been increased to 2048 bytes, to increase the usability. These changes improved the bandwidth by more than 259 percent from 355 to 921 MB/s. The latency of VELO has been reduced by 16 percent, from 1.14 to 0.99 μ s, breaking the μ s barrier.

The contribution with the most impact however, is the newly designed EXTOLL cross-

bar. Its feature rich design supports HW based multicasting, effectively improving the collective operations performance of EXTOLL. The introduced credit scheme enables the network to support more in-flight packets, improving the buffer utilization from 25 to 97.7 percent for a typical packet size. This allows more in-flight packets in the network without the need to increase the internal buffers. In addition, adaptive routing has been introduced to the crossbar design to reduce the probability of congestions in EXTOLL powered networks. All these features have been added without a negative impact on the route-through latency of the crossbar. Although the number of pipeline stages increased from 13 to 16, the latency has been reduced to 80 ns. This has been enabled through a higher achievable clock frequency due to the revised pipeline structure. The crossbar design will offer a very good performance with a latency as low as 22 ns in the upcoming application specific integrated circuit (ASIC) implementation.

The current FPGA implementation of EXTOLL already proves that it is capable of competing with other high performance interconnection networks. The upcoming ASIC implementation will show that the herein proposed concepts and modules are capable of delivering unreached performance values.

Acronyms

AMD	Advanced Micro Devices.
ASIC	application specific integrated circuit.
ATOLL	ATOMIC Low Latency.
ATU	Address Translation Unit.
BAR	base address register.
BTS	binary tree search.
CAG	Computer Architecture Group of the University of Heidelberg.
CPI	cycles per instruction.
CPU	Central Processing Unit.
CRC	cyclic redundancy check.
DHS	Distributed Header Store.
DMA	direct memory access.
EOP	end of packet.
EOP_E	end of erroneous packet.
ETH	Ethernet Protocol.
EXTOLL	Extended ATOLL.
FAST	FIX Adapted for Streaming.
FCC	flow control channel.
FIFO	first-in-first-out.
FIX	Financial Information Exchange Protocol.
flit	flow control unit.
FLOPS	floating point operations per second.

Acronyms

FPGA	field programmable gate array.
FSM	finite state machine.
FU	functional unit.
GB	GigaByte.
GPGPU	general purpose graphics processing unit.
HDL	hardware description language.
HFT	high frequency trading.
HOL	head-of-line blocking.
HPC	high performance computing.
HT	HyperTransport.
HTAX	HyperTransport Advanced Crossbar.
HToC	HyperTransport on-chip Protocol.
HTX	HyperTransport expansion slot.
HW	hardware.
ICC	InterConnect Controller.
ID	identifier.
IGMP	Internet Group Management Protocol.
IPC	inter process communication.
ISA	Industry Standard Architecture.
kB	KiloByte.
LP	Link Port.
LUT	look-up table.
MB	MegaByte.
MMU	Memory Management Unit.
MPI	Message Passing Interface.
MTT	Message Type Tag.
MTU	maximum transfer unit.
NIC	Network Interface Controller.

NOP	no operation.
NP	Network Port.
NRC	Network Routing Chip.
OPRA	Options Price Reporting Authority.
OS	operating system.
PCB	printed circuit board.
PCI	Peripheral Component Interconnect.
PCI-X	PCI-eXtended.
PCIe	PCI Express.
PDID	Protection Domain Identifier.
PGAS	Partitioned Global Address Space.
phit	physical unit.
PIO	Programmed Input/Output.
PMAP	Presence Map.
PR	posted send request.
PRQ	Posted Send Request Queue.
QPI	QuickPath Interconnect.
R1	Revision 1.
R2	Revision 2.
RAM	random access memory.
RDMA	remote direct memory access.
RF	Register File.
RFS	Register File Surrogate.
RMA	Remote Memory Access.
RTL	register transfer level.
sint32	32 Bit wide signed integer.
sint64	64 Bit wide signed integer.
SMFU	Shared Memory Functional Unit.
SOC	start of command.
SOD	start of data.

Acronyms

SOP	start of packet.
SW	software.
TC	traffic class.
TCP	Transmission Control Protocol.
TID	Template Identifier.
TLB	Translation Look-Aside Buffer.
TMU	Tag Matching Unit.
TNI	Tofu Network Interface.
TNR	Tofu Network Router.
UDP	User Datagram Protocol.
UE	unexpected receive.
UEQ	Unexpected Receive Queue.
uint32	32 Bit wide unsigned integer.
uint64	64 Bit wide unsigned integer.
UPL	upper layer protocol.
VC	virtual channel.
VCT	virtual cut-through switching.
VELO	Virtualized Engine for Low Overhead.
VHRD	Virtual Ring Buffer Handler.
VOQ	virtual output queuing.
VPID	virtual process identification.
XML	Extensible Markup Language.

Bibliography

- [1] LINPACK Benchmark. <http://www.netlib.org/linpack/>, retrieved 2011.
- [2] K-Computer. http://www.nsc.riken.jp/K/diary_eng.html, retrieved 2011.
- [3] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *Proceedings of the April 18-20 1967 spring joint computer conference*, 23(4):483–485, 1967.
- [4] Top500 List of June 2011. <http://top500.org/lists/2011/06>, retrieved 2011.
- [5] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. *Access*, pages 83–87, 2010.
- [6] D. Sima, T. Fountain, and P. Kacsuk. *Advanced Computer Architectures: A Design Space Approach*. Addison Wesley, 1997.
- [7] Infiniband Trade Association. <http://www.infinibandta.org/>, retrieved 2011.
- [8] Solarflare Communications Inc. Solarflare 10GbE Server Adapters with OpenOnload. Technical Report 0, 2011.
- [9] QLogic. <http://http://www.qlogic.com/Pages/default.aspx>, retrieved 2011.
- [10] QLogic. The Impact of InfiniBand Architecture on CPU Utilization. Technical report, QLogic, 2011.
- [11] Message Passing Interface Forum. <http://www.mpi-forum.org/docs/docs.html>, retrieved Sept. 2008.
- [12] H. Bellm. Entwurf und Implementierung eines parametrisierbaren Arbitergenerators in Verilog. Technical report, University of Mannheim, 2002.

5 Bibliography

- [13] P. Gupta and N. McKeown. Designing and Implementing a fast Crossbar Scheduler. *Ieee Micro*, 19(1):20–28, 1999.
- [14] C.E. Savin, T. McSmythurs, and J. Czilli. Binary Tree Search Architecture for Efficient Implementation of Round Robin Arbiters. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings.(ICASSP'04). IEEE International Conference on*, volume 5, pages V–333. IEEE, 2004.
- [15] D. E. Knuth. *The Art of Computer Programming, Volume 3*, volume 3 of *Computer Science and Information Processing*. Addison-Wesley, 1973.
- [16] The Perl Programming Language. <http://www.perl.org/>, retrieved 2011.
- [17] SystemVerilog. <http://www.systemverilog.org/>, retrieved 2011.
- [18] C.E. Cummings. Simulation and Synthesis Techniques for Asynchronous FIFO Design. *Synopsys Users Group*, pages 1–23, 2002.
- [19] M.T. Oliveira. Multi-threaded FIFO Memory Generator with Speculative Read and Write Capability. *US Patent 7,756,148*, 2010.
- [20] Y. Tamir and G.L. Frazier. Dynamically-allocated Multi-Queue Buffers for VLSI Communication Switches. *Computers, IEEE Transactions on*, 41(6):725–737, 1992.
- [21] S. Secatch. Pushback fifo, September 2003.
- [22] M.D. Ward, K.L. Williams, and K.J. Craig. Fifo with fast Retransmit Mode, November 1994.
- [23] G.E. Ehmann. FIFO Buffer that can Read and/or Write Multiple and/or Selectable Number of Data Words per bus cycle, March 2004.
- [24] Linux Foundation. <http://www.linuxfoundation.org/>, retrieved 2011.
- [25] HyperTransport Link Specifications. <http://www.hypertransport.org/default.cfm?page=HyperTransportSpecifications>, retrieved 2011.
- [26] B. Kalisch, A. Giese, H. Litz, and U. Brüning. HyperTransport 3 Core: A Next Generation Host Interface with Extremely High Bandwidth. In *Proceeding of the First International Workshop on Hyper-Transport Research and Applications (WHTRA'09)*, 2009.

- [27] Intel QuickPath Technology. <http://www.intel.com/content/www/us/en/io/quickpath-technology/quickpath-technology-general.html>, retrieved 2011.
- [28] D. Ziakas, A. Baum, R. Maddox, and R. J. Safranek. Intel® QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. *2010 18th IEEE Symposium on High Performance Interconnects*, pages 1–6, August 2010.
- [29] PCI Special Interest Group. <http://www.pcisig.com/specifications/pciexpress/>, retrieved 2011.
- [30] O. Ichikawa. Computer System having a Data Buffering System which includes a Main Ring Buffer comprised of a Plurality of Sub-Ring Buffers connected in a Ring. *US Patent 5,948,082*, 1999.
- [31] IBM International Business Machines. <http://www.ibm.com/>, retrieved 2011.
- [32] A. G. Shet, P. Sadayappan, D. E. Bernholdt, J. Nieplocha, and V. Tipparaju. A Performance Instrumentation Framework to Characterize Computation-Communication Overlap in Message-Passing Systems. *2006 IEEE International Conference on Cluster Computing*, 2006.
- [33] Myricom Inc. Myrinet Express (MX): A High-Performance, Low-Level Message-Passing Interface for Myrinet; version 1.2. Technical Report Mx, Myricom, Inc., 2006.
- [34] R. Brightwell, R. Riesen, B. Lawry, and A.B. Maccabe. Portals 3.0: Protocol Building Blocks for low Overhead Communication. *Proceedings 16th International Parallel and Distributed Processing Symposium*, page 164, 2002.
- [35] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K.D. Underwood. Implementation and Performance of Portals 3.3 on the Cray XT3. *2005 IEEE International Conference on Cluster Computing*, pages 1–10, September 2005.
- [36] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1), 2002.
- [37] L. Dickman, G. Lindahl, D. Olson, J. Rubin, J. Broughton, and PathScale. PathScale InfiniPath: A First Look. In *Proceedings of the 13th Symposium on High Performance Interconnects*, pages 163–165, 2005.

5 Bibliography

- [38] R. Brightwell and K.D. Underwood. An Analysis of NIC Resource Usage for Offloading MPI. *Proceedings of the 2004 Workshop on Communication*, 2004.
- [39] K.D. Underwood, A. Rodrigues, and K.S. Hemmert. Accelerating List Management for MPI. *2005 IEEE International Conference on Cluster Computing*, pages 1–10, September 2005.
- [40] K.D. Underwood, K.S. Hemmert, A. Rodrigues, R. Murphy, and R. Brightwell. A Hardware Acceleration Unit for MPI Queue Processing. *19th IEEE International Parallel and Distributed Processing Symposium*, pages 96b–96b, 2005.
- [41] MPICH2. <http://www.mcs.anl.gov/research/projects/mpich2/>, retrieved August 2008.
- [42] MVAPICH MPI. <http://mvapich.cse.ohio-state.edu>, retrieved 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, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. *Proceedings of the 11th European PVM/MPI Users' Group Meeting (Euro-PVM/MPI04)*, pages 97–104, 2004.
- [44] J. Park, B.W. O’Krafka, S. Vassiliadis, and J. Delgado-Frias. Design and Evaluation of a DAMQ Multiprocessor Network with Self-Compacting Buffers. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 713–722. ACM, 1994.
- [45] Aite Group. New World Order: The High Frequency Trading Community and Its Impact on Market Structure, 2009.
- [46] J. Brogaard. High Frequency Trading and its Impact on Market Quality. *Management*, pages 1–6, 2010.
- [47] M. Chlistalla. High Frequency Trading: Better than its Reputation?, 2011.
- [48] FIX Adapted for SStreaming. <http://www.fixprotocol.org/fast>, retrieved 2010.
- [49] Financial Information eXchange (FIX) Protocol. <http://www.fixprotocol.org/>, retrieved 2011.
- [50] H. Subramoni, F. Petrini, and V. Agarwal. Streaming, Low-latency Communication

- in on-line Trading Systems. *Parallel & Distributed*, 24972, 2010.
- [51] Myrinet Network Adapters. <http://www.myricom.com/products/legacy-products/myrinet-2000/network-adapters.html>, retrieved 2011.
 - [52] V. Agarwal, D. A. Bader, L. Dan, L. Liu, D. Pasetto, M. Perrone, and F. Petrini. Faster FAST: Multicore Acceleration of Streaming Financial Data. *Computer Science Research and Development*, 23(3-4):249–257, 2009.
 - [53] G. W. Morris, D. B. Thomas, and W. Luk. FPGA Accelerated Low-Latency Market Data Feed Processing. *2009 17th IEEE Symposium on High Performance Interconnects*, pages 83–89, 2009.
 - [54] F. L. Herrmann, G. Perin, J. P. J. De Freitas, R. Bertagnolli, and J. B. Dos Santos Martins. A Gigabit UDP/IP Network Stack in FPGA. In *16th IEEE International Conference on Electronics Circuits and Systems*, pages 836–839. IEEE, 2009.
 - [55] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H. Jacobsen. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *Proceedings of the VLDB Endowment*, 3(2):1525–1528, 2010.
 - [56] Options Price Reporting Authority. <http://www.opradata.com/>, retrieved 2012.
 - [57] D. Pasetto, K. Lynch, R. Tucker, B. Maguire, F. Petrini, and H. Franke. Ultra low Latency Market Data Feed on IBM PowerEN. *Computer Science - Research and Development*, 26(3-4):307–315, April 2011.
 - [58] The JAVA Programming Language. http://java.com/de/download/whatis_java.jsp, retrieved November 2011.
 - [59] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, volume 18 of *Software series*. Prentice Hall, 1988.
 - [60] F. Lemke. Fsmdesigner4 - development of a tool for interactive design and hardware description language generation of finite state machines. Diploma thesis, University of Mannheim, 2006.
 - [61] FSMDesigner 4. <http://sourceforge.net/projects/fsmdesigner/>, retrieved 2011.

5 Bibliography

- [62] J. Chao, J.S. Park, and W.S. Su. Performance Study of Commercial ATM Switches. Available at citeseer.nj.nec.com/491482.html.
- [63] J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: an Engeneering Approach*. IEEE Society Press, 1997.
- [64] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, and N. Chrysos. Variable Packet Size Buffered Crossbar (CICQ) Switches. *2004 IEEE International Conference on Communications (IEEE Cat. No.04CH37577)*, (June):1090–1096 Vol.2, 2004.
- [65] N. McKeown, M. Izzard, A. Mekkittikul, W. Ellersick, and M. Horowitz. Tiny Tera: a Packet Switch Core. *IEEE Micro*, 17(1):26–33, 1997.
- [66] R. Rojas-Cessa, E. Oki, and H.J. Chao. CIXB-1: Combined Input-One-Cell-Crosspoint Buffered Switch. *2001 IEEE Workshop on High Performance Switching and Routing (IEEE Cat. No.01TH8552)*, 00(718):324–329, 2001.
- [67] R. Rojas-Cessa, E. Oki, and H.J. Chao. CIXOB-k: Combined Input-Crosspoint-Output Buffered Packet Switch. *GLOBECOM'01. IEEE Global Telecommunications Conference (Cat. No.01CH37270)*, 4:2654–2660, 2001.
- [68] M. Marsan, A. Bianco, E. Filippi, and P. Giaccone. A Comparison of Input Queuing Cell Switch Architectures. *IEEE BSS*, 1999.
- [69] A. Mekkittikul and N. McKeown. A Practical Scheduling Algorithm to Achieve 100% Throughput in Input-Queued Switches. *Proceedings IEEE INFOCOM 98 the Conference on Computer Communications Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies Gateway to the 21st Century Cat No98CH36169*, 2:792–799, 1998.
- [70] R. O. LaMaire and D. N. Serpanos. Two-Dimensional Round-Robin Schedulers for Packet Switches with Multiple Input Queues. *IEEEACM Trans Netw*, 2(5):471–482, 1994.
- [71] H.J. Chao. Centralized Contention Resolution Schemes for a Large-Capacity Optical ATM Switch. *ATM Workshop Proceedings, 1998 IEEE*, 1998.
- [72] N. McKeown. The iSLIP Scheduling Algorithm for Input-Queued Switches. *Networking, IEEE/ACM Transactions on*, 7(2):188–201, April 1999.

- [73] P. Gupta. Scheduling in Input Queued Switches: A Survey. *citeseer. nj. nec. com/246798. html*, 1996.
- [74] Prof. Dr. U. Brüning. Rechnerarchitektur 2. Script for Lecture, 2003.
- [75] W.J. Dally. Virtual-Channel Flow Control. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):194–205, March 1992.
- [76] W.J. Dally and H. Aoki. Deadlock-Free Adaptive Routing in Multicomputer Networks Using Virtual Lanes. *IEEE Transactions on Parallel and Distributed Systems*, 1993.
- [77] W.J. Dally and C.L. Seitz. Deadlock-free Message Routing in Multiprocessor Interconnection Networks. *Computers, IEEE Transactions on*, 100(5):547–553, 1987.
- [78] T. Nachiondo, J. Flich, and J. Duato. Efficient Reduction of HOL Blocking in Multistage Networks. In *Proc 19th IEEE Int Parallel and Distributed Processing Symp*, 2005.
- [79] Y. Tamir and G. L. Frazier. High-Performance Multi-Queue Buffers for VLSI Communication Switches. *Micro*, pages 343–354, 1988.
- [80] S. Scott, D. Abts, J. Kim, and W.J. Dally. The Blackwidow High-radix Clos Network. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 16–28. IEEE Computer Society, 2006.
- [81] Y. Ajima, S. Sumimoto, and T. Shimizu. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *Computer*, 42(11):36–40, 2009.
- [82] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu. The Tofu Interconnect. *IEEE Symposium on High Performance Interconnects*, pages 87–94, 2011.
- [83] Top500 List of November 2010. <http://top500.org/lists/2010/11>, retrieved 2011.
- [84] M. Xie, Y. Lu, L. Liu, H. Cao, and X. Yang. Implementation and Evaluation of Network Interface and Message Passing Services for TianHe-1A Supercomputer. *IEEE Symposium on High Performance Interconnects*, pages 78–86, 2011.
- [85] X. Yang, X Liao, and J. Song. The TianHe-1A Supercomputer: Its Hardware and Software. *Science And Technology*, 26(2009):344–351, 2011.

5 Bibliography

- [86] B. Steinmacher-burow, D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, and J. J. Parker. The IBM Blue Gene / Q Interconnection Network and Message Unit. *Network*, pages 1–10, 2010.
- [87] U. Brüning, H. Fröning, P.R. Schulz, and L. Rzymianowicz. ATOLL: Performance and Cost Optimization of a SAN Interconnect. In *Parallel and Distributed Computing and Systems*, number figure 1, pages 2–4. ACTA Press, 2002.
- [88] H. Fröning, M. Nüssle, D. Slogsnat, P. Haspel, and U. Brüning. Performance Evaluation of the ATOLL Interconnect. *Proc. of IASTED Conf. on Parallel and Distributed Computing and Networks (PDCN)*, pages 26–29, 2005.
- [89] F. Ueltzhöffer. *Design and Implementation of a Virtual Channel Based Low-Latency Crossbar Switch*. Diploma thesis, University of Mannheim, 2005.
- [90] B. Geib. *Improving and Extending a Crossbar Design for ASIC and FPGA Implementation*. Diploma thesis, University of Mannheim, 2007.
- [91] H. Montaner, F. Silla, H. Fröning, and J. Duato. MEMSCALE: a Scalable Environment for Databases. *13th IEEE International Conference on High Performance Computing and Communications (HPCC-2011)*,, pages 225–238, 2011.
- [92] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, and U. Brüning. The HTX-Board: A Rapid Prototyping Station. In *3rd annual FPGAWorld Conference*. Citeseer, 2006.
- [93] M. Nüssle, B. Geib, H. Fröning, and U. Brüning. An FPGA-Based Custom High Performance Interconnection Network. *RECONFIG*, 2009.
- [94] M. Nüssle. *Acceleration of the Hardware - Software Interface of a Communication Device for Parallel Systems*. Phd. thesis, University of Mannheim, 2008.
- [95] M. Nüssle, H. Fröning, and U. Brüning. SWORDFISH: A Simulator for High-Performance Networks. In *Parallel and Distributed Computing and Systems*. ACTA Press, 2005.
- [96] A. X. Widmer and P. A. Franaszek. A dc-balanced, partitioned-block, 8b/10b transmission code. *IBM Journal of Research and Development*, 27(5):440 –451, sept. 1983.
- [97] D. Slogsnat. *Tightly-Coupled and Fault-Tolerant Communication in Parallel Systems*.

- Phd. thesis, University of Mannheim, 2008.
- [98] N. Burkhardt. *Fast Hardware Barrier Synchronisation for a Reliable Interconnection Network*. Diploma thesis, University of Mannheim, 2007.
 - [99] Xilinx Virtex-4 Devices. <http://www.xilinx.com/support/documentation/virtex-4.htm>, retrieved 2011.
 - [100] Xilinx Virtex-6 LXT Devices. <http://www.xilinx.com/products/silicon-devices/fpga/virtex-6/lxt.htm>, retrieved 2011.
 - [101] EXTOLL GmbH. <http://www.extoll.de/>, retrieved 2011.
 - [102] P. Kermani and L. Kleinrock. Virtual Cut-Through: A new Computer Communication Switching Technique. *Computer Networks*, 3(4):267–286, 1979.
 - [103] J. Duato. A necessary and sufficient Condition for Deadlock-free Adaptive Routing in Wormhole Networks. *IEEE Transactions on Parallel and Distributed Systems*, 6(10):1055–1067, 1995.
 - [104] N. Burkhardt. Extoll network protocol specification. Internal Working Paper of the CAG, 2010.
 - [105] N. McKeown. Fast Switched Backplane for a Gigabit Switched Router. *Business Communications Review*, 27(12):1–30, 1997.
 - [106] D. Pan and Y. Yang. FIFO-Based Multicast Scheduling Algorithm for Virtual Output Queued Packet Switches. *IEEE Transactions on Computers*, 54(10):1283–1297, October 2005.
 - [107] Xilinx Inc. Virtex-6 FPGA Memory Resources User Guide, 2011.
 - [108] D. Abts and D. Weisser. Age-based Packet Arbitration in Large-Radix k-ary n-Cubes. *Proceedings of the 2007 ACM/IEEE conference on Supercomputing - SC '07*, page 1, 2007.
 - [109] L.M. Bhuyan. Achieving Fairness and Throughput for Best-Effort Traffic in Input-Queued Crossbar Switches. In *GLOBECOM '05. IEEE Global Telecommunications Conference, 2005.*, volume 1, page 6 pp. Ieee, 2005.
 - [110] C. Leber. Rfs - register file surrogate. Internal Working Paper of the CAG, 2010.

5 Bibliography

- [111] Xilinx ISE Design Tool. <http://www.xilinx.com/products/design-tools/ise-design-suite/index.htm>, retrieved 2011.
- [112] H. Fröning. *Architectural Improvements of Interconnection Network Interfaces*. Phd. thesis, University of Mannheim, 2007.
- [113] H. Litz. *Hardware Techniques to Extend the Scalability of future high Performance Computer Systems*. Ph.d., University of Mannheim, 2010.
- [114] AMD Advanced Micro Devices. <http://www.amd.com/>, retrieved 2011.
- [115] AMD Opteron Processor. <http://www.amd.com/us/products/server/processors/Pages/server-processors.aspx>, retrieved 2011.
- [116] Intel. <http://www.intel.com/>, retrieved 2011.
- [117] Intel® Many Integrated Core Architecture. <http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html>, retrieved 2011.
- [118] H. Litz. HyperTransport Advanced X-Bar (HTAX) Specification. 2008.
- [119] H. Litz. HyperTransport On-Chip (HTOC) Protocol Specification. 2010.
- [120] M. Nüssle, M. Scherer, and U. Brüning. A Resource Optimized Remote-Memory-Access Architecture for Low-Latency Communication. In *2009 International Conference on Parallel Processing, ICPP 2009*, pages 220–227. IEEE, 2009.
- [121] H. Froning and H. Litz. Efficient Hardware Support for the Partitioned Global Address Space. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–6. IEEE, 2010.
- [122] 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, and T. Wen. Productivity and Performance using Partitioned Global Address Space Languages. *International Conference on Symbolic and Algebraic Computation*, 2007.
- [123] B. Holden. Latency Comparison between HyperTransport and PCIexpress in Communications Systems. *HyperTransport™ Consortium*, 2006.
- [124] H. Litz, H. Froening, M. Nüssle, and U. Brüning. VELO: A Novel Communication

Engine for Ultra-low Latency Message Transfers. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 238–245. IEEE, 2008.

[125] GASnet. <http://gasnet.cs.berkeley.edu/>, retrieved 2008.

[126] Mellanox Connect-X Infiniband Performance Numbers. http://www.mellanox.com/content/pages.php?pg=performance_infiniband, retrieved 2012.

List of Figures

1.1	Performance Gain when Parallelizing a Computational Task [3]	2
1.2	An Example Design Space Analysis	4
1.3	An Example Block Diagram	4
1.4	An Example Timing Diagram	5
2.1	FSM based Arbiter Resource and Performance Evaluation	10
2.2	Simplified Block Diagram of the BTS Arbiter [14]	11
2.3	BTS based Arbiter Resource and Performance Evaluation	12
2.4	FIFO Block Diagram [18]	12
2.5	Multi-Queue-FIFO Block Diagram	15
2.6	Address Generation Using a Complex Logic	16
2.7	Address Generation Using a Reversed Queue Selection	16
2.8	Resource Requirements of N conventional FIFOs	17
2.9	Resource Requirements of a Multi-Queue FIFO with N Queues	18
2.10	Speculative FIFO Block Diagram	19
2.11	Write Combining Buffer Block Diagram	23
2.12	Memory Mapping of Segments to User Space	25
2.13	VRHD Design Space Analysis	27
2.14	VRHD Block Diagram	28
2.15	VRHD Descriptor Queue Element	29
2.16	VRHD Queue State RAM	30
2.17	Example 1	32
2.18	Example 2a	33
2.19	Example 2b	34
2.20	TMU Block Diagram	36
2.21	TMU Header Format	38
2.22	FAST Example Template	44
2.23	Fast Messages Encapsulated in UDP Frame	45
2.24	Design Space for FAST Decoding	47
2.25	FAST Processor Block Diagram	48
2.26	FAST Data Flow	49

List of Figures

2.27	FAST Ring Buffer Usage	50
2.28	FAST Example Assembler	52
2.29	FAST Message Distribution	55
2.30	FAST Decoding Bandwidth Capabilities	61
2.31	FAST Bandwidth Comparison [53][57]	61
2.32	FAST Message Decoding Latency Comparison [53][57]	62
3.1	Design Space Diagram for Switching Elements [62]	63
3.2	Connecting Multiple Units Using a Bus	64
3.3	Connecting Multiple Units Using a Crossbar	65
3.4	Detailed View of a Crossbar	66
3.5	Crossbar Buffering Design Space Analysis	67
3.6	Detailed View of a Crosspoint Fabric	67
3.7	Scheduling and Packet Length Design Space Diagram	68
3.8	Topology Design Space Diagram	69
3.9	Indirect Topologies	70
3.10	Direct Topologies	70
3.11	An Example for a Deadlock Condition	71
3.12	An Example for Virtual Channels	71
3.13	An Example for Head-of-Line Blocking	72
3.14	An Example for Virtual Output Queuing	73
3.15	Gemini Block Diagram [5]	77
3.16	ICC Block Diagram [81]	79
3.17	BlueGene/Q Network Device Block Diagram [86]	81
3.18	The HTX Testbed in the 4th Revision	82
3.19	EXTOLL Crossbar R1 Packet Format	85
3.20	EXTOLL Crossbar Routing Phit	86
3.21	EXTOLL Crossbar Top Block Diagram	87
3.22	EXTOLL Crossbar R1 Inport Block Diagram	88
3.23	EXTOLL Crossbar Outport Block Diagram	90
3.24	EXTOLL R1 Crossbar Message Rate	91
3.25	EXTOLL R1 Crossbar Bandwidth	91
3.26	EXTOLL R1 Crossbar Bandwidth Efficiency	92
3.27	EXTOLL Crossbar R1 Route-Through Latency	92
3.28	EXTOLL R2 Routing Design Space Diagram	94
3.29	Network Partitioning	97
3.30	EXTOLL R2 Routing Table Entry	98
3.31	EXTOLL R1 Routing Problem (1)	99
3.32	EXTOLL R1 Routing Problem (2)	100

3.33	EXTOLL R2 Packet Format	101
3.34	Multicast Example	103
3.35	Multicast Design Space Analysis	103
3.36	EXTOLL R2 Multicast Routing Table Entry	106
3.37	Crossbar Buffer Usage	109
3.38	Crossbar Buffer Usage when Storing the last Packet	110
3.39	Crossbar Buffer Efficiency	111
3.40	Early Arbitration Timing Diagram	112
3.41	Credit Administration Optimization Code Example	114
3.42	Optimization Code Example	118
3.43	Code Optimization Example	119
3.44	EXTOLL R2 Crossbar Inport Block Diagram	120
3.45	EXTOLL R2 Crossbar Resource Utilization	121
3.46	EXTOLL R2 Crossbar Message Rate	123
3.47	EXTOLL R2 Crossbar Bandwidth	123
3.48	EXTOLL R2 Crossbar Bandwidth Efficiency	124
3.49	EXTOLL R2 Crossbar Route-Through Latency	124
3.50	EXTOLL Ventoux	125
3.51	Dropping Erroneous Packets Design Space Diagram	128
4.1	EXTOLL Block Diagram	134
4.2	EXTOLL Host Interface Connectivity Design Space	136
4.3	EXTOLL Register File Tool Flow	138
4.4	Ping-Pong Latency Results for RMA and VELO in EXTOLL R1 [94]	140
4.5	EXTOLL R2 Block Diagram with highlighted Modules	141
4.6	HTAX Bridge Design Space Diagram	142
4.7	HTAX Bridge Block Diagram	145
4.8	HTAX Bridge Tag Mapping Example	146
4.9	PCIe Bridge Design Space Diagram	149
4.10	PCIe Bridge Block Diagram	151
4.11	NP Block Diagram	152
4.12	VELO R1 Block Diagram	154
4.13	VELO R1 Status Word	155
4.14	VELO R1 Address Encoding	155
4.15	VELO R1 Ring Buffer Usage	156
4.16	VELO R2 Address Encoding	159
4.17	VELO R2 Status Word as the Requester receives it from SW	160
4.18	VELO R2 Status Word as it is Transmitted Inside the Network	161
4.19	VELO R2 Block Diagram	162

List of Figures

4.20 Main Memory Queue Design Space Diagram	163
4.21 VELO Memory Slot Usage	165
4.22 VELO R2 Status Word as the Completer writes it into Main Memory . . .	165
4.23 EXTOLL VELO Latency Comparison	166
4.24 EXTOLL VELO Bandwidth Comparison	167

List of Tables

2.1	FIFO Cost Comparison	20
2.2	FIFO Cost Comparison 2	21
2.3	Performance of Baseline Implementation	54
2.4	Costs of Baseline Implementation	54
2.5	Performance of Template Profiling Implementation	56
2.6	Costs of Template Profiling Implementation	56
2.7	Performance of Pointer Prefetching	57
2.8	Costs of Pointer Pre-fetching	57
2.9	Performance of Immediate Integer Decoding	58
2.10	Costs of Immediate Integer Decoding	58
2.11	Performance of Using a Two Shift-Out FIFO	59
2.12	Costs of Using a Two Shift-Out FIFO	59
3.1	Overview Over Different Interconnect Families	83
3.2	EXTOLL Crossbar R1 Synthesis Result	92
3.3	EXTOLL R1 Source-Path Routing Table Size	94
3.4	Minimum Size Requirements for Table-Based Routing	96
3.5	EXTOLL R2 SOP Field Definitions	102
3.6	Cost of different Credit Administration Techniques	113
3.7	Costs of Free Slot Management Implementations	115
3.8	Costs of Free Slot Management Implementations	116
3.9	Costs of Multi-Queue-FIFO Implementations	117
3.10	EXTOLL R2 Multicast Counter Code Optimization	120
3.11	EXTOLL R2 Crossbar Resource Utilization	122
3.12	Crossbar Synthesis Results	125
3.13	Read Slot Look-Ahead Resource Utilization for an Example Size	127
4.1	HT and corresponding PCIe Nomenclatures	147
4.2	HT and PCIe Comparison [123]	148
4.3	EXTOLL R2 VELO Performance	168