Improving the Scalability of High Performance Computer Systems

Inauguraldissertation

zur Erlangung des akademischen Grades

eines Doktors der Naturwissenschaften

der Universität Mannheim

vorgelegt von

Heiner Hannes Litz

aus Heidelberg

(Diplom-Informatiker der Technischen Informatik)

Mannheim, 2010

Dekan:	Professor Dr. W. Effelsberg, Universität Mannheim
Referent:	Professor Dr. U. Brüning, Universität Heidelberg
Korreferent:	Professor Dr. R. Männer, Universität Heidelberg

Tag der mündlichen Prüfung: 17. März 2011

Für Regine

Abstract

Improving the performance of future computing systems will be based upon the ability of increasing the scalability of current technology. New paths need to be explored, as operating principles that were applied up to now are becoming irrelevant for upcoming computer architectures. It appears that scaling the number of cores, processors and nodes within an system represents the only feasible alternative to achieve Exascale performance. To accomplish this goal, we propose three novel techniques addressing different layers of computer systems. The Tightly Coupled Cluster technique significantly improves the communication for inter node communication within compute clusters. By improving the latency by an order of magnitude over existing solutions the cost of communication is considerably reduced. This enables to exploit fine grain parallelism within applications, thereby, extending the scalability considerably. The mechanism virtually moves the network interconnect into the processor, bypassing the latency of the I/O interface and rendering protocol conversions unnecessary. The technique is implemented entirely through firmware and kernel layer software utilizing off-the-shelf AMD processors. We present a proof-of-concept implementation and real world benchmarks to demonstrate the superior performance of our technique. In particular, our approach achieves a software-to-software communication latency of 240 ns between two remote compute nodes.

The second part of the dissertation introduces a new framework for scalable Networks-on-Chip. A novel rapid prototyping methodology is proposed, that accelerates the design and implementation substantially. Due to its flexibility and modularity a large application space is covered ranging from Systems-on-chip, to high performance many-core processors. The Network-on-Chip compiler enables to generate complex networks in the form of synthesizable register transfer level code from an abstract design description. Our engine supports different target technologies including Field Programmable Gate Arrays and Application Specific Integrated Circuits. The framework enables to build large designs while minimizing development and verification efforts. Many topologies and routing algorithms are supported by partitioning the tasks into several layers and by the introduction of a protocol agnostic architecture. We provide a thorough evaluation of the design that shows excellent results regarding performance and scalability.

The third part of the dissertation addresses the Processor-Memory Interface within computer architectures. The increasing compute power of many-core processors, leads to an equally growing demand for more memory bandwidth and capacity. Current processor designs exhibit physical limitations that restrict the scalability of main memory. To address this issue we propose a memory extension technique that attaches large amounts of DRAM memory to the processor via a low pin count interface using high speed serial transceivers. Our technique transparently integrates the extension memory into the system architecture by providing full cache coherency. Therefore, applications can utilize the memory extension by applying regular shared memory programming techniques. By supporting daisy chained memory extension devices and by introducing the asymmetric probing approach, the proposed mechanism ensures high scalability. We furthermore propose a DMA offloading technique to improve the performance of the processor-memory interface. The design has been implemented in a Field Programmable Gate Array based prototype. Driver software and firmware modifications have been developed to bring up the prototype in a Linux based system. We show microbenchmarks that prove the feasibility of our design.

Zusammenfassung

Die Verbesserung der Skalierbarkeit zukünftiger Computer Systeme ist eine wichtige Vorraussetzung, um die Geschwindigkeit dieser zu erhöhen, da bisher angewendete Prinzipien zur Geschwindigkeitssteigerung in Zukunft nicht mehr anwendbar sein werden. Der Grund hierfür ist hauptsächlich im zu hohen Stromverbrauch begründet welcher durch die Erhöhung der Taktfrequenz verursacht wird. Eine Lösung für dieses Problem besteht darin, die Anzahl von Komponenten innerhalb eines Mikrochips, eines Rechenknotens und innerhalb von Rechennetzwerken zu erhöhen um eine höhere Gesamtleistung zu erzielen. Um dieses Ziel zu erreichen schlagen wir einen neuen Ansatz vor, genannt "Tightly Coupled Cluster". Unser Ansatz erlaubt es, Computer-Netzwerke zu realisieren, welche die Kommunikation mit extrem geringer Latenz ermöglichen. Dies ermöglicht das Ausnutzen fein granularer Parallelität welche in Applikationen vorhanden ist, was wiederum die Skalierbarkeit erhöht. Unser vorgeschlagener Mechanismus integriert die Netzwerkschnittstelle gewissermassen in den Prozessor, was ermöglicht zusätzliche Integrierte Schaltkreise zu umgehen, und eine Protokollübersetzung überflüssig macht. Wir präsentieren eine Implementierung unserer Technologie welche auf AMD Prozessoren aufsetzt. Tests belegen eine Software zu Software Kommunikationslatenz von 240 ns welche anderen Verfahren weit überlegen ist.

Im zweiten Teil der Dissertation wird ein Rahmenwerk zur Realisierung von Mikrochip internen Netzwerken vorgestellt. Sogenannte "Networks-on-Chip" bieten eine gute Möglichkeit um viele Komponenten innerhalb eines Mikrochips miteinander zu verbinden. Da sich die Anzahl von Funktionseinheiten innerhalb eines Mikrochips ständig erhöht, stellt die Kommunikationsschnittstelle eine wichtige und geschwindigkeitsbestimmende Komponente dar. Das Rahmenwerk welche die Struktur vorgibt um Netzwerke unterschiedlicher Topologie und Architektur zu entwerfen, stellt eine Software zur Verfügung, welche es ermöglicht auf Knopfdruck verschieden Designs automatisch zu generieren. Auf Basis einer abstrakten Beschreibung wird synthetisierbarer Code generiert. Die Evaluierung des Rahmenwerks belegt die gute Skalierbarkeit und Performanz unseres Ansatzes.

Im dritten Teil der Arbeit wird ein Mechanismus zur Verbesserung der Prozessor-Speicher Schnittstelle vorgeschlagen. Die Integrationsdichte zukünftiger Prozessoren stellt immer höhere Anforderungen an die Speicherkapazität und Speicherbandbreite. Um diese zu befriedigen, schlagen wir eine Speichererweiterung vor, welche auf Hochgeschwindigkeits-Transceivern basiert. Unsere Speichererweiterung lässt sich transparent in die Systemarchitektur integrieren, und ist Cache Kohärent. Die Möglichkeit Speicherhierarchien zu unterstützen sowie unserer vorgeschlagener “Asymmetric Probing” Mechanismus erhöht die Skalierbarkeit unserer Architektur. Zusätzlich stellen wir einen “DMA offloading” Mechanismus vor. Unser Ansatz wurde mittels eines “Field Programmable Gate Array” realisiert. Software Benchmarks belegen die Qualität unseres Ansatzes.

Acknowledgements

This dissertation is the result of many years of effort, training and support by other people. In particular, I want to thank the following people (in alphabetical order):

Parag Beeraka, Ulrich Brüning, Holger Fröning, Alexander Giese, Benjamin Kalisch, Berenike Litz, Mondrian Nüssle and Maximilian Thürmer.

Furthermore, I want to thank the companies AMD and SUN Microsystems for supporting my work.

Table of Contents

CHAPTER 1
INTRODUCTION

CHAPTER 1 INTRODUCTION

The past decades have shown a tremendous improvement of computer system performance. As a consequence, modern mobile phones offer an equal processing power as the largest mainframe computer twenty years ago. The key enabler of this development is the constantly increasing amount of transistors that can be placed inexpensively on integrated circuits. In fact, transistor density has doubled every two years over half a century. This trend, also described by Moore's Law [103], has enabled exponential improvements in many areas of integrated circuits, including advancements in processor speed, memory capacity and the pixel density in digital cameras. However, although Moore's Law will be intact for at least another 10 years it will be increasingly difficult to transform the abundance of transistors into increased performance. Recent developments yield that sustaining the pace of performance improvements enjoyed until today will hardly be possible.

So far, four important operating principles have been applied to transform additional transistors into greater performance: (1) increasing the operating frequency of the processor, (2) increasing the number of instructions that can be executed per clock cycle, (3) increasing the number of processing units per processor and (4) increasing the number of processors within a system. While these principles proved successful in the past, it has recently become evident that it will be impossible to apply options (1) and (2) to future processor generations. Raising the processor frequency above current levels is unfeasible due to power consumption. The so-called *power wall* does not further permit to increase the number of transistors as well as their frequency without exceeding the amount of heat that can be dissipated. Increasing the number of instructions per clock cycle, on the other hand, is limited by the *instruction level parallelism wall*, which is defined as the maximum amount of parallel executable instructions that can be extracted from a sequential application by the compiler. Research has shown that most of the existing applications provide a maximum parallelism of three to four instructions per clock [77]. The remaining two operating principles are scaling the number of cores within a processor as well as scaling the number of processors within a system. Unfortunately, current architectures are not capable to scale well in these regards. This is the main issue that needs to be addressed in order to improve performance, wherefore, in this thesis we discuss all aspects that lead to a better system scalability.

1.1 Limitations Of Scalability

The technique of increasing the clock frequency to boost performance has become impractical. Chips have reached a critical power consumption that does not permit a further increase of clock speeds. In order to understand the factors that lead to the power wall it is required to analyze its main contributor which is the dynamic power consumption of transistors,

$$P_{(d)} = C \times V^2 \times f$$

where C represents the load capacity, V the supply voltage and f the processor frequency. In the past, every transistor generation allowed scaling down the operating voltage which helped to compensate the increased power consumption of higher clock rates. By reducing the voltage in every transistor generation by approximately 30%, a 50% decrease of $P_{(d)}$ could be achieved. By introducing additional power saving techniques, the clock frequency could be increased although the transistor count doubled every generation. At the beginning of the current millennium, however, it became evident that voltage scaling would slow down significantly as a minimum amount of electrical charge (the number of electrons stored in a transistor) is needed to guarantee the stable operation of a circuit. As the transistor count and, thereby, the power consumption still doubles every two years following Moore‘s Law, clock frequency is even likely to decline in the future.

A feasible solution to this problem is increasing the number of cores within a processor while decreasing the size and individual performance of each core. The motive behind this approach is that many small cores provide a better total performance than a small number of large cores. An explanation for this behavior is given by Pollack‘s rule [23] which states that the performance increase is roughly proportional to the square root of complexity. As an example, adding a superscalar out-of-order execution capability to a core may increase its size by 100% while increasing its performance by only 40%. Although, Pollack’s Rule may be oversimplified as microarchitecture improvements certainly exhibit different ratios of performance increase per silicon area, it has proven applicable for most architectures in the past [23]. The paradigm shift towards many-core

architectures that utilize a large number of small cores is further supported by the Berkeley View on "the Landscape of Parallel Computing" [9] which expresses the view of some of the most distinguished computer architects towards future computing systems. In this report, the group predicts that upcoming computer systems will feature a heterogeneous design consisting of thousands of cores and accelerators.

Critics of the many-core approach refer to Amdahl's Law [7] which explains the inability of many applications to perform well on said architectures. Amdahl's Law defines a limit for the parallel speedup that can be achieved with many-core architectures,

$$Parallel\ Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

where P represents the percentage of the application that can be executed in parallel and N equals the number of available processor cores. Figure 1-1 lists the parallel speedups for different applications plotted against different numbers of cores. It shows that many applications cannot benefit from multi-core and even less from many-core architectures. Although, Amdahl's law held, in terms of that present architectures have never contained more than a handful of cores, Gustafson insists in "Reevaluating Amdahl's Law" [62] for many-core architectures. He arguments that those architectures won't be utilized to accelerate present applications but instead to increase the problem size, which automatically leads to a higher degree of parallelism. Hill and Marty emphasize the importance of finding more parallelism in applications and to accelerate the sequential parts of an application to address "Amdahl's Law in the Multicore Era" [67].

Considering the increasing amount of cores and functional units on microchips, the interconnect structure between the components is getting increasingly important. As the sequential part of applications is mostly determined by thread synchronization overhead, it is mandatory to focus on optimizing the communication hardware. Recent architectures have applied networks on-chip (NoCs) to address this problem. However, designing efficient and fast NoCs is a complex and challenging task. The design space for NoCs is huge as every application has different requirements. To address this important challenge a NoC design framework has been developed that will be presented in this work.

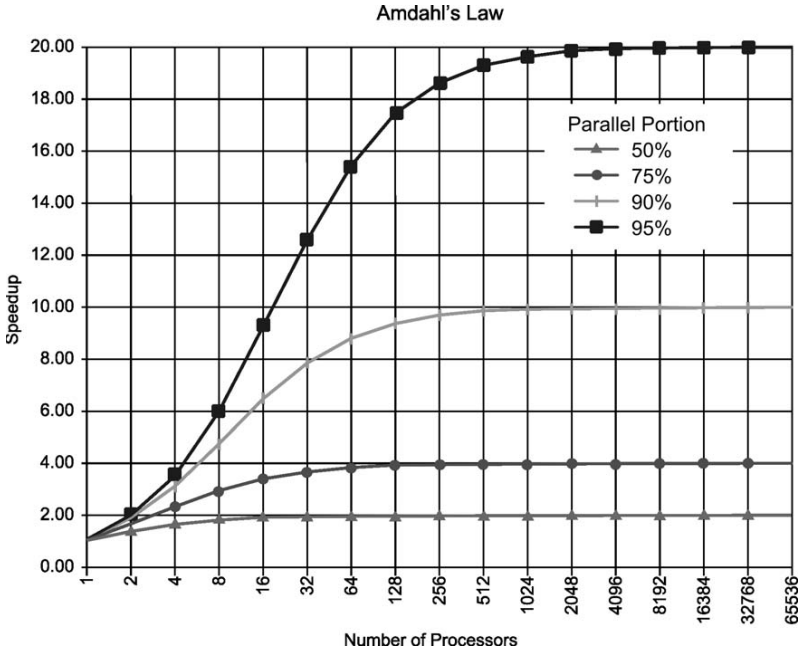

Figure 1-1: Amdahl's Law limits the parallel speedup[1]

The limitations that Amdahl's law imposes on multi-core architectures are also valid for multi-node systems. Multi-node systems represent a well known technique for increasing processing power by aggregating several nodes into compute clusters. While providing a tremendous processing capability, they even more depend on the ability to extract parallelism from applications. The network protocols that are applied for internode communication within the cluster add a significant amount of overhead which increases the cost of communication. In order to utilize future Exascale supercomputers efficiently it will be mandatory to exploit more fine grain parallelism which requires a considerable

1. *Source: Wikipedia, Wikimedia Commons, licensed under CreativeCommons-License by Daniel, URL: http://creativecommons.org/licenses/by-sa/2.0/de/legalcode*

improvement of the network interconnect. In particular, the optimization of communication latency will be of paramount importance as it enables tightly coupled systems in which threads can synchronize their results much more efficiently. To address this issue we have developed the Tightly Coupled Cluster (TCCluster) mechanism which can improve latency by an order of magnitude compared to traditional approaches. The key contribution of our approach is to virtually move the interconnect into the processor which renders any intermediate bridging and protocol conversion unnecessary. In addition, the approach is very cost and power efficient as it reuses already available resources within the processors. As a result, TCCluster can increase the scalability of a multi-node system considerably.

The fastest processor is unable to deliver adequate performance if it cannot be provided with sufficient data. Therefore, a key challenge of future high performance computer systems will be to scale the performance of the processor-memory interface (PMI). The PMI is important as even The PMI has been studied extensively [27][82] to address the divergence of the processor and the memory latency, also referred to as the memory wall which emerged in the 1980'ies [133]. The solution to this problem included the introduction of a memory hierarchy [57] with different levels of caches that can absorb the high latency of the slow DRAM based memory. The technique required increasingly large caches, however, it worked well until recently. Now, computer system architects are starting to face a new challenge while this time, memory capacity and bandwidth appear to be the limiting factors for increasing the processor performance.

Memory bandwidth is determined by the amount of processor pins that are dedicated to the PMI and the data rate. Current architectures already need to allocate a large amount of resources for the memory interface, for example the AMD Magny Cours processor implements four 64-bit memory controllers that consume more than 50% of the package pins. It is expected that the pin count at the socket level will increase only moderately [75] and that increasing the data rate of such wide memory interface will be problematic due to the power consumption of the PMI. By now, the memory controllers in the AMD Magny Cours processor already consume 15 Watts, in addition to the power consumption of the individual DIMMs.

An important contribution of this dissertation, therefore, is the presentation of a technique that addresses the memory bandwidth and capacity problem. Our proposal utilizes a novel low pin-count, high frequency transceiver based interface to attach the memory to the processor. The approach increases the bandwidth per pin ratio significantly and, furthermore, supports hierarchies of memory which enable to scale the amount of supported DIMMs, effectively addressing the memory capacity problem.

1.2 SCALABLE COMPUTER SYSTEMS

Design for scalability represents the novel paradigm for future computer systems. As predicted in [85], upcoming Exascale systems will require to execute one billion of threads concurrently. Such a degree of parallelism poses extensive demands on the scalability of the processor microarchitecture, the inter-node system architecture and the interconnection network. Furthermore, a software environment is required which can extract fine grain parallelism from applications and map it to the many resources provided by the hardware architecture. Figure 1-2 shows the different software and hardware layers that compose a scalable computer system. Although, this work focuses on the three lowest layers, an optimal scalability and system performance can only be achieved by looking at the complete system. Therefore, an overview of the tasks that are involved in the different layers and the challenges that need to be addressed therein will be presented.

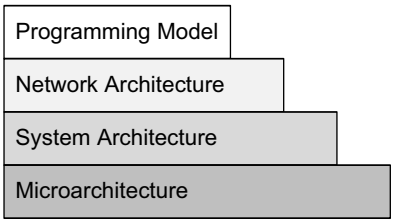

Figure 1-2:Layers within Scalable Computer Systems

1.2.1 Microarchitecture

This layer defines the hardware architecture on the chip level. Every application specific integrated circuit (ASIC) represents a complex system that consists of a large number of components. As an example, the microarchitecture of an AMD Opteron processor core is shown in Figure 1-3. The core pipeline operates by fetching instructions from the cache and decoding them into micro operations simultaneously in three decode units. The micro operations are then packed and forwarded to the instruction control unit which dispatches them to the execution units. Instructions are ready to execute as soon as the operands are ready which are retrieved by the load/store unit from the data cache. The

described core communicates with the main memory controller and other cores via an on-chip network implemented through a X-BAR. In addition to the structure on the component level, the microarchitecture also defines the internal capabilities of the modules. In the scope of a processing engine this may include operating principles like branch prediction [120], superscalar processing [121] or simultaneous multithreading [88]. The microarchitecture defines the most important properties of a design including the clock frequency, the power consumption, the degree of parallelism and the size of specific building blocks. On the microarchitecture layer the main future challenge will be to efficiently integrate the ever increasing number of on-chip components. Novel architectures need to emphasize on flexibility to be able to scale the number of components in every generation. NoCs represent a powerful mechanism to address this issue. Therefore, in this dissertation a novel framework for the generation of flexible and scalable on chip networks for general purpose architectures will be presented.

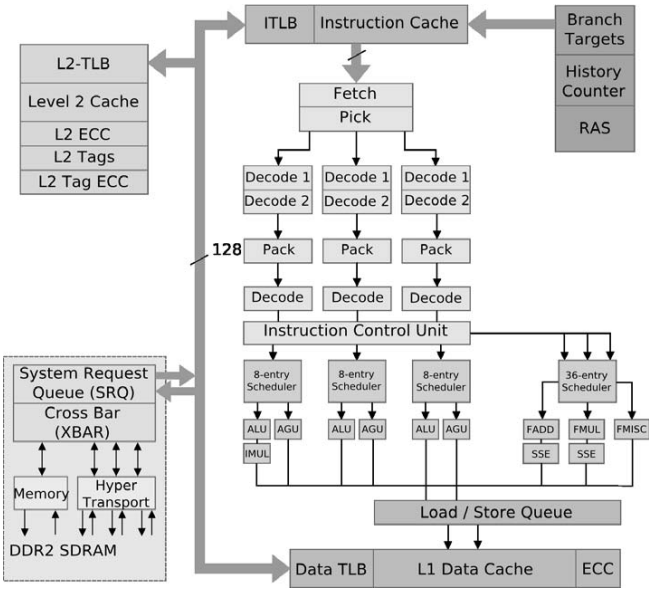

Figure 1-3:AMD Opteron Processor Microarchitecture[1]

1.2.2 Node Architecture

This layer describes the system architecture of a compute node which traditionally consists of a processor, memory and I/O devices. Symmetric multiprocessor (SMP) systems represent an advancement as they aggregate multiple processors into a single system. The most important components defined by the node architecture are the host processor interface and the memory subsystem. While the former enables multiple processors to share memory via a cache coherent interface, the latter specifies the data exchange between the processor and main memory. As the memory access latency has improved at a much slower rate then the processor performance, computer system architects have introduced a memory hierarchy. It consists of very fast components with a limited size (registers) as well as slow components with high capacity (DRAM). The memory hierarchy accelerates performance because of the existing spatial and temporal locality of data within applications. While the concept of a memory hierarchy has reduced the mean access latency of memory it has not solved the memory bandwidth and capacity problem. Upcoming many-core systems require a significant increase of memory capacity which cannot be realized using the current architectures. Extending the scalability of the memory subsystem - a key challenge in future computer systems - is the focus of chapter four within this dissertation. Therein, we introduce a cache coherent yet scalable technique to increase the memory capacity of compute nodes by a significant factor.

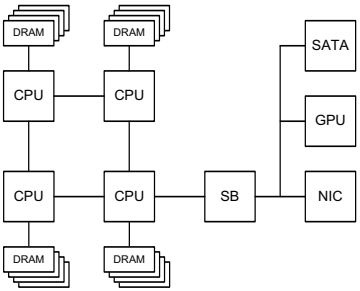

Figure 1-4:Node Architecture

1. *Source: Wikipedia, Wikimedia Commons, licensed under CreativeCommons-License by Appaloosa, URL: http://creativecommons.org/licenses/by-sa/2.0/de/legalcode*

1.2.3 Network Architecture

This layer defines the architecture of the interconnection network that forms multiple compute nodes into a cluster. Today's largest computer systems consist of thousands of nodes interconnected by networks as Ethernet or InfiniBand while future Exascale systems will contain more than 200.000 nodes [85]. Therefore, a scalable and high performance network interconnect is the key component of such systems. The network architecture comprises many aspects including the topology, routing algorithm, fault tolerance and protocol. Key properties of a network are communication latency and bandwidth as they define its performance to a large degree. The network protocols utilized by current supercomputers including Cray's XT6 or Intel and AMD based clusters, require a protocol conversion in the network adapter which causes a significant overhead. To address this issue we propose the Tightly Coupled Cluster (TCCluster) architecture which virtually moves the network controller into the host processor. Our approach avoids protocol bridging and, thereby, improves latency and bandwidth by an order of magnitude which increases the scalability of clusters significantly.

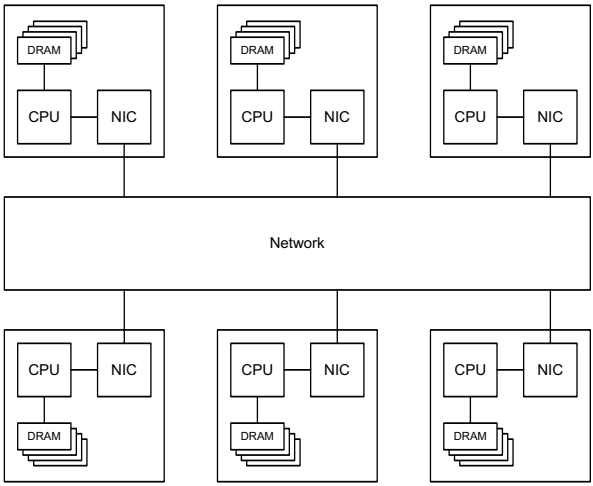

Figure 1-5:Network Architecture for Interconnecting Compute Nodes

1.2.4 Programming Model

The programming model defines a set of techniques to express algorithms in a form that can be executed on computers. The programming model comprises programming languages, compilers and libraries, but most importantly the theoretical concept behind. The goal of a programming model has always been to optimize productivity while providing an adequate performance. The traditional, sequential Von Neumann programming model has not been able to cope with the emergence of multi-core and multi-node architectures. As a result, a large number of novel parallel programming models have been proposed which will be briefly outlined in the following. Although this work does not contribute in the field of programming models, their comprehension is required as they define many properties of the underlying hardware. As shown in Figure 1-6, programming models can be roughly categorized in the groups of distributed and shared memory systems. Furthermore, there exist a number of novel approaches that try to combine the best of two worlds.

In distributed memory systems, threads communicate with each other over a network using message passing. The approach utilizes explicit messages for the data transfer and synchronization which is very efficient as the programmer has total control over the communication. While the message passing model provides a good performance it is the least productive due to the high burden on the programmer. Nevertheless, all of the large distributed clusters are employing message passing for its good scalability.

Shared memory based systems on the other hand aggregate the memory of multiple processors and present it to the application as a global, unified address space. Such systems are convenient to program as all data transfers are implicitly handled by the hardware. To enable such functionality a cache coherency mechanism is required, which keeps the data consistent between all threads and memories. Unfortunately, these coherency mechanisms introduce a lot of overhead which penalizes performance in larger systems. As a result, the shared memory model performs well in systems that employ only a small number of cores, however, it does not provide adequate performance in larger systems due to its limited scalability.

Transactional memory represents another programming paradigm that enables a shared memory address space. However, in contrast to shared memory systems which keep the system consistent at any time whether required or not, the transactional memory model is optimistic. It assumes that most of the times threads will not interfere with each other by accessing shared variables consecutively but not simultaneously. To ensure correct results in cases where this assumption does not hold a roll back mechanism is provided. Therefore, threads are combining multiple memory accesses into transactions which are only committed in the absence of any conflicts. A conflict for example may exist in the case of two threads that write to a single shared variable simultaneously. In this case the latter transaction will fail and need to be restarted and re-executed. As long as the number of conflicts is small this is a very efficient model, especially if specific hardware support is provided.

An approach that blurs the line between distributed and shared memory is the Partitioned Global Address Space (PGAS) model [134]. It presents a single, global, virtual address space to the application which may be physically distributed over a large number of nodes. To permit high scalability no cache coherency is maintained, however, specific hardware that enables efficient synchronization between the threads may be supported. PGAS languages like UPC leverage the fact that not all memory needs to be kept consistent between the threads at all times but only at specific synchronization points defined by the application programmer.

Some high performance systems employ heterogeneous architectures that utilize different types of processors to accelerate specific applications. Such systems include clusters as Roadrunner [12] which combines x86 CPUs and IBM Cell processors as well as systems that employ General Purpose Graphic Processing Units (GPGPUs). Programming such hybrid systems is very challenging and a large number of software approaches have been proposed [90][95] to handle this additional complexity. While current GPGPU systems only achieve an efficiency of around 50%, hybrid accelerator based architectures may be the only solution to constrain power consumption in future Exascale systems.

It remains to be seen which programming models can prevail in the future. The transactional memory and PGAS programming model have not yet found acceptance. The popular shared memory programming paradigm will only persist if new cache coherency schemes can be developed that provide a better scalability, which is doubtful. Message Passing schemes provide the necessary scalability, however, they are inflexible and currently do not support hybrid systems efficiently. For the near future hybrid programming models that use message passing for node to node communication and shared memory techniques as OpenMP for the internal data transfers represent the best solution.

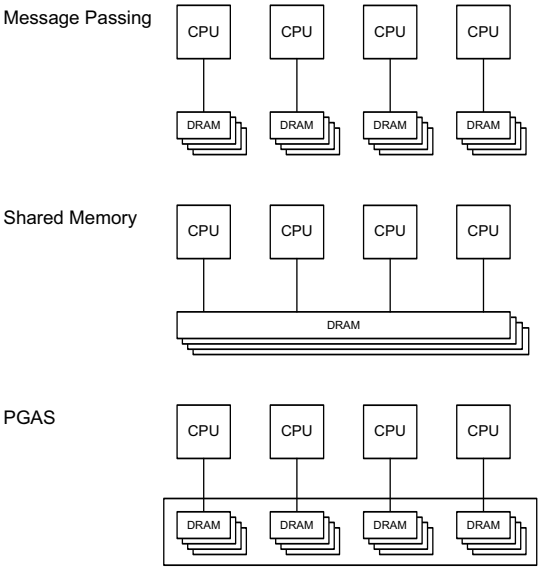

Figure 1-6:Programming Models

1.3 DISSERTATION METHODOLOGY

This work presents three novel approaches to enhance the scalability of future high performance computing systems. The dissertation is, therefore, partitioned into three main sections that describe the individual techniques. All main chapters exhibit the same outline, starting with an introduction and a detailed design space exploration which outlines the scope of the section. This part is in each case followed up with the implementation, and evaluation section as well as a subsequent conclusion. This consistent outline facilitates the orientation and comprehension.

Throughout this work we emphasize the importance of a thorough evaluation of the developed techniques. Each hardware mechanism that is presented herein has been implemented in synthesizable register transfer level (RTL) code and evaluated in a real system with the use of Field Programmable Gate Arrays (FPGAs). Each developed code has been verified exhaustively and simulated in a cycle accurate fashion. Although this methodology consumes significantly higher efforts than an abstract simulation based approach it provides considerable advantages. While a prototype implementation delivers results that correctly represent the performance of a real system, simulations always need to use abstracted models which are prone to errors if not correctly designed. Black and Shen showed [11] that microprocessor simulators contain numerous bugs which distort results by 3% to 5% challenging the validity of simulation based approaches. A cycle accurate RTL model in contrast provides exact results. FPGA based prototypes provide another advantage by significantly increasing the verification and evaluation space. Due to the hardware acceleration the test set can be several orders of magnitude larger and by employing the FPGA in a host system, third order effects like the influence of the Operating System can be incorporated.

CHAPTER 1 INTRODUCTION

CHAPTER 2
SCALABLE MULTINODE CLUSTERS

2.1 INTRODUCTION

For many applications, the performance delivered by a single compute node is not sufficient. Applications in the field of weather forecasting, fluid dynamics and military simulations demand for much greater processing power which can only be delivered by supercomputers like the ones in the Top 500 list [126]. More than 80% of these systems are so-called clusters that consist of off-the-shelf compute nodes integrated by a network interconnect, in most cases Ethernet or InfiniBand. While this approach scales well and enables large low cost systems the efficiency is limited due to the high overhead introduced by the network protocol. Therefore, the application performance often equals a fraction of the theoretical peak performance of the cluster.

To increase the efficiency of such systems, network communication bandwidth needs to be improved but even more importantly the latency needs to be reduced. Current compute clusters consist of loosely coupled machines, that communicate using large messages to evade this problem. While this method works for so-called embarrassingly parallel applications that communicate infrequently, it significantly limits the performance of workloads that exhibit fine grain parallelism. To address this issue we are proposing an entirely new architecture called Tightly Coupled Cluster (TCCluster) [97]. The key idea of our solution is to exploit the processor's host interface as a network interconnect. As the host interface resides within the processor, our approach enables tightly coupled systems that can communicate with very high bandwidth and very low latency.

The technique neither applies any modifications to the processor nor does it require any additional hardware. Instead, it utilizes commodity off-the-shelf AMD processors by reconfiguring the HyperTransport host interface as a cluster interconnect. Our approach is purely software based and is completely implemented in BIOS firmware as well as in a Linux kernel driver. In this work, we analyze the programming model and the software stack as well as the logical and physical implementation of such a design. Further questions which are addressed in this work include the routing, addressing and topology schemes that can be used. We present a detailed description of the tasks that need to be solved and provide a proof of concept implementation. The evaluation of our technique is

conducted with the help of a two node TCCluster prototype. Therefore, the BIOS firmware, a custom Linux kernel and a small message library have been developed. We present micro benchmarks that show a sustained bandwidth of up to 3420 MB/s for messages as small as 64 Byte and a communication latency of 240 ns between two nodes outperforming other high performance networks by an order of magnitude.

2.1.1 Motivation

In High Performance Computing (HPC) there still exist Grand Challenges that demand for more powerful, less expensive and less power consuming machines than currently available. Since Roadrunner [12] has broken the petascale barrier in 2008, the HPC industry and scientists from all over the world are competing to build the first Exascale computer. Analyzing the last ten TOP500 lists [126] a trend for HPC platforms can be observed. They are moving from big symmetric multiprocessor (SMP) machines towards large clusters consisting of thousands of interconnected nodes. The main reasons for this trend are that clusters scale better and that they can use commodity off-the-shelf components which provide a much better price/performance ratio than specifically developed supercomputers.

Almost all clusters are x86 CPU based and utilize processor hardware from either Intel or AMD. A cluster consists of several nodes which are comprised of one or more processors, memory and input/output (I/O) devices. To form a cluster out of such nodes a network interconnect is required. The traditional technology is Ethernet which is more and more getting replaced by faster and more efficient interconnects like InfiniBand. While network interconnect technology has been improved significantly over the years in terms of latency and bandwidth, it still represents the bottleneck within a cluster for many applications. For example, one of the fastest currently available implementations, namely the ConnectX InfiniBand adapters from Mellanox [123] achieve a bandwidth of 1.4 GB/s and an end-to-end latency of about 1.4 us. This is only a fraction of the amount that modern processors are able to deliver at their chip interface. In an SMP system, multiple processors are interconnected by their native host interface like HyperTransport (HT) in the case of AMD or Quick Path Interconnect (QPI) in the case of Intel. Such interfaces are

one order of magnitude faster, delivering bandwidths of 10 GB/s and access latencies of as low as 50ns [73].

In an SMP machine, threads can communicate very efficiently via shared memory. This requires a cache coherency mechanism [8] like MESI which guarantees data consistency in the system at all times. While such a coherency model facilitates programmability of shared memory systems it dramatically limits their scalability. The coherency mechanism requires the exchange of cache state information between the processors, which limits the performance gain. The shared memory approach performs well for small scale systems of up to 8 or 16 nodes, however, as the coherency overhead grows with the number of nodes it cannot be applied to large clusters.

To overcome the disadvantages of the current approaches we propose a fundamentally new cluster architecture which we refer to as Tightly Coupled Cluster (TCCluster). TCCluster uses the processor's host interface as an interconnection network leveraging its high bandwidth and low latency characteristics while providing scalability of up to thousands of nodes without cache coherency. The high scalability is achieved by diverting the naturally coherent host interface between the processors from its intended use, by operating it in a non-coherent fashion.

Typical architectures limit the usage of non-coherent interfaces to access I/O devices like southbridges or coprocessors. Our presented technique, however, enables the use of that interface as a cluster interconnect to circumvent the scalability limitation of coherent shared memory systems. In contrast to other architectures, our technique allows to interconnect thousands of machines with the bandwidth and latency characteristics that normally only an SMP can provide.

For the implementation of our ideas we chose the AMD Opteron processor over an Intel based system. The main advantage of AMD is that their processors use HyperTransport (HT) as their host interface which is an open protocol. Furthermore, AMD provides comprehensive publicly available processor documentation in the form of the BIOS and Kernel Developers Guide (BKDG) [2]. AMD also supports the coreboot project [102] which has the goal of developing a generic open source BIOS firmware for x86 platforms.

2.1.2 Related Work

Research on high performance interconnects to enable scalable parallel computer systems has a long history in computer science. Most approaches apply the message passing programming model which uses explicit messages for inter process communication. One of the very first milestones in this area was the *Transputer* [130] developed in the 1980ies. It was specifically designed for parallel systems and combined the processor with four serial links on a single chip. The in-built network allowed for very efficient communication between multiple Transputers which could be interconnected to form a so called *computing farm*. In many aspects our approach is similar to the Transputer approach, however, we have applied it to modern x86 based systems which dramatically differ from the old Transputer design. Since then, processor builders like Intel and AMD have abandoned the idea of incorporating network interconnects directly into their processors, although this would offer a dramatic increase in performance.

In [64] Joerg and Henry analyze the benefits of a tightly coupled processor-network interface which is realized through additional processor registers. Although this approach performs well it requires a modification of the processor hardware. This is well known to be very different endeavour.

More recent approaches like Infinipath [41], Cray's XT3's seastar interconnect [25] as well as our own VELO technique which we propose in [98] focus on optimizing the interface between the processor and the network interface. In all three approaches, AMD's processor host interface HyperTransport is used to reduce latency and to increase bandwidth. Although, HyperTransport allows for a direct connection between the network adapter and the processor without intermediate bridging, the three approaches still have a significantly higher latency than TCCluster due to the latency introduced by the network hardware and protocol conversion. The network interface which currently (2010) can be regarded as the state of the art as it offers very good performance is the ConnectX InfiniBand adapter manufactured by Mellanox. It provides bandwidths of up to 40Gbit per link and a latency as low as 1.4 us [123]. This technology will be used as a baseline for the evaluation of TCCluster.

In addition to the message passing based systems, a lot of research activities focus on scalable coherent shared memory based systems. Such systems offer a very good performance for small systems as they employ a much faster interconnect in the form of the host interface. Their biggest disadvantage, however, is their limited scalability. There exist various approaches which address this issue, like the virtual shared memory architecture by Li [92], or the scalable coherent interconnect by Gustavson [62]. More recent approaches, like Horus [87] and 3-Leaf [1] target modern x86 based systems. They enable large Cache Coherent Non Uniform Memory Architectures (ccNUMA) by extending AMD's HyperTransport protocol. By applying a directory based coherency mechanism they can moderately increase the scalability to 32 nodes.

All approaches focus on either increasing the performance of the network interconnect or increasing the scalability of shared memory systems. Neither has been able to combine the best of the two worlds and to develop a truly scalable tightly coupled architecture. In order to address these shortcomings, we present the TCCluster mechanism which offers the same bandwidth and latency performance of a shared memory based system combined with a much better scaleability in terms of system size.

2.1.3 Background

Although the principles of TCCluster can be applied to other processor families for instance from Intel or ARM, we target AMD processors exclusively. AMD processors offer a significant advantage as they employ HyperTransport as a host interface which is an open protocol specification maintained by the HyperTransport consortium. Therefore, before presenting our approach, an analysis of the AMD Opteron architecture and the HyperTransport protocol will be provided. The current AMD Opteron processors, named family 10th [33][2], implement several components on a single chip. In addition to the processing elements named cores, a processor node comprises memory controllers, several levels of cache memory, up to four outgoing HyperTransport links and a crossbar that interconnects the different components.

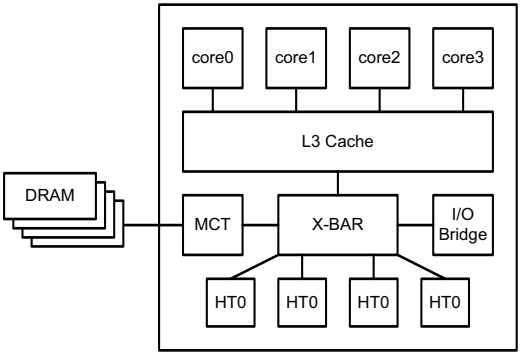

Figure 2-1:AMD Opteron Chip Architecture

Figure 2-1 shows the AMD Opteron chip architecture of the *Shanghai* processor that implements four cores that have their individual L1 and L2 caches and a large shared L3 cache, utilized for inter-core communication. Furthermore, the processor chip contains an I/O bridge that converts between coherent and non-coherent HyperTransport packets, a DDR2 memory controller (MCT) and four HyperTransport links. Those can be used to communicate with off-chip devices as other processors or as I/O devices.

HyperTransport is a packet based protocol that offers low latency (approximately 50 ns per hop) and a unidirectional bandwidth of 12.8 GByte/s per link. It has low overhead and defines fault tolerance mechanisms on the link level. HyperTransport packets consist of a command header and payload data. The command header contains information about the type of the packet, the amount of appended data and the destination of the packet. HyperTransport defines a credit based flow control mechanism to avoid buffer overflow and to guarantee reliable transmission. Packets have a maximum size of 64 Bytes and travel in different virtual channels to avoid deadlock. Packets are distributed into virtual channels based on their message class. Write requests and broadcasts travel in the posted channel while read request packets travel in the non-posted channel. The split-phase read response is transmitted in a separate packet, utilizing the response channel. HyperTransport links can be operated either in a coherent (cHT) or non-coherent (ncHT)

fashion. In a symmetric multiprocessor node the individual processors always communicate over coherent HyperTransport. Only I/O devices as accelerators and network interface cards are connected using non-coherent HyperTransport. Coherent links add a fourth traffic class and hence, virtual channel to transmit so-called probes. Cache coherency is required for multi socket systems which are available in two, four and eight socket configurations. The system shown in Figure 2-2 combines the physical memory modules which are connected to each processor to form a single shared memory address space. Such a system requires a mechanism like the modified, exclusive, shared, invalid (MESI) protocol to ensure cache consistency. The cache consistency guarantees that data, stored at a specific address which is shared by multiple cores, has the same value in every cache at any point in time. Every time a data value is modified in a cache or loaded from main memory the other cores that participate in the coherent domain have to be informed and probed for a response. The transaction can only be completed if all nodes have responded to the probing. While this technique allows multiple nodes to share a common memory address space, its scalability is limited. By increasing the number of nodes, the number of probe messages is increased proportionally. This negatively effects bandwidth and latency as the last incoming response is pivotal. Furthermore, as the Opteron processor only contains four links, fully connected systems are only possible for two and four processor configurations. Larger systems have to utilize multi-hop topologies which increase the latency even further.

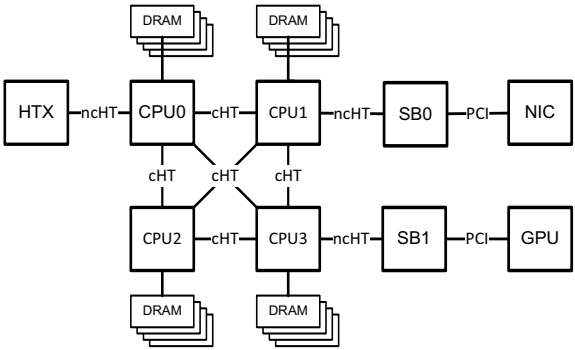

Figure 2-2:AMD Based Multiprocessor System

Non-coherent HyperTransport is used to integrate I/O devices into the system. This includes southbridges or directly connected accelerators and network devices. In [99] we propose “A HyperTransport NIC for Ultra-low Latency Message Transfer” that exploits the direct, low latency connection to the processor. The non-coherent HyperTransport protocol defines a reduced set of commands which are used to transport data via Programmed I/O (PIO) or Direct Memory Access (DMA). A typical system that is comprised of multiple CPUs, two southbridges and a HyperTransport Extension (HTX) slot is shown in Figure 2-2. All four CPUs are directly connected via cHT links. In addition, the system features two southbridge chips that are connected to the CPUs via ncHT links. These chips allow to attach PCI-Express, USB and SATA I/O devices to the system, for example Graphic Processing Units (GPUs). The HTX slot is a PCI like connector that can be used to attach ncHT devices with very low latency directly to the CPUs without any intermediate bridging.

2.1.4 Approach

This paragraph will introduce the basic operation of TCCluster. The design is based on the following observations.

- Coherent shared memory systems do not scale beyond a restricted number of end-nodes efficiently
- Better exploitation of parallelism in very large multi-node systems can only be achieved with fine grain communication schemes
- To support fine grain communication the processor and the network interface need to be tightly coupled with very low latency
- Better coupling can only be achieved by removing the manifold protocol bridging that takes place in current computer architectures

A self-evident solution is to integrate the network interface on the processor die. However, due to extra costs of this solution there exists no processor manufacturer who supports this approach. In this work we introduce a solution that can be implemented using current processor architectures without any changes to the hardware. The TCCluster approach exploits the native processor host interface which already exists for other

purposes and exploits it as a network interface. This technique allows to reuse the protocol and transceiver hardware within the processor chips, thereby, being the most cost and power efficient solution. The omittance of I/O bridges like PCIe and network protocol chips virtually moves the network controller into the processor and is expected to reduce the communication latency by an order of magnitude.

The scalability of the system is accomplished by abandoning cache coherency. TCCluster introduces a non-coherent globally shared address space in which packets can be sent by using remote store operations. Packets are routed through the network based on their address which is embedded in the packet header. The complete mechanism is a pure firmware and software based approach. To discuss the requirements of TCCluster and to determine the issues which have to be solved to enable a successful implementation a design space exploration is required which will be provided in the next paragraphs.

2.2 Design Space Exploration

Unlike previous approaches our presented solution does not require any intermediate NIC hardware and, therefore, offers superior performance at a reduced cost. Connecting multiple processors directly with HyperTransport, however, raises a number of challenges. A design space exploration has to be performed which analyzes the programming model, the routing mechanism, the topology and fault tolerance requirements of TCCluster. In addition, it will be discussed how a physical implementation can be realized. A further section examines the limitations of TCCluster and compares the technique to other network interconnects.

The design space exploration will enable to define the principle properties of TCCluster. This is required to develop a prototype implementation. It will be shown that a proof-of-concept can be realized that consists of two discrete Opteron nodes that are interconnected by a TCCluster interconnect. Custom BIOS firmware, a modified Linux kernel and specific driver software has been developed to realize such a system.

2.2.1 Programming Model

To develop a suitable programming model for TCCluster, the capabilities of a non-coherent HyperTransport link have to be analyzed. HT defines three main types of message transactions: posted writes, non-posted reads and responses. While posted writes are completed from the sender's perspective as soon as they are transmitted, non-posted requests require some sort of book keeping. In HyperTransport read requests are realized using split phase transactions in which the read requests are routed identically to write requests, however, the response to the request is not. Each read request creates an entry in the response matching table located in the northbridge and receives a tag. A matching response will carry the same tag and can be thereby routed without having to carry an address. However, the number of these tags is limited to 32 and, furthermore, they are always mapped to a specific NodeID. Therefore, TCCluster does not support to route responses, restricting the communication to writes. TCCluster will, therefore, utilize a remote store programming (RSP) model.

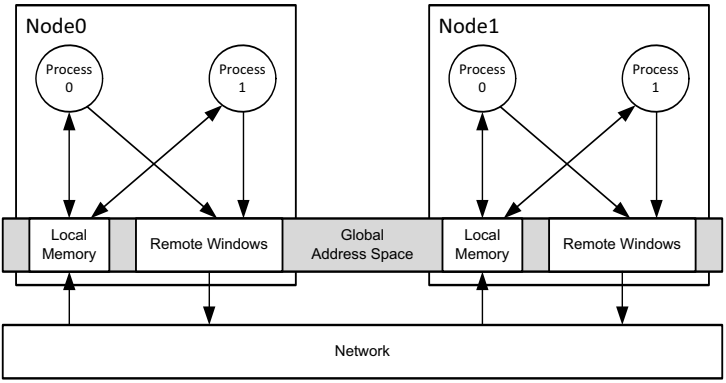

Figure 2-3:Remote Store Programming Model

Remote Store Programming

In the remote store programming model, applications can directly access remote nodes with store instructions in addition to the regular load and store instructions that target local main memory. To enable efficient communication, remote stores need to be supported in hardware directly which is accomplished by TCCluster. As remote load operations are not supported, the communication scheme is always two-sided. Figure 2-3 shows four processes on two compute nodes that use a partitioned global address space for communication with remote stores. Furthermore, parallel programming paradigms can be distinguished by analyzing three important characteristics [70]: the process model, the communication mechanism and the synchronization mechanism.

The process model. In RSP, each process has its own local, private memory. A process can make certain parts of its memory, a so-called *window*, accessible to remote processes. A remote process only has write permission to these windows, the local process can both read and write the remotely writable window. No cache coherency is applied to keep the memory consistent between different processes.

The communication mechanism. In RSP, processes communicate by directly writing into physical remote memory using the standard store instruction. A kernel level security mechanism guarantees that only specific memory windows are remotely writable. A process receives data by reading data from remote memory windows that reside in local physical memory. Remote stores can be reordered in the processor pipeline.

The synchronization mechanism. In RSP, synchronization is implemented through software barriers. The required hardware function is a flush instruction which enforces that all local and remote stores have been completed. The network needs to support in-order delivery of barrier synchronization messages.

Given this description, the RSP model offers the following attributes:

- By ensuring that no process reads remote memory and by avoiding coherency overhead the RSP model is very lean and efficient
- Programmers are encouraged to exploit spatial locality by being forced to distribute data before processing it in local memory
- Communication and synchronization is explicit giving programmers a good control of the system
- Remote reads or gets are not supported. This minimizes read latency which is required for efficient execution as delayed read responses stall the CPU pipeline
- Reordering of writes optimizes performance
- RSP can be used as a basis for other programming paradigms like message passing

Message Passing on Top of RSP

RSP primitives can be used as a foundation for other programming paradigms like message passing. To support a message massing interface (MPI) protocol like MVAPICH [106] an underlying application programming interface (API) is required that enables sending and receiving of messages. In TCCluster communication between processes is carried out through unidirectional connections using *send* and *receive* buffers. The receiver allocates its receive buffer first and communicates the physical address of said buffer to the sender. The sender then allocates a send buffer which points to this particular physical address. Sending is performed by writing to that specific address, resulting in a

HyperTransport packet that is delivered to the receive buffer on the remote node. The send and receive buffers are managed as ring buffers. To prevent a buffer overflow the read and write pointers of the receive buffer are periodically communicated to the send node using a lazy pointer scheme. Receiving of messages is implemented by polling the corresponding address on the receive node. As soon as a change in main memory is observed the API can extract the data from the buffer, copy it into local main memory and increment the read pointer. There exists no hardware support for sToring messages at the correct location within the receive buffer, hence, the sender is responsible for writing the packets to consecutive addresses. Therefore, it is impossible to share receive buffer queues between multiple endpoints which may limit scalability for very large systems. However, using 4 KB receive buffers, 65 K endpoints can be supported with a memory footprint of a mere 256 MB. Remote stores can also be utilized to implement one-sided rendezvous like communication. This approach imposes an initialization penalty due to the registration overhead of the connection, however, it offers zero-copy messaging as the data is directly delivered to the correct memory location. The receiver can work with the data in place without another copy transaction. As a result, two-sided send-receive communication is applied for small messages, in which larger messages are transmitted using a one-sided approach. This enables high performance message passing implementations.

2.2.2 Routing

The routing algorithm defines the path which is followed by each message sent throughout a network. Routing algorithms are tightly coupled to the underlying network topology and define various properties of a network including packet latencies, congestion behavior and deadlocks. There exists an almost endless amount of routing algorithms while a good summary is given by Duato and Yalamanchili in [45] and Dally and Towles in [39]. To come up with a routing algorithm for TCCluster only a subset of the possible approaches needs to be analyzed as the capabilities of the hardware represent a limiting factor. Our approach uses off-the-shelf processors, hence, it is necessary to choose a mechanism that is compatible with it. In the following, an analysis of the Opteron northbridge and its routing capabilities will be provided.

Routing Capability of the Opteron Processor

The Opteron northbridge utilizes different table based interval routing schemes (IRS) to deliver packets within the HyperTransport fabric. In IRS multiple endpoints are aggregated into an interval and mapped to a specific outgoing link. This enables scalable systems as the interval routing table only requires an entry per interval in contrast to routing tables which contain an entry for each endpoint in the network. In the Opteron architecture, depending on the packet type, one out of three tables is queried to look up the outgoing link for a specific packet. All memory accesses initiated by processor cores are routed in the coherent HyperTransport fabric and directly carry the target node identifier (*tgtNodeID*) in their packet header. In such cases the *nodeID-to-link* table returns the required information.

Memory writes from I/O space or processor writes to memory mapped I/O are sent as non-coherent HyperTransport packets. Non-coherent devices which may either be the source or the target of such packets do not carry a nodeID and thereby utilize another routing scheme. In principle, an Opteron system supports an infinite number of non-coherent devices which are accessed through the system's memory address space that offers a total size of 256 Terabytes. Non-coherent HyperTransport packets contain the target memory address in their packet header. In every routing stage a table lookup is performed which compares the address against eight different memory address ranges that can be defined individually. Each address range points to a tgtNodeID which in return can be used to determine the outgoing link.

The third type of packets are non-coherent read responses which utilize a source tag based routing mechanism. In HyperTransport read requests are implemented using split phase transactions. The read request travels in the non-posted channel and carries a destination memory address that enables it to be routed identically as posted requests. In addition, each read request generates an entry in the source tag matching table in each routing stage. The corresponding read response carries the same tag information and by performing a table lookup the outgoing link can be determined which, in return, delivers the response to the originating node.

The Opteron northbridge also supports translation of coherent packets into non-coherent packets and vice versa using its I/O bridge. If the packet was received from an IO link and targets main memory, it is forwarded to the I/O bridge which converts the non-coherent packet into a coherent packet. A coherent packet is either forwarded to the on-chip memory controller or to an outgoing HyperTransport link in the case of the physical memory address residing on another node. Coherent packets that target an I/O link are also forwarded to the I/O bridge which converts it into a non-coherent packet and forwards it to the corresponding I/O link. Non-coherent packets originating at an I/O link that target another I/O link are simply forwarded without bridging. As TCCluster only supports non-coherent posted write transactions only the according routing mechanism is supported. Consequently, TCCluster uses table based interval routing supporting up to eight table entries.

Interval Routing Scheme

Interval routing was introduced by Santoro and Khatib [117] and generalized by Leeuwen and Tan [91] to be applicable to arbitrary networks. Its main advantage over traditional table based routing is the compactness of its routing tables. In the interval routing scheme (IRS) all endpoints that can be reached over a link are grouped together in an interval. The routing tables, therefore, require k entries where k is the number of intervals mapped to an outgoing link.

In the following we will provide some definitions concerning interval routing. The underlying network topology for interval routing can be described as a graph

$$G = (V, E),$$

V being the set of nodes in the graph and E the set of bidirectional edges between the nodes. $(x, y) \in E$ and $(y, x) \in E$ are defined as the unidirectional arcs between the nodes x and y. The interval labelling scheme (ILS) defines a one-to-one labelling for each $v \in V$. For each $(x, y) \in E$ there exists an arc-labeling which defines the nodes that can be reached over this arc. As all intervals are consecutive, interceptions are only supported by mapping multiple intervals to an arc. Figure 2-4 shows a node labeling scheme and an arc labeling scheme for a simple network.

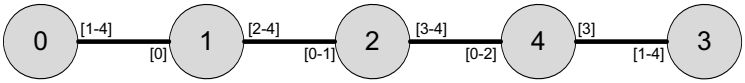

Figure 2-4:Interval Labeling Scheme for a Simple Network

It is interesting to analyze the topologies that can be effectively supported with interval routing. Although, Leeuwen and Tan have shown that there exists a valid IRS for every network topology [91] there is no hope to determine a compact and optimal routing scheme for arbitrary topologies. The compactness of an IRS is defined as the maximum number of intervals that are mapped to an arc. A routing scheme with *k* intervals per arc is denoted as *k*-IRS. An IRS is optimal if all routes in the network use a shortest path. Another important property is the linearity of an IRS. In a linear IRS (LIRS) all arcs may only define linear routing intervals that do not contain any cycles. The interval [5, 1] for example would not be linear as it contains a wrap-around. IRS can be, furthermore, classified into strict and non-strict schemes. An IRS is strict (SIRS) if no interval assigned to the edges of a node contain the label of that node. For this reason, the ILS shown in Figure 2-4 is not strict as the interval of node 3 includes itself. Depending on linearity and strictness, hence, there exist four different IRS variants: *k*-IRS, *k*-LIRS, *k*-SIRS and *k*-SLIRS. Furthermore, we can denote optimal routing schemes with an appended "*" which leads to *k*-IRS*, *k*-LIRS*, *k*-SIRS* and *k*-SLIRS*. In the following, topologies are analyzed that can be mapped to TCCluster and which support a 1-SLIRS* IRS. Therefore, a summary of related work will be given that discusses the application of IRS to different popular topologies. A more detailed survey on interval routing can be found in [55].

Meshes

The *k*-ary *n*-dimensional Mesh is a direct network that has *n* dimensions and $k_0 \times k_1 \times \cdots \times k_{n-1}$ nodes, k_i nodes along each dimension *i*, where $k_i \geq 2$ and $0 \leq i \leq n-1$. Each node is identified by n coordinates $(x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ where $0 \leq x_i \leq k_i - 1$ for $0 \leq i \leq n-1$. Two nodes *x* and *y* are neighbors if and only if $y_i = x_i$ for all *i,* $0 \leq i \leq n-1$, except one, *j*, where $y_j = x_j \pm 1$. Thus, nodes have from *n* to 2*n* neighbors, depending on their location in the Mesh and, hence, the Mesh in not regular

[44]. Meshes are a common topology used in many commercial network implementations for example in Intel's Paragon [50] machine.

Meshes support interval routing schemes, in particular, Bakker, Leuween and Tan prove in [10] that *n*-dimensional Meshes belong to the class of 1-SLIRS*. Furthermore, it is interesting to analyze the deadlock-freeness of IRS for Meshes. Zerrouk proved in [135] that every 1-SLIRS* also supports a deadlock-free 1-SLIRS*, however, only if the network supports multiple virtual channels. TCCluster does not provide support for changing the virtual channel within a route, but fortunately, in [136] it is shown that for *n*-dimensional Meshes there exist 1-SLIRS* that are deadlock-free using dimension order routing with a single virtual channel. This is an important result as it implicates that there exists a scalable topology for TCCluster that provides minimal routing paths as well as deadlock-freeness using the interval routing technique.

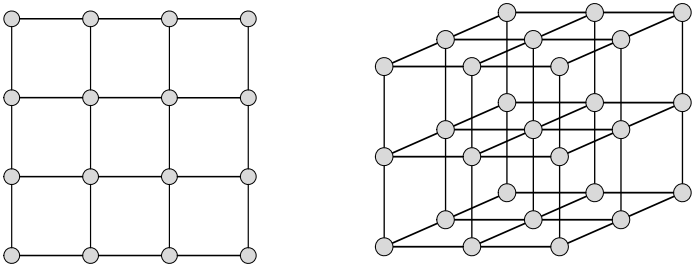

Figure 2-5:a) 4-ary 2-dimensional Mesh, b) 3-ary 3-dimensional Mesh

Hypercubes

A Hypercube is an n-dimensional Mesh with the additional properties of being regular and symmetric. It is also referred to as the *binary n-cube*. An n-dimensional Hypercube contains 2^n nodes and has $2^{(n-1)} \cdot n$ edges. The scalability of Hypercubes is limited as each dimension adds an additional link to each node. Nevertheless, there exist a range of successful implementations, most notably the Connection Machine [68] by Thinking Machines which interconnected 65,536 processors using a Hypercube topology. The Hypercube offers grateful properties regarding interval routing as it belongs to the class of

1-SLIRS* [51]. Furthermore, there exist deadlock-free routing schemes that do not require multiple virtual channels [136] for the Hypercube topology. The *p*-cube routing algorithm [56] and the *e*-cube algorithm both provide minimal, deadlock free routing in hypercubes. In both mechanisms packets are first routed in the lowest dimension, then step-by-step in the next higher dimensions and only in the last step routed in the highest dimension. These properties turn the hypercube into another excellent topology candidate for TCCluster.

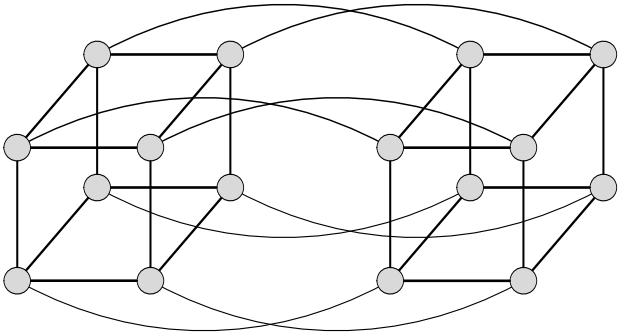

Figure 2-6:4-Dimensional Hypercube

Cubes

A *k*-ary *n*-cube or Torus is a Mesh with additional wrap around links. Through the addition of these channels the cube gains regularity and symmetry. Every node in a cube has n neighbors if $k = 2$ and $2n$ neighbors if $k \geq 2$. A *k*-ary cube with a single dimension is referred to as a ring. A two-dimensional *k*-ary cube is also known as a grid. A *k*-ary *n*-cube contains k^n nodes. The additional wrap around links double the bisection bandwidth in comparison to Meshes and divide the maximum diameter of the network by two.

Tori are used in many commercially available networks including the Cray XT supercomputer family. Cray's proprietary Seastar [25] interconnect provides six links per node to enable highly scalable 3-dimensional cubes. Another prominent example of the 3D-Torus topology is the IBM bluegene [4]. Tori are particularly interesting as, from a deployment perspective their complexity is comparable with Meshes, however, they provide a significantly better bisection bandwidth and lower diameter.

In contrast to the previous topologies, the IRS properties of Tori depend on k. For $k < 5$ the k-ary n-cube belongs to 1-SLIRS*, while for $k \geq 5$ the k-ary n-cube belongs to 2-SLIRS* [51]. While the overhead of two routing intervals per arc is in principle tolerable, for TCCluster it represents a limitation. As TCCluster only supports eight address intervals, for $k \geq 5$ the maximum number of links that can be supported per node is four, hence TCCluster would be restricted to a 2D-grid. By restricting k to 4, 4-ary 4-dimensional cubes with a maximum of 256 endpoints can be realized. For this size the cube represents a feasible alternative for TCCluster thanks to its low diameter and high bisection bandwidth. Another problem of large k-ary n-cubes is that dimension-order routing is not deadlock free anymore as the wrap around links introduce cycles even for packets that travel in a single dimension. For n-cubes with $k > 4$ deadlock freeness can only be achieved through the introduction of virtual channels as proposed by Dally [38] or by applying a turn model based routing algorithm as introduced by Glass [56]. In the turn model, certain turns within the network are prohibited which break the cycles. The approach does not require additional links nor virtual channels, however, it sacrifices the shortest path for some routes which leads to a non minimal routing scheme. In the virtual channel approach packets are assigned a new virtual channel as soon as they pass a wrap around link. As packets in different virtual channels can overtake each other, the cycle is broken. However, the approach is not feasible for TCCluster as in HyperTransport packets cannot change their virtual channel whilst travelling the network.

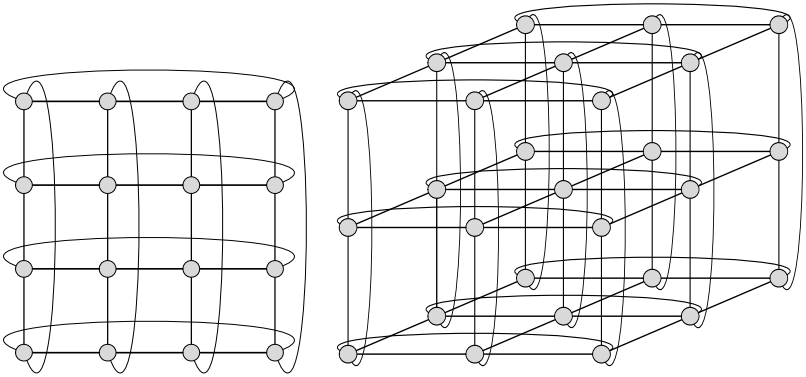

Figure 2-7:a) 4-ary 2-dimensional Cube, b) 3-ary 3-Dimensional Cube

Other Direct Topologies

Many other topologies have been proposed. However, none of them appears to be a good fit for TCCluster. One popular topology is the tree. It defines a root node with *n* descendant nodes which in return can have further descendants. A node with no descendants is called a leaf node. A tree with a tree-width of 4*k* belongs to *k*-SLIRS as shown by Bodlaender et al. in [20]. Trees are deadlock free as they contain no cycles and for balanced trees the distance from every leaf node to the root is identical. The drawback of trees is that the root node and adjacent nodes become a bandwidth bottleneck as all traffic from the left to the right side needs to pass the root node. This disadvantage can be eliminated by Fat-Trees which utilize multiple or higher bandwidth links to connect the root node. However, as Opteron processors only provide a fixed amount of links with equal bandwidth capacities, Fat-trees cannot be supported. Hence, tree topologies are not the preferred choice for TCCluster.

A topology that supports a very high bisection bandwidth and a minimal network diameter is the fully connected graph. However, in this topology the routing tables get very large as one entry for each node in the network is required. It is obvious that the topology is not scalable due to the amount of links required for large networks. Another

topology that seems promising is the star Mesh. It is based on a 2D-Mesh topology extended with additional diagonal links to reduce the network diameter and to increase bisection bandwidth. However, as the star Mesh is $\notin$ 1-SLIRS it is an unsuitable topology for interval routing [55].

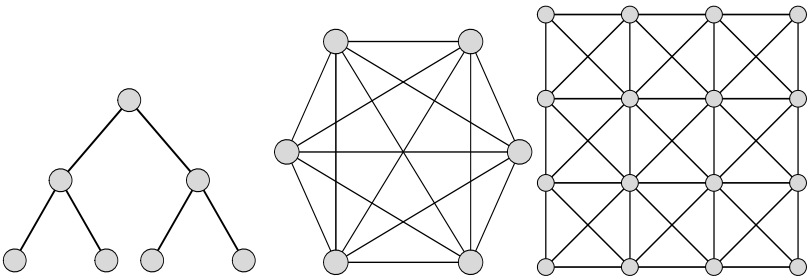

Figure 2-8:a) Tree, b) Fully-connected Graph, c) Star Mesh

Interval Labeling Schemes

One challenge of interval routing is to label the nodes correctly to enable all messages reaching their destination in finite time without any cycles in their route. For *n*-dimensional grid networks, which are a good fit for TCCluster, Leeuwen *et al.* provide an algorithm that generates a valid, compact and optimal ILS [91]. This ILS, however, ignores deadlock behavior. Therefore, we propose a modified interval labeling algorithm that is based on dimension order routing and which eliminates deadlocks completely. Let *G* be an *n*-dimensional Mesh with the dimensions $D_0 \cdots D_n$ and the elements $E_0 \cdots E_n$ within these dimensions. Label the nodes consecutively by traversing the dimensions in reverse order starting with D_n. Each time dimension D_i has been traversed completely, proceed in the next dimension D_{i-1} and continue labeling the nodes consecutively. Figure 2-9 shows valid interval labelling schemes produced with the algorithm that are deadlock free for *n*-dimensional order routing. From the labeling schemes above the interval routing tables for each node can be deduced. The routing tables for the network shown in Figure 2-9 a) is presented in Table 2-1. For each node, the table lists the four links, denoted by X+, X-, Y+, Y-, and the intervals that are mapped to these links.

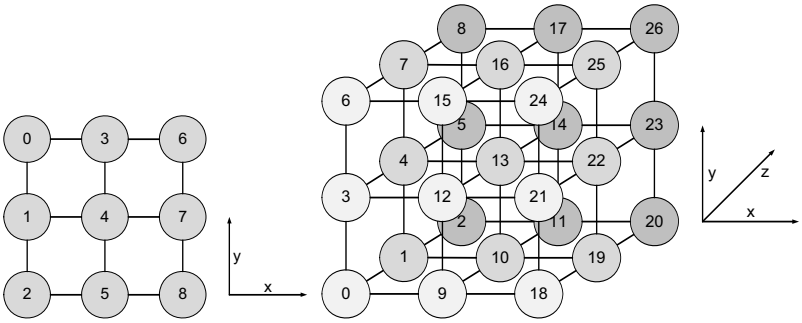

Figure 2-9:a) 2D-Mesh labeling scheme, b) 3D-Mesh labeling scheme

Table 2-1: Interval Routing Table for 3x3 Mesh

Channel	Node	Interval	Node	Interval	Node	Interval
X+	0	3-8	3	6-8	6	
X-	0		3	0-2	6	0-5
Y+	0		3		6	
Y-	0	1-2	3	4-5	6	7-8
X+	1	3-8	4	6-8	7	
X-	1		4	0-2	7	0-5
Y+	1	0	4	3	7	6
Y-	1	2	4	5	7	8
X+	2	3-8	5	6-8	8	
X-	2		5	0-2	8	0-5
Y+	2	0-1	5	3-4	8	6-7
Y-	2		5		8	

2.2.3 Topology

An adequate network topology needs to be defined for TCCluster. Therefore, five topology characteristics that are required for a comparison will be defined. These attributes determine important properties of the network including the communication bandwidth and latency.

- Node degree: The number of links per node
- Diameter: The maximum distance between any two nodes in the system
- Bisection Bandwidth: The combined bandwidth of all links which are cut when dividing the network into two halves.
- Symmetry: A network is symmetric if it looks the same from every node
- Regularity: A network is regular if each node has the same degree

As shown in the previous paragraphs, *n*-dimensional Meshes, hypercubes and *k*-ary *n*-cubes represent feasible topologies for TCCluster. Using the characteristics defined above a comparison of the different topologies can be made. The results are shown in Table 2-2. Due to its high bisection bandwidth and low diameter, the k-ary n-Cube appears to be the preferred candidate. However, as discussed in the last paragraphs only systems with $k < 5$ are feasible due to their deadlock liability. Therefore, for larger systems the Mesh and the Hypercube topology is applied

Table 2-2: Topology Characteristics

Topology	Node degree	Diameter	Bisection Bandwidth	Symmetric	Regular
k-ary *n*-Mesh	$2n$	$nk/2$	k^{n-1}	no	yes
k-ary *n*-Cube	$2n$	n	$2k^{n-1}$	yes	yes
Hypercube	n	n	2^{n-1}	yes	yes

Based on these findings the Opteron architecture will now be analyzed in more detail in order to develop a concrete implementation. The current high performance server architecture from AMD named Magny Cours supports 12 compute cores on a single processor socket [33]. Magny Course is a multichip module that deploys two six-core

Opteron processors in a single package. The six-core processors share the same architecture as the four-core Opteron presented in Figure 2-1 and they are interconnected on the package by two HyperTransport links. Magny Cours contains four memory channels that provide an aggregated streaming bandwidth of up to 100 GByte/s. Most important for TCCluster is the fact that Magny Cours features four 16 bit HyperTransport links that support link-unganging. Link-unganging allows to split a 16 bit link into two 8 bit links which effectively increases the link count to eight. Figure 2-10 shows the architecture of the processor. It shows the two internal nodes P0 and P1 which are interconnected through one 8 bit and one 16 bit link. One of the shown external HyperTransport links is unganged into two 8 bit links All HyperTransport links can operate at 3.2 GHz double data rate. Furthermore, each node implements a two channel DDR3 memory controller.

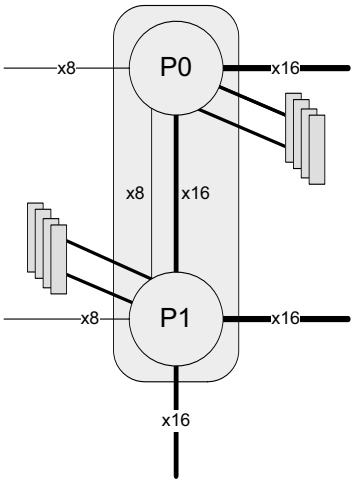

Figure 2-10:AMD Magny Cours processor

The Magny Cours processor as shown in Figure 2-10 can be used as a basic component for TCCluster. Each link can be used to interconnect to further processors, thereby, forming a large network. Although, the processor provides eight 8-bit HyperTransport

links, a maximum of seven links are available for TCCluster as at least one link is required to connect to a southbridge which is required to access I/O devices attached over PCIe, USB or SATA. Even in systems that do not require I/O, the link to the southbridge is required to fetch BIOS firmware code to initialize the system. Utilizing a node degree of seven, 3D-Meshes and 3D-Tori as well as 7-dimensional Hypercubes can be realized.

To reduce cost and to save valuable HyperTransport links we introduce the notion of *Supernodes*. A *Supernode* consists of two to eight processors which are interconnected by coherent HyperTransport links and form a shared memory system. All nodes within a *Supernode* can share a single southbridge. This approach enables a more efficient utilization of HyperTransport links for TCCluster and hence provides better performance. A *Supernode* is completely transparent to the TCCluster system and appears like a single node. This concept also enables a Hybrid programming model as threads within the *Supernode* can communicate via shared memory utilizing the coherent HT fabric while communication between Supernodes is carried out through the non-coherent TCCluster fabric.

A *Supernode* consisting of two Magny Cours processors is shown in Figure 2-11. Such a two node configuration opens up several possible interconnection schemes. In particular, the number of links as well as the link bandwidth needs to be distributed between internal coherent links and TCCluster links. Furthermore, it is possible to aggregate multiple 8-bit links to increase bandwidth or to maximize the degree of the network. It is important to note that internal links share coherent traffic between the processors in the *Supernode* and remote store traffic which is injected via TCCluster links. Depending on the communication scheme of the application that runs on TCCluster external traffic can thereby reduce the internal bandwidth of a node leading to a congestion. It is, therefore, recommended to assign a higher link bandwidth to the internal links than to the TCCluster links. This enables computation progress within a node even if the TCCluster links inject traffic into the *Supernode* at their maximum rate. It also enables exploitation of spatial locality as the communication bandwidth between threads within a *Supernode* is generally higher.

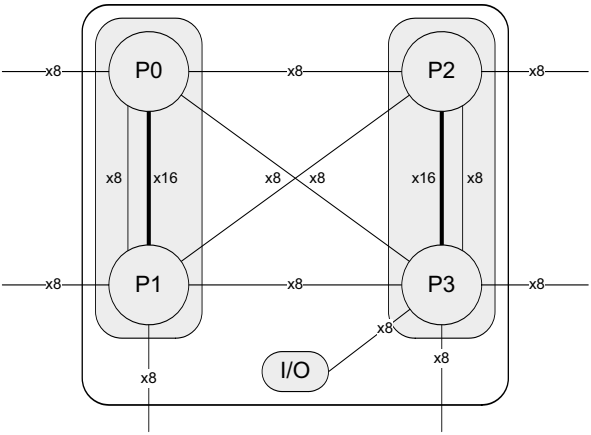

Figure 2-11:Two-processor Supernode

The *Supernode* shown in Figure 2-11 implements two Magny Cours processors and offers 24 cores in total. Internally, the processors are interconnected through two 16-bit HyperTransport links, six 8-bit links are assigned as TCCluster links. To reduce the hop count for internal communication, the 16-bit links have been unganged to enable a direct interconnect between the four processors $P_0 \cdots P_3$ which are distributed over the two Magny Cours dies. The topology offers an internal bisection bandwidth of 51.2 GByte/s and a maximum distance of one. In this topology one eight bit link on the left node is unused. It can be utilized to connect to an I/O device or be reconfigured as a TCCluster link enabling the additional seventh link. The two-processor *Supernode* allows to build 3D-Meshes, 3D-Tori 7-Dimensional Hypercubes.

A different two-processor *Supernode* is shown in Figure 2-12. It offers a TCCluster node degree of 10 which enables 5-dimensional Meshes and Tori as well as 10-dimensional Hypercubes. This topology enables highly scalable TCClusters with a very moderate diameter. However, for shared memory applications that utilize the coherent HyperTransport fabric intensively, the internal links can represent a bottleneck.

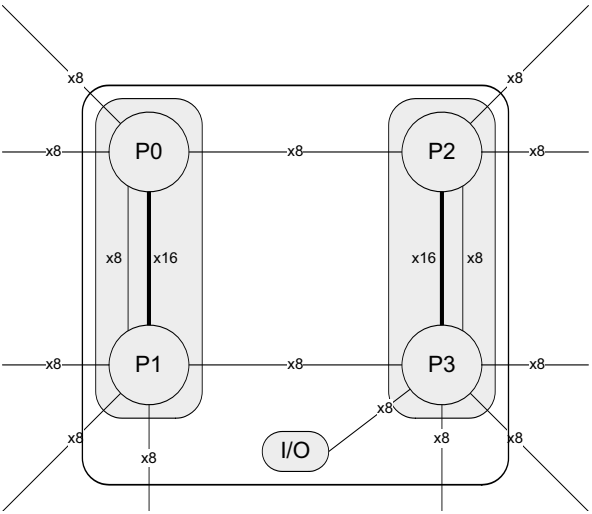

Figure 2-12:Two-Processor Supernode with Maximum Node Degree

Figure 2-13 shows a four processor, 48 core *Supernode*. The topology optimizes internal connectivity by assigning a large portion of the HyperTransport links to the internal coherent fabric. The internal links are unganged into x8 and are routed crosswise to offer a minimal distance between all processors. Thereby, it provides a bisection bandwidth of 104.4 GByte/s, a maximum distance of 2 and an average distance of 1.5. It offers seven external TCCluster links for 3D-Mesh, 3D-Tori and 7-Dimensional Hypercubes.

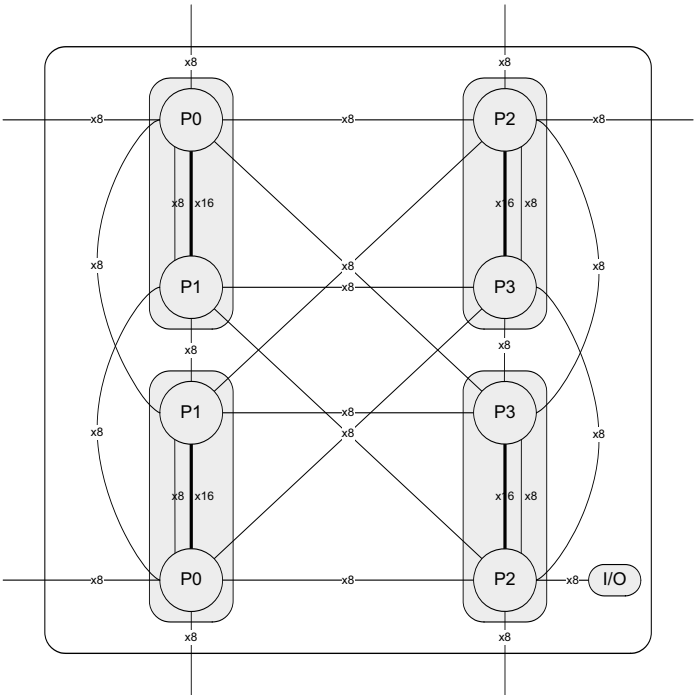

Figure 2-13:Four Processor Supernode

2.2.4 Fault Tolerance

The reliability of large computer systems is of paramount importance. Nevertheless, each integrated circuit and each transmission line is prone to physical errors. These errors can be classified into two different groups: hard errors and soft errors. A hard error is a permanent failure that can only be corrected with redundant hardware replacing the defect part. Soft errors, also referred to as *single event upsets* (SEU), are much more frequent in integrated circuits. They are caused by external hazards as cosmic rays or alpha particles tampering individual bits which lead to corrupted data. All integrated logic circuits and

transmission lines are, in principle, affected, however, semiconductor memories like static and dynamic RAMs are most vulnerable. As technology scales into the deep submicron area, the error rate and the probability for soft errors increases. Due to the lower operating voltages required to limit power dissipation and due to the reduced size of the gate area, the number of electrons constantly decreases. This allows the plentiful low energy particles that surround us to generate enough parasitic charge to cause a soft error. Shivakumar *et al.* model the probability of soft errors within a chip projecting an increase of nine orders of magnitude between the 600 nm and future technologies [119]. Furthermore, the error rate of compute clusters scales with the size of the system. Integrated circuits and network transmission lines may have a very low bit error rate (BER), however, for large systems the BER will add up to inacceptable levels. As applications are partitioned over all nodes of a cluster, a single bit error is sufficient to compromise the consistency of the complete system. In current chip technologies it is, therefore, absolutely necessary to employ error correcting code mechanisms to guarantee an adequate degree of reliability.

The theory of error correcting codes, as originated by R. W. Hamming in the 1950ies [63], is comprehensive. Therefore, it will solely be focused on the reliability mechanisms that can be supported in TCCluster and its consequence on scalability. As TCCluster utilizes HyperTransport as the network protocol it is required to analyze its provided fault tolerance mechanisms. Within HyperTransport 3 individual packets are protected using a 32-bit cyclic redundancy check (CRC). The CRC [112] is a hash function which is applied to the bits of a packet. It allows to detect accidental changes (errors) to the data within a packet. Therefore, the CRC is calculated by the sender and attached to the end of a packet. On the other side, the receiver performs the same calculation on the packet and compares the result with the appended CRC. This allows to detect errors that may occur during the transmission of the packet. To compute the packet CRC, a polynomial division is performed by *XOR-ing* the data packet with a generator polynomial. The polynomial is a sequence of ones and zeros and its form defines the properties and error detection capability of the CRC. HyperTransport utilizes a standard IEEE802.3 32-bit polynomial to protect the data portion of the packet as well as the command header. As shown by Koopman in [86] this polynomial achieves a hamming distance of five for packets of a

maximum length of 2975 bit, while HyperTransport packets have a maximum size of 608 bits. For smaller HyperTransport packets (172-268 bits) the hamming distance is six and for a minimum sized packet like a read request the hamming distance is 7. This implicates that for all HyperTransport packet sizes up to four, individual bit errors can be detected. Furthermore, all burst errors with a maximum length of 31 can be reliably detected.

In the case of an error, HyperTransport provides a retry mechanism to recover from and to return the system in a consistent state. Therefore, the sender stores a copy of each packet in the so-called *retransmission buffer*. The receiver acknowledges each successfully received packet and in the case of an error requests a retransmission of the corrupted packet. As long as occurring bit errors can be reliably detected by the CRC, the approach is perfectly reliable. The number of bit errors that occur within a single HyperTransport packet depend on the BER of the channel which is affected by many factors: The length (signal attenuation) of the channel, the impedance mismatch of the channel and crosstalk to other signals. All of these factors that impact the signal integrity are caused by the environment which consists of the traces on the PCB, the connectors and the cabling which implement the TCCluster link. The BER is, furthermore, determined by the signal frequency of the TCCluster link. At frequencies of beyond 2 GHz extensive pre-emphasis and receiver equalization techniques are required to enable operation. In the evaluation chapter TCCluster links running at different frequencies will be evaluated regarding their BER in a prototype system. It will be shown that the links can be reliably operated at frequencies of up to 5.2 Gbit/s.

2.2.5 Physical Implementation

The proposed approach enables tightly coupled clusters from commodity off-the-shelf hardware. All main components including processor, memory and I/O can be used unmodified, however, it is necessary to develop a mainboard that fans out multiple HyperTransport links to a backplane or cable connectors. In contrast to traditional systems in which nodes are connected by means of network cards, cables and switches, TCCluster utilizes the processor internal links directly. A possible TCCluster implementation,

therefore, consists of several multi processor mainboards that act as *Supernodes* which are interconnected via TCCluster links through a backplane or optical and electrical cables.

There exist two physical constraints that aggravate the implementation of such a design. First, AMD Opteron processors that communicate via HyperTransport require a mesochronous link clock that is derived from the same oscillator. Second, physical trace length of the links between two processors is limited to 24 inches by the specification.

A solution for the first constraint is straightforward. As the input clock of all processors has to be mesochronous (same frequency) but not synchronous (same frequency and phase) it is sufficient to employ a single clock for the complete system which is then fanned out to the different processors via clock distribution ICs. The clocks can be distributed either through the backplane or via cables. Jitter cleaners can be applied to compensate for the jitter that is introduced by the backplane traces and cabling.

To address the second constraint, it is required to match the topology of TCCluster with the location of the *Supernodes* within the system. For an $n \times n$ Mesh the distribution that minimizes trace length between the *Supernodes* is represented by employing n *Supernodes* in the horizontal axis as well as n *Supernodes* in the vertical axis. A blade type rack server in which the nodes are arranged vertically next to each other and then stacked in multiple rows provides a very balanced distribution of nodes in the x and y axis. Furthermore, the trace length limitation is specified for copper traces on FR4 PCB material. By using well shielded copper cables, signal integrity can be improved and the length can be increased to about one meter. Optical cables finally enable cable lengths of up to 100 meters with minimal signal degradation. Optical cables with in-built electrical connectors, thereby, enable arbitrary topologies and very large networks.

2.2.6 Limitations

The TCCluster interconnect technology provides high scalability and greater performance than traditional network protocols due to its closely coupled nature. Applications that communicate using many small messages can greatly benefit from the low latency TCCluster provides. On the other hand, there exist latency insensitive

applications that only require a very high network bandwidth. Although, TCCluster is capable of providing a very high bandwidth, communication is more expensive in terms of compute power when compared to other approaches. As in TCCluster messages can only be send using PIO, each data packet requires involvement of the processor. Many Ethernet or InfiniBand network interfaces additionally support DMA which can reduce the processor overhead. In this case, the processor only provides a descriptor to the network adapter. The data itself is then automatically fetched by the host channel adapter. As TCCluster directly uses processor internal hardware it is impossible to add DMA support without a modification of the CPU. This limitation holds true for all other CPU offloading techniques which move computation efforts from the CPU to the network device. This disadvantage of TCCluster is put into perspective by the increasing core count and computing resources of the upcoming processors. Current CPUs already deploy 12 and more processor cores which has lead to the reversed trend of CPU onloading. In fact, Regnier *et al.* propose "TCP onloading for data center servers" [114] to efficiently utilize CPU resources for networking.

TCCluster systems are, furthermore, limited in terms of scalability. As memory addresses are used to route packets to their destinations in the network, the number of address bits effectively limits the number of nodes that can be addressed in the system. As a shared global address space is deployed each memory address is globally unique and can, therefore, exist only on a single node. Current Opteron processors support 48 address bits resulting in a maximum physical memory space of 256 terabyte. Utilizing compute nodes with 64 GByte of local DRAM restricts the total amount of nodes that can be supported to 4096. Future processor revisions will incrementally increase the size of the physical address space to the full 64 bit which will gradually increase the maximum global address space. Nevertheless, for mid range clusters 4096 compute nodes represent an appropriate size.

2.3 IMPLEMENTATION

The discussion in the previous paragraphs motivates a real world implementation of TCCluster. A proof-of-concept implementation has been developed which interconnects two AMD Opteron based compute nodes via a TCCluster link. The prototype which is shown in Figure 2-14 consists of two Tyan S2912E motherboards interconnected with the HTX-2-HTX cable adapter that has been developed in the course of this work. The prototype permits booting into Linux OS and to execute application software that can utilize TCCluster as a regular network interface. The prototype implementation resulted in a number of contributions which include the development of an HTX-2-HTX cable adapter, the design of a global address space mapping scheme, the concept of an initialization protocol and the algorithm of setting up an operational TCCluster link.

.

Figure 2-14:TCCluster Two-Node Prototype

2.3.1 HTX-2-HTX Cable Adapter

In regular AMD based multiprocessor systems, the CPUs are interconnected via coherent HyperTransport links implemented as copper traces on the mainboard. As those links are inaccessible for TCCluster, a solution is required which routes one or multiple HyperTransport links to a connector which can be utilized for an inter-node connection. HyperTransport defines the HyperTransport Extension (HTX) socket for direct connections between Opteron CPUs and I/O devices. The HTX socket is a PCI like connector and there exist various plug-in cards like the InfiniBand host channel adapter Infinipath [42] as well as computation offloading accelerators. The TCCluster prototype exploits the HTX socket as a high speed cable connector. Therefore, the HTX-2-HTX cable adapter, shown in Figure 2-15, has been developed which consists of two PCBs interconnected by a high-speed differential Samtec cable. The cable supports data transfers of up to 20 Gbit/s over 2 meters with minimal signal degradation. The PCBs act as interposer PCBs to fan out the signals from the cable to the HTX socket. All signal traces on the PCB have been routed skew matched and impedance controlled to improve signal integrity and thereby enabling highest operating frequencies.

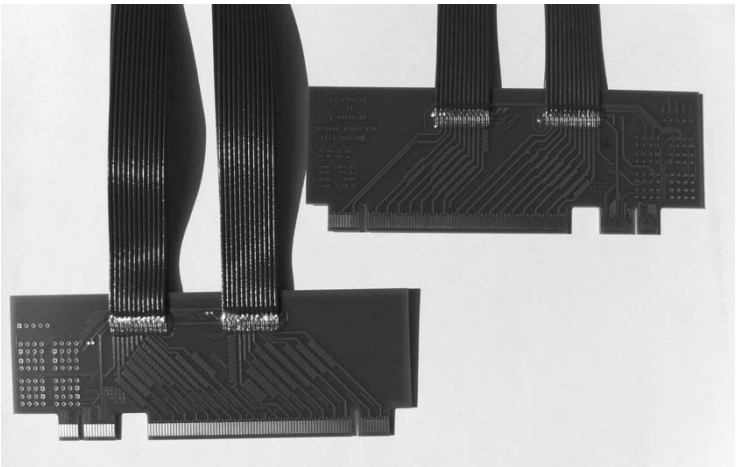

Figure 2-15:HTX-2-HTX Cable Adapter

HTX-2-HTX Signal Integrity Analysis

The HyperTransport specification defines operation for a maximum distance of 10 inches on FR4 material. As TCCluster realizes a node-to-node interconnect it is required to support significantly longer distances using copper cables. Electrical signals in copper experience a degradation proportional to the length of the track or wire and the amount of reflections within the transmission line. The longer a copper trace is, the more high frequency components are lost leading to a longer rise time which finally results in a distortion of the received signal. Similarly, impedance discontinuities caused by vias or connectors induce reflections of high frequency components which are sent back to the transmitter without reaching the receiver. TCCluster links increase the length of the transmission line and introduce impedance discontinuities due to the additional connectors. It therefore, needs to be analyzed which frequencies can be supported reliably on the TCCluster system by performing eye measurements at the receiver side of the cable. Signal measurement directly at the Opteron's receiver pins which provide more accurate results could not be conducted due to mechanical obstructions. Figure 2-16 shows the monitored eye diagram for a link running at 2.4 Gbit. The eyes are still visible, however, noise can be observed even close to the center of the eye which leads to an absolute eye opening of only 68 ps in the horizontal and 71 mV in the vertical dimension. Such small eyes can lead to incorrect sampling of specific receive patterns (intersymbol interference) increasing the bit error rate of the system.

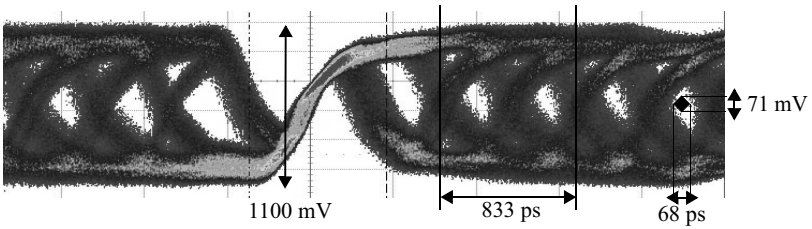

Figure 2-16:HT1200 Link Default Parameters

To increase the signal integrity for long copper traces, the Opteron transceiver provides digital compensation circuits. Depending on the data pattern that is transmitted, the sender can attenuate or boost the amplitude of the signal. By de-emphasizing long static bit

sequences the voltage amplitude for such symbols is decreased, leading to an improved rise time for the following, high frequency signals. The Opteron transmitter allows different de-emphasis settings of 3, 6, 8 and 11db. Figure 2-17 shows the monitored signal with a maximum de-emphasis setting of 11db. It clearly shows two different peak-to-peak amplitudes, one of 1100mV and another of 487mV. While the deemphasis leads to a wider eye opening in the horizontal dimension due to the improved slew rate, the deemphasis negatively effects the horizontal eye opening.

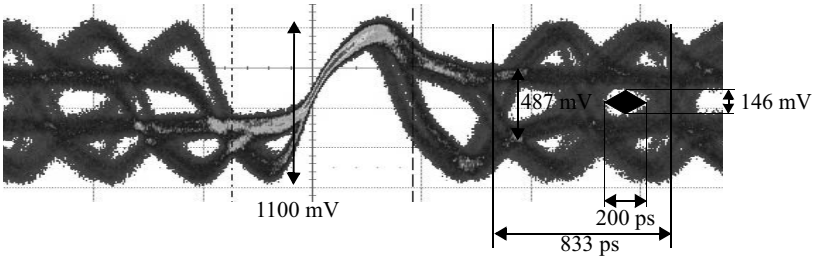

Figure 2-17:HT1200 with Maximum De-Emphasis

The available de-emphasis modes have been analyzed for different operating frequencies for the TCCluster system. It appeared that 6db de-emphasis represents the optimal setting for our system. By enabling the decision feedback equalizer in the receive block of the Opteron transmitter, stable operation could be achieved at the maximum link frequency of 2.6 GHz, representing a bit rate of 5.2 Gbit/s. Figure 2-18 shows a monitored HT2600 signal measured at the receiver.

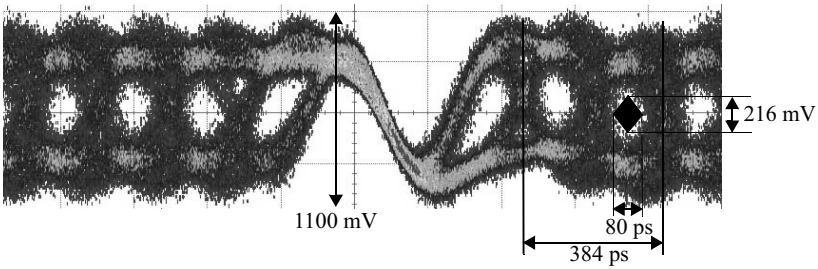

Figure 2-18:HT2600 with 6db De-Emphasis

2.3.2 Address Mapping Scheme

A regular Opteron-based shared memory system consists of multiple processors with individual physical memory modules attached to each processor. The complete physical memory is aggregated to form a single shared physical address space and each processor has an identical memory map. In a TCCluster, parts of the local physical memory are made accessible for remote nodes. This memory area can then be used to exchange messages between the nodes. In TCCluster, local memory is treated as DRAM and remote memory is treated as MMIO space. This requires a different and unique address map for each node as depicted in Figure 2-19.

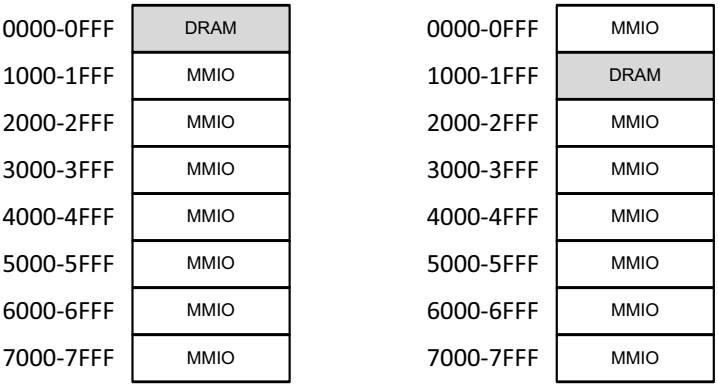

Figure 2-19:Example Address Map for two different nodes

Write accesses from Node0 to the address range 0000-0FFF result in a local physical memory access. The same address range accessed by Node1, however, results in a network package that is transmitted over a non-coherent TCCluster link towards Node0. The mapping of local DRAM space into remote node's MMIO space represents a security issue as each node can overwrite physical memory of other nodes in the network. To address this issue, it is required to implement a remote memory management and security policy on the kernel level. By configuring only segments so-called *windows* as remotely writable, remote nodes cannot overwrite sensitive data in remote memories.

There exist three different memory types in TCCluster.

- Local memory: This is the regular type of memory, used to execute applications on the local node. It must not be accessible by remote nodes.
- Remote TCCluster memory: A window of remote memory which is mapped as MMIO and which allows to send packets to other nodes using remote stores.
- Local TCCluster memory: A window of local memory which is made accessible to remote nodes for receiving remote stores.

The TCCluster memory management scheme must support two different tasks. One is to manage remote TCCluster address windows to guarantee that applications on the local node can only write to specific locations on the remote nodes. The second task is to manage local TCCluster memory. As local TCCluster memory resides in DRAM just as regular local memory it is, a priori, managed by the operating system that performs allocation, paging and enforcement of a segmentation mechanism. To use local memory as a *receive buffer* for remote stores the TCCluster driver needs to cooperate with the operating system. To perform these tasks two different schemes have been developed.

In the first scheme the BIOS firmware reserves a portion of local main memory as remotely accessible address space during boot time. This memory space is kept hidden to the operating system and managed separately by the TCCluster kernel driver. The reserved memory area is static and known by the kernel driver. Applications can allocate receive buffer space by requesting a certain amount of memory. If a contiguous range of physical memory of that size is available in the reserved memory region, the driver maps the physical address range to a virtual address range and returns it to the application. The driver does not support paging, hence, it is impossible to map more virtual address space for remotely accessible receive buffer space than available physical memory, reserved by the BIOS during startup. This does not represent a limitation as receive buffer space has to be pinned anyway as the sender would otherwise not be able to determine whether the receive buffer is currently residing in main memory or swapped to disk. To enable sending of messages, remote TCCluster memory needs to be allocated. Therefore, a send buffer that points to a specific remote node is requested from the driver. While this approach is

lean and easy to implement it is also inflexible as the amount of memory which is reserved by the BIOS for TCCluster needs to be known at boot time.

In the second approach, both local and remote TCCluster memory is completely managed by the operating system. To enable communication, the sending and receiving processes need to negotiate a unidirectional connection. Therefore, the receiver allocates pinned local TCCluster memory dynamically by using a kernel driver. The driver returns a virtual address to the application and communicates the physical address to the kernel driver on the send node. On the sender side the physical remote TCCluster memory address is mapped into a virtual address. While this approach is very flexible and enables to allocate TCCluster memory at runtime, it requires a separate communication mechanism to initialize a TCCluster connection. Possible solutions are using a separate ethernet connection or by using a TCCluster connection with static and well defined addresses. Such a connection can be established by using the first scheme.

From the performance perspective both schemes behave identical. Therefore, for the prototype presented in the evaluation paragraph it was decided to utilize the first approach. An implementation of the second scheme using a standard ethernet connection to negotiate TCCluster connections between endpoints will been implemented in the future. As the initialization process is non-critical for the performance, it can be implemented using a Gigabit ethernet connection which is available on all computer systems. After the initialization is complete the applications can communicate over TCCluster exploiting the bandwidth and latency advantages.

2.3.3 Initialization

Before computer systems are able to execute software and to run an operating system, they have to be initialized through BIOS firmware. Therefore, code that resides in non-volatile flash memory is executed by the processor to setup all devices in the system. In an AMD environment the code is retrieved via the southbridge which is connected to one specific processor, the boot strap processor (BSP) via a non-coherent HyperTransport link. A multiprocessor system, therefore, consists of a single BSP and a number of Application Processors (APs). The BSP first initializes itself and then performs the coherent link

enumeration to initialize the HyperTransport fabric which spans the BSP and all APs. Before the BSP is able to configure the BSP's and AP's routing table it has to determine the topology of the system. This can be done either in a static way by passing the topology description to the firmware at compile time or dynamically at runtime. Therefore, the processor performs a depth-first search for all APs. The BSP traverses all APs and assigned a new nodeID to every AP in the system. Using the nodeID information, the BSP configures all routing table entries within the system. In the end, the topology consists of a structure of the nodes and the HyperTransport links interconnecting them. As the southbridge occupies a HyperTransport link, TCCluster utilizes the concept of *Supernodes* which enables sharing of a southbridge by multiple processors as shown in Figure 2-20.

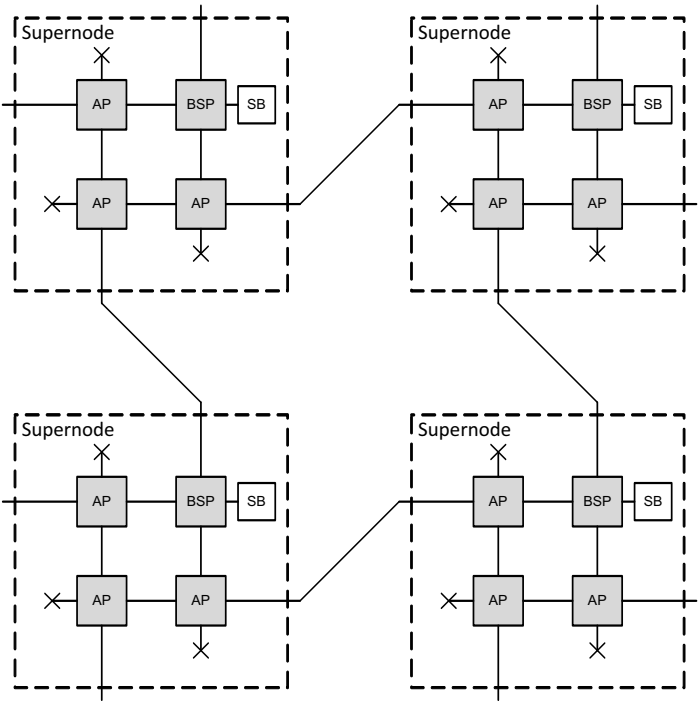

Figure 2-20:Four Interconnected Supernodes.

2.3.4 TCCluster Initialization Algorithm

In this paragraph step-by-step instructions will be given to enable operational TCCluster links. Many actions need to be carried out by the BIOS, hence a proprietary firmware has been developed. As a foundation for our code we use coreboot [102], also referred to as LinuxBIOS, which is an open source BIOS firmware project that already supports many AMD and Intel platforms. It was required to rewrite most parts of the coherent and non-coherent link enumeration code to implement the following sequence which configures a HyperTransport link as a TCCluster link.

- Cold Reset: Both platforms exit cold reset simultaneously and perform their HyperTransport low level link initialization. The TCCluster link is configured as coherent.
- Coherent Enumeration: Both platforms perform the usual boot sequence including coherent link enumeration. At this point the TCCluster links are still configured as coherent which would cause the regular firmware to perform a search for all coherent links thereby building the system topology. The modified TCCluster firmware avoids this by ignoring such links and only performs coherent link enumeration for the nodes within a *Supernode*.
- Force Non-Coherent: Each TCCluster link in the system is forced into non-coherent mode. Furthermore, the link speed is increased from 400 to 5.200 Mbit/s.
- Warm Reset: Both platforms or nodes issue a warm reset, which results in another low level link initialization. The modified settings now become effective and the TCCluster link gets configured as non-coherent.
- Northbridge Init: Both platforms configure their northbridge including nodeID, DRAM address range, MMIO address range registers and routing registers as described in the previous paragraphs. For the first prototype, reconfiguration is performed via the coherent link between Node0 and Node1, in the second setup each machine configures itself individually.
- CPU MSR Init: The Memory Type Range Registers (MTRR) on both nodes are reconfigured to map a large uncachable address space to the TCCluster MMIO link. This causes the processor's system request queue to generate non-coherent posted HT packets which are required for TCCluster.

- Memory Init: The machines initialize their memory controllers and report the size and type of memory which is attached to the processors.
- EXIT CAR: Until now the firmware is executed in cache as RAM (CAR) mode in which the code is fetched from the firmware ROM and the L3 cache is treated as main memory. At this point, the system is comparatively slow as the performance is limited by the read bandwidth of the ROM. To exit CAR, the firmware is copied into main memory and the program counter gets pointed to main memory.
- Non-Coherent Enumeration: The processors interconnected by a TCCluster link appear as non-coherent devices which causes regular firmware to perform I/O device enumeration for this link. This needs to be disabled for each TCCluster link.
- Post Initialization: The firmware performs TCCluster independent tasks and loads the operating system
- Loading Operating System: After the firmware configuration is completed the operating system, in our case Linux, can be loaded. The OS also switches the system from 32 bit protected mode into 64 bit user mode.
- Enabling Remote Access: The device driver maps the remote address range as memory mapped I/O and provides access to the API.
- Data Transmission: The API requests page wise memory mapping of remote addresses into user space. User software can now access remote memory.
- Data Reception: On the remote node the same physical memory address, which is DRAM based in this case, has to be mapped into user space which enables to receive data from remote nodes.

2.3.5 Non-Coherent Configuration

HyperTransport links between two processors are generally configured as coherent. As soon as the Opteron processor emerges from its reset state, it enters the low level initialization and begins to configure its HyperTransport links. Therefore, it drives some specific data patterns on the wires trying to detect another device that may reside on the other side of the link. If both endpoints conform to that sequence the link connection is established. Then, both endpoints identify themselves as a coherent or non-coherent

device to determine the type of the link. In the case of two Opterons the link type will be coherent and the boot strap processor (BSP) can now use this link to configure itself and all other devices in the fabric.

TCCluster requires non-coherent links between the processors. While this mode is neither intended nor referenced in any of the processor or HyperTransport specifications it is still possible to enforce such a behavior. The processors implement a specific register for debug purposes enabling non-coherent operation. This possibility is exploited by our approach. After the initialization phase, the HyperTransport links are configured coherent which enables the BSP to access and set the debug register. The modifications become effective at the next warm reset that causes a re-initialization of the link. At this time, the processors identify themselves as non-coherent devices. This technique allows to enforce a non-coherent link between processors.

2.4 EVALUATION

To evaluate the TCCluster interconnect we provide software micro benchmarks that show the latency and bandwidth performance of our technique. We developed a Linux driver that maps remote TCCluster memory addresses into the user space and a rudimentary message library that can be utilized to send and receive messages. As the operating system we run Linux with a custom 2.6.34 kernel. We needed to compile our own Kernel to comply with a limitation of TCCluster caused by interrupts. Within the HyperTransport fabric, interrupts are broadcasted to inform coherent and non-coherent devices within the system about specific events. It is required to avoid broadcasting of interrupts over TCCluster as interrupts have to be handled within the system and must not be sent over the network. Therefore, all system management calls (SMC) need to be disabled which can be only achieved with a custom kernel.

The user space message library provides the following functionality. It can open local and remote memory addresses by calling the TCCluster device driver. The *send* function can then be used to transfer data to remote memory addresses. The *receive* function is called on the remote side to receive data from local memory. TCCluster transactions cannot generate cache invalidation requests on the receiver side. Therefore, the receiver needs to map the local memory which is accessible by the remote nodes as *uncachable*. This guarantees that all reads to remote-accessible memory bypass the cache and directly hit main memory. Although, this approach generates additional processor-memory bus overhead when polling the memory, it represents the only way to guarantee that incoming data is seen by the processor. The message library will offer support for synchronization primitives using the *Sfence* machine instruction. *Sfence* enforces a strict ordering between store transactions that can be utilized by the library to implement a synchronization mechanism. The message library provides the basis for higher level middleware, for example the MPI message passing protocol.

2.4.1 Benchmark Methodology

In the following, we present the latency and bandwidth performance of a two-node system interconnected by a single bidirectional TCCluster link as shown in Figure 2-21. Both nodes deploy two AMD Opteron Istanbul six core processors running at 2.6 GHz and offer 6 MByte of L3 cache. The machines were equipped with 16 GByte of main memory per node, running at DDR800 using both memory controllers in the processor in dual rank configuration. The HyperTransport links used for the node-to-node interconnection were operated in three different modes. We analyzed the performance using a 16 bit gen1 link at HT800, an 8-bit gen3 link at HT1600 and an 8-bit gen3 link at HT2600. All three configurations showed stable operation over a period of 24 hours with zero link errors after configuring the pre-emphasis and DFE accordingly. The analysis of a high frequency 16-bit gen3 link was not possible due to a physical limitation of the mainboard. The reason is that HyperTransport Gen3 modes require an additional signal which is not supported on the Tyan platform.

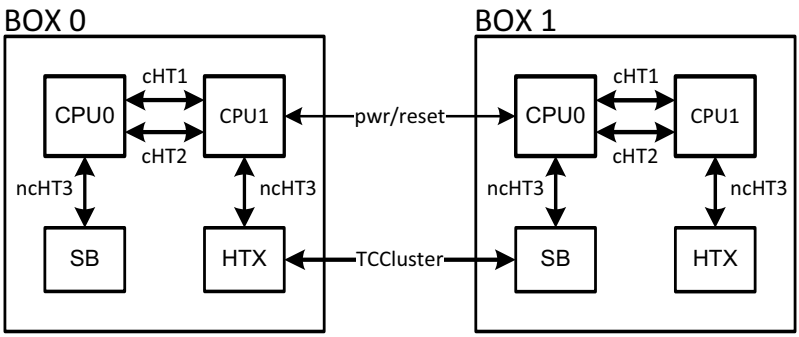

Figure 2-21:Evaluation Setup

We developed two user level software benchmarks that utilize the TCCluster API to access the hardware. As the operating system we used Linux with a 2.6.25 kernel. The application code has been developed in C and compiled using GCC 4.4. We utilized OpenMP for semi-automatic parallelization enabling multithreaded execution of the

program. Multithreading has been supported to assure that enough store instructions can be generated by the execution pipelines which is required to fully saturate the bandwidth of the TCCluster link. The same approach is applied by the popular STREAM benchmark which measures main memory bandwidth and only achieves maximum performance when executing multiple threads concurrently. To conduct the tests we varied the number of concurrent threads and processes and tried different permutations thereof. Furthermore, we utilized the program numacontrol which allows to bind software threads to specific cores within a multi-node, multicore system. As the TCCluster link connects specific processors it was expected that the performance results depend on the processor which is used to execute the benchmark. To increase the bandwidth of the TCCluster link we enable *write-combining* for the memory range that points to the TCCluster link. It enables to pack multiple consecutive 64 bit store instructions and to send them out as a single packet. Such maximum sized HyperTransport packets reduce the command overhead and thereby increase link utilization.

2.4.2 TCCluster Bandwidth

Figure 2-22 shows the bandwidth of the TCCLuster link running at different link speeds. The HyperTransport frequencies we analyzed are HT800-16 bit, HT1600-8 bit and HT2600-8 bit. As all three modes use DDR the theoretical peak performance is 3.2 GByte/s, 3.2GByte/s and 5.2GByte/s respectively. As HyperTransport packets have a minimum overhead of 96 bits (64 bit command plus 32 bit CRC), the raw bandwidth that can be realized represents 84% of the theoretical peak performance. While the HT800 mode performs worse than the faster modes for small messages, it provides better results than the HT1600 mode for large packets. This implies that the 16 bit mode is implemented more efficiently in the AMD Opteron processor. The maximum bandwidth that can be achieved is 2690 MByte/s for the HT800 case, 2100 MByte/s for the HT1600 case and 3420 MByte/s for the HT2600 case. It is worth noting that the maximum bandwidth is already achieved with 1K byte messages while other interconnects including InfiniBand only achieve their maximum performance with much larger (4K byte) messages. The message size which delivers 50% of the peak performance, also referred to as $n/2$

performance, is 64 bytes which represents an outstanding result. Both facts prove the applicability of TCCluster for fine grain communication.

As a baseline, the InfiniBand ConnectX network adapter from Mellanox can be referenced [123]. It provides an MPI bandwidth of 2500 MB/s for 1 MB messages, 1500 MB/s for 1K messages and 200 MB/s for cache line sized messages. Although, our evaluation does not include the overhead of the MPI middleware it can be seen that TCCluster provides a significant performance edge over InfiniBand especially for small messages.

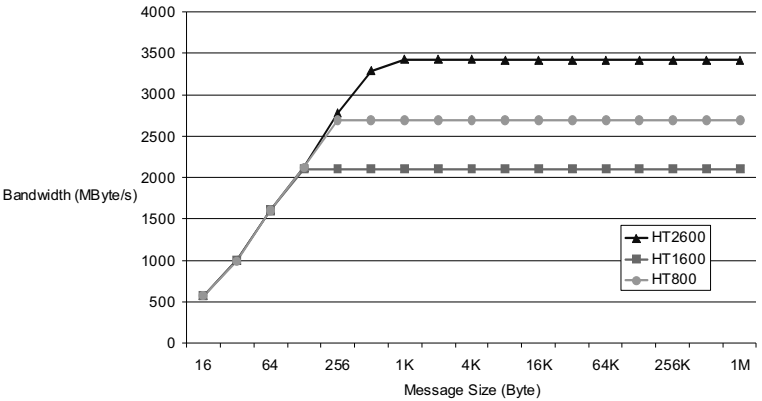

Figure 2-22:TCCluster Bandwidth

2.4.3 TCCluster Latency

In the next benchmark we measured the communication latency that can be achieved with TCCluster. We used a standard ping-pong kernel in which the receive node polls a specific memory location and sends back a response as soon as the first message arrives. The full round-trip latency is divided by two to calculate the one-way latency. As shown in Figure 2-23, TCCluster provides a very low half-round-trip latency for 64 byte packets between two nodes of 240 ns. It can be seen that smaller packets show a slightly larger

latency. The reason for this behavior can be explained with the microarchitecture of the write combining buffer. Only cache lines sized packets are sent out immediately after writing the last quadword into the buffer and, hence, deliver the best performance. For all smaller packet sizes the buffer is not entirely full after writing the message. The result is that the buffer does not need to be flushed immediately which leads to a slightly worse latency performance. However, even for 1 KByte messages the latency is still below 1 us which represents, to the best of our knowledge, the best performance ever measured on a x86 architecture. Other high performance networks like InfiniBand currently achieve end-to-end latencies of around 1 us for minimal sized packets which leads to a 4X performance advantage for TCCluster.

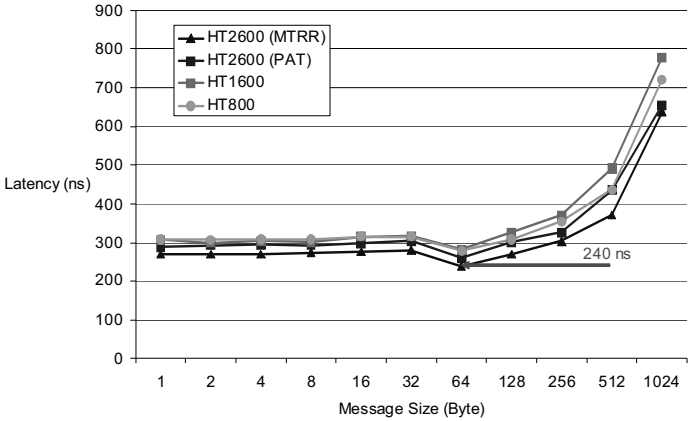

Figure 2-23:TCCluster Latency

Furthermore, we performed measurements to determine the influence of the processor location on which the benchmark is executed. By utilizing the Linux application numacontrol the latency benchmark was bind to either node 0 or node 1 within the system. As both boxes contain two processor nodes and the ping-pong kernel consists of a send and receive process there exist four different permutations for scheduling the processes. Figure 2-24 shows the benchmark results using an HT2600 link, while for instance

(*n0, n*1) defines that the sender thread runs on node 0 and the receiver thread runs on node 1. As the TCCluster link is connected to node 1 in both boxes the (*n*1, *n*1) configuration shows the best result. Adding another hop by moving the process to node 0 increases the latency by approximately 50 ns. Consequently, the (*n*0, *n*0) configuration introduces a latency penalty of 100 ns. This benchmark is of particular interest as it provides a performance estimation for larger systems. As TCCluster utilizes a direct topology the median communication latency increases with the number of hops, and respectively, the number of nodes in the system. A very low switching latency per hop is, therefore, of paramount importance. Following these results, in a hypothetical 1024 node, 32×32 Mesh system, the end-to-end latency between remote nodes would still not exceed 1 us.

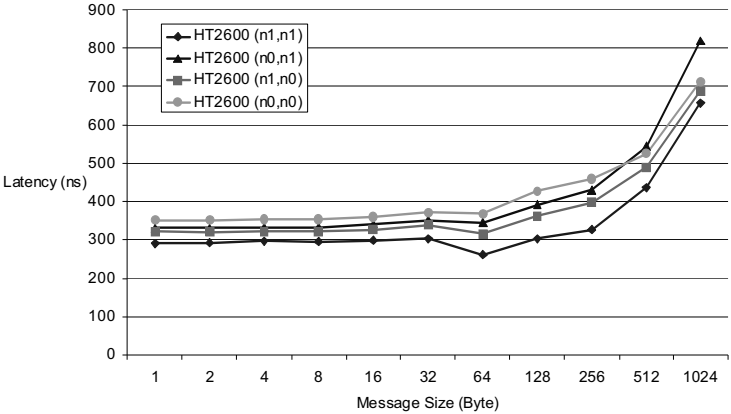

Figure 2-24:NUMA Impact on Communication Latency

2.4.4 TCCluster Message Rate

The next benchmark analyzes the message rate performance of the TCCluster interconnect. The message rate is determined by counting the maximum number of minimum sized messages that can be transmitted per second. The message rate is coupled to the bandwidth that can be achieved in a system, however, focusses on small messages.

In the measurements we report true hardware message rates, without combining multiple messages in software into a single message, to synthetically increase the message rate. Within a computer system, there exist various factors that determine the maximum message rate. As shown in Figure 2-25, running the message rate benchmark using a single thread, a message rate of 48 million messages/s can be achieved. While this result outperforms comparable technologies like InfiniBand that achieve a rate of 20 million messages/second, it is significantly lower than the theoretical capability of the link. It appears that a single core is not able to saturate the TCCluster link when sending minimum sized messages. Therefore, we performed measurements with 2, 4, 8, 16 and 32 threads. A significant speedup until the maximum message rate of 430 million messages/s is achieved using 32 threads can be observed. This represents a 20x performance advantage over InfiniBand.

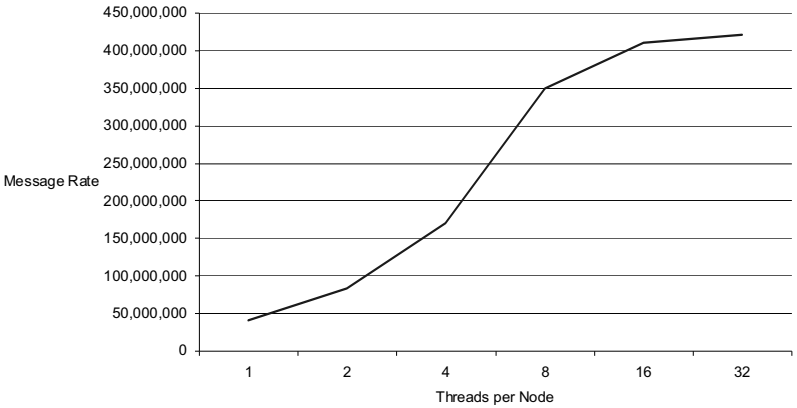

Figure 2-25:TCCluster Message Rate

2.5 CONCLUSION

We have presented a novel technique named TCCluster for interconnecting large cluster systems by utilizing the processor host interface as a direct network interconnect. By virtually placing the network interface into the processor we achieve a much higher bandwidth as well as a lower latency than traditional interconnects including Ethernet and InfiniBand. Our approach does not require any hardware modifications as it is realized entirely in firmware and kernel layer software. In particular, the HyperTransport host interface implemented in any AMD Opteron processor is exploited and reconfigured as a network interface, thereby, virtually moving the interconnect into the processor. Our solution achieves an outstanding software-2-software network latency of 240 ns and a bandwidth of more than 3400 MByte/s for 1 KByte messages. The contributions of this chapter are, in particular: A novel interconnection technique which enables AMD Opteron processors to communicate with ultra low latency and very high bandwidth. A new remote store programming model that uses a partitioned global address space to exchange data between processes on different nodes. An in-depth summary of interval based routing mechanisms for a specific set of topologies and the introduction of a interval labeling scheme that allows deadlock free routing for the proposed TCCluster technology. A proof of concept implementation of the complete technique including the development of a new BIOS firmware together with a complete software stack. And finally, a thorough evaluation of the mechanism based on a two-node prototype.

As outlined before, we applied a prototyping methodology to verify TCCluster and to prove the merit of our technique. Although, this approach has been extremely time-consuming, it turned out to be very beneficial. While simulations that have been conducted at the beginning of the project showed the operation of our technique, the prototype revealed different issues that could not have been discovered by simulations. One of these issues that frequently lead to the crash of the prototype system, has been induced by the power management policy of the operating system. In particular, each time the operating system triggered the *halt* instruction to put the system into a temporarily sleep mode, the system crashed. Tracing the root of the problem has been extremely challenging and was finally achieved with a logic state analyzer connected to the pins of

the HTX-2-HTX cable adapter. This technique enabled to monitor the discrete packets on the TCCluster link which lead to the comprehension and the solution of the problem. Understanding the complex interactions between hardware, power management, operating system, and software is impossible with simulation-based techniques. This fact proves the feasibility of our design and evaluation methodology.

To the best of our knowledge, TCCluster currently provides the best interconnect performance for x86 based systems, outperforming comparable approaches by an order of magnitude. The main reason for this performance advantage originates in the fact that TCCluster virtually moves the interconnect into the processor, rendering all protocol conversions unnecessary. While TCCluster utilizes an undocumented feature in the AMD processor architecture, the question arises why processor manufacturers not officially support such interconnect capabilities, particularly, when taking into account that the mentioned Transputer already provided such features. There exist both economical and technical reasons for excluding the interconnection network from the processor. On the one hand, not all applications utilize an interconnection network, wherefore, it may be inefficient to spend silicon on such features. On the other hand, the x86 instruction set does not provide networking primitives to handle the data transfer between nodes in a network. Hence, a major contribution of this work represents the development of a technique that reduces message-sending including packetization and way-finding to a simple store instruction. The TCCluster mechanism addresses both economical and technical concerns by providing a cost efficient implementation that only reuses available hardware. It remains to be seen whether manufactures will explore that opportunity by adopting the technique in future processor generations.

While the TCCluster technique represents an effective solution for addressing the scalability problem in high performance compute clusters, it does not improve the performance of the processors within the cluster. Therefore, in the next chapter we will present a technique to improve the scalability of on-chip components within integrated circuits.

CHAPTER 3
SCALABLE NETWORKS-ON-CHIP

3.1 INTRODUCTION

Recent advancements in silicon technology have provided the possibility of integrating billions of transistors on a single chip. Processor manufacturers have leveraged this abundance of transistors to integrate more and more functionality into the processor, including memory controllers, multiple layers of caches and I/O interfaces. In addition, the single thread performance of processors plateaued which encouraged designers to work on multi- and many-core architectures. Recent publications [9] predict a two-fold increase of cores per year and Intel has already introduced prototype implementations of 80 core processors [127]. One of the most important aspects of such architectures is the communication infrastructure between the components as it determines the overall performance of the chip to a large degree. The design of a resource and power efficient architecture which, in addition, provides good performance is one of the greatest challenges within many-core designs.

As a solution, Dally and Towles have proposed the use of networks-on-chips (NoCs). In contrast to previous architectures in which components were simply connected through wires, they advocate to “route packets not wires” [40]. NoCs provide a number of significant advantages over traditional architectures. One is that they address the increasing significance of wire delays [69][21] which in current chip designs have started to dominate the gate delays. As the operating frequencies reach into the gigahertz range, timing closure gets increasingly difficult as signals require several pipeline stages to travel a complete chip. NoCs represent an elegant solution for this problems as they decouple the different on-chip components, in regards to area and time. Components are not required to operate in a fixed cycle relationship anymore as the NoC separates the computation and communication task from each other.

Another advantage of NoCs is the possibility of reusing wires within a chip much more efficiently than in traditional architectures that utilized shared medium networks (bus) to interconnect the components on a chip. In bus based architectures, the bandwidth is shared between all endpoints, thereby, limiting scalability. NoCs in contrast, provide point-to-point links as well as switch stages which enable sharing of resources while providing a good scalability.

On the architecture level, NoCs increase the modularity and reusability of designs. By defining a standard interface between the components and the NoC, the flexibility of designs is significantly improved. Components can be interchanged easily enabling new architectures to be extended efficiently. A feasible approach to handle the complexity of future computing systems is the application of a component-based design methodology that supports component reuse in a plug-and-play fashion as described by Benini [16]. The adoption of well defined components interconnected by a high performance NoC represents the only feasible solution for reducing the integration and verification effort of large Systems-on-Chip (SoCs).

Novel developments, including reconfigurable NoCs [132], address the issue of defects and process variations. This issue is becoming increasingly important in chip designs as the constant scaling of technology size into the deep sub micron area increases the impact of process variations. The non-uniform width of an individual transistor can effect the functionality and maximum operating frequency of a design significantly. Manufacturers, therefore, divide their chips into good and bad lots and operate them at different frequencies. Reconfigurable NoCs, on the other hand, allow to adjust the operating frequency at a much finer grained level. Each individual component can be operated at its own maximum frequency while a globally asynchronous, locally synchronous (GALS) NoC [29][105] acts as the synchronization interface. This allows a frequency adjustment at runtime which enables sophisticated power management techniques. Completely errorneous logic can be easily replaced by redundant components through a simple reconfiguration of the NoC.

The prevailing NoC architectures can be subdivided into two categories which are the heterogeneous and the homogenous architectures. Heterogeneous NoCs as shown in Figure 3-1 are prevalent in almost every modern SoC covering smartphones to large embedded systems. In this case, a number of components like processor cores, radio, graphic accelerators and memory controllers, possibly all from different vendors, have to be integrated on a single chip. The development and verification effort of such complex platforms is only feasible by introducing a standardized interconnect structure which enables the communication between the different IP blocks.

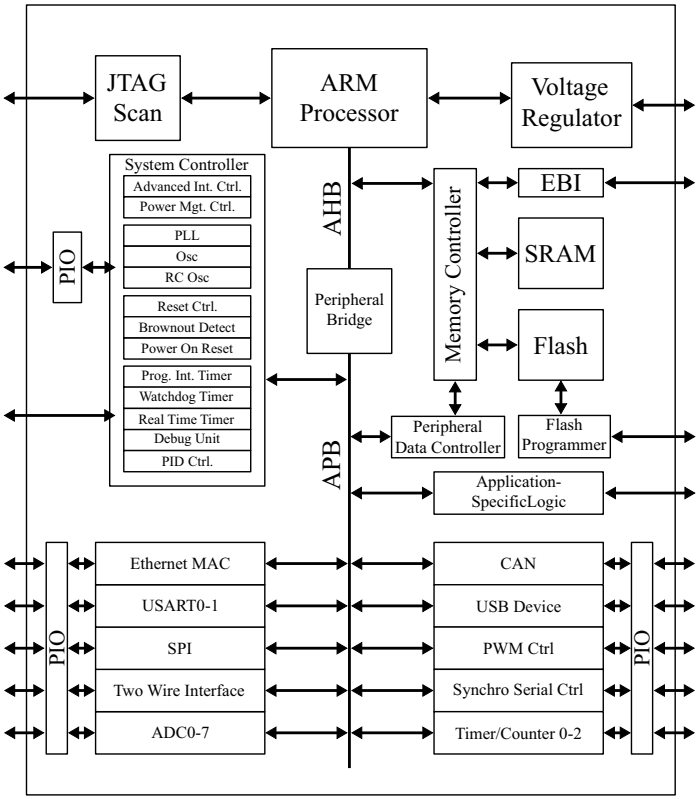

Figure 3-1:Heterogeneous System-on-Chip[1]

1. *Source: Wikipedia, Wikimedia Commons, licensed under CreativeCommons-License en:User:Cburnett, URL: http://creativecommons.org/licenses/by-sa/2.0/de/legalcode*

Currently, there exist two prevailing IP interface specifications which are the Advanced Microcontroller Bus Architecture (AMBA) [6] maintained by Advanced RISC machines (ARM) as well as the *wishbone interface* [131] that is supported by the opencores project. The main disadvantage of both specifications is that they are primarily bus based. They define different interfaces for master and slave devices and both approaches provide complex, high overhead interfaces with very low performance. Furthermore, AMBA is constantly being modified, now in its fourth generation, defining several subinterfaces which are each optimized for different purposes. The following enumeration is only a subset of the existing AMBA specifications: AHB, AHB-Lite, ATB, APB, multi-layer AHB, AXI and AXI4 add up to quite a number of mostly incompatible protocols. AMBA AXI has been enhanced to support point-to-point NoCs, however, the adoption has been ineffectual as obsolete bus oriented concepts have been reused. This results in inefficient intermediate layers with even higher overhead. Hence, there exists the need for a scalable, flexible and highly efficient infrastructure optimized for NoCs. A possible solution is represented by the HTAX framework that has been developed in the course of this work.

Homogenous designs consist of a large number of identical components interconnected by a NoC. In principle, every multicore processor including IBM's Cell [31] or AMD's Opteron that implements a number of identical cores and a switched interconnect can be regarded as a homogenous NoC architecture. However, in general, the Multiprocessor System-on-Chip (MPSoC) category embraces tile based designs as shown in Figure 3-2. Examples like the TILE64 [3] from Tilera, Intel's 80 core Polaris [127] or the TRIPS [116] microprocessor consist of three main basic blocks which are compute elements, memory controllers and the NoC. The advantage of homogenous architectures is their similar structure which greatly facilitates the development and the layout process. Homogenous designs are focused to a much larger extend on performance than on reusability and flexibility. Therefore, they are implemented in the form of custom manufactured ASICs with proprietary NoC interconnects. Designing an MPSoC NoC from scratch is a complex and time consuming task. There exists the demand for electronic design automation (EDA) tools that assist designers to specify and implement their NoCs. The HTAX approach proposed in this work not only provides the framework for such architectures but also the tools for automatic generation of NoCs, suitable for many different

applications. This technique optimizes the design and verification flow of new designs and reduces risk by reusing silicon proven IP.

To ensure the future scalability of both heterogeneous and homogenous architectures the NoC component is of paramount importance. For this purpose, all layers of the NoC including the protocol, the network and the physical link layer have to be carefully designed. A NoC architecture requires scalability in terms of size and performance while being flexible to enable adaption to upcoming challenges. Accelerating the design process by employing NoC specific EDA tools and a component based methodology is a key factor. Addressing these requirements in conjunction is the goal of the HTAX framework.

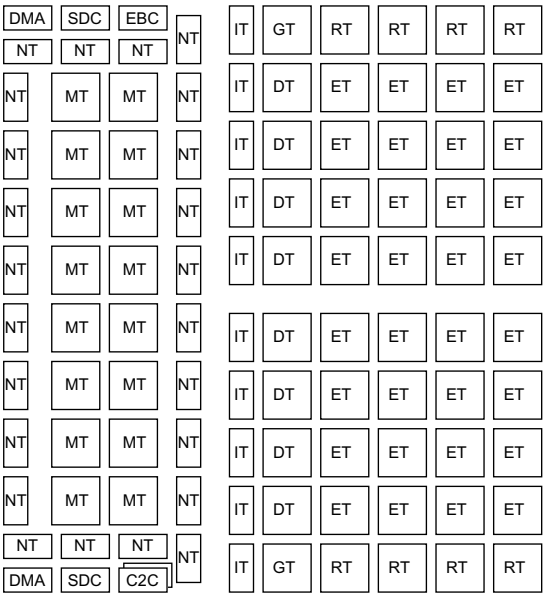

Figure 3-2:Homogenous TRIPS Multiprocessor System-on-Chip[1]

1. *Source: Wikipedia, Wikimedia Commons, licensed under CreativeCommons-License by MovGP0, URL: http://creativecommons.org/licenses/by-sa/2.0/de/legalcode*

3.1.1 Related Work

The research field of NoCs is broad and has stimulated numerous efforts resulting in a large number of publications. Fundamental contributions on NoC architecture and design methodology has been contributed by Kumar, Holsmark and Jantsch [83][76][111][65]. The authors have studied NoCs extensivley, in particular, homogenous Mesh architectures and their design methodology, including topologies, routing algorithms, and applied quality of service policies.

Important work has also been contributed by Benini in the form of the definitive book "Networks-on-Chips" [15], his work on NoCs for SoCs and the XPipes project. Xpipes [37][17] is an advanced NoC architecture, that targets high performance and reliable communication for on-chip multi-processors. It consists of a library of soft macros (switches, network interfaces and links) that can be composed and tuned at design-time, such that domain specific heterogeneous architectures can be instantiated and synthesized. It features the XpipesCompiler which takes a design description file as an input that defines the NoC's topology, clock speed and pipeline stages for each link to generate an application specific network described in SystemC.

A design flow for application specific NoCs to accelerate SoC Design and Verification has been introduced by Goosens in the form of the Aetherial NoC [58][59]. The Aetherial NoC has been commercialized by NXP and provides a tool flow to generate RTL NoC descriptions from an abstract XML description. It targets SoC platforms and features a compiler, verification and simulation capability of the synthesized NoCs.

Coppola has introduced the Network-on-chip Modeling and Simulation Framework OCCN [35] which has been adopted by STMicroelectronics in the form of STNoC [110]. It implements a layered approach to decouple the different components of the NoC methodically including the network interface, the router and the physical layer. Pre-configured IP blocks which implement a specific functionality can be assembled to form complete NoCs. Another contribution by Coppola is the Spidergon topology [36] for NoCs which targets low cost SoCs. The topology uses a ring bus with additional links as short cuts to provide good performance with a minimal amount of links.

Beigne has provided many important contributions in the field of Globally Asynchronous Locally Synchronous (GALS) NoCs [14]. GALS systems consist of communication endpoints that define their own synchronous clock domain and an asynchronous network to interconnect the local regions. The advantage of such a design is the simplified clock routing as clock domains do not cover the whole chip. Furthermore, the GALS approach allows to run each local clock domain at different clock speeds which can be used for coarse grained power management and to improve the yield rate of a chip by restricting the operating frequency of specific chip areas.

Although NoCs differ significantly from local area networks, many properties are similar and have been reinvented for on-chip networks. This particularly applies for the research on topologies and routing mechanisms whose issues tend to be similar as in local area networks. Most valuable research contributions, on the other hand, have been made in the area of SoCs in the form of the previously mentioned XPipes, Aetherial and STNoC projects. They provide a good solution for designing complex SoCs like mobile devices, however, they are not practicable for general purpose applications. Due to their SoC heritage they show high overhead and resource consumption as well as a limited application due to the restriction to proprietary protocols. A more lightweight approach that is easily extendable and provides the necessary flexibility to support the full range of applications is provided by the HTAX framework that will be introduced in this work.

3.2 NoC Design Space Exploration

Before the HTAX framework can be introduced it is necessary to perform a design space exploration of NoCs. Some basic principles of networks and on-chip networks will be introduced which are necessary to define the purpose of the framework.

3.2.1 NoC Topology

Networks can generally be classified into different groups according to their topology. The four common types are shown in Figure 3-3.

Shared Medium Networks consist of a single communication infrastructure called a bus which is shared by all communication endpoints. This simple interconnect structure requires an arbiter which guarantees that only one master can access the bus at the same time. Although, shared medium networks are a good fit for small networks they are not feasible for NoCs with a large number of endpoints due to their limited scalability. Simultaneous communication between multiple endpoints is not supported which limits the bidirectional bandwidth of the NoC, independent of the number of endpoints in the system. An advantage of bus based topologies is the capability of supporting broadcasts without additional hardware as each transaction on the bus is visible to every endpoint.

Indirect networks typically have a single central switch with an amount of ports equal to the number of connected endpoints in the system. To allow scaling to a large number of nodes, high radix switches are composed of multiple switch stages in the form of a Fat Tree or Clos topology [72]. Nonblocking indirect networks provide a very high bisection bandwidth and a low deterministic latency as the distance between all endpoints in the network is identical. In the area of NoCs, indirect networks are preferably used in small to medium sized SoCs as for large chips the distance between the far edges of the chip and the switch is too large.

In *direct networks*, each endnode implements an individual switch allowing it to directly connect to adjacent nodes. Depending on the number of links, 2D and 3D Tori as well as higher dimensional topologies can be realized. Most NoCs implement a 2D Mesh

topology which scales extremely well as it can be easily extended by adding additional rows and columns of components. Furthermore, NoCs provide lowest latency for nearest neighbor communication and fault tolerance against broken links as each component can be accessed by multiple routing paths.

Hybrid Networks combine attributes from shared medium, direct and indirect networks. Arbitrary topologies can be realized with Hybrid Networks using the different technologies provided by bus and switch based architectures. One example are heterogeneous networks that employ a hierarchy to cope with different bandwidth requirements of the endpoints. In this case low speed endpoints are connected to the system via an additional hierarchy to avoid communication performance degradation of the higher speed endpoints.

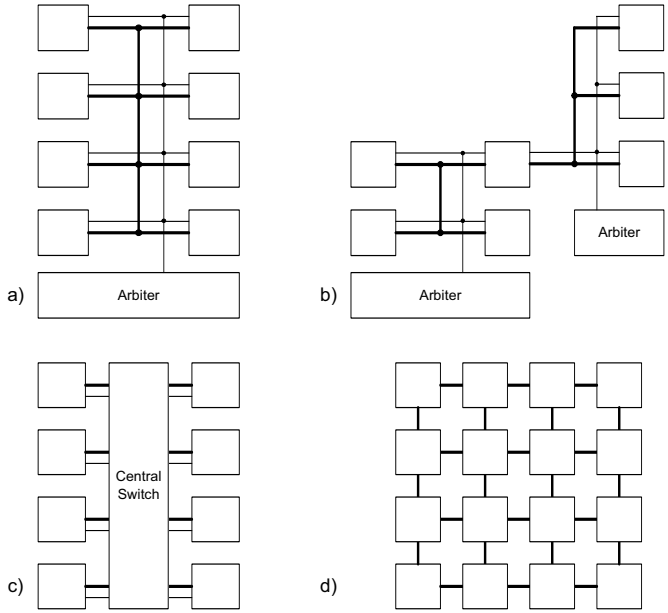

Figure 3-3:a) Shared Medium Network, b) Hybrid Network, c) Indirect Network d) Direct 2D Mesh Network

In the following, it will be focused on switch based topologies which provide higher scalability than shared medium networks and are, therefore, commonly used in current NoCs. As a result, the HTAX framework only supports direct, indirect and hierarchical topologies. For on-chip networks, the effort which is assigned to the interconnection structure continuously increases. Not only do the chips scale in terms of cores and modules, there exists the trend towards more and more heterogeneous chips including memory controllers, caches and accelerators. This results in a new set of topologies which are highly heterogeneous and combine both indirect and distributed characteristics. To support the different topologies the HTAX framework defines a component based design methodology. Each component implements a unique functionality as e.g. the switching capability or the buffer management. As a result, high radix indirect switches can be realized by combining multiple switch components in a multistage topology. Direct topologies are implemented by many switch components that are distributed over the complete chip.

3.2.2 Communication Protocols

The protocol defines all aspects that are required to enable communication between two or more endpoints. The properties which may be defined by the protocol include the supported topologies, the routing mechanism, the size and format of transactions, as well as the reliability and fault tolerance requirements. Similar to the different existing topologies, protocols can be classified into groups.

Traditional shared medium networks utilize transaction based protocols like the Advanced Microcontroller Bus Architecture (AMBA) or the Open Core Protocol (OCP). In general, a master initiates a transaction by requesting a grant from the arbiter to gain ownership of the bus. Now the slave target can be determined by driving the corresponding address onto the bus. As soon as the slave accepts the transaction, data can be exchanged between the master and the slave. The family of AMBA protocols is supported by ARM and represents the *de facto* standard for bus based SoC architectures. AMBA AHB represents the first specification released by ARM which supports master/slave shared medium networks with a central arbiter. The combination of a bus based

architecture with a relatively high arbitration overhead, however, restricts the performance and scalability of the protocol. To address this issue, multi-layer AHB has been introduced which effectively duplicates the bus structure to enable concurrent communication between multiple endpoints. Although, multi-layer AHB can increase the performance its scalability remains limited due to its high resource consumption. Systems, that implement a single master and, hence, require no arbiter can utilize the AHB-Lite protocol.

For switch based topologies, packet based end-to-end communication protocols have been proven successful. These protocols support concurrent communication between multiple endpoints and transport data in so-called packets. This technique not only enables multiple transaction streams but also multiple concurrent packets between two specific endpoints. Furthermore, read accesses can be implemented as split-phase transactions decoupling the read request from the read response. This allows to free network resources between transactions reducing overall congestion and bus contention. Examples are the AMBA 4.0 AXI and the Open Core Protocol (OCP). They support multiple outstanding transactions, out-of-order delivery and different transaction categories. Due to their pervasiveness in the SoC area, most NoC frameworks support at least one of the two protocols. The HTAX framework introduces a layered architecture which decouples the transaction layer protocol from the underlying NoC and its hardware. By applying this methodology, AXI, OCP as well as most proprietary protocols can be realized on top of an HTAX NoC. This approach provides an additional benefit, as only the features that require direct hardware interaction need to be mapped to the HTAX while higher layer functions are implemented by the endpoints, transparently to the underlying NoC.

3.2.3 Reliability and Fault Tolerance

Integrated circuits are inherently prone to errors. By scaling the operating frequency and, in particular, the voltage of ICs, the probability of soft errors (single-event upsets) increases constantly. Soft errors are caused by alpha particles within the chip material or by cosmic rays from space [13]. Components which have the highest failure rates are SRAM based memory blocks [28] and high speed interconnection structures. Depending on the target technology a specific error rate needs to be accepted and handled by the

system architecture. One possible solution is to protect data in SRAM memories through error correcting codes. A feasible technique are Hamming error correcting codes (ECC). Most commonly used approaches use an additional parity bit to offer single error correction and double error detection (SECDED) protection. If the probability for multi bit errors is extremely low this approach can transform an IC with a known bit error rate into a reliable system. The cost for increasing SRAM reliability in terms of resources are additional bits that have to be stored for each data word. Each word requires $\log(n)$ parity bits, where n is the length of the data word. The HTAX framework supports SECDED for all instantiated SRAM structures transparently to the surrounding logic.

On the nanoscale level, NoC interconnect structures are prone to errors due to crosstalk between global wires. CAD software that performs the layout process of the wires minimizes the failure rate by applying an extremely conservative approach which limits the operating frequency of the design. However, if errors can be corrected on a higher layer, a low failure rate on the transistor level is tolerable. This enables much higher performance and reduces the costs of manufacturing, verification and test. This motivates the analysis of error detection and correction mechanisms.

Different approaches have been proposed to increase NoC reliability. On the one side flooding schemes have been analyzed [47] which duplicates messages by sending them out over different routes simultaneously to increase the probability of a successful transaction. This simple solution requires few resources, however, entails a high bandwidth overhead which negatively effects performance. Vellanki et al. [128] have analyzed two different error control schemes for NoC architectures: *single error correction (SEC)* as well as *single error detection and retransmission*. The first approach applies an ECC technique while the latter stores the packets in a buffer and uses an acknowledgement scheme to enable retransmission of errorneous packets. It is shown that while SEC exhibits a marginally higher power consumption it is less complex and provides better performance, especially for systems that involve higher failure rates. The HTAX framework supports SEC protection throughout the complete switch fabric transparently to the endpoints. The HTAX fault tolerance scheme, therefore, provides security for SRAM blocks as well as for the interconnect structure.

3.2.4 Network Congestion

As networks are operated near their saturation point, network congestions appear which reduce the throughput of the network and subsequently limit the performance. The reasons for congestion are manifold. Application specific traffic patterns and fluctuating packet injection rates cause an unbalanced traffic distribution throughout the network. Very active regions called hotspots block the packet flow and induce congested areas. Similar to traffic jams, congested areas can spread and affect larger parts of the network over time. This results in an even greater negative impact on network performance.

The main issue of congestions which allows them to reduce the throughput of the complete network is the head of line (HOL) blocking effect. HOL blocking appears if a packet that is heading towards a congested region blocks other packets that would be otherwise unaffected. Due to this effect a congestion tree emerges that may spread over the whole network and that reduces the throughput by up to 58% of its peak value [79]. To address the appearance of congestions two approaches have been pursued. One costly solution in terms of resources and power is to oversize the network to be able to cope with hotspot traffic. More effective approaches are source throttling techniques [125] which dynamically reduce the injection rates at specific endpoints to avoid congestions. The main problem of these techniques is that global communication between the endpoints is required to effectively determine when to throttle the injection rate at a specific endpoint. Although, throttling techniques reduce network congestions effectively, performance gains can only be achieved in a well coordinated system.

More recent approaches as proposed by Duato *et al.* [46] focus on avoiding HOL blocking instead of the congestion itself. Solutions like the RECN mechanism [53] or Virtual Output Queuing (VOQ) [79] utilize additional buffers to reorder the packet flow. This enables non-congested packets to bypass congested packets which limits the HOL blocking effect. Although, buffer space is costly, these techniques allow to operate the network closer to its saturation point resulting in a considerable increase in performance. An interesting observation is that both the RECN and the VOQ approach share many aspects of the virtual channel technique. This fact is exploited by the HTAX framework to provide a congestion management scheme.

3.2.5 Virtual Channels

A general switch consists of a set of input ports, output ports and an interconnection matrix. The matrix consists of physical wires which connect the input ports to the output ports via multiplexers. A physical wire supports the transmission of one packet at a time. However, for various techniques, including congestion management and Quality of Service, it is desirable to send multiple traffic classes over the same link. Instead of duplicating the physical wires for this purpose, virtual channels have been proposed. Virtual channels are implemented by allotting multiple sets of queues in the input and output ports. The inport and outport buffers are usually implemented as FIFOs and are used to enqueue packets which cannot be immediately routed by the switch. In general, each inport implements a single queue. Virtual channels, in contrast, deploy multiple buffers in each port. This allows to transmit different packet streams “concurrently”. In effect, only one packet can be transferred over a physical wire at a time, however, the input appears to support multiple virtual channels. The advantage of this technique is that only the memory structures have to be replicated as the amount of ports and wires in the switch matrix remains constant.

The application area of virtual channels is manifold. Many protocols use virtual channels to distinguish different traffic classes, for example, read and write transactions. Depending on the application protocol this may be necessary to avoid deadlocks and to enable specific synchronization mechanisms. Another technique is to assign different priorities to the virtual channels which can be used to implement Quality of Service.

As the different virtual channels are managed independently, packets in unblocked virtual channels can bypass packets in blocked virtual channels reducing HOL blocking. This technique can be extended with additional buffers to support virtual output queuing which also allows to bypass packets that target different outports. Both virtual channels and VOQ can be combined and implemented by using the same buffering technique.

Another useful application of virtual channels is deadlock avoidance in wormhole-switched direct networks. In wormhole-switched networks packets are split into smaller parts (flits) which can be transmitted individually. Wormhole switching provides lower latency and fewer buffer requirements than the *store and forward* technique which always

receives complete packets before retransmitting them. However, as in-flight packets are distributed over multiple nodes, cyclic dependencies between these nodes can appear in Meshes and Tori. Dally and Seitz propose a solution for this problem by switching the virtual channel of the flits at specific locations in the network [38]. The manifold application possibilities of virtual channels are supported by the HTAX framework.

3.2.6 NoC Scalability

To determine the scalability of a NoC architecture, four different aspects need to be analyzed. NoCs need to be able to scale in terms of bandwidth, latency, operating frequency and their resource consumption. The bandwidth issue can be addressed by applying switches and point-to-point links instead of shared-medium buses. Using this technique, each additional endpoint increases the intersection bandwidth linearly resulting in a constant bandwidth per port ratio for arbitrary network sizes.

The transmission latency depends on the NoC topology and the utilization of the network. When the saturation point of a network is reached, packet latency increases significantly as congestions appear. Solutions to this problem are to modify the traffic patterns and the message injection rate as well as to oversize the network capacity. Furthermore, a congestion management technique can be applied. While the impact of network utilization is highly dynamic the effect of the network topology is static and well defined. In direct networks multiple switches are deployed in a Mesh structure resulting in moderate latency increase for each hop. The latency for nearest neighbor communication remains constant for arbitrarily large networks while the far-end communication latency increases with the square root of the size of the network. For indirect networks, in theory, the latency remains constant for all network sizes, however, due to timing issues multistage switches need to be deployed which lead to a moderate increase in latency.

Indirect networks that implement a single central switch exhibit a limited scalability on the physical level. High radix switches require arbitration logic at each outport whose complexity increases quadratically with the number of ports. This fact limits the degree of the switch to a number of up to 8 to 16 ports if high operating frequencies need to be supported. A solution for this problem is the support of multistage or hierarchical

switches. Both approaches partition the switch into smaller subswitches to reduce complexity and to alleviate timing closure. While this approach requires intermediate buffers between the switch stages which increases the latency, the additional pipeline stages also allow to distribute the switch over the chip reducing the length of global wires. Another way to increase the operating frequency of the arbitration logic is to insert additional intermediate pipeline stages. This increases the latency by a single clock cycle while improving the timing considerably. The HTAX framework supports multi stage and hierarchical switch topologies as well as pipelined arbitration logic. All three solutions do not affect bandwidth, providing 100% throughput.

Direct networks, on the other hand, show a good scalability technology-wise as the switches only need to implement a small number of ports. In order to increase the number of endpoints, replication of the switches is sufficient.

The occupied silicon area of NoCs is to a large part determined by the SRAM based buffers that are instantiated in the design. To address this issues the HTAX framework supports buffered as well as buffer-less NoCs. In many designs the endpoints already implement buffers which obviates the need for redundant memory cells. For performance reasons and to reduce head of line blocking, the framework supports input buffered switches with virtual output queuing.

3.3 HTAX: A NOVEL FRAMEWORK FOR NOCS

The HTAX framework we propose in [98] defines a general network-on-chip architecture for SoCs and MPSoCs. Its goal is to significantly reduce chip design and verification overhead for a wide range of applications. To satisfy the different requirements of the diverse hardware implementations the Layered Component Based Design Methodology is introduced.

3.3.1 Layered Component Based Design Methodology

Our proposed design methodology, shown in Figure 3-4, decomposes the several tasks into two dimensions. In the vertical dimension, the HTAX framework is divided into four layers which are the physical, network, transport and application layer. These layers provide specific functionality and are directly built on each other. While the network layer provides a collection of mandatory features which are common for all HTAX NoC implementations, the transport layer is partitioned into different components within the horizontal dimension. In this layer optional features provided by the framework are implemented that can be instantiated depending on the application protocol.

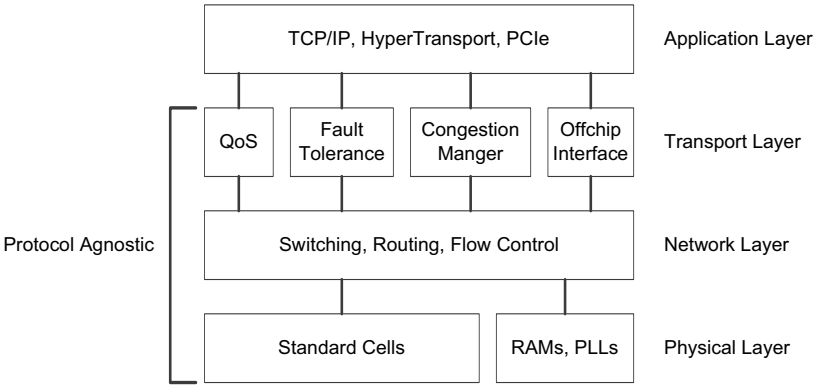

Figure 3-4: Layered Framework Architecture

Physical Layer

All components of the framework are implemented in the form of portable register transfer level (RTL) code that can be synthesized technology independent. We target a standard cell design flow and, in principle, support all FPGA and ASIC technologies. Specific building blocks like RAMs and PLLs which are required for the clocking infrastructure are technology specific. These blocks are encapsulated in a way such that they can be easily exchanged depending on the target technology.

Network Layer

The network layer provides the core functionality including switching and routing capability as well as arbitration, flow control and virtual channels. This common layer is shared by every HTAX instance. The complete functionality is hidden in the HTAX microarchitecture and transparent to the application which connects to the NoC through well defined interfaces. The interface as shown in Figure 3-5 shows a low complexity and supports advanced functionality through additional components in the transport layer. The interface distinguishes a transmit (TX) and receive (RX) part. It deploys control signals which implement a low level request/grant mechanism to control the access of the outports. To hide the arbitration latency, the optional *release_gnt* signal can be asserted to release a grant in prior. Data packets that are transmitted through the NoC are framed using the start of transaction (*sot*) and end of transaction (*eot*) signals.

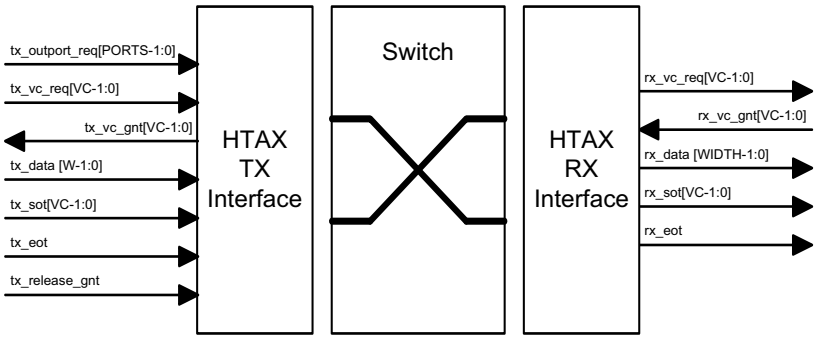

Figure 3-5:HTAX TX and RX Interfaces

Transport Layer

The Transport Layer implements higher level functionality based on the services provided by the network layer. All blocks within the transport layer are optional as the network layer itself already provides the necessary functionality of switching packets within an HTAX NoC. The Transport Layer offers a high degree of flexibility for the HTAX NoC designer by implementing a component based methodology. From a number of different components the user can choose a subset to extend the functionality as required. The supported components include buffering structures, congestion management, a capability for fault tolerance and the extension interface as described in the Design Components paragraph.

Application Layer

An important feature of the HTAX framework is its independence from higher level communication protocols. As the network and transport layer is protocol agnostic, the HTAX framework can be deployed in a large number of applications. More complex communication protocols including TCP/IP, HyperTransport or any proprietary protocol can be supported via the Application Layer. To offer the best flexibility, the framework supports all important features that are required by common protocols. By providing QoS, Virtual Channels and a reliable communication even heavyweight protocols can be supported.

3.3.2 Design Components

The HTAX framework provides a modular concept to build application specific NoCs with a high functional flexibility. Therefore, a component based design methodology is applied which provides pre-configured blocks that implement a different functionality. All components support the network layer interface so they can be plugged together without modifications. From a design perspective the components can be regarded as black boxes that can be combined without knowledge of their internal functionality. The approach facilitates the development of new design components as long as they adhere to the frameworks' specification. To give an example of an HTAX implementation, Figure 3-6

shows a highly heterogeneous central switch consisting of different design components. The switch utilizes a multistage topology. All arrows resemble the interface shown in Figure 3-5. The functional units (FUs) are interconnected by the switch via their transmit (TX) and receive (RX) interfaces. The switch incorporates both non buffered input ports as well as input queued ports to enable QoS policies and to reduce the impact of congestion. Specific modules are protected with error correcting codes (ECC) to improve fault tolerance. A real world implementation is likely to be configured more homogeneously, however the purpose of Figure 3-6 is to show the wide range of possibilities and the flexible concept of the architecture.

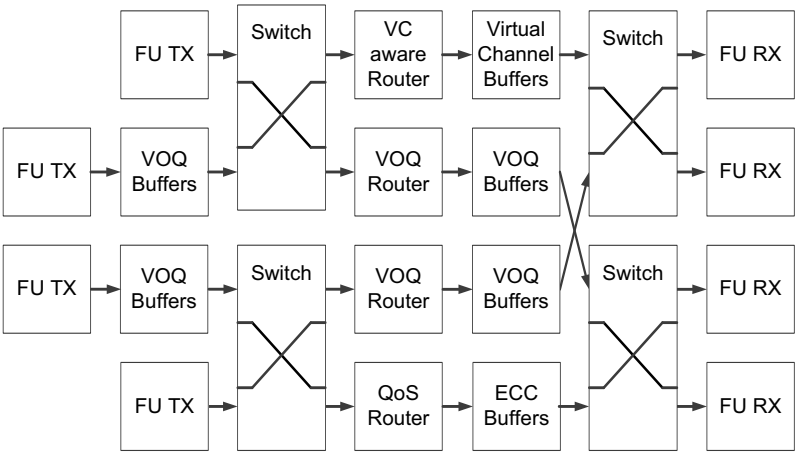

Figure 3-6:Heterogeneous HTAX Switch

Buffering Structure

Buffering structures are required in networks for different reasons. Their main task is to decouple computation from communication. Most applications do not generate constant steady traffic but tend to show bursty communication patterns in between the computation phases. To avoid mutual blocking of both tasks it is desirable to overlap the computation and communication task. This can be realized through buffers which are able to store a number of packets that cannot be immediately injected into the network. As long as the

buffers can accept additional packets, the processor can continue its work instead of being stalled by the backward flow control of the communication fabric. However, in many applications the communication endpoints implement buffering structures themselves. As SRAM based buffers are cost intensive it can be beneficial to avoid this redundancy. The component based technique of the HTAX framework supports a flexible buffer strategy to optimize resource consumption, latency behavior and throughput of the NoC.

Furthermore, buffers increase the scalability of switches as they enable non blocking multistage interconnection networks (MINs) [72]. MINs are implemented by cascading multiple smaller switches with intermediate buffering structures. Figure 3-6 shows a two level MIN. In principle, the HTAX framework supports arbitrary levels of switch stages. While MINs enable high radix switches that can operate at high clock frequencies, physical scalability remains limited as for non blocking MINs the silicon footprint scales quadratically. To address this problem hierarchical switches have been proposed which scale linearly. Hierarchical switches implement tree-like structures of switches in which the leaves represent the communication endpoints. They show indeterministic latency and bandwidth characteristics as the distance between nodes is variable depending on the number of levels that have to be traversed in the tree to reach the destination.

The buffering structure within an HTAX switch is transparent to the endpoints. Packets are transmitted over the same interface using the low level request/grant protocol. As long as the buffers can accommodate packets, requests are immediately granted. The inport then handles requesting of the actual outport and transmission of the packet.

Congestion Management

The HTAX framework applies a virtual output queueing technique on a switch level to limit the negative effect of congestions. VOQ reduces HOL-blocking by utilizing multiple buffer queues in each inport. Each inport instantiates a queue for each outport resulting in a total amount of

$$Q = NUM(inports) \cdot NUM(outports)$$

queues. Each incoming packet is stored in the buffer which maps to the corresponding output. Packets in different output queues can overtake each other which results in a

significantly reduced HOL blocking effect as long as each buffer has free slots to accommodate further packets. In multistage switch topologies or MINs there still exists the possibility of HOL blocking. This is due to the fact that packets targeting the same outport on the first switch level may target different outputs in the second switch level. If the first packet is blocked it will block the second packet which could, in principle, proceed. Complete HOL blocking can only be achieved by applying VOQ on the network level which requires a queue in each inport for every endpoint in the network. It is obvious that this approach does not scale due to its enormous resource requirements. However, it has been shown [79] that VOQ shows significant reduction of HOL blocking with acceptable costs. In contrast to other congestion management techniques, VOQ requires no propagation of congestion information throughout the network. Therefore, it can be implemented independently in each switch stage.

Multicast Capability

HTAX switches offer broadcast and multicast capability. Multicasts can be used to send a single packet to multiple recipients concurrently. The switch hardware automatically duplicates the packets which considerably improves network performance. Instead of sending a number of p consecutive packets to n different endpoints a single multicast packet can be used. A requirement for multicasts is that all recipients are able to receive the packet. To avoid that a busy endpoint is able to block the multicast, buffering structures are utilized which accommodate duplicated packets until they can be delivered to the receiver. An inport implements multiple buffer queues, one for each existing outport. The structure is similar to the one used for VOQ which allows to combine VOQ congestion management and multicast capability. A multicast is triggered by the sender by requesting multiple outports concurrently. If all requested buffer queues for the corresponding outputs can accommodate the packet, the request is granted. The packet is received from the sender by the input and inserted concurrently in multiple buffers. After the packet has been duplicated the queues start requesting the corresponding outport individually. A blocked outport, therefore, cannot stall the other packets within the multicast. Multicast reception is not synchronous which implicates that the targets can receive the multicast packets at different times.

Quality of Service Support

QoS enables applications to transmit information in different traffic classes to prioritize specific types of traffic. QoS is useful to guarantee latency boundaries for high priority packets combined with a best effort mechanism which maximizes the throughput of the network. To implement QoS the HTAX framework utilizes virtual channels. Each virtual channel is mapped to a specific traffic class which enables to transmit packets within different channel between two endpoints. Packets in different channels can bypass each other, which allows the high priority packets to reach their destination as fast as possible. The framework supports two different policies to implement QoS.

In the privileged approach the bandwidth is unequally divided between the data streams. According to their priority, a relation is defined on the different virtual channels. An exemplary architecture may define three virtual channels with a 3:2:1 relation. In this case packets traveling in the first virtual channel are assigned three times as much bandwidth as packets that travel in the third virtual channel. Due to the higher bandwidth a similar decrease in latency can be achieved for class one packets. To maximize throughput, bandwidth is assigned dynamically. In the case of no outstanding packets in virtual channel one and two the complete bandwidth can be utilized by channel three.

In the exclusive privileged approach, only two different channels are supported while one virtual channel is strictly favored over the other. As long as there exist packets to send within the privileged channel they are always preferred. As this can cause starvation of the low priority channel the application has to enforce idle times on the high priority channel.

The QoS implementation utilizes multiple buffers, one for each virtual channel. As long as there exists buffer space, packets are immediately accepted and stored into the queue according to the requested virtual channel. The inport contains a prioritized arbiter which, depending on the policy, chooses packets from the queues and requests the corresponding outports. The best effort approach of the switch is subordinated to the priority scheme which can cause HOL blocking of low priority packets if the high priority packets are stalled.

Burst Capability

The HTAX switch supports a burst mode to reduce arbitration overhead and to increase throughput for small packets. When sending minimum-sized packets over the HTAX, bubbles are introduced through the delay of the arbitration mechanism. To address this issue the HTAX supports a burst mode. A single grant can be used to transmit multiple packets consecutively. Each packet is framed by start of transaction and end of transaction signals so the packets can be differentiated from each other. The grant, however, is only released by assertion of the *release_gnt* signal. The endpoints need to define a maximum burst size (MBS). If a grant is given by a receiver it must be capable to receive multiple packets as defined by the MBS. The endpoints may support burst capability only for small messages to reduce buffer space. The definition of the MBS is part of the application protocol.

Excellerate Interface

The HTAX framework delivers a scalable technique for on-chip networks. However, scalability is at least limited by the physical size of the chip. Due to the manufacturing yield which is inverse proportional to the chip size, small chips are much more cost efficient. For this reason multichip modules (MCM) as well as multichip PCBs have been proposed. The proximity connection approach [48], for example, connects large amounts of chips over a high speed interface, using capacitive coupling, to form a 2D grid of ICs. To enable multichip approaches the functionality of the NoC has to be extended to support off chip communication. Due to the different characteristics of off-chip transmission lines, additional features need to be defined for such an interface. A physical layer is required which defines driver and receiver to enable data transmission over the PCB or the MCM substrate. As the distance between the chips is relatively small, communication latency and bit error rate is limited. This allows to define a protocol with much less overhead compared to wide area network protocols like Ethernet and InfiniBand which can tolerate much wider distances between the endpoints.

The HTAX enables off-chip data transmission via the Excellerate interface. Excellerate is a lightweight protocol that provides transparent communication over chip boundaries. Excellerate hides the additional PHY and protocol overhead completely and behaves

exactly like the HTAX on-chip interface despite some marginal increase in latency. This feature avoids any modification of the components regardless of the applied physical interface and, thereby, maintains the modularity and flexibility of the HTAX framework. Figure 3-7 shows two chips which both implement an HTAX based NoC interconnected by the Excellerate interface. Components that reside within one chip can access components on the other chip as if they were interconnected by a single NoC. This architecture provides especially interesting possibilities for FPGA-to-FPGA or ASIC-to-FPGA tandems. In this case, the reconfigurability of the FPGA can be exploited to exchange components conveniently even at runtime by applying the partial reconfiguration capability of state of the art FPGAs [43]. Besides tandem configurations, many-chip topologies can be implemented by utilizing multiple Excellerate interfaces.

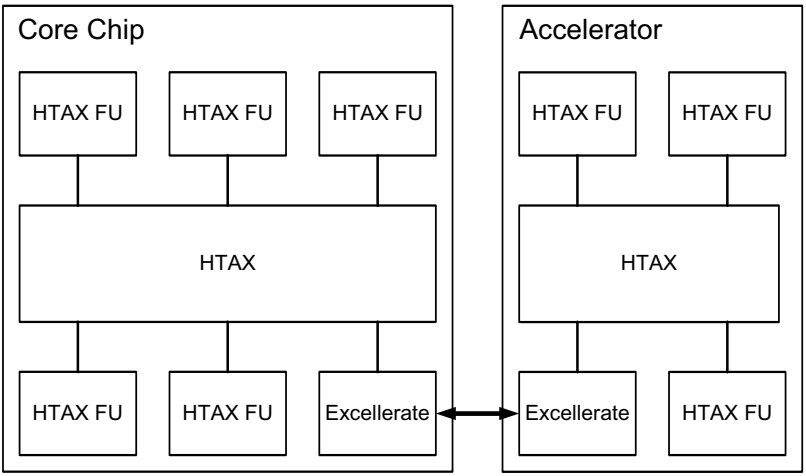

Figure 3-7:Excellerate Interface for Interconnecting Multiple Chips

For seamless integration into an application, Excellerate implements the standard HTAX interfaces as defined in Figure 3-5. To enable sending HTAX packets to remote Excellerators, HTAX packets are encapsulated into the EXTOLL off-chip protocol. EXTOLL is network protocol the enables routing of packets between different chips and,

therefore, represents a suitable solution to implement the off-chip interface. The Excellerate unit, shown in Figure 3-7, reuses many components of the EXTOLL protocol to provide virtual channel aware buffers, a credit based flow control over the Excellerate interface and fault tolerance capability. For this purpose, the HTAX packet is framed with two command characters, the EXTOLL start of frame and end of frame characters. The framed packets are then forwarded to the EXTOLL network port which adds protection by appending an ECC encoding to the command cells as well as a 32 bit packet CRC to the HTAX data frame. The packet CRC adds less overhead than the ECC mechanism, although it can only detect errors instead of correcting them. For data packets this drawback is tolerable as they can be simply retransmitted in the case of an error. Command frames, however, need to be correctable as corrupted command characters can put the system into an unpredictable state. To enable resending of erroneous data packets the EXTOLL link port implements a retransmission buffer which stores all packets until they are acknowledged by the receiver. If multiple internal or external Excellerate ports are required an optional EXTOLL X-BAR can be deployed which switches between the different ports.

The goal of the Excellerate interface is to act as a transparent channel, however, the additional functionality increases the request-grant latency preceding any communication within the HTAX NoC. The problem of a higher arbitration latency is that it causes bubbles between two packets in the data stream. Back to back sending is then rendered impossible which effectively decreases the interface bandwidth. To prevent these negative effects a credit based flow control mechanism is applied which virtually gives out grants in prior to the transmit side. Additional buffer space is provided on the receiver side to accommodate packets for which credits have been released. Credits are controlled on a per virtual channel basis by the link port. It controls a remote credit counter for every virtual channel which is permanently updated by credit information packets sent by the remote side. Based on the credit information, it controls the HTAX TX interface to guarantee that only packets with available buffer space on the remote are accepted. Although, supporting multiple virtual channels consumes resources in terms of buffer space and logic, head of line blocking between channels is removed which is mandatory for efficient off-chip communication.

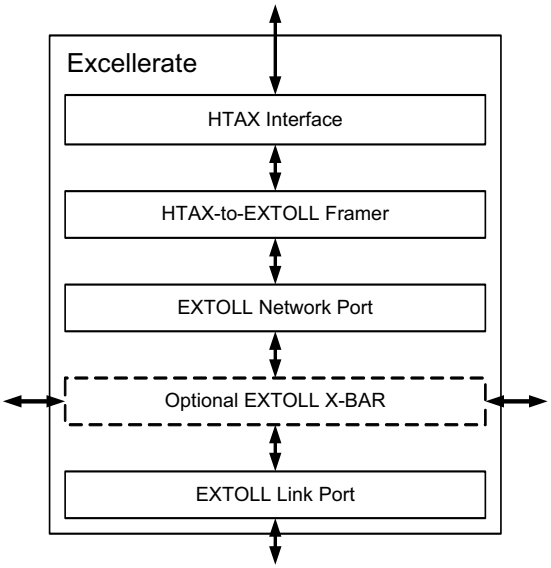

Figure 3-8:Excellerate Functional Unit

The next module is the arbiter which multiplexes the HTAX data stream and the credit information onto a shared link before providing it to the physical layer interface. The PHY can be subdivided into a transmit path that drives the data stream onto the physical wire and the receiver which samples the data and forwards it to the Excellerate module. To reduce the amount of pins which are required at the chip level data is transmitted at both the rising and falling clock edge using the double data rate (DDR) technique. The transmit clock is derived from a common external clock which allows to operate the two endpoints synchronously. The synchronization point between the link clock domain and the chip's internal clock domain resides in the PHY and is implemented by a FIFO structure which can compensate for the phase offset between the read an write clock. As the main application for Excellerate is to enable chip-to-chip communication over PCBs the requirement of synchronous operation between the ICs is not a serious limitation as both can be sourced by the same input clock.

Reliability and Fault Tolerance

Each queue that is instantiated through one of the HTAX building blocks can be configured to support ECC protection. The costs are additional bits required to be stored in the memory structure while improving fault tolerance significantly. Each data word retrieved from a memory structure is directly checked for errors and single bit errors are automatically corrected.

Another fault tolerance mechanism is the automatic retransmission scheme. It requires bidirectional communication between each endnode in the network and applies additional modules to the transmit and receive paths. The transmit module is responsible for calculating a CRC which is attached to the end of a packet and to store the packet in its retransmission buffer. The receive module checks the CRC and if the result is successful, generates an ACK packet otherwise rejects the packet and issues a NACK. On receiving the ACK the transmitter removes the packet from the retransmission buffer and in case of a NACK the packet is retransmitted.

3.3.3 HTAX Design Flow

In this section the design flow for the generation of HTAX NoCs is presented. An electronic design automation (EDA) tool suite has been developed that supports different target technologies. The presented technique alleviates NoC design significantly and enables a rapid prototyping approach as new NoCs can be generated in a time efficient way. Furthermore, correct functionality of the NoC can be guaranteed by means of the HTAX verification environment. To enable automatic generation of an application specific NoCs, the user provides a description which is then transformed into a gate level netlist by the tool chain. The HTAX design flow, as shown in Figure 3-9, is comprised of four main components which are the HTAX Configuration Wizard, the HTAX NoC Compiler, the HTAX Verification Environment and the Backend Synthesis.

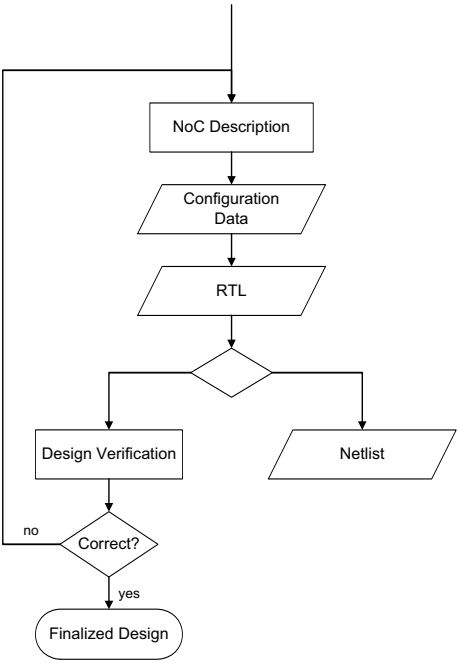

Figure 3-9:HTAX Design Flow

HTAX Configuration Wizard

The HTAX Configuration Wizard provides a convenient solution to generate HTAX NoC descriptions. By utilizing the graphical user interface as shown in Figure 3-10, all necessary properties of the NoC can be defined. Currently, the number of endpoints, the number of virtual channels, the amount of switch stages, the buffering technique and the width of the data bus including ECC protection can be selected. Multistage NoC topologies are as well supported as QoS requirements.

The HTAX Configuration Wizard produces a configuration file that uniquely describes the designated NoC. This configuration file is then handed over to the HTAX NoC Compiler to generate the actual design files.

Figure 3-10:HTAX Configuration Wizard

HTAX NoC Compiler

The HTAX NoC Compiler transforms the HTAX description as defined in the configuration file into synthesizable RTL code. It is implemented in the form of a Pearl based scripting engine that utilizes two different techniques to generate the designated output. The first technique exploits the component based design methodology to instantiate pre-configured IP blocks in the design as requested by the user. Examples are the arbiter IP block and the buffering structures that are used in a specific NoC. To reduce the amount of pre configured blocks that have to be supported, all IP blocks are parametrizable. The NoC Compiler is aware of the IP block attributes and configures them accordingly.

The remaining parts of the design are generated by the tool directly. This includes the logic to interconnect the instantiated IP blocks as well as the multiplexers and the switching matrix which is the central part of the NoC. The RTL code that is generated by the compiler is optimized for area and speed which allows the backend process to generate highly efficient NoC implementations. The quality of the generated code and the performance of the designs is comparable to hand written HDL code as shown in the evaluation chapter. In addition to the actual design, the tool generates a toplevel file which contains the module instantiations as well as the NoC specific verification environment. This verification environment provides the tools to test the design under different conditions and to verify its correct behavior.

Verification Environment

To prove the correct functionality of a specific HTAX NoC implementation, a verification environment was developed. The major challenge of the verification environment is to completely support the modular feature set of the HTAX framework. It is not sufficient to verify correct behavior of a number of designs and then deduce correctness for all HTAX NoC instances, as designs tend to increase in complexity and susceptibility to errors with their size. Therefore, the verification environment is completely configurable to support any HTAX switch that can be generated with the framework. The NoC generator tool has been customized to automatically generate a verification environment that fits the corresponding design.

The verification environment is based on the Open Verification Methodology (OVM) from Cadence which defines a general architecture for large verification designs. It is an open source product and runs under the Apache License 2.0. An OVM verification environment consists of several different Open Verification Components (OVC) including module, interface and system OVCs as shown in Figure 3-11. The main tasks of the verification environment are to provide mechanics for interaction with the Design Under Test (DUT) via the interface and to ensure the connectivity of all components and subcomponents. Each component is defined in the most general way using object oriented programming techniques to create OVCs with a high degree of reusability. Interface OVCs for example abstract the DUT specifics to provide a general interface for higher level OVCs. Other DUTs can then be imported into the verification environment by solely modifying the interface OVC.

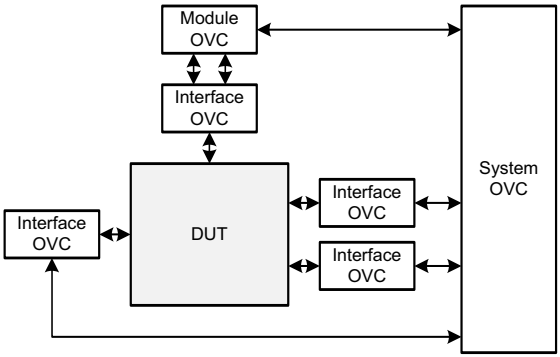

Figure 3-11:Open Verification Methodology Architecture

Interface OVCs define an environment which includes so-called *agents*. An agent in turn can contain *sequencers*, *monitors* and *drivers*. Sequencers generate sequences or stimuli that can be driven into DUTs by a specific driver. The driver also provides feedback to the sequencer to be able to react to the behavior of the DUT. Finally, monitors are connected to drivers to capture the so-called sequence items that are driven into the DUT. Figure 3-12 presents the layout of the HTAX transmit component (TX-OVC). It contains the HTAX master agent which defines a single driver, monitor and sequencer. In

the HTAX verification architecture each TX-OVC is responsible for a single HTAX port. In the case of an eight port HTAX there exist eight instantiated TX-OVCs as well as eight additional receive components (RX-OVC). The modular architecture of the OVM allows to easily scale the number of OVCs and interconnect them in the toplevel environment.

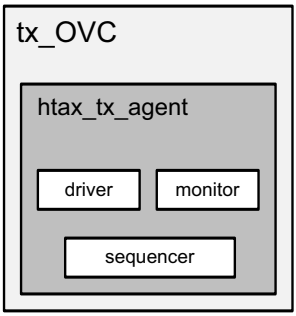

Figure 3-12:HTAX Transmit OVC

The sequencer within the TX-OVC generates HTAX packets in the form of sequence items which are driven into the DUT by the driver. The sequencer is a highly configurable block which can generate any legal HTAX packet description either in a user constrained form or completely at random. The monitor which connects to the driver captures the HTAX packet and stores it into a data structure, referred to as the *scoreboard*. The RX-OVCs which are connected to every receive port of the HTAX implement a monitor which captures received HTAX packets and compares them with the entries in the scoreboard. This approach guarantees that every packet is checked for data integrity as well as every packet is received at the correct outport using the proper virtual channel.

Verification of the correct virtual channel management behavior is a complex task for the proposed OVC. The driver of the TX-OVC requires the ability to request multiple virtual channels simultaneously and to react to the virtual channel specific grant signals of the DUT accordingly. Therefore, the sequencer needs to generate multiple packets at once that target a single outport using different virtual channels. For this purpose, a novel data structure named *sequence item pool* has been developed which is shown in Figure 3-13.

The sequence item pool contains HTAX packets that target a single outport using multiple virtual channels. It contains a separate queue for each virtual channel that can hold multiple HTAX sequence items. The sequence items can be enqueued at arbitrary times and the driver immediately requests all virtual channels that contain at least one entry. Only if all items of the pool are driven into the DUT, the item pool is deallocated and a new item pool targeting a different outport can be installed. The driver itself is a multithreaded entity in which one thread is responsible for requesting the outport and the virtual channels for which packets are available. Running in parallel, the packetizer thread interprets the grant signals provided by the DUT. It matches the existing requests and grants before retrieving the according packets from the pool to drive them into the DUT.

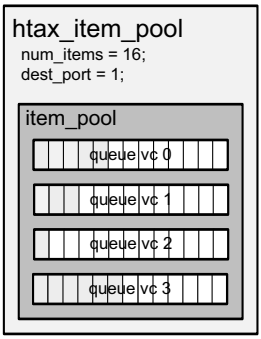

Figure 3-13:Item Pool Datastructure

The scoreboard of the verification environment has the task of comparing packets transmitted through the TX-OVC with packets received by the RX-OVC. For this purpose, it manages different queues, one for each outport. Packets which are injected by the transmit side are inserted into the corresponding queue, depending on their target outport. Ordering is maintained within virtual channels for a specific outport and checked for correctness. As packets of different virtual channels are allowed to be reordered in time, the outport queues contain subqueues for each virtual channel. This functionality is implemented through a datastructure that reuses the concept of the item pool as shown in Figure 3-14.

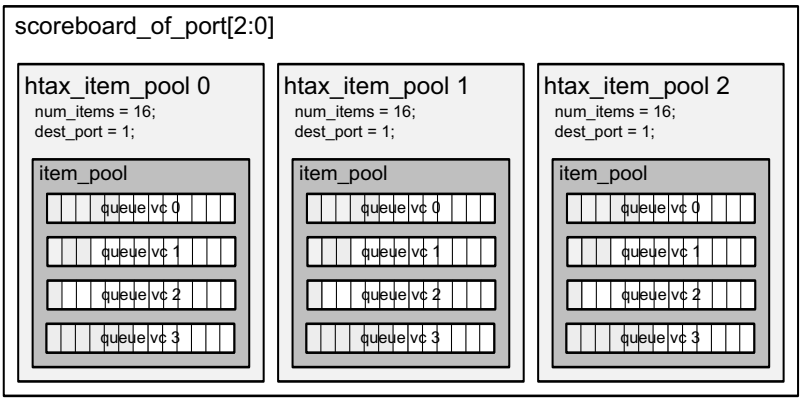

Figure 3-14:Multi Queue Scoreboard

A complete HTAX testbench is created by combining TX-OVCs, RX OVCs and the scoreboard unit. As the amount of required OVCs as well as their parameters are dependent on the design under test, the instantiation and customization of the OVCs is handled by the HTAX development environment tool. An example environment for a 2×2 HTAX NoC is shown in Figure 3-15.

The main purpose of the presented verification environment is to verify the correct functionality of HTAX implementations. The environment exhibits a high degree of flexibility to support any type of HTAX NoC that can be generated with the framework. In addition, the verification environment provides the necessary tools to extend its area of application. It can be utilized to verify the correct behavior of modules that are attached to the HTAX NoC. By reusing the monitors connected to the HTAX interfaces it is possible to observe the interface compliance of these modules. Furthermore, the provided driver OVCs can be utilized to generate test stimulus for any module that supports the HTAX interface. The HTAX verification environment has been successfully applied to a number of NoC designs including high radix implementations with a size of up to 64×64 ports.

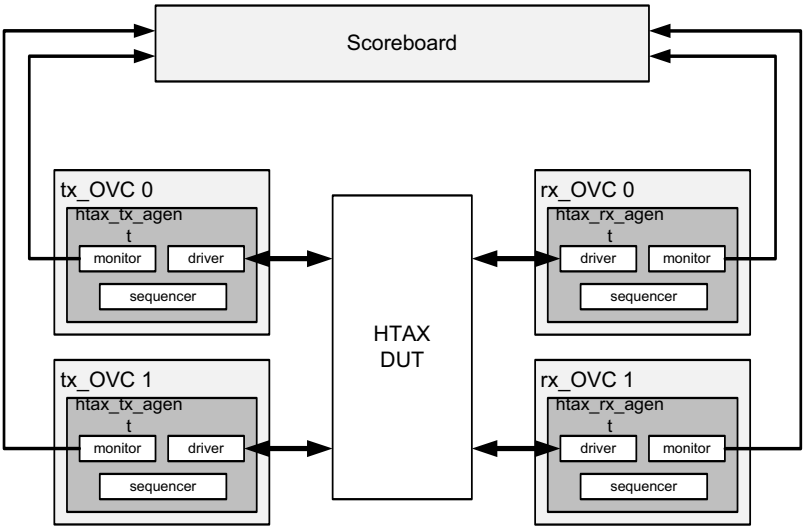

Figure 3-15:Complete HTAX Verification Environment

3.3.4 Microarchitecture

The output of the HTAX design flow is highly optimized RTL code. In combination with a synthesis tool like the Cadence RTL Compiler a gate level netlist can be produced which comprises standard logic cells and registers. Figure 3-16 shows the pipeline structure of an HTAX switch implementation. The transmit unit is depicted on the left while the receive unit is shown on the right. The HTAX is a two pipeline stage design in which the first stage is required to compute the grant signal and the second stage to route the data through the switch fabric. The switch architecture is pipelined to overlap the request-grant latency. This allows bubble-free back-to-back data transfer of packets with maximum bandwidth. To relax timing closure a third pipeline stage can be inserted into the critical path which resides in the arbitration logic of the outport. In this case the operating latency of the switch can be increased by approximately 30% while increasing the propagation latency to three cycles. In addition to the pipeline structure, the diagram shows the control and data path and how they are routed through the module.

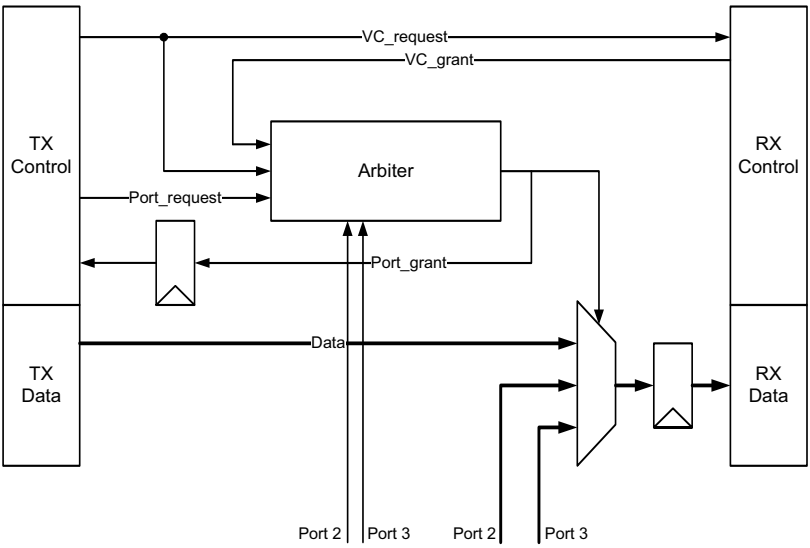

Figure 3-16:HTAX Pipeline Structure

A detailed view of the arbiter logic instantiated in the toplevel model is shown in Figure 3-17. The arbiter module, which is the central component of the switch, is a combinatoric logic block. It is implemented in the form of a discrete module to enable a modular design that supports to interchange the arbiter implementation. It can be utilized to support different arbitration algorithms or architecture dependant timing optimizations. In the example, a switch with three inports and three outports is presented. The request signal bus which can be seen on the left is comprised of 12 bits as the switch supports four virtual channels for each port. The request bus is masked with the virtual channels that are acknowledged by the receiver and then forwarded to the actual arbiter logic. The arbiter logic is implemented through a hierarchical sub block which can be seen on the bottom of the image. Incoming requests are combined with information about previous requests to calculate the grant result. Individual requests can be granted directly. In the case of multiple requests, a fair round robin arbitration scheme is used that is implemented by a binary tree search (BTS) mechanism in conjunction with a unit-weighted representation of the priority index [118]. The arbiter has been carefully designed as it is the timing critical component which defines the performance of the NoC. The result of the arbiter is then forwarded to a multiplexer structure as shown in Figure 3-18 which maps the incoming data busses onto a single outport data bus. Furthermore, the start and end of transaction signals which frame the packets are controlled by the multiplexer. The multiplexer structure instantiates output registers for each data and control signal.

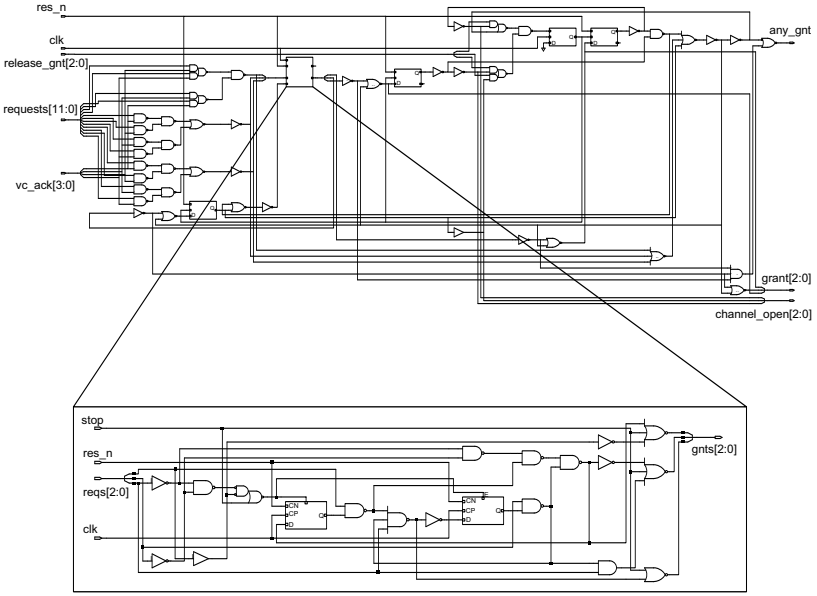

Figure 3-17:HTAX switch arbiter with three ports and four virtual channels

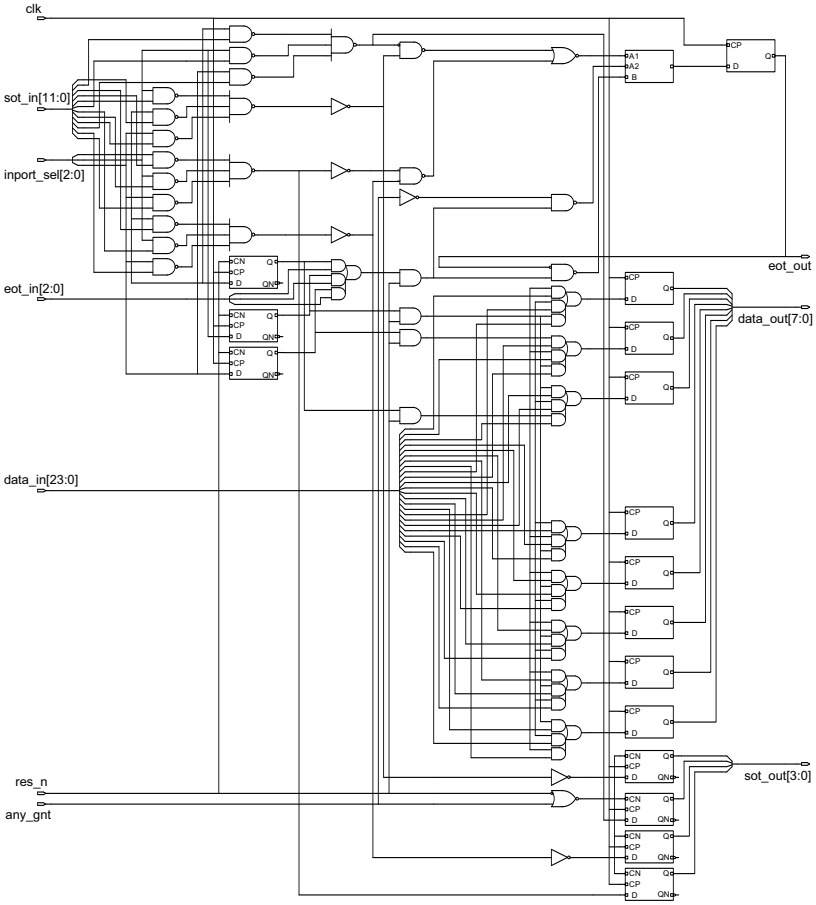

Figure 3-18:HTAX switch multiplexer structure

3.4 EVALUATION

The main goal of the HTAX framework is to facilitate the design and implementation of network-on-chip architectures. By applying the HTAX methodology, the design process can be significantly accelerated enabling faster time to market as well as less risk. In addition, by applying rapid prototyping methodologies, the quality and performance of a design can be improved. The possibility of generating synthesizable RTL code from an abstract description provides helpful information about the scalability, the performance and the resource consumption of the design already in the early stages of a design. This knowledge can be used to adjust the design parameters early, instead of in one of the later implementation stages. Also, by sweeping the NoC's parameters many design alternatives can be tested which enables a much broader design space exploration.

In many cases, automatically generated code is less performant and efficient than hand written code. To address this issue and to prove the feasibility of our approach in terms of performance, different switch implementations have been generated and evaluated. For our tests we have varied the number of endpoints, the datasize of the packets and the number of stages within the NoC. The designs have been synthesized using two different chip technologies, one is FPGA based and the other one ASIC based. To conduct the measurements, two different back end processes within the HTAX design flow were applied. As for the target FPGA technology the Xilinx Virtex5 LX330T device was chosen which is one of the most complex and sophisticated devices available. It provides 330.000 look up tables to realize arbitrary logic functions, 11 kbits of RAM, 24 high speed serial transceivers, 960 IO pins and 192 digital signal processing (DSP) slices. To transform the RTL code into a LUT level netlist the Xilinx ISE 10 synthesis tool has been used. As in general, RTL modules do not instantiate input registers static timing analysis for the input signals is incorrect. To address this issue, the designs have been embedded into wrapper files that implement a register for each input signal.

To confirm the obtained results, a second silicon technology has been evaluated, to be specific, the 65 nm TSMC standard cell ASIC library. This state-of-the-art technology enables high performance designs with up to billions of transistors, some prominent examples are the recent GPUs from Nvidia and ATI. The backend design flow for the

ASIC technology has been conducted with the Cadence RTL Compiler suite. To provide most significant results, we opted for a physical synthesis which performs intermittent placement and routing of design to improve the estimation of wire delays. This approach is computationally intensive, however, it provides much better results than other approaches using wire load models. In fact, we have compared the physical synthesis results with post placed and routed designs, and could not observe deviations of more than 3% regarding area and speed.

3.4.1 Single stage Switch Performance

The first test evaluates the maximum achievable operating frequency (*Fmax*) for different switch implementations targeting the Xilinx FPGA technology. The results are shown in Figure 3-19. The implementation shows an excellent result of 411 MHz for small switches with 4 ports and still a maximum frequency of 152 MHz for 64 port switches. Considering that most FPGA designs run at 100-200 MHz, this HTAX architecture appears to be a good fit for this technology.

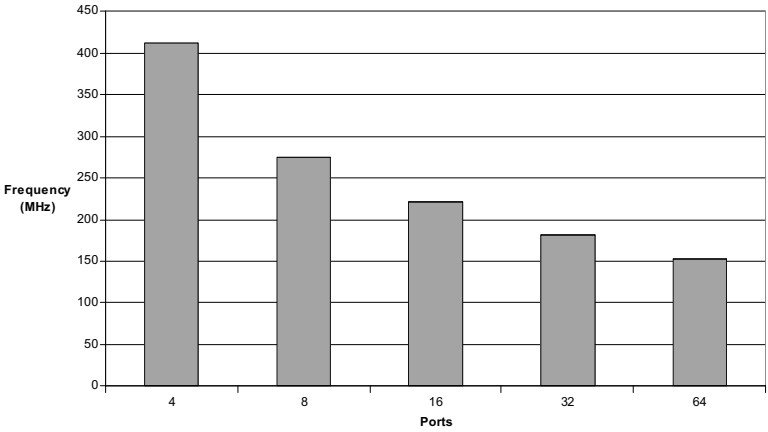

Figure 3-19:Maximum Operating Frequency of FPGA based HTAX Switches

The results of the ASIC switch implementations are shown in Figure 3-20. Four port switches can be clocked at 1 GHz, eight port implementations support up to 800 Mhz, while 32 port implementations still operate at 567 MHz. In contrast to the FPGA results which show a grateful degradation of performance, the ASIC performance decreases linearly with the number of ports. As a result four port switches operate more than twice as fast using the ASIC technology while 64 port switch implementations only show a performance increase of 60%. This suggests that the ASIC timing is influenced to a higher degree by the size of the module and the resulting wire delays. The switch latency can be determined by multiplying the cycle time with the pipeline depth of the architecture. This leads to the following results: 3 ns for the 4 port switch, 3.7 ns for the 8 port switch, 4.5 ns for the 16 port switch, 5.7 ns for the 32 port switch and 9.4 ns for the 32 switch. Correspondingly, the throughput of the architecture can be calculated. Based on a 128 bit wide datapath the port bandwidths of the switches are 15.9 GByte/s, 12.9 GByte/s, 10.7 GByte/s, 8.4 GByte/s and 5.1 GByte/s respectively.

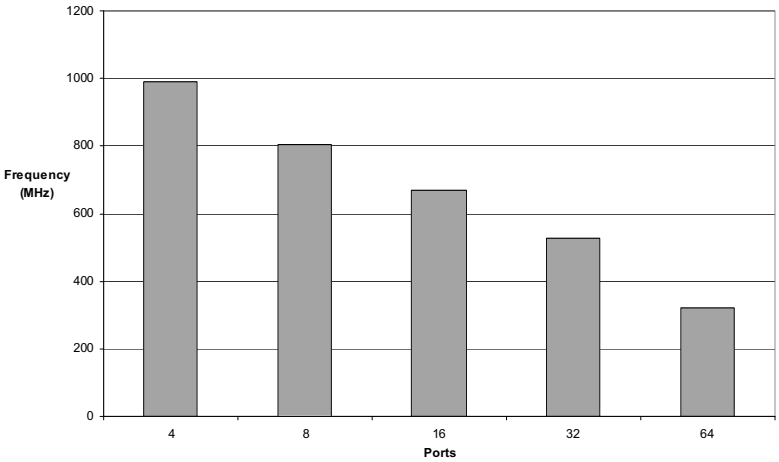

Figure 3-20:Maximum Operating Frequency of ASIC based HTAX Switches

3.4.2 Single Stage Switch Resource Consumption

The size in terms of transistors of different HTAX switch implementations is shown in Figure 3-21 and Figure 3-22. For the FPGA technology, the metric is the amount of six input lookup tables (LUT) as reported by the synthesis tool. Such a lookup table implements any logic function of six inputs and contains a register to store the result. The diagrams show results for 4, 8, 16, 32 and 64 port switches with supported data width of 32, 64, 128 and 256 bit on a logarithmic scale. As shown in the diagrams, the size of the switch implementations not only depends on the number of ports but also on their datawidth. However, while the datawidth increases the resource consumption by a linear factor of ~1.6, the number of ports effects the amount of resources exponentially. Doubling the amount of ports leads to four-fold increase of complexity. The reason for this exponential increase in complexity is explained by the fact that the logic within a port arbiter, as well as the number of arbiters, scales with the number of ports. The datawidth, on the other hand increases the complexity of the multiplexer structures and the amount of registers only linearly.

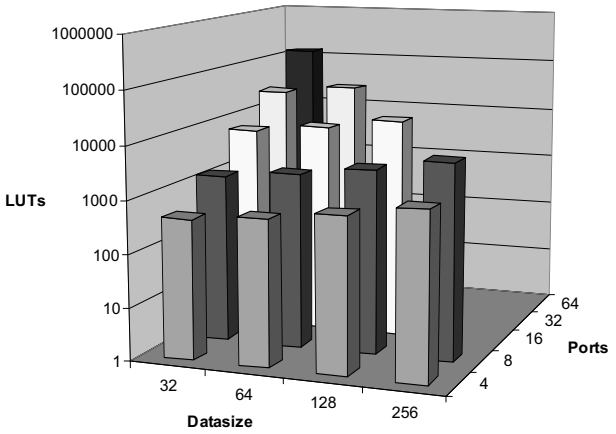

Figure 3-21:Single Stage Switch FPGA Resource Consumption

Figure 3-22 shows the number of transistor gates which are required for different switch implementations using the TSMC 65nm library. The reported numbers are higher for the ASIC than for the FPGA implementation as gates only realize simple NAND or NOR functions while lookup tables can implement arbitrary functions based on up to six inputs. However, the relative impact of the datasize and the amount of ports to the complexity of the switch is similar. While the datasize increases complexity by a linear factor of ~1.45, scaling the number of ports has the same exponential effect as in the FPGA technology case. In summary, while the gate count of small switches is insignificant, 64 port implementations contribute to the entire size of the design significantly. For large designs, the resource consumption in the area of 1 million gates represents a smaller issue than the limited operating frequency of such designs. To provide a solution for those architectures, the HTAX introduces multi-stage topologies.

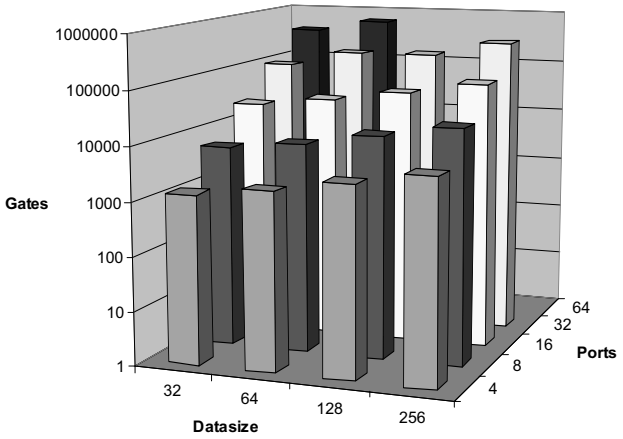

Figure 3-22:Single Stage Switch ASIC Resource Consumption

3.4.3 Multistage Switch Performance

The previous paragraphs show that high radix switch implementations exhibit a limited capability of supporting high frequency designs. Especially for designs that feature 32 and more endpoints it is desirable to develop a solution that supports higher frequencies. One possible approach is to deploy multistage switches. In this case, multiple smaller switches are cascaded in a topology to form a larger switch. The switch stages can be interconnected arbitrarily, which enables topologies including Benes[22], Banyan[22] and Butterfly[81]. In our case, we have applied a two-stage MIN composed of different switch components. The architecture utilizes $2n$ times $n \times n$ subswitches to compose a single switch of $n^2 \times n^2$ ports. Figure 3-23 shows examples for a 9 port 3×3 and a 16 port 4×4 implementation. In traditional circuit switched networks, this represents a blocking topology, however, as HTAX is a packet switched architecture, switches are non blocking by definition. The downside of this approach is that it requires packet buffers in front of each intermediate switch stage.

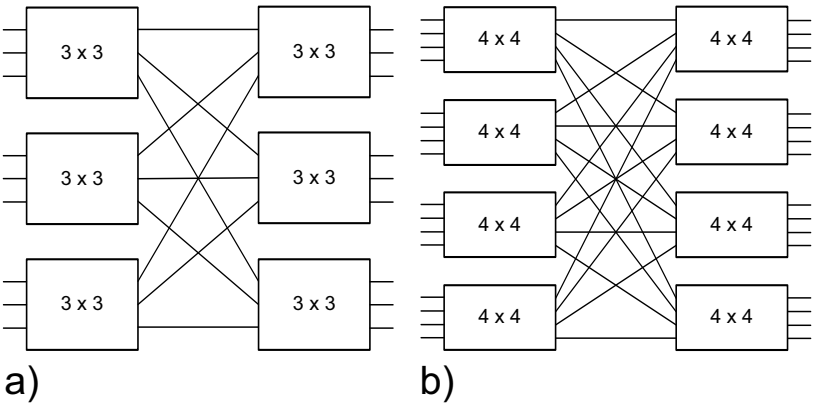

Figure 3-23:a) 9×9 *X-Bar, b)* 16×16 *X-Bar*

Figure 3-24 shows the operating frequencies that can be achieved with the multistage approach utilizing the 65nm ASIC technology. In contrast to the single stage switch architecture, the multistage approach provides a much better scalability in terms of operating frequency. Especially the large switches with 32 and 64 ports show a significant improvement of 778 vs. 520 MHz, and 713 vs. 320 MHz respectively. For the 64 port switch this represents a speedup of 125%. Due to its improved scalability, the multistage approach also enables larger switches, as a 256 port 16×16 switch is still capable of running at 660 MHz. The multistage approach, hence, appears to be a feasible solution for high radix switch architectures that need to support high clock frequencies. The pipeline depth of multistage switches increases from three to six clock cycles, however, the increased clock frequency compensates this disadvantage by reducing the latency of each stage. Therefore, the packet switching latency is 6.2 ns for the 9 and 16 port switches, 7.3 ns for the 25 port switch, 7.7ns for the 36 port switch, 7.8 for the 49 port switch, 8.4 ns for the 64 port switch and 9.1 ns for the 256 port switch. The per port bandwidth of the switch ranges between 15.2 GByte/s for the 9 port switch and 10.5 GByte/s for the 256 port switch. For all MIN switches, the datapath width was configured to 128 bits.

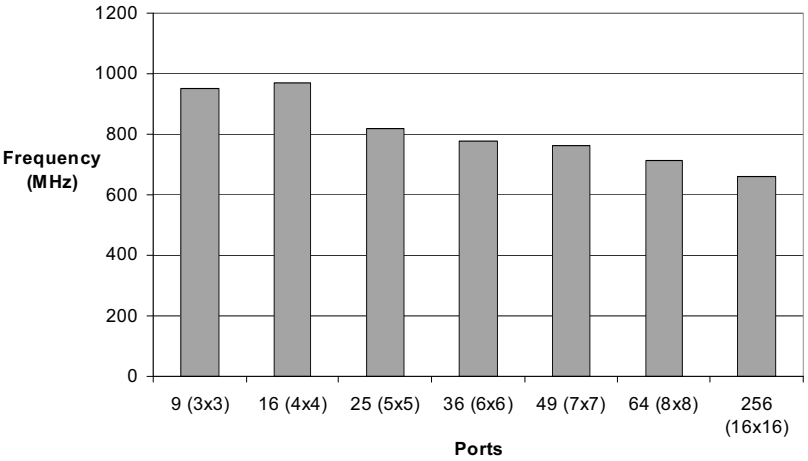

Figure 3-24: Multi-stage HTAX Operating Frequency

The multistage switch approach has proved advantageous for high radix switch implementations as is scales considerably better in terms of clock frequency. It needs to be analyzed how this approach performs regarding resources consumption to determine whether the performance gain is achieved on the expense of a larger design. Figure 3-25 shows the gate count plotted against the switch radix. A linear increase in resource consumption can be observed, in which doubling the number of ports leads to a two-fold increase in gates. Comparing these results with the singlestage switch solution shows that singlestage switches require significantly less resources for smaller switches of up to 8 ports. For switches of 16 ports and greater, the exponential increase in resources leads to larger designs for the singlestage switch approach.

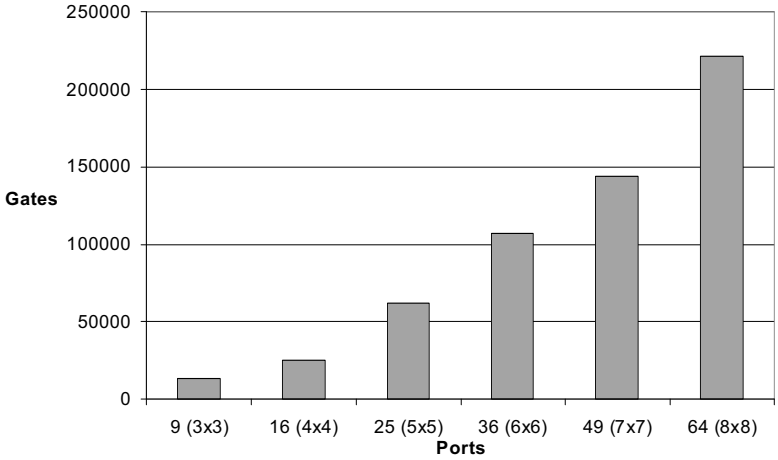

Figure 3-25:Multi-stage HTAX Resources

Until now, only the gate count of the switching and multiplexer logic has been considered for the area evaluation. However as described, the multistage switch approach requires intermediate buffers between the switch stages. These buffers add to the resource count and need to be embraced in the analysis. The size of the buffers depends on a number of properties, while the most important property is the maximum transfer unit size

of the packets, as the buffers need to be able to buffer at least one complete packet. Additional buffer space needs to be allocated for congestion management techniques like virtual output queuing. The size of the buffers, hence, depends on the employed communication protocol. For our analysis we assume a hypothetical protocol that uses an MTU of 512 bytes. To provide limited decoupling capability we allow to buffer two complete packets in the intermediate buffers. The protocol does not utilize virtual output queuing. As a result, an *n*-port multistage switch requires *n* buffers of 1 KByte in size. For the evaluation, the 65 nm TSMC library has been used, all results are reported in square um. As can be seen in Figure 3-26 the buffers contribute a significant factor to the combined area. While for this protocol both the logic cells and buffers contribute to a similar amount to the total area in other protocols which define larger MTU sizes or use multiple buffers per port the buffer area quickly starts to dominate. As a rule of thumb, duplicating the number or size of the buffers leads to a two-fold increase in silicon area. To draw a conclusion, it appears that multistage switches provide superior performance at a reduced cost in terms of gate count. However, it is important to note that the buffer space can considerably increase the area, especially in MINs.

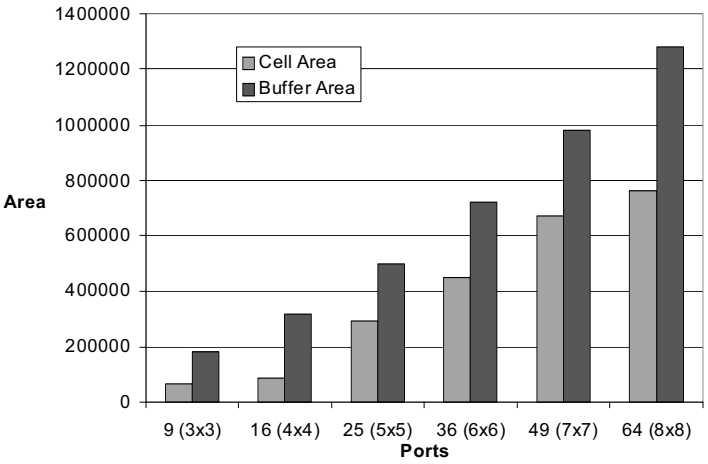

Figure 3-26:Multi-stage HTAX Gate and Buffer Area

3.5 Application Example: EXTOLL

The HTAX framework has been applied to a number of hardware designs. We will present one of these examples, the EXTOLL network interface adapter architecture which utilizes the HTAX NoC for interconnecting its on-chip modules. EXTOLL is a multi million gate chip implementation that currently exists in the form of an FPGA prototype and which is being manufactured as an ASIC device in 65 nm technology. EXTOLL is a network adapter that enables high performance computing systems by interconnecting large clusters of x86 processors. The architecture is designed to support very low latency communication and high bandwidth. It supports communication offloading primitives and device virtualization capability in hardware. Furthermore, it provides a significant performance advantage over competing network technologies including 10G Ethernet and InfiniBand.

3.5.1 Introduction to EXTOLL

EXTOLL is a modular, high performance architecture for network interconnects. It is based on multiple building blocks that can be combined to form a powerful communication device. The different building blocks implement all the required functionality to communicate over a network including packetization, routing, switching and to provide main memory access. The central component of the EXTOLL architecture is the HTAX NoC which interconnects the different building blocks. Due to its flexibility the architecture is very modular which allows for instance to increase the number of message transport engines to enable simultaneous message streams with higher bandwidth. An example network device that is based on the EXTOLL architecture is shown in Figure 3-27.

The EXTOLL architecture is comprised of four domains. The host interface domain, the NoC domain, the network engine domain and the network switch domain. The host interface domain provides access to the host processor and main memory. The EXTOLL architecture supports multiple host interface implementations that support different protocols. In particular, the architecture provides support for PCI-Express,

HyperTransport 1 and HyperTransport 3. In principle any other host protocol can be deployed as long as the internal interface is compliant with the HTAX specification. Highest performance regarding latency and bandwidth is provided by the HyperTransport 3 core which we propose in [78]. Furthermore, the HyperTransport 3 core supports cache coherency which enables the attached device, e.g. EXTOLL, to participate in the cache coherent domain of the host processors.

The central component of EXTOLL which resides between the host interface and the network engines is the HTAX NoC. In this case, the HTAX is configured as a nine port switch with a 64 bit wide data path. For EXTOLL, the HTAX supports six virtual channels to decouple *write*, *read* and *response* data operations as well as different management layer streams from each other.

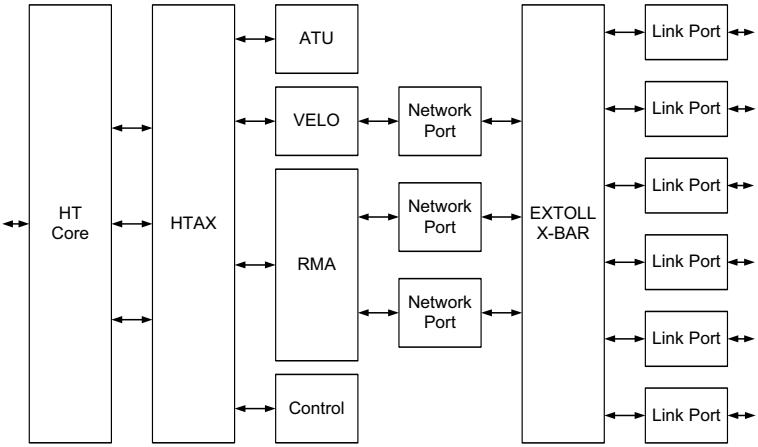

Figure 3-27:EXTOLL Toplevel Architecture

The network engine layer consists of two main components which are the engines of Very Low Overhead (VELO) and for Remote Memory Access (RMA).VELO is the transaction engine for small messages that provides the lowest possible communication latency. In [96] we show that VELO achieves a half round trip latency of below 1 us for small messages. This low latency capability provides significantly better performance for

fine grain communication schemes than other technologies, including Ethernet and InfiniBand. The key idea of VELO is to reduce the amount of operations required to setup a message transfer to an absolute minimum. This is achieved by tightly integrating hardware and software components which allows to trigger VELO messages atomically with a single store instruction to the network device. The hardware assisted device virtualization technique enables secure access to multiple software threads by encoding additional state information in the virtual address that is used for the store instruction. By restricting the address range that is made available to the user process, security policies can be enforced. Furthermore, VELO offers an efficient scheme for receiving messages and storing them to local memory. Message arrival is detected by polling a status queue while the payload is transferred into DRAM using VELO's integrated DMA engine.

Sending VELO messages involves copying the payload from main memory or its caches to the EXTOLL network device using PIO on the sender side. For large message payloads, this technique is inefficient as it generates a significant CPU overhead. The RMA engine provides a solution for this problem. The RMA engine provides direct access to main memory in remote machines. Therefore, *put* and *get* functions are supported that are initialized by a descriptor. This enable true one-sided communication. One-sided communication defines that either the sending or the receiving process take care of the complete transaction without required interaction of the other. The complete message transfer is handled by the RMA units by using direct memory access (DMA), thereby, unburdening the CPU. Nuessle *et al.* show in [109] that the EXTOLL RMA engine provides an excellent latency and bandwidth for medium and large sized messages.

The EXTOLL architecture avoids any kind of trapping into the operating system to prevent unpredictable high latencies for message transfers. Therefore, all network engines support user level access and hardware assisted virtualization. The architecture supports security mechanisms to allow direct access of the hardware by unprivileged user processes. Although powerful, this mechanic is problematic regarding memory management, as user processes reside in virtual memory. However, the DMA engines of EXTOLL can only access physical memory. To allow EXTOLL to move data between physical memory locations based on descriptors that hold virtual addresses, an address

translation unit (ATU) is required. The ATU module contains a translation lookaside buffer (TLB) for virtual to physical address translations which is managed by driver software. To avoid deadlock scenarios in which an address translation request blocks or is blocked by a data transaction to or from the RMA engine, the applied HTAX NoC supports extra virtual channels for stream decoupling.

Finally, EXTOLL implements the network layer which performs packetization and data injection into the actual physical wire. Fault tolerance mechanism including ECC, CRC and a retransmission scheme are supported as well. As EXTOLL supports direct network topologies like the Mesh and the Torus, the network switch is included on the host channel adapter. This approach is very cost and power efficient as no external switches are required. Due to its multiple links, it can provide a higher bandwidth and a lower latency for nearest neighbor communication than traditional, central switch based networks.

3.5.2 EXTOLL NoC Integration

The NoC implemented in EXTOLL is a single stage HTAX switch providing eight bidirectional ports. To increase the host interface bandwidth, the HTAX implements three ports that directly connect to the HyperTransport core. This approach enables up to three packet streams in parallel for different virtual channels. The remaining ports are connected to the VELO and RMA engines, as well as to the register file and to the Excellerate interface. The RMA engine implements multiple ports to increase bandwidth and to support two simultaneous message streams between the engine and main memory. This architecture allows to serve PUT, GET and address translation requests in parallel.

Transactions within the HTAX layer are processed using the low overhead HyperTransport On-Chip (HTOC) protocol. The HTOC protocol extends HyperTransport for on-chip communication capability by increasing the word size from 32 to 64 and to 128 bit respectively. This enhancement is required to support higher on-chip bandwidth communication at moderate operating frequencies. Furthermore, HTOC increases the number of addressable endpoints to 256, adds additional virtual channels and new features which facilitate decoding of packets. HTOC packets contain a command frame, shown in Figure 3-28 which may be followed by up to 64 bytes of data.

byte	7	6	5	4	3	2	1	0
0	SeqID[3:2]		Cmd[5:0]					
1	PassWD	SeqID[1:0]		UnitID[4:0]				
2	Mask/Count[1:0]		Compat	SrcTag[4:0]				
3	Addr[7:2]						Mask/Count[3:2]	
4	Addr[15:8]							
5	Addr[23:16]							
6	Addr[31:24]							
7	Addr[39:32]							

8	extAddr[47:40]							
9	extAddr[55:48]							
10	extAddr[63:56]							
11	ext_srcTag[7:5]			rsvd	stomp	command type[1:0]		D_att
12	ext_count[9:4]						ext_srcTag[9:8]	
13	htaxSourceInport[7:0]							
14	htaxReqOutport[7:0]							
15	extVC[7:0]							

Figure 3-28:HTOC Command Frame Specification.

The HTOC command frame is 128 bits wide and supports the following fields. Bytes zero to seven are directly adopted from the HyperTransport specification [73] which alleviates bridging between HT and HTOC packets. Bytes eight to fifteen contain the protocol extensions introduced by HTOC. The extended fields including *extended address*, *extended srcTag* and *extended count* provide additional bits to increase the address size as well as the number of concurrent packets and the size of the appended data frame. Additional bits specify the type of the command (*command_type*), e.g. write request or read response and define whether a data frame is attached to the command (*d_att*). The *htaxSourceInport* and *htaxSourceOutport* fields specify the origin and target of a request which is used for on-chip routing and response to request mapping. This field is particularly important as it introduces the possibility of routing HTOC packets between different components on a chip. HyperTransport on the other hand is an off-chip protocol which only supports communication between discrete chips. Finally the *extVC* field specifies additional virtual channels for on-chip communication. A more detailed description of the HTOC protocol can be found in the HTOC specification [74].

3.5.3 EXTOLL Implementation

The HTAC NoC employed in EXTOLL has been implemented using two different technologies, in particular the 90nm XILINX Virtex4 FPGA and the 65nm TSMC ASIC technology. For the FPGA, the complete design has been synthesized, placed, routed and tested on the HTX rapid prototyping board we propose in [52]. The HTAX NoC used in this design instantiates eight ports, five virtual channels and implements a 64 bit data path. The complete EXTOLL design contains the components as described above and runs at 156 MHz. The HTAX NoC provides a per port bandwidth of 1.248 GByte/s and a bisection bandwidth of 9.984 GByte/s. The direct connected topology connects every on-chip component within one hop which results in a packet latency of two cycles in low latency mode, respectively 12.8 ns. Table 3-1 lists the occupied resources of the FPGA implementation. The first row lists the different components which are provided by the Xilinx Virtex4 FX100T device including the number of four-input look up tables, flip flops, multiplexers and RAM blocks. The second row show the number of resources which are available on the FPGA. The third row shows the amount of resources that are consumed by the complete EXTOLL design while the fourth row presents the consumed resources of the HTAX.

Table 3-1: EXTOLL HTAX Resource Consumption

Resource	Available	EXTOLL Total	HTAX
Look Up Tables	84,382	66,002	2,380
Flip Flops	84,382	32,387	702
Multiplexer	NA	5,412	208
Block Rams	376	144	0

3.6 CONCLUSION

In this chapter we have presented a novel technique to facilitate the design and implementation of generic on-chip networks. As microchips are integrating more and more functionality and are becoming increasingly complex, a feasible solution is required to interconnect the components within such large designs. To address this issue, the HTAX framework provides chip architects with a convenient solution allowing to generate on-chip networks from an abstract description. The introduced methodology assists in building modular and flexible designs that offer highest performance. We presented a NoC generator tool that produces synthesizable RTL code and a full system verification environment reducing the design and verification time significantly. The chapter provides the following contributions. A detailed network on-chip design space exploration which summarizes the state of the art and presents an elaboration of the challenges that need to be addressed. A layered component based design methodology which addresses these challenges by introducing a protocol agnostic network on-chip architecture. The modular approach enables the HTAX framework to be applied to a large number of applications and communication protocols. In addition, we provide a detailed insight into the NoC architecture that covers all levels of abstraction, from the application layer to the microarchitecture. The HTAX design methodology is introduced and the NoC generator tool is presented. The chapter is concluded with a comprehensive evaluation of the approach and an applied use case.

The HTAX framework has been applied to multiple chip designs. In addition to the previously mentioned EXTOLL design, the framework was utilized in a highly parallel NAND flash controller. The design implements 24 high speed flash controllers which are interconnected via an HTAX on-chip network. Further components include a PCIe host interface and two serial links that connect to additional flash devices. The whole design has been implemented on a Lattice FPGA that resides next to 192 GByte of flash memory on a dense PCB. Other designs that deploy the HTAX framework include a data acquisition design that is used within the Compressed Baryonic Matter experiment at the physics institute Helmholzzentrum für Schwerionen GmbH (GSI). Finally, the HTAX NoC is applied in an FPGA design for high frequency trading acceleration.

The broad application range of the HTAX framework demonstrates its flexibility. However, during the course of the HTAX development, the specification underwent different modifications to accommodate the requirements of two specific applications. Both modifications lead to a generalization of the design, extending its applicability. While we believe that the current specification is flexible and powerful, an in-depth design space exploration in the initial project stage had allowed to design an improved specification from the beginning. Both specification revisions required a renewed verification of the HTAX design as well as modifications to the existing functional units that utilized the NoC. An adaption of the specification is, therefore, undesirable and should be avoided if possible.

Another issue that appeared during the development stages was induced by the design decision of optimizing for minimum latency. This appeared problematic for achieving timing closure in high frequency ASIC designs especially as the monolithic architecture of the arbiter aggravated the integration of further pipeline stages. This has lead to the modular architecture that is deployed in the current design. By instantiating the arbiter as a separate module it can be easily improved or replaced by other implementations.

While the HTAX framework has proved successful, there still exist opportunities for further improvements. An important task is to explore further topologies, for example the ring. While rings were dismissed due to their limited scalability and high latency, they recently experienced a renaissance within Intel's network-on-chip designs. Rings provide the advantage of a very simple architecture that is capable to operate at very high frequencies. This fact may compensate the disadvantages, wherefore, the ring topology should be explored in the future. Another task worth exploration is the integration of a NoC simulation engine to the framework. This feature allows to simulate different NoC architectures without traversing the complete design flow, further decreasing the development efforts.

Scaling the number of on-chip components leads to dense systems that deliver lots of processing power on a single chip. As a result, these architectures show a significantly increased demand for memory bandwidth and capacity. To address this issue, the next chapter will propose a scalable memory extension technique.

CHAPTER 4

SCALABLE MEMORY EXTENSION FOR SYMMETRIC MULTIPROCESSORS

4.1 INTRODUCTION

Over the last years, the growing disparity between processor and memory speed has led to the so-called *memory wall*. This issue has been addressed and partially solved by applying a memory hierarchy using large and highly efficient caches. However, a new memory wall appears on the horizon [26]. Latency is not the main issue this time, but rather the limited bandwidth and the size of main memory. While the number of transistors on a processor and likely the number of cores will double every 18 months [9] following Moore's Law [103], memory density only grows by a factor of two every three years [71]. Even more dramatic is the growing disparity between the advancements in processor- and package technology which defines the maximum pin count that can be supported per chip [75]. Current packages employ up to 2000 pins and are not expected to scale significantly in the future, thereby, limiting I/O bandwidth. This will lead to an abundance of cores whose compute power is limited by memory bandwidth and capacity.

The situation is further aggravated by the fact that the memory footprint of applications is increasing. Virtualized servers that run multiple operating system instances show a constantly increasing demand for more memory. In addition, there still exist a lot of single threaded "memory-bound" applications like electronic design automation tools, finite element analysis simulations and database accelerators like Memcached that can benefit more from increased memory than from additional compute cores [54]. The current processor architecture from Intel and AMD [34] directly implements the memory controller on the chip supporting only a limited number of DRAM modules. Hence, one possibility to increase the memory capacity is to form cache coherent non uniform memory systems (ccNUMA), combined of multiple processors. This technique enables aggregated shared memory systems with increased capacity. However, it is very power and cost inefficient solution to deploy a large number of processors which are not used for computation but only for increasing memory capacity.

To address this issue, we have developed a new technique that supports high memory footprint applications in a much more efficient way. Our approach allows building Symmetric Multiprocessor (SMP) systems that combine a limited amount of processors

with terabytes of memory. Our proposed memory extension device supports cache coherency and can be transparently integrated into a system. Similarly, as in tightly coupled ccNUMA systems, accessing the memory extension is only slightly slower than accessing local memory.

Our custom memory extension controller has been further optimized to increase the performance of the memory subsystem. We introduce an offloading technique that allows transferring certain tasks to the memory controller reducing CPU overhead. For this purpose, we propose a cache coherent DMA controller to offload memory-to-memory transactions to the memory extension device. Memcpy is a frequent task that applies high CPU burden as each data block has to be moved between memory locations using programmed I/O (PIO). By introducing a DMA offloading engine that can process such transactions independently, system performance is significantly increased.

4.1.1 Related Work

Many approaches aggregate multiple machines to form a large distributed shared memory system leveraging the fact, that in most cases a network access is faster than the process of swapping data between memory and hard disk. Virtual shared memory techniques as proposed by Li [92] and Raina [113] implement full software coherency on top of such a distributed shared memory system. Recently, the startup ScaleMP has picked up the idea to develop a product for server aggregation called the versatile SMP architecture [115]. Based on InfiniBand interconnected Intel Xeon nodes, ScaleMP forms virtual SMPs with up to 128 cores and 4 TB of shared memory. All these approaches utilize a software based cache coherency protocol to support the shared memory programming paradigm. Programmers often prefer shared memory architectures due to their convenience, however, the performance of cache coherency protocols is reduced significantly when communicating over a high latency interconnect. Our approach solves this issue by employing a significantly faster interconnect which reduces the latency by orders of magnitude and, thereby, enables large and efficient shared memory systems.

Hardware assisted approaches like the scalable coherent interface (SCI) as proposed by Gustavson [62] shift some of the cache coherency functionality to hardware to speed up

the operation. The SGI Origin [89] is a powerful implementation that provided cache coherent systems with a very high number of cores. A more recent approach based on SCI is provided by Numascale [108]. Another cache coherent approach is the Horus system [87] which leverages HyperTransport (HT) to build SMPs of up to 32 nodes. Although, such SMPs provide a large distributed shared memory address space they come with the cost of additional CPUs and, thereby, do not improve the compute to memory ratio. The approach, furthermore, introduces significant costs in the form of the Horus chips itself.

Some approaches avoid the cache coherency penalty by increasing the memory capacity using block IO devices. In this case, the DRAM extension is rather treated as storage than as system memory. Texas Memory Systems sells DRAM based solid state disks [124] with a maximum capacity of 512 GB which are attached via FibreChannel. Panda et al. propose swapping to remote memory over InfiniBand [93] to reduce the negative effect of swapping. While both approaches can speedup high memory footprint applications that need to swap to the hard disk they still suffer the latency of an interconnect and do not provide a real system memory extension.

Lim *et al.* propose the design of a disaggregated memory blade [94] that is attached via PCI-Express. The memory extension is transparent to applications by supporting cache coherency, however, only on a page sized, course grained granularity.

More radical approaches try to increase memory capacity and bandwidth by introducing 3D stacking of memory chips and the processor [80][19]. By stacking multiple memory chips on top of the processor using through-chip vias large memory capacities can be accessed with high bandwidth and low latency. The proximity connection approach [48] employs short range capacitive coupling to vertically interconnect multiple chips in a single package. By interleaving processing and memory elements, large devices with a high memory capacity can be realized within a single multichip package. While these approaches look promising and capable to significantly improve the processor-memory interconnect, they require a major modification of the current processor technology and, therefore, only provide a mid term solution.

4.2 Design Space Exploration

4.2.1 Existing Memory Architectures

Different architectures have been proposed to increase the computational capability and addressable amount of main memory within a computer. While the traditional Von Neumann computer architecture defines a single processor with attached memory, recent techniques utilize multiple processors and memory controllers. Modern computer systems can be separated into different classes depending on their memory architecture. The resulting properties thereof have a significant impact on the scalability, bandwidth and access latency of the memory subsystem and, therefore, overall performance of the system.

Uniform Memory Architecture

The Uniform Memory Architecture (UMA) describes shared memory multiprocessor systems in which the processor-memory interface is identical for all processors. As a consequence, the bandwidth and access latency to main memory is equal for all applications independent of the physical processor on which they are executed. Figure 4-1 shows a common UMA system that deploys four processors, an interconnection network and memories. An example for this architecture is the Intel Xeon which uses a so-called *front side bus* (FSB) to interface the processors to a single memory controller. The front side bus not only transports memory addresses and data, but also snoop messages issued by the cache coherency mechanism to guarantee the consistency of the processors' caches. In the Intel Xeon architecture the original FSB is only capable of supporting one memory transaction at a time. As this limits the scalability in terms of memory bandwidth and number of supported processors, Intel introduced a double pumped FSB that supports two simultaneous transactions. In principle, UMA systems can support an arbitrary number of processors and memories, however, besides the scalability limitations introduced by the cache coherency protocol, the shared front size bus and the single memory controller represents a severe bottleneck. On the upside, the UMA architecture does not require an intelligent placement of processes and data in memory as every processor accesses every memory location in an identical way.

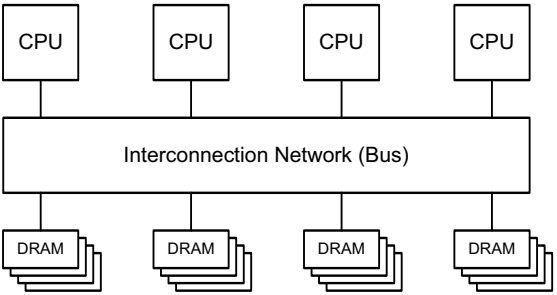

Figure 4-1:Uniform Memory Architecture

Non-Uniform Memory Architecture

Non-Uniform Memory Architectures (NUMA) pursue the goal of increasing system scalability by removing the bottleneck of the shared front side bus and the memory controller. Therefore, the bus is replaced by an interconnection network and every processor deploys its own memory controller in most cases on the same chip. The advantage of this technique is that the bandwidth scales with the number of processors and, furthermore, the latency is reduced by the memory controller integration. Most architectures, like the AMD Opteron deploy a point-to-point interconnect like HyperTransport to replace the front side bus. This interface is used to access remote memories and to propagate information required by the cache coherency protocol. In the AMD architecture, a packet based protocol is utilized that supports multiple simultaneous transactions, further increasing the efficiency and performance. Figure 4-2 shows a four processor NUMA system. As can be seen, the distance between local and remote memories to the processors is unequal. This results in different memory access latencies, depending on the location of the process and its data. It is therefore, important that frequently accessed data is located close to the process, preferably on the local main memory. Operating systems therefore, support explicit placement of applications and data to specific processors and physical memory devices. This fact complicates development of applications and the resource management as the principle of temporal and spatial locality

is expanded from the processor caches to the memory subsystem. However, the extended scalability of this architecture compensates this disadvantage. While NUMA represent an improvement over UMA architectures, their scalability is still limited due to the cache coherency protocol. To overcome this limitation and to increase the scalability, distributed shared memory systems have been proposed.

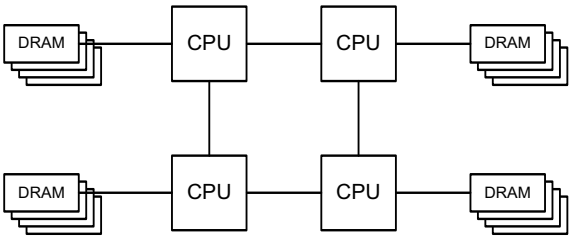

Figure 4-2:Non-Uniform Memory Architecture

Distributed Shared Memory

A further memory architecture is represented by the Distributed Shared Memory (DSM) paradigm in which the local memory of multiple nodes is aggregated into a single globally shared address space as shown in Figure 4-3. Applications can thus access remote memory using regular read and write operations in addition to their local memory. Most existing implementations utilize loosely coupled network protocols as Ethernet and InfiniBand to interconnect the systems in a non-coherent fashion. This implicates that, although, applications access global memory in the same way as local memory, they cannot rely on cache coherency as cache lines may be shared between remote nodes without their mutual knowledge. There exist two common solutions for this problem, by either applying programming models that do not require consistency or the deployment of a software based cache coherency mechanism. In the partitioned global address space programming model as used by the UPC [30] and Fortress [5] programming languages, the programmer needs to be aware of the memory architecture and has to use specific mechanisms like strict operations to synchronize data between nodes. The consistency is, therefore, explicitly enforced by the application or software library. Due to their low

overhead and scalability, PGAS style languages show a great promise for future parallel systems although they are still in their early stages of development and have not been widely accepted yet. Software cache coherency mechanisms as provided by ScaleMP [115] and Symmetric Computing [32] apply modifications to the memory management within the Operating System to realize a globally coherent memory address space. To improve the performance of the system, the techniques introduce a directory based cache coherency protocol and they utilize a portion of main memory as a network cache, managed on a per page basis. Shared memory applications can be executed on such a system without modifications, however, the cache coherency overhead limits the scalability of such systems. Applications that continuously update globally shared variables experience a severe performance penalty due to the high latency of the network interconnect. These shortcomings are addressed by hardware supported scalable shared memory systems as provided by Numascale [108] and 3Leaf [1]. Both systems offload the coherency protocol to hardware allowing to bypass the Operating system by accessing remote memory directly from user space. The techniques pledge higher performance by decreasing latency while simultaneously increasing the scalability of such systems. Nevertheless, both software and hardware based globally shared memory systems will always suffer from the cache coherency overhead and hence do not represent a feasible alternative for future massively scalable systems.

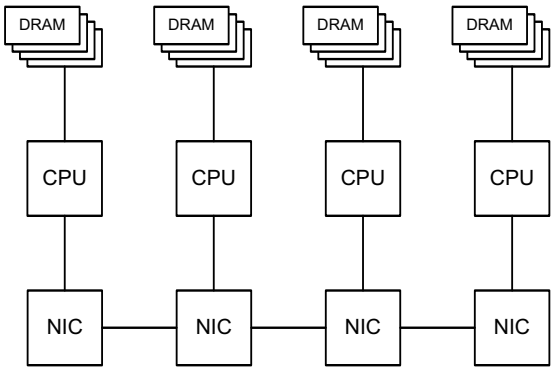

Figure 4-3:Distributed Shared Memory Architecture

4.2.2 Memory Subsystem Analysis

System performance is significantly affected by the speed of the memory subsystem. In the following, we analyze three important aspects of computer systems related to main memory. Those are swapping, the overhead of the cache coherency protocol and the significance of the *memcpy* instruction.

Demand Paging and Swapping

Modern operating systems support a virtual memory architecture to provide each process with an individual memory space using segmentation. This technique increases system security and offers an almost infinite amount of virtual memory to each process. To limit the amount of required main memory, only the addresses that are currently in use are mapped into physical memory. As data in the remaining address space resides on low performance, solid state devices like hard drives, the memory footprint of applications should not exceed the capacity of the main memory. In the case of a page fault (the page doe not reside in main memory), the OS is required to load the page from hard disk by replacing another page. This process, called *swapping* is time-consuming and, therefore, degrades system performance substantially. Figure 4-4 shows the results of a benchmark we performed on a standard x86 system with 4 GB of main memory. The benchmark accesses a working set of increasing size. As long as the working set fits into main memory, the memory bandwidth is almost constant at below 1800 MByte/s. However, once the memory footprint of the program exceeds the size of main memory (minus a system reserved part), the resulting memory bandwidth decreases considerably. Although, this is a known fact, the benchmark shows two important issues. First, it can be observed that instead of showing a graceful degradation, the performance drops abruptly after crossing the critical size. Second, the decrease in performance is absolutely significant, as the remaining bandwidth is reduced by three orders of magnitude. It is obvious that swapping needs to be avoided whenever possible, however, as systems run multiple virtualized operating systems at once and the memory footprint of applications increases, physical memory becomes a scarce resource. Our proposed memory extension device minimizes the need for swapping and thereby increases performance.

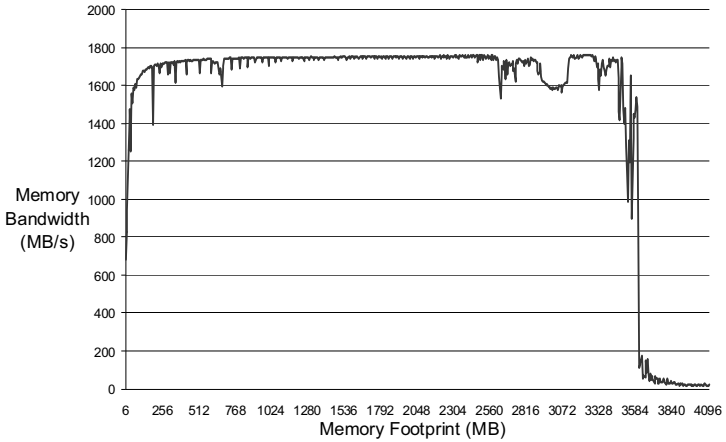

Figure 4-4:Impact of Page Faults to Bandwidth Performance

Cache Coherency Overhead

The symmetric multiprocessor technique allows multiple processors to share a common memory address space. As processors implement caches that hold copies of data from main memory, a cache coherency mechanism is required to guarantee data consistency between the processors. AMD based processors implement a broadcast protocol which involves probing of all caches in prior to a data modification in main memory. Probing is implemented as a two way handshake which adds latency for the completion of each memory access. The added delay increases with the amount of processors in the system due to multi hop topologies. Figure 4-5 shows the negative effect of multi processor configurations we have observed by executing the Stream benchmark. Four different system configurations were evaluated.

(1) A system based on a single AMD Opteron CPU with 2.2GHz, 4 MB L3 cache and 4 GB of RAM. (2) A system with two CPUs of the same type. (3) A dual CPU system in which the Stream benchmark was executed on CPU0 and the memory was allocated physically on CPU1. (4) A system consisting of a single CPU and a cache coherent FPGA based coprocessor running at 200 MHz.

.

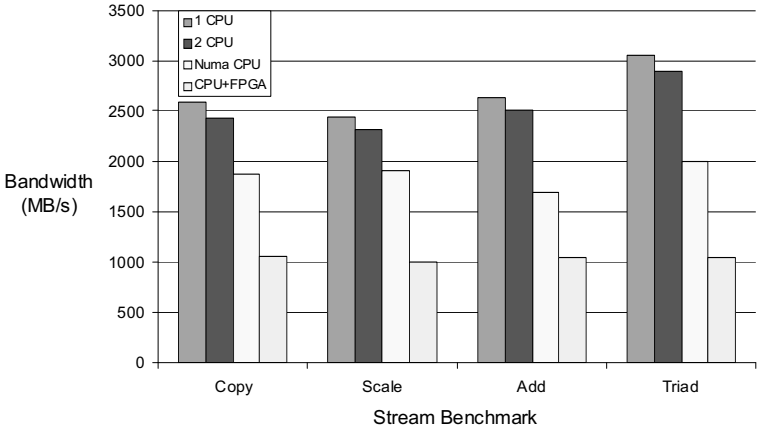

Figure 4-5:Cache Coherency Protocol Penalty to STREAM

It can clearly be seen that even a single additional Opteron CPU that does not participate in the benchmark effectively reduces Stream performance which is due to the cache coherency overhead. By accessing memory on the remote CPU, the NUMA effect can be observed which further penalizes performance. In the last test configuration a slow cache coherent FPGA based HyperTransport device was employed in the system which is completely passive and only responds to cache coherency communication. The performance drop that can be observed is dramatic. Tests with similar results have been conducted by IBM [129].

To avoid the cache coherency penalty entirely, we propose the asymmetric probing technique [24]. By deploying this technique, devices that implement a memory controller but no caches can issue probe request but do not receive probe requests from other memory controllers. As a result no probes are ever received by our memory extension device which addresses the problem exposed by the Stream benchmark. Using this technique, even multiple extension devices can be supported without any negative effect on overall system performance.

Impact of Memory-to-Memory Transactions

The *memcpy* library call is frequently used by applications to copy data from one memory location to another. Benchmarks have been conducted that copy data of different granularity using the C-library memcpy call. The results are shown in Figure 4-6. The diagram shows the performance of the memcpy call for different CPU locations within the system. The used CPUs comprise three levels of caches, proving capacities of 64 KB, 512 KB and 2 MB. The impact of the cache sizes can be clearly seen in the graph, resulting in drops of performance when the next level has to be used. Although, the performance is adequate it is important to note that the *memcpy* task utilizes the processor by 100%. Offloading this task to our memory extension device, therefore, frees up processor resources for computation.

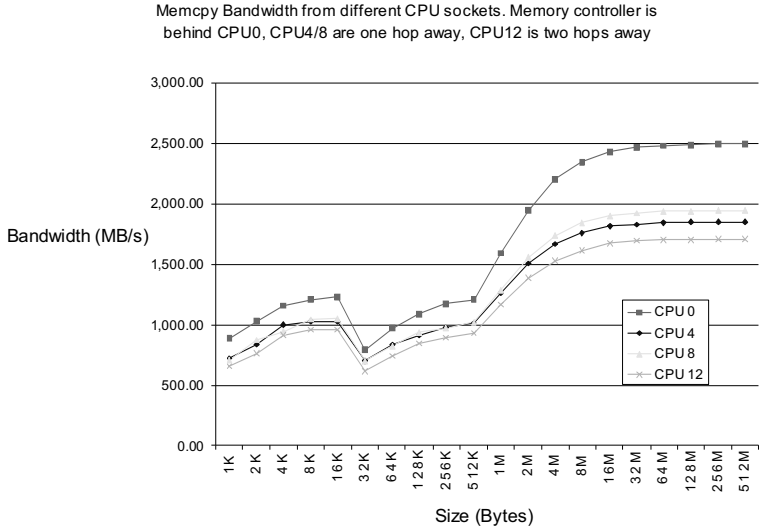

Figure 4-6:Memcpy Performance in Different System Configurations

4.3 IMPLEMENTATION

Our memory extension architecture was guided by the following observations:

- There exists the need for low power, cost efficient system memory extensions.
- The processor pin limitation prohibits further scaling of parallel memory controllers within CPUs.
- Shared memory extensions need to be closely coupled to the system to avoid performance degradation.
- Cache coherency needs to be supported to comply with the standard shared memory programming model.

Given these observations, we propose a novel approach to extend the memory capacity of commodity-off-the-shelf systems. Our technique leverages the serial, low pin count HyperTransport processor interface to access additional memory controllers, thereby, enabling tightly coupled cache coherent systems with terabytes of main memory. The technique enables to increase the memory capacity of a system as well as the memory bandwidth considerably using a very low pin count. The analysis will show that our architecture provides a significantly increased capacity per pin ratio compared to current systems.

Figure 4-7 illustrates our proposed memory extension device within an AMD Opteron based system. The central element remains the host processor which implements an on-chip memory controller (MCT) to access a small amount of local DRAM. In addition, each processor implements up to four HyperTransport links which either provide cache coherent communication with other processors or non-coherent communication with I/O devices. Each system contains at least a single non-coherent device, the southbridge, which bridges to I/O like PCI and SATA. In the described system, multiple memory extension devices which implement additional DRAM controllers are connected to the CPU via cache coherent HyperTransport links. Our flexible architecture can support multiple HyperTransport links which allows to daisy-chain several memory extension devices to further increase memory capacity.

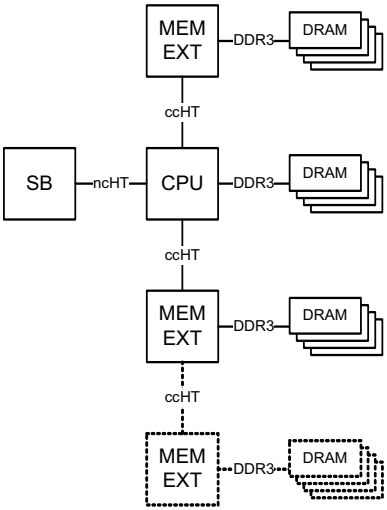

Figure 4-7:System Architecture with the Proposed Memory Extension

As in ccNUMA systems, the memory access latency depends on the number of hops between the origin and target device. For ASIC implementations each HyperTransport hop adds around 50ns of latency which needs to be considered by the memory management policy. Nevertheless, other approaches that extend the main memory through loosely coupled interfaces as PCI-Express or Ethernet and InfiniBand suffer latencies in the range of microseconds.

An important aspect of our design was its ability to scale in terms of memory capacity and bandwidth. Therefore, our system architecture supports multiple memory extension devices while the memory extension devices in turn support multiple DRAM controllers. The central component of our internal architecture is the network-on-chip (NoC) that supports a variable number of HyperTransport links and memory controllers. Further components include the cache coherence protocol handler and the DMA engine. An overview of the internal memory extension architecture, which will be presented in detail in the following paragraphs, is shown in Figure 4-8.

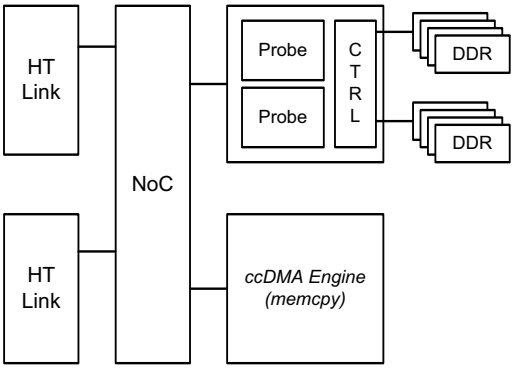

Figure 4-8:Internal Memory Extension Device Architecture

4.3.1 HyperTransport Domain

The memory extension device is directly connected to the host processor via the cache coherent HyperTransport protocol. HyperTransport represents an excellent solution as it provides lowest latency, a very high bandwidth and cache coherency. To leverage the HyperTransport technology our memory extension technique utilizes the HyperTransport 3 core which we proposed in [78]. The core is fully compliant to the HT 3.1 link specification and supports 16 bit links with up to 5.2 Gbit/s per lane. This enables a maximum theoretical bidirectional bandwidth of 20.4 GByte/s which exceeds the performance of a DDR3 memory interface. The HyperTransport core supports different traffic classes which are decoupled from each other by the use of virtual channels. By separating read, write, response and cache coherent transactions from each other, head of line blocking can be minimized and deadlocks can be avoided. This approach, furthermore, enables split-phase read transactions enabling multiple outstanding transactions. This is in particular beneficial for a high performance memory controller. Besides providing virtual channel multiplexing and demultiplexing, the HyperTransport core takes care of data packetization, serialization, flow control, buffer management and the complete initialization sequence. To increase the reliability of the system the core

supports cyclic redundancy check (CRC) protection and a retry mechanism which allows recovering from link errors. Each individual packet is appended with a 32 bit CRC by the transmitter. In addition, the packet is stored into a retransmission buffer. Packets reside in the retransmission buffer as long as they are positively acknowledged. In the case of a link error which can be detected by the receiver by recalculating the CRC, a negative acknowledge is sent to the transmitter which then forces a re-initialization of the link and the packet retransmission. As the core uses a 128 bit wide internal datapath, the CRC calculation is a particularly challenging task. To compute the CRC, the complete 128 bit datapath needs to be combined using XOR operations. Furthermore, as the computation is cyclic the computed CRC from the previous clock cycle needs to be XOR'ed with the data stream. This permits pipelining of the algorithm as the result of one stage is required in the next clock cycle. To address this problem we have recomputed the mathematical expressions for calculating the CRC by parallelizing them into multiple equations [78]. Furthermore, we extracted the part of the computation that does not exhibit cyclic dependencies, which we could then process using multiple pipeline stages. The combination of both techniques enabled timing closure of the CRC modules even at high operating frequency. Due to the wide datapath of 128 bit, protocol decoding within the core is a complex task. As we show in [78] the decoding complexity increases exponentially with the internal data width of the core. However, to support the high data rates of HyperTransport, wide data busses are a necessity. To support the highest data rate of HyperTransport (HT3200) with a 64 bit datapath an internal operating frequency of 1600 MHz would be required. As such high frequencies are unfeasible in standard cell ASIC technologies, it was decided to implement a 128 bit datapath. The main challenge of this architecture is that the core needs to be capable to decode multiple packets during a single clock cycle. As the smallest packet in HyperTransport is 64 bit, it is possible to receive segments of three different packets during a single clock cycle which need to be decoded, CRC-checked and processed in parallel. This increases the parallelism of the design as well as the depth of the pipeline. As a result the core implements a 3-way receive pipeline and only shares the most resource intensive components like the buffers which are used to decouple the threads. The transmit pipeline is also capable of processing multiple packet streams in parallel before multiplexing them onto the link.

The queue interface of the core is connected to the on-chip network to enable packet distribution to the memory subsystem. The NoC has been created with the HTAX framework described in Chapter 3. This approach enables flexible and congestion free communication within all components of the design. In addition, it provides the capability of scaling the HyperTransport link interfaces within the chip. By utilizing two interfaces as shown in Figure 4-8, daisy chained configurations that include multiple memory extensions can be realized. By further increasing the number of links other direct topologies including trees and Meshes can be supported. The architecture is truly scalable as in a daisy chain or tree topology, there exists no limit for the memory capacity of the system. The only limitation is caused by the restricted size of the physical address space.

4.3.2 Memory Controller

The memory controller is connected to the on-chip network and processes all incoming load and store transactions. It furthermore implements the cache coherency protocol and, hence, is able to issue probe packets targeting the host processors within the system. After processing the cache coherency protocol, the read and write transactions are forwarded to the DRAM controller. The DRAM controller which in our case controls DDR2 SODIMM memory banks then actually writes or reads the data to or from the DIMM. In the following, we will discuss the functionality of the memory controller which implements the sub-blocks shown in Figure 4-9.

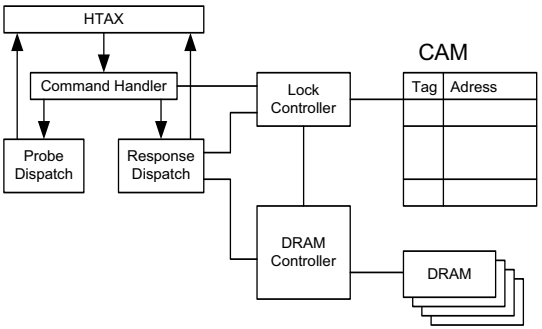

Figure 4-9:Memory Controller Sub blocks

The *Command Handler* block receives the HyperTransport packets and decodes the command header of each packet. Depending on the type and destination, the packet is then forwarded to the according unit within the memory controller to be processed. The command handler block, furthermore, communicates with the lock controller to guarantee consistency of all operations. In addition, it also enforces a throttling mechanism necessary to compensate the latency of the DDR2 SODIMM. In general, the memory controller processes the commands much faster as the DRAM is capable to deliver the data. A possible future implementation, therefore, may include a larger cache which can buffer read and write requests.

The *Lock Controller* block keeps track of all the transactions in flight to the memory controller block at any point in time using content addressable memories (CAM). Since HyperTransport implements a split transaction protocol each transaction is identified by a *SrcTag* and the originator of the transaction. The *Lock Controller* block uses CAM modules to keep track of outstanding transactions based on various stages of the transaction. The first CAM is used to keep track of all the transactions in flight based on the *SrcTag* and the source of the transaction. The second CAM is used to keep track of the state of the cache lines based on the address of the transaction. Upon receipt of a new command by the command handler module, a new entry is added to both CAM modules. The entries in the CAMs are updated based on the progress of the transaction and finally overwritten when the transaction has been completed.

The *Probe Dispatcher* block is responsible for issuing cache snoop requests to all processors in the system. Snoop requests are required to implement the cache coherency protocol defined by the Opteron processors. For example, if one processor wants to load a cache line, all other processors need to be consulted whether they are sharing that line. This is considered a vital process in a cache coherency system because the data that is stored in the memory has to be up to date. Upon receipt of a snoop request the processors accumulate the responses from their respective CPU caches (there exist L1 and L2 caches for each core within the processor) and send one final response back to the memory controller module which either contains a current snapshot of that specific cache line or a response indicating that the cache line does not exist in their caches.

The *Response Controller* block handles the HyperTransport response packets which are received by the MCT in response to snoop requests. In the case of a cache line update the data is extracted from the packet and forwarded to memory. In all cases, upon reception of the response packets, the *Lock Controller* block updates the CAM modules with the correct information.

The final stage in the memory controller module consists of the *Memory Module block* which interfaces the data from all the cache coherent transactions to the DDR2 SODIMM. This module is also responsible for synchronization of all the data from the application interface clock domain to the DDR2 SODIMM clock domain.

4.3.3 Prototype Implementation

Our proposed memory extension architecture was implemented in the form of an FPGA based prototype. We have developed register transfer level (RTL) code for each module and synthesized it using the Xilinx FPGA design flow. As a hardware platform, we designed a PCB that features state of the art FPGA technology to accommodate our memory extension device. Although, such a development is very time consuming, it enables realistic measurements in an actual system which provides a much better performance evaluation as simulation-only approaches. To deploy our design, it is plugged into a HyperTransport Extension (HTX) slot of a commodity-off-the-shelf server. This provides the possibility to directly connect our device to the Opteron processor and to participate in the cache coherent fabric. As memory extension devices are neither supported by standard firmware and operating systems, we have developed our own BIOS and kernel level software to support the Linux operating system.

4.3.4 RTL Design

Each module, including HyperTransport core, NoC, MCT, DMA controller and caching engine has been implemented in the hardware description language Verilog. Every module is written in synthesizable RTL code to maintain technology independence. This not only allows us to test the design on an FPGA but also alleviates porting the

design to an ASIC technology. While this approach is feasible for most components, there exist some technology dependent modules which need to be developed for the individual platforms. In fact, one of the most challenging tasks has been the implementation of the FPGA specific physical layer interface (PHY) block within the HyperTransport link transceivers. We have proposed a solution for multi-mode PHYs in [100] which was successfully applied in this design.

Due to the complexity of our implementation, a comprehensive verification environment has been developed allowing to test the design in a simulator before loading it onto the FPGA. Simulations greatly facilitate debugging and the bringup of the system, although they only cover a certain set of errors due to its abstract nature. We developed a Cadence Open Verification Methodology based verification suite that is capable of issuing arbitrary HyperTransport packets while checking the behavior of the core. Any action that is non-compliant to the specification is detected and reported. As our testsuite generates random constraint test patterns, a verification coverage analysis has been conducted that guarantees that all important aspects and use-cases are effectively covered. Finally, the design has been transformed into a gate level netlist using the Xilinx ISE FPGA design flow. Our design requires more than 200.000 lookup tables (LUTs) and is capable to operate at a clock frequency of up to 225 MHz whereas the external HyperTransport interface supports serial 8 bit links of 4 Gbit/s per lane resulting in a total bandwidth of up to 4 GByte/s.

4.3.5 HTX Board

To evaluate our architecture, the HTX3 board was deployed, which we propose in [101]. The card as shown in Figure 4-10 employs three Xilinx Virtex5 FPGAs which provide a combined amount of up to 500k LUTs. The board, furthermore, contains an SODIMM socket for DDR2 memory, USB and various expansion connectors. The host processor is interfaced through a HyperTransport 3 Extension interface which supports link frequencies of up to 4 Gbit/s. To realize the HyperTransport interface, it is required to utilize the FPGA's serial transceivers (GTX) which are the only I/O components that operate at such high frequencies. However, as the GTX components do not natively support the HyperTransport standard, it was required to develop a custom Physical Layer Interface (PHY) block. In [100] we have proposed a HyperTransport 3 PHY for FPGAs which implements the electrical and low level protocol layers of HyperTransport. Due to the complexity of the specification this represents a challenging task. In particular, the PHY needs to operate at a very large spectrum of line-rates comprising low speed operation at 200 MHz as well as high speed modes of several Gigahertz. To aggravate the implementation, HyperTransport defines different data sampling mechanisms. While the low speed modes utilize a source synchronous clock to sample the incoming data lanes, the high speed modes utilize a clock data recovery mechanism to extract a sampling clock from each individual lane. As HyperTransport systems are initialized in the low speed mode and then change to higher frequencies at runtime, a dynamic, partial reconfiguration of the GTX blocks is required. This functionality is realized through a dynamic reconfiguration state machine within the PHY. Further components include an oversampling mechanism to sample the low frequency signals and a bitslip logic to align to the bytes within the stream and a channel bonding block which aligns the different lanes.

Figure 4-10:FPGA based Prototype

4.3.6 BIOS Firmware

To utilize our memory extension device in a compute system, the BIOS firmware has to be modified. Although, this represents a complex task it represents the only way to enable transparent access of our memory extension device by software. The benefit is a seamless integration without any modifications to the operating system, middleware or application software. The challenge is that our solution represents a new device class that is not supported by the standard BIOS firmwares. Our device supports a subset of features

provided by a processor like the memory controller, however, it is not able to execute code nor does it implement caches. The particular tasks that need to be solved are described in the following.

- While performing cache coherent device enumeration the device needs to be marked as a special memory extension device.
- The DRAM initialization needs to be performed and a specific memory range needs to be mapped to the device.
- The routing and memory address range tables on all coherent devices (in a multiprocessor system) need to be modified such that the physical memory addresses used by all processors point to the extension device correctly.
- The resource allocation tables which represent the interface between the firmware and the operating system need to be modified. These tables describe the architecture of the memory subsystem, in particular the size, the location and the caching attributes of specific memory ranges.
- The device can be configured for asymmetric probing by disabling cache coherent probes targeting the memory extension device to increase the performance.

These fundamental modifications of the BIOS require access to the firmware code. The Coreboot project, previously known as LinuxBIOS provides open source firmware code for a large number of platforms, including the Tyan S2912E motherboard which has been employed for our prototype system. We have successfully added support of our memory extension device to the coreboot firmware which enables any software application to access the memory extension as regular memory without any modifications. Full cache coherency is supported to enable multi-threaded applications to share the memory extension.

4.4 EVALUATION

4.4.1 Evaluation Methodology

To evaluate our architecture we have opted for a real world prototype implementation instead of a simulation based approach. Although, this approach requires greater efforts to generate first evaluation results, our experience shows that simulations do not provide sufficiently meaningful results. Most simulators do not model the real system accurately enough to deliver comparable results, especially for highly complex systems. To evaluate our proposed approach it is required to embrace the whole system architecture including multiprocessor, multicore configurations, the memory subsystem, BIOS firmware, Operating System, Middleware and software applications, preferably in a cycle accurate fashion. Such requirements are only provided by a full-fledged prototype implementation. Furthermore, it was not obvious that our architecture could be realized in available commodity-off-the-shelf server systems without major hardware and software modifications. However, our successful bringup of the FPGA based prototype in a real system including software integration, proves the feasibility of our design. In the following we present different microbenchmarks to evaluate our architecture. We, furthermore, evaluate our asymmetric probing and DMA offloading technique and analyze the architecture in terms of pins, performance and power.

4.4.2 Memory Extension Benchmark

Stream benchmark performance results of the presented memory extension architecture were obtained. The measurements were carried out on a Tyan S2912e based system supporting two 2.8 GHz Opteron Shanghai processors with a single DDR2 667 RAM module (KVR667D2D8P5K2/4G) connected to each node. The system is running a Linux kernel versioned 2.6.31 with NUMA support enabled. System memory available to Linux was limited to 1 GB in order to allow direct DRAM access via the device file system (Linux allows to map the main and extension memory as a memory device into the device tree). In order to compare accesses of a CPU to local DRAM, remote DRAM or DRAM provided by the prototype design, the BIOS firmware was modified as described above.

Via *numactl*, the benchmarking process was bound to the first node (node0) in the system. The HTX link is connected to the second node (node2) of the system resulting in a two hop distance to the prototype design. When binding the benchmarking process to the second node, the latency, therefore, decreases by approximately 40 ns. This shows the effect of the NUMA architecture to the Stream benchmark. The results of the benchmarking process are presented in Table 4-1 below.

Table 4-1: Stream Benchmark Results

	DRAM node0	DRAM node0 with FPGA	DRAM node1	DRAM node1 with FPGA	FPGA Memory Extension
Bandwidth	4935 MB/s	4935 MB/s	3708 MB/s	3708 MB/s	384 MB/s
Latency	80 ns	80 ns	140 ns	140 ns	866 ns

The first observation of the results is that the Stream performance is not degraded due to the slower FPGA based cache coherent device in the system. The architecture successfully avoids the cache coherency overhead by applying the asymmetric probing technique. This attribute is vital for scalable coherent memory extensions as it allows to scale the amount of cache coherent memory extension devices is the system without affecting performance negatively. The second observation can be made regarding the bandwidth and latency performance numbers of the FPGA prototype. They are perceptibly lower than of the local DRAM which can be explained by the significantly lower clock frequency of the FPGA prototype. However, regarding the fact that the MCT of the prototype is clocked at approximately a tenth of the speed of the Opteron controllers, the results are very promising. To the best of our knowledge the full round trip latency of a read access of 866ns already provides a significant improvement over all other available memory extension approaches. Texas Memory systems states a memory access latency of 15 microseconds [124] while swapping to remote memory over InfiniBand consumes several microseconds [93] for minimal-sized data values. The latency advantage of our solution is crucial to support large cache coherent shared memory effectively. A possible ASIC implementation and higher HyperTransport link frequencies will allow our architecture to provide a DRAM performance that is comparable to the Opteron's.

4.4.3 DMA Offloading Engine

We have conducted similar measurements of the DMA offloading engine. The benchmark utilized the same Tyan based server system together with the FPGA based memcpy accelerator. In contrast to the previous measurements which were initiated by the processor we now present DMA measurements conducted by the FPGA with no processor interaction. In this case, our architecture provides an even more reduced latency value of 688 ns for the full round-trip read access. Furthermore, the bandwidth is improved to up to 1600 MB/s for write accesses. The results for different transfer sizes are shown in Figure 4-11. As for DMA transactions, zero CPU load is involved. Our approach noticeably disburdens the CPU while maintaining high memory-to-memory copy performance.

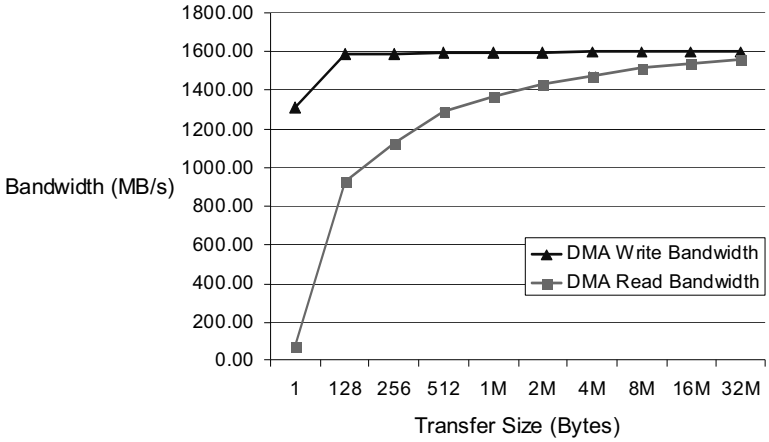

Figure 4-11:DMA Bandwidth Performance of different Transfers

4.4.4 Architecture Analysis

We have analyzed our architecture in terms of resources, memory capacity and power consumption. As already discussed, the package pin limitation of future processors will limit the scalability of memory bandwidth. To support a two DIMM channel the Opteron requires over 160 signal pins, while our memory extension device only requires a 44 pin HyperTransport link (power and ground pins are not included for both approaches). Figure 4-2 provides a comparison of different system configurations with 8 GByte memory extension devices. It shows that attaching memory using our proposed low pin count serial interface significantly improves the capacity per pin ratio. Needless to say, memory access latency increases especially for the daisy-chained configurations.

Table 4-2: Memory Capacity per pin Ratio

	Single Opteron	1 Memory Extension	2 Memory Extension	3 Memory Extensions Daisy Chained
Capacity	8 GByte	8 GByte	16 GByte	24 GByte
Pin count	160 pins	44 pins	88 pins	44 pins
Ratio	50 MB/pin	181 MB/pin	181 MB/pin	545 MB/pin

Our prototype design has been furthermore analyzed in terms of power consumption. While an AMD Opteron node typically consumes between 65 and 100 Watt, our solution requires a moderate 11.1 Watts. This value has been reported by the Xilinx Power Estimator whose results match real world measurements within 5%. While it is evident that a full fledged processor has a higher power consumption than a memory extension controller, a processor represents a very power inefficient device for extending memory capacity. For example, a system that employes three memory extension devices offers the same memory capacity as a four-way (65 Watt) AMD Opteron system while reducing power consumption from 260 to 110 Watts. Power constrained, memory bound applications, therefore, can greatly benefit from our proposed memory extension technique.

4.5 CONCLUSION

In this chapter, we have proposed a novel cache coherent memory extension device to build large shared memory AMD Opteron based SMP systems. Due to package pin limitations and DRAM scalability issues, we predict that the memory capacity and bandwidth will not be able to scale at the same speed as processor technology. This will result in a severe bottleneck for future computing systems. Our architecture addresses this issue by introducing a scalable approach that can extend the memory capacity of a system by terabytes of memory using the low pin count and cache coherent HyperTransport host CPU interface. Furthermore, we introduce a technique to minimize the overhead of the cache coherency protocol and a DMA offloading technique that enables execution of memory-to-memory copy transactions with zero CPU overhead. We have implemented our complete architecture in real hardware using a specifically developed FPGA based prototype. The memory extension device has been successfully employed in a commodity-off-the-shelf server system including full software and operating system integration. For the evaluation of our architecture we present promising micro benchmark results that show sub-microsecond memory access latencies, outperforming comparable approaches.

The main goal of our proposed memory extension device is to address the compute to memory ratio. Due to the low cost of off-the-shelf processors it has been proposed to deploy additional CPUs that are solely used as additional memory controllers. While this may be correct for existing systems due to the lack of commercially available memory extension devices it represents an inefficient solution. The amount of silicon that is required to manufacture a memory extension device as proposed in this work is significantly smaller than that of a processor. A high-volume ASIC production of the memory extension should, hence, provide significant cost advantages as well as reduced power consumption over a processor based solution. Furthermore, in contrast to the SMP approach, our solution is scalable. We can support a large number of memory extension devices without increasing the cache coherency overhead due to the asymmetric probing technique. Finally, our solution can provide higher performance by offloading tasks to the memory extension device.

While we have presented a novel, unique solution to the memory capacity problem there still exists a number of challenges which we have not addressed in this work. Most significantly, additional efforts need to be invested in benchmarking more meaningful applications. As we are relying on relatively slow FPGA based hardware, results will not reflect the performance of an ASIC implementation, however, applications like in-memory databases that are performance limited by the page swapping overhead should experience a visible improvement. To transform the technique into a working product an ASIC implementation is required.

Another perception is that there still exist a number of techniques which can further increase the performance of our architecture including optimizations of the DMA engine like pre-fetching. Furthermore, the memory extension approach enables to increase the memory capacity through compression and data de-duplication mechanisms [60]. By compacting data and by avoiding redundant data copies within the memory, the capacity can be virtually increased. In addition our approach could be easily modified to support other types of memory like flash or phase change memory. This feature would enable designs that support large amounts of non-volatile memory combined with fast accessible DRAM. For all such techniques, our design represents an excellent base architecture.

CHAPTER 5
CONCLUSION

5.1 SUMMARY

Extending the scalability of future computer architectures by increasing the number of functional units appears to be the only method that allows to maintain the increase in compute performance we have enjoyed in the past. There exist no other approaches besides the proposed massively parallel architectures that are capable of transforming the increasing amount of transistors provided through Moore's Law into higher performance. Future Exascale computer systems need to execute in the order of a billion of threads which challenges the scalability of current systems in all layers. It is evident that the number of functional units need to increase significantly both on-chip as well as on the compute node and on the network level. However, a naïve, manifold duplication of existing components will not lead to the desired result due to the inability of current architectures to scale. The main reason is the limited performance of the communication interface between the components which prevents them to deliver their full potential. In most architectures the communication overhead is too large, rendering the potential performance advantage of parallelizing the problem irrelevant. To address this problem we have proposed three techniques in this dissertation that improve the communication infrastructure, and therefore, the scalability of current architectures. The proposed mechanisms approach the problem in three different layers of the computer architecture.

The first technique addressed the shortcomings within the communication subsystem between multiple nodes in a cluster system. Traditional systems are interconnected using loosely coupled interconnects like Ethernet and InfiniBand which enable to interconnect a large number of nodes, however, they only provide a moderate communication performance. Especially, the high latency makes fine grain communication very costly which limits the speedup of applications through parallelization rendering it impossible to exploit the available compute power of large clusters. To address this issue, we introduced the TCCluster approach that virtually integrates the network interface into the processor by reusing available interconnect structures. This enables a tightly coupled cluster which capitalizes much more fine grain parallelization, leading to much better overall scalability. The approach which was implemented in an AMD Opteron based system shows an order of magnitude improvement in latency and bandwidth for specific applications over

comparable approaches. As our technique does not require additional hardware for the network interconnect it in addition represents a very dense and power efficient solution. The further contributions of the TCCluster chapter were manifold, including a comprehensive summary of interval based routing techniques that can be applied in this context as well as a novel class of network topologies that have been developed for this particular technique. Furthermore, a detailed analysis of the AMD Opteron architecture was provided which exposes its bottlenecks and solutions to cope with them. The TCCluster approach introduced a disrupting class of highly scalable and tightly coupled systems with the potential of outperforming established cluster based systems considerably.

In the third chapter we introduced a novel framework for on-chip networks that represents a scalable alternative for the interconnection of different functional units within a microchip. While the transistor count and, thereby, the number of functional units within integrated silicon devices constantly increases, the architecture of the on-chip communication infrastructure gets increasingly relevant. Apart from providing a high throughput and a low latency, the network needs to be scalable to keep up with the ever increasing number of interconnected components of future architectures. We presented different techniques to achieve this goal including different topologies and multistage distributed architectures. Another increasingly important requirement in chip design is the minimization of development and verification efforts to reduce the time to market. More complex designs, in general, lead to a greater design space and, thereby, increase the development and verification efforts of a new architecture. Our framework introduced a novel component based design methodology which enables flexible designs that can be generated automatically. Our engine generates efficient, synthesizable hardware structures that can be directly integrated into chip designs which significantly accelerates the design process. It furthermore enables a rapid prototyping design methodology as different design aspects can be modified by regenerating the design using different parameters. In conclusion, the framework provides a very efficient solution to build efficient, scalable networks-on-chips for future, highly integrated circuits.

The third technique addresses the memory bandwidth and capacity problem which processor architectures are facing currently. As the number of components on a microprocessor increases, more memory bandwidth and capacity is required. However, as it is predicted that the number of I/O pins on the next generation's chip packages will not scale at the same rate as the silicon transistors it will become increasingly difficult to satisfy the memory demands of future devices. The processor-memory interface may evolve as a critical bottleneck limiting compute performance. We addressed this challenge by introducing a scalable, cache coherent memory extension controller connected to the processor via a high-speed low pin count interface, and thereby, by-passing the pin-limitation of future microprocessors. By utilizing SERDES technology an increased amount of silicon area is invested to increase the per-pin bandwidth. This approach is beneficial, as the number of transistors increases much faster than the number of package pins. By deploying our technique the number of memory interfaces can be increased creating a new class of NUMA memory topologies. As our architecture provides the capability of relaying memory transactions to further memory extension devices, different topologies consisting of a large amount of memory extensions can be realized. The approach enables to add additional memory extension devices without increasing the number of processor I/O pins, representing a scalable solution even for future processor architectures. This in turn results in a significantly increased memory capacity as well as in a much better capacity-to-power ratio. We presented additional contributions in the form of a DMA offloading technique that allow to handle large memory copy transfers by the memory controller itself, freeing up a significant amount of compute resources. The approach, which has been implemented in a real system using FPGAs and AMD Opteron based processors, demonstrates a feasible solution for the memory capacity and bandwidth problem of current and future microprocessors.

5.2 Looking Back

In this paragraph I look back at my doctoral thesis and discuss the impact of my dissertation to the field of computer architecture. The most cited papers are VELO [96] which received the best paper award at the 37th International Conference on Parallel Processing (ICPP 2008) and the HTX-Board [52] publications which, thus, had the biggest impact on the research community. The novel communication mechanism we proposed in the VELO paper has influenced research in the area of distributed shared memory and has been applied in these projects successfully. The success of the HTX-Board paper originates from the fact that the proposed rapid prototyping platform has been applied by several research groups in the area of communication and computation acceleration. Another excellent contribution has been provided in the form of the "A HyperTransport 3 Physical Layer Interface for FPGAs" paper that received the best paper award at the 5th International Workshop on Applied Reconfigurable Computing (ARC 2009).

Apart from impacting the research community, the HTX-Board as well as the HyperTransport 3 verification platform [101] have been applied in joint projects with leading companies from industry including AMD, SUN Microsystems and IBM. In the scope of these projects my work had a significant impact on the products and research activities carried out by the three companies. AMD utilized the HyperTransport 3 verification platform [101] in combination with our proposed HyperTransport 3 core [78] for internal debugging, qualification and verification of Opteron processors. Our FPGA platform enabled a flexible test environment for testing AMD processors and third party hardware in a real world system. In particular, the HyperTransport interfaces of the AMD Magny Cours 12 core processor released in 2010 were tested and evaluated using our verification platform. We, thereby, have actively contributed to the success of one of the most complex x86 processors ever made.

SUN Microsystems has applied our technology in the Scalable Interface project which aims at creating large, high performance computing clusters based on InfiniBand. In the course of our collaboration, a 65 nm ASIC was developed that bridges cache coherent HyperTransport to InfiniBand. We actively contributed with the code of the

CHAPTER 5 CONCLUSION

HyperTransport core and developed a sophisticated verification environment within this work. The joint research lead to significant improvements in the cache coherent domain of the design which resulted in a performance increase of the entire architecture. IBM has utilized our technology in research projects focused on cache coherent memory controllers.

5.3 LOOKING FORWARD

The techniques presented in this dissertation influence the research field of computer architecture by introducing a new set of scalable high performance computer systems. In this final paragraph I will envision an architecture that combines the three proposed techniques into a single integrated system offering the required scalability of future Exascale computing systems. It will be further pointed out which challenges need to be addressed and which tasks need to be solved to enable such systems. All three principles that have been introduced in this work focus on improving the communication infrastructure between components on different layers inside a computer system. Furthermore, two of the proposed techniques leverage the existing HyperTransport communication interface component for novel purposes. In TCCluster, the host interface is exploited as a cluster interconnect, while in the memory extension part of this work the same interface is utilized to access a greater memory capacity with higher bandwidth. By combining these techniques we introduce the concept of *interface consolidation*.

The principle idea behind this concept is that the main components of a system should be logically separated from each other and communicate by utilizing a single, unified interface. Similarly, to the HTAX methodology we propose for on-chip architectures, this principle introduces a component based design methodology on the system level. By deploying a unified interface for chip-to-chip, node-to-node and processor-to-memory transactions, off-the-shelf components can be utilized in different configurations to optimize the design for a specific purpose. For example, memory bound applications can utilize systems that deploy a large number of memory controllers while embarrassingly parallel applications can leverage designs that implement many interconnected processor cores. Applications that rely on high single thread performance, on the other hand, can utilize symmetric multiprocessor systems, consisting of a limited number of processors connected via a cache coherent interconnect like HyperTransport. This flexible approach optimizes both cost and power efficiency for a large set of applications. Current x86 architectures are lacking this flexibility as they follow an all-in-one approach. To minimize development costs, they are required to cover a large application space which leads to non-optimal efficiency. The solution to this problem is to apply a common unified

interface between all components which allows to interchange them arbitrarily, increasing the flexibility of the system. In lieu of developing optimized architectures for particular applications, the component based approach allows to increase efficiency simply by choosing the correct combination of basic components.

The flexibility of the architecture can be furthermore increased by applying the same protocol not only for chip-to-chip but also for on-chip communication. The EXTOLL application we describe as the use-case for HTAX utilizes the HTOC protocol for communicating between on-chip components which represents a one-to-one mapping of the external HyperTransport protocol. Deploying this concept, components can be arranged flexibly on the system level as well as on the chip level.

The design modifications of current architectures which are required to realize such a flexible architecture are moderate. As shown in this dissertation, the HyperTransport interface provides the necessary flexibility to be utilized as a cache coherent SMP interface, as well as a memory extension and a cluster interconnect. The current AMD Opteron architecture we analyzed for our work shows limitations regarding the supported topologies and lacks many features that are common in interconnection networks. More flexible routing mechanisms as well as congestion management techniques and an increased number of addressable endpoints represent necessary features for increasing the scalability. Nevertheless, these techniques are well understood and they could be incorporated into current architectures with moderate efforts. While these extensions extend the scalability and will be required for Exascale systems, the component based approach presented herein showed that current architectures can be utilized to build large systems that utilize the interface consolidation concept. Figure 5-1 shows such a system. It combines the ideas of this work and represents a practical implementation that can be deployed using existing components. The center of the block diagram shows the compute blades which each deploy two AMD Opteron processor modules. The compute blades are interconnected via the TCCluster interconnect and are attached to memory extension blades on the right. The left side shows the EXTOLL interconnection network which is linked to the blades using the non-coherent HyperTransport interface. The advantages of this architecture over traditional systems are the increased interconnect performance

provided by the TCCluster interconnect, the increased memory capacity offered by memory extension devices and the adaptability to specific workloads and applications.

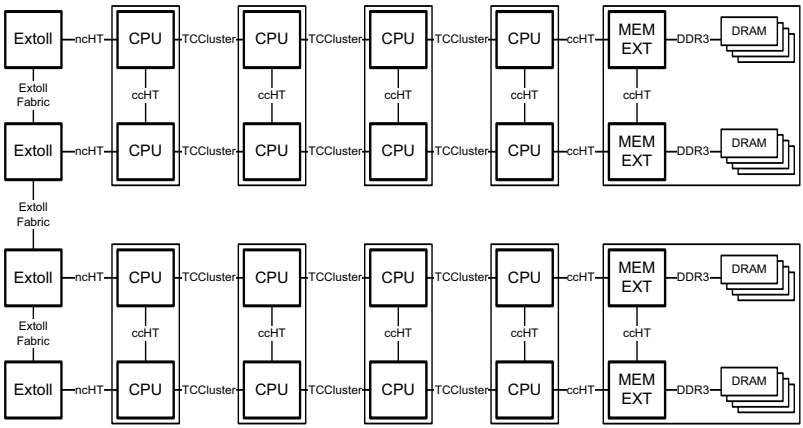

Figure 5-1:Component Based System Design

The scalability of a TCCluster system is limited by the restricted physical address space of the AMD Opteron architecture to medium sized systems of up to 1K processor nodes. Using 12-core Magny Cours Opteron chips such a system provides 12K cores. To enable massively parallel systems of a much larger scope the proposed solution deploys the scalable EXTOLL interconnect which supports systems of up to 64K compute nodes. Utilizing this technique, more than half a million cores can be interconnected in a high performance compute cluster. As the EXTOLL interconnect uses non-coherent HyperTransport to interface to the processors it blends nicely into the interface consolidation approach. Employing our component based approach enables a whole new set of system architectures of different size, with variable processor-memory configurations and different topologies by adjusting the layout of the TCCluster and EXTOLL links. We believe that this flexibility makes our architecture an excellent candidate for future Exascale systems.

CHAPTER 5 CONCLUSION

List of Figures

List of Tables

REFERENCES

[1] 3 LEAF SYSTEMS: Enabling the Dynamic Data Center, whitepaper, 2009.

[2] Advanced Micro Devices: BIOS and Kernel Developer's Guide (BKDG) For AMD Familiy 10h Processors, rev. 3.00.

[3] A. Agarwal, "The Tile processor: A 64-core multicore for embedded processing," Proceedings of HPEC Workshop, 2007.

[4] S. Alam, R. Barrett, M. Bast, and M. Fahey, "Early evaluation of IBM BlueGene/P," Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC08), 2008.

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele Jr, S. Tobin-Hochstadt, J. Dias, C. Eastlund, and others, "The Fortress language specification," Sun Microsystems, vol. 139, 2005, p. 140.

[6] AMBA Specification, rev. 2.0, ARM.

[7] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," Proceedings of the AFIPS Spring Joint Computer Conference (Reston, Va.), AFIPS Press, Arlington, VA, 1967.

[8] J. Archibald, J. L. Baer: Cache coherence protocols: evaluation using a multiprocessor simulation model. In Proceedings of ACM Transactions on Computer Systems (TOCS), 1986.

[9] K. Asanovic, R. Bodik, B. Catanzaro, and J. Gebis, "The landscape of parallel computing research: A view from berkeley," *UC Berkeley Tech Report*, 2006.

[10]E. Bakker, J.V. Leeuwen, and R. Tan, "Linear interval routing," The Computer Journal, 1991.

[11]B. Black and J. Shen, "Calibration of microprocessor performance models," Computer, 2002.

[12]K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, J. C. Sancho: Entering the petaflop era: the architecture and performance of Roadrunner. In Proceedings of the 2008 ACM/IEEE Conference on Supercomputing - Volume 00 (Austin, Texas, November 15 - 21, 2008). Conference on High Performance Networking and Computing. IEEE Press, Piscataway, NJ, 1-11.

[13]R. Baumann, "Soft errors in advanced computer systems," IEEE Design & Test of Computers, vol. 22, 2005, p. 258–266.

[14]E. Beigné, F. Clermidy, J. Durupt, H. Lhermet, S. Miermont, Y. Thonnart, T. Xuan, A. Valentian, D. Varreau, and P. Vivet, "An asynchronous power aware and adaptive noc based circuit," IEEE Journal of SolidState Circuits, vol. 44, 2009, pp. 1167-1177.

[15]L. Benini and G. D. Micheli, "Networks on chips: technology and tools," 2006.

[16]L. Benini and G.D. Micheli, "Networks on chip: a new paradigm for systems on chip design," Design, Automation and Test in Europe Conference and Exhibition, 2002., 2002, p. 418–419.

[17]D. Bertozzi and L. Benini, "Xpipes: A network-on-chip architecture for gigascale systems-on-chip," IEEE Circuits and Systems magazine, vol. 4, 2004, p. 18–31.

[18]T. Bjerregaard, S. Mahadevan, R. Olsen, and J. Spars, "An OCP compliant network adapter for GALS-based SoC design using the MANGO network-on-chip."

[19]B. Black, M. Annavaram, N. Brekelbaum, and J. DeVale, "Die Stacking (3D) Microarchitecture," *39th Annual IEEE/ACM International Symposium on Microarchitecture*.

[20]H. Bodlaender, R.B. Tan, D.M. Thilikos, and J. Leeuwen, "On Interval Routing Schemes and Treewidth," Information and Computation, vol. 139, 1997, pp. 92-109.

[21]M.T. Bohr, "Interconnect Scaling - The Real Limiter to High Performance ULSI," Solid State Technology, 1996.

[22]R. Boppana and C. Raghavendra, “Designing efficient Benes and Banyan based input-buffered ATM switches,” Communications, 1999. ICC’99. 1999 IEEE International Conference on, IEEE, 2002, p. 1826–1830..

[23]S. Borkar, "Thousand core chips: a technology perspective," Annual ACM IEEE Design Automation Conference, 2007, p. 746.

[24]J. Borowski, U. Bruening, “Asymmetric Probing Prototype,” AMD Accelerated Computing Symposium, 2008.

[25]R. Brightwell, K. Pedretti, K. D. Underwood: Initial Performance Evaluation of the Cray SeaStar Interconnect. In Proceedings of 13th Symposium on High Performance Interconnects (HOTI'05), 2005.

[26]D. Burger, J. Goodman, and A. Kaegi, "Memory bandwidth limitations of future microprocessors," *ACM SIGARCH Computer Architecture News*, vol. 24, 1996, p. 89.

[27]D. Burger, "Hardware techniques to improve the performance of the processor/ memory interface," Dissertation, 1998.

[28]E. Cannon, D. Reinhardt, M. Gordon, and P. Makowenskyj, "SRAM SER in 90, 130 and 180 nm bulk and SOI technologies," IEEE international reliability physics symposium, 2004, p. 300–304.

[29]D. Chapiro, "Globally-asynchronous locally-synchronous systems," Phd. thesis, 1984.

[30]W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, Introduction to UPC and language specification, Citeseer, 1999.

[31]T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell Broadband Engine Architecture and its first implementation-A performance view," IBM Journal of Research and Development, 2007.

[32]S. Computing, "Trio Departmental Supercomputer DSMP ™ System Software," Life Sciences, 2008.

[33]P. Conway, N. Kalyanasundharam, G. Donley, and K. Lepak, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," IEEE Micro, 2010, pp. 16-29.

[34]P. Conway and B. Hughes, "The AMD Opteron northbridge architecture," *IEEE Micro*, vol. 27, 2007, p. 10–21.

[35]M. Coppola, S. Curaba, and M.D. Grammatikakis, "OCCN: a network-on-chip modeling and simulation framework," Proceedings of the, 2004.

[36]M. Coppola, R. Locatelli, G. Maruccia, and L. Pieralisi, "Spidergon: a novel on-chip communication network," System-on-Chip, 2004.

[37]M. Dall'Osso, G. Biccari, L. Giovannini, D. Bertozzi, and L. Benini, "Xpipes: a latency insensitive parameterized network-on-chip architecture for multi-processor SoCs," Computer Design, 2003.

[38]W. Dally and C. Seitz, "Deadlock-free message routing in multiprocessor interconnection networks," IEEE Transactions on computers, vol. 100, 1987, p. 547–553.

[39]W. Dally and B. Towles, "Principles and Practices of Interconnection Networks. 2004," Morgan Kaufmann, San Francisco.

[40]W. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," Design Automation Conference, 2001, p. 684–689.

[41]L. Dickman, G. Lindahl, D. Olson, J. Rubin, J. Broughton: Pathscale InfiniPath: a first look. In Proceedings of 13th Symposium on High Performance Interconnects, 2005.

[42]D. Doerfler, "An Analysis of the Pathscale Inc. InfiniBand Host Channel Adapter, InfiniPath," Sandia report SAND2005-5199 (Aug. 2005), 2005.

[43]S. Donthi and R. Haggard, A survey of dynamically reconfigurable FPGA devices, IEEE, 2003.

[44]Y. Dourisboure and C. Gavoille, "Interval Routing for Generalized Hypercube-Like Graphs," 2007.

[45]J. Duato, S. Yalamanchili, and L. Ni, "Interconnection networks: An engineering approach," 2003.

[46]J. Duato, I. Johnson, J. Flich, F. Naven, P. Garcia and T. Nachiondo "A new scalable and cost-effective congestion management strategy for lossless multistage interconnection networks," 2005.

[47]T. Dumitras, S. Kerner, and R. Marculescu, "Towards on-chip fault-tolerant communication," Proceedings of the 2003 Asia and South Pacific Design Automation Conference, ACM, 2003, p. 232.

[48]H Eberle, P.J. Garcia, J. Flich, J. Duato, R. Drost, N. Gura, D. Hopkins, and W. Olesinski, "High-radix crossbar switches enabled by proximity communication," *Conference on High Performance Networking and Computing*, 2008.

[49]A. Ehliar, J. Eilert, and D. Liu, "A comparison of three FPGA optimized NoC architectures," Swedish System-on-Chip Conference, 2007.

[50]R. Esser and R. Knecht, "Intel Paragon XP/S-Architecture and Software Enviroment," Anwendungen, Architekturen, Trends, Seminar, 1993.

[51]P. Fraigniaud and C. Gavoille, "Interval routing schemes," Algorithmica, 1998.

[52]H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, and U. Brüning, "The HTX-Board: A Rapid Prototyping Station," 3rd annual FPGAWorld Conference, Citeseer, 2006.

[53]P. Garcia, F. Quiles, J. Flich, J. Duato, I. Johnson, and F. Naven, "Efficient, scalable congestion management for interconnection networks," IEEE Micro, vol. 26, 2006, p. 52–66.

[54]H. Garcia-Molina and L. Rogers, "Performance through memory," *ACM SIGMETRICS Performance Evaluation Review*, vol. 15, 1987, p. 131.

[55]C. Gavoille, "A survey on interval routing," Theoretical Computer Science, 2000.

[56]C.J. Glass and L.M. Ni, "The turn model for adaptive routing," Journal of the ACM (JACM), vol. 41, 1994, p. 874.

[57]J. Goodman, "Using cache memory to reduce processor-memory traffic," 25 years of the international symposium on Computer Architecture (ISCA), 1998.

[58]K. Goossens, J. Dielissen, O. Gangwal, S. Pestana, a. Radulescu, and E. Rijpkema, "A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification," Design, Automation and Test in Europe, pp. 1182-1187.

[59]K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal Network on Chip:Concepts, Architectures, and Implementations," IEEE Design and Test of Computers, vol. 22, 2005, pp. 414-421.

[60]D. Gupta, S. Lee, M. Vrable, S. Savage, A. Snoeren, G. Varghese, G. Voelker, and A. Vahdat, "Difference Engine: Harnessing Memory Redundancy in Virtual Machines," Communications of the ACM, vol. 53, 2010, p. 85–93.

[61]J.L. Gustafson, "Reevaluating Amdahl's law," Communications of the ACM, vol. 31, 1988, p. 532.

[62]D. Gustavson: The Scalable Coherent Interface and related standards projects. IEEE Micro. 1992.

[63]R. Hamming, "Error detecting and error correcting codes," Bell System Technical Journal, vol. 29, 1950, p. 147–160.

[64]D. S. Henry, C. F. Joerg: A Tightly-Coupled Processor-Network Interface. In Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V), 1992 .

[65]A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, "Network on chip: An architecture for billion transistor era," Proceeding of the IEEE NorChip Conference, Citeseer, 2000, p. 166–173.

[66]M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems, 1993.

[67]M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era," Computer, vol. 41, 2008, pp. 33-38.

[68]W. Hillis, "The connection machine," 1989.

[69]R. Ho, K. Mai, and M. Horowitz, "The future of wires," Proceedings of the IEEE, vol. 89, 2001, p. 490–504.

[70]H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote Store Programming," Lecture notes on High Performance Embedded Architectures and Compilers, 2010.

[71]Hp, "Memory technology evolution: an overview of system memory technologies," *http://tinyurl.com/ctfjs2*.

[72]F. Hwang and A. Jajszczyk, "On nonblocking multiconnection networks," IEEE Transactions on Communications, 1986.

[73]HyperTransport Consortium: HyperTransport I/O Link Specification, revision 3.10, 2008.

[74]"Heiner Litz, "HyperTransport On-Chip (HTOC) Specification, ver. 14, 2010.

[75]SIA International Technology Roadmap for Semiconductors Edition 2007," 2007.

[76]A. Jantsch and H. Tenhunen, "Networks on chip," 2003.

[77]N. Jouppi and D. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," ACM SIGARCH Computer Architecture News, 1989.

[78]B. Kalisch, A. Giese, H. Litz, U. Brüning, and G. Ausgabe, "HyperTransport 3 Core: A Next Generation Host Interface with Extremely High Bandwidth," Proceeding of the First International Workshop on Hyper-Transport Research and Applications (WHTRA'09), 2009.

[79]M. Karol, M. Hluchyj, and S. Morgan, "Input versus output queueing on a space-division packet switch," IEEE Transactions on Communications,1987.

[80]T. Kgil, SD'Souza, A. Saidi, N. Binkert, and R. Dreslinski, "PicoServer: Using 3D Stacking Technology To Enable A Compact Energy Efficient Chip Multiprocessor," *ACM SIGOPS Operating Systems Review*.

[81]J. Kim, W.J. Dally, and D. Abts, "Flattened butterfly: a cost-efficient topology for high-radix networks," Proceedings of the 34th annual international symposium on Computer architecture, ACM, 2007, p. 126–137.

[82]D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," ISCA '98: 25 years of the international symposia on Computer architecture, 1998.

[83]S. Kumar, A. Jantsch, J.P. Soininen, M. Forsell, M. Millberg, J. Oeberg, K. Tiensyrja, and A. Hemani, "A network on chip architecture and design methodology," IEEE Symp. on VLSI ISVLSI02, Apr, 2002, pp. 105-112.

[84]A. Kumar, I. Ovadia, J. Huisken, and H. Corporaal, "Reconfigurable Multi-Processor Network-on-Chip on FPGA," Proceedings of the Annual Conference of the Advanced School for Computing and Imaging, 2006.

[85]P. Kogge, S. Lead, D. Campbell, J. Hiller, M. Richards, and A. Snavely, "Exascale Computing Study : Technology Challenges in Achieving Exascale Systems," Government PROcurement.

[86]P. Koopman, "32-Bit Cyclic Redundancy Codes for Internet Applications," International Conference on Dependable Systems and Networks (DSN'02), 2002.

[87]R. Kota, R. Oehler, N. Inc, and T. Austin, "Horus: large-scale symmetric multiprocessing for Opteron systems," *IEEE Micro*, vol. 25, 2005, p. 30–40.

[88]D. Koufaty and D. Marr, "Hyperthreading Technology in the Netburst Microarchitecture," IEEE Micro, vol. 23, 2003, pp. 56-65.

[89]J. Laudon and D. Lenoski, "The SGI Origin," *ACM SIGARCH Computer Architecture News*, vol. 25, 1997, pp. 241-251.

[90]S. Lee, S. Min, and R. Eigenmann, "OpenMP to GPGPU: a compiler framework for automatic translation and optimization," Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2009.

[91]J.V. Leeuwen and R. Tan, "Interval routing," The Computer Journal, 1987.

[92]K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 7, 1989.

[93]S. Liang, R. Noronha, and D. Panda, "Swapping to remote memory over InfiniBand: An approach using a high performance network block device," *IEEE Cluster Computing*, Citeseer, 2005.

[94]K. Lim, J. Chang, T. Mudge, P. Ranganathan, S.K. Reinhardt, and T.F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," *ACM SIGARCH Computer Architecture News*, vol. 37, 2009, p. 267.

[95]M. Linderman, J. Collins, and H. Wang, "Merge: a programming model for heterogeneous multi-core systems," Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, 2008.

[96]H. Litz, H. Froening, M. Nüssle, and U. Brüning, "VELO: A Novel Communication Engine for Ultra-low Latency Message Transfers," Parallel Processing, 2008. ICPP'08. 37th International Conference on, 2008, p. 238–245.

[97]H. Litz, M. Thuermer, and U. Bruening, "TCCluster: A Cluster Architecture Utilizing the Processor Host Interface as a Network Interconnect," CLUSTER '09. IEEE International Conference on Cluster Computing., 2010.

[98]H. Litz, H. Fröning, and U. Brüning, "HTAX: A Novel Framework for Flexible and High Performance Networks-on-Chip," Fourth Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC) in conjunction with Hipeac, 2010.

[99]H. Litz, H. Fröening, M. Nuessle and U. Brüning, "A HyperTransport Network Interface Controller for Ultra-low Latency Message Transfers," HyperTransport Consortium Whitepaper, 2007.

[100]H. Litz, H. Fröning, U. Brüning "A HyperTransport 3 Physical Layer Interface for FPGAs" 5th International Workshop on Applied Reconfigurable Computing (ARC 2009) , March 16 - 18, 2009, Karlsruhe, Germany.

[101]H. Litz, H. Fröning, M. Thürmer, U. Brüning "An FPGA based Verification Platform for HyperTransport 3.x" 19th International Conference on Field Programmable Logic and Applications (FPL 2009) , August 31 - September 2, 2009, Prag, Czech Republic.

[102]R. Minnich, J. Hendricks, and D. Webster, "The linux BIOS," Atlanta Linux Showcase, 2000, p. 21.

[103]G. Moore and others, "Cramming more components onto integrated circuits," *Proceedings of the IEEE*, vol. 86, 1998, p. 82–85.

[104]H. Moussa, O. Muller, A. Baghdadi, and M. Jézéquel, "Butterfly and benes-based on-chip communication networks for multiprocessor turbo decoding," Proceedings of the conference on Design, automation and test in Europe (DATE), EDA Consortium, 2007.

[105]J. Muttersbach, T. Villiger, and W. Fichtner, "Practical design of globally-asynchronous locally-synchronous systems," Asynchronous Circuits and Systems, 2000.

[106]MVAPICH: MPI over InfiniBand and iWARP. http://mvapich.cse.ohio-state.edu/.

[107]L. Ni and P. McKinley, "A survey of wormhole routing techniques in direct networks," COMPUTER, 1993.

[108]Numascale Whitepaper, “SMP Redux: You Can Have It All”.

[109]M. Nüssle, M. Scherer, and U. Brüning, "A resource optimized remote-memory-access architecture for low-latency communication," 2009 International Conference on Parallel Processing, ICPP 2009, IEEE, 2009, p. 220–227.

[110]G. Palermo, G. Mariani, C. Silvano, R. Locatelli, and M. Coppola, "Mapping and topology customization approaches for application-specific STNoC designs," Application-specific Systems, Architectures and Processors, 2007. ASAP. IEEE International Conf. on, 2007, p. 61–68.

[111]M. Palesi, R. Holsmark, and S. Kumar, "A methodology for design of application specific deadlock-free routing algorithms for NoC systems," Proceedings of the 4th …, 2006.

[112]W. Peterson and D. Brown, "Cyclic codes for error detection," Proceedings of the IRE, 1961.

[113]S. Raina, "Virtual Shared Memory: A Survey of Techniques and Systems," *cs.bris.ac.uk*.

[114]G. Regnier, R. Iyer, D. Newell, L. Cline, and A. Foong, "TCP Onloading for Data Center Servers," Computer, 2004.

[115]ScaleMP: Whitepaper, "The Versatile SMP ™ (vSMP) Architecture and Solutions Based on vSMP Foundation ™," *ReVision*, pp. 1-12.

[116]K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, and D, "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture," International Symposium on Computer Architecture, 2003.

[117]N. Santoro and R. Khatib, "Routing without routing tables," Techn. Report SCSTR-6, School of Computer Science, 1982.

[118]C. Savin, T. McSmythurs, and J. Czilli, "Binary tree search architecture for efficient implementation of round robin arbiters," ICASSP, 2004.

[119]P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," International Conference on Dependable Systems and Networks, 2002.

[120]J. Smith, "A study of branch prediction strategies," 25 years of the International Symposium on Computer Architecture, 1998.

[121]J. Smith and G. Sohi, "The microarchitecture of superscalar processors," Proceedings of the IEEE, vol. 83, 2002, p. 1609–1624.

[122]H. Sullivan and T.R. Bashkow, "A large scale, homogeneous, fully distributed parallel machine, I," International Symposium on Computer Architecture, vol. 5, 1977, p. 105.

[123]S. Sur, M. J. Koop, L. Chai, D. K. Panda: Performance Analysis and Evaluation of Mellanox ConnectX InfiniBand Architecture with Multi-Core Platforms. In Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects, 2007.

[124]Texas Memory Systems, "Increase Application Performance with Solid State Disks," *Memory*.

[125]M. Thottethodi, A. Lebeck, and S. Mukherjee, "Self-tuned congestion control for multiprocessor networks," hpca, Published by the IEEE Computer Society, 2001, p. 0107.

[126]J. Dongarra, M. Meuer, “TOP500 Supercomputer List,” June 2009 URL: http:// www.top500.org

[127]S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, and others, "An 80-tile sub-100-w teraflops processor in 65-nm cmos," IEEE Journal of SolidState Circuits, vol. 43, 2008, p. 29–41.

[128]P. Vellanki, N. Banerjee, and K. Chatha, "Quality-of-service and error control techniques for Mesh-based network-on-chip architectures," INTEGRATION, the VLSI journal, vol. 38, 2005, p. 353–382.

[129]D. Watts, "System x3755 Technical Introduction," *IBM Redpaper*, pp. 2006-2007.

[130]C. Whitby-Stevens: The Transputer. In Proceedings of the 12th ISCA, 1985.

[131]Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores Specification, Revision B.3, 2002.

[132]P. Wolkotte, G. Smit, G. Rauwerda, and L. Smit, "An energy-efficient reconfigurable circuit-switched network-on-chip," 19th IEEE International Parallel and Distributed Processing Symposium, 2005.

[133]W. Wulf and S. McKee, "Hitting the memory wall: Implications of the obvious," ACM SIGARCH Computer Architecture News, 1995.

[134]K. Yelick, D. Bonachea, W. Chen, and P. Colella, "Productivity and performance using partitioned global address space languages," PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation, 2007.

[135]B. Zerrouk, S. Tricot, B. Rottembourg, L. Patnaik, “Proper linear interval routing schemes”, Technical Report N 94-29, IBP-MASI, October 1994.

[136]B. Zerrouk, J.M. Blinn, and A. Greiner, "Encapsulating Networks and Routing," Proceedings of the 8th International Symposium on Parallel Processing, 1994, p. 546.