Tightly-Coupled and Fault-Tolerant Communication in Parallel Systems

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

vorgelegt von
Dipl.-Inf. David Christoph Slogsnat
aus Heidelberg

Mannheim, 2008

Dekan: Prof. Dr. Matthias Krause, Universität Mannheim

Referent: Prof. Dr. Ulrich Brüning, Universität Heidelberg

Koreferent: Prof. Dr. Reinhard Männer, Universität Heidelberg

Tag der mündlichen Prüfung: 4. August 2008

Abstract

The demand for processing power is increasing steadily. In the past, single processor architectures clearly dominated the markets. As instruction level parallelism is limited in most applications, significant performance can only be achieved in the future by exploiting parallelism at the higher levels of thread or process parallelism. As a consequence, modern “processors” incorporate multiple processor cores that form a single shared memory multiprocessor.

In such systems, high performance devices like network interface controllers are connected to processors and memory like every other input/output device over a hierarchy of peripheral interconnects. Thus, one target must be to couple coprocessors physically closer to main memory and to the processors of a computing node. This removes the overhead of today’s peripheral interconnect structures. Such a step is the direct connection of HyperTransport (HT) devices to Opteron processors, which is presented in this thesis.

Also, this work analyzes how communication from a device to processors can be optimized on the protocol level. As today’s computing nodes are shared memory systems, the cache coherence protocol is the central protocol for data exchange between processors and devices. Consequently, the analysis extends to classes of devices that are cache coherence protocol aware. Also, the concept of a transfer cache is proposed in this thesis, which reduces latency significantly even for non-coherent devices.

The trend to the exploitation of process and thread level parallelism leads to a steady increase of system sizes. Networks that are used in such large systems are very susceptible to both hard and transient faults. Most transient fault rates are constant per bit that is stored or transmitted. With increasing system sizes and higher clock frequencies, the number of faults in time increases drastically. In the end, the error rate may rise at a level where high level error recovery becomes too costly if lower layers do not perform error correction that is transparent to the layers above. The second part of this thesis describes a direct interconnection network that provides a reliable transport service even without the use of end-to-end protocols. Also, a novel hardware based solution for intermediate routing is developed in this thesis, which allows an efficient, deadlock free routing around faulty links.

Zusammenfassung

Der Bedarf an Rechenkraft von Computer-System wächst ständig. Insbesondere auf dem Massenmarkt wurde dieser in der Vergangenheit vor allem durch Einprozessorsysteme gedeckt. Die parallele Abarbeitung von Operationen ist dabei ein wesentlicher Faktor zur Geschwindigkeitssteigerung. Da die Parallelität auf Instruktionsebene in den meisten Anwendungen sehr beschränkt ist, sind weitere Leistungssteigerungen nur möglich, wenn auch die Parallelität auf Prozess- und Thread-Ebene genutzt wird. Daher bestehen heutige Prozessor-Chips meist aus mehreren Prozessor-Kernen, die einen gemeinsamen Speicher mit einem globalen Adressraum nutzen.

In solchen Systemen sind hochperformante Netzwerkschnittstellen genauso über eine Hierarchie von Verbindungsnetzwerken und Bussen mit dem System verbunden wie klassische Eingabe/Ausgabe Geräte. Um die Kommunikationsleistung zwischen Prozessor und Netzwerkschnittstelle zu verbessern, ist es erforderlich diese Verbindungsstruktur zu optimieren. Ein solcher Ansatz ist die Entwicklung von Geräten, die über das HyperTransport Protokoll direkt mit dem Prozessorchip verbunden werden können. Eine Umsetzung dieses Konzeptes wird in dieser Arbeit vorgestellt.

Darüber hinaus werden in dieser Arbeit weitere Möglichkeiten zur Verbesserung der Kommunikation untersucht. In heutigen Computersystemen ist das Cache-Kohärenz Protokoll das zentrale Protokoll, welches den Datenaustausch zwischen den Kernkomponenten des Rechners regelt. In dieser Arbeit werden Klassen von Geräten vorgestellt, die direkt als Kommunikationspartner an diesem Protokoll teilnehmen. Als bedeutende Neuerung wird außerdem das Konzept des Transfer Caches in dieser Arbeit entwickelt und vorgestellt, welches die Kommunikationslatenz zwischen Gerät und Prozessor bedeutend verbessert.

Die bessere Ausnutzung der Parallelität auf der Ebene von Prozessen und Threads führt außerdem zu ständig komplexer werdenden Systemen. In Netzwerken, die solche Systeme verbinden, muss mit dem häufigen Auftreten von statischen und transienten Fehler gerechnet werden. In einem solchen System können die Fehlerraten dabei auf ein solches Maß steigen, dass eine ausschließlich in höheren Softwareebenen erfolgende Fehlerbehandlung sehr ineffizient wird. Mit einer Fehlerbehandlung direkt in Hardware kann dieses Problem umgangen werden. In diesem Sinne beschreibt der zweite Teil dieser Arbeit ein fehlertolerantes Verbindungsnetzwerk, welches eine fehlertolerante Übertragung auf der Ebene 8b/10b kodierter serieller Links sicherstellt. Eine weitere Komponente des Protokolls ist ein neuartiger hardwarebasierter Mechanismus, der über ein "intermediate routing" eine effiziente und blockierungsfreie Lösung darstellt, um Pakete um fehlerhafte Komponenten herumzuleiten.

Contents

1 Introduction

The demand for processing power is increasing steadily. In many application fields, there can never be enough computing power. Simulations in the field of engineering, like virtual crash tests, or in the field of bioinformatics, as protein folding, are examples for applications that require enormous computing power. But even consumer PCs continue to demand for more and more computing power.

Moore's Law, predicting that the performance of microprocessors doubles about every 18 months, has proven to be true in the past, and will most likely stay true for the near future. One contributing factor to this performance increase are technological improvements. However, the direct influence of technology on computing performance is limited. Architectural improvements are another main source for sustained performance improvements.

In the past, single processor performance has been in the main focus for computer architecture. But even in this case, the exploitation of parallelism at instruction level is a key element.

As instruction level parallelism is limited in single processor applications, further performance increases can only be achieved by exploiting parallelism at the higher levels of thread or process parallelism. As a consequence, modern "processors" incorporate multiple processor cores that together form a single shared memory multiprocessor. While the architecture of the processor cores does not fundamentally differ from the architecture of single processors, architectural research must optimize communication among the processors.

In large parallel systems, which are typically message-passing multicomputers, a network interface controller connects the individual nodes to the network. Classically, the network interface controller is connected to its home node like every other input/output device over a hierarchy of peripheral interconnects. While this is an appropriate solution for slow devices like hard disks, it has become a significant bottleneck for network interface controllers (NIC) and coprocessor devices like field-programmable gate arrays (FPGA).

Thus, one target must be to couple coprocessors physically closer to main memory and to the processor of a computing node. This removes the overhead of today's peripheral interconnect structures. Such a step is the direct connection of HyperTransport (HT) devices to Opteron processors. The development of a HyperTransport intellectual property (IP) core and the integration into an FPGA coprocessor environment is part of this thesis.

Additionally, the classical assumption that a computing node consists of a single processor with memory and I/O components is outdated. Multi-core processors have turned every computing node into a small-scale shared memory system. The trend towards higher parallelism is obvious: dual core processors are standard even for consumer PCs, and all major vendors are currently introducing four or eight core processors. Research prototypes of multi-socket systems feature up to 80 cores on a single die. Today's network interface architectures do not consider this fact sufficiently.

One area that is being investigated is the virtualization of network interfaces, which provides direct access from the user space to a device for multiple processes and threads at the same time. However, little research has been performed so far to analyze new mechanisms of low-level data transport between devices and processors in these systems. Almost all data transport in a shared memory system is controlled by the cache coherence protocol, which ensures that conflicts of parallel access to the same data objects are resolved. Cache coherence protocols thus determine how efficient and fast data transport in these systems is. Traditionally, NICs and coprocessors are connected to the system over noncoherent protocols, and thus are unaware of the coherence protocol. As a result, processors cannot hide latency by caching device memory that is accessed using programmed I/O (PIO). The second way of data transport from device to processor is direct memory access (DMA). Here, the device writes data into coherent main memory, which allows processors to cache this data. However, this path includes write and read accesses to DRAM, and thus exhibits a relatively high latency.

Thus, another target of this thesis is to analyze how this communication path can be improved to exhibit lower latencies. Two types of latency are relevant: a processor's read access latency to data that has previously been produced by the device affects the throughput of the processor. The other important latency is the overall latency of data transport from device to processor, which is important if the process is waiting for the respective data.

The analysis extends to classes of devices that take part in the cache coherence protocol. Among those are devices with coherent caches, and devices that provide a coherent memory view on device memory. Besides a potential increase in performance and efficiency, coprocessors may functionally benefit from coherent caches.

The growing demand for computing power and the exploitation of thread and process level parallelism does not only increase the size and complexity of single computing nodes. Net-

works of such nodes, mainly supporting message passing, are also increased in their size. The most prominent example is the IBM BlueGene system, featuring 106,496 computing nodes connected over a 3D torus direct interconnection network.

Such large networks are very susceptible to faults. Failures that occur in hardware can be classified into hard failures, where the hardware of a system is physically broken, and transient faults. In a transient fault, the information that is stored in a system is altered, for example due to radiation or Gaussian noise on a channel. Within the last years, transient bit faults have maintained an almost constant fault rate per bit that is stored in static random access memory (SRAM) or transmitted over cable. With an ever increasing complexity and size of computers, the likelihood of transient bit fault per system is increasing steadily. To keep the availability of parallel computers at a high level, error correction and fault-tolerance are becoming a more and more important issue. In the end, the error rate may rise at a level where high level error recovery becomes too costly if lower layers do not perform error correction that is transparent to the layers above.

In any cable based network, link bit faults and complete link failures due to hard faults are the most frequent faults. In particular if direct-current-free (DC-free) high-speed serial transmission is used, coding for error correction and detection is difficult. A fault-tolerant network protocol is presented in this thesis. In contrast to state-of-the-art network protocols, errors are corrected directly by hardware on the link and network levels. On the link level, control information is protected using error correcting codes, while data is retransmitted in the case of errors. Besides the correction of erroneous bits and packets, another important topic in direct interconnection networks with a regular topology is that faulty links destroy the regularity of the topology. In this case, nodes become unreachable if deadlock-free routing mechanisms are used that have been optimized for the specific topology. A novel hardware based solution for intermediate routing is developed in this thesis, which allows an efficient, deadlock free routing around faulty links.

The result is a direct interconnection network that provides a reliable transport service even without the use of end-to-end protocols.

The outline of this thesis is as follows. Chapter 2 summarizes the state of the art in parallel computer architecture, and thus is the foundation for the subsequent chapters. Chapter 3 analyzes device to processor communication in HyperTransport based direct network NUMAs. Proposed improvements include devices that take part at the coherent HT protocol, and the completely new concept of a transfer cache. HyperTransport-based prototypes that realize the concepts are described in Chapter 4. A potential future improvement for device to processor communication are direct cache access mechanisms. An outlook on these is given in Chapter 5. With Chapter 6, the focus switches to the other side of the NIC:

the interconnection network. Transient faults that occur in such networks are analyzed, and a fault-tolerant network protocol for Extoll is described. Chapter 7 concludes this work.

1.1 The Extoll Project

The Extoll project from the University of Mannheim combines different new methodologies in SAN communication into one network. Extoll is based on the Atoll network [26][27]. Just like Atoll, Extoll combines both the network interface and a part of the network into a single chip. Although centralized switch resources are supported, Extoll is designed as a direct network. Every NIC has a crossbar and 6 bidirectional network links, thus, a 3D torus topology is recommended for Extoll.

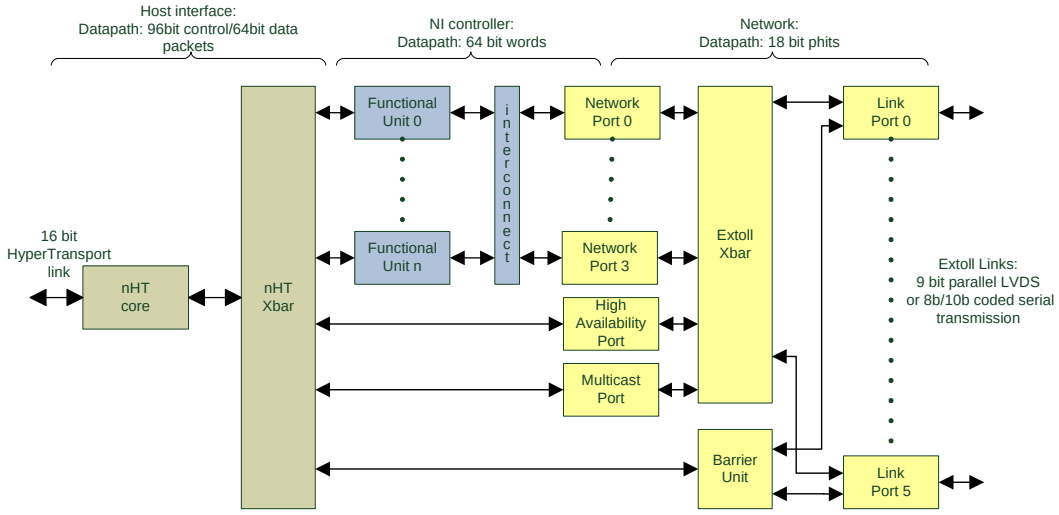


Figure 1-1. Block diagram of the Extoll NIC

The following aspects of Extoll are improvements to Atoll, and at the same time new contributions to the design of efficient SANs:

- Closer coupling of NIC, processors and memory. The design space analysis for such a closer coupling is performed in Chapter 3, the specification of the current implementation is described in Chapter 4.
- A virtualization of the NIC to allow direct user-level communication for a high number of processes or threads at the same time [121].

- Improved routing schemes in the network, including a mechanism for congestion avoidance [123].

Improved fault tolerance, including link-level error detection and retransmission of packets and link-level forward error correction of control flits. The High Availability Port (HAP) allows a rerouting of packets in the case of temporary or permanent hardware failures. The Extoll network itself, and in particular fault tolerance in the network, are described in Chapter 6.

The actual network interface controller logic is implemented in a set of functional units (FU) that execute communication instructions. One communication paradigm in Extoll is message based communication with short messages that are smaller than one cacheline. This communication mechanism is implemented in the non-virtualized ULTRA functional units, which will be described later in more detail.

The other communication paradigm in Extoll is communication in a fully virtualized device. It allows a large number of processes and threads to access a device directly using user-level-communication. Here, send-receive and put/get communication is supported. A superscalar functional unit executes the communication instructions. Multiple such units may be used in implementations to parallelize work. It is still a topic of research how an I/O memory management unit (IOMMU) and translation lookaside buffers (TLB) are integrated into Extoll to allow an efficient translation of virtual into physical addresses. A context cache keeps the most recently used contexts for the processes, which are loaded into the FUs on a user process request.

The right hand side of Figure 1-1 implements the Extoll interconnection network. The network port is the instance that translates packets into and from the network protocol format. Virtual channels and lanes are used to decrease the impact of head of line blocking and to avoid deadlocks in the system [123].

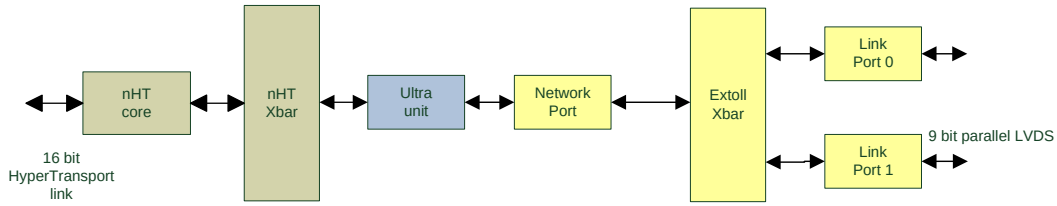


Figure 1-2. Ultra NIC

Extoll supports a direct interconnection network (IN) by integrating a crossbar-based switch for a 3D torus network. Such a direct IN provides distributed routing resources. This

means that the routing resources automatically scale with the number of nodes in the network. Between every two crossbars, a credit based flow control is used.

Current Implementation. The first offspring of the Extoll project is the design depicted in Figure 1-2. This is also the design which is used for the optimizations using a coherent protocol. With only two links, it looks more like a conventional NIC that requires centralized switching resources. The Ultra unit is the only functional unit. In ULTRA communication, a process sends a message by writing the message to the device using PIO writes. On the receive side, ULTRA writes messages into a user-space queue in main memory using a DMA write.

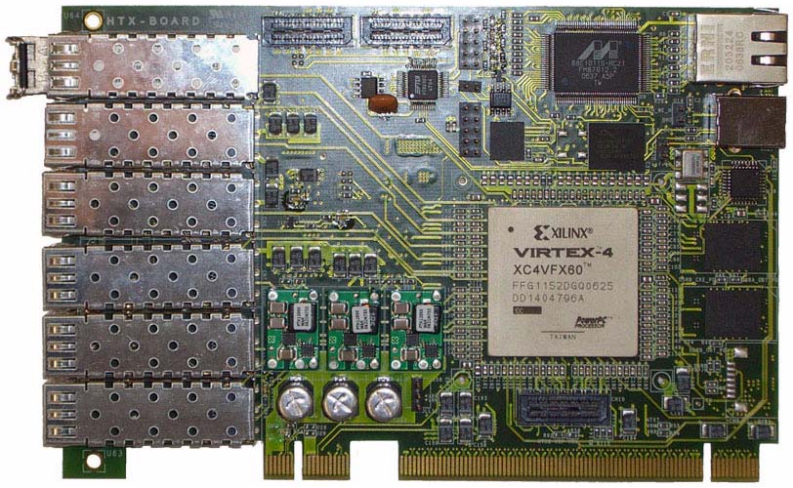


Figure 1-3. The HTX board

1.2 Physical Implementation

The hardware platform for the Extoll NIC prototype is the HTX board [128]. It contains an HyperTransport expansion (HTX) connector and a Virtex-4 FX FPGA which can be programmed via JTAG or USB. For communication, six small form factor pluggable (SFP) serial transceivers are on the board that are connected to the high-speed serial transceivers of the FPGA, featuring bit rates of up to 4 Gbit/s. Alternatively, the board can be equipped with two bidirectional parallel connectors. The HTX-Board can be plugged into any motherboard providing an HTX slot. The initial verification has been performed using the Iwill DK8-HTX motherboard, equipped with two AMD Opteron 246 processors.

1.3 Graphical Representations

Design space diagrams. An important goal of this work is to analyze and explain design spaces and design choices. A graphical representation of different computer architectures is the design space diagram (see Figure 1-4), which has been introduced by Sima [4]. The diagram shows the different aspects in the design space, as well as the design choices for every such aspect.

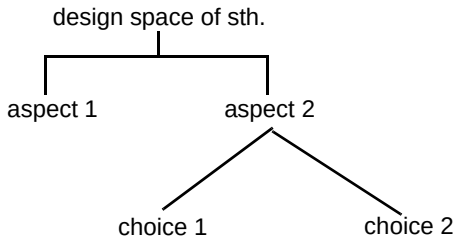


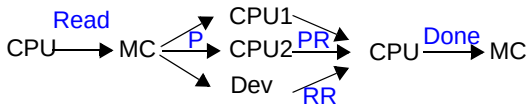
Figure 1-4. Design space diagram

Flow diagrams. In modern NUMA architectures, nodes are interconnected through a packet-based direct network. Every transaction on the system consists of a sequence of packets that is exchanged between a number of master and slave devices. Flow diagrams are being used to visualize the path of packet flow, as shown in Figure 1-5. Most diagrams refer to the packet flow in Opteron based systems that are interconnected with a coherent HyperTransport fabric. As the coherent HyperTransport (cHT) protocol is confidential, flow diagrams are based on publicly available information only [108][44].

Cache coherence state diagrams. Cache coherence protocols can be seen as state machines. When describing them, there are two alternatives: in every protocol, every memory location is in a determined state. For example, the state may be invalid, that means not being cached at all. So, one way of describing the protocols is to describe how this global state of a memory location is affected by the cache actions.

The second way of describing the protocols is from the cache viewpoint: every cacheline entry in a cache is in a determined state too. The protocols can therefore be described by showing how these states are affected by the cache actions. State diagrams, as shown in Figure 1-6 and Figure 1-7, are a good way to describe these state transitions. Separate state diagrams are required for the caches that issued a request and those caches which are snooping the request. To be able to describe a protocol this way, by convention both a present

cacheline entry with the state invalid and a non-present cacheline entry will be called invalid.



Actors:

CPU - processor/cache
MC - memory controller
Dev - device
Br - I/O bridge

Packets:

Requests:

Read
Write
C2D - change to dirty
P -probe request

Responses:

PR - probe response
RR - read response
Done

Figure 1-5. Flow diagram

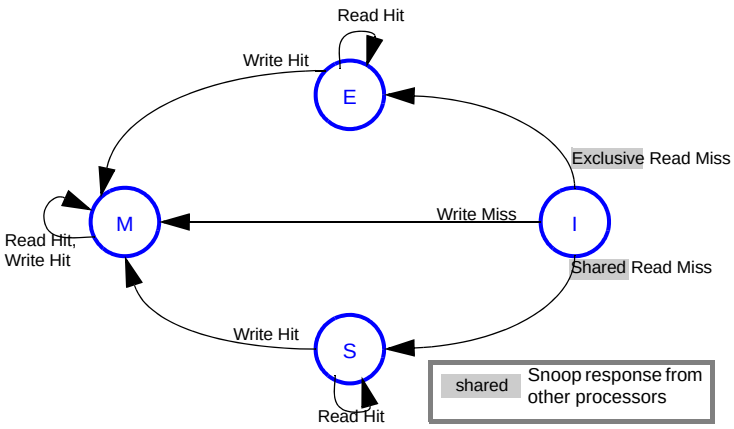


Figure 1-6. MESI state diagram for a requesting cache

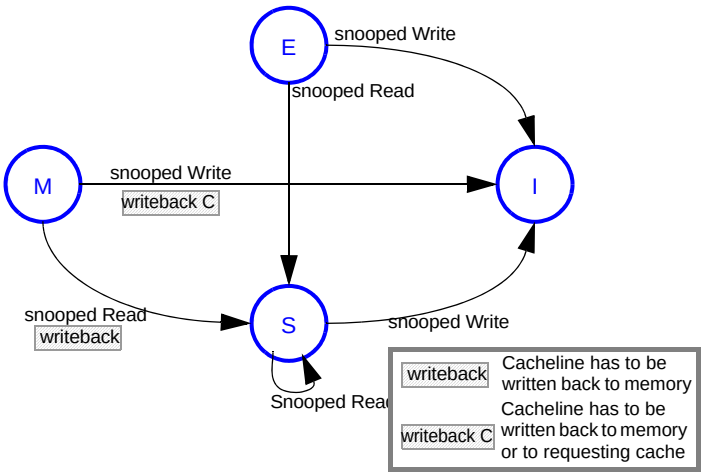


Figure 1-7. MESI state diagram for a snooping cache

1.4 Methodologies

New designs or design variants of a device in a computer system must be evaluated for their benefits and in particular for their performance. As the performance of a device usually depends on the hardware and software of the whole system, this evaluation is a complex task. The methods for the exploration of a system are shown in Figure 1-8.

The idea for a specific design can be expanded to a **theoretical model**. In this model, the performance of the system's inherent mechanisms can be estimated. Usually, only worst- or best-case estimations can be made in complex systems. A theoretical model cannot deliver good results for complex traffic patterns which influence transactions in the system in the form of background traffic. The theoretical model is based on assumptions about the behavior of system components. If these assumptions are right, a theoretical model can be efficiently used to estimate at least the order of magnitude of the performance of the choices in the design space.

An **architectural simulation** is an effort to increase the precision of performance estimates for a design. In the best case, the simulation is a cycle-accurate one-to-one image of the system, so that simulation results match results in the real system. At the same time, a simulation environment is usually being implemented faster than the real device or a prototype,

and changes to the system as part of a design space exploration are possible with less effort. Nevertheless, architectural simulation is not free of problems:

- The only way to ensure the correctness of a simulator is to verify it against the real system - which is difficult if the system does not exist yet.

The implementation of a simulation framework is very time-consuming, with limited reuse potential for the actual system implementation. This may increase the time-to-market for a product significantly. Thus, simulators are often being reused to reduce this problem.

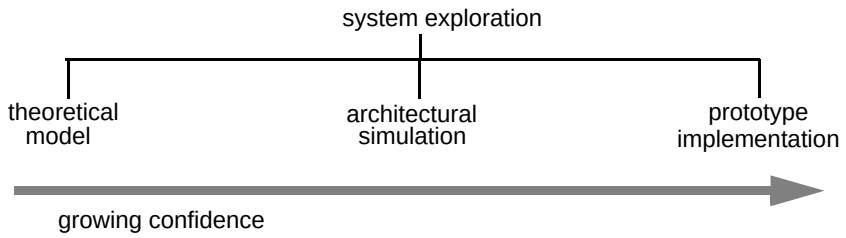


Figure 1-8. Design exploration design space

Besides these general problems, the computer architecture research community faces additional problems:

- Frequently used simulators like RSIM [16] simulate older architectures, it is thus not clear how mechanisms would behave in modern systems. As many of these simulators simulate processor instruction sets that aren't used any more, it is difficult to compile applications for use with the simulator. As a result, a small set of older benchmarks is being run on the simulators. Again, it is questionable whether this is good practice.
- Most scientific publications do not give many details about what functionality has been implemented in the simulator. Also, the source code is usually not contributed to the community. This prevents other researchers from verifying and comparing results.
- Also, publications frequently do not describe a feasible hardware implementation of proposed new features. Thus, assumptions that have been made about the hardware implementation cannot be verified. Also, it is not clear how feasible and expensive a hardware implementation would be. Due to the intrinsic differences between software and hardware design, computer architects with little knowledge and experience in hardware design are likely to make false assumptions in this field.

Although architectural simulations are a very powerful tool in general, these deficiencies reduce the significance of simulations as performed and presented in today's research com-

munity. Under these circumstances, it is not clear why most simulations that are performed are more accurate than a “order of magnitude” estimation made based on a good theoretical model. Although the experienced computer architect can avoid most of the above mentioned problems, the lack of an up-to-date simulation framework and a both critical and supportive community is a major problem.

A **prototype implementation** is the only bullet-proof exploration and verification technique. However, the development of the prototype is very expensive and time consuming. Thus, prototypes are not well suited for the exploration of a multitude of different design choices. If the prototype does not run at the same speed as the final product, for example because it is implemented in an FPGA, while the product is supposed to be an application specific integrated circuit (ASIC), performance of the final system must be extrapolated from the prototype performance.

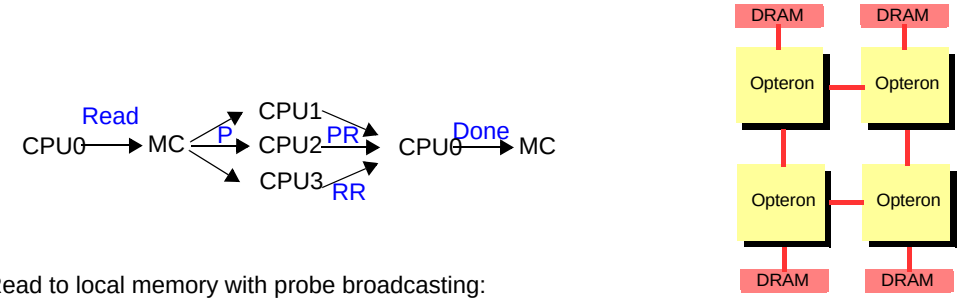
Due to the problems of simulation frameworks described above, this thesis combines the usage of a theoretical model with prototype implementations. Assumptions about the performance of subcomponents are mostly based on implementations in the prototype system, and therefore ensure that the theoretical model can be safely applied for the comparison of design space alternatives. The following subsection details these system parameters.

1.5 A Theoretical Model for cHT/HT Performance

The in-depth analysis of efficient data transport from device to processor in Chapter 3 is performed for directly interconnected NUMA systems that are interconnected using the HyperTransport protocol. The model considers the influence of all components of the HT fabric, as well as attached memory and caches. The HT components are the switches, links, coherent caches, coherent memory controllers and I/O bridges as depicted in Figure 2-28 on page 57. Processor core internal paths are not considered.

Actions that take place in these components have a certain latency, the relevant ones are depicted in Figure 1-10. Except for the memory access delay, all latencies depend on the clock frequency of the HT fabric, only the DRAM latency is fixed. The parameters in the table are given for HT1000 and are derived from the FPGA prototype implementations. Virtual-cut-through routing is being assumed in the cHT network.

Figure 1-9 shows an example calculation of a processor’s coherent read access to memory that is homed on the same node, and no other cache holds the respective cacheline. The resulting value is best-case. In a real system, background traffic and congestion will negatively influence this latency.



Read to local memory with probe broadcasting:

$$Latency_{Read} = t_{xbar} + \max(t_m + t_{xbar}, t_{probeg} + 2(2t_{link} + 3t_{xbar}) + t_{pm} + t_{probec}) = 114ns$$

Figure 1-9. Four-node example

Name	Abbrev.	Latency in ns	
Memory Access Delay	t_m	45	Read delay of memory controller including DRAM latency
Probe Hit Delay	t_{ph}	4	Probe requests hits in probed cache. Cache must deliver data
Probe Miss delay	t_{pm}	2	Probe requests hits in probed cache. Probe response must be sent
Probe generate delay	t_{probeg}	2	Time to generate a probe broadcast or a directed probe
Probe collect delay	t_{probec}	2	Time to process responses after last response has been received
Response processing delay	t_{rpr}	4	Time to process a read response containing data
Link delay	t_{link}	21	One-way latency of HT links
Xbar delay	t_{xbar}	4	Delay of HT Switch
Bridge delay	t_{br}	4	Delay of cHT/HT bridge

Figure 1-10. System parameters for HT1000

2 Communication in Parallel Computers

This chapter summarizes the state of the art in parallel computer architecture, and thus is the foundation for the subsequent chapters.

After a short overview about caches in Section 2.1, an introduction to parallel computers is given in Section 2.2, including an overview about the communication patterns in parallel systems. Section 2.3 analyses the design space of network interface (NI) locations within a node. This work concentrates on a realization of tightly-coupled NIs and devices under consideration of the cache coherence protocols in shared memory nodes. Therefore, Section 2.4 discusses cache coherence protocols in depth.

The prevalent type of computing nodes are based on the x86 architecture, which is mainly due to the good price to performance ratio of these off-the-shelf systems. Section 2.5 introduces such server systems. Section 2.6 gives an overview of parallel systems that have been implemented in order to illustrate of the most important mechanisms in parallel architectures.

2.1 Caches

The memory hierarchy in a single processor system consists of the register file at the top of the hierarchy, followed by a number of levels of caches. At the bottom of the hierarchy, there is the main memory. This hierarchy is depicted in Figure 2-1. As can be seen from the figure, speed and size of a memory component are contrary to each other: large memory components are generally slower than smaller ones. Thus, the only reason to use caches is to hide the latency and bandwidth restrictions of main memory accesses.

The terminology used in this thesis is as follows: Every location in a cache can hold a datum called *cache block*. Usually, such a cache block consists of multiple data words. Many current processors have a cache block size of 64 bytes. The term *cacheline* is frequently used as a synonym for cache block. It is not only used for the cache block located in the cache, but also when such a datum is transferred in the system.

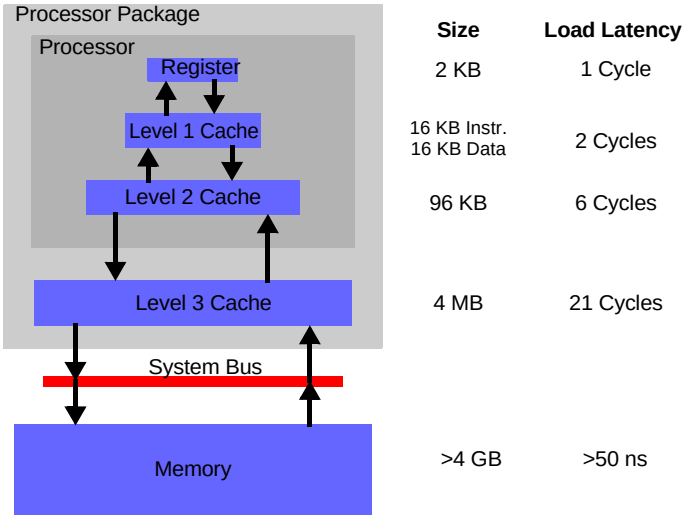


Figure 2-1. The memory hierarchy of the Intel Itanium processor [5]

The second component belonging to a cacheline is the control field. It holds information about the current state of the cacheline, for example its cache coherence protocol state (see Chapter 4.3). The union of cacheline, tag and control field is called *cache entry*.

A cacheline is indexed by a part of the address. The higher part of the address, which is not used for indexing, has to be saved in a tag field with the cacheline, so that cachelines can be uniquely identified. The group of cache blocks that can be accesses with the same index is called a *set*. If the number of cache blocks in a set is 1, the cache is called a *direct mapping cache*. If all cache blocks are in one single set, the cache is called *fully associative cache*. In all other cases, the cache is called an *n-way set-associative cache*, where n specifies the number of cache blocks per set.

2.2 Parallel Computing Architectures

The interface between every shared memory node and the rest of the system is called the network interface controller (NIC). The communication patterns for which a NIC should be optimized strongly depend of the architecture of the parallel computer. This chapter will thus briefly describe the different types of parallel computer architectures, thereafter, the communication paradigms of the most common systems are described.

Flynn's taxonomy [3] distinguishes computer architectures by looking at the parallelism in the data and instruction streams. Four classes exist: Single Instruction - Single Data (SISD), Single Instruction - Multiple Data (SIMD), Multiple Instruction - Single Data (MISD) and Multiple Instruction - Multiple Data (MIMD). Although this classification scheme is still in use, it has some major weaknesses: The class of MISD systems has never really been populated. The original proposal of the sequential von Neumann computer [1] [2] is a typical SISD system. Due to the trend towards multi-core processors and simultaneous multi threading, the class of SISD systems is emptying.

Arrays of processing elements are a typical example for SIMD architectures. Most present parallel systems are MIMD computers. An MIMD computer usually consists of a number of processing units working independently of each other, each with its own instruction and data stream. MIMD architectures are more versatile than SIMD architectures, since they are not reduced to one single stream of instructions. Therefore, MIMD architectures can generally exploit more parallelism. MIMD architectures can be differentiated into shared memory and message passing architectures. This distinction is based on the hardware mechanisms of communication. It does not specify the communication paradigm that is used by user applications, as this may be different than the mechanism that is used in hardware.

Sima's taxonomy [4] classifies parallel architectures into data parallel architectures and function-parallel architectures. Data parallel architectures are vector, associate and neural, SIMD and systolic processors. Functional-parallel architectures can be distinguished based on the level of parallelization. Instruction-level parallelism (ILP) can be exploited within every single processor by means of pipelining, superscalar designs or very-long-instruction-word (VLIW) processors. According to Sima, the process-level and thread-level parallel architectures combined form the same class as MIMD.

Shared Memory Architectures. In a shared-memory architecture, every processor is connected to every memory via the system interconnect. Although the physical memory modules may be distributed throughout the system, they form one global memory space which can be addressed by all processors.

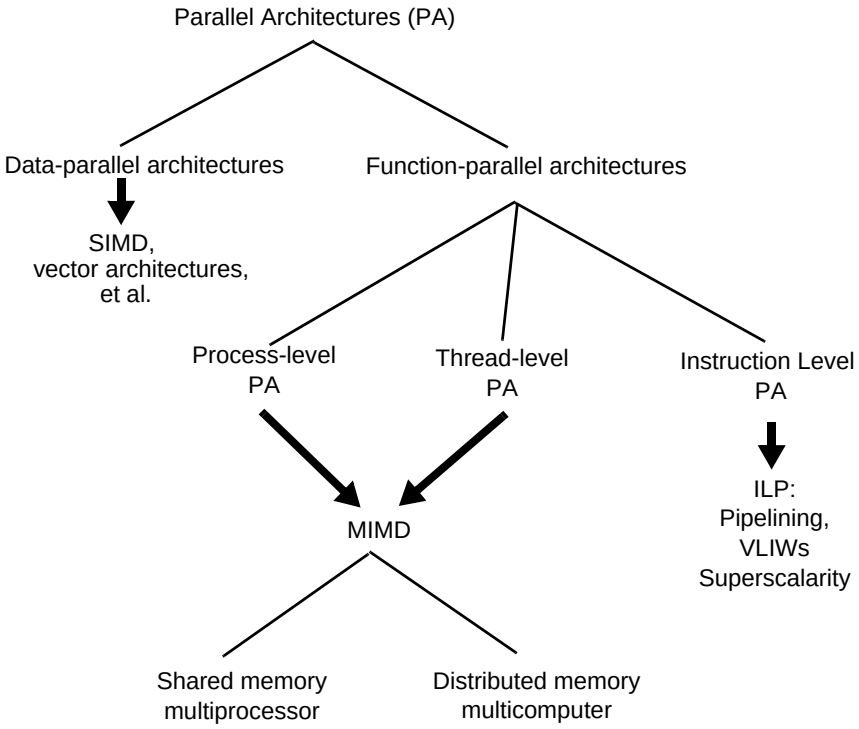


Figure 2-2. Classification of parallel architectures according to Sima

A closer evaluation of shared memory systems reveals different access models: the uniform memory access model (UMA), the non-uniform memory access model (NUMA), and the cache-only memory access model (COMA), as shown in Figure 2-3.

UMA architectures consist of n processors and m physical memories. Processors are interconnected with all memories so that processors can access all memories in the very same way. In particular, access latencies and bandwidths are the same for every processor-memory path. The scalability of this topology is limited: with a growing number of processors and memories, the complexity of the interconnect is increasing as well. Thus, larger systems will exhibit higher memory access latencies.

In NUMA architectures, physical memory is assigned to every processor, which this processor can access directly without having to use the global system interconnect. To access any other memory, the system interconnect has to be used. Thus, accesses to the local

memory will typically exhibit a lower latency than global accesses. Also, the bandwidth to the local memory may be higher.

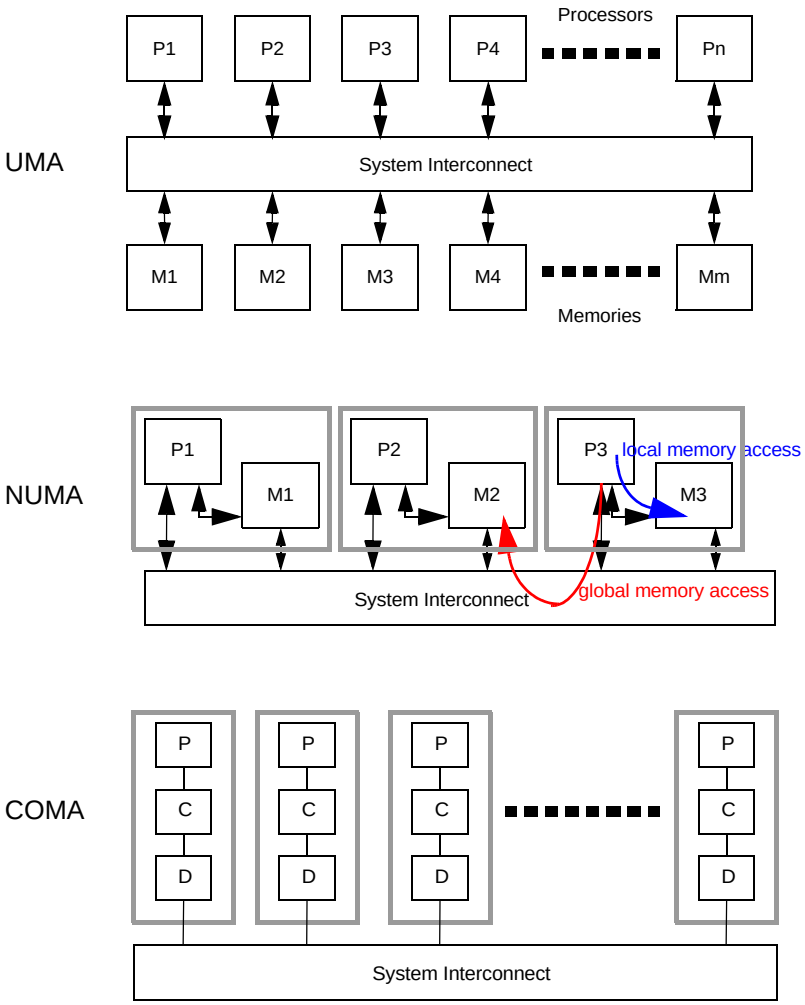


Figure 2-3. UMA, NUMA and COMA architectures

The motivation for NUMA systems is that in most parallel applications, the largest part of the accessed memory is privately used by one thread, only a part of the memory is really shared among threads. If the operating system in a NUMA machine allocates the memory of a process or thread on the same processor as the process or thread is running on, most

memory accesses of the processes or threads should target the local memory. As a result, NUMA systems are much better scalable. Firstly, a larger system interconnect affects only a part of all memory accesses. The latency and bandwidth of local memory accesses is independent of the system size. Secondly, the load on the system interconnect is much lower.

The third category of shared memory systems are COMA architectures, in which all memories are converted to caches. A memory word in a cache-only architecture does not have a permanent home address in one of the memories. Instead, it can be in any of the caches at any time. Particularly, it can be in more than one cache at a time. Processors have direct access only to the local caches. The access on memory words residing in a remote cache is performed implicitly by the cache-coherence mechanism.

Another criterion for shared memory system is the one of symmetry. In a symmetric multiprocessor (SMP), all processors do have the same capabilities including I/O access and running the operating system. In asymmetric multiprocessing, processors are designated to special purposes: Master processors are able to execute the operating system and to perform I/O. Slave processors cannot perform I/O access, but only execute code under supervision of the master processors.

Often, shared memory systems use a hierarchy of interconnects, and may use different coherence mechanisms at the different interconnect levels. The NUMA implementations ExtendiScale [108], AzuzA [98] or Dash [97] show that significantly more has to be done at the interface between nodes and the network than just routing. Optimizations include for example remote caching, address remapping and probe filtering. Thus, such a system will comprise a shared memory network interface controller (SM-NIC). The fundamental difference between NIC and SM-NIC is that a NIC has to be addressed explicitly, while communication over the SM-NIC happens implicitly based on the address of requests.

Distributed Memory Architectures. The architecture of a distributed-memory system is depicted in Figure 2-4. In this type of system, memory is not globally shared. Instead, the system consists of so called nodes, which consist at least of one processor, local memory and an interface to the interconnection network. For inter-node communication, messages are passed between the nodes. The nodes of such a system are computers acting autonomously. Therefore, these systems are also called multicomputers.

Today, most large parallel systems are distributed memory architectures. However, the nodes typically consist of small UMA or NUMA shared memory multiprocessor systems. Cluster computers that are constructed using off-the-shelf AMD or Intel processors (see Section 2.5) and SAN interconnection networks (see Section 2.6.5) are typical examples for this type of system. In these systems, any communication with other nodes requires explicit communication with the NIC.

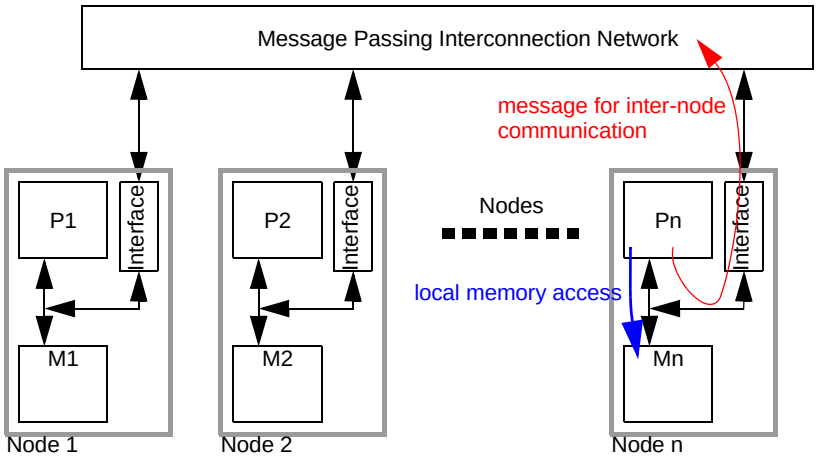


Figure 2-4. Distributed memory architecture

A NIC is required due to the fundamental differences between intra-node and inter-node interconnects. A node can be integrated in a small physical space, i.e. on board level. Communication paths on chip are in the range of micrometers, off-chip in the range of centimeters. In contrast, the interconnect between the nodes has to connect nodes that may be meters apart from each other, connected by cables instead of traces on a printed circuit board (PCB). Particular problems are:

- The high latency of the transmission over longer distances requires sophisticated flow-control mechanisms over every single cable.
- Bandwidth in the interconnect is limited, due to the high costs of adding bandwidth compared to on-chip or on-board interconnects. Thus, congestion and the need for node-to-node flow control are a serious issue.
- A significant bit error rate requires such errors to be corrected, using retransmissions or forward error correction. Also, a large network must cope with the failure of components, as cables and nodes.

Thus, the separation into a node with network interface controller, and the interconnection network between the nodes is very useful.

2.2.1 Communication Paradigms

A different point of view on this problem simply distinguishes communication primitives that may be present in such a system, which are:

- Remote memory access using remote load/store operations,
- Message passing using a send-receive semantic,
- Remote memory access (RMA) using a put/get semantic.

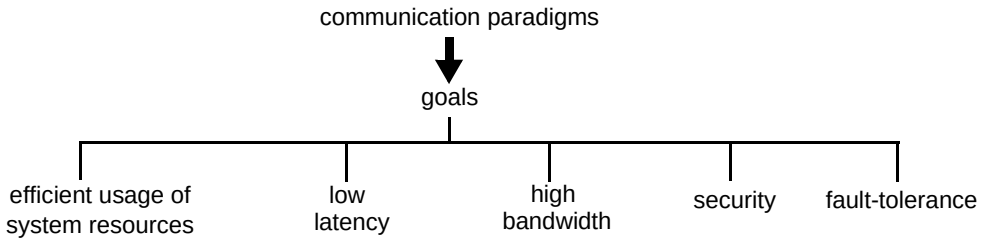


Figure 2-5. Goals of all communication paradigms

Remote loads and stores and the RMA mechanism are both one-sided communication mechanisms. This means that only one process is actively participating in the communication. Thus, this kind of requests always involves access to remote memory: communicating processes are coupled by using shared memory. In contrast, message passing using a send-receive semantic is a two-sided communication mechanism, as both communication processes are involved. A system may support both message passing and shared memory mechanisms at the same time.

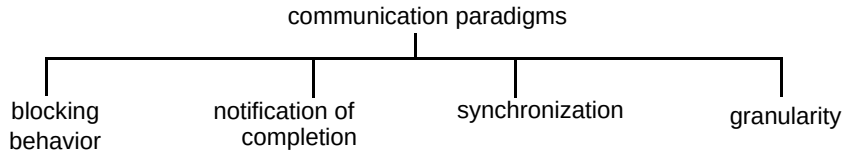


Figure 2-6. Aspects of communication paradigms

Figure 2-5 shows the general goals of all communication paradigms. It depends on the application rather than on the specific paradigm how these goals are weighted. The para-

digms differ in the aspects shown in Figure 2-6. The next three sections will analyze the paradigms with regard to these aspects.

2.2.2 Remote Load/Store

The remote load/store mechanism is the type of communication that is used in shared memory systems. It is based on load and store instructions of the processor instruction set. Typically, there is no differentiation between remote and local instructions, so that they are treated the same way as every other load instruction.

A processor's load instruction classically loads one value from memory into one register, a store instruction stores the content of one register to memory. Thus, load and store instructions work on one single processor-native data object that typically has a size of 32 or 64 bits. Vector instructions may work on larger data types. However, current implementations in processors as the different types of SSE [23] or AltiVec [24] support only up to 128bit loads and stores. If such an instruction misses in a cache, the cache will create a read or write request with the size of a cacheline in this cache to the next level in the memory hierarchy.

A remote read/write may thus use the cacheline size of the last cache hierarchy, or a multiple of it. The largest unit of data transport is used for virtual shared memory (VSM) systems. These are message-passing based distributed memory systems that emulate remote loads and stores in software. Due to the high overhead for remote memory accesses in these systems, the granularity of remote accesses is typically one page [25].

Remote loads and stores are blocking operations. A load that has been issued blocks until the response data arrives. In particular, a thread or process cannot be retired. However, the processor may execute other instructions if they do not have data dependencies with the stalled load, and if this does not violate memory ordering constraints.

Similarly, a store may block other memory requests. Most memory consistency models employ strict ordering among stores.

Shared memory programming models as OpenMP [75] may allow a differentiation of remote and local memory. Thus a compiler can optimize parallel code by inserting early prefetch instructions for remote memory loads. A manual optimization by the programmer would also build up on prefetch instructions. A prefetch instruction is basically a hint to load a specified memory block into the cache in a non-blocking way. However, finding the right point in time for a prefetch is difficult. If a prefetch is started too early, a succeeding write to the address by another processor will lead to an invalidation of the cache entry that has been allocated by the prefetch, which results in the same situation is if no prefetch had been executed. In the worst case, a prefetch is executed at the same time while another pro-

cess is writing to the same memory location. In this case, the write is delayed, as the prefetch of the cacheline cancels the writer's write permission to the cacheline temporarily.

The notification of completion happens implicitly, due to the blocking behavior of loads and stores.

In a shared memory system, the cache coherence protocol makes changes to shared data visible to the whole system immediately. Thus, a certain grade of synchronization is already performed by the underlying hardware. However, many applications require mutual exclusions to access critical sections of the parallel application.

A non-coherent implementation of remote loads/and stores is possible as well. In this case, changes become only visible if they are written back using a remote store, and cached copies are invalidated. An application of this scheme are systems with a very relaxed ordering scheme, as for example transactional memories [8].

2.2.3 Put/Get

Communication using put and get operations is often called remote memory access (RMA) communication. In analogy to the remote load operation, a get request is used to access remote memory. However, there is a number of fundamental differences to the load operation:

The most fundamental difference is that put and get operations do not operate on processor registers, but on main memory or a dedicated set of registers that is not part of the processor register set. For example, implementations of MPI [73] and the Extoll put/get unit use main memory. The T3E (see Section 2.6.2) uses a set of memory-mapped device registers.

Put and get can be implemented as non-blocking operations. The request for a get starts the operation that is performed asynchronously to the process. Before the process can access data that has been transferred using a get, it has to check for completion of the operation. The third and last difference is that the put/get semantic does not support an automatic hardware coherence. Instead, this must explicitly be managed by the application.

The notification of completion may occur in three different ways, depending on the implementation. One possibility is a blocking put/get operation, which only returns if the operation completed. A second possibility is a nonblocking get, where read accesses to the local get destination are blocked until data is available. Such a mechanism is implemented in the T3E (see Section 2.6.2). The third and most popular solution is a test for completion that is performed by the user process. An MPI implementation will typically be based on notifications of completion, which are inserted by the device into a notification queue that resides in main memory. The process then has to check this queue to retrieve the status of the operation.

For a get operation, a notification is generated if all data has been written to the requestor's destination. For a put operation, there are two points in time at which a notification may be generated. A first notification may be generated if all user memory that relates to a specific put operation can safely be reused by the requester. The second notification is generated if the put operation completed at the target, so that it is globally visible.

The synchronization in the RMA scheme can be done using mutual exclusions. MPI, for example, uses the notion of memory windows. A window is a part of the address space that is accessible via puts and gets by remote processes. These windows can be locked explicitly for mutually exclusive access. Another method of synchronization is the use of epochs. An epoch is framed by a barrier at the beginning of the epoch and a barrier when the epoch ends. A process enters a barrier operation only if all puts and gets it has issued in the previous epoch, i.e. since the last barrier, have finished. So, puts and gets from different epochs cannot collide.

2.2.4 Send-Receive

In the send-receive scheme, a processes explicitly send messages to other processes. The target process can obtain the message data by an explicit receive operation. Thus, send-receive is a two-sided communication mechanism. Besides the two sided scheme, message passing communication may also support n-sided communication using broadcast, multicast and other collective operations. The send-receive paradigm offers a wide variety of operations that differ in their behavior. User-level libraries as MPI offer the whole variety of function calls to the user.

The standard send and receive functions are blocking. The send function returns if the message has been successfully send from the sending processes perspective. This means, that the user can safely reuse buffer space that contains the message data. It does not mean that the message has arrived on the remote node. A receive function returns after the message has been received. Nonblocking send and receive functions may return immediately, the user application has to explicitly check if the operation succeeded before accessing the respective data. Another type of send is the synchronous send. This function only returns if the receive operation for that message has been called.

The underlying hardware does not have to directly implement all these sending mechanisms, instead they can be emulated. A nonblocking send function can be implemented in the message passing library using a blocking send hardware mechanism: first, all message related user data is copied into a buffer. Then, a separate thread performs the actual blocking send operation. Synchronous sends are usually implemented using a rendezvous protocol. The sender first sends a request to the receiver, when the receiver calls the

corresponding receive operation, it sends back an acknowledgement. Only after this acknowledgement has been received, the sender may start sending the actual message.

In order to uniquely identify messages, they carry a user-application tag and information to distinguish processes.

Efficient implementations of the send-receive make use of nonblocking send-receives, as this allows the overlapping of communication and synchronization. Also, blocking sends and receives are prone to race conditions that may lead to deadlocks. Thus, in the general case, the NIC has to generate a notification that is checked explicitly by the user application. As the synchronization is explicitly performed by the send and receive operations, no other mechanisms are used to synchronize bidirectional communication. For global synchronization, barriers may be used.

2.3 Device Integration Design Space

This chapter analyzes at which locations in a node a NIC or SM-NIC may be located. The overview presented in Figure 2-7 makes use of the same scheme that originally has been introduced by Bruening [28].

The closest coupling can be reached if the NIC is integrated into the processor core. Examples for such implementations are the iWarp [11] and Transputer [12] computers, as well as a suggestion by Henry & Joerg [83]. Communication in these systems is performed by writing to or reading from a special set of registers. In these implementations, the register set is connected to the network over a FIFO queue. A general problem with these approaches occurs if packets arrive that are destined to a process other than the one that is currently loaded in the processor. Gang scheduling together with draining the network on every process switch is used in most such closely coupled systems to resolve this problem. Henry & Joerg suggest to interrupt the currently active process and to schedule the process that is the destination for the packet. Also close to the processor core are NICs that are attached to the cache interconnect, as the *T [94] for example.

While such a close coupling of computation and communication may be desirable, it is very difficult to propose realistic implementations. The cache interconnect and the processor core of commercial processors are usually neither physically or legally accessible. They are within the processor chip, confidential, proprietary, not compatible with other processors and also may change frequently.

The coherent processor interconnect is the typical network interface location for shared memory systems that have a global memory address space. Supercomputers in the early and mid nineties were frequently of that type [99]. One major vendor of this type of systems has

been Sun Microsystems, an overview of their system architectures is given in [15]. Another prime example of shared memory systems is the Dash architecture [97], a more recent example is the Horus chip of the Extendiscale architecture [108]. *T-Voyager acts like a snooping processor at the processor interconnect to allow coherent shared memory communication between nodes.

This processor interconnect is also the closest location to the processor that is physically accessible, as it is designed to support inter-chip communication. It is also less confidential. Some message passing NICs, as in the PowerManna [76] or in the T3E were attached to this interconnect as well. They implement the functionality of I/O devices.

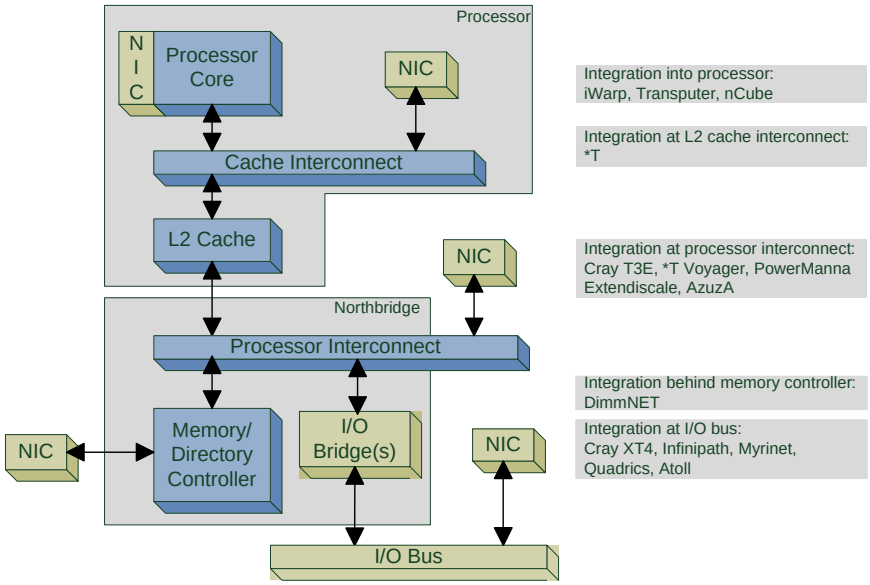


Figure 2-7. Device integration design space

The I/O bus is the standard interface for devices, including NICs and accelerators. I/O buses like PCI Express are standardized, and available on different platforms. Thus, there are numerous examples for such systems, as the Cray XT3 and XT4. Also, I/O buses allow the development of NICs that may be used in all host systems that support the I/O bus. Examples for such NICs are Atoll, Infinipath, Myrinet and Quadrics.

The MEMOnet NIC architecture [77] is one of the few examples for a memory controller-attached NIC. This architecture shall leverage the fact that a processor can access memory with higher bandwidth and lower latency than a device connected over a PCI bus. The

MEMOnet NIC is implemented on a DIMM. Besides the NIC itself, SDRAM memory is also included on the DIMM.

The sending of a message to the NIC makes use of the PIO mechanism: A send memory region maps to the DIMM module. The processor writes message header and data to this region. Thus, as for any PIO operation, the memory region must be uncacheable so that writes are directly written to the memory controller and thus to the memory module.

A serious problem of this architecture is the passive nature of memory modules. The NIC cannot issue interrupts. The receive process can only work as follows: Messages that have been received are written to the SDRAM that is embedded on the memory module. A process can then access the received data just as it would for a classical memory-mapped PIO device.

Analysis in this work. Due to the low feasibility of processor integrated NICs, this work focuses on NIC implementations on the I/O bus and the processor interconnect. Also, it focuses on the integration into a node that is a small-scale shared memory system. This is due to the fact that such systems just recently started to prevail on the market, as technological and architectural solutions to increase the performance of single processor cores have been exploited. Section 2.4 will describe shared memory systems in detail, as this is the basis to analyze integrations of devices in such systems.

2.3.1 Process-Device Interaction

Most of today's architectures know two different basic types of memory: main (physical) memory and I/O memory.

Traditionally, I/O space can be accessed by device drivers and the operating system, leveraging the PCI software model. This model is used by virtually all of today's devices, as in PCI-X, PCI-Express (PCIe), HT2 and HT3 devices. In the user space communication scheme, I/O space is mapped into the virtual memory space of applications. This avoids operating systems calls and memory copy operations between user and operating system memory spaces.

Process-device interaction can be differentiated into synchronization and data communication. There exists a very typical basic scheme of communication between process and device. To submit a job to a device, the process inserts a job descriptor into a dedicated queue. This queue is necessary to decouple the operation of processor and device. The job descriptor may contain all data that is required for the device to process the job. It may also be the case that the job descriptor contains direct or implicit pointers to additional data required for the job. In any case, this is a classical producer-consumer situation. Synchronization in this process is necessary to determine the fill level of the queue.

Communication in the other direction, i.e. from device to process, works the very same way. This path may be used for example to submit the result of a job that has been previously issued to the device. In a network interface device, data that has been received from the network is forwarded this way.

For data communication in today's x86 systems, there exist two mechanisms, which both include the required synchronization:

Programmed Input/Output (PIO). In this communication pattern, data is moved between applications and devices by explicitly performing processor memory operations. Before moving data to a device, the processor has to check whether the device is capable to hold the data. If there is enough free space, data is written by I/O writes into the address space of the device. Address bits of the device address space may be used to submit additional control information to the device. For example, one bit of the address might be used to mark the end of a data packet. Data transfer in the other direction is performed in a similar fashion. First, the process checks for new data by performing I/O reads to a status register. If this is successful, the processor can issue subsequent I/O reads to read the data from the device. This interface is called register-mapped interface.

Direct Memory Access (DMA). Data is not copied by the processor, but by a DMA engine that is external to the processor. Usually, the DMA engine is part of the device. For processor to device communication, a job descriptor that specifies source and target of the copy operation must be set up by the processor. Then the processor triggers the DMA engine, which is typically done using an I/O memory access to a register of the DMA engine. A typical location for the job descriptors is a queue in main memory. For data transport in the other direction, the device performs the DMA and creates a notification that is also in main memory. The processor uses PIO accesses to check for new entries in the queue. This interface is a descriptor-based interface.

It can be noted that both communication methods currently rely on PIO concerning the synchronization of data transport. In essence, there are three basic types of data movement used:

- Synchronization using I/O read and write accesses from processor to device
- PIO data transport between processor and device
- DMA data transport between device and memory

A fourth possibility is not used in the classical approach, but will be introduced in the following sections:

- Synchronization over memory

Queue Organization. In order to decouple the producer and the consumer of data, buffers are required. The hardware representation is a first-in-first-out (FIFO) buffer. This structure cannot be implemented efficiently in software, where data always resides in main memory. Therefore, pointer-based wrap-around queues as shown in Figure 2-8 are used.

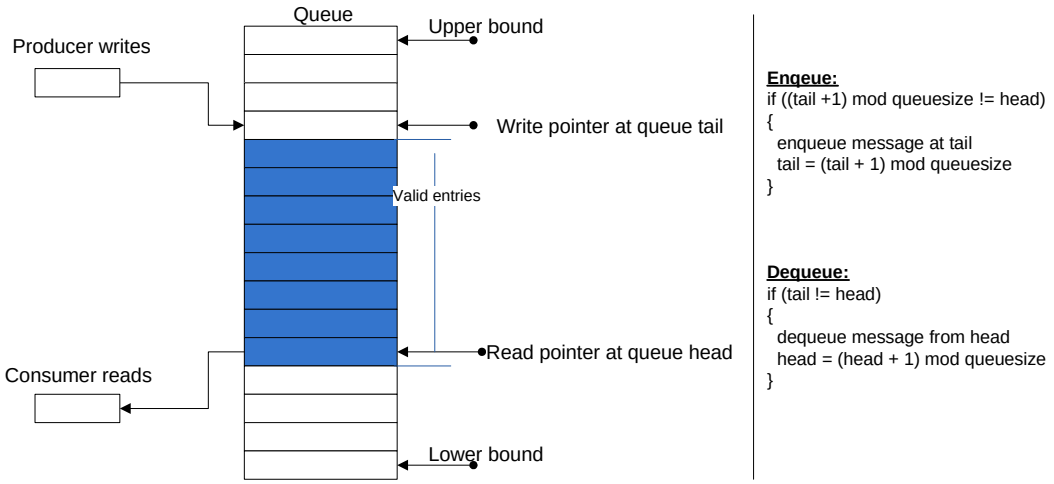


Figure 2-8. Pointer-based wrap-around queue

The basic pointer scheme has the disadvantage that pointers have to be exchanged between consumer and producer for every access to the queue. Queue pointers are usually communicated using memory operations to memory-mapped I/O space of the device.

Improvements to the scheme are depicted in Figure 2-9. Lazy pointers improve the enqueue operation: for an enqueue, it is sufficient to know that enough free space is in the queue. It is not necessary to know the amount of free space precisely. Thus the producer has a copy of the read pointer that is potentially stale. This copy has to be updated only if the free space in the queue is under a certain threshold. For the write pointer, the lazy pointer does not work. Not updating the write pointer means that the consumer is not notified of new entries in the queue, which results in a higher latency.

Another improvement are valid bits that are embedded in every queue entry. The software-based approach, which is implemented in the *T-ng [95] for example, embeds valid bits in the data object that represents a queue entry, i.e. the valid bits are part of the queue payload. Thus, the processor can read a queue entry speculatively, and determine whether the entry is valid or not. In one scheme, a bit with the value ‘0’ means invalid, ‘1’ valid. The producer sets this bit to ‘1’ in every entry it enqueues, the consumer has to reset this bit to ‘0’ after it

has consumed the entry. In this scheme, the consumer does not need to observe the write pointer at all. The drawback of this approach is that the consumer has to write to the queue entry, causing additional memory traffic. Sense reverse avoids this problem by changing the meaning of the sense reverse bit for every pass through the queue, so that the bits in the entries do not have to be changed.

A general drawback of software-based valid bits is that they consume entry payload. This may not be a major problem if the queue entries are control information, but it may be a severe limitation if queue entries contain payload data.

Hardware-interpreted valid bits have been implemented for example in the Cray T3E (see Section 2.6.2 on page 60). The advantage of hardware-interpreted bits is that they allow advanced notification mechanisms, as explained in the next paragraphs. Also, they do not reduce the payload that can be used by software.

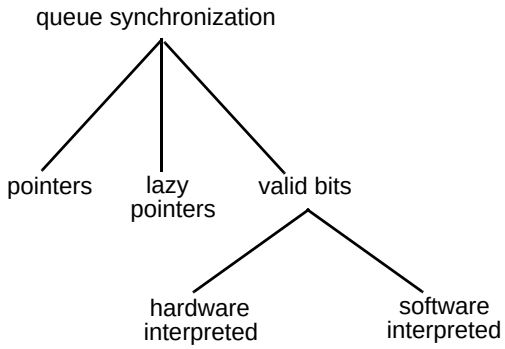


Figure 2-9. Queue synchronization design space

Process Notification (see Figure 2-10). A process has to be notified that a queue entry has been added by a device, and thus should check queue pointers or valid bits. Most devices use interrupts for signaling to the processor. However, today's interrupt mechanisms only signal that something happened, they do not carry any information about what happened in a device. Interrupts are signaled to a processor, which usually calls the interrupt handler of the operating system or the device driver. Thus, interrupt signaling involves a much higher latency than polling mechanisms and is less useful for tightly coupled coprocessors.

In the polling scheme, a process is continuously reading on a synchronizing data structure, as a queue pointer for example, to detect a change. The downside of this method is that the process occupies processor resources, and may cause traffic on the path between processor and device. Another option is to suspend the execution of the thread until a change

occurred. In the Cray T3E, a load to a memory location whose hardware valid bit is ‘0’ stalls, thus stalling the process. However, the processor is blocked by the process. SMT processors as the Sun T2 (see Section 2.6.1) or HEP [32] avoid this blocking, as the processor resources are still available for the other threads running on the processor.

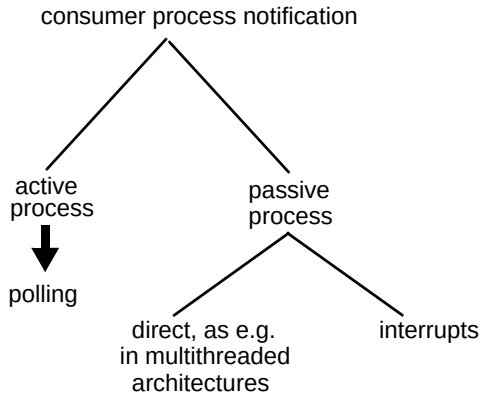


Figure 2-10. Consumer process notification design space

The ATOLL NIC [26] [29] introduced a mechanism to mirror device registers into main memory, so that a process can access those instead of the real device registers. Coherence is maintained by updating the main memory image, called replicator page, periodically. The main advantage of this method is that a polling process will only access the cache instead of continuously generating I/O traffic. A very crucial issue is the update frequency. A low frequency increases the latency until a process gets hold of a value change of a device register. A high frequency in the best case has the same latency as a PIO access, but will generate unnecessary bus traffic.

2.3.2 Device Virtualization

Virtualized devices allow user-space access to the device for multiple processes at the same time. Three major challenges have to be solved:

- How is the context of a process, i.e. status registers and queues handled, and where is the home of these data structures?
- How can requests from processes be controlled so that they do not overflow the device?
- How can the device reliably protect processes from each other?

If a virtualized device supports a large number of contexts, the corresponding data structures cannot efficiently be held on the device. A solution to this is to make main memory to the home of all context information. The device caches these contexts.

At the same time, it must be ensured that device buffers do not overflow with requests from different processes. To allow processes to determine whether they can submit a request to the device, an submission of such a request must be an atomic operation as read-modify-write. Many systems do not support these operations on the I/O bus. A conditional store buffer [124][120] is an alternative to implement an atomic access to buffers on a virtualized device. Information about the request is encoded in the address bits of a read request to device I/O space. The device answers with a read response, which contains information whether the request insertion into the buffer was successful or not.

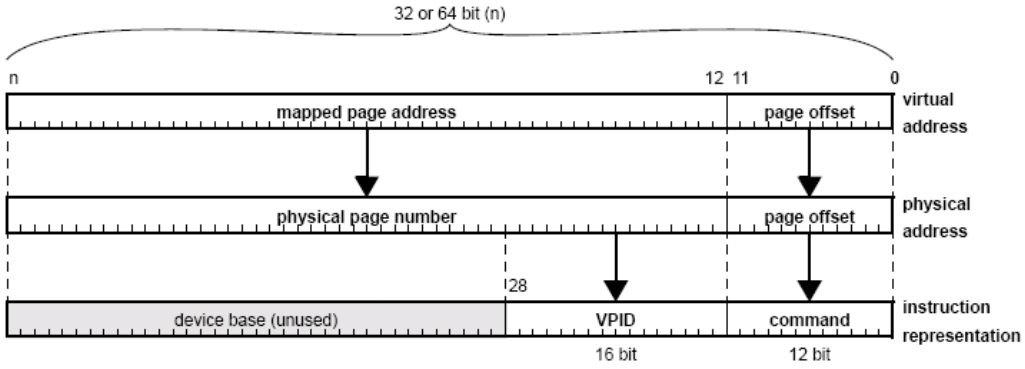


Figure 2-11. Address decoding for a read request to a conditional store buffer in Extoll [121]

The requesting process can be identified if a process or context ID is encoded in the read request as well. Processes are protected from each other if only that memory-mapped I/O space is mapped into process memory that corresponds to the assigned process ID. An example for such an encoding is the one used in Extoll, as shown in Figure 2-11. It uses 16 bit of the address to decode the virtual process identifier (VPID), and additional 12 bit to decode the command which should be enqueued.

A more detailed description of these problems and possible solutions in the context of Extoll is given in [121].

2.4 Cache Coherence for Shared Memory Systems

The most important aspect of shared memory systems is the single address space view. A differentiation of the memory access models UMA, NUMA and COMA has already been performed in Section 2.2.1. Other important aspects, shown in Figure 2-12, will be described in the following sections.

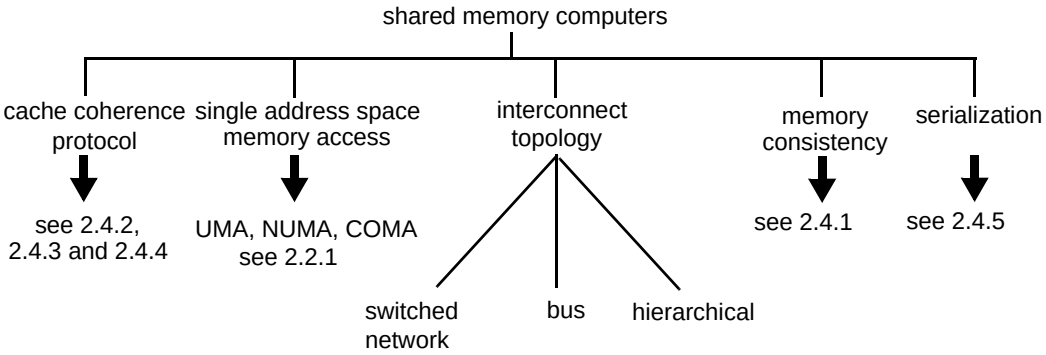


Figure 2-12. Design space of shared memory computers

A key point is the memory consistency model of a system. The memory consistency model defines how operations on memory that are performed by one processor in a shared memory system affect the memory as seen by the other processors in the system. The consistency model requires some form of serialization of simultaneous memory requests to the same address.

Processors in shared memory systems usually have private caches. If a processor modifies a cached line, a mechanism is required that keeps these caches coherent. These aspects will be analyzed in detail in the next sections.

Another important criterion is the topology of the interconnect between processor caches and memory. In principle, any type of interconnection network may be used. Bus-based interconnection networks have been very common in the past. Very small scale system may use a single bus as interconnect, while larger systems may use multiple parallel or hierarchical buses, as for example in the NEC AzuzA [98] and Sun's XDBus architecture [14]. Broadcast-based cache coherence protocols can be implemented on buses easily. However, both broadcast protocols as well as busses generally do not scale well. In contrast, the scalability of switched interconnection networks is much better. Today, there is a clear trend towards direct networks of NUMA nodes, as depicted in Figure 2-13.

Hierarchical combinations of different interconnection networks are possible as well. For example, the Dash [97] shared memory computer connects clusters of 4 processors with a bus, while the clusters are interconnected using a 2D torus direct network.

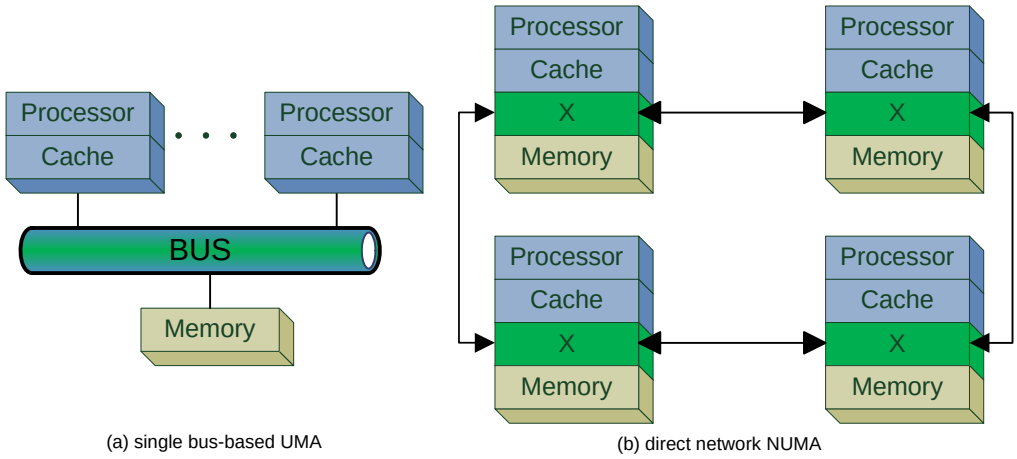


Figure 2-13. Common topologies for small scale shared memory computers

2.4.1 Consistency Models for Shared Memory

In order to develop parallel applications for shared memory systems, programmers need a solid idea of how memory behaves with respect to reads and writes to memory addresses. The *memory consistency model* of a system is the set of rules that a system employs on memory accesses.

Strict consistency is the most stringent model of memory consistency. It requires that every read or write must be globally visible before any other read or write can be issued. Thus, it imposes a static ordering of memory accesses of all processors. Such a system would not require any other mechanism for synchronization, as it is synchronous by compile time. However, is not applicable to multiprocessor systems at all, as they gain from issuing and executing operations independently of each other.

A realistic model of consistency in multiprocessor systems is the model of sequential consistency. The original definition from Lamport [55] is: “A system is sequentially consistent if the result of any execution is the same if any operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.” A more handy definition has been derived by [56]:

"A system is sequentially coherent if the following conditions are true:

(A) All processors issue memory accesses in program order.

(B) If a processor issues a STORE, then the processor may not issue another memory access until the value written has become accessible by all other processors.

(C) If a processor issues a LOAD, then the processor may not issue another memory access until the value which is to be read has both been bound to the LOAD operation and become accessible to all other processors."

The classical consistency model is sequential consistency. However, modern processors may issue instructions out of order and speculatively, which suggests the same for read operations. Section 3.1.2 shows that the standard memory type of AMD Opteron processors does not perform any ordering among reads. Reads may also pass writes if they do not go to the same address. Systems with consistency models that are weaker than the sequential consistency model are called relaxed consistency models. A summary of possible relaxations and their implementations is given in [57]. A commonality between all those optimizations is that they do not reorder reads and writes that go to the same destination.

In the weakest consistency models, no ordering constraints at all are applied to memory operations per se. Ordering and thus consistency can only be maintained using synchronization primitives as fences. One of those models is weak consistency [58], an improved definition is given by Gharachorloo [59]:

"(A) before an ordinary load or store access is allowed to perform with respect to any other processor, all previous synchronization accesses must be performed, and

(B) before a synchronization access is allowed to perform with respect to any other processor, all previous ordinary load and store accesses must be performed, and

(C) synchronization accesses are sequentially consistent with respect to one another."

Obviously, weak consistency can emulate sequential consistency by performing an explicit synchronization after every memory access. The idea of weak consistency is nevertheless to perform synchronization on a coarse grain level. Thus, it is well suited for systems were a fine grain synchronization is to expensive, as in SANs or LANs. For example, MPI[73][74] uses a weak consistency model for its RDMA protocol.

Transactional memory coherence and consistency [117][118] is an approach to raise the level on which consistency is performed from instructions to transactions. A transaction is a block of instructions that is executed and completes as an atomic unit. Among transactions, sequential consistency is maintained. Only if a transaction completes, the changes it made to memory are made globally visible by broadcasting these changes to all other processors. If another processor detects a change of a memory location that is used by that pro-

cessor, it must roll back and reprocess the current transaction. A transaction may only complete if all previous transactions completed. Although the idea of hardware-based transactional memory systems is relatively old, intense research only started within the last years.

2.4.2 Cache Coherence Protocols

In a shared memory system, multiple caches may hold copies of the same memory location at the same time. A basic requirement of all consistency models is that caches do not hold different values for the same memory location at the same time. The point in time at which this is guaranteed for a memory operation is called the commit point of the operation. The mechanism that ensures that is called the *cache coherence protocol*.

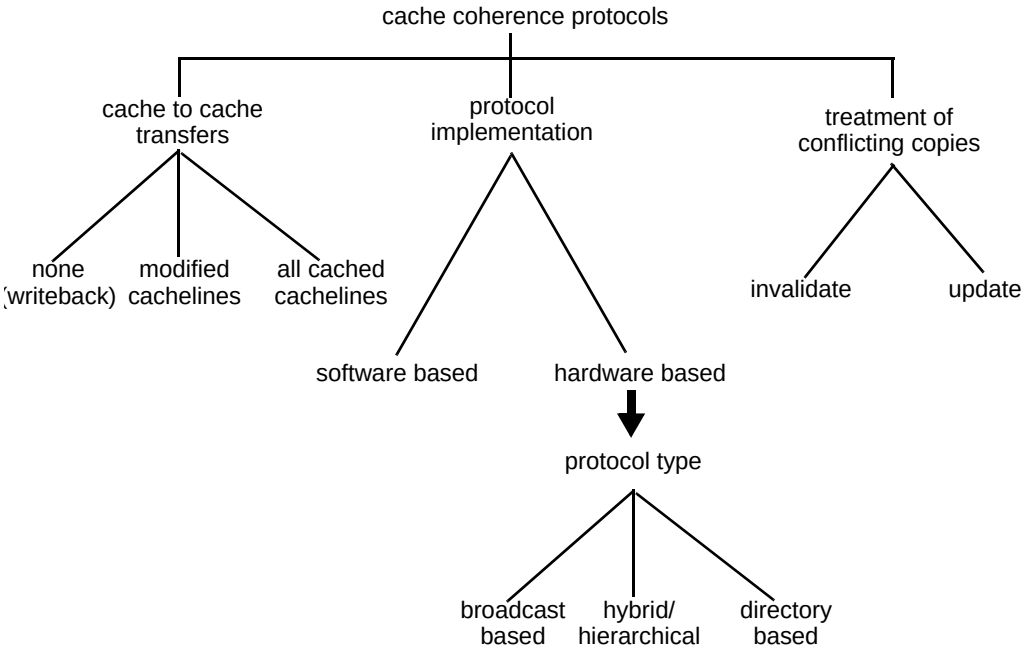


Figure 2-14. Design space of cache coherence protocols

Invalidate vs. Update protocols. Inconsistencies between caches may occur only when at least one processor writes to a memory location, either into its own cache or back to main memory. If the same location is cached in other caches, a cache coherence protocol must ensure that these caches do not keep the old value of the memory location. There are two

possible ways to ensure this: One solution is that all other caches have to invalidate their copies before a processor is allowed to write to a memory location. The mechanism how this is done depends on the cache coherence protocol. One possibility is to explicitly send invalidation messages to all other caches that may have the memory location cached.

Instead of invalidating other cache entries, these entries might as well be updated with the new data. The advantage of this method is that a subsequent read from the remote processors will result in cache hit instead of a miss. In a bus-based system, this is can be implemented easily, as all caches snoop the bus and can copy the updated value into their caches. However, the scalability of other topologies would be smashed if these updates would have to be broadcast to every node. Update protocols have a second problem: a processor does not write cachelines, but words of a much finer granularity, as for example 32bit. Theoretically, every such small grain writes would have to trigger an update of the cacheline. A write-buffer mechanism could be used to perform updates at a cacheline granularity.

For these reasons, basically all cache coherence protocols are invalidation-based protocols. Mukherjee [112] started with an analysis of how prediction mechanisms may be used to control update mechanisms. This research is being continued by other groups [114] [115] [116]. As prediction is speculative, all such protocols are based on invalidation-based protocols.

CC protocols and memory coherence. A processor that maintains sequential ordering among writes can only execute a write when the previous write has committed, i.e. it has been observed globally. A necessary condition, and usually also a sufficient one, is that all caches have seen the write. Thus, the cache coherence protocol must acknowledge that a memory operation has been seen by all caches to the requesting cache. Only then a cache is capable of putting the respective memory coherence model into effect.

Another requirement of memory consistency models is that a write transaction on the coherent interconnect appears to be an atomic operation. To be more precise, it is sufficient for the sequential consistency model and all derived models of consistency that operations to the same location appear to be atomic. Writes to different memory locations from different processors do not have to appear atomic among each other.

If operations on the same memory location were not atomic, two writes from different processors to the same memory location at the same time could cause an inconsistent state of the respective cachelines. In an update-based cache coherence scheme, this could result in different caches holding different values for the same memory location at the same time. In an invalidation-based protocol, it could result in the invalidation of both writes, thus reestablishing the previous value of the memory location, which is not legal either. As a write operation may cause communication among multiple caches, the memory controller and a

directory, it is by no means a true atomic access. Chapter 2.4.5 will show how this problem may be resolved.

Cache to cache transfers. If a processor requests a memory location that is cached as modified in another cache, this modified value has to be transported to the requester. One possible solution is a write-back of the dirty line back to memory, so that the memory can satisfy the read request. A faster solution is the direct cache-to-cache transfer of the modified line that is started by the probe. Depending on the protocol, an additional write-back may also be triggered.

Another possible improvement is a direct cache to cache transfer of unmodified cachelines. The reason to do so is that a remote cache can access data much faster than the memory controller that has to access slow DRAM memory. On the other hand, there is no guarantee at all that the requested memory location can be delivered by a cache.

To achieve the lowest memory access latency in all cases, cache coherence protocols that support cache to cache transfers may request the memory location from the physical main memory at the same time when sending out the probes. As it is not known if the DRAM access is required, and if it contains the recent value at all, this memory access can be seen as a *speculative memory access*. In case a cache could deliver the data, a special memory cancel packet may be send to the memory controller to stop the memory access. In contrast, a *non-speculative memory access* is given if the memory access occurs after it has been found out that remote caches cannot satisfy the request. This can be determined by the probe responses in broadcast based protocols, or by a directory lookup in directory based protocols.

2.4.3 Broadcast Protocols

Broadcast-based cache coherence protocols are a natural choice for shared memory computers that are interconnected with a single bus. Write access to a bus is granted to at most one bus master at a time, while all other endpoints on the bus listen to the transaction of the bus master. This is precisely a broadcast communication pattern.

In such a bus-based system, a coherent cache has to snoop on all memory requests on the bus. The cache has to check whether it caches the same memory location. In the case of a possible coherence conflict, the cache has to respond on the bus within a certain timeframe to resolve this problem. Thus, broadcast-based cache coherence protocols in a bus based system are also called snoopy protocols.

In all other topologies than a bus, a broadcast must be implemented explicitly by sending broadcast messages to every cache. Due to network contention, it is difficult to guarantee that all caches can respond to a broadcast within a fixed, short period of time. Therefore,

every cache has to send a response, no matter if it detects a conflict or not. The requesting cache has to collect all responses before it may proceed.

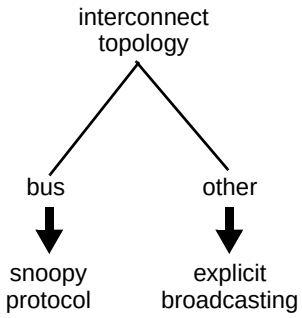


Figure 2-15. Influence of the interconnect topology on broadcast based protocols

The most basic cache coherence protocol would allow a memory location to be cached by at most one cache at the same time. The most significant problem of this protocol is that processes or threads that run on different processors in a shared memory system often share read-only memory, as for example program code. In this case, the simple protocol would be highly inefficient, as the ownership of the cacheline would have to toggle with every access in case more than two processors access the cacheline at the same time.

Thus, the simplest protocol that has been used in systems allows shared reading. It is implemented in the MSI protocol, named after the three states in which a cache entry may be:

- *Invalid*: The cache entry is not valid.
- *Shared*: The cache entry is valid; the respective cacheline may be shared with other processors. The cacheline is unmodified, i.e. it contains the same data as the corresponding memory location. Before writing to a cacheline in the *shared* state, other entries in other caches have to be invalidated.
- *Modified*: The cacheline contains data that has been written by the processor. The corresponding memory location has not been updated yet and therefore does not hold the actual value. The cacheline is exclusively held by this cache, therefore read and write operations can be performed on this line without notification of the other processors. If the cache decides to evict the cacheline, it has to be written back to memory.

The MESI protocol improves the MSI protocol by adding the *exclusive* state. It is one of the most popular protocols. For example, it is implemented in Intel's IA-32 and IA-64 processors [70]. The *exclusive* state is used to indicate that the respective cacheline is exclusively cached by the processor, but, in contrast to the *modified* state, unmodified. This reduces

snooping or probing traffic for data that is not shared between caches: a processor can write to a cacheline that is in *exclusive* state due to a previous read, and thus silently change the state to *modified* without notifying other caches. In order to decide whether a cacheline should be allocated as exclusive or as shared upon a read miss, different read or prefetch instructions may be supported by the processors. A *read_exclusive*, sometimes also called *read_with_intend_to_modify*, may be used instead of a "standard" read if the processor intends to write to the cacheline soon. A good choice for the standard read is to allocate the cacheline in the shared state if others share it, and otherwise on *exclusive* state. A *read_shared* may also be beneficial in some systems.

These instructions have their equivalent in the bus or interconnect protocol:

- A *read_exclusive* will be issued for the respective instruction, or for a write miss.
- A *read_shared* will be issued for the respective instruction, or for a read miss in the instruction cache.
- For all other read misses, a standard *read* will be issued.

The MERSI protocol, which is implemented in the IBM PowerPC 970 [72], is another improvement to the MESI protocol. Upon a read request, it allows to transfer a shared, unmodified cacheline directly between caches. The motivation to do so is that a cache's access time is significantly lower than the one of main memory DRAM. For not wasting bandwidth, the MERSI protocol ensures that at most one cache is delivering a cacheline in such a transfer, even if it is cached in multiple caches. The most recent reader of a shared cacheline will hold the line not in the *shared* state, but in the *recent* state. Only the cache that has the line in the *recent* state may forward the cacheline to the requester, after that, it has to change the state to *shared*. The reason why the most recent reader of the cacheline is the one that has to forward the data is the assumption of temporal locality: the most recent reader is also assumed to evict the cacheline after the caches that read the line before. If the most recent reader of the cacheline evicts the line, a subsequent read miss from any processor to the same address will have to fetch the data from memory, even though the cacheline may be present in the *shared* state in some caches.

In the following, two other improvements of the MESI protocol will be described in detail, the MOESI and the MESIF protocols.

2.4.3.1 MOESI

MOESI is an extension of the MESI protocol, and being used by AMD in their AMD64 architecture processors [20]. The five states of a cache entry are *modified*, *owned*, *exclusive*, *shared*, and *invalid*.

The *global state* of a memory location in MESI corresponds substantially with the states of the respective cache entries. The global states of MESI, which also exist in MOESI, are:

- *invalid*, i.e. not present in any cache,
- *modified exclusive*, i.e. present in exactly one cache in the *modified* cache entry state,
- *unmodified shared*, i.e. present in one or more caches in the *shared* cache entry state,
- *unmodified exclusive*, i.e. present in exactly one cache in the *exclusive* cache entry state.

MOESI adds a new global state, which does not correspond directly with the cache entry state, but with a combination of two cache entry states:

- *modified shared*. The location is present in exactly one cache in the *owned* state, and in any number of caches in the *shared* state at the same time.

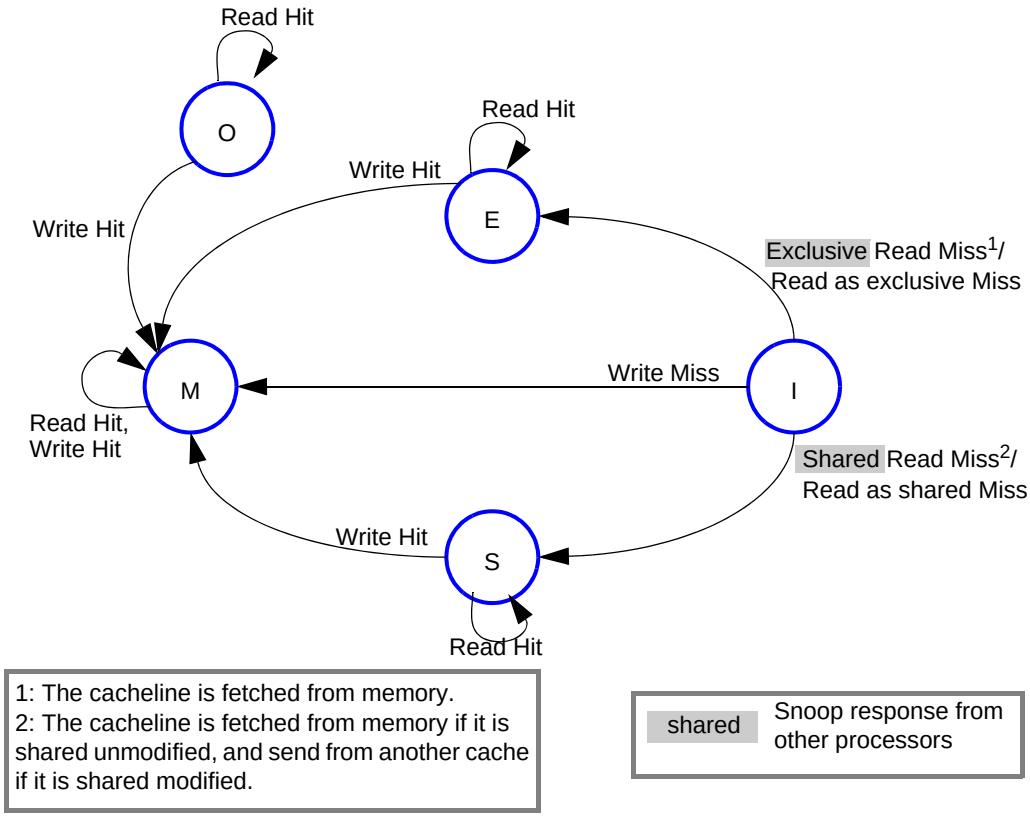


Figure 2-16. MOESI state diagram for a requesting cache

So, MOESI can share a cacheline among caches that is modified. In contrast to the MESI, protocol, a modified cacheline does not need to be written back to memory in order to share the memory location. The owned state has been introduced to mark exactly one cache entry that the respective cache it is the one who must forward the line in the case of subsequent read request from other nodes. Also, it must write the line back to memory if the cacheline is evicted.

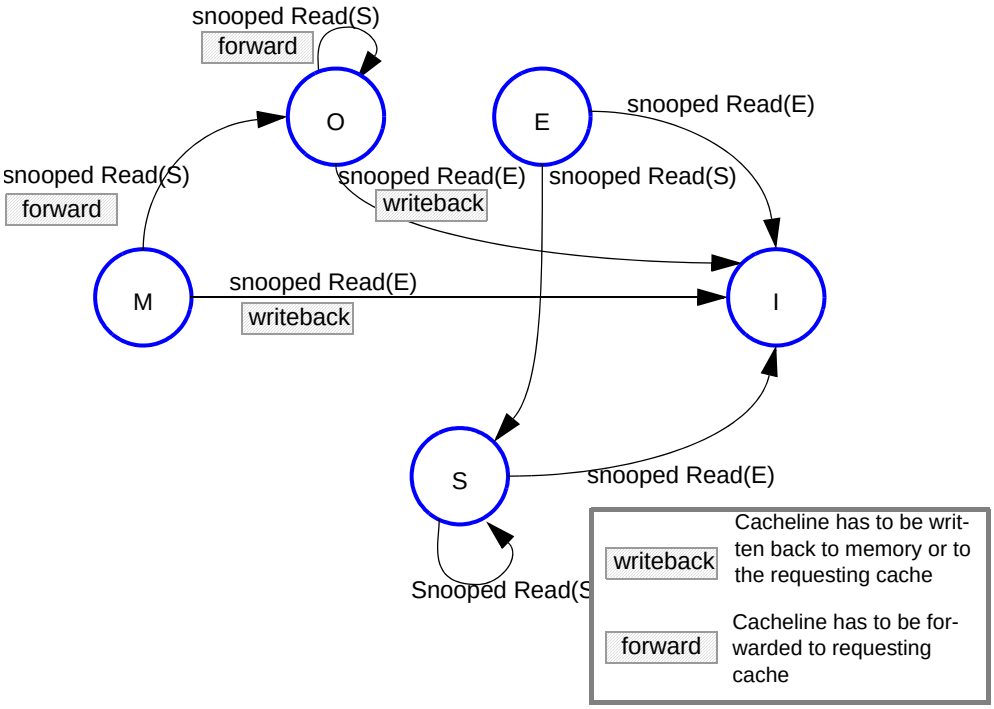


Figure 2-17. MOESI State diagram for a snooping cache

The description of the MOESI protocol from the requester's viewpoint, as depicted in Figure 2-16, is as follows:

- Read Hit: No coherency actions have to be taken.
- Read Miss: if there are other caches in the states *exclusive*, *shared* or *modified*, the cacheline goes from *invalid* to *shared*, otherwise it goes to *exclusive*. If another cache is in *exclusive*, this other cache goes to *shared* too; if another cache is in *modified*, this other cache goes to *owned*.

- Write Hit: The master's new state is *modified*. If the cacheline has been in *exclusive* or *modified*, no coherency bus transactions have to be done. If the cacheline has been in *shared* or *owned*, all other caches in state *shared* have to be invalidated.
- Write Miss: The cacheline goes from *invalid* to *modified*. All other caches holding this line go to *invalid*.

Figure 2-17 shows the state transitions of snooping caches.

2.4.3.2 MESIF

The MESIF protocol has been suggested by Intel [62]. Basically, it is an improvement of the MERSI protocol. MERSI allowed only the forwarding of unmodified cachelines. MESIF allows the forwarding of both modified and unmodified data.

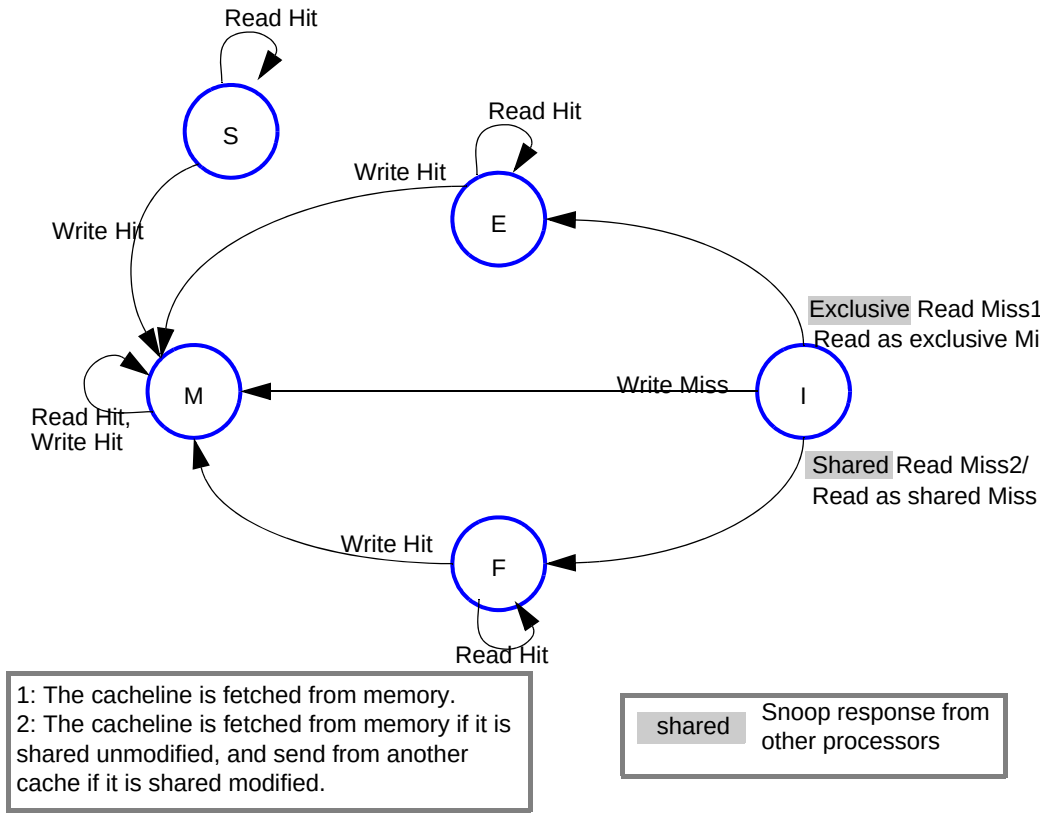


Figure 2-18. MESIF state diagram for a requesting cache

In the case of forwarding a modified cacheline, the line is simultaneously written back to memory, so that MESIF does not have to distinguish between *modified shared* and *unmodified shared*: there is only the latter global state. In all other respects, the F state behaves exactly like the R state: the most recent requesting cache has the cache entry in F state, all other sharers in S.

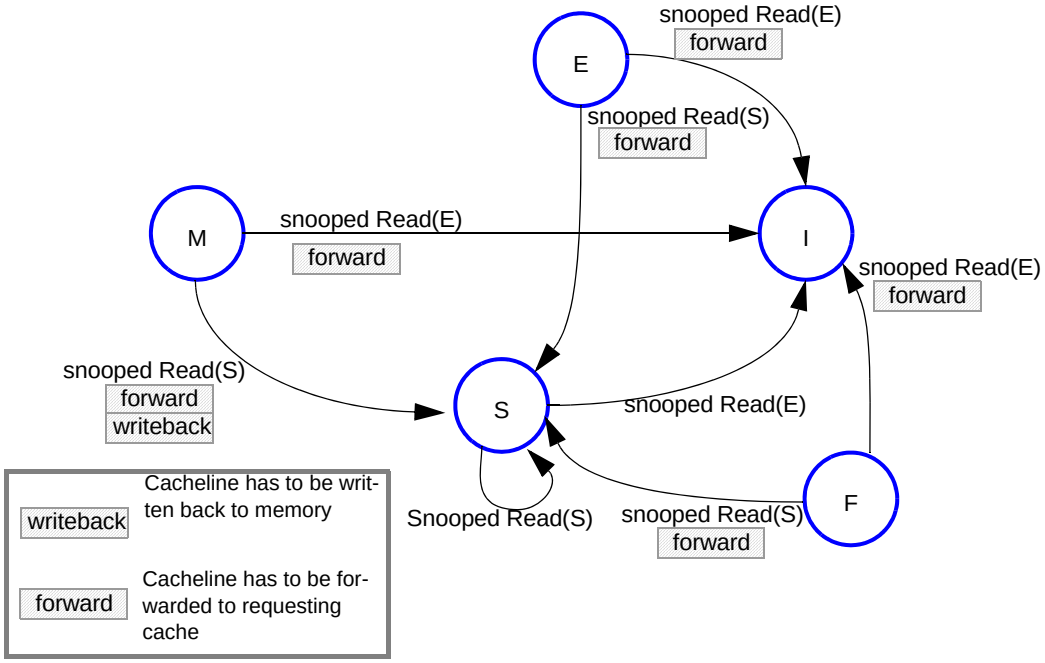


Figure 2-19. MESIF state diagram for a snooping cache

The description of the MESIF protocol from the requester's viewpoint, as depicted in Figure 2-18, is as follows:

- Read Hit: No coherency actions have to be taken.
- Read Miss: if there are other caches holding the line in any state, the cacheline goes from *invalid* to *forward*, otherwise it goes to *exclusive*. If another cache is in *exclusive*, this other cache goes to *shared*. If another cache is in *forward*, this other cache goes to *shared*, and forwards the line to the requesting cache. The same applies if another cache is in *modified*, additionally, the cacheline is written back to memory.

- Write Hit: The requester's new state is *modified*. If the cacheline has been in *exclusive* or *modified*, no coherency bus transactions have to be done. If the cacheline has been in *shared* or *forward*, all other caches holding the same cacheline have to be invalidated.
- Write Miss: The cacheline goes from *invalid* to *modified*. All other caches holding this line go to *invalid*.

Figure 2-19 shows the state transitions of snooping caches. It can clearly be seen that the idea of the protocol is to forward data directly from cache to cache if possible. On the other hand, it ensures that at most one processor is forwarding data

Hierarchical Buses. The scalability of a bus is very limited. As the available bandwidth of a bus is constant, the per-processor bandwidth decreases with a larger number of processors. Hierarchical snoopy buses are a better scalable solution, but only if the traffic on the individual bus segments of the hierarchy can be reduced. The two-level hierarchy of the NEC AzuzA [98] is an example of such a system (see Figure 2-20). The addresses of all memory requests on the local buses are broadcast on the system address bus. The system address controllers (SAC) listen to all these addresses. The SAC incorporates a snoop filter, so that snoop requests are only forwarded to the local bus if there is any chance that one of the processor's caches holds the requested memory block.

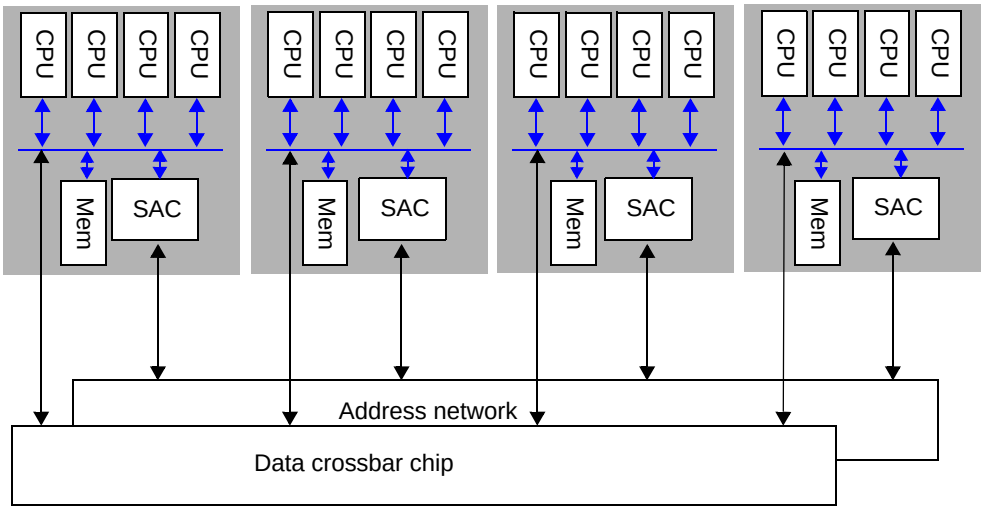


Figure 2-20. Hierarchical snoopy-bus NUMA system

2.4.4 Directory-Based Protocols

Directory-based cache coherence has been proposed by Censier & Feautrier in 1978 [65], even before bus based coherence was introduced. In such a protocol, a directory keeps track of the state of memory blocks. The directory contains the state of every block that is currently being cached, and information about which processors have copies of that memory block in their caches, and in which state it is.

Generally, directory lookups replace broastcasts. For example, a *read_exclusive* request from a cache upon a cache miss will cause a lookup. If the lookup shows that no other cache holds a copy of the memory location, the requester can allocate the cache entry in the exclusive state. If the directory entry shows that other caches hold the same location, invalidation messages will be send to those caches. If another cache holds a modified copy of the cacheline, it must also be notified to forward this line.

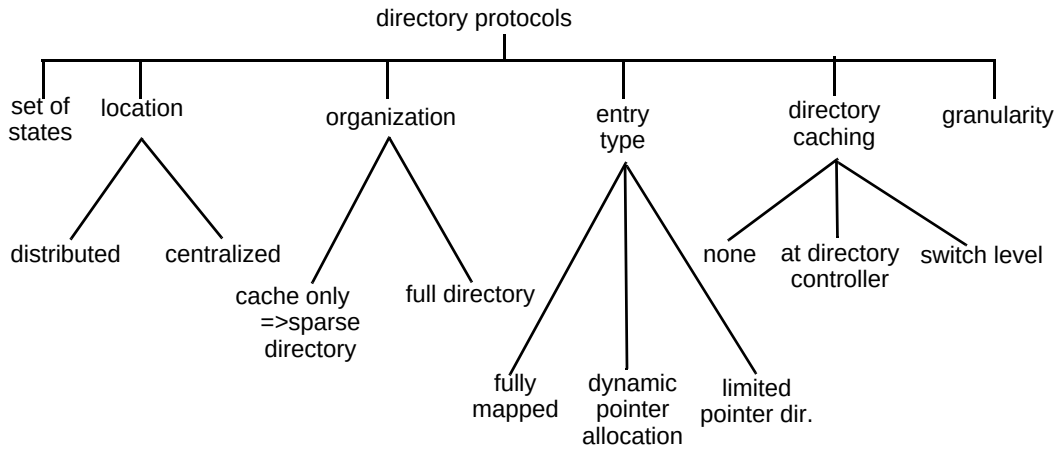


Figure 2-21. Design space of directory cache coherence protocols

The design space of directory cache coherence protocols is shown in Figure 2-21.

The set of states. Theoretically, directory protocols can work if the state of a memory location is held in the directory only. However, this would mean that a cache has to perform an directory access every time it wants to change the state. Therefore, the state is included in the cache entries as well. Generally, it must be ensured that directory and cache entry state are consistent. Some inconsistency may be allowed: A cache entry may transition from the exclusive state to a modified state without requesting the update from the directory, which would involve communication latency in the memory access. As well, a cache may evict a

cacheline in a non-modified state without updating the directory. This reduces traffic for cache evictions. On the other hand, it increases traffic in the case another cache wants to acquire the same memory location later on, as an invalidation has to be sent to the first cache. Usually, only a small fraction of the memory locations that a processor caches is shared and written to by processes, so that this strategy usually leads to an overall decrease of traffic

The MESI state set is well suited for the use with directories and has frequently been implemented, for example in the Dash system [97]. If the protocol allows caches to silently transition from E to M, the directory does not need to distinguish between both states, but only knows the MSI state set.

Instead of the MESI state set, the MOESI state set might be used as well as a basis for directory-based protocols. For cache to cache forwarding of unmodified cache lines, extensions like MERSI and MESIF are not required. A node that should forward a cacheline could just be selected from the list of nodes in the directory. A variety of selection criteria could be used. For example, the most recent requester could be chosen, if it is marked in the directory entry. Another criterion could be to select a cache that is close to the original requester.

Location. As there is exactly one directory entry for every (cached) memory block, the directory should be placed at the respective memory controller. Most shared-memory multiprocessors use distributed memory. In such an architecture, distributed directories should be used: to every physical memory component, a directory is attached that contains the directory information regarding the memory component.

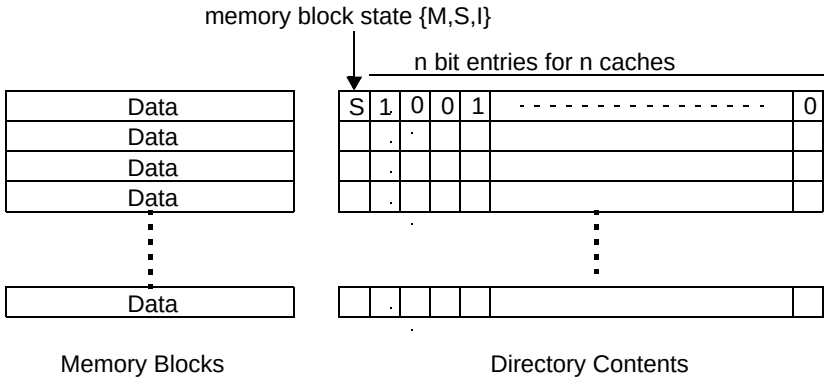


Figure 2-22. Directory contents in a full mapped directory. There is a bit for every cache, stating whether the memory block is cached by that cache (bit=1) or not (bit =0)

Organization of Directory Entries. Depending on the way this information is held by the directory, directories can be divided into three classes:

- Fully-Mapped Directories [65]: in a directory of this type, the directory information for a memory block has a bit for every processor cache in the system (see Figure 2-22). This is an inexpensive and fast solution for small scale multiprocessor systems. For systems with a higher number of processors, the directory becomes quite large. Another drawback is that the maximum number of processors is determined by the directory structure. As a result, full mapped directories do not scale well.
- Limited (Pointer) Directories [68] are similar to full-mapped directories. They differ in so far that they have entries for a fixed number n of caches, which may be smaller than the total number of caches in the system. Therefore bit flags are not unique and have to be replaced with pointers to caches. For example, the directory might support 128 processor caches, which would require 7 bit pointers. The directory might now have 8 pointer entries, resulting in a memory use of 7*8=48 bit to store the pointers, compared to 128 bit for a full-mapped directory. In this example, only 8 caches can share a specific cacheline. If a 9th cache requests to share it too, one of the other caches has to invalidate the cacheline.
  These directories overcome the problem of a directory line getting too large in large systems. However, they do have a scaling problem too, since the number of caches that can hold a memory block simultaneously is fixed by the directory line size.
- Dynamic Pointer Allocation Directories [69] work with pointers as well. But instead of storing them in a fixed number of fields, they are stored in a dynamic list. The space for storing list elements can be shared among all memory block entries. There is more overhead both in the directory logic and the directory memory space, although memory space is much more efficiently used since it can be shared among all memory blocks. This is the most flexible model.

Organization of the directory. A full directory implementation has one entry for every memory block. A scalable solution would be to keep the directory in main memory. On the other hand, the access to the directory is timing critical. Thus, the directory should be implemented in a fast memory technology, and be implemented on the same chip as the memory controller. This limits the scalability of the memory size of a node.

An idea to decrease the size of a full directory is to increase the memory block size of directories to a multiple of the cacheline size. However, the larger the memory block size becomes, the more drastic is the effect of false sharing: if one processor writes to a memory location, all cached copies of the memory block will have to be invalidated.

In contrast, a sparse directory [66] keeps directory information only for a part of the memory blocks. Due to the limited size of processor caches, only a small fraction of main memory is cached at the same point in time. It is sufficient to have directory entries for those memory blocks that are currently cached. A sparse directory is build like a cache without a back-up store. Every entry in the cache consists of an address tag and the directory entry itself. If a directory entry does not exist for a memory location, the memory block is uncached. If a new entry is allocated, an old entry has to be removed, which is selected by a cache replacement mechanism as in any other cache. However, a replacement means that the respective cachelines have to be invalided. Also, the cacheline may have to be written back if it is dirty.

For a system with a central directory, the number of entries in the directory must be at least the same as the accumulated number of entries in the system's caches. This theoretically limits the scalability of the number of processors in the system, but a central directory is very limited in its scalability anyway.

Systems with distributed directories have the potential to scale better. If every distributed part of the directory would have to be able to hold the status of all cache entries of all in the system, this would destroy the scalability. Thus, the size of the distributed directories must be fixed. This is not a problem in itself: if a system is scaled up in a smart fashion by increasing the number of processors, memory controllers and (distributed) directories, the number of memory accesses per memory controller should not increase significantly. In particular, the number of memory blocks per directory that is cached does not necessarily increase.

While a sparse directory significantly reduces the number of entries that have to be stored, the number of bits to store per entry increases, as a tag has to be stored with every entry. An example of a central, sparse directory is the Intel Xeon architecture (see Section 2.5.1).

Caching. A full directory that is implemented in slow memory may be cached directly at the directory controller [67]. Another way to speed up directory lookups upon a memory operation of a processor would be to cache directory entries close to the processor. This is difficult, as such directory caches must be kept coherent with the main directory. An interesting approach are switch directories [119], which are used to speed up cache to cache transfers. In a directly interconnected NUMA architecture, every switch has a small directory. The state of the directory is maintained by keeping track of memory requests and responses that cross the switch. Requests from caches are sent to the main directory. If there is a hit in any of the switch directories in the path, this directory may redirect the request to a cache that holds the cacheline, so that a cache-to-cache transfer of this cacheline is sped up.

2.4.5 Serialization of Conflicting Accesses

Cache coherence protocols as presented above ensure that caches can be kept consistent. They do that by giving only one cache write access to the same memory location at a time, and invalidating cached copies that hold old values of a memory location. Also, these protocols support sequential consistency, if every component acknowledges invalidation requests and a processor waits for arrival of all those acknowledgements before issuing subsequent memory accesses. However, one problem has not been discussed yet: what happens if two or more processors access the same memory location at the same time?

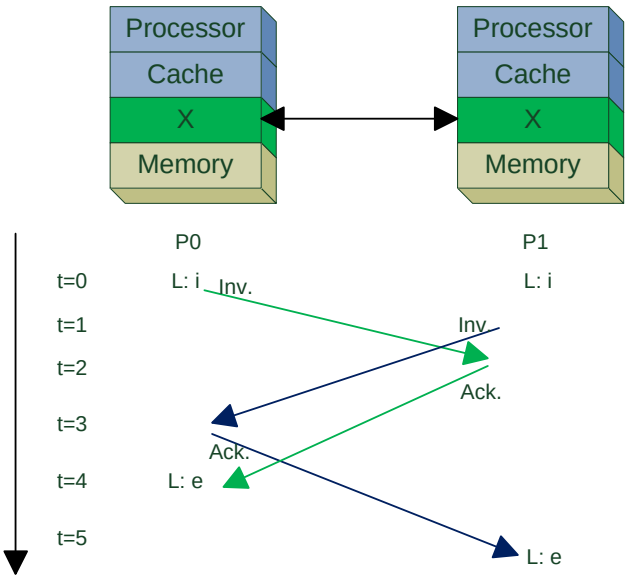


Figure 2-23. Conflict caused by simultaneous access to the same memory location

Assume for example a 2-node direct network system, using a MESI cache coherence protocol. Both processors have the same memory location L not cached yet. If one processor P0 decides to obtain an exclusive copy, it sends an invalidation message to P1. Upon the receipt of the acknowledgement, the cache of P0 gets its exclusive copy. However it might also be the case that both processors decide to obtain an exclusive copy of L at about the same time, as shown in Figure 2-23. P0 sends an invalidation request at t=0, P1 at t=1, so that both requests overlap due to the latency of the network. Both processes will have to acknowledge the requests immediately in order to avoid a deadlock. If both processors con-

tinue to pursue their requests, both will have an exclusive copy of the request at the same time, which is an inconsistent state.

Conflicts may always occur when at least one of the simultaneous requests tries to acquire a cacheline in *exclusive* state. Depending on the protocol and the implementation details, conflicts may also occur in other cases.

There are two very different approaches how such conflicting accesses can be treated. One solution introduces a centralized serialization of all requests to the same memory location. A request may only proceed beyond the serialization point if all previous requests to the same location finished, i.e. have been globally observed. The other possible solution is to allow such conflicting accesses, but to resolve conflicts if they occur. Figure 2-24 shows an overview about which strategy may be used in which system.

Conflict Avoidance. In bus-based systems, a central serialization is intrinsic to the system. Even if memory accesses are issued at the very same time by different processors, they will have to compete for bus arbitration. Depending on the arbitration policy, one of the requests will “win” and go first. The other processor’s cache snoops on the address of this request, and thus can determine that is not allowed to issue the request until the request for the first processor has completed.

Other network topologies do not have such a serializing characteristic. A solution to this problem is a central instance that every request to a memory location has to pass. It keeps track of memory operations that did not commit yet. If an operation to the same memory location arrives, it is queued and will only be handled if the previous request committed.

In directory-based system, a request from a cache will always go to the respective directory first. Thus, the directory is a point where serialization of accesses to the same memory location can be performed by buffering incoming requests until all previous requests to the same memory location have completed. If the directory is distributed throughout the system, the ordering point is distributed just the same way.

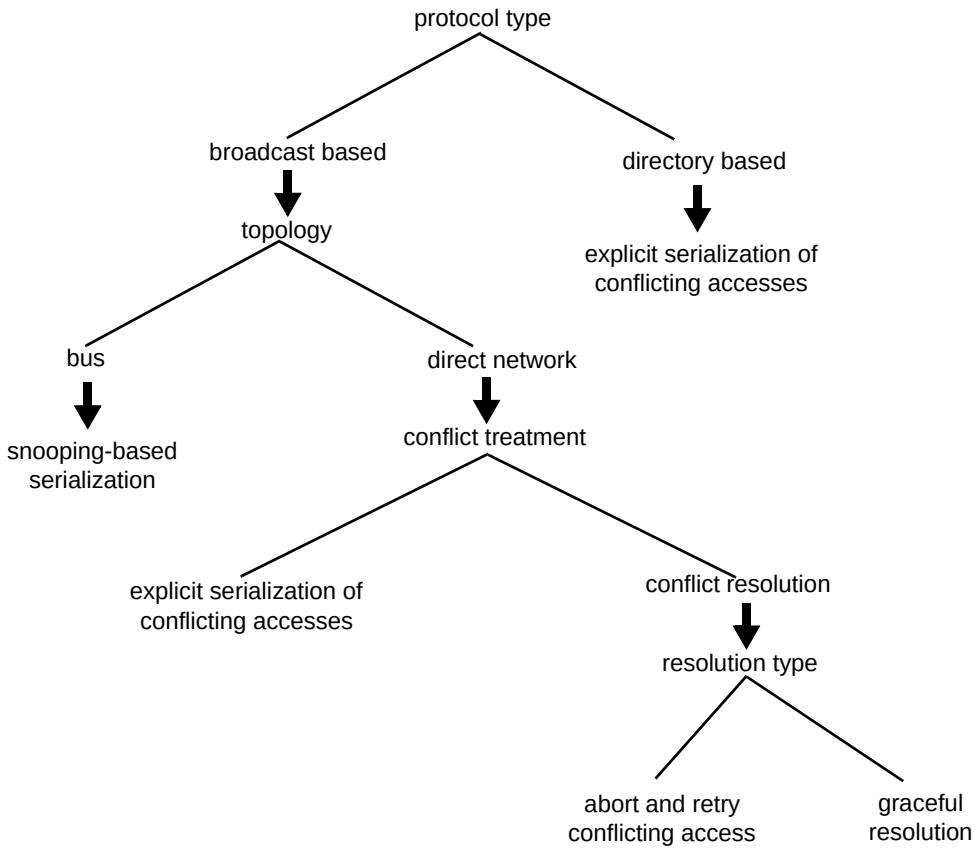


Figure 2-24. Treatment of conflicting accesses

In systems that implement an explicit broadcast, ordering can be done the same way at the memory controller. In MOESI, a cache does not broadcast invalidations directly to other caches. Instead, it sends a request to the memory controller. The memory controller queues these requests, and issues the invalidation requests only if there is no active request to the same memory location. Acknowledgements may be sent to the original requester directly, in order to reduce the waiting time for this processor. In this case, the requester has to notify the memory controller that the request completed. For example, AMD Opteron processors use this method.

Figure 2-25 (a) and (b) show the respective flow of packets. An important characterization criterion for these protocols is the maximum number of hops that have to be taken on the path from the processor's request until the processor gets all required replies. The serialization based broadcast protocol requires three hops to be taken. A directory based protocol will also require three hops if it has to send directed probes to caches. Otherwise, it is a two hop protocol.

Conflict Resolution. Another strategy is to allow conflicts, but to resolve them when they occur. The idea is to speed up accesses for which a conflict does not occur. The resolution of a conflict, however, will usually be more expensive in terms of latency. So conflict resolution may offer benefits over conflict avoidance if conflicts occur only for a small fraction of all memory accesses.

Figure 2-25 (c) shows how a read access without conflicts may look like. All requests can be sent out in parallel, without having to take an extra hop to the memory controller. This may reduce latency compared to the serializing solution. If a conflict occurs, it will be detected from at least one of the conflicting requesters. A conflict occurs if the requester gets probes from another cache for the same memory location for which an own request is currently active. After it has been detected, there are different ways how a conflict may be resolved. In the decentralized solution, every processor that sees a conflict could for example abort its memory request, and start the request over again after a short waiting period. This mechanism is similar to the store conditional instruction (STWCX) of the PowerPC processor architecture [71]. A conditional store is executed only if since the last load and reserve instruction (LWARX), requests to the same address have not been observed on the processor bus. However, in the case of cache conflict resolution, aborts and restarts of memory request should be done in hardware, transparently to the software. If both requesters observed the conflict, they will both restart their requests. In order to avoid these requests to collide again, they should not restart the request immediately, but e.g. with a random delay.

In the case of the MESI protocol, this method works as described above. Forwarding protocols as MOESI are more challenging: a dirty copy of the requested memory block may be transmitted to the requester that detected a conflict. Thus, the request cannot be simply aborted. Instead, it must be assured that the dirty copy does not get "lost", and in particular, that it reaches the other requester in the conflict, as it may not have detected the conflict and thus needs the dirty copy.

.

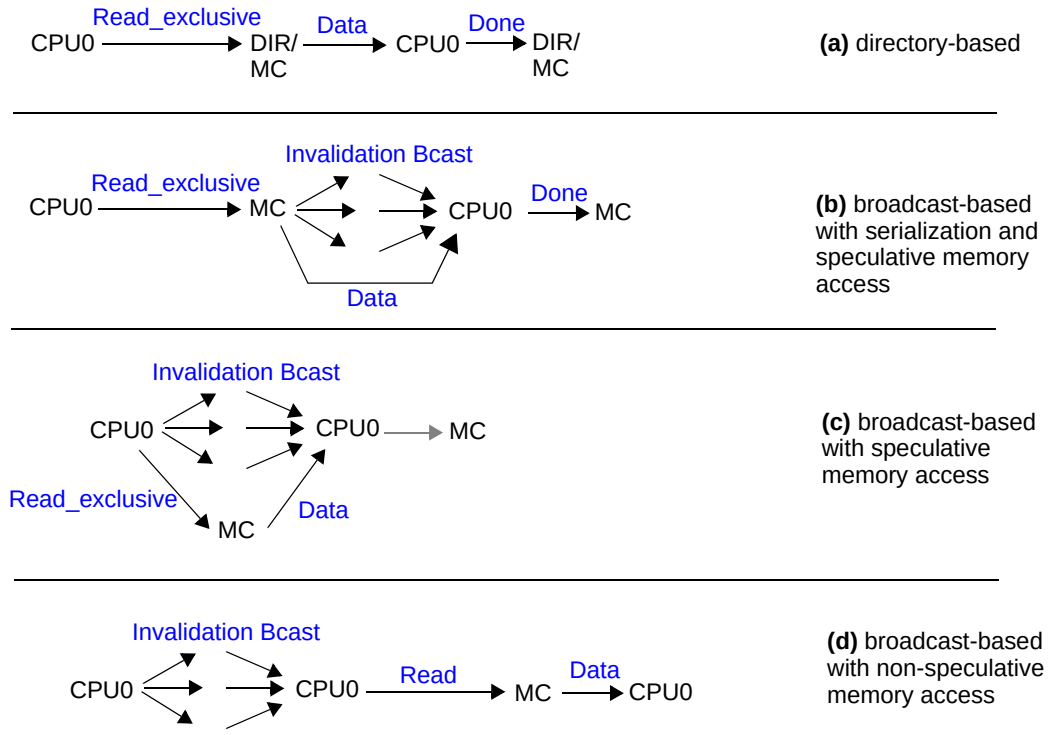


Figure 2-25. Transfers for a *read_exclusive* request for different conflict treatment strategies

Hum et al. [63] propose a mechanism as in Figure 2-25 (d) for the MESIF protocol. Here the conflict is not resolved by canceling the current request. Instead, every requester reports the conflict to the other requesters. This assures that all know of the conflict, as it may be the case that only one detected it. The basic idea is as follows: any node that wants to obtain a cacheline in an exclusive state must read it using the *port_read_invalidate_line* (PRIL) command, even if it has been cached before in the shared state. If the requested memory block is being cached in a remote cache C_r in a state that allows forwarding, this cache will forward the line to the first request it received. Thus, one of the requesters is the winner C_w, which may use the cacheline. The loosing requester C_l will be blocked, as C_r does not answer any requests to the memory location until C_w acknowledges to C_l that potential conflicts have been resolved. Before doing so, C_w will report the conflict to the memory con-

troller, which thus can defer memory requests that might come from C_l. Also, the memory controller will advise C_w to forward the cacheline to C_l and invalidate it in C_w.

Similarly, if no cache can forward the data, a winner will get a read response from the memory with the valid data. Again, the memory will advise C_w to forward the cacheline to C_l. The read request from C_l will not be answered with delivering the data, but with a notification that C_w will deliver the data.

There is also a variant of this protocol [64], in which the winner is not allowed to use the data. Instead, it is stored in a buffer. The memory controller then decides about the order in which the caches are served with the data.

2.5 Introduction to x86 Systems

The market for high-performance processors is dominated by x86-architecture processors from Intel and AMD. Like a perpetual motion machine, their large market share yields a good performance to price ratio, which again leads to a large market share.

Intel's x86 architecture is very conservative: the northbridge is an external chip to the processor, the interface is based on a snoopy-bus protocol. On the other hand, Intel has a unique technological advantage. As a result, Intel can integrate more SRAM memory on chip, which allows the integration of larger caches as well as the implementation of directories.

AMD's architecture is different. The integration of the northbridge functionality into the processor chip, and an efficient, serial HyperTransport protocol between chips allow the construction of glueless NUMA shared memory systems. Both architectures will be presented in the following two sections.

2.5.1 Intel Xeon Architecture

The block diagram of a typical Intel Xeon dual processor system is shown in Figure 2-26. For example, the Bensley platform integrates two Xeon 5000 processors and the 5000x northbridge chipset. Processor and northbridge are connected over the Intel Front Side Bus (FSB). Both FSBs are independent buses and connected to a switch in the northbridge. The FSB consists of a parity protected 64 bit wide data bus, and a 36 bit wide address bus. With a data rate of up to 1333MHz, the raw bandwidth of the data bus is 10.5 GByte/s. The address bus has half the data rate, and thus has a peak bandwidth of 3 GByte/s.

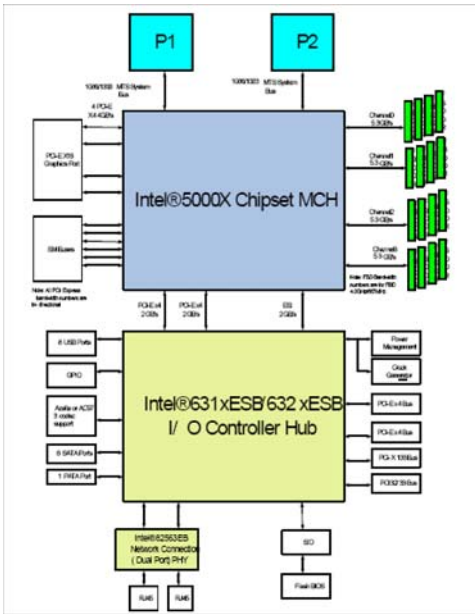


Figure 2-26. A 2-processor Intel Xeon system

The Bensley platform integrates a coherence directory, called snooping filter, in the north-bridge chip. The directory holds entries for all memory blocks that are cached in the system, i.e. it is a sparse directory. The directory has a size of 1 MB, and is organized in two affinity groups, which each consist of 8 kilo sets that are 16-way each. The pseudo-LRU algorithm is used as replacement policy. If a new entry is allocated in the directory, it will be allocated in the affinity group that is assigned to the processor from which the memory request was issued. Thus, the affinity group implicitly encodes the processor cache that caches the memory block. This saves one bit per directory entry (see Figure 2-27).

A directory lookup including ECC check is done within one clock cycle. Every lookup is followed by a write to the directory. Either, a new line is allocated, or the pseudo-LRU bits are updated. The directory is clocked with 533 MHz, which means that the directory can perform 267 MLUU (Mega Look-Up Update) Operations per second, which fits to the incoming request rate of 267 MHz.

The directory can hold information for up to 16 MB of processor cache. To support larger caches or a larger number of processors, the size of the directory scales with the total

amount of processor cache in the system. Announced have been the Cranberry Lake platform, which supports 2 Xeon 5000 processors with 24MB cache, and the Caneland platform, supporting 4 Xeon 7000 with 64MB total cache [54].

Snoop Filter Physical Address Partitioning

Tag (21b)	Set (13b)	Byte Offset (6b)
A[39:19]	A[18:6]	A[5:0]

Snoop Filter Entry

Bits	Value
[31]	**Redundant bit.**
[30:24]	**ECC check bits**
[23]	**State of the cache line** 1 The cache line is in E/M state, i.e. the line is either exclusive (but clean) or modified (dirty) state. 0 The cache line is in non-E/M state, i.e. S state if Bus presence vector is non-zero or I state if Bus presence vector is zero.
[22:21]	**Bus presence vector** [00] The entry is invalid [xx] The entry is present in any of the processor L2 in the corresponding FSB. Bus0 is the least significant bit. Bus0 corresponds to FSB0 on the Intel 5000 series chipset MCH. Bus1 corresponding to FSB1
[20:0]	**Tag portion of the address**

Figure 2-27. Snoop filter entry format and address partitioning [38]

2.5.2 AMD

The AMD Opteron architecture is designed to build up small-scale NUMA systems. It employs a direct network architecture. Every node integrates processor cores, caches, cHT routing resources and the northbridge on one single chip, as shown in Figure 2-28.

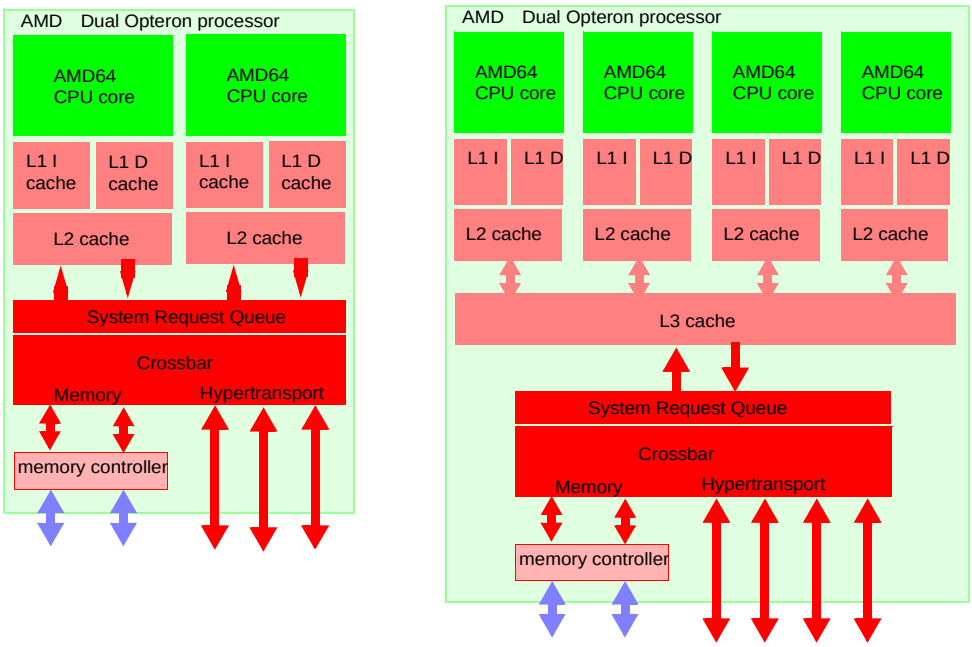


Figure 2-28. 2nd and 3rd generation AMD Opteron processors

The number of HyperTransport links and processor cores per chip depends of the processor generation. The third generation Opteron, codenamed Barcelona, supports up to four cores and four HT links as well. HyperTransport links can be configured to be either coherent links to connect other processors, or noncoherent links. In this case, the HT link interface functions as an I/O bridge. An integration of more cores is foreseeable: an 8-core processor, codenamed Sandtiger, is announced for 2009.

Figure 2-29 shows the topology for an 8-chip system, based on 2nd generation Opterons that support up to three HT links. As the broadcast-based MOESI coherence protocol significantly increases the latency of memory accesses for networks with a larger diameter, 2- and 4-chip systems prevail. The Opteron system architecture, and in particular the integra-

tion of devices, will be evaluated in detail in the next chapter. Intel is developing a very similar NUMA processor interconnect, called CSI or QuickPath [43].

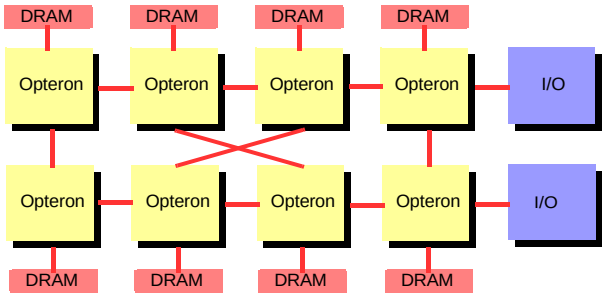


Figure 2-29. An 8-node Opteron topology

2.6 Examples of Parallel Systems

The following sections detail some other very interesting architectures. They are good examples to demonstrate state of the art parallel computing. The following systems will be described: the Sun UltraSPARC T2, Cray T3E, XT3 and XT4, and the IBM BlueGene/L. Additionally, network interfaces that are connected over standardized peripheral interfaces will briefly be described.

2.6.1 Sun UltraSPARC T2

Multithread architectures as the HEP [32] and Tera [33] supercomputers exploit thread level parallelism (TLP). The UltraSPARC T2 processor [17] [18] is the most recent commercial implementation of a TLP-exploiting processor.

As shown in Figure 2-30, the T2 features 8 processor cores with L1 caches. In every core, 8 different strands can be loaded at the same time; two of these may run simultaneously. Every core has two integer pipelines, one floating point and one memory pipeline. The execution of strands is switched every cycle using a last-recently-issued policy. Only those strands are considered that are marked as available. A strand may become unavailable for different reasons, the most important one is an L1 cache miss. A strand becomes available again as soon as the event is resolved.

The L1 cache consists of an 18 kByte, 8-way set associative instruction cache with a cacheline size of 32 byte, and an 8 kByte, 4-way set associative data cache with a cacheline size of 16 byte.

A remarkable difference to most other architectures are the L2 caches. They are not located at the processor core, but at the memory controllers and hold entries of the memory range of the respective memory controller. Every of the four memory controllers, called memory control unit (MCU), has two sets of L2 cache. Thus, coherence does not need to be maintained among the larger L2 caches, but only between each of the L2 caches and the smaller L1 caches. A directory-based protocol is being used, the directory controller is located at the L2 cache. The total size of the L2 caches is 4 MByte. All sets are 16-way set associative with a cacheline size of 64 bytes. The cache hit delay is 26 cycles for data and 24 cycles for instructions. L2 and L1 caches are interconnected by two unidirectional 8x8 crossbars. Arbitration prioritizes the oldest requests.

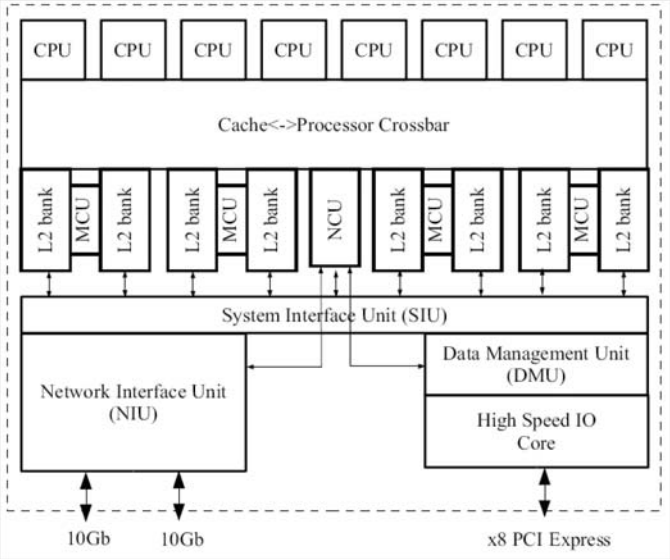


Figure 2-30. The Sun UltraSPARC T2 processor [17]

The System Interface Unit (SIU) is the interface to I/O devices. It directly interfaces the L2 caches. However, writes are bypassed directly to the DRAM controller. A packet based protocol with credit based flow control is used between SIU and I/O components, which are 2 integrated 10 GB Ethernet MACs and one x8 PCI express link.

All parts of the chip above the system interface unit are in the core clock domain of 1.5 GHz. The SIU has a 350 MHz interface to the I/O modules. The T2 protects memory and datapaths outside of the processor core with parity bits.

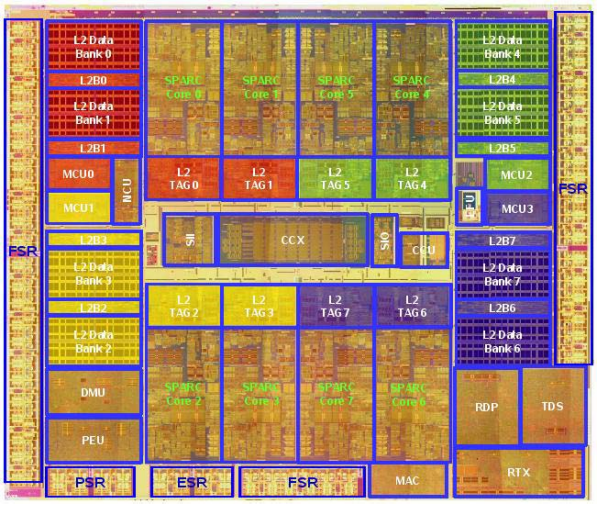


Figure 2-31. The Sun T2 die with an area of 342 mm^2

2.6.2 Cray T3E

The Cray T3E [84] was the successor of the T3D and has been presented in 1996. It connects up to 2048 processing elements (PE) using a direct interconnection network with a 3D torus topology. Routing in this network is fully adaptive and minimal path [85]. As depicted in Figure 2-32, every PE consists of one DEC Alpha 21164 processor, up to 2 GB local memory, control logic and a router for the interconnection network.

The T3E is a not a true distributed shared memory system: A processor can directly access only its local memory. However, a global shared memory view allows the processor to access remote memory using put/get semantics. All communication is done using the so-called E-registers, a set of 512 user and 128 system registers that are part of the control logic of a PE. In contrast to the local memory, which may be cached by the processor, these registers are memory mapped I/O space to the processor, and thus uncacheable. Puts and gets can be initialized by writing the respective command to the E-register file. One parameter is the E-register which is the local source for a put or the local target for a get. The remote address is specified using an address index, which will be used to lookup the global virtual address and also the logical PE number, which then is used for a lookup in the routing table.

Upon arrival of a request on a remote PE, the global virtual address is translated into a local physical address.

Both puts and gets can work on 32bit and 64 bit words or on vectors of 8 of these words with an arbitrary stride. The result of a put will be placed in the specified E-register. As long as the put does not complete, the register is marked “invalid”. A load operation from the processor on an invalid register will stall until the register content is available. It is assumed that 128 E-register are sufficient to generate enough overlap to be able to utilize the full maximum PE-to-PE bandwidth.

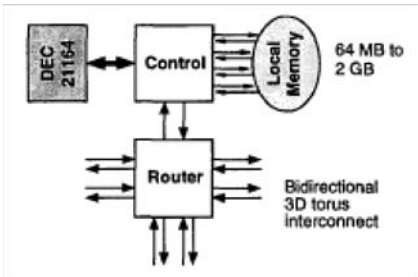


Figure 2-32. T3E PE Block diagram [84]

With a mechanism that is also based on the E-registers, it is also possible to send and receive messages. There is also a hardware barrier mechanism: Instead of a dedicated barrier network as in the T3D, barrier messages are sent as packets in a dedicated virtual channel over the interconnect fabric. Every PE has 32 Barrier synchronization units (BSUs). Every such unit can implement a node in a barrier tree. A register within the BSU indicates which network directions are children to the tree, and whether the local PE is a child. If the BSU is not the root node, it also contains the direction of the parent node in the tree. The BSU also keeps track of which child nodes have entered the barrier. If all child nodes entered, a corresponding message is sent to the parent node. As soon as the root node has been reached, completion messages are multicast downwards the tree.

2.6.3 Cray XT3 and XT4

Cray calls the XT3 [91] and XT4 [92] systems to be successors of the T3E, and indeed, the system architecture is very similar. As in the T3E, a direct network with a 3D torus topology connects up to 30508 compute PEs in both systems. From the available documentation, the only difference between both systems seems to be that the SeaStar 2 interconnect of the

XT4 offers a higher bandwidth than the SeaStar interconnect of the XT3. So, while the remainder of the section describes the XT4, most facts also apply to the XT3.

Every PE consists of an AMD Opteron processor with up to 8GBytes of local memory. A SeaStar chip is directly connected to the processor over a non-coherent HyperTransport link. With a width of 16 bit and a clock of 800MHz, it offers a bidirectional bandwidth of 6.4Gbyte/s. Additionally to the compute PEs, a system may contain service PEs, which may be configured to provide login, I/O, system or network services. The operating system on the compute PEs is a UNICOS/Ic microkernel, Linux is running on the service PEs.

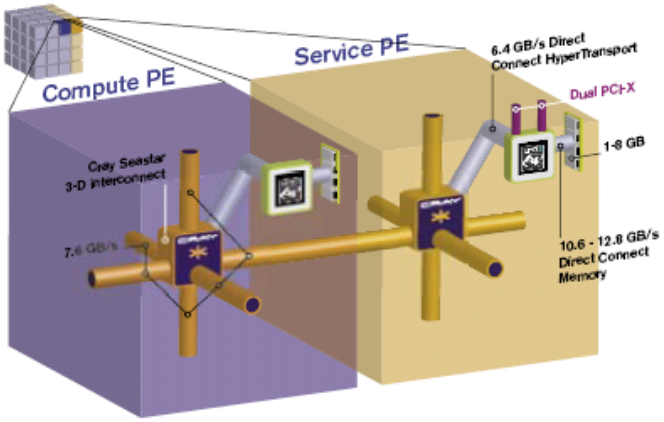


Figure 2-33. XT4 processing element block diagram [92]

The block diagram of the SeaStar2 chip, shown in Figure 2-34, resembles very much the diagrams of the dedicated NICs presented in Section 2.6.5. And indeed, it is not build to support a fine-grain, hardware-based communication scheme known from the T3E, but to support the MPI 2.0 [74] and SHMEM software libraries. I/O is done using the Lustre cluster file system [93].

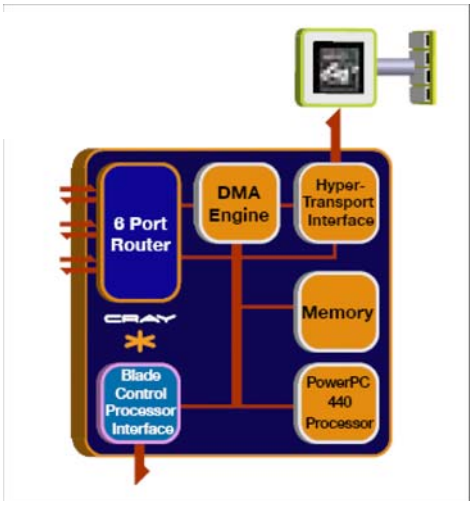


Figure 2-34. Cray SeaStar2 block diagram [92]

2.6.4 IBM BlueGene/L

The IBM BlueGene/L [101] is designed for high numbers of computing nodes. The currently fastest supercomputer in the world is a 106,496 node BlueGene system [99]. These nodes are interconnected by three dedicated networks [100]. The most important network is a 3D torus network with virtual cut-through, adaptive routing. This network is used for point-to-point message passing between the nodes. Deadlocks are avoided by the use of four virtual channels. Every hop in the network adds a latency of 100 ns, the unidirectional bandwidth of every of the six links of a node is 1.4 Gb/s. A barrier network is implemented with four global OR structures over all nodes. A collective network allows to statically built up a broadcast topology which may be used for one-to-all and all-to-one communication patterns. As the collective network interface of every node has three bidirectional links, a natural choice for this network is a binary tree.

Every node of the system, as depicted in Figure 2-35, hosts two PowerPC 440 processors that have been enhanced with an additional floating point unit.

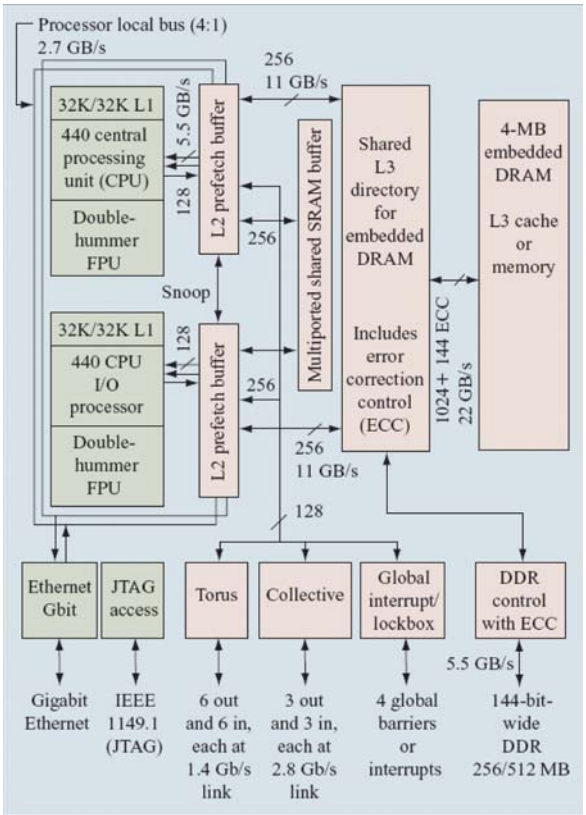


Figure 2-35. BlueGene/L node architecture

2.6.5 NIs on Standardized Peripheral Interfaces

In contrast to the full system solution, many networks are built up by equipping standard computers with a network interface adapter at a standard peripheral interface. PCI Express is by far the most frequently used interface. Such NI adapters either connect to a Gigabit Ethernet, 10 Gigabit Ethernet or Inifiniband network [90], or to one of the few proprietary networks as Extoll, Quadrics [88] or Myrinet [87]. In their functionality, they are very similar to the Cray SeaStar NIC (see Section 2.6.3), for example.

3 Improving Device to Processor Communication

This chapter analyzes device to processor communication in HyperTransport based direct network NUMAs. Compared with processor to device communication, the device to processor direction suffers from the following problems:

- While a processor can directly communicate to a device, there is no mechanism for a device to directly notify a thread from a device in an efficient way.
- Section 3.1 shows that data transport to the processor using PIO reads has a much worse performance than PIO writes. The latency of a data transport using DMA is also not optimal, as this mechanism involves slow DRAM accesses.

Besides these inefficiencies, memory and I/O bottlenecks further increase the impact of communication latency between device and processor. A tighter coupling of device and processors is thus required. This tighter coupling must be carried out on a physical level, e.g. by system-on-chip integration. As well, a closer coupling on the protocol level is required.

This chapter focuses on the evaluation of coherent devices, i.e. devices that take part in the cache coherence protocol. Only little research has been carried out in this area in the past, so that the evaluation in this thesis is a major contribution to the scientific community. Another significant contribution is the concept of the *transfer cache*, which is being developed in this chapter.

The remainder of this chapter is organized as follows: The classical PIO access to a device is analyzed in Section 3.1. Section 3.2 will give an overview of the design space for improvements. Section 3.3 discusses memory and interconnect bottlenecks, and shows why

an on-chip integration of latency sensitive devices is necessary. The design space for coherent devices is analyzed in Section 3.4. Section 3.5 examines the performance of these devices, while Section 3.6 presents the transfer cache, a caching solution for non-coherent devices.

3.1 HyperTransport Devices and Accelerators

The various offsprings of the PCI protocol have for a long time been the standard for connecting devices, including accelerators and network interface controllers. After the PCI and PCI-X protocols, PCI Express is the currently predominating protocol.

Usually, peripheral devices are connected to the processor over one or more bridges, which may be implemented in separate chips, called the chipset. AMD Opteron processors currently provide the potentially best connection to devices, as they integrate the northbridge functionality into the processor chip (see Figure 3-1). Thus, devices and accelerators may be directly connected to the processor over a HyperTransport link. This decreases the number of crossed chip boundaries to the minimum. Additionally, it avoids the latency that is introduced by the HT-to-PCI Express bridge, which is mainly caused by protocol conversions and synchronization between the different clock domains.

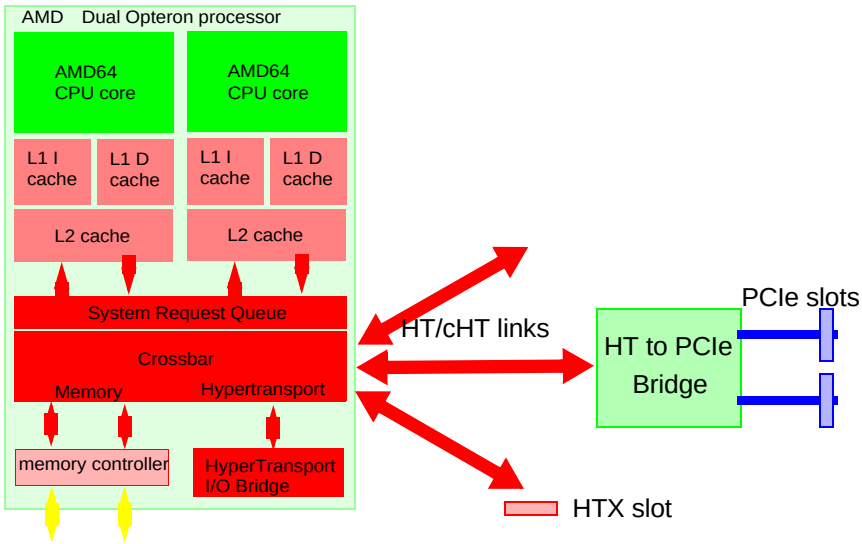


Figure 3-1. Comparison of HTX and PCI Express connections to the processor

Opteron processors can use up to four HT links for cache-coherent communication between different processors in multiprocessor systems. In this case, they use the cache-coherent HT protocol, which is not part of the public HT specification, but AMD proprietary and confidential. To connect devices or other bridges to the processor, the respective HT links are configured to be non-coherent and thus use the open HyperTransport specification. In this case, the HT link within the processor has to translate accesses from the noncoherent domain into the coherent domain and vice versa. By doing so, every such noncoherent HT link has the functionality of an I/O bridge. Due to the similarity of non-coherent and coherent HT protocols, protocol conversion and synchronization overhead can be minimized.

Thus, the direct connection of HyperTransport devices and accelerators to an Opteron processor, also referred to as direct connect architecture (DCA), is the best possible solution to connect to the processor via an I/O bus. The following sections will analyze the HT connection using a theoretical model, while Section 4.1 will give details about the physical implementation.

3.1.1 The HyperTransport Protocol

HyperTransport is a packet-based communication protocol for data transfer. There are three versions of HyperTransport: HT 1.05 has been developed in 2001, and was updated by HT 2.0 in 2004. In April 2006, HT 3.0 [47] has been defined as the next successor. Current Opteron processors follow the HT 2.0b specification [46], HT 3.0 devices or systems are not available yet. Therefore this work focuses on the implementation of an HT 2.0b device. Additionally to the HyperTransport specification, the precise behavior and in particular the initialization of HT devices in Opteron based systems is specified by AMD [22].

A HyperTransport link consists of two sets of unidirectional signals. Each set can be distinguished into three signal types: CAD (command, address, data), CTL (control) and CLK (clock). The CAD lines are used to transport command and data packets, while the CTL line distinguishes between command and data packets on the CAD lines. The HT protocol supports CAD buses with a width of 2, 4, 8, 16 or 32 bit. The width of the CAD bus is usually called the width of the HT link. If more than 8 CAD lines are used per link and direction, every group of 8 signals has its own CLK signal. These groups of signals are synchronously transmitted with the source associated CLK signal. This means that one CLK and its associated group of CAD signals must be routed with equal length traces in order to minimize skew. The data transferred on the CAD bus is 32bit aligned, independently of the bus width. All transferred packets have at least a size of one doubleword, i.e. 32bit. HT 2.0 allows frequencies from 200MHz to 1.4GHz. Current Opteron processors use link widths of 16 bit and frequencies of up to 1GHz. In Opteron systems, all devices start at power up of the system with 200MHz and 8bit wide links. The BIOS checks the capabilities of all devices

by accessing the HT register space of each device, and sets new values for frequency and width for every link according to the capabilities of the two devices that share the link. After that, it forces a re-initialization of all HT devices to establish the new parameters.

HyperTransport topologies consist of three different device types, which are distinguished by their connection to other HT devices (see Figure 3-2). Generally, HyperTransport devices are connected in chains. There can be up to 32 devices in one single chain. Different chains can be connected with each other by HyperTransport bridges. The top of a chain is always a bridge. Caves have a single link, thus they form the lower end of an HT chain. Tunnels have two links and are connected at least with the upstream link with one device, or with both links to different devices.

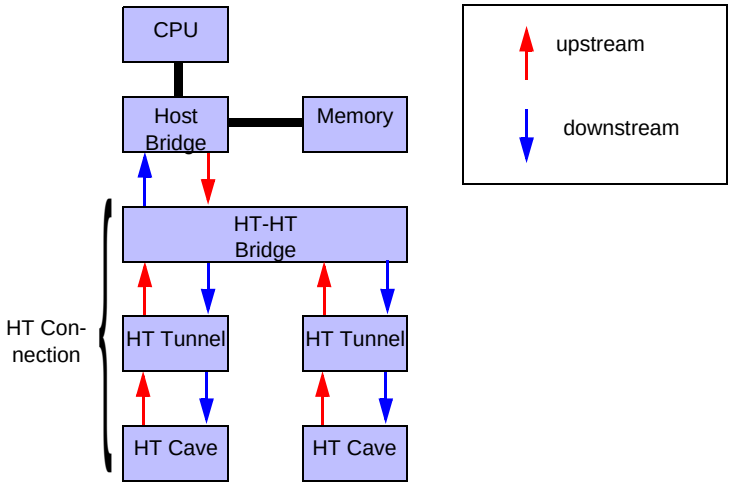


Figure 3-2. HyperTransport topology [51]

In order to decouple response from their requests, the packets are transferred in split phase transactions. This basic function is shown in Figure 3-3 with an example of read and write operations. A transfer always starts with a control packet. Three types of control packets can be distinguished: information, request, and response packets. Information packets are used for flow control and synchronization. Request packets initiate a transaction. Response packets contain the answer to a corresponding request. Control packets have a size of 4 or 8 bytes or, if they use addresses of 64bits instead of 40bit addresses, the extended format with a size of 12 bytes. If a transfer contains payload data, the next data packet which is sent on the link belongs to this packet. A data packet can have a maximum size of up to 64 bytes. Sending other control packets during a stream of data packets at every 32 bit boundary is

allowed, but only if this control packet is not followed by data. Otherwise it could not be possible to determine which control packet the data belongs to. This mechanism makes it possible to send urgent control packets with priority.

Packets travel in different virtual channels in order to avoid deadlocks. Within these channels, all data packets move along with the control packets. The virtual channels are classified into three sets: posted requests, non-posted requests, and responses. Posted requests do not get a response packet from the receiver. Non-posted requests always need a response to complete the outstanding transaction. However, these sets are not totally independent of each other, as there is the option to order non-posted requests and responses in relation to posted packets on a packet by packet basis.

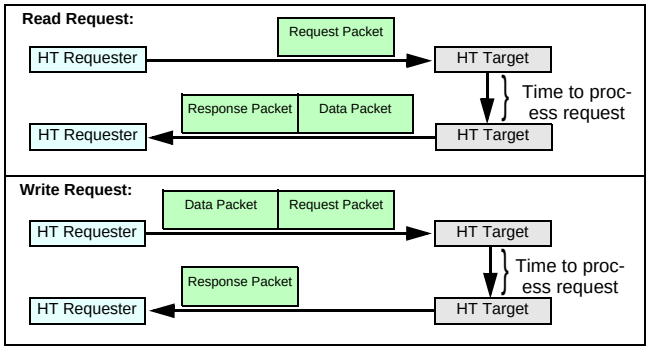


Figure 3-3. HyperTransport read and write request packet flow

Comparison with PCI Express. Brian Holden [45] [50] showed that the HyperTransport protocol offers significantly lower latencies than PCI-Express. This is due to the fact that PCI-Express uses a small number of high-speed serial lines, instead of a larger number of lines with reduced frequencies as HT. This high-speed serialization and de-serialization process together with DC-free 8b/10b coding generates significant latency.

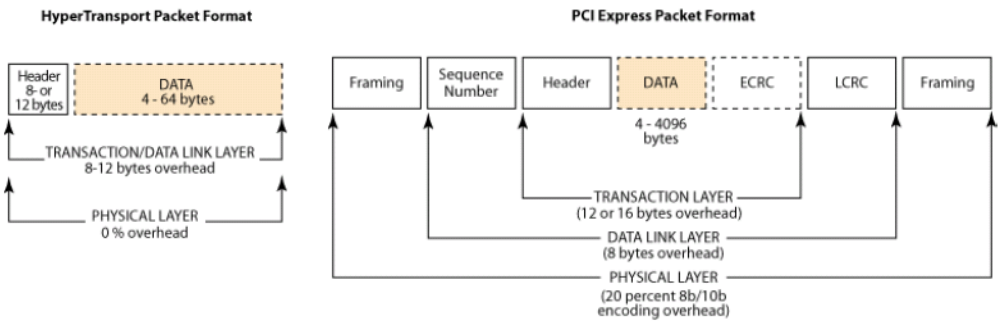


Figure 3-4. HyperTransport and PCI Express packet formats [49]

3.1.2 I/O in HTX Systems

Memory types. The AMD 64bit architecture specification [20] differentiates memory into six subtypes (see Figure 3-5). The classical I/O memory is of type uncached (UC), having the highest ordering restrictions for both reads and writes. “Reads from, and writes to, UC memory are not cacheable. Reads from UC memory cannot be speculative. Write-combining to UC memory is not allowed. Reads from UC memory cause the write buffers to be written to memory and invalidated.” The second type of memory that is often used for I/O memory is write-combining memory (WC). It has a more relaxed ordering scheme, reads do not automatically cause the write buffers to be written out. It further improves write performance by combining stores to the same memory block in a buffer, so that they can be written out on the interconnect in a single access. There are several occasions when the buffer will be written back, the most important ones being writes to WC memory outside of the memory block of the buffer, and UC memory reads. Reads from WC memory can be speculative.

Main memory is usually cacheable memory of type write-back (WB), allowing speculative reads. There are no ordering constraints between different reads. Reads may also pass write accesses, if not destined to the same address. Stores of the processor will be written to the cache. Writes to the physical memory only occur if a modified cacheline is evicted. In contrast, writes in a write-through (WT) memory will update the main memory always, as well as the cacheline. Allocation of new cachelines does not occur for WT writes. Write-protected (WP) memory is the third type of cacheable memory.

All memory types have in common that writes are committed only in order.

Memory Access Allowed		Memory Type				
		UC/CD	WC	WP	WT	WB
Read	Out-of-Order	no	yes	yes	yes	yes
	Speculative	no	yes	yes	yes	yes
	Reorder Before Write	no	yes	yes	yes	yes
Write	Out-of-Order	no	yes	no	no	no
	Speculative	no	no	no	no	no
	Buffering	no	yes	yes	yes	yes
	Combining[1]	no	yes	no	yes	yes

Note:
1. *Write-combining buffers are separate from write buffers.*

Figure 3-5. Memory accesses and memory types in the AMD 64bit architecture [20]

3.1.3 Ordering in PIO

Ordering in interconnects is a key issue, as it can directly impact the performance of the interconnect. The following paragraphs detail ordering for PIO accesses to memory-mapped I/O devices. Any processor in an Opteron-based x86 system may issue write and read requests to any device outside of the coherent HT fabric, thus including any non-coherent HT devices. The immediate destination for such requests is the particular I/O bridge that connects to the device. The I/O bridge then has the responsibility to forward the request to the device. Different ordering mechanisms are used for the coherent and the noncoherent HT links.

Ordering in noncoherent HT. In the nHT domain, ordering can be established quite easily. By default, ordering is performed as follows: Packets within every virtual channel (VC) are ordered among each other. Also, packets of the non-posted virtual channel may not pass packets of the posted channel. As read requests and non-posted write requests travel in the non-posted VC, while posted writes travel in the posted channel, this has the following implications:

- Read requests are ordered among each other.
- Non-posted write requests are ordered among each other.
- Posted write requests are ordered among each other.
- Read requests cannot pass any write request.

- Non-posted write requests cannot pass posted write request.
- Posted write requests may pass both read requests and non-posted write requests.

In system, non-posted writes can be observed in the configuration phase of the system, i.e. during the boot up phase. In normal operation, posted writes are used. However, the north-bridges may be configured to support legacy ISA devices. The use of posted writes in combination with non-posted reads on the HT bus leads to an ordering scheme which matches the ordering requirements of the UC memory type of the AMD 64bit architecture specification, except for the fact that nHT imposes ordering among read requests.

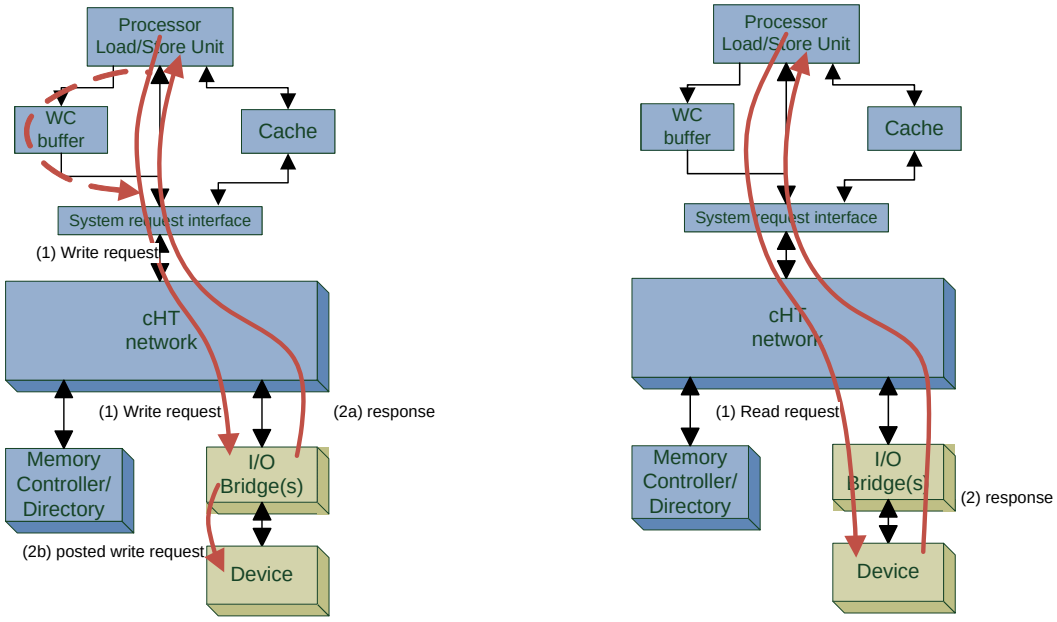


Figure 3-6. Device access using memory-mapped I/O in Opteron systems

Ordering in the coherent HT. How exactly routing of I/O requests is done in cHT is out of the scope of this thesis, mainly because the cHT protocol is confidential. Nevertheless, it is sufficient to say that cHT does not apply ordering constraints between VCs in the fabric, for the sake of better performance of the interconnect. Also, the nHT ordering solution is useful for the nHT chain topology, but may not work in the switched cHT fabric supporting arbitrary topologies.

Instead, ordering in the cHT fabric can only be established using an end-to-end protocol. Therefore, all requests in cHT have to generate a response, including requests in the posted VC. If a processor wants to ensure that a request *r1* should commit before request *r2*, it will have to wait for the response of *r1* before issuing *r2*. Theoretically, this is a much more promising way to establish ordering, as it allows processors to order only those accesses that really require ordering. The disadvantage is, however, that a series of ordered accesses cannot be simply streamed out as it is possible in nHT. Instead, every access can only be issued to the fabric if the previous response arrived, which may decrease the bandwidth available to the device significantly.

Figure 3-6 shows how cHT and nHT ordering play together for a read and a posted write request. End-to-end ordering is performed between processors and I/O bridges within the cHT domain. In the case of a read, end-to-end ordering is performed between processor and the device.

3.1.4 Ordering PIO Write Requests

The performance impact of ordering for a sequence of ordered accesses from a processor to a bridge in the cHT domain can be calculated. All calculations assume virtual cut through routing, which implies that the response to a packet can be generated just after receiving the header. Packet header sizes are neglected. The system parameters as introduced in Section 1.5 are used, the relevant ones are repeated in Figure 3-7:

Name	Abbrev.	Latency in ns	
Response processing delay	t_{rpr}	4	Time to process a read response containing data
Link delay	t_{link}	21	One-way latency of HT links
Xbar delay	t_{xbar}	4	Delay of HT Switch

Figure 3-7. System parameters

The round-trip latency, which is the time it takes after starting to send a packet till the response is received, is:

$$t_{rtl} = 2hops(t_{xbar} + t_{link}) + t_{xbar} + t_{rpr}$$

where *hops* is the number of hops between processor and bridge. The time in HT cycles to inject a packet into the HT network depends on the packet size s_p and the link width per clock cycle *w*:

$$t_{inj} = \frac{s_p}{w}$$

In ordered accesses, the next packet can be injected at time $t_{gap} = max(t_{inj}, t_{rtl})$ after the previous packet. Obviously, there is only an ordering impact on performance if $t_{inj} \leq t_{rtl}$. In this case, the effective send bandwidth of the link used in the transfer is decreased by a factor of t_{inj}/t_{rtl}:

$$BW_{eff} = \frac{t_{inj}}{t_{rtl}} BW_{inj}$$

Assumed that the processor does not buffer requests that it committed to the fabric[1], the time the processor is occupied with the sending process is increased by the same factor:

$$t_{eff} = \frac{t_{rtl}}{t_{inj}} t_{inj} = t_{rtl}$$

Figure 3-8 shows the effect for BW_{eff} and t_{eff} for different packet sizes and different number of hops between processor and bridge. The performance decrease for streams of small packet sizes is dramatic. For a direct connection, only packets with a size of 32 byte or more can fully saturate the link. The direct connection performs much better than the one- and two hop configurations, as all communication is done within one chip. For reference, the diagram also shows how 128byte packets would perform, however, this packet size is not supported by the HyperTransport protocol.

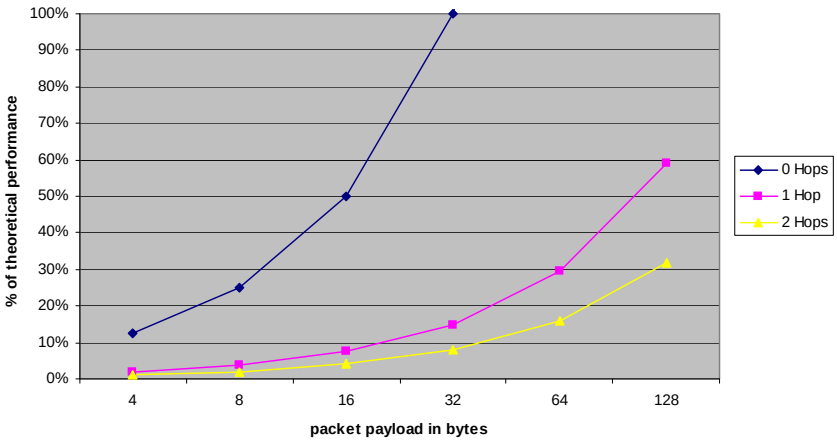


Figure 3-8. Relative performance for streams of different packet sizes

1. As described later in this chapter, this assumption is valid for current Opteron processors.

Impact of bridge behavior. Up till now, the response processing delay t_{rpr} was considered to be a fixed, small number. This behavior corresponds to a queue in the bridge of indefinite size. In reality, t_{rpr} may be influenced by the following issues:

- The device may temporarily not be capable to consume or process the packet.
- The link speed between bridge and device may be smaller than the one in the cHT fabric.
- Background traffic, i.e. packets that are send to the bridge by other processors.
- Buffer space in the bridge. Buffers within the bridge can generally reduce the impact of above problems.

Device and application behavior determine whether a device may temporarily not be capable to process packets. There is no direct influence of the interconnect, however, buffers anywhere in the path between bridge and device may help to relieve temporary problems. Of course, the maximum bandwidth between a single processor and the device cannot be increased, but the processor could be done with the sending process earlier, thus decreasing the negative impact on t_{eff}. In essence, such a buffer would decouple processor and device during a data transfer. The buffer must be located after the ordering point in the bridge. t_{rpr} is only improved if the response is sent immediately, buffering within the write-request-response loop would not have an impact on t_{rpr} as long as a strict ordering scheme of writes is used.

If the link between bridge and device is slower than the link between processor and bridge, t_{rpr} may also increase. In current systems, 1GHz links in the cHT domain and 400MHz links to the device are popular. Large enough buffers within the bridge could thus significantly reduce the impact on processor execution time for writes.

To find out about buffer sizes, the following experiment has been performed in an 9th generation Opteron-based system with a directly connected nHT device: the device is configured to not react on write request, but to leave them in the HT queues, so that no credits are send back. Then a sequence of write requests is issued to different address locations within the BAR address range of the device. The processor stalled when trying to do the 16th write. At the same time, the HT queue in the device contained 14 entries. This means, that there is virtually no queue space in the bridge. As a result, it can be noted that buffer space should be introduced into the bridge to improve processor execution times for I/O accesses.

Further improvements in performance for write requests with smaller sizes or from processors farther away from the bridge could probably only be made if ordering is handled differently, or if ordering is weakened.

3.1.5 Ordering PIO Read Requests

I/O read accesses suffer from the same problem as writes. However they are even more affected, as the ordering point is the device, and thus the round-trip latency of a read access is even higher. A theoretical analysis of this problem, as performed in the previous section for write access, shall be waived. However, the performance has been measured in the real system and is described in Chapter 4.1.1.

3.1.6 Potential Incremental Solutions

The currently used workaround to cope with the bad PIO performance is to reduce PIO accesses as possible. Instead data is transferred using DMA. A DMA mechanism also has the advantage that main memory can be used as buffer space, which is significantly larger and better scalable in size than device memory. Also, main memory is cacheable.

To raise the PIO read performance, a processor could read larger chunks from device memory that is marked prefetchable. In analogy to the write buffers, a read buffer could be used to read cacheline sized blocks. This could improve PIO read performance significantly.

3.2 The Space of Analysis

This section analyzes the design space for devices at the coherent processor interface or below. Two key issues have to be solved in this analysis: which data is sensitive to latency, and how can a low latency be reached? The latency depends on how data is buffered on the path between producer and consumer. The location and size of buffers thus is the second important topic.

3.2.1 Latency-Sensitive Data

Communication from a device to a process can be considered to be a stream of data. There is at least one stream from the device to every process that uses the device. An important consideration is how these streams of data are organized, and how and where the streams can be buffered. The previous chapter showed that queues are a very important form of communication between NIC and processor. Thus, in most cases the communication streams will be organized as queues. However, non-queue-based data streams, as in the put/get communication mechanism, can also be observed.

An important question is: which data streams or parts of data streams are sensitive to latency and thus require an improvement in performance? Two types of latency can be dis-

tinguished: the latency of the data-transport from device to processor when the processor is already waiting for it, either actively using polling or passively as in multithreaded architectures. The other type of latency is the read access latency to data that has been made available by the device.

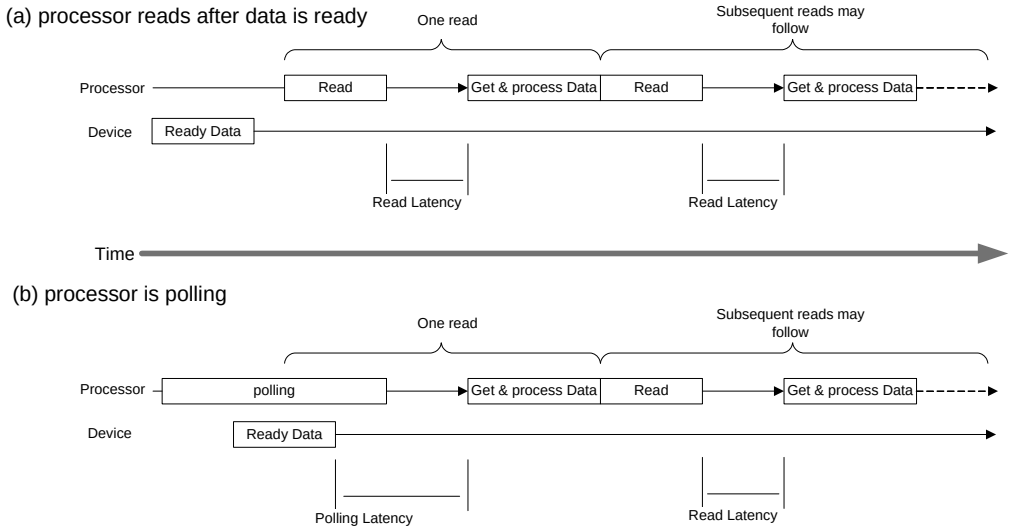


Figure 3-9. Latencies

The overall device to processor latency is a so called startup latency, as it occurs only once for every block of data (see Figure 3-9). For large blocks of data, it composes only a fraction of the total latency, and thus can be neglected. For small data blocks, this latency becomes the most significant latency in the transfer.

The latency of processor read accesses to data that already has been made available by the device is important because this latency increases processor execution time and thus decreases processor throughput. If DMA accesses are performed by the device, the read access latency equals the physical main memory access latency. The figure shows that the use of larger data packets to transport data to the processor has the potential to decrease the overall latency, as fewer memory requests are required to obtain a fixed sized block of data.

In contrast to the device to processor latency, the read access latency can be hidden. The prefetch engines of processors can detect access patterns to memory when they have a fixed stride, and in particular if they have a unit stride. Start-up latency occurs when the first cach-

elines are read until the prefetch logic starts prefetching. Thus, latency can be hidden for larger objects, but small data objects to which reads cannot be predicted do not profit at all.

An intermediate conclusion is that both the device to processor latency and the processor read latency should be minimized for small data objects.

In a modern NIC like Extoll, the relevant data structures that profit from a latency reduction are:

- Descriptor and notification queues. Entries in these queues are typically not larger than one cacheline.
- Fast and small grain communication mechanisms, as for example small grain send/receive as used in the Extoll Ultra mechanism. While this mechanism is queue-based, non queue based mechanisms as a fine grain put/get mechanism could profit as well.

In applications that are sensitive to end-to-end communication latency between nodes in the system, queues will typically be empty or almost empty, as processors are waiting for entries to be placed in the queue. In this case, device to processor latency is critical. If throughput is more important than end-to-end communication latency, the read latency should be minimized.

The fact that only small data objects must have low latency, and that these will typically be consumed soon from the queue is an excellent basis for optimizations. It opens up the possibility to implement fast data buffering solutions without excessive hardware overhead.

3.2.2 Buffering

To decouple the execution of processor and device, there must be buffering capabilities in between the communication partners. The design space of such buffers is shown in Figure 3-10. With such buffers, the latency of a communication between device and processor depends on the access latency of the memory technology, as well as on the logical and physical location of the buffer.

The **logical location** determines which address or register space the buffer belongs to. Buffers that are logically placed in the device are accessible from the processor using memory accesses to the memory-mapped device. Besides the classical I/O device memory, a coherent device may exhibit the memory range as coherently cacheable.

A system’s physical main memory is not only the largest memory in a system, it is also the one that scales best. A logical placement in the system’s physical main memory thus has the big advantage that it allows buffered streams to inexpensively grow in size. This is particularly useful for NICs that are using lossless networks, as a full receive side buffer in one of the NICs will cause congestion in the network. Also, data structures for virtualized

devices supporting the simultaneous direct user-level access of hundreds or thousands of threads may need the size and scalability of physical main memory.

For the **physical implementation**, two options can be distinguished: implementations in small-capacity, fast memory technologies as SRAM, or ones in higher latency, high capacity memory technologies as DRAM. Physical main memory is slow DRAM, all other buffer implementations use faster memory. However, slower memory technologies typically offer more capacity than faster ones.

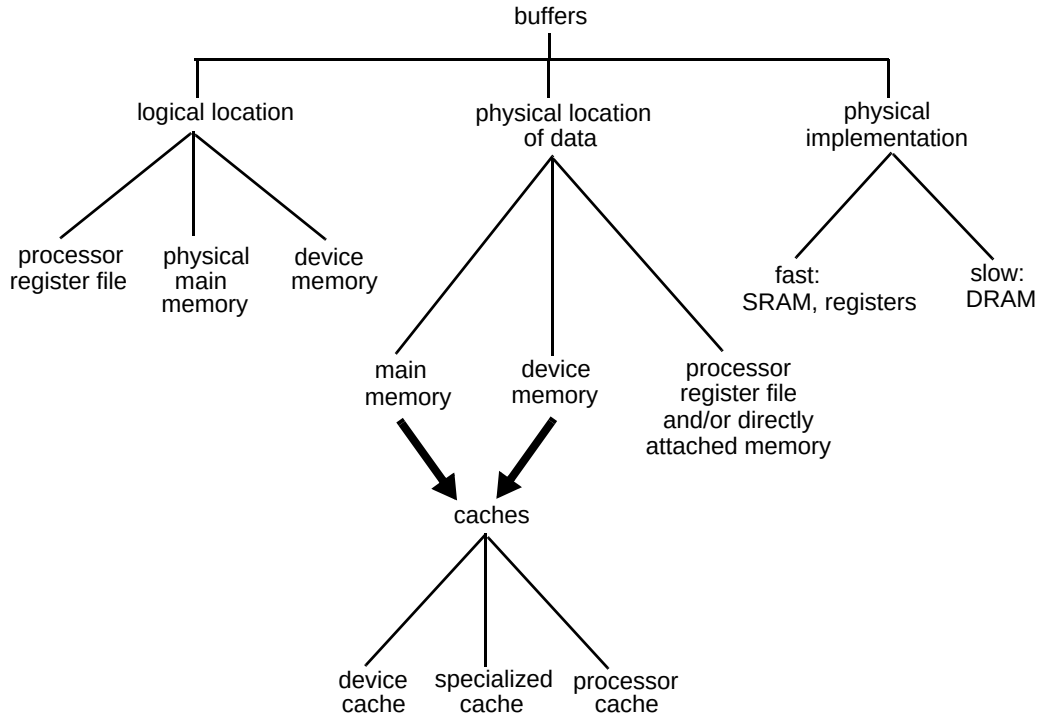


Figure 3-10. Buffer design space

The **physical location of buffered data** may be anywhere on the path between device and processor. However, the logical location may reduce the number of choices. All logical locations have their natural physical locations. For the logical locations in physical main memory or in a coherent device memory, caching is possible. Caches may improve performance as fast memory technologies can be used due to their relatively small size. Also, a good placement of caches may improve access latency. In the best case, the use of caches

avoids any DRAM accesses in the timing critical path between device and processor, and performs DRAM accesses only for victim writebacks, or if queues grow large.

The cache can be implemented in the device that provides data to the requesting processors via a direct cache-to-cache transfer. Another option is to stream data into the processor's cache that is likely the one that will work in that data. Chapter 5 will give an outlook about this approach. A third possibility is the use of other caches, as e.g. dedicated message caches or the transfer cache proposed in this work.

3.2.3 Feasible Solutions

Based on the previous two subsections, the following conclusions can be drawn that guide the development of improved mechanisms:

- Queue-based communication must be supported. Other mechanisms can and should be supported as well.
- Support for efficient, low latency data transport between device and processor if the queues are filled sparsely.
- At the same time, queues should be allowed to grow large, which means their home should be physical main memory.

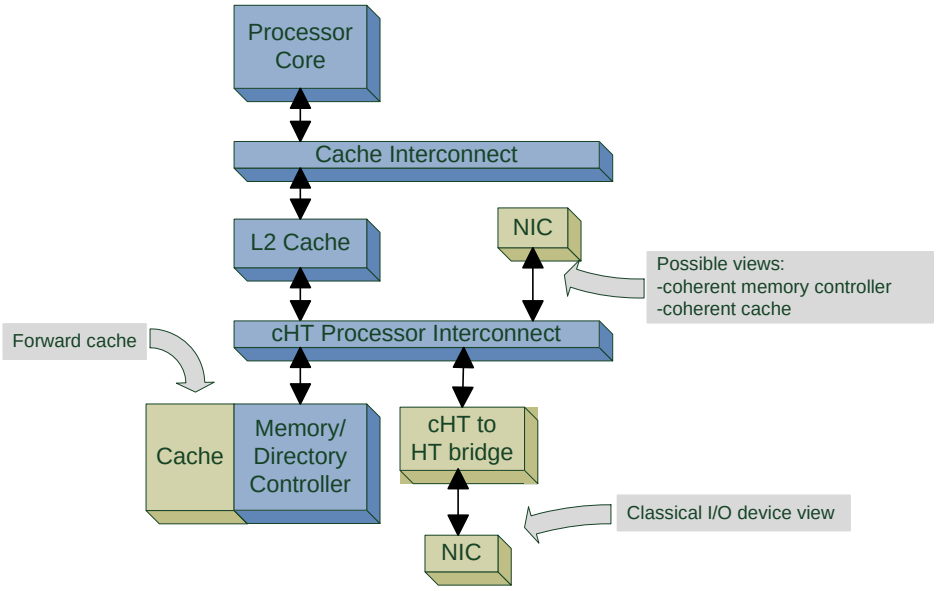


Figure 3-11. NIC locations

Two suggested architectures place a coherent device at the coherent processor interconnect (see Figure 3-11). The first one is a device that acts as a coherent memory controller, i.e. queues are located in device memory. The second suggestion is a device with a device cache. The logical home for queues is physical main memory. A third suggestion also homes queues in physical main memory and adds a special transfer cache to the memory controller. As DMA is currently the best method to transfer data from device to processor, all solutions have to compare with a corresponding DMA solution.

The fourth potential solution is to stream data directly into the processors caches. This approach exhibits some problems that are not present in the other solutions. One major problem that occurs in multiprocessor systems is how the target cache can be identified. Chapter 5 provides an outlook on direct cache access mechanisms.

3.3 Memory and Interconnect Bottlenecks

The memory bottleneck is well known [7]. It is less commonly known that the interconnect between chips in computers has become a similar bottleneck. This chapter demonstrates the impact on small-scale NUMA systems, and concludes with a recommendation for tighter system-on-chip (SOC) integration.

Memory and I/O bottlenecks. Gordon Moore's prediction [106] that the number of transistors that can be placed economically onto a single chip doubles every two years still holds true. This leads to a continuous increase in processor performance. Figure 3-12 shows that the theoretical peak performance in million instructions per second (MIPS) has increased thousand fold within the last 15 years. This theoretical peak performance is not a good measure for the real systems performance, as it is influenced by the memory and I/O subsystem as well as by the utilizable parallelism of applications. It is, however, a good measure to show the capabilities of the processor core itself.

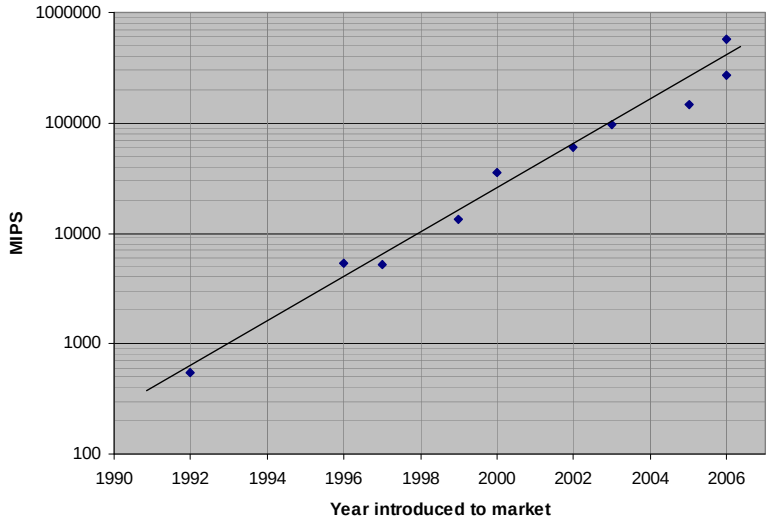


Figure 3-12. The development of processor speeds of x86 processors

In contrast, I/O buses have doubled their performance only every 36 months [103]. Figure 3-13 shows that the maximum bandwidths of DRAM memories and I/O buses[1] have increased by a factor of ten within the last ten years. Compared to the increase in processor speed within the same time period, the performance gap increased tenfold!

Besides bandwidth, latency is another important performance criterion for memories and I/O buses. The figure shows that the random access latency increases similarly to the bandwidth, although the increase is less stable. The development of the latency of a direct chip-to-chip I/O link is more difficult to trace. Usually, target latencies are not part of the specification, but are considered implementation details. The latency of a direct link of a chip-to-chip interconnect is composed of the following parts:

- The time for the physical transmission of signals between two chips. As this time only depends on the distance of the chips, it remains constant.

1. Only PCI-derivatives have been used, as they are the de-facto standard I/O bus.

- Processing on transmit and receive side scales with technology. There is no room for architectural improvements to reduce latency. Instead, the pin count limitation of processors led to a development of I/O interfaces that use high-speed serial transmission. This adds even more latency, caused by the de-/serialization, 8b/10b encoding, scrambling, error correction and detection steps that have to be performed.

Thus, the scalability of the link latency is very limited with current technologies. Technologies as proximity communication [107] might resolve such restrictions in the future.

The processor Interconnect. An all-embracing comparison of processor interconnect performance is difficult, as these interconnects are usually confidential. Generally, these interconnects are subject to the same technological conditions as I/O interconnects. A case in point is a comparison of the latencies in processor clock cycles for the Motorola 68030 processor, released in 1987, and one of today's Opteron processors. The 68030 accessed DRAM memory over a bus interconnect. The latency for a DRAM read operation consisted of 3 processor clock cycles for the bus arbitration, and additional 5 clock cycles until the DRAM delivered the requested data. In contrast, a point-to-point HT link has a latency of at least 60 3GHz Opteron clock cycles, the DRAM access accounts for at least 120 clock cycles. To send the response back to the processor, additional 60 clock cycles pass.

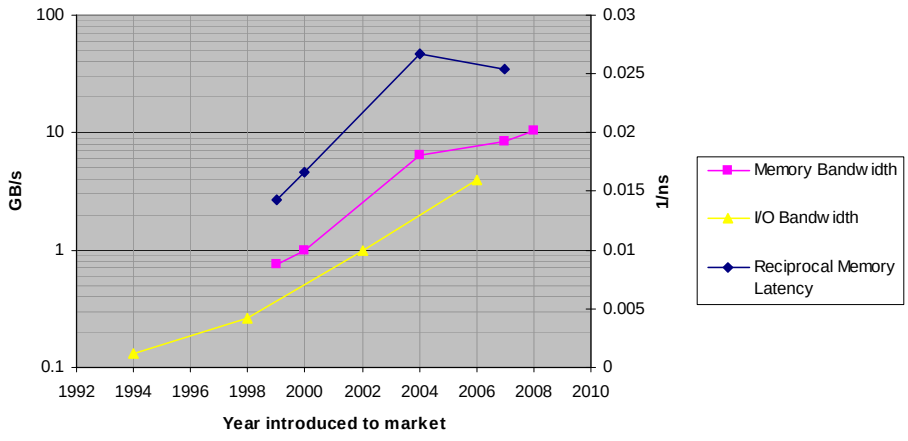


Figure 3-13. The development of DRAM memory and I/O bus speeds

Memory and I/O hierarchies. Memory technologies that are currently being used allow the integration of fast SRAM memory on CMOS logic processes. The downside is that

SRAM cells consume a large real estate. Thus, SRAMs have only small storage capacities. DRAM, on the other hand, can be integrated much denser, but access latencies are high. Also, few solutions exist to integrate DRAM into logic processes. The solution to this is the use of a memory hierarchy, as explained previously in this work.

Figure 3-14 shows the latency of memory requests, separated into the impact of memory and interconnect technology. It shows that the performance difference in terms of latency between the hierarchical levels are of the same magnitude for both memory and interconnect technology, with the exception of the lowest level in the memory hierarchy.

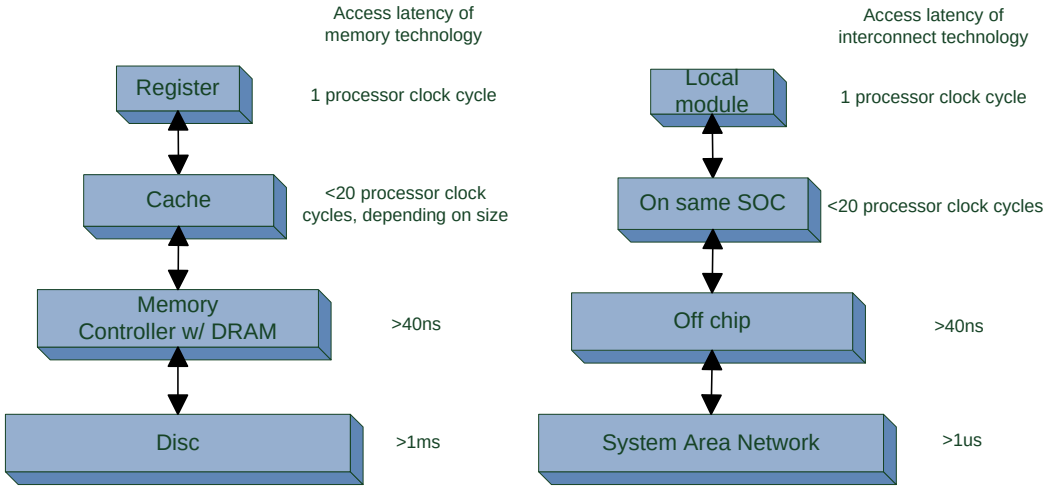


Figure 3-14. Read access latency, depending on memory and interconnect technology

The usage of a memory hierarchy aims to reduce both interconnect and memory latency by placing fast caches close to the processors. This works well in particular if memory accesses are predictable and thus may be prefetched.

Figure 3-15 displays the access latency of DRAM memory in dependence of the distance between requestor and memory in an HT network in hops. Cache coherence mechanisms are not considered, the diagram simply shows the latency of the interconnect and the latency of the DRAM access itself. The access latency of the DRAM is the limiting factor only if

the memory request's source is on the same chip. As soon as chip boundaries have to be crossed, the interconnect latency is larger than the DRAM access latency.

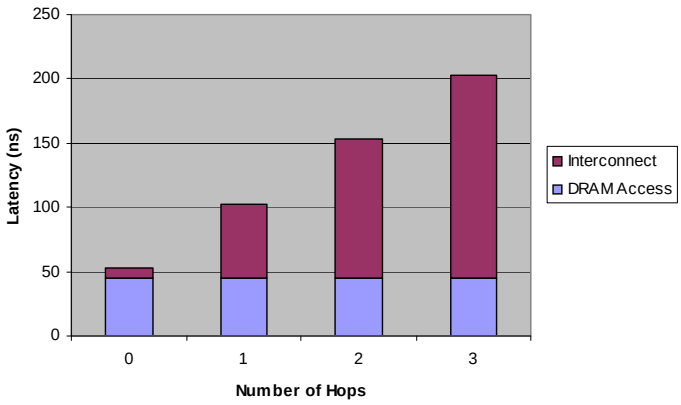


Figure 3-15. Overall DRAM read access latency in Opteron system relation to number of hops to take

3.3.1 Influence of the Cache Coherence Protocol

If the coherent fabric that connects processors and memories spans over multiple chips, the interconnect latency also impacts broadcast based cache coherence protocols. Figure 3-16 shows the latencies that are observed for a processor read access on physically local memory in the NUMA-type of system that is assumed in this work. Even in the two processor configuration, which is the smallest configuration where probes have to cross chip boundaries, the latency of the probing over the interconnect is worse than the DRAM latency and thus affects overall latency slightly. For larger configurations, the probing latency is clearly dominating the overall memory access latency.

In a three hop coherence protocol, as analyzed here, the latency of a memory access to remote memory is increased by the time the request travels to the remote memory controllers. In two-hop protocols, this does not happen, but the latency of a read to local memory will be about the same as in the three hop protocol, as the hop from a processor to a local memory controller has a relatively low latency.

Thus, an effective solution for both types of protocols must take the broadcasting of probes out of the critical path of the memory access. Directories are a way to do so by decreasing

the number of probes that have to be sent out. In the best case, probing has not to occur at all for a memory access, thus decreasing the overall latency of the memory access drastically for a 4 processor system. A directory also decreases traffic on the interconnect, and is less vulnerable to congestion in the interconnect.

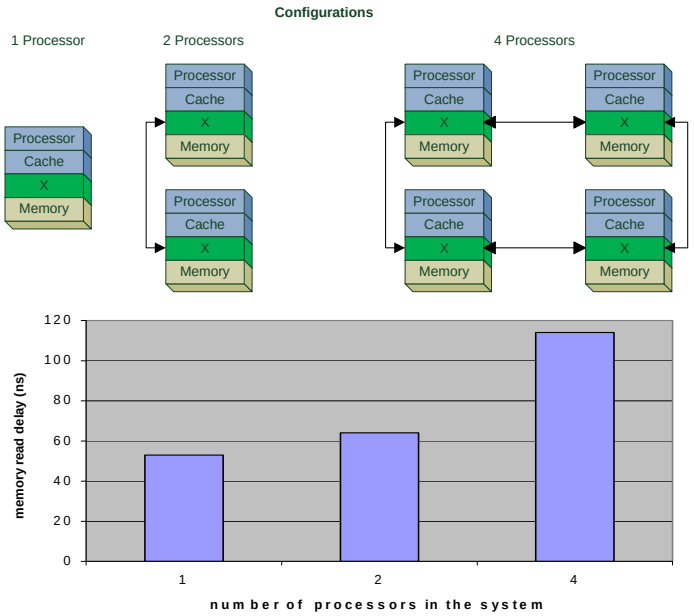


Figure 3-16. Latency of a read operation on physically local memory with broadcast based coherence

The theoretical results from Figure 3-16 can also be observed in the real system. An evaluation of the memory bandwidth on a dual-socket Opteron system using the STREAM benchmark [110] has been performed. The dual processor performance has been measured by starting one STREAM process and using the “numa tools” [109] to bind the process and its memory to the same node. The single processor performance has been measured by removing one processor. This evaluation showed that the memory bandwidth that is reported by the benchmark drops by 5% to 7% for the two processor system.

Thus, the usage of a directory to reduce memory latencies becomes beneficial even for small-scale NUMA systems.

Also, a tighter integration of the system will reduce the radius of the coherent fabric, and thus generally decrease the latency of coherence protocols.

3.3.2 Summary

The best possible solution to reduce the communication latency is the integration of all performance critical components of a system into a single system-on-chip (SOC). Off-chip communication should be avoided just as DRAM accesses are avoided. The Sun T2 is such a system that leads the way for a tight physical integration of processors, memory, and NIC.

If multiple such SOCs are interconnected to build up a shared memory system, the communication latency impact of broadcast-based coherence protocols should be avoided even for small-scale systems. Instead, coherence on a chip-to-chip level should be maintained using directory based protocols.

Only such devices should be integrated into the SOC that are used widely and where low access latency is required. Traditionally, only network interface controllers belong to this class. However, one can imagine that FPGA or GPU coprocessors may become popular as well.

Thus, the remainder of this chapter will analyze the integration of a NIC device into such a SOC. As the transition to such a system should rather be an evolutionary process than a revolution, it will also be analyzed how mechanisms would perform in traditional systems.

Problems of SOC integration. A SOC integration does not come for free. In the following, problems are discussed that are in the way of such a solution:

- **Limited on-chip resources**. NIC buffers and directories need memory, which is expensive both in terms of silicon area and power consumption. As long as this problem is not solved by the use of new memory technologies as Z-RAM [34], it may be necessary to use NICs with lower buffer requirements.
- **Reduced Yield**. Additional functionality on a chip increases the die size. The larger the size of a die, the higher is the probability of faults on the die. This will always lead to a decreased yield.
- **Reduced Modularity**. Traditional systems allow replacing or recombining of parts of the system as needed if these parts are implemented on different chips. For example, for x86 processors several northbridges and southbridges exist that may be developed at different points in time. On a SOC, such a functional part is not a chip, but an IP block on the single die. That means that a part of the system can only be exchanged by modifying the whole system. The reduced modularity may also increase the risk: if one IP block of a SOC fails, it may turn the whole system useless.

- **Pin Limitation**. Complex designs like SOCs will usually be limited by the number of available pins. A NIC requires a number of pins for communication, these must be taken away from other components on the chip. A potential solution may be the sharing of pins for different functions. For example, one of the HyperTransport links of an Opteron could be shared, so that either the HT link can be used, or the integrated NIC.
- **Multi-SOC systems.** It is very likely that such SOCs will be used to build up larger shared memory systems. Not much will be won if this system does not decrease inter-chip communication significantly. Processor cores should preferably use local memory, and inter-chip probing must be decreased with directories for example. As well, processors should use the local NIC device for best performance. Of course, this also changes thread scheduling policies. Threads should preferably be scheduled to a “home node”.

3.4 Devices at the Coherent Interconnect

A processor interconnection network in a small-scale shared memory multiprocessor system usually distinguishes three main classes of ‘clients’ that are attached to the network.

One client is a processor core with its associated caches. Usually, the processor core and the caches share one common interface to the network, called the system request interface (SRI). An SRI may also be shared among multiple processor cores, which is in particular the case if these cores share a cache. An SRI has master functionality in the classical sense: it may issue coherent and noncoherent read and write requests, and collects the associated responses. The SRI is also target for probe requests, and must respond to these.

The coherent memory controller is the classical slave device: it responds to memory requests. In the analysis performed here, it is also responsible for generating probe requests. A cache coherence directory is, from the interconnect’s viewpoint, part of the memory controller, as it is simply a means to reduce probing traffic.

I/O bridges are the third kind of clients. Like an SRI, a bridge must be able to create coherent and noncoherent requests as a consequence of requests on the I/O interconnect side. From the coherent interconnect’s viewpoint, it is the target for memory requests to I/O devices, and the source of responses. In contrast to the coherent memory controller, it does not generate probe requests, as memory behind the bridge, and thus on the I/O bus, is non-coherent by definition. In essence, an I/O bridge’s functionality from the coherent interconnects viewpoint is a subset of the combined functionality of SRI and memory controller.

Accordingly, Figure 3-17 shows the possible views of a device that is attached to a coherent interconnect. (a) is a non-coherent device that is connected as an I/O device. The views of

a device as a memory controller or a cache are depicted in (b) and (c). A device might as well be a combination of these views.

Functionality. A coherent device may use a cache that is kept coherent by the hardware protocol. The possible benefit of such a cache in a networking device is being discussed in Section 3.4.1. However, this thesis will focus on performance improvements.

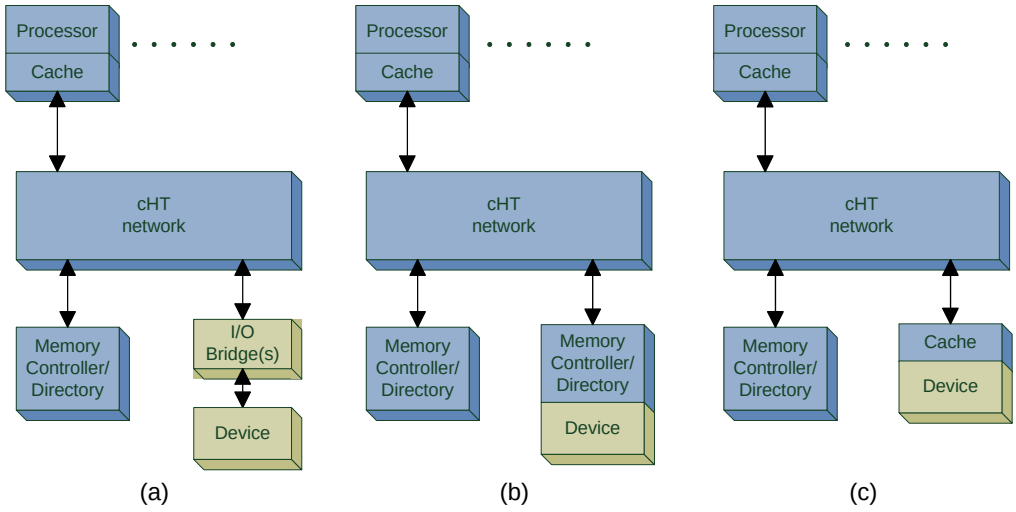


Figure 3-17. Views of a device in a coherent processor interconnection network

Performance Improvements. The coherent protocol allows other mechanisms of data buffering and transport between device and processor than pure I/O protocols. This may allow faster or more efficient data transport. In particular, the over-all latency for data transport between device and processor register may be decreased. As well, the read latency of the processor on such data may be lower, thus increasing processor throughput. An analysis of the different design choices is done in Section 3.5.

3.4.1 Devices with Coherent Caches

Processors benefit from caches because of the spatial and temporal locality of a program's memory accesses. For most programs, it can be observed that after any access to memory, there is a high probability that a nearby memory is accessed very soon. Therefore it is advantageous to fetch a larger memory block upon a load request from a processor and to store it in a cache frame. Subsequent accesses to this block will hit in the cache and thus have a much lower latency than an access that has to be forwarded to physical main mem-

ory. In the optimal case, the whole working set of an application fits completely into the cache. In this case, cachelines do not get evicted from the cache, all accesses to the memory will hit in the cache after the very first access to the respective memory block.

Besides the reduction of read latency, caches also improve the write behavior. In a write-back cache, a memory location that is repeatedly written to by a processor will just update the cache. An access to the physical main memory occurs only on an eviction of the cacheline. This saves bandwidth both on the system interconnect and the memory controller.

Devices may display a similar spatial and temporal locality for their memory accesses. However, there may be a difference in granularity of those accesses. While general purpose processors usually only support load and store operations with a size of up to 128bit, devices are optimized for their specific tasks and thus may support memory accesses that are much larger. The impact of this difference can be made clear with the example of a linear access to subsequent memory addresses. In a processor, a case in point for this behavior is the instruction cache. Here, a cache clearly improves read latency: loops in the instruction stream lead to a good temporal and spacial locality of instruction references.

A typical example of a device that is accessing memory is a NIC that is reading a memory area in order to transmit it over the network. As the NIC knows the size of the transfer beforehand, it can directly fetch the memory area using overlapping memory accesses in an optimal way. In this example, there is also no temporal locality of this data, as it will not be used by the NIC again. A caching of such data in the NIC would not introduce any benefit.

A NIC's memory accesses to other data structures show access patterns that can be improved by caching them on the device. In the case of Extoll, virtual device contexts and associated data structures that reside in main memory should be cached on the device.

Cache Coherence. The important question is how a device cache is being kept coherent with the system. Noncoherent devices cannot take part at the hardware cache coherence protocol. With noncoherent devices, coherency can only be maintained explicitly by software. If devices are part of the coherent domain, caches may be kept coherent by the hardware protocol.

The design space of device cache coherence is presented graphically in Figure 3-18. If data is not shared between device and system, it is obviously not necessary to maintain any coherency at all. However, such private data does not need to reside in system memory. Instead, it could be stored in memory directly attached to the device. This would not only result in lower memory access latencies, but also avoid the occupation of system interconnect and memory. Thus, it can generally be assumed that most of the memory a device will cache is indeed shared.

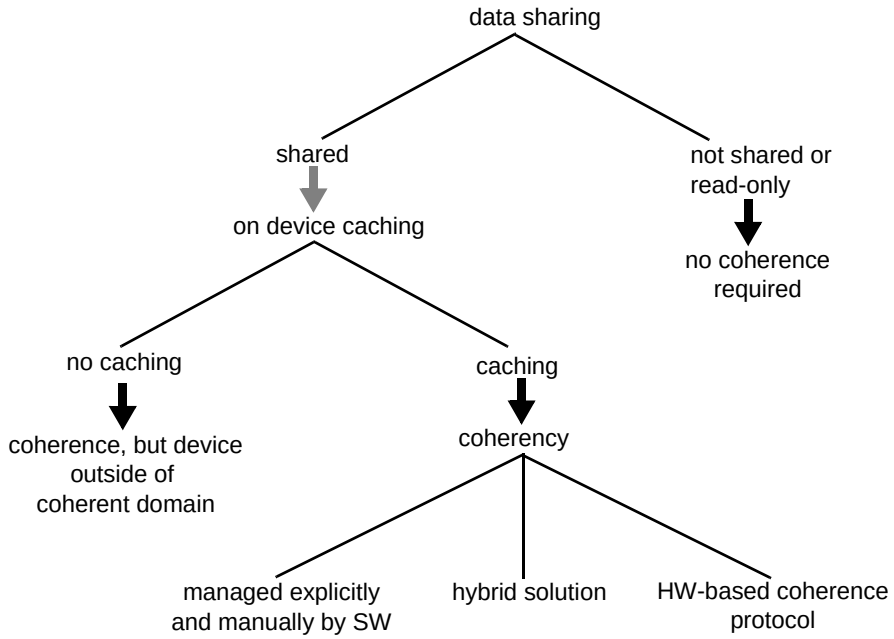


Figure 3-18. Coherence of device caches

If shared data is cached, the only solution that is possible for non-coherent devices is that coherence is managed by software on the host system, usually this is done on an API or driver level. No matter how this is implemented in detail, if a processor writes to a memory location that might be cached in a device, it has to either invalidate or update the respective cache entry in the device. If the same quality of consistency should be achieved that is used within the system, software will have to wait for a response before it can proceed. The device, on the other hand, simply has to move data out to the coherent domain if it is written by the device. An option for this are write-through caches. As a result, maintaining coherence is expensive for modified data that might be cached, but it imposes no overhead at all for all other accesses.

In contrast, broadcast based hardware cache coherence protocols do not provide a good differentiation between data that might be cached somewhere else and data that is not cached somewhere else. For all accesses, there is a certain amount of overhead introduced by the coherence protocol. However, this overhead is much smaller than with software based coherence schemes.

In comparison of both mechanisms, software based coherence performs better if the memory region that might be cached in devices is very limited. It performs better if the coherence scheme can be relaxed. Hardware based schemes are to be preferred if the memory region that might be cached is larger.

3.5 The Performance of Coherent Transfers

This section analyzes the design space for coherent devices in terms of latency improvements. The analysis focuses on the device-to-processor communication path. It compares the latencies that can be expected using coherent devices with the performance of DMA.

The main question is: how fast can a cacheline, which is the smallest granularity of data in a coherent environment, be transmitted? Such a cacheline may be of any of the data types that require a low latency transmission, as described in Chapter 3.2.2.

Figure 3-19 shows the flow diagram for device to processor communication using conventional DMA. A write access by a device is followed by a read request from a processor to the same memory block.

The timing between both accesses depends on the notification mechanism (see Section 2.3.1) that is being employed. The lowest latency can be achieved if the data that is written contains information about its validity, as valid bits for example, and the process is polling on it. In this case the processor is polling on the line in the cache. The write request by the device causes an invalidation probe to be sent to this cache. After the cacheline has been invalidated, the next processor read causes the line to be fetched from memory. The second dependency between both requests occurs at the memory controller. Here the read request is queued until the previous write request finishes. This has to be done to maintain coherence, as explained in Section 2.4.5 on page 49.

If interrupts are used to signal a new queue entry, there is a significant time interval between both accesses, caused by the delay introduced by the interrupt handling mechanisms. In this case, the overall latency of the data movement from device to processor is determined by the interrupt mechanism and can hardly be optimized using coherent data transport mechanisms. Here, another number is more interesting: the latency of the processor’s read.

Some assumptions have to be made regarding the behavior of the memory controller. A classical memory controller works on request after request. It may reorder accesses to memory internally to optimize access to the banks, but a read-after-write hazard is avoided only by forcing the write to memory before reading the address. This mechanism is employed in the SUN T2, for example. It is a very inefficient mechanism. Its performance influence is particularly bad with high background traffic, as a read request may have to

wait a significant amount of time until the previous write to the same address has been performed. But even in the absence of background traffic, this may affect latency. In these cases, this work will also present the timing for a memory controller in which such data is being forwarded to respond to the read request earlier.

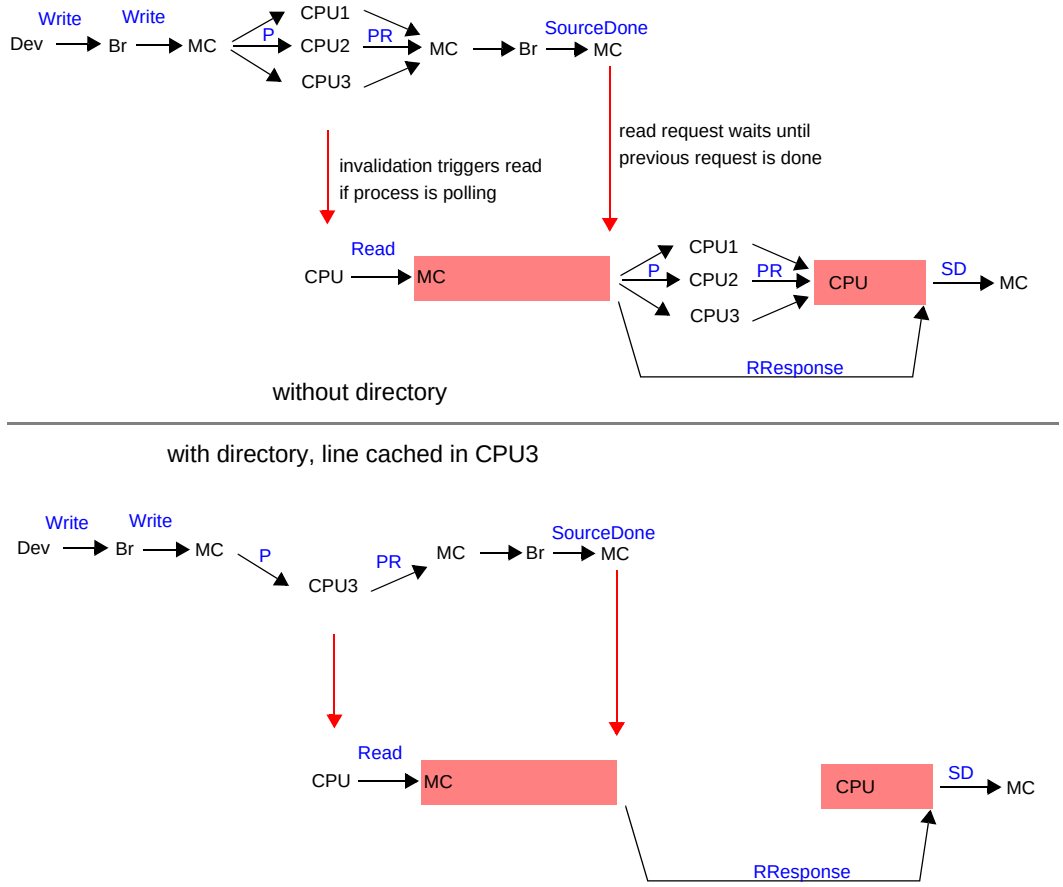


Figure 3-19. DMA transfer by device with subsequent processor access

Another assumption has been made regarding the target memory controller for DMA accesses. The following comparison assumes that the target memory controller for device memory accesses is always the memory controller that is local to the processor, i.e. within the same NUMA node. The reason to do so is that this is the best case in the DMA scheme. It is a feasible assumption for modern virtualized accelerator devices that provide direct

user-space access: Every thread may have its own queues in its own memory range, which may be allocated on the same node on which the thread is running.

3.5.1 Devices with Coherent Caches

The first variant of a device that uses the cache coherence protocol is a device with a coherent cache. In a queue-based communication scheme, the queue's home is main memory. However, if the device inserts a new entry into the queue, this element is not copied to DRAM using a DMA mechanism. Instead, it is allocated in the device's cache. The entry must be allocated in a modified state, as the value is different from the value in DRAM, and copies of this cacheline in other caches must be invalidated. A subsequent read request from a processor will cause the cacheline to be forwarded from the device's cache to the processor's cache. The corresponding flow diagram is depicted in Figure 3-21. The idea behind this mechanism is that an access to a cache's fast on-chip SRAM is much faster than DRAM accesses that are performed in the DMA scheme.

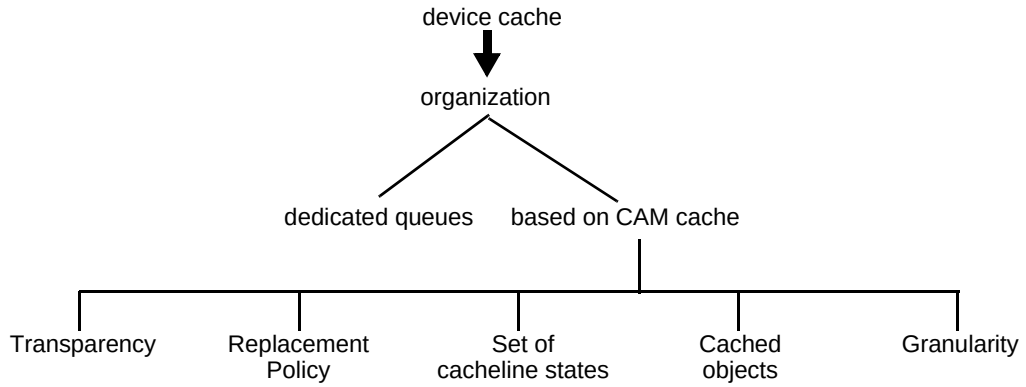


Figure 3-20. Design space for device cache implementation to speed up queues

In the following, the design space of such implementations (see Figure 3-20) is analyzed.

Cache Organization. A cache can be organized as an explicit set of queues, just as it would be implemented in a memory-based scheme. A suggestion for such an implementation are „cacheable queues“[10]. As already outlined in Chapter 3.2.2, dedicated queue structures are less advantageous for the use in virtualized devices. Also, the implementation of such queues as a coherent memory controller, as described in Section 3.5.2, is more promising. Therefore, the following paragraphs will focus on the second choice: an implementation as

a CAM-based cache. Such a cache may be used not only for queues, but other data transfers as well.

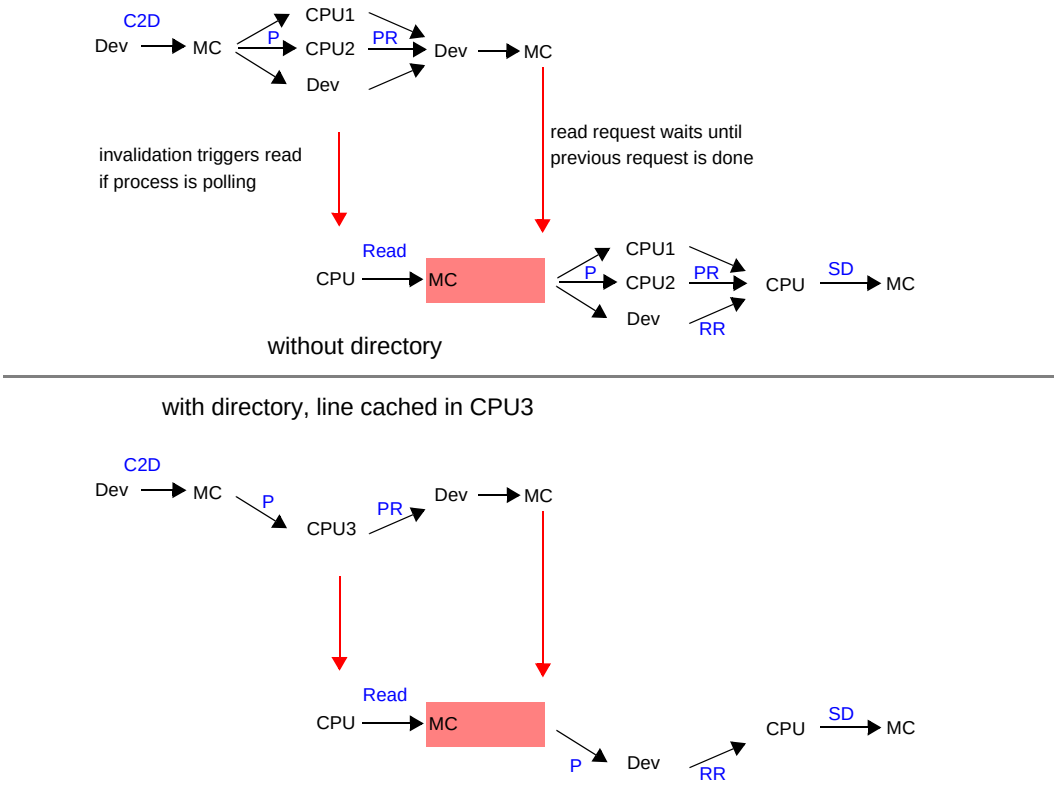


Figure 3-21. Caching instead of DMA transfer

Cache replacement policy. A device that caches data for its own use may use the same replacement policies as a processor. However, this is different for queue mechanisms. Queue entries will never be read by the device, and they will be read only once by the processor in most cases. Thus, cachelines should be evicted as soon as a read access by a processor has been seen. New cache blocks are allocated using empty, i.e. invalid, cache entries. Nevertheless, depending on the associativity of the cache, it may happen more or less easily that an empty cache frame cannot be found for a line. Now, there are two choices: either the cacheline bypasses the cache and is directly written to main memory. Or, an entry in the cache is replaced. Assuming that a small number of queues is cached, a replacement

means that entries that are close to the head are being replaced by tail entries. In the worst case, all process reads would be served from main memory, as they are being evicted from the cache before they can be read by the processor.

Bypassing means that the heads of the queue stay in the cache and are not replaced, so that at least a part of the processor read requests could be served from the device cache. On the other hand, a scheme without replacement may leave stale entries in the cache, for example if a receiving process terminates before reading the entry.

Generally, the efficiency of these mechanisms depends on the number of applications that are communicating with the device, the cache size, and the communication pattern. The communication pattern directly influences queue sizes and residence time in the cache. Thus, it is difficult to predict which of the mechanism performs better. This can only be found out with an evaluation in the system.

Cacheline states. In order to allow a cache to cache forwarding of the cacheline in a MOESI based system, it must be allocated on the M state. A subsequent read-exclusive request from a processor would cause the devices cache to forward the value to the requester and invalidate its copy. If the subsequent read is a read-as-shared request, it would cause the device's cacheline to change the state to O, and establish a shared copy in the processor's cache. In the case of a device-to-processor queue, there is no use for the cacheline any more, so that the cacheline may be evicted as soon as it changes its state to O.

An alternative for the device is to always transfer ownership by signaling in the read response to the requesting cache that the new cacheline state must be M. This frees the device from a writeback of the cacheline. However, the original MOESI and MESIF protocols do not support this.

Cached Objects. As explained in Section 3.2.2, only some data structures need to be made available with a low latency. Only those should be cached, all other data should bypass the cache and be written to main memory. This avoids that relevant cachelines are squeezed out of the cache, and also keeps the cache as small as possible. The question how data can be distinguished into worthy of caching and not worthy of caching on the device leads to the question of how transparent a device cache should be.

Transparency. In a device or accelerator context, it is beneficial if the cache is transparent to the device itself. In this case the cache is only part of a specific interface, and can easily be replaced by a different interface without the need to change the structure of the device itself. In a fully transparent design, the question what should be cached can only be determined at the cache, for example using address-based prediction [112]. If the constraint of a totally transparent cache is weakened, such a decision can be made at the level of a functional unit in the device. The most basic implementation is a single cache hint bit, stating if

the cacheline should be cached or not. Such a hint is the hardware equivalent to prefetch instructions of a processor, which also allows to define the caching behavior for individual cachelines.

Granularity of cached objects. As data transport for coherent communication is always performed by transferring cachelines, lowest latency can always be reached by aligning data structures as queue entries to cacheline boundaries. If a queue entry has a smaller size and thus does not fill out an entire cacheline, there will be a gap between entries. Of course, a queue entry is allowed to have a size of multiple cachelines as well.

A device cache may cache other structures than queues as well. For example, received data for get operations can be cached. If this data is not aligned to and sized as cachelines, there is a problem, as the remaining part of an incomplete cacheline may contain valid user data that must not be overridden. Thus, the memory block must be read from the coherent system, before the device's data can be inserted to the cacheline. This is not very efficient in terms of latency and hardware complexity of the device cache. Thus, such unaligned or miss-sized data should be written to the memory controller instead of being cached by the device.

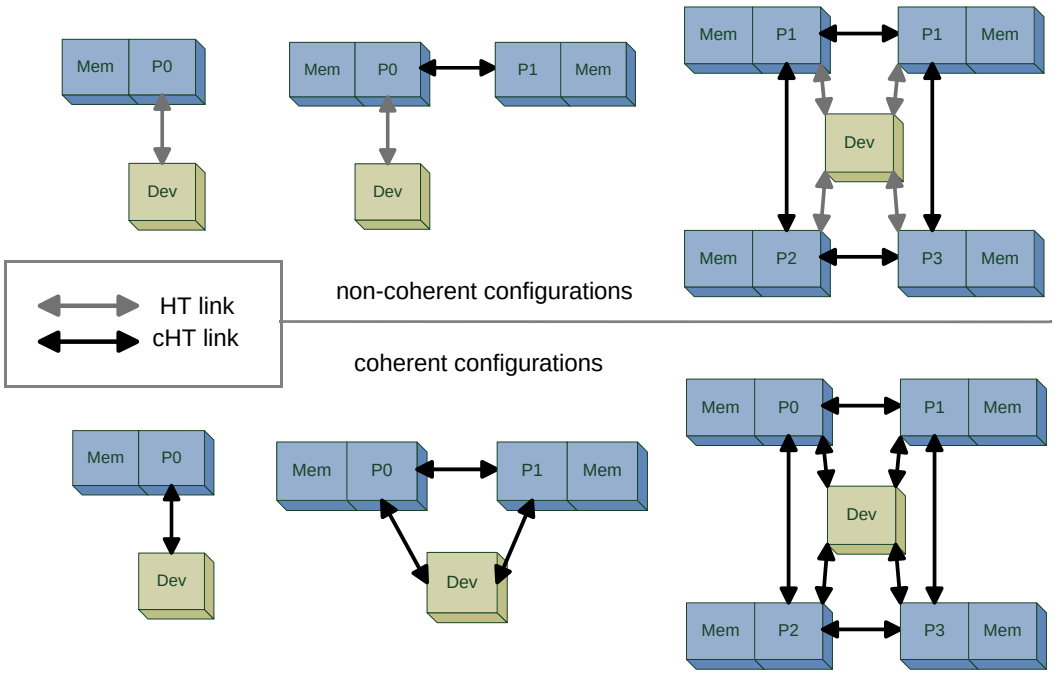


Figure 3-22. Configurations with coherent device caches

In the following two sections, the performance of device to processor transfers of such coherent device caches will be analyzed for off-SEC and on-SOC devices.

3.5.1.1 Off-SOC Devices

Figure 3-22 shows configurations of one, two and four node system topologies that have been evaluated, the respective latencies are given in Figure 3-23.

In the single node configuration, the coherent cache has a worse performance than the DMA scheme. This is due to the fact that the coherent domain now crosses chip boundaries, which increases probing latency drastically compared to the single processor system, where probing occurs on chip. Also, the single node configuration is the only one in which buffer-forwarding in the memory controller has a positive effect.

Thus, coherent devices should always be integrated into the coherent fabric so that they do optimally not increase the depth of a invalidation broadcast tree, measured in the number of hops. The topologies for two and four node systems thus have been chosen to be optimal, assuming that every NUMA node has three coherent links available. To have a fair comparison between protocols without the influence of the topology, the same dense topology has been applied for the noncoherent device.

With an optimal topology, the dual NUMA node configuration performances are equal. The latencies for the probe broadcasts are the same, and so are the polling and memory access latencies. Differences can only be observed if the topology is not selected in an optimal way. In the noncoherent example, P1 observes a higher polling latency if it is not directly connected to the device.

In the quad node configuration, probing latency is much higher than DRAM access latency and clearly dominates timing. Here, the cache-transfer mechanism is faster: if data comes from memory, the processor has to wait for all probes before it can use the respective data. If a processor receives a read response from a cache, it can immediately use the data. As the device is connected to the processors so that is has only 1-hop distance to every CPU, latency is lower as if the CPU has to wait for every devices response, due to the fact that there exist 2-hop connections between processors.

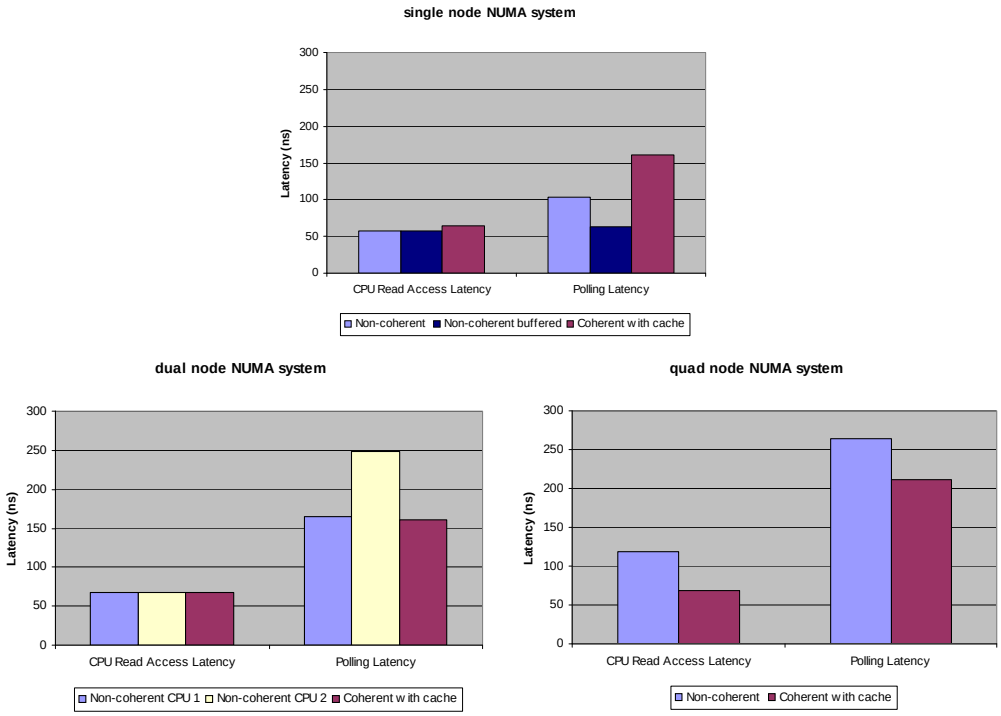


Figure 3-23. Performance of off-SOC device with coherent cache

Influence of a coherence directory. A directory may reduce the latency of memory accesses if the latency of the probing is higher than the latency of the DRAM access. This is due to the fact that using a directory, probes may be avoided at all in the best case.

In the topologies that are discussed here, only the four node topology displays a probing latency that is significantly higher than the DRAM access latency. In the best case, i.e. the accessed memory is on the same node as the processor, and no other processor is caching the line, performance increases up to the performance of the single node system.

Thus, the presence of a directory may improve performance, but will not make a direct cache to cache transfer more efficient.

What would the use of MESIF change? The implementation is analyzed for the MOESI protocol. In MOESI, a request is sent to the memory controller first for every memory access. This is avoided in the MESIF protocol, so that a cache to-cache transfer generally

has a lower latency. This work assumes a best-case DMA, i.e. that DMA memory is on the same chip as the processor that is requesting it. In this case, the overhead for this first hop is relatively small, so that a coherent cache in a MESIF system would not perform significantly better. However, MESIF would avoid a performance decrease in case the processor is not on the same chip as the memory controller.

3.5.1.2 Devices with Caches in SOCs

If a device is integrated onto the same chip as processor and memory, the situation is different. The device cache can be accessed with a low latency, as no chip boundaries have to be traversed. An on-SOC device with a coherent cache decreases latency drastically. In particular, the processor read latency would decrease by a factor of four (see Figure 3-24).

In contrast, if classical DMA is used, processor read latency does not improve compared to an external device. The polling latency would decrease by about 20% compared to the off-SOC solution.

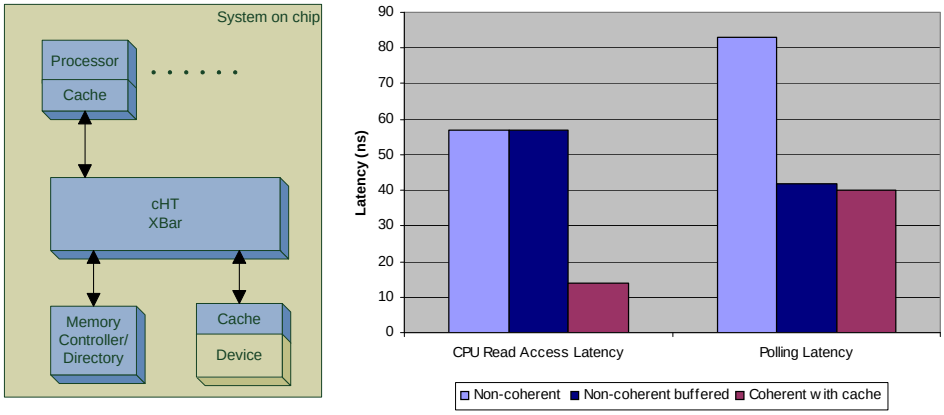


Figure 3-24. CPU read latency for on-SOC devices with a cache

The model that has been used for calculation assumes that the SOC features multiple processor cores with either separate or shared caches. Cache coherence is maintained by a broadcast or directory based protocol. If the number of caches that has to be kept coherent is relatively small, these design choices influence the latency of the transfer only marginally, assumed that contention is being neglected.

SOCs that integrate a number of processor cores, memory controller and a networking device are frequently used for processors that target networking applications, as for exam-

ple the XLR processor family from Raza Microelectronics, which features up to 8 multi-threaded processor cores, DDR2 memory controllers and two 10Gbit Ethernet devices [104]. Another example is the Sun T2 (see Section 2.6.1).

Multi-Chip environments of SOCs. In a multi-chip environment, every chip has a device instance. Even if the processor core and the cache that communicate with each other are on the same chip, the performance gain would be nullified if communication involves inter-chip probing. With the use of directories for the main memory, latency can be maintained at the same level as in a single SOC system.

3.5.2 Devices with a Coherent Memory Controller

A very different approach is a device that acts as a coherent memory controller. As in PIO, data flows from the device to the processor through read accesses to device memory. However, the processor's performance on cacheable memory is higher.

Another improvement over PIO are better premises for efficient consumer process notification (as introduced in Section 2.3.1). If valid bits are used for queue synchronization, a processor using PIO on uncachable memory has to continuously poll on the device memory, both wasting process and interconnect bandwidth. Using a coherent memory controller in the device, the processor polls on the cache, not occupying any interconnect bandwidth. A processor instruction that waits until a cacheline with a certain address is invalidated externally could even save processor resources, which is particularly useful for SMT architectures.

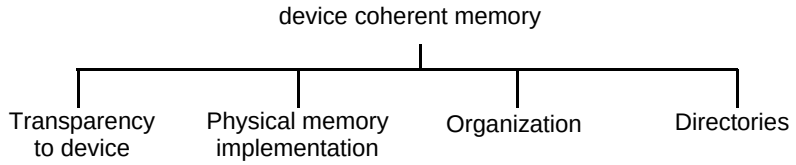


Figure 3-25. Design space of coherent memory on the device

The design space for devices with coherent memory controllers is shown in Figure 3-25.

Organization. This work focuses on a queue-based communication between processor and device. This means, that the coherent memory controller implements physical memory for these queues. RMA operations that target memory that is homed in another memory controller are thus not supported. Nevertheless, a device acting as a coherent memory controller does have applications in other fields, as for example shared memory controllers as Exten-

diScale [108], transactional memories or simply memory that is implemented in a different technology, as FLASH memory for example.

The physical memory implementation. As explained earlier, DRAM technology accounts for a large part of memory access latency, and thus should be avoided in the communication path between device and processors. Thus, a fast memory technology as embedded SRAM or ZRAM should be used if performance improvements are the goal. Another choice may be slower but larger memory, as external SRAM or DRAM, in combination with a transparent cache on the device.

The performance for a device using a fast embedded memory is shown in Figure 3-26. As expected, a speed-up cannot be achieved for off-chip devices. The on-chip solution shows a latency decrease by a factor of four in both the polling latency and processor read access latency.

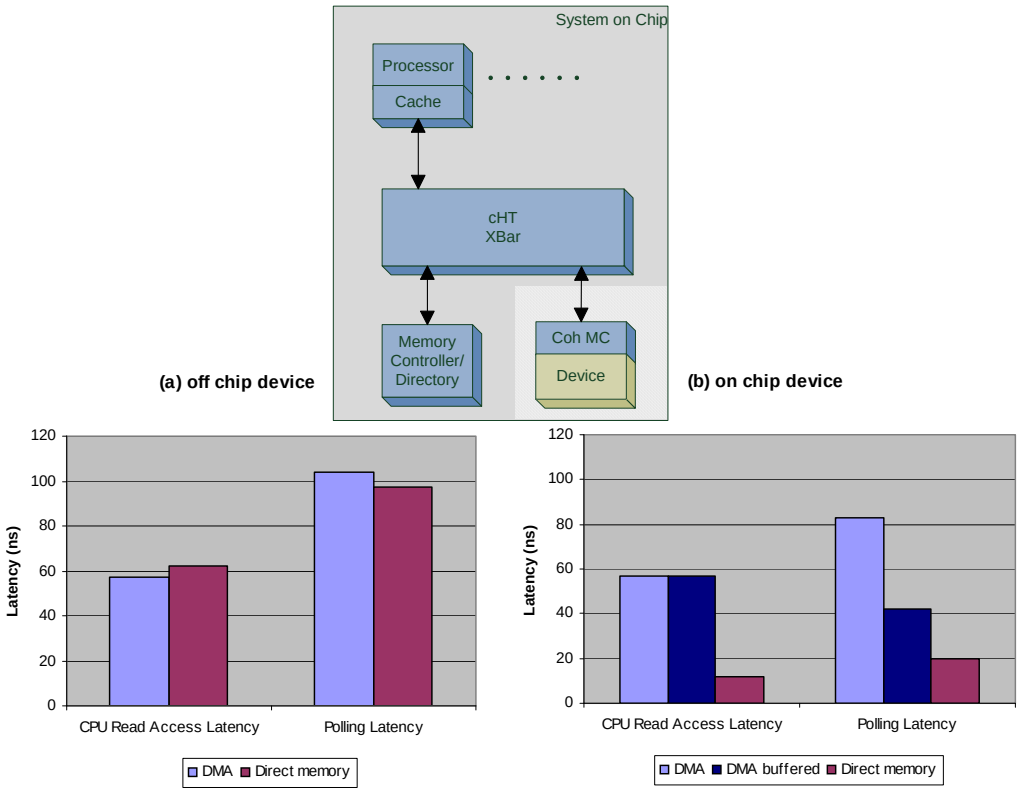


Figure 3-26. Latency of a device acting as coherent memory controller

Multi-chip systems. So far, these numbers are for a one chip solution. In a multi-chip system, communicating processor cache and device should be on the same cache to achieve best latencies. Even in this case probing over chip boundaries would destroy the performance benefit. In a small device memory, the implementation of a directory is not very expensive and solves this problem.

3.6 Transfer Cache

The previous approaches showed that a performance gain with coherent devices can only be achieved for systems on a chip. External coherent devices suffer from the fact that in a transaction, chip boundaries have to be crossed multiple times. In contrast, a DMA transfer crosses the boundary only once, as it pushes the data to memory.

Thus, an improvement for external devices is to move only the device cache into the processor chip, and leave the device off the processor chip. Thus, a chip boundary has to be crossed only once.

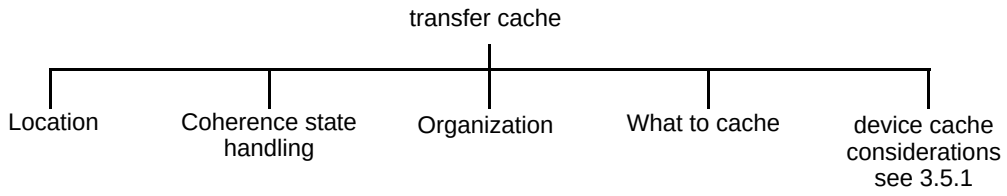


Figure 3-27. Design space for transfer caches

Location. In a multi-chip system, data can be cached in a coherent cache in the processor node to which the device is connected. Or, it can be cached at the node that is the home to the respective memory address, as shown in Figure 3-28. In this case, the cache can be implemented either as a coherent cache or behind a memory controller. As requests in a MOESI system are always directed to the memory controller first, an implementation in the same node is suggestive.

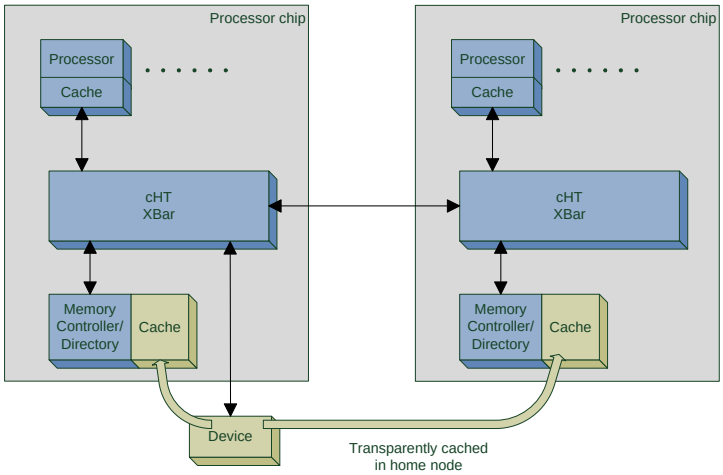


Figure 3-28. Transparent caching in memory controller of home node

Organization. The transfer cache can be implemented as any of the types presented in Section 3.5: as a coherent memory controller, or as a coherent cache. While the coherent cache can be used for all purposes, the memory controller implementation is rather limited. On the other hand, it is more efficient if a request is directly served by a memory controller, rather than forwarding the request to the cache. A combination of the advantages of both methods is a cache in the main memory controller that is transparent to the coherence protocol. Due to the transparency, the access pattern of such a transfer cache is the same as for a DMA access (see the flow diagram in Figure 3-19).

For such a cache, the considerations about device caches from Section 3.5.1 hold valid. However, there is a difference regarding the coherence state of cacheline:

State of the cachelines. A transparent cache within the memory controller must be kept coherent with the main memory. This can be done easily if the cache does not contain older values of a cacheline than physical main memory. Also, dirty entries must be marked so that they are written back eventually.

In a broadcast based protocol, this consistency scheme would work, but also would be inefficient. A processor cache reading a line that is present in the transparent cache will receive the response that contains the data very soon, but must wait until all probe responses arrive before it can forward data to the processor. Thus, the beneficial effect of a fast cache would be reduced. As a solution, a transparent cache should contain directory information for all

cached lines. Depending on the size of the system and the type of directory, the size of the required field is relatively small compared to the cache entry itself.

For the transfer cache suggested here, coherence state information can be encoded implicitly. A cacheline is filled by a normal DMA access from a device. All processor caches have to evict this line from their caches. As described in Section 3.5.1, a subsequent read hit in the transfer cache should lead to the eviction of the cacheline from the transfer cache. Thus, the presence of a cacheline in the transfer cache implies that the line is modified and not cached elsewhere. Then, probing does not need to occur upon a read request to that line.

What to cache. For the transfer cache, the differentiation between data that should be cached and other data is even more important than for a cache on the device. A large part of all write requests to the memory controller will be victim write-backs from the processor's caches. Those must not be cached, as this would pollute the cache.

Data can be distinguished at two locations: either in the transfer cache, or by the device. In the transfer cache, the decision can be made based on the source of the request, which will be an I/O bridge, and the address, either by a table-lookup or prediction. Besides the question whether such mechanisms have the desired effect, they add significant logic overhead into the path of every write request to the memory controller.

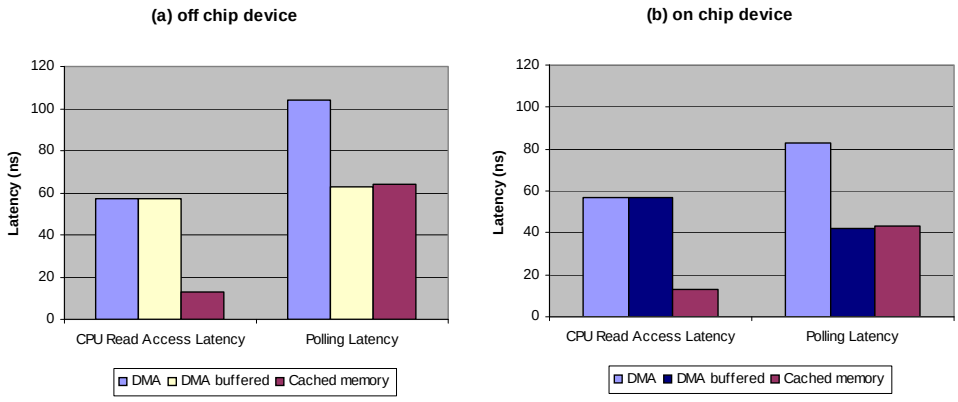


Figure 3-29. Transfer cache latencies

Thus, I suggest using the same methodology as for the device integrated cache: the functional units of the device determine what data should be cached. This information must be transmitted to the memory controller with the write requests. The most simple "cache hint" is a one bit field in the HT packet.

Figure 3-29 shows the transfer cache latencies. Processor read accesses to the cache have about the same latency as the other on-chip coherent solutions. This latency stays the same for the off chip implementations as well. Polling latency can be decreased by ~40% for the off-chip implementation, and ~50% for on-chip implementations.

An interesting system for a transfer cache implementation is the Sun T2. A device's write requests in the T2 pass a level of cache on their way to the home memory controller: the L2 cache. Instead of building a dedicated transfer cache, the general purpose L2 cache might be used to cache such transfers.

3.7 Results

3.7.1 Conclusion

Off-chip devices. Both coherent queue-based device implementations show no performance benefit for an off-chip device. This analysis assumed the same link speeds for all links. In reality, a device may not be able to run at the same link speed as the processors. This would further decrease the performance of coherent solutions!

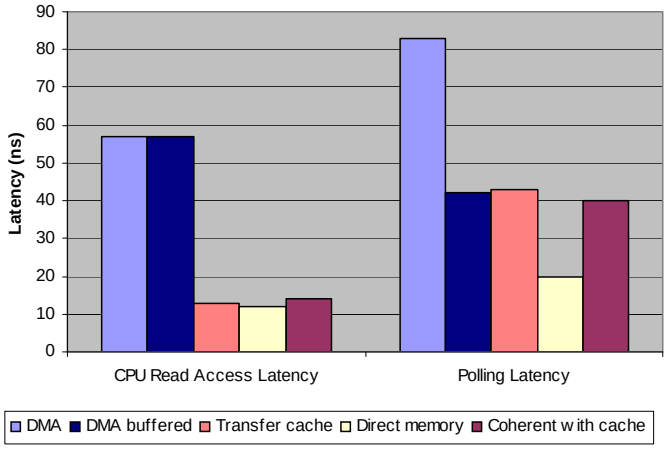


Figure 3-30. Latency summary for on-chip devices

Another important consideration is that such a cache-coherence device may increase the diameter of the coherent fabric. Thus, probe-broadcast may take longer, thus increasing the latency of every memory request in system. If the device is connected over a slower link,

or cannot keep pace with the number of incoming probes, this may have a severe impact on system performance. However, a device acting as a coherent memory controller without a coherent cache does not need to receive probes. Here, only asymmetric probing is required.

All in all, cache coherent interfaces cannot be recommended for external devices that implement queue based interfaces. This is in contrast to previous research done by Mukherjee [9] in 1998. The prime reason for this is that Mukherjee assumed memory latencies to be 3x the one-way hop latency (120 ns vs. 40 ns), while the factor is ~2 today.

The proposed transfer cache is the only solution to significantly decrease latencies for device to processor communication for off-chip implementations. It does not increase the diameter of the coherent network, and implicitly includes coherence state information.

SOC devices. Figure 3-30 shows the latencies for a SOC implementation on the device. A device with a coherent memory space offers the lowest latencies.

The only disadvantage of such devices is that this interface is less universal, as only a relatively small amount of memory can be implemented on devices. Caching, in contrast, is more universal, as it allows caching of the complete main memory range of the system. Device caching and a transfer cache have about the same performance in a chip SOC.

In a multi-SOC-chip environment, access latencies stay the same for direct memory devices, if processor memory and device are on the same chip and directories are being used to maintain coherence. If processor, device and memory controller are distributed on different chips, latencies mainly depend on the number of hops between processor, device and memory controller that are in the critical path. Table 3-1 shows the number of hops, which can directly be obtained from the respective flow diagrams. If processor, device and memory controller are distributed randomly in the system, direct memory has the lowest number of hops to take, followed by the transfer cache, traditional DMA and the device cache.

Table 3-1. Number of hops in the critical path between device and processor

# hops	**processor read**	**polling**
DMA without (with) directory	3 (2)	4
Direct Memory	2	3
Transfer Cache	2	4
Device Cache	3	5

The decision making process to select the architecture with the lowest latency is shown in Figure 3-31. A short summary of the conclusions follows:

- External coherent devices do not decrease the latency of device to processor communication. In contrast, coherent caches in devices may significantly slow down all memory accesses of the system. Only a transfer cache improves performance of external devices.
- A system-on-chip implementation of processor and device is inevitable to reach lowest latencies. In this case, all mechanisms that have been analyzed in this thesis offer a significant performance improvement over classical DMA.
- For SOCs in which processor cores communicate with devices on the same SOC, the direct memory mechanisms clearly offers the lowest latency.

All in all, the concept of the transfer cache as proposed in this work is particularly promising. Besides the performance advantages, devices can stay outside of the coherent domain. In practice, getting access to the proprietary, non-standardized coherent protocols may be difficult. Another point is that the cache hints that are embedded in the request packets can be used for direct processor cache access (DPCA) mechanisms as well. Thus, these mechanisms can be compatible with each other. An outlook on DPCA solutions is given in Chapter 5.

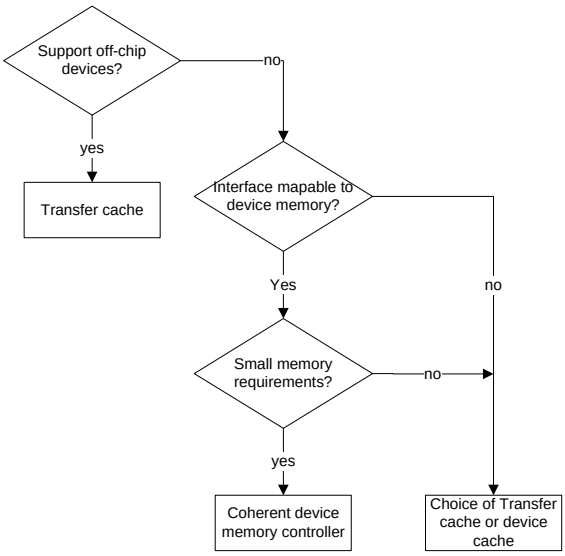


Figure 3-31. Decision process for coherent devices

3.7.2 Related Work

This section summarizes related work to coherent device interfaces or SOC integration that has not been presented as a complete system in Chapter 2.

SOC integration. [80] compares the placement of an Ethernet NIC on-die in a single processor system with classical off-die approaches. The finding that an on-die NIC has a slower performance than an off-die NIC cannot be explained by the authors. A second suggested architecture streams all received data of the NIC into the processors L2 cache, although the mechanism how this is supposed to work is not described.

[83] proposes a register-mapped interface to message-passing NICs. Besides an implementation in the register file of a processor, implementations as an either on- or off-chip cache are proposed. In both cases, a bus is assumed as interconnection network between processor and caches.

Coherent Devices. Except for coherent shared memory systems, coherent NICs are not commonly used. Muckherjee [9] presented and simulated so called “cacheable queues”, which are very similar to the device cache and coherent memory device solutions presented in this work. Muckherjee analyzes systems where device and processor are on different chips, and finds that coherent transfers are faster than conventional DMA. This is in contrast to this thesis. This difference can clearly be traced back to the worsening of interconnect and memory bottlenecks.

A device based on the idea of Muckherjee is the JNIC prototype system of a 10 GBit Ethernet NIC [89]. JNIC onloads much of the protocol processing overhead to one of the general purpose processors in the system. Queues that reside in the coherent memory of the device are for communication between the hardware and the software part of the NIC.

4 HT and cHT Prototypes

This chapter describes the specification and implementation of coherent HyperTransport (cHT) and noncoherent HyperTransport (nHT) solutions for the Extoll NIC. The focus of this chapter lies on the noncoherent and coherent infrastructures that handle the data exchange between device and the other components of the system.

The first section of this chapter describes the specification and implementation of a HyperTransport core to utilize the direct connect architecture as described in Section 3.1. A direct connection to an Opteron processor is achieved by mapping the core on the HTX board. The current implementation of the nHT Extoll NIC is the Ultra NIC, as depicted in Figure 4-1.

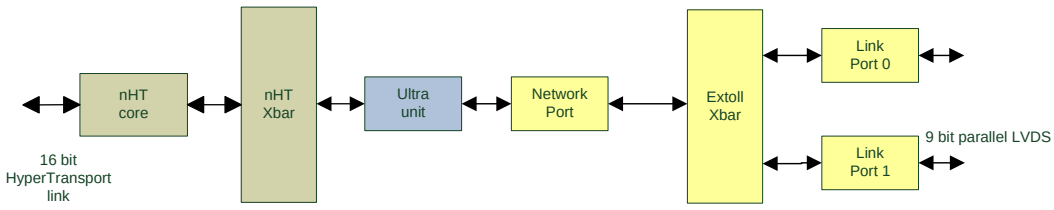


Figure 4-1. Ultra NIC

Section 4.2 details the coherent HyperTransport interface for Extoll, which is based on the ideas of devices with a coherent cache or a coherent memory controller, as developed in Section 3.4 and Section 3.5.

4.1 The HT Core and Interface

The HyperTransport core has two different interfaces: On the one side are the HyperTransport send and receive links, as specified in the HyperTransport protocol. On the other side, there is the application interface, which allows FPGA designs to access the HyperTransport core. This interface consists of three queues in each direction, one for every virtual channel. Applications can access these using a valid-stop synchronization mechanism. Control and attached data packets can be delivered simultaneously over the 160-bit-wide interface (96 bit to handle extended control packets, 64 bit for data packets).

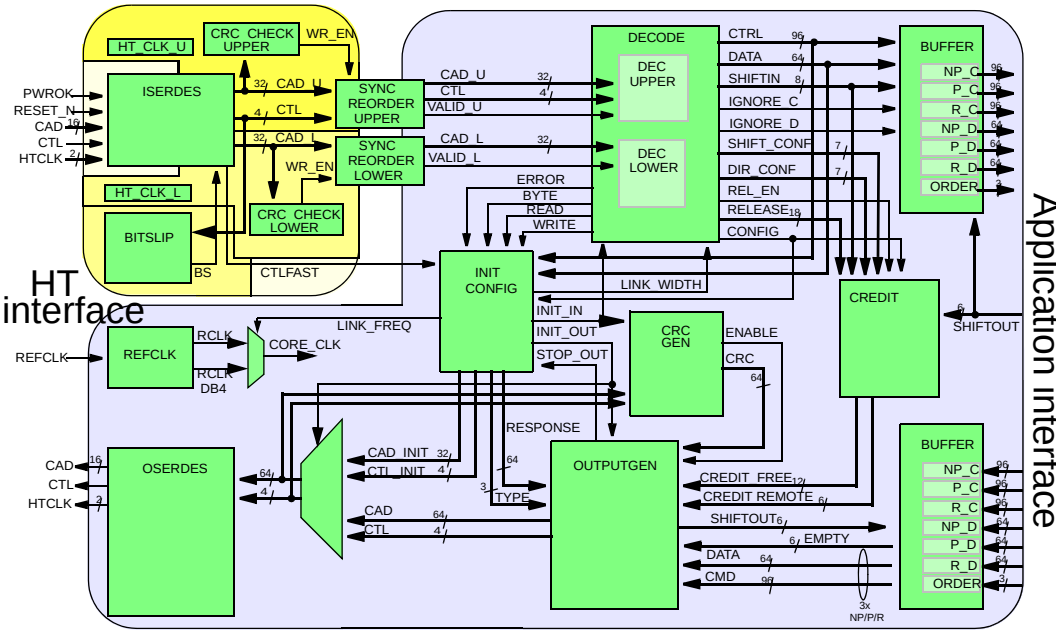


Figure 4-2. Block diagram of the nHT core [51]

With a maximum HT link width for Opteron systems of 16 bit, the performance of the link depends mainly on the link clock frequency. One limiting factor is the speed of the serial I/Os. In the FPGA used on the HTX boards, they limit the speed to 400-MHz DDR, thus HT400. Xilinx serializer/deserializer (SERDES) blocks parallelize/serialize the link by a factor of four, so that the frequency of the internal core clock is 200 MHz, and the data path

has a width of 64 bit. The SERDES blocks are controlled by a “bitslip” module to generate proper alignment to 32-bit boundaries.

An important constraint is to process this data stream with the lowest number of pipeline stages and reasonable resource requirements.

Scalability and portability. The HT core is implemented in the Verilog hardware description language. This makes the design easily portable to other platforms, as FPGAs or ASICs from other vendors. Only device- or process-specific hard macro blocks have to be exchanged. These are SRAMs, DLLs or PLLs, I/O cells and the serializers and deserializers.

The scalability of HT core implementations in FPGAs to higher HT link clock frequencies is limited (see Figure 4-3). Faster I/O cells are already available in the newest generation devices, but the maximum internal clock speed is unlikely to scale up by the same factor. Thus, FPGA implementations of an HT core for higher link clock frequencies require a completely new design of the core, with a higher parallelization degree. The downside of this is a significantly increased complexity, which increases the number of utilized FPGA resources and the length of pipelines. It is also questionable if applications in an FPGA could take advantage of the resulting high bandwidth.

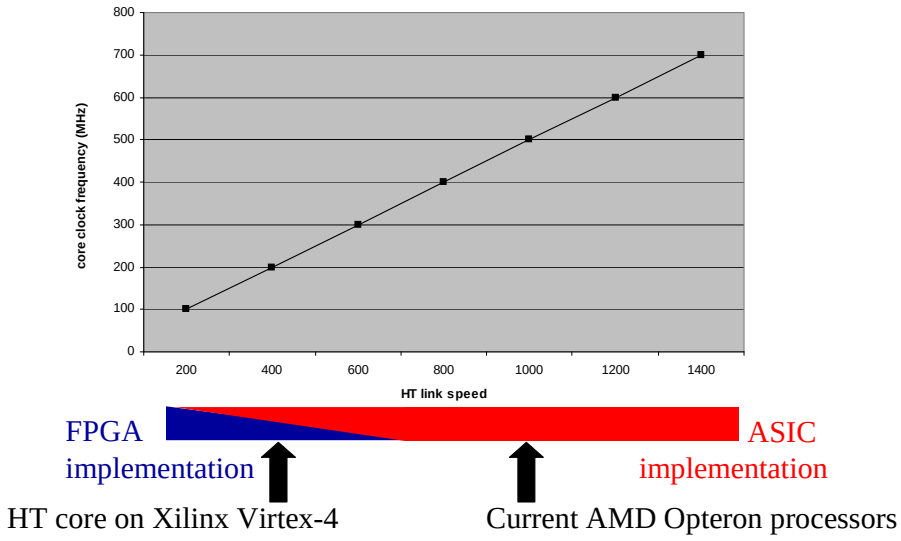


Figure 4-3. Scalability of the HT core

Significant performance improvements can only be expected if ASIC are used instead of FPGAs. ASIC implementations in modern process technologies with internal clock frequencies of 500 or 600MHz are very feasible. Thus, the current design is the perfect choice for FPGA implementation and verification of high-speed designs that will be implemented in an ASIC, as the HT core is the same for both. If different core architectures would be used for different implementations, an FPGA-based verification could not prove that the ASIC implementation works.

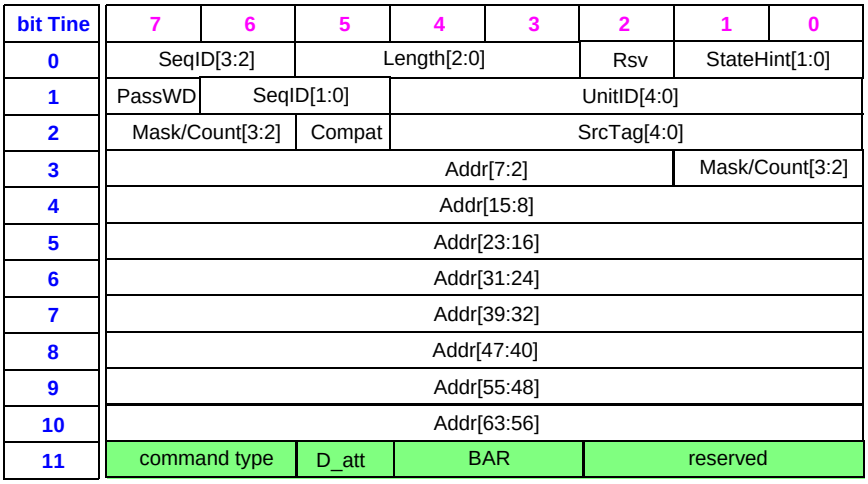


Figure 4-4. Command packet format at application interface

The application interface. The 96 bit sized on-chip control packet is basically the same as the largest HT control packet, which is an extended packet that uses 64 bit addresses. 8 bits of redundant information have been removed and are used for an internal tag (marked green in Figure 4-4). Associated data is transmitted 64 bit parallel.

As in Extoll, most devices will have to connect multiple internal units to the HyperTransport interface. A crossbar is the most universal switching structure, as it allows multiple concurrent transactions. The Extoll on-chip HT crossbar is such an implementation that is based on the HT protocol [122] and is directly connected to the HyperTransport core.

4.1.1 Results

The resource requirements of the 16bit HT400 core implementation on a Xilinx Virtex-4 FX 60 are shown in Table 4-1. The hardware latency of the crossbar is 12 internal clock cycles for the inbound path from link to application interface, and 6 cycles for the outbound path.

Table 4-1. HT core resource requirements in a Virtex-4 FX 60 FPGA

Resource	**absolute**	**relative**
Logic Slices	5,222	20%
Look-up tables	6,371	12%
Flip-flops	2781	5%
FIFO16/ RAMB16s	33	14%
DCMs	3	25%
ISERDESs	10	1%
OSERDES	9	1%

Table 4-2. Hardware latencies of the core

Direction	**Clock Cycles**	**Delay@ HT200**	**Delay@ HT400**	**Delay@ HT1000**
In	12	120 ns	60 ns	24 ns
Out	6	60 ns	30 ns	12 ns

Performance in the system. The 16bit HT400 core has been evaluated [127] in the system described earlier in this chapter. PIO accesses from the process to the device are one type of transaction. For write accesses, the memory type “write combing” has been used, which combines stores to subsequent addresses into one write access with the maximum size of 64bytes. For these writes, the sustainable bandwidth is 874 MB/s.

For read accesses, both the bandwidth and latency can be measured. The latency is 39 HT core clock cycles, which corresponds to 195 ns. For read accesses, only 32 bit read accesses

can be observed. This of course implies that larger processor reads result in a fairly high latency, as e.g. an 128 bit read is executed using 4 individual ordered 32 bit reads, resulting in a latency of 780 ns. The bandwidth for reads is thus quite low: 20 MByte/s.

The second type of accesses are those initiated by the device: DMA operations. The achievable bandwidth depends on the HT packet size. Using the largest HT packet size of 64 bytes, write bandwidth goes up to 1410 MB/s, while read bandwidth is 1040 MB/s.

Performance of the Ultra NIC. The Ultra unit targets low latency communication with small messages. Figure 4-5 shows the half round trip latency, measured with the NetPIPE benchmark. The latency seen on the Extoll API level is below $1\mu s$, which is an excellent result when compared to other NICs [125]. These are the results of an PFGA prototype design with an HT200 core and an internal clock frequency of only 100MHz. Due to technical difficulties, Ultra has net been evaluated using the HT400 core. However, significant performance improvements can be expected.

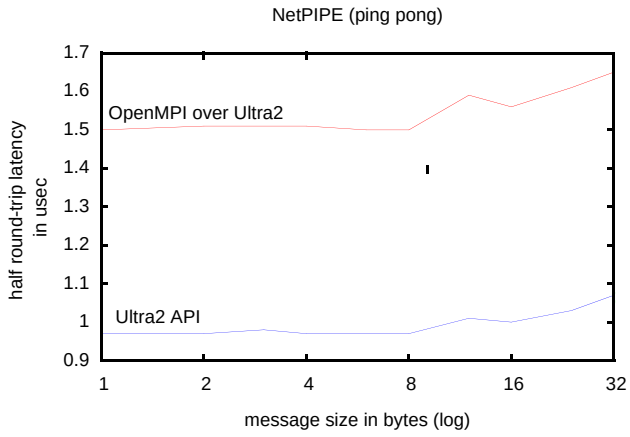


Figure 4-5. Ultra ping-pong latencies in a two-node network [125]

All in all, the HT core successfully exploits the potential of the used FPGA in terms of bandwidth, latency and resource utilization. Offering an HT400 connection, and thus a bidirectional bandwidth of 3.2 GByte/s, the HT core can be used for more than just prototyping. The performance is sufficiently good to serve as a production coprocessor board as well.

4.2 The Coherent HT Infrastructure

The guiding idea of the coherent device infrastructure is that the noncoherent Extoll device should be able to use the coherent communication mechanisms that have been described in the previous chapter. At the same time, changes to the Extoll core should be as minimal as possible.

While the previous chapter showed that performance increases cannot be expected from an implementation in an external device, such an implementation is the only way to proof that the mechanism works.

A second thought is that the cHT infrastructure should be designed in such a way that other applications may be using it, and in particular those that intrinsically need a coherent memory view. Such applications are in particular NI devices that provide a coherent shared memory view of the system, or testbeds for transactional memory systems.

This section describes the coherent solutions that have been proposed and analyzed in Section 3.4 and Section 3.5: a coherent device cache and coherent device memory space by embedding a transparent memory controller.

4.2.1 The Coherent Fabric

Figure 4-6 shows a block diagram of the coherent HyperTransport device infrastructure.

The coherent device fabric is organized very similar to that of an Opteron northbridge. The cHT crossbar is the central unit in the coherent part, over which all other components get connected with each other. The configuration of the coherent part, and in particular the configuration of the routing tables, is done using the same set of configuration registers as Opteron northbridges. Due to the similarities with a Opteron northbridge, changes in the BIOS can be kept at a minimum.

As specified by AMD [22][1], a maximum of four HT units IDs per NUMA node exist. Individual generations of Opteron processor may have restrictions about the functionality of units. In the 9th generation Opteron's used in this project, unit IDs are: "0" for I/O bridges, "1" and "2" for processors/caches, and "3" for coherent memory controllers.

1. All following details about AMDs architecture are also taken from this document.

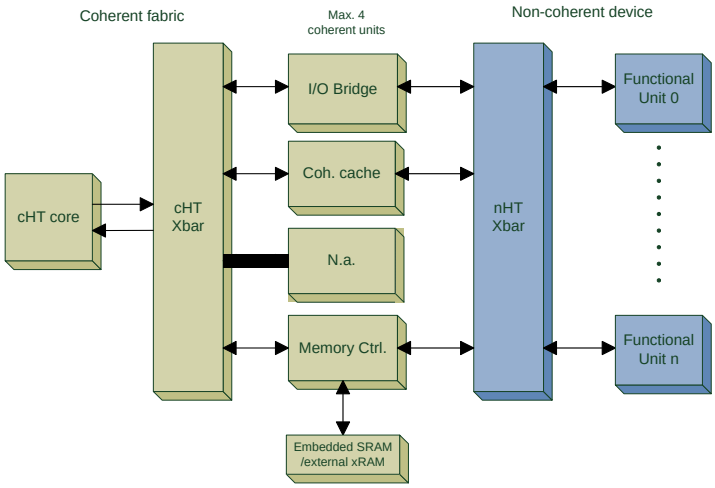


Figure 4-6. The coherent device infrastructure

4.2.2 Units and Crossbars.

As cHT units, the following types of units exist: a cHT/nHT bridge, a cache, and a coherent memory controller. Not all units have to be present, instead, they may be combined with each other depending on the requirements of the specific application. All these units have a noncoherent HT Crossbar compatible interface to the application side. This allows to implement noncoherent devices behind the coherent part.

The need for two crossbar switches. The coherent fabric has two crossbar switches, that are relatively close to each other. Every crossbar has a latency of three clock cycles and also introduces some wiring complexity. This may be a problem especially for implementations on FPGAs, where routing resources are limited. However, the following advantages clearly outweigh the disadvantages of having two crossbars:

- In order to keep changes to the noncoherent functional units as small as possible, their on-chip nHT interface should not be changed. Thus, the nHT crossbar is essential to connect the various functional units, and thus is required.

- The previous chapter showed that a coherent fabric should be as densely interconnected as possible. A cHT crossbars opens up the possibility of adding more cHT links to the device. While an implementation in the HTX slot does currently not allow the use of more than one link, implementations in the processor socket could easily do so.

The interface between cHT crossbar and the coherent units is just as specified by the cHT protocols.

The noncoherent HT crossbar routes requests based on the address. Thus, every unit at a switch port that may be target of requests has a unique address space. Responses are routed based on the *SourceTag* field in the HT packet. The coherent memory controller can be integrated just the same way. For an access to the coherent cache, the cache access (*C_acc*) bit must be set, as this overrides address based routing. Thus, the command packet forward must contain the *C_acc* bit in the tag.

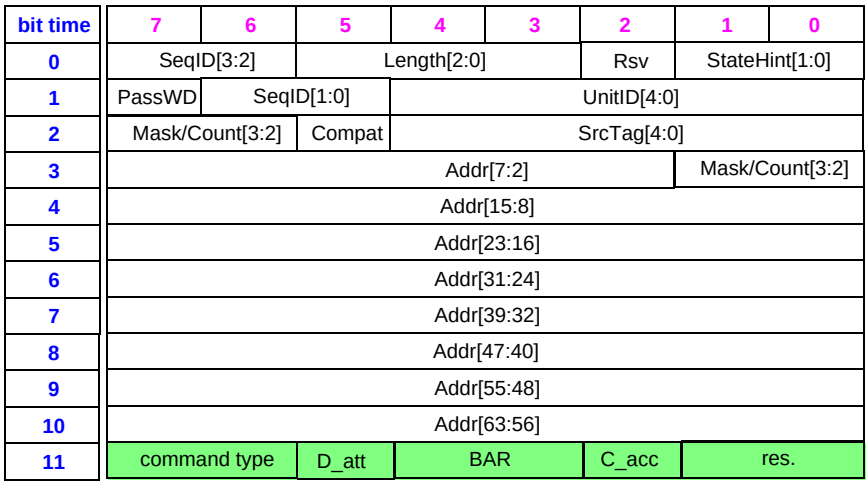


Figure 4-7. Coherent cache-aware command packet format at the nHT crossbar.

4.2.3 cHT/nHT Bridge

The cHT/nHT bridge allows noncoherent device functionality without any changes in the device behind the coherent fabric or the corresponding software. Due to the similarities between cHT and nHT protocols, the essential functionality is small.

Accesses from the coherent domain may can only be noncoherent read and write requests. The bridge translate these from the cHT protocol to the nHT protocol, and vice-versa for

responses from the device. For some requests, the bridge also has to send a response back to the requester in the coherent fabric.

Device-initiated sized read or sized write requests to the coherent domain must be forwarded into the coherent domain. This includes a routing table lookup. For reads, the bridge will not only get a read response, but also has to collect probe responses. For writes, ordering must be maintained. Thus, write requests are only forwarded if all previous writes have been acknowledged.

4.2.4 Cache Design

There are 2 potential applications for a coherent device cache. A general data cache, as discussed in Section 3.4.1 on page 89, and a queue cache. A general cHT data cache has been implemented [129]. Only small modifications are required to use this cache as a queue cache. The following section gives a brief overview about the cache implementation and modifications that are required for a queue cache.

For a general cache, the following transactions go over the coherent interface:

1. Read requests due to misses in the cache, and corresponding read and probe responses.
2. Change to dirty request due to a write hit to a non-exclusive cacheline, or due to the new allocation of a complete cacheline.
3. Write requests due to cache evictions of modified data.
4. Probing requests caused by accesses of remote processors or devices. These must be answered by either a probe response or a read response with the cacheline data.

Transactions 1 through 3 are all initiated by the device. Only transactions type 4 is initiated externally. In a queue cache, transactions as in 1 do not occur. Mechanism 2 takes place when new entries are inserted into the queue by the device. Data is transported to the processors caches via 4. Write-backs, item number 3, occur if a processor reads a queue entry, but the obligation to write the dirty cacheline to memory does not move to the processor cache.

Figure 4-8 shows the top level diagram. Two units can access the cache data. The probe handler is responsible for all requests from the outside. Its main task is to answer incoming probe requests either with probe responses, or with read responses.

The cache logic module is responsible for all requests that originate from the device. It is organized as a pipeline and can thus process multiple requests simultaneously (see Figure 4-9). The XBar input stage obtains control packets over the nHT crossbar. In the next pipeline stage, a cache lookup occurs. As the cache may be busy with other requests, this operation may stall for some clock cycles. Depending on the type of request and the hit/

miss information from the cache, the cache FSM unit schedules the request to the next pipeline stage.

Upon a *read hit*, the direct response unit will generate a response. For a *read miss*, the request store unit will generate a request on the cHT side, and also allocate an entry in the request queue and in the cache. The allocation of a new cache entry may evict another entry. If this entry is modified, it is written back to main memory by the CacheLineAdmin unit. Responses to these requests arrive at the request match unit, which matches the request with the entries in the request queue. Then, the cache store unit writes the entry to the cache and at the same time forwards the response the requester on the nHT side.

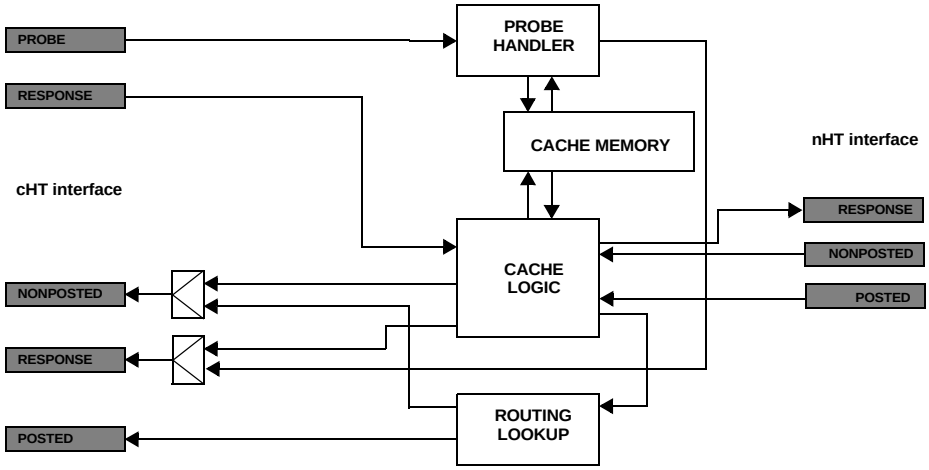


Figure 4-8. Cache top level diagram [129]

A *write hit* will be forwarded to the cache store unit, which inserts the data into the cacheline. If the state of the cacheline is not an exclusive one, the CacheLineAdmin stage must send invalidations to all other caches. In this case, the cache entry is marked “busy” and may not be used unit all caches have acknowledged.

A *write miss* will generally bypass the cache and proceed directly to the CacheLineAdmin stage. An exception are write misses that write a complete cacheline of data. As for a read miss, a cache entry has to be allocated. After that the CacheStore unit writes the data into the cacheline. The CacheLineAdmin unit must also generate invalidations on the coherent HT.

It requires two modifications to turn the cache into a queue cache:

- As analyzed in Section 3.5.1 on page 94, it depends on factors like cache size and the number of queues in the system whether cacheline replacement or bypassing should be used. Thus, the cache should support both methods. The mode can be selected by the device driver by setting a field in the configuration register space of the cache. This can happen anytime during normal operation.
- Upon an external read via a probe request, the cacheline should be evicted from the cache.

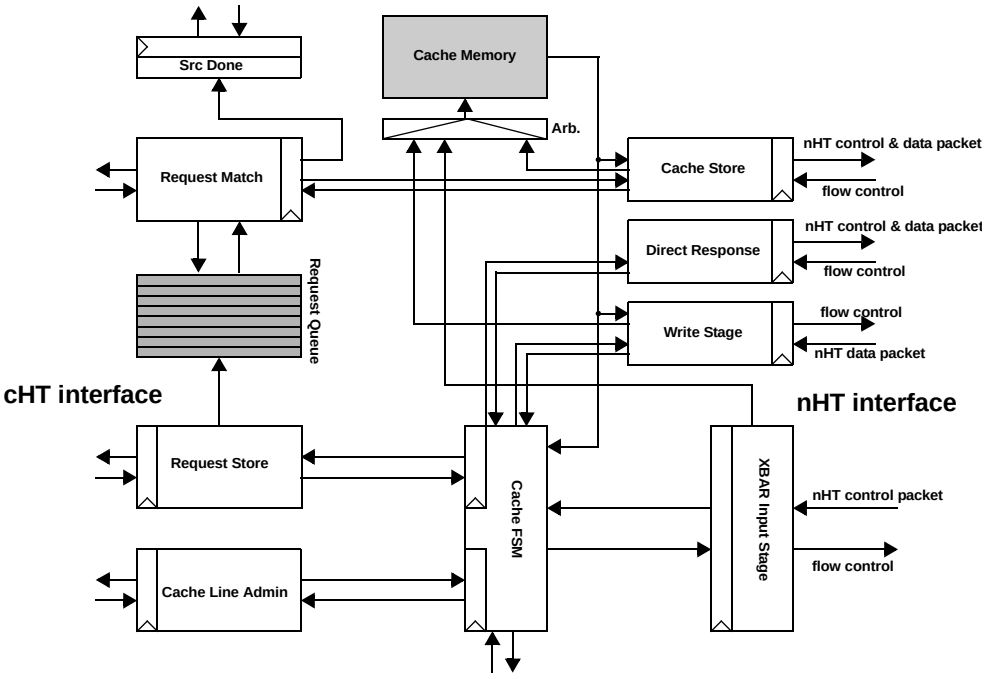


Figure 4-9. Block diagram of the cache logic module [129]

Ultra Implementation with coherently cached queues. The only modification to the Ultra unit when it is used with the coherent cache to implement cacheable queues is that it must set the C_acc bit for such data that should be cached in the device. As Ultra is designed for low-latency, fine-grain communication, all received data and the corresponding control data are ideally cached. Thus, the Ultra receiver must set C_acc for all writes of such data to efficiently make use of the queue cache.

4.2.5 Transparent Memory Controller in the Device

The memory controller provides a coherent memory space to the host system. A top level block diagram is shown in Figure 4-10. It is designed for a data communication from device to the host system as described in Section 3.5.2. The device needs write access to the memory space, while the host system should obtain both read and write access. An additional write access for the device makes the memory controller a much more universal module.

Requests to the memory space by the host behave just exactly like requests to the system's coherent memory. The device driver controls which memory regions reside on the on-device memory and which reside on the system's main memory, and configures address registers in the functional units correspondingly. For the functional units of the device, the on-device memory controller is fully transparent. The FUs simply perform memory accesses, which will be routed by the crossbar either to the host bridge or to the on-chip memory.

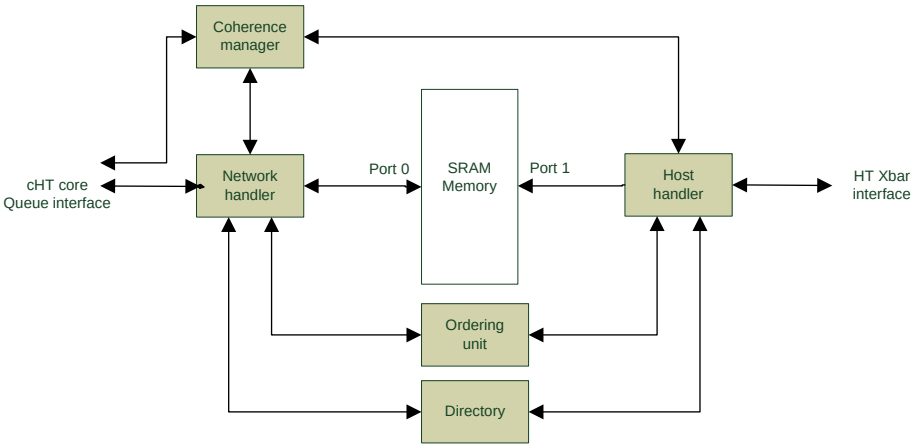


Figure 4-10. Coherent memory controller

4.3 Summary

This chapter detailed the implementation of the noncoherent HyperTransport core, which is the heart of the direct connect architecture. Performance results are excellent, and very promising for faster implementations of the core. The HT core implementation is already being used for both research and production systems.

This chapter also specified the coherent environment and its components. A full specification is essential to prove the functionality and has been performed with success. Also, the coherent HT core, the cHT crossbar, and the coherent cache have been implemented and tested in the FPGA prototype system. As predicted in Chapter 3, the performance of an external coherent device is worse than for noncoherent devices, especially if the link to the device is a only an HT400 link. Thus the coherent framework will not be used for an FPGA production system. However, it is an excellent testbed for the verification of ASIC-implemented coherent designs, and for the research on coherent networks in SOCs.

5 Suggestions for Direct Processor Cache Access

This chapter provides an outlook on direct processor cache access (DCA) architectures. DCA has the same goal as the transfer cache that has been proposed in Chapter 3: data that is written by the device and will very soon be consumed by a processor is cached in order to minimize access latencies. DCA moves the cache much closer to the processor, as it uses the standard data caches of the processor. Thus, the read access latency that occurs when a processor reads this data is further reduced. In contrast, significant reductions of the overall latency of a transfer between device cannot be expected when compared with the coherent solutions and the transfer cache as presented in Chapter 3. Just as in the other mechanisms, DCA must occur in a cache coherent fashion, which cost some time.

Nevertheless, the reduction of the processor read access latency that DCA brings is significant for overall application performance [40].

Many considerations that have been made for the coherent device cache and the transfer cache in Section 3.5.1 and Section 3.6 hold true for DCA as well. This includes the types of data that should generally be cached, and which component of the system can best decide this.

However, two key challenges have to be solved to enable DCA architectures. The first one occurs in multiprocessor systems, where the right target processor has to be identified. The second one is the development of a data transport mechanism into the processor caches.

Section 5.1 first shortly describes the design space. Then, the design space is narrowed down by creating a mapping from device contexts over processes to processors, and analyzed in Section 5.1.1. Section 5.2 proposes four alternative mechanisms how DCA could be implemented in a HyperTransport based system that uses an IOMMU.

5.1 The Design Space

The most important design aspects of direct processor cache access mechanisms as shown in Figure 5-1 shall be briefly discussed.

Initiation. A DCA is per definition triggered by the producer of data, i.e. the device. However, the device does not necessarily need to be the initiator for the actual transfer of data. For example, the device may send a prefetch hint message to the processor. The processor then can decide to actively pull the respective data into the cache. If the device is the initiator, it pushes data into the processor cache.

Mapping. A key problem for DCA mechanisms in multiprocessor systems is to identify the right processor cache that should be the target for a DCA. Assuming virtualized devices, the task is to map a device context to a thread. In a second step, the processor must be identified which executes this thread. Other information besides the device context, as for example the source of a message may be included in the mapping process as well.

Theoretically, address based mapping can be used instead of context based mapping. In this case, large address mapping tables must be maintained.

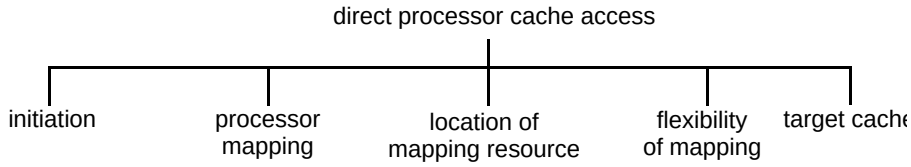


Figure 5-1. Design aspects of DCA mechanisms

The **Location** of mapping resources can be at the device, at the I/O bridge for noncoherent devices, or at the memory controller. A device based mapping has the advantage that mapping resources scale with the number of devices. The interconnect to the device, as HyperTransport for example, must support DCA packets.

If the mapping is performed by the bridge, most details of the DCA protocol may be hidden from the device, which increases inter-platform compatibility. DCA mapping could be integrated into existing data structures, as the I/O memory management unit (IOMMU). An IOMMU may allow a device to perform the address translation itself using an IOTLB. Similarly, the device could be allowed to perform the mapping using the same TLB.

Flexibility. A process or thread may be statically bound to a processor. In this case, the mapping does not have to be changed during lifetime. However, it may be disadvantageous if the process can never be scheduled to a different processor. If processes are allowed to be scheduled to other processors, mapping tables must be updated. Also, there is a penalty for every such processor switch, as the newly assigned processor has a cold cache. Thus, even for a dynamic scheduling of threads, processor switches should be avoided.

Target Cache. As a target cache, any of the data caches that is associated with the target processor can be used. In some systems, multiple processor cores share a level of cache. For example, four-core Opteron processors share an L3 cache. If this L3 cache is selected as a target cache, a process does not need to be bound to a single processor core. Instead, it has to be bound to the group of cores, on which it can be scheduled freely.

5.1.1 Device - Thread - Processor Relations

In a symmetric multiprocessor environment, DCA is complicated by the fact that a thread typically may be scheduled to any processor of the system. However, DCA can only work if this processor can be identified. This chapter discusses how a device context, a thread and the processor can be associated with each other.

A device context may be a real, explicit context, as in the virtualized Extoll architecture. In devices where no explicit context exists, it can be implicitly given by the information in a packet header. For TCP/IP packets, the combination of the target TCP port and target IP address may define such a context.

Every device context is mapped to a single process exclusively. The relation between context and thread is not bijective: a process may use several contexts at the same time.

The device-thread relation. When analyzing how incoming requests are distributed to processes or threads in the system, two application models have to be considered. One possibility is that a number of different and independent processes is running in the system. As a process can only run one processor at a time, a binding between a device context and a process may thus explicitly define the processor which is assigned to the process.

The second application model is a number of worker threads that are used to process requests. There are no differences between the worker threads, any thread may work on any request that is coming in from the network. This type of application is typical for commercial workloads as web servers and database servers. In a socket based environment, every worker thread has an exclusive set of ports that it uses to work on incoming requests. This port has been assigned to the thread when the socket based connection was established. Thus, there exists a device context to thread relation as well.

The thread-processor relation. If a process is in execution on a processor, there is a clearly defined relation between the thread and the processor. In particular, a thread can determine the current processor on which it is running. However, this is not sufficient for a implementation of DCA. Packets for a process may arrive as well while the process is not in execution.

In essence, DCA requires a thread-processor relation that does not change frequently. This is the case if the operating system's scheduler avoids to shift threads between processors. Besides DCA, there are other potential benefits of such a policy. In a NUMA system, it is advantageous if a thread is running on the same die that holds the processes memory (see Section 3.3). Also, if a thread has been de-scheduled only for a short period of time, a part of the current working set of the thread may still be present in this processor's cache. A rescheduling to the same processor can thus significantly decrease the start-up penalty that is caused by a cold cache. Under Linux, the kernel-level "numalib" library and the "numactl" tool [109] give users explicit control over the allocation of memory and processors. Using these functions, a process can for example be bound to a specific processor so that it executes only on this processor. It also can be bound so that it executes preferably on this processor.

As the operating system or the thread itself know the processor that is currently assigned to the process, this information can easily be communicated to create a context-thread-processor relation.

A different approach is to create a direct context-processor relation. Such a solution is implemented with the Receive Side Scaling (RSS) [39] mechanism. Intel claims [37] that the distribution of incoming packets to processors is one of the large bottlenecks in TCP/IP processing using modern 10GbEthernet NICs. RSS specifies a way how a processor can determine the target processor that has to process an incoming packet. This is done by computing a hash function on certain parts of the packet header, including source and destination IP addresses. Source and destination port may also be part of the address. The hash result is used to select a target processor over an indirection table in the device. The intention of RSS is not to fill the processor's cache, but to send a directed interrupt to the processor. However, this mechanism can be used for DCA purposes just as well, and in particular, both applications may work very well together.

Mapping in virtualized environments. A context-thread-processor mapping is trivial for non-virtualized devices, as the number of contexts and threads is one. In a virtualized device, some form of table must be present, which is indexed with the context identifier and returns the related processor ID. Several logical locations of this table are possible. The processor ID field may just be integrated into the context information. Another location may be the data structures that are used by the IOMMU.

One of the main tasks of the IOMMU is to map virtual device addresses that a device is issuing to physical addresses. IOMMUs for x86 have been suggested by both AMD [42] and Intel [41]. In these architectures, the IOMMU manages a device table[1]. The entry of such a table holds, as the most important value, the root pointer of the page table that is used for address translation.

One useful application of this model is to give a virtual device the opportunity to access the address space of a process using process virtual addresses, i.e. the virtual address space of the processor and the device are identical. This can be accomplished if every virtual device has a device entry in the device entry table of the IOMMU, and the page table root pointer points to the page table root of the process. In this usage model, a processor ID or CPU ID field could be integrated into the device table entry, as shown in Figure 5-2. In contrast to other solutions, this solution could not only work in cases where the device decides based on it's functionality if a memory access should use DCA. It is also possible to use a memory mapping by marking pages table entries as DCA-enabled pages.

In the case that a device uses an IOTLB to cache translations from the IOMMU page tables, the device should cache the processor ID as well in order to avoid an IOMMU lookup.

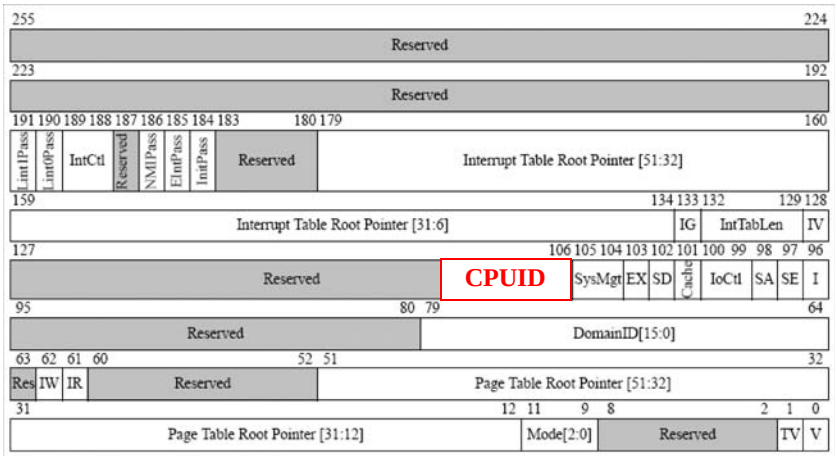


Figure 5-2. A potential integration of a CPU ID filed in a device table entry, based on the AMD IOMMU specification [42]

1. While both architectures are very similar to each other, they use different naming conventions. Here AMD's naming conventions will be used.

5.2 DCA for HyperTransport

The analysis performed above sufficiently shows on what a DCA transport protocol may build upon. Methods have been proposed in which the device or the IOMMU have the knowledge which processor will access the data.

Now, suggestions for DCA data transport mechanisms for the coherent HyperTransport network are presented. The next subsection introduces an indirect cache access mechanism, where the data is prefetched by processor logic based on prefetch hints of the device or IOMMU. Thereafter, three mechanisms for a push-style data transport into a processor's cache are presented. All three mechanisms require some packet format changes to existing packets and new commands for control packets. Drafts of these modifications will be presented, except for the Write Allocate (WRA) packet, as this is a packet of the coherent HT protocol, and the presentation in this work might violate non-disclosure agreements.

5.2.1 Indirect Cache Access via Prefetch Hint

This solution reuses the prefetching logic that is implemented in processors. Usually, these prefetching engines are triggered by regular accesses to data like burst and strides. Now, this prefetching engine is triggered by the device, which sends a prefetch hint to the processor.

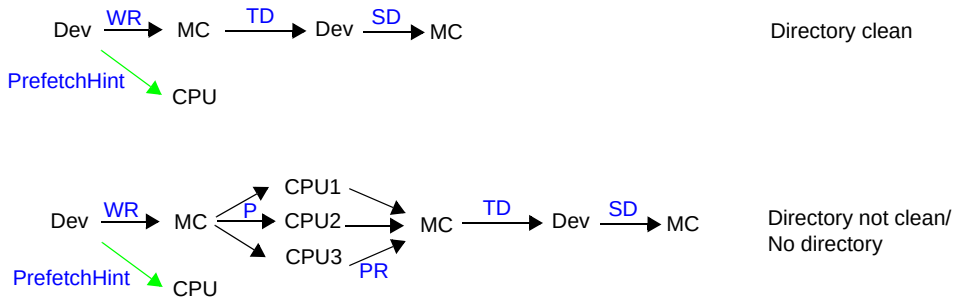


Figure 5-3. Indirect cache access via prefetch hint

An important question is at what point in time the prefetch request is sent. Lowest latency can be reached if the prefetch hint is sent directly after sending the write request. Theoretically, there is the chance that the processor prefetches the line before the write request has reached the memory controller. The most critical situation occurs if the line is already

cached, and the prefetch arrives before the invalidation from the memory controller. Experimentation or simulation must be performed to determine the best strategy.

bit time	7	6	5	4	3	2	1	0
0	Addr[7:6]		Length[2:0]			Rsv	StateHint[1:0]	
1	Addr[15:8]							
2	Addr[23:16]							
3	Addr[31:24]							
4	Addr[39:32]							
5	Addr[47:40]							
6	Addr[55:48]							
7	Addr[63:56]							

Figure 5-4. Sized-write payload for prefetch hint

In the case of the HyperTransport protocol, the prefetch hint packet can be a standard HT sized posted write packet. Thus, changes to the HT protocol are not necessary. The packet is targeted to processor address space, which may be integrated into the configuration address space of the processor. The packet payload, shown in Figure 5-4, consists of a single 64bit word, which must contain the address of the cacheline. The remaining space can be used for a coherence state hint and a length-field to allow prefetch hints for multiple consecutive cachelines at once:

- *StateHint[1:0]*: Hint to the processor in which state of the coherence protocol the line should be fetched.
- *Length[2:0]*: If >0 number of following cachelines with subsequent addresses that should be fetched as well.
- *Addr[63:6]*: Address of the cacheline that should be prefetched in a cacheline-sized granularity.

5.2.2 Direct Cache Access

Direct access to the processor cache by the device without the use of prefetch hints promises lowest latencies. However, in a system that uses the memory controllers to serialize simultaneous requests to the same memory address, any DCA transfer must also be subject to this serialization process to avoid memory inconsistencies. A direct point-to-point transfer without involvement of the memory controller is not allowed. The three following options mainly differ in how they access the serialization point.

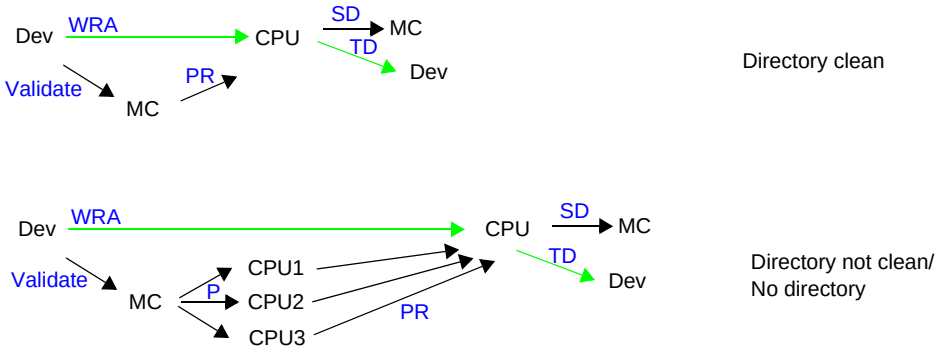


Figure 5-5. Cache update with parallel access to MC and CPU

Option one. The device sends an update directly to the target cache, and a validation request to the memory controller at the same time (see Figure 5-5). The CPU collects both the *write_allocate* (WRA) packet and all probe responses that are send to it due to the validation request of the device. However, the matching of WRA and PR packets poses some problems. First of all, probe responses are always generated as a direct or at least indirect response to a request from the processor that will later receive the probe response. Thus, matching can be done using a sequence identifier that has been assigned by this processor. The scheme presented here is different, as the request is generated by the device. Matching can only be performed if additional information is provided. Either the probe response must carry the memory address in it's header, which means a significant overhead. Alternatively, it must contain both the devices ID and it's sequence ID. Another issue is the size of the matching queue: in order to avoid deadlocks, it must be large enough so that it never blocks incoming requests. Due to these problems, this option is not a solution for DCA.

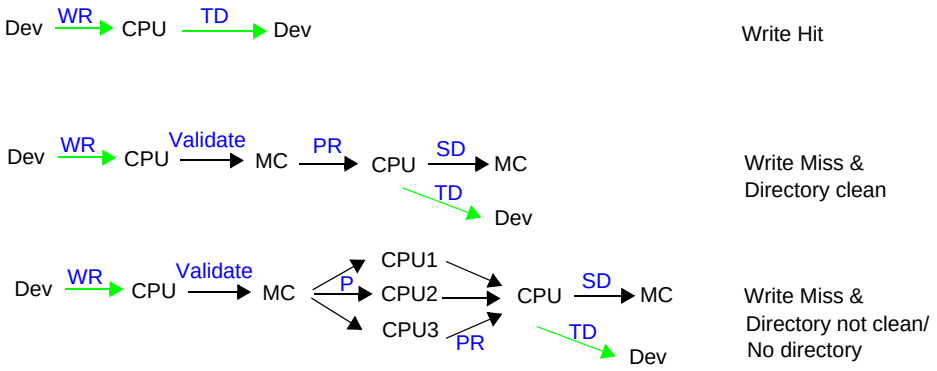


Figure 5-6. Cache update with serial access to CPU and MC

Option two. An alternative is to send data to the target cache without notifying the memory controller directly. The target cache then has to do just exactly the same as if the processor had written to the cacheline: If the line is not in an exclusive or modified exclusive state, a message has to be send to the memory controller, and eventually invalidation-probes have to be sent to other caches. A flow diagram for this case is shown in Figure 5-6.

This mechanism provides the best possible performance in the case that the respective cacheline is already present in the target cache in an exclusive state, which may be caused by a previous prefetch by the processor.

A critical issue regarding the latency of a DCA operation is that the processor is in the flow path that makes the request globally visible. Considering that DCA is an speculative optimization to speed up memory accesses that are likely to be performed in the future, it is clear that requests from the processor should have priority over DCA requests. If this leads to a situation in which DCA-initiated requests are queued at the processor, the time until the request will be observed globally increases. As a result, the write bandwidth of the device decreases if it performs write ordering.

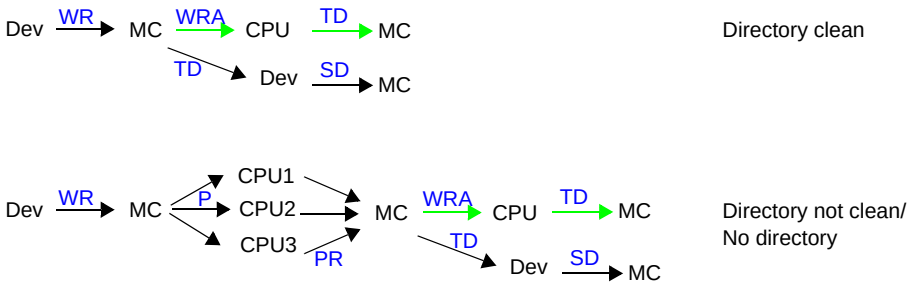


Figure 5-7. Cache update with serial access over MC and CPU

Option three. The memory controller is the target for the device's writes, as in conventional DMA. The memory controller then can update a processor cache, either with a direct update, or with a prefetch hint. A benefit of this scheme is that processors are not required to perform the requested updates to guarantee consistency.

For a mapping by the device, the format of write request packets must be modified, as it has to include the target processor ID. Current HT packet headers do not provide sufficient space for this field. In HT3-based systems, this can be done by introducing a new extension packet. This extends HT control packets to a size of 96 bit for standard HT packets using 40 bit sized addresses, and to a size of 128 bit for HT packets using 64 bit sized addresses. The extension has to be appended to the header only for such packets that should be forwarded. A draft of such an extension is shown in Figure 5-8.

- *StateHint[1:0]*: Hint to the processor in which state of the coherence protocol the line should be fetched.
- *DCA Unit ID[7:0]*: Unit ID of the DCA destination.

bit time	7	6	5	4	3	2	1	0
0	01		Cmd[5:0]= 111110					
1	rsv.						StateHint[1:0]	
2	DCA Unit ID[7:0].							
3	rsv.							

Figure 5-8. Proposed HT 3.0 packet extension for write packets with a cache hint

From this first analysis, option three seems to be the most advantageous solution. However, future work must perform an in-depth analysis and comparison of these mechanisms. Another consideration is to support processor-to-processor DCA with the same mechanism.

5.3 Related Work

A patent by Intel [61] describes the sketchy idea of a data structure that can be used for an DCA mechanisms based on address ranges. This mechanism is embedded into an IOMMU-like structure.

A dedicated network cache to speed up the receive operation in message passing NICs is suggested in [78]. The cache is parallel to the normal data cache of the processor, and can be accessed using special *network_load* and *network_store* instructions. Besides the message data, a cache entry contains the message ID, which is used to unambiguously identify the message and to bind it to it's target address. Also, it contains the network tag and process tag fields, which point to the respective network memory or process memory home address of the cacheline. Before a process performs a receive operation on a message, the memory tag is the valid pointer to the home address, after it has been received, the process tag is the pointer to the message, and buffer space in the NIC can be freed. As all 3 tags have to be searched associatively, the authors suggest an implementation as 3 different caches. In multiprocessor environments, a message predictor [79] shall be used to find the appropriate processors cache.

6 Reliability in a Direct Interconnection Network

Direct interconnection networks are the state-of-the art network topology to interconnect processors in small-scale shared memory multiprocessor systems. Besides small-scale systems, direct interconnects can also be applied to large networks, as for example in the Cray XT3 and XT4 (see Section 2.6.3), IBM BlueGene (see Section 2.6.4) or Extoll, which all employ a 3D torus topology.

Reliability of such large networks is of highest importance: either, because systems are used in environments where failures are unacceptable, as for example in banks. Another reason for the need for reliability is the sheer size of systems. The likelihood of most faults scales with the system size. Thus, the mean time between failures may be reduced to a level where a system may become useless. At the same time, a reliable network should be able to offer lowest latencies and high scalability.

With Extoll, a network has been developed that provides a reliable service. The Extoll network protocol significantly improves communication over high-speed 8b/10b encoded serial links by providing a loss-less service. A significant improvement over state-of-the-art protocols is in particular the development of a protocol that uses error correcting control characters and link based retransmission.

A second significant improvement over state-of-the-art networks is a hardware implemented mechanisms to allow deadlock-free routing in regular networks with faulty links.

This chapter describes the Extoll network and network protocol, focusing on how reliability is achieved. Section 6.1 gives an overview about the design space for a reliable direct interconnect networks. Section 6.2 and Section 6.3 detail the Extoll network, focusing on the

fault tolerance for faults on the link. An outlook about on-chip mechanisms is given in Section 6.4.

6.1 Faults

A fault is a defect or abnormal condition that potentially may lead to the failure of a system. For example, a bit flip in a memory due to cosmic radiation is a fault. Not all faults turn into an error, which is the invalid state of the system. If an error occurs which cannot be recovered, the system has a failure.

Hardware faults, i.e. faults that are not caused by software behavior, can be categorized into hard faults and soft faults. In a hard fault, the physical hardware is broken and does not operate in the way it is supposed to. Typical hard faults in computer networks are power failures of individual nodes and link cable faults.

Soft faults are transient faults. They affect only the information that is stored on the chip or link. As soon as the system has been brought into a valid state, which in the worst case may require a reset, it continues to operate normally.

Faults can be resolved at different levels. Hardware bit faults for example can be efficiently corrected locally in hardware. Even some hard failures, as a defective memory cell, may be resolved efficiently in hardware. Other faults must be treated on higher levels. In the end, the highest tolerance against faults can only be achieved by high-level mechanisms as checkpointing and redundant processes and nodes. However, there are strong reasons to treat at least some faults directly in hardware:

- Methods like checkpointing or high-level redundancy are relatively expensive, even in the absence of faults. If faults can be treated more efficiently on lower layers, the performance/price ratio may be significantly better if the system relies on these mechanisms and reduces high-level mechanisms.
- From Amdahl's law, it follows that faults that occur frequently should be resolved faster for a better system performance. The resolution of rare faults may be expensive but still will not affect performance significantly. Due to the ever increasing system sizes and speeds, in particular transient faults are becoming much more likelier, and the need for an efficient resolution is increasing.
- The detection of a fault must occur in any case. Depending on the type of fault, the immediate resolution in hardware may not cost significantly more.

Relevant faults that may occur in the Extoll network are:

- Transient faults with bit errors on the links. As long as components of the link do not have design flaws, these faults have the characteristics of Gaussian noise.

- Other transient faults on the links, e.g. burst errors when the bit synchronization at the receiving side of a serial link is lost.
- Transient faults with bit errors on chip.
- Permanent faults of links, e.g. if cables are broken or accidentally unplugged.
- Permanent faults of Extoll chips. Except for the burn-in and wear-out phases, the only relevant hard faults are power faults. These can efficiently be reduced by using redundant power supplies. If a power failure still occurs, it will affect not only the Extoll chip, but the entire node. Thus, such a failure must be resolved at a higher level. The network only has to ensure that routing around the faulty components is possible.

6.1.1 Units

In contrast to the burn-in and wear-out phases, the failure rate for most components in a digital system during the useful life period is constant. The occurrence of failures in any system or subcomponent is frequently measured as failures in time (FIT):

$$1FIT = \frac{1failure}{10^9 hours}$$

The reciprocal value of the failures in time is the mean time between failures (MTBF). Together with the mean time to repair (MTTR), the availability of a system can be specified as:

$$Availability = \frac{MTBF}{MTBF + MTTR}$$

Besides the availability of a system, the correctness of results is an important property of a system. This section aims to increase the availability of the system by increasing the MTBF due to soft errors either in chips or on links, and at the same time to ensure correctness.

If the probability of errors depends on the amount of data that is processed or transmitted rather than on the time that passes by, an error ratio is a better measuring unit than FIT. The bit error ratio (BER) for the transmission over a physical media is such a parameter: it is the ratio of erroneous bits per transmitted bits. More frequently, BER is translated as bit error *rate*. This reflects the fact that bit errors are caused mostly by random noise. In such systems, the BER equals the bit error probability *p(e)*. The BER can be converted to FIT if the data frequency is known:

$$BER = \frac{FIT}{10^9 \cdot 360} \cdot \frac{1}{FREQ}$$

6.1.2 Soft Error Nature and Rates

Today, the most important source of soft errors on chips are cosmic particles. High energy neutrons may produce ions when they hit silicon nuclei. These ions may change the charge especially of reverse-biased junctions [146]. Figure 6-1 shows that the cosmic ray flux increases with the height above sea level. Soft errors due to radiation thus increase by the same amount.

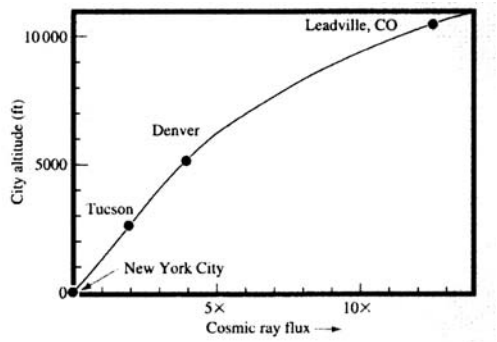


Figure 6-1. Cosmic ray flux increases with the altitude [148]

Another source of ions are radioactive impurities in the packaging materials. The same applies for solder bumps, although lead free solder has much lower impurities than leaded solder. In the past, neutron interactions with borophosphosilicate glass (BPSG), a material that has been used to form insulator layers, also were a source of ions.

Soft errors caused by radiation affect DRAM, SRAM, sequential and combinational logic. In memories, more than one cell may be hit by a single event, causing a multi-bit error [154]. Therefore, RAMs are usually organized so that physically adjacent cells are not logical adjacent, so that only single bit errors occur per word.

Flip-flops and logic nets are also subject to soft faults caused by radiation. In logic nets, such an event may cause a glitch. In sequential logic, the logic net drives one or more flip-flops, either directly or indirectly over other logic nets. A fault only occurs if a flip-flop samples the glitch. The strength of the glitch is determined by the capacitance of the net. Timing critical paths are more affected. Currently, it is being assumed that the error rate in static logic is significantly lower than for sequential logic.

The most interesting question is how the soft error rate will impact future chip designs. Ever-shrinking geometries and voltages lead to lower charges in logic and memory. Thus, an ion's impact becomes more important. Against that works that logic elements occupy a smaller area, which decreases the chance of a hit. Figure 6-3 shows how the soft error rates

for SRAM scale with technology. It shows a steady increase of the system SER, which is mainly caused by the growing size of systems and a nearly constant bit SER. Hazucha et al. [147] find a similar development, with a slight increase of bit SER by 18% per technology generation. In contrast to SRAM or logic, the DRAM vulnerability is decreasing constantly when measured per bit, as shown in Figure 6-2.

A 6 mm^2 example in a modern 90nm logic CMOS process[1] based on the vendors SER specifications shows that raw FIT rates are around 13 FIT/kbit for flip/flops, and around 8 FIT/kbit for SRAM at an altitude of 300 m above sea level. Thus, for a chip with 10 kbit of flip-flops and 32 kbit SRAM memory, the FIT rate is 385 FIT. In a system with 1000 chips, an upset in the SRAM occurs every $3.85*10^4$ hours, i.e. every 1.2 years.

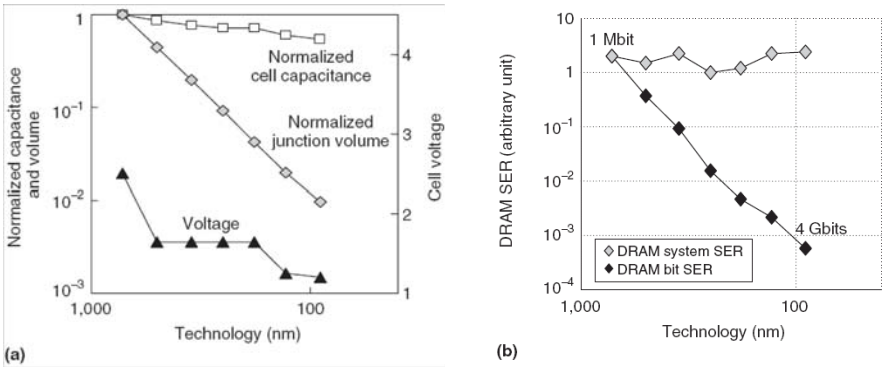


Figure 6-2. Soft fault rate scaling for DRAM [146].

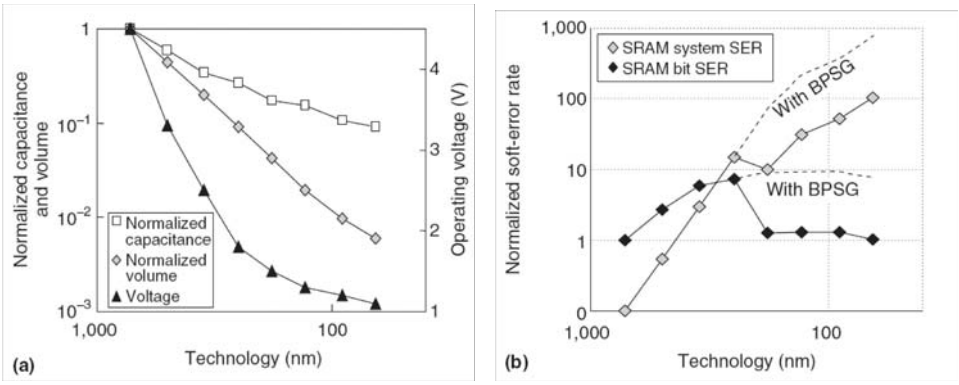


Figure 6-3. Soft fault rate scaling for SRAM [146]

1. Such data is strictly confidential. Thus, more details cannot be given here.

Bit errors on paths between chips. Noise on cables or traces between chips can also lead to soft errors. All components on the path between two chips have an influence on the error rate: transmitters and receivers on the chip, the package, PCB traces, connectors and cables. Thus error rates depend on the detailed configuration of the system and can only be obtained by detailed measurements in the system [141]. Extoll is not limited to one such configuration, instead, a variety of configurations is thinkable: electrical or optical high speed serial transmission over cables, parallel electrical transmission using low-voltage digital signaling (LVDS) or backplane transmission. Thus, it is sufficient to use the rule of thumb estimation that raw bit error rates are in the order of 10^{-12} to 10^{-15} for cable based transmission, as for example in an optical cable that is specified for up to 20 GBit/s [143].

With an BER of 10^{-15} on a single 10GBit/s link, a bit error will occur every 10^5th second, i.e. ever 27 hours. A 1000-node 3D torus topology has 6000 unidirectional links, a link bit error occurs every 16 seconds. Assuming that bit errors are unrelated, multi-bit errors in data words are rather unlikely. With the same BER, the probability of a two bit error $w=2$ within a code word size of $n=10$ bit and $p_s=BER$ is:

$$p = \binom{n}{w} p_s^w (1-p_s)^{n-w} = 4{,}5 \cdot 10^{-29}$$

In the above example, this occurs every 1.1 billion years. It can be concluded that single bit error correction is essential for systems with a larger number of links. Double-or more bit detection is not necessarily required when random bit faults are assumed.

Obviously, bit errors on the link are by orders of magnitude more frequent than SEUs on chip. Error correction on links is in any case required. However, such rules of thumbs must be treated with care.

Which faults turn to errors? A classification of the outcomes of a bit fault is shown in Figure 6-4. This classification is a modified version of the classification introduced in [149]. Not every fault leads to an error and failure of the system, even if it cannot be corrected. Such a fault is called benign fault. An example for such an fault is a bit error in the empty part of a FIFO queue. As such an empty entry will never be read and used, this fault does not generate an error. If a faulty bit is used and the fault is relevant to the program or application and cannot be corrected, an error occurs. Such errors can be classified into detected errors, and errors that silently corrupt data.

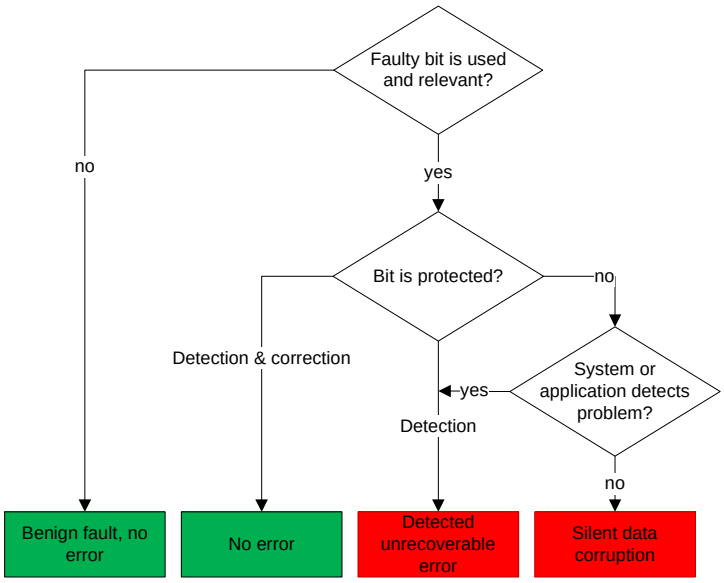


Figure 6-4. Classification of the possible outcome of soft bit faults

6.1.3 Error Correcting and Detecting Codes

Bit error detection and correction both rely on redundancy, i.e. they require additional resources. For digital systems this implies that information in this system is represented, or coded, in a redundant way. Depending on their capabilities, these coding schemes are called error detection codes (EDC) or error correction codes (ECC).

A code consists of a set of code words. The number of bits in which two distinct code words differ is called the Hamming distance (HD) of those words. Increasing the Hamming distance between two valid code words, and thus adding redundancy in the code, means that an increased number of bits may change without flipping one code word into the other.

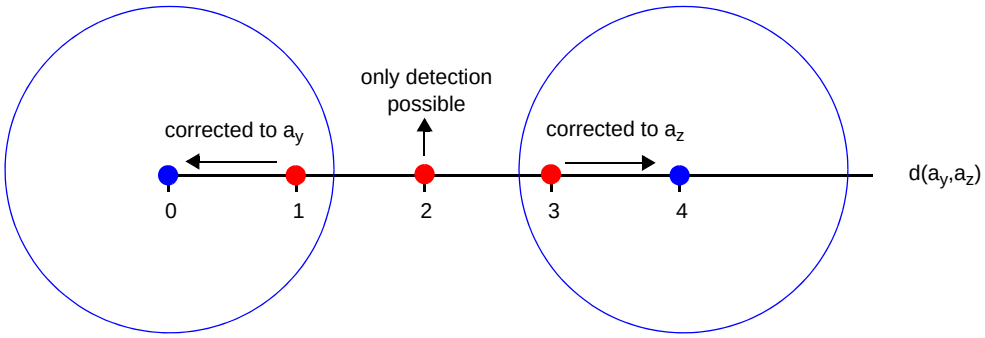


Figure 6-5. Geometrical interpretation of Hamming distances

The minimum of the Hamming distances between every single pair of words in a code is the minimal Hamming distance of a code, often simply called the Hamming distance of this code. A code that is able to detect all errors with f_e faulty bits must have a minimal Hamming distance d_{min} of at least:

$$dmin_{det} = f_e + 1$$

To correct a code word in which some bits have been altered, this erroneous word can be mapped to the valid code word with the lowest Hamming distance. In a geometrical interpretation, all words on the cloud around a_y in Figure 6-5 will be mapped to this code word. Thus, the minimum Hamming distance must be greater to assure that the clouds around different valid code words do not intersect with each other:

$$dmin_{corr} = 2 \cdot f_e + 1$$

Often, codes are used that correct errors with up to f_{ec} faulty bits, while at the same time detecting errors with $f_{ed}>f_{ec}$ bits. The most popular codes from this group have single error correction, double error detection (SECDED) capabilities. Although a *dmin* for this case is usually not given in the literature, it can easily be constructed: In the first step, a SEC code is constructed from the original code. In the second step, a single error detection (SED) code is applied to the SEC code. Thus, the required Hamming distance is:

$$dmin = (f_{ed} - f_{ec}) + 2 \cdot f_{ec} + 1 = f_{ed} + f_{ec} + 1$$

This approach to classify error correcting codes is very useful if errors that may occur in the system are rare and by their physical nature limited to a fixed burst length. In particular, single bit errors are a frequent phenomenon of errors in systems.

All error correcting codes have to introduce a minimum redundancy to reach a given $dmin_{det}$ or $dmin_{corr}$. With a given code word length l, and the number of redundant positions k,

the error correcting code will have the length $n=l+k$. The following equation shows how k can be computed for a desired *dmin*. The derivation of this equation can be found for example in [134].

$$2^k \geq \sum_{i=0}^{\left\lfloor \frac{dmin-1}{2} \right\rfloor} \binom{l+k}{i}$$

Besides correcting all bit errors that are below the Hamming limit, codes differ in how good the can detect and/or correct errors with a higher number of faulty bits. This is of importance for systems with other fault models, as burst faults for example.

While a large variety of error correcting or error checking codes exists, only a small selection adds little redundancy while at the same time having a low coding and checking complexity. A low complexity is important for the implementation in timing critical hardware components.

Hamming Code. The Hamming code is a linear block code, i.e. it encodes fixed length channel words. It has a Hamming distance of *dmin=3*, and corrects single bit errors. The number of check bits that are required can be determined by evaluating the equation above, which simplifies to: $2^k \geq k + l + 1$. The extended Hamming code has a *dmin=4* and adds the capability to detect two-bit errors. The construction of these codes for a given l can be found in the literature [134][135]. [142] shows that for a system with a purposely increased raw BER of $3.2*10^{-9}$, Hamming coding increases the BER to $3.86*10^{-16}$.

Cyclic Redundancy Codes (CRC) are cyclic codes over the Galois field of two elements *GF(2)*. CRCs are used for error detection. The CRC algorithm can be described as a polynomial division in *GF(2)*. Binary data is represented as a polynomial, where the single bits of the word are the coefficients. For example, the word '10011101' is represented as '$x^7+x^4+x^3+x^2+x^0$'. The data word *u(x)* is divided by a generator polynomial *g(x)*. The remainder of this division is the CRC value. This value is appended to the data word to form a code word. A CRC check consists of a recomputation of the CRC of the data word part of the code word, and a comparison with the CRC value in the code word.

A commonly used notation is CRC*l*, where l denotes the most significant term for which the coefficient is 1. CRCs are capable to detect any burst errors where the distance between the first and the last erroneous bit is smaller or equal to l.

Finding the best CRC polynomial and length for an application is generally difficult, as the quality depends on the data word length and a Hamming distance requirements. Evaluations of polynomials can be found in [136][137].

A straightforward hardware implementation of the CRC calculation is a linear feedback shift register (LFSR) as shown in Figure 6-6. The drawback of this implementation is that

only one bit of the data word can be shifted into the register at a time. Parallel implementations can be constructed by expanding the equations for multiple clock cycles [138], which can be done automatically in the HDL code [139]. With additional parallel logic, the timing critical path can be reduced to one XOR gate [140].

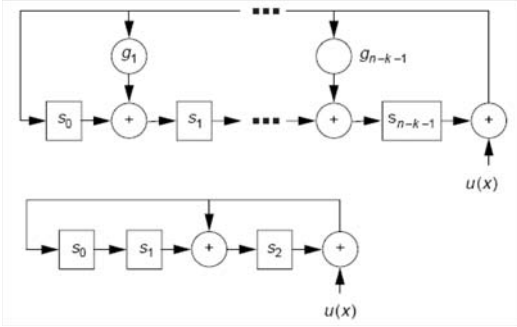


Figure 6-6. Linear feedback shift register for $g(x) = x^3 + x^2 + 1$ [139]

6.1.4 SEU Tolerant Design

On chip. Logic on a chip can be differentiated into control logic and data paths. The treatment of bit faults that are caused by single event upsets is different for both types, as shown in Figure 6-7. Control logic can be replicated on a module level. The most frequently used replication is triple modular redundancy (TMR), which has originally been proposed by von Neumann [144]. Here, three instances of a module work in parallel, and a "majority organ" determines the output that is generated by the TMR block. If information redundancy is used, fault checking and fault correction state machines can be constructed, for example using Hamming-coded state vectors [145].

For data paths, a triple redundancy does not offer any benefits over a correction scheme like Hamming, but requires significantly more resources. Thus, error correcting codes are usually employed to protect data paths. Theoretically, backwards error correction (BEC) protocols could be used in chip for communication between components that are protected otherwise. In the general case, this would drastically increase communication protocol overhead and lead to unpredictable latencies of data paths. At the same time, additional buffers and acknowledgements cannot decrease resource utilization compared to FEC mechanisms.

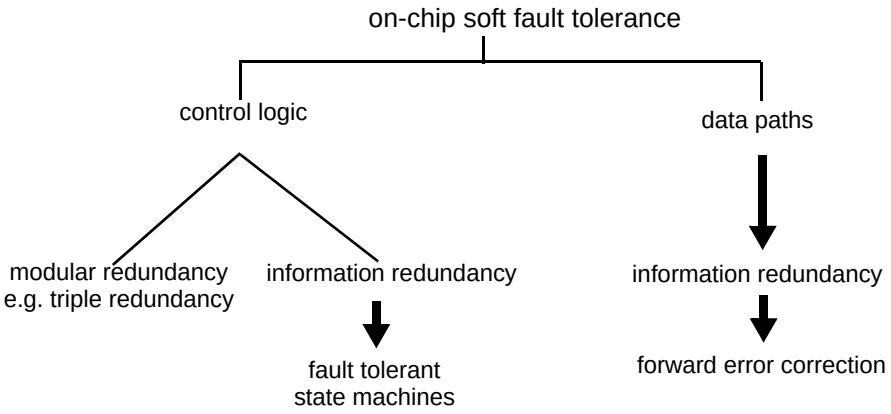


Figure 6-7. Chip soft fault tolerance design space

Links. Soft fault correction on links is somewhat different. While the physical link itself is always a data path without logic, data that is transmitted on the link can be distinguished into payload data and link control information. Control information is such data that controls the flow of payload data and thus determines the state of both the transmitting side and the receiving side. For both, FEC and BEC mechanisms are possible, as shown in Figure 6-8.

Both FEC and BEC mechanisms consume link bandwidth, the total occupied bandwidth for FEC codes is the sum of the codeword length *l* and the number of redundant positions *k*:

$$N_{FEC} = l + k_{FEC}$$

For BEC mechanisms, a part of the overhead is caused be the retransmission and the transmission of acknowledgements. However, for the low BERs that are present, the influence of the retransmission can be neglected:

$$N_{BEC} \approx l + k_{BEC} + l_{Ack}$$

Thus, the overhead of both mechanisms depends only on the bits that are appended for correction and to acknowledge the reception. For very small *l*, for example a link character, an correcting code typically has less overhead than a BEC approach, as the acknowledgement will have about the same size *l*. As k_{FEC} grows faster than k_{BEC} with *l*, backward error correction will have less overhead for a large *l*. However, assuming 64byte data words, a FEC using a Hamming code with $k_{FEC}=10$ has still less overhead than a BEC with a parity bit $k_{BEC}=1$ and an acknowledge character size of 18 bits. But a FEC has some major limitations:

- If a low latency hardware implementation is the goal, scalability of *l* is limited by computational complexity. Also, a check can only be performed if all data of the code word is present at the same time.
- FEC protocols cannot correct words that get lost either completely or partially, which may happen in cable transmission.
- If line codes like 8b/10b are used, an FEC may be difficult or impossible to implement (see Section 6.1.6).

As a result, BEC is being used for bit error correction in Extoll, while small control words are protected using forward error correction.

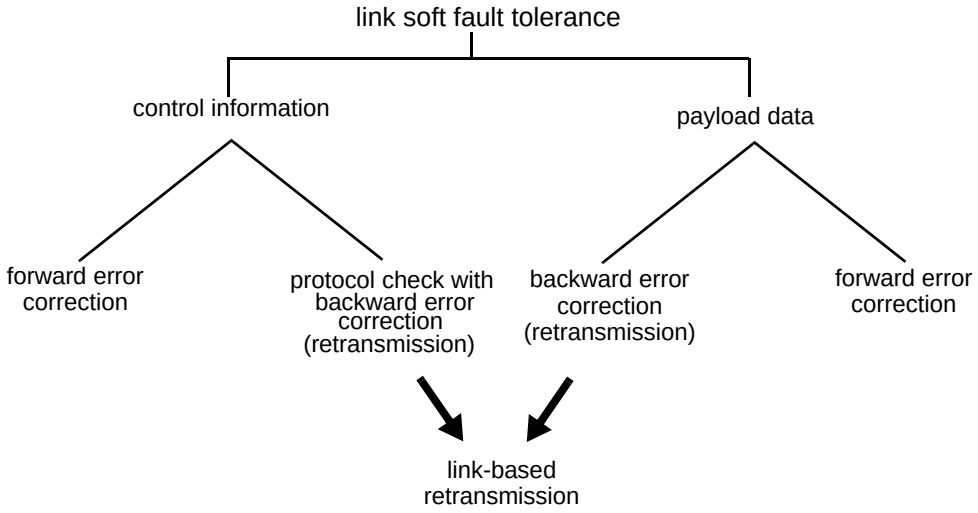


Figure 6-8. Link soft fault tolerance design space

Besides the overhead on the link, backward error correction requires buffer space on the transmit side of a link. The size of the retransmission buffers depends on the link round-trip latency t_{rt} in clock cycles:

$t_{rt} = 2(t_{propagate} + t_{logic}) + t_{insert} + t_{flit}$, with $t_{propagate} = \frac{v_{signal}}{t_{cyc}} \cdot l_c$.

The propagation delay $t_{propagate}$ depends on the length of the cable and the velocity of the electrical or optical signal, which can be estimated to v_{sp}=*3.3ns/m*. All other delays occur on chip and are directly measured in clock cycles. Logic delays of the transmit and receive side are summarized in t_{logic}. The length of a flit in clock cycles is t_{flit}. On the receiver side, the link may be busy, so that an acknowledgement may be inserted only after t_{insert}. In the

case of Extoll, acknowledgements may be inserted only in between flits, so in the worse case $t_{insert} = t_{flit}$.

The required size of buffer space is then t_{rt} times the width of the link. For an FPGA-based Extoll implementation with serial links, $t_{cyc}=100ns$, $l_c=20m$, $t_{logic}=24\ cycles$ which is mainly caused by the serial transceivers, $t_{insert} = t_{flit} = 32\ cycles$, $t_{rt} = 140\ cycles$, so that buffers must have a minimum size of 280 bytes, which corresponds to the size of 4.4 flits.

6.1.5 Retransmission Endpoints

A link-based retransmission protocol scales with the network size, as every link that is added to the network brings all resources with it that are required to fully utilize the link. Naturally, it protects only against errors on the link.

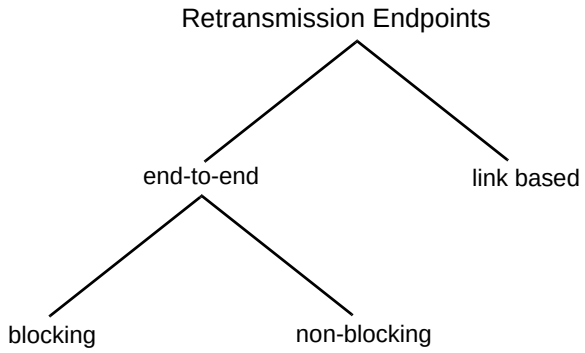


Figure 6-9. Retransmission in networks

In contrast, an end-to-end retransmission protocol covers the complete path between two communicating nodes, and thus may protect against all types of errors on the path in between. End-to-end retransmission can be either blocking or non-blocking.

A blocking retransmission delays the termination of a network transaction. Depending on the type of transaction, this may delay communication processes. Nevertheless, it is frequently being used, as no or little hardware is required to implement this protocol. The Extoll functional units implement such an end-to-end acknowledgement, which can optionally be switched on.

A non-blocking retransmission between end nodes does not introduce this delay. It works very similar to the link retransmission: in both protocols, transmitted data is buffered in a dedicated buffer until the reception has been acknowledged. To fully utilize the bandwidth of the network, buffers must be designed so that they can hide the round-trip latency to all other nodes in the network. This includes link delays, on-chip delays, and delays through

congestion. Thus, the scalability of this solution is limited: if the network is extended, buffers must be sized up.

However, the biggest problem of end-to-end protocols are the acknowledgements. They must carry a sequence ID, and must be routable. Thus, they are flits on its own, consuming link and crossbar bandwidth. Thus, acknowledging flits is inefficient. Instead, packets are acknowledged. For small packets, for which Extoll is optimized, this does not significantly improve the situation. For larger packets, this increases the round-trip latency and thus the required end-to-end retransmission buffer sizes for a non-blocking end-to-end retransmission protocol.

As a consequence out of these considerations, Extoll uses link-based retransmission.

6.1.6 Serial Transmission

The Extoll protocol is designed so that parallel and serial links are supported. In a parallel link, data is transmitted over parallel lines. Additional lines carry control and clock signals. In a serial protocol, these additional signals have to be multiplexed together with the data signals onto one single line. Thus, a line code has to be used. The code must assure that there are sufficient transitions between 0 and 1 in the transmitted code words.

Also a much higher data signaling rate is being used on serial links. This is done using AC-coupled transmission. The generation of very low frequency patterns on the link must be avoided. This is also called a direct current (DC) free transmission. This can be achieved when the line coding ensures that the number of 0's and 1's is equal within a short timeframe. The timeframe must be shorter than the controlling interval of the receiving amplifier.

Another important issue in serial transmission is the alignment of the serial data stream to word boundaries. Here a line code may provide alignment information in the code words. The 8b/10b code has a set of comma symbols which can be used for alignment, due to their unique bit patterns with respectively 5 consecutive 0's or 1's. However, there is no real need for such comma symbols. The alignment must be found once at the initialization of the link. Afterwards, it simply must be ensured that the clock is recovered properly. Thus, it is better to find the alignment at link initialization using longer training patterns.

The standard line code for serial transmission is the 8b/10b transmission code [150], which is being used for example in PCI Express, Infiniband, Gigabit Ethernet and HyperTransport 3.0.

In the 8b/10b code, an 8bit data word is translated into a 10 bit line code word, also called character. The difference between the number of 0's and 1's is called the disparity of a code word. Some 8 bit words have a translation with a disparity of zero. All others have two

translations: one with a disparity of +2, i.e. with six 1's and four 0's. The second translation is the inverted code word, which then has a disparity of -2. The running disparity is the sum of the disparities of all words that have been sent in the past. The running disparity will be either +2, 0, or -2, i.e. the link is completely DC free in the long term. This can be achieved in the coding step by selecting the code word with the inverse disparity than the current running disparity. Also, every two code words that are transmitted one after another will have a maximum combined disparity of $|2|$. Another feature of the code is that a sequence of more than 5 1's or 0's cannot occur in any combination of code words.

Besides the 2^8 data words, there exist 12 control characters. The naming conventions for data and control characters are D.*x*.*y* and K.*x*.*y*, with $0 \le x \le 31$ and $0 \le y \le 7$. 8b/10b coding is performed by coding the first 5 bits with 5b/6b coding, and the remaining 3 bits with 3b/4b coding. *X* and *y* denominate the respective 5b/6b and 3b/4b characters. K.28.1, K.28.5 and K.28.7 are comma characters. They are the only ones with the sequence "1100000" and "0011111" and thus can be used to align the serial stream to word boundaries.

In the context of fault tolerance, an important topic is how bit errors on the 10b characters behave and how they can be detected at the receiver. Although the 8b/10b coding is widely used, I could not find an in-depth analysis of the 8b/10b protocol in the presence of transient bit faults in the literature. Thus, the following analysis is a significant contribution that revalues the behavior of the 8b/10b protocol in the presence of faults.

Some errors in the 10b domain will turn characters into invalid characters. In this case, the 8b/10b decoder detects an out-of-table error. As a single bit error changes the disparity of a code word, other single bit errors will be detected by the disparity check. However, the disparity check may not precisely detect which character was the faulty one: characters with a zero disparity delay the error detection. Also, multi bit errors, either within a single character or in adjacent characters, may not be detected by checking the disparity.

State of the art 8b/10b based transmission relies on the checks for out-of-table and disparity errors. If an error occurs, a possible solution is to re-initialize the link, as for example HyperTransport 3.0 does.

Hamming Distances of Control Characters. The set of control characters can be used to encode protocol information. For fault tolerance, large Hamming distances are desirable. However, Figure 6-10 shows that Hamming distances vary. An important character is K.28.7, as this is the only character that cannot be turned into a data character by a single bit error. A maximum sized set including K.28.7 that has a minimum Hamming distance of 3 and thus can correct single bit errors is {K.28.7, K.27.7, K.28.3}. This set is being used in the Extoll protocol. If error correction among K characters does not play a role, a larger

set with HD=2 can be constructed: {K.23.7, K.27.7, K.28.1, K.28.2, K.28.3, K.28.5, K.28.6, K.29.7, K.30.7}.

	K_23_7	'K_27_7'	'K_28_0'	'K_28_1'	'K_28_2'	'K_28_3'	'K_28_4'	'K_28_5'	'K_28_6'	'K_28_7'	'K_29_7'	'K_30_7'
K_23_7	0	2	4	5	3	3	4	5	3	4	2	2
K_27_7	2	0	4	5	3	3	4	5	3	4	2	2
K_28_0	4	4	0	3	1	3	2	3	1	2	4	4
K_28_1	5	5	3	0	2	2	3	2	4	1	3	3
K_28_2	3	3	1	2	0	2	3	4	2	3	5	5
K_28_3	3	3	3	2	2	0	1	2	2	3	5	5
K_28_4	4	4	2	3	3	1	0	1	1	2	4	4
K_28_5	5	5	3	2	4	2	1	0	2	1	3	3
K_28_6	3	3	1	4	2	2	1	2	0	3	5	5
K_28_7	4	4	2	1	3	3	2	1	3	0	2	2
K_29_7	2	2	4	3	5	5	4	3	5	2	0	2
K_30_7	2	2	4	3	5	5	4	3	5	2	2	0

Figure 6-10. Hamming distances of 8b/10b control characters in the 10b domain

General Fault Correction or Detection mechanisms. When fault detection or correction methods like Hamming codes or CRCs are used for additional protection of transmitted data, these can be applied either on the unencoded binary data, or on the line code (see Figure 6-11). Checks then will be performed either in the 8b domain or in the 10b domain.

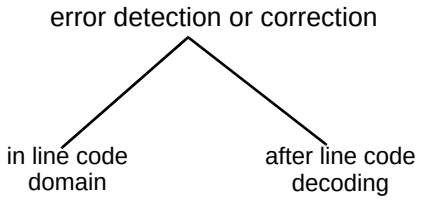


Figure 6-11. Error detection or correction for line codes

Checks in the binary coded domain. Systems usually perform error detection or correction in the normal binary coded domain. This is the most flexible solution, as the check may be performed at any place in the normal binary coded domain. In particular, end-to-end checks in software are possible. However, bit errors on the line code words may cause more complex error patterns in the binary domain. Figure 6-12 shows that single bit errors on 10b encoded words may cause bit errors of up to 5 bits in the 8b domain, even though the probability for 5bit errors to be generated is very low.

This means that an SEC code would have to provide a Hamming distance of 5+3=8: error correction becomes very inefficient. Error detection also becomes less efficient. In particular, CRC checks on data that have been designed to reliably detect bit faults of up to *n* bits may fail to do so. However, due to the burst error detection capability of CRCs, a 16 bit

CRC can detect all bit errors within adjacent 16bits, and thus detects all errors that occur within two adjacent 10b characters. Thus, CRC calculation in the binary domain is still very valuable.

	HD>1	HD in 8bit domain							
		1	2	3	4	5	6	7	8
Number of 10b word pairs*	141044	685	660	396	285	3	0	20**	8**

*= the two different encodings per character have been treated as different words
**= these pairs consist of one K and one D character each - faults thus can be distinguished by the protocol

Figure 6-12. 10b word pairs with a Hamming distance of 1 and their Hamming distances on the 8bit domain

Checks in the 10b domain. Error detection and correction in the 10b domain would circumvent the previously described problem of error multiplication. Basically, the design space shows two potential approaches to do so:

- Error control bits may be inserted into the 10b stream after the 8b/10b encoder of the sender. On the receiving side, these characters must be removed before feeding them into the decoder. One proposal [158] is to add FEC bits in the 10b coded stream. In order to not destroy the 8b/10b's guarantees like DC-balance, every inserted FEC bit is directly followed by its complement. They protect every 8 10b words using 8 FEC bits. As the 8 complement bits are also included, the overhead of this mechanism in terms of bandwidth is 17%. Error correction can only be performed in the 10b domain on the complete block of 96 bits, which adds additional latency.
  The insertion of a packet based CRC directly into the 10b domain would avoid the problem of latency, and reduce the bandwidth overhead. In such a solution the CRC must be extracted based on the protocol, which essentially means that the link port logic must operate in the 10b domain. The 10b CRC calculation may protect packet payloads better against multi-bit errors, but not control information that is outside of packets.
- Error control information is inserted in the 10b domain as well, but in the form of valid 10b characters. The author tried to map a Hamming (7,4) code into 10b characters: this is simply not possible due to the restrictions of the 10b coding space. A different approach is to search the set of 10b characters for a set of characters that has a minimum Hamming distance. Using a brute force search, a set of 16 D characters with a minimum HD=4 has been found (see Figure 6-13). As there is no intrinsic logic function to correct errors, a lookup based decoding must occur.
  The combination of this set of D characters with a set of K characters can be used to

construct control words with a large Hamming distance. This is done for the Extoll protocol (see Section 6.3.2). The set can be used to correct single bit errors in 10b characters.

	D_14_1	D_20_1	D_20_6	D_22_3	D_28_2	D_28_5	D_6_2	D_14_6	D_25_3	D_17_5	D_3_6	D_5_3	D_10_3	D_3_1	D_11_2	D_17_2
D_14_1	0	4	8	4	4	4	4	4	6	8	8	6	4	4	4	8
D_20_1	4	0	4	4	4	4	4	8	6	4	8	4	6	4	8	4
D_20_6	8	4	0	4	4	4	4	4	6	4	4	4	6	8	8	4
D_22_3	4	4	4	0	4	4	4	4	4	6	6	4	4	6	6	6
D_28_2	4	4	4	4	0	4	4	4	4	8	8	6	6	8	4	4
D_28_5	4	4	4	4	4	0	8	4	4	4	8	6	6	8	8	8
D_6_2	4	4	4	4	4	8	0	4	8	8	4	4	4	4	4	4
D_14_6	4	8	4	4	4	4	4	0	6	8	4	6	4	8	4	8
D_25_3	6	6	6	4	4	4	8	6	0	4	6	4	4	6	4	4
D_17_5	8	4	4	6	8	4	8	8	4	0	4	4	6	4	8	4
D_3_6	8	8	4	6	8	8	4	4	6	4	0	4	4	4	4	4
D_5_3	6	4	4	4	6	6	4	6	4	4	4	0	4	4	6	4
D_10_3	4	6	6	4	6	6	4	4	4	6	4	4	0	4	4	6
D_3_1	4	4	8	6	8	8	4	8	6	4	4	4	4	0	4	4
D_11_2	4	8	8	6	4	8	4	4	4	8	4	6	4	4	0	4
D_17_2	8	4	4	6	4	8	4	8	4	4	4	4	6	4	4	0

Figure 6-13. Set of 16 D characters with a minimum HD=4

Alternatives. An alternative to 8b/10b is a combination of 64b/66b coding and scrambling, which is being used in 10 Gigabit Ethernet. While it has less overhead than 8b/10b coding, it also guarantees a transition between 0 and 1 only every 66 bit. By means of scrambling, it is considered likely that there are enough additional transitions in the code words to recover a clock, and also to be DC free. As a self-synchronous scrambler is being used, bit errors in the 66b domain do not convert into multi-bit errors in the 64b domain. A coding like 64b/66b is not considered for use in Extoll, as the minimum data unit on the link has a size is 3x 64 bit. A different 64b/66b protocol could have been developed that does not have this restriction. However, a 64bit code word width increases latency on the transmit side if the internal with is much smaller. Also, more should be known about the reliability of such a coding for use in a loss-less network. Due to the previously described problems with 8b/10b coding and the overhead of 20% that is introduced by this coding, future work should closer evaluate this type of coding.

6.1.7 Faults in Regular Networks

Routing in topologies like 3D tori is optimized for the specific topology. In particular, deadlocks are usually avoided by restricting the number of allowed routes between nodes.

Figure 6-14 shows a network using dimension order routing in a 2D grid topology. In dimension order routing, all routing steps into the X direction must be performed before any routing step into the Y dimension is allowed. Thus, there is no legal path from the sending node to the receiving one.

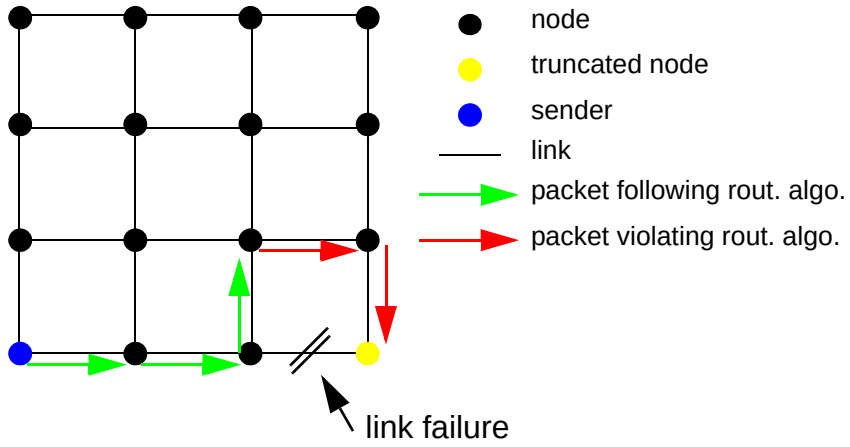


Figure 6-14. Deadlock-free routing violation due to link failure

One solution to this problem is to change the routing algorithm so that irregular topologies are assumed. This usually results in a much lower performance throughout the network. In large networks, there is a good likelihood that there is at least one failed link or node in the system at any given time, so that the network will rarely work using the optimal routing algorithm.

Caused by this problem, many techniques have been proposed to avoid this solution in wormhole-routed networks. Instead, these techniques concentrate to allow packets on paths that are affected by the failure to take paths through the network that violate normal routing rules, but still do not cause deadlocks. Most route schemes introduce additional virtual channels to allow to route around a fault [152][153][155]. These additional virtual channels have the drawback that they require additional buffer capacity. Also, the complexity of virtual channel handling in a switch is relatively high, as can be observed in the Extoll switch.

A different approach is to partition the path into as many partitions that every partition is deadlock free in itself (see Figure 6-15). Packets are extracted from the network at every node at a partition boundary, called intermediate node, before it is inserted again. As result, the complete route is deadlock free as well. Multiple intermediate nodes may be used.

In a software-based solution [156], such packets are extracted at intermediate nodes by writing them to the system's main memory. Software must reinject these packets into the network. Thus, this mechanisms has a relatively high latency. [157] proposes to use a set of nodes that act as intermediate nodes only, called lamb nodes.

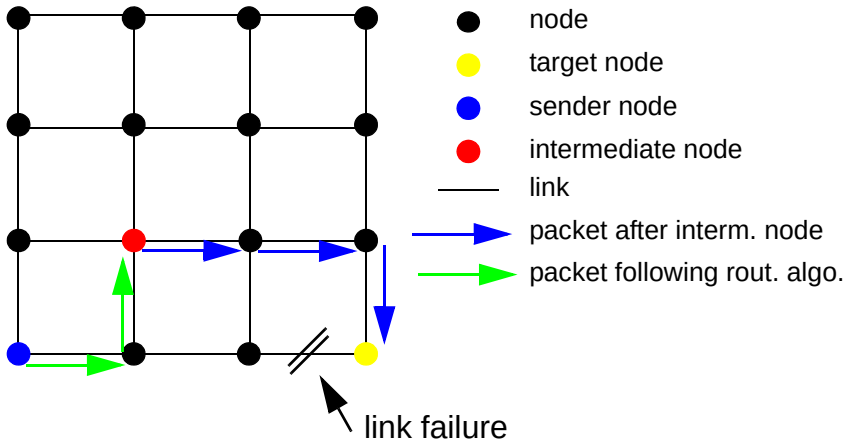


Figure 6-15. Fault-tolerant routing over intermediate nodes

However, the software-based solution can be significantly improved, when the following points are considered:

- In the case of static source path routing, complex routing decisions have not to be performed at the intermediate nodes. Thus, the control logic can be implemented in the NIC in hardware.
- It is not necessary to remove packets completely from the network. In particular, a store-and-forward architecture is not required. Instead, an intermediate node can directly forward a packet. Deadlocks can sufficiently be avoided if the tail of a packet is removed from the network in the case the head of the packet blocks.
- Especially for small packets, fast NIC internal buffers may be used for extracted packets. If this buffer fills, the node's main memory must be used to store extracted packets. The on-chip buffer may hide the memory access latency completely in this case.

Such an hardware implementation is the HAP of Extoll, descried later in Section 6.3.6.

6.2 The Extoll Network

Extoll is a direct interconnection network (IN), and thus connects computing nodes in an Extoll network without the need for centralized switches. With six bidirectional links per node, the target network topology for Extoll is a 3D-torus, as shown in Figure 6-16. Extoll

is based on the experiences made with the Atoll network [27], which is a direct IN with a 2D-torus topology. The advantage of direct INs is the good scalability of such a system: if the number of nodes in a network is increased, buffering and switching resources automatically increase by the same amount.

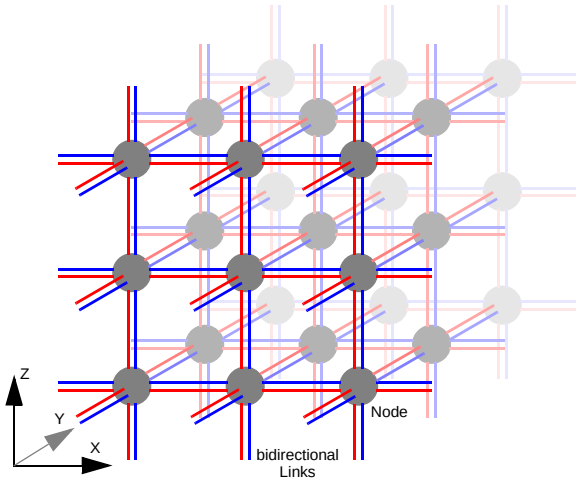


Figure 6-16. 3D-Torus topology

The Extoll network is optimized to achieve high bandwidths and low latencies for the transfer of small packet sizes. Wormhole routing is being used in the network. In wormhole routing, a packet traverses the network in a pipelined fashion. Flow control is performed on small blocks of data, called flow control digits (flit). Buffering in the network occurs only on flit level. Buffering on packet-level, as in store-and-forward and virtual-cut-through networks, is not done.

Packets find their way through the network using source path routing. In contrast to table-lookup-based routing schemes, the path through the network is determined by the source node. Thus, routing is fixed, which is in contrast to adaptive routing schemes. The advantage of source path routing is that routing decisions can be made extremely fast in every switch. In a direct IN, this is of particular importance, as the number of switches on the path of a message may become much larger than for topologies with centralized switches.

The network that is implemented in every Extoll chip is shown in Figure 6-17. At the heart of the network is the Extoll switch, which is based on a unidirectional 12x12 crossbar. 6 ports of the switch connect to the link ports. Towards the functional units of the NIC, four network ports translate between NIC and network protocols. A multicast port is used for

hardware-based support of multicast and broadcast packets. The high availability port (HAP) is being used mainly to resolve problems that arise out of link failures.

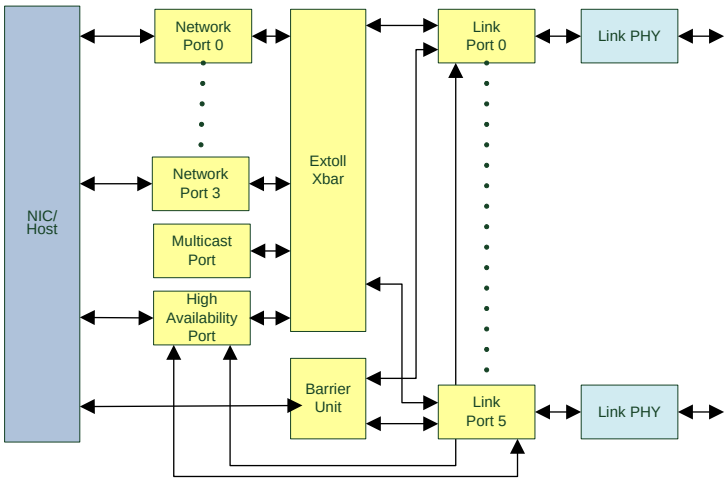


Figure 6-17. A node of the Extoll network

Every packet's payload starts with a routing string (see Figure 6-18). It consists of units that have the size of physical transfer digits (phits), which is 16 bit. In standard source path routing, routing pits contain only the destination port of the next switch, and are striped of after each switch. Extoll integrates a delta routing: every routing phit is valid for the number of hops specified in the counter fields. In every switch, counters are decremented, and the phit is only removed if both counters are zero. In a regular 3D torus topology and if link cables are connected in a consistent fashion (e.g. positive X direction on port 7, negative X direction on port 8, positive Y direction on port 9, and so on) three routing phits are sufficient to address 256,000 nodes. A fourth routing phit is needed to select the desired network port.

For deadlock avoidance in the network, two virtual channel groups can be used. Every of these groups consists of 4 virtual channels (VCs) which may be used to reduce the impact of head of line blocking.

Credit based flow-control is used between every two Extoll switches to avoid buffer overflows in the switches and to make buffer space guarantees for the different virtual channels. Extoll is an input buffered switch, with a buffer capacity of 32 flits. This buffer size has been selected so that two buffer slots are reserved for every VC, and a single packet stream on one virtual channel can saturate the link [130].

Parallel applications frequently use barriers to synchronize among the different threads or processes. A hardware-based barrier mechanism has been integrated into Extoll. The barrier unit supports up to 16 hardware barrier groups at the same time. Barrier information is multiplexed on the Extoll links.

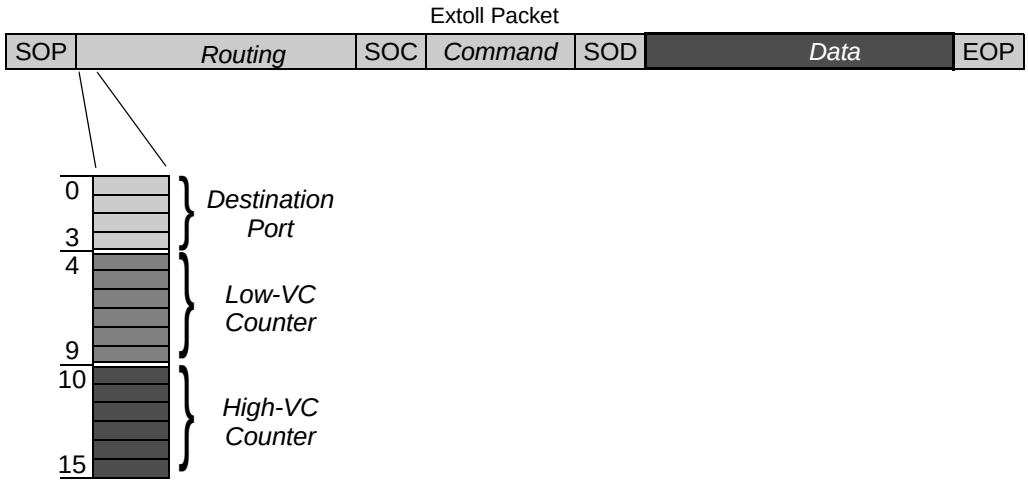


Figure 6-18. Extoll packet and routing format

6.2.1 Packet and Flit Protocol

The Extoll has been optimized for small message transfer. Thus, the overhead for small packets in the network must be as small as possible. A packet consists of the segments: routing, command and data. The packet injection and extraction points into and from the network are the network ports.

The Extoll packet size does not have an upper bound defined by the protocol. It is left to layers on top to specify a maximum transmission unit (MTU) if this should be desired. A packet is transmitted with a sequence of flow control digits (flits), as shown in Figure 6-19. A flit can hold up to 32 phits of payload. With a physical transfer digit (phit) size of 16 data bits plus 2 control bits, the payload of one flit is 64bytes, which corresponds to the cacheline size in most systems. Within the payload, transitions to the command and data sections are marked with start-of-command (SOC) and start-of-data (SOD) phits. Every flit is framed with start and end phits and a 16bit CRC. Although theoretically, a framing on the start of a flit would be sufficient, the end-of-flit character provides additional security at little cost.

The first and last flits of a packet are marked with start-of-packet (SOP) and end of packet (EOP) phits respectively. All other framing words in the flits are start-of-flit (SOF) and end-

of-flit (EOF). The start phits include information about the virtual channel in which a flit flows. Extoll supports 8 different virtual channels. The link protocol has been optimized to provide maximum Hamming distances for this number of channels. It can be slightly modified to support more virtual channels, at the cost of a reduced Hamming distance. As all flits of one packet flow in the same virtual channel, flits of different packets can be interleaved in the links.

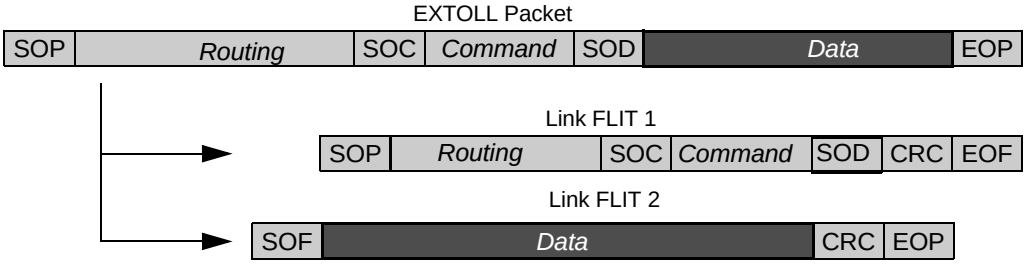


Figure 6-19. Extoll packet and phit framing

6.3 Extoll Link Error Correction

Extoll uses bidirectional links to connect nodes in the network (see Figure 6-20). This allows to transmit flow control and retransmission protocol information backwards. The link can be separated into a physical layer (PHY) and a logical link layer. Different physical layers may be used for Extoll, although it is optimized for an 8b/10b-coded high speed serial transmission.

The logical link layer, consisting of link in- and out-ports, multiplexes data streams from the crossbar and the barrier module to the link, as well as credit information that comes from the crossbar. The link layer is also responsible to correct errors on the link. In a classical layered protocol, all link errors should be handled by the link port, and be transparent to the switch layer of the network. Unfortunately, such a complete encapsulation can only be implemented in a store and forward fashion. As Extoll is designed as a low latency network, store and forward at every link port is not feasible. Thus, error correction by the link is not fully transparent to the layers above.

The remainder of this section describes the mechanisms to correct errors on the links.

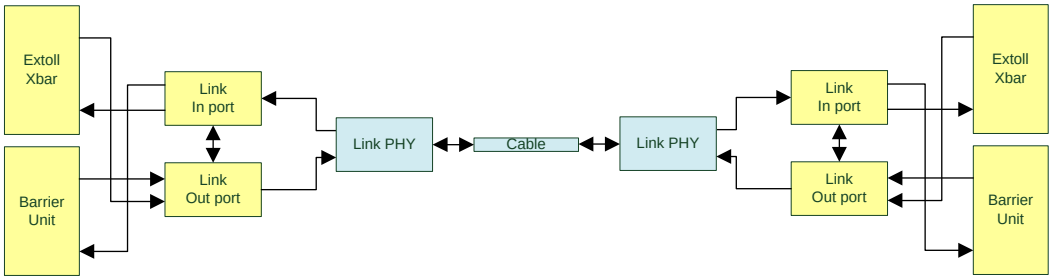


Figure 6-20. A link between two nodes in the Extoll network

6.3.1 The Physical Link

The target link media for Extoll is a high-speed serialized optical transmission. The development of a high-speed serial link PHY is complex and out of the scope of this work. However, experiences in the design of such links in OASE and with FPGAs [133] allow to presume a general architecture of such a link, as depicted in Figure 6-21.

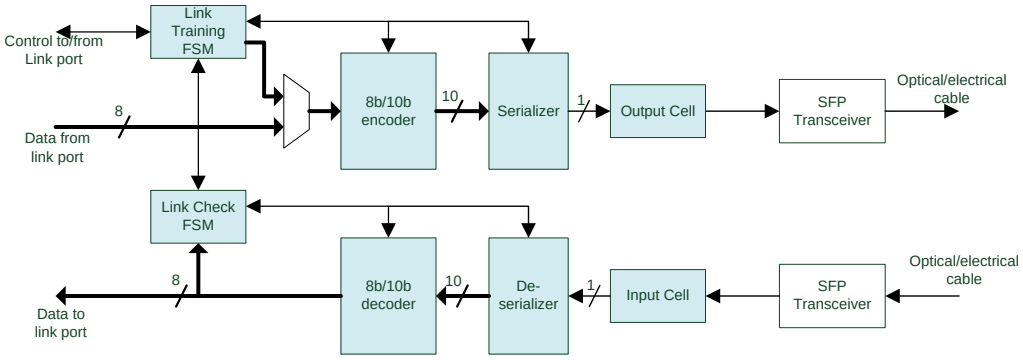


Figure 6-21. Functional block diagram of the PHY in the FPGA prototype

Such a link PHY performs a low-level initialization, which includes the use of training patterns to align the 8b/10b decoder. The alignment will not be verified any more during normal operation. The state of the link and errors in the 8b/10b decoder are signaled to the link port. Expected errors on a serial link that are visible to the link port are:

- Transient, random single bit errors.

- Very long bursts of bit faults, e.g. if the receiver looses bit alignment.
- Permanent link failure, e.g. through an unconnected cable.

Errors of the latter kind can be detected by the physical link itself. However, it may take a while until a receiver may take notice. Experiments showed that the high-speed serial transceivers of an FPGA detect this after a time period much longer than milliseconds. During this period, arbitrary data is being forwarded to the link port. Thus, the link port must be able to detect this situation early.

Parallel cabling is also supported in Extoll, although parallel cables are not considered to be the link that will be used in production Extolls. Due to the much lower complexity, the parallel link has also been used in the design and evaluation phase. The cabling technology from Atoll has been reused for Extoll. An Atoll link cable is a 68 pin twisted pair cable. Using source synchronous transmission, every direction of the link has eight data, one control and one clock line. An additional line per direction is used to detect whether a cable is connected on the other end. Using DDR signaling, data frequencies of up to 300 Mbps have been verified.

Expected faults on a parallel link that are visible to the link port are:

- Transient, random single bit errors.
- Static faults on one or multiple LVDS pairs that lead to constant or frequent bit errors.
- Permanent link failure, e.g. through an unconnected cable.

A link detection mechanisms based on the cable detect line and clock detection are used. However, these mechanisms alone have turned out to be not sufficient for reliable link detection. For example, static faults in LVDS pairs require testing at link initialization. Thus, an initialization sequence has been developed [132], which is similar to the unitization and training of a serial link. The following section will detail the coding for serial links. If parallel links would be used in production systems, the coding would have to be adapted to provide Hamming distances in the binary coded domain. Such a development is simple, and thus is not presented here.

6.3.2 Protocol Encoding for Serial Links

Besides start and end characters, an Extoll protocol must support credits and acknowledgements. These characters must encode additional information as e.g. the virtual channel number, which must be bit error protected. Additionally, idle, retransmission and management characters exist. As the number of K characters is by far not sufficient to encode these Extoll control characters, a combination of K characters and the set of D characters with HD=4 is used. This choice also fits well the Extoll phit size of two 8b/10b characters. To

avoid confusions between Extoll and 8b/10b characters, Extoll characters are referred to as phits.

Encoding for single bit FEC. Section 6.1.4 shows that for control phits, FEC is suited well. For packets, a retransmission based protocol is more efficient. As explained above, forward error correction on 10b characters requires a lookup table. Here, a design choice is whether one lookup occurs for a complete phit, or if the characters of the word are looked up independently of each other. The size of the table is critical for the feasibility of an implementation. The simplest implementation is a ROM, where the incoming phits or characters are used as an address to read the corrected and 8b decoded value. For a lookup of a 20 bit phit, the table would have a size of at least $2^{20} \cdot 18bit \approx 2MByte$, and thus is not realizable. If the two10b characters are looked up separately, a single table with two read ports and a size of $2^{10} \cdot 9bit \approx 1kByte$ is sufficient. However, the separate lookup requires that the first and the second character position of the phit exhibit an $HD \geq 3$ individually. This restricts the set of usable K characters to the set of {K.28.7, K.27.7, K.28.3}, which limits the coding space for Extoll control phits. Figure 6-22 shows the Extoll control phits.

Name	Upper Byte	Lower Byte	Name	Upper Byte	Lower Byte
SOP_VC0	K_28_3	D_14_1	**NACK0**	K_27_7	D_3_1
SOP_VC1	K_28_3	D_20_1	**NACK1**	K_27_7	D_11_2
SOP_VC2	K_28_3	D_20_6	**NACK2**	K_27_7	D_17_2
SOP_VC3	K_28_3	D_22_3	**NACK3**	K_27_7	D_25_3
SOP_VC4	K_28_3	D_28_2	**NACK4**	K_27_7	D_17_5
SOP_VC5	K_28_3	D_28_5	**NACK5**	K_27_7	D_3_6
SOP_VC6	K_28_3	D_6_2	**NACK6**	K_27_7	D_5_3
SOP_VC7	K_28_3	D_14_6	**NACK7**	K_27_7	D_10_3
SOF_VC0	K_28_3	D_3_1	**CREDIT0**	K_28_7	D_14_1
SOF_VC1	K_28_3	D_11_2	**CREDIT1**	K_28_7	D_20_1
SOF_VC2	K_28_3	D_17_2	**CREDIT2**	K_28_7	D_20_6
SOF_VC3	K_28_3	D_25_3	**CREDIT3**	K_28_7	D_22_3
SOF_VC4	K_28_3	D_17_5	**CREDIT4**	K_28_7	D_28_2
SOF_VC5	K_28_3	D_3_6	**CREDIT5**	K_28_7	D_28_5
SOF_VC6	K_28_3	D_5_3	**CREDIT6**	K_28_7	D_6_2
SOF_VC7	K_28_3	D_10_3	**CREDIT7**	K_28_7	D_14_6
SOS	K_28_7	D_10_3	**EOP_ERR**	K_28_7	D_3_1
ACK0	K_27_7	D_14_1	**EOF_ERR**	K_28_7	D_11_2
ACK1	K_27_7	D_20_1	**EOP**	K_28_7	D_17_2
ACK2	K_27_7	D_20_6	**EOF**	K_28_7	D_25_3
ACK3	K_27_7	D_22_3	**RETRANS**	K_28_7	D_17_5
ACK4	K_27_7	D_28_2	**IDLE**	K_28_7	D_3_6
ACK5	K_27_7	D_28_5	**MNGT**	K_28_7	D_5_3
ACK6	K_27_7	D_6_2	**SOD**	K_29_7	K_28_3
ACK7	K_27_7	D_14_6	**SOC**	K_30_7	K_28_3

Figure 6-22. Extoll control phits

Multi-Bit errors. Due to the Hamming distance of 3, only single bit errors can be corrected. Multi-bit errors within the same character may not be detected. The set of D characters with HD=4 has SECDED capabilities.

While Section 6.1.2 shows that multi-bit errors within a 10b character are very unlikely, and thus are not considered to occur by most system designers. Assumed they would occur, the vast majority of multi-bit errors will be detected, either by the 8b/10b decoder or through protocol checks. However, there are cases where a multi-bit error within a single character leads to an undetectable error (see Figure 6-24). Multi-bit error detection on the Extoll control phits could be improved by doing only error checking instead of correction. The link protocol does not have to be changed for this.

All in all, the Extoll link protocol encoding that has been proposed in this section provides a reliable transmission of control information on links in the presence of single bit faults. This avoids the loss of data and inconsistent states of the links, which may occur in other state of the art implementations of 8b/10b encoded links. Single bit FEC for control information is a significant improvement over state of the art 8b/10b protocols.

6.3.3 The Logical Link Layer: the Link Port

A link port consists of two parts: the link in-port as the receiver, and the link out-port as the sender (see Figure 6-23). A central point in the link out port is the arbiter, which multiplexes the link between packets from switch and barrier module, and control phits in a round robin fashion. The buffer space which is used for the retransmission buffers is also used to buffer incoming data streams. Therefore, barrier and switch have separate retransmission buffers. The flit CRC is also recomputed and inserted into the flit here.

Link level error checking is performed by the link in-ports. Operation of the link can be differentiated into the initialization phase and normal operation mode. Link port initialization is started as soon as the physical link layer has successfully initialized. During initialization, bit errors on the link can either be ignored or lead to a restart of the initialization. During normal operation, a receiving link may be in any of the three super states: intra-flit, inter-flit, and waiting-for-retransmission. The last possible state is link failed. These states are from a theoretical point of view, an implementation will feature a much more detailed set of states.

The **inter-flit state** is the typical state for an idle link. In this case, only IDLE phits cross the link. It is also the default state after initialization. Other allowed phits in this state besides IDLE are: SOP_VCx, SOF_VCx, SOS, ACKx, NACKx, RETRANS and CREDITx. If any other phit is received in this state or the 8b/10b decoder signals an error that cannot be corrected by the FEC mechanism, it can only be caused by an error on the link.

In this case, the link port requests a retransmission from the sender. Although this is only required in the case that a packet got corrupted, error detection mechanisms are more light-weight if a retransmission is requested in any case. The reception of an SOP, SOF or SOS phit leads to a transition into the intra-flit state, an detected error to waiting-for-retransmission.

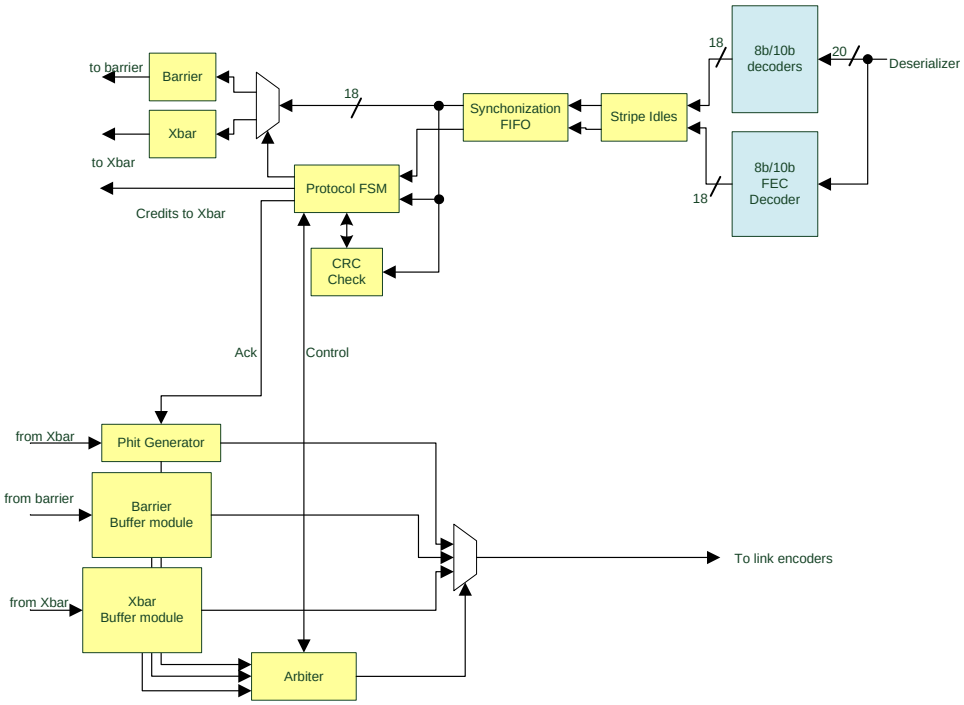


Figure 6-23. The link port

During the **intra-flit state**, all data phits and SOC, SOD, EOP, EOF, EOP_ERR and EOF_ERR control phits are allowed. If any other flit is received or the 8b/10b decoder signals an uncorrectable error, the flit in transition is terminated directly after the erroneous phit with an end-error character. If the error occurs in the very first data phit of a flit, the minimum size of 3 is violated. As the Extoll switch removes flits that are too small, it is not necessary to fill the flit with dummy data to increase the flit size. If a CRC error is detected, the end phit is replaced with an end-error phit in the same place. Erroneous flits that started with an SOP or SOS will be concluded with EOP_ERR. This closes the virtual channel after the flit, which is important as the routing string in the flit may be affected, and thus the retransmitted flit must open the virtual channel. Erroneous flits that started with an SOF will

be concluded with an EOF_ERR, which keeps the virtual channel open for the retransmission of the flit and for following flits. An error also leads to the transition into the waiting-for-retransmission state. Otherwise, the reception of an EOP, EOF, EOP_ERR or EOF_error leads to a transition to the inter-flit state. To avoid oversized packets due to a corrupted end of a flit, such flits will be truncated at the maximum allowed size and ended with an error character.

If an uncorrectable **error** occurred, the current and all subsequent flits are ignored. It is a necessity to ignore flits to maintain strict ordering among them. Only the other control characters will be interpreted. If a RETRANS character is sampled, a transition into the inter-flit-state, and thus normal operation, occurs. Protocol checking as described above has to continue.

If multi-bit errors in single 10b characters are assumed to occur or forward error correction is switched off, a major issue is that an erroneous control phit may be one of those that start or end a flit. Thus, the state after the error can only be determined by looking at the phit that follows the erroneous phit.

If any further error occurs between the first error and the starting retransmission, there is a good likelihood that the physical link has a permanent problem that causes a burst error. This may happen for example if the serial receiver has not detected yet that a cable has been removed. Thus, a receiving link transitions into the link failed state, and waits for resolution by the management software as described in Section 6.3.4.

The retransmission protocol. The retransmission protocol ensures that packets are not corrupted. If the receiving side of a link detects an error that might possibly affect a phit like described above, a retransmission request is sent to the sending side. All flits that are received between the detection of an error and the start of the retransmission are ignored, as this is the only way to guarantee that ordering among phits is being maintained. Theoretically, it would be sufficient to maintain ordering only among flits of the same virtual channel. As the information about the virtual channel may be subject to link errors, all flits must be retransmitted. The cost for the retransmission in terms of bandwidth and latency can be neglected in any case, as retransmissions occur infrequently.

The basic idea is as follows: All flits that are transmitted over the link are also copied to a retransmission buffer in the sending link. If the receiving part of the link positively acknowledges the reception of a flit, the buffer space can be freed. If the acknowledgement is a negative acknowledgement, the sending part will instead initiate a retransmission of all flits in the retransmission buffer. The retransmission begins with a RETRANS phit. The retransmission of the flits follows the same rules as the normal transmission of flits. In particular, retransmitted flits are also protected by the retransmission protocol.

In order to allow the sending link to detect lost acknowledgements, an indirect relation of acknowledgements to flits is introduced. Both sides of a link implement counters. On the sending side, the counter is incremented for every flit that is sent. The receiver increments it for every received flit. The acknowledgements ACK0...ACK7 and NACK0...NACK7 correspond to the counter values for the acknowledged flit in the receiving side of a link. The size of the counters is a trade-off, as the available code space for control phits in the 10b domain is limited.

The following link errors can be corrected using the retransmission protocol:

- Bit errors in data flits and framing control phits.
- Loss of up to 7 ACKs in a row.
- Loss of an infinite number of NACK or RETRANS. Both are protected using time-out-based resending of NACKS. If the retransmission does not start after the time-out period, the NACK is being resent.
- The complete loss of flits can only happen in the case of a link failure. Thus, this case does not need to be covered by the retransmission protocol, but will be covered by the link failure resolution mechanism that is described in the next section.

Credit phits can be lost if a bit error occurs in a credit phit, as credits cannot be recovered by the retransmission protocol. Although the loss of a single credit is not critical, it may degrade performance. Thus, software must be notified in the event of unexpected phits on the link via the Extoll register file, so that it can check the credits in the crossbars.

Using the protocol described here, a reliable and order-maintaining retransmission of phits can be ensured in the case of bit errors in flit payloads.

Using FEC coded control phits, the protocol is highly reliable for the expected source of error: rare, noise-induced single bit errors. For multi-bit errors in control phits, the vast majority should be detected by out-of-table, disparity, or out-of-control-phit space checks. The protocol can detect and resolve some of such undetected errors (see Figure 6-24). This provides some additional security.

from character	to character	Always Detected by Protocol Check?	Minimum errornous bits in single character for non detectable error (correction/detection)	impact
SOP_VCx, SOF_VCx	SOP_VCy, SOF_VCy	No	3bit, 4bit errors	data corruption
	SOS	No	2bit, 3bit error	data corruption
	ACKn, NACKn, CREDITn, EOX, IDLE	Yes		
ACKy, NACKx	ACKy, NACKy	No	3bit, 4bit errors	data corruption in case of retransmission
	CREDITn	No	2bit, 3bit error	Xbar buffer overflow
	EOx, IDLE, Sox	Yes		
SOS	SOP_VCx, SOF_VCx	No	2bit, 3bit error	data corruption
	ACKn, NACKn, CREDITn, EOX, IDLE	Yes		
EOx	any	Yes		
CREDITx	ACKm, NACKn	No	3bit, 4bit errors	data corruption in case of retransmission
	IDLE	No	2bit, 3bit error	Credit loss
	SOP_VCy, SOF_Vcy, Eox	Yes		
	CREDITy	No	3bit, 4bit errors	Xbar buffer overflow or data corruption
IDLE	ACKn, NACKn	No	2bit, 3bit error	data corruption in case of retransmission
	CREDITn	No	3bit, 4bit errors	Xbar buffer overflow or data corruption
	SOP_VCy, SOF_Vcy, Eox	Yes		

Special Characters	
SOC, SOD	belong to CRC protected flit payload
MNGT, RETRANS	reinitialization of link
D.x.y	

Figure 6-24. Protocol detection of multi-bit errors in phits

6.3.4 Temporary or Permanent Link Failure

A link failure can have several reasons: cables that are accidentally removed, dysfunctional cables or transmitters, or temporary problems like a lost bit alignment. In the classical layer model, a temporary failure of a link may be considered to be resolved by the link layer. A permanent link failure must be resolved by the network layer to re-establish a valid routing in the system. As both faults can be resolved with similar resolutions, they will be described here together.

In Extoll, a permanent link failure requires the modification of the static routing through the network by the management software. As this process usually takes a while, the network should continue to operate until new routing tables take effect, even though the performance may be reduced. As Extoll shall be a lossless network, a requirement is that packets and flits that are traversing a failing link do not get lost. Therefore, three key issues can be identified when resolving a link failure:

- Packets that traversed the link while the fault occurred must be reassembled.
- All other packets that request the link later on must be rerouted.
- Routing tables must be modified to reflect the changed topology of the network.

Immediate Reaction. The first two points are the immediate reaction on faults in order to keep the network working. Figure 6-25 (b) shows the algorithm implemented in Extoll. This is the one where most actions can directly be taken in hardware, and thus lowest latencies can be expected. For reference (a) shows an implementation where the route of the detour is handled by the management software.

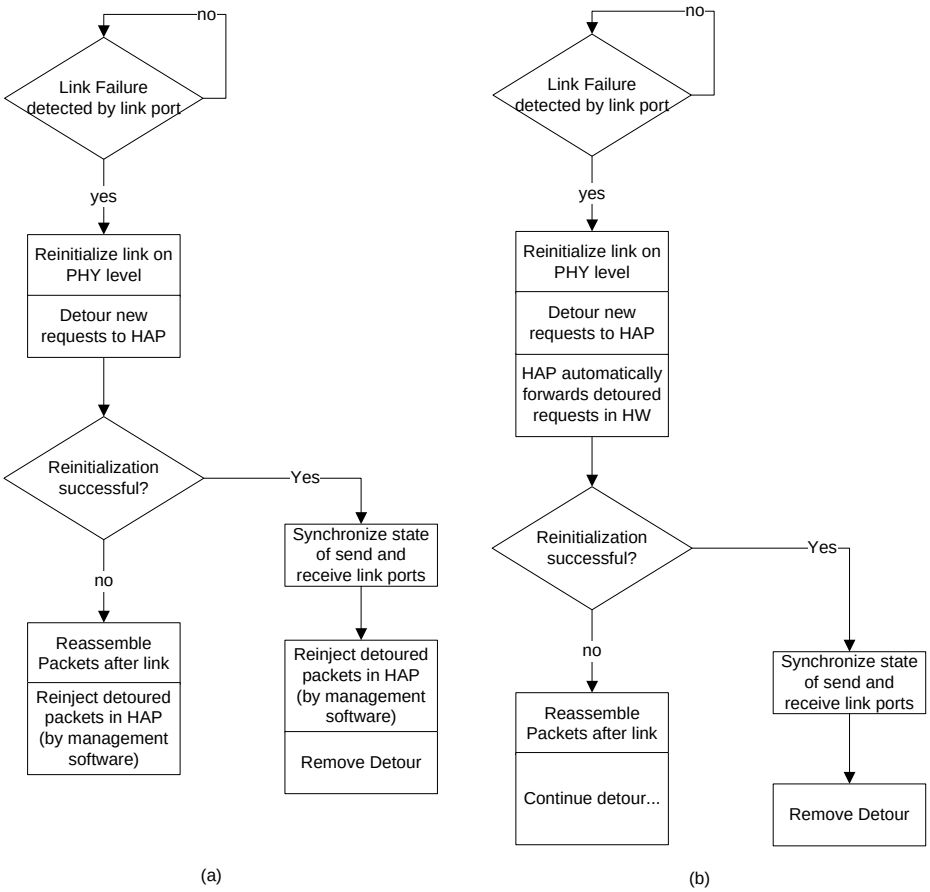


Figure 6-25. Link failure treatment by the Extoll network (b) and software based alternative (a)

If a failure is detected, a re-initialization of the link can be tried, either by software or automatically in hardware. Something which has to happen in hardware is to detour routing requests from packets that are destined to the broken link to the HAP. This avoids that such packets cause a head of line blocking, which soon can congest the whole network. This sit-

uation is depicted in Figure 6-26, where packet p1 is stuck on a broken link and packet p2 is detoured to the HAP.

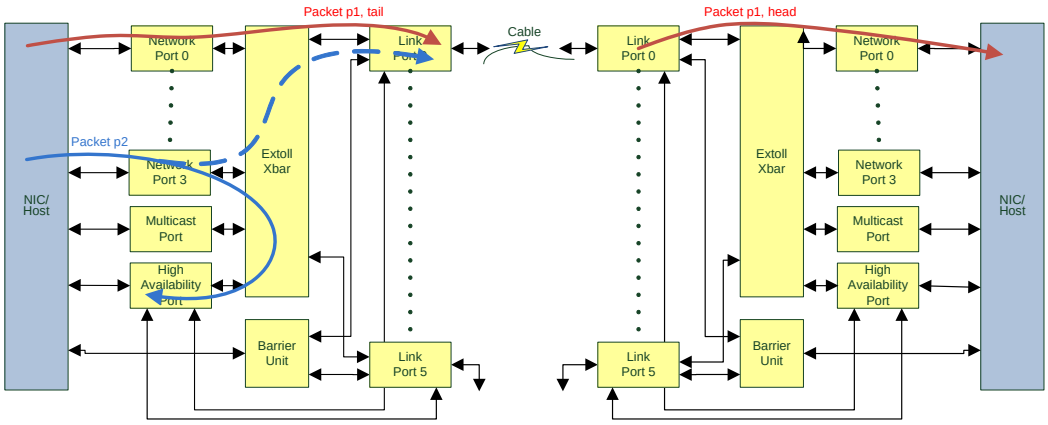


Figure 6-26. Scenario after a detected link failure

Further actions depend on whether the link has been successfully re-initialized. In the case of a permanent failure the following steps have to be performed, which are all software controlled over register interfaces to the network units:

- All packets that have been in transfer on the link have been split into a head and a tail, which must be recombined. The tails are being extracted over a dedicated path to the HAP. A loopback over the link port receive side and the crossbar is not possible, as the crossbar may be blocked by packet heads that flow in the other direction: all virtual channels and/or buffer space may be in use. Also, the number of the virtual channel must be maintained, as this number is required in order to stick the right packet flows together. The Crossbar would change the virtual channel of packets. Thus, a separate data path is being used between HAP and link port. The tail will then be injected into the receiver's link port, so that the packets continue to flow through the network (see Figure 6-27).
  The correct joint between head and tail must be determined before the injection of the tail. As the breaking link may have lost some acknowledges, the retransmission buffer may hold some flits that already have been transferred correctly. These must be removed from the tail. A network using sequence numbers in the flits could use those to determine the joint. In Extoll, sending and receiving part of the link count a flit sequence number when sending or receiving error-free flits. As flits are only lost on a link failure, a comparison of the counters is sufficient to determine the joint.

Another consideration is how packet tails travel to the other link port. In an HPC cluster environment, there is typically a second network, which can be used in this. It may also be possible to use the Extoll network. However, it must be ensured that packet tails do not get stuck at a link which is congested by the header of the same packet, as this would result in a deadlock.

- Packets that have been detoured to the HAP will be reinjected into the network. As ordering among packets is not required, reinjection can take place in parallel to the recombination of messages on the link. In the currently implemented solution, the HAP automatically looks up an alternative route to the neighboring node in a small lookup table, and prepends this route to the original routing string of the packet. A management software based solution would determine the target of the packet by analyzing the routing string. The old routing string is replaced with a new routing string that avoids the faulty link.

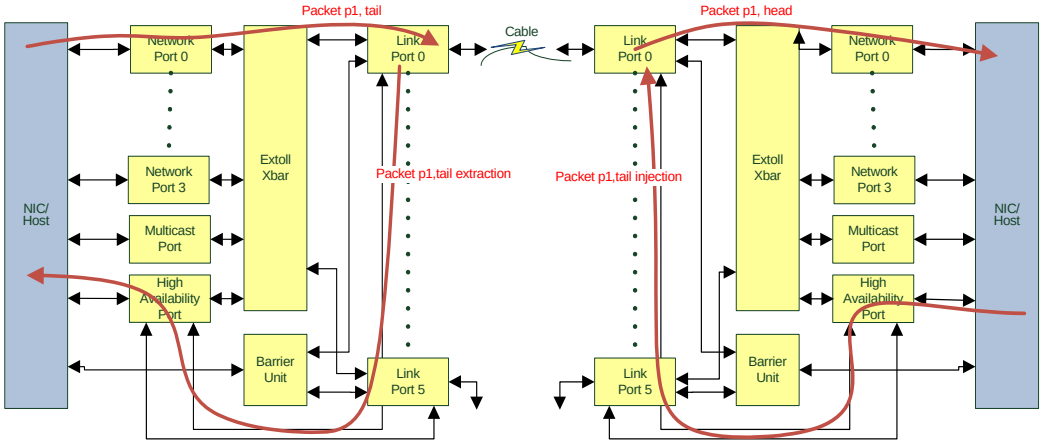


Figure 6-27. Packet tail extraction and injection due to permanent link failure

In the case of a temporary failure, the actions are different:

- As soon as the physical link is initialized again, transmission on the link can continue as normal. Credits may have been lost, so that credits have to be recounted.
- The detour can be removed.

Long-term reaction. For a longtime reaction on a permanently failed link, routing tables must be changed. The fast and efficient computation and distribution of routing tables may be a difficult task, in particular as the network will have an irregular topology after a link

failure. A routing method for such network topologies using routing over intermediate nodes has been developed in Section 6.1.7.

6.3.5 The Extoll Switch

An in-depth description of the crossbar can be found in [130] and [131]. Here, only fault correction mechanisms are described.

Actions due to link layer fault correction. In a classical layered protocol, all link errors should be handled by the link port, and be transparent to the switch layer of the network. Unfortunately, such a complete encapsulation can usually be only implemented in a store and forward fashion. As Extoll is designed as a low latency network, store and forward at every link port is not feasible. As a result, the Extoll switch has to cope with the following issues:

- Flits coming from the link ports may have a length of 3 instead of the minimum length of 4 phits. The crossbar must extract and discard these phits. This solution reuses crossbar logic that discards empty routing phits due to the stripping of routing characters.
- Erroneous flits, marked with EOP_ERR or EOF_ERR, flow through the network. Flits starting with SOF and ending with EOF_ERR simply follow the flow of the virtual channel. Flits starting with SOP and ending with EOP_ERR may have a corrupt routing. They will be routed through the network until they hit a network port, or until all routing characters have been stripped off, and thus can be discarded. A random bit error in the routing sting may violate the rules of the deadlock free routing algorithm, and thus such a packet may cause a deadlock. In the case of a deadlock, the flit will be in the flit buffer of a switch in-port in one piece and wait for the grant of a virtual channel. Extoll changes the destination port of such flits to the HAP, which never blocks. In the HAP, the flit can be deleted.
- Credit re-count. In particular after a link failure, credits may have been lost. Software must be able to stop individual link ports of a crossbar, read credit counts, and set credit counters.

Switch level fault correction. On the switch level, the only fault that can happen in the absence of bit errors on chip is a packet that requests an output port that is not available. A port may not be available either because it does not exist in the physical implementation, or because of an unconnected link cable or another failure on the link. Such packets will be routed to the HAP and treated there. The switch detects both cases by checking availability information from the link.

6.3.6 The High Availability Port

The main task of the high availability port (HAP) is to resolve problems caused by temporary or permanent failure of the links. As already explained in the previous sections, the HAP is being used to detour packets that request a link that is unavailable. Also, the HAP is being used to extract and reinject packet tails respectively from and into the links. Both events occur only until routing tables have been changed.

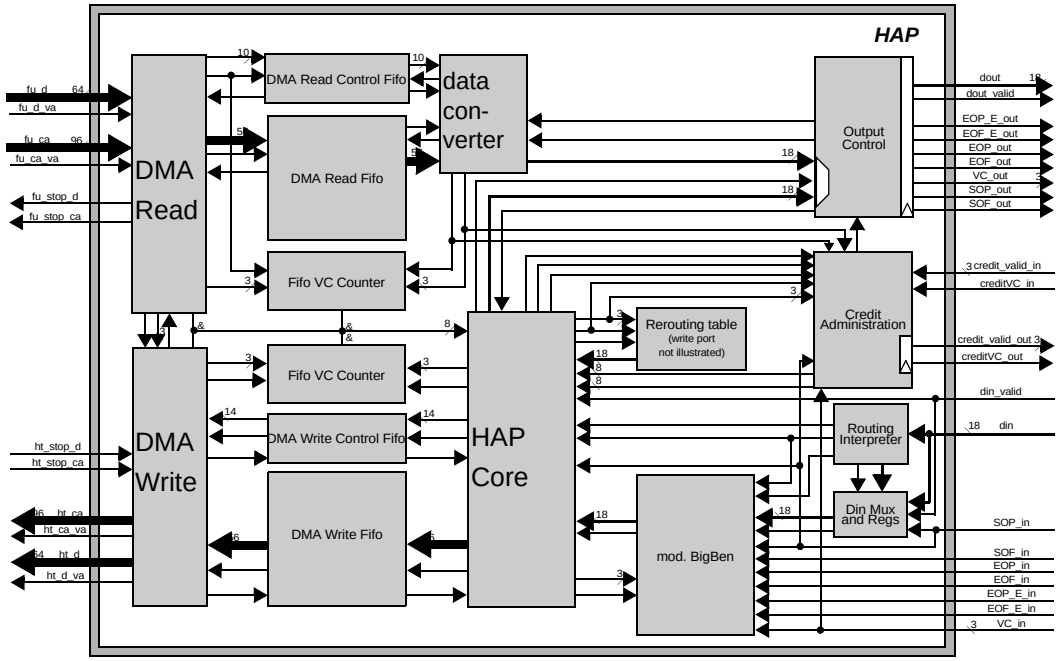


Figure 6-28. High Availability Port [131]

A third task is to improve routing in a regular network which has turned to a network with an irregular topology due to link or node failures, as described in Section 6.1.7.

The HAP has three interfaces: to the Extoll switch, to the host via the HT interface, and to the link ports (see Figure 6-28, where the direct interface to the link ports is not yet shown).

The HAP is virtual channel aware. It uses the same credit based input buffer as the crossbar. Packets or flits from the network will stream into this buffer first. The further proceeding depends on the type of the flit, which the HAP may determine by looking at sideband signals from the crossbar, and by interpreting the first routing phit:

- If the HAP has been directly addresses, the first routing phit contains the HAP address, as HAP routing phits are not consumed by the crossbar. If a HAP is directly addressed, it must act as an intermediate node. Thus, such a packet will be forwarded to the out port of the HAP. If the in-port buffer is full, it will be forwarded to the DMA buffers. Succeeding flits of the same packet follow in this path. However, once a flit of a packet has been written to host memory, all succeeding flits have to follow to maintain ordering among them.
- Packets that have been rerouted can be determined by sideband signals. A routing lookup in the rerouting table occurs. Then the path is the same as for the previous type of packets.
- Flits from the direct interface from the link ports are written into a dedicated host memory region via DMA.
- All other flits have been routed to the HAP as they end with an error phit. They will we written into a separate DMA region as well.

Data that is written to the node's main memory is stored in a raw format. To simplify address handling, every burst of three consecutive 18 bit phits is stored in a 64 bit aligned memory word.

The output control module multiplexes the interfaces to the crossbar between the different packets that come from the input buffer or main memory. It uses round-robin arbitration among those packets for which credits are available.

The DMA read buffer hides RAM read latency by reading head data as soon as DMA queues in RAM contain valid entries.

6.3.7 Barrier

Extoll implements barrier logic in hardware [132]. All nodes that take part in a barrier belong to the same barrier group. Extoll provides support for up to 16 barrier groups at a time in the network.

Software that enters a barrier signals this via the register interface of the barrier module, and also requests the status of a barrier there. Extoll barriers are tree-based, where the nodes of the barrier tree are mapped onto the Extoll nodes.

To achieve lowest latencies, tree-node logic is independent of the Extoll crossbar switch. However, barrier data is multiplexed on the Extoll links to avoid the cost of an extra barrier network. Thus, the barrier logic can be seen as a second switch layer.

As depicted in Figure 6-29, the barrier module hosts 16 barrier group units, i.e. one separate unit for ever barrier group. The scalability of the number of barrier groups is only limited by the hardware resources.

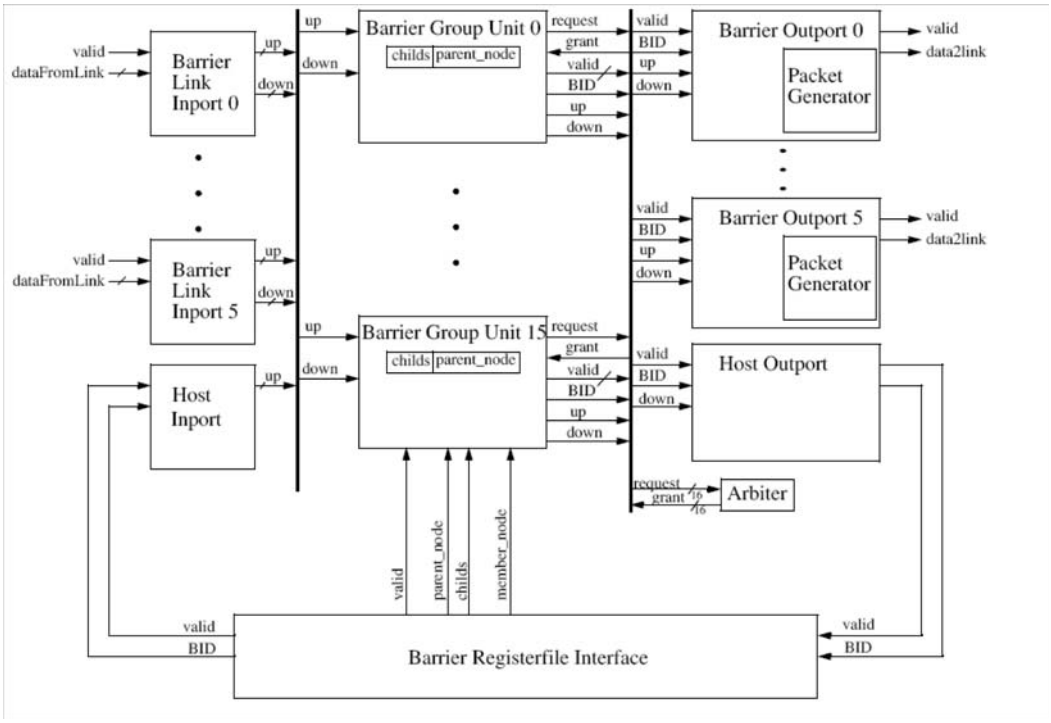


Figure 6-29. The barrier module [132]

The barrier input and output ports decode and encode the barrier messages into/from Extoll packets. To distinguish them from normal, crossbar-routed packets, they start with the Start-of-Special (SOS) phit. From the network protocol point of view, a barrier packet behaves just the same way as crossbar packets. Theoretically, a barrier packet may span over multiple flits. However, this is not required, barrier packets are minimum size packets of 4 phits, as shown in Figure 6-30.

As barrier messages are Extoll packets, the same fault tolerance mechanisms that protect other Extoll packets protect barrier packets. However, one difference exists: barrier packets are not routable, and thus are not reroutable by Extoll hardware. If a link is marked as failed, management software must be notified, which must then extract these flits and inject them to the other end of the faulty link until it has changed barrier tree mapping.

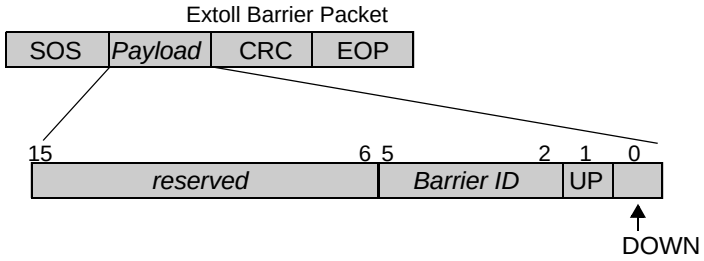


Figure 6-30. Extoll barrier packet format

6.3.8 The Network Port

The interface between NIC functional units and the network is the network port. The generator serializes the 64 bit data words from the functional units to the 18 bit wide phit format, breaks up data into flits and does the framing. At the interface to the network, it must be virtual channel and credit aware.

The network port analyzer receives packets from the network. As the endpoint of a communication as seen by the network layer, the analyzer hides errors in the network from the functional units.

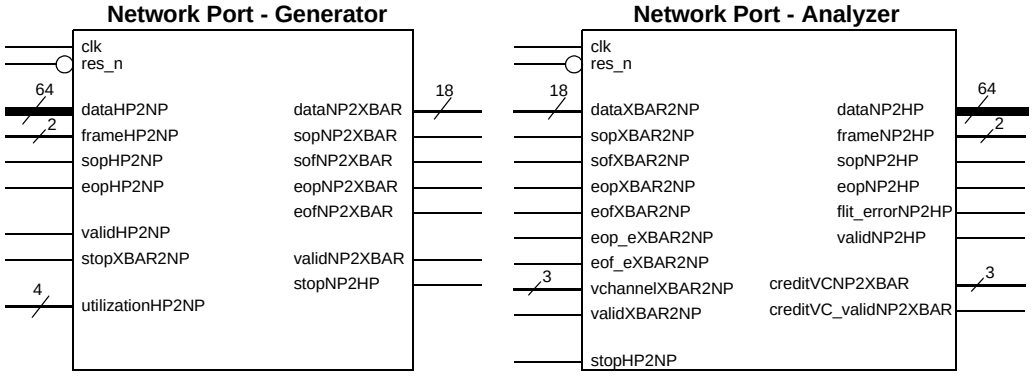


Figure 6-31. Network Port Generator and Analyzer [126]

Impact of faults on the analyzer. In order to minimize latency and avoid a store-and-forward implementation, the generator may brake up packets into flits arbitrarily. Packets will

be delivered to the receiving network port in the same way as they have injected by the sender network port, with the following restrictions:

- Flits must have a minimum size of 4 phits, i.e. there must be at least one payload phit in each flit. The maximum payload size, including SOC and SOD words, is 32 phits.
- Switches in the network may consume routing phits, these phits are removed from a flit. Thus, the routing section of a flit shrinks with phit granularity. A flit that contains routing only may be removed completely during its path through the network.
- To verify the command of a packet, the end character must be checked not to be an EOP_ERR or EOF_ERR. To reduce the latency of this check, sender network ports must end a flit as soon as the command frame ends.
- As a result of a link error, erroneous flits may continue to travel through the network and arrive at network ports. Erroneous flits end with one of the two error fits: EOP_ERR and EOF_ERR. A network port can simply ignore and discard both types of erroneous flits, as a retransmission will follow up. A flit that is market as erroneous may contain all types of errors. Particular errors are data bit errors, and wrong payload sizes. Both EOP_ERR and EOF_ERR end phits may be present for those errors. A third type of erroneous packets are misrouted packets. These start with SOP and end with EOP_ERR.

The straight forward implementation of the network port is a store and forward architecture, in which data is forwarded to the functional units only after it has been verified. This introduces an additional latency that depends on the flit size. For maximum sized flits, this latency sums up to ~30 clock cycles to the overall packet transmission latency.

An alternative implementation might speculatively forward data to functional units, and verify or disapprove it later on. However, functional units must not change the state of the system, unless it is recoverable, before the command has been verified. Otherwise, all types of undefined and illegal behavior might occur.

6.4 On Chip Protection

The previous sections described how the Extoll network copes with link faults. Paths on chips are not protected against bit faults with this protocol. To establish a protection against transient bit errors, which is the relevant type of fault, several potential solutions exists. This section shall give an outlook about the design space.

The most important question is whether error detection or correction is performed. As the Extoll network shall be lossless even without an end to end retransmission protocol, error

correction is recommended. Figure 6-32 shows the design space, which also depends on the granularity of protection.

Using a flit-based granularity, the space for the CRC phit which is being used for off-chip error detection could be used for CRC on the chip as well. As the link out-port must recompute the CRC to be able to differentiate between link errors and on-chip errors, this is also the place where an on-chip CRC error is being checked. As the packet based CRC is already being used, this solution has very low overhead and thus is the best option if error correction is not required.

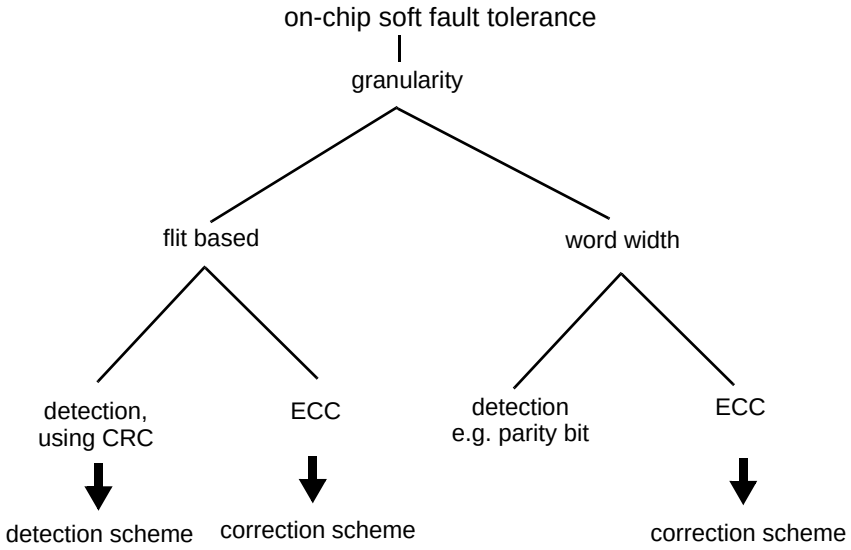


Figure 6-32. On-chip data path protection

For forward error correction, a general requirement is that a check is to be performed every time data interpreted by control logic. In the Extoll network, phits that are interpreted are control characters and routing phits. One solution is to protect the 16 bit wide data path together with the predecoded control character signals with a Hamming code, which adds 5 control bits.

A different approach could use the CRC phit for a Hamming code over the complete flit. Error correction can be done at the link out-ports: if an on-chip error is detected, the port concludes the flit with an EOP_ERR or EOF_ERR, corrects the flit in the retransmission buffer and retransmits it. Control phits can be protected by changing the coding to a Hamming code instead of using the link protocol encoding. The only problem are routing phits,

which are interpreted and modified by the Extoll Crossbar. Future research may find a solution for this issue.

To achieve further improvements in reliability, protection of control structures like state machines may be considered.

6.5 Summary

The Extoll protocol and control character encoding efficiently protects against all types of link faults. Single bit FEC for control information is a significant improvement over state of the art 8b/10b protocols. This avoids the loss of data and inconsistent states of the links, which may occur in other state of the art implementations of 8b/10b encoded links. A correction of on-chip transient bit errors can be added.

The hardware-based routing over intermediate nodes provides a mechanism to improve routing in regular networks with faulty links.

The direct overhead per flit is three phits. When maximum sized phits are used, this overhead reduces the bandwidth that is available to 91.5%. In the backwards direction, acknowledgements and credits are an additional overhead, reducing the effective payload bandwidth to 85% of the raw link bandwidth. Besides the overhead that is introduced by the protocol, line coding adds additional overhead. As 8b/10b coding itself has a 20% overhead, the total effective payload bandwidth is 67%.

The closest competitor to the Extoll protocol is Infiniband, which is a lossy network. Compared to framing in Infiniband, the flit framing of Extoll is only 2 bytes larger, as Inifiniband uses single K characters for that.

The only significant difference in bandwidth between Extoll and Infiniband occurs for larger packets: Infiniband sends packets in one piece, whereas Extoll partitions packets into flits, which have to be framed individually. This is not caused by the fault tolerant protocol, but by the general architecture of the network. Infiniband uses virtual cut-through routing, while Extoll uses wormhole routing, which requires much less buffer space. If Extoll's flit size, and thus the buffers in Extoll, would be doubled to 128 bytes, efficiency would rise from 85% to 92%.

7 Conclusion

The steady performance growth rate of computer systems can only be maintained in the future by exploiting parallelism at the level of threads and processes. Parallelism requires efficient and fast communication methods among the components of a system. One major goal of this thesis was to research and find solutions to remove the performance bottleneck of network interface controller to processor communication.

Besides the link bandwidth, the latency that is observed on this path plays a fundamental role for overall system performance. This latency directly increases the overall communication latency between two nodes. Additionally, the access latency that is observed by the processor affects the throughput of the processor adversely.

To achieve a significant decrease in latency at all, on-chip communication is mandatory. Communication over chip boundaries is just as expensive in terms of latency as a DRAM memory access. The integration of network interface, processor cores and memory controller onto the same die will become even more important in the future, as the processor to I/O performance gap is still widening.

A closer coupling has not only to occur on the physical level, but on the protocol level as well. For processors, caching of data is one of the most crucial factors for processor performance. For device to processor communication, caching of data is currently not employed. A tighter integration of devices into the system must consider these protocols.

As most of today's computing nodes are now shared memory multiprocessors, the essential communication mechanism in today's computing nodes is the cache coherence protocol. A comprehensive and up-to-date description of shared-memory design space is given in this work. Although explicit broadcast protocols in direct interconnect networks have changed the conditions under which coherence protocols have to operate, cache coherent communication has not been a main focus of both researchers and authors in the last years. Thus, the in-depth description of the state of the art for small scale shared memory systems is a pre-

requisite for any further work in this area, and cannot be found in its up-to-dateness elsewhere.

A first implementation of a closer coupling of device and processor is the HyperTransport direct connect architecture, based on the HyperTransport IP core and the HTX board. The HyperTransport IP core already has evolved from a research project to a product, and is freely available under an open-source license. The coherent version of the core is distributed to licensees of the coherent protocol by AMD. Both cores find large interest in both industry and academia. Thus, they are a significant practical contribution to the community, leveraging research and design of coherent devices and coprocessors of different kinds.

Another contribution of this work is the evaluation of cache coherent devices, focusing on the critical path of device to processor communication. Performance estimations in this work are based on RTL implementations using the HyperTransport and coherent HyperTransport interconnects. Compared to abstract, high level simulations, this guarantees a solid quality of the results. Also, these prototypes verify the proper functioning of the proposed concepts.

To the best of my knowledge, only Mukherjee [9] has proposed a similar architecture: external coherent devices in bus-based systems. Mainly due to the widening processor-to-memory and processor-to-I/O gaps, the situation has changed since then: performance increases through coherent transfers cannot be expected for external devices. However, for highest performance, devices and coprocessors must be integrated into the chip in a SOC-like fashion. In this case, coherent devices exhibit a significantly improved performance over classical DMA.

A completely new idea in this work, and thus a key contribution, is the transfer cache. It is the only architecture of those that have been analyzed in this work that improves the processor read latency even for external devices. Also, devices do not need to participate at the coherence protocol. In practice, getting access to the proprietary, non-standardized coherent protocols may be difficult due to legal and political reasons. Thus, the transfer cache is a very promising concept, which fits particularly well with for example the Sun UltraSparc T2 memory architecture, as the existing second level caches can be used as transfer caches.

Another option for future improvements are direct processor cache access mechanisms. The placement of data directly into the processor caches leads to best processor read latencies. Compared to NICs in the processor core or specialized network message caches, DCA is a universal mechanism that can be used by all devices in the system. Such mechanisms have virtually not been researched yet. The outlook presented in this thesis is an excellent starting point for intense work on DCA mechanisms.

While the first part of this work deals with the optimization of device to processor communication, the second part concentrates on the opposite side of the NIC: the network itself.

Again, research in this field is driven by the growing exploitation of parallelism. Higher parallelism leads to increased network sizes with larger numbers of nodes. For example, the currently fastest supercomputer [99] has 106,496 computing nodes. At the same time, parallelism and complexity of the individual nodes are increasing as well. With growing system sizes, the likelihood of faults per system is increasing. Transient bit faults on chips become more significant. On links, both transient and permanent faults have always been a problem. But not only the larger number of links increases fault rates. Transient fault rates per time increase with higher link data frequencies. Such faults can now be considered to be regular events, so that fault handling must be efficient and thus occur on the lower levels of the network.

This work presents the fault tolerant Extoll network protocol, aimed to be used in small- to large-scale direct interconnection networks. The protocol protects against transient faults of the link that are due to Gaussian noise. Control information is protected using forward error correction. Payload data is protected by a link-based retransmission protocol. Besides transient bit faults, the protocol protects against temporary and permanent link failures without any loss of data. As a result, the Extoll link protocol is, to the best knowledge of the author, the only successful implementation of a truly fault-tolerant network protocol based on 8b/10b link coding. Under the constraints given by the 8b/10b protocol and Extoll network requirements like small flit sizes and flit-based credits, the over-all maximum bandwidth utilization of the Extoll protocol of 85% is the best that can be achieved to guarantee fault tolerance. 8b/10b's efficiency of 80% further reduces the bandwidth. Thus, future work must evaluate closer how 8b/10b coding can be replaced, for example using mechanisms that are based on scrambling.

The High Availability Port is another significant improvement of fault tolerance. It is the first and only hardware-based routing mechanism that uses partitioned routing between intermediate nodes. In every large network, there is a good chance that at least one link is broken at any time. Thus, networks with a regular topology will in fact be irregular networks. The HAP allows an efficient and deadlock-free rerouting around faulty components, while all packets that are not directly affected by the fault can continue to use optimal routing algorithms.

All in all, this thesis developed new architectures and methods to solve two significant problems of modern network interface controllers: a significant increase in the latency of device to processor communication, and efficient and reliable error correction on network links. For both areas, solutions have been developed and implemented that are already successfully used in practice. As well, methods have been proposed and analyzed that may be used in future systems.

A Acronyms

API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
BAR	Base Address Register
BEC	Backward Error Correction
BER	Bit Error Rate, also Bit Error Ratio
CAM	Content-Addressable Memory
cHT	Cache Coherent HyperTransport protocol.
CMOS	Complementary Metal–Oxide–Semiconductor
DC	Direct Current
DCA	Direct Processor Cache Access
DIMM	Dual Inline Memory Module. A printed circuit board with a specified interface, typically holding a number of SDRAM chips.
DMA	Direct Memory Access
ECC	Error Correcting Code
FEC	Forward Error Correction
FIFO	First-In First-Out, a strategy for buffers

FIT	Failures In Time
FLIT	Flow-Control digIT, i.e. the smallest unit of flow control
FPGA	Field-Programmable Gate Array
FU	Functional Unit
HD	Hamming Distance
HPC	High Performance Computing
HT	HyperTransport protocol. May be followed by a number that specifies the maximum clock frequency in MHz.
HTX	HyperTransport eXpansion, the standard for HT slots
IN	Interconnection Network
IP	Intellectual Property
LAN	Local Area Network
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
MTU	Maximum Transmission Unit
nHT	noncoherent HyperTransport, see HT.
NIC	Network Interface Controller
NUMA	Non-Uniform Memory Architecture
PCB	Printed Circuit Board
PE	Processing Element
PIO	Programmed Input/Output
RMA	Remote Memory Access
SAN	System Area Network
SMP	Symmetric Multiprocessor

SMT	Simultaneous Multi-Threading
SO-DIMM	Small Outline DIMM.
SRAM	Static Random Access Memory
SRI	System Request Interface
TCA	Tightly Coupled Accelerator
TLB	Translation Lookaside Buffer
VC	Virtual Channel
VSM	Virtual Shared Memory
ZRAM	Zero Capacitor Random Access Memory

B Bibliography

[1] J. von Neumann. *First draft of a report on EDVAC*. Technical Report, University of Pennsylvania, 1945

[2] M. D. Godfrey, D. F. Hendry. *The Computer as Von Neumann Planned It*. IEEE Annuals of the History of Computing. Vol. 15, No.1, 1993

[3] M. J. Flynn. *Some Computer Organizations and Their Effectiveness*. IEEE Trans. on Computers C-21(9), pp. 938-960, September 1972.

[4] D. Sima, T. Fountain, P. Kacsuk. *Advanced Computer Achitectures: A Design Space Approach*. Addison Wesley, 1997.

[5] H. Sharangpani, K Arora. *Itanium Processor Microarchitecture*. In *IEEE Micro*, p.24-43, Sept.-Oct. 2000.

[6] X. Zang, A. Dasdan, M. Schulz, R. K. Gupta and A. A. Chien. *Architectural adaptation for application-specific locality optimizations*. In Proceedings of the 1997 IEEE international Conference on Computer Design, 1997.

[7] W. A. Wulf and S. A. McKee. *Hitting the Memory wall: Implications of the Obvious*. Computer Architecture News, 23(1), pp. 20-24, March 1995

[8] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C Koryrakis, K. Olukotun. *Transactional coherence and consistency: simplifying parallel hardware and software*. Micro, IEEE Volume 24, Issue 6, Nov-Dec 2004

[9] Shubhendu Sekhar Mukherjee. *Design and Evaluation of Network Interfaces for System Area Networks*. PhD. Thesis, University of Wisconsin-Madison, 1998

[10] Shubhendu Sekhar Mukherjee, M. D. Hill. *The impact of data transfer and buffering alternatives on network interface design*. Fourth International Symposium on High-Performance Computer Architecture, pp.207-218, 1-4 Feb 1998

[11] T. Gross, D. R. O'Hallaron. *IWarp - Anatomy of a Parallel Computing System*. MIT Press, Cambridge, Massachusetts, 1998

[12] C. Whitby-Strevens. *The Transputer.* Proceedings of the 12th Annual International Symposium on Computer Architecture (ISCA85), Boston, Massachusetts, U.S., June 1985

[13] Joseph Carbonaro and Frank Verhoorn. *Cavallino: The Teraflops Router and NIC.* Hot Interconnects IV. pages 157 160, 1996

[14] Ben Catanzaro. *Multiprocessor System Architectures*. Prentice Hall, Englewood Cliffs, NJ, 1994

[15] David Slogsnat. *Simulation and Architectural Exploration of a Shared-Memory Multiprocessor Node for Scientific Algorithms*. Diploma Thesis, University of Mannheim, 2002

[16] V. S. Pai, P. Ranganathan, and S. V. Adve. *RSIM reference manual, version 1.0*. Technical Report 9705, Rice University, 1997

[17] Sun Microsystems Inc. *OpenSPARC™ T2 System-On-Chip (SOC) Microarchitecture Specification*. July 2007

[18] Sun Microsystems Inc. *OpenSPARC™ T2 Core Microarchitecture Specification*. July 2007

[19] Harlan McGhan. *Niagara2 Opens the Floodgates*. Microprocessor Report, December 2006

[20] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Revision 3.11, 2005

[21] Advanced Micro Devices. *Software Optimization Guide for AMD Family 10h Processors*. Revision 3.04, September 2007

[22] Advanced Micro Devices. *AMD BIOS and Kernel Developer's Guide for the AMD Athlon 64 and AMD Opteron Processors*. Revision 3.3, 2006

[23] Wikipedia. *Streaming SIMD extensions*. Wikipedia Article, http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions, September 2007

[24] Wikipedia. *AltiVec*. Wikipedia Article, http://en.wikipedia.org/wiki/AltiVec, September 2007

[25] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. Dissertation, Yale University, 1986

[26] Lars Rzymianowicz, Ulrich Brüning, Jörg Kluge, Patrick Schulz and Mathias Waack. *ATOLL: A Network on a Chip*. Cluster Computing Technical Session (CCTEA) of the PDPTA'99 conference, in Las Vegas, June 28 - July 1 1999

[27] H. Fröning, M. Nüssle, D. Slogsnat, P. R. Haspel, U. Brüning. P*erformance Evaluation of the ATOLL Interconnect*. IASTED Conference, Parallel and Distributed Computing and Networks (PDCN), Innsbruck, Austria, February 2005

[28] U. Brüning, W. Giloi. *Future Building Blocks for Parallel Architectures*. In Proceedings of the 2004 International Conference on Parallel Processing (ICPP04), Montreal, Canada, 2004

[29] Lars Rzymianowicz. *Designing Efficient Network Interfaces For System Area Networks*. Dissertation, University of Mannheim, 2002

[30] Jon Beecroft, David Addison, David Hewson, Moray McLaren, Duncan Roweth, Fabrizio Petrini, Jarek Nieplocha. *QsNetII: Defining High-Performance Network Design*. IEEE Micro, vol. 25, no. 4, pp. 34-47, Jul/Aug, 2005

[31] Brightwell, Pedretti, Underwood. *Initial performance evaluation of the Cray SeaStar interconnect*. 13th Symposium on High Performance Interconnects, 17-19 Aug. 2005

[32] Smith, B. *The architecture of HEP.* On Parallel MIMD Computation: HEP Supercomputer and Its Applications, Ed. Massachusetts Institute of Technology, Cambridge, MA, 41-55, 1985

[33] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. *The Tera computer system*. SIGARCH Comput. Archit. News 18, 3b, Sep. 1990

[34] T. Halfhill. *Z-RAM shrinks embedded Memory.* Microprocessor Report, www.MPRonline.com, 2005

[35] Intel. *Intel® I/O Acceleration Technology.* Technology Brief, www.intel.com/go/ioat, 2006

[36] Intel. *Intel® 82598 10 Gigabit Ethernet Controller.* Product Brief, 2007

[37] Intel. *Intel® QuickData Technology Extends Flexibility of I/O Acceleration*. Technology@Intel Magazine, Volume 4, Issue 9, December 2006

[38] Intel. *Intel® 5000X Chipset Memory Controller Hub (MCH)*. Datasheet, September 2006

[39] Microsoft. *Scalable Networking: Eleminating the Receive Processing Bottleneck - Introducing RSS*. WinHEC, April 2004

[40] R. Huggahalli, R. Iyer, S. Tetrick. *Direct Cache Access for High Bandwidth Network I/O*. In Proceedings of the 32nd Annual international Symposium on Computer Architecture (June 04 - 08, 2005). International Symposium on Computer Architecture. IEEE Computer Society, Washington DC, pp50-59, 2005

[41] Intel. *Intel® Virtualization Technology for Directed I/O*. Architecture Specification, Revision 1.0, May 2007

[42] AMD. *AMD I/O Virtualization Technology (IOMMU) Specification*. Architecture Specification, Revision 1.20, February 2007

[43] David Kanter. *The Common System Interface: Intel's Future Interconnect*. Real World Technologies, http://www.realworldtech.com/includes/templates/articles.cfm?ArticleID=RWT082807020032, August 2007

[44] P. Conway, B. Hughes. *The AMD Opteron Northbridge Architecture*. IEEE Micro , vol.27, no.2, pp.10-21, March-April 2007

[45] Brian Holden. *Latency Comparison between HyperTransport and PCI-Express in Communications Systems*. HyperTransport Consortium White Paper, November 2006

[46] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification Revision 2.00b*. Document #HTC20031217-0036-0009, 2005

[47] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification Revision 3.00*. Document #HTC20051222-0046-0008, 2006

[48] HyperTransport Consortium. *HyperTransport EATX Motherboard/Daughtercard Specification.* www.hypertransport.org, 2004

[49] HyperTransport Consortium. *The Future of High Performance Computing: Direct Low Latency Peripheral-to-CPU Connections*. www.hypertransport.org, November 2005

[50] Duncan Bees, Brian Holden. *HyperTransport reduces delays in some applications*. EETimes 2004

[51] Alexander Giese. *Development and Verification of a HyperTransport-Interface with Optimizations for FPGA Environments*. Diploma Thesis, Universität Mannheim, 2006

[52] PCI-SIG. *PCI Express Base Specification 1.1*. 2005

[53] PCI-SIG. *PCI Express Base Specification 2.0*. 2007

[54] Heise Newsticker. *Neue Server Plattformen fuer zwei oder view Xeons*. http://www.heise.de/newsticker/result.xhtml?url=/newsticker/meldung/88368&words=Clarksboro&T=clarksboro, April 2007

[55] Leslie Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. IEEE Transactions on Computers, C-28(9):241-248, September 1979.

[56] Christian Scheurich, Michael Dubois. *Correct Memory Operation of Cache-based Multiprocessors*. In Proceedings of the 14th Annual International Sumposium on Computer Architecture, pages 234/243, June 1987

[57] Sarita V. Adve, Kourosh Gharachorloo. *Shared Memory Consistency Models: A Tutorial*. Computer, vol. 29, no. 12, pp. 66-76, Dec., 1996

[58] M. Dubois, C. Scheurich, and F. Briggs. *Memory access buffering in multiprocessors*. Proc. 13th Int'l Symp. Comp. Arch., pp. 434-442, June 1986.

[59] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. *Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors*. Proc. 17th Ann. Int'l Symp. Computer Architecture, 1990.

[60] D. Culler, J. Singh. *Parallel Computer Archtecture. A Hardware/Software Approach*. Morgan Kaufman Publishers, 1999

[61] R. Madukkarumukumana et al. *Performing Direct Cache Access Transactions Based on a Memory Access Data Structure*. WIPO patent, publication number WO/2007/078958 , 2007

[62] H. Hum, J. Goodman. *Forward State for use in Cache Coherency in a Multiprocessor System*. WIPO patent, publication number WO 2004/060678 A2, July 2004

[63] H. Hum et al. *Speculative Distributed Conflict Resolution for a Cache Coherency Protocol*. WIPO patent, publication number WO 2004/061677 A2, July 2004

[64] Beers, R. et al. *Non-Speculative Distributed Conflict Resolution for a Cache Coherency Protocol*. US Patent No. 6,954,829 B2. October 2005.

[65] L. Censier, P. Feautrier. *A new solution to Coherence Problems in Multicache Systems*. In IEEE Transactions on Computers, C(27):1112-1118, December 1978.

[66] A. Gupta, W.-D. Weber, and T. Mowry. *Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes*. In International Conference on Parallel Processing, pages I:312--321, Aug 1990

[67] M. M. Michael, A. K. Nanda. *Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors*. In Proceedings of the 5th international Symposium on High Performance Computer Architecture, HPCA, 1999

[68] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz. *An Evaluation of Directory Schemes for Cache Coherence*. In Proc. of the 15th Znt. Sym. on Computer Architecture, pages 280-289, May 1988

[69] Richard Simoni and Mark Horowitz. *Dynamic Pointer Allocation for Scalable Cache Coherence Directories*. In Proc. of the Int. Sym. on Shared Memory Multiprocessing, pages 72-81, April 1991

[70] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Version 023, 2007

[71] IBM. *PowerPC Architecture Book.* Version 2.02, 2005

[72] IBM. *IBM PowerPC 970FX RISC Microprocessor User's Manual*. Version 1.41, November 2004

[73] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Available from http://www.mpi-forum.org/docs, 1995

[74] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. Available from http://www.mpi-forum.org/docs, 1997

[75] OpenMP Architecture Review Board. *OpenMP Application Program Interface*. Version 2.5, Mai 2005

[76] P. M. Behr, S. Pletner, A. C. Sodan. *PowerMANNA: a parallel architecture based on the PowerPC MPC620*. Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, HPCA-6. , pp.277-286, 2000

[77] Tanabe, N.; Yamamoto, J.; Nishi, H.; Kudoh, T.; Hamada, Y.; Nakajo, H.; Amano, H. . *MEMOnet: network interface plugged into a memory slot*. Proceedings of the IEEE International Conference on Cluster Computing, pp.17-26, 2000

[78] Khunjush, F. and Dimopoulos, N. J. . *Lazy direct-to-cache transfer during receive operations in a message passing environment*. In Proceedings of the 3rd Conference on Computing Frontiers (Ischia, Italy, May 03 - 05, 2006), ACM Press, New York, NY, pp. 331-340, 2006

[79] Afsahi, A. and Dimopoulos, N. J. . *Efficient Communication Using Message Prediction for Cluster Multiprocessors*. In Proceedings of the 4th international Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications. Lecture Notes In Computer Science, vol. 1797. Springer-Verlag, London, 162-178, 2000.

[80] Binkert, N. L., Saidi, A. G., and Reinhardt, S. K. . *Integrated network interfaces for high-bandwidth TCP/IP.* In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems (San Jose, California, USA, October 21 - 25, 2006). ASPLOS-XII. ACM Press, New York, NY, pp. 315-324, 2006

[81] L. Spracklen, S. G. Abraham. *Chip Multithreading: Opportunities and Challenges*. In Proceedings of the 11th international Symposium on High-Performance Computer Architecture (February 12 - 16, 2005). HPCA. IEEE Computer Society, Washington, DC, pp. 248-252, 2005

[82] Manolis Katevenis. *Towards Light-Weight Intra-CMP Network Interfaces*. Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems, Stanford, California, 6-7 December 2006

[83] D. S. Henry and 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 (Boston, Massachusetts, United States, October 12 - 15, 1992). R. L. Wexelblat, Ed. ASPLOS-V. ACM Press, New York, NY, pp. 111-122, 1992

[84] S. L. Scott. *Synchronization and communication in the T3E multiprocessor.* SIGPLAN, pp. 26-36, Sep. 1996

[85] S. Scott, G. Thorson. *The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus*. HOT Interconnects IV, Stanford University, August, 1996

[86] Michael Schlansker, Nagabhushan Chitlur, Erwin Oertli, Paul M. Stillwell, Jr., Linda Rankin, Dennis Bradford, Richard J. Carter, Jayaram Mudigonda, Nathan Binkert, Norman P. Jouppi. *High-performance Ethernet-based Communications for Future Multi-core Processors*. Supercomputer Conference, SC07, November 2007

[87] C. Seitz, N. Boden, J. Seizovic, W.-K. Su. *Myrinet: A Gigabit-per-second Local Area Network*. IEEE Micro, vol 15, no.1, pp 29-36, 1995

[88] F. Petrini, Wu-chun Feng; A. Hoisie, S. Coll, E. Frachtenberg. *The Quadrics network (QsNet): high-performance clustering technology.* Hot Interconnects 9, 2001, pp.125-130, 2001

[89] M. Schlansker, N. Chitlur, E. Oertli, P. M. Stillwell, Jr., L. Rankin, D. Bradford, R. J. Carter, J. Mudigonda, N. Binkert, N. P. Jouppi. *High-performance Ethernet-based Communications for Future Multi-core Processors*. Supercomputing Conference, SC07, Nov 10-16, 2007

[90] Infiniband Trade Association. *Infiniband Architecture Specification Volume 1*. Release 1.2, October 2004, and *Infiniband Architecture Specification Volume 2*. Release 1.2.1, October 2006

[91] Cray Inc. *Cray XT3 Datasheet*. http://www.cray.com/products/xt3/index.html. 2005

[92] Cray Inc. *Cray XT4 Datasheet*. http://www.cray.com/products/xt4/index.html. 2006

[93] *Lustre Wiki*. http://wiki.lustre.org/

[94] Nikhil, R. S., Papadopoulos, G. M., and Arvind 1992. **T: a multithreaded massively parallel architecture*. In Proceedings of the 19th Annual international Symposium on Computer Architecture (Queensland, Australia, May 19 - 21, 1992). ISCA '92. ACM Press, New York, NY, 156-167, 1992

[95] Derek Chiou, Boon S. Ang, Arvind, Michael J. Becherle, Andy Boughton, Robert Greiner, James E. Hicks, James C. Hoe. *StartT-ng: Delivering Seamless Parallel Computing*. In Proceedings of EURO-PAR '95, Stockholm, Sweden, 1995

[96] B. S. Ang, D. Chiou, D.L. Rosenband, M. Ehrlich, L. Rudolph, Arvind. *StarT-Voyager: a flexible platform for exploring scalable SMP issues.* In Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (San Jose, CA, November 07 - 13, 1998). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, pp. 1-13, 1998

[97] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, John Hennessy. *The DASH Prototype: Implementation and Performance*. In Proceedings of the 19th International Symposium on Computer Architecture, pages 92-103, Gold Coast, Australia, May 1992

[98] F. Aono and M. Kimura. *The AzusA 16-Way Itanium Server.* In IEEE Micro September-October 2000, p.54-60

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

[100] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. *Blue Gene/L Torus Interconnection Network*. IBM J. Res. & Dev. 49, No. 2/3, 265–276, 2005

[101] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. *Overview of the Blue Gene/L system architecture*. BM J. Res. & Dev. 49, No. 2/3, pp. 195-212, 2005

[102] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Haring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura, and M. E. Wazlowski. *Blue Gene/L Compute Chip: Memory and Ethernet Subsystem*. IBM J. Res. & Dev. 49, No. 2/3, pp. 255–264, 2005

[103] The HyperTransport Consortium. *HyperTransport Technology I/O Link*. White Paper, July 2001

[104] Raza Microelectronics, Inc. *XLR Processor Product Overview.* May 2005

[105] U. Brüning. *Lecture Notes for Computer Architecture I*. University of Mannheim, Germany, 2007

[106] Intel Corporation. *Intel's Official Moore's Law Page*. http://www.intel.com/technology/mooreslaw/

[107] R. J. Drost, R. D. Hopkins, R. Ho, I. E. Sutherland. *Proximity communication*. IEEE Journal of Solid-State Circuits, vol.39, no.9, pp. 1529-1535, Sept. 2004

[108] Newisys. *ExtendiScale™ Technology: Large-Scale SMP Using AMD® Opteron™ Processors and Newisys® Horus ASIC*. White Paper, 2006

[109] Andi Kleen. *A NUMA API for Linux*. Technical Report, 2004

[110] John McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. http://www.cs.virginia.edu/stream/

[111] S. S. Mukherjee and M. D. Hill. *The Impact of Data Transfer and Buffering Alternatives on Network Interface Design*. Proceedings of HPCA98, Feb. 1998

[112] S. S. Mukherjee, M. D. Hill. *Using prediction to accelerate coherence protocols*. Proceedings of the 25th Annual International Symposium on Computer Architecture, pp.179-190, 27 Jun-1 Jul 1998

[113] Manuel E. Acacio, José González, José M. García and José Duato. *Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in cc-NUMA Multiproces-*

sors. Proc. of the SC2002 High Performance Networking and Computing, November 2002

[114] S. Kaxiras, C. Young. *Coherence communication prediction in shared-memory multiprocessors*. Sixth International Symposium on High-Performance Computer Architecture, 2000. HPCA-6. Proceedings, pp.156-167, 2000

[115] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki, B. Falsafi. *Memory coherence activity prediction in commercial workloads.* In Proceedings of the 3rd Workshop on Memory Performance Issues: in Conjunction with the 31st international Symposium on Computer Architecture (Munich, Germany, June 20 - 20, 2004). WMPI '04, vol. 68. ACM Press, New York, NY, pp. 37-45, 2004

[116] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki and B. Falsafi. *Temporal Streaming of Shared Memory.* In Proceedings of the 32nd Annual international Symposium on Computer Architecture (June 04 - 08, 2005). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, pp. 222-233, 2005

[117] M. P. Herlihy and J. E. B. Moss. *Transactional Memory: architectural support for lock-free data structures*. In Proceedings of the 1993 International Symposium on Computer Architecture (ISCA), San Diego, CA, May 1993

[118] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg., M.K. Prabhu, H. Wijaya, C. Kozyrakis, K. Olukotun. *Transactional Memory Coherence and Consistency.* In Proceedings of the 31st Annual international Symposium on Computer Architecture (München, Germany, June 19 - 23, 2004). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, p. 102, 2004

[119] R. Iyer, L. Bhuyan, A. Nanda. *Using Switch Directories to Speed Up Cache-to-Cache Transfers in CC-NUMA Multiprocessors*. IPDPS , p. 721, 2000.

[120] T. Schlichter. *Exploration of Hard- and Software Requirements for one-sided, zero copy user level Communication and its Implementation*. Diploma Thesis, Computer Architecture Group, University of Mannheim, 2003

[121] H. Fröning. *Architectural Improvements of Interconnection Network Interfaces*. Inaugural Dissertation, University of Mannheim, Jul. 9, 2007.

[122] H. Litz. *HTAX Specification*. Technical Report, Computer Architecture Group, University of Mannheim, 2007

[123] P. Haspel. *Researching methods for efficient hardware specification, design and implementations of a next generation communication architecture*. Inaugural Dissertation, University of Mannheim, 2007

[124] L. Schaelicke, A. Davis. *Improving I/O performance with a conditional store buffer.* MICRO-31. Proceedings. 31st Annual ACM/IEEE International Symposium on , pp.160-169, 30. Nov-2. Dec 1998

[125] H. Litz, H. Froening, M. Nuessle, U. Bruening. *A HyperTransport NIC for Ultra-low Latency Message Transfers*. Technical Report, 2007

[126] D. Bayer. *Designing the Network Port Element for the ExTOLL Network Chip*. Project Report, University of Mannheim, 2005

[127] D. Slogsnat, A. Giese, M. Nuessle, U. Bruening .*A Versatile, Low Latency HyperTransport Core*. Technical Report, 2008

[128] H. Fröning, M. Nüssle, D. Slogsnat, H. Litz, U. Brüning. *The HTX-Board: A Rapid Prototyping Station*. 3rd annual FPGAworld Conference, Stockholm, Sweden, Nov. 16, 2006

[129] S. Kapferer. *Design Space Analysis and Implementation of a Cache Coherent Device for HyperTransport*. Diploma Thesis, University of Mannheim, 2007

[130] F. Ueltzhöffer. *Design and Implementation of a Virtual Channel Based Low-Latency Crossbar Switch*. Diploma Thesis, University of Mannheim, 2005

[131] B. Geib. *Improving and Extending a Crossbar Design for ASIC and FPGA Implementation*. Diploma Thesis, University of Mannheim, 2007

[132] N. Burkhardt. *Fast Hardware Barrier Synchronisation for a Reliable Interconnection Network*. Diploma Thesis, University of Mannheim, 2007

[133] S. Schenk. *Configuration and Implementation of the Xilinx Multi Gigabit Transceivers*. Project Report, 2007

[134] H. Klimant, R. Piotraschke, D. Schoenfeld. *Informations- und Kodierungstheorie*. B.G. Teubner 2003

[135] D. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge: Cambridge University Press, 2003

[136] P. Koopman, T. Chakravarty. *Cyclic Redundancy Code (CRC) Polynomial Selection For Embedded Networks*. International Conference on Dependable Systems and Networks (DSN'04), p. 145, 2004

[137] J. Ray, P. Koopman. *Efficient High Hamming Distance CRCs for Embedded Networks*. International Conference on Dependable Systems and Networks, DSN 2006, pp.3-12, 2006

[138] T.-B. Pei and C. Zukowski. *High-Speed Parallel CRC Circuits in VLSI*. IEEE Trans. Comm., vol. 40, no. 4, pp. 653-657, Apr. 1992

[139] Sprachmann. *Automatic Generation of Parallel CRC Circuits*. IEEE Des. Test 18, 3 (May. 2001), pp. 108-114, 2001

[140] Yin-Tsung Hwang, Jiun-Yan Chen, Ming-Hwa Sheu. *Automatic Generation of Programmable Parallel CRC & Scrambler Designs*. IEEE Workshop on Signal Processing Systems Design and Implementation, SIPS '06, pp.286-291, Oct. 2006

[141] P. Subbiah. *Bit-Error Rate (BER) for high speed serial data communication*. Technical Paper, Cypress Semiconductor, 2006

[142] L. Thon, H.-J. Liaw. *Error-Correction Coding for 10Gb/s Backplane Transmission*. DesignCon 2004

[143] Intel. *Intel® Connects Cables. High-Performance 20 Gbps Optical Cables*. Product Brief, 2007

[144] J. von Neumann. *Probailistc logics and the synthesis of reliable organisms from unreliable components*. Automata Studies, in Annals of Mahtematical Studies no. 34, pp. 43-98, Princeton University Press, 1956

[145] P. Lala. *Self-Checking and Fault-Tolerant Digital Design*. Academic Press/Morgan Kaufmann Publishers, 2001

[146] R. Baumann. *Soft errors in advanced computer systems*. Design & Test of Computers, IEEE, Volume 22, Issue 3, Page(s):258 - 266, May-June 2005

[147] P. Hazucha, T. Karnik, J, Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Hareland, P. Armstrong, S. Borkar. *Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25-um to 90-nm generation*. Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International, pp. 21.5.1-21.5.4, 8-10 Dec. 2003

[148] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, C. W. Wahaus. *IBM experiments in soft fails in computer electronics (1978-1994)*. IBM Journal of Research and Development, Vol. 40, No.1, 1996

[149] S.S. Mukherjee, J. Emer, S.K. Reinhardt. *The soft error problem: an architectural perspective*. 11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11, pp. 243-247, 12-16 Feb. 2005

[150] A. X. Widmer, P. A. Franaszek. *A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code*. IBM Journal of Research and Technology, Volume 27, Number 5, Page 440, 1983

[151] Rick Walker, Birdy Amrutur, Tom Knotts, Richard Dugan. *64b/66b coding update*. IEEE 802.3ae, Albuquerque, 3/6/2000

[152] Suresh Chalasani, Rajendra V. Boppana. *Communication in Multicomputers with Nonconvex Faults*. IEEE Transactions on Computers ,vol. 46, no. 5, pp. 616-622, May, 1997

[153] Chun-Lung Chen, Ge-Ming Chiu. *A Fault-Tolerant Routing Scheme for Meshes with Nonconvex Faults*. IEEE Transactions on Parallel and Distributed Systems ,vol. 12, no. 5, pp. 467-475, May, 2001

[154] R.A. Reed, M.A. Carts, P.W. Marshall, C.J. Marshall, O. Musseau, P.J. McNulty, D.R. Roth, S. Buchner, J. Melinger, T. Corbiere. *Heavy ion and proton-induced single event multiple upset*. IEEE Transactions on Nuclear Science, vol.44, no.6, pp.2224-2229, Dec 1997

[155] Maria Engracia Gomez, Nils Agne Nordbotten, Jose Flich, Pedro Lopez, Antonio Robles, Jose Duato, Tor Skeie, Olav Lysne. *A Routing Methodology for Achieving Fault Tolerance in Direct Networks*. IEEE Transactions on Computers ,vol. 55, no. 4, pp. 400-415, April, 2006

[156] Young-Joo Suh, Binh Vien Dao, Jose Duato, Sudhakar Yalamanchili. *Software-Based Rerouting for Fault-Tolerant Pipelined Communication*. IEEE Transactions on Parallel and Distributed Systems, vol. 11, no. 3, pp. 193-211, March, 2000

[157] C. Ho and L. Stockmeyer. *A New Approach to Fault-Tolerant Wormhole Routing for Mesh-Connected Parallel Computers*. IEEE Trans. Comput. 53, 4 , pp. 427-439, Apr. 2004

[158] Donald H. McMahon, Alan A. Kirby, Bruce A. Schofield, Kent Springer. *Data and forward error control coding techniques for digital signals*. US Patent Number 5144304, 1992

C List of Figures