Institutt for Informatikk
Universitetet i Oslo

Preface

This is our thesis for the degree Candidatus Scientarium at the University of Oslo, Department of Informatics.

Our advisors to this thesis have been Ernst H. Kristiansen, Oddvar Søraasen and Yngvar Lundh. Oddvar Søraasen and Yngvar Lundh are employed at the Department of Informatics. Ernst H. Kristiansen was previously working at Dolphin Server Technology. Currently he is employed at SINTEF, Oslo. We would like to thank them all for their help and guidance.

The following at the Department of Informatics gave us some valuable advice: Stein Gjessing, Øystein Gran Larsen, and Sverre Johansen.

Thanks to NTNF for covering our traveling expenses to the Open Bus Systems '92 conference in Paris.

A special thanks to all our friends at the Institute, and also to all our friends in the "Realistforeningen" and the "Cybernetisk Selskab". Thanks to them all our years at the university had one important element: fun ☺☺☺!

University of Oslo, February 9, 1993

John W. Bothner Trond Ivar Hulaas

Set in 11pt NewCentury Schoolbook using LaTeX.
Drawings made with idraw and graphs with the spread-sheet Wingz.

Contents

List of Figures

List of Tables

Glossary

agent A switch or bridge between the requester and the responder. During normal operation the agent's intervention is transparent to the requester and responder.

busies A "busy" is the term for a negative acknowledgement in SCI. If the input-buffers of an SCI-interface is occupied, it sends a "busy" to the node wishing to access the input-buffers.

coherence Multiple copies of data are coherent if they are logically consistent.

diameter The diameter of a network is the maximum distance between two points in a network. Sometimes called the maximum number of hops.

flit is the smallest unit of information that a queue or channel can accept during one clockcycle. A "symbol" in the parallel version of SCI is a flit.

latency is the time for a packet to traverse a network from the sender to the receiver.

node has unfortunately three different meanings, depending on the context:

1. [IEEE-SCI] use the word to mean "something" that has an SCI-interface. This "something" is either a switch (agent), or a unit including for example a processor and a SCI-interface.
2. In various articles, eg. [JohnGood,Agarwal,Dally90] the word "node" is used to mean an intersection in a k-ary n-cube, which typically has a switch with a processor attached to it. We use instead the word "vertex".
3. In our simulation program, a node is an entity containing a processor and a SCI-interface. We will try to use the term "active node" to stress the fact that we speak of a node that takes the initiative to send requests and responses.

requester The node that initiates a transaction. This is done by initiating a request subaction.

responder The node that completes the transaction. This is done by initiating the response subaction.

ringlet The closed path formed by the connection that provides feedback from the output link of a node to its input link. This connection may include other nodes and/or switch elements.

scalable A concept is efficient in a system independently of the system's size.

symbol See *flit*.

transaction An information exchange between two nodes. A transaction consists of request subaction and a response subaction. The request subaction transfers commands (and sometimes data) between a requester and a responder. The response subaction returns status(and sometimes data) from the responder to the requester.

throughput The amount of traffic in a network per unit of time. Usually measured in bytes per second.

vertex (plural: vertices) Webster: "the termination or intersection of lines or curves". In this thesis it refers to the intersections in a k-ary n-cube. Examples: a square has 4 vertices, a 3-cube has 8 vertices, and a 3-ary 2-cube has 9. See chapter 4 for further explanation.

1

Introduction

High-performance computers with hundreds of processors are increasingly needed for applications like weather-forecasting, simulations and expert systems. To achieve an overall high performance in a computer system, every part of the system has to deliver a performance which equals the other parts of the system. In a system with a large number of processors working together, the interconnection system is of great importance to the performance. How the interconnection is made up is the main focus in this thesis.

We will study various interconnection networks for multiprocessors. Of special interest is the class of interconnection networks called k-ary n-cubes. To evaluate the various interconnects, we have made a simulator that simulates various aspects of k-ary n-cubes. Aspects to be studied are first of all the latency and throughput of k-ary n-cubes. These k-ary n-cubes run protocols as specified by SCI[1].

SCI is an IEEE multiprocessor standard approved in march 1992. Work on SCI began in late 1987 when several members of the IEEE Futurebus project felt that buses would not meet the demands of future multiprocessors due to mainly the following factors: the one-broadcast-at-a-time property, clock rates limitations, and poor scalability [Gustav]. A study group soon concluded that a solution could involve the use of packet-based signaling over multiple point-to-point links, thus avoiding the physical limitations of buses.

The most active participants in the evolution of SCI are the following: Apple Computers, CERN, Dolphin Server Technology, Hewlett-Packard, Stanford Linear Accelerator Center, the University of Wisconsin, and the University of Oslo.

We make an informal comparison of SCI and SWIPP[2]. SWIPP is a multi-computer-study at the University of Oslo. The main idea of the SWIPP concept is to connect heterogeneous nodes by using fiber-optics and switches. Each node has an extra device which performs the networking/communication tasks on behalf of the node.

[1] **SCI :** The Scalable Coherent Interface IEEE 1596-1992

[2] **SWIPP :** Switched Interconnection for Parallel Processors

Several students are writing their thesis as part of the SWIPP-project. A prototype SWIPP-network has been made, which is currently under testing [NerSmaTor]. Contact has been made with CERN with the aim of using SWIPP in data-acquisition.

This thesis is organized in the following manner:

Chapter 2 In this chapter the field of parallel processing is presented. First the basic architecture of a traditional computer is presented. Then we explain essential terms such as streams, coupling, shared memory, message passing, multiprocessors, multicomputers and cache coherency.

Chapter 3 The basics of interconnection networks are explained in this chapter: wires, switches and switching-techniques. The difference between static and dynamic topologies is explained. Meshes and cubes are presented, along with single-stage and multi-stage networks.

Chapter 4 In our simulation we use the interconnection network called k-ary n-cubes. In this chapter the more theoretical aspects of k-ary n-cubes are discussed.

Chapter 5 The Scalable Coherent Interface (SCI) standard forms the basis of our study of k-ary n-cubes. In this chapter SCI is introduced.

Chapter 6 In this chapter we present SWIPP. An informal discussion of SWIPP and SCI is then made.

Chapter 7 presents several ways of constructing topologies using rings.

Chapter 8 In this chapter we present our simulator. Objects and algorithms that represent the nodes and protocols, are presented.

Chapter 9 We here show the results of our simulations.

Appendix A In this appendix we take a brief look at previous and existing standards for computer-buses.

Appendix B Here we explain briefly the simulation tools we considered using.

Appendix C This appendix contains two articles. The first article a reprint of the article "Various interconnects for SCI-based systems", published in the proceedings of Open Bus Systems '91.

The second article was published in the proceedings of CAMAC '92. It is entitled "Behavior of Scalable Coherent Interface in larger systems", and is based to a large extent on the findings presented in this thesis. It also serves as a rough summary to this thesis.

Appendix D This appendix explains how others may use our simulator.

2

A Brief introduction to parallel processing

This chapter presents the background material for this thesis. First a brief overview on how most current computers work is presented. Then a general introduction to parallel processing and some of the important terms used are explained.

2.1 von Neuman

Since the 1940's and up to the present, computers have had the same basic architecture. It has been made up of three basic elements: a processor, a memory, and some form of input/output [Tanen90]. A bus has then been used to connect these elements together, as shown in figure 2.1. This is often referred to as the von Neuman architecture. Closely

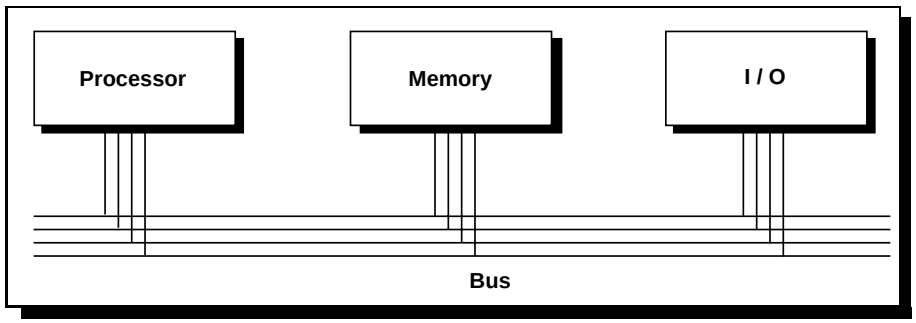


Figure 2.1: Traditional computer architecture.

associated with this architecture is the following, very sequential procedure carried out by the processor (the instruction cycle).

1. Fetch the next instruction from the memory over the bus.
2. Update the "program counter" in the processor. A program counter is a reference (placed in a register) to the next instruction to execute.

3. Decode the instruction (in other words: find out which instruction it is).
4. Fetch if necessary operand(s) from the memory over the bus.
5. Execute the instruction.
6. Put the result somewhere, either in a register in the processor, or send it to the memory over the bus.
7. Go to step 1.

This procedure and the architecture that goes with it has remained basically unchanged during the last 40 years. Nevertheless the performance of computers has improved dramatically. This is due to to the following (mainly from a hardware-perspective):

- ❏ The invention of the transistor
- ❏ Virtual memory
- ❏ Use of caching
- ❏ VLSI – Very Large Scale Integration
- ❏ Pipelining

The present trend is that processors are getting faster and memories larger. The bus connecting them has not improved within the same order of magnitude (the *von-Neuman bottleneck*) [Tanen90]. The bus is a shared resource (one user at a time) that tends to become saturated. As the wish to put more devices (e.g. more processors and memories) on the bus becomes more pronounced, this saturation becomes a greater problem.

2.2 Goal: Faster computers

The demand for more processing power will continue to increase. There are 2 ways to satisfy this demand.

1. Improve technology. This can be done by increasing the level of integration or increase the switching speed of the chip. Increasing the level of integration implies more logic gates per unit area on the chip. A speed increase means that the chip will use more power, which will put a restrain on the scale of integration (mainly due to heat development). The quest for higher performance then leads to more research on new chip technologies, resulting in for example new chip materials (eg. GaAs), lower signal switching levels etc. The limits of these approaches are the speed of light and the unavoidable capacitances in the materials used.
2. Make the computer do tasks in parallel. For that multiple processors and memories are needed. This in turn demands an improved interconnect. This is the topic of this thesis.

2.3 The basics of parallel processing

What is parallel processing? It is basically to subdivide a task into multiple parts, each which can be executed on independent pieces of hardware simultaneously. This division is most likely carried out by software (typically the operating system). The net result should be a reduction in the overall execution-time.

A more formal definition has been given by [HwaBri]: [1]

> *"Parallel processing is an efficient form of information processing which emphasizes the exploitation of concurrent events in the computing process. Concurrency implies parallelism, simultaneity, and pipelining. Parallel events may occur in multiple resources during the same time interval; simultaneous events may occur at the same time instant; and pipelined events may occur in overlapped time spans. ..."*

Note especially the keywords *concurrent events* and *multiple resources*: events are happening concurrently on multiple pieces of hardware (processors).

There are three main points to consider when analyzing or synthesizing a parallel computer [Tanen90]:

1. The nature of the processing elements themselves. Their size and speed, and whether there is a single processing element on a chip, or several.

2. The nature of the memory modules. Memories are often physically split up into multiple modules.

3. The nature of the network connecting the processing elements and the memory modules together. This is the main topic of this thesis. This is discussed further in chapter 3.

A computer with n processors should ideally be n times faster than a computer with a single processor (assuming processors are identical). That is *not* the case in practice, for a number of reasons:

1. Communication is not instantaneous.

2. A processor has to wait for data from a memory-node.

3. A processor has to wait for data from another processor.

4. Parts of the code *has* to be sequential because of the nature of the application.

5. A processor has to wait for available bandwidth, it cannot yet access the interconnection network.

[1] On page 6.

6. Inefficient algorithms that do not exploit the available hardware are used. In other words: unnecessary sequentialism.

7. Sharing of writable data – the cache coherence problem.

For the future: little can be done about 1-4, but with 5-7 progress can definitely be made.

The estimated speed of a parallel computer relative to using a single equivalent processor is often referred to as **speedup**. We here formally define speedup as [Seitz]:

$$S(n) = \frac{\textit{time on 1 node}}{\textit{time on n nodes}} \tag{2.1}$$

As explained above , it is impossible for the speedup to be n. The pessimists say speedup is $\log_2 n$ (Minsky's conjecture). A more optimistic view is that the speedup is $n/\ln n$. See figure 2.2.

Amdahl points to the fact that the speedup is dependent on the percentage of sequential code f in a given program [Quinn]:

$$S_{Amdahl}(n) \leq \frac{1}{f + \frac{1-f}{n}} \tag{2.2}$$

If we assume that 10% of the code is sequential, then

$$S_{Amdahl}(n) \leq \frac{10}{1 + \frac{9}{n}} \tag{2.3}$$

The speedup can then never be more than 10 (according to Amdahl). This is shown in figure 2.2.

If f approaches zero (naive) then S_{Amdahl} approaches the ideal case:

$$\lim_{f \to 0} S_{Amdahl}(n) \leq n \tag{2.4}$$

Most commercial multiprocessors today typically have 2-4 processors. Using Minsky's conjecture a system containing for example 4 processors, only work twice as fast as a single processor. If one leans toward $n/\ln n$ then it works roughly three times as fast as a single processor. Also Amdahl's law points to the latter (for $f = 10\%$).

A commonly used term specifying the amount of parallelism used is the **grain-size** of the system. For example, a time-sharing system with multiple users demands relatively little communication between independent pieces of software. This is typical of *coarse-grained parallelism*. The opposite is *fine-grained parallelism*. An example of the latter is vector-processing, where multiple processors are working closely together on the same problem.

It is a probable trend that the grain size of parallel computers will decrease. This results in messages being sent more often and fewer instructions are then executed in response to each message. Thus communication latency becomes an increasingly important factor [Dally90].

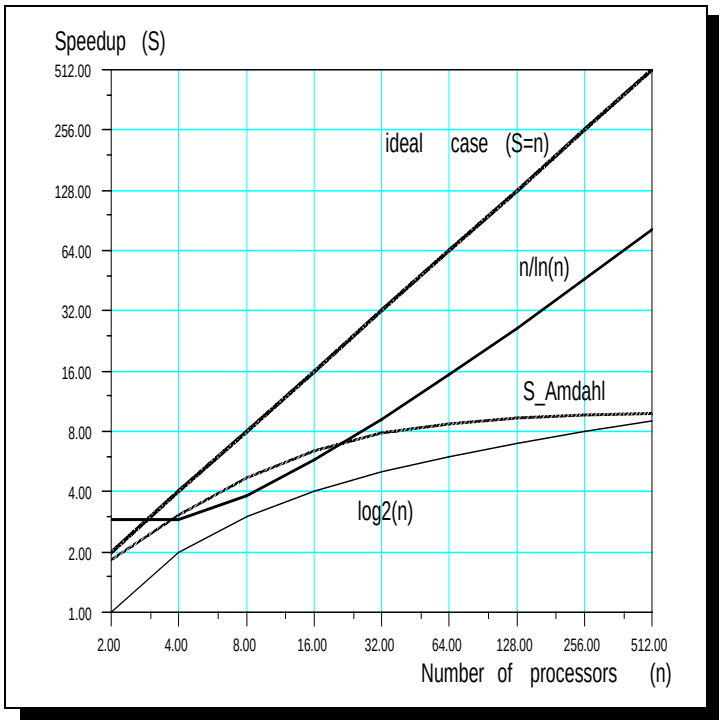


Figure 2.2: The estimate of the speedup S of a system of n processors. Adapted from [HwaBri].

2.3.1 Flynn's streams

The most classic classification of computer-architectures is probably that done by Flynn. To Flynn the main task of a computer is the execution of instructions on a set of data. An instruction stream is the set of instructions executed by processor(s) on one or multiple sets of data. A data stream is the set or sets of data the instruction stream operates on. He divided architectures into 4 basic categories [HwaBri]:

SISD Single instruction stream, single data stream. This is the same as the traditional von Neuman-configuration with a single processor and a single memory. Most computers today fall into this category.

SIMD Single instruction stream, multiple data streams. In this category a single instruction is performed simultaneously on multiple operands at the same time (synchronously). This is also known as *vector-processors*, or *array-processors*. Examples are the Illiac IV, the Connection Machine and Crays.

A variation of SIMD-machines is M-SIMD: multiple SIMDs. In contrast to a SIMD-machine, it has multiple control units, each operating on a subset of the processors.

MISD Multiple instruction streams, single data stream. Various instructions are simultaneously performed on the same operand. No real implementation of this class has been constructed.

MIMD Multiple instruction streams, multiple data streams. In this category multiple operations are performed simultaneously on independent operands. A program must be broken into several independent pieces which execute independently (asynchronously). This category is better known as *multiprocessors* or *multicomputers*, depending on how memory is organized. See section 2.3.3. Well-known MIMD-machines include the BBN Butterfly, the Intel iPSC, and some implementations using the Inmos Transputer.

2.3.2 Coupling

Another important classification scheme is that of coupling. A parallel architecture is said to be either *tightly* or *loosely* coupled [HwaBri]:

Loose The architecture consists of relatively independent entities, each with its own processor, memory, and perhaps its own I/O channels. The network latency is usually not so critical. This is also known as a *multicomputer*. See figure 2.3.

Tight The architecture consists of a set of processors that share a common main memory and is controlled by an operating system that provides for interaction between processors and their programs. The processors may have their own private memory. See figure 2.4. With a tight configuration the network latency becomes a critical parameter, because more memory-fetches have to be done via the interconnection network.

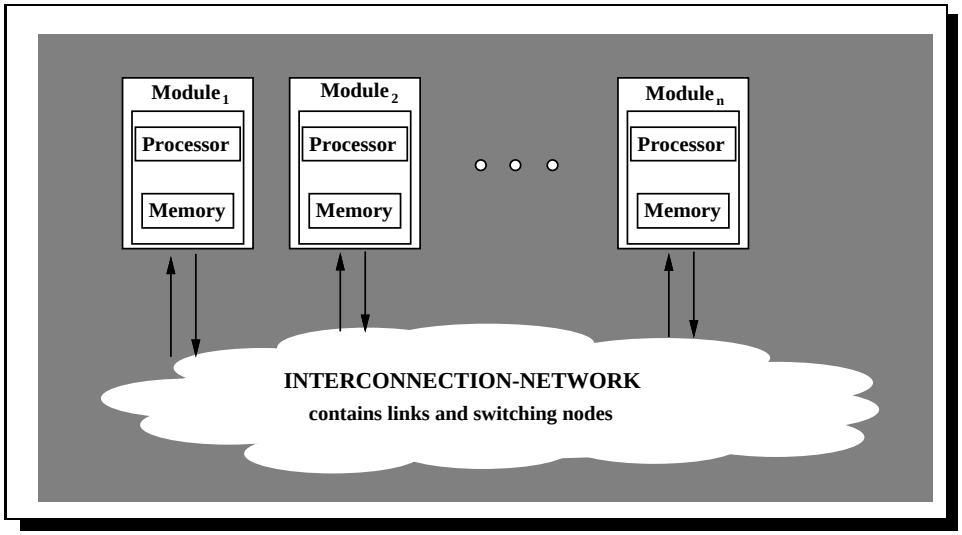


Figure 2.3: Loosely coupled architecture.

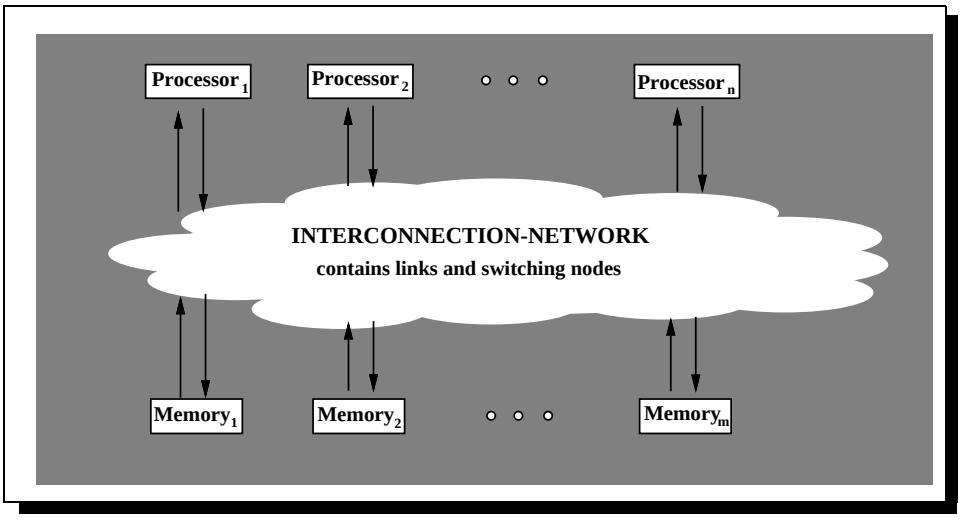


Figure 2.4: Tightly coupled architecture. Note that the processors may have some private memory. The memory modules are together a shared memory.

2.3.3 The organization of memory

There are 2 major ways to organize memory: by using a **shared memory** or by using **message passing** [Tanen90]. With shared memory there is one common address-space for all the processors. This simplifies the sharing of code and data-structures among the processors. If instead the processors only have their own private memory, they share data by sending messages. The latter, called "message passing" is often referred to as a **multicomputer** (loosely coupled).

A computer with shared memory is often called a **multiprocessor** [2]. There are various ways to organize shared memory. One way could be to have a single, large memory module. That is the conservative approach. A better way could be to divide them into multiple memory-modules, maybe with interleaved (low-ordered) addressing.

It is important to distinguish between physical sharing and logical sharing. With physical sharing there is one single shared address-space. A value written to a word with one processor can be read by another processor. If such physical sharing does not exist, data must be sent.

Logical sharing is used if the software (and the programmer) *assumes* it is accessing a shared memory, when it is not. Instead an abstract single memory-space is accessed by the programmer that hides the fact that message-passing is used.

The two concepts physical and logical sharing can both appear at the same time, giving a total of four combinations. If the software and the hardware both "see" only private unshared memory, message passing must be used to share data. If the software and the hardware "see"

[2] Multiprocessors and multicomputers are terms used relatively unformally in the literature. Therefore some might disagree with the stated definitions.

the same address-space we have a "straightforward" shared memory. A less common alternative is to simulate message passing and a logically unshared memory on top of a physically shared memory.

The most interesting combination consists of simulating a logical shared memory "on top of" a physically disjoint memory. It is easier for programmers to relate to shared memory instead of message passing. The Linda programming model [Carrier] is an example of an implementation of logical shared memory. In Linda processes access an abstract "tuple space" (the logical shared memory). This "tuple space" could be distributed to independent workstations. In table 2.1 the various combinations of physically and logically shared memory are shown.

Shared memory versus Message passing What are the pros and cons of using a shared memory or a message-passing architecture? We here present some of them, largely based on our own impressions:

- Shared memory: "Easier to use, but more difficult to construct."
  - \+ This is the "traditional" programming environment. Shared memory is presented to the programmer as one large, uniform address-space. This is familiar to all programmers. Thus more programmers can be persuaded to program a shared memory computer. If this is the case more programs will be available for shared memory computers.
  - +/– There is less copying of data by the nodes. Messages often contain only references to the data in question and the operation to perform on the data.
  - +/– Latency is more critical. This is because requests are sent more often, and these requests are often for small amounts of data. Thus it is reasonable to assume that the grain-size of a shared memory computer is likely to be smaller. Since packets tend to be smaller, the relative overhead due to headers is larger.
  - – Prefetching is somewhat more difficult. Prefetch is the ability to guess in advance the next block of data desired by the processor, and thus fetch it over the network in advance.

  According to [Bell] the trend the last few years has been leaning toward shared memory computers.

- Message passing: "Easier to construct, but more difficult to use."
  - \+ Prefetching is easier.
  - +/– Latency is not so critical and grain-size is larger. Header overhead is smaller, but in message-passing there is a "data-overhead": more data than necessary are often sent.
  - – A drawback with message passing is that the programming environment is more complex than on a traditional one-memory-space computer. This is a major obstacle if someone is

to program a message-passing computer. It can be overcome by having an abstract shared memory "on top of" message-passing which the programmer uses (a logically shared memory).

- Messages contain the operation and the data itself, not just a reference to the data. This could result in more copying of data by the nodes. More memory-space is then needed in the nodes.

Both message passing and shared memory have their strength and weaknesses. Message passing is probably best for problems of larger grain-size, while shared memory is maybe more optimal for problems of smaller grain-size.

Physically	**Logically**	
Unshared	Unshared	Multicomputer. Uses message passing.
Unshared	Shared	Distributed virtual memory. E.g. the Linda paradigm
Shared	Unshared	Message-passing by shared buffers possible.
Shared	Shared	"Straightforward" shared memory multiprocessor

Table 2.1: Various combinations of shared memory in software (logically) and in hardware (physically). Adapted from [Tanen90].

Organization of memory versus coupling We have here chosen to separate the concept of coupling from the concept of the organization of memory. In reality there is a close relationship, as can be seen in figure 2.3 and 2.4. In the loose architecture shown in figure 2.3 it is natural to assume the modules communicate by using message passing. The tight architecture in figure 2.4 is a typical shared memory architecture. [Dally87][3] and others do not distinguish between the two concepts.

2.3.4 The cache coherence problem

A problem with designing shared memory multiprocessor-systems is the contention for memory-access that arise, and also the contention for the interconnect itself. These problems contribute to increase memory access time. This in turn results in the processors having to slow down. Thus the processors are often idle, while waiting for data from memory. The use of caches in each processor reduces this problem, but then the problem of data consistency in the caches arises. This is often referred to

[3] On pages 8-10.

as "the cache coherence problem" [Stenström, Chaik et al, Goor, PatHen]. It gives rise to the following problems:

Sharing of writable data : Multiple copies of the same data result in problems if the data is writable. Data incoherency might arise in two forms: between multiple caches and between a cache and memory. An example of the former: cache C_i and cache C_j both has a copy of line X from memory. C_i then modifies X. C_j must then somehow be made aware of the change.

Process migration : Sometimes there is a wish to move a process from a processor P_i to a processor P_j. A reason could be to achieve load balancing. A problem may occur when P_i modifies line X in its cache before the process migrates to P_j and before X is updated in memory. P_j will then read the *old* X value in memory.

I/O : There is also the risk of incoherent data being written to disk. A block from main memory could be written to the disk before a line X is modified by a cache. Thus main memory had an old value of X, resulting in the wrong data being written to disk.

There are many schemes proposed to solve the cache coherence problem. Some solve it by using an interconnect with broadcast properties (usually a bus). This is referred to as "snooping": the cache-controller keeps constantly a close watch on the interconnect. Another way is to somehow keep a list of the cached blocks (directory-based schemes). It is also possible to solve the cache coherence problem in software.

Figure 2.5 shows a taxonomy of various coherency-schemes.

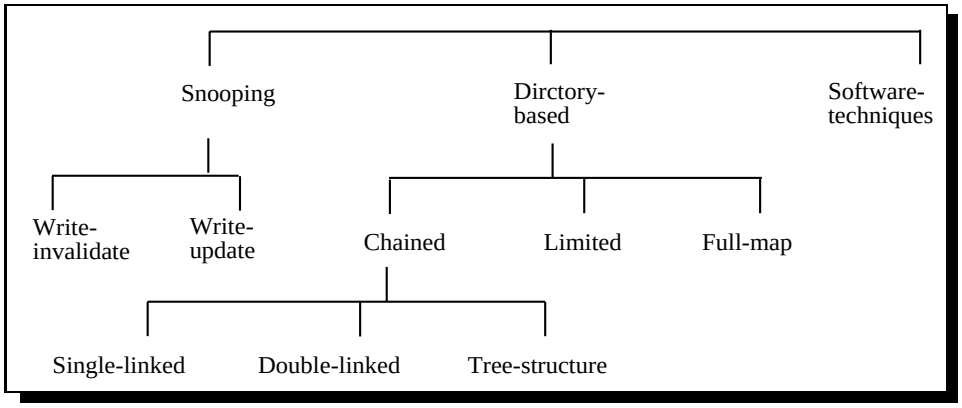


Figure 2.5: A taxonomy of cache-coherency schemes, somewhat hardware-biased.

2.3.4.1 Snooping

The coherency scheme called snooping is probably the most popular today. It is relatively easy to implement, most commercial multiprocessors available today use this scheme.

This technique is closely tied to the use of a broadcast-type interconnect – a bus. The cache controllers keep a close "watch" on what is transmitted on the bus. If they "hear" that "someone else" is modifying a block that they have in their own cache they take the appropriate action to maintain cache coherency.

There are two types of snooping protocols. They differ in the way they handle writes. They are called *write invalidate* and *write update*. The latter is also known as write broadcast.

Write invalidate works the following way: the writing processor invalidate the copies of the block in other caches. This is often done by a special invalidation signal on the bus. When this is done the writing processor is free to modify the block. If one of the other caches or the memory later wants to read the block, the cache which last modified it, supplies the block.

With write update, the writing processor broadcasts the new data on the bus. The other cache controllers then see if they have the block in their caches. If that is the case, they all update their caches simultaneously with the new data.

The drawback of snooping is its dependence on a broadcast-type interconnect. Broadcasts are difficult to route efficiently through switches [James et al], thus snooping is constricted to bus-type interconnects. That limits the scalability of snooping.

2.3.4.2 Directory-based schemes

The problem with snooping protocols is that they depend on the use of buses. Buses represent a bottleneck with respect to bandwidth (saturation).

If one is to avoid the use of broadcasts then the location of the cached copies of the block must be stored somehow. This list of the copies' location is commonly called a directory. It can be centralized (e. g. in the memory) or distributed among the caches. There are three types of directory-based schemes: full-map, limited or chained. Directory based schemes are suited to various interconnection networks.

Full-map scheme: In the full-map scheme the directory is associated with memory. For each block in memory there is a vector containing several bits (as many bits as there are caches). Each bit is associated with a specific cache in the system. The bit is set if the block is in the respective cache.

Associated with each block is also an additional dirty bit, which is set when the cache is modifying the block. For a cache to do so it must have "valid" data and write permission.

The full-map protocol is relatively straight-forward, but it has limited scalability with respect to overhead in memory.

Limited directory scheme: The limited scheme is an alternative to reduce the overhead of the full-map scheme. Here we restrict the number of cache-pointers in the directory entry to less than the total

number of caches. The difficulty is to handle the case when more cache-pointers are requested. There are two alternatives, either to disallow more copies of the block or somehow broadcast that more copies exist.

Chained directory: In the chained directory scheme the directory is physically spread out among the caches. Starting at the cache line in question there is a chain of pointers "connecting" the caches together. The memory-overhead is made up of a clean/dirty -bit and a pointer to the first cache in the chain. The advantage with this scheme is that it is scalable with respect to the directory-size. If another cache wishes to enter the sharing list, the directory is increased by one. The location of one cache is put in another cache.

In figure 2.6 and 2.7 the general idea is shown. In figure 2.6 cache_1 is the only cache having a copy of cache line x. The memory has a pointer to the first cache in the chain, presently cache_1. If more caches were sharing the cache line, cache_1 might have a pointer to the next cache sharing the cache line. Presently only cache_1 is in the sharing list, so cache_1 only has a "chain-termination"(CT) pointer. If cache_2 also wants to access cache line x, it puts itself as the new head in the chain. This is illustrated in figure 2.7. Cache_2 requests cache line x from memory. The memory sends a copy to cache_2, along with a pointer to cache_1. The memory then deletes its pointer to cache_1 and points to cache_2 instead. Cache_2 keeps its pointer to cache_1, which remains at the end of the chain.

For a processor to write to the cache line it must become the head of the chain. The rest of the chain must then be purged, along with the copies of the cache line.

The difficulty with these chains arises when a cache in the middle of the chain has to be removed. This could happen when a cache line in a cache was purged because of a cache miss. Since the cache does not have a reference to the previous cache in the chain, there has to be a sequential search along the chain. The time for this search increases linearly with the length of the chain.

The best solution to this problem is probably to have a two-way chain, so that a cache also has a reference to the previous cache in the chain. The drawback is that the overhead of the cache tags increases, although the overhead remains independent of the length of the list. SCI uses this scheme (see section 5.4).

Not everybody thinks scalable cache coherency is a major issue for large systems. [Matloff] argues that scalable cache coherency is inefficient for large systems. He proposes instead to have multiple domains of "local coherency". Only the processors within such a domain should share variables.

2.3.4.3 Software techniques

It is possible to let software take care of the cache coherence problem, instead of the hardware. One conservative solution could for instance be to forbid shared writable data to be cachable. Another possible approach

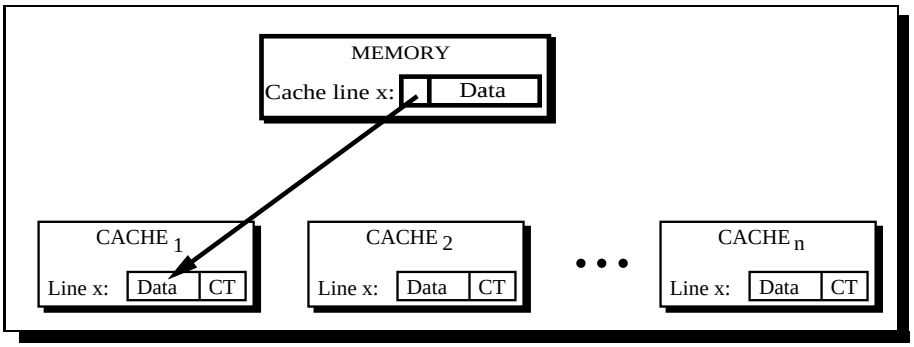


Figure 2.6: A chain with only a single cache. Adapted from [Chaik et al].

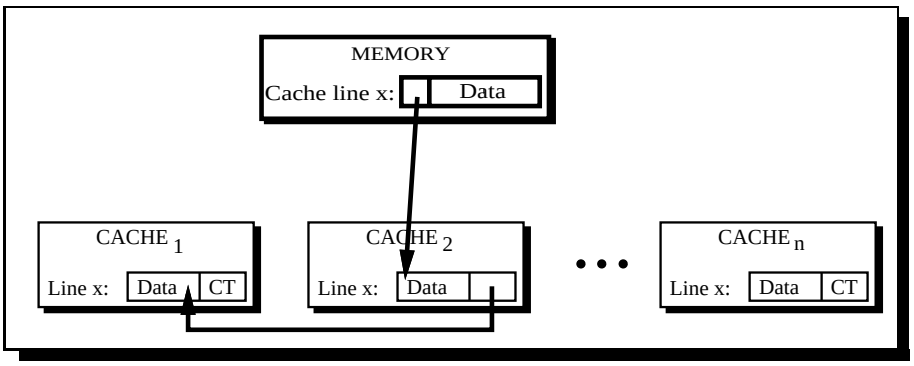


Figure 2.7: A new cache has inserted itself into the chain. Adapted from [Chaik et al].

could be to let the compiler analyze the program to see when it is safe to cache a shared variable [Stenström]. There are other possibilities but this is somewhat beyond the scope of this thesis.

3

Interconnection Networks

The previous chapter gives a rough overview of aspects of parallel processing. But what about the interconnection network? In figure 2.3 and 2.4 it was only shown as a vague cloud! In this chapter we are going to study aspects of interconnection networks in some more detail.

To tie modules in a multiprocessor together an interconnection network is needed. The interconnection network is a very important part of a multiprocessor. It consists of wires and switches. Wires are relatively cheap, in contrast to switches.

Wires: The most important property of wires is its delay.

The delay of wires depends on its length l. For "short" wires the delay t_{short} is dominated by the charging of the capacitance of the wire:

$$t_{short}(l) = \tau_{invert} e \log_e Kl \tag{3.1}$$

where K is some constant and τ_{invert} is the unit time to switch the state of the wire [Dally90]. Thus for short wires the delay increases logarithmically with the length. For longer wires, transmission line effects dominate [Dally87], thus the delay is limited by the speed of light c:

$$t_{long}(l) = \frac{l\sqrt{\varepsilon_r}}{c} \tag{3.2}$$

and increases linearly with wire length. The crossover between a capacitive wire (short) and transmission line (long) wire are dependent on technology. An analysis is given in [Dally87] [1]: with a 0.5μ technology the crossover is about 10mm, which is equivalent to the length of a chip. Thus wire-delay between switches in an interconnection network increases linearly with the length of the wire, as in formula 3.2.

Switches: It is important to design the switches carefully with respect to the following issues:

- What should the size of the buffers in the switches be? Should they be able to contain a complete packet? The choice might put constraints on the packet size.

[1] On page 141.

- How many buffers should the switches have ?
- It is desirable that the time to transfer packets from an input channel to an output channel be as short as possible.
- The routing algorithm should ideally be simple and fast.
- How many channels and how many pins should the switch have? This issue is related to the number of switches in the network. If the switches has "many" input and output channels ("many" dimensions) then the pinout on the chip grows, but the number of switches in the network decreases (holding the number of nodes constant). Simply put: large, expensive switches imply fewer switches (we're here presuming the switch would be put on a single chip, that is not necessarily the case).

There are four main characteristics of interconnection networks [HwaBri, Feng]: the mode of operation, the strategy for controlling the routing, the form of switching, and the nature of the topology.

The mode of operation: There are three major methods of establishing communication paths: *synchronous*, *asynchronous* and *source-synchronous* communication.

With synchronous communication the system has a common global clock. It is therefore clearly defined what constitutes "data" on the links in the network. All time is divided into discrete intervals, e.g. bus cycles. The drawback is that time is sometimes wasted while waiting for the start of a new interval.

In asynchronous mode time is not divided in such a manner. Instead "data" must somehow be encoded, or protocols must be used, e.g. "handshaking" in asynchronous buses. Asynchronous communication is used when it is desirable to issue communication requests dynamically.

A third method is to have a source-synchronous sender. The method is interesting mostly when using point-to-point links. The sender-part of a node determines the clock on the links between two nodes. SCI (see chapter 5), and also parts of Futurebus+ (see appendix A) uses this scheme.

Control of routing: A typical network contains switches and links. These switches can have various settings, depending on the control strategy. This control strategy can be done centrally, or by the individual switching elements. The first is called *centralized* control, the latter *distributed* control.

Type of switching: There are two major switching methods: *circuit switching* and *packet switching*.

In circuit switching an actual physical path is established between a sender and a receiver. Generally circuit switching is more suited to transmit large bulks of data.

In packet switching data is encapsulated in packets, containing the data and some information necessary for the transportation of the data.

The packets are routed through the interconnection network without establishing a physical path. Packet switching is more suited to smaller bursts of data. Circuit switching reflects how the telephone system works while packet switching reflects how computers work. In section 3.3 we divide packet switching into 3 switching techniques: store & forward, wormhole and virtual cut-through.

The network topology: Interconnection networks can be divided into two rough categories: *static* networks and *dynamic* networks.

A static topology implies a network with unique paths (routing) between any 2 given nodes, like a ring or a star network. A dynamic topology routes packets through a network of switches. The routing is not fixed; packets between 2 given nodes may take various paths. This is discussed further in section 3.1.1 and 3.1.2.

When discussing interconnection networks further we will concentrate on the topology of the network.

3.1 Static and dynamic topologies

3.1.1 Static

In a static network the links between the modules are passive and cannot be reconfigured in any way. Typical examples are: a linear array (typically a bus), a ring, star, tree, mesh, a completely connected network, a cube, or a cube-connected cycle.

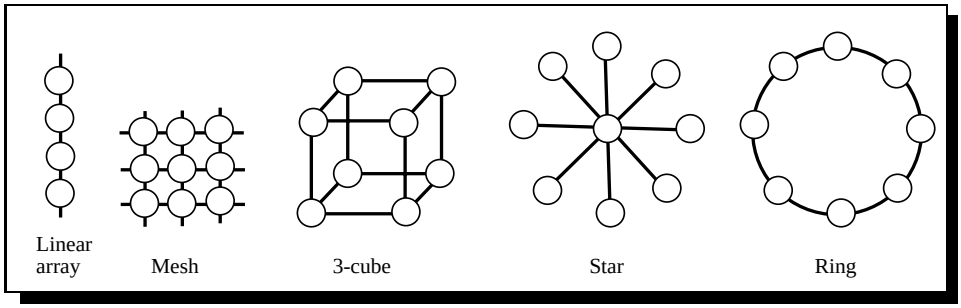


Figure 3.1: A selection of static topologies. Adapted from [Feng]. Note that the circles represent modules (processors and memories), and not switches. The lines represent links.

3.1.1.1 The Mesh

An important and popular interconnection network is the mesh. The mesh is a structure of nodes spread out in 2 dimensions. A classical example is the interconnection network of the Illiac-IV array processor [Siegel]. It consists of $N = 64$ processors connected together in a 8x8

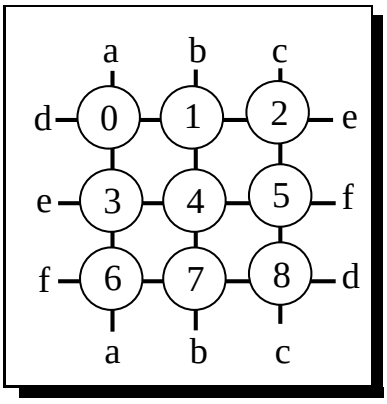


Figure 3.2: The mesh connection of the Illiac-IV, somewhat simplified. Identical letters are connected together. Adapted from [Siegel].

matrix. For simplicity a reduced 3x3 version is shown in figure 3.2. Each processor is allowed to communicate directly with any of its 4 neighbors. This is formally characterized by the following routing equations:

$$\begin{array}{lrcl} \text{right neighbor:} & \mathit{Illiac}_{+1}(P) & = & (P+1) \bmod N \\ \text{left neighbor:} & \mathit{Illiac}_{-1}(P) & = & (P-1) \bmod N \\ \text{below neighbor:} & \mathit{Illiac}_{+r}(P) & = & (P+r) \bmod N \\ \text{above neighbor:} & \mathit{Illiac}_{-r}(P) & = & (P-r) \bmod N \end{array} \tag{3.3}$$

where $r = \sqrt{N}$ and P is the address of the processor wishing to send. The processors are interrupted in their work when they receive a packet which must be routed on to another processor (static topology). It is therefore best suited to problems where the processors only communicate with their nearest neighbors.

3.1.1.2 The nCube

The cube is also a well-described interconnection network [Siegel,Feng]. It is a cube of N nodes stretching into $n = \log_2 N$ dimensions. Each processor is connected to n other processors. To address a neighbor in the i'th dimension the i'th address-bit is complemented. The n routing functions is formally described by the following equation:

$$\mathit{Cube}_i(P_{n-1} \cdots P_i \cdots P_0) = P_{n-1} \cdots \overline{P_i} \cdots P_0 \tag{3.4}$$

where $i =< 0, n-1 >$ and $P_{n-1} \cdots P_i \cdots P_0$ is the n address bits of the processor addresses.

For $n = 3$, shown in figure 3.1, this results in the following 3 routing functions:

$$\begin{array}{ll} \text{“dimension 0 - neighbor”:} & \mathit{Cube}_0(P_2P_1P_0) = P_2P_1\overline{P_0} \\ \text{“dimension 1 - neighbor”:} & \mathit{Cube}_1(P_2P_1P_0) = P_2\overline{P_1}P_0 \\ \text{“dimension 2 - neighbor”:} & \mathit{Cube}_2(P_2P_1P_0) = \overline{P_2}P_1P_0 \end{array} \tag{3.5}$$

This static cube topology can also be mapped onto a dynamic multistage network. [Siegel] proposes such a "multistage cube/shuffle-exchange" network, where dimension i of the $n$cube is mapped to stage i of a multistage network of n stages. The switches perform the exchange function (see equation 3.7) and the connection pattern between the switches use the shuffle function (see equation 3.6). Figure 3.5 shows a general multistage network.

Like the mesh, the $n$cube is better suited to problems where the processors only communicate with their nearest neighbors.

Please note that when $n = 2$ the $n$cube is equivalent to $2 * 2$ mesh.

3.1.2 Dynamic

Unlike static networks, dynamic networks contain switching elements. They can reconfigure the links in the network. Thus there are alternative routes between two given modules. This is a desirable property in case a part of the network becomes faulty for some reason.

[Feng] divides dynamic networks into 3 groups: *crossbar* networks, *single-stage* networks and *multi-stage* networks.

Crossbar: m memory-modules are connected to p processors through an interconnection network containing $m * p$ switches. $Processor_p$ is connected to $memory\text{-}module_m$ through $switch_{pm}$ The advantage with a crossbar network is that all processors has then a path to a memory-module whenever it so desires (non-blocking). The drawback is that this is achieved at a great cost in the number of switches.

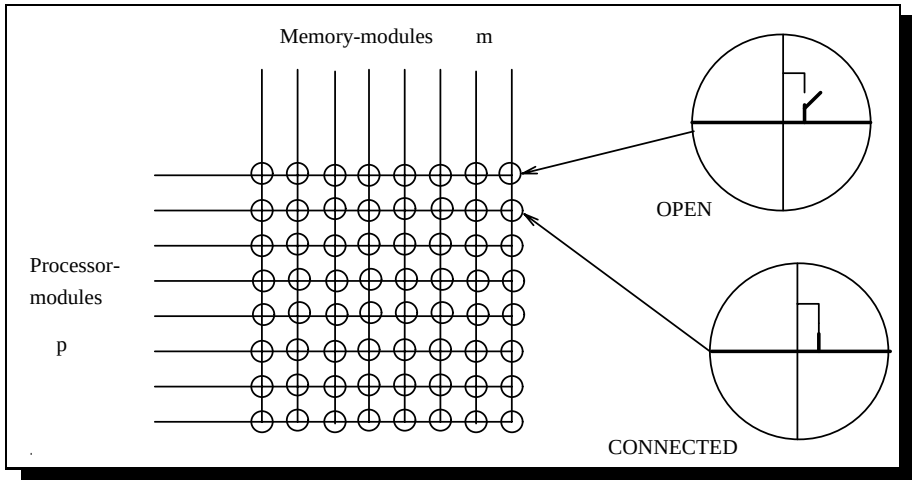


Figure 3.3: An example of a crossbar network. At each crosspoint a switch is placed. These switches are in one of two states: either it connects a horizontal line with a vertical one (CONNECTED) or there is no connection between the two (OPEN).

Singlestage: A single-stage network is made up of one stage of switches cascaded to a connection pattern of links. An example is the

shuffle-exchange network shown in figure 3.4. Generally, the shuffle routing function is performed by the cyclic shuffling of the address bits $a_{n-1} \cdots a_1 a_0$ in the following way:

$$\textit{shuffle}(a_{n-1} \cdots a_1 a_0) = a_{n-2} \cdots a_1 a_0 a_{n-1} \tag{3.6}$$

The exchange routing function the switches may perform is:

$$\textit{exchange}(a_{n-1} \cdots a_1 a_0) = a_{n-1} \cdots a_1 \overline{a_0} \tag{3.7}$$

Complementing the least significant bit means that modules with adjacent addresses exchange data. In figure 3.4 The interconnection pattern performs the function:

$$\textit{shuffle}(a_2 a_1 a_0) = a_1 a_0 a_2 \tag{3.8}$$

The switches can perform the exchange function:

$$\textit{exchange}(a_2 a_1 a_0) = a_2 a_1 \overline{a_0} \tag{3.9}$$

depending on the control-signal to the switches.

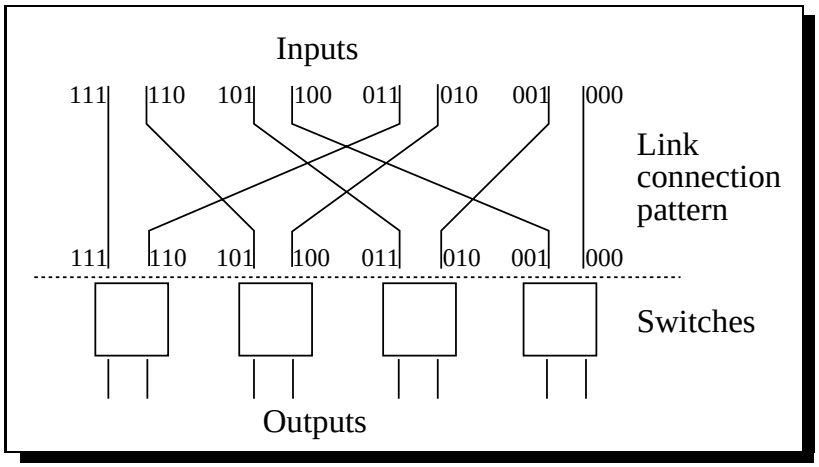


Figure 3.4: Example of a singlestage network: a shuffle-exchange network. Adapted from [Feng].

Singlestage networks are also called a recirculating network because data items may have to circulate through the network multiple times before reaching a destination. Conceive that the outputs of the switches are connected back into the network, so that the network has a cylindrical form.

Multistage: A multistage network (called by some an indirect network, and by others yet, a butterfly network) consists generally of n stages with $N = k^n$ input and output lines. Each stage is constructed of $k * k$ crossbar switches. k is usually 2. The delay in the network is proportional to number of stages. The network forces all requests through

all n stages to its destination. Thus multistage networks cannot take advantage of locality.

An important characteristic is the interconnection-pattern between the stages, which could for example be the shuffle function. How are the switches controlled? With common stage control, the same control signal sets all the switches in the same stage. Thus all switches in the same stage are set to the same state. An alternative is to set the switches individually. This requires $n^2/2$ control signals.

Well-known examples in the literature of multistage networks include Clos network [Feng], Benes network [Feng] and the Omega network [HwaBri] (the latter uses the shuffle-exchange interconnection pattern).

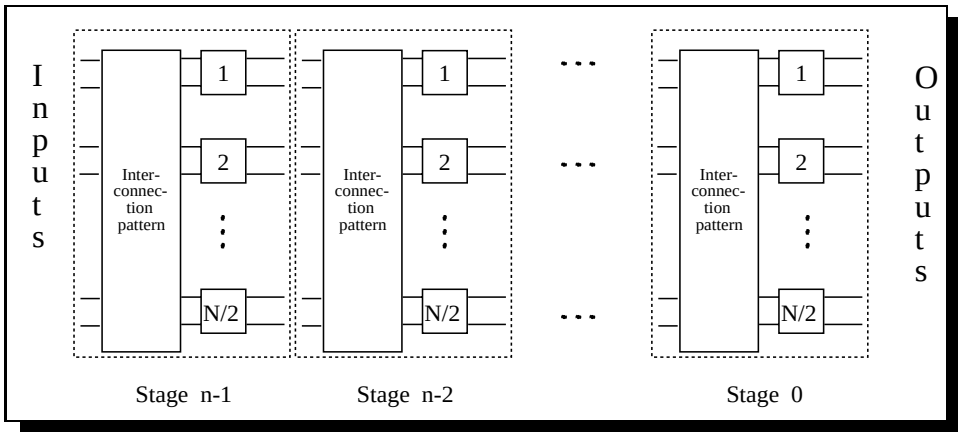


Figure 3.5: General multistage network containing n stages used to connect together 2^n inputs and 2^n outputs. k is here 2.

Multistage networks are often classified as a *blocking* or *non-blocking* network. In blocking networks conflicts often arise over the simultaneous use of network communication links. An interconnection network capable of handling all possible connections without blocking is a non-blocking network. Clos network [Feng] is an example of a non-blocking network.

Please note: When Feng and others refer to e.g. a 3-cube as being static, they conceive that the vertices in the cube as containing processors. If the vertices in a cube instead only contain switches, then obviously the 3-cube is not at all static. It is then best described as a dynamic network. We find this point to be ignored in the literature.

As time goes on the distinction between static and dynamic topologies may fade. The difference is really only where the logic doing the routing is placed.

3.2 Deadlock

An important problem with networks is the possibility that it may block. It is essential that packets arrive at their destination in a reasonable amount of time and that a packet does not block parts, or all, of the network. Deadlock is a property of the routing algorithm and the topology of the network. An example of deadlock is shown in figure 3.6. If the four nodes attempt to send a packet to the opposite corner at the same time, and the routing algorithm routes the packet in a clockwise direction, a deadlock situation arises. Here a packet buffer in each node would remove the deadlock.

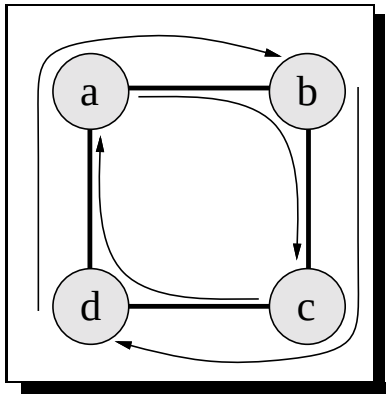


Figure 3.6: Example of a deadlock. The arrows reflect the situation that all nodes wishes to send to the "opposite" node. 'a' wish to send to 'c', 'b' to 'd', and so on. Adapted from [ShMayTho].

3.3 Switching techniques

One of the basic problems when designing an interconnection network is to choose a switching method. A selection of methods are: circuit switching, store & forward switching, wormhole routing, and virtual cut-through.

Circuit switching originates from the time computer networks were based on the existing telephone network. With circuit switching a complete path of communication links must be established before two nodes can communicate. Once a path has been set up, no more addressing information is necessary. The path implicitly provides the addressing information. To set up this path some signal must first be sent (shaded box in figure 3.7a). The path set up remains during the entire session. Since data is often sent in bursts it is not ideal for interconnection networks. The path is similarly closed by a signal when the session is over.

Store & forward switching In store & forward switching a packet is buffered in an intermediate node before it is passed on to the next node. See figure 3.7b. Thus the packet is effectively removed from the network while the node determines where it should send the packet (when it can). This buffering effectively limits the size of the packets. Each node must be able to store one or more packets.

Wormhole routing This switching technique is called wormhole routing, despite the fact that strictly speaking it is not a routing algorithm.

In wormhole routing the first flits of a packet are transmitted before the complete packet has been received. An intermediate node does not have buffer-space for a complete packet. Thus, if the header (essentially the target address) can be kept short, the delay through an intermediate node can be minimalized. See figure 3.7c.

Virtual cut-through is a hybrid of store & forward switching and wormhole routing. If the desired output channel of a node is busy the packet is buffered (see figure 3.7d). Otherwise the node sends the packet and does not buffer it (see figure 3.7e). Like store & forward, virtual cut-through puts restrictions on the size of the packets. In figure 3.7f the desired output channel from the sender is vacant so the packet is sent. The intermediate nodes' output channel is not vacant, so there the packet is buffered until it becomes available [KerKlein].

3.4 Relationship with software

So far much has been said about the hardware properties of parallel processing. But what about the relationship between the hardware and the programs it is expected to execute? In figure 3.8 an attempt is made to show the relation between the interconnection network and a program. On the highest level of abstraction is a user program. Below a compiler or interpreter partitions the program into multiple processes. These multiple processes are assigned to various processors through the network. Communication between the processes is carried out by the network.

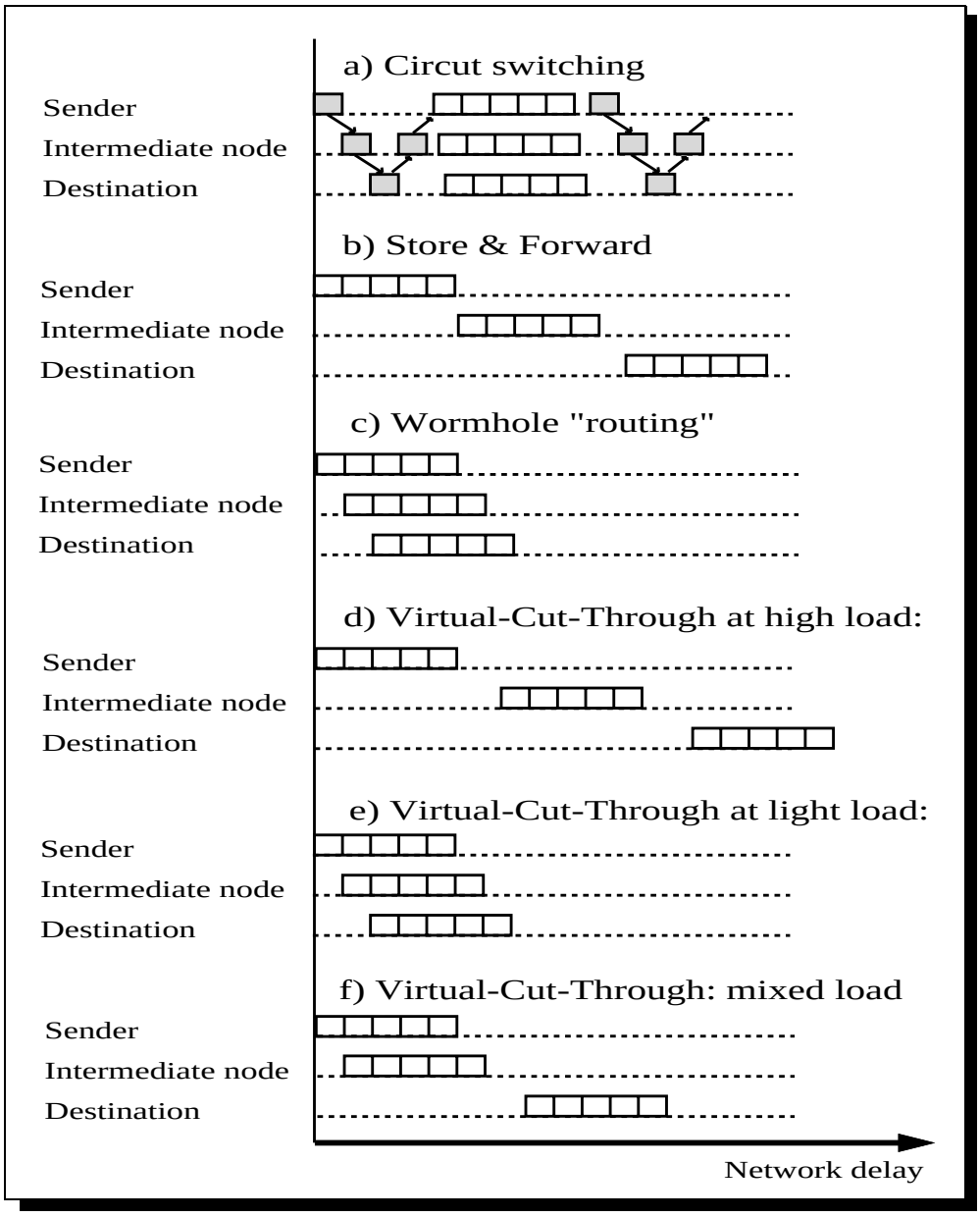


Figure 3.7: Various switching techniques. Shown is an example packet of 5 flits. Adapted from [Dally90] and [KerKlein].

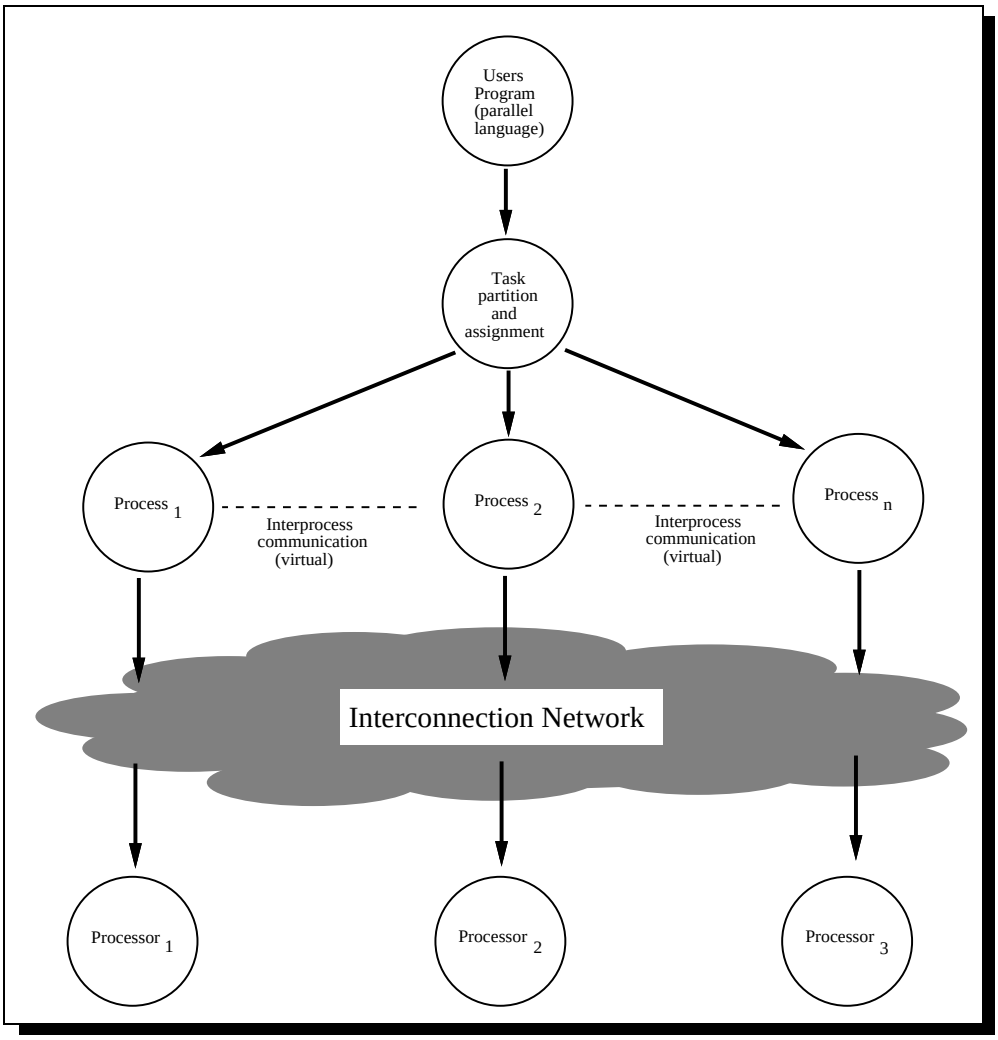


Figure 3.8: Relationship between an interconnection network and an executing program. Partly adapted from [Feng].

4

The k-ary n-cube interconnection network

As mentioned previously the goal of this thesis is to study various interconnection networks for large SCI-systems. We have considered several kinds of topologies:

- A single ring. The advantage of using just a single ring is its simplicity. It is easy to connect. Practically no routing is needed. The main drawback is the limitations in bandwidth and latency as the size of the ring grows. It is in practice useless for a large system, as shown in chapter 9 and [HulBot].

- Crossbar-networks. Their advantage is that they are non-blocking. The drawback is the large number of switches as the network-size increases. It is thus too costly and not scalable.

- Singlestage/multistage. Possible to scale, but the latency grows with the increase of the number of stages. This in itself is not so bad, but the distance between nodes in the network is constant irrespective of which two nodes are communicating (when no contention). Thus no form of optimalization (locality) is possible, if desired. Another drawback is that some multistage networks have the possibility of deadlock, while others are difficult to prove deadlock-free [ScottGood].

- k-ary n-cubes. They are regular, independently of size. They are thus scalable [1]. It is easy to vary the size by changing the parameters (k and n) and thus see how the performance is affected by the change of the parameters. The routing algorithm can be made simple. See section 8.5. If desired fault-tolerant network can also be implemented. k-ary n-cubes can be made deadlock-free, as proved in [ScottGood].

[1] At least in theory. The scalability-issue is of course implementation-dependent.

Of the interconnection described in the literature we have chosen to use the k-ary n-cubes as the topology to investigate in this thesis. The reasons mentioned above are the most important for the choice. The rest of this chapter is an in-depth presentation of k-ary n-cubes.

The k-ary n-cubes topology has been used successfully in machines such as the Connection Machine and the Cosmic Cube [Seitz]. It is constructed of cubes with dimension n, and k vertices in each dimension. The relation between dimension (n), radix (k) and total number of vertices (N) are as follows[2]:

$$N = k^n, \qquad (k = \sqrt[n]{N}, n = \log_k N) \tag{4.1}$$

To address these N vertices, n fields of length $\log_2 k$ (rounded up to nearest integer) bits are needed, giving a maximum of $\log_2 N$ address bits.

Most parallel computer topologies are of the k-ary n-cubes -category, or can in some way be mapped to it. Examples of k-ary n-cubes are meshes, cubes and omega-networks [Feng]. Networks that can be mapped to it are rings, trees, multistage, butterfly, and others.

The maximum number of links between 2 nodes farthest away from each other is often called the *diameter* of the network. The diameter of a k-ary n-cube is:

$$D = n(k-1) \tag{4.2}$$

In figure 4.1 we show some of the smaller, less complex variations of k-ary n-cubes. Each point in the cubes are switching nodes. Some special cases of interest:

- When $n = 1$ the cube is a linear array.
- When $k = 2$ the cube is better known as a *hypercube* (called by some a binary n-cube).
- When $n = 2$ the cube is strictly not a cube at all, but a mesh.

A set of links in a dimension can either be unidirectional or bidirectional. In addition a set can be torus-connected at the "sides". Thus there are four possibilities, as shown in figure 4.2. (Note that this is a characteristic of networks in general, and not specially for k-ary n-cubes.)

The vertex in a k-ary n-cube is connected to n other vertices through $2n$ links (inputs and outputs). A torus-connected unidirectional k-ary n-cube is then assumed.

From now on, unless otherwise stated, torus-connected unidirectional k-ary n-cubes are assumed.

How does k-ary n-cubes fit into the scope of the previous chapter? Is a k-ary n-cube a static or a dynamic network? k-ary n-cubes are certainly dynamic in the sense that the vertices contain active elements (switches)

[2] This is the most important equation in this thesis. Please make note of it ☺

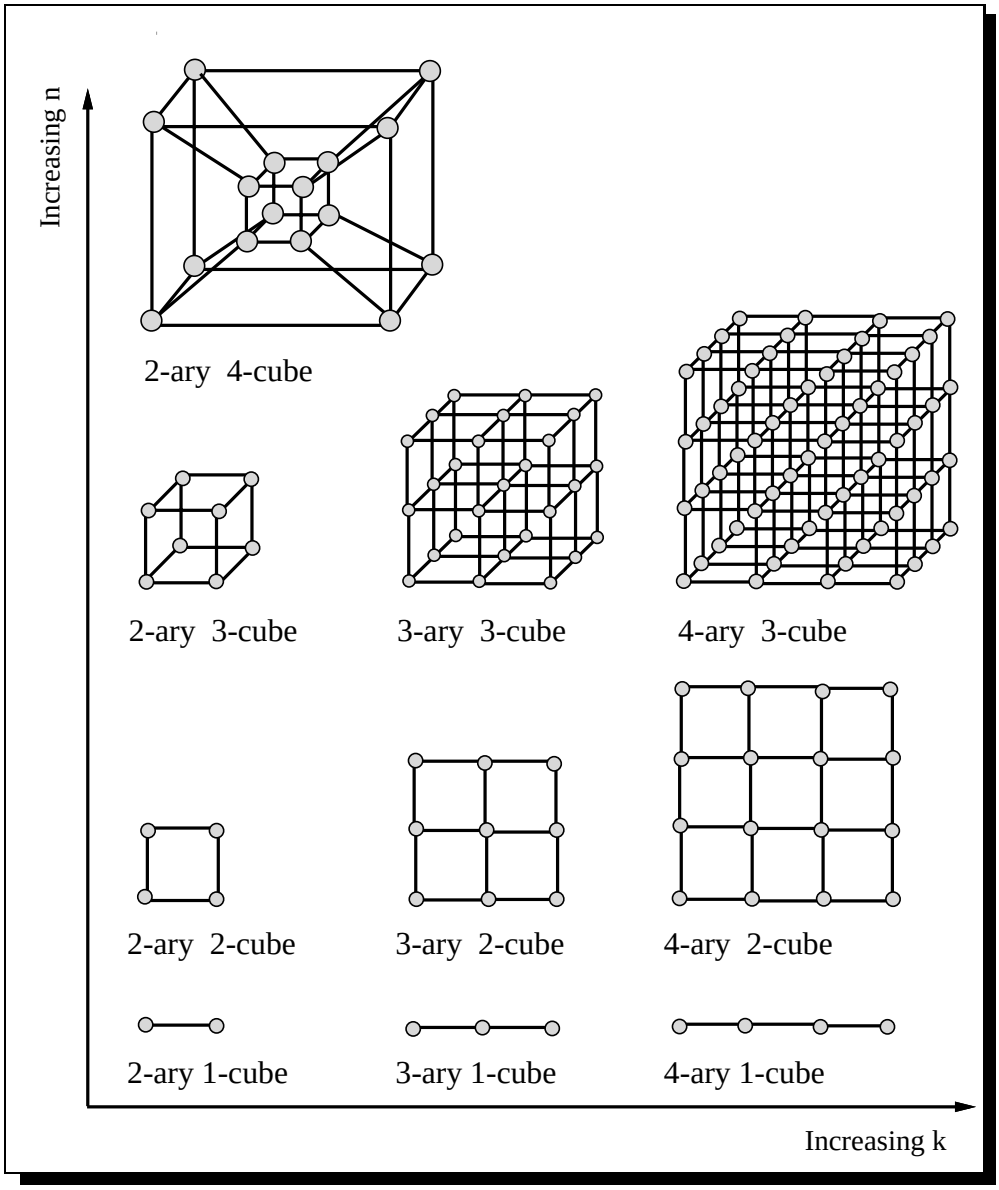


Figure 4.1: A small selection of k-ary n-cubes. As k, and especially n grows, it becomes too complicated to draw.

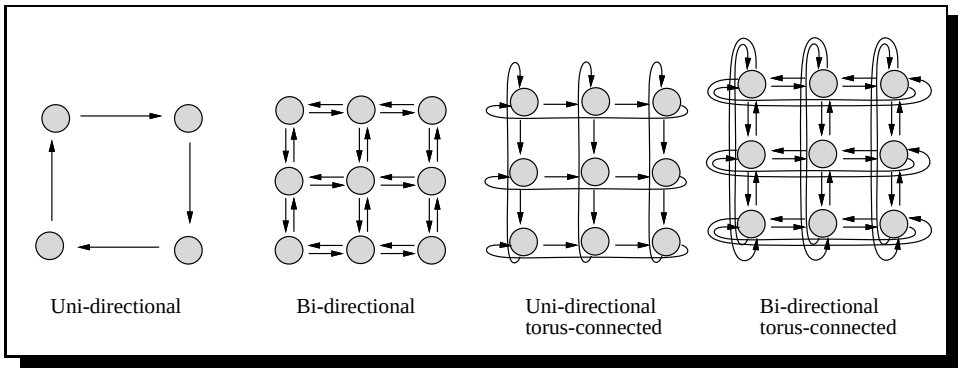


Figure 4.2: Different connection schemes in point-to-point networks. The figure shows various ways of implementing the links in a k-ary n-cube. The examples shown here are for 2-ary– and 3-ary– 2-cubes. Note that some of these methods are very redundant for so small examples.

capable of routing the packet in various directions (dimensions). But within the scope of a single dimension k-ary n-cubes could be viewed as static. One dimension would probably be implemented as a ring or a linear array.

4.1 Properties of k-ary n-cubes

One of the important properties of k-ary n-cubes is the the unloaded latency, which is also possible to discuss without a simulation or complicated mathematical modell. This presented in depth in section 4.1.2. We will also say some words about wire delay (section 4.1.1) and throughput (section 4.1.3).

4.1.1 Wire delays

The length of wires is an important parameter in k-ary n-cubes, as in all networks. In [Agarwal] it is suggested that advances in technology will make ever-faster switches possible. Wire speeds will remain roughly constant, and will thus dominate switch delays.

The length of the wires is not necessarily equal for all the links [ScottGood]. The network must obviously physically be constructed in 3 dimensions. If the network contains more than 3 logical dimensions, modules must be placed somehow unevenly in three physical dimensions. Each physical dimension must then contain $n/3$ logical dimensions. Thus the delay of the longest wire is:

$$T_{wire\_max} = k^{\frac{n}{3}-1} \cdot T_{wire}(n=3) \qquad n \geq 3 \qquad (4.3)$$

where $T_{wire}(n=3)$ is the delay of the wire in a uniform network with 3 dimensions.

In a synchronous network the clock must accommodate this longest link. Thus the wire delay is determined by the *maximum* wire length. In a pipelined, source-synchronous network, like SCI, the mean wire delay is determined by the *mean* wire length. In [ScottGood] the mean wire delay, T_{wire_ave}, is found to be approximately equal to the mean wire length, or:

$$T_{wire\_ave} \approx \frac{k^{n/3} - 1}{k - 1} \cdot \frac{3}{n} \qquad n \geq 3 \tag{4.4}$$

Folding If a torus-connection is used, the long end-around link will be much longer (see figure 4.2c). In [Dally90] a *folded* torus-connection is proposed to reduce the length of the longest link. Then the longest wire is approximately the mean wire-length. This reduces the maximum wire length. Thus the clock frequency can be increased if using synchronous clocking. See figure 4.3.

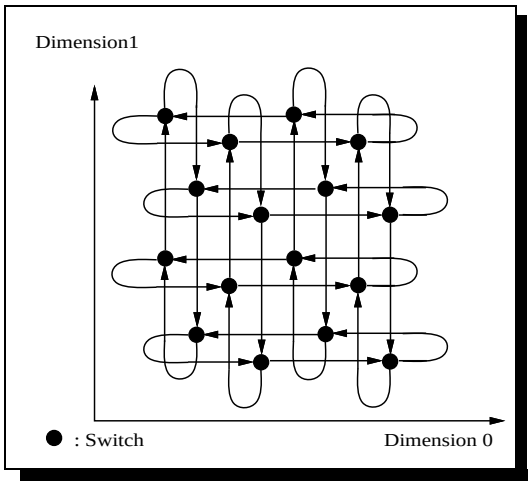


Figure 4.3: Folding a 4-ary 2-cube to shorten and even the wire lengths. Adapted from [Dally90].

4.1.2 Unloaded latency

One of the most important metric for an interconnection network is the transmission time for a packet through the network, when the network is not loaded. This is often referred to as the unloaded latency. Let us assume the following:

- A packet has L bits.
- Channels are W bits wide.
- A packet contains L/W flits.

- If two random switching nodes are selected, then there is on average D_{ave} number of hops between them where:

$$D_{ave} = n\frac{k-1}{2} \tag{4.5}$$

  (equation 4.2 divided by 2)

- The k-ary n-cube has unidirectional channels and end-around torus connections.

- The switching technique is "wormhole routing" or "virtual cut-through" [KerKlein].

- The bisection width of a network is the minimum number of wires cut when the network is divided into two equal halves [Dally90]. The bisection width of a bidirectional k-ary n-cubes is:

$$B = \frac{2WN}{k} = 2Wk^{n-1} \tag{4.6}$$

We will here in this section discuss the unloaded latency as presented in [Dally90,Agarwal,ScottGood]. [Dally90,Agarwal] both assume the use of synchronous clocking, and thus the latency is dependent on the *maximum* wire length. [ScottGood] assumes the use of pipelined (source-synchronous, see chapter 3) networks, dependent on the *mean* wire length. [ScottGood] also goes more into detail with respect to the speed of switches.

Ignoring the switching delay, [Dally90] defines the latency as the time on wires plus time to send a packet L/W flits long:

$$Latency_{[Dally90]} = T_{cycle}(D_{ave} + L/W) \tag{4.7}$$

where T_{cycle} is the cycle time on the channel. In [Dally90] k-ary n-cubes are studied under the constant bisection width constraint. The conclusion was that 2-dimensional and 3-dimensional networks were optimal as system size N increased, instead favoring an increase in the value of the radix k. This conclusion was motivated by the 2-dimensional nature of VLSI and of the use of synchronous clocking.

In [Agarwal] the unloaded latency is the product of the time through one switching node, the sum of the nodes passed and the message length. [Agarwal] includes the switching delay T_{switch} through the switching nodes:

$$Latency_{[Agarwal]} = (T_{wire} + T_{switch})(D_{ave} + L/W) \tag{4.8}$$

[Agarwal] studied k-ary n-cubes under the constant bisection width and under the constant wire density. The conclusion was that moderate high dimension is optimal.

Both in [Dally90] and in [Agarwal] it is assumed that synchronous clocking is employed. Thus T_{cycle} in equation 4.7 and T_{wire} in equation 4.8 is T_{wire_max} (equation 4.3) and thus dependent on n.

In [ScottGood] it is assumed that bits are pipelined: source-synchronous clocking is used [3] and the wire delay is as in equation 4.4. Also, [ScottGood] makes a difference between the time for the message to switch dimensions, T_{in_switch}, and the time to continue along the the same dimension, T_{pass}. It is assumed that the switching node uses T_{decode} cycles to decode the message-address:

$$T_{decode} = \frac{\log_2 A}{W} \tag{4.9}$$

where A is the number of address-bits, placed in the first flit of the packet.

A message travels a mean of $k/2$ hops in one dimension. It continues in the present dimension with a probability of $(k-2)/2$ (-2 because of the local sender and the local receiver of the packet). It enters the n dimensions with a probability of $\frac{k-1}{k}$ The unloaded latency, in cycles, is then:

$$\begin{aligned} Latency_{[ScottGood]} = \\ & n\tfrac{k-1}{k}\left[\tfrac{k}{2}(T_{wire\_ave} + T_{decode}) + \tfrac{k-2}{2}T_{pass} + T_{in\_switch}\right] \\ & + L/W - 1 \end{aligned} \tag{4.10}$$

In [ScottGood] pipelined and synchronous k-ary n-cubes are studied under the constant link width W constraint, the constant node size $2nW$ constraint, and the constant bisection width B constraint. The conclusion of [ScottGood] is that in all 3 cases, pipelining argues for higher dimensionality as system size increases, while the radix k should be kept constant.

4.1.3 Throughput

Throughput is the amount of data the network can handle per unit of time.

A network must be expected to carry traffic. Latency must be studied together with the throughput X of the net. To do so requires either a queueing model or simulation. The former is beyond the scope of this thesis, though it is basis of the discussion in [Dally90, Agarwal, ScottGood]. The latter is discussed in chapter 8 and 9.

A definition of the maximum throughput for a k-ary n-cube , sometimes also called the network capacity, is the following (adapted from [Dally90]):

$$X_{max} = \frac{X_v \cdot N}{D_{ave}}$$

X_v is the maximum bandwidth out of a vertex. It is the number of outgoing links (n) multiplied by the link-bandwidth (X_l).

The maximum throughput for a k-ary n-cube is thus:

$$X_{max} = \frac{X_v \cdot N}{D_{ave}} = \frac{nX_lN}{D_{ave}} = 2X_l\frac{N}{k-1} \tag{4.11}$$

[3] The analysis in [ScottGood] has been made with SCI in mind.

If X_l is in bytes per second, X_{max} will be likewise. Note that equation 4.11 is for the *complete* k-ary n-cube.

4.1.4 Reducing the maximum distance in the k-ary n-cube

The maximum distance in a k-ary n-cube is $n(k-1)$. Several schemes have been proposed to reduce the distance. In [Dally91, ElAmSha] the use of additional links is discussed. In [ElAmSha] it is proposed that each switching node has $n+1$ outgoing links, the last link being connected to the most distant node. The maximum distance is thus halved.

5

The Scalable Coherent Interface

In the late 1980's several people working on bus standardization[1], initiated a study on how to increase the performance of buses. They soon realized that no bus could satisfy the needs of the next generation of multiprocessors. Satisfying these needs would demand connecting together a large number of processors, providing a speed of communication to live up to the rapidly increasing speeds of the processors. The interconnect should scale well with increasing size of the topology, as well as offering a coherent memory system.

Buses don't fit into such a scheme due to their inherent behavior. They allow only one transmission at any time. The clock rate on a bus is limited by the physics of transmission lines with a variable load. Scaling is largely restricted by the propagation delays involved with handshaking and the general arbitration on the bus. This adds up to a system not very applicable to supply the necessary communication needed in a large multiprocessor system.

So, if the performance of computer communication was to increase more than just marginally, a different scheme had to be chosen. The conclusion was to use packet-based communication over independent point-to-point links. This solution implies a reduction in the problem of non-ideal transmission lines. There will never be any problem with contention on the transmission medium.

The loss of a bus generates a new problem, though: It is no longer possible to use snooping (see section 2.3.4.1) to maintain a coherent memory system. Instead a directory-based cache coherence protocol was designed (see section 2.3.4.2).

The project was adopted within IEEE and named the "Scalable Coherent Interface" (SCI). A working group within IEEE has worked for 2.5 years defining all aspects of the now approved standard: IEEE std. 1596–1992.

In the following sections a closer overview of the different aspects of SCI is presented.

[1] See appendix A for an overview of bus-standards.

The Goal of SCI The goal of the SCI-project was to define some interconnection scheme connecting together processors and memories in a multiprocessor system. The following section describes the most important properties which were taken into consideration during the development:

To get the use of SCI more widespread, the interconnect should be able to support a high performance multi-computer system, at a low cost. It should be possible to build a high-end multi-processor at a reasonable price, as well as making it cost effective to build small systems.

To achieve this goal the protocols had to be fairly simple. The pin count on the chips had to be as small as possible, allowing connection of processor, memory, I/O and bus adaptor cards from multiple vendors. On the other hand the high-performance and the scalability behavior should not be sacrificed.

Each link connection should offer a speed of 1 GigaBytes/sec. The addressing scheme should be able to address as much as 64k nodes. And to provide a more user friendly system the cache-coherence problem should be solved.

5.1 SCI basics

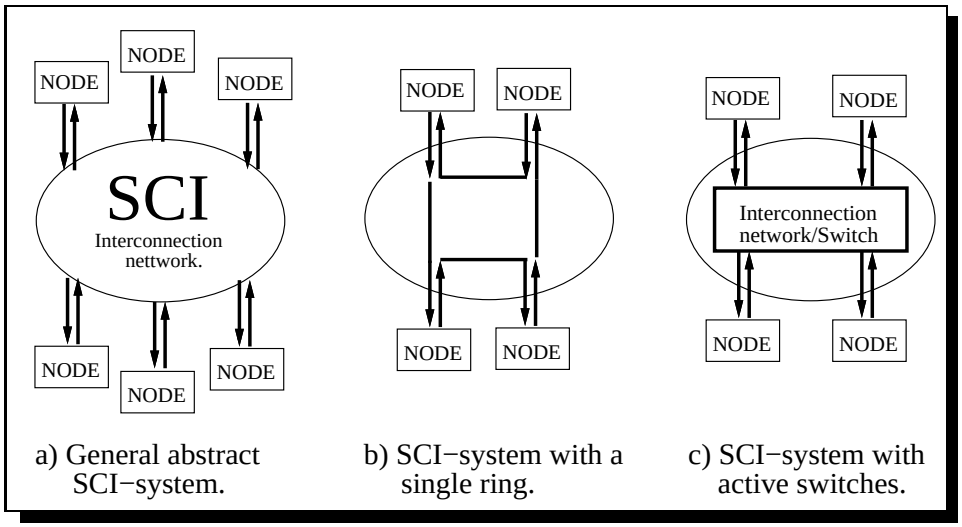


Figure 5.1: SCI system topology in general.

This is meant to be a brief introduction to SCI. For a full and more detailed description see the SCI standard [2].

In a computer system using SCI all communication between nodes in the system uses uni-directional point-to-point links. A general system topology using SCI is shown in figure 5.1. This figure shows that every node has one input and one output link. The links consist of 18 parallel

[2] Most of the figures in this chapter are adapted from [IEEE-SCI].

lines. Of these lines, 16 are used for data and the remaining 2 for control purposes[3]. A view of the node including the links is shown in figure 5.2.

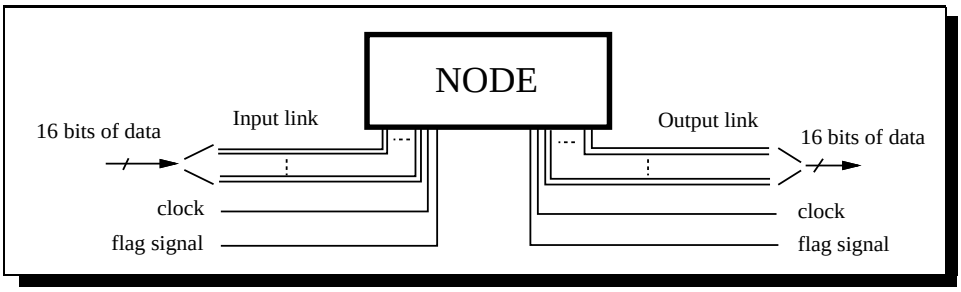


Figure 5.2: Symbols on the link.

On these links nodes transmit/receive the *symbols* (flits). The symbols are the smallest entity of information in SCI, they consist of one word (16 bits, omitting the control lines) transmitted in parallel on the links.

Some of the symbols are used to form packets [4] and others are used to carry information used by the protocol to control arbitration, synchronization, etc. Packets are formed by consecutive symbols. Between each packet there has to be at least one *idle*-symbol. The idle symbols carry information concerning the bandwidth allocation. It is used by the nodes to control the flow on the output link. This means that the node is dependent of the incoming symbols to control the behavior on the output link. In other words the arbitration protocol depends on the circulation of idle-symbols in the system. The bandwidth allocation protocol implements a concept of partial fairness using priorities and a round-robin protocol. See section 5.3.5.5 and 5.3.6 for an explanation on idle-symbols.

The dependency of incoming idles for a correct flow behavior is the reason why the basic configuration of a SCI system always has to be a ring. The small rings of which the whole system are build up of is often called *ringlets*. On every ringlet there is one node with the dedicated task of performing the ring maintenance (initialization, time-out etc). This node is called the *scrubber*. Other configurations of topologies using SCI-system are not defined. This work is left to the implementor depending on the task which the system is intended to solve. See figure 5.1.

Each node in a SCI based system will be a combination of processors, SCI-memories, I/O, bridges and switches. Based on the contents of a node it will have either requester or responder capabilities. Nodes with active elements (e.g. processors and bridges) will typically have request capabilities. They request (demand) some resource from a node with

[3] A serial link is also defined [See section 5.2], but every link reference in this thesis is to the parallel link.

[4] Examples in section 5.3.5

responder capability. The respond capable node (memory, I/O etc) then provides the resource.

The cache-coherence problem is solved using a distributed-directory concept. This is described in section 5.4 and section 2.3.4.2.

The throughput and latency are the most common measure on the performance of computer-system. Thus a short discussion on theoretical throughput and latency on a ring, is presented. The discussion below covers rings only, since this is the basic building element in any topology using SCI. See also section 9.1 and [HulBot] for our simulation results on single rings.

Throughput A node theoretically outputs g Gigabytes per second. A ring with x nodes should theoretically result in throughput of $x * g$ Gigabytes per second, but only if packets are addressed to the senders downstream neighbor. This is not so realistic. Let us instead assume that packets are addressed randomly. Then the traffic is roughly symmetric. On average, packets will traverse half-way around the ring. Then the average number of segments traversed by each packet is half the total number of segments ($\frac{x}{2}$):

$$\text{Ring bandwidth} = g \cdot \frac{x}{\frac{x}{2}} = 2 \cdot g \tag{5.1}$$

Thus the ring bandwidth is independent of the ring-size.

Some of this bandwidth is used by echo-packets (containing $e = 4$ symbols) and packet-headers (containing h symbols). Echo packets is described in section 5.3.4. In addition, there is at least one idle-symbol between packets . If we assume "move64"-packets (section 5.3.5.1) of 40 symbols, with $h = 8$ we must multiply the previous equation with:

$$\text{Fraction of real data} = \frac{40 - h}{40 + e + 1} = \frac{32}{45} \approx 71\% \tag{5.2}$$

If larger packets are used, this fraction will improve.

g is 1 Gigabytes per second. Thus the maximum effective throughput of an SCI-ring (independently of size) is:

$$\text{Effective throughput} = \frac{32}{45} \cdot 2 \cdot g \approx 1.4 \quad \textit{G bytes/second} \tag{5.3}$$

Latency It is difficult to say much off-hand about latency under loaded conditions. That requires a thorough analysis (queueing theory) or simulation (see chapter 9). Here we will only present the average latency on an unloaded ring.

The same argument as for throughput goes for latency. Thus we assume a random distribution of addresses, each packet travels half way around a ring on average. This makes the latency the sum of the elements it passes on its way, pluss the packet length. In equation 5.4 x

represents the size of the ring, t_{pass} is the time taken to pass through a node and t_{wire} the time used in wires.

$$\text{Minimum average latency} = \frac{x}{2} \cdot (t_{pass} + t_{wire}) + r \qquad (5.4)$$

r is the packet-length in nanoseconds.

5.2 Physical Part

The physical part of SCI defines the electrical, the thermal and the mechanical environment for a SCI system. SCI defines three types of physical links. These links offer a reliable packet transmission to the logical protocols. There is one parallel electrical link for use over short distances (meters). Then there are two serial implementations, one using electrical signal (tens of meters) and one using fiber optics (kilometers). The mechanical module and rack used in SCI is defined using the IEEE 1301 and IEEE 1301.1 standards.

In a SCI system the links continually transmit symbols, whether there are any data to send or not. The symbols are made up of 16 bit data, 1 bit packet delimiter and 1 bit clock information. The clock information is used by the receiver to physically be able to extract the data from the incoming signals. The defined links, to carry the signals in SCI are:

Type 18 - DE - 500 (DE = Differential ECL)

Type 1 - SE - 1250 (SE = Single Ended)

Type 1 - FO - 1250 (FO= Fiber Optics)

The notation for the links are defined as follows:

Type <number_of_signals>–<kind_of_signal>–
<bit_rate_per_signal_in_Megabits/sec>.

Type 18 - DE - 500 This link is intended for applications within a backplane and distances up to a few meters. It uses differential signalling on the lines. The links and node-link interface are implemented in ECL-technology. The symbols are sent in bit parallel. The use of differential signalling demands the use of two wires per signal-line. Using differential ECL signalling has some rather nice properties: The signal levels are relatively small (gives greater speed), it gives a high transmission impedance and the lines become less sensitive to noise. All this makes the link able to operate at a rate of 1 GByte/sec.

Type 1 - FO/SE - 1250 This is the serial link. It is thought that it should be used for longer distances where the cost of wires would be rather expensive using bit parallel transmission of the symbols. The transmission medium can either be coax or fiber-optics. For shorter distances low-cost LED's or coax should offer the best alternative. To communicate over longer distances one would need high-cost lasers.

5.3 Protocols

Logic function SCI-nodes have to be able to receive data on its incoming link at the same time as it transmits symbols on the outgoing link. To deal with this a node needs FIFOs to store symbols intermediately. The input-output intermediate store (INPUT-OUTPUT fifos) is necessary to interface between the higher speed on the link and the lower speed of the rest of the node. The BYPASS-fifo is used for storing symbols passing through to other nodes, while the node is sending one of its own packets. The INPUT-fifos receive symbols (packets) destined for this node. There is usually at least one INPUT-fifo for requests and one INPUT-fifo for responses. The size of the fifos should be at least the size of the largest packet-size allowed in the system, with the exception of the BYPASS-fifo who is a little bit larger. The BYPASS-fifo should be able to store an echo-packet [5] in addition to the largest packet. The internal logical structure of a node is shown in figure 5.3

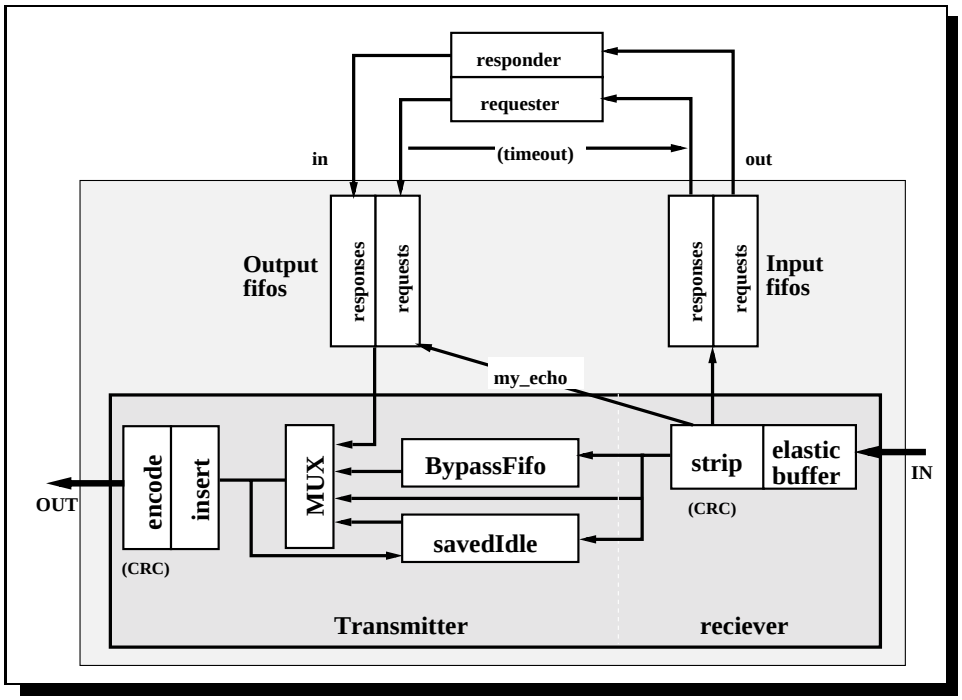


Figure 5.3: Node logical structure.

5.3.1 Input synchronization and elastic buffers

All SCI nodes are synchronous. This goes for both their internal logic and the data transmission. The data transmission is what one would call *source-synchronous* (see chapter 3). The synchronous nature of SCI

[5] See section 5.3.5 and 5.3.4 respectively for use and definition.

very much simplifies the logic in the nodes. So the receiving part of the node is fairly simple. It consists mainly of a receiver containing an elastic buffer. See figure 5.3 for the actual placement in a node. The elastic buffer has the task of compensating for the drift in phase over time between the incoming data clock and the local receiving clock. The other means of phase compensation, is the information contained in the sync packets. This helps to correct for phase drift between the individual bits (skew between the lines).

One of the reasons for having idle-symbols (see section 5.3.5.5) between every packet is to make the elastic buffer model work. The way the elastic buffer works is that it introduces a short delay (a buffer) of 2 symbols . Choosing symbols correctly from this buffer allows the elastic buffer to delete and insert idles from the ring to make the necessary phase corrections. The symbols used for this purpose are called elasticity symbols. They can be both idle and sync packets. It is very important that there always are enough of them to make the whole system work as supposed to. To understand how the elasticity buffer works with insertion and deletion see figure 5.4. When the receiving clock is faster than the internal clock idle-symbols have to be deleted (see figure 5.4 b). When the receiving clock is slower than the internal clock idle-symbols have to be inserted (see figure 5.4 a).

5.3.2 Flag coding of incoming data

For the node to be able to recognize the various packet types arriving on the link it makes use of the flag signal. The flag signal has a bit-pattern (transitions) which is exclusive for each kind of packet. Thus the interface can by decoding the flag-signal distinguish between the packet types and also their duration. The concept is graphically presented in figure 5.5 to make it easier to understand. The transitions of the flag signal are specified as follows:

$0 \longrightarrow 1$	This is the start of a packet.
$1 \longrightarrow 000\cdots$	The number of trailing zeroes identifies the packet type.

Coding-decoding specification for the different types of packets:

Send packets:	At least the 4 first bits high with 4 trailing zeroes
Sync packets:	First bit high with 7 trailing zeroes
Echo packets:	3 bits high with 1 trailing zero

5.3.3 Switching techniques

How well does SCI fit into the switching techniques described in section 3.3 ? On a ringlet packets are buffered in the bypass-fifo if the output-channel of the interface is busy. Otherwise the symbols (flits) are passed on immediately. Thus, on a ringlet, cut-through switching is used.

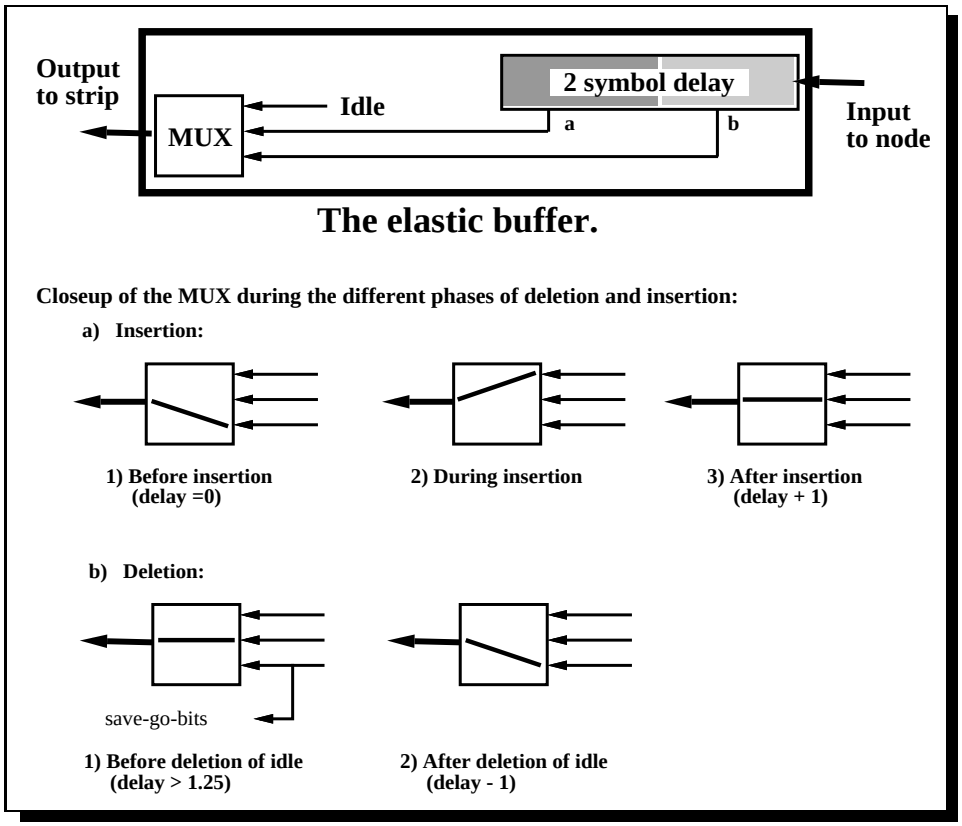


Figure 5.4: Elastic buffer models.

The switching technique *between* rings is not defined in SCI. It is up to the network-designer to decide. In our simulation of SCI (see chapter 8 and 9) we have studied both the effects of store & forward switching and virtual cut-through switching in the switches.

5.3.4 Transactions

All the communication in SCI is defined by transactions. Transactions are initiated by a requester and completed by a responder. Every node in the system can have both responder and/or requester capabilities. In other words memory and I/O nodes typically need only to be able to respond to requests from others, while the processor nodes almost always needs to execute both requests and response transactions. Transactions consist of one or two sub-actions dependent on the type of transaction. The sub-actions are then of cause called response- and request-sub-action. Each sub-action involves two packet transmissions. One sent on the output link and one received on the input link. Thus a typical transaction is made up of four packets being transmitted in the network. This is illustrated in figure 5.6.

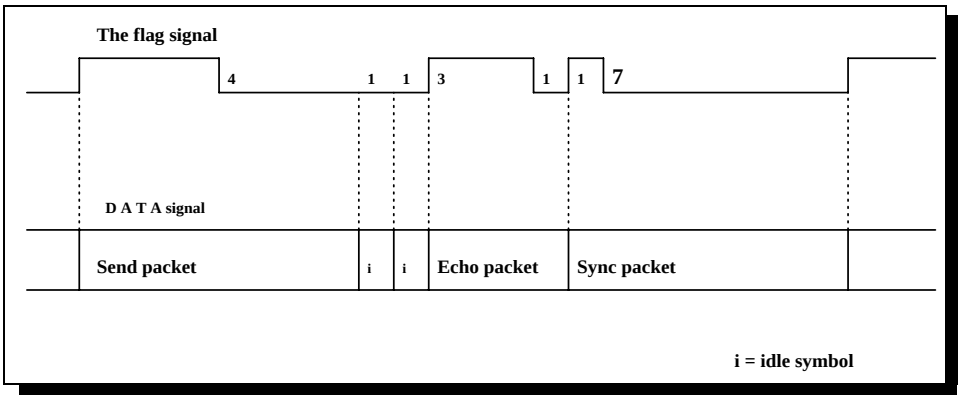


Figure 5.5: Flag-signal coding.

If the responder in a transaction does not have available buffer space to receive the incoming packet, it generates an echo with the busy flag set (echo busy). This tells the requester that the responder could not process the packet (the packet has been busied), because of the lack of queueing space. The requester then either has to resend the packet until it is received successfully or give up. When a responder has to send an echo-busy it must make a space reservation. This is done to increase the possibility for acceptance of the resent packet. The protocol to implement this is a simple A/B aging scheme. This allows a node to in a simple way choose between B, resent packets, and A, not busied (not acknowledged) send packets.

When a transaction takes place between nodes on different ringlets it is called a remote transaction. This involves the use of an agent. The agent could be a switch or bridge. A switch is used to internally route between the ringlets in a SCI-system. A bridge is used to interface between SCI and another system. This could for example be a VME-bus.

The agent is used to remove packets from one ring and to insert it into another ring. When doing this the agent takes responsibility of further forwarding of the packet. It sends a local echo (acknowledgment) on the ring which it received the packet from. See figure 5.7.

There are four main types of transactions:

Name	Request		Response	
read*xx*	Header		Header	0,16,64,256
write*xx*	Header	0,16,64,256	Header	
move*xx*	Header	0,16,64,256		
locksb	Header	0,16	Header	0,16

xx indicate one of the legal data block sizes, indicated by the corresponding numbers in the response and request columns.

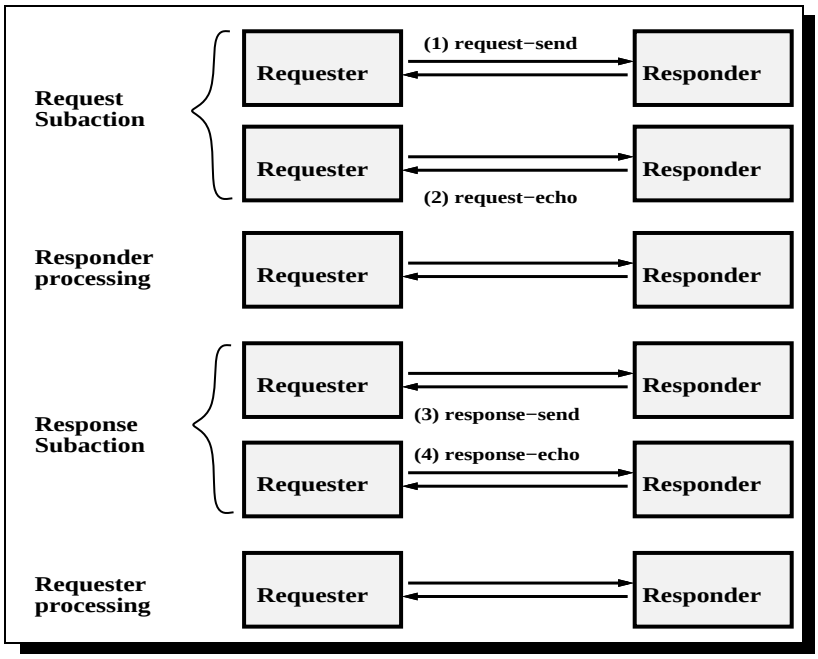


Figure 5.6: Example of a typical transaction.

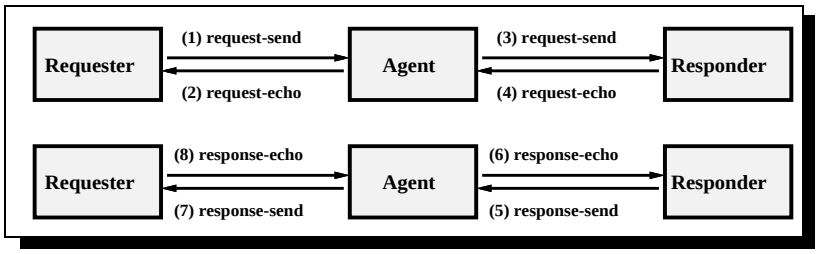


Figure 5.7: Example of a typical remote transaction.

read*xx*: Copy data from responder to requester. With response subaction.

write*xx*: Copy data from requester to responder. With response subaction.

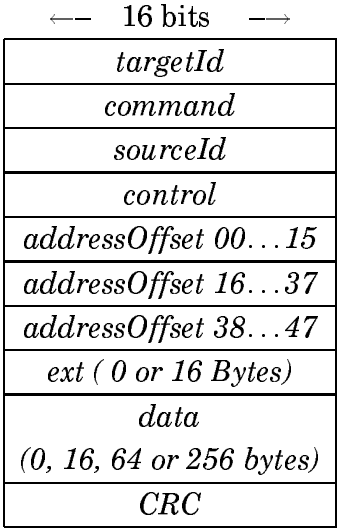
move*xx*: Copy data from requester to responder. Without response subaction.

locksb: Selected byte-lock. Copy data from requester to responder. The responder updates the address to lock, according to the command-field in the packet, and returns the previous address contents to the requester. This is a non-coherent transaction, used to implement indivisible operations (also known as “atomic operations”). See [IEEE-SCI].

5.3.5 Packet formats

SCI has many types of packets. Explained here are only the types used in our simulator. The request packets, the response packets and the echo-packets are explained below. The request packets are those originating from the initiator of a transaction and response packets are sent back in response to the initiator's request. Echo-packets are acknowledgements of received packets.

5.3.5.1 Request-Send Packet Format



Description of the various fields in the packet.

1. *targetId* is the id of the node for which the packet are destined. I.e. the destination address.

2. The *command* field gives the type of packet and flow control information.

3. *sourceId* is the id of the node which sends the packet.

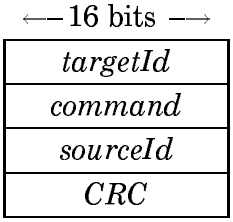
4. *addressOffset* is the internal address in the requester node. It may be addresses or registers.

5. *ext.* A portion of the extended header.

6. *Data.* 0, 16, 64, 256 bytes of data.

7. *CRC* Cyclic Redundancy Check. 16 bit CCITT CRC.

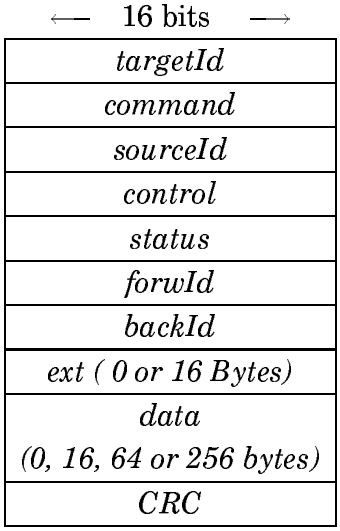
5.3.5.2 Request-Echo Packet



Description of the various fields in the packet.

1. *targetId* As for the Request-Send packet.
2. *command* Contains part of both the command and control symbol from the send packet. See description below.
3. *sourceId* As for the Request-Send packet.
4. *CRC* Cyclic Redundancy Check.

5.3.5.3 Response-Send Packet Format



Description of the various fields in the packet.

1. *targetId* is the id of the node which the packet is destined for. Used to route from the responder to the requester.
2. The *command* field gives the type of packet and flow control information.
3. *sourceId* is the id of the node which sends the packet.
4. *status* gives the status of the transaction. Used to indicate transaction status including cache-coherence information.
5. *forwId* and *backId* are used by the cache-coherence protocol to maintain the distributed directory list.

6. *ext.* A portion of the extended header.
7. *CRC* Cyclic Redundancy Check. 16 bit CCITT CRC.

5.3.5.4 Response-Echo Packet

←−16 bits −→
targetId
command
sourceId
CRC

Description of the various fields in the packet.

1. *targetId* As for the Response-Send packet.
2. *command* Contains part of both the command and control symbol from the send packet.
3. *sourceId* As for the Response-Send packet.
4. *CRC* Cyclic Redundancy Check.

5.3.5.5 Idle symbols

Idle symbols fill the part(s) of the ring that is idle. They are also used for synchronization. There is always an idle symbol between packets.

←−8 bits −→	←−8 bits −→
Various bit-fields	*error correction*

Of the "various fields", the go-bits are the most important. They control the bandwidth (see section 5.3.6). The " error correction" is just the inverted of "various fields".

5.3.6 Bandwidth Allocation

For allocation of bandwidth on the ringlets SCI implements a concept of partial fairness. This involves a fair allocation of bandwidth among nodes with equal priority. Unfair partitioning of bandwidth is used between the different priority levels. This means that the nodes with the *current* highest priority on the ringlet get the most of the bandwidth. The remaining bandwidth is fairly allocated among the rest of the nodes. This is illustrated in figure 5.8 with the four priority levels that SCI defines: it ranges from P0 (lowest priority) to P3 (highest priority). The highest priority in use in this example is P2. This means that nodes with priority P2 get the most of the bandwidth. The remaining bandwidth is shared equally between nodes with priorities P1 and P0.

Having the allocation protocols organized in such a manner gives some benefits on the following:

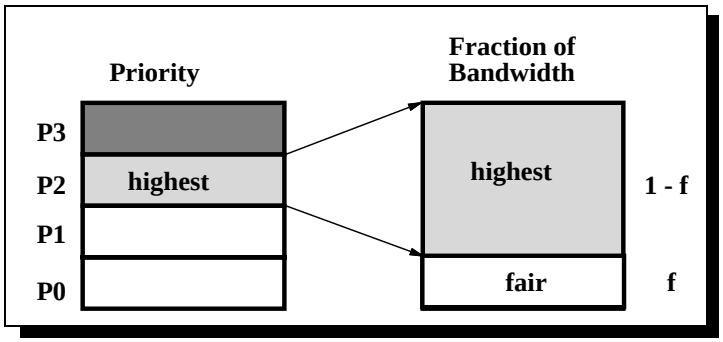


Figure 5.8: Bandwidth Partitioning.

- Guarantee forward progress on the ringlet. Allowing temporarily blocking of high priority packets without any deadlocks.
- Deterministic upper limit for time-outs, ie. the worst case transaction time-out values can be calculated.
- Queue-Allocation protocols. Using partial fairness gives an upper bound for time to retry busies. This makes the queue-allocation protocols simpler.

Fair Bandwidth Allocation Fair bandwidth allocation means that nodes with the same priority get the same amount of bandwidth. To accomplish this a round-robin protocol is used. The protocol makes use of the go bits in the idle symbols to decide which nodes get to transmit and which ones doesn't.

In every idle-symbol there are two go-bits. One for the highest priority level and one for the lowest. We will make no distinction between them for the rest of the explanation, because they have the same functionality in each priority group (highest/lowest).

By setting the go-bit on/off you indicate a go- or no-go-idle. Packets can only be sent after an idle-symbol with a set go-bit. By controlling the amount of go- and no-go-idles on the ringlet the bandwidth can be partitioned between nodes according to the protocol.

There are two conditions that have to be satisfied for a node to be able to send a packet. The bypass-fifo must be empty and there must be a go-idle present, to send in front of the packet. See figure 5.9 part (1).

When the transmission starts the node enters state BLOCKED. In this state further forwarding of go-bits is delayed. The incoming go-bits are saved and merged into a save-go-queue. The bypass-fifo will increase in size. This is illustrated in figure 5.9 part (2).

It stays in state BLOCKED after the transmission of the send-packet is ended. Figure 5.9 part (3). After ending the transmission the node still stays in state BLOCKED until the bypass-fifo is empty. Then it releases the save-go-queue and leaves state BLOCKED. Figure 5.9 part (4).

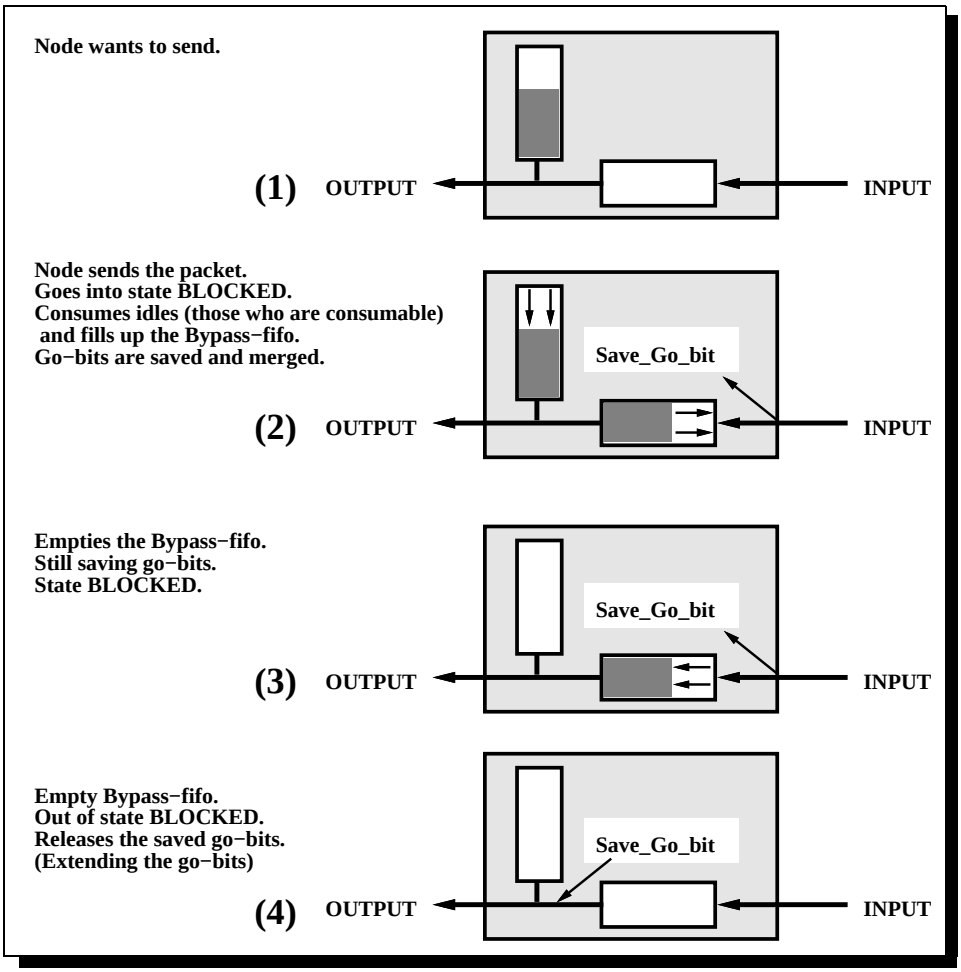


Figure 5.9: Using go-bits with the fair bandwidth allocation protocol.

5.4 Cache coherence

Almost every processor developed today makes use of caches to reduce the latency involved in memory access. A cache is a small high speed local memory in which the processor stores data temporarily. In a multiprocessor system, like SCI, this leads to some problems. The essence of the problem is that several processors often share the same line of memory in their cache. What should be done to keep the consistency of the data when independent processors do writes and reads on their local copies? This is called the cache coherence problem. This is discussed in section 2.3.4. SCI has defined a protocol to handle cache coherence. This protocol also allows both the use of non-coherent and coherent cache operations. It is based on a concept called *"distributed directories"*. This is the same as "chained directory" explained in section 2.3.4.2.

The concept of "chained directory" fits nicely into the policy of SCI, which demands a high scalability. In theory one has no upper limit to

the number of nodes in the list. The directory does not have to allocate space which in most cases will be useless. Another point is that by using a list with pointers one doesn't have to relay on broadcasts. All cache-information can be directed only to the nodes involved. The labor by updating the list is also shared between the processors , not only left to the memory management unit.

The list in SCI is a doubly-linked chain of pointers, connecting every sharing node. By making it doubly-linked, the operations on the list are very much simplified. An example of such a list is shown in figure 5.10.

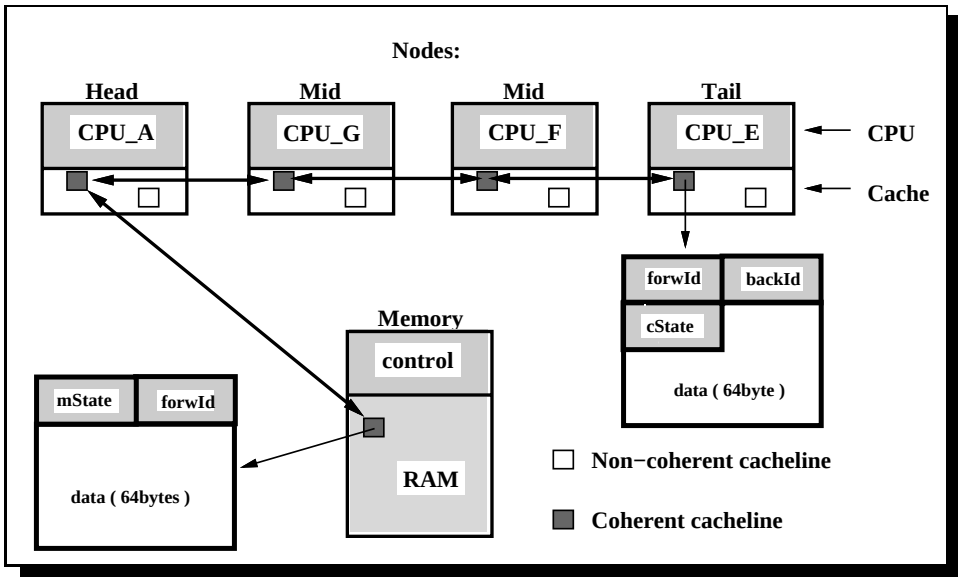


Figure 5.10: Distributed cache-line list.

The placement of the nodes in the figure does not reflect anything about the physical connections and orientation of the nodes.

The figure show that every cache has two pointers, pointing to next and previous one in the list. In addition to pointers, each cache has a 7-bit tag field; *cstate*. This introduces an overhead of approximately 7%. In the memory there are one pointer, to the head of the list, and a 4-bit *mstate* field. The memory overhead is approximately 4% .

Every node in the list can read data from the cache. It is only the first node in the list that has write access. The node executing a write is responsible for purging data stored in the other caches. There are transactions for operations on the list available. For a closer description see [IEEE-SCI].

The cache-line size in SCI is fixed to 64 bytes. Having a fixed length gives some nice properties:

Small Tag Overhead. Tags do only occupy a very small part of total space.

Reasonable Efficient. The 64-byte transaction in SCI is efficient. It uses almost two thirds of the bandwidth for data.

Uniformity. Some other buses use the same cache-line size.

When the number of processors in the sharing list gets very large this scheme is too slow. IEEE P1596.2 is an extension of the cache coherence scheme with this in mind (see section 5.5).

5.5 Other SCI-related projects

There are several projects going on with SCI related topics [Gustavson]. A brief list of these IEEE projects:

P1596.1 SCI/VME Bridge. A specification of a bridge allowing connection between a SCI-system and a VME-bus.

P1596.2 Cache Optimization for Large number of SCI processors. Cache optimization scheme that uses a tree-structure to represent the cache-list and request-combining to further increase efficiency. Useful when the number of processors is very large (1000 or more).

P1596.3 Low-Voltage Differential Interface for The Scalable Coherent Interface. Specifies the use of low voltage swings in signals used to implement the SCI-chips. Making low-cost CMOS chips to be used in workstations and PCs at speeds of at least 200 MBytes/sec.

P1596.4 High Bandwidth Memory Interface. This project is called the RAMLINK and tries to utilize the SCI technology to provide a high-bandwidth interface to memory chips.

P1596.5 Data Transfer Formats Optimized for SCI. Specification of data types and formats that will work efficiently on SCI for transferring data among heterogenous nodes in a SCI multi-processor system. The work is now finished.

P1596.6 SCI Real Time Applications. Specifications made to satisfy the needs of real-time applications for SCI.

6

Swipp & SCI

In this chapter we present the SWIPP-concept and make a comparison of SWIPP[1] and SCI. We will also make some suggestions on how the switching network in SWIPP might be organized.

6.1 Presentation of SWIPP

SWIPP is a multicomputer study under development at Department of Informatics, University of Oslo.

SWIPP is a multicomputer system for heterogeneous computing nodes. It offers fast communication channels between the computing nodes. High bandwidth and flexibility are obtained by employing custom-designed VLSI-switches, together with point-to-point links. Special modules (protocol engines) between the computing nodes (compute engines) and the network are used to off-load communication tasks from the compute engines. There is no physically shared memory, instead a logically shared memory (distributed in the compute engines) might be employed. To achieve this "shared data-space" message passing is employed. Presently a number of master-of-science students are writing their thesis as a part of the SWIPP-project: [Lundh, La et al, Østby, BlekHag, Baltz, Karlsen, Roseth, Larsen, EsvSchrø, NerSmaTor].

SWIPP is composed of the following components:

Compute engines are independent hosts with their own processor and memory. They may have different architectures (heterogeneous). They are henceforth called CE's. The CE's are connected together through a communication system of switches.

Protocol engine is the module between the CE and the network. Its main task is to carry out communication duties for the CE, thus increasing the total performance of the CE. The protocol engines (henceforth called PE's) receive data from the CE's. This data is put into packets by the PE and then sent out on the network to another CE.

[1] SWIPP = Switched Interconnection for Parallel Processors

Switches The switches contain a crossbar-matrix with s input channels and s output channels ($s = 4$ in figure 6.1 and 6.2). Each channel is internally a 9 -bit bus (8 for data, 1 for control). Each input channel includes a buffer (a first in first out buffer) called the input port. At the output is the output port. s is planned to be 16.

Links There are two schemes for the links connecting the switches and the PE's.

1. Optical. The links are made up of two fibers (one for each direction). An advantage in using optical fibers is that high bandwidth is achieved, independently of distance. In addition, the fibers are not sensitive to electrical noise. See figure 6.2. An optomodule (see item below) is then needed at both ends of the fiber.
2. Electrical. The links are made up of 9 parallel wires in each direction . Electrical links are for shorter distances.

Optomodule The optomodule connects the optical links to the switch or PE. It transforms parallel electrical signals from the PE or switch to a serial optical signal (and vice versa). Henceforth called OM.

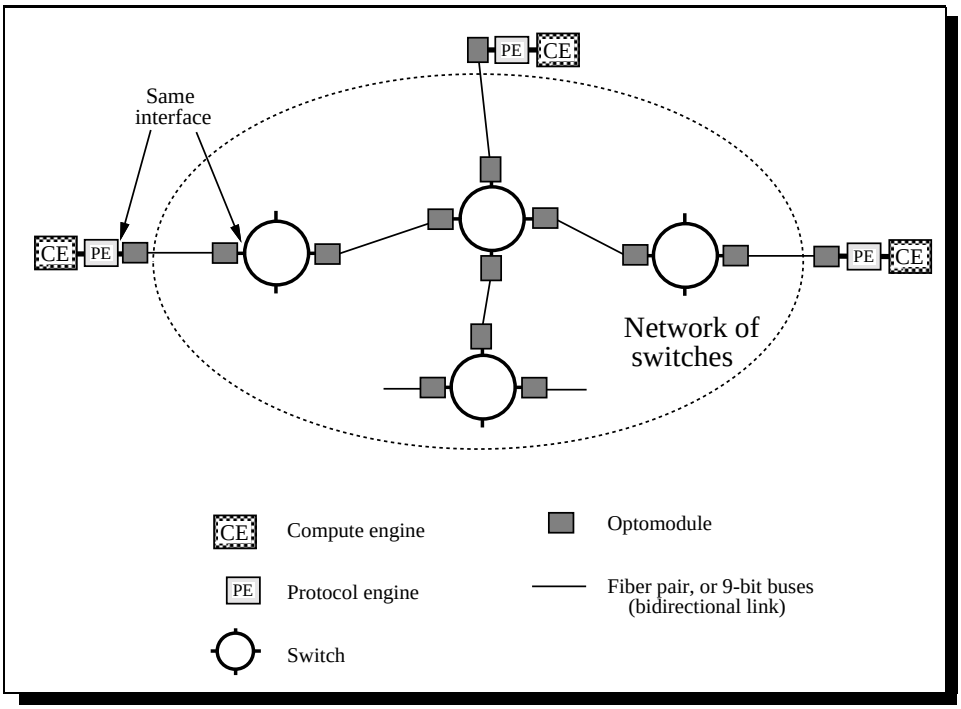


Figure 6.1: The SWIPP concept. OM omitted if fiber links are not used.

Routing in the network SWIPP uses source-routing. The PE's have a routing table. Before sending a packet, the PE's prepend it with a

list of addresses, giving the route through the network. The advantages with this scheme:

- It reduces the work of the switches. There is no need to have tables in the switches. The complexity is put in the PE's instead. Less logic is needed to handle an incoming packet.
- The time to make a connection in the switches is reduced.
- It eases upgrading of a switch to use optical technology when that becomes available.

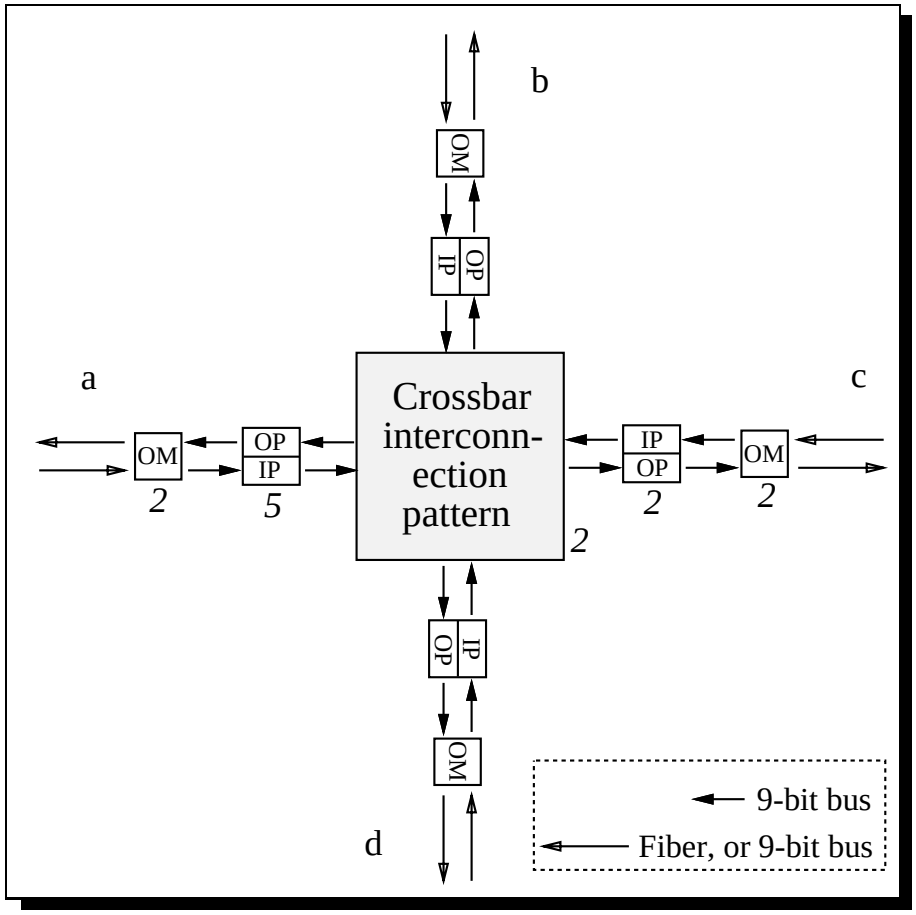


Figure 6.2: The switch used in SWIPP. In this example the switch has 4 channels ($s = 4$) Each channel has an input port (IP), an output port (OP), and an optomodule (OM) (if fiber-links are used). The numbers suggest *roughly* the number of clockcycles a flit needs to traverse the switch from 'a' to 'c'.

6.2 Comparative study of SWIPP and SCI

Both SCI and SWIPP have much of the same basic functionality at a low level. They both offer a module allowing the connecting of several

processors and memories together forming an interconnection system. But the ideas leading towards the final result were quite different.

SCI had its origin from a group of people working with bus standards in the late 1980's. On their search to increase the performance on buses, they realized that if the performance should have more than a marginally increase one would have to choose a new scheme for communication. This leads to the development of SCI. In SCI the means of communication are point to point links instead of the traditional bus. In addition to communication, a protocol to make SCI capable of handling the cache coherence problem is offered as a major part of the standard. When developing SCI as a concept, one had to put much thought in the problem of what kind of technology and protocols it should depend on. It should make optimum use of the current available technology as well as making it possible to take advantage of the coming advances in technology.

SWIPP on the other hand has its birth in a university environment. It is based on the idea that if a multi-computer should make the most out of its resources, the task of communication should be left to a separate unit. This unit should take care of the communication and communication protocols. Developing such a unit should involve the study of several ways of implementation. It should have in mind predictions on what technology should offer in the next decade or so. This project has been going on for quite some time and has resulted in several theses. This has lead to a rather lengthy process where much of the main lines of the project were floating back and forth, as each of the theses in the project had to be time limited and with an academic content.

Some differences between SWIPP and SCI:

The physical difference of the links. Both concepts have possibilities to use both electrical links and fiber-optical links for transmission. The main focus so far has been with fiber-optics in SWIPP and electrical in SCI. The electrical links are as one would expect in both concepts parallel and the fiber-optics serial. The difference then between the two concepts is mainly the width of the communications channels. SWIPP uses 9 bits in parallel while SCI uses 18 bits. 8 of the bits in SWIPP are used for data and 1 for controll. In SCI 16 of the bits are used for data and 2 for control. The other difference is the speed of which to operate the links. It is much dependent on the technology in which the "final" implementation is done. This is the current specifications. SCI can send 8G bits per second over its parallel electrical links and 1G bits per second on its fiber-link. SWIPP is capable of at least 1G bits per second over its fiber-link.

Switching technique As mentioned in section 5.3.3 SCI uses virtual cut-through on the rings. The behavior of the switches is implementation dependent. In any case the packets must be buffered if a conflict

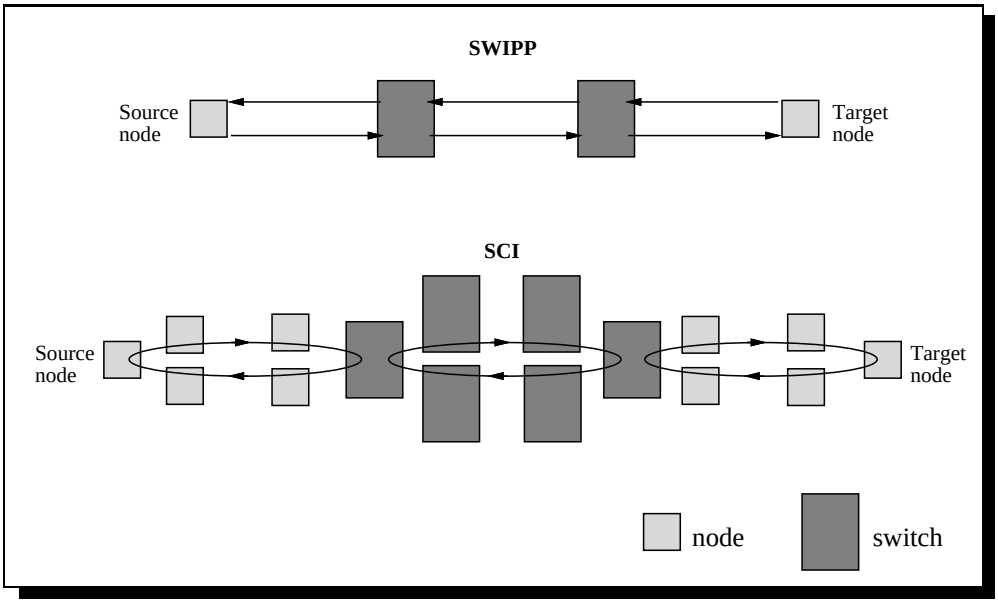


Figure 6.3: The figure shows an arbitrary path between two active nodes that a packet must traverse. We see that both SCI and SWIPP has bidirectional links, logically.

happens, thus setting an upper limit on the packet-size according to the fifo-size.

Swipp uses wormhole "routing". Since the input buffers are much smaller than the size of the packets, part of the return bandwidth of the links is reserved for signalling purposes. If a buffer in the switch/PE's receiving the packet is nearly full it must somehow pause the sender of the packet. Thus a signal is sent to the sender (on the return link) telling it to stop for a while.

The switches. The switches has two tasks to perform. One is to connect the network (the k-ary n-cube-structure) together. The other is to connect the local nodes to the network (and thus to the k-ary n-cube-structure).

The former can be done similarly in both SWIPP and in SCI, as shown in the left parts of figure 6.4 and 6.5, using switches, and for example rings in a cube. The latter is done with a switch in SWIPP and with a ring in SCI, as seen in the right parts of figure 6.4 and 6.5, respectively.

Building a switch in either concepts depends on whether one uses the parallel or serial versions. It is obvious that the parallel versions set a limit to the number of input and output connections. Thus allowing a serial implementation to connect more nodes.

Switches in SCI need more logic in each switch to make the routing decisions. This is due to the fact that each packet has only the target and source node address contained within it. Thus some sort of address

table has to be stored and accessed in the switches. The output buffers also have to be able to store a whole packet. The protocol demands the sending interface to store the packet until an acknowledgement is received.

Transmission-time through the switches In SCI the switches are not standardized, they are implementation-dependent. If the switching technique virtual-cut-through is used, a flit (a symbol) should be able to accomplish roughly 10 clock-cycles through a switch (not presently).

SWIPP uses wormhole, and should accomplish roughly the same speed as SCI.

Measuring the speed is more than just comparing the number of clock-cycles. In SCI the cycle-time is 2-3 nanoseconds. The cycle-time in SWIPP is uncertain since a real implementation has not been designed yet.

A point to mention is that one has to balance the speed on the links and the the time to decode and route the packets in the switches. For the logic in the switch to handle the high speed one might have to introduce extra delay (buffer).

It is expected that SCI-switches will have a "favored switch setting" (page 66). Thus the communication will be faster between pairs of input and output ports on the switch.

SWIPP does not have this property. The switches have not been constructed with this in mind. The usefulness of this property depends on the topology. It is handy when constructing k-ary n-cubes.

Packets and formats. In SCI packets can be as long as the size of the packet-buffers. They thus have a fixed header-overhead. See section 5.3.5.1.

In SWIPP packets can be of variable length. Packets can be as long as desired, thus effectively becoming circuit-switched. Thus having larger packets reduce the overhead of the header.

The application layer. A large difference between SCI and SWIPP is in their respective use. SCI is mainly intended for a shared-memory multiprocessor. SWIPP has in mind an environment of heterogenous computers that communicate by sending messages. To permit multiple processes to interact the "shared data-space" is used [Larsen].

See also page 10 in section 2.3.3 for a discussion of shared memory versus message passing.

Addressing The addressing scheme used by SCI and SWIPP is different. SCI uses an absolute addressing scheme to address up to 64 K (2^{16}) nodes. SWIPP uses a relative addressing scheme to address the output links of the switches. There is no explicit limit to the number of nodes to address, the addresses can be as long as the largest "hop" in the network (diameter of the network). 4 address-bits per hop.

With the SWIPP-addressing scheme it is relatively easy to attach additional nodes to the system at a later stage. The routing tables in all the PE's must then be updated, though.

With the absolute addressing scheme of SCI this becomes more difficult. Care must be taken when designing a routing algorithm. Otherwise addresses may have to be changed when attaching additional nodes.

Connecting to other buses/networks. How much has SWIPP and SCI thought about interacting with other systems? SWIPP has made a thorough study on how to connect to ethernet (IEEE 802.3). A bridge between SWIPP and ethernet is presented in [Roseth].

Within the SCI-community work has so far concentrated on defining a standard for a bridge between SCI and VME-bus (see section 5.5). In addition there has been studies on how to connect SCI to Futurebus+, Turbochannel and HIC, among others.

	SWIPP	SCI
Addressing	relative	absolute (max 2^{16})
Switching	wormhole + signaling	virtual cut-through
Routing	source-routing	undefined
Memory model	message passing	shared memory or message passing
Cache scheme	none	distributed linked list
Packet length	no limit	limited by fifo-size
Links	2*9 unidir. wires or 2 fibers	18 unidir. wires or serial fiber
Status	prototype made	standard, chips available
Error correction	CCITT 16-bit CRC	CCITT 16-bit CRC

Table 6.1: Summary of comparison of SCI and SWIPP.

6.3 k-ary n-cubes implemented in SWIPP & SCI

As shown in the discussion in the previous section the difference is mostly in the protocols. Both can use serial and parallel communication in and out of a node. Each will use whatever is most suitable for a specific problem. All kinds of topologies can be connected with both schemes.

SCI has thought little about switches and how to connect topologies. SWIPP has done some more thinking about the switches, but not so much about topologies.

As a more interesting form of comparison let us discuss the performance of k-ary n-cubes as implemented in SWIPP and in SCI. In systems

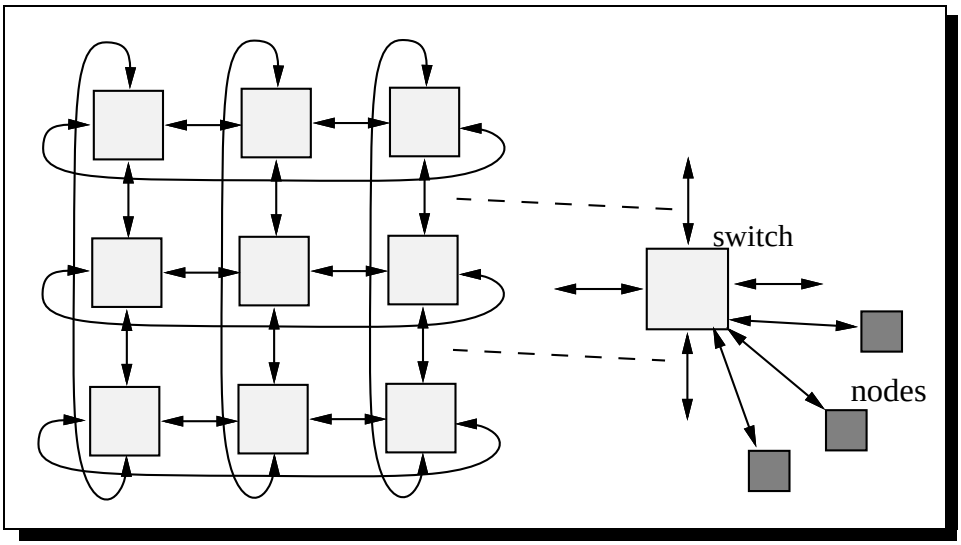


Figure 6.4: k-ary n-cubes implemented with SWIPP. Each vertex has one switch and p nodes attached to it. The switch has then $s = 2n + p$ channels. Here $p = 3$ and $n = 2$. 7 channels for this 3-ary 2-cube.

like SCI most of the traffic generated is triggered by cache operations. This behavior is more thoroughly discussed in section 8.6. In systems where the greater part of traffic is caused by cache operations, the latency is the most important parameter to observe. Thus we will concentrate on latency in the following comparison.

We assume:

1. Each vertex in the k-ary n-cube has one switch and p active nodes.

2. The switches for the SCI scheme connect n dimensions, plus one extra link for node(s). That requires $n + 1$ interfaces in a switch [2]. See figure 6.5.

   The switches for the SWIPP scheme connect n dimensions, plus one extra link *per* active node. That requires $2n + p$ channels in the switches. p for SWIPP is constrained by the size of the switch. The maximum number of nodes per vertex is then $s - 2n$. See figure 6.4.

3. A k-ary n-cube implemented with SCI has unidirectional links, as shown in figure 6.5. SWIPP must have bidirectional links (important requirement of the protocol), as shown in figure 6.4. We assume both use a torus-connection scheme.

[2] **Note:** we will not (for simplicity) use the scheme presented in section 7.1.2 on connecting k-ary n-cubes for SCI. This comparison between SCI and SWIPP would then be needlessly complex.

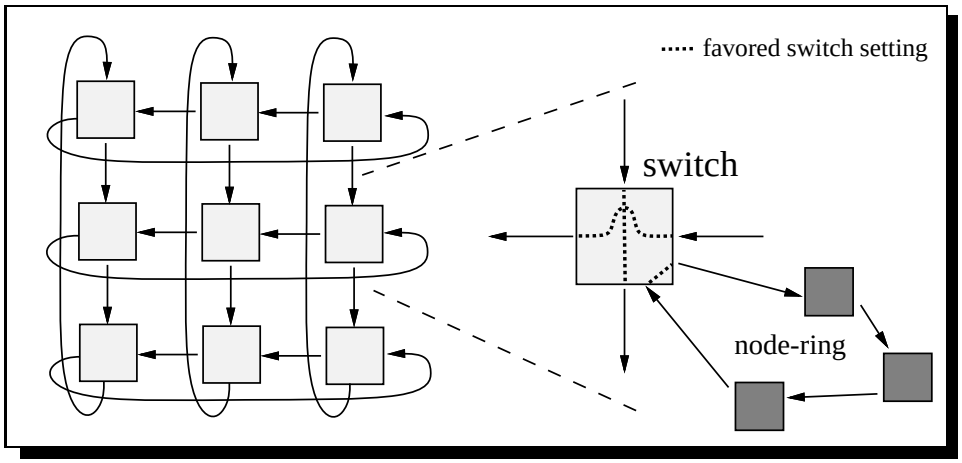


Figure 6.5: k-ary n-cubes implemented with SCI. Each vertex has one switch and a ring with p nodes attached to it. The switch has $n+1$ SCI-interfaces. 3 SCI-interfaces for this 3-ary 2-cube.

SCI can also be bidirectional, each link is then functionally a ring. In this discussion though, we assume SCI is unidirectional. This is a more efficient allocation of hardware resources in SCI.

4. SCI has a 2 byte wide link. The first implementation of SWIPP is expected to have a 1 byte wide link.

Unloaded Latency An interesting and straightforward parameter to compare SWIPP and SCI with is the unloaded latency. We assume k-ary n-cubes implemented as shown in figure 6.4 and 6.5.

All latency-equations and figures are in the number of clockcycles.

In the following discussion the following parameters are used:

1. t_w is the wire-delay. We assume 2 clockcycles-time both for SCI and SWIPP.

2. t_{br} is the switch-delay. SCI-switches might use an additional cycle or two more for address-decoding,[3] but on the other hand the PEs in SWIPP need additional time to construct the route of the packet (lookup into the routing table). Such implementation-dependent details remain uncertain. We assume for simplicity 10 clockcycles-time both for SCI and SWIPP.

3. t_{bp} is the bypass-delay for SCI. Its value is here assumed to be 5 clockcycles-time. We differentiate in SCI between the time to switch dimension (t_{br}) and the time to continue in the same dimension (t_{bp}), due to the "favored switch setting"-property of SCI.

[3] If our coordinate-addressing scheme (section 8.1.11) is used for for SCI, the time to decode the address bits should be just as fast as the source-routing scheme of SWIPP.

4. r is the time, in clockcycles, to transmit an 80 byte packet (both header and data). r is 80 for SWIPP, and 40 for SCI. The reason being the 1 byte wide SWIPP-links in contradiction to the 2 bytes used in SCI.

First we will derive the unloaded latency for SWIPP:
If the k-ary n-cube lacks torus-connection the maximum unloaded latency is:

$$Latency_{max} = n(k-1)(t_{br} + t_w) + 2t_w + t_{br} + r \tag{6.1}$$

On average the latency then is:

$$Latency_{ave} = \frac{n(k-1)}{2}(t_{br} + t_w) + 2t_w + t_{br} + r \tag{6.2}$$

We will assume the k-ary n-cube has torus-connections. Thus the maximum latency is:

$$\begin{aligned} Latency_{torus\text{-}max} &= n\tfrac{k-i}{2}(t_{br} + t_w) + 2t_w + t_{br} + r \\ & \qquad\qquad i = 0 \text{ , } k \textit{ even} \\ & \qquad\qquad i = 1 \text{ , } k \textit{ odd} \end{aligned} \tag{6.3}$$

Our goal is the average unloaded latency using the scheme in figure 6.4, which has torus-connections. Since it is the goal, it is first presented more verbally:

$Latency_{torus\text{-}ave}$
= *packet length*
\+ *time to traverse "source-wire" + "source-switch"*
\+ *time to traverse "target-wire"*
\+ *with a penalty* $t_{br} + t_w$ *change dimension*
on average $n\frac{k-i}{4}$ *times*

More formally this is:

$$\begin{aligned} Latency_{torus\text{-}ave} &= n\tfrac{k-i}{4}(t_{br} + t_w) + 2t_w + t_{br} + r \\ & \qquad\qquad i = 0 \text{ , } k \textit{ even} \\ & \qquad\qquad i = 1 \text{ , } k \textit{ odd} \end{aligned} \tag{6.4}$$

The average unloaded latency for k-ary n-cubes implemented with SCI using the scheme of figure 6.5:

$Latency_{sci}$
= *half of source vertex*
\+ *half of target vertex*
\+ *packet length*
\+ *change dimension* $n-1$ *times with probability* $k-1/k$
\+ *traverse each of* n *dimensions*

or, more formally:

$$\begin{aligned} Latency_{sci} &= pt_w + (p-1)t_{bp} + r \\ &+ (n-1)\tfrac{k-1}{k}t_{br} \\ &+ n\tfrac{k-1}{k}\left[\tfrac{k-2}{2}t_{bp} + \tfrac{k-1}{2}t_w\right] \end{aligned} \tag{6.5}$$

$Latency_{sci}$ is also dependent on the number of nodes in each vertex p. We assume in figure 6.6 and 6.7 that $p = 3$.

The average unloaded latency equation for k-ary n-cubes implemented with SCI is more complex than the SWIPP-equation because SCI switches are expected to have a "favored switch setting" (page 66), thus it costs more for the packet to switch dimension rather than continuing in the same dimension

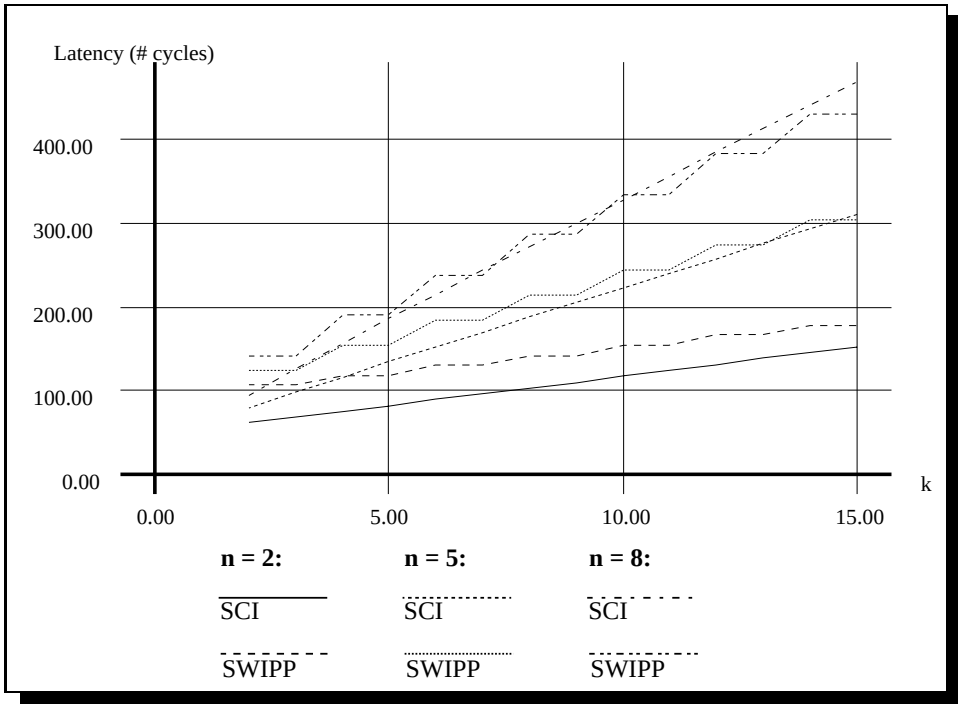


Figure 6.6: Unloaded latency, in clockcycles, as a function of the network-size (given by k and n). Equation 6.4 versus 6.5. The "staircase" -form of the SWIPP-latency is explained by the odd/even part of equation 6.4.

Let us now compare the mean unloaded latency (in cycles) for SCI and for SWIPP. We assume k-ary n-cubes as in figure 6.4 and 6.5. We thus compare equation 6.4 and 6.5. Remember though, that there might be a difference in the cycle-time in the two concepts (see section 6.2 on page 57). For simplicity we assume that they are equivalent, but that is not certain!

We see that in most cases the SCI-latency is lower, but not to a great extent. There is also indication that SWIPP-latency is smaller for very large n and k's. As a rule of thumb: we see that as we increase the

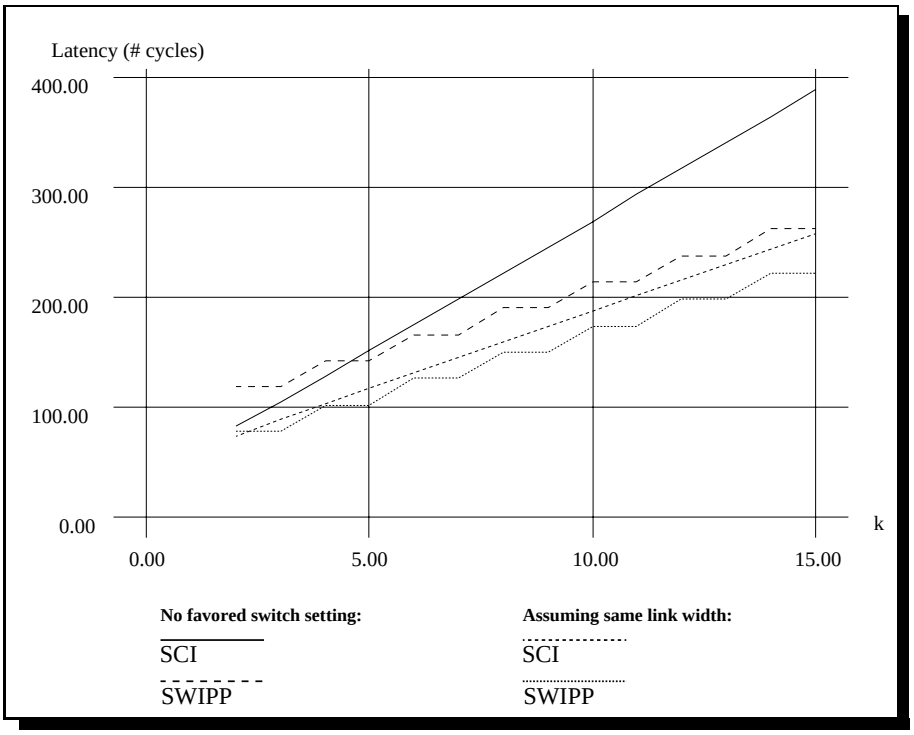


Figure 6.7: Latency. First we see that SWIPP has a much lower latency if we assume that SCI has no favored switch setting ($t_{bp} = t_{br}$). Also, if the link width of SCI and SWIPP were the same (2 bytes wide), the latency would be approximately the same. n is here constant ($n = 4$).

distance from the origin (the network size increases), the latency for SWIPP is the same, or lower than the latency for SCI.

Figure 6.7 attempts to explain the differences. The 2 upper lines show the latency if SCI has no "favored switch setting" (page 66). Thus $t_{br} = t_{bp}$. We see that there is a larger difference as the network size grows, in SWIPP's favor. Thus the fact that SCI-switches are expected to have a "favored switch setting" is important to the latency comparison. The 2 lower lines in figure 6.7 plot the latency if both SWIPP and SCI has 2 bytes wide links. We then see there is a negligible difference only. Thus the lower latency figures for SCI in figure 6.6 are explained by the larger link width of SCI and the favored switch setting of its switches.

Another matter is that as n grows the question arises of how large the switches can be.

Loaded performance The unloaded latency does not say much about the performance under real working conditions. What about conditions under load? What about throughput?

What happens when contention arises? It is then up to the bandwidth allocation protocols to balance the resources between the various parts of a system. They control an eventual difference in performance

between the two concepts under load. To study the difference one needs either an analytical queueing model or a simulation model. Both are beyond the scope of this comparison. We here informally mention some of the more interesting points, which may make a difference on the resulting performance.

- Release of buffers. The SCI-protocol operates on a ring, while SWIPP operates on a single bidirectional link. SWIPP might therefore be able to release buffers faster. Also, since the SWIPP-protocol operates on a single link, the unavailability of resources (typically buffer-space) is discovered more quickly. It is therefore probable that the SWIPP-protocol might generate fewer retransmissions. In fact most of the time SWIPP will not retransmit packet. It just slows down the sending node. It continues the transmission when the sender receives a "go-on" signal or the sender times out. Both these factors should result in a larger throughput. On the other hand packets which stop because of contention, occupy several buffers/switches (wormhole routing). This is because the complete packet cannot be buffered in a single buffer. This will slow down other packets trying to use the same route. The network, or parts of it, might become temporarily blocked. This might degrade the total performance.

- Forward progress. Forward progress is the ability to ensure the flow of packets through the network under all conditions. This is a point where SCI and SWIPP differ. SCI has a working mechanism to ensure this. This part of SWIPP is not yet complete.

- Connectivity. With bidirectional links the nodes are on average closer to one another. The average distance is then reduced. This should give a shorter latency, and in turn more throughput. There is more connectivity with bidirectional links together with switches, than with a ring.

The above points indicate that SWIPP might give a higher throughput. On the other hand, the wider link of SCI might make up for them. This is difficult to say off-hand.

Cost As a parameter for cost, let us use the nodes/switches – ratio. As can be seen in figure 6.5, it is relatively simple to add nodes to the vertices by adding nodes to the "node-ring". Thus improving the nodes/-switches – ratio. The change to latency and throughput is negligible.

It is not so simple to add nodes to the vertices in the SWIPP-implementation. One method is to enlarge the switches. A switch could have more than one channel connected to nodes. That might increase the cost of the switches. Another method: connect the switch to a new switch, which one in turn connects several nodes to. This would double the amount of switches.

The SWIPP-protocol is somewhat less complex in comparison to the SCI-protocol [4]. This could result in cheaper switches.

Summary of Comparing We have only discussed communication-aspects in this comparison. Topics like cache coherency and message passing have been kept out.

In SCI latency is more critical since cache coherency is an important part. This is not so important in SWIPP, high throughput is a more important goal.

[4] This is partly because SCI is a standard, thus more people (with conflicting wishes) are involved.

7

Topologies using SCI

In chapter 5 we presented SCI. In this chapter we show how we suggest that multiple SCI-rings be connected into a k-ary n-cube. A rough theoretical analysis of these k-ary n-cubes is then presented. This analysis is partly based on theory in chapter 4 (with some simplifications), particularly equation 7.6 and 7.7. We compare these two equations with the simulation-results in chapter 9.

Most of the classical topologies (those described in chapter 3, for instance) can be synthesized from a set of rings. The use of rings connected together with switches presents certain tradeoffs [JohnGood]:

Favored switch settings: A switch can have multiple input and output links, but each input link will have a preferred output link. In a ring-based topology it will be faster, by a considerable factor, to route within the same ring, instead of routing to another ring. For example, in the switch in figure 7.1 there are two output links. A packet entering the input link in ring 1 will leave the switch faster if it chooses the output link in ring 1, rather than the output link in ring 2. This constraint has nothing to do with rings, it is a property of switches.

Different ring size: The optimal ring size depends on the degree of favored switch setting. In other words, if the relative penalty for changing rings is very high, then the optimal ring size increases. The degree of favored switch setting is typically faster with a factor of roughly 3 - 4. This argues for large rings, but there are also arguments for small rings. One is that the echo-packets must traverse the remaining part of all the rings visited, thus wasting a part of the bandwidth. Another argument for small rings is that echo-packets then arrive faster to release buffers at the destination, thus enabling a new transaction.

Deadlock avoidance: SCI has been defined so as to avoid deadlock on a single ring, but special care must be taken to avoid deadlock within a topology of multiple rings.

7.1 Synthesizing k-ary n-cubes with SCI-rings

We will here take a look at various methods of implementing k-ary n-cubes. First we must consider a switching element to use.

The simplest way of connecting two rings is by using a bridge as shown in figure 7.1. Here 2 SCI-interfaces are connected "back-to-back". This is probably the simplest and cheapest SCI-bridge that can be manufactured. It should also be possible in due course to place it on a single chip. This is the kind of switch we have chosen to use in our simulation study.

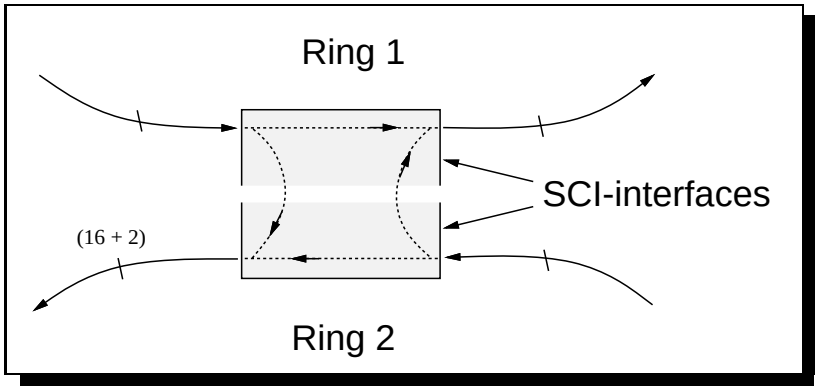


Figure 7.1: A switch connecting 2 rings.

With such a bridge multiple rings can be connected together. We will in the following concentrate on constructing k-ary n-cubes using the bridges of figure 7.1. A detailed view of this bridge, as we simulate it, is shown on page 81.

7.1.1 Surfaces

One method is to connect the cube by using "surface-rings", as shown in figure 7.2. Each surface-ring is then connected to $4(k-1)$ corner-rings (this is one possible method, anyway). We considered this alternative, but viewed it as being too complex to connect. In addition the routing is unnecessary complex.

7.1.2 Edges

In this thesis k-ary n-cubes are mainly constructed of SCI-rings in the following way: "edge-rings" connect each vertex in the k-ary n-cube. Each edge-ring has k corner-rings attached to it as shown in figure 7.3b. They are labeled as x-, y-, and z-rings in figure 7.3a. One advantage with this method is the relatively simple routing algorithm possible (described on page 94). It also has a relatively understandable topology. A further example of our k-ary n-cubes is shown in figure 7.4. Note the direction of the arrows, they show the direction the packets go along a ring. In our simulation the packet is always first routed in the first dimension (the "x-ring"). Then the second dimension, and so on. This to make the routing algorithm simple. A side effect of this is shown in fig-

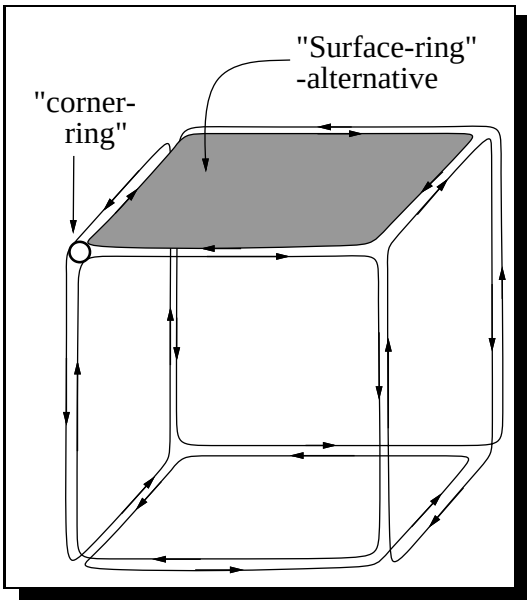


Figure 7.2: Connecting a cube by using rings: here the rings follow the *surfaces* of the cube. They are connected together with "corner-rings", as shown in figure 7.3a.

ure 7.4, where the node furthest from the one labeled 'a' is 'b' and not 'c'.

7.1.2.1 Placing the active nodes

A question is: where are active nodes placed? In this study we place these nodes in the vertices. We have studied two possibilities:

Placing the nodes in the corner-ring This is shown in figure 7.5a. It is important that the corner-rings do not become a "bottle-neck", there is a limit to the number of nodes and switches connected to the corner-ring. Thus it is not ideal to place too many nodes on the corner-ring since its main purpose is to connect dimensions (edge-rings) together.

With this scheme the number of switches in the network are

$$S_{nic} = n * k^n \tag{7.1}$$

and the total number of active nodes, P, is

$$P = p * k^n \tag{7.2}$$

where p is the number of active nodes in each vertex. The subscript "nic" refers to the fact that the active $n$odes are in the $c$orner-ring.

An interesting parameter is the maximum latency in an idle network. We will here derive the maximum unloaded latency for k-ary n-cubes as we have implemented it. First some variables:

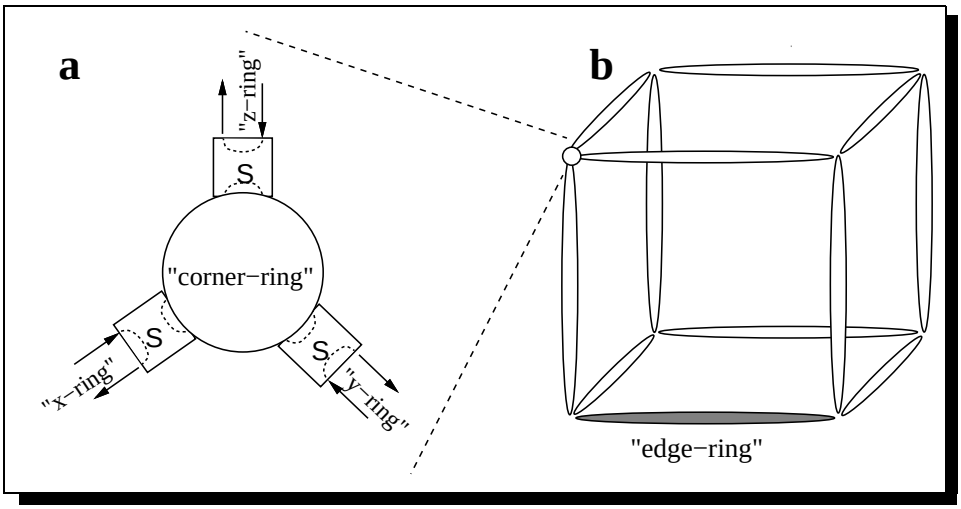


Figure 7.3: Connecting a cube by using rings: here the rings follow the *edges* of the cube. The edges are connected together with "corner-rings", as shown in **a**. 'S' denotes the switches in figure 7.1.

Description	name
wire delay (4 ns)[1]	t_w
bypass delay (12 ns)	t_{bp}
bridge delay (22 ns)	t_{br}
packet length (80 ns)	r
number of processors in vertex	p

We assume the switching-technique virtual-cut-through is used. The max unloaded latency is :

$$\begin{aligned} \textit{Max latency} &= \textit{the time to leave source vertex} \\ &+ \textit{the maximum time to switch dimension } n-1 \textit{ times} \\ &+ \textit{time to enter target vertex} \\ &+ \textit{time to receive packet of length 40 symbols} \end{aligned} \tag{7.3}$$

more formally:

$$\begin{aligned} T_{\textit{max-latency-nic}} &= t_w p + t_{bp}(p-1) + t_{br} \\ &+ (n-1)(t_w + 2t_{br}) + n\left[t_{bp}(k-2) + t_w(k-1)\right] \\ &+ t_{br} + t_w p + (p-1)t_{bp} \\ &+ r \end{aligned} \tag{7.4}$$

$$\begin{aligned} &= 2\left[t_w p + t_{bp}(p-1) + t_{br}\right] + r \\ &+ (n-1)(t_w + 2t_{br}) + n\left[t_{bp}(k-2) + t_w(k-1)\right] \end{aligned}$$

[1] For simplicity we ignore variable wire lengths, as discussed in section 4.1.1.

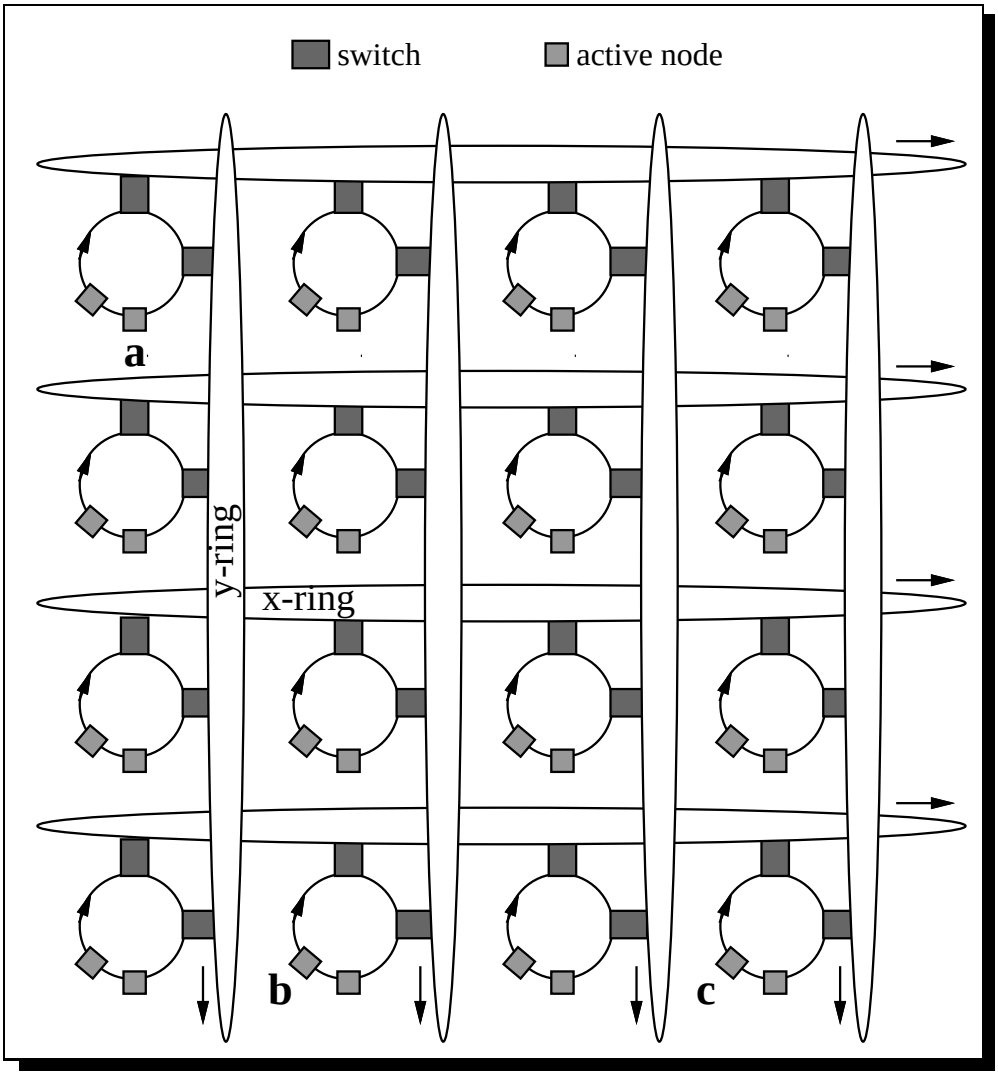


Figure 7.4: A 4-ary 2-cube with 2 active nodes in each vertex. The switches are the switches shown in figure 7.1.

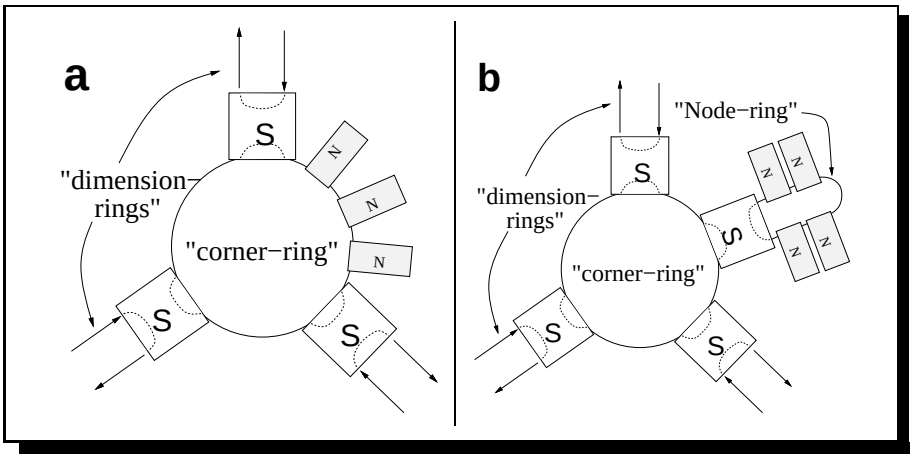


Figure 7.5: How active nodes are placed in the vertex. In **a** the nodes are placed in the "corner-ring". In **b** the nodes are placed in an additional ring. 'N' denotes the active nodes.

From equation 7.4 the mean latency can be derived. The packet enters the n dimensions with probability $k - 1/k$. It traverses halfway through the source and target vertex. r is constant. The minimum mean unloaded latency is :

$$\begin{aligned} & \textit{Mean latency} \\ = \; & \textit{time to traverse half of source vertex, and "source-bridge"} \\ + \; & \textit{time to traverse half of target vertex, and"target-bridge"} \\ + \; & \textit{time to receive packet of length 40 symbols} \\ + \; & \textit{time to switch dimension } n-1 \textit{ times, with probability } k - 1/k \\ + \; & \textit{time to traverse on average halfway through } n \textit{ dimensions} \end{aligned} \tag{7.5}$$

more formally:

$$\begin{aligned} T_{\textit{mean-latency-nic}} &= t_w p + t_{bp}(p-1) + 2t_{br} + r \\ &+ (n-1)\tfrac{k-1}{k}(t_w + 2t_{br}) \\ &+ n\tfrac{k-1}{k}\left[\tfrac{k-2}{2}t_{bp} + \tfrac{k-1}{2}t_w\right] \end{aligned} \tag{7.6}$$

Another interesting parameter is the theoretical maximum throughput. We base an expression for the theoretical maximum throughput on equation 4.11. Then the maximum theoretical throughput for the k-ary n-cube with SCI-technology is:

$$X_{max} = 2 * 10^9 * k^n \tfrac{1}{k-1} \tag{7.7}$$

in bytes per second. This for the complete k-ary n-cube.

This bandwidth is achievable if the vertices are sending only to the neighboring vertices. Except perhaps for a few applications, this is unrealistic. Saturation of the network will probably effectively block the network with much less load.

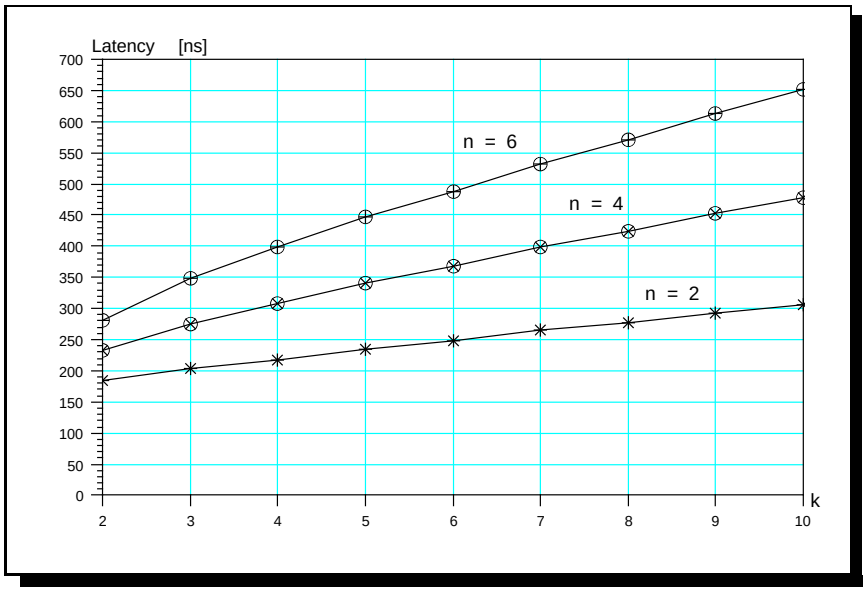


Figure 7.6: Unloaded latency for k-ary n-cubes as defined by equation 7.6. $p = 3$.

Placing the nodes in a ring by itself This is shown in figure 7.5b. An advantage with this is that the corner-ring is then reserved for switches. The size of the corner-ring can thus be restrained. Packets from other vertices, which are just switching dimensions, and not addressed to the vertex, do not have to traverse the bypass-fifo's of the active nodes. A drawback, however, is that a packet from the node-ring will use the additional time to pass through the extra switch connecting the node-ring to the corner-ring. So then the mean latency is equation 7.6 plus $t_{br} + t_w$. The rough throughput with this scheme is the same as equation 7.7.

This scheme has its strength if there is a large amount of locality in the addressing of packets (eg. 50% of packets are addressed to other nodes on the local node-ring). As we address nodes at random in our simulation-study, this scheme is not optimal.

This scheme has

$$S_{nr} = (n + 1) * k^n \tag{7.8}$$

switches and the same number of active nodes as in equation 7.2. The subscript "nr" means the vertex has the extra $n$ode-$r$ing.

Preliminary simulation results indicate that method 'a' gives shorter latency and better performance than method 'b' (when there is no optimal addressing). This is mainly due to the extra time needed to traverse the extra switch and the fact that our addressing scheme implies no locality.

Alternatives Processor nodes do not *have* to be placed in the vertices, but we feel it is more optimal. An alternative could be to place the nodes along the edges, but this has some drawbacks. First of all: rout-

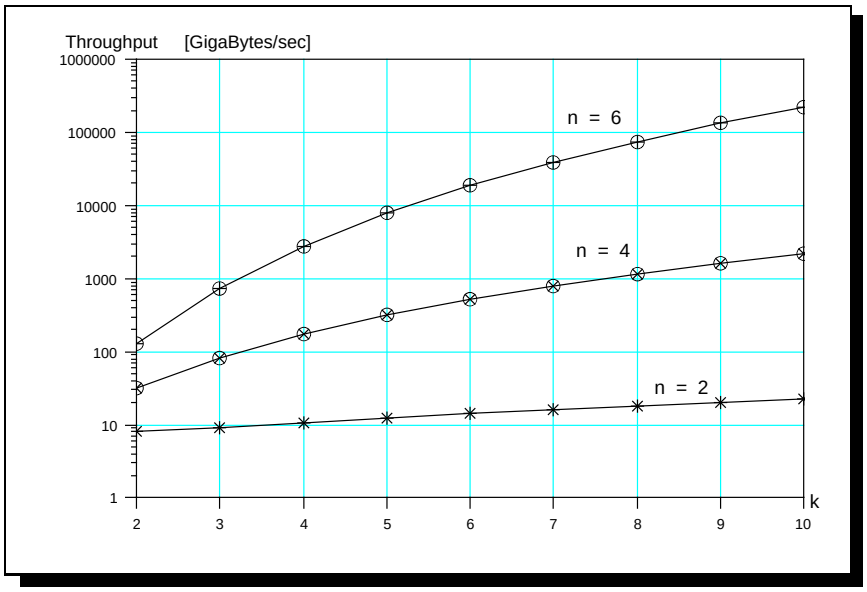


Figure 7.7: Throughput for k-ary n-cubes as defined by equation 7.7.

ing is much simpler if packets are addressed to the vertices. Second, the edge-rings are communication-links. It is important they do not become saturated.

7.1.3 Bidirectional edges

An alternative to edges is to use switches along the edges of the cube. An example is shown in figure 7.8 for a 3-ary 2-cube. An advantage with this scheme is that we effectively have a bidirectional link between the vertices. Thus the traversal of a long ring is avoided. This advantage is complicated by the fact that the delay through a switch is longer than the delay through a bypass-fifo (favored switch setting, see page 66). Thus this scheme is probably slower than the "edge"- scheme. It uses the same amount of switches (equation 7.1 if the nodes are in the corner-ring). We presume this type of topology is best suited for communicating with the nearest neighbors. If the traffic is directed at random it is probably inferior.

7.1.4 Using larger switches

All topologies above use a switch with 2 interfaces. It is conceivable that a large switch with 3 or 4 interfaces could be made. An example is shown in figure 7.9, with the four interfaces back-to-back, and a form of crossbar-interconnection pattern between them. An example of topologies with such switches is shown in [HulBot]. We early considered using such switches, and did some preliminary simulation studies with them.

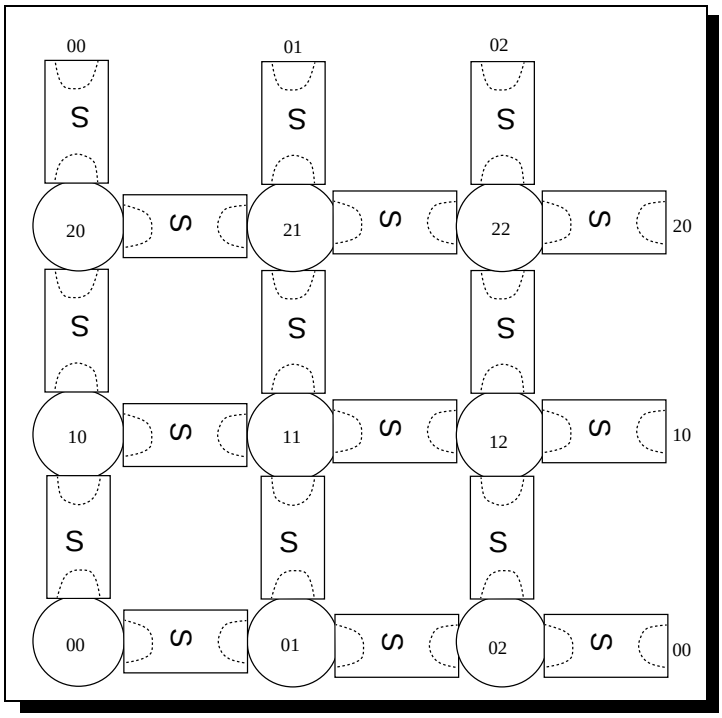


Figure 7.8: Example of a 3-ary 2-cube with bidirectional links between the corner-rings, effectively. A torus-connection is used (see figure 4.2). Nodes can be attached to the corner-rings as shown in figure 7.5a and 7.5b. The numbers indicate which switches are connected to which vertices, to illustrate a torus-connection.

We did not proceed mainly because we felt 2-port switches are probably more likely to be constructed in the nearer future [2].

They are therefore more interesting. It is obvious though, that a well constructed 4-port switch will have much larger connectivity, thus giving both smaller latency and larger throughput.

7.2 Other topologies

In this section we have only seen how k-ary n-cubes can be implemented with SCI-rings. Synthesizing networks with SCI-rings are only limited by the imagination. k-ary n-cubes represent just some of the possibilities. As mentioned previously, most of the classical topologies can be implemented with SCI-rings. In [JohnGood], for example, there is an analysis of multistage networks using SCI-rings.

[2] We may in fact be wrong on this point. There are studies under way at Dolphin SCI Technology, and at SINTEF–SI, to consider SCI-switches which connect more than two rings.

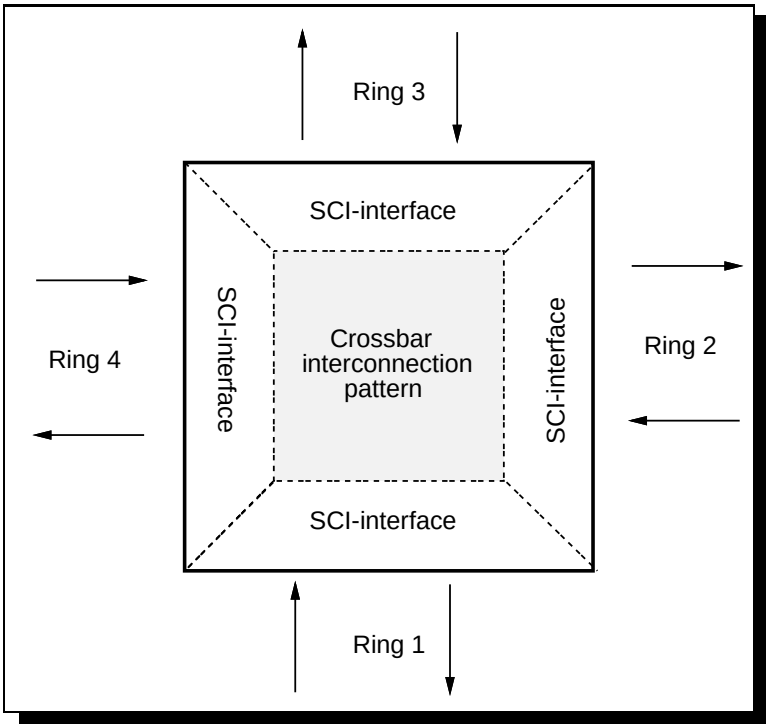


Figure 7.9: Using larger switches: a switch connecting 4 rings.

8

Construction of the simulator

In the previous chapter (specially section 7.1.2) we explained how we could connect multiple SCI-rings to form large networks. In this chapter we are going to explain the construction of the simulator, and the various modules and algorithms.

First a presentation of the sections in this chapter. The various modules of the program are presented in section 8.1. These modules represent physical parts (nodes and switches) and packets. Figure 8.1 shows most of these modules and their relationship. Section 8.4 - 8.5 explains important characteristics like bandwidth arbitration and packet-routing. In section 8.7 the production of statistical output is explained.

Our simulator simulates `nodes` [1] sending packets to each other through an interconnection network. Packets are addressed randomly. These `nodes` have both requester– and responder– qualities (see chapter 5). They could be processors, memories, or a combination. They produce requests and response-packets. Response-packets are produced in reaction to received request-packets. Request-packets are produced at random intervals. The length of these intervals determines the load put into the interconnection network. This is explained further in section 8.6. We simulate k-ary n-cube networks that are made up of multiple rings connected together by `switches`. Section 8.8 takes up the subject of how `node`- and `switch`-objects are connected together to make these networks.

Very roughly our simulator works in the following way: it simulates k-ary n-cubes made up of SCI-rings (as presented in section 7.1.2). It reads a "topology-file" (see section D.4). This file describes the network. Thus it tells the simulator how nodes and switches are connected together. When the network then is constructed the simulation is started, running for a specified number of clock-cycles. During the simulation various counters is incremented. When the simulation is complete, it reads these counters, and from them writes various statistical data to a "result-file".

[1] Most program-names will be written in a "typewriter-font".

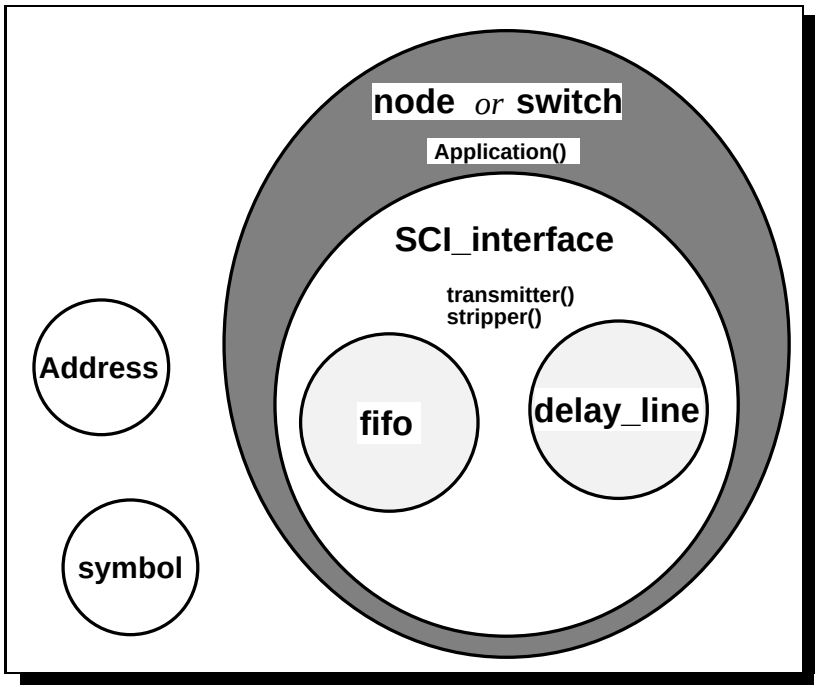


Figure 8.1: The relationship of major objects and functions making up the program. `node`-objects and `switch`-objects are made up of one or two `SCI_interface`-objects plus an instance of the function `Application()`. A `SCI_interface` is in turn made up of `fifo`-objects and `delay_line`-objects plus an instance of the functions `stripper()` and `transmitter()`. In addition we have `Address`-objects and `symbol`-objects to make packets. Figure 8.2 and 8.3 show example of their use.

Our simulation program implemented using the programming language of C++. C++ is an extension of C with strong use of typing and object orientation. It has a class and sub-class concept like the one used in the programming language Simula [Dahl et al].

Object-oriented programming makes it very convenient to program each of the small logical entities of an SCI-interface as a class. All the objects of a class have their own private data. Procedures operating on this data is provided as a user interface to the class. Using these classes as building blocks simplifies the construction of the larger classes (figure 8.1). This reason is in general the major objective for using an object-oriented language. This results in a hierarchy of classes on top of each other. This is explained in detail in section 8.1.

8.1 Classes and functions

In this section we present the major data-structures in our simulator. Figure 8.1 displays all the major modules of the program.

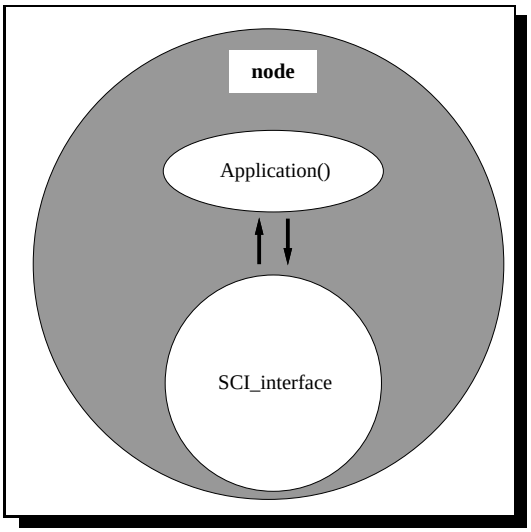


Figure 8.2: Rough overview of a `node`.

On the top levels there are the `node`-class and the `switch`-class. Each representing the complete nodes and switches . They are connected together and thus form the network topology. The `node`-class uses an instance of the function `Application()`, to simulate the behavior of the processor/memory part of a node. In the `switch`-class the `Application()` function performs the necessary forwarding of packets between each of the input- and output-ports.

To perform the lower level tasks representing the SCI-protocol, the `node` and `switch` - classes use the `SCI_interface`-class.

The `node` and `switch` - classes uses respectively 1 and 2 instances of the `SCI_interface`-class. (see section 8.1.1 and 8.1.2). The receiving and transmitting parts of an SCI-node are represented as instances of the functions `stripper` and `transmitter` respectively.

Other parts of the SCI-interface are modeled as classes simulating the physical fifos and delays. The `fifo`-class implements a general fifo and contains several functions allowing the most usual fifo-operations to be carried out. The `delay_line`-classes are used to simulate the physical delay of devices and the transmission medium. It is simply an array of symbols, shifted one step each clock-cycle.

Packets are made up of the `symbol`-class and the `address`-class. The `symbol`-class represents the symbols being sent between the nodes on the links and from which the packets are made up of. It does not contain the flag and clock line (see section 5.3.2) because the simulation does not take into consideration skew and noise on the physical links. Instead the `symbol`-class has internal flags from which a node logically can differ between the various packet types and idle symbols on the link. The `symbol`-class uses internally the `address`-class to represent the addresses (those symbols that present the address-fields). This makes

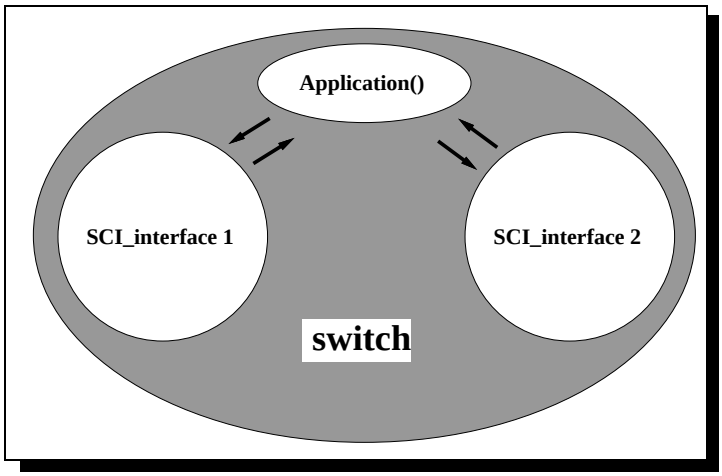


Figure 8.3: Rough overview of a `switch`.

it easier to have dynamic address fields, suiting the various topologies (this is explained further in section 8.1.11).

8.1.1 The `nodes`

The `nodes` are active nodes (e.g. processors) sending packets to each other. I.e. they generate traffic. The `node`-objects are simply made up of an `SCI_interface`-object and an instance of the `Application()`-function. The `Application()` generates requests and receives responses. Figure 8.2 gives a rough overview over the `nodes`.

8.1.2 The `switches`

The `switch`-objects are made up of two `SCI_interface`-objects put back-to-back and an instance of the `Application()`-function. The `Application()` is responsible for moving packets between the 2 interfaces. Figure 8.3 gives a rough overview of a switch. Note that the `SCI_interface` in a switch is slightly different from the `SCI_interface` in a node. There are no input-fifos. Also, to model the physical delay that symbols use when passing through a switch we have a switch-delay, an object of class `delay_line`, between the interfaces. See figure 8.5 for a more detailed drawing of a `switch`.

4 Switching strategies In reality we have made *four* simulators, but most of the code in them is identical. The distinction is the type of bridge strategy used. We vary the switching technique used in the switches and whether the switches have an extra buffer or not. The active nodes in each simulator behave alike.

Store & forward A k-ary n-cube where the switches use store & forward switching (see section 3.3).

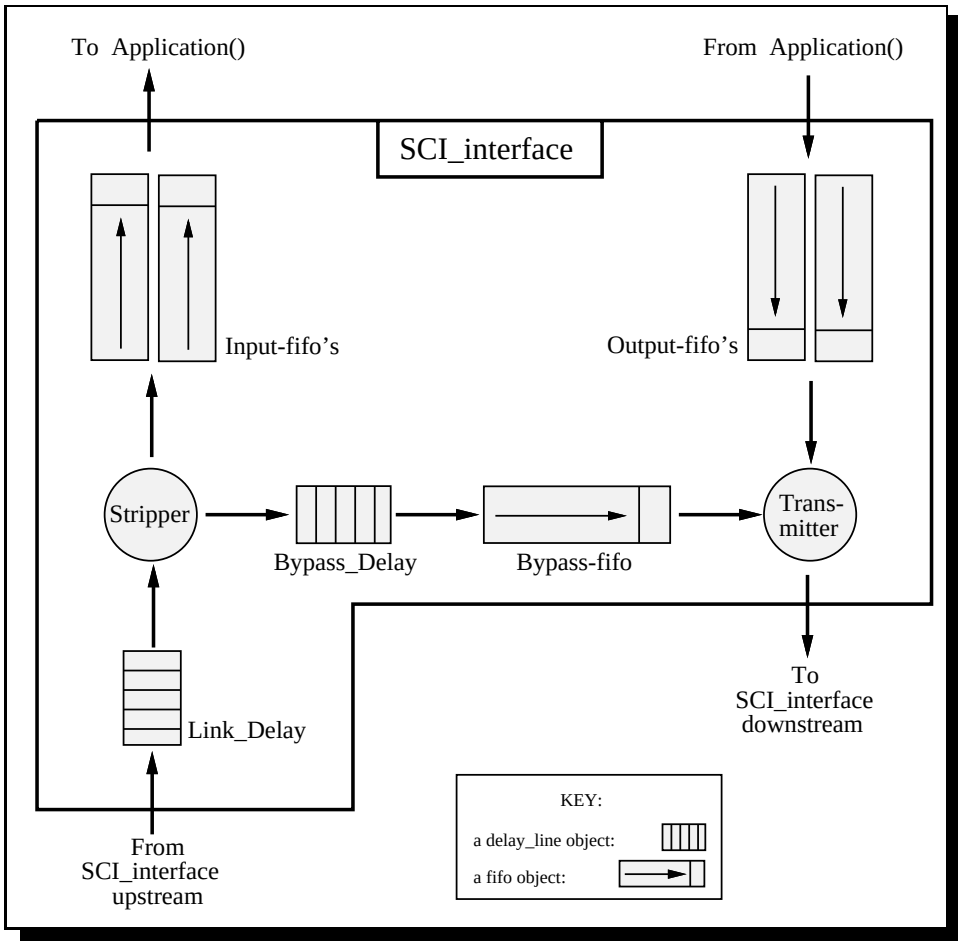


Figure 8.4: The objects that make up an `SCI_interface` in a `node`.

Virtual-cut-through A k-ary n-cube where the switches use virtual-cut-through switching (see section 3.3).

Store & forward with extra buffer A k-ary n-cube where the switches use store & forward switching and have an extra pair of buffers per interface. Thus the switches are like in figure 8.5 on page 81, except that they have an additional pair of output-fifo in each interface. Thus switches may have 2 outstanding packet-requests and packet-responses. We suspect that such a choice will reduce conflicts in the input-buffers, thus reducing retransmissions.

Virtual-cut-through with extra buffer A k-ary n-cube where the switches use virtual-cut-through switching and have an extra pair of buffers per interface.

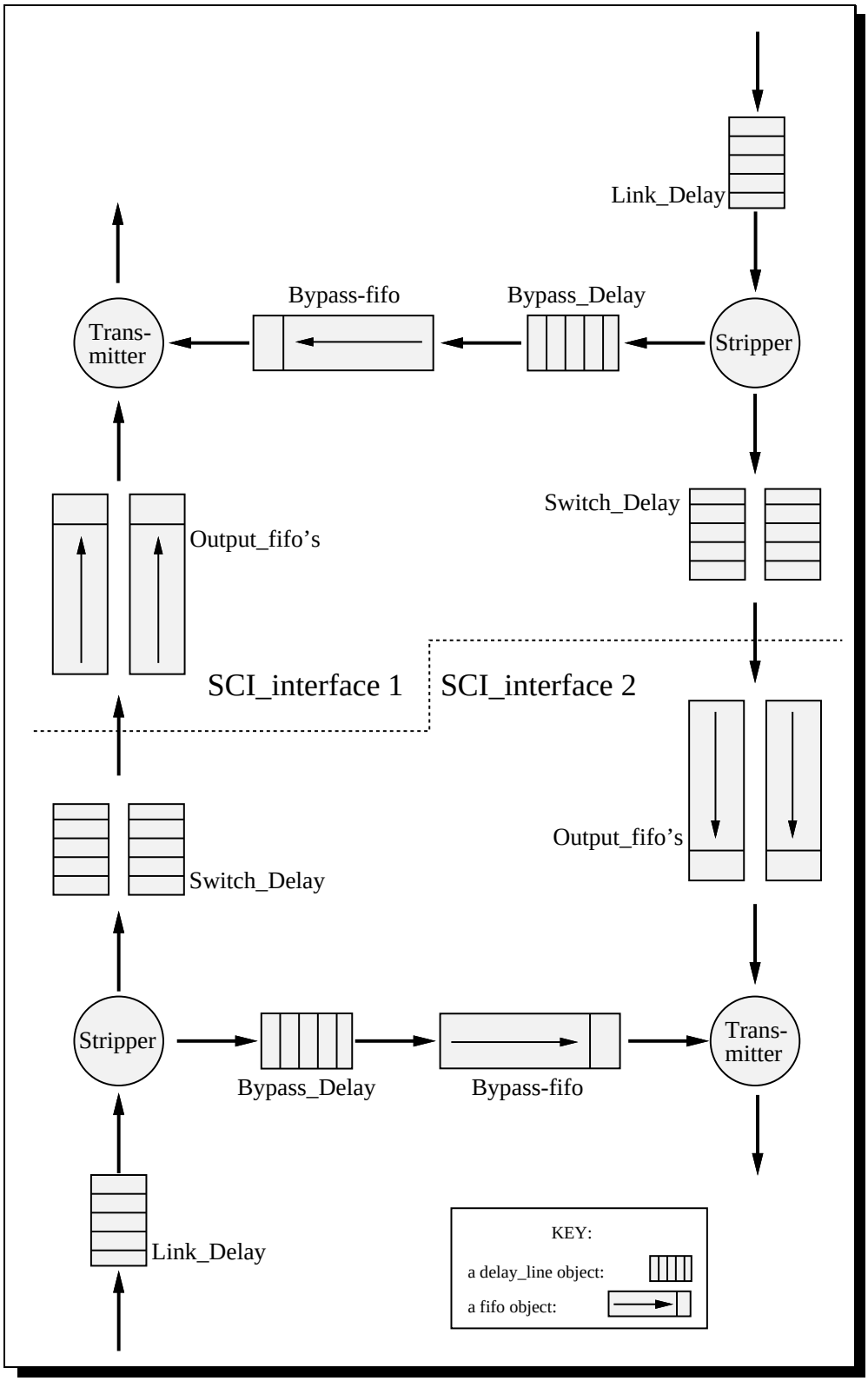


Figure 8.5: The objects that a `switch` is constructed of. As in `nodes`, there are 2 output-fifo's: one request-output-fifo and one response-output-fifo, but no input-fifos.

8.1.3 The `SCI_interface`

Nodes and switches are as we have seen in figure 8.2 and 8.3 made up of interfaces. Let us now see how they, in turn, are constructed.

An `SCI_interface` receives packets from the `Application()` to transmit on the ring and also receives packets that are given to the `Application()`. In addition the `SCI_interface` must route packets not addressed to it further on the ring.

An `SCI_interface` in a `node` (see figure 8.4 for reference) is made up of the following objects and functions:

- Two input-`fifo`s of class `fifo`. One input-`fifo` for incoming request-packets and one for incoming response-packets. The `fifo`-class is described in section 8.1.7.
- Two output-`fifo`s of class `fifo`. One output-`fifo` for outgoing request-packets and one for outgoing response-packets.
- A bypass-`fifo` of class `fifo` . It moves symbols not addressed to itself from the input-link to the output-link.
- A `link_delay` of class `delay_line` to model the physical delay on the links. It is made up of a simple array of length `LINK_DELAY` to simulate the delay of a wire. Symbols are shifted one step up the link (array) each clock-cycle. The `delay_line`-class is described further in section 8.1.8.

  The `link_delay` is referenced by the `SCI_interface` upstream, this is how the `SCI_interfaces` are connected together into rings (see section 8.8).
- A `bypass_delay` of class `delay_line` to model the physical delay through the bypass-fifo.
- An instance of the `stripper()`-function. The `stripper()`-class is described in section 8.1.6.
- An instance of the `transmitter()`-function. The `transmitter()`-class is described in section 8.1.5.
- In order to have a reference to the node downstream each `SCI_interface` has a pointer to the input-link of the `SCI_interface` downstream.

This section describes an `SCI_interface` in a `node`. An `SCI_interface` in a `switch` is almost identical, only there the input-fifos are removed. In addition an `SCI_interface` in a `switch` has a `Switch_Delay` of class `delay_line` to model the physical delay in the switches. See figure 8.5 for a detailed drawing of a switch.

Note that we have not bothered to add a scrubber-function (see section 5.1) in our simulation.

We assume that one clock-cycle takes 2 nanoseconds. This is the goal of the SCI-standard [IEEE-SCI]. Dolphin SCI Technology achieves about 3 nanoseconds for its first node-chips in ECL-technology.

8.1.4 The `Application`

The `Application` is a function that communicates with one or several `SCI_interfaces`. It moves packets into the output- fifos and takes packets from the input-fifos. It behaves in 2 ways depending on if the `SCI_interface` being in a `node` or in a `switch`.

8.1.4.1 The `Application` in a `node`

If the `SCI_interface` is in a processor-`node` the `Application` is responsible for the following:

- Consuming packets. Whenever there is a new packet in one of the input-fifos, the `Application` removes it from the fifos.
- Producing packets. At random intervals the `Application` puts packets into the output-fifos. The loading of the interconnection network is thus varied by changing the next time the output-fifos will be filled. This is discussed further in section 8.6. These packets are addressed randomly to any of the other active nodes in the network.

  As the symbols in the packets pass through various parts of the network, they are referenced by various objects. There is no copying of symbol-objects, all are only referenced by different parts of the network.

8.1.4.2 The `Application` in a `switch`

In a `switch` the `Application` is responsible for moving a packet from one `SCI_interface` to the other `SCI_interface` in a `switch`. It moves symbols from the `switch_delay` to the other interface's output-fifos. See figure 8.5. Also, here the field `echo_return` is set to the address of the destination interface in the switch. Then the receiver on the other ring knows who to send an echo to.

8.1.5 The `transmitter`

The main objective of the `transmitter()` is to send symbols from the SCI-interface to the SCI-interface downstream. It has to decide whether to send symbols from one of the following fifos: the `req_out_fifo`, the `resp_out_fifo` or the `bypass_fifo`. It can only send symbols from the `req_out_fifo` or the `resp_out_fifo` if the `bypass_fifo` is empty and the SCI-interface has an idle-symbol available with the go-bit set. If both the request-out-buffer and the response-out-buffer have a packet, they take turns to transmit in a round-robin-fashion.

The `transmitter()` is a state-machine with 5 states. Their relationship is shown in figure 8.6. The states:

IDLE Sending an idle-symbol. It leaves this state if the `bypass_fifo` or one of the output-fifos has a packet to send.

REQ_OUT Sending a symbol from the `req_out_fifo`

RESP_OUT Sending a symbol from the `resp_out_fifo`.

BYPASS Sending a symbol from the `bypass_fifo`.

LAST_SYMB Sending the last symbol in a packet from the `bypass_fifo`, `req_out_fifo`, or `resp_out_fifo`. A transitional state, the `transmitter()` is never here more than a clock-cycle. This state is to ensure that there is at least one idle between packets.

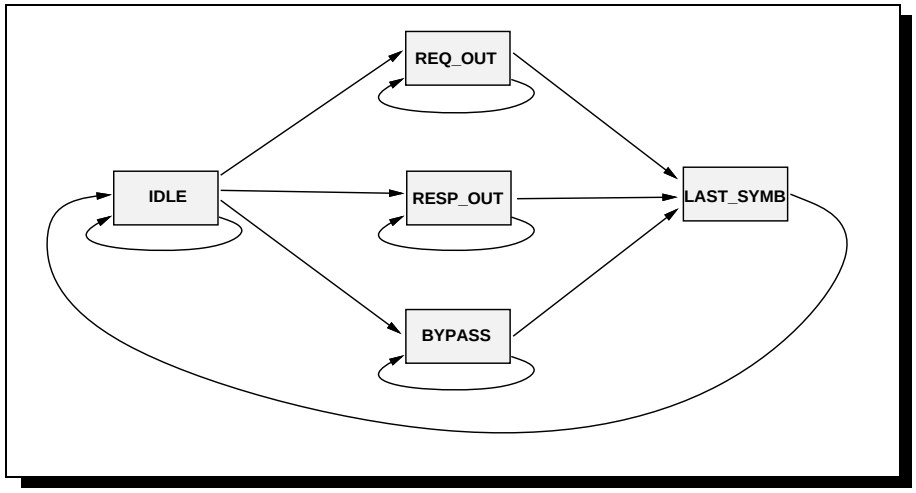


Figure 8.6: State diagram of transmitter.

8.1.6 The `stripper`

The main objective of the `stripper()` is to receive symbols from the `link_delay` (connected to the transmitter of the SCI-interface upstream). It has to decode addresses, and route symbols (packets) either to the `req_in_fifo` (for requests) or the `resp_in_fifo` (for responses) if the packet is addressed to itself, or to the `bypass_fifo` if the packet is for another node further downstream. The `stripper()` is a state-machine with the following states:

IDLE In this state the `stripper()` receives incoming idle-symbols. No idle-symbols enter the bypass-fifo, in our simulator they are deleted in the `stripper()`. Before deletion the value of the go-bit is registered.

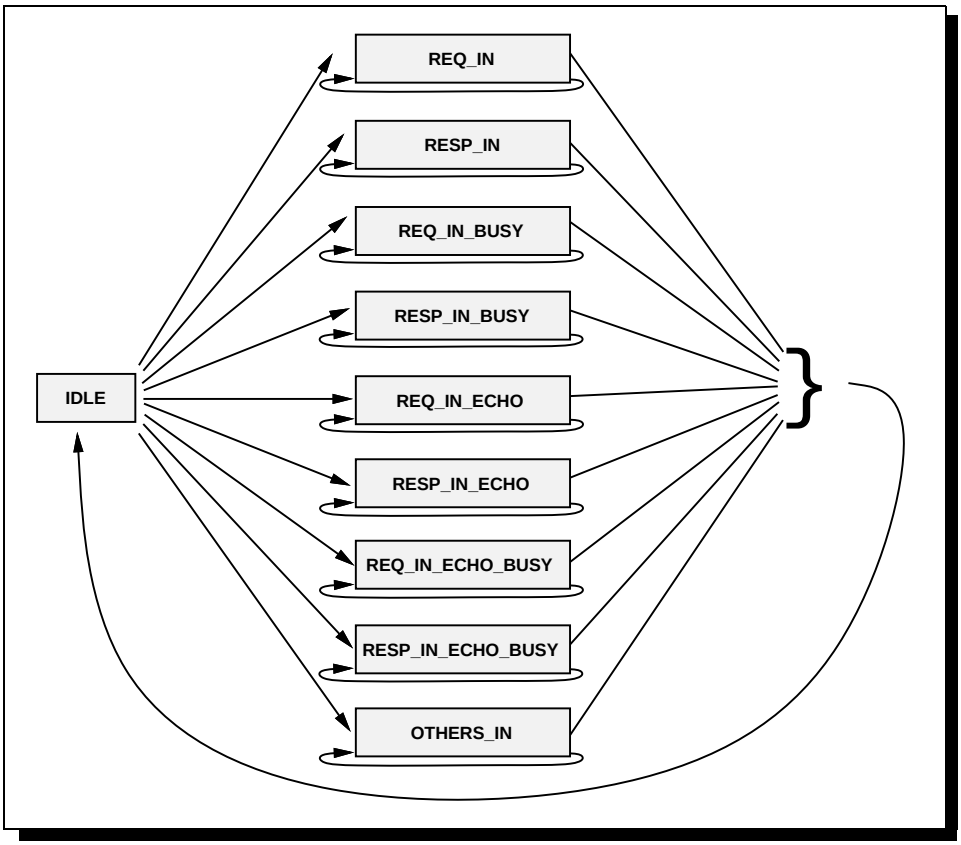


Figure 8.7: State diagram of stripper.

REQ_IN In this state the `stripper()` receives a symbol and puts it in the `req_in_fifo`. If it is the last symbol in a packet (and the packet was received completely) an echo is sent back to the (local) sender.

RESP_IN Just like the previous state, except that the symbol is bound for the `resp_in_fifo`.

REQ_IN_BUSY Similar to the state `REQ_IN`, except that the `req_in_fifo` was busy. An echo is sent back to the (local) sender, informing it about the failure.

RESP_IN_BUSY Just like the previous state, except that the symbol is bound for the `resp_in_fifo`.

REQ_IN_ECHO The incoming symbol is part of an echo-packet, acknowledging the safe arrival of its' request packet at another node. The packet in the `req_out_fifo` is released.

RESP_IN_ECHO Just like the previous state, except that a response-packet is acknowledged.

REQ_IN_ECHO_BUSY The incoming symbol is part of an echo-packet informing the node that a request packet sent to another node was

discarded because of busy buffers. The node must therefore retransmit the packet.

RESP_IN_ECHO_BUSY Just like the previous state, except that it was a response packet that was lost.

OTHERS_IN The incoming symbol is part of a packet addressed to some other node on the ring. The symbol is put into the `bypass_delay` and then into the `bypass_fifo`.

Busy input buffers What happens if a packet is sent to an input-buffer that turns out to be busy? If a buffer is busy this is discovered as the beginning of packet is entering the interface. The packet continues to enter, but its contents is ignored/destroyed. When the end of the packet is received, an echo is constructed. This echo-packet is addressed to the original packet-sender, and informs it of the failure. This node will then immediately resend the packet (if bandwidth is available). We have no limit to the number of retries in our simulators.

8.1.7 The class `fifo`

An `SCI_interface` has 3 or 5 fifos. An interface in a switch has generally 3 fifos, interface in a processor-node has 5. A `fifo` is a first-in-first-out queue. The first symbol entering the queue is also the first symbol leaving the queue. One important note should be made. The fifos here are to be able to model the switching technique virtual-cut-through (see section 3.3). Thus if the fifo is empty, the symbol is to speed through the fifo as fast as physically possible. If the fifo is not empty, the symbol to be placed right behind the previous symbol that entered the fifo. This important property is always used in `fifo`-objects that are used as bypass-fifos. This property of virtual-cut-through is also used in the other fifos that are in an interface, if virtual-cut-through is simulated between rings. Input and output fifos in active nodes are always filled up completely before any action is taken.

The class `fifo` is simply a linked list – a chain – of symbol-objects. At the head and tail of the chain are "dummy"-objects. The chain contains the symbols that are in a fifo. The last symbol entering the fifo is placed at the end of the chain. The oldest symbol is at the start of the chain. Each clock cycle a symbol may leave and/or enter the fifo.

The `fifo`-objects have 3 states: `EMPTY`,`HALF_FULL`, and `FULL`. In the `EMPTY` state the fifo contains just the head and tail objects, no other symbols. See figure 8.8. In the `HALF_FULL` state the fifo contains symbols, but less than `max_in_fifo`. See figure 8.9. In the `FULL` state the fifo contains exactly `max_in_fifo` objects.

Four functions put and take symbols from the `fifo`-objects:

`in_symbol()` puts a single symbol into a fifo. Typically called by the `stripper()` when it wishes to put a symbol into one of the input-fifos or the bypass-fifo.

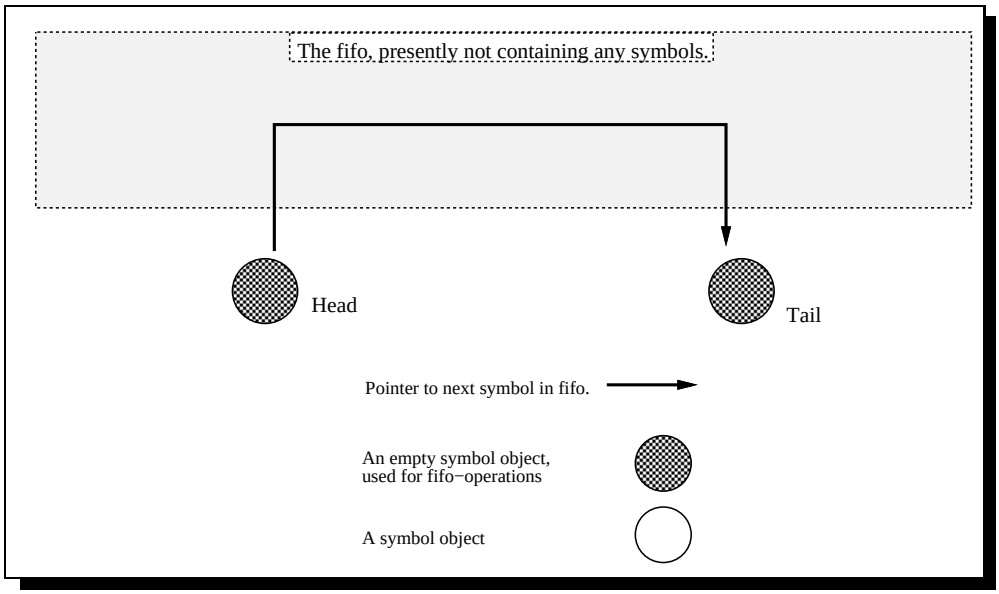


Figure 8.8: A `fifo` in state `EMPTY`.

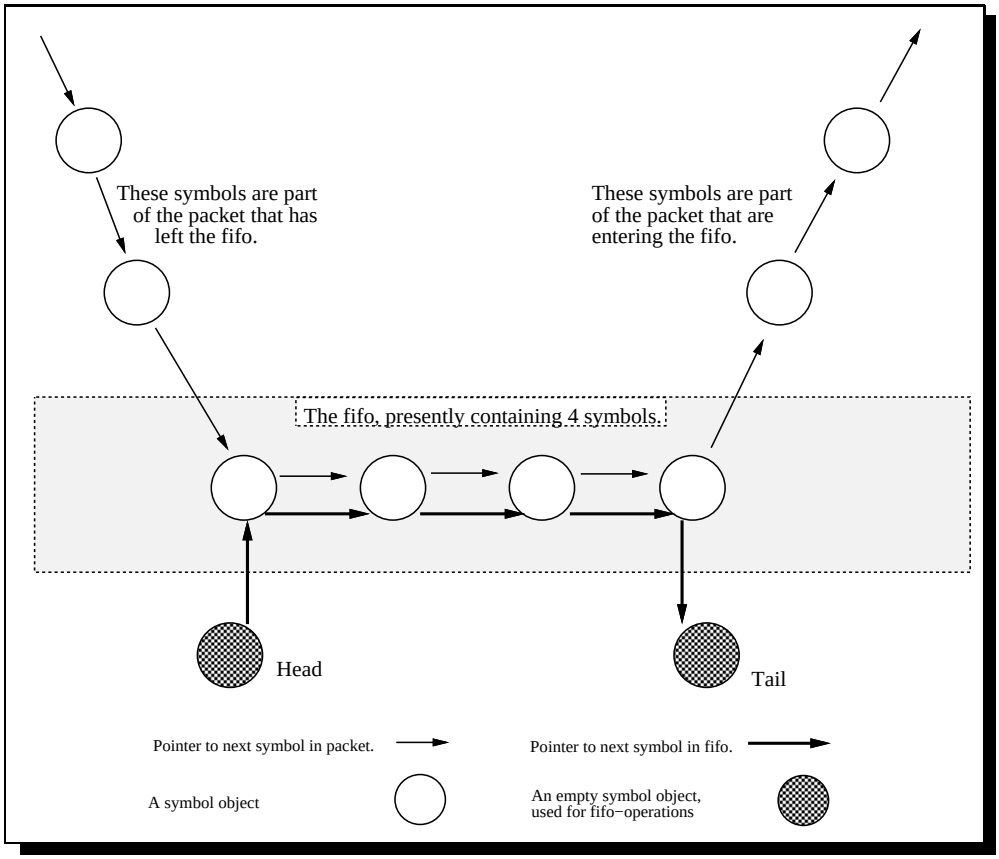


Figure 8.9: A `fifo` in state `HALF_FULL`.

`out_symbol()` moves single symbol out of a fifo. Typically called by the `transmitter()` when it wishes to move a symbol out of one of the output-fifos.

`in_packet()` moves a complete packet into a fifo.

`out_packet()` moves a complete packet out of a fifo.

8.1.8 The class `delay_line`

We are simulating physical devices and physical links. Signals do not pass through them infinitely fast. The class `delay_line` models physical delays. It is implemented as an array (a table) of pointers to symbol-objects. The length of the array depends on the length of the delay to be modeled. In a `delay_line`-object the symbol must pass through each entry in the `delay_line`. An example: a `delay_line` of length 3 means a delay of 3 clock-cycles (6 nano-seconds).

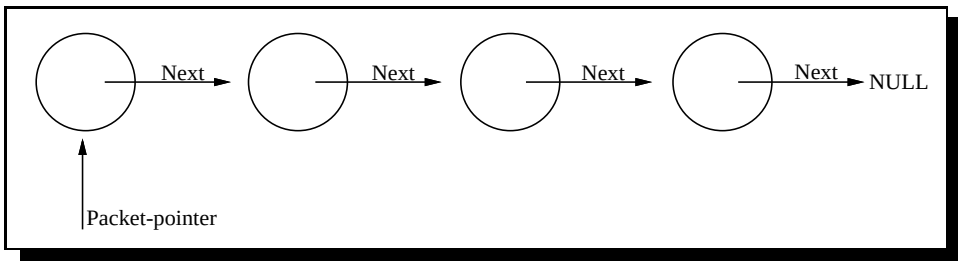


Figure 8.10: A simple packet of 4 symbols. Each packet has a pointer to the next symbol in the packet.

8.1.9 Packets

Packets in SCI are made up of symbol-objects. To make a packet all the symbols in it are linked together into a chained list, as shown in figure 8.10. To access a packet in the program one simply accesses the first symbol in the packet.

In our simulator we use the following types of packets:

Request-packets are made up of 40 symbols. 7 symbols are regarded as header-data and the last symbol is a CRC-code. Thus the remaining 32 symbols (= 64 bytes) are real data. This is the "`move64`"-packet of SCI (see section 5.3.4).

Response-packets are, for simplicity, like the request-packets. They are sent in response to incoming request-packets.

Echo-packets are made up of 4 symbols (= 8 bytes). They are acknowledgments of packets addressed to the node. They are generated by the `stripper()` when packets addressed to the node enters. If the packet is received completely, an ok-acknowledgment is sent. If the

input-buffer which the packet wishes to enter is busy, an echo informing the (local) sender of the failure is sent. The sender must then try again.

Idle-symbols are strictly not a packet, just a single symbol. They have several purposes, most of which are ignored in this simulation. The one thing about idle-symbols we do not ignore are the go-bits in them. They are used to regulate the bandwidth on the ring.

8.1.10 Symbols

The 16 bits sent on a link simultaneously constitute a symbol. All packets are constructed of `symbol`-objects. Below are listed some of the major attributes of `symbol`-objects. Note that most of these attributes are void when the symbol is not part of a packet, but an idle symbol.

The `flag` attribute is used to mark what type of packet the symbol belongs to: a request or a response packet, a request-echo or a response-echo packet, or a request-echo-busy or a response-echo-busy packet, or an idle. The `flag` is also used to mark packet-boundaries.

The `target_address` is an object of the class `Address`. It contains the coordinates (the position) of the target-node.

The `source_address` Similarly, the `source_address` is an object of the class `Address`. It contains the coordinates of the sender-node.

The `echo_return_address` is also an object of the class `Address`.

A node or bridge that receives a packet must know whom to send an echo to. The `echo_return_address`-field contains the address of the local sender of the packet to send the echo to. The `echo_return_address` is a variable field that changes value as the packet moves from ring to ring.

The `next-pointer` As shown in figure 8.10 a packet is a linked list of symbol-objects. The next pointer is a reference to the next symbol in the packet.

The `fifo-next-pointer` Is also a reference to the next-symbol in a packet, but it is only used to handle the symbols when it is in a fifo.

The `go_bit` This attribute only has meaning for idle-symbols. The go-bit is used to ensure fair use of bandwidth. See section 8.4.

`send_time` All symbols leaving their sender are stamped with the time they left the node. This value is later used to calculate the transmission-time.

`switches_passed` Each time a send-packet enters a switch this counter is incremented by 1. See section 8.7 for the use of this variable.

`bypasses_passed` Each time a send-packet enters a bypass-fifo this counter is incremented by 1. See section 8.7 for the use of this variable.

8.1.11 The class `Address`

All SCI-interfaces have an address. Instead of having one absolute address, we have divided the address into multiple parts. So, addresses are not large numbers like "12579". Instead we have something like "5, 11, 4, 7". Generally, addresses have the following form:

$$A_0, A_1, A_2, \cdots A_{n-1}, A_n$$

A_0 to A_{n-1} represents the interface's coordinates in a k-ary n-cube and varies from 0 to $k-1$. A_n represents the interface's local address in the vertex. An example for a 3-dimensional cube ($n = 3$):

Name of Field:	A_0: "x-coord."	A_1: "y-coord."	A_2: "z-coord."	A_3: "local"
Example address:	5	11	4	7

When we refer to addresses later in the text, we will usually avoid the form $A_0, A_1, A_2, \cdots A_{n-1}, A_n$. Instead we will usually refer to the x-, y, z-coordinates, and the local part. This is a less general form, but it hopefully increases readability. In the text we use 2- or 3-dimensional examples for simplicity. Remember though, that the simulator works for any dimension we wish to simulate. [2]

The advantage with this scheme is that it is relatively simple to route (section 8.5). Also no tables are needed in the nodes and switches. Each interface only has to store its own address (16 bits), the stripper only needs some comparator logic. In addition the switches must have a "dimension-index", so that the interface knows which dimension it is connected to. This requires $\log_2 n$ bits (typically 2-3 bits). This is relatively little information to check each time a packet enters an SCI-interface.

In a sense this is a form of source-routing. Source-routing usually implies that the message contains the explicit path which the message must traverse. This path is often specified as a list the links the message traverses. In our scheme we instead specify target-planes for the packet.

In the above example the packet must first go to the yz-plane with $x = 5$, then the xz-plane with $y = 11$, and finally the xy-plane with $z = 4$.

8.2 Various development problems

We had many problems to solve when we constructed the simulator. We will here only mention some of the major ones:

What kind of simulation-model ? There are several classification-schemes for simulation [Shannon]. We early decided on a simple discrete time-step model. All `nodes` and `switches` in the network execute the SCI-protocol for one clock-cycle in round-robin order. Then the

[2] Currently an upper limit of 11 dimensions, but that can be raised, if desirable.

time is increased on cycle and the protocol is again executed in all the `nodes` and `switches`. This continues until the desired number of cycles has been executed.

At first we used a coroutine-library [3] to achieve this. Later we changed the simulator a bit; instead of using a coroutine-library we managed with a single program-loop that executes the `nodes` and `switches` in a round-robin-fashion. This change made the program much more portable.

What programming language should we use? And which compiler? This decision was in practice related to the above problem. We decided finally to program in the language C++. We chose the compiler "g++" from GNU. See appendix B for more details.

Bandwidth-arbitration We decided to implement the *fair* bandwidth arbitration scheme as defined in [IEEE-SCI]. Our interpretation of it is described in section 8.4. Dolphin SCI Technology has also made this choice for their first node chip implementation [Alnæs et al].

Large numbers Our simulator was to generate statistics about networks. To produce these statistics numerous counters are needed. The values some of these counters reach can be exceptionally large. Ordinary types in C++ (and in C) are not large enough. We studied various available libraries, but decided not to use them.

Fortunately the solution was extremely simple, once we became aware of the type " long double". This type was actually defined in our chosen compiler (and most other standard C and C++ compilers). Variables declared as "long double" allocates 128 bits, 104 bits for the integer part, and 24 for the fraction part. This was more than enough for our purpose.

Acknowledgments All acknowledgments in SCI are done locally. No echo-packets traverse more than one ring. This is referred to as a remote transaction in [IEEE-SCI], the switch being the agent. See figure 5.7.

But how does a node know whom to send an echo to? If the network is only a single ring, then the source-address field holds the necessary information. But if the network has multiple rings it is not so simple. The problem is when the node wishing to send an echo-packet does not know the address of the switch (on the local ring) that is to have the echo-packet. This topic is not discussed in [IEEE-SCI]. We therefore had to decide on a solution ourselves.

There are 2 possible schemes, as we see it:

- The send-packet has an extra field containing the address of the switch that is supposed to have the echo-packet. This field must then change value each time just before it enters a new ring. The value is the identity of the interface connected to the new ring. This is the method we have used in our simulation-model. In SCI

[3] See appendix B for an explanation on coroutines.

provisions are made for one or two extra header-fields, if desired. See the "extended header", *ext* in section 5.3.5.1. So it is not unreasonable to assume there is an extra field available.

- Use the source-field of the packet and send the echo-packet in the direction of the source-node (assuming k-ary n-cubes). That puts restrictions on the topology and the routing algorithm. Every packet from a node A to a node B must then traverse the same path. Only then would an echo be sent to the correct switch on the local ring. Thus alternative paths between two nodes are not permissible. Assuming fixed paths and our coordinate address-scheme one should be able to construct a routing algorithm, which does not need extra fields in the packet. When refering to coordinate address-scheme we mean both the partition of the address-field and the indexing of the switches (ports) according to the dimension they belong to.

Variation in the network load How do we vary the loading of the network? For a long time we were uncertain about the best strategy for this, before we finally settled on the scheme described in section 8.6.

8.3 Priority

For simplicity, we have chosen not to implement any priority scheme. This is not unreasonable. Also Dolphin SCI Technology has decided to ignore implementing priority in their first implementation of the SCI node chips [Alnæs et al].

8.4 Bandwidth arbitration on the rings

The go-bits in the idle-symbols act like tokens, to ensure fair use of the links. The nodes on a ring must first send an idle-symbol with a set go-bit before it can send its own packet. In addition their bypass-fifo must be empty. These two criteria ensure fair bandwidth arbitration among the nodes on a ring. Nonetheless, the process of getting permission to send a packet is somewhat complex. This is to guarantee that there is always enough go-bits for stable operation. We have chosen to divide the process into several states (look at figure 8.11 as you read this):

PASS In this state the node is letting others nodes packets pass through its bypass-fifo. Information in idle-symbols passes through unchanged until it wishes to send a packet of its own. Then, if its bypass-fifo is occupied, it starts to set the go-bits in the idle-symbols to zero.

SEND_OWN When the node receives a set go-bit and has emptied its bypass-fifo, it leaves the state `PASS` and enters state `SEND_OWN`. During this phase the node sends its packet. Incoming go-bits are

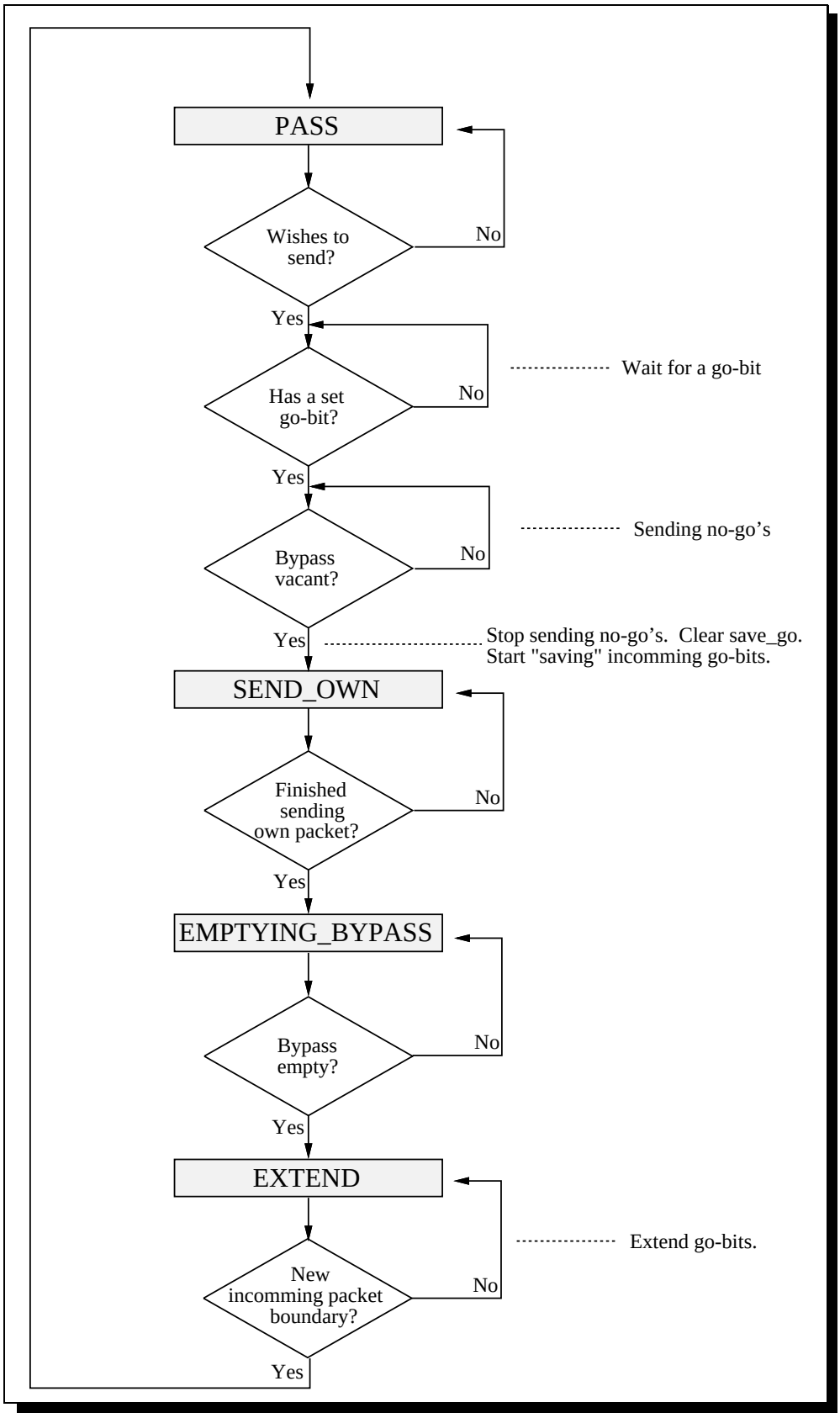


Figure 8.11: State diagram showing the bandwidth arbitration algorithm in an interface.

"saved" (ORed) in the variable `save_go` for later use in the following manner:

```
save_go   =  incoming_go || save_go;
```

EMPTYING_BYPASS When the node has completed sending its packet, it leaves the state `SEND_OWN` and enters state `EMPTYING_BYPASS`. It continues to save incoming go-bits as described above.

When leaving this state an idle-symbol will be sent. If `save_go` is '1', the symbol's go-bit is set to '1', otherwise it is set to the value of the last go-bit that entered the interface. Thus no *new* no-go bits are generated. Only when nodes downstream generated a '0', is a '0' sent on.

EXTEND When the bypass-fifo is emptied the node leaves the state `EMPTYING_BYPASS` and enters the state `EXTEND`. During this phase so-called go-bit-extension takes place. Effectively all go-bits leaving the node are set to one until the next packet boundary. This ensures that there is enough go-bits to go around, with value 1. This guarantees stable operation. Then the node returns to the state `PASS`.

8.5 Routing

We use a distributed routing algorithm in our model.

We divide the description of the routing into 2 parts: routing on a global scale, and routing on a local scale. In the global routing we describe how packets are routed from one part of the k-ary n-cube to another part of the k-ary n-cube. In the local routing description we describe in detail the routing decision done by routing-elements in the vertices in the k-ary n-cube.

Routing on a global scale The routing of packets in our k-ary n-cubes is relatively straightforward. It is best shown with an example (Adapted from [Leighton]). We presume here a 3-cube with x, y, and z representing the 3 dimensions. Also recall that in our corner-ring simulation a torus-connection is used. It might be helpful to look at figure 8.14 and 8.15 as you read this.

1. Route the packet along the sender's x-ring, until the packet reaches a switch whose x-value is the same as the x-value in the packet. Exit at the other output port of the switch (e. g. interface marked as '3' in figure 8.14). The packet is now in the correct yz-plane.

2. Enter the closest y-ring (through interface '4' and '7' in figure 8.14). Route the packet along the y-ring until it reaches a switch whose y-value is the same as the y-value in the packet. Exit at the other output port of the switch (e. g. interface marked as '4' in figure 8.14).

3. Enter the z-ring (through interface '5' and '8' in figure 8.14). Route the packet along the z-ring until it reaches a switch whose z-value is the same as the z-value in the packet. Exit at the other output port of the switch (e. g. interface marked as '5' in figure 8.14). The packet has now reached the destination-ring.

The packet must pass through 2 switches each time the packet switches dimension (occurs maximum $n - 1$ times).

Note that when a packet enters a corner-ring just to switch dimension, the packet traverses only one segment of the ring. Thus the majority of the traffic is not contesting for bandwidth.

A drawback with this scheme is that this routing algorithm is not "fault tolerant", it will not adapt to route differently in a faulty network. There are no multiple paths between two nodes. We sacrificed such a property for simplicity. Thus there is only one unique path between two nodes.

Routing on a local scale So far we have only presented routing on a global scale. What routing decisions are made by the `SCI_interfaces` in the switches? A vertex is constructed of multiple `SCI_interfaces`, but the routing decisions are not similar in all the `SCI_interfaces`. How is routing within a vertex? A vertex in a k-ary n-cube is typically made like the example shown in figure 8.14. We here describe in detail the routing decisions made by the various `SCI_interfaces` in a vertex. Use figure 8.14 as reference as you read this.

Criteria for removing the packet off the ring:

1. The nodes have a simple job: either the address in the packet is identical to its own address, otherwise just forward it.

2. The "inner" interfaces in the corner-ring (marked as '3', '4' and '5', in figure 8.14): If the "dimension-index" in the packet and interface are identical, then the packet is not removed from the corner-ring. Example: if the x–coordinate of the packet arriving in interface '3' (figure 8.14) is identical to the x–value of the vertex, then the packet continues to interface '4'. Interface '4' will then check the y–value of the packet. If the value is different from the vertex' y–value, the interface will take the packet off the ring. The packet will then enter a "y–ring" for further routing.

3. The "outer" interfaces in the corner-ring (marked as '6', '7' and '8', in figure 8.14) behave in a similar manner. A packet arriving in interface '6' on the "x–ring" will have its x–value checked. If the x–value is identical to the x–value of the vertex, the packet is removed from the x–ring and forwarded to the corner-ring.

4. If the scheme of figure 7.5b is used:

   The interface in the switch connected to the node-ring (marked as '1' in figure 8.16): If a packet arrives here with other address-coordinates than those of the vertex (different x–, y–, or z–value)

then obviously the packet is for a processor-node in another vertex. The packet must be taken off the "node-ring" and passed to the "corner-ring" for further routing.

5. If the scheme of figure 7.5b is used:

   The interface in the switch connected to the corner-ring (marked as '2' in figure 8.16). If the packet entering the interface has the correct x–, y–, and z–value, then the packet must be sent to the node-ring.

Alternatives At first we considered another routing scheme, based on *"interval labeling"*. Then each switch-output has an associated interval – a consecutive set of addresses. See [ShMayTho]. But we found that this scheme is rather inflexible for our purpose, when studying various topologies. On the other hand, interval labeling could be a good idea when constructing actual physical networks. This is because such a scheme will probably use little table-space in the switches.

Bidirectionality In our bidirectional simulation (the scheme of figure 7.8) the routing is somewhat modified from the one above. To fully utilize the fact that bidirectional links are used, the switches must calculate which direction is shortest along the current dimension. Shown in the following pseudo-code:

```
distance  =  target_index  - current_switch_index ;
if (distance < 0)   distance = distance + k ;
```

If then the `distance` is less than $k/2$ (rounded down to closest integer) it is shorter not to route in the default direction (e.g. not decreasing the x-value, but increasing it instead).

8.6 Variation in the network load

How do we vary the loading of the network? This loading is determined by the frequency of the active nodes outputting request-send-packets. We have in mind a multiprocessor SCI system. We assume then that it is cache misses that make the nodes generate requests to memory in another node, thus generating network load.

In [PatHen] a model of cache behavior is presented. The cache behavior attributes are divided into three parts. Compulsory, Capacity and Conflict ("The three C's").

- Compulsory. The misses caused by the first access to a block of memory. In other words a block not not used before is called. This is called "cold-start misses".
- Capacity. These misses are occurring when the cache is to small to contain all the memory blocks needed by a program at the same time.

- Conflict. This is related to the block replacement strategy used in the cache. When using a set associative or direct mapped cache, several blocks of memory will map into the same place in the cache. This will result in cache-blocks to be discarded and inserted, when having a program referring frequently to memory locations with the same mapping-address.

The model presented is not perfect, but gives a general view of the memory demands of caches. Then the question arises: How do we represent this model (of the three C's) within our simulation model? The most obvious solution is to try to represent the loading as a statistical distribution. We have chosen to use the *normal*, *negative exponential*, and *uniform* distributions. It is not clear which of these distributions which model cache-behavior in the best way. To investigate these phenomena we have run simulations with each of these distributions, comparing the results. These results are presented in section 9.2. The conclusion drawn from the results is that the difference between the various distributions is very small. Therefore we will mostly use the uniform distribution.

Let us describe our implementation of the uniform distribution in more details:

How often should the nodes send a request? Should the active node send a new request immediately after a transaction is complete? We felt that this is unrealistic. Instead we let the active node wait a random interval before starting a new transaction. This random waiting is modeled with the j-parameter. j is the *maximum* number of cycles to wait before starting a new transaction. The active node draws a random number between j and the minimum value (10 cycles). If j is 15 the node draws a number between 10 and 15 and waits that many cycles before starting a new transaction. A small j means that the node starts a new transaction almost immediately after the previous one was completed. We simulate with the following j-values: 15, 500, 1000, 2000, 3000, 4000, and 7000. With j= 7000, an active node will on average wait 3500 clockcycles before initiating a new transaction.

Thus, the time from the return of a response until a new request is sent varies from 30 and up to 14000 nanoseconds, depending on the selected j-parameter.

How do we justify the selected j-values? We assume the active node has a 2-level cache with a hit-rate of 95% and 90% in the primary and secondary cache, respectively. The hit-time for the primary cache is 20 nanoseconds and for the secondary cache 80 nanoseconds. The time for the network to service a memory-request over the network is roughly 1000 nanoseconds. The average time between two memory-requests over the network [PatHen, Bugge et al] we then calculate to be 5800 nanoseconds. This corresponds to $j = 5800$. $j = 7000$ signifies very light load, then we measure near optimal transmission-times. j= 15 signifies

a very high load, perhaps a processor with a very small cache. Thus our choices of j-values seem reasonable.

These j-values may seem somewhat peculiar to present in the graphs in chapter 9. There we render the load-variation in what is more common in the literature: amount of requests per unit of time. In the graphs in chapter 9 we use the amount of Kilorequests per second when varying load. The above j-values then corresponds respectively to 40000, 2000, 1000, 500, 340, 250 and 140 Kilorequests/second.

This model of requests per second might seem a bit artificial. The amount of requests per second is of course limited by the physical resources available. In practice it is limited by the amount of buffering available in the active nodes. This is what restricts the maximum amount of un-responded requests (transactions) a node can have at a given time.

The p-parameter varies the number of outstanding requests (transactions) that an active node can have. We simulate for 1, 2, and 4 outstanding packet-requests. We expect the first implementation of SCI will have a single transaction at a time only.

8.7 Gathering statistics

To produce the graphs of chapter 9 the simulator must produce various parameters. A presentation of the basis for these parameters:

Average throughput on the links is measured by counting all the symbols (=2 bytes) leaving the `transmitter()`, except idle symbols. This is done in all the `SCI_interfaces`. This is then averaged at the end of the simulation.

The effective total system throughput is calculated in the following way: all received request- and response-packets are counted up for all the active nodes. Echo-packets are not counted. The overhead due to packet-headers is subtracted. Lastly the total number of bytes received is normalized to the length of the simulation. This leaves us the effective total throughput in bytes per second, for the whole system.

Latency The latency of a packet is measured from the time the sender puts the packet into its output buffer, until the packet is received completely in its destination-buffer. This is then averaged for all packets sent during the simulation.

The mean number of bypass-fifos passed by the packets Each packet has a counter that is incremented by 1 each time the packet enters a bypass-fifo. When the packet is received at the destination-node, a global counter is incremented by the packets' value. Note that this counter is also incremented if the packet must be resent due to busy input-buffers. At the end of simulation this global value is then normalized with respect to the total number of packets sent (not counting echo-packets).

The mean number of switches passed by the packets Each packet also has a counter that is incremented by 1 each time the packet enters a switch. This is similarly then normalized with respect to the total number of packets sent (not counting echo-packets).

Uncertainties in the results There are several factors that give rise to statistical uncertainties in the simulation results.

One factor is the length of the simulation-run. Currently we run and simulate 100 000 clock-cycles, that is 200μ seconds. Ideally we should simulate much longer, but even 200μ seconds take a long time to simulate. Simulating 200μ seconds for a 10-ary 2-cube (with 300 active nodes) takes roughly 3 hours to simulate on a dec-station 5000.

Another factor is the random addressing of packets. The active nodes address other active nodes in the system (but not itself) at random. Thus the traffic pattern varies for each simulation. The addressing is done at random by the random-number generator in the compiler (at run-time). It is the randomness of the random-number generator that "steers" the traffic pattern.

The traffic pattern is also shaped by the random-number generator in another way: packets are partly emitted on a random basis. See section 8.6 for a full description.

The two latter factors are dependent on the quality of the random-number generator and the first on computing resources.

8.8 Constructing networks

We have now seen how the nodes and switches are built up using various modules. But how do we model a network of multiple nodes and switches? We have already mentioned the downstream-pointer each interface has. It is a reference to the next interface on a ring. By using this pointer multiple `SCI_interfaces` can be connected together.

These pointers are set during the startup of the simulation. During the startup, a topology-file is first read in [4]. The topology file contains the information needed by the simulator to connect the nodes in the network: First, all the `SCI_interfaces` are listed. Its coordinates (the position in the k-ary n-cube) are given. Also, the address of the downstream `SCI_interface` is given. Then the downstream pointer of the interface can be set. In addition the topology-files indicate whether the `SCI_interface` is in a `node` or `switch`.

In figure 8.12 and 8.13 we show how nodes are connected using the downstream-pointer. Figure 8.12 shows how nodes are connected together in a ring. Figure 8.13 shows how 2 such rings are connected together with a switch.

[4] These files are made up by other programs that understand the topology of the network. See appendix D.

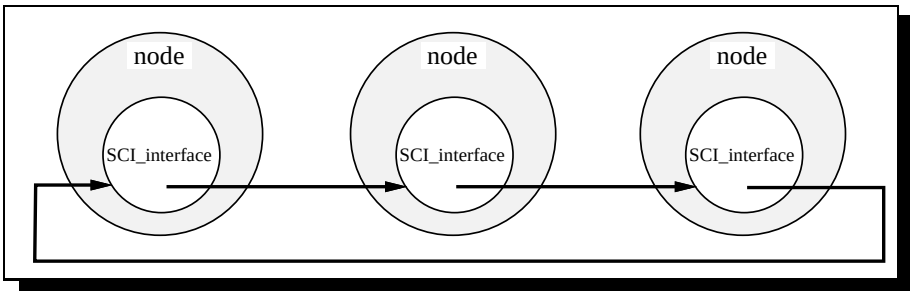


Figure 8.12: Connecting the `nodes`-objects together: a ring of 3 nodes. The arrows are the `SCI_interface`-pointers to the downstream `SCI_interface`.

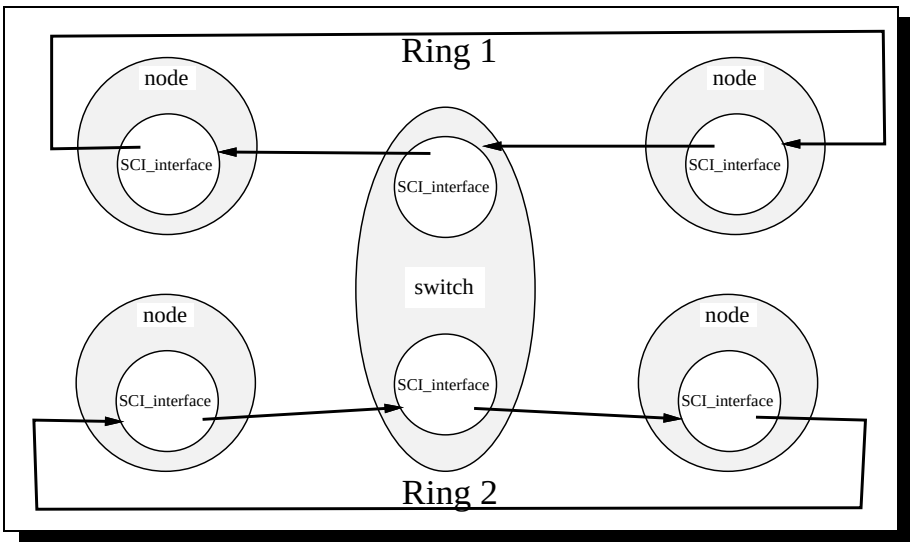


Figure 8.13: Switches: connecting 2 rings.

8.8.1 k-ary n-cubes

We have now shown how we connect nodes and rings together. How do we from this go to constructing larger structures like k-ary n-cubes? In figure 8.14 a possible vertex in a k-ary n-cube is shown. Figure 8.15 shows how $N = k^n$ such vertices might be arranged in a 3-ary 2-cube, using the scheme of figure 7.5.

Unless otherwise stated, all simulated networks use the unidirectional torus-connected scheme (see figure 4.2 on page 30). We have made this choice for 2 reasons: the routing algorithm is then simpler to implement, and we expect the performance to be acceptable. Also, it is plainly very natural to map ring-based networks to a torus-connection scheme.

When simulating k-ary n-cubes with an extra node-ring (the scheme in figure 7.5b) the vertices are organized as shown in figure 8.16.

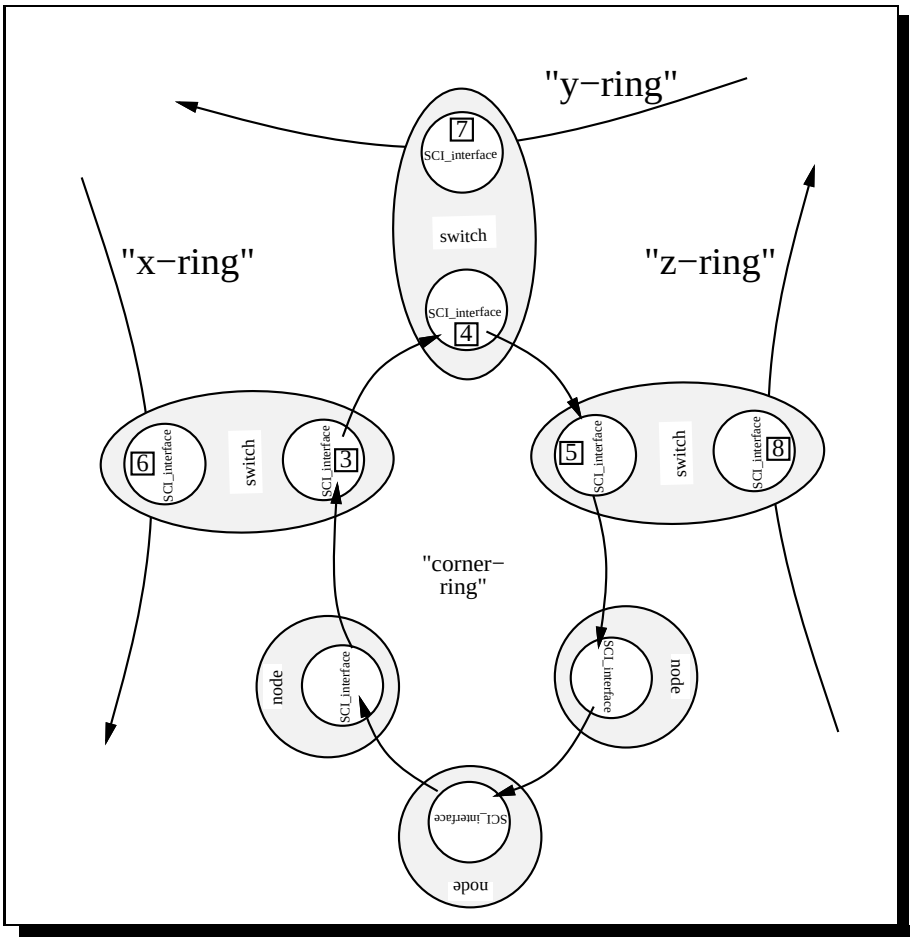


Figure 8.14: A vertex in a k-ary n-cube for $n = 3$, using the scheme of figure 7.5a. The vertex contains 3 switches connecting 3 dimensions, and 3 active nodes.

8.9 X11-animation

Due to the complexity of the simulator we have found it necessary to display the contents of the SCI-interfaces on a workstation. Only with the help of extensive graphic displays were we able to convince ourselves that the simulator behaved as we wished. To create the display, we used the graphic libraries for the X11 window system [5].

Animation was only used during the debugging phase, not during the actual simulation.

8.10 Randomization

We use random numbers for two purposes:

[5] X11 Athena widgets library and X11 toolkit intrinsics library

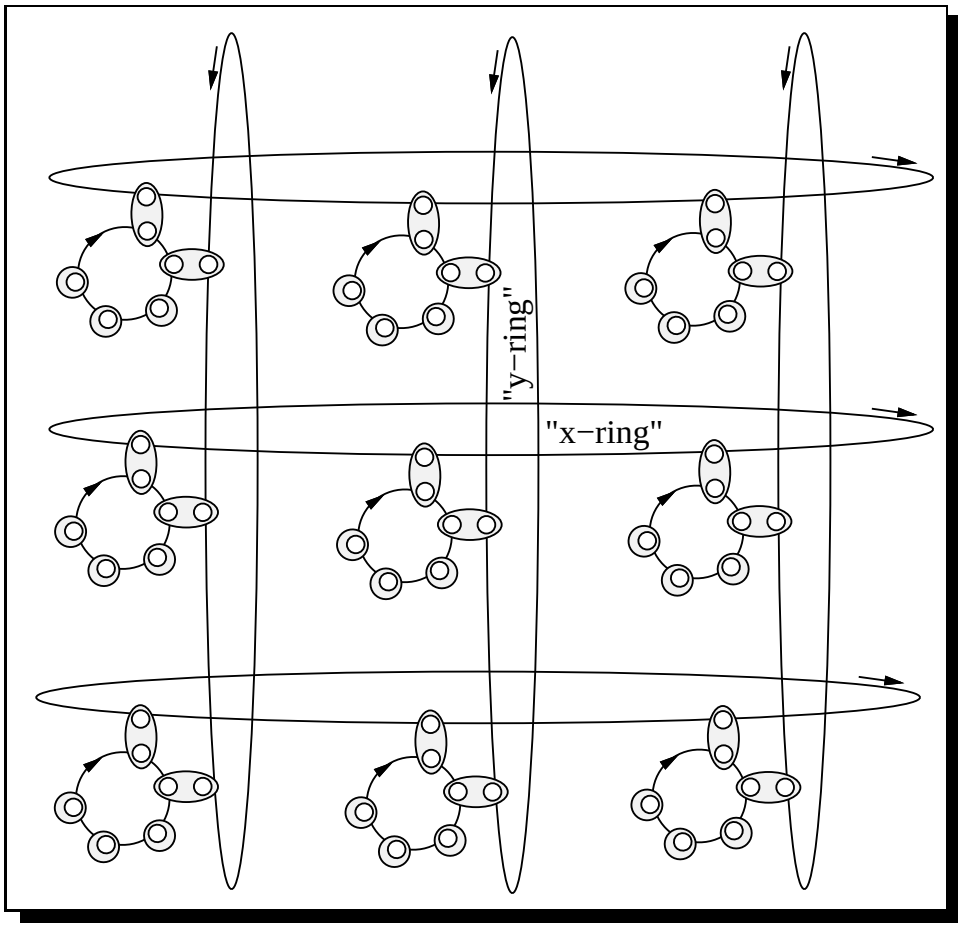


Figure 8.15: Connecting vertices in a k-ary n-cube together. A 3-ary 2-cube.

1. To vary the loading of the network. For this we have used 3 different probability distributions (section 8.6): the uniform distribution, the normal distribution, and the negative exponential distribution [Shannon].

2. To draw addresses for the request-packets. For this we have only used the uniform distribution. This spreads the traffic, and avoids (to some degree) hot spots.

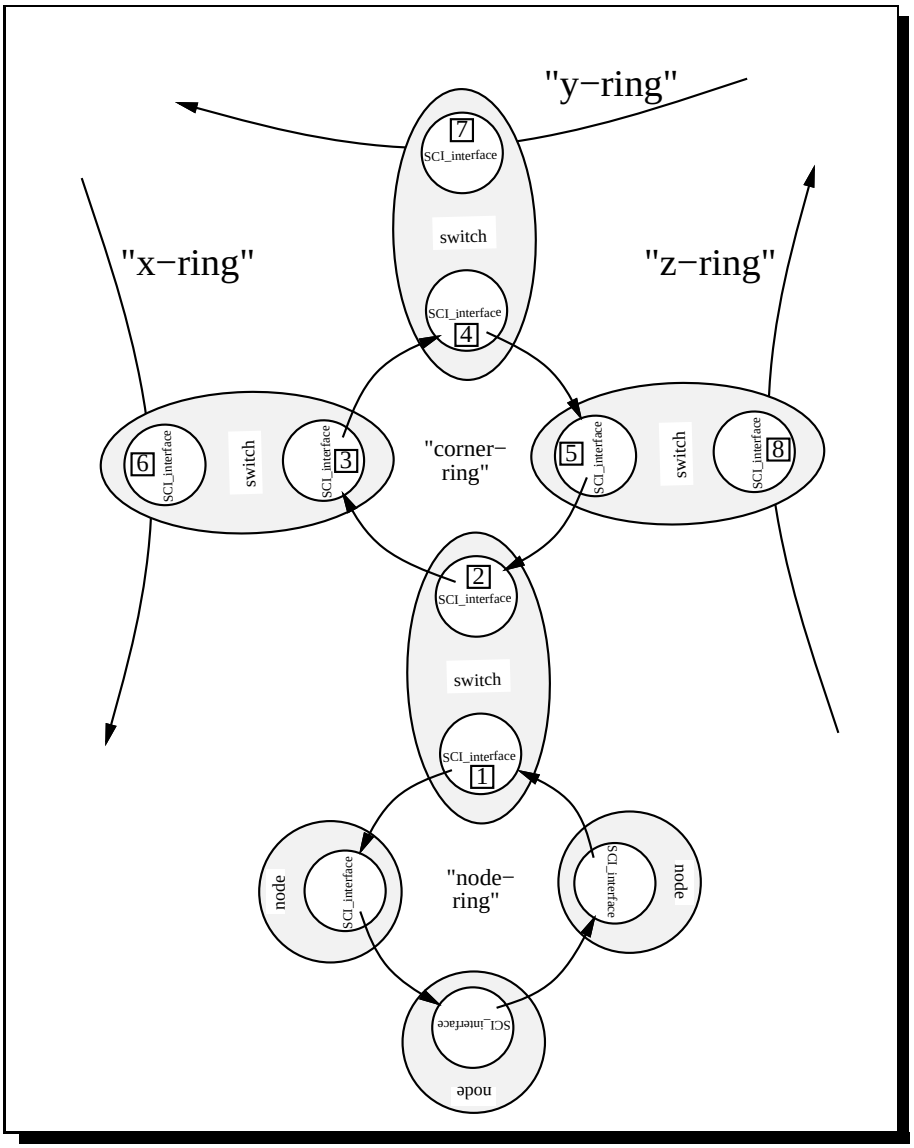


Figure 8.16: A vertex in a k-ary n-cube for $n = 3$, using the scheme of figure 7.5b. Here 3 active nodes are shown.

9

Results

Section 7.1.2 presents a solution on how to construct k-ary n-cubes with SCI-rings. In chapter 8 we explained how the simulator is constructed. This chapter presents the results of using our simulator on a range of selected k-ary n-cube networks.

First we have a short comparison between a selection of possible statistical distributions to use (uniform, normal or negative exponential), for both rings and k-ary n-cubes. This is a continuation of the discussion in section 8.6.

We continue with a discussion on rings, and view the limitations in the performance of a ring.

Then we study the variation of the parameters k, n, and the amount of active nodes separately.

Lastly, we discuss various bridge strategies, and the effect of locality.

Notation A short explanation of the abbreviations used in the figures:

- **k**: The number of vertices in each dimension.
- **n**: The number of dimensions.
- **a**: The number of active nodes in each vertex.
- $(2, 4, 7)$ means a 2-ary 4-cube with 7 nodes in each vertex.
- **Rxx**: A single ring with xx nodes.
- Latency and throughput (simulated values as defined in section 8.7), and other related abbreviations:

  TH Simulated Throughput, in [Mbytes/sec].
  THT Theoretical throughput, in [Mbytes/sec].
  L Simulated latency, in [nanosec].
  LT Theoretical latency, in [nanosec].
  min minimum.
  max maximum.

Wire delay on the links[1]	4 ns
Bypass delay	12 ns
Bridge delay	22 ns
Length of a simulation	200μ seconds
Minimum response time in the responding node	200 ns
Outstanding requests (transactions)	1, 2, & 4
Minimum delay from response to new request	20 ns
Maximum delay from response to new request	30 - 14000 ns
Packet length	80 bytes
Packet header	16 bytes
Data in packet (counted in throughput)	64 bytes

Table 9.1: Parameter constants common to all simulations. All times are in nanoseconds.

ave average.
NX Negative exponential distribution used to vary load.
NO Normal distribution used to vary load.
UN Uniform distribution used to vary load.
#S The total number of switches.
#P The total number of processors.

- Type of simulator (see section 8.1.2)

  sf Store and Forward.
  sf_e Store and Forward with one extra buffer in the switches.
  w Virtual cut-through.
  w_e Virtual cut-through with one extra buffer in the switches.

- Type of vertex (see section 7.1.2.1 and figure 7.5)

  nic The nodes in each vertex is placed in the corner-ring.
  c The nodes in each vertex is placed in a ring by them selves.

- Load variation parameters

  j KiloRequests/second, this is limited by the amount of buffers in the active nodes (maximum outstanding requests). It is abbreviated as "kreq/sec" in the figures.
  p Maximum outstanding requests per active node (# transactions)

[1] For simplicity we assume *even* wire-delays in the system, despite the discussion in section 4.1.1.

When not stated particularly, the "Wormhole" (virtual cut-through switching technique) with extra buffers (in the switch-interfaces) –simulator is used. In most simulations here the nodes are placed in the "corner-ring" (see figure 7.5a). The exception is section 9.3 and 9.6. Also, load variation using the uniform distribution is used in all simulations unless otherwise stated (the exception is section 9.2).

There are very many variations of k-ary n-cubes. It will take to much time to simulate *all* of them. What factors were decisive when we chose the topologies we did? Some were too large to simulate, they demanded too much memory in the workstations. Others had a high switch/node ratio that we felt were unrealistic. Otherwise, the selection presented here shows a fair sample of k-ary n-cubes, with k up to 11, n up to 7 and with a up to 16.

9.1 Simulation of a single ring

The most simple SCI-network is just to have a single ring.

Analysis of the SCI-ring has been done before, perhaps most thoroughly in [ScGodVe]. Also, we ourselves have published a preliminary study of the SCI-ring [HulBot]. By comparing our simulation model-results on a ring, with the results presented by [ScGodVe], we show that the model behaves as expected.

We have simulated ring-sizes from 2 nodes and up to 20 nodes. The load has been varied in the way explained in section 8.6.

Throughput In figure 9.1 the simulated throughput is shown for rings near saturation, for rings of increasing sizes (up to 20). Also plotted is the latency then (near saturation: L_max). As figure 9.1 shows, the maximum throughput of a ring is 1.2 to 1.3 Gigabytes per second, independently of the ring-size. We compare the throughput of 16-node ring (1.2 Gbytes) with simulations of [ScGodVe]. They have approximately 1.2 Gbytes for a 16-node ring [2]. Thus there seems to be little deviation. The theoretical limit for this is 1.4 Gigabytes (see equation 5.3 on page 38), so our simulation results seem reasonable. [3]

But why does the measured throughput vary between 1.2 and 1.3 Gigabytes? Since we vary the loading and traffic-pattern using random numbers, the throughput-line in figure 9.1 will not be completely straight. Note that this is the case for all the simulations presented in this chapter.

Latency As can be seen in figure 9.1, the latency of a ring increases as the ring-size increases. This is rather obvious. The maximum latency under near saturation conditions (L_max) in figure 9.1 seems to increase

[2] The simulation model with flow control, not the analytical model.

[3] We could probably get a higher throughput by sending longer packets. That would reduce the echo- and header overhead. Also less of the available bandwidth is then spent on idle-symbols.

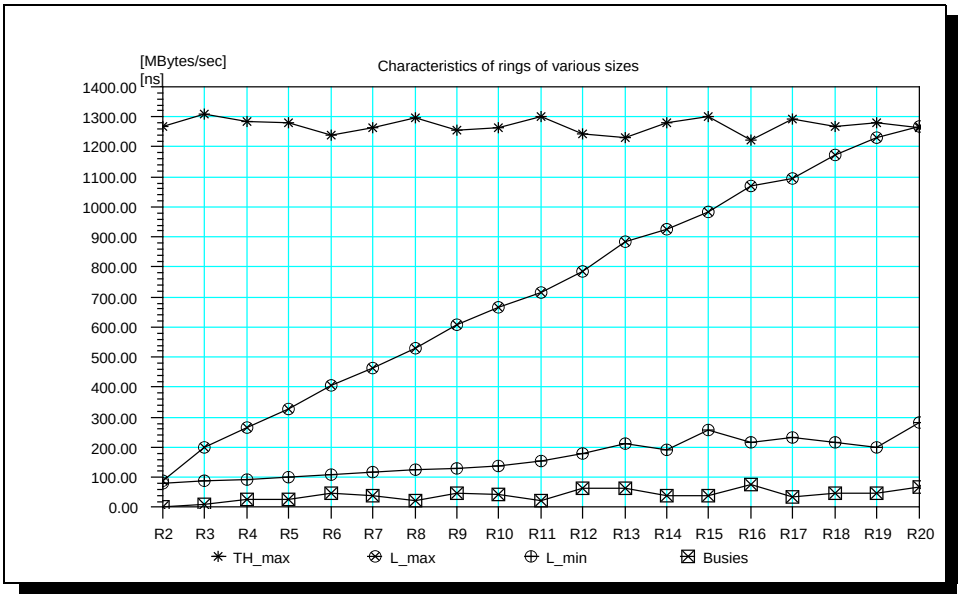


Figure 9.1: Various characteristics of rings for rings with 2 to 20 nodes. The following characteristics are plotted: the maximum throughput (MBytes/sec), the latency measured when maximum throughput is measured (nanosec), the minimum latency (at light load), and the amount of busies measured at highest throughput.

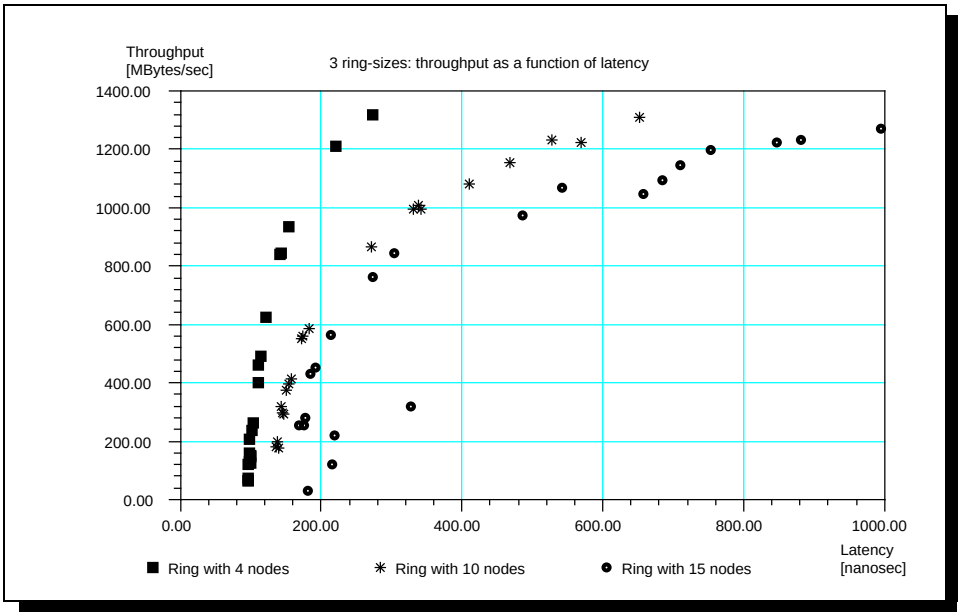


Figure 9.2: Throughput and latency for rings under various loading conditions. 3 different rings are presented: with 4 nodes, 10 nodes, and with 15 nodes.

roughly linearly. This is not surprising. As the ring-size increases, the number of links and bypass-fifos the packets have to traverse, will increase. They both result in constant increase in delay. We have much larger latency, compared to the result of [ScGodVe]. This difference can largely be explained by the fact that [ScGodVe] assume a shorter node and wire delay.

At light load (e.g. at 140 kreq/sec) the latency (marked as L_min in figure 9.1) increases in the same, roughly linear manner.

The table shows a rough comparison between equation 5.4 and 3 selected points on the L_min -curve.

	Measured	Equation 5.4
4 nodes	95	112
10 nodes	138	160
15 nodes	257	200

Note that the L_min -curve represents conditions that are not completely unloaded (but light load), and with a varying traffic pattern. The equation 5.4 represents the average unloaded latency. On the whole, the figures are roughly of the same order.

Figure 9.2 shows the throughput as a function of latency for 3 different ring-sizes. All have a maximum throughput of about 1.3 MBytes/sec. The latency is very dependent on the ring-size.

9.2 Statistical distributions

The active nodes load the interconnection network with their packets. An interesting point is whether our implementation of k-ary n-cubes is sensitive to the form of statistical distribution of the frequency of this loading. In this section we will attempt to see if the interconnection network behaves differently when loaded with different distribution functions. We study the behavior under the negative exponential distribution, the normal distribution, and the uniform distribution.

The background is presented in section 8.6. As one can see from the graphs (figures 9.3, 9.4, 9.5 and 9.6) presented there is little deviation between the simulation results of the three different distributions. In most graphs the uniform distribution presents medium values between the normal and negative exponential distributions. This is the case for all loading situations (the various combinations of p- and j-values). Thus we have chosen to use the uniform distribution on the rest of the simulations we present in this chapter. This is also partly because the uniform distribution is the fastest of the three. All the other distributions calls the uniform distribution as a basis for generating their values.

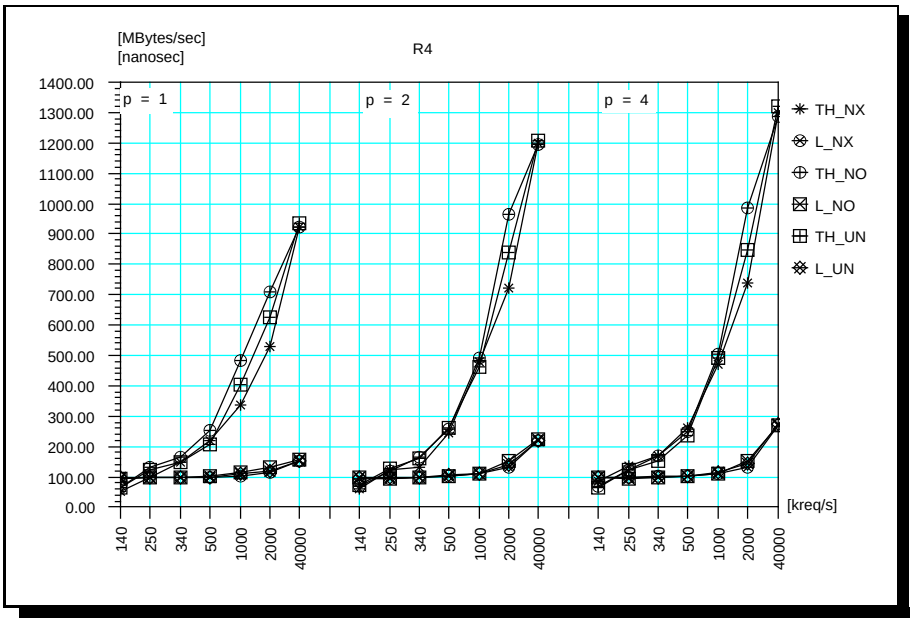


Figure 9.3: Comparison of a ring with 4 nodes using negative exponential, normal and uniform distribution. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

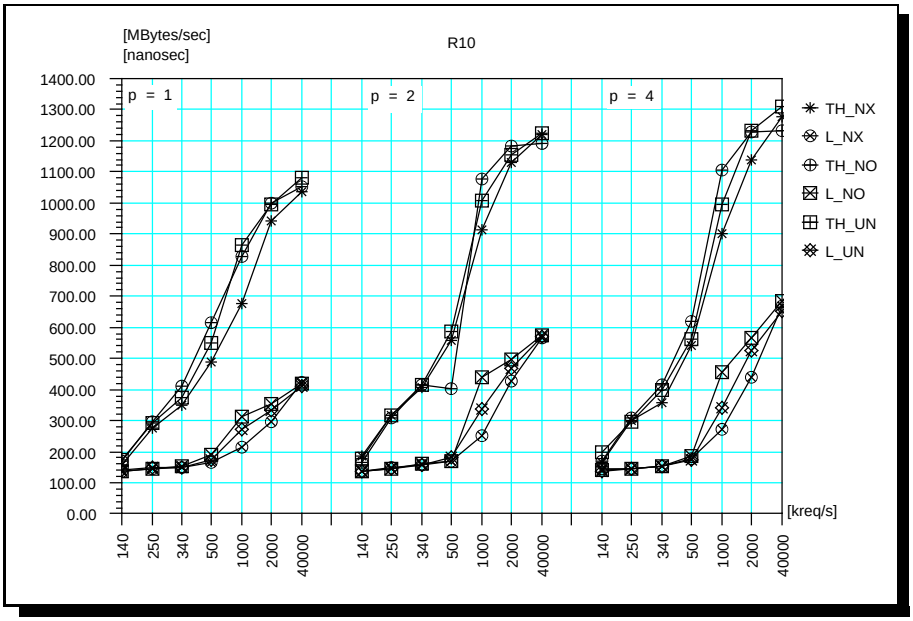


Figure 9.4: Comparison of a ring with 10 nodes using negative exponential, normal and uniform distribution. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

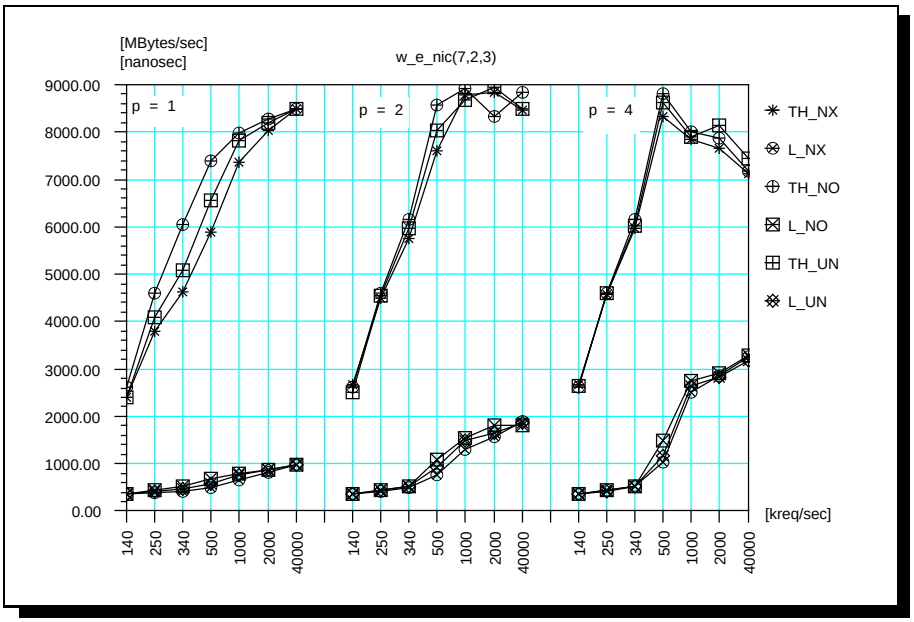


Figure 9.5: Comparison of a (7,2,3) using negative exponential, normal and uniform distribution. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

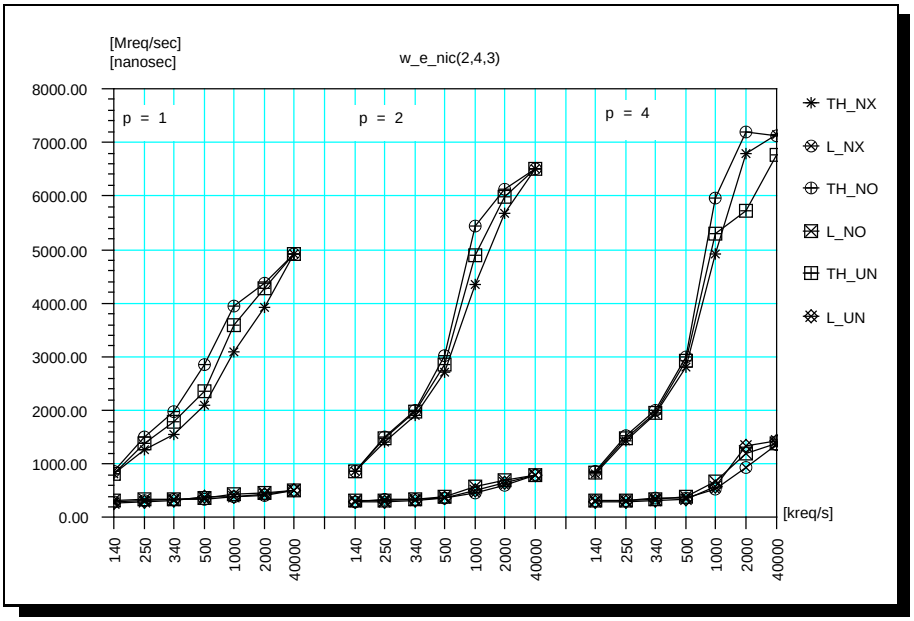


Figure 9.6: Comparison of a $(2,4,3)$ using negative exponential, normal and uniform distribution. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

9.3 Placement of the active nodes

How should active nodes be connected in the vertices? There are two choices which are illustrated in figure 7.5. In figure 7.5a (scheme A) the the active nodes are placed in the corner-ring. Using scheme B in figure 7.5b the active nodes are placed on an additional ring added to the vertex.

Off-hand, it is likely that the latency for a packet is larger in scheme B, since it must traverse the extra switch. This is probably true in most cases, but there might be cases when scheme A is not best: When n is large, the size of the corner-ring is large (it has n switches are connected to it, plus the additional active nodes). The corner-ring might then too easily become saturated. Thus it might be more optimal to use scheme B.

Figure 9.7 shows if this is the case. It shows the difference between the two schemes for 3 different topologies, using scheme A and scheme B.

The top pair of graphs (subfigure a and b) and the middle pair (subfigure c and d) show the typical response. Then the size of the corner-ring using scheme A is moderate (8 and 5 connections respectively). In both the upper and the middle pair of graphs we see that scheme A (subfigure a and c) tends to utilize the available bandwidth better. The ideal is subfigure a, where the latency increases evenly as the throughput rises. The extra switch used in scheme B tends to act as an unnecessary delay (subfigure b and d). This results in the latency being slightly better in scheme A, and also allowing slightly more packets to be sent (larger throughput). The performance of scheme B is specially bad in subfigure d, where the data tends to cluster in the bottom left part of the graph. The available bandwidth doesn't seem to be utilized properly. Scheme B also has more retransmissions (not shown).

The bottom graphs show the response when the size of the corner-ring using scheme A is larger ($n + a = 5 + 5 = 10$). With scheme A the corner-ring now seems to be saturated (becomes a bottleneck). There is more clustering of the data in the bottom left part of the graph. This clustering does not take place using scheme B (subfigure f). This bottleneck seems to be lessened then. Thus we conclude that when the size of the corner-ring becomes large using scheme A, one should consider scheme B instead.

Note that the addressing of packets is completely random. Scheme B is therefore more interesting if some degree of locality is applied. We will therefore return to scheme B in section 9.6. In all other simulations we use scheme A.

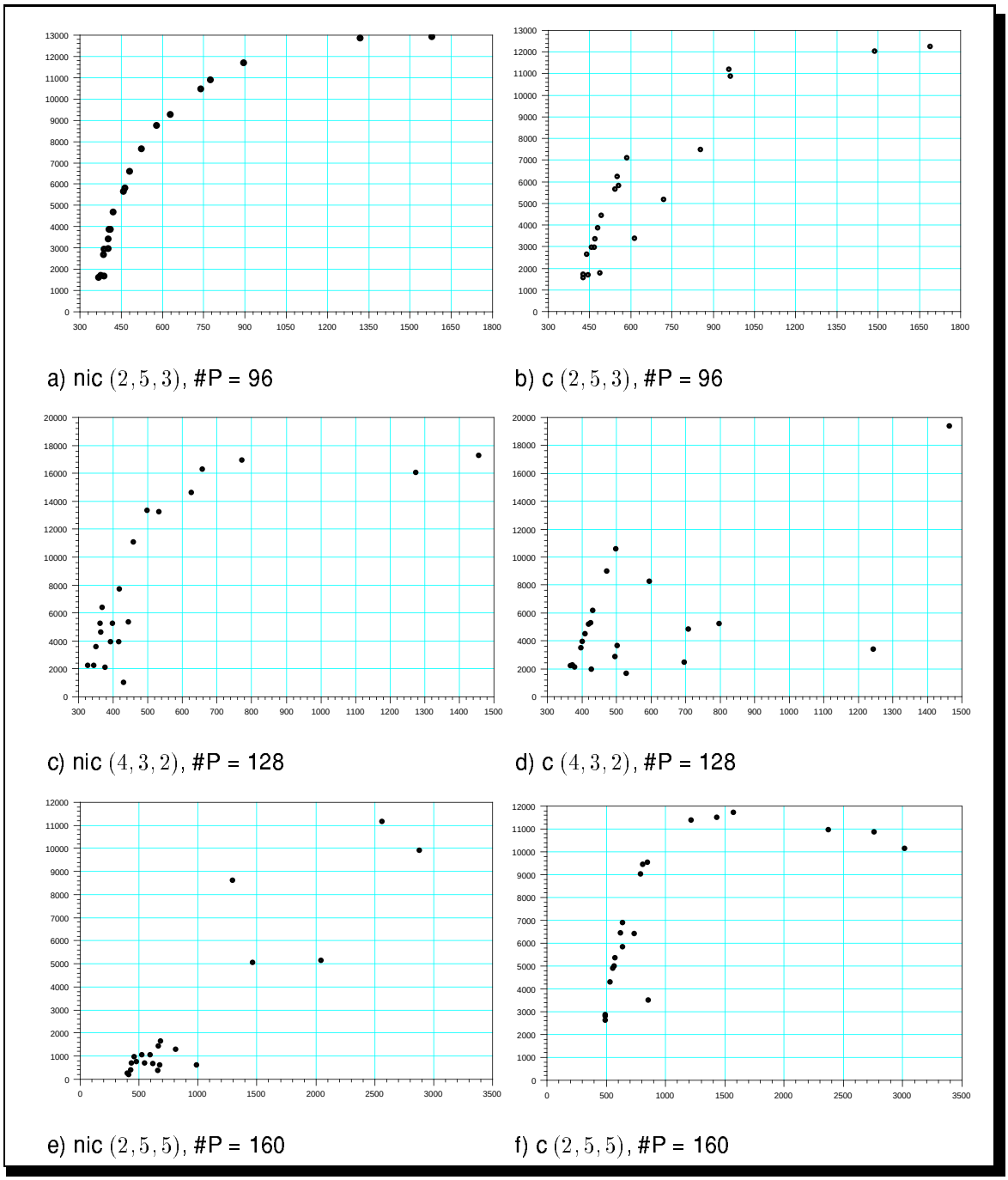


Figure 9.7: Placement of the active nodes. The figures show throughput [Mbytes/sec] as a function of latency [ns] under various loading conditions. Three topologies are shown, using scheme A and scheme B, left and right, respectively. Discussed in section 9.3.

9.4 k-ary n-cubes

The simulations of k-ary n-cubes in this section is made with topologies using nodes connected into the corner-ring (scheme A in the previous section).

The k, n, and the number of nodes (a) in each vertex are varied to investigate the effect on the throughput and latency figures.

In order to make a good comparison one will have to make some sort of measure of what makes some network better than others. The most obvious is the throughput and latency figures, which is the most important parameters describing the performance of a specific network. These parameters are presented and discussed in the first part. Following is a short discussion on the relation between theoretical figures and the simulated. They are presented with graphs giving the throughput per active node in the system in connection with the number of active nodes.

Designing a high performing network is relatively easy, but a theoretical model does not always reflect the implementation-cost. A good measure of the cost factor of k-ary n-cubes (and other networks) is the relation between the number of switching elements (#S) used to connect a certain number of nodes (#P). Combining this with the performance per node with respect to throughput and latency, should give a picture of how to make a network specific for your needs.

A comparison of the simulated results with the theoretical formulas in chapter 7 is also presented.

The last part of this comparison will present some numbers on the amount of busied packets (retries) and their location in the network.

The choice of which topologies to simulate was made taking into account the simulation time, switch/node factor and the conclusions drawn from preliminary simulations. This giving an indication of the maximum values to use for either k, n and a.

9.4.1 Comparing the model with results of [JohnGood]

To make the figures in this chapter more credible we will here attempt to compare a sample of our results with that of others. The work that relates best is the work done by [JohnGood].

[JohnGood] views the vertices as consisting of a processor plus a crossbar-connection to connect the dimensions together. See figure 9.8. We of course, use the scheme of figure 7.5a.

Since the physical representation of k-ary n-cubes is very different in the two schemes, this is a very rough approximate comparison.

We first compare figures for latency at light load, and then figures for throughput.

Latency [JohnGood] shows the worst-case latency at light load, while we have studied the *average* latency at light load. This comparison thus has strong limitations.

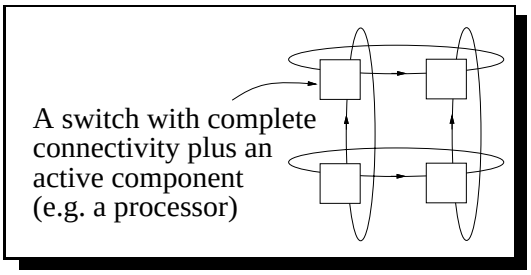


Figure 9.8: A 2-ary 2-cube as presented in [JohnGood].

The physical differences in the two schemes create the following differences in the latency-figures.

1. The cost of switching dimension (occurs on average $n-1$ times) is different. In our scheme the packet must traverse a switch, then a wire, then a switch again. This adds up to 48 ns. [JohnGood] models the cost of switching dimension as 40 ns. This is a difference of $8(n-1)$ ns.

2. We model wire-delay plus bypass-delay as 16 ns. [JohnGood] has this as 10 ns. This is on average $6\frac{k}{2}$ ns difference per dimension.

3. We model packets as 80 bytes, [JohnGood] uses 50 bytes packets. This means 30 ns (15 extra symbols sent) extra in our model.

   In addition our scheme models the time for the packet to leave and enter the vertex (the corner-ring). We assume for simplicity it has only one active node attached. If t_w is the wire-delay, and t_{br} is the bridge-delay, we must then add (to [JohnGood] figures) $2(\frac{t_w}{2}+t_{br}) = 2(2+22) = 48$ ns.

   This must be added to the difference in packet-length: $30+48 = 78$.

With item 1-3 we have explained some of the difference between our measured figures with that of [JohnGood]. The difference is also due to the fact that our figures here represent conditions with "light" load, not completely unloaded, as the figures of [JohnGood] seem to be.

Thus we explain the difference as a result of the physically different solutions selected, the slightly different load, packet length, and the fact that we compare average figures with maximum figures.

The table shows 3 sample comparisons for latency

	Latency (ns)			
Topology	[JohnGood] I (max)	our results II (ave)	II - I	point 1,2,3 above
$(8,2,1)$	$\approx$ 200	293	93	$8+48+78 = 134$
$(5,2,2)$	$\approx$ 130	265	135	$8+30+78 = 116$
$(2,5,2)$	$\approx$ 200	317	117	$32+30+78 = 140$

Throughput We see a much larger deviation in the throughput-figures.

	Throughput (Gbytes/sec/vertex)	
Topology	[JohnGood]	our results
$(2,5,2)$	≈ 1.5	0.4
$(2,5,1)$	≈ 1.5	0.33

Why this large deviation? Some differences:

1. [JohnGood] ignores retransmissions (this applies to latency too)
2. Lastly, and most important: larger switches of [JohnGood] will have much more connectivity than our scheme. This will affect throughput to a very large extent, though it is uncertain how much.

Summary What to conclude from this comparison? First it confirms our model. When we take in account the physically differences in the vertices, our model generates *roughly* the same figures as [JohnGood].

Secondly, larger and more expensive switches seem to have a much larger capacity than our corner-ring concept. This is reasonable. A larger switch has the ability to offer pair of nodes to establish a connection simultaneously. Thus the flow through the switch (vertex) will increase, giving a higher throughput.

9.4.2 Variation of k

To study the effect of various values for k, the value of k is varied, keeping the other parameters constant. The values we have chosen for the topologies (k,2,3), $k = 2,4,6,7,8,9,10$ and (k,2,5), $k = 2,4,6,7,8,9,10$. The graphs are presented in figures 9.9 and 9.10.

The increase of k implies a "stretch" in each dimension. Increasing k results in more nodes on each dimension ring. And then more traffic nodes which generate traffic on these.

The maximum throughput shows a steady increase in value corresponding to the increase in the radix k with both topologies. The latency also follows the increasing k and shows no signs of a rapid increase in relation to maximum throughput. This could mean that one can increase the size of k to a large number.

But as the radix k increases (around $k = 8$ and above) we also get a larger variation on the throughput. This is shown in figure 9.11. Specially in simulations with a high load. This implies that the network is more sensitive for the traffic-pattern applied during high load. This seems reasonable. If a network increases the value of k keeping the other parameters constant, the amount of total applied load to the network increases. The network to handle this extra load only has the same basic rings to transport the packets, leading to a higher load on the dimension-rings. This gives more congestion in each of the vertices. It is evident

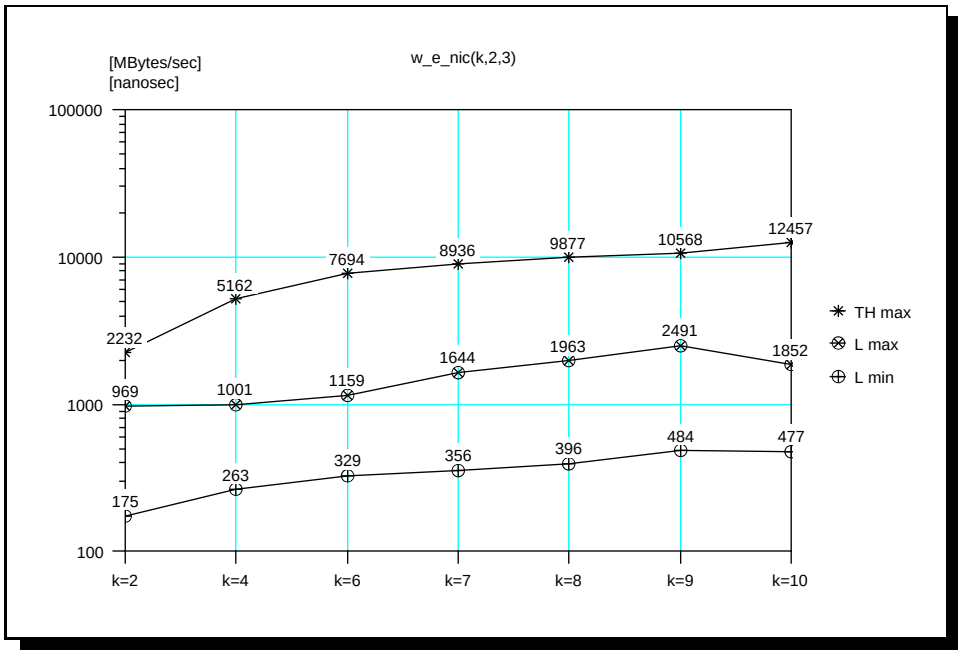


Figure 9.9: Variation of k: The maximum throughput and corresponding latency, plus the minimum latency of (k,2,3).

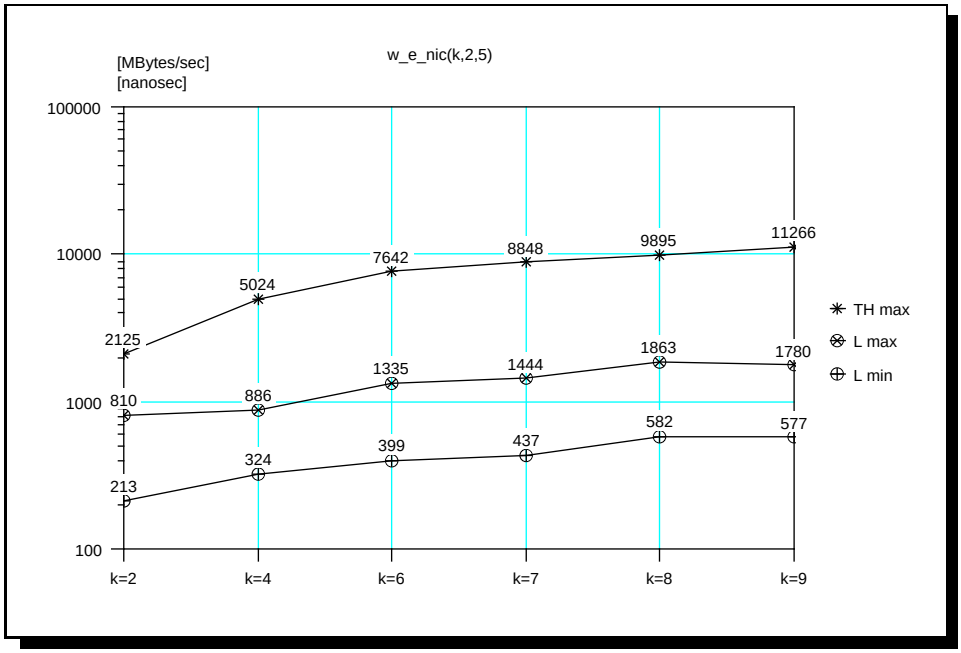


Figure 9.10: Variation of k: The maximum throughput and corresponding latency, plus the minimum latency of (k,2,5).

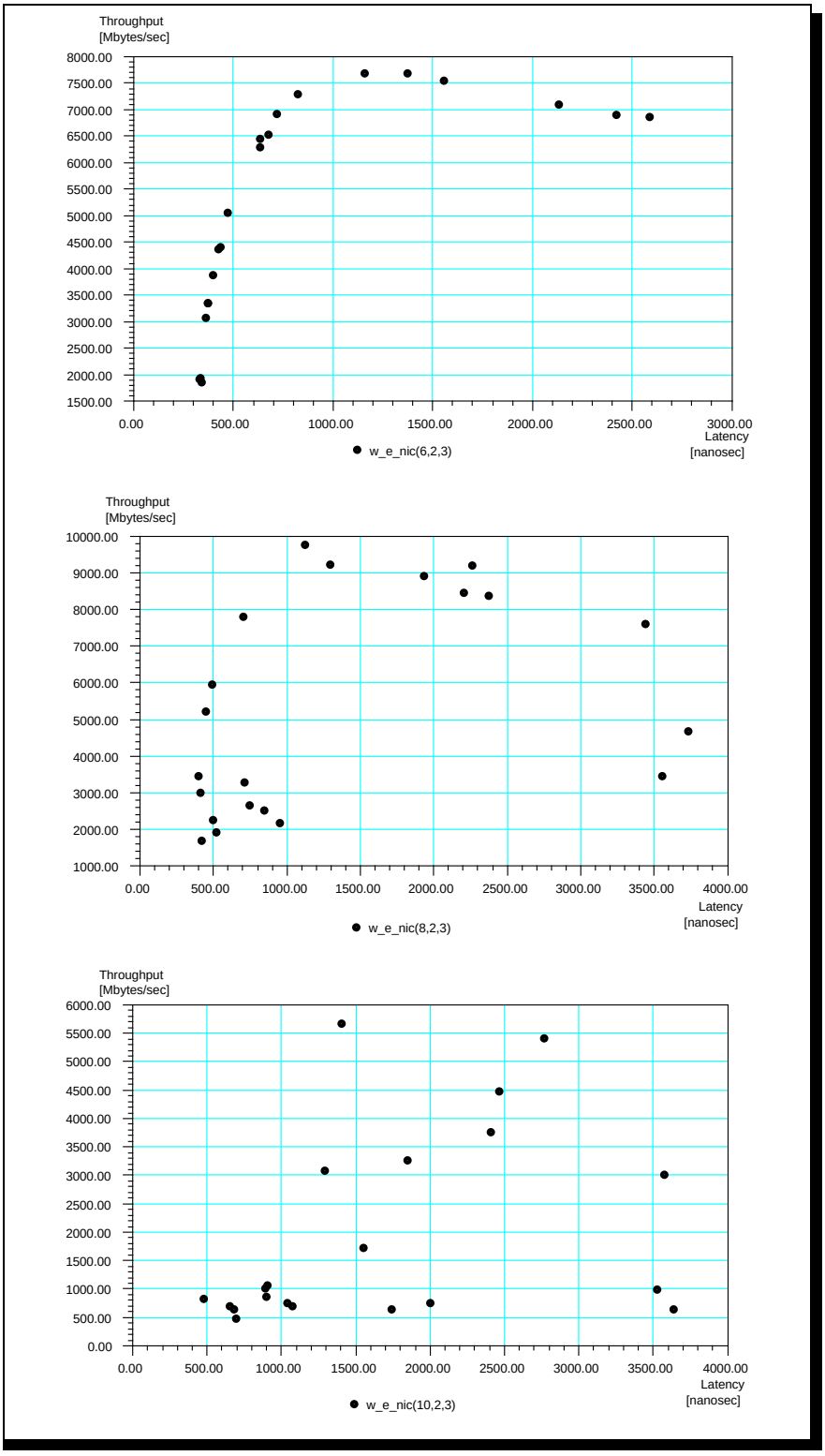


Figure 9.11: Variation of k: The Throughput as a function of latency for 3 different topologies: $(6,2,3)$, $(8,2,3)$, and $(10,2,3)$. This illustrates that the network has a more erratic behavior as k grows.

when looking at the number of busied packets and seeing where they occur. The major part of busy-retries (up to approximately 70 %) is located in the vertices, with packets destined for the dimension-rings [4].

In systems with an average *typical* load, one can let the parameter k get larger than the approximate maximum value 8-10. This by not pushing too much load on the network. This should give good performance figures for the throughput and latency. Calculating the load as in section 8.6 gives a time between each transmission of packets from nodes, of 2900 clock-cycles. It corresponds approximately to the number 250 kreq/sec in the graphs. This is clearly low compared to 40000 kreq/sec. With such a load the performance of a network with rather high value for k one will get a consistent throughput combined with an acceptable latency.

In a system with very high average load, a possible solution to increase the overall performance, is to make use of some locality in the network. This should give a increase in performance to compensate for the extra amount of load. See section 9.6.

The comparison between the number of nodes and the throughput per node is shown to indicate the performance for each node.

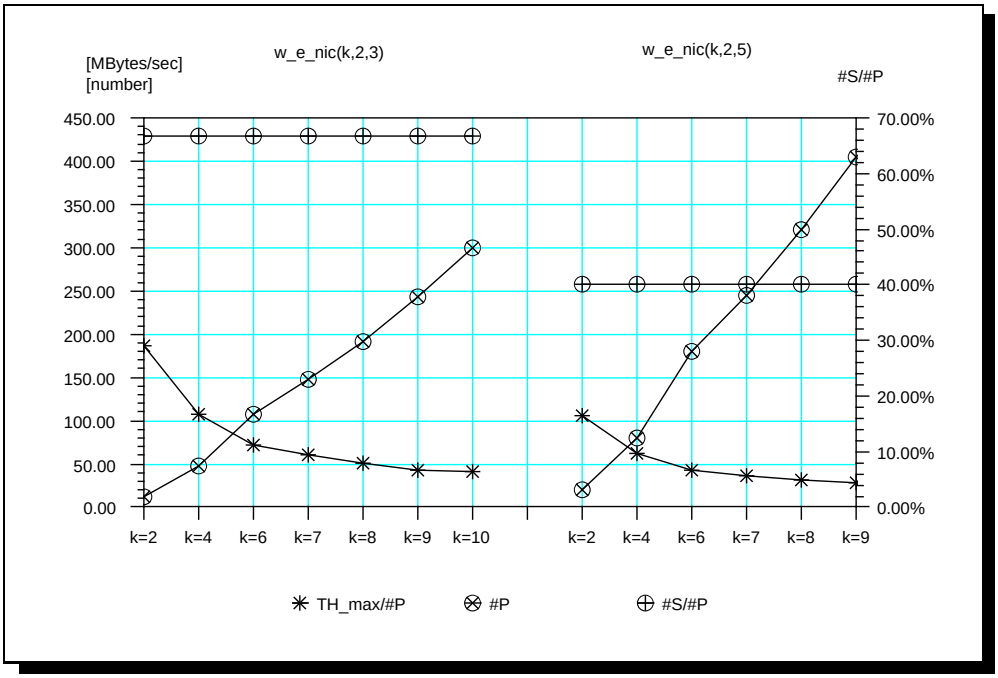


Figure 9.12: Variation of k: Number of active nodes and throughput per active node. Also shown (using the right axis) is the number of switches/active-nodes ratio.

[4] Figure 9.26 is a plot of the number of busies for a $(5, 2, 3)$. It shows that the largest number of busies come from the interfaces labeled as '3', '4', and '5' in figure 8.14.

What it tells us is that the usable throughput for each of the nodes gets considerable lower as the value of k increases.

The ratio between the number of nodes and number of switches is the same for all values of k. This is merely a result confirming that one does not get any extra resources with an increase in k.

9.4.3 Variation of n

As for the variation of k, to study the effect of having various values for n (the number of dimensions), is done by varying the value of n while keeping the other parameters constant. The values we have chosen for the topologies (2,n,3), $k = 2, 3, 4, 5, 6$ and (2,n,5), $k = 2, 3, 4, 5$. The graphs are presented in figures 9.13 and 9.14.

The main difference between varying the value of n instead of k, is that one gets more resources available to handle the larger amount of nodes connected. In other words, the number of rings increases with the value of n, illustrated in figure 9.15. This is quite visible when observing the throughput in figures 9.13 and 9.14 in comparison with the throughput in figures 9.9 and 9.10. For increasing values of n the throughput curve shows an increasing rate of inclination. While the same curve for variation of k the rate is decreasing for an increase in the value k. But as it uses more resources (passes through more switches/vertices) it has a latency which is larger than a topology with high k. Each packet has on average to pass through more switches on its way through the network.

When the value of n increases the throughput seems to get some what unstable. It has the same tendencies as for the large values of k, as shown in figure 9.11. The reason for this behavior is that as the value of n increases, the number of nodes connected to each vertex increases. On average each packet has to pass through one more vertex as n increases by one. In other words the number of packets which passes the vertices becomes higher. The network then becomes more sensitive to the traffic applied.

9.4.4 Varying the amount of active nodes (a)

Varying the number of nodes in each vertex is just another way of setting the load of the system. Increasing the amount of active nodes in each vertex gives an increasing amount of load into the system. In the graphs presented (Figure 9.16 and 9.17) one can see that the throughput decreases when the number of nodes goes beyond 6 to 8 while the latency then increases. This is the same problem as for large values of k and n, the traffic (load) in each vertex-ring is large. Then the total system performance is reduced.

9.4.5 Connecting about 50 nodes

A typical problem might be to choose the best way to connect about 50 active nodes together. The figures 9.19 and 9.20 show the performance

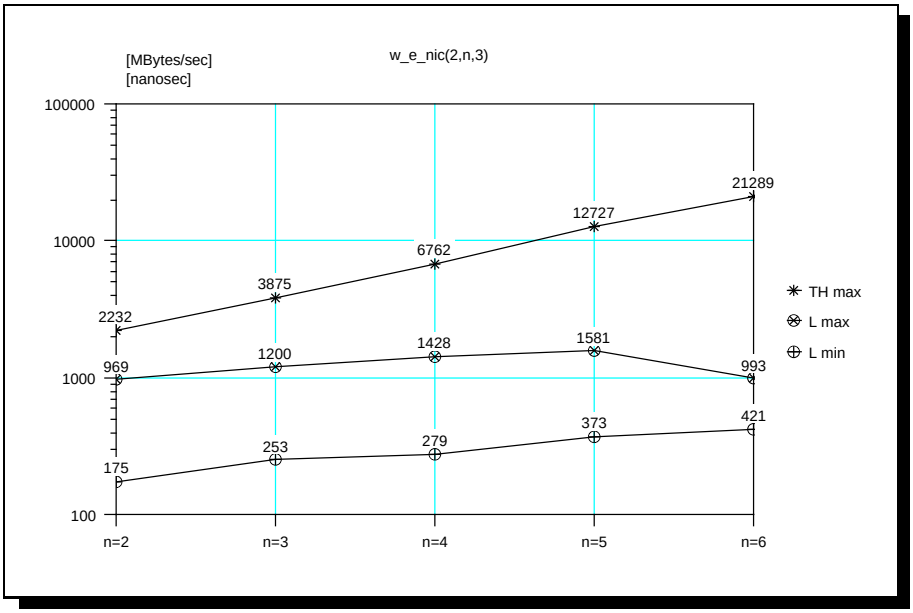


Figure 9.13: Variation of n: The maximum throughput and corresponding latency plus the minimum latency of $(2,n,3)$.

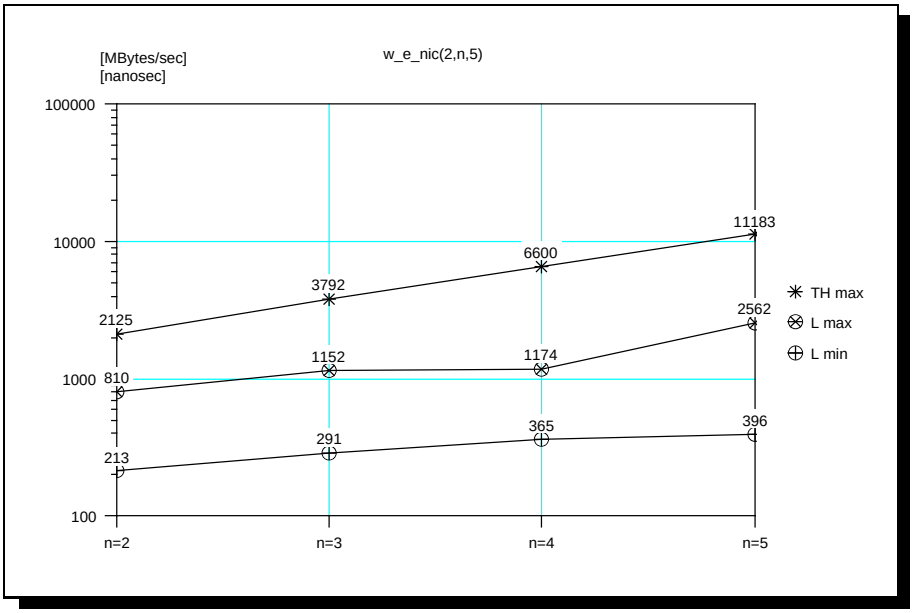


Figure 9.14: Variation of n: The maximum throughput and corresponding latency plus the minimum latency of $(2,n,5)$.

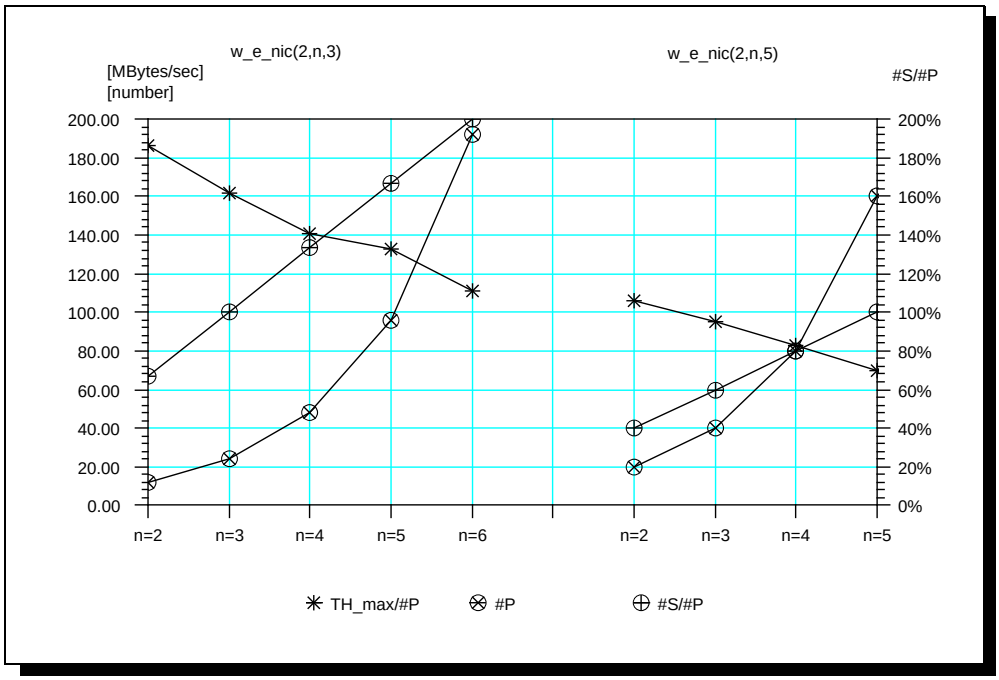


Figure 9.15: Variation of n: Number of active nodes and throughput per active node. Also shown (using the right axis) is the number of switches/active-nodes ratio.

of all combinations [5] of k-ary n-cubes with 40 to 64 active nodes. All combinations are included, except 2 cases where the implementation-cost would be too high.

We will look at three parameters when discussing these topologies: latency, throughput, and cost.

As can be seen in figure 9.19, the mean unloaded latency is approximately the same for all the selected topologies. It varies from 263 ns $(4,2,3)$ to 349 ns $(2,2,11)$. As can be seen in figure 9.20 the measured average unloaded latency is between 1.15 and 1.35 times the theoretical (equation 7.6).

Our measured values for maximum throughput are far below the theoretical (equation 7.7). In figure 9.20 the largest measured throughput-value, relative to the theoretical, varies from 18% ($(2,2,11)$,$(2,2,10)$,and $(2,3,8)$) to 54% ($(7,2,1)$ and $(8,2,1)$). The latter figure is very good, but generally this shows that saturation effectively takes place long before the theoretical maximum throughput can be achieved.

The "L_max" -curve is the average latency measured when the throughput is highest. It varies from 847 ns $(3,2,6)$ to 3000 ns $(2,2,11)$. We see clearly a relationship: when the max-latency is highest $(2,2,10)$,

[5] All combinations of k-ary n-cubes with n in the range 2 to 5, k in the range 2 to 8, and the number of active nodes per vertex from 1 to 11.

Except $(4,3,1)$ and $(2,6,1)$, they have 3 and 6 switches per node, respectively. This is not the best way to utilize resources.

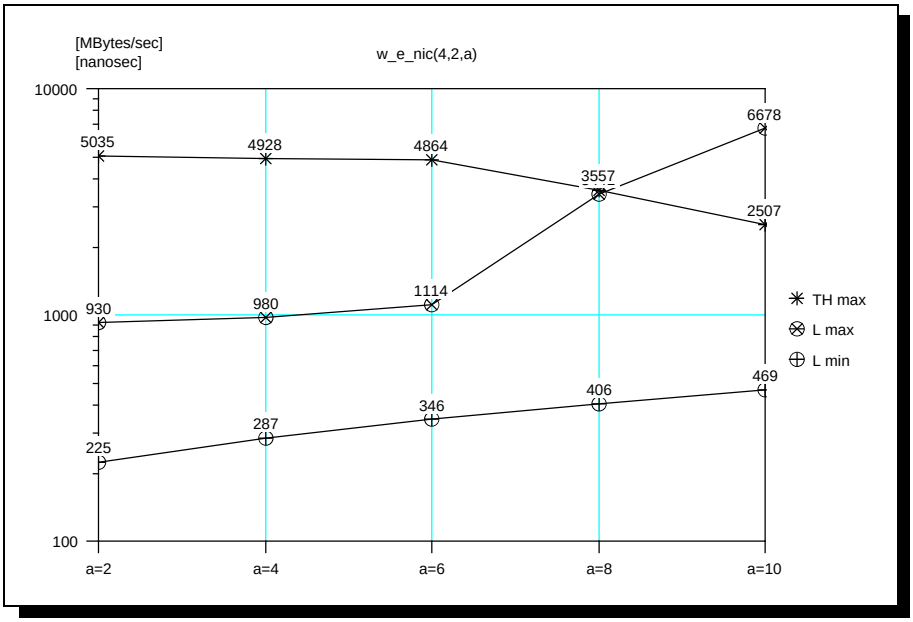


Figure 9.16: Variation of a: The maximum throughput and corresponding latency plus the minimum latency of $(4,2,a)$.

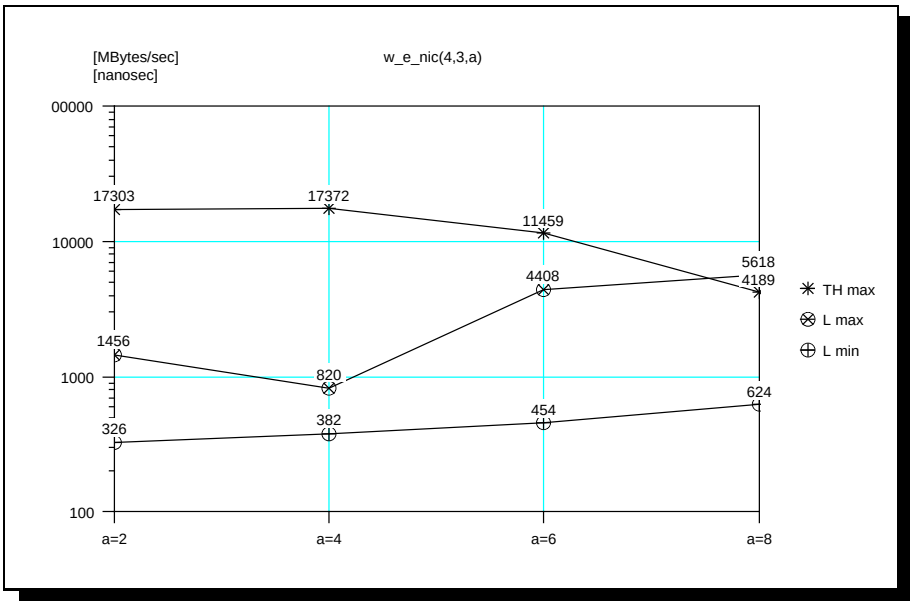


Figure 9.17: Variation of a: The maximum throughput and corresponding latency plus the minimum latency of $(4,3,a)$.

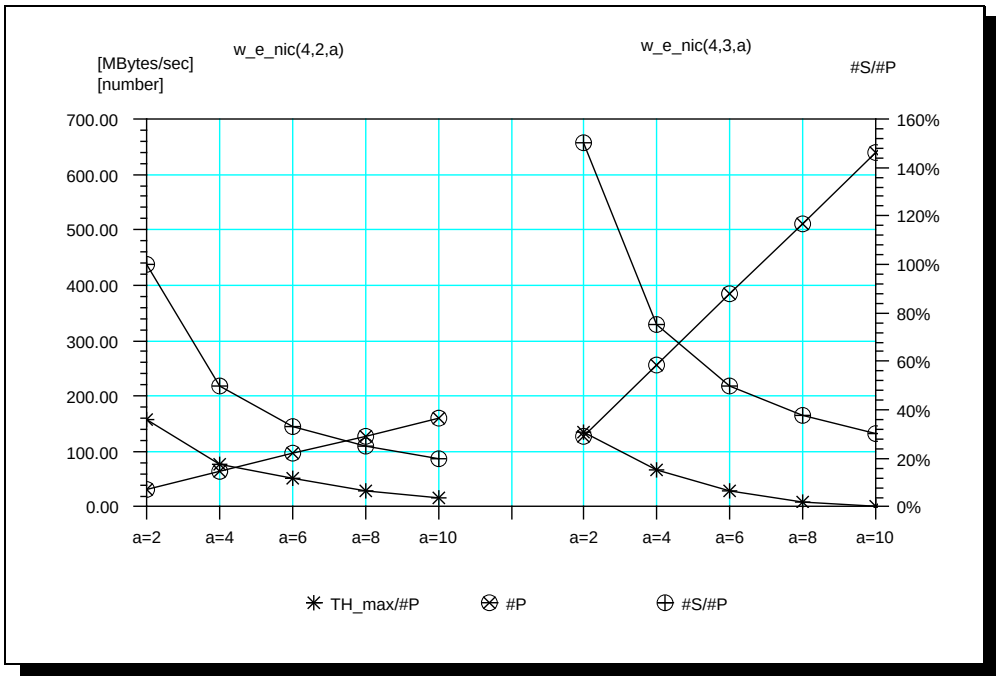


Figure 9.18: Variation of a: Number of active nodes and throughput per active node. Also shown (using the right axis) is the number of switches/active-nodes ratio.

$(2,2,11)$, $(2,3,7)$, $(3,2,7)$, and $(2,3,8)$, the throughput is the worst. In all these cases the size of the corner-ring is 8 or higher. That might explain the bad utilization (saturated corner-ring).

Some of the topologies in figure 9.19 with good throughput and acceptable latency are $(2,4,3)$, $(7,2,1)$, $(5,2,2)$, $(3,3,2)$, $(2,5,2)$, and $(8,2,1)$. In figure 9.20 we see that these topologies have the highest switch/node-ratio. Thus they will cost more to construct.

We can see in figure 9.19 that best performance is when k, n, and a all have "small values". On the right side in the graph, these parameters are "close" in value. If one of them had a high value the performance would be reduced. Thus there is no ring in the system with "many" numbers of nodes/switches connected to it. The opposite is the case in the left part of figure 9.19. Then some of the rings in the system are relatively bigger, thus representing a potential bottleneck.

9.4.6 Connecting about 250 nodes

Another typical problem is how to connect about 250 active nodes together. The figures 9.21 and 9.22 show the simulation results of all combinations [6] of k-ary n-cubes with 240 to 260 active nodes.

[6] All combinations of k-ary n-cubes with n in the range 2 to 6, k in the range 2 to 11, and the number of active nodes per vertex from 2 to 16.

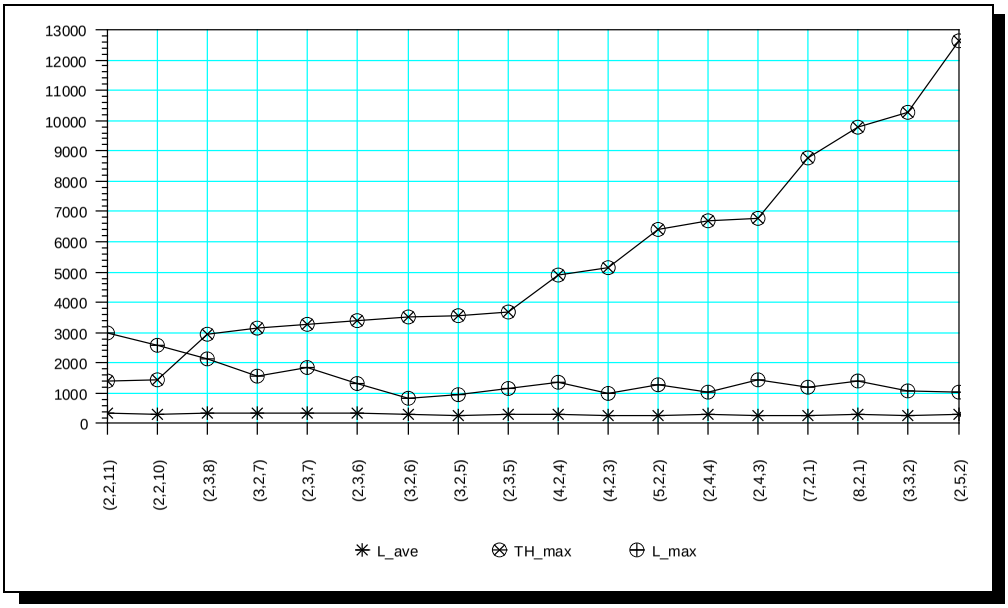


Figure 9.19: Measured performance of various k-ary n-cubes with 40-64 active nodes.

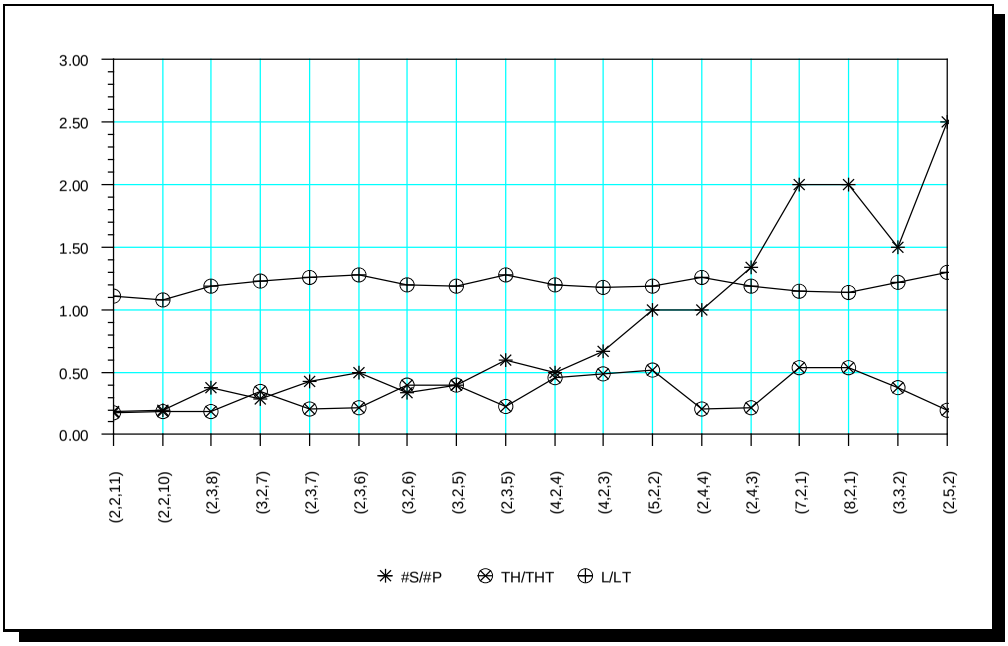


Figure 9.20: Comparing theoretical and simulated results for k-ary n-cubes with 40-64 active nodes. "TH/THT" is the highest measured throughput (from figure 9.19) divided by the theoretical throughput (equation 7.7). "L/LT" is the simulated latency divided by the theoretical unloaded latency. Also, the switch/node-ratio is shown.

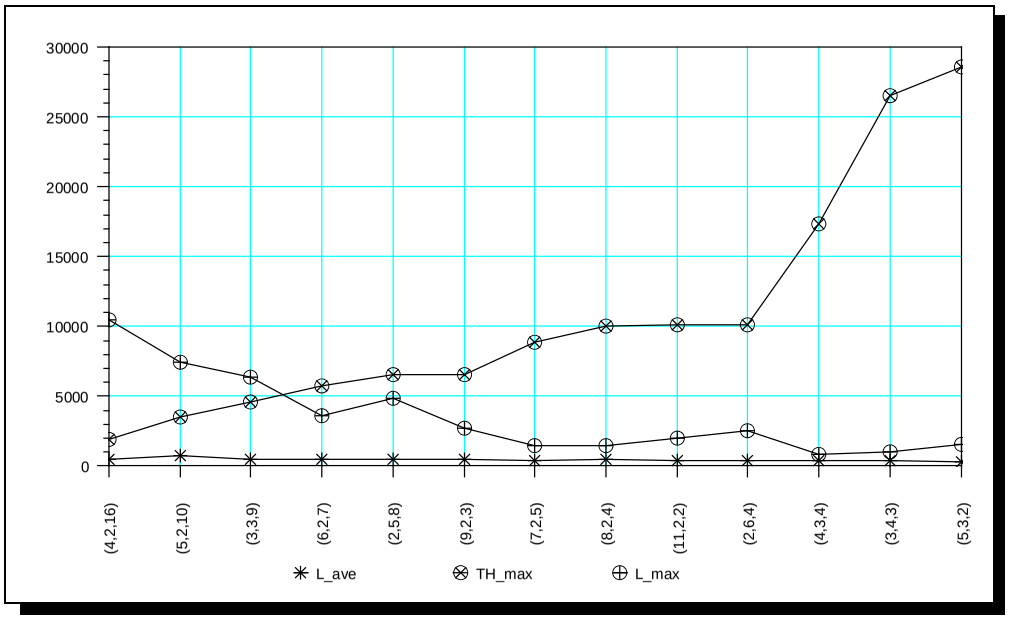


Figure 9.21: Measured performance of various k-ary n-cubes with 240-260 active nodes.

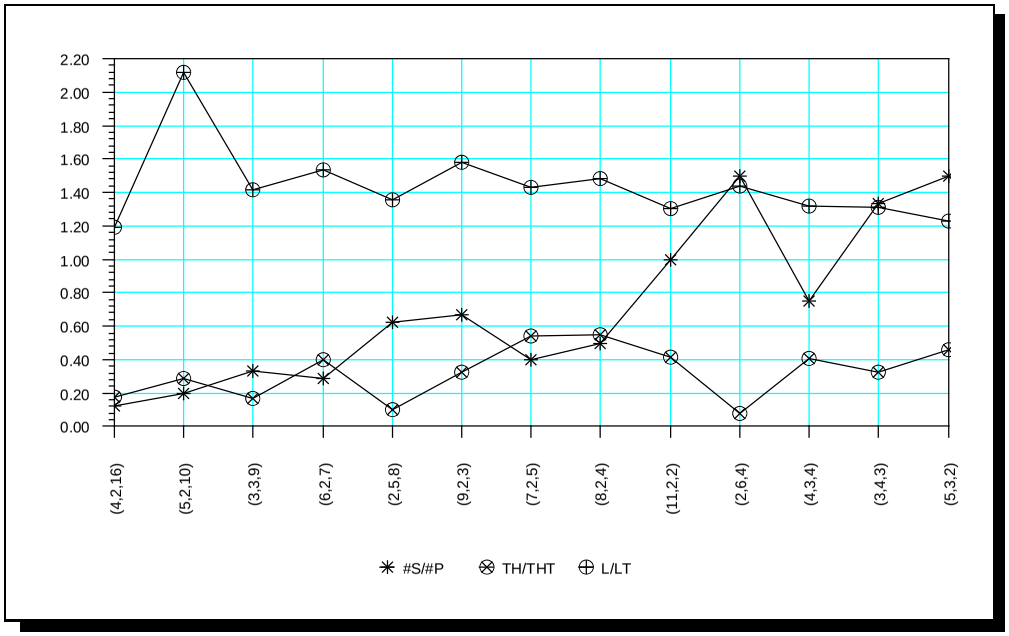


Figure 9.22: Comparing theoretical and simulated results for k-ary n-cubes with 240-260 active nodes. "TH/THT" is the highest measured throughput (from figure 9.19) divided by the theoretical throughput (equation 7.7). "L/LT" is the simulated latency divided by the theoretical unloaded latency. Also, the switch/node-ratio is shown.

Here too, the unloaded latency is roughly the same for all topologies. It varies from 347 ns $(5,3,2)$ to 745 ns $(5,2,10)$. This is between 1.19 $(4,2,16)$ to 2.12 $(5,2,10)$ times the theoretical (figure 9.22).

The maximum utilized throughput is between 10% $(2,5,8)$ and 55% $(8,2,4)$ of theoretical throughput.

The best performance is given by $(3,4,3)$ and $(5,3,2)$. They both have a measured throughput over 25 Gbytes/sec. The latency measured at this peek is 1020 ns and 1559 ns, respectively. Their unloaded latency is 371 ns and 347 ns, respectively. The drawback is, they both have a high switch/node-ratio.

Here too, we see (in figure 9.21) that best performance is when k, n, and a all have "small values".

9.5 Difference in various switch strategies

As mentioned previously we have 4 different simulators, each with different switch strategies (see section 8.1.2).

We had initially planned to simulate the switching technique virtual cut-through from the start, but we first simulated with store & forward in the switches, since that was the easiest to implement. Having simulators for both, this gave us the opportunity to investigate the differences, if any. This is discussed in subsection 9.5.1.

We also thought that adding another pair of buffers in the interfaces in the switches would increase performance. It should reduce the amount of retransmissions. Thus, in subsection 9.5.2 we study the effect of having an additional pair of buffers in the interfaces in the switches.

9.5.1 Store & forward versus virtual cut-through

The switching technique used on a ring in SCI is virtual-cut-through (the use of the bypass-fifo). How does the switching technique in the switches affect performance? We will here discuss the use of store & forward and virtual-cut-through in the switches.

In both cases we use the simulators that has an extra pair of buffers in the interface in the switches.

With virtual cut-through the packet is stored only if contention arises. Thus, intuitively virtual cut-through should behave as store & forward when there is high load (see figure 3.7d). At low load virtual cut-through should only use the buffers sparingly (see figure 3.7e).

Simulation indicates that it is not so simple in reality. We see in figure 9.23 that as the loading of the network is increased, the difference in the average latency of the two schemes decreases. Thus, as the load increases, more packets are buffered in the virtual-cut-through scheme. Virtual-cut-through approaches store & forward with respect to latency. But even at the highest loading there is a notable difference between the

Except where the switch/node-ratio is too high (subjectively chosen): $(4,4,1)$, $(3,5,1)$, $(2,7,2)$, and $(2,8,1)$.

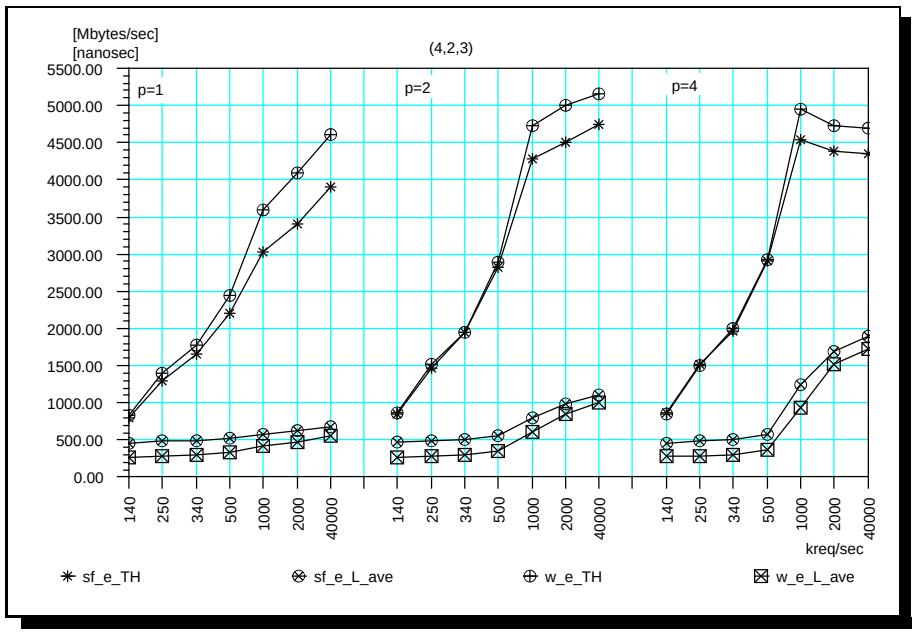


Figure 9.23: A $(4,2,3)$ with 48 active nodes. The use of store & forward (marked as sf_e) or virtual-cut-through (marked as w_e) in the switches. The difference in latency-performance (marked as L) and throughput-performance (marked as TH) under increasing load. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

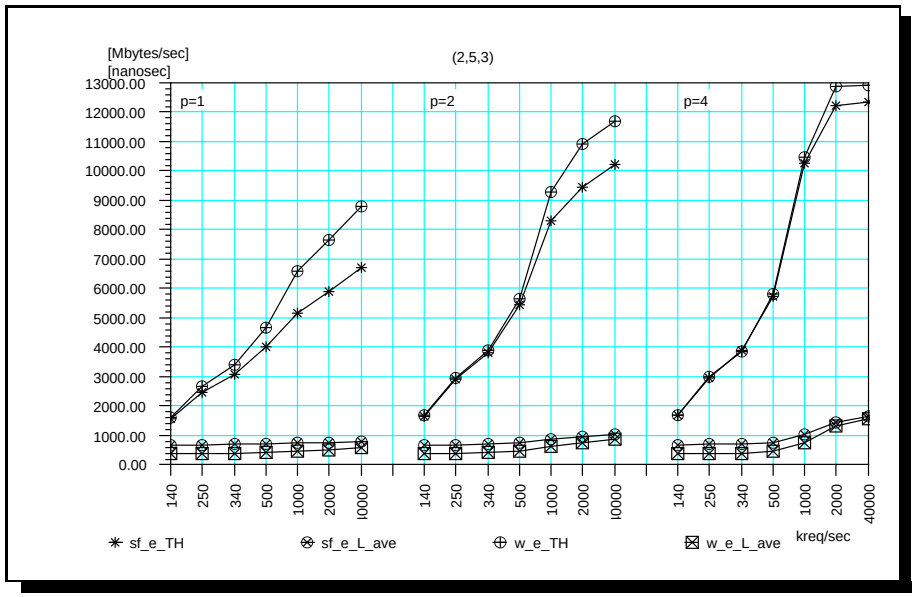


Figure 9.24: A $(2,5,3)$ with 96 active nodes. The use of store & forward (marked as sf_e) or virtual-cut-through (marked as w_e) in the switches. The difference in latency-performance (marked as L) and throughput-performance (marked as TH) under increasing load. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

two. Since the latency for virtual-cut-through is lower at high loading, the active node can send more packets, thus increasing the throughput.

When the loading is decreased, the number of packets sent is decreasing. The amount of packets sent for both schemes approach one another, thus the throughput for the two approaches one another.

Figure 9.23 $(4,2,3)$ is a "small" topology of 48 active nodes. We see the same trends in figure 9.24, a larger topology of 96 nodes $(2,5,3)$.

9.5.2 Varying the amount of buffering in the switches.

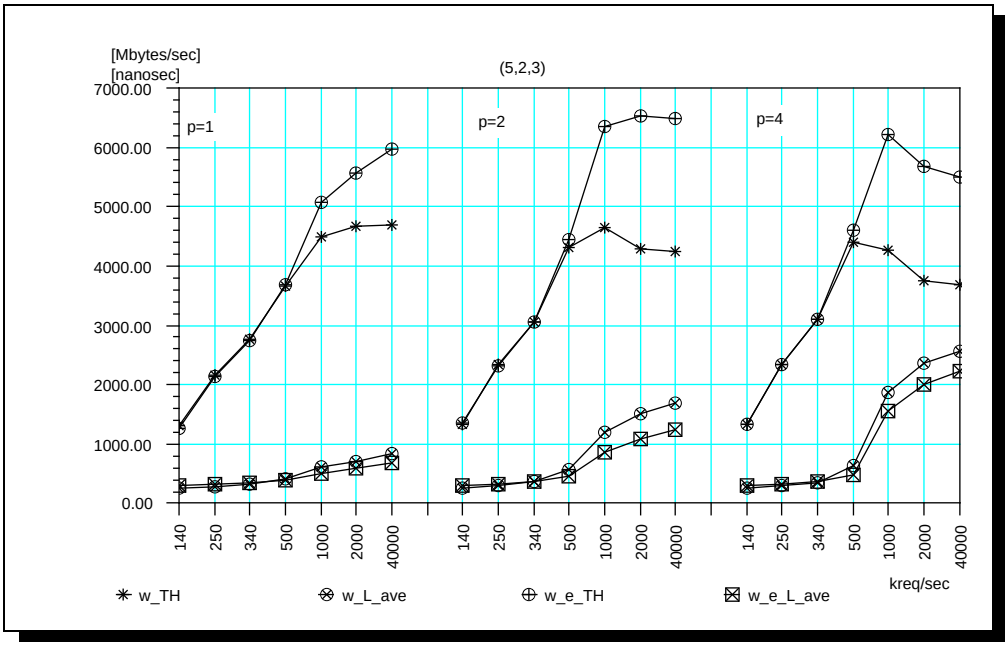


Figure 9.25: The difference between having, or not having an additional pair of buffers in the switches. A $(5,2,3)$ (has 75 active nodes) is shown as an example. The 3 graphs show the performance when having 1, 2 and 4 outstanding requests, respectively.

As mentioned previously, we have the possibility of simulating with the switches having an additional pair of buffers. An interesting point is to what degree this affects performance. Adding the buffering capacity should reduce the number of retries. That should increase the amount of packets sent, which in turn increase the throughput. Also, since the number of retransmissions should be reduced, the average latency should be reduced.

We see in figure 9.25 that this behavior appears, but only at the highest loading conditions. We simulated a $(5,2,3)$ with an extra pair of buffers in the switches, and a $(5,2,3)$ without extra buffering. Both using the virtual-cut-through switching technique.

We made similar comparisons for $(2,2,3)$ and $(2,5,3)$ and saw similar trends (not shown). The difference is negligible when the load is less than 500 – 1000 kilo-requests/second.

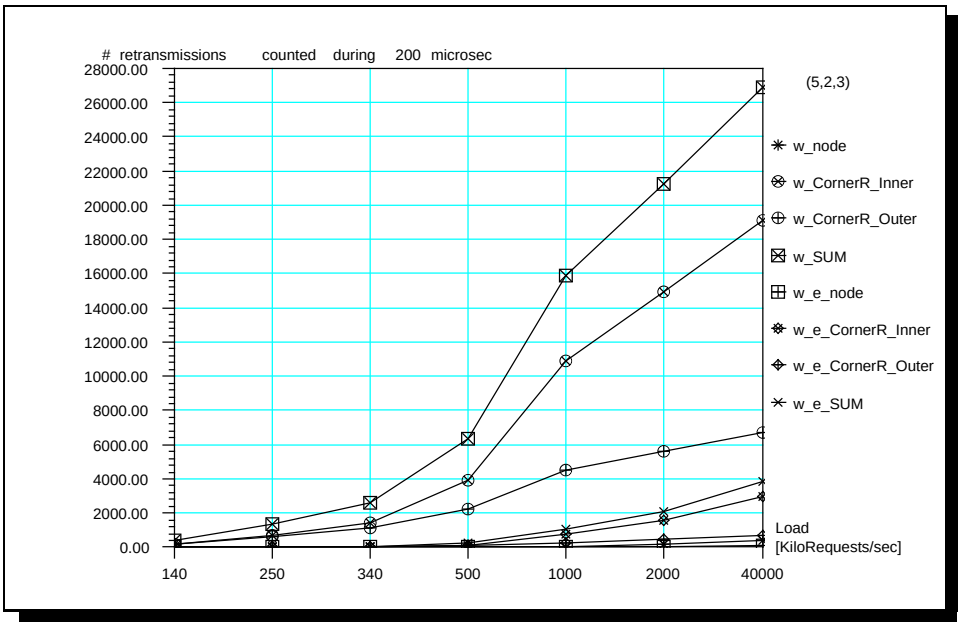


Figure 9.26: The amount of busies (as a function of load) counted when simulated with and without extra buffering in the switches. "node" is the interface of an active node, "CornerR_Inner" is the interfaces marked as 3,4,5 in figure 8.14, "CornerR_Outer" is the interfaces marked as 6,7,8 in figure 8.14. The graph shows the performance when having 1 outstanding request.

Figure 9.26 shows the number of retransmissions counted during the same simulation (lasts 200μ seconds). At high load (e.g. at 40 000 kilo-requests/second) there is a large difference in the amount of retransmissions generated when not having extra buffering (lines marked as w_SUM or w_e_SUM). A total of almost 27 k retransmissions were generated when not having extra buffering (at 40 000 Kilo-requests/second). When an extra pair of buffers per interface were added, this dropped to about 4 k retransmissions. The simulation period was in both cases 200μ seconds and the loading the same. Note that in both cases the largest portion of retransmissions was generated inside the "corner-ring". This is seen by the lines marked as "CornerR_Inner" in figure 9.26. Thus the corner-rings are the bottlenecks in our method of implementing k-ary n-cubes.

9.6 Various levels of locality

All simulations so far have randomly addressed packets. Packets are addressed evenly to all parts of the k-ary n-cube.

What performance can we get if we put some degree of "locality" into the addressing of packets?

With locality we mean that the probability of communication between two nodes taking place increases if they are physically placed

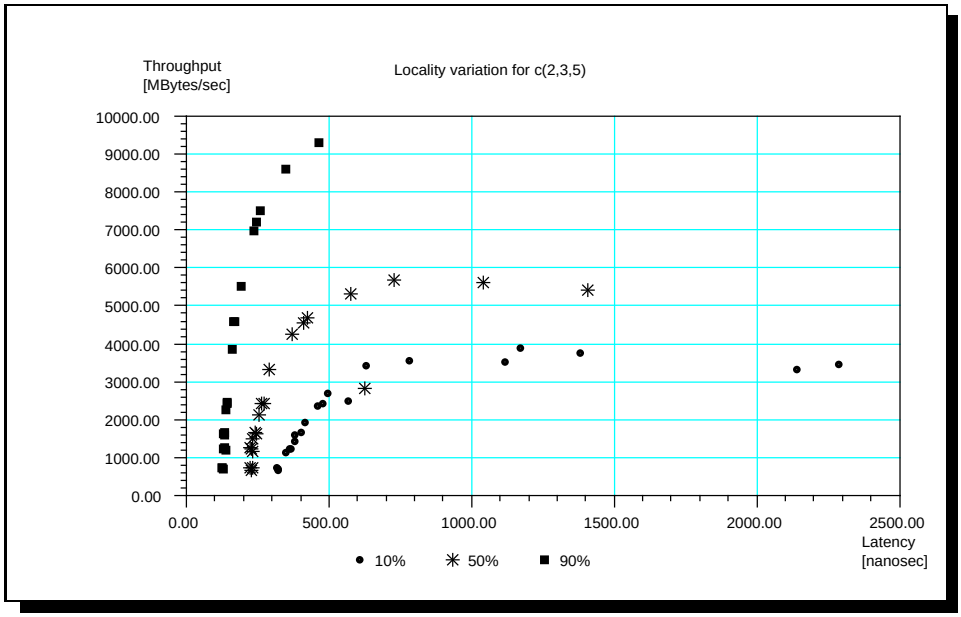


Figure 9.27: Scatter-plot for a c$(2,3,5)$ (has 40 active nodes) with various levels of locality. Each level of "locality" represents 21 simulation-runs with various loading conditions. As a larger percentage of packets are addressed within the local vertex, the average latency decreases, and the maximum total throughput increases.

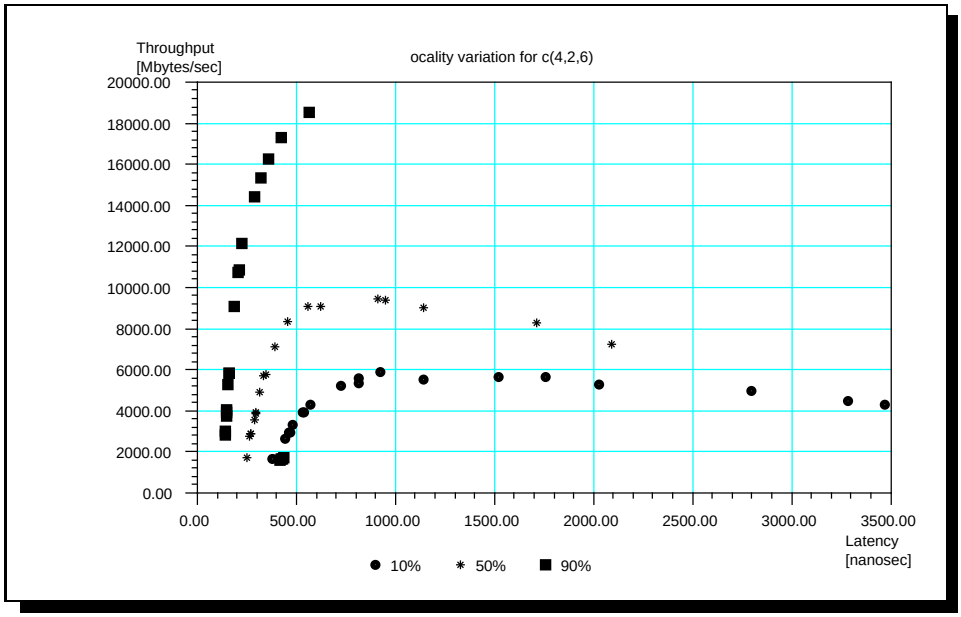


Figure 9.28: Scatter-plot for a c$(4,2,6)$ (has 96 active nodes) with various levels of locality.

"close" to one another. By close we here mean within the same vertex in the k-ary n-cube. Locality will increase the throughput and decrease the average latency [Agarwal]. Latency should improve because the risk of contention and the number of hops is decreased.

In some applications it is reasonable to assume that the program is divided into several parts, with various amount of communication between the different parts.

An example of this is if the local processors mostly access data and instructions in a part of memory that is physically stored in the same local vertex.

We have studied the effects that locality has on performance by varying the fraction of packets that are addressed to the active nodes in the local vertex only. The rest of the packets are addressed at random to all the other nodes in the topology (including nodes in the local vertex). For example, in the data marked "90%" in figure 9.27, 90% of packets are addressed to active nodes in the local vertex. The additional 10% of the packets are addressed randomly to any part of the k-ary n-cube.

This scheme will probably be further enhanced if we physically separate the "local" traffic from the more global traffic. To achieve this an additional ring (within the local vertex) is added to the vertex, reserved active nodes only. This is scheme B of section 9.3.

Simulations using this model is presented in figure 9.27 (has 40 active nodes). We here see that as the degree of locality grows, the variation in latency decreases. It approaches the average latency of independent rings (compare with figure 9.2). In addition, a much larger throughput is possible with locality (compare figure 9.27 with $(2,3,5)$ in figure 9.19).

We can see the same trend in larger topologies. Figure 9.28 (has 96 active nodes) is an example.

9.7 Summary

Rings are not optimal when they are allowed to grow in size. The latency grows linearly with the size of the ring. This makes it unsuitable to use a single ring when a large number of nodes are to be connected. In addition: Irrespective of applied load, the total throughput figure for a single ring has a clear upper limit.

When constructing k-ary n-cubes the size of the dimension-rings (k) and the corner-rings ($n + a$) should be chosen with approximately the same value. This gives the best performance. Then the size of the corner-rings and the dimension-rings are balanced (assuming scheme A in section 9.3).

The value of k should be held lower than 8 – 10. A further increase in the value of k gives too much applied load into the system. The number of vertices connected to the dimension-ring then gets too large.

The range best suited for n is up to approximately 6. Increasing this value beyond this gives a relatively high latency, degrading the total

system performance. This mainly because it takes more time to switch to another dimension instead of continuing in the same dimension (the favored switch setting -property on described on page 66). One should also note the dependency between n and a. Since the sum of a and n gives the number of connected interfaces (nodes and switches) on each of the corner-rings.

Increasing the number of active nodes in each of the vertices (a) effectively increases the applied load. If the applications running on a system can make use of locality, greater values of k, n, and a can be acceptable. Another possibility in order to increase the number of active nodes in each vertex is to put them on an additional ring (scheme B in section 9.3).

We conclude that none of the rings used to make up the network-topology should be allowed to have more than approximately 8 – 10 nodes or switches connected to it. In addition, each of the rings (corner-rings and dimension-rings) should be roughly equal in size. In other words: All ring-sizes should be balanced.

If lower performance (throughput) is acceptable, or locality is applied, the limitations of the above can be somewhat modified. For instance, by having a large degree of locality, the size of the corner-ring can be enlarged. In addition, we saw in section 9.6 that this can be enchanced if scheme B of section 9.3 is used.

10

Conclusions

This thesis started out as a study of interconnection networks for use in SCI and as a comparison between SWIPP and SCI.

To study topologies for SCI we decided to build a simulator. The two most important decisions we had to make were: the type of topology and the type of switches. After a thorough study we decided to use k-ary n-cubes made up of 2-port switches.

The SCI standard defines a network interface in a multiprocessor system. We have seen that SCI works well with the k-ary n-cube topology. A routing algorithm for k-ary n-cubes can be made very simple, and cheap in terms of extra hardware resources. With our implementation the network is also provenly deadlock-free. We have observed that the networks are stable under all loading situations, and that they are not sensitive to any particular statistical distribution. The use of SCI-rings in k-ary n-cubes has been shown to be a very robust combination. This is due to the simple routing algorithm, combined with the virtual cut-through switching technique.

On the other hand, everything is not perfect. The drawback with our way of implementing k-ary n-cubes is that every corner-ring has a potential of becoming a bottleneck in the system.

To get the best performance out our topology implementation, one should carefully balance the sizes of the corner-rings versus the dimension-rings. A deviation from this rule is if the size of the corner-rings get beyond a certain limit. If the number of nodes get too large one can put the active nodes on a separate ring of their own, this giving better results.

There are two factors that contribute most to improve performance: more complex switches and locality.

A more complex switch would yield a higher performance than using our implementation. With complex we mean more input and output ports and a larger degree of connectivity between them. Thus the study of larger switches with more connectivity is an important factor worth further study.

The other means of getting an improved performance is to introduce locality. When introducing locality in each of the vertices in the network

the overall performance increases. The performance increases with the increasing degree of locality.

We have verified our simulation model by comparing it with the *roughly* similar work done by others. When comparing our single-ring-simulations with others work, we see that there is relatively little difference. This is also partly the case when studying more complex networks, though such a comparison is much more complex.

In all our simulations we have seen that the unloaded latency grows relatively little with an increase in the topology size. This is a very important factor in a SCI-system (cache-coherency operations). An equation giving a first order approximation for the latency with our implementation of k-ary n-cubes is presented. The throughput is dependent on the traffic applied and the capacity of the network.

We have shown that large networks connecting up to several hundred active nodes can be built using simple 2-port switches with very acceptable performance. This thesis has shown some possible solutions to the design-issues involving such networks. Hopefully we have thus made a contribution to future uses of SCI.

We have made a rough comparison of SWIPP and SCI, and shown that both yield an average unloaded latency of roughly 150 cycles for a network consisting of about a thousand active nodes. The latter is assuming that both networks are using switches with high connectivity. Since at present there are no simulation result on the behavior of SWIPP, it is difficult to say much about the performance under load.

Literature

[Agarwal] Agarwal, A. : *"Limits on Interconnection Network Performance"* IEEE Trans. on Parallel and Distributed Systems, vol. 2, No. 4, Oct 1991

[Alnæs et al] Alnæs, K. & Rongved, E. & Kristiansen, E.: *"Chip Set for Scalable Coherent Interface"* Open Bus Systems 91 – Proceedings, page 209

[Baltz] Baltzersen, P. M. Lie: *"Svitsjenoder for et Pakkesvitsjet Multiprosessornett"*. Master thesis (in Norwegian), Department of Informatics, University of Oslo, August 1989.

[Bell] Bell, Gordon: *"Ultracomputers: A terraflop before its time"* Communication of the ACM, pages 27-47, Aug. 1992

[BlekHag] Ingeborg Blekastad & Monica Hagen: *"Protokollmaskin, en kommunikasjonsenhet for en høyhastighets multiprosessor"*. Master thesis (in Norwegian), Department of Informatics, University of Oslo, January 1990.

[Borrill] Borrill, Paul:*"High-speed 32-bit buses for forward-looking computers"*. IEEE SPECTRUM July 1989

[Bugge et al] Bugge, Håkon & Kristiansen, Ernst & Bakka, Bjørn: *"Trace Driven Simulations For A Two-Level Cache Design In Open Bus Systems"* Procceedings of the 17th Annual International Symposium On Computer Architecture, Seattle Washington, May 28-31, 1990

[Carrier] Carriero & Gelernter: *"Linda in context"* Communication of the ACM, pages 444-458, April 1989

[Chaik et al] Chaiken D., Fields C., Kurihara K. & Agarwal A.: *"Directory-Based Cache Coherence in Large-Scale Multiprocessors"* IEEE COMPUTER, June 1990

[Dahl et al] Dahl, O.J. & Myrhaug, B. & Nygaard, K. : *"Simula 67 Common Base"* Norwegian Computing Center, Oslo 1968

[Dally87] Dally, W.J.: *"A VLSI Architecture for Concurrent Data Structures"* ©1987 by Kluwer Academic Publishers, ISBN 0-89838-235-1

[Dally90] Dally, V. J.: *"Performance Analysis of k-ary n-cube Interconnection Networks"* IEEE Trans. Comput., vol. 39, No. 6, June 1990

[Dally91] Dally, V. J.: *"Express cubes: Improving Performance of k-ary n-cube Interconnection Networks"* IEEE Trans. Comput., vol. 40, No. 9, Sept. 1991

[DallySeitz] Dally & Seitz: *"Deadlock-free message routing in multiprocessorInterconnection Networks"* IEEE Trans. Comput., vol. 36, No. 5, May 1987

[DawDob86] Dawson & Dobinson:*"A Framework for computer design"*. IEEE SPECTRUM Oct. 1986

[DawDob87] Dawson & Dobinson:*"Buses and Bus Standards"*. Computer Standards & Interfaces, Vol.6, No.4, 1987.

[ElAmSha] El-Amawy, Ahmed & Shahran, Latifi: *"Properties and Performance of Folded Hypercubes"* IEEE Trans.on Parallel and Distributed Systems ., vol. 2, No. 1, Jan. 1991

[EsvSchrø] Runa Esvall & Anne Schrøder Bonde: *"Protokollmaskinen PE"*. Master thesis (in Norwegian), Department of Informatics, University of Oslo, august 1992

[Feng] Feng, Tse-yun: *"A survey of interconnection networks"*. IEEE COMPUTER, Dec. 1981

[Goor] Goor, A.J. van de: *"Computer Architecture and Design"* ©1989 by Addison-Wesley Publishing Company, Inc.

[Gustavson] Gustavson, David:*"The Scalable Coherent Interface and Related Standards Projects"*. IEEE Micro Feb. 1992

[Gustav] Gustavson, David:*" SCI is Approved"*. An electronic mail distributed to the SCI-community, dated Thu, 19 Mar 1992

[HulBot] Bothner, John W. & Hulaas, Trond Ivar: *"Various interconnects for SCI-based systems"* Open Bus Systems 91 – Proceedings, page 197
Reprinted in appendix C

[HwaBri] Hwang, Kai & Briggs, Fayé A.: *"Computer Architecture and Parallel Processing"* ©1984 by McGraw-Hill, Inc.

[IEEE-SCI] IEEE: *"The Scalable Coherent Interface"* IEEE Std 1596-1992

[James et al] James D., Laundrie A., Gjessing S. & Sohi G.: *"Scalable Coherent Interface "*, IEEE COMPUTER, June 1990

[JohnGood] Johnson, R. & Goodman, J.: *"Synthesizing General Topologies from Rings"* Article submitted to ICPP 1992, Internet: pipe.cs.wisc.edu – /TRs/rings.ps

[Karlsen] Karlsen, Frode Redigh: *"Et høyhastighets, fiberoptisk flerprosessornett"*. Master thesis (in Norwegian), Department of Informatics, University of Oslo, October 1989.

[KerKlein] Kermani & Kleinrock: *"Virtual cut-through: A new computer communication switching technique"* Computer Networks vol. 3, pp. 267-286, 1979

[KrisBotHul] Kristiansen, E. & Bothner, J. & Hulaas, T. : *"Behaviour of Scalable Coherent Interface in larger systems"* Proceedings CAMAC '92, Reprinted in appendix C

[La et al] Larsen, Liao, Lundh, Søråsen & Østby: *"SWIPP - Switched Interconnection of Parallell Processors"*. Department of Informatics, University of Oslo. Internal paper 1991

[Larsen] Larsen, Øystein Gran: *"Development and emulation of interaction mechanisms for a heterogeneous multicomputer"*. Ph.d thesis, Department of Informatics, University of Oslo, October 1991

[Leighton] Leighton, F. Thomson: *"Introduction to Parallel Algorithms and Architectures"* ©1992 by Morgan Kaufman Publishers

[Lundh] Yngvar Lundh: *"Skisse av multiprosessorstruktur"*. Internal paper, October 1987, Department of Informatics, University of Oslo

[Matloff] Matloff, Norman: *"An Argument Against Scalable Cache Coherency"*, Computer Architecture News Jun 01 1991 vol. 19 no. 4 page 117

[NerSmaTor] Nergård, R. & Småstuen, S. & Torp, P. H. : *"SWIPP'em"*. Master thesis (in Norwegian), Department of Informatics, University of Oslo, august 1992

[PatHen] Patterson & Hennessy: *"Computer Architecture: A quantative approach"* ©1990 by Morgan Kaufman Publishers

[Roseth] Roseth, Øyvind: *"Brokobling mellom et multicomputernettverk og et lokalt nettverk"*. Master thesis (in Norwegian), Department of Informatics, University of Oslo, August 1991.

[ScGodVe] Scott, S. & Goodman, J. & Vernon, M..: *"Analysis of the SCI Ring"* Tech report #1055 University of Wisconsin-Madison, November 1991

[ScottGood] Scott, S. & Goodman, J.: *"Performance of Pipelined K-ary N-cube Networks"* Tech report #1010 University of Wisconsin-Madison, February 1991

[Seitz] Seitz, Charles L.: *"The Cosmic Cube"* Communication of the ACM, pages 22-33, Jan. 1985

[Shannon] Shannon, R. E. : *System Simulation, the art and science* ©1975 by Prentice-Hall, Inc.

[ShMayTho] Sheperd, Roger & May, David & Thompsom, Peter : *"Transputers and Routers: Components for Concurrent Machines"* Inmos internal note dated Aug 28, 1991

[Siegel] Siegel, Howard Jay: *"Interconnection Networks for Large-Scale Processing"* ©1985 by D.C. Heath and Company - Lexington Books

[Stenström] Stenström, Per: *"A survey of Cache coherence Schemes for Multiprocessors"* IEEE COMPUTER, June 1990

[Stroust] Stroustrup, Bjarne: *"The C++ Programming Language"* ©1986 by Addison-Wesley Publishing Company, Inc.

[StrouSho] Stroustrup, Bjarne & Shopiro, J. E.: *"A Set of C++ Classes for Co-routine Style Programming"* AT & T C++ Language System Release 2.0, Library Manual

[Tanen89] Andrew S. Tanenbaum: *"Computer Networks"*. ©1989 by Prentice-Hall, Inc.

[Tanen90] Andrew S. Tanenbaum: *"Structured Computer Organization 3rd. ed."*©1990 by Prentice-Hall, Inc.

[Quinn] Quinn, M. J.: *"Designing Efficient Algorithms for Parallel Computers"* ©1987 by McGraw-Hill

[Østby] Østby: *"Definition of some of the cuts in the MultiComputer Network: Draft 2.0"*. Internal paper, November 1990, Department of Informatics, University of Oslo

Index

Appendix

A

Bus standards

SCI is a standard with the scope of replacing buses in many applications. It is therefore natural to take a look at some previous and existing bus standards

Until the 1980's most computer buses were proprietary. But gradually the demand for standardization has come about [DawDob86, DawDob87, Borrill, Tanen89].

For a bus to become a formal standard (standard *de jure*) it has to be approved by one of the standardization organizations. These are both national and international. International standards are produced by ISO (International Organization for Standardization) and the IEC (International Electrotechnical Commision), the latter which has its roots in the industry. Sorting under ISO are all the national members, like ANSI (American National Standards Institute), BSI (Great Britain), AFNOR (France), and DIN (Germany). Under IEC are several national committees. ISO and IEC communicate through JTC1 (Joint Technical Committee on Information Technology). Another major contestant in international standardization is the IEEE (Institute of Electrical and Electronic Engineers)

Standards come about in at least three ways:

- Proprietary buses become standards *de facto*, by widespread use. DEC's Unibus and the IBM PC bus are well-known examples.
- The owners of proprietary buses suggest that their bus be adopted as a formal standard by an international standard organization. This is the case for the GPIB bus, originally developed by Hewlett Packard, the VMEbus from Motorola, Multibus from Intel, Nubus from Texas Instruments, among others.
- Special-interest-groups who are manufacturer-independent take the initiative to develop a standard, under the guidance of an international standard organization. Manufacturers are encouraged to freely use such standards for their products. Since such standards are developed as standards from the beginning, they more easily approach an optimal solution. Examples of such standards include CAMAC, Fastbus, Futurebus, and SCI.

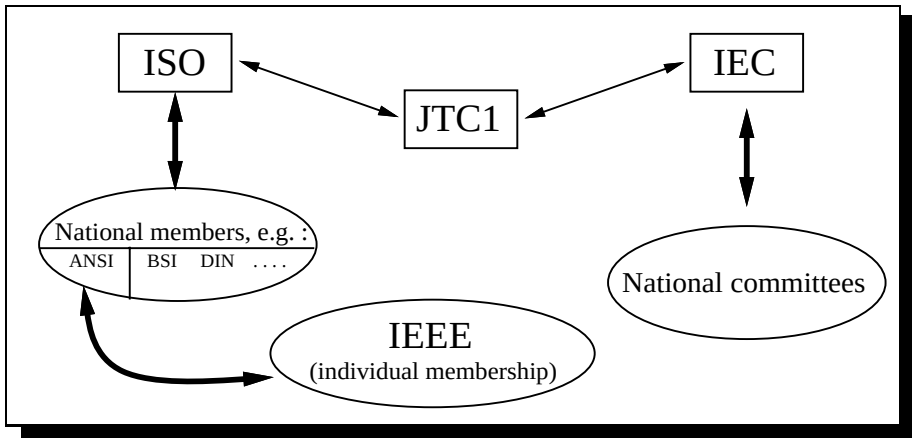


Figure A.1: The relationship between various standard organizations.

There are many standards, partly because different applications require buses with different characteristics, at least if some optimum performance is asked for. Some types of buses:

- A *memory bus* is used to take care of communication between processor and memory. Often called the "internal bus".
- An *I/O bus*, also called a *peripheral bus*, is used to connect a computer system to its discs and tape storage.
- An *instrumentation bus* is a bus tailored for measurement and controll devices in a laboratory environment.
- A *system bus* is a general-purpose backbone bus to connect processors, memory, and peripherals together.
- There are also developed buses for special applications. An example is Fastbus, developed for data acquisition in high-energy physics.

An overview of some of the more successful bus standards:

Unibus was a *de facto* system bus standard, introduced by DEC in the late 1960's for the PDP11. It was perhaps the first widespread bus standard.

GPIB (IEEE 488, General Purpose Instrumentation Bus). It is maybe the most popular instrumentation bus, developed by Hewlett Packard.

CAMAC (IEEE 595,596..., Computer Aided Measurement and Control) is another instrumentation bus, favored for data acquisition in physics.

IBM PC bus is a *de facto* standard for the IBM PC and its clones.

Multibus (IEEE 796) is a popular bus for microprocessors from Intel.

VMEbus (IEEE 1014) is a popular system bus from Motorola. It was originally intended for the MC68000, but is now used in a much wider scope.

Fastbus (IEEE 960) is a high-performance instrumentation bus, developed for high-energy physics.

SCSI (ANSI). Small Computer System Interface. A very popular I/O bus.

S-bus is a *de facto* standard made for SUN's SPARC architecture.

Turbochannel is a *de facto* standard from DEC.

Futurebus (IEEE 896) is a high-performance system bus designed for multiprocessing. It has protocols to maintain cache coherence. Developed by IEEE, it is the first standard to use "hard metrics" [1] in its mechanical specification.

[1] The distance between the connector pins is 2 mm and the board size come in multiples of 600 mm.

B

Tools

There are four different concepts we wish in a language to implement our simulator:

- Object-oriented. The real world is mostly considered to be made up of objects, this is also true for multiprocessors. The following is naturally considered as objects in our simulator: a queue, a fifo-register, a latch, a piece of wire, a complete computer, a set of processors and even a process itself can be viewed as an object. Object-orientation has become very “trendy” in the last years. Nevertheless, here at Institute for Informatics there is a 20-year-old tradition for object-oriented programming! Another advantage with object-oriented code is that it is much more “cleaner”, it is more easily read.

- Capable of partial parallelism. We wish to see how different nodes communicate with each other. These nodes are ”intelligent” devices, at least they are independent of each other. They each take some initiative at ”unpredictable” times. This demands that different parts of our program show a level of independence with respect to each other.

- Acceptable speed.

- Portability. The simulator should execute on various architectures. A minimum was that it could execute on “sparcs” and “dec-stations”, which are the work-stations used at Institute for Informatics.

With these four concepts in mind we have considered the following:

Simula We have a strong tradition for programming in Simula here at the Institute for Informatics. It was developed not too far from here, at “Norsk Regnesentral” in the late ’60s. It seems to be ideal for simulation purposes: it is completely object-oriented, has

coroutines [1], is strongly typed, has a wide range of statistical functions, and has a rather comfortable "run-time-system" (garbage-collector). Its major fault is its relatively slow execution-speed.

C & LWP The Lightweight Process (LWP) Library is a coroutine-library from Sun for programming in a C environment. The main drawback is that C is not an object- oriented language.

C++ C++ is object-oriented and fast. Since it is syntactically similar to C, it has become very widespread in use. For an introduction to C++, see [Stroust].

C++ & Smurph A community at the University of Manitoba, Canada, has made a rather comfortable C++- environment for simulating LANs. The main problem is that it is too comfortable, that is, it is difficult to see what the code is really doing. It is also too LAN-oriented for our purpose.

C++ & tasks The task-library [StrouSho] is useful for implementing coroutines. It is provided with the AT&T CC compiler and the Objectworks (OWC from ParcPlace Systems) programming environment for C++. To our knowledge only available on the "sparc"-architecture.

We early decided to use C++, the only object-oriented language with acceptable execution-speed. The compiler we finally decided to use was "g++" from the "Free Software Foundation". This compiler is available on most architectures.

[1] **Coroutines** is a mechanism which allows the programmer to have multiple "threads of control". The threads runs in a quasi-parallel manner. In other words the thread of control can be switched between the various parts of the program, each with its own stack of private data and program counter.

C

Articles

In November 1991 we attended the conference “Open Bus Systems ’91” in Paris, arranged by the “VFEA International Trade Association. There we had the chance to present some of our work to a wider audience. We also wrote an article which was printed in the “proceedings” of the conference. Since it is related to the contents of this thesis, it is included in the following pages.

Various interconnects for SCI-based systems

Bothner, John Weding & Hulaas, Trond Ivar
University of Oslo / Dolphin Server Technology

September 13, 1991

Abstract

In this paper we wish to show some of the possible ways in which small SCI-rings can be connected together to form large systems of switches, processors and memories.

Choosing a topology is very much dependent of the use of the network. A general multiprocessor requires an entirely different system than for example a data acquisition system. The topologies mentioned in this paper are relevant to general multiprocessors.

The authors are currently implementing a simulation model for some of these topologies as part of their thesis.

1 Introduction

The Scalable Coherent Interface (SCI) – IEEE P1596 – is a high performance interface standard for multiprocessors. The nodes communicate by sending packets on unidirectional point-to-point links. Each link consists of 18 bit-lines, 16 for data, 1 for flag and 1 for clock. The lines between nodes in the figures represent such a link. They are capable of an output of 1 GigaByte/second. [1] The nodes can consist of a processor with no memory, memory banks only, or a mixture of both.

SCI can be configured in many ways. The basic configuration is a simple ring. See figure 1.

[1]For a complete presentation of the SCI-protocols and the physical interface please read [1] and also other papers in this Proceedings.

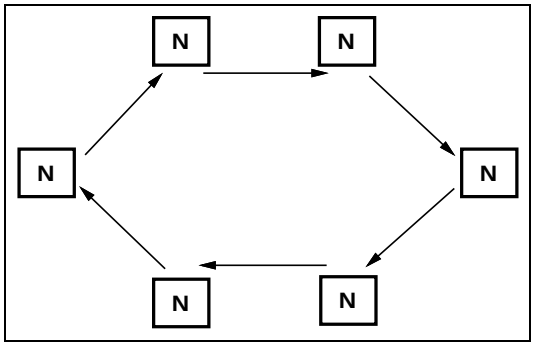


Figure 1: A simple ring-configuration with 6 nodes.

Simulations we have done show that a ring-configuration becomes saturated as more nodes are put on the ring. See figure 2. As the number of nodes increase the data-emission-rate per node will rapidly decrease from the theoretical upper limit of 1 GigaBytes/second. The total throughput of the ring remains stable at about 1.8 GigaBytes/second. The theoretical limit is 2 GigaBytes/second, because on average each packet traverse only halfway around the ring. Figure 2 also shows that transmission-time (queue time in the nodes + transmission time on the ring) obviously increase as the number of nodes increase. The results are preliminary. It does not, for instance take into account the overhead of packet-headers. Also, the results are "worst-case" since the nodes always transmit if it gets the chance. But the trend in the figure is clear enough.

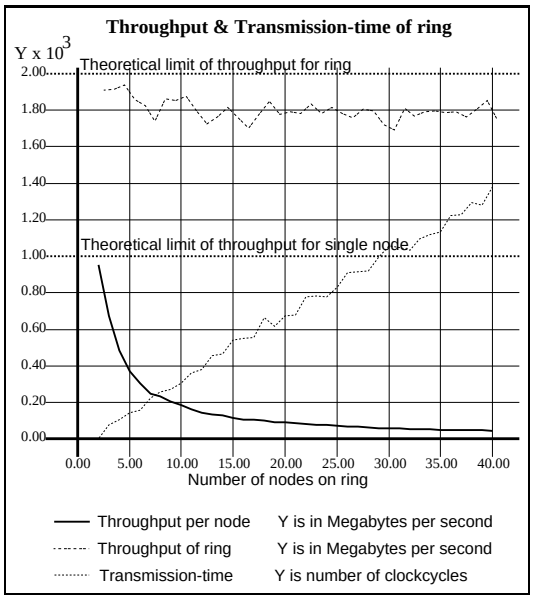


Figure 2: Results of the simulation of a ring.

Thus bandwidth – and transmission-time – limitations become a problem as the number of nodes on a ring increase. The solution to this problem is to have many small rings instead, these being connected together with switches. Figure 3 is a small example of an interconnection network with 7 (very) small rings connected by 2 switches. In this paper we will present various ways of implementing this interconnection network with switches as defined by the SCI-standard. In all the figures an 'N' indicates either a memory-node or a processor-node. An 'S' indicates a switch. Sometimes we will be more explicit and write 'M' for a memory-node and 'P' for a processor-node.

When designing a network for multiprocessors there are several aspects to consider. It is desirable to have a high rate of connectivity (number of hops between any nodes). This can be achieved by either having a large number of links connecting each node or a large number of switches. Either extremes will be rather expensive, so a solution in between should be chosen.

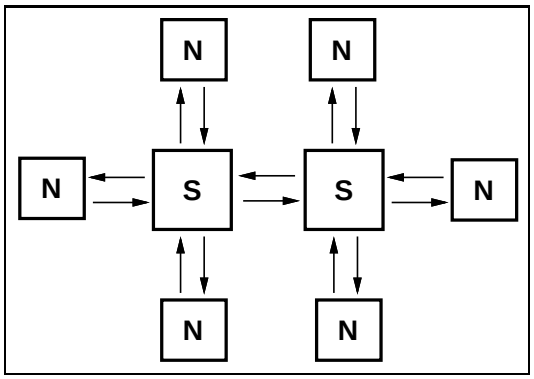


Figure 3: A simple switch-configuration with 2 switches and 6 nodes. All the pairs of arrows is a logical ring.

Due to pinout-limitations there is also a limit to the number of links connected to a switch, if implemented on one chip. Thus the maximum number of SCI-interfaces in a single-chip SCI-switch is probably 4.

We also want to avoid "hot spots". Thus we have to spread the memories among the processors in the network as much as possible.

The switches mentioned here are able to connect 2-4 SCI-rings, thus they have 2-4 SCI-interfaces. The switches are shown somewhat abstractly in figures 4 and 5. Such switches have yet to be designed.

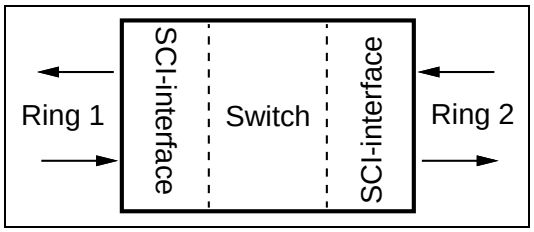


Figure 4: A 2-port switch connecting 2 rings.

2 Topologies

There are innumerable ways of using SCI as an interconnect for multiprocessors. Here we

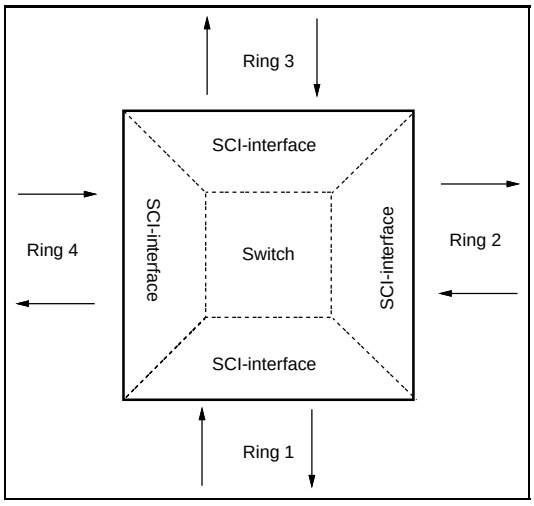


Figure 5: A 4-port switch connecting 4 rings.

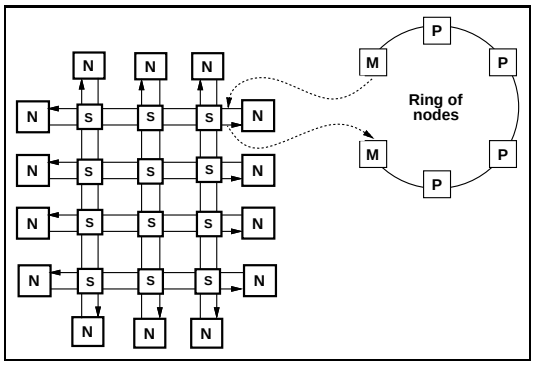


Figure 6: A mesh-configuration. All the nodes can be a single processor or memory, or a ring with both.

will consider three relatively well-known topologies for computer-architectures: meshes, k-ary n-cubes, and hypercubes. These are relatively well understood in the literature, and are scalable.

2.1 Mesh

A possible "mesh" -configuration is shown in figure 6. The switches can either have 2 ports or 4 ports. The 2-port switch would probably be cheapest, and maybe also fastest. With a 4-port switch a packet does not have to risk traversing on average halfway around a ring, but because of some more queueing delay, it is uncertain that it is faster than a 2-port switch. This needs to be simulated before we can say anything of certainty.

In figure 6, 'N' represents a single memory, or a single processor, or a ring of processors/memories/combined.

2.2 k-ary n-cubes

k-ary n-cubes [2], [3]have been used successfully in machines such as the Connection Machine and the Cosmic Cube. It offers the possibility of alternate paths and can be made deadlock free. It is constructed of cubes with dimension n and k switches in each dimension. The relation between dimension (n), radix (k) and total number of switches (N) are as follows:

$$N = k^n, (k = \sqrt[n]{N}, n = \log_k N)$$

Using large radix k (in SCI the number of switches on a ring) with low n makes it easy to construct and scale rather large structures. With low dimensionality the complexity of the switches are reduced, but the number of hops may increase. The opposite, a small k and a large n, may reduce the number of hops but demands more complex switches. It has been shown [2] that low-dimensional networks will outperform (run faster and cost less) high-dimensional networks with the same bisection width[2]. You have three basic k-ary n-cubes:

Unidirectional.

Figure 7 shows a unidirectional 4-ary 2-cube using 2-port SCI-switches. The nodes (processors and memories) can be connected in the shown rings as in alternative 1. Alternatively on separate rings connected to the

[2]The bisection width [4] of a network is the minimum number of wires cut when the network is divided into two equal halves.

4-ary 2-cube with a 2-port SCI-switch (alternative 2).

If 4-port SCI-switches are used then 2 more interfaces ($\Rightarrow$ 2 more rings) per switching node is made available. The memories and processors should then be put on these extra rings.

The constraining of the number of ports on a switch to 4, will allow the use of up till 4 dimensions (n = 4) using an arbitrary radix k. To ensure that this structure can be made deadlock free the number of input/output queues must be larger than one.

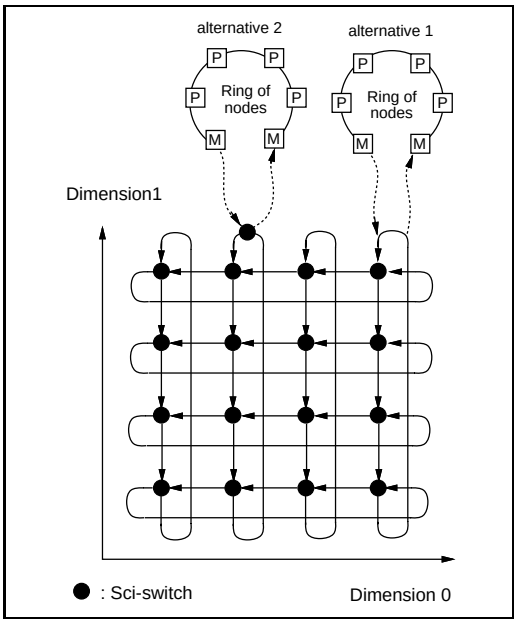


Figure 7: A unidirectional 4-ary 2-cube.

Torus connected bi-directional.

Figure 8 shows a torus connected bidirectional 4-ary 2-cube. It has two channels in opposite directions between each pair of connected nodes. Thus the number of hops from source to destination node can be halved, at the cost of using more complex switches. Another point in using bidirectionality is a better exploitation of the locality of communication. When node A sends a message to node B, node B will always have to send a response message back to A.

It is here necessary to use 4-port SCI-switches in the switching nodes shown in figure 8. In order not to saturate the rings shown only 2-port switches should be added. To these new switches, rings of processor-nodes and memories are connected.

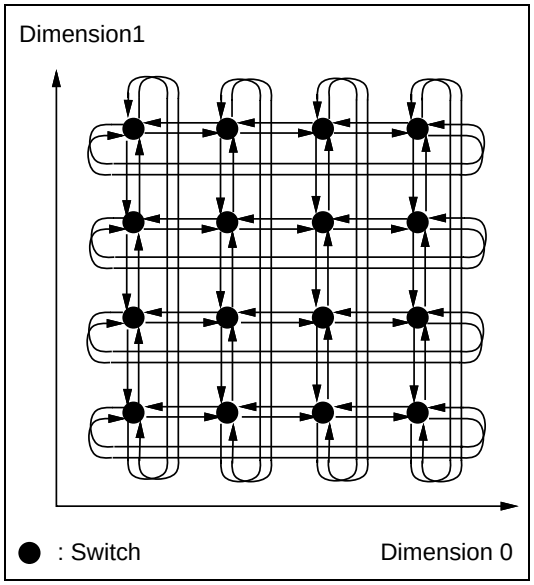


Figure 8: A torus connected bidirectional 4-ary 2-cube.

Mesh connected bi-directional.

If you eliminate the end around links in a bidirectional k-ary n-cube you get a mesh connected k-ary n-cube. But this modification has the penalty of doubling the longest path in the network and also resulting in a loss of symmetry.

2.3 Binary hypercubes

Binary hypercubes is a subset of k-ary n-cubes where k = 2, that is to say there are 2 vertices for each dimension in the cube. There are a total of $N = 2^n$ vertices in a hypercube. That is where to put the switching nodes. Using a 4-port switch implies a maximum dimension of 4 ($\Rightarrow N = 2^4 = 16$ vertices).

To expand beyond 4 dimensions it is possible to combine 2 or more 4-port switches to "build" larger switches. For instance, to construct a switch to connect 6 rings (6 dimensions) one could combine 2 switches as shown in figure 9. The relationship between the number of switches (S) and dimensions (D) is: $D = 2S + 2$. The drawback with such constructs is increased path-length and increased queueing delay.

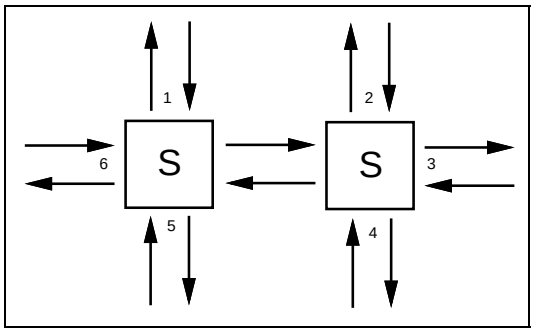


Figure 9: Connecting 6 dimensions ($\Rightarrow$ 6 rings). Note that the connection between the 2 switches is a small SCI-ring, logically.

An interesting variation of the binary hypercube is *the cube-connected-cycle* [5] [6]. At each vertex in the hypercube a ring of 2^r switches is inserted, where r is any integer. Note that if r = 0 we have an ordinary binary hypercube. A decrease in r (for a given number of vertices, and keeping N constant) results in a cube-like appearance, an increase results in a more ringlike appearance.

A possible SCI-implementation of the cube-connected-cycle is shown in figure 10 for 3 dimensions. The "corner-ring" does not necessarily have 2^r switches. Each of the 6 surfaces is a ring. At the corners of the cube 3 surfaces meet. They are connected to a "corner-ring" through 2-port switches, as shown in figure 11. These corner-rings should contain only switches, not processors and memories, so as to reduce the possibility of it becoming a "hot spot".

This is relatively straightforward to extend to more than three dimension by increasing the number of switches on the "corner-ring". For an increase in number of dimensions by one, a new surface is attached to the "corner-ring".

An alternative, shown in figure 10, is to have SCI-rings along the edges of the cube.

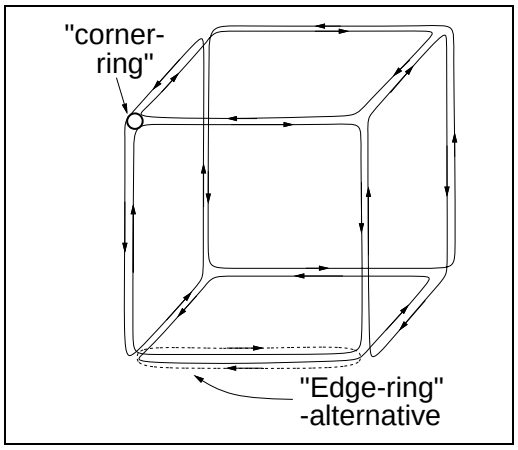


Figure 10: A cube-connected-cycle in 3 dimensions where the 6 surfaces are rings of nodes. The corners are shown in figure 11. An alternative is to have rings of nodes along the edges, instead of in the surfaces. This is shown with a lighter brush in the figure.

3 Conclusion

Simulation-results show that to build large systems with SCI, one ring is not enough. Multiple rings must be used, connected together with switches. Future work will consist of the simulation of these scalable systems of meshes, k-ary n-cubes, and hypercubes.

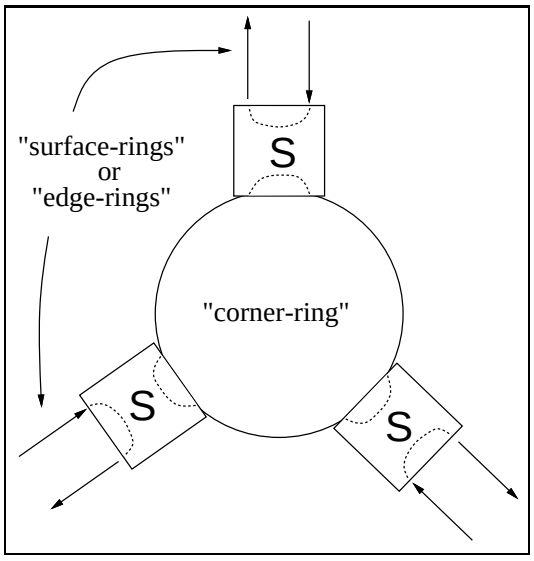


Figure 11: Connecting dimensions in a cybe-connected-cube for 3 dimensions.

If you have any questions, please contact us, either by electronic mail:

trondh@ifi.uio.no & johnb@ifi.uio.no

or by ordinary mail:

Bothner/Hulaas
c/o Søraasen
Institutt for Informatikk
Universitetet i Oslo
Pb. 1080, Blindern
0316 OSLO 3
Norway

4 Acknowledgments

We would like to thank our advisors Ernst Kristiansen and Oddvar Søraasen for their help with the preparation of this article. We would also like to thank Sverre Johansen and Stein Gjessing for enlightening discussions regarding SCI and parallel processing.

References

[1] The Scalable Coherent Interface, DRAFT 1.0, Jan. 1991, IEEE P1596 working group

[2] W. J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks" IEEE Trans. Comput., vol. 39, No. 6, June 1990

[3] Dally and Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks" IEEE Trans. Comput., vol. 36, No. 5, May 1987

[4] C. D. Thompson, "A Complexity Theory of VLSI" Dep. Comput. Sci., Carnegie-Mellon Univ., Tech. Rep. CMU-CS-80-140, Aug 1980

[5] Preparata and Vuillemin, "The Cube-Connected Cycles: A Versatile Network for Parallel Computation", Communication of the ACM, May 1981

[6] Haynes, Lau, Siewiorek, Mizell, "A Survey of Highly Parallel Computing,", Computer, vol. 15, No. 1,January 1982

The following article is written by Ernst Kristiansen. It was published in the proceedings of the conference CAMAC '92 [1] , arranged in Warsawa in September/October 1992. The topic of the article is mostly based on the contents of this thesis. We therefore find it natural to include it in the appendix. It also serves as a rough summary of this thesis.

[1] Arranged by the Polish CAMAC committee. CAMAC is a bus standard (see appendix A).

D

Use and modifications of program

This is meant to be a short introduction to those who wants to use the simulators. It is assumed the reader is familiar with compiling (using Make) and running a program in a UNIX environment.

D.1 Porting programs

The source code is available to all. To get a copy, please contact us.

Most of the code should compile with the g++ compiler from gnu. The X11 window interface demands the Athena widgets library together with a full X11 library release (X11, Xt etc.). The statistical routines is linked from the Simula library. To port to something else, they should be changed to something available on your own system. The class using them is defined in the file *main.H*. It concerns particularly the routines `rnegexp` and `rnormal`. The GNU C++ -library has routines that should be available to most. We advice you use them.

In addition to the simulator there are some programs to generate the appropriate topology-files read by the simulator. The topology-programs (section D.4) should compile easily with the g++ – compiler from GNU.

All delay/time parameters is in clock cycles. Every clock cycle is 2 nanoseconds.

D.2 Options

Each simulator read the command input line to decide it's behavior. The command line options are as follows.

-i *filename* Input topology file. See section D.4 on how to make these files.

-o *filename* Output result file.

-t *number* Simulation time in clock cycles.

-p *number* Number of outstanding transactions.

-j *number* The interval which decides the frequency of packets sent from each node (using uniform distribution). See section 8.6 for an explanation of the p and j parameters. *number* is the upper limit of the interval to draw from.

-x Pops up a window for each SCI-interface on a X-terminal, showing the contents of each interface (node/switch), if program is compiled with X11-options (see section D.5).

-l *number* Simulate with various degree of locality-addressing (see section 9.6). The degree of local addressing is *number* percent. Must be compiled with LOCAL_ADDR_CORNER set (in the makefile). Only implemented in the simulator for virtual-cut-through with extra buffer.

-negx *number* Use the negative exponential distribution to vary loading conditions, with mean set to *number*. Only implemented in the simulator for virtual-cut-through with extra buffer.

-norm *number* Use the normal distribution to vary loading conditions, with mean set to *number*. Only implemented in the simulator for virtual-cut-through with extra buffer.

D.3 Defined constants

In addition to the options previously described, there are several defined constants in the source code. They specify additional parameters. These are collected in the file: *defs.H*. To have any affect on the behavior of the simulator one has to recompile the source code. An exception from this is MAXNODES which is defined in the file: *main.H*.

defs.H: #define CPU_PROS_TIME 100 The time in clock cycles the nodes needs to process an incoming request and produce a response.

defs.H: #define PACK_ARRAY_SZ 10000 For debugging purposes (#ifdef END_DEBUG). To examine packets remaining in the network when the simulation is ended. It is printed into a log-file at the end. Allows 10000 packets to be active and registered in the network at any time.

defs.H:7:#define Link_Delay 2 The delay in time introduced by traversing the links between nodes.

defs.H:8:#define Bypass_Delay 5 The time to pass through the bypass-fifo pluss the 1 extra cycle delay introduced by the fifo itself. The physical delay is then 6 cycles.

defs.H:9:#define Switch_Delay 10 The time to pass through a switch pluss the 1 extra cycle delay for the fifo. The physical delay is then 11 cycles.

#define BIDIR 1 Uses the routing algorithm for a bi-directional k-ary-n-cube, when BIDIR is defined.

#define 2500 MAXNODES is the maximum total number of interfaces. If you wish to simulate larger networks with more interfaces, this constant must be increased. Note then that your computer will have to have a very large memory.

D.4 Making topologies

For a simulator to run, it must have an input-file describing the network. These input-files are made by different, smaller programs, depending of the type of network:

Rings To construct a network with a single ring, use the program `ring-only-gen`.

prompt> `ring-only-gen` x

where x is the size of the ring. The result is a file with name: `ring`x`nodes.top`.

k-ary n-cube with the active nodes in the corner-ring To produce k-ary n-cubes with the nodes placed in the "corner-ring"(see section 7.1.2.1 and figure 7.5a), the program `nodes-in-corner` is used thus:

prompt> `nodes-in-corner -k` c `-n` d `-a` e

where c is the k-value, d is the n-value, and e is the number of active nodes in each vertex. The result is a file with name: `node-in-corner`c`ary`d`cube`e`nodes.top`.

k-ary n-cube with the active nodes in an extra ring To produce k-ary n-cubes with the nodes placed in a separate ring (see section 7.1.2.1 and figure 7.5b), the program `nodes-in-corner` is used thus:

prompt> `corner-gen -k` c `-n` d `-a` e

where c is the k-value, d is the n-value, and e is the number of active nodes in each vertex. The result is a file with name: `corner`c`ary`d`cube`e`nodes.top`.

"Bidirectional" k-ary n-cube To construct k-ary n-cubes with bidirectional links as described in section 7.1.3, use the program `long-switch-gen`:

prompt> `long-switch-gen -k` c `-n` d `-a` e

where c is the k-value, d is the n-value, and e is the number of active nodes in each vertex. The result is a file with name: `long-switch`c`ary`d`cube`e`nodes.top`.

The simulator must then be compiled with `#define BIDIR 1` set.

D.5 Compiling the programs

With each simulator there are two make-files. These files contains the instructions how to compile the complete program using the make program. make allows an automatic compilation of programs. For further information see the manual pages under an Unix system.

One file is for compiling the simulator with the X11 display routines. It is the XMakefile. It uses the AT&T CC compiler. The other one is the Makefile, compiling the program without the possibility to use X11 to display the action. It uses the g++ compiler.

To use either of these makefiles, one types:

prompt> make -f XMakefile
or
prompt> make
(make reads from the file; Makefile; on default).