Contents

List of Figures

Chapter 1

Introduction

This chapter aims to give a short introduction to the terms used throughout the project. Because this project is dealing with the study of interconnection networks with a simulator, this introduction includes basic concepts from interconnection networks as well as from the design of simulators.

1.1 Interconnection Networks

People seem never satisfied with the speed and performance of computers. Nowadays, there are many areas in which problems are not solved quick enough, or not with the adequate precision. This is, for example, the case of weather forecasting. Although processor manufacturers are working very hard on speeding up their designs, this seems never enough. Thus, at some point in history somebody, having this in mind, must have thought *if one computer does not suit my needs, how about two?* This is where the need for interconnection networks comes.

Initially, one would think that having two computers would perform twice as good as a single one, but this is not true. The foundation for this statement is twofold. First there is the unavoidable communication overhead, and second there is the impossibility of adequately balancing the computing load among the computers. As an example we can think of two workers doing a certain job. This job should be divided into work packages and distributed among the two people. Depending on the job division and the dependencies between work packages, there will be times when one worker will be waiting for the other to complete a work package, keeping him from working on the next. This lack of efficiency can be solved, to a certain extent, by carefully studying the job and adequately dividing it. But what can not be eliminated is the time the workers spend telling each other the outcome of their work packages. The people involved with interconnection network design aim to minimize these degradations on the network's

performance.

1.1.1 Network Topology

The case of connecting two computers together is very simple. Only one link is required to transfer information between them. If a higher number of *nodes*[1] is needed, the number of links required to totally connect the nodes grows geometrically as equation 1.1 shows.

$$l(N) = \frac{(N-1)N}{2} \tag{1.1}$$

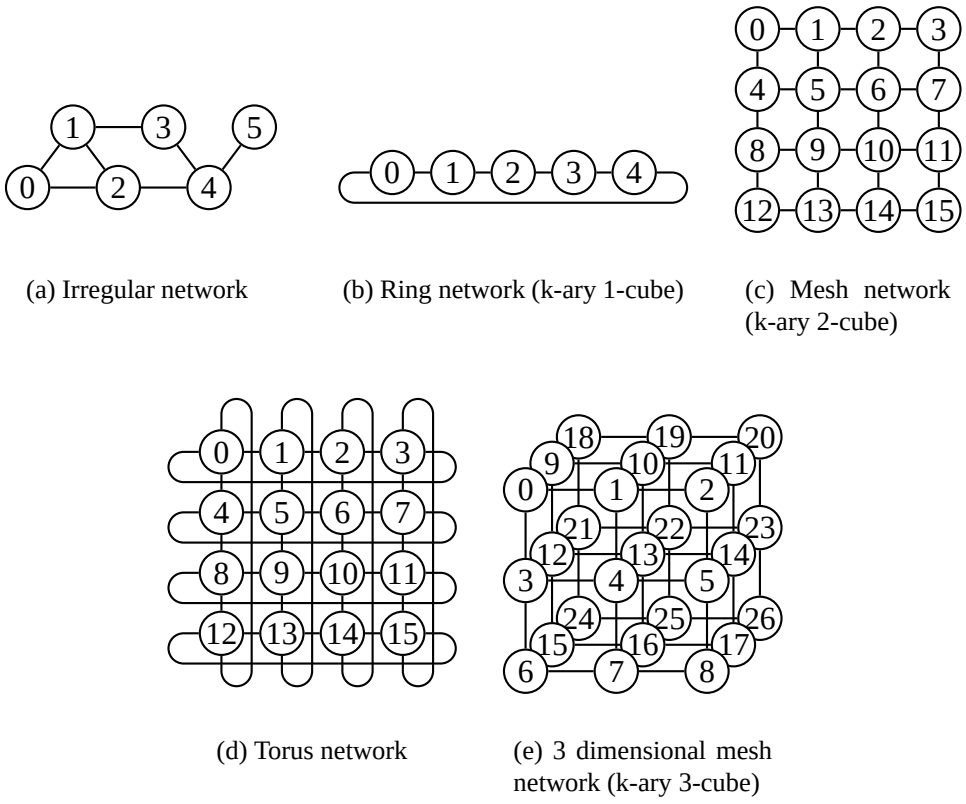
Where l is the number of links and N the number of nodes. For example, to totally connect 16 nodes the amount of links needed is 240. If we think of linking the computers in an office, this does not result affordable. Therefore, there is a need of reducing the network complexity. By adding a special element to every node, a piece of information would be able to be conveyed through the network between two nodes not necessarily directly connected. This special element is able of reading the destination of a piece of information and sending it to another node nearer to the destination if necessary. Such an element is called a *router*, in the sense that it finds routes through the network for the information to travel in.

The addition of routers to a network, enables a subset of the links in the totally connected network to be used. Then each link will not only carry information between the nodes at its ends, but also the information forwarded by these coming from nodes further away. This kind of network, in which the links are shared, are called *switched networks*.

Depending on which subset of links from the total interconnection network is chosen we get different network types or topologies. If the connection scheme does not respond to a certain rule, the network is *irregular* (Figure 1.1(a)), otherwise it is *regular*. In the case of regular networks, there is a further classification; if the view of the network from every node is the same, the network is said to be *symmetric* (Figures 1.1(b) and 1.1(d)) and if not, the network is *asymmetric* (Figures 1.1(c) and 1.1(e)). Also, depending on the number of neighbors a node can have, networks are said to be n-dimensional. Figure 1.1(b) shows a network where each router has two neighbors so it is 1-dimensional. Figures 1.1(c) and 1.1(d) show 2-dimensional networks and figure 1.1(e) shows a 3-dimensional network.

Interconnection networks can also be described by certain figures of merit. These are topological measures that allow to compare, with an objective point

[1] This is a generic way of referring the element that is to be connected. It can be a computer in a local area network, a processor inside a machine, a telephone in a telephone network...



(a) Irregular network

(b) Ring network (k-ary 1-cube)

(c) Mesh network (k-ary 2-cube)

(d) Torus network

(e) 3 dimensional mesh network (k-ary 3-cube)

Figure 1.1: Network examples

of view, different networks. The *diameter* of an interconnection network is the number of *hops*[2] necessary to communicate the two farthest apart nodes. This value can give an idea of the maximal transit time[3] a piece of information needs to cross the network.

The other value, the *average distance*, is the average number of hops necessary to communicate any pair of nodes. Therefore, this value suggests the idea of average transit time[4].

1.1.2 Parallel Computers

The networks treated in this project are intended for the use in parallel computers. A *parallel computer* is a set of computing elements that cooperate in order to

[2]the number of links a piece of information goes through.

[3]Assuming there is no congestion in the network. This is, the information flows without stopping form the source to the destination.

[4]Assuming no congestion and random selection of source and destination nodes.

solve a big problem. In order to cooperate, these elements are connected together with an interconnection network as those described above. However, there must be a higher level mechanism to structure the communication between the different processor elements. Attending to this, parallel computers can be divided into two families. Although the computers in each family have specific hardware particularities, at a software level they are able of emulating the behavior of the other family.

Message Passing Computers

These machines are also known as multicomputers. The main feature is that the programmer has to write in the code when and where should the communication occur. The communication is structured in send and receive primitives. This is done by using special libraries like MPI[5]. The processor elements in message passing computers are normally have all the typical elements of a normal computer (processor, memory, I/O) as well as a connection to the interconnection network.

Shared Memory Computers

These machines are also known as multiprocessors. Opposed to the multicomputers, the programmer needs not to care about how the communication occurs. From the software point of view, multiprocessor machines have one huge memory shared by all the processors of the machine. However, at a hardware level, the memory is divided among the processors and the network provides each with the memory of the others. Typically the processors of a multiprocessor machine share the I/O resources.

1.1.3 Routing

Focusing again on the network infrastructure of parallel computers we have a set of nodes connected to each other with a certain network topology. On each node there is a router unit that enables it to forward pieces of information to other nodes, therefore allowing networks to reduce the number of links. But now there must be a algorithm to allow information to reach its destination quickly. This is called *routing*. The routing algorithm is distributed among all the routers, this is, a router does not decide the whole path of piece of information, it just decides to which of its neighbors it will send the information to.

The routing algorithm must, at least, have knowledge of the network topology and the source and destination nodes of the information. In more complex algo-

rithms, congestion status may also be taken into account. This knowledge must enable the algorithm to find routes through the network.

Depending on which route is assigned to a pair of source and destination nodes, the routing algorithm can either be *deterministic*, in that the information traveling between two given nodes gets assigned always the same path, and *adaptive*, in which information finds its way through the network depending on, for example, traffic conditions.

In addition to getting the information to its destination, it is desirable that this is done fast. Therefore routing a piece of information can be done through a *minimum path*. This is obviously the path that connects two nodes with the minimum number of hops. Nevertheless, there are some networks, such as the irregular, or traffic conditions in which *misrouting*, i.e. routing through non-minimum paths, can give better results.

In this project we will see a static, minimum path, deterministic routing called *dimensional order routing*(DOR). In a k-ary n-cube network, such as a mesh, a piece of information has to move a certain amount of hops in both dimensions, dimension ordered routing forces the completion of the movement in one dimension before proceeding in the next. This eliminates any resource cyclic dependencies between different dimensions.

1.1.4 Flow Control

Up to now we have discussed about 'pieces of information' crossing the network. But this information should be organized somehow. The structure used in this project is as follows, we consider arbitrary sized *messages* divided into a certain number of equally sized[5] *packets*, each of these packets is composed of a sequence of *flits*. Flits are the minimum amount of information the flow control can distinguish. This is, flits travel as atomic units through the network and can not be divided whatsoever. A flit is composed, as well, by a fixed number of *phits*, which are the amount of information the physical infrastructure handles at a time, i.e. the width of the link, and are composed by bits. While the packets in a message can arrive in any order, the flits of a packet must be received properly ordered and not mixed with those form other packets.

Having the idea of the information structuring we can now describe the different flow control functions. On one side we have *store-and-forward*[6], in which a router waits until it receives the end of the packet before sending it to the next router. In the ideal case, the time a packet needs to get to its destination is proportional to its length and the number of hops.

[5]In order to have equally sized packets, the message must be padded so its length is divisible by the packet length.

On the other side we have *worm-hole* [2] flow control. With this function, a router does not wait for the end of the packet to send it over to the next router. The benefit of this approach is that the time a packet needs to get to its destination is only proportional to the number of hops and the buffering space needed is much smaller. However there is a drawback, when the head of the packet is blocked, the packet body stops advancing and blocking some channels from other routers.

There is also a mixture of both functions explained above. It is called *cut-through*[6]. This function behaves like the worm-hole, in that it sends flits to the next router as soon as possible, but once the head of the packet is sent, there must be enough space in the receiver router for the rest of it. This avoids having packets blocking several routers at a time. It can be thought of a worm-hole function when traffic is low and a store-and-forward function when the traffic is high.

None of the flow control functions presented in this project consider packet discarding or retransmission. If the network gets congested no more packets are injected.

1.1.5 Anomalies

When the design of the routing algorithm and flow control functions is not carefully done, there are various anomalies that might occur. The avoidance of some of these is critical as they can draw the network to an unusable state.

Deadlocks

It is said that, when there is a cyclic dependence on resources with no chance of being solved, the network reaches a *deadlock* state. Most of the network topologies used in this project are composed by a set of interconnected rings, therefore they are specially deadlock-prone.

There is a large amount of work on preventing and recovering from deadlocks. Deadlock recovery normally involves packet discarding so we will be concentrating in deadlock avoidance. This can be done by ordering the acquisition of resources in such a way that static dependencies among packets cannot form a cycle. One technique to avoid this involves the use of *virtual channels*. A link with various virtual channels works as various logically independent links. However, at a physical level the virtual channels share the data-path and have independent control signals. To avoid deadlock situations the traffic is split among the virtual channels so that a cycle does can not occur. Nevertheless, this kind of solution suffers from high hardware complexity and nonuniform use of link resources. Therefore other techniques have appeared based on different concepts. The *bubble flow control*, for example, is a low cost deadlock avoidance mechanism. This

consists in restricting the injection of packets to a ring, so that the packets in the ring can move[1].

Starvation

When a resource is to be shared among various consumers, these need to have a fair chance of getting the resource. If not, *starvation* may occur. In this situation, there might be a router that favors some of its inputs and neglects others, causing the latter to wait for and undetermined length of time.

Livelock

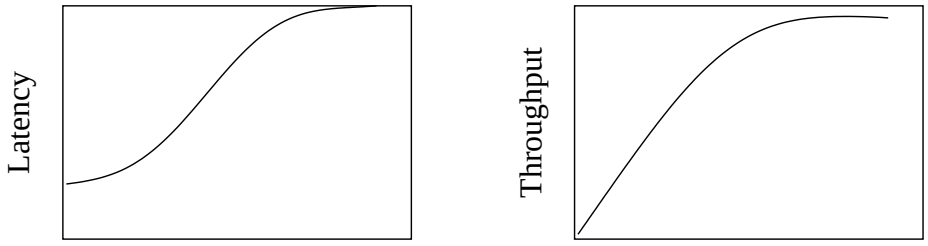
When routing packets through the network, care must be taken in order to ensure that the packets will finally reach their destination and do not wander within the network for ever. This situation is called *livelock* and can occur in when *non-minimum path* routing is used. Opposed to the concept of minimum path routing, non-minimum path routing allows packets to advance in any direction, including those that do not approach the destination.

1.1.6 Network Performance

Interconnection networks are a means to allow information to be sent among a number of nodes. A way of measuring the amount of information that traverses the network is to add up the traffic of every input or output. These will give the units of information (flits, phits, bytes, bits...) per unit time (cycles, seconds...) that enters or exits the network. This is commonly known as input or output load. However, the load is strongly dependent on the number of nodes of the network and it is sometimes interesting to be able to compare networks of different sizes, thus there is also a normalized load that can vary from 0 to 1 independently of the size of the network[4].

Without going into too much detail, there are two parameters that represent the performance of the network. These are the packet *latency* and the *throughput*. Packet latency is the time elapsed since a particular packet enters the network until its last flit reaches its destination. Of course this value is different for every packet, so to give a measure of the overall performance, the mean latency is calculated. The throughput is the amount of information per unit time, that the network has transmitted, i.e. the output load.

The values of mean latency and throughput can be calculated while varying the input load. If this data is represented in two graphs, as in figure 1.2, the relation between both values can be seen. In these graphs, two zones can be observed. A linear zone, when traffic is low. Here the latency is almost flat and



(a) Latency (b) Throughput

Figure 1.2: Typical network behavior

throughput grows linearly. In the linear zone, all packets get to their destination in a reasonable time. The other zone is the saturation zone it happens when traffic is high, here latency and throughput stabilize at their maximum values. This means that the network is not able of managing that amount of traffic, therefore, the message latency and output traffic are constant.

When the input load is at minimum, the latency is also at minimum, this latency value receives the name of *base latency*. It is important because, as there is very little traffic and there is no congestion, it tells the minimum time a packet needs to traverse the average distance network[6].

1.1.7 Network Traffic

When studying the behavior of networks under real parallel applications, it can be observed that the stress put on all nodes is not the same. Depending on which application is running, there are different *traffic patterns* appearing on the network.

Because the cost of simulating real applications is very high, synthetic traffic patterns have been used in order to analyze a network at a lower computational cost. Traffic patterns also provide a 'standard' set of stimuli that can be easily generated. Among others, there are the following traffic patterns.

Random Messages are set between pairs of random nodes.

Local Messages are sent form each node to a range of neighboring nodes.

Broadcast Messages are sent from a node to every other.

Transpose Messages are sent from node (i,j) to node (j,i).

[6] Assuming random selection of source and destination nodes.

If more realistic studies are needed there is also a possibility of using traces. These are files containing the network activity of a real parallel machine. This information is then fed to the simulator to see how the network behaves. This method offers very good results while not being very costly.

In order to simulate real network traffic with higher accuracy, a processor simulator is connected to each node of the simulated network, then a parallel application is run on the processor so that the network traffic can be studied. As this last approach simulates the processors and the network it is more costly but the results have a higher accuracy.

1.2 Simulation

Simulation is a technique that allows a cheap and fast way of studying, tuning and testing real systems without actually building them. This technique has its origins a long time ago, when scientists built scaled models of a system in order to check how it worked before building the real system. Nowadays, with the dawn of computers, these scaled models are represented by mathematical equations and rules inside a computer. Then scientists are allowed to apply some kind of stimuli to the model, in order to understand the system. The computer programs that do this are generically called simulators.

In order to test, tune and compare the performance of the different interconnection networks and routers presented in this project, we need a simulator program. To justify the structure of the simulator that will be used, a brief introduction to some simulation concepts will be given here.

1.2.1 Time Axis

Most simulator programs progress along the time axis performing calculations at certain points until the end of the simulation is reached. The nature of the system to be simulated puts some restrictions to how the time axis is treated. For example, the accuracy when integrating differential equations is highly influenced by the time step used. In the case of the simulation of synchronous digital systems, which is the case treated here, the time axis is discrete, so the simulator must be able of resolving the clock period of the system.

The simulators advance along the time axis can be done in fixed or in variable steps. The variable step approach enables the simulator to *skip* time periods of inactivity reducing the simulation time. In discrete systems this is usually done with events. Events are generated with a certain timestamp in the future and stored in a queue. The events in the queue are processed in timestamp order by the main loop of the simulator. The current simulation time corresponds to the timestamp

of the event being processed in each moment. The processing of an event causes the state of the system to change and more events to be generated.

Event-driven simulators have a drawback, when the activity of the system is high the overhead of creating and sorting events is too big and it is sometimes preferable to use the fixed step approach.

1.2.2 Accuracy vs. Simulation Time

A different aspect of simulator design, is the tradeoff existing between simulation time and the accuracy achieved. This is, the higher accuracy needed, the longer the simulation takes.

When modeling a system, there must be a selection of the aspects that will conform the model. This is, the model should only have the features that determine the functionality of the system, and not others that can unnecessarily complicate it. Therefore, the difficulty of building a model lies on the appropriate choice of the system's features depending on the purpose of the analysis.

Simulation is a numeric technique that performs experiments on a given model in order to extract data that describes the functioning of the modeled system. Normally, a simulation involves a huge amount of operations and variables. Thus, simulation is nearly always done with the aid of computers.

When using a simulator, there are various things the must be kept in mind:

- The execution of a simulation can be very long. Therefore the model must be kept as simple as possible while resembling the system's features in maximum detail.

- As the results depend on the manipulation of random variables. There should be an averaging of many experiments in order to get a proper result.

- Seldom there is a model that mimics the system completely. There is always some detail that is not taken into account. Hence, the results of the simulation are never to be interpreted as the performance of the real system. This error must be taken into account when reaching conclusions out of the results of a simulation.

In general, the error in the results of a simulation grows as the model is simplified. Therefore, at the beginning of the study of a system, the model may be fairly simple, giving high error but short simulation times. And as the study advances, a more refined model can be designed to achieve higher accuracy.

1.3 Motivation and Aims

The simulation of interconnection networks is a key issue if there is a need to study these systems. To this respect, a simulator called SICOSYS was developed and has been very successful. However, the simulator is nowadays obsolete. Although its not structurally limited, the simulation times it offers are too big to make it a useful tool.

Historically the use of super-computers has been something reserved for scientific purposes. Therefore, the study and design of interconnection networks has been done under the scientific application environment. The analysis of this kind of applications has been possible with the aid of execution driven simulators. These are able of executing an application without system code[12] and analyze the network traffic it generates.

Looking at the Top500[13] list, there are starting to appear more and more systems devoted to commercial, finance, database or WWW applications. Thus, there is an urge to study the impact of the interconnection network in these applications. When trying to analyze these applications in the same fashion as with the scientific ones, it has been noticed that those make an extensive use of operating system functions and this can not be overlooked. Therefore, the application can not be run on its own in an isolated environment.

The release of a *complete system simulator* called SimOS[11] has initiated the idea of joining SICOSYS to it. SimOS is a very complex program that simulates a parallel machine completely, it simulates the file system, network interfaces, etc. The virtual machine is capable of running almost any application in a slightly modified operating system. By replacing SimOS built-in interconnection network code with SICOSYS, there would be the possibility of studying real application traffic of almost any application wanted.

One of the aims of this project is to reduce the simulation time of SICOSYS in order to make it a feasible candidate to be joined to SimOS.

The second aim is to allow the study of very big networks. In order to solve very complex problems, the tendency is to use bigger networks with thousands of nodes[7]. The enormous size of these networks cause that small reductions in their topological dimensions make big improvements in their performance. Therefore, making SICOSYS able to test new topologies for large number of nodes is very interesting.

These two aims require an optimization of SICOSYS. This will be done in two ways. On one hand, there will be a thorough analysis of the structure of SICOSYS trying to find ways of making it run faster. These optimizations will not affect the precision of the simulator at all. On the other hand, a way of writing simplified models will be added. This will speed up the simulator greatly at the cost of a loss of accuracy.

1.4 Overview

The contents of this project can be summarized as follows:

Chapter 2 presents the simulator used in this project. This powerful tool has been very successful in the past, but nowadays the network size and complexity has grown to overwhelm the capacity of this simulator. This chapter will describe the general structure of the simulator, point out some problems and propose solutions for them. In addition, a new family of router models will be presented.

Chapter 3 quantifies the improvements in the performance of the simulator caused by the optimizations proposed in chapter 2.

Chapter 4 summarizes the achievements of the whole project and suggests future developments for the simulator.

Appendices are a collection of documents that conform the manual of the simulator. These documents were developed to give an introduction to its usage as well as a exhaustive user manual and an introduction to the programming concepts needed to extend the simulator.

Chapter 2

Simulation Infrastructure

This chapter presents the simulator choice to develop this project. It will be explained why it does not suit the simulation time requirements and how it will be modified in order to reduce this time.

2.1 Simulation and Interconnection Networks

It is frequently seen, in the analysis of interconnection networks, that analytical and simulation techniques often cooperate. The use of analytical tools is compromised by the complexity of the system under study. Though analytical methods are very precise and allow extrapolation and generalization, when the system under analysis is complex, the effort required is too big and the system has to be simplified. But simplifying the system is not always a good approach because then, the analysis does not represent all the features of the real system and the study of these was the goal of the analysis. This is, for example, the case of comparing two alternative systems, the errors that appear when simplifying their modeling can disguise the subtle differences between both alternatives. In cases like these, the only way of obtaining reliable data would be by experimenting with the real system itself. But then again this is not affordable because the system is not built at the time of its analysis. This only leaves the possibility of simulating the system. Though this involves modeling the system as well, the computing infrastructure supporting the simulation process allows a greater approximation to the system than the analytical tools.

Nowadays most of the development of digital systems is done using hardware description languages, such as VHDL or Verilog. The tools available to develop hardware descriptions include simulators that can analyze the system from an early prototype to the final implementation of the system. Because the routers in interconnection networks are no exception, these tools are the most precise way

of analyzing, testing and benchmarking them. However, though hardware description simulators are very precise, the computational cost of performing these simulations is enormous. Hence, the group of Computer Architecture and Technology at the University of Cantabria developed a simulation environment that can perform simulations of interconnection networks with less resources while achieving a very high accuracy. This tool gets the name SICOSYS form "Simulator of Communication Systems" and it will be used extensively throughout the project [9].

2.2 SICOSYS

SICOSYS is a time driven simulator developed in C++ having in mind modularity, versatility and connectivity with other systems. The models used are intended to resemble the hardware implementations in some aspects while keeping the complexity as low as possible. In this way, the simulator mimics the hardware structure of the routers instead of just implementing their functionality.

In order to benchmark the routers and networks against other alternatives and to enable the system to keep up with new developments while presenting a homogeneous user interface, the design of the simulator has payed much attention at its extensibility and the simplicity of user interface. Thus, SICOSYS has a collection of hardware inspired components like multiplexers, buffers or crossbars. Routers can be built by connecting components to each other, as they are in the hardware description.

2.3 Operation With SICOSYS

SICOSYS simulates an interconnection network scenario composed by various elements, these are described in various configuration files. The configuration files are written in a user-friendly format called SGML. The advantage of this format is that it is also processed easily by the simulator. SGML is a structuring language very similar to HTML, in fact it can be thought of as a superset of HTML. A SGML file consists of a set of tags, each with a variable number of parameters. A tag can contain other tags with in it, conforming a hierarchical structure. This suits the the hierarchical descriptions of routers and simulations in SICOSYS perfectly. Every tag in a SGML file has an identification parameter. SICOSYS uses this to make references of a particular tag defined elsewhere.

The description of the simulation scenario is distributed among three files, these are:

`Router.sgm` The router description, indicating the components that conform it, how these are connected and their individual parameters, such as delays,

protocols, etc.

`Network.sgm` The network parameters, including topology, size, wire delay, etc.

`Simula.sgm` The simulation parameters, such as simulation length, seed for random numbers or traffic pattern, injection rate, etc.

In file `Simula.sgm`, there is a tag for each simulation, it includes the definition of various simulation parameters but, within it, there is also a special tag that references the network identifier of the network that will be simulated. This identifier must be defined in the `Network.sgm` file. In turn, each tag in the `Network.sgm` file has a reference to the router that will be placed on every node of the network. The router is defined in the `Router.sgm` by defining and connecting its components together.

To start a simulation SICOSYS must be run from the command line passing it as a parameter the identifier of the desired simulation tag from `Simula.sgm`. SICOSYS then builds a memory representation of the network with multiple instances of the router and feeds it with the traffic pattern until the the simulation time is over. During the simulation, SICOSYS monitors the flow of packets through the network. In this way, it is able to present statistical measures of the simulation when the program ends. Among these, various latencies, throughput or sent and received packet counts can be found.

2.4 SICOSYS' Performance

SICOSYS was designed while searching for a tool that could simulate interconnection networks at a low level. Up to then this had to be done using the hardware implementations of the routers and networks. Although these tools provided the most accurate results, the simulation times needed were very high.

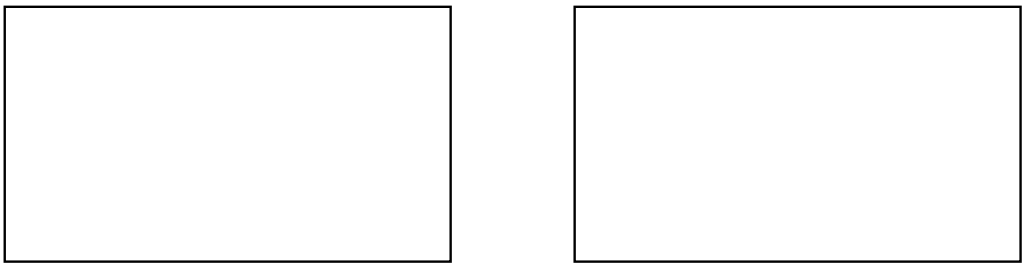


Figure 2.1: Comparison between SICOSYS and a VHDL simulator (Leapfrog).

While keeping certain resemblance with the hardware structure of the routers SICOSYS had simpler components. The behavior of these components was not based on signals and registers as in the hardware implementation. Rather, they where based on higher level elements such as state machines and messages. This approach proved that there is very little accuracy gain when simulating the hardware implementation. Figure 2.1 compares the performance of SICOSYS with that of a VHDL simulator (Leapfrog) using an adaptative router in a 64 node torus network. It can be seen that the error hardly exceeds 3%. This is really outstanding, specially when the speedup is as high as 45. The excellent performance of SICOSYS will allow the comparison of the new simple models against their more complex alternatives already built in it.

2.5 Class Structure

SICOSYS has a large number of classes closely interrelated. Nevertheless, to give a brief idea of how SICOSYS internally represents components and networks only a small subset of the classes must be known. In figure 2.2, a simplified view of the class diagram of SICOSYS is shown.

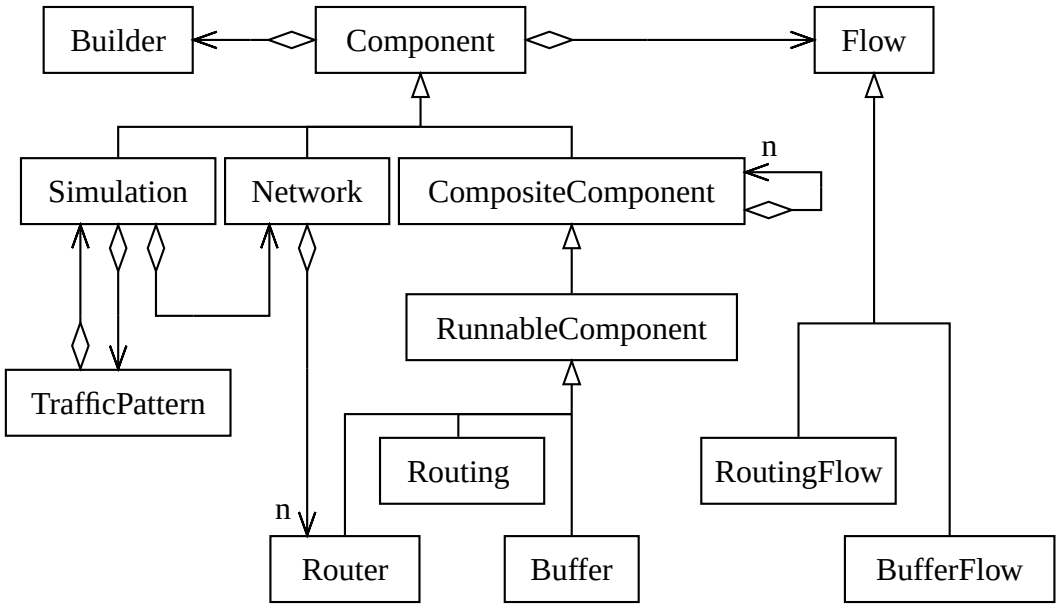


Figure 2.2: Simplified class diagram

2.5.1 Components and Builders

As seen above, SICOSYS reads simulation information from three SGML files. Each of these describe the set of components that will conform the simulation in memory. SICOSYS defines an abstract `PRZComponent` class that will be base class for all the components that are constructed as described in the SGML files. One of the aggregators of class `PRZComponent` is the abstract class `PRZBuilder`, its children specialize in reading one of the three SGML files. Thus, `PRZSimulationBuilder` specializes in reading `Simula.sgm` to create `PRZSimulation` components and similarly happens with `PRZNetworkBuilder`. But a router is defined as a set of components itself, therefore the `PRZRouterBuilder` class specializes in constructing a family of components that derive from `PRZCompositeComponent`. This class is an abstract class that can contain more components, enabling the creation of hierarchically structured components.

Once the building process is finished, SICOSYS has a structure of components that can simulate. The structure of components in memory is as follows. The simulation component has a specific traffic pattern object, derived from `PRZTrafficPattern`, and an instance of a particular network, derived from `PRZNetwork`. In turn, the network object has a set of identical `PRZRouter` objects, one for each node of the network. Each router, as they are derived from `PRZCompositeComponent` has a set of components, such as buffers, routings or crossbars connected together.

2.5.2 Components and Flow Control

As explained above, router components contain many other components that draw up its behavior. Nevertheless, these components do not have the functionality implemented within them. Each component delegates the implementation of the functionality to an aggregator derived from the `PRZFlow` class. In this way a component can have a variety of flow control classes that implement different behaviors of the component. For example, the class `PRZFifoMemory` has the `PRZCTFifoMemoryFlow` class to implement the cut-through behavior and the `PRZWHFifoMemoryFlow` class to implement the worm-hole behavior.

2.5.3 Message Life Cycle

Once the building process is done, the simulation can start. During simulation, messages are created, sent to the network and received. SICOSYS has a class `PRZMessage` that represents the information flowing through the network, it can represent either a single flit, when it is in the network, or a full message, when it is generated by the traffic pattern class.

In the simulator main loop, there is a call to the traffic pattern object to generate a message for each router, this message is represented by a single `PRZMessage` object. The message is sent to the network, then to the appropriate router and then enqueued in the injector component within the router. The injector reads the message object and finds out how many packets and flits it has. Then, it sends flits, each represented by a new `PRZMessage` object, through its output.

The `PRZMessage` object is copied from the output of a component to a `PRZ-Connection` object and then to the input of the next, eventually crossing to other routers and finally reaching a `PRZConsumer` object. This component notifies the network the arrival of the flit in order to perform statistical calculations.

2.6 SICOSYS' Improvement

Now the needs have gone beyond the capability of SICOSYS. On one hand, the demand of higher computing power is forcing the study of bigger networks, connecting more and faster processors. On the other hand, the use of synthetic traffic patterns is giving way to the use of real application traffic. With this approach a processor simulator[8] is connected to each node of the network, then a parallel application is run on the processor simulators so that the network traffic can be studied. Up to now SICOSYS has been able of running various benchmarks from the SPLASH2 suite[12] on 64 processors. However, SICOSYS faces a new challenge, the integration with SimOS[11]. This is a simulation infrastructure that is able of completely simulating a parallel system.

In order to fulfill these new challenges, SICOSYS needs to overcome some modifications. First of all, the internal structure shows a time consuming design flaw that will be solved, and second, a new family of lighter models will be developed for faster performance.

2.6.1 Optimization of Internal Structure

To perform the simulation of a given network and router, SICOSYS builds in memory a representation of the problem with a set of components. When this is done, it injects flits into the network with a probability given by the current traffic pattern. Once in the network, the flits progress from router to router and within each router from component to component until they reach the consumer at their destination.

In order to move a flit from one component to the next, the destination component makes a copy of the flit held by the source component. This way of sending the information through the components is seriously time consuming, as each time

a message is passed to another component, the copy constructor of the class is executed. A close look at the nature of flits suggests that the copying mechanism does not seem right. The flits are created at a node and progress through the network to their destination node almost unchanged, only some routing information is changed. Therefore it was thought that flits should be created once in the source and not destroyed until they are consumed in their destination. Thinking in terms of programming language, the components should be passing each other references of flits instead of the actual flits.

However this approach has a drawback because dynamic allocation of the flits is needed. Before, the flits were kept by the components. They had attributes that held a copy of the flits they were processing. But now these attributes have been changed to pointers and the flits must be dynamically allocated once at the source and deallocated at the destination. At first this seems no problem, but the amount of flits needed in a simple simulation can be enormous so dynamic allocation can slow the simulator down as well as make it use the memory of the machine inefficiently.

The storage space within a router is fixed. The size of its buffers and the number of flits it can hold at a time can be determined beforehand. This fact allows the calculation of the maximum number of flits that will be in the network at a given time. Bearing this in mind and knowing that the size of the flits in memory is also constant, a work around to the dynamic allocation problem can be designed. At the beginning of a simulation a pool of flits is created. This pool is implemented by a linear array of flits. Then the the flits are allocated from the pool instead of performing a normal memory allocation from the heap. The improvement of this approach is twofold. On one hand the search for a free flit in the array is much quicker than performing an allocation system call. And on the other hand, having all the messages neatly organized in a single array makes a better use of the memory.

The size of the pool is calculated at the beginning of the simulation. Once the size of the network is known, a method of the router tells the amount of flits it can hold. By multiplying the size of the network by the storage space of a router the total amount of flits that can be in the network is obtained. Then the simulator creates the pool two or three times bigger. This is to make room for the packets in the injector queues. When the pool is full the new messages are discarded, as they will never reach their destination.

In order to allocate elements in the pool, an index shows the last allocated element. When a new flit is needed, this index is increased until a free element in the array is found. In the case of reaching the last element of the array, the index is wrapped to the first position and the search continues. Initially this could lead to time consuming searches through the array. However the nature of messages is to have an average *lifetime* with small variance. This fact showed that the usage of

the array is quite well organized and by the time the index goes around the array, the initial flits are already deallocated. Figure 2.3 shows a typical allocation status of the pool. The flits allocated for the packets traveling in the network are close behind the index are normally free.

Figure 2.3: Typical allocation status of the flit pool

In addition to these major changes, there was a minor change in the management of collections that actually proved it self very convenient. SICOSYS has its own dynamic data structure classes, such as queues or stacks. These are implemented as doubly linked lists. However, only the first element of the list is referenced by the collection class. This becomes a problem when large queues are used, in which new elements are added to the beginning of the list but extracted at the end. As no reference of the last element is kept, the end can only be found by traversing the whole list. This was easily solved by adding a reference to the end element in the collection class. A graphical representation of this modification can be seen in figure 2.4, where the new reference is drawn in a dashed line style.

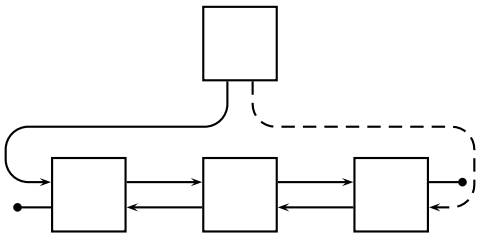


Figure 2.4: Modification of collection classes

2.6.2 Model Simplification

Although the modifications in the internal structure proved very good, there are some cases in which there is a need for faster performance and a bigger error is not a problem. For these cases a new family of lighter models are designed.

Having in mind the resemblance with hardware and the modularity of the simulator, routers are built as an interconnection of several components. This approach has been very successful and reliable, but now that there is a need for

shorter simulation times it might be interesting to head for the implementation of routers in a more compact manner. Packing the totality of the router in a single component could drastically reduce the computational overhead of messages entering and exiting the components as well as reduce the size of the memory representation of the router. Nevertheless, this monolithic scheme is bound to show a loss of accuracy compared to its predecessor. In order to establish limits to the accuracy of this new approach, two routers will be implemented: a simple router, to show the upper bound of the accuracy, and a complex router, to give the lower bound.

The `SFSSimpleRouter` Component

In SICOSYS, components are connected to each other through special objects. Inputs and outputs are modeled by an association of two components. First the `PRZInterfaz` handles the multiplexing and demultiplexing of the various virtual channels associated with an input or output. Second each virtual channel is represented by a `PRZPort` object. Then each pair of input and output ports of the same virtual channel are connected by a `PRZConnection` object. Figure 2.5 shows the structure of a connection between two components with three virtual channels.

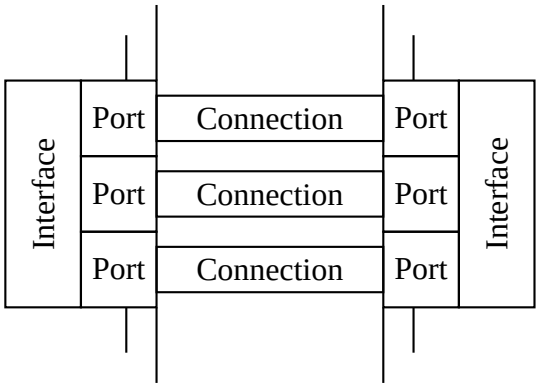


Figure 2.5: Structure of a connection between two components with three virtual channels

The new routers are implemented as a new component that lies within a classic SICOSYS router as usual. As shown in figure 2.6 all the components of the router are grouped into a single one. The reduction of inter-component communication is clear.

As the injector and consumer componets are responsible for the generation and destruction of the flits, they are likely to be modified in order to connect

SICOSYS to other simulator programs. Therefore the injector and consumer are the only components of the router that are not bundled in.

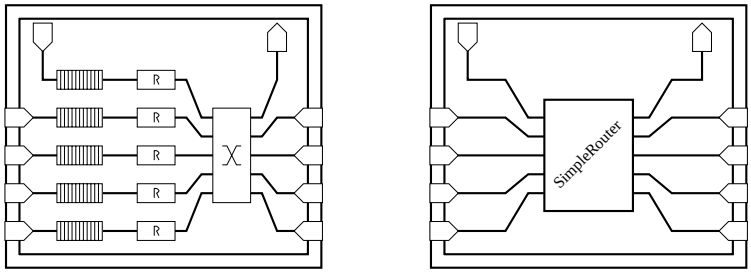


Figure 2.6: Substitution of the router components by a single one

The `SFSSimpleRouter` class can be thought of as an empty shell that only has the ports to communicate to the outside world. The totality of the router model is contained within the flow control class. This is done to enable fast and easy manipulation of the flits within one class and not bind future developments to a particular component architecture.

The Simple Bubble Router

The simplest router considered is a cut-through bubble DOR router[1]. Figure 2.7 shows its internal structure. This router has one virtual channel. Incoming flits are temporarily stored in input buffers until there is a chance to progress into a routing component. There, the output port through which the flit should be sent is calculated. The crossbar module tries to transfer as many flits as possible form its inputs to their corresponding outputs at the same time.

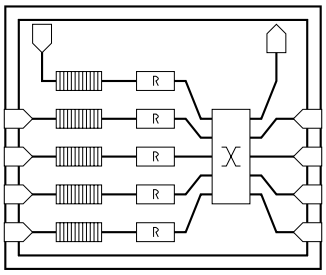


Figure 2.7: Diagram of the simple bubble router

The cut-through flow control is implemented so once a route is granted to a packet all its flits must follow. To achieve this behavior, when a buffer has space for no more than one packet it signals it to the previous component. If a component receives such a stop signal form its receiver counterpart it knows that it can still

send the remaining flits of the packet, but can not start to send a new packet. This allows packets to be transferred completely from a router to the next.

Deadlocks are avoided in two ways. First, multidimensional dependencies are solved using DOR routing. Second, deadlock within a ring in a single dimension is prevented by the bubble algorithm [1], this restricts the access of packets to a new dimension favoring the mobility of the packets that are already in it.

The Complex Adaptative Router

To calculate a lower bound of the accuracy of the simplified model approach, a complex router has been implemented. This router was designed to offer best characteristics for increasing system performance while being able to be integrated within the processor die. Thus it is a serious candidate for providing microprocessors with on-chip network router [10].

The *Head-of-Line Blocking*(HLB) effect is is a problem form which many routers suffer. Routers normally store incoming packets in input buffers in a FIFO fashion. If the head of a packet get blocked because the output it requires is unavailable, the packets that follow it can not advance even if their outputs are free. To reduce the head-of-line blocking effect, the router has smaller buffering at the input and uses multiport memories at the outputs. This allows that the blocking of one of the outputs does not disturb the packets that head for the other outputs. In order to improve the router performance at high loads, it implements an effortless path selection function based on credits. This adequately balances the network traffic increasing throughput. The internal structure of the router is shown in figure 2.8.

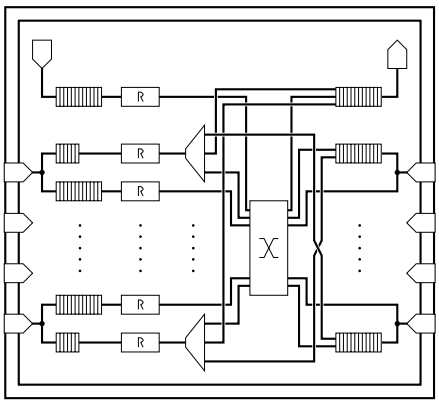


Figure 2.8: Diagram of the complex adaptative router

The router works with two virtual channels, an adaptative and a deterministic one. The deterministic part of the router is very similar to the simple router as it

is based on a crossbar with DOR routing and it has the bubble deadlock avoidance mechanism. However, the adaptative part is rather different. The crossbar is substituted by a set of demultiplexers. These send the incoming packet to the less occupied output that makes it approach its destination, i.e. balances traffic and keeps minimum path. To avoid deadlocks in this part, the packets that can not progress through any of the adaptative outputs are sent to the deterministic virtual channel, also called escape channel. Being this last deadlock free confers this virtue to the adaptative channel as well [3].

Chapter 3

Performance Analysis

The previous chapter has described the different modifications that SICOSYS has overcome in order to become a useful tool for the future. This chapter will quantify the improvement levels obtained by such modifications.

3.1 Optimization of Internal Structure

The internal structure of SICOSYS has three major changes, the management of messages by pointers, the improvement in the queue implementation and the use of a message pool. To visualize the effect of the different changes, four snapshots of the code have been used to simulate a wide selection of situations. The four snapshots represent the state of SICOSYS through its development as follows.

a) is the original simulator with no changes at all.

b) is a simulator which manages the messages with pointers, though it uses `new` and `delete` for their allocation and deallocation.

c) it includes the new queue in which the final element does not have to be searched for.

d) is the final simulator with the message pool that reduces the cost of the allocation and deallocation of the messages.

These four simulators fill be fed with a wide range of simulations, spanning typical simulation scenarios. Three torus networks with sizes 4, 64 and 1024 nodes have been used. The message injection is done with a random pattern with normalized load of 0.01, 0.2, 0.5 and 0.9. Each of these environments is simulated for 50000 and 200000 cycles.

The simulation time of each modified simulator (b,c,d) is compared to that of the original (a) and put in graphs. Figure 3.1 shows the speedup of the simulator with a deterministic bubble router.

The graphs show the improvement of the simulation time obtained by each modification in different situations. In general, it can be observed that with low load the most significant improvement is the manipulation of messages with pointers (b), this is because the messages spend most of their lifetime traversing the network and do not wait a long time at injection queues. However, at higher loads messages get held for long times in the injection queues, causing these to grow unlimitedly. Therefore, especially at very high loads, it is the new queue (c) that speeds up the simulations most. However, when bigger networks are simulated the major improvement is provided by the message manipulation (b).

The modifications in the internal structure have lead to cut down simulation times at least in 80% (1.25 times faster) when low load cases are simulated, and down to 1.4% (70 times faster) when small networks are simulated for very long. It must be noted that the accuracy of the simulator remains untouched, as only the simulator's structure has been modified and the models are kept the same. In addition, the amount of memory used by the simulator has not diminished either.

3.2 Model Simplification

Even though the performance of the improved simulator is good, there are occasions in which there is a need for faster simulations. To this respect a new way of modeling routers was posed. It was capable of grouping all the components of a router in a single one. Thus reducing the overhead of inter-component communication.

In order to evaluate the effectiveness of this new approach, its accuracy bounds must be found. Therefore, two routers have been modeled, a very simple one and a complex one. This section will try to quantify the advantages in the performance of these new models and measure the loss of accuracy they exhibit.

To achieve this, several sets of simulations have been performed with the two versions of both routers. The selected simulations try to cover typical simulation scenarios.

3.2.1 The Deterministic Bubble Router

The simplicity of the bubble router leaves little room for optimizations and at the same time allows a simplified model to approach the real router fairly well. This fact can be observed in the following graphs (See figures 3.2, 3.3 and 3.4). Each

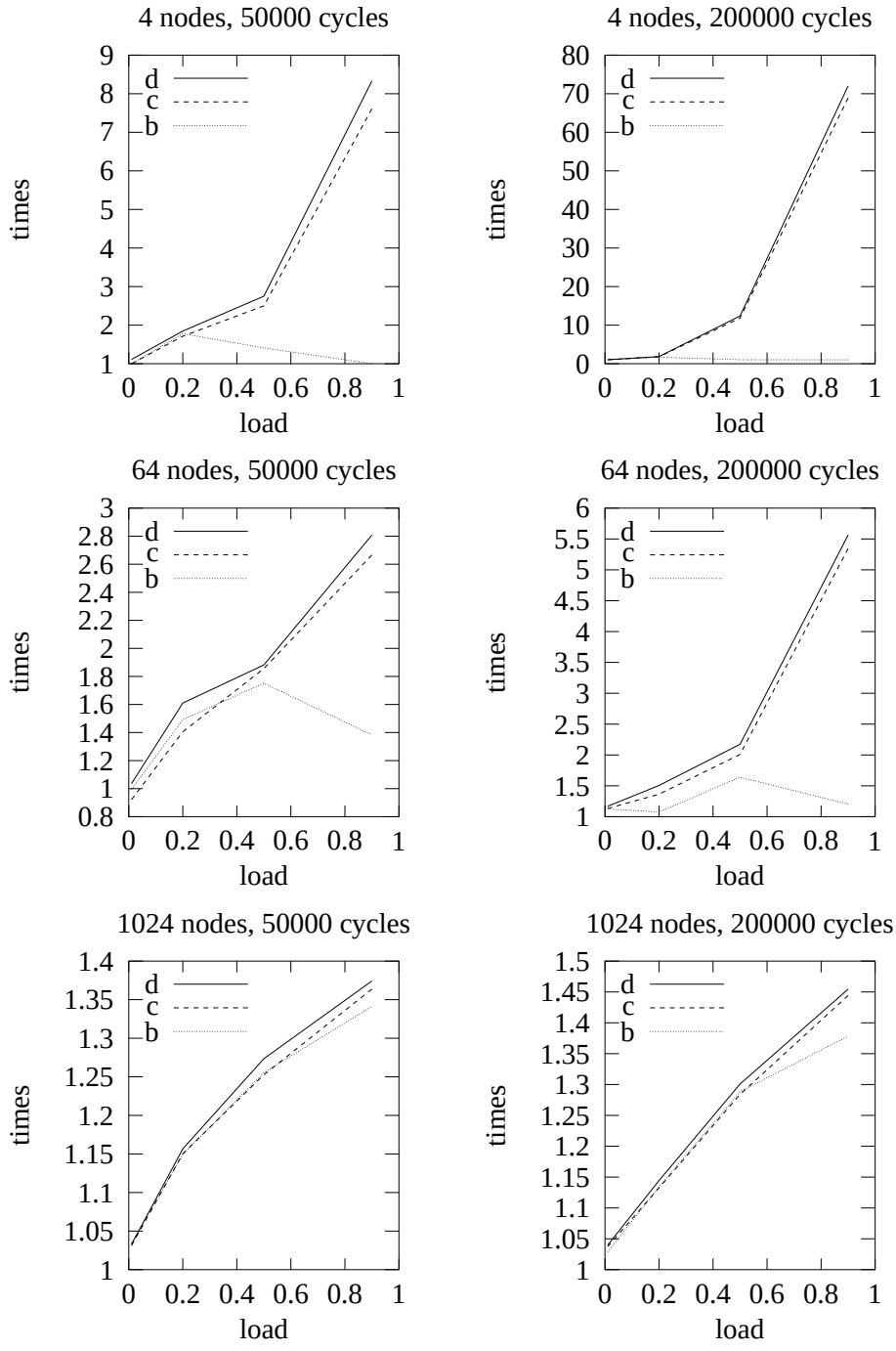


Figure 3.1: Improvement of SICOSYS's performance with a DOR bubble router

figure represents the results obtained from a set of simulations with a torus network of a particular number of nodes that spans different load conditions. Within each figure, there are four graphs. The latency and throughput of each model is represented by the leftmost graphs. On the right the simulation time improvement (speedup) is at the top and the relative error of both, latency and throughput, is at the bottom.

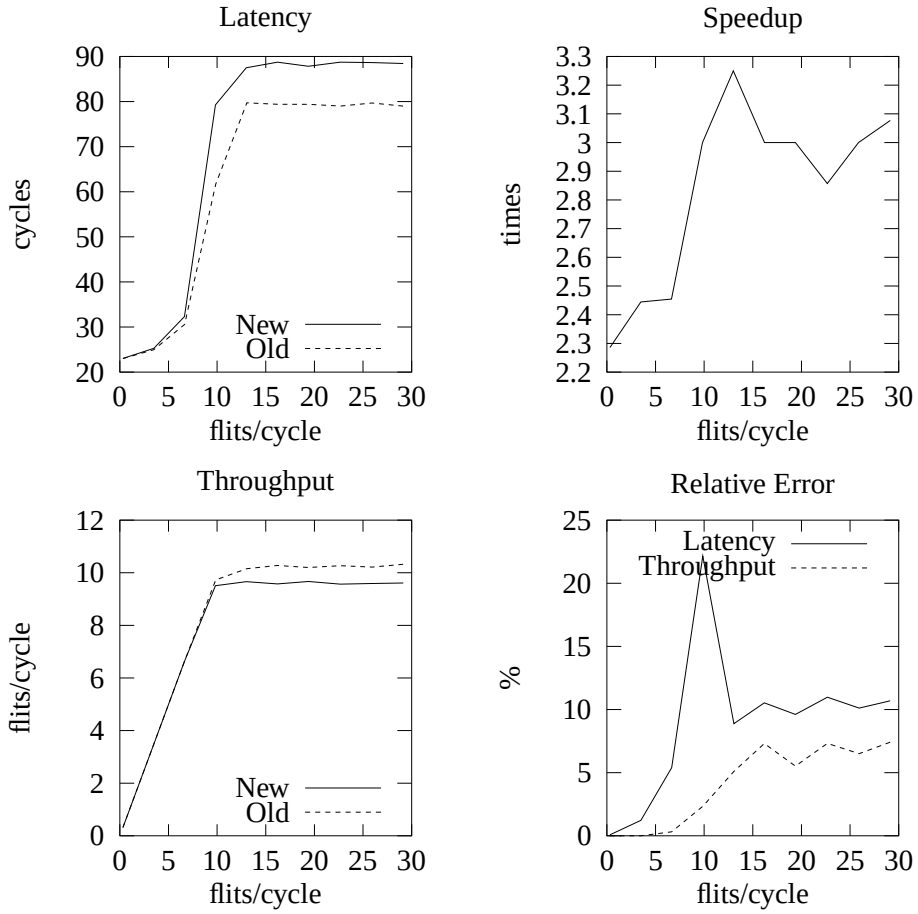


Figure 3.2: Performance of the simplified model of the deterministic bubble router. 16 node torus network.

When constructing the new model, a special effort was put in adjusting the base latency. This is the latency of the router when packets flow without contention. Thus the latency at low loads has zero error. Nevertheless, the simplification of the model does not mimic the crossbar precisely and, as load increases, the discrepancy in the latency grows. The latency error becomes maximum when

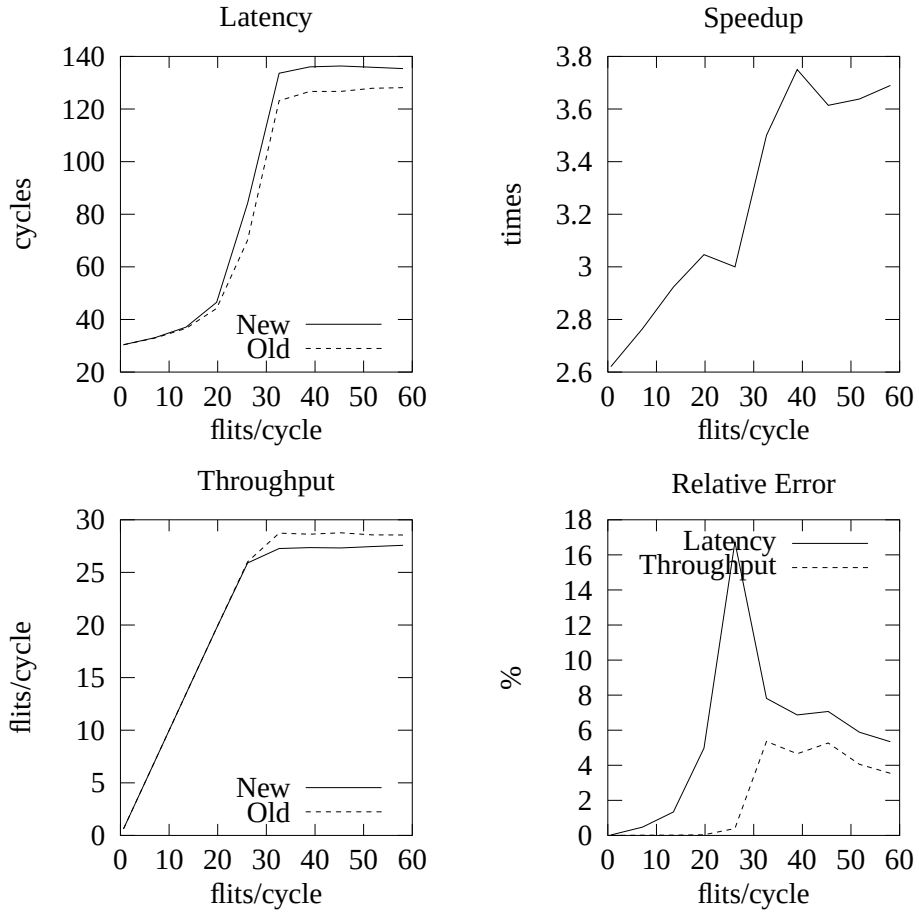


Figure 3.3: Performance of the simplified model of the deterministic bubble router. 64 node torus network.

the network is between the linear and the saturation areas. This is due to the huge slope of the latency in this area that provokes big errors even when minor details of the router are left out. However the error decreases just as drastically to a reasonable value when the network is saturated.

On the other hand, the throughput has a different behavior. When in the linear zone, the throughput is equal to the input load, this must be true for both routers. However, when approaching the saturation zone, a discrepancy between both routers appears. Unlike the latency error, the throughput error grows slightly and stabilizes in the saturation zone. As can be seen, the results obtained by the simple model are somewhat pessimistic. This means that the details left out by the simple model make the router perform worse and thus give an upper bound for the latency and a lower bound for the throughput. So being the average speedup

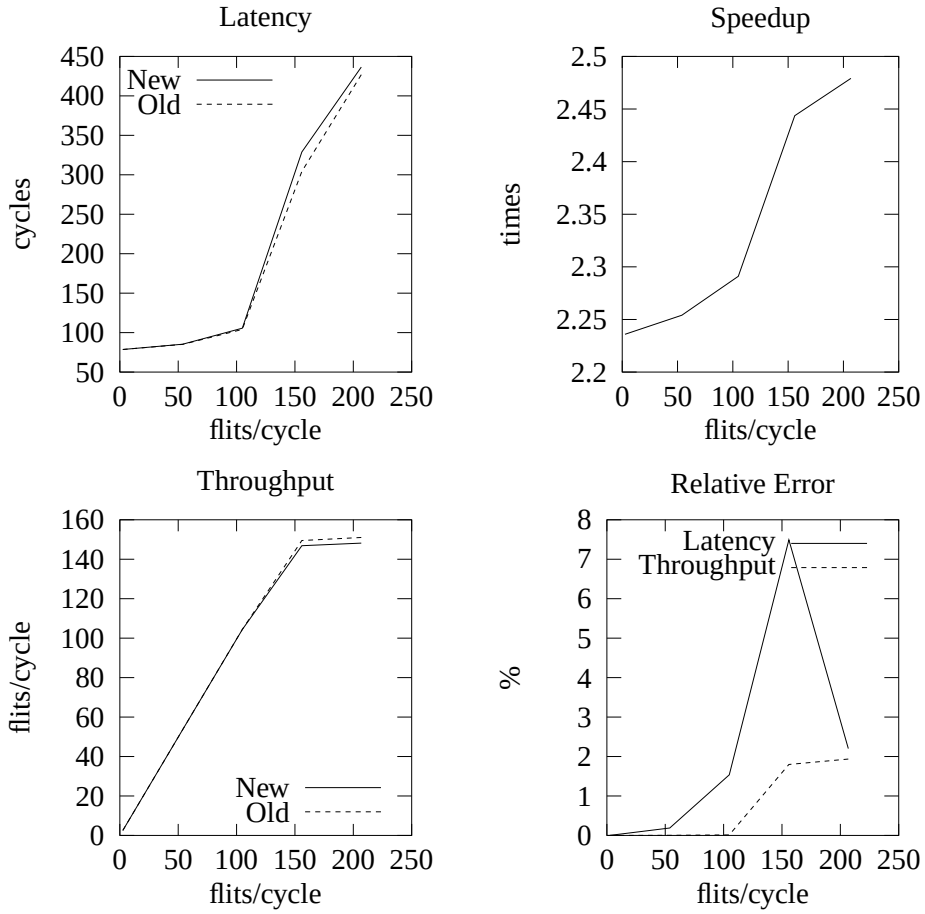


Figure 3.4: Performance of the simplified model of the deterministic bubble router. 1024 node torus network.

around 3, a worst case bound of what the real router will surely achieve can be obtained saving 66% of the simulation time.

In addition to the speedup reached by the new model, there is also an interesting parameter that is also reduced. This is the memory usage of the simulator. The reduction of the size of the simulator can make it perform even faster. As there is a bigger chance of fitting the process in the L2 caches of the machine, a reduction of time-consuming cache faults can be noticed. When running SICOSYS with the new router with various network sizes an average 80% reduction in memory usage was observed in the size of the simulator.

3.2.2 The Complex Adaptative Router

To get an estimation of the upper bound of the error a complex adaptative router has been implemented. The complexity of it leaves a lot of room for simplifications and, therefore, the saving of a lot of simulation time and memory.

In fact the adaptative router has many components and connections within it. When using the new monolithic scheme, all the connections and interface objects are eliminated. In addition, the individual models of all the components are compound in a single model. This enables the drastic simplification of the router, cutting down simulation time and memory usage. Then again, the error that this router is bigger than in the deterministic one.

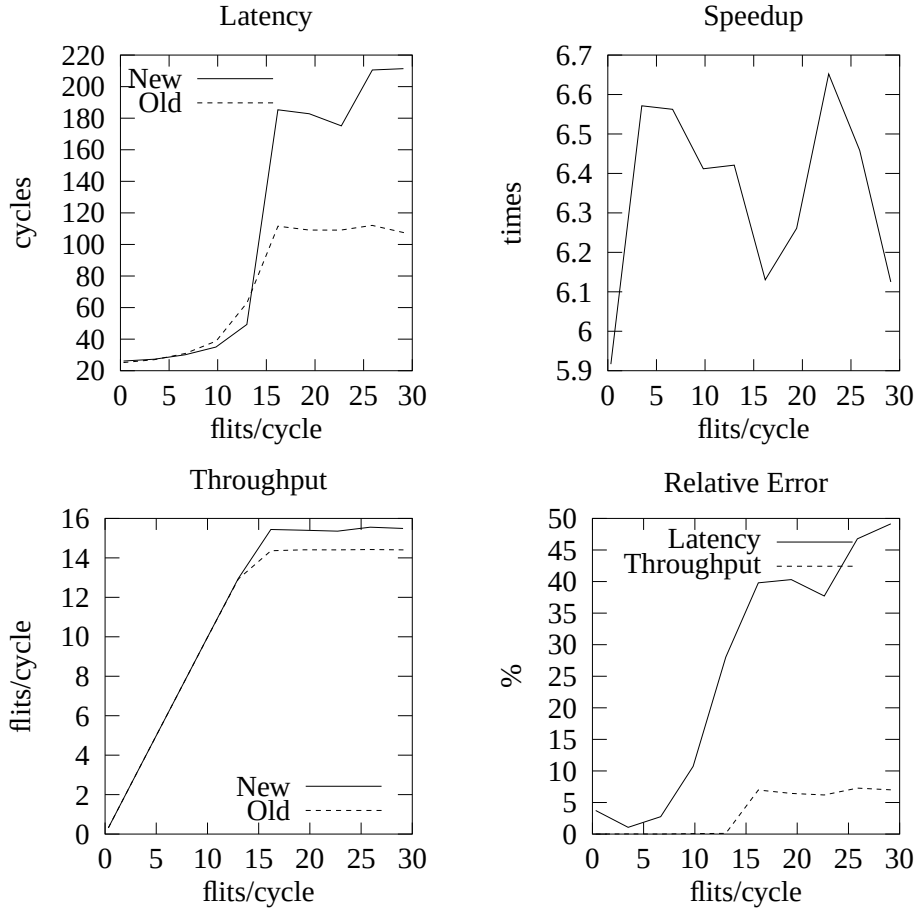


Figure 3.5: Performance of the simplified model of the adaptative router. 16 node torus network

To profile the adaptative router a smaller range of network sizes have been used. The cost of simulating a 1024 node torus network with the old adaptative

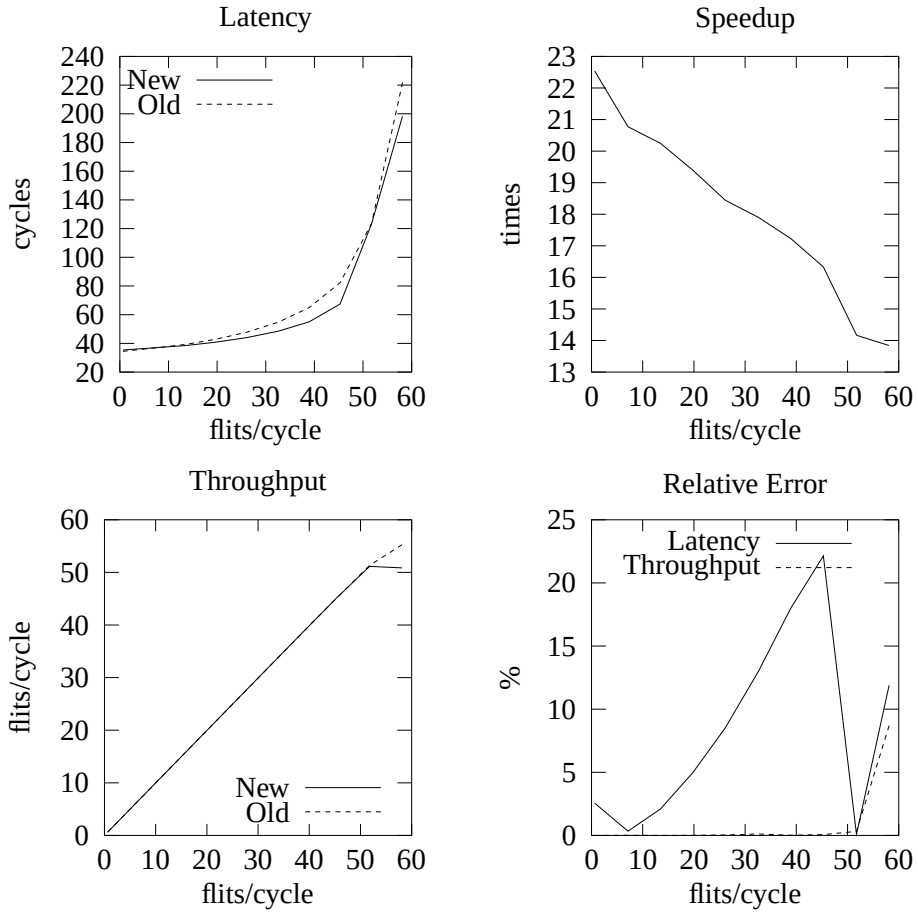


Figure 3.6: Performance of the simplified model of the adaptative router. 64 node torus network

router would be too big. Thus, the chosen network sizes are 16, 64 and 256. The results of the simulations are shown in figures 3.5, 3.6 and 3.7.

Just as with the deterministic router, the design of the internal structure payed special attention to the base latency. This fact is exposed by the the results of the simulations. The error in the latency is very small at low loads. However, as the load is increased the discrepancy between both routers grows. On the other hand, the throughput of both routers necessarily follow each other when working in the linear zone, but when in the saturation area, the new router shows a different performance.

As was expected, the error shown by the adaptative router is somewhat bigger than in the deterministic one. This is caused by the stronger simplifications made to the model to enable its monolithic implementation. It must be kept in mind that,

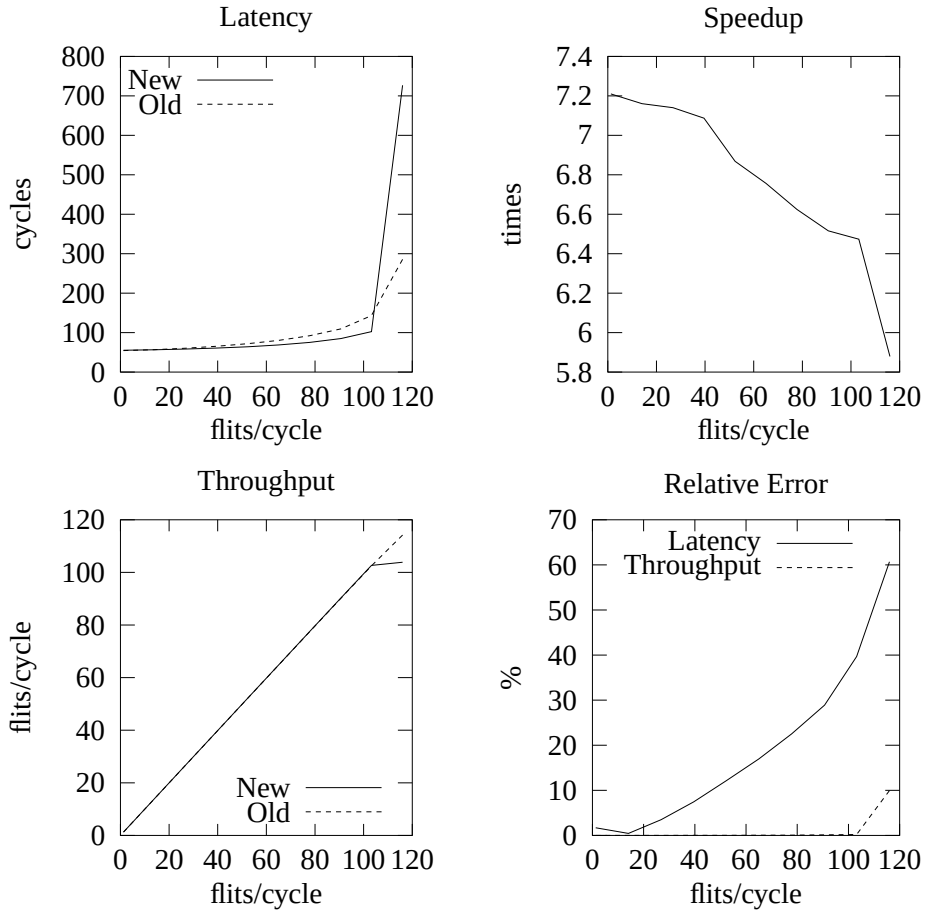


Figure 3.7: Performance of the simplified model of the adaptative router. 256 node torus network

although this router has a significant error, it is still a perfectly working router. It may not be suitable for tuning router parameters, but its a very good alternative to experiment other parts of a parallel machine. Having reached speedup ranges from 6 up to 22 and reducing the size of the simulator's memory usage down to 47% this router constitutes a very good tool to test networks with more than 1024 nodes or simulate real traffic.

3.2.3 Conclusions

The accuracy and performance of the two models has been measured, this has given a set of bounds for the accuracy of this new model strategy. Globally the latency error, the biggest of both, ranges from 16% to 60% while speedup lies in

the range of 2 to 22. However the higher errors and lower speedups occur in the saturation zone. When connecting SICOSYS to SimOS, only the lower load zone is used. If only the lower load area is observed, say 0 to 0.4, the latency error lies in the range of 5% to 10%. This will help giving a good approximation to the behavior of the network at a lower computational cost.

Chapter 4

Conclusion

4.1 Achievements

At the beginning of the project, there was a simulator that did not satisfy current demands. This simulator, SICOSYS, had very good qualities and there was a lot of effort put in it. Although its design posed no limits to the size and complexity of the simulations, the time it took to perform them was exceedingly high. By thoroughly studying its code a way of optimizing it has been found. This has allowed to deliver a renewed simulator capable of meeting actual simulation demands.

Being this optimization not enough for certain applications, a new way of implementing routers has been devised. This method allows the implementation of the whole router as a single component. Opposed to the multi-component architecture available in SICOSYS, this new technique speeds up the simulations and reduces the size of the simulator in memory at the cost of a loss of accuracy.

These improvements have made SICOSYS a tool for the future. The versatility of its design in combination with its optimized core allowed its integration with SimOS. At the time of writing several simulations have been performed showing excellent results.

Although the multicomponent models exhibit better performance, the monolithic router component allows the implementation of routers in a very compact an optimized way. This makes it an ideal candidate to build early prototypes of new routers.

4.2 Future Developments

The ever growing computational cost of the simulations will compel SICOSYS to improve its performance even further. Although SICOSYS has been run on parallel machines it is a single threaded application that takes no advantage of them.

Up to now multiple instances of the program were run at a time with different conditions. Nevertheless, when performing coosimulation, such as with SimOS, there must be only one instance of SICOSYS and it may not meet performance requirements in the future. One possible solution for this may be the parallelization of the simulator in order to make it take full advantage of the available parallel infrastructure.

Bibliography

[1] C. Carrión, R. Beivide, J.A. Gregorio, F. Vallejo, "A Flow Control Mechanism to Avoid Message Deadlock in k-ary n-cube Networks", International Conference on High Performance Computing, India, December 1997.

[2] W.J. Dally, "Performance Analysis of k.ary n-cube Interconnection Networks", IEEE Transactions on Computers, Vol. 39, no. 6, pp. 775-785, June 1990.

[3] J. Duato, "A Necessary and Sufficient Condition for Deadlock-Free Routing in Cut-Through and Store-and-Forward Networks", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, no. 8, pp. 841-854, August 1996.

[4] J. Duato, S Yalamanchill, L. Ni, "Interconnection Networks", IEEE Computer Society, 1997.

[5] R. Hempel, "The MPI Standard for Message Passing", Lecture Notes in Computer Science, Vol. 797, pp. 247-252, 1994.

[6] P. Kermani, R Kleinrock, "Virtual cut-through: a new computer communication switching technique", Computer Networks, Vol. 3, no. 4, pp. 267-286, 1979.

[7] A.R. Larzelere, "Creating Simulation Capabilities", IEEE Computational Science & Engineering. Vol 5, no. 1, pp 27-35, January/March 1998.

[8] V.S. Pai et al., "RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors". IEEE TCCA Newsletter, October 1997.

[9] J.M. Prellezo, V. Puente, J.A. Gregorio, R. Beivide, "SICOSYS: Un Simulador de Redes de Interconexión para Computadores Paralelos", VIII Jornadas de Paralelismo, Septiembre 1998.

[10] V. Puente, J.A. Gregorio, C. Izu, R. Beivide, "Impact of Head-of-Line Blocking on Parallel Computer Networks: Hardware to Applications", Europar'99, September 1999.

[11] M. Rosenblum, E. Bugnion, S. Devine, S. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems", ACM TOMACS Special Issue on Computer Simulation, 1997.

[12] S.C. Woo, M. Ohara, E. Torrie, J.P. Sing, A. Gupta, "The SPLASH-2 programs: Characterization and Methodological Considerations", Proceedings of the 22nd ISCA, pp. 24-37, June 1995.

[13] `http://www.top500.org`

Appendix A

Introduction to SICOSYS

It is frequently seen, in the analysis of interconnection networks, that analytical and simulation techniques often cooperate. The use of analytical tools is compromised by the complexity of the system under study. This is because normally the system must be simplified in order to allow its analysis. Frequently, these simplifications make the model not very similar to the system. In these cases, the way of obtaining convincing data could be by doing experiments with the real system. But this is not always affordable because the system is not normally built at the time of its study. This only leaves the possibility of simulating the system. Though this involves modeling the system as well, the computing infrastructure supporting the simulation process allows a great approximation of the model to the system.

When modeling a system, there must be a selection of the aspects that will conform the model. This is, the model should only have the features that determine the functionality of the system, and not others that can unnecessarily complicate it. Therefore, the difficulty of building a model lies on the appropriate choice of the system's features depending on the purpose of the analysis.

Simulation is a numeric technique that performs experiments on a given model in order to extract data that describes the functioning of the modeled system. Normally, a simulation involves a huge amount of operations and variables. Thus, simulation is nearly always done with the aid of computers.

When using a simulator, there are various things the must be kept in mind:

- The execution of a simulation can be very long. Therefore the model must be kept as simple as possible while resembling the system's features in maximum detail.

- As the results depend on the manipulation of random variables. There should be an averaging of many experiments in order to get a proper result.

- Seldom there is a model that mimics the system completely. There is always some detail that is not taken into account. Hence, the results of the simulation are never to be interpreted as the performance of the real system.

- In many cases the simulation operates in a transient phase before reaching a stationary state. To get results of the stationary state, no statistical measures should be taken during the transient phase.

The most precise way of simulating interconnection networks is by simulating a hardware implementation (at logical or physical level) of the network and routers. This procedure allows the simulation of details down to the transistor level. Anyway these simulations are very expensive in terms of simulation time and memory usage. Hence, the group of Computer Architecture and Technology at the University of Cantabria developed a simulation environment that can perform simulations of interconnection networks with less resources while achieving a very high accuracy. This tool gets the name SICOSYS form "Simulator of Communication Systems".

SICOSYS is a time driven simulator developed in C++ having in mind modularity, versatility and connectivity with other systems. The models used are intended to resemble the hardware description. In this way, the simulator mimics the hardware structure of the routers instead of just implementing their functionality.

Another objective of the project was giving a tool that could implement a wide variety of routers and networks and presented a homogeneous and simple interface to the user. Thus, SICOSYS has a collection of components like multiplexers, buffers or crossbars. And these components can connected to each other to conform a router and these, in turn, are connected in a certain network fashion. All this is defined in SGML which can be thought of as a superset of HTML. In fact, all the configuration that SICOSYS needs is written in SGML that makes it easy to understand and code to the user.

Appendix B

SICOSYS Tutorial

In this chapter will give rough description of the user interface to SICOSYS illustrated by a simple example. This will help the user get acquainted with the simulator.

B.1 Work Environment

In order to successfully run SICOSYS there are several files that have to be accessible to the simulator executable. This section will describe the directory structure needed by SICOSYS.

Basically the SICOSYS work environment is composed by five files:

`ATCSimul` Is the executable program. It can be anywhere accessible by the user. It can be in a system directory like `/usr/bin`, so it is accessible to many users, or it can be in a user directory when only one user needs it.

`ATCSimul.ini` It is read by SICOSYS at startup and tells it where the three SGML files are. It should be in the working directory. Therefore every user must have his own copy (or more than one).

`Router.sgm` Describes the structure of the routers that can be used. Its location must be pointed by `ATCSimul.ini`.

`Network.sgm` Describes the networks that can be used. Its location must be pointed by `ATCSimul.ini`.

`Simula.sgm` Describes the parameters of the different simulations. Its location must be pointed by `ATCSimul.ini`.

A more detailed description of all files is given in the next section. In addition, there are some tools to help the debugging tasks. These are explained in the Reference Manual

`BView` is a simple program to convert the buffer information file to a readable format.

`xbuffer` is a graphical application to represent the buffer information file.

`xsimul` is a graphical interface to SICOSYS. It executes simulations while presenting the evolution of the latency.

B.2 Configuration Files

All configuration files are coded in SGML. SGML is a markup language similar to HTML, in fact SGML can be considered a superset of HTML. A SGML file consists of a set of tags, each with a variable number of parameters. A tag can contain other tags with in it, conforming a hierarchical structure. This suits the the hierarchical descriptions of routers in SICOSYS perfectly. A simple introduction to the format of the files will be given throughout this section. For a detailed explanation of all the tags that are available see the reference manual.

B.2.1 `ATCSimul.ini`

The first, and most simple, configuration file is `ATCSimul.ini`. SICOSYS uses this file to find the rest of the configuration files it needs. This is very simple, as can be seen in the following example.

```
<SimulationFile  id="../sgm/Simula.sgm"  >
<NetworkFile     id="../sgm/Network.sgm" >
<RouterFile      id="../sgm/Router.sgm"  >
```

The three tags that can be found in this file give the location of each of the three files describing routers, networks and simulations. The locations can be absolute or relative to the working directory.

B.2.2 `Simula.sgm`

This file describes the scenarios that will be simulated. In addition to selecting a network from `Network.sgm`, other simulation parameters are given. These parameters include simulation length, seed for random numbers or traffic pattern and injection rate. Here follows a simple example:

```
<Simulation id="TORUS8x8-CT">
  <Network            id="TORUS8x8-CT">
  <SimulationCycles   id=20000>
  <TrafficPattern     id="MODAL" type="PERFECT-SUFFLE">
  <Seed               id=113>
  <Load               id=0.1>
  <MessageLength      id=1>
  <PacketLength       id=16>
</Simulation>
```

B.2.3 Network.sgm

The Network.sgm file describes the way the routers are connected to each other. This is, it describes the topology of the network to be used. The format of this file is fairly simple. Each network is defined by a single tag specifying the topology, the name, the router to be used, the size of the network and the delay of the wires between neighboring nodes. As an example, a 64 node torus network using the router shown above can be described easily as follows.

```
<TorusNetwork id="TORUS8x8-CT" sizeX=8 sizeY=8 router="
   DOR2D-CT" delay=0>
```

B.2.4 Router.sgm

This file has a description of all the routers that can be simulated. In addition to the router structure, there are other parameters that are specified in this description, like the buffer capacity or the delay of each component of the router.

In the following example the description of a simple 2D router is shown.

```
<Router id=DOR2D-CT inputs=4 outputs=4 bufferSize=64
        bufferControl=CT routingControl=DOR-BU >
   <Injector id="INJ">
   <Consumer id="CONS">

   <Buffer id="BUF1" type="X+"   dataDelay=2  size=32>
   <Buffer id="BUF2" type="X-"   dataDelay=2>
   <Buffer id="BUF3" type="Y+"   dataDelay=2>
   <Buffer id="BUF4" type="Y-"   dataDelay=2>
   <Buffer id="BUF5" type="Node" dataDelay=2>
```

```
<Routing id="RTG1" type="X+"   headerDelay=1 dataDelay
   =0>
<Routing id="RTG2" type="X-"   headerDelay=1 dataDelay
   =0>
<Routing id="RTG3" type="Y+"   headerDelay=1 dataDelay
   =0>
<Routing id="RTG4" type="Y-"   headerDelay=1 dataDelay
   =0>
<Routing id="RTG5" type="Node" headerDelay=1 dataDelay
   =0>

<Crossbar id="CROSSBAR" inputs=5 outputs=5 type="CT"
      headerDelay=1 dataDelay=1>
   <Input  id=1  type="X+">
   <Input  id=2  type="X-">
   <Input  id=3  type="Y+">
   <Input  id=4  type="Y-">
   <Input  id=5  type="Node">
   <Output id=1  type="X+">
   <Output id=2  type="X-">
   <Output id=3  type="Y+">
   <Output id=4  type="Y-">
   <Output id=5  type="Node">
</Crossbar>

<Connection id="C01" source="INJ" destiny="BUF5">
<Connection id="C02" source="CROSSBAR.5" destiny="CONS">
<Connection id="C03" source="BUF1" destiny="RTG1">
<Connection id="C04" source="BUF2" destiny="RTG2">
<Connection id="C05" source="BUF3" destiny="RTG3">
<Connection id="C06" source="BUF4" destiny="RTG4">
<Connection id="C07" source="BUF5" destiny="RTG5">
<Connection id="C08" source="RTG1" destiny="CROSSBAR.1">
<Connection id="C09" source="RTG2" destiny="CROSSBAR.2">
<Connection id="C10" source="RTG3" destiny="CROSSBAR.3">
<Connection id="C11" source="RTG4" destiny="CROSSBAR.4">
<Connection id="C12" source="RTG5" destiny="CROSSBAR.5">

<Input  id="1" type="X+"   wrapper="BUF1">
<Input  id="2" type="X-"   wrapper="BUF2">
<Input  id="3" type="Y+"   wrapper="BUF3">
<Input  id="4" type="Y-"   wrapper="BUF4">
<Output id="1" type="X+"   wrapper="CROSSBAR.1">
```

```
   <Output id="2" type="X-"   wrapper="CROSSBAR.2">
   <Output id="3" type="Y+"   wrapper="CROSSBAR.3">
   <Output id="4" type="Y-"   wrapper="CROSSBAR.4">
</Router>
```

In the `Router.sgm` file there can be any number of routers, each defined by a ROUTER tag. In the ROUTER tag, there are various attributes defining the name of the router, the number of inputs and outputs, the size of the buffers, the buffer control algorithm and the routing function. The complete definition of the internal structure of the router has to be between the ROUTER and the corresponding /ROUTER tag.

B.3 Running Simulations

Having the definition of the router, network and simulation, every thing is ready to execute the simulation. This is done with the `ATCSimul` program. It accepts various command line switches, all are optional except for one, *-s <simulation_ID>*. It tells `ATCSimul` which simulation must be executed. The rest of the switches override the simulation parameters set in the SIMULATION tag like *-l <load>* sets the traffic load and *-c <cycles>* modifies the number of cycles to simulate. There are also some switches that enable tracing and debugging functions.

In order to run the simulation shown in the examples above, the following command should typed:

```
$ ATCSimul -s TORUS8x8-CT
```

The output of the simulation is fairly simple. It consists of a listing of the statistical values observed in the simulation. By executing the command above, the following output is obtained.

```
**************************************
 Network        = Toro(8,8,1)
 buffer control = CT
 routing control= DOR-BU
 Started at : Tue May  8 08:16:31 2001
 Ended at   : Tue May  8 08:18:29 2001
**************************************
Traffic Pattern         = Perfect Shuffle
Seed                    = 113
Cycles simulated        = 20000
Buffers size            = 64
Messages length         = 1 packet(s)
```

```
Packets length          = 16 flits
Supply load             = 9.76125 %
Real load               = 9.73375 %
Supply Thr.             = 6.2472 flits/cycle
Throughput              = 6.2296 flits/cycle
Average distance        = 4.13395
Messages generated      = 7809
Messages received       = 7787
Messages to inject      = 8
Total message latency   = 42.6705
Network message latency = 40.8305
Buffer message latency  = 1.83999
Maximum message latency = 159
Simulation time         = 00:01:58
```

The header presents information of the network and router that were simulated. Following are the simulation parameters and then the simulation results, the meanings of which are explained below:

`Supply load` the traffic load presented to the network. This value is nomalised to the network bisection.

`Real load` the traffic load actualy transmited by the network. This value is normalized to the network bisection.

`Supply Thr.` a different view of the load presented to the network, in terms of injected flits per simulated cycle.

`Throughput` the number of flits the network consumed per simulated cycle.

`Average distance` the average distance traveled by the messages.

`Messages generated` the amount of generated messages.

`Messages received` the amount of received messages.

`Messages to inject` the amount of messages that are still to be injected.

`Total message latency` The mean total latency. This is, since the message is generated until it is consumed.

`Network message latency` The mean network latency. This is, since the messages is injected in the network until it is consumed.

`Buffer message latency` The mean time the messages spend waiting to be injected in the network.

`Maximum message latency` the maximum total latency registered in the simulation.

`Simulation time` The time needed to complete the simulation.

B.4 Helper Applications

The SICOSYS work environment includes three applications that are very useful to detect and locate malfunctions. The two first are concerned with the buffer usage within the routers. The third is an interface to SICOSYS that presents a graphical view of the evolution of the simulation while its running.

B.4.1 Examinig Buffers

When designing a router it is very interesting to know how full do the routers get in order to optimize their size. SICOSYS can write a file containing samples of the buffer ocupation, this is done by specifying a filename after the parameter *-b*. The samples are taken every 100 cycles. Because the size of this file can get very big, it is written in a binary format.

There are two tools to inspect the buffer ocupation file. `BView` is a very simple program that calculates the average ocupation for each buffer and presents it in a table. For example, running random traffic in a 4 node torus network with bubble routers gives an the following output.

```
Network size      = (2,2,1)
Number of buffers = 5
Buffer Size       = 32

Mean buffer ocupation
 (0,0,0)   (1,0,0)   (0,1,0)   (1,1,0)
BUF5  size=32
   0.945      1.17      1.15      1.38
BUF4  size=32
   0.525      0.62      0.55      0.52
BUF3  size=32
   0.485      0.66     0.445      0.45
BUF2  size=32
   0.625      0.39     0.695      0.32
BUF1  size=32
   0.355      0.38       0.5     0.605
```

The other tool is graphical tool designed using Trolltech's QT GUI application framework. The tool gives a graphical view of the buffer ocupation. `xbuffer` loads the buffer file and does an animation of the ocupation of a certain buffer of each router. The View→Select Buffers option presents a lists of the buffers in a router. The user can choose which buffers will be represented. By pressing the Start button the program opens a window for each selected buffer. The windows are divided in cells, one for each router in the network, the colour of the cells vary from white to black depending on the ocupation of the buffer in that router. The figure B.1 shows a screenshot of a 64 node torus.

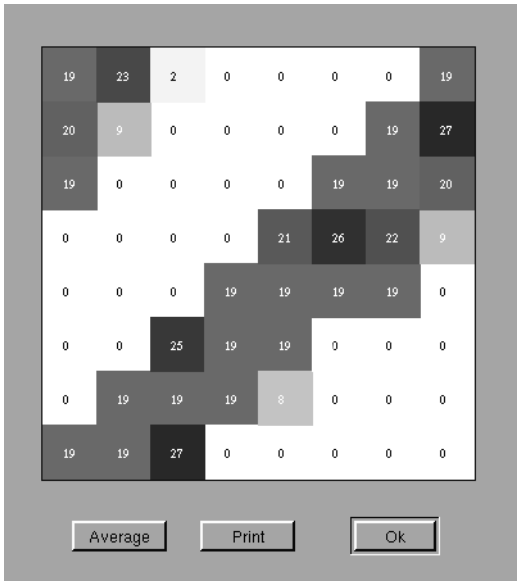


Figure B.1: Screenshot of `xbuffer`

B.4.2 Observing Latency Evolution

Another interesting point in the performace analisys of routers and networks is to know the evolution of the latency. `xsimul` is a simple front-end to SICOSYS, also based on Trolltech's QT GUI application framework. With then File→Open option the user can select a simulation from the `Simula.sgm` file. To further configure the simulation, the Options→Parameters option enables the user to set simulation parameters, such as traffic pattern or simulation lenght (See figure B.2). And by clicking the Run button the simulation starts. During the simulation the latency is plotted versus the simulation cycles in real time, enabling the user to ver the evolution of the network easily. Figure B.3 shows a screenshot of `xsimul`.

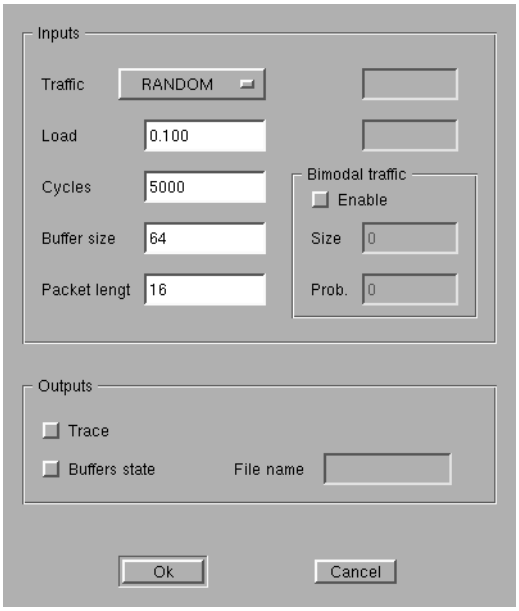


Figure B.2: Screenshot of Options→Parameters dilog box

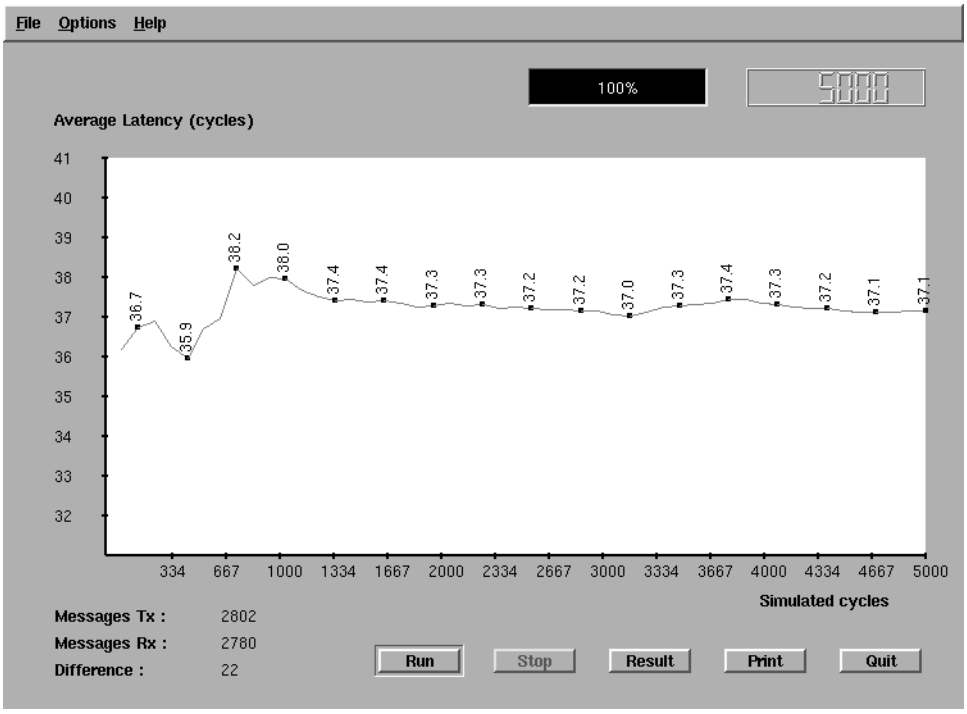


Figure B.3: Screenshot of `xsimul`

Appendix C

Reference Manual

C.1 Command Line Syntax

-s <simulation_id> simulate the entry <*simulation_id*> of file `Simula.sgm`. This switch is always necessary.

-l <load> set the traffic load. This value is normalized to the torus bisection. It overrides the value set by the tag LOAD in `Simula.sgm`, it can not be used together with *-p*. <*load*> must be in the range of 0.0 to 1.0. See also LOAD in page 53.

-p <probability> set the injection probability. It overrides the value set by the tag PROBABILITY in `Simula.sgm`, it can not be used together with *-l*. <*probability*> must be in the range of 0.0 to 1.0. See also PROBABILITY in page 54.

-c <simulation_cycles> set the simulation length. It overrides the value set by the SIMULATIONCYCLES tag in `Simula.sgm`. See also SIMULATIONCYCLES in page 54.

-u <buffer_size> set the size of all buffers of each router. It overrides the value set globally by the ROUTER tag, so the buffers that have their size individually set by the BUFFER tag in `Router.sgm` are not modified. See also ROUTER in page 59 or BUFFER in page 56.

-L <packet_lenght> set the packet length. It overrides the value set by the PACKETLENGTH tag in `Simula.sgm`. See also PACKETLENGTH in page 54.

-t <pattern>[,<param1>[,<param2>]] set the traffic pattern. <*pattern*> can be any traffic pattern name. <*param1*> and <*param2*> can be added if

traffic-specific parameters a needed. There may be no blank space between *<pattern>*, *<param1>* or *<param2>*. It overrides the value set by the TRAFFICPATTERN tag in `Simula.sgm`. See also TRAFFICPATTERN in page 55.

-B <long/short_packet_ratio>,<long_packet_probability> set bimodal traffic. This is, two types of packets are generated, a longer one and a shorter one. The short packet's size is set by the *-L* or the PACKETLENGTH. *<long/short_packet_ratio>* sets how many times bigger than the short packet is the long packet. *<long_packet_probability>* is the probability of generating long packets. See also TRAFFICPATTERN in page 55 and PACKETLENGTH in page 54.

-d <message_count> show the evolution of the simulation over time. When the simulation finishes, a table shows the traffic load offered to the network and the mean latency. A sample presented in the table every time the network transmits *<message_count>* messages. In addition a histogram of the distance traveled by the messages is also shown.

-b <bufferFileName> set the name of the file that will hold buffer occupation information. The buffer occupation is sampled every 100 cycles and written, in a binary format, in *<bufferFileName>*. The information stored in this file can be examined with `BView` or with `xbuffer`. See also `BView` in page 42 and `xbuffer` in page 42.

C.2 SGML Tag Reference

C.2.1 Tags in `ATCsimul.ini`

NETWORKFILE

NetworkFile id=<path>

Specifies the path where the network description file, normally `Network.sgm`, is found. The path can be absolute or relative to the current directory of the simulation.

ROUTERFILE

RouterFile id=<path>

Specifies the path where the router description file, normally `Router.sgm`, is found. The path can be absolute or relative to the current directory of the simulation.

SIMULATIONFILE

SimulationFile id=<path>

Specifies the path where the simulation description file, normally `Simula.sgm`, is found. The path can be absolute or relative to the current directory of the simulation.

C.2.2 Tags in `Simula.sgm`

LOAD

Load id=<load>

Sets the normalized traffic load to *<load>*. The injection probability is calculated as shown in[4]. It may be overridden by the the switches *-l* or *-p*. This tag can not be used together with the tag PROBABILITY. Value *<load>* must be in the range of 0.0 to 1.0. See also PROBABILITY in page 54.

MESSAGELENGTH

MessageLength id=<message_length>

Specifies the number of packets a message has.

NETWORK

Network id=<Network_id>

Sets the name of the network that will be used in the simulation. The identifier *Network_id* must be defined in `Network.sgm`

PACKETLENGTH

PacketLength id=<packet_lenght>

Sets the number of flits a packet has. It may be overridden by *-L*.In the case of bimodal traffic patterns, *packet_lenght* defines the length of the smallest packet. See TRAFFICPATTERN in page 55.

PROBABILITY

Probability id=<probability>

Sets the injection probability to *<probability>*. It may be overridden by the the switches *-l* or *-p*. This tag can not be used together with the tag LOAD. Value *<probability>* must be in the range of 0.0 to 1.0. See also LOAD in page 53.

SEED

Seed id=<seed>

Sets the seed for the random number generator.

SIMULATION

Simulation id=<simulation_id>

This is a group tag. It bundles together all the simulation parameters and associates them to *simulation_id*. This is the identifier that must follow *-s* when performing a simulation.

SIMULATIONCYCLES

SimulationCycles id=<simulation_cicles>

Sets the number of cycles that should be simulated. It may be overridden by *-c*.

STOPINJECTIONAFTER

StopInjectionAfter id=<simulation_cicles>

Sets the number of cycles after which no more packets should be injected to the network.

TRAFFICPATTERN

TrafficPattern id=MODAL type=<pattern>[,<param1>[,<param2>]]

TrafficPattern id=BIMODAL type=<pattern>[,<param1>[,<param2>]] prob=<long_packet_probability> numMsg=<long/short_packet_ratio>

Sets the traffic pattern that is injected to the network. *<pattern>* can be any traffic pattern name. *<param1>* and *<param2>* can be added if traffic-specific parameters a needed. There may be no blank space between *<pattern>*, *<param1>* or *<param2>*.

If *BIMODAL* traffic is needed. The short packet's size is set by the *-L* or the PACKETLENGTH tag. *<long/short_packet_ratio>* sets how many times bigger than the short packet is the long packet. *<long_packet_probability>* is the probability of generating long packets.

The effect of this tag may be overridden by parameters *-t* and *-B*. See also PACKETLENGTH in page 54.

C.2.3 Tags in `Network.sgm`

MIDIMEWNETWORK

MidimewNetwork id=<network_id> sizeX=<size_x> router=<router_id> [delay=<delay>]

Connect *<size_x>* routers with identifier *<router_id* using a midimew topology. The connections between the routers can have an optional delay of *<delay>*.

SQUAREMIDIMEWNETWORK

SquareMidimewNetwork id=<network_id> sizeX=<size_x> router=<router_id> [delay=<delay>]

Connect <*size_x*> routers with identifier <*router_id* using a square midimew topology. Currently <*size_x*> must be a even power of 2 (2^{2i}). The connections between the routers can have an optional delay of <*delay*>.

TORUSNETWORK

TorusNetwork id=<network_id> sizeX=<size_x> sizeY=<size_y> sizeZ=<size_z> router=<router_id> [delay=<delay>]

Constructs a 1D, 2D or 3D torus with <*size_x*>, <*size_y*> and <*size_z*> routers in each dimension. The connections between the routers can have an optional delay of <*delay*>.

C.2.4 Tags in `Router.sgm`

BUFFER

Buffer id=<buffer_id> type=<dimension> [dataDelay=<delay>] [size=<buffer_size>] [ackgranul=<granularity>]

This component implements a simple FIFO queue for flits. It accepts flits from the component before it and serves them to the component behind. The buffer control function, also knows as flow control unction, tells when the buffer is full and can not admit more flits. This function is defined as follows:

wormhole The stop signal goes high when there is no room for another flit.

cutthrough The stop signal goes high when there is no room for another whole packet.

CONNECTION

Connection id=<connection_id> source=<source_id> destiny=<destiny_id>

Connects two components within the router. <*source_id*> and <*destiny_id*> must be a string composed by the identifier of the component and, if necessary, a dot followed by the number of the input or output.

CONSUMER

Consumer id=<consumer_id>

Accepts incoming flits. It performs some checks to ensure that the flits are reaching their destination, that they arrive in proper order and not mixed with flits from other packets. In addition it performs statistical measures on the incoming flits.

CROSSBAR

Crossbar id=<crossbar_id> inputs=<inputs> outputs=<outputs> type=<type> [mux=<mux>] [headerDelay=<header_delay>] [dataDelay=<data_delay>]

Connects any of its inputs to any of its outputs. This component performs the spatial switching ability of the router. Depending on the *<type>* parameter, it establishes a specific protocol with the previous component, normally a routing, in order to grant or deny its requests.

<inputs> and *<outputs>* specify the number of inputs and outputs the crossbar has. The crossbar may have different number of inputs and outputs when the *<mux>* parameter is not 1. This *<mux>* parameter indicates the crossbar it has to multiplex the outputs in order to bundle various virtual channels into a physical one. *<header_delay>* and *<data_delay>* set the delays for header flits and data flits respectively. See also ROUTING in page 60.

FIFOMUXED

FifoMuxed id=<buffer_id> type=<dimension> outputs=<outputs> control=<control> [dataDelay=<delay>] [size=<buffer_size>] [ackgranul=<granularity>]

This component is like a normal BUFFER but it has a demultiplexer attached to the output, this enables various virtual channels to share the space in the buffer. The number of outputs of the demultiplexer is set with the *<outputs>* parameter. The rest of the parameters are set like in the normal BUFFER tag. See also BUFFER in page 56.

INJECTOR

Injector id=<injector_id> numHeaders=<headers>

It is the component in charge of converting the messages that arrive to the node into sequences of flits organized into packets. The *<headers>* value specifies how many flits per packet are used as headers.

INPUT

Input id=<input_id> type=<dimension> { wrapper=<id_list> | [channel=<channel>] }

This tag declares each of the inputs of a CROSSBAR tag or a ROUTER tag. Each INPUT must have a different *<input_id>* from the rest of the inputs in the component. The *<dimension>* must tell to which dimension is the input connected. This can be X+, X-, Y+, Y-, Z+ or Z- if the input handles flits coming from other routers, or NODE if it is a crossbar input handling flits from the local injector.

When specifying router inputs, the *<id_list>* specifies to which component is the input connected to. In case of an input that supports multiple virtual channels, the *<id_list>* holds a list of the components that will be connected to each virtual channel.

If the input belongs to a CROSSBAR that has multiple virtual channels for each dimension, there must be a different input for each virtual channel and it must be specified using the *<channel>* parameter.

See also ROUTER in page 59 and CROSSBAR in page 57.

MPBUFFER

MPBuffer id=<buffer_id> type=<dimension> inputs=<inputs> control=<control> [dataDelay=<delay>] [size=<buffer_size>] [sizeShort=<buffer_size_short>]

MULTIPLEXORCV

MultiplexorCV id=<multiplexor_id> inputs=<inputs>

This component does a multiplexing of various virtual channels to a single physical channel. The number of channels it multiplexes is defined by *<input>*.

OUTPUT

Output id=<output_id> type=<dimension> { wrapper=<id_list> | [channel=<channel>] }

This tag declares each of the outputs of a CROSSBAR tag or a ROUTER tag. Each OUTPUT must have a different *<output_id>* from the rest of the outputs in the component. The *<dimension>* must tell to which dimension is the output connected. This can be X+, X-, Y+, Y-, Z+ or Z- if the output handles flits going to other routers, or NODE if it is a crossbar input handling flits that must go to the local consumer.

When specifying router outputs, the *<id_list>* specifies to which component is the output connected to. In case of an output that supports multiple virtual channels, the *<id_list>* holds a list of the components that will be connected to each virtual channel.

If the output belongs to a CROSSBAR that has multiple virtual channels for each dimension, and there are not multiplexed, there must be a different output for each virtual channel and it must be specified using the *<channel>* parameter.

See also ROUTER in page 59 and CROSSBAR in page 57.

ROUTER

Router id=<router_id> inputs=<inputs> outputs=<outputs> bufferSize=<buffer_size> bufferControl=<buffer_control> routingControl=<routing_control>

This tag encloses the definition of a router in terms of interconnected components. The *<router_id>* specifies the identifier of the router that will be used in the `Network.sgm` file. *<inputs>* and *<outputs>* set the number of inputs and outputs the router has, normally both values are equal. *<buffer_size>* sets a default size for the buffers in the router. <buffer_control> specifies the buffer control

function, it can be WH, for worm-hole, or CT, for cut-through. And *<routing_control>* establishes the routing function of the ROUTING components.

ROUTING

Routing id=routing_id> type=<dimension> [channel=<channel>] [headerDelay=<header_delay>] [dataDelay=<data_delay>]

A ROUTING component analyzes incoming packages, calculates the output they should go and communicates with the crossbar to request a connection for it.

Routings must have information of which dimension and virtual channel there are working in. This must be stated with parameters *<dimension>* and *<channel>*. *<header_delay>* and *<data_delay>* set the delays for header flits and data flits respectively. See also CROSSBAR in page 57.

ROUTINGMUXED

RoutingMuxed id=routing_id> type=<dimension> [channel=<channel>] inputs=<inputs> [headerDelay=<header_delay>] [dataDelay=<data_delay>

This component has similar functions as the ordinary ROUTING but it can multiplex a number of virtual channel allowing them to share the output to the crossbar, thus reducing its complexity.

The only parameter that differs from the ROUTING is the number of *<inputs>* to the component. See also ROUTING in page 60.

SIMPLEROUTER

SimpleRouter id=<simple_router_id> inputs=<inputs> outputs=<outputs>

The SIMPLEROUTER component is intended to pack all of the components (Except injector and consumer) of a router into a single component. There is a slight loss of accuracy but, on the other hand, the simulation time is considerable shorter.

Within this tag there must be a list of the inputs and outputs that the component has, very much like the CROSSBAR.

C.3 Known Problems

The SGML processing routines are written on a text-line basis, therefore tags can not span more than one line of text. Similarly two tags can not be in the same line.

Appendix D

SICOSYS Programming Guidelines

D.1 Introduction

SICOSYS was designed having in mind its versatility and modularity, while keeping the user interface as simple as possible. This idea gave the project a chance of keeping up to date with new developments, therefore lasting a long time. This chapter aims to give a simple introduction to the way SICOSYS can grow by showing how to add a new network, component or traffic pattern.

D.2 Coding Style

Throughout the code of SICOSYS there are various conventions in the programming style. Most of them are just object oriented programming common practice, like keeping class attributes private and writing inline methods to access them or observing `const` correctness. Nevertheless there are two topics that might be new to the reader, constants and runtime type information.

The coding style can be resumed as follows:

- Class names should be preceded with three uppercase letters chosen by the programmer in order to avoid name collisions. The names should be lowercase with first letters of words in uppercase. As in `PRZCompositeComponent`.

- Class method names should be lowercase with first letters of words in uppercase except for the first word. As in `updateMessageInfo`.

- Class attribute names should follow the method naming convention and preceded by `m_`. As in `m_messageQueue`.

- Local variable names should help understand contents of the variable. Single letters should not be used, except for integer indexes in `for` loops.
- When a boolean variable is needed, it should be declared as `short` and the constant variables `true` and `false` should be used to assign and test it.
- Class definitions are written in files with the same name as the class with extension `.hpp` and stored in directory `inc`.
- Class implementations are written in files with the same name as the class with extension `.cpp` and stored in directory `src`.
- Three space indenting should be used.
- Comment should explain tricky pieces of code.
- Class members should be kept private. If they need to be used from outside the class, there should be inline members to access them.
- `const` correctness should be observed.
- References are preferred to pointers in most cases (but not all).

D.2.1 Constants

All constants should be centralized in the same file to allow changing the builtin limits and parameters. These include nearly all of the strings that are used in SICOSYS, including tag names and parameters. All this is grouped in a file called `PRZConst.hpp`. This file implements the strings as extern variables that can be accessed from every class that includes it. There is also a naming convention for the string identifiers. The names should be all uppercase using underscore as word separator and preceded by three letters chosen by the programmer in order to avoid name collisions. Similarly, error message strings are stored in a file called `PRZError.hpp`.

D.2.2 Runtime Type Information

In order to safely cast a base class pointer to a derived class pointer. SICOSYS has a runtime type information class that is included in most classes. The way this is done is very easy. First The definition of the class should have a line like the following:

```
DEFINE_RTTI(<class_name>);
```

Where <class_name> is the name of the class. And second, the class implementation file should include the following:

```
IMPLEMENT_RTTI_DERIVED(<class_name>,<base_class_name>);
```

Where <class_name> and <base_class_name> are the names of the class and its parent class respectively.

D.2.3 Adding a module to SICOSYS

SICOSYS has a directory called `mak` that contains what is necessary to build the binary program. Basically, it has a `Makefile` and `OBJS` directory that contains the object and other intermediate files.

In order to add a new module to SICOSYS, add the name of the module (no extension is needed) to the `MODULES` variable in the `Makefile`. Next execute the command:

```
make depend
```

This will analyze the dependencies between modules and put them in `Makedepend`. Now executing `make` should build SICOSYS.

D.3 Class Structure

SICOSYS has a large number of classes closely interrelated. Nevertheless, to implement new networks, components or traffic patterns, only a small subset of the classes must be known. In the following subsections, a small introduction to the structure of SICOSYS will be given. In figure D.1, a simplified view of the class diagram of SICOSYS is shown.

D.3.1 Components and Builders

Simulations are defined with the three SGML files `Simula.sgm`, `Network.sgm` and `Router.sgm`. Each of these describe the set of components that will conform the simulation. SICOSYS defines an abstract `PRZComponent` class that will be base class for all the components that are constructed as described in the SGML files. One of the aggregators of class `PRZComponent` is the abstract class `PRZBuilder`, its children specialize in reading one of the three SGML files. Thus, `PRZSimulationBuilder` specializes in reading `Simula.sgm` to create `PRZSimulation` components and similarly happens with `NetworkBuilder`. But a router is defined as a set of components itself, therefore the `PRZRou-`

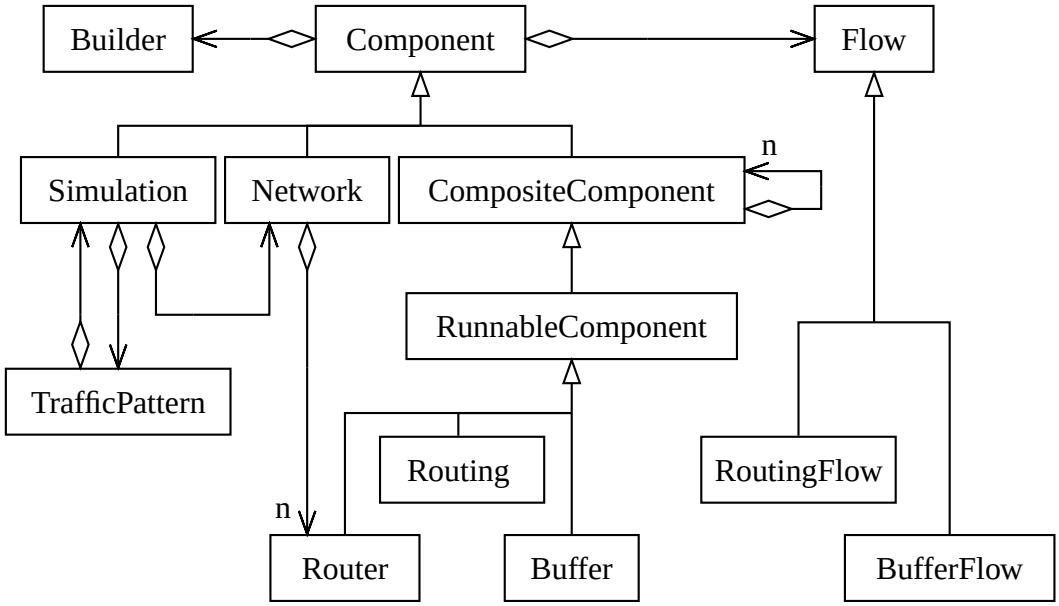


Figure D.1: Simplified class diagram

`terBuilder` class specializes in constructing a family of components that derive from `PRZCompositeComponent`. This class is an abstract class that can contain more components, enabling the creation of hierarchically structured components.

Once the building process is finished, SICOSYS has a structure of components that can simulate. The simulation component has a specific traffic pattern object, derived from `PRZTrafficPattern`, and an instance of a particular network, derived from `PRZNetwork`. In turn, the network object has a set of `PRZRouter` objects, one for each node of the network. Each router, as they are derived from `PRZCompositeComponent` has a set of components, such as buffers, routings or crossbars.

D.3.2 Components and Flow Control

As explained above, router components contain many other components that draw up its behavior. Nevertheless, these components do not have their functionality implemented in them. Each component delegates its functionality to an aggregator derived from the `PRZFlow` class. In this way a component can have a variety of flow control classes that implement different behaviors of the component. For

example, the class PRZFifoMemory has the PRZCTFifoMemoryFlow class to implement the cut-through behavior and the PRZWHFifoMemoryFlow class to implement the worm-hole behavior.

D.3.3 Message Life Cycle

Once the building process is over, the simulation can start. During simulation, messages are created, sent to the network and received. SICOSYS has a class PRZMessage that represents the information flowing through the network, it can represent either a single flit, when it is in the network, or a full message, when it is generated by the traffic pattern class.

In the simulator main loop, there is a call to the traffic pattern object to generate a message for each router, this message is represented by a single PRZMessage. The message is sent to the network, then to the appropriate router and then enqueued in the injector component within the router. The injector reads the message object and finds out how many packets and flits it has. Then, it sends flits, each represented by a new PRZMessage object, through its output.

The PRZMessage object progresses from the output of a component through a PRZConnection object to the input of the next, eventually crossing to other routers and finally reaching a PRZConsumer object. This component notifies the network the arrival of the flit in order to perform statistical calculations.

PRZMessage objects are created in the PRZTrafficPattern and PRZInjector classes and destroyed in the PRZConsumer[1]. But in a simulation, there are thousands of messages to be created and destroyed, so SICOSYS has a special class that creates a pool of messages and manages their allocation and deallocation. The SFSPool class' allocate and release methods are much faster than the standard C++ operators new and delete.

D.3.4 Message Exchange Protocol

In order to transmit messages from a component to the next, a simple protocol has been implemented in SICOSYS. This protocol resembles the one used in real hardware, as show in figure D.2. It is based on two signals, ready tells the receiver component that there is a message ready to be read and stop tells the sender component that the receiver can not process messages.

To send a message one component calls sendData on the output with the message as a parameter. This sends the message to the output port and rises the ready signal causing the onReadyUp method to be called on the receiver. Normally the receiver class has an overridden version of the onReadyUp method

[1] In fact, there are complex routings and crossbars that exchange messages between them.

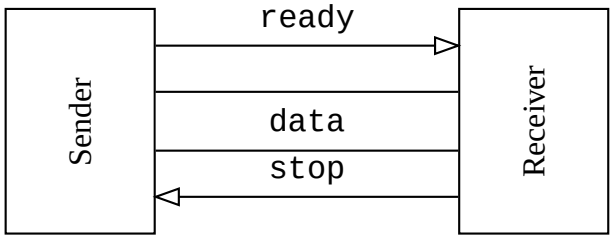


Figure D.2: Signals in component connections

that reads the message from the input port calling `getData` on the input port. Assuming the message is read, the sender component can call `clearData` to reset the `ready` signal at the end of the cycle.

The case might arrive that the receiver can not process more messages. This can happen when a crossbar has the desired output port engaged or when a buffer is full. In these cases the `sendStop` method is called on the input of the component. This in turn calls the `onStopUp` method on the sender component. Normally this method calls `sendStop` on the input, thus, propagating the `code` signal backwards. The `onStopUp` method can be overridden in case the propagation should not happen. For example, a buffer can still process incoming messages though its output is blocked by a `stop` signal.

D.4 Creating a new Traffic Pattern

There are many different types of traffic patterns already implemented in SICOSYS, nevertheless it might be interesting to add new kinds in order to test networks and routers under new conditions. Thus, the development of a simple traffic pattern and how it is coded into SICOSYS is explained bellow.

D.4.1 The New Traffic Pattern class

Each traffic pattern is implemented by a single class derived from `PRZTrafficPattern`. During the simulation the method `injectMessage` is executed any time a new message should be created. By overriding this method the new traffic pattern can be executed. If the traffic pattern needs some parameters to be set, the class must have the appropriate attributes to store them.

The example that will be illustrated sends messages from the source node to a specific neighbor. The parameters that it will take are the relative coordinates of the neighbor `dx`, `dy` and `dz`. These will be kept in the private part of the class and will be set by a public method.

D.4.2 The `injectMessage` method

For a simple traffic pattern like this, most of the code can be coded inline in the class definition. The only method that needs more than a line is the `injectMessage` method. This method takes a single parameter, the position of the source node, and should always return `true`. The `PRZPosition` is a vector class that represents a position within the network by holding the three coordinates that define it.

```
Boolean TUTTrafficPatternJump :: injectMessage(const
   PRZPosition& source)
{
   /* Create the message with source node */
   PRZMessage* msg = generateBasicMessage(source);

   /* Calculate target node */
   unsigned x, y, z;
   x = (source.valueForCoordinate(PRZPosition::X) + getX
      () + dx ) % getX();
   y = (source.valueForCoordinate(PRZPosition::Y) + getY
      () + dy ) % getY();
   z = (source.valueForCoordinate(PRZPosition::Z) + getZ
      () + dz ) % getZ();
   PRZPosition destiny = PRZPosition(x,y,z);
   /* Set target node in message */
   msg->setDestiny(destiny);
   /* Send message */
   getSimulation().getNetwork()->sendMessage(msg);

   return true;
}
```

In general, the `injectMessage` method has to:

- Create a message object, that will eventually be sent to the network.
- Calculate the target node from the source node and other external data, in this case, the parameters. Nodes should not be sending messages to themselves.
- Set the calculated target node in the message.
- Send the message to the network.

D.4.3 Traffic Pattern Creation

Once the traffic pattern class is defined, SICOSYS must be able to create instances of it. This is done by modifying the `createTrafficPattern` method of the `TrafficPattern` class. This method gets the traffic pattern tags from the `Simula.sgm` file and creates the appropriate traffic pattern object. It has a huge `if-else` chain to select the traffic pattern.

In order to create an instance of the traffic pattern class described above, the following code is required:

```
else if( type == TUT_TAG_JUMP )
{
   /* Create the traffic pattern object */
   TUTTrafficPatternJump* newPattern = new
      TUTTrafficPatternJump(simul);
   /* Read tag parameters */
   PRZString param;
   int x = 0, y = 0, z = 0;
   if( tag.getAttributeValueWithName(PRZ_TAG_PARAM1,
      param) )
      x = param.asInteger();
   if( tag.getAttributeValueWithName(PRZ_TAG_PARAM2,
      param) )
      y = param.asInteger();
   if( red3D && tag.getAttributeValueWithName(
      PRZ_TAG_PARAM3, param) )
      z = param.asInteger();

   /* Show error if parameters aren't valid */
   if( !x && !y && !z )
   {
      PRZString err;
      err.sprintf( ERR_PRZTRAFI_001, (char*)
         PRZ_TAG_PARAM1 );
      EXIT_PROGRAM(err);
   }
   /* Set paraemter values in traffic pattern class */
   newPattern -> setJump(x,y,z);
   /* Set traffic pattern object */
   pattern=newPattern;
}
```

Observe that the `TUT_TAG_JUMP` tag string must be declared in the `PRZConst.hpp`

file and that it can not be used by to different traffic patterns. The instantiation of a traffic pattern normally involves four steps:

- Creation of the new traffic pattern object.
- Read tag parameters.
- Check validity of the parameters
- Set the read parameters and other information in the traffic pattern object.

D.5 Creating a new Network

The networks that SICOSYS simulates are based on a 2D or 3D arrangement of nodes. The way the connections between the nodes are made, leaves a great degree of freedom to implement many regular networks. In the following example, a variation on the concept of midimew network is implemented into SICOSYS, the square midimew network. An example with 64 nodes is shown in figure D.3.

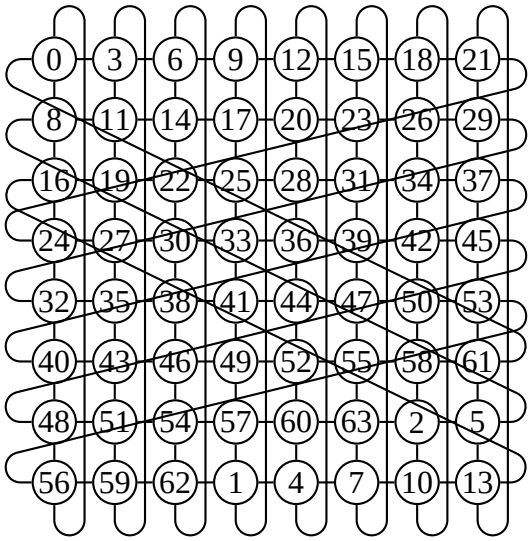


Figure D.3: 64-node square midimew

D.5.1 The New Network Class

As shown in the introduction, new network classes are to be derived from `PRZNetwork`. The new network class must implement the special aspects of the network

type. A list of methods that must be overridden and what should they do is shown below:

`newFrom` Instantiate the class from the tag.

Constructor Initialize statistics.

`initialize` Create and connect the routers of the network.

`initializeConnectionsFor` Connect a given router to its neighbors.

`routingRecord` Calculate the routing record, given source and target nodes.

`distance` Calculate topological distance between two nodes.

`getDiameter` Calculate the diameter of the network.

`asString` Give description of the component.

Some of these methods are fairly simple and need not further explanation. Anyway, the methods involved with the creation process and those concerning the routing will be covered in detail in the following subsection. However, the complete code can be read in `inc/PRZSquareMidimewNetwork.hpp` and `src/PRZSquareMidimewNetwork.cpp`.

D.5.2 The Creation Process

All networks are created from their sgml description in `Network.sgm`. The `PRZNetworkBuilder` class reads the file and calls the `newFrom` method of the appropriate class, passing it the tag read from the file. This can be done without an instance of the class because this method is `static`. In fact this method does the network object creation after reading and checking the tag passed by the builder. A close view of the `PRZNetworkBuider` shows that its method `parseComponentDefinition` has a big `if-else` chain. It compares the tag string against the tag names of the all network classes SICOSYS has already implemented. When adding a new network class, the new tag string must be declared in the `PRZConst.hpp` file and it can not be used by another network class.

After the creation, there are some parameters passed to the object, and then the `initialize` method is executed. The `initialize` method does some parameter checks before it is ready for creating the routers for each node of the network. Then with the aid of `initializeConnectionsFor` the routers are connected together. This method is called once for each router. It calculates all the neighbors and the port numbers associated with each routing direction.

Then it connects all the outputs of the router with the corresponding inputs of its neighbors. A little care must be taken in this part to avoid missing or duplicate connections.

D.5.3 Routing Through The Network

Once the router arrangement is constructed and connected, the network class is used to calculate the routing records of the messages. This is done by the `routingRecord` method. For other networks implemented in SICOSYS, there are direct ways of calculating the routing records, but for the `PRZSquareMidimewNetwork` there is no such direct method. Thus, a routing table is constructed by a brute force approach during initialization and the `routingRecord` just looks it up.

D.6 Creating a new Component

In order to compose the behavior of a router, SICOSYS has a selection of components that can be connected to each other. This section will explain how new components can be added to SICOSYS and how to implement their functionality. In SICOSYS, the structure and the behavior of a component are coded separately in two different classes. In fact, a component class can be associated to an arbitrary number of behavior classes.

D.6.1 The Component's Structure

SICOSYS' components are all derived from the `PRZRunnableComponent` class. It encapsulates the functionality needed by the simulator to execute the component. The component class stands for the structure of the component and is also responsible for the creation process. In the following example, the construction of a simple component will be presented. The component is the simple routing block that is implemented in SICOSYS. It has one input and one output, and it admits the following parameters in its description tag:

control the type of routing function to use.

headerDelay the number of cycles it takes to process the header of a packet.

dataDelay the number of cycles it takes to process data flits.

The definition of the `PRZRouting` class must have attributes to hold the parameters shown above. Anyhow, the base class already provides support for *headerDelay* and *dataDelay*, so only a member for the *control* is needed.

In order to properly read the tag, interpret its parameters and instantiate the class, the `PRZRouterBuilder` class calls the `newFrom` method when it needs to create a routing. This method is `static` and can be called when there is no object created of its class. The `newFrom` has to create a new object of the class `PRZRouting`, set all the parameters read from the tag and return it to the `PRZRouterBuilder` class.

Once the `PRZRouterBuilder` class has the instance of the `PRZRouting` class, it initializes it. This involves the creation of the flow class, that encapsulates the functionality of the component. When the component is initialized, the method `buildFlowControl` is called. This method must be overridden by the component class in order to create the flow class correctly. In the example there is a wide choice of flow classes and, depending on the value of the *control* parameter, one of them will be created.

D.6.2 The Flow Control Class

SICOSYS has many flow control classes for the `PRZRouting` class. Each provide a different protocol towards the crossbar or handles different packet structures. Control flow classes are like the brains of the component, they are able of reading parameters from the component, receive and send packets through the components inputs and outputs, etc.

Though all the flow control functionality of the component might be implemented in one class, it is sometimes worth considering designing a class that holds the common features of a component. Then, more specialized classes can be derived from this one to complete the behavior of a particular component. This is the case of the `PRZRouting` component in which `PRZRoutingFlow` provides basic methods for `PRZRoutingBurbleFlow`, implementing a bubble routing, or `PRZRoutingCVFlow`, for virtual channel routing.

This example will explain the `PRZRoutingBurbleFlow` class, it implements the DOR bubble routing. This routing reads packets from the input port and calculates to which, of the router ports, they should be routed to. If the packet is to be routed onto another dimension, the routing checks that the buffer lying in the new dimension has space for, at least, two more packets. This is the bubble condition. The packets have one header flit, specifying the number of hops, for each dimension. Therefore, before routing the packet to another dimension, the routing cuts the first header flit.

The methods that the `PRZRoutingFlow` class provides include some utility functions, that are called by the derived classes, and some virtual methods, that are called by the simulator. Among the first group there are:

`cutCurrentHeader` tells the caller if it should remove the current header flit

of the packet. This is done by finding if the packet has reached the end of the current dimension.

`changeDirection` informs the caller if a flit should be routed to a new dimension. This is done by comparing the routing port required by the packet and the dimension on which the routing is located.

`getDirection` tells the caller on which dimension lies the component.

The virtual methods overridden by the `PRZRoutingFlow` class include two that are involved with an event queue that is used to implement the delays on the component's activity. These are:

`outputWriting` finds out if there is an event that should be processed and calls `dispatchEvent` to process it if necessary.

`dispatchEvent` process events. Events in routing components are very simple, they specify at what time should the message be sent through the output. The `dispatchEvent` method sends the message only if the `controlAlgorithm` method returns `true`, if not it generates a new event that should be processed on the next cycle. Performing, in this way, a retry mechanism. The `controlAlgorithm` method should be overridden by the derived class to properly control the sending of messages.

In addition to these methods there are `onReadyDown` and `onStopDown` that manage the flow control signals.

Mainly, the derived `PRZRoutingBurbleFlow` class needs only to override three methods, these are:

`onReadyUp` the simulator calls this method whenever there is a flit that can be read. When called, this method reads the new flit and, if it is not a header, it sends it immediately to the output. If the flit is a header, it calls `updateMessageInfo`, cuts the header if necessary, generates an event to be processed when the header delay has elapsed and tells the previous component to stop sending flits.

`updateMessageInfo` reads the message and calculates to which port it should be routed. Because the routing information is distributed between the header flits, this is done in several steps. The method return `true` if the packet must change dimension.

`controlAlgoritm` this method is called by the `dispatchEvent` method to know if it should send a message. It checks that the bubble condition is true before allowing the message to be sent.