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.

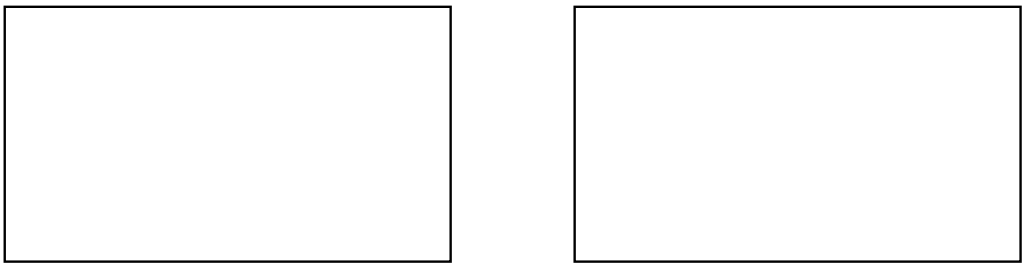

Figure 2.1: Comparison between SICOSYS and a VHDL simulator (Leapfrog).

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.

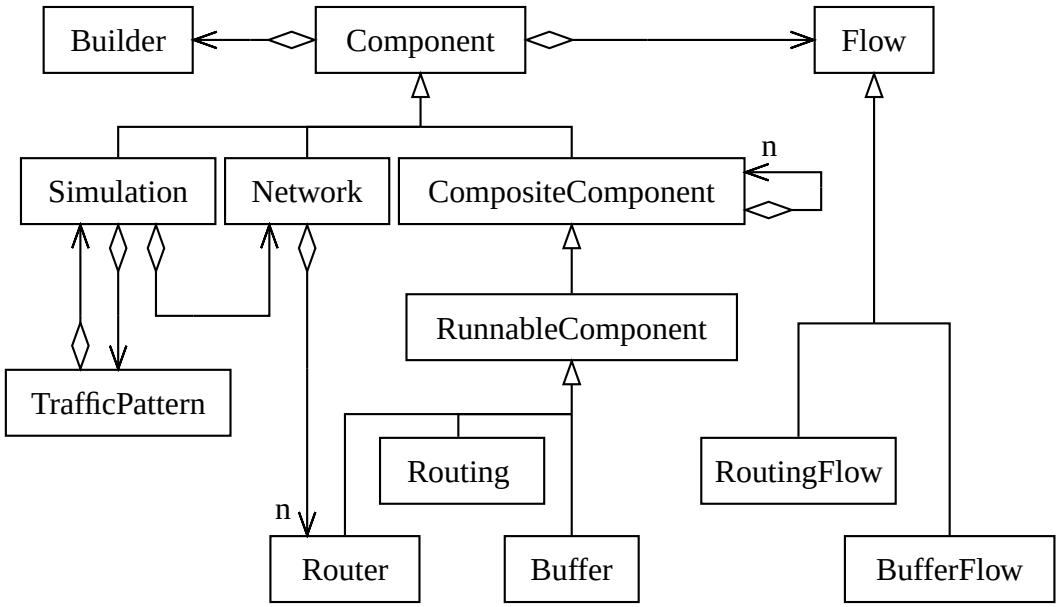

Figure 2.2: Simplified class diagram

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.

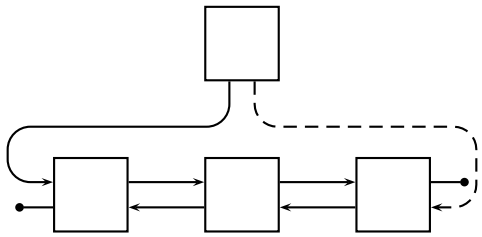

Figure 2.4: Modification of collection classes

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.

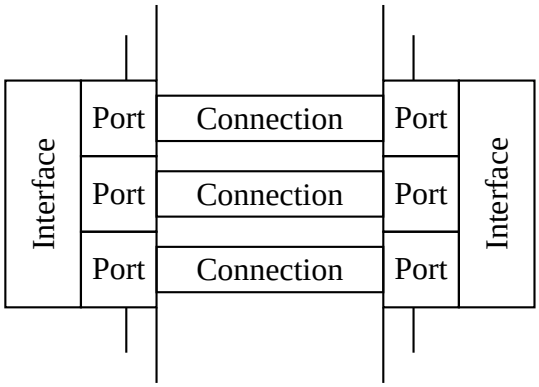

Figure 2.5: Structure of a connection between two components with three virtual channels

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.

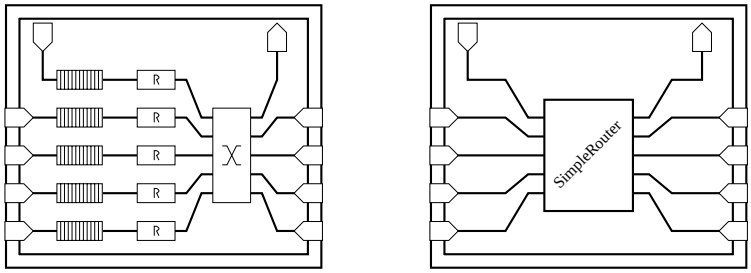

Figure 2.6: Substitution of the router components by a single one

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.

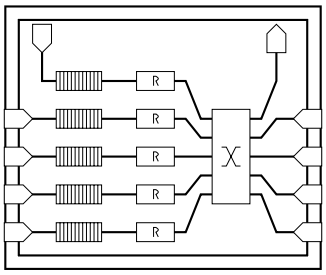

Figure 2.7: Diagram of the simple bubble router

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.

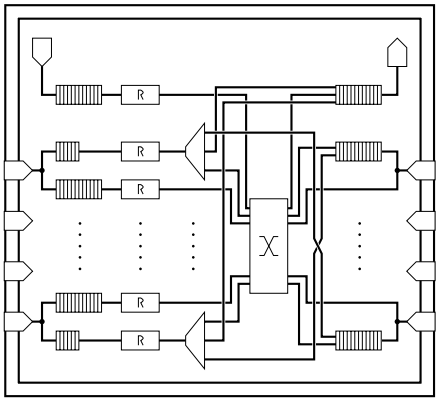

Figure 2.8: Diagram of the complex adaptative router

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].