Contents lists available at ScienceDirect

Simulation Modelling Practice and Theory

journal homepage: www.elsevier.com/locate/simpat

Simulating and evaluating interconnection networks with INSEE

Javier Navaridas *, Jose Miguel-Alonso, Jose A. Pascual, Francisco J. Ridruejo

Department of Computer Architecture and Technology, The University of the Basque Country, P. Manuel de Lardizabal 1, 20018 Donostia, Spain

ARTICLE INFO

Article history:
Received 12 May 2010
Received in revised form 13 July 2010
Accepted 17 August 2010
Available online 22 August 2010

Keywords:
Interconnection networks
Modeling and simulation
Performance evaluation

ABSTRACT

This paper describes INSEE, a simulation framework developed at the University of the Basque Country. INSEE is designed to carry out performance-related studies of interconnection networks. It is composed of two main modules: a Functional Simulator of Interconnection Networks (FSIN) and a TRaffic GENeration module (TrGen), together with several other modules that extend INSEE's functionality. The router models and topologies implemented in FSIN are included in this description. Likewise, the available methods to generate traffic are described thoroughly. Finally, the resource consumption of INSEE when managing different systems and workloads is also evaluated. INSEE has been used to conduct a variety of studies, including evaluation of novel topologies, diagnosis of causes of network congestion and evaluation of parallel scheduling algorithms.

1. Introduction

Supercomputing is a very valuable resource which is continuously growing in importance in business and science. Most current scientific studies rely on modeling and analyzing different natural phenomena and/or technological processes, which often require a huge amount of computing power impossible to attain using regular *off-the-shelf* computers. For instance, physicists, chemists or pharmaceutical researchers simulate, for different purposes, interactions between huge amounts of molecules. Likewise, in the business context, corporations demand large amounts of computing power in order to use data mining software over large databases, with the objective of extracting knowledge from raw data, and to use that knowledge to their advantage. Obtaining patterns of consumer habits, boosting sales, optimizing costs and profits, estimating stocks or detecting fraudulent behavior are just a few interesting application domains. At any rate the required computing power is only limited by the available resources. In general, when resources are increased then the number of runs, the grain-size (whichever this means in the particular application context), the size of the datasets or any other parameter that affects execution time is increased accordingly to fully utilize the compute power. In other words, the magnitude of the experiments is scaled up to the available assets. Briefly, there is a permanent demand of supercomputers able to cope with these challenging workloads.

A supercomputer is not only a piece of hardware. It is actually a multipart system that integrates a large collection of hardware and software elements. Therefore, the design of a supercomputer is a complex task that comprises the selection and design of multiple components, such as compute elements, storage, interconnection network, I/O infrastructure, operating system, high performance libraries and parallel applications. The performance of all these components has to be properly evaluated in order to select the most effective (again, the exact meaning of *effective* depends on the context) taking into account the purpose of the system and the workloads that are planned to be executed on them. Furthermore, the complete

* Corresponding author.
E-mail addresses: javier.navaridas@ehu.es (J. Navaridas), j.miguel@ehu.es (J. Miguel-Alonso), joseantonio.pascual@ehu.es (J.A. Pascual), franciscojavier.ridruejo@ehu.es (F.J. Ridruejo).

1569-190X/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.simpat.2010.08.008

system has to be evaluated as a whole, because unexpected interactions among components can make the system suffer severe performance losses.

The interconnection network (IN, in short), a specific-purpose network that allows compute nodes to interchange messages with high throughput and low latency, is a key element of any supercomputer because its performance has a definite impact on the overall execution time of parallel applications, especially for those that are fine-grained and communication intensive. Any delay suffered by the messages while traveling through the network will harmfully affect the execution time of applications. This is the reason why we should not decide lightly about the network that interconnects compute nodes in a high-performance computing site. The evaluation of an IN is a complex task that requires, among other concerns, deep knowledge about how parallel applications make use of the network.

Our interest revolves exactly around this topic: the evaluation of INs. In our research, simulation is the key means to evaluate them. The tool to carry out these simulations is INSEE, an Interconnection Network Simulation and Evaluation Environment. A preliminary description of the design of INSEE was performed in a previous paper [40]; however, INSEE has evolved noticeably since then. Some of the features explained in that paper were *planned*, and now they are a reality. Enhancements and modifications have improved markedly the usefulness of the tool in many aspects: execution speed, memory requirement, router models and topologies, workload generators, attainable statistics, etc. For this reason in this paper we review the updated version of INSEE, describing all its current features. We also include references to several research works in which INSEE has been used as the main evaluation tool.

The two main characteristics of INSEE when compared to other simulation tools are *flexibility* and *frugality* in the use of resources. As will be seen throughout the paper, INSEE can be used to simulate a wide variety of topologies and router models, and the networks can be fed using traffic models with different degrees of fidelity to actual application traffic. This flexibility may tip the scale in favor of INSEE when it comes to select a simulation tool. Furthermore, the requirements to build and use INSEE, in terms of memory and CPU speed are frugal, and therefore it may be the environment of choice for quick deployment and fast obtention of results. It provides the capability to simulate, on a desktop computer, systems composed by tens of thousands of nodes in reasonable time, hours or a few days at most.

INSEE basically consists of a set of ANSI-C source files that can be compiled with any compliant compiler. It can be built and executed without problems in both POSIX and Microsoft Windows environments. Most simulation parameters are given at execution time, so that only a few decisions have to be taken at compilation time, which, in turn, simplifies compilation. These decisions affect the complexity of the data structures included in the simulation, which increase memory needs. For example, as trace-driven and execution-driven simulation require specialized data structures for their operation (as will be described later), giving support to these modes is optional and can be defined at compilation time. The source code of INSEE (released under GPL) together with the required information for its operation (user manual) can be found at SourceForge [50].

For trace-based studies, the availability of a customized MPICH implementation is needed only to obtain the traces [22], but not to use them. A Simics installation is required for full-system simulation. Special care has been taken regarding memory footprint, in order to be able to cope with large-scale networks. The amount of required memory varies depending on the characteristics of the simulation (number of routers, virtual channels, buffer sizes, complexity of injected traffic, etc.). Some extreme configurations simulated in an off-the-shelf desktop computer were: a massively parallel high-performance system with the size of the largest BlueGene/L ($64 \times 32 \times 32$ nodes) [9], a complete SpiNNaker network (256×256 nodes) [24], and tree-like topologies connecting over 16 K nodes [26]. It is to be said that the main limiting factor is the amount of RAM (2 GB in our environment), as the simulation were executed quite fast (in the order of hours). With more RAM (a resource that is increasingly cheap and available) it would be possible to deal with even larger configurations; note that, in that case, the

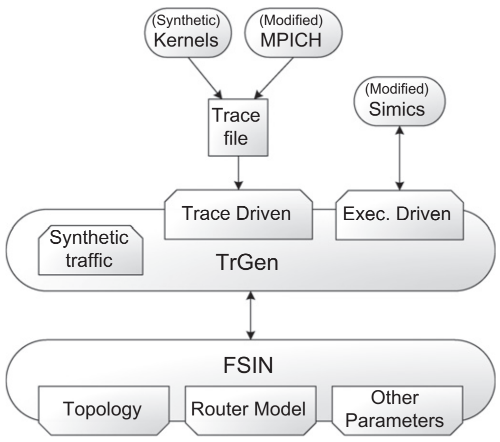

Fig. 1. Overall design of INSEE.

execution time may become the main limiting factor. For more details on resource requirements of INSEE, reader can refer to Section 5.

INSEE is composed of two main modules: FSIN and TrGen. A schematic depiction of the structure of INSEE is shown in Fig. 1. FSIN, described in Section 2, is a Functional Simulator of Interconnection Networks that allows modeling different topologies and routers. TrGen is a TRaffic GENeration module that is in charge of feeding FSIN with the desired traffic model. INSEE also includes some other additional modules that interface with TrGen to provide application-driven workloads: a modified MPICH library to obtain traces, and some modules implemented within Virtutech's Simics to carry out full-system simulation. TrGen and its companion modules are explained in Section 3. Section 4 discusses the limitations of our environment. A performance evaluation of the simulation environment is performed in Section 5, in which we show the memory requirement and execution time of several configurations of interest in our research. Section 6 performs a review of several related simulation environments, showing how they differ from INSEE. Finally, Section 7 closes the paper with some concluding remarks and an outlook of future plans to improve INSEE.

2. FSIN

The core of our environment is FSIN, a flexible, lightweight functional simulator (meaning that the router functionality is modeled in detail, but the hardware is not) that allows us to rapidly assess the performance of large-scale systems. Time is measured in terms of an abstract *cycle*, defined as the time required by a routing element to route and forward a *phit* (physical transfer unit). FSIN is able to simulate a wide variety of router models and topologies; we review those in this section. We want to remark that FSIN can be expanded in functionality by means of code additions/modifications. For example, to add a new topology, the only requirement is to formulate neighborhood and routing functions (using the C programming language). Other additions of higher complexity, such as adding a new router model, may require a deeper knowledge of the internal data structures.

2.1. Topologies

Several topologies, both direct and indirect, have been implemented in FSIN. Some of them are widely used in actual systems and thoroughly studied by the community. Some others are not as well known, and have been the subject of our own research work.

2.1.1. Direct topologies

Direct topologies are those in which every switching element or router is connected to a compute node. FSIN has topological models of standard meshes and tori for 1, 2 and 3 dimensions. Furthermore models for several alternative topologies have been added: two- and three-dimensional twisted tori, the Midimew topology and the SpiNNaker topology. All these

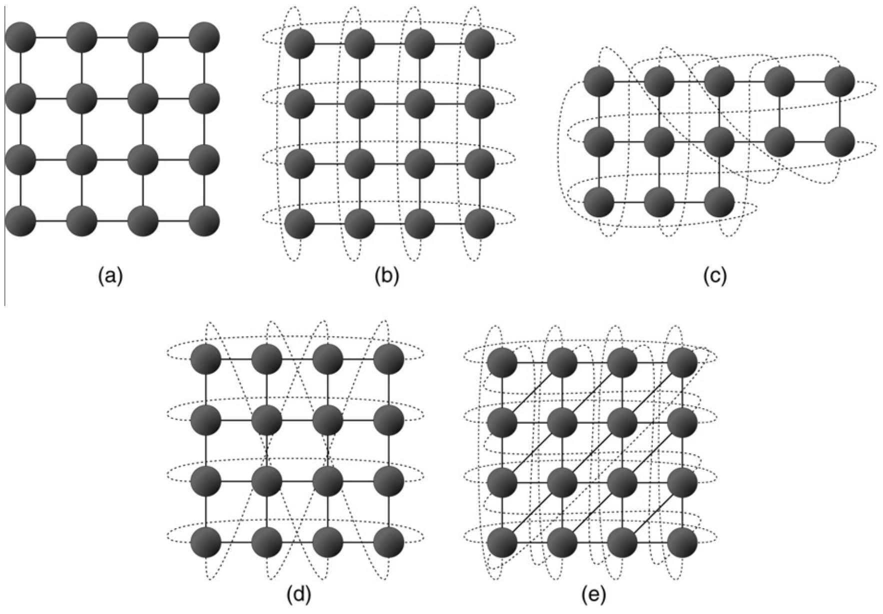

Fig. 2. Examples of the direct topologies implemented in FSIN: (a) 4 × 4 mesh; (b) 4 × 4 torus; (c) 13-node midimew; (d) 4 × 4 twisted torus with skew 2 and (e) 4 × 4 SpiNNaker topology.

topologies will be discussed in the following paragraphs, and are depicted in Fig. 2. In these graphs, nodes (and their corresponding routers) are represented by dark circles and links are represented by lines between them. Note that in FSIN all direct topologies (but meshes) can be built using unidirectional or bidirectional links.

The implemented direct topologies are the following:

- A *mesh* [11] is probably the simplest direct topology; nodes are arranged in a *d*-dimensional array and identified by their Cartesian coordinates. Some links at the peripheral nodes are disconnected—there are no wrap-around links. A 4×4 mesh topology is shown in Fig. 2a. This topology was historically used in high-performance computing systems because of its ease to scale up, as new nodes can be attached to the unconnected links. Currently, this topology has been abandoned in this context, in favor of networks with better topological characteristics, such as the torus. It is noticeable that the mesh topology is gaining popularity in the context of networks-on-chip because of the simplicity to deploy in silicon [23].
- The *torus* [11] is another well-known topology that has been historically used to interconnect massively parallel processors. Nodes in a torus are arranged in a *d*-dimensional array and identified by their Cartesian coordinates. In opposition to the mesh, the nodes in the boundaries of the topology are connected among them by means of wrap-around links. A 4×4 torus is shown in Fig. 2b.
- The *midimew* [7]—standing for MInimal DIstance MEsh with Wrap-around links—is a two-dimensional symmetric topology based on circulant graphs. It provides the best distance-related characteristics for any given number of nodes using routers of radix 4. A representation of a 13-node midimew network is shown in Fig. 2c.
- The *twisted torus* [9] is an optimization of the regular torus as it provides, for the same number of nodes, better topological characteristics: bisection bandwidth, average and maximum distance. An interesting property of this topology is that, for mixed-radix networks in which the number of nodes in one dimension doubles the number of the other dimensions ($2a \times a$ and $2a \times a \times a$) symmetry can be maintained, which help balancing the use of the channels of the different dimensions: the network does not present bottlenecks. A representation of a 4×4 twisted torus with a twist of 2 from dimension *Y* over dimension *X* is shown in Fig. 2d.
- The *SpiNNaker* topology [34] is a triangular toroidal network, in other words, a torus network with additional diagonal links to add redundancy to the design. This redundancy is deliberately devised to be exploited for fault-tolerance. A 4×4 instance of the SpiNNaker topology is represented in Fig. 2e.

2.1.2. Indirect topologies

Indirect topologies comprise all topologies in which there are switching elements that are not directly attached to computing nodes. This includes multi-stage and multi-level topologies. In multi-stage interconnection networks (MINs) all the traffic within the network flows in the same direction (usually represented from left to right) and the distance between every pair of nodes is the same: the number of stages of the network. In contrast, in multi-level topologies, the traffic must go up from the source to one minimal common ancestor of source and destination, and then down to the destination. This way, the distance between two nodes depends on the number of levels needed to reach a common ancestor.

Two indirect, tree-based multi-level topologies are implemented in FSIN, the *k*-ary *n*-tree and a reduced version of this well-known topology that we have called *k*:*k*′-ary *n*-thin-tree. In the graphical representations of the topologies (see Fig. 3), boxes represent switches and lines represent links between them. Note that we neither show the compute nodes connected to the first-level switches and their links, nor the last-level of upward links (which are left unplugged). These elements are hidden for the sake of clarity. We call the relation between the number of downward ports of a switch and the number of upward ports the *slimming factor*. For example, taking a look at the switches in the topology shown in Fig. 3b, four ports are downward ports, linked to switches in the next lower level. The remaining two ports of each switch are upward ports that connect to switches in the next higher level; therefore the slimming factor is the relation between 4 and 2, that is, 2:1 (or simply 4:2). Details of the indirect topologies implemented within FSIN are as follows:

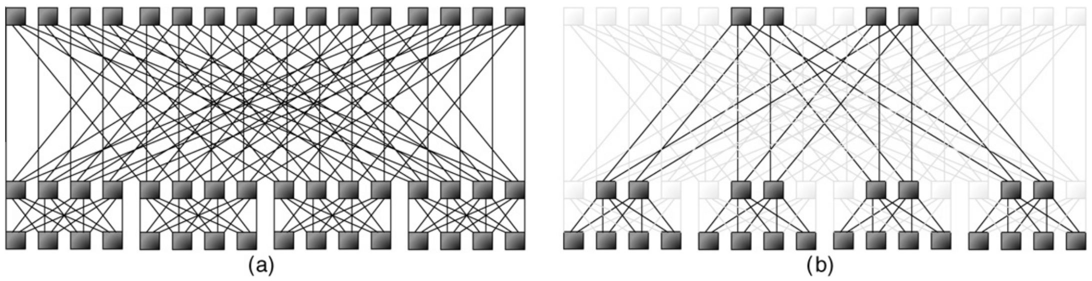

Fig. 3. 64-Node example networks of the multi-stage topologies: (a) 4-ary 3-tree and (b) 4:2-ary 3-thin-tree.

- In *k-ary n-trees* [33], k is half the radix of the switches—actually, the number of links going upward (or downward) from the switch—and n the number of levels. They can be seen as particular cases of the thin-trees (to be formally defined later), with a slimming factor 1:1. Fig. 3a shows a depiction of a 4-ary 3-tree. The main advantages of this topology are the high bisection bandwidth and the large number of routing alternatives for each pair of source and destination nodes—a path diversity that can be exploited via adaptive routing. Nevertheless, it might be expensive and complex to deploy, because of the large number of switches and links required.
- We define a *k:k'-ary n-thin-tree* [26] as a cut-down version of a k-ary n-tree in which we apply a given slimming factor. k is the number of downward ports, k' is the number of upward ports and n is the number of levels. The slimming factor is, obviously, the ratio between k and k'. Note that k does not need to be a multiple of k' so that we can produce a thin-tree with arbitrary values of k and k'. A 4:2-ary 3-thin-tree is depicted in Fig. 3b. Note the shadowed switches and links, that represent those elements that would be removed from a complete 4-ary 3-tree to form the thin-tree. In this topology the bisection bandwidth has been reduced, as well as the number of switches and links. For this reason, thin-trees are easier to deploy than regular trees and, if k and n values are kept, the radix of switches is smaller.

2.2. Modeled routers

Models of several router architectures with different purposes have been implemented in FSIN. The Dally router and the bubble router are designed to be used in direct networks, arranged in the typical 2D or 3D topologies commonly used in massively parallel processors (MPPs). The multi-stage switch allows the simulation of tree-based, indirect topologies, such as those used in many large-scale clusters. Finally the SpiNNaker router is a bespoke router architecture designed for fault-tolerance and low power consumption.

2.2.1. Dally router

A router architecture designed to be used in toroidal k-ary n-cube topologies was introduced by Dally in [10]. This architecture, which we call *Dally* router, uses a virtual channel scheme that allows deadlock-free routing using two virtual channels as Escape channels following oblivious dimension-order routing (DOR). The cycles embedded in each dimension ring are eliminated by means of restrictions in the use of the Escape channels, which cut the physical cycle into non-cyclic combinations of virtual channels. In Fig. 4, a depiction of the behavior of several modes of this scheme is shown. Adaptivity is possible increasing the number of virtual channels: two virtual channels (usually 0 and 1) are used as Escape channels (using deadlock-free DOR) while adaptive routing can be used in the remaining ones. The overall arrangement (according to Duato's theorem [13]) is an adaptive, but deadlock-free routing algorithm. Several strategies to route packets are implemented in FSIN to be used together with the Dally router:

- In *trc* all packets are injected in the same Escape channel. Packets remain in that channel, until reaching the wrap-around link of a toroidal topology. Then, it is switched to the other one. In other words, the cycle is transformed into a spiral (acyclic) as can be seen in Fig. 4b.
- In *basic* those packets that have to cross the wrap-around links of a dimension—let us call them P_W—circulate through one Escape channel in that dimension. Those that do not have to use the wrap-around links—denoted P_D—use the other Escape channel. This way, the physical ring is split into two separate virtual chains, which do not include cycles, see Fig. 4c.

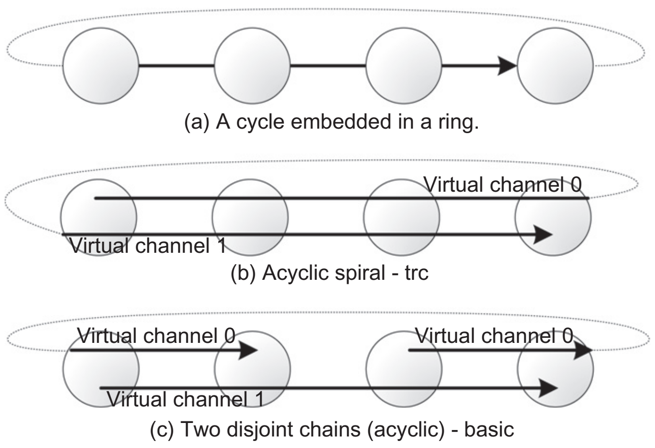

Fig. 4. Dally scheme to avoid deadlock in a unidirectional ring. (a) A cycle in a ring. (b) The cycle is cut by changing the virtual channel at crossing the wrap-around link (*trc* policy). (c) The cycle is removed by splitting the physical ring into two separate virtual chains (*basic* policy).

- *Improved* is an optimization of the basic policy to obtain better balancing in the utilization of both VCs. P_W packets are forced to transit by one of the Escape channels. P_D packets can use any of the two Escape channels. This way, P_W packets cannot block P_D packets, and thus P_D packets eventually will use the other channel and reach their destination. Note that this policy may lead to starvation of P_W packets, as P_D packets may make intensive use of the two Escape channels.
- *Adaptive* works as the improved policy, but adding routing adaptation capabilities. Two virtual channels work as Escape channels following the improved policy and the remaining virtual channels are used as adaptive channels (always using minimal paths). The packets randomly select a viable adaptive output port and, only in the case that no adaptive channel is available, the packets try to use an Escape channel as dictated by the improved strategy (P_W packets are restricted to one of the Escape channels, while P_D packets are free to use any of the two Escape channels).

The modeled router has queues in the input ports, and a crossbar that interconnects input and output ports as depicted in Fig. 5. Note that the crossbar has as many ports as virtual channels (plus additional ports for injection and consumption). Therefore, the utilization of VCs increases crossbar complexity.

2.2.2. Bubble router

The *bubble* router [39] is also designed to be used in *k*-ary *n*-cube topologies, having the same internal architecture of the Dally router (see again Fig. 5) but follows a different approach to avoid deadlocks; instead of avoiding cycles, this scheme avoids all the buffers in a ring (cycle) becoming full. To do so, packets can only enter an Escape channel from injection or from other channel when there is room to store at least two packets, one for the entering packet and another one to ensure that there is, at least, one free slot (*bubble*) to be used by the packets inside the Escape channel. This behavior, depicted in Fig. 6, requires a single Escape channel to avoid deadlock. In the depiction, only node 0 and node 3 can inject into the ring, as they have room enough in the queues of the ring to comply with the bubble restriction. Alternatively, the other two nodes (node 1 and node 2) are not allowed to inject as they do not comply with that restriction. Adaptivity can be easily incorporated by adding additional virtual channels with adaptive routing, but always checking the bubble restriction when moving packets from an adaptive channel to the Escape channel.

The bubble router can route packets following several strategies to decide, when a packet is awaiting at the head of the input queue (or at the injection queue), to which output VC a request will be made to forward the packet.

- With *oblivious* request mode all VCs follow the bubble restrictions (all of them behave as Escape channels), using DOR routing. Once a packet enters into a VC, it never abandons it—in short, there is no adaptivity.
- When using the **random** request mode, the router tries to route packets waiting at an input port through the output port of the same VC. If this is not available, any profitable adaptive VC can be requested, choosing it randomly. Finally, if no profitable adaptive VC is available then packets try the Escape channel.
- The **shortest** request mode works as the previous one but, when trying a profitable adaptive VC, selection is done considering the space available in the channel's queue, choosing the one with more room.
- If the **smart** request is selected, a packet is injected initially into a random channel and tries to continue in the same dimension and virtual channel. If this is not possible, then it is moved to a profitable adaptive virtual channel in another dimension. If no adaptive, profitable channel is found, the packet tries to move to the Escape channel. This strategy tries to avoid congested links by changing the traveling dimension every time it reaches a port that is in use.

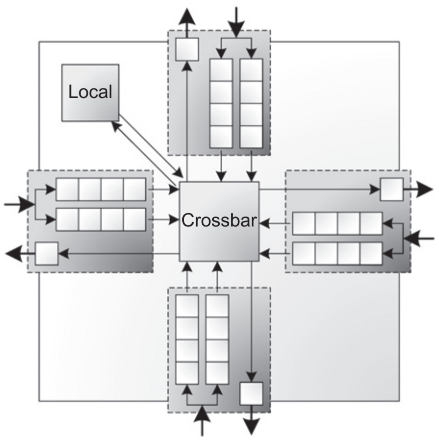

Fig. 5. Architecture of Dally and bubble routers for a 2D topology and two virtual channels per physical link.

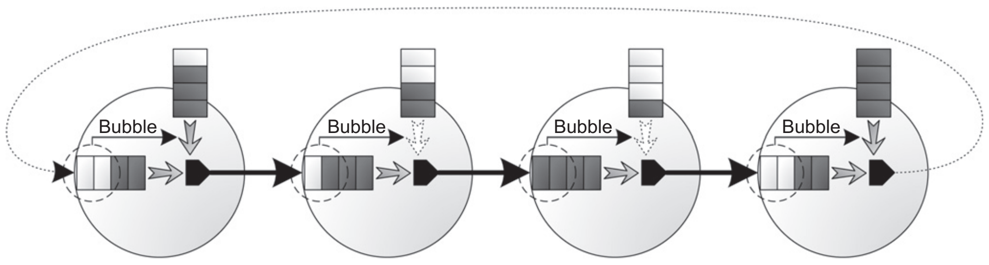

Fig. 6. Bubble scheme to avoid deadlock in a unidirectional ring. Grey arrows represent allowed data movements. Dotted arrows represent not allowed data movements.

Several other experimental strategies have been implemented for the bubble router but are not discussed here for the sake of brevity.

2.2.3. Multi-stage router

Routers for multi-stage INs are simpler than those for k-ary n-cubes, as routing in multi-stage topologies is deadlock-free provided that acyclic routing algorithms (such as up$*$/down$*$ [44]) are used. The router has a centralized crossbar and can have any arbitrary number of ports and virtual channels. Queues are located at the input ports. The model of the router for multi-stage topologies is depicted in Fig. 7. Routing can be static or adaptive, but in both cases shortest path routing is used to guarantee deadlock-freedom:

- In *static* routing the upward path is defined statically and depends on the source of the packet (*source_id* mod k). The downward path is also static and depends on the destination of the packet (*destination_id* mod k). Furthermore, if physical links are split into several virtual channels, one virtual channel is randomly selected at injection and the packet never leaves it.
- When using **adaptive** routing the downward path is static and depends on the destination, but the packets can adapt when traveling upwards. A credit-based adaptive routing is used, which works as follows: a packet at the head of an input queue tries to go through the profitable output channel with most available room in the queue of the neighbor input port (credit). If several output channels have the same credit, one of them is selected randomly. Note that it is assumed that credit transmission is performed out-of-band and, therefore, does not interfere with regular traffic.

2.2.4. SpiNNaker router

The SpiNNaker router [34] is also implemented within INSEE. This is a specific-purpose router designed for a large-scale machine with austere hardware constraints. The main focus of the SpiNNaker system is on low power consumption and fault-tolerance, and the design of the router also follows these concerns. As crossbar-based routing engines require excessive hardware resources, they can not be part of the SpiNNaker router; therefore, it was designed using a more frugal approach. All the ports of the router are hierarchically merged in such a way that the routing engine is accessed by a single packet at once. This way, a packet that can not be forwarded will force all the packets in the router to wait until it is forwarded or discarded, an undesired effect known as Head-of-Line blocking. Fortunately, as the routing engine is relatively faster than the transmission ports, this situation is unlikely to happen. A depiction of the architecture of the router is shown in Fig. 8.

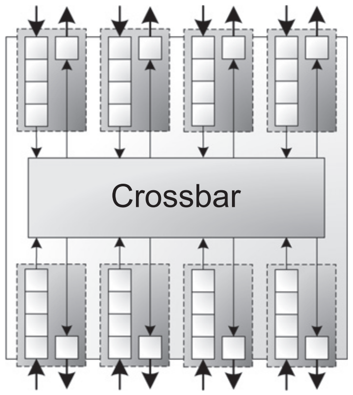

Fig. 7. Architecture of a multi-stage router with eight ports and one virtual channel per physical link.

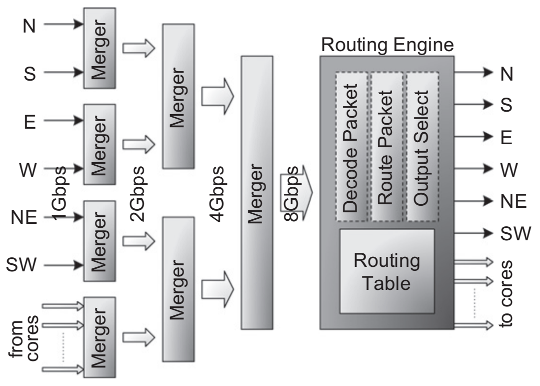

Fig. 8. Architecture of the SpiNNaker router.

Routing is performed by means of the routing tables inside each router. However instead of being destination-based, the routing tables choose the output port(s) for a given packet taking into account the source of the packet. This behavior makes the source nodes and the packets unaware of the destination node(s). The way routing tables are filled is application-dependent.

2.2.5. Other router parameters

The *phit length* (physical transfer unit, measured in Bytes) and the *packet length* (measured in phits) can be set to any arbitrary value. Note that the packets have a fixed length and when a smaller packet is required, it will carry some empty phits. The *transit queue length* and the *injection queue length* (measured in terms of room to store packets) can be set to any arbitrary value larger than one packet. Similarly, any desired number of virtual channels can be used.

The *arbitration* of output ports (selection among all the requesting input queues) can be performed in several ways; some of them are unaware of the traffic, and some others take into account some attributes of the packets to give priority to certain flows. The arbitration policy can be defined for all the router models except the SpiNNaker router, because it treats packets once at a time. The defined policies are the following:

- *Round Robin* arbitration.
- *Random* arbitration selects randomly among all the requesting input ports.
- *FIFO* arbitration selects the first input port that requested the output.
- *Longest* arbitration selects the input port having the highest queue occupation.
- *Age* arbitration selects the input port containing the oldest packet, measured since the packet was injected into the network.

Furthermore, we have implemented and evaluated several *congestion control mechanisms* around the Dally and the bubble routers. Two of them are *local* mechanisms in which routers detect congestion taking into account only the state of their own queues. The other one is a *global* mechanism that throttles the injection taking into account the state of the whole system. These mechanisms are described as follows:

- In-transit Priority Restriction (IPR). For a given fraction P of cycles, priority is given to in-transit traffic, meaning that, in those cycles, injection of a new packet is only allowed if it does not compete with packets already in the network. P may vary from 0 (no restriction) to 1 (absolute priority to in-transit traffic). This mechanism can be used along with the two router models and is the method applied in the IBM's BlueGene/L torus network [2].
- Local Buffer Restriction (LBR) mechanism was designed specifically for adaptive bubble routers. A previous study showed that the bubble restriction, in addition to guaranteeing deadlock-freedom, also provides congestion control for the Escape sub-network [17]. LBR extends this mechanism to all new packets entering into the network. That is, a packet can only be injected into an adaptive virtual channel if such action leaves room for at least B packets in the transit buffer associated to that virtual channel. The parameter B indicates the number of buffers reserved for in-transit traffic. In other words, congestion is estimated by the local buffer occupancy.
- Global Congestion Control (GCC) estimates network congestion by examining the status of the whole network. Injection is stopped when network occupancy reaches a given threshold G, which is given as a fraction of the whole network capacity. In our implementation, the network utilization is measured every T cycles (being T a simulation parameter), and then transmitted out-of-band to all nodes, in such a way it does not interfere with application packets.

2.3. Node model

In INSEE, nodes are modeled in a rather simplistic way: a traffic generator/consumer plus an arbitrary number of injection queues or *injectors*. In the case of direct topologies, nodes are attached to the routers through a specialized connection. Several *injection* policies are implemented to be used with direct topologies:

- *Shortest* is the simplest injection policy. It inserts a newly generated packet into the injection queue with more room.
- When using *dor* policy, there is a dedicated injection queue per network dimension/direction, which significantly reduces Head-of-Line blocking effects [17]. A dimension-order pre-routing phase is carried out to decide in which queue will be injected the new packet. If the selected queue is not available, the injection will be stopped.
- The *dor + shortest* policy tries to inject using the *dor* policy but, if the selected injection queue is full, then it uses the shortest policy.
- In the *shortest profitable* policy, the packet will be injected in the valid queue (once pre-routed) that has most empty space.
- *Longest path* policy injects packets in the queue associated to the direction through which the packet has to travel a larger number of hops.

Furthermore, the specialized connection of direct routers allows for two alternative *consumption* modes:

- *Single* consumption policy considers the consumption port as a regular output port following the selected port-arbitration strategy.
- *Multiple* consumption policy assumes a consumption port wide enough to accept simultaneously a phit from each input port, meaning that several packets can be consumed simultaneously.

In the case of indirect topologies, nodes are connected to the network through regular links with the same characteristics of the links within the network. For this reason, both injection and consumption are serialized and do not allow any special policy (shortest injection and single consumption).

2.4. Output data generated by FSIN

Depending on the kind of research work that is being carried out with INSEE, different types of results are needed: throughput/delay analysis, channel utilization, time to deliver a workload, dropped packets, system evolution, etc. For this reason FSIN is able to capture and report a wide variety of statistics from the performed simulations. These statistics can be captured for the whole simulation or, in contrast, can be averaged from different time intervals. For debugging purposes, cycle-by-cycle information can be generated as well. As the execution of FSIN is deterministic (given a seed used to generate random numbers, if needed), a summary report of the input simulation parameters is printed in order to allow repeating a given execution.

2.4.1. Run-time statistics

Run-time summaries of statistics can be obtained at pre-configured time intervals. This allows us to follow the evolution of the system. Note that applications usually pass through different stages of communication patterns and, therefore, of network usage. The statistics captured along the execution are the following (always related to the measurement time interval):

- *Injected and accepted load*: The average injection and consumption rates of network nodes, measured in phits/node/cycle.
- *Packets*: The number of packets that have been injected, consumed and dropped due to different reasons (blocked at the head or tail of injection queues and/or dropped in-transit).
- *Network utilization*: The average number of packets in the queues of the system.
- *Distance*: The (averaged) distance traversed by consumed packets.
- *Latency*: Several latency-related measurements are captured including: average, standard deviation and maximum latencies. Note that latencies are measured since the moment at which packets are generated, as well as since the moment they are injected into the network.

2.4.2. Traffic evolution mode

Using this mode, the wandering of packets and phits can be shown in order to have a better understanding of how the network evolves. Its main interest, though, is for debugging purposes. Selecting *packet evolution*, the simulator will show in its output the following events: packet generation, packet injection, packet arriving to a node, packet leaving a node, packet dropping and packet consumption. For *phit evolution*, the output of simulator will show the following events: phit generated, phit moved, phit dropped and phit consumed.

2.4.3. Final report

Once the simulation is concluded, FSIN shows a comprehensive report with a summary of the whole simulation. All the previously discussed figures are averaged for the whole simulation. In order to allow a better understanding of the results, the standard deviation and the maximum of all these figures are also shown, as well as the total simulation time, measured in cycles. The actual time required to perform the simulation is also provided.

If requested, FSIN can capture statistics to allow an even more detailed understanding of the simulation. A histogram of the queue occupancy along the simulation can be shown, which is useful to understand which (and how) network queues were occupied. This can be used, for example, to diagnose anomalies in network utilization.

A map of the load managed by each (virtual) port of the system can also be generated. This is useful to find network bottlenecks, and also to understand the virtual channel management strategy used by the system.

If our interest revolves around the distances traversed by packets, FSIN can show a histogram of distances, as well as a map showing pair to pair communication. Both histogram and map are generated for the distance distributions at the injection *and* at the consumption. If the captured data for injection and consumption are notably different, this can suggest that there are nodes that are suffering starvation. Note that the pair-to-pair map gives more information, but may require excessive memory if the system is large, as the memory requirements scales quadratically with the number of system nodes.

2.4.4. Using the results generated by FSIN

Performance data generated by a single run of FSIN can be useful for some studies, but often many runs with different random seeds are required in order to obtain statistically meaningful results. Additionally, for many studies a parameter sweep is required. It is up to the researcher to combine the results of the simulation runs in the form of suitable tables and graphs, although FSIN helps by means of outputting structured text files (in CSV format) that can be easily imported into spreadsheets and other data analysis tools. For illustrative purposes, Fig. 9 depicts some graphs obtained from FSIN results, which are similar to those used in our research works.

2.5. Validation of results

Given that, in most cases, we do not have access to the systems we simulate with INSEE, a direct comparison of the behavior of actual versus simulated systems can not be carried out. However, the results obtained with INSEE have been cross-validated with mathematical models of the different implemented topologies. These mathematical models include throughput

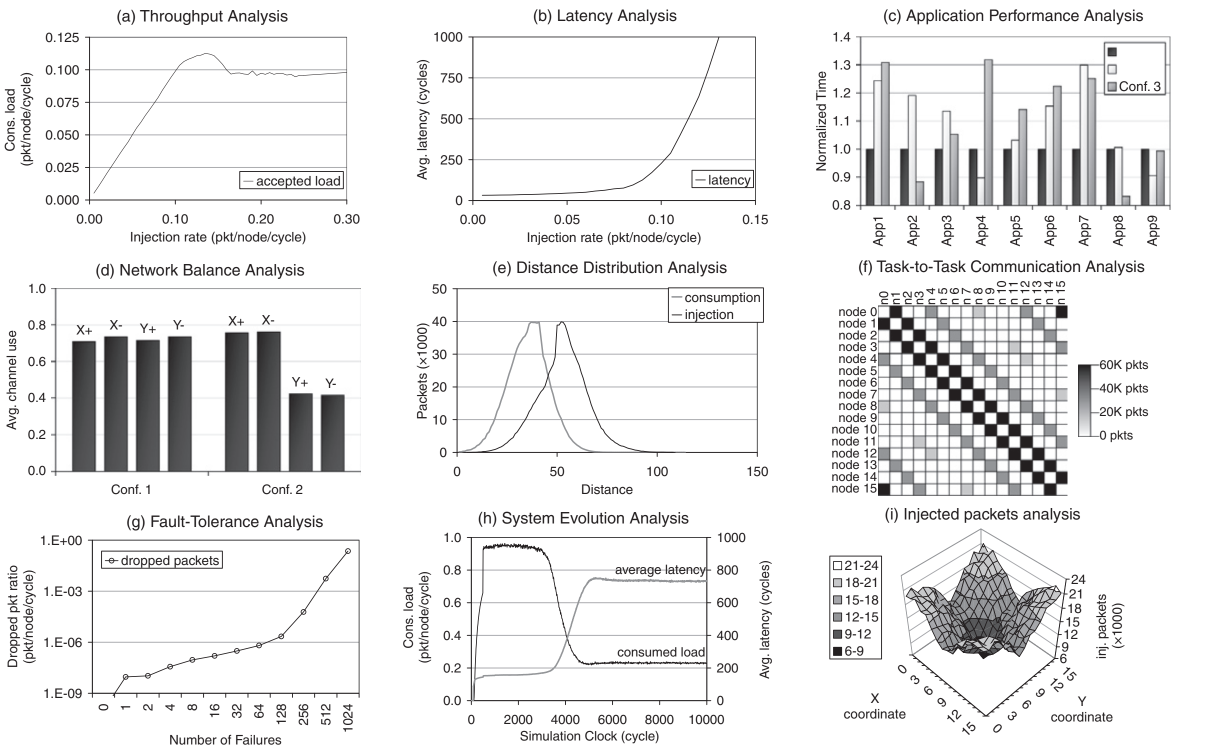

Fig. 9. Examples of IN analysis supported by INSEE: (a) throughput analysis; (b) latency analysis; (c) application performance analysis; (d) network balance analysis; (e) distance distribution analysis; (f) task-to-task communication analysis; (g) fault-tolerance analysis; (h) system evolution analysis and (i) injected packets analysis.

analysis for uniform traffic as well as distance-related characteristics of the topologies. In some of our previous works, the reader can find examples of those validations for different topologies: twisted tori [9], regular tori [21], 2D triangular torus such as the SpiNNaker topology [24] and tree-like topologies [26]. Interestingly, in [9], an erroneous mathematical model for the network throughput when managing uniform traffic was detected using INSEE.

We have also carried out cross-validation with other simulation tools, specially with SICOSYS [36,48], see details in Section 6. Results obtained with INSEE have always been consistent with those obtained with other tools, for similar configurations and parameters.

3. TrGen

TrGen—standing from TRaffic GENerator—is the module in charge of feeding the simulation with the desired kind of traffic. It interfaces with FSIN and passes the traffic in the form of network packets. TrGen is capable of generating purely synthetic traffic patterns, as well as of extracting communication patterns from traces or from the actual execution of parallel programs in a simulated computing environment (interfacing with Simics).

3.1. Some considerations about workloads

When it comes to evaluate the interconnection network of parallel or distributed systems, the way we model the workload imposed to the network is of crucial importance. In the first phases of the design of a system, it may be desirable to obtain results as fast as possible, even if they are not fully accurate. Alternatively, a complex model providing high fidelity may be required in the last phases just before deployment, to ensure that all the pieces of the system perform as expected when they are put together and also that the applications make efficient usage of the underlying hardware.

Synthetic traffic patterns from independent sources [11] provide a good first approach to evaluate a network, because they allow us to rapidly asses the raw performance of a network, and because it can be supported by any of the previously discussed analytical studies. Very often, randomly generated traffic is used to evaluate systems: uniform, hot region and hot spot traffic patterns have been used in a large collection of studies. Other commonly used patterns are those that send packets from each source node to a destination one as indicated by a certain permutation, defined as a function that takes as input the address of the source.

Nevertheless, actual applications that communicate internally using patterns like these synthetic ones, in which traffic-generating nodes produce messages without coordinating among them, are rare. For this reason, trace-driven simulation is often preferred, in order to perform a more realistic evaluation of a system. Feeding a simulator with a trace is not an easy task. To evaluate only the network of a parallel system we could implement a *dummy* model of the processing node, allowing it to inject messages into the network as fast as it can, ignoring the causality of messages and the computation intervals. This approach is a stress test of the network, because of the contention caused by all nodes injecting at the maximum pace.

Alternatively, it would be more realistic to maintain the causal relationship between the messages in the trace; in other words, if the trace states that there is a reception before a send, the node has to wait for that reception to be completed before starting with the send. This mechanism provides more fidelity than the inject-at-will model. To further improve the simulation accuracy, compute intervals (periods in which nodes do not inject load into the network) should be taken into account, maybe applying a *CPU-scaling-factor* in order to simulate a system with faster (or slower) CPUs than those used to capture the trace.

Still, there are some problems with the trace-driven approach that we should not ignore. Firstly, the information captured within the trace could be inexact due to the intrusion effects of the trace logging mechanism. Secondly, traces may reflect some of the characteristics of the system in which they have been obtained. Finally, traces from actual applications running in a large set of processors are difficult to obtain, store and manage, and these are precisely the ones of interest in our performance evaluation studies.

A hybrid between the utilization of synthetic traffic patterns and traces is the estimation of probability distributions for destinations, inter-generation times and message lengths, using data extracted from actual traces to feed some distribution-fitting program. For example, the spatial distribution of several application prototypes were shown in [5]. Once we have the distributions that model the application used to obtain a trace, we can generate random traffic resembling it. However, as stated before, in actual applications causal relationships among messages are common, and this technique does not capture them. And, again, the inexactitudes of the information within the trace (due to the characteristics of the system in which the trace was captured, and the intrusion of the logging process) may generate estimated distributions, or parameters of those, that are not valid.

In order to introduce causality in the simulation, and to fill the gap between trace-driven simulation and independent sources traffic, a bursty traffic model can be used. This model uses the previously discussed synthetic traffic patterns, but emulates application causality using a coarse-grained approach. The message generation process passes through a certain number of *bursts* or steps, during which nodes can inject at will until the burst finishes, and then stall until the starting of the next burst. Synchronization among nodes is included in this model, but in a very primitive way (roughly a barrier); fine-grained synchronization among messages/tasks are not considered.

A further refinement of the bursty model is by means of application kernels. These kernels are implemented using point-to-point synchronization and communication primitives, and can include different levels of causality such as long chains of dependencies. Application kernels emulate the behavior of small parts of actual applications, but when compared with regular traces they are more flexible because they are fully configurable in terms of number of communicating tasks, message size, task coupling, etc. This gives an advantage when compared to traces, as the latter are difficult to capture and manage for large-scale systems. Furthermore, as kernels are only small parts of applications, their execution is orders of magnitude faster than trace-based configurations, while still providing a reasonable level of accuracy. These application kernels are inspired in communication patterns observed in actual applications. In some cases they reproduce virtual topologies, or implementations of collective communication primitives, while in others they reproduce programming models such as master–slave.

Finally, the most accurate methodology to evaluate a parallel computer would be running a detailed full-system simulation that includes the interconnection network, the compute nodes, the operating system, some support libraries and the applications running on them. This is a very complex, error-prone task, as well as a high resource-consuming methodology that could need a system similar in dimension to the one we want to evaluate. These are the main reasons to justify the limited utilization of execution-driven simulation to evaluate medium-to-large size distributed memory parallel computers.

Table 1 closes this section summarizing the methodologies to generate traffic to fed simulations. Columns show the spatial pattern of the workload, the causality among messages, the complexity of generating, managing and performing simulation, and the kind of evaluation supported by the traffic model. Note that all these workload generation modes are supported by TrGen.

3.2. Synthetic workloads

Our simulation environment provides a wide variety of synthetic workloads that can be used to measure the performance of the communication infrastructure. These workloads have different levels of fidelity to actual application workloads, in terms of spatial and temporal/causal patterns.

3.2.1. Independent traffic sources

A widely accepted and used mechanism to feed simulations is the use of synthetic traffic patterns from independent sources [11]. This kind of workload allows tuning the injection rate of nodes; they try to inject at this rate, following a Bernoulli distribution. There is no causality among receptions and injections. When using this kind of traffic, the simulation run is split in three phases. The first phase simulates a given number of cycles without capturing statistics and is used as a warm-up phase. When warm-up finishes, a phase in which convergence is checked starts. During this second phase network statistics are captured every a given number of cycles (convergence intervals). If three consecutive intervals are within a given range, it is assumed that convergence has been reached. Finally a statistics-capturing phase starts. During this phase a given number of samples or batches are run, capturing the statistics during these phases, showing their average and their standard deviation.

The spatial traffic patterns supported by TrGen are the following:

Table 1
Description of different traffic models used for simulation-based evaluation of parallel systems.

Traffic model	Spatial pattern	Causality	Complexity	Evaluates
Independent traffic sources				
Random	Random	No	Very low	Raw performance
Permutation	Worst case	No	Very low	Raw performance
Estimation of distributions	Application-like (origin-dependent)	No	Low/medium	Selected application (origin-dependent)
Bursty traffic sources				
Random	Random	Coarse-grained	Very low	Raw performance
Permutation	Worst case	Coarse-grained	Very low	Raw performance
Estimation of distributions	Application-like (origin-dependent)	Coarse-grained	Low/medium	Selected application (origin-dependent)
Application-based				
Application kernels	Application-like	Application-like	Low/medium	Usual communication patterns
Trace-driven (inject-at-will)	Application-like (origin-dependent)	No	Medium	Raw performance when congested
Trace-driven (causal)	Application-like (origin-dependent)	Application-like (origin-dependent)	Medium/high	Selected application (origin-dependent)
Execution-driven	Actual application	Actual application	Very high	Selected application

- *Random*: When a packet is generated at a node (the source), the destination is randomly selected following a given probability distribution. The built-in modes are *uniform* (UN), in which all the nodes have the same probability of being selected as destination, and the non-uniform *hot spot* (HS) and *hot region* (HR), in which a given node or group of nodes, respectively, have higher probability of being selected as destination, increasing the risk of generating congestion in some regions of the network. In local (LO), the probability of selecting destination nodes decreases with the distance (so that most packets are sent to nearby nodes). Furthermore, TrGen can read and follow user-defined distributions (for example extracted from actual applications) which can be introduced as a *histogram* or as a *population*.
- *Distribution*: Distribution patterns send packets sequentially to each one of the remaining nodes. The initial destination node can be the next one in order of identifier (SD), or can be selected randomly (RD).
- *Permutation*: Given a source node, the destination node is always the same, and is computed as a permutation of the source node identifier (generally bit permutations) [11]. The permutations implemented in TrGen are the following: Bit Complement (BC), Bit Reversal (BR), Bit Transpose (BT), Butterfly (BU), Perfect Shuffle (PS) and Tornado (TO). Their mathematical descriptions are shown in Table 2. Identifiers of source and destination are denoted as s and d, respectively. For bit permutations, l is the number of bits, s_i and d_i are the ith bit of the source and the destination respectively. For the tornado permutation n_x is the number of nodes in the X and Y dimension of a 2D cube topology. $\langle s_x, s_y\rangle$ and $\langle d_x, d_y\rangle$ are the Cartesian coordinates in the 2D cube of the source and destination nodes, respectively.
- In some scenarios we study the performance of a parallel job composed of a collection of communicating tasks that uses only one part of the network. This job could run in solitary, while the non-used processors are empty. However, it would be more realistic if the job experienced interference from other jobs sharing the machine. To model these scenarios, is it possible to combine application-based (or application-inspired) workloads to realistically simulate the target job, while using traffic from independent sources (normally, random traffic) to emulate background network utilization. We used this combination in some studies on parallel application mapping [31].

3.2.2. Reactive traffic

As part of the evaluation of the SpiNNaker system we have implemented a traffic model that resembles biologically plausible neural traffic [29]. The definition of this traffic model is quite simple: simple independent traffic in which nodes may generate, with a given probability, a packet (or a collection of packets) after the reception of a packet. The parameters for this kind of simulation are the same that for the independent traffic (spatial pattern and injection rate) plus a probability to trigger new traffic and the number of packets triggered (given as an interval). Note that this behavior models the activation and firing of a neuron.

3.2.3. Bursts

Bursty traffic sources provide a simple model to introduce system-level synchronization when managing synthetic traffic, emulating the execution of a barrier synchronization operation every b packets. To do so, nodes are allowed to inject b packets as fast as they can, and then stall until all packets of the burst have been injected and consumed by all the nodes. Note that small values of b resemble tightly-coupled applications while large values of b resemble loosely-coupled applications. In this mode, the figure of merit to measure network performance is normally the time taken to deliver all the packets in one or several bursts, being the faster the better. All the spatial patterns discussed in Section 3.2.1 can be used with *bursty* traffic sources. This burst-synchronized behavior avoids starvation of nodes, as all of them are allowed to inject exactly the same amount of traffic into the network.

3.2.4. Application-inspired workloads

One of the contributions of our group, within the INSEE environment, is a set of synthetically generated traffic patterns that resemble the way actual scientific applications communicate. These traffic patterns are a further refinement of bursty traffic in which point-to-point synchronization is supported. They can be considered application micro-kernels as they mimic different communication patterns widely used on parallel applications. This mimicry is done both in terms of spatial patterns and causality. The utilization of this kind of workloads is based on the same infrastructure used for trace-based simulation, described later. A (standalone) INSEE module takes the appropriate parameters and generates a synthetic trace file. This file can be used in a simulation as a regular trace file.

Table 2
Mathematical description of the permutation patterns implemented in TrGen.

Pattern	Destination	Example
Bit Complement	$\forall i: 0 \leqslant i \leqslant l-1, \quad d_i = \sim s_i$	BC(11011000) = 00100111
Bit Reversal	$\forall i: 0 \leqslant i \leqslant l-1, \quad d_i = s_{l-i-1}$	BR(11011000) = 00011011
Bit Transpose	$\forall i: 0 \leqslant i \leqslant l-1, \quad d_i = s_{(i+1/2) \bmod l}$	BT(11011000) = 10001101
Butterfly	$d_{l-1} = s_0, \quad d_0 = s_{l-1}$	BU(11011000) = 01011001
Perfect Shuffle	$\forall i: 0 \leqslant i \leqslant l-1, \quad d_i = s_{(i-1) \bmod l}$	PS(11011000) = 10110001
Tornado [46]	$d_x = (\frac{n_x}{2} + s_x) \bmod n_x, \quad d_y = s_y$	$TO_{8\times 8}(\langle 3, 2\rangle) = \langle 7, 2\rangle$

We can arrange the application-inspired workloads into three different groups. Some of them are reproductions of the way MPI collective operations are implemented relying on point-to-point communications (this is, when hardware support for collectives is not available), taking into account optimized as well as non-optimized implementations. The second group includes communication patterns that mimic those applications that rely on virtual topologies, passing messages to immediate neighbors. The third group is more generic, implementing different modes of random generation of synchronized interchanges of messages. The exact definition of all these kernels can be found in [25,27] and is not discussed here for the sake of brevity. Application micro-kernels provide a reasonable level of accuracy at a reduced cost in terms of computing power required by the simulator. Moreover, they have the capability of generate workloads with thousands of communication nodes which are difficult, if not impossible to obtain with application-guided workloads (described next).

3.3. Application-guided workloads

Synthetic sources provide very useful insights into a network's potential. However, obtained performance metrics can be unrealistic as applications use more sophisticated communication patterns than synthetic models. For this reason TrGen can also use traces from applications to perform trace-driven simulation, and even interact with Simics to perform full-system simulation. This subsection is devoted to discuss these execution modes.

3.3.1. Traces

We use a modified version of MPICH [20], one of the most popular implementations of MPI, to obtain trace files usable with TrGen. MPICH includes an easy-to-use mechanism to obtain trace files from running applications. However, these traces are not useful for our purposes because collective operations (such as barriers, broadcasts and reductions) appear as such in the trace files, that is, the actual interchange of messages necessary to implement those operations in networks without native support for collectives is not reflected. Internally, MPICH implements collective operations using point-to-point operations (when no better alternative is available). We modified MPICH in order to make those point-to-point operations visible, registering them in the trace files along with the corresponding collective operation. The intervals of time between MPI operations are considered as computation intervals. A CPU-scale factor can be applied to these intervals in order to simulate processors faster or slower than those of the system in which traces were captured.

Trace files are slightly pre-processed before using them with TrGen. Only a few, relevant fields are selected (event type, source node, destination node, message size and tag) and organized in a simplified format more suitable for TrGen. It would be possible to use this format to feed simulations with traces obtained from sources different to MPICH, a network sniffer being a good example, just building the right pre-processor.

To reproduce the causal relationships between events in the trace files, TrGen requires a special data structure to store past and future events, shown in Fig. 10. Each node of the simulated applications has an event queue, which is fed from the trace file. A packet is sent to the network when an **S** (send) event is in the queue's head. If an **R** (receive) event is in the head, it is necessary to access the pending notifications queue to check if the expected event has happened already; otherwise, processing of events is blocked until the network notifies the awaited reception. The pending notifications queue at each node, thus, stores reception events that arrive before the application requests them, and it is a crucial element to keep event causality. The complete process of trace-driven simulation is as follows:

(1) Enqueue in each node's event queue all the events it has to execute.
(2) Initialize the pending notifications list as an empty list.
(3) Nodes sequentially execute the events in their event queue.
 (a) If the first event is a send, remove the event and inject the corresponding message into the network.

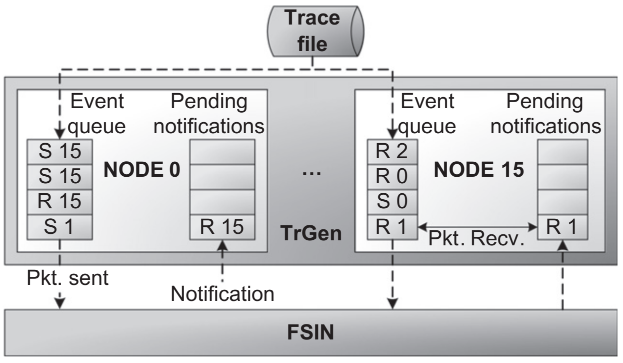

Fig. 10. Data structure used to maintain causality in the trace-driven simulation.

   (b) If it is a reception, check if a corresponding message (matching origin, destination, tag and size) is in the pending notification list. If it is there, remove both entries. Otherwise, keep in this state until the required message is received by the node and is accordingly found in such list.
   (c) If it is a computation event, put the node on hold for the required period of time, using if selected a CPU-scale factor.
(4) When the simulator delivers a message, put it in the pending notifications list.

An example of this procedure is depicted in Fig. 10, note that **R** represents a reception and **S** represents a send; computation events (**C**) are not considered for the sake of simplicity. In the figure, node 0 cannot advance, because it is waiting for a message from node 1, even if a message from node 15 has been already received. In contrast, node 15 can advance because the required message from node 1 has been delivered. This mechanism reproduces the actual way messages were interleaved when running the application, complying with the causal order between a reception and the subsequent sends it may trigger.

Traces obtained from one system are often used to evaluate via simulation the performance potential of another, different, target system. However, this approach has some drawbacks. In the context of IN design and evaluation, traces obtained with the same collection of nodes running a parallel application with two different INs A and B may be different, because properties of A and B differ, and those properties have an influence on the way nodes interchange messages. For this reason, performance results obtained with traces may not be totally accurate [14]. A thorough discussion of the design, limitations and issues of the trace-driven simulation within the INSEE environment was carried out in [22].

3.3.2. Interaction with Simics

TrGen can interface with Virtutech's Simics [19] to perform a full-system simulation. Simics is in charge of simulating a cluster of multiprocessors, while INSEE simulates the IN. In our experimental environment, each Simics instance is executed in a different computer and can simulate up to eight compute nodes (limited by the available RAM of our machines). A single instance of INSEE simulates the network that interconnects all the simulated nodes. In order to perform a correct simulation of the parallel system, a set of interfacing modules were implemented, some for INSEE and some others for Simics. These modules implement the following functions:

- Transference of application traffic from Simics instances to FSIN, and vice versa.
- Synchronization among all the elements taking part in the simulation (FSIN and the collection of Simics instances).

Every simulated computing node has a simulated Ethernet network interface card which is instrumented to put the messages into the Traffic manager of its Simics instance. The Traffic manager sends meta-information about the messages to TrGen and is in charge of sending the complete message to the corresponding Simics instance and node, once it has reached its destination in the simulated network. In Fig. 11 we can see a depiction of all the elements involved in the execution-driven simulation.

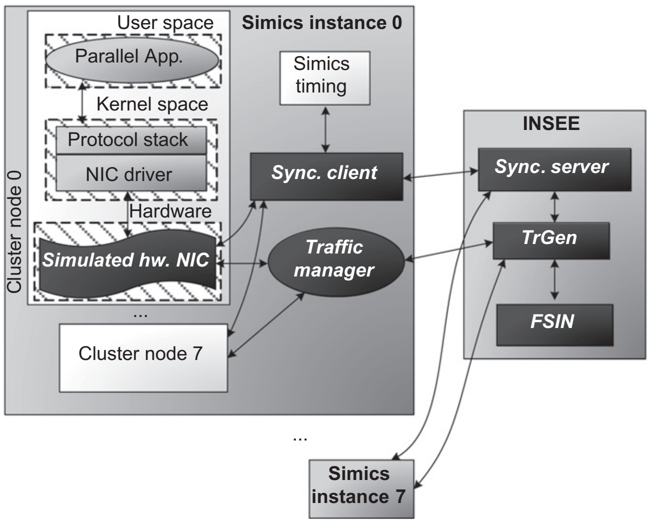

Fig. 11. Elements of our full-system simulation environment. Black elements belong to INSEE.

It is remarkable that Simics gives support to the synchronized execution of all the nodes within a given instance. Still, we need to add a synchronization client to keep synchronization among the different Simics instances, and also between them and FSIN. To this purpose INSEE is complemented with a synchronization server which keeps the whole simulation running at the same pace.

The interchange of application traffic is managed by TrGen, which stores meta-information of each message taking part in the simulation. TrGen is also in charge of injecting traffic into FSIN and of monitoring it to recognize when a message has arrived to its destination. Once a message arrives to its destination in FSIN, TrGen sends a confirmation to the source Simics instance in order to send the complete message to the destination instance and node. The traffic manager inside each Simics instance is in charge of storing the complete message and sending it to the destination node. When the Traffic manager receives a message to a simulated node it injects the message into the NIC module of the proper node.

Synchronization is fairly more complex as a two-level mechanism is implemented in order to synchronize the multiple elements taking part on the simulation. On the one hand, all the nodes simulated within a Simics instance are executed in a step-wise fashion: nodes go to execution sequentially for a given number of CPU cycles (*slice*) and, once the running one finishes its slice, the next node enters into execution for the same amount of simulated time. This mechanism is part of Simics. A synchronization client implemented inside each Simics instance is in charge of sending a message to TrGen to make it know that it has finished its slice; this is carried out when all the simulated nodes finish their corresponding slices. Once this message is sent, the synchronization client suspends the execution until a message from TrGen is received, allowing the execution of a new slice. The synchronization server, implemented within TrGen, waits until all Simics instances have finished their slices and then makes FSIN run for a period of time equivalent to the Simics slice. After FSIN finishes its slice, TrGen sends a multicast message to the Simics instances, allowing them to resume their execution. The ratio between Simics and FSIN slices (one measured in Simics cycles and the other one in FSIN cycles) determines the bandwidth of the simulated network.

Further details about performing execution-driven simulation within INSEE can be found in [41]. This work includes a thorough description of the full-system simulation in INSEE, and discusses several different approaches that may be followed in order to obtain usable results from this kind of simulation. Emphasis is put on where to capture the traffic in the simulated node, and also in synchronization, showing the need to fine tune the value of the slices in order to obtain a balance between simulation accuracy and execution time. Furthermore it discusses some issues encountered when setting up the simulation environment and carrying out performance-related experiments.

3.4. Task placement

An important consideration to take into account when launching parallel applications is the mapping of application tasks onto computing resources. Parallel applications are usually implemented following spatial distributions that can be exploited effectively by means of an adequate allocation onto the nodes. Furthermore, supercomputing sites are often used by many users that share machine resources, with several applications being executed simultaneously on different system partitions. Our previous research showed that a bad application placement may slow down the communication of applications sharing a parallel computing up to 10 times [28]. We also showed that an improvement of 10–15% in execution time of applications, obtained through a good placement policy, can compensate the high cost of a topology-aware scheduling [32].

When dealing with application (or application-inspired) traffic, INSEE has the ability of arranging the tasks of a workload onto the whole network, or onto network partitions, following different policies. It can also execute several application instances concurrently, to measure the effects of the interactions among them. Currently, INSEE supports only the simultaneous execution of several instances of the same application [28].

Some examples of the different placement strategies available for direct networks are depicted in Fig. 12. In the depictions, four applications, composed by four nodes each, share the network (these applications are represented by white, grey, black and crossed circles, correspondingly). Similarly, some example depictions of the different strategies for tree-like topologies are shown in Fig. 13. The definitions of the placement policies are as follows:

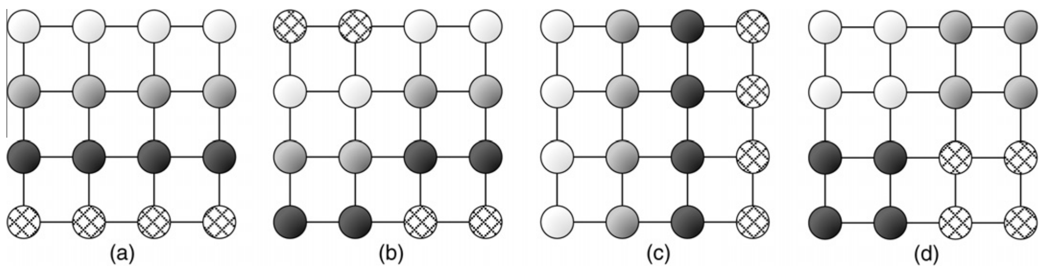

Fig. 12. Placement strategies for direct topologies: (a) row; (b) shift 2; (c) column and (d) quadrant.

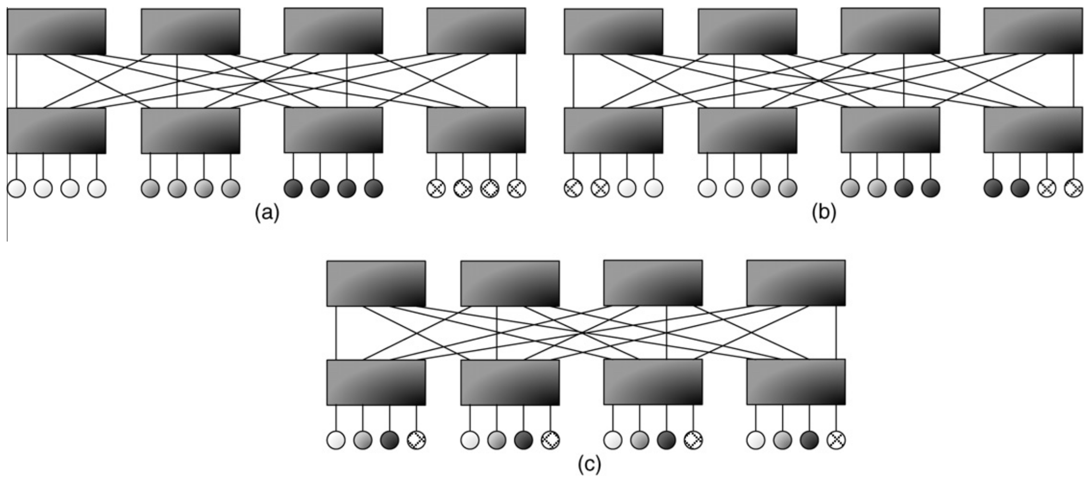

Fig. 13. Placement strategies for tree-like topologies: (a) consecutive; (b) shift 2 and (c) shuffle.

- *Consecutive/row*: Tasks and applications are placed consecutively—note that in the case of cube topologies this means filling rows in order. For example if we have an application of n nodes, it will be placed from nodes 0 to $n - 1$, being its tasks arranged in order. Examples of these placements are plotted in Fig. 12a for cubes and Fig. 13a for trees.
- *Shift*: This policy is like the previous one, but adding a given shift s to the allocated node, that is, task t is located in node $t + s$. Examples of these placements are plotted in Fig. 12b for cubes and Fig. 13b for trees.
- *Shuffle*: Tasks are placed in order in the nodes attached to the first port of each switch, then to those nodes attached to the second port of each switch, and so on. If in each switch there are p ports connected to compute nodes, then the tasks will be placed in nodes: $0, p, 2p, \ldots, 1, p + 1, 2p + 1$, and so on. An example of this placement is plotted in Fig. 13c. Note that this policy only makes sense in those topologies with many compute nodes attached to each network element, i.e. indirect topologies.
- *Column*: This policy only makes sense in cube-like topologies with, at least, two dimensions. Assignment is done selecting the nodes by columns, which can be seen as partitioning the network in rectangular sub-networks, taller than wide. An example of this placement is plotted in Fig. 12c.
- *Quadrant*: This policy only makes sense in cube-like topologies with, at least, two dimensions. When using several application instances, we can partition the network in perfect squares (or cubes). Within each square an application is placed following consecutive order. An example of this placement is plotted in Fig. 12d.
- *Random*: All the tasks are randomly placed along the network independently of their application. To do so, we generate a random permutation of the network nodes.
- *File*: INSEE can read the mapping information from a file. The format of this file is very simple: $\langle$*node*, *task*, *application*$\rangle$, meaning that the task *task* from the application *application* is mapped onto node *node*. Note that this mode allows us to test more complex mapping techniques, such as optimized task-to-node placement [31].

4. Limitations of INSEE

Although INSEE has proven to be a useful tool to evaluate interconnection systems, it has some limitations that we should not forget. Some of these limitations are easily solvable, but may require considerable effort in terms of implementation time. Additionally, some modifications may increase excessively memory requirements and/or execution times, something that would go against the philosophy of INSEE. Still, in this section we discuss some modifications that could increase the usefulness of this environment.

FSIN uses internally a time-driven engine that it is not suitable to simulate workloads with long computation times between communication events, because it wastes too much time doing nothing but making the clock advance. We are currently developing an event-driven engine for FSIN which would accelerate the simulation with low density of events.

The node model, simulated as a simple traffic generator and consumer without any internal structure, is too simplistic. We designed it this way because we are interested in the behavior of the IN, and this model is sufficient for this purpose. Another limitation closely related to this one is that INSEE only allows allocating a single task per node. This is not very realistic because, in current systems, nodes attached to the network are actually multiprocessors. The simulation of multiprocessor nodes can be implemented in TrGen just by allowing several application tasks (from Simics, traces or application kernels) to share a FSIN node. However it will require making some implementation decisions: how to arbitrate the injection infrastructure and how to perform intra-node communication. The added complexity would result in slower simulation times and larger footprints.

When performing trace-driven simulations, the causality of the messages is maintained, but the MPI semantics are not followed accurately by TrGen. For instance, all messages sends are treated equally, regardless of them being immediate, *rendevouz*, one-sided, etc. Implementing MPI semantics accurately would require increasing the complexity of the node which, as stated before, is not desirable. Furthermore, MPI has very complex (and precise) semantics, and implementing every operation supported by the standard would require huge efforts in terms of fully understanding the involved details and coding them within TrGen.

A remarkable limitation of INSEE is that FSIN does not include detailed models of some *state-of-the-art* networking technologies such as InfiniBand [45] or Myrinet [8]. The reason for this is two-folded. On the one hand, implementing these technologies would require a deep knowledge of every detail of them, as well as a non-negligible effort in terms of coding. On the other hand, given the complexity of these technologies, it would not be possible to perform large-scale simulations because of the associated requirements in terms of computing resources.

An arguably less significant limitation is that INSEE does not allow for parallel or distributed simulation. Although FSIN could be easily extended in this line—note that its time-driven engine is split into two separate loops and, in each loop, iterations are completely independent among them—this has not been done for two simple reasons. The first one is that the low footprint of INSEE allows for large-scale simulations in a single off-the-shelf computer, and therefore we did not *need* to parallelize it, even when we have parallel computers. This leads to the second reason: in our evaluations we usually require the execution of *many* experiments (hundreds, even thousands), modifying parameters such as network size, workload, placement and random seed, without any dependency between the different runs. The utilization of a cluster as a high-throughput computing resource is enough for this purpose.

5. Performance evaluation of INSEE

As stated before, INSEE is a lightweight tool that has been developed with resource scarcity in mind. In this section we show how the resource requirements scale with the size of the system to simulate, using a collection of realistic experiments in which we vary those parameters that affect memory and CPU usage. The values plotted in the figures correspond to the average value of 10 runs. Confidence intervals (99%) are three orders of magnitude below the average and therefore are not plotted.

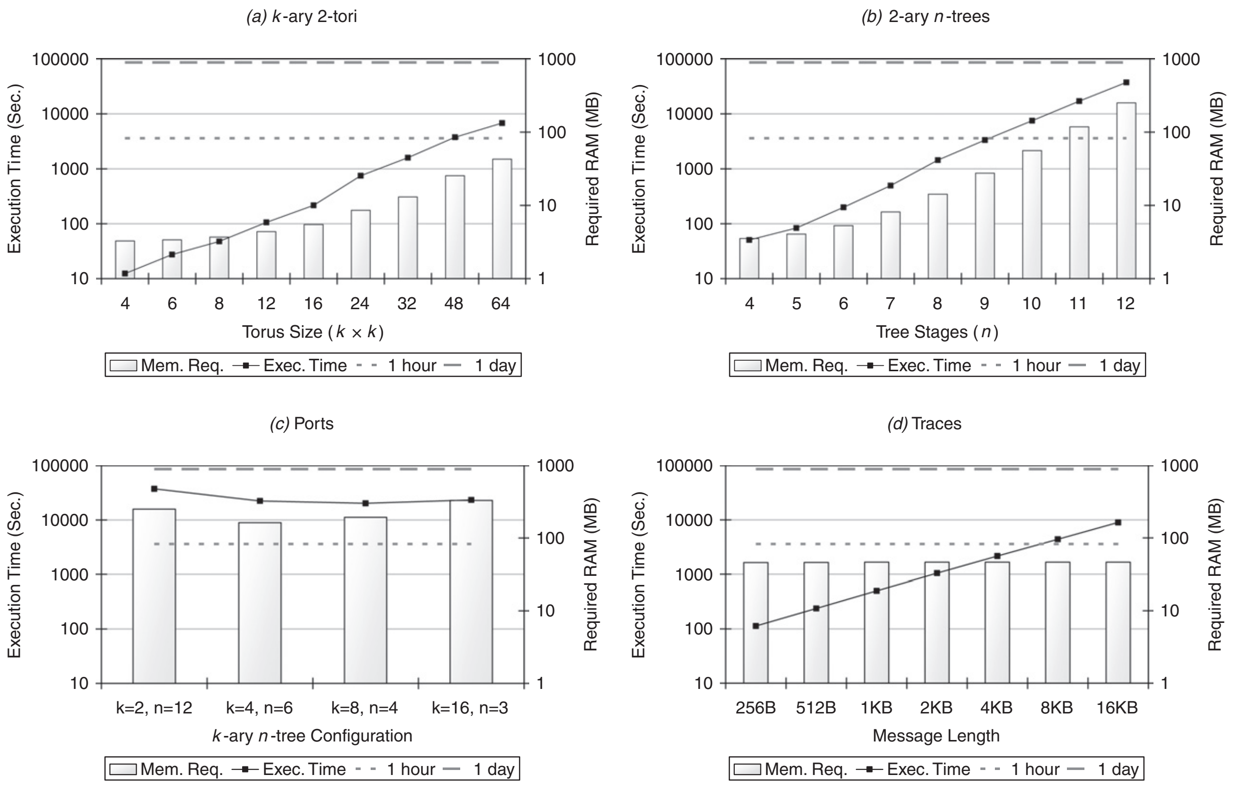

Fig. 14. Resources needed to perform simulation.

The simulation runs used in this section use parameter values that are commonly used in our day-to-day research. Tori use 2D bubble routers using the smart request strategy, while trees use the adaptive routing strategy. All physical channels are split into two virtual channels, each of them with a queue with room for storing four packets of 16 phits (physical units – 4 bytes). Fig. 14 shows plots of simulation time and memory footprint obtained from a series of simulation runs performed in an Intel(R) Pentium(R) 4 CPU working at 3.00 GHz with 1.5 GB of RAM. Note that the instrumentation introduced to obtain these measurements has a negative effect on them, and also that executions on more recent processors would be significantly faster.

In the plots both *X* and *Y* axis show logarithmic values. Two dotted lines representing respectively an hour and a day are added as hints to better understand the reported times.

A first study whose results are depicted in Fig. 14a, shows the scalability of INSEE when simulating tori. Systems arranged in a wide variety of sizes, from 4×4 (16 nodes) to 64×64 (4096 nodes), are studied,with a workload modeled as uniform traffic from independent sources at the maximum possible pace. We can see how execution time and memory consumption scale roughly linearly with the number of nodes. This is because the main memory consumption is due to the room in the queues, and the number of queues depends on the number of nodes.

In the case of the trees, depicted in Fig. 14b, we simulated 2-ary *k*-trees with the *k* parameter varying from $k = 4$ (16 nodes) to $k = 12$ (4096). The workload was also uniform traffic from independent sources at full pace. The number of switches scales in $O\ (n \log n)$ with the number of nodes and, therefore, so do memory consumption and execution time, as can be seen in the figure.

In Fig. 14c, four different fat-tree topologies, all able to connect 4096 nodes, are fed with uniform traffic from independent sources. We can see how the choice of topology affects noticeably the resources required to perform a simulation. In this case, both memory usage and execution time can increase up to a 400% just by using one topology or another. In the plot we can clearly seen that the sweet spot seems to be the 4-ary 6-tree.

Finally, in Fig. 14d, we generated several instances of butterfly, an application inspired workload, all of them for 4096 nodes but varying the message length from 256 bytes (1 packet) to 16 kB (64 packets). We run these workloads in a 2D torus with 64 nodes per dimension. In the plot we can see how the message size affects linearly the execution time, but the memory does not change from configuration to configuration. This is because the size of the data structures used to store messages in trace-driven simulation do not depend on the length of the (simulated) messages. However, as the simulation operates at the phit level, longer messages translate on more phits managed by the network, and this requires more time.

In general, we can conclude that INSEE's memory requirements grow roughly linearly with the number of simulated ports, which in turn depends on the number of switches and their complexity. Execution time grows with the number of ports too, but also strongly depends on the characteristics of the simulated workload. At any rate, authors want to highlight that INSEE's footprint and execution speed in current computers (more up-to-date than the one used in this evaluation) are austere: most of our experiments are completed in a few hours, using only a few hundred MB of RAM. In other words, INSEE does not have prohibitive requirements.

6. Related work

We want to remark that INSEE is only one of the network simulators available to the community. In the literature we can find references for many of them. Let us review a small selection, pinpointing the main differences with INSEE. Note that, as explained before, the main characteristics of INSEE are its flexibility and its low resources requirement, and therefore it outperforms in these two aspects to most of the tools revised here.

SICOSYS [36,48], developed at the University of Cantabria, has very detailed models of several router architectures, which allows obtaining very accurate performance measurements, close to those obtained with a hardware-level simulator, but at a fraction of the required computing resources. It is noticeably more complex and resource-demanding than INSEE and, therefore, it is restricted to simulate networks composed by a few hundred nodes at most. SICOSYS is implemented in C++, and has been used for evaluations focusing on performance and on fault-tolerance of networks-on-chip. In addition, it allows feeding the simulator with several traffic models with different levels of detail: synthetic traffic and traces of MPI or ccNUMA applications. It can also interface with RSIM [30] and SIMOS [43] to perform full-system simulation of INs.

The Chaos router simulator [51], from the University of Washington, has detailed models of *k*-ary *n*-cube networks (meshes, tori and hypercube); tree-like topologies are not implemented. Switching can be both packet-switching and wormhole. Routers can implement dimension-order oblivious routing or, alternatively, Chaotic adaptive routing [18]. It also incorporates a wide variety of synthetic traffic patterns to feed the simulation. A distinguishing characteristic of this simulator is that it may also run in animated mode, in which the temporal evolution of the simulation is graphically represented. One of the main drawbacks of this simulator is that all the changes in the design of the network to evaluate must be done at compilation time, in other words, every change in the model requires the application re-compilation. It is remarkable that, as far as we know, development of this simulator was stopped in 1996, but its source code is still available.

FlexSim [49], developed at the University of Southern California, is a C-based simulator for *k*-ary *n*-cube networks with any number of dimensions greater or equal than 2, any power of two number of nodes per dimension, and any arbitrary number of virtual channels. It has support for several router models, synthetic traffic patterns and failure models. This simulator has been used to conduct research in fault-tolerance and deadlock-free routing.

MARS [12] is a simulator of parallel systems developed at IBM and based on the OMNeT++ simulation framework [52]. Its design is oriented to the evaluation of parallel systems and parallel applications, and to that purpose it includes detailed models of both the communication side and the compute nodes. MARS allows us to use several multi-stage topologies, and a variety of switching and routing functions. It supports multi-core configurations in which each processing core has its own MPI stack. The main difference between MARS and INSEE trace-driven simulations is the model of the node, and the conformity to MPI semantics; these differences make INSEE faster, while MARS is *a priori* more accurate.

MINSimulate [47], developed at the Technical University of Berlin, is a simulator designed to evaluate multi-stage INs. It implements Clos and Delta networks and supports both wormhole and store and forward switching. Note that currently INSEE does not support this kind of networks but their inclusion would require insignificant efforts as an implementation of a multi-stage switch is already available.

The NS-2 simulator [15], from the University of Southern California, is designed to research on wired and wireless TCP-based communication networks. Although high-performance computing systems use to rely on high performance interconnects such as InfiniBand [45] or Myrinet [8] for parallel computing, most of them support Ethernet-based networks for storage and control purposes. Furthermore, new versions of Ethernet 10 GB or the early-to-come 100 GB Ethernet can be used as low-cost INs. For these reasons we include NS-2 in this review.

BigNetSim [53], developed at the University of Illinois at Urbana Champaign, is a trace-driven parallel discrete event simulator. It simulates, with reasonable detail, an integrated model for computation (processors) and communication (network). The simulator allows different levels of detail to evaluate the IN: from simple latency models to detailed models of the network including k-ary n-cubes and k-ary n-trees. One of the main advantages of this system is its extreme modularity, with easy mechanisms to model new topologies and routing algorithms. BigNetSim has a parallel implementation that allows carrying out large simulations of current and future systems, and to study the behavior of applications developed for those systems. In contrast with INSEE, in which system configuration is given as parameters at execution time, BigNetSim is configured at compilation time, in such a way that any change in the models require to re-compile the target modules.

Dimemas [6], developed and maintained at the Barcelona Supercomputing Center, was designed with the evaluation of applications behavior in mind. It can reconstruct the execution of a parallel application in any supported architecture using a trace of that application. Dimemas philosophy is similar to INSEE: keep it simple. Dimemas models computing elements with accuracy but models the INs in a rather simplistic way: a collection of buses. The workloads used with Dimemas are modeled in detail, with lots of significant states available for each application thread. A drawback of this workload's complexity is that obtaining traces with sufficient level of detail requires an instrumented kernel. Dimemas is designed to search for bottlenecks and/or unbalances that may harm the performance of parallel applications. Note that while INSEE can be used for this purpose, it is not specifically designed for this function.

The COTSon Infrastructure for system-level simulation by HP Labs [4] provides a simulation environment very similar to that given by the combination FSIN+ Simics, but based on AMD's SimNow [3]. The tool has been open sourced in January 2010. It is able to simulate multi and many-core machines, and also clusters using a functional simulator of a network switch. We plan to further study the potential of its pluggable architecture, in order to check if FSIN could be integrated into this infrastructure.

We close this section with Table 3, in which the differences among the discussed simulators are summarized. Note that the NS-2 is not included because of its completely different nature.

7. Final remarks and future work

In this paper, we have thoroughly described INSEE, our Interconnection Network Simulation and Evaluation Environment [50]. This description includes all the router architectures and network organizations currently implemented in FSIN, our Functional Simulator of Interconnection Networks, as well as all the accepted router parameters and captured statistics. Furthermore, all the traffic models and procedures to generate workloads provided by TrGen, our traffic generator module, are

Table 3
Summary of the characteristics of the reviewed simulators.

	Family of topologies			Level of detail		Traffic models			Resources requirement
	Cube	Tree-like	Multi-stage	Network	Nodes	Synthetic	Traces	Full-system	
INSEE	√	√	×	High	Low	√	√	√	Low
SICOSYS	√	×	×	High	Low	√	√	√	High
Chaos	√	×	×	High	Low	√	×	×	Medium
Flexsim	√	×	×	High	Low	√	×	×	Medium
Mars	×	√	√	High	High	×	√	×	High
MINSimulate	×	×	√	High	Low	√	×	×	Medium
COTSon	√	√	√	Low	Very high	×	×	√	Very High
BigNetSim	√	√	√	Variable	Variable	×	√	×	Variable
Dimemas	√	√	√	Low	High	×	√	×	Low

explained in detail. The workloads can be synthetic, with different degrees of fidelity to actual applications, or application-based using traces and full-system simulation.

INSEE tools have been successfully used in our research group to carry out a wide variety of studies in the field of performance evaluation of INs, including: the effect of Head-of-Line blocking at injection [17], the impact in the performance of several injection interfaces as congestion control mechanisms [16], the performance of local congestion control mechanisms [21,42], the evaluation of several topological proposals [9,24,26], and the effect of task and node allocation on the performance of parallel applications [28,31], among others. A remarkable feature of INSEE is that it allows simulating large-scale networks of up to 64 K nodes in a regular off-the-shelf desktop computer with 2 GB of RAM.

We plan to add new features and operation modes to the environment. Some of them will be driven by the requirements of our research work. Some others are oriented to improve the behavior of the environment. These new additions include, but are not limited to the following.

- A new, event-driven simulation engine for FSIN is under development, in order to allow faster simulation of applications while properly modeling the network. This implementation will be further extended to be used in the field of job scheduling, allowing running together several applications as well as their management. Together with the integration of this new engine, we plan to add the capability to simulate several processing units in each compute node in order to properly simulate current clusters and supercomputers.
- We plan to model new router architectures. For example, a more detailed model of the SpiNNaker router will be added to perform a more realistic evaluation using neural-activity workloads for this target system. Other router architectures than can be added to INSEE are the HPAR router [37] or the rotary router [1]. Other subsystems to be modeled are the Immunet [38] and ImmuCube [35] fault-tolerance schemes.
- We will continue with our work on characterizing realistic workloads, in order to increase our collection of application kernels. These kernels will be used in future evaluation studies in which realistic traffic models are required to provide accurate results. One interesting starting point is the review of typical high-performance computing applications that can be found in [5], in which 13 *dwarves* are identified. Each of these dwarves represents a prototype of application because of the communication patterns, the coupling of the task or several other details.

Acknowledgements

This work has been supported by the Ministry of Education and Science (Spain), Grant TIN2007-68023-C02-02, and by Grant IT-242-07 from the Basque Government. Dr. Navaridas is supported by a post-doctoral grant of The University of the Basque Country. Mr. Pascual is supported by a doctoral grant of the Basque Government.

References

[1] P. Abad, V. Puente, P. Prieto, J.A. Gregorio, Rotary router: an efficient architecture for CMP interconnection networks, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, San Diego, CA, USA, June, 2007, pp. 116–125. doi: 10.1145/1250662.1250678.
[2] N.R. Adiga, M.A. Blumrich, D. Chen, P. Coteus, A. Gara, M.E. Giampapa, P. Heidelberger, S. Singh, B.D. Steinmacher-Burow, T. Takken, M. Tsao, P. Vranas, Blue Gene/L torus interconnection network, IBM Journal of Research and Development 49 (2/3) (2005).
[3] Advanced Micro Devices Inc., AMD SimNow Simulator. <http://developer.amd.com/cpu/simnow>.
[4] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, D. Ortega, COTSon: infrastructure for full system simulation, ACM SIGOPS Operating Systems Review 43 (1) (2009) 52–61.
[5] K. Asanovic, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, EECS Department, University of California, Berkeley, Technical Report No. UCB/EECS-2006-183, December 18, 2006. <http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf>.
[6] R.M. Badia, J. Labarta, J. Gimenez, F. Escale, DIMEMAS: Predicting MPI applications behavior in Grid environments, Workshop on Grid Applications and Programming Tools, June, 2003.
[7] R. Beivide, E. Herrada, J.L. Balcazar, A. Arruabarrena, Optimal distance networks of low degree for parallel computers, IEEE Transactions on Computers 40 (10) (1991) 1109–1124, doi:10.1109/12.93744.
[8] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.K. Su, Myrinet: a gigabit-per-second local area network, IEEE Micro 15 (1) (1995) 29–36, doi:10.1109/40.342015.
[9] J.M. Camara, M. Moreto, E. Vallejo, R. Beivide, J. Miguel-Alonso, C. Martinez, J. Navaridas, Twisted torus topologies for enhanced interconnection networks, Accepted for Publication in the IEEE Transactions on Parallel and Distributed Systems, in press. doi: 10.1109/TPDS.2010.30.
[10] W.J. Dally, C.L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, IEEE Transactions on Computers 36 (5) (1987) 547–553, doi:10.1109/TC.1987.1676939.
[11] W.J. Dally, B. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann Series in Computer Architecture and Design, 2004. ISBN: 0-12-200751-4.
[12] W.E. Denzel, J. Li, P. Walker, Y. Jin, A Framework for End-to-End Simulation of High Performance Computing Systems, SIMUTools'08, Marseille, France, March 3–7, 2008.
[13] J. Duato, A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks, IEEE Transactions on Parallel and Distributed Systems 6 (10) (1995) 1055–1067, doi:10.1109/71.473515.
[14] S. Goldschmidt, J. Hennessy, The accuracy of trace-driven simulation of multiprocessors, in: ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems, May, 1993, pp. 146–157. doi: 10.1145/166962.167001.
[15] Information Science Institute, Network Simulator ns-2. <http://www.isi.edu/nsnam/ns/>.
[16] C. Izu, J. Miguel-Alonso, J.A. Gregorio, Evaluation of interconnection network performance under heavy nonuniform loads, in: Lecture Notes in Computer Science, Proc. ICA3PP, vol. 3719/2005, 2005, pp. 396–405.
[17] C. Izu, J. Miguel-Alonso, J.A. Gregorio, Effects of injection pressure on network throughput, in: Proc. PDP 2006 14th Euromicro Conference on Parallel, Distributed and Network Based Processing, Montbéliard-Sochaux, France, February 15–17, 2006. doi: 10.1109/PDP.2006.32.
[18] S. Konstantinidou, L. Snyder, The Chaos router, IEEE Transactions on Computers 43 (12) (1994) 1386–1397, doi:10.1109/12.338098.

[19] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, Simics: a full system simulation platform, IEEE Computer 35 (2) (2002) 50–58, doi:10.1109/2.982916.
[20] Message Passing Interface Forum, MPI: A Message-Passing Interface Standard. <http://www-unix.mcs.anl.gov/mpi/standard.html>.
[21] J. Miguel-Alonso, C. Izu, J.A. Gregorio, Improving the performance of large interconnection networks using congestion-control mechanisms, Performance Evaluation 65 (3) (2008) 203–211, doi:10.1016/j.peva.2007.05.001.
[22] J. Miguel-Alonso, J. Navaridas, F.J. Ridruejo, Interconnection network simulation using traces of MPI applications, International Journal of Parallel Programming 37 (2) (2009) 153–174, doi:10.1007/s10766-008-0089-y.
[23] M. Mirza-Aghatabar, S. Koohi, S. Hessabi, M. Pedram, An empirical investigation of mesh and torus NoC topologies under different routing algorithms and traffic models, in: Proceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools, Lübeck, Germany, August 29–31, 2007, pp. 19–26. doi: 10.1109/DSD.2007.28.
[24] J. Navaridas, M. Luján, J. Miguel-Alonso, L.A. Plana, S.B. Furber, Understanding the interconnection network of SpiNNaker, in: proceedings of the 23rd International Conference on Supercomputing, Yorktown Heights, NY, USA, June 8–12, 2009, pp. 286–295. doi: 10.1145/1542275.1542317.
[25] J. Navaridas, J. Miguel-Alonso, Realistic evaluation of interconnection networks using synthetic traffic, in: 8th International Symposium on Parallel and Distributed Computing, Lisbon, Portugal, June 30–July 4, 2009.
[26] J. Navaridas, J. Miguel-Alonso, F.J. Ridruejo, W. Denzel, Reducing complexity in tree-like computer interconnection networks, Accepted for Publication in the International Journal on Parallel Computing, in press. doi: 10.1016/j.parco.2009.12.004.
[27] J. Navaridas, J. Miguel-Alonso, F.J. Ridruejo, On synthesizing workloads emulating MPI applications, in: The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing, Miami, Florida, USA, April 14–18, 2008. doi: 10.1109/IPDPS.2008.4536473.
[28] J. Navaridas, J.A. Pascual, J. Miguel-Alonso, Effects of job and task placement on parallel scientific applications performance, in: Proc 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. Weimar, Germany, February 18–20, 2009, pp. 55–61. doi: 10.1109/PDP.2009.53.
[29] J. Navaridas, L.A. Plana, J. Miguel-Alonso, M. Luján, S.B. Furber, SpiNNaker: effects of traffic locality and causality on the performance of the interconnection network, Submitted to the ACM International Conference on Computing Frontiers, 2010.
[30] V.S. Pai, P. Ranganathan, S.V. Adve, RSIM: an execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors, in: IEEE Technical Committee on Computer Architecture Newsletter, October, 1997.
[31] Jose A. Pascual, Jose Miguel-Alonso, Jose A. Lozano, Optimization-Based Mapping Framework for Parallel Applications. Technical Report EHU-KAT-IK-02-10, University of the Basque Country, April, 2010. Submitted to Elsevier's Journal of Parallel and Distributed Computing.
[32] J.A. Pascual, J. Navaridas, J. Miguel-Alonso, Effects of topology-aware allocation policies on scheduling performance, in: Proc. 4th Workshop on Job Scheduling Strategies for Parallel Processing in Conjunction with IPDPS 2009, Rome, Italy, Lecture Notes in Computer Sciences, vol. 5798/2009, May 29, 2009, pp. 138–156. doi: 10.1007/978-3-642-04633-9_8.
[33] F. Petrini, M. Vanneschi, *k*-ary *n*-trees: high performance networks for massively parallel architectures, in: Proceedings of the 11th International Parallel Processing Symposium, Geneva, Switzerland, 1–5 April, 1997, pp. 87–93. doi: 10.1109/IPPS.1997.580853.
[34] L.A. Plana, S.B. Furber, S. Temple, M.M. Khan, Y. Shi, J. Wu, S. Yang, A GALS infrastructure for a massively parallel multiprocessor, IEEE Design and Test of Computers 24 (5) (2007) 454–463, doi:10.1109/MDT.2007.149.
[35] V. Puente, J.A. Gregorio, Immucube: scalable fault-tolerant routing for *k*-ary *n*-cube networks, IEEE Transactions on Parallel and Distributed Systems 18 (6) (2007) 776–788, doi:10.1109/TPDS.2007.1047.
[36] V. Puente, J.A. Gregório, R. Beivide, SICOSYS: an integrated framework for studying interconnection network performance in multiprocessor systems, in: Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing, Canary Islands, Spain, January 9–11, 2002, pp. 15–22. doi: 10.1109/EMPDP.2002.994207.
[37] V. Puente, J.A. Gregorio, R. Beivide, C. Izu, On the design of a high-performance adaptive router for CC-NUMA multiprocessors, IEEE Transactions on Parallel and Distributed Systems 14 (5) (2003), doi:10.1109/TPDS.2003.1199066.
[38] V. Puente, J.A. Gregorio, F. Vallejo, R. Beivide, Immunet: dependable routing for interconnection networks with arbitrary topology, IEEE Transactions on Computers 57 (12) (2008) 1676–1689, doi:10.1109/TC.2008.95.
[39] V. Puente, C. Izu, R. Beivide, J.A. Gregorio, F. Vallejo, J.M. Prellezo, The adaptive bubble router, Journal of Parallel and Distributed Computing 61 (9) (2001) 1180–1208, doi:10.1006/jpdc.2001.1746.
[40] F.J. Ridruejo, J. Miguel-Alonso, INSEE: an interconnection network simulation and evaluation environment, in: Lecture Notes in Computer Science, Proc. Euro-Par, vol. 3648/2005, 2005, pp. 1014–1023.
[41] F.J. Ridruejo, J. Miguel-Alonso, J. Navaridas, Full-System Simulation of Distributed Memory Multicomputers, Cluster Computing, Published Online, March 28, 2009. doi: 10.1007/s10586-009-0086-y.
[42] F.J. Ridruejo, J. Navaridas, J. Miguel-Alonso, C. Izu, Realistic evaluation of interconnection network performance at high loads, in: 8th International Conference on Parallel and Distributed Computing Applications and Technologies, Adelaide, Australia, December 3–6, 2007, pp. 97–104. doi: 10.1109/PDCAT.2007.73.
[43] M. Rosenblum, S.A. Herrod, E. Witchel, A. Gupta, Complete computer system simulation: the SimOS approach, Parallel and Distributed Technology: Systems and Applications 3 (4) (1995) 34–43, doi:10.1109/88.473612.
[44] M.D. Schroeder, A.D. Birrell, M. Burrows, H. Murray, R.M. Needham, T.L. Rodeheffer, E.H. Satterthwaite, C.P. Thacker, Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-point Links, SRC Research Report 59, December, April 21, 1990.
[45] T. Shanley, InfiniBand Network Architecture. Addison-Wesley, 2002 (November). ISBN: 978-0-321-11765-6.
[46] B. Towles, W.J. Dally, Worst-case traffic for oblivious routing functions, IEEE Computer Architecture Letters 1 (1) (2002), doi:10.1109/L-CA.2002.12.
[47] D. Tutsch, M. Brenner, D. Luedtke, A. Walter, MINSimulate. <http://dontcry.pdv.cs.tu-berlin.de/minsimulate/index.html>.
[48] University of Cantabria, SICOSYS. <http://www.atc.unican.es/SICOSYS/>.
[49] University of Southern California, Information on FlexSim1.2. <http://ceng.usc.edu/smart/FlexSim/flexsim.html>.
[50] The University of the Basque Country, INSEE. <http://insee.sourceforge.net/>.
[51] University of Washington, The Chaos Router Simulator. <http://www.cs.washington.edu/research/projects/lis/chaos/www/simulator.html>.
[52] A. Vargas, The OMNeT++ Discrete Event Simulation System. <http://www.omnetpp.org/download/docs/papers/esm2001-meth48.pdf>.
[53] G. Zheng, T. Wilmarth, P. Jagadishprasad, L.V. Kalé, Simulation-based performance prediction for large parallel machines, International Journal of Parallel Programming 33 (2–3) (2005), doi:10.1007/s10766-005-3582-6.