Full-System Simulation of Distributed Memory Multicomputers

Fco. Javier Ridruejo Jose Miguel-Alonso Javier Navaridas
Dep. of Computer Architecture and Technology, The University of the Basque Country
P. Manuel de Lardizabal, 1 (20018) Donostia-San Sebastian, Spain
franciscojavier.ridruejo@ehu.es j.miguel@ehu.es javier.navaridas@ehu.es

Technical Report EHU-KAT-IK-01-09

Abstract

In this paper we discuss environments for the full-system simulation of multicomputers. These environments are composed of a large collection of modules that simulate the compute nodes and the network, plus additional linking elements that perform communication and synchronization. We present our own environment, in which we integrate Simics with INSEE. We reuse as many Simics modules as possible to reduce the effort of hardware modeling, and also to simulate standard machines running unmodified operating systems. This way we avoid the error-prone effort of developing drivers and libraries. The environment we propose in this paper enables us to show some of the difficulties we found when integrating diverse tools, and how we were able to overcome them. Furthermore we show some important details to have into account in order to do a valid full-system simulation of multicomputers, mostly related with synchronization and timing. Thus, a trade-off has to be found between simulation speed and accuracy of results.

1. Introduction

Supercomputing is a very valuable resource which is continuously growing in importance in business and science. Current scientific studies rely on analysis and modeling of different natural phenomena that require a huge amount of computing power, which is not attainable using regular off-the-shelf computers. For example, physicists, chemists or pharmaceutical researchers simulate, for different purposes, interactions between large numbers of molecules. Likewise, in the business context, corporations demand large amounts of computing power in order to use data mining software over their huge data bases, 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, in both contexts, the required computing power is only limited by the available resources. In general, if available computing resources are doubled the number of runs, the grain-size (whichever this means in the particular application context), the size of the datasets or whatever other parameter that affects execution time will be increased in order to fully use the new resources. In other words, the magnitude of the experiments is scaled up to the available resources. This means that there is a permanent demand of high-performance computers 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, access elements, operating system, high performance libraries, parallel applications, etc. Depending on budget and availability, elements can be designed from scratch; often, however, off-the-shelf components are reused, either directly or modified to fulfill tasks different to those they were designed for.

During the preliminary phases of the design of a supercomputer, elements are tested and evaluated separately, in order to assess (and, if possible, improve) their performance. These evaluations are carried out using synthetic loads, based on statistical distributions, which allow for fast simulations, but may not be truly representative of actual workloads. Simulation speed is important in this phase, in order to be able to explore a wide range of options to help in the decision-making process, choosing the more promising alternatives.

In subsequent design phases the simulated model grows in complexity, and more realistic evaluations are performed, mixing a complex model of the component under evaluation with simpler models of the rest of the system, usually working with traces obtained in actual machines. Traces are more realistic than synthetic workloads, but may comprise some characteristics of the system in which they were taken that are not valid in the system under evaluation [18].

Once system components have been chosen, a validation of the whole design is required to confirm that the behavior is the expected one, and to check that there are no undesirable interactions between components that may have passed unnoticed before, due to the

simplification of models and simulations. This validation is usually done with full-system simulators made from scratch or, in most cases, using different simulators for each component of the system, and doing some linking work to put them to operate together.

Our interest is mainly focused on interconnection networks for distributed memory parallel systems, a kind of specific-purpose network that allows compute nodes to interchange messages with high throughput and low latency – which is required to run efficiently parallel applications. In the rest of this paper we will use "IN" as the acronym for interconnection network, and multicomputer as a shorter way of naming distributed memory parallel computers.

We will describe the components that take part in a full-system simulation of a multicomputer, making a proposal that mixes two very different tools: Interconnection Network Simulation and Evaluation Environment [32] (INSEE for short) in charge of the IN, and Simics [14], used to simulate the compute elements. We will also discuss several approaches to interface these two classes of simulators, and problems that may arise when doing full-system simulation, some due to reutilization of components of the simulated hosts, and some due to unexpected interactions between components. Moreover, we will explain the complexity of fine-tuning all components and their interfaces, in order to find a trade-off between simulation accuracy and resource usage (simulation time).

The rest of the paper is organized as follows. In section 2 we review related work. Section 3 explains the elements that can take part in a full-system simulation of a multicomputer, as well as some approaches to glue together an IN simulator with the simulators of the compute nodes. Section 4 discusses how to synchronize these simulators. Section 5 introduces our proposal for full-system simulation of multicomputers, combining INSEE with Simics. In order to show the capabilities of this tool, a set of experiments related with congestion control are defined in Section 6. Results are shown and analyzed in Section 7. Finally, in section 8 we enumerate the conclusions of this work.

2. Related work

There are other research groups around the world interested in full-system simulation. However most of them are only interested in either the performance evaluation of workloads on

servers, or in the assessment of a particular micro-architectural improvement. We can enumerate several full-system simulators specifically designed to perform these kinds of studies: RSIM [26], ML-RSIM [34], SimOS [33], Simics [14], M5 [2], SIMFLEX [37], GEMS [15], and Mambo [7]. A longer list is available in [16].

Models used for the IN are often too simplistic. These tools implement networking systems based on Ethernet, which is valid for most of the usual performance evaluations of server systems that run OLTP (On-Line Transaction Processing) workloads. As far as we know, none of them implement a sophisticated IN such as those used in high-performance clusters or massively parallel processors.

It would be possible to integrate more complex IN models into available full-system simulation tools; however, IN simulators already exist (SICOSYS [28], the Chaos Router Simulator [5], FlexSim [35], OPNET [25], NS [24] and many others), and it would be easier for designers to integrate available tools, instead of starting from scratch with, for example, a collection of Simics modules for an IN. For instance, SICOSYS can interface with RSIM to do full-system simulation of shared-memory parallel computers, providing an accurate timing model, as discussed in [30]. However, this setup consumes a huge amount of resources (memory, CPU time) and only allows for the simulation of a few tens of interconnected compute nodes.

Simics memory timing mechanism is too simplistic: all instructions and memory accesses take the same amount of time. Consequently several tools have been designed that extend Simics functionality to perform detailed memory simulation. Two of these are GEMS [15] and SIMFLEX [37]. GEMS was designed to study a wide variety of memory hierarchies and systems ranging from broadcast-based SMPs to hierarchical directory-based Multiple-CMP systems. It can model current high-end, non-uniform memory access (NUMA) multiprocessor systems which have their memory controllers connected via an interconnection network, including directory-based and snooping-based systems. It has not been designed to provide support to large simulation of multicomputers with their compute nodes connected through another kind of IN.

SIMFLEX uses statistical sampling to accelerate simulations. It estimates the performance of an application by measuring several samples of it. Those samples are carefully chosen using established statistical sampling methods from a library of application checkpoints that represent the behavior of the whole application with a high confidence level, depending on the number of

samples being executed. It also requires finding out the warming period previous to the measurement of each sample. SIMFLEX can model chip multiprocessor and distributed-shared-memory multiprocessor systems. However, it does not save the state or keep account of the queues in the interconnection network, it runs the example more times to warm up the system to allow the queues to fill into a steady state. That approach could alleviate a congestion state in the interconnection network, hiding or attenuating effects of congestion. Moreover, there may be different execution paths when trying with different topologies and other parameters. A big effort has to be done to integrate an interconnection network simulator and SIMFLEX, and to prepare every library of samples for each benchmark.

The M5 simulator is specifically designed to conduct research in TCP/IP networking and provides features necessary for simulating networked hosts, including full-system capability, a detailed I/O subsystem, bus timing, coherence effects of network DMA transfers [3] and the ability to simulate multiple networked systems deterministically in a single instance of the M5 simulator. M5 connects network interface cards of individual systems using an Ethernet link object, modeled as a lossless, full-duplex channel of configurable bandwidth and delay [2], a very simple approach to do IN simulation. New components can be developed and incorporated into the M5 structure.

As we can see, most of these tools are heavyweight, and focus either on LAN technologies, or on shared-memory machines. Our interest is on distributed-memory parallel machines, and we already have a simulation environment to evaluate proposals for the INs used in this kind of architectures: INSEE [32]. This environment, depicted in Figure 1, includes FSIN, a time-driven, lightweight, flexible, and functional IN simulator that allows us to simulate distributed memory parallel computers with thousands of nodes arranged in different topologies (k-ary n-cubes, multistage networks). The TrGen module allows synthetic traffic generation, trace based generation and, as we will describe later, full-system simulation. The integration of INSEE with Simics, the tool of choice to provide full-system simulation of compute elements, provides a great environment to experiment with cluster and MPP technologies.

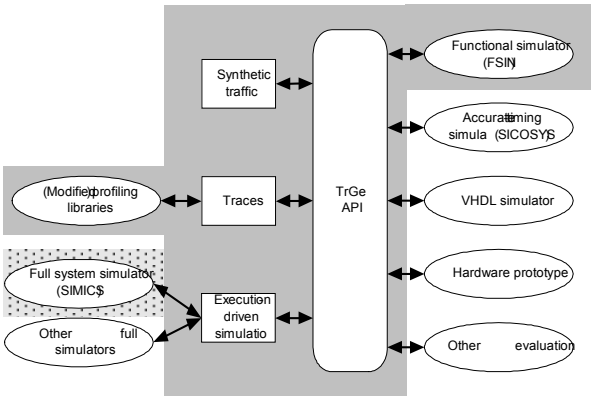


Figure 1. Overall design of INSEE (from [32]). Elements with grey background form part of INSEE, while the remaining parts are external modules. In this paper we discuss the integration with Simics

3. Interfacing IN and compute element simulators

There are many possible approaches to perform full-system simulation of a multicomputer. In Figure 2 we can observe a collection of components that take part in the simulation. All of them are software-based components, but some simulate pieces of hardware.

A single instance of an IN simulator simulates the flow of packets through a network. Several instances of full-system simulators mimic in detail the operation of the compute elements. The level of detail includes the simulation of the hardware devices, as well as the ability to run unmodified software, including operating system, support libraries (for example, an MPI library) and parallel applications.

We will pay special attention to these elements:

- The NIC (Network Interface Card) that interfaces with the IN; this is a software module that simulates the NIC.
- The driver, in kernel space, for that NIC.
- The protocol stack, in kernel space, providing higher-level access to the IN.
- The support library, on top of the protocol stack, that provides the MPI API and the necessary run-time support for parallel computing.
- A process of a parallel application.

Note that there are many instances of these elements, at least one per simulated compute node. Additionally, the simulation environment includes a synchronization module that make all

simulators advance in synchrony, and a traffic manager module that allows the interchange of information (frames, packets) between node and IN simulators, making format translations if required.

Now we focus on the different mechanisms available to implement the traffic manager. Issues about synchronization will be discussed later.

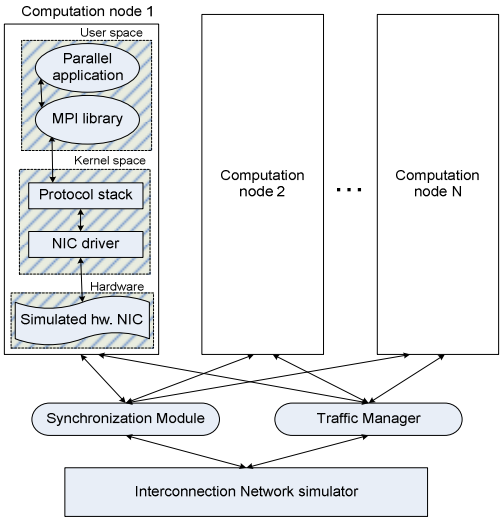


Figure 2. Elements taking part in a full-system simulation of a multicomputer

In order to better understand the following sub-sections, we show in Figure 3 several protocol stacks commonly used in cluster environments. The first three are for Ethernet hardware, commonly used in low-cost clusters: MPICH [19] over TCP, LA-MPI [11] over UDP and LAM [12] over UDP using *lamd* daemons. More expensive clusters use dedicated system area networks such as Myrinet and Infiniband. For Myrinet, Myricom provides two low-level interfaces (GM and MX), and a corresponding modification of MPICH to run over them [20]. In the case of Infiniband, the figure corresponds to MVAPICH, a modification of MPICH that runs on top of either VAPI or OpenIB low-level interfaces [13]. Note that there are many other possible stacks, for different combinations of MPI implementations (both commercial and free) and drivers for interconnection networks.

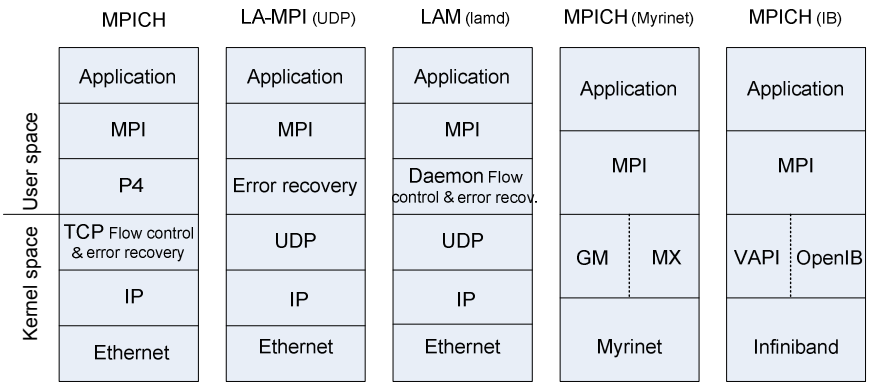


Figure 3. Examples of protocol stacks involved in the communication at every compute element for different MPI implementations and IN.

3.1 Substitution of the NIC driver

A possible approach to interface compute nodes with IN (obviously, in terms of simulation) would be via a substitution of the NIC driver, using another one that, in addition to doing its regular work, intercepts network traffic and passes it directly to the Traffic Manager. The main advantage of this option is that we can reuse the NIC model that comes with the full-system simulator, and also the Linux protocol stack. However, this approach has the disadvantage of requiring us to program a network driver for Linux that must register itself in the kernel as a general network driver. Furthermore, this driver must interact properly with the protocol stack because otherwise it would be impossible to reuse this stack. Note that the trickiest part here would be writing a driver that, while running in a simulated machine, interacts with an external module running in the simulation environment.

3.2 Substitution of the NIC simulated module

Another option would be to implement our own NIC module, which would mimic the operations of the original one while adding capabilities to interact with the Traffic Manager (sending and receiving frames). This way we can reuse protocol stack and NIC driver. However, we need to implement all the details of the simulated hardware, with all its control registers and low level accesses (writes in memory mapped registers, interruption handling, DMA accesses, etc.) Neither this option nor the previous one requires us to manipulate or re-implement the protocol stack.

Usually full-system simulation environments come with default hardware and drivers for that hardware, like Ethernet NICs. Support for other INs such as Myrinet [4], Infiniband [27] or the torus network of the Bluegene/L [1] is not readily available. These environments provide mechanisms to add new, user-designed hardware modules that can be integrated into the simulators. If we have an accurate description of a certain NIC, and we program a module that simulates this NIC, we could reuse existing software (drivers and protocol stacks) designed to run on actual hardware. For example, if we implement a very realistic Myrinet card module, we could re-use the GM drivers and the MPICH-GM MPI implementation. However, due to the difficulty of doing this accurate hardware modeling, this approach is often rejected, and multicomputer experimentation is done using default hardware (Ethernet) and protocol stacks (TCP/IP-over-Ethernet), drastically simplifying setting up the experiments.

3.3 Substitution of the full protocol stack

The third and most complex option is to program the NIC module for the simulator, the driver to run in Kernel space, and a full protocol stack – including a customized MPI implementation – on top of it. The NIC module would interface with the Traffic Manager, and the driver would take advantage of the (simulated) high-speed IN. The obvious advantage of this option is that we would have full control of the IN; experiments could be done evaluating the hardware, the software, the MPI implementation, or a combination of them. Results would be very realistic, but only if we are able to provide good-quality, bug-free software. This is, in fact, the main drawback of this option: the implementation effort is huge, and difficult to reuse. Any improvement in the simulated hardware propagates upwards: it may require driver changes, and probably MPI changes in order to take advantage of it. We need, thus, to find a trade-off between programming effort and flexibility. Reutilization of components allows us to use in our experiments good quality, well-proven software, but at the cost of using off-the-shelf components. The accurate simulation of a completely new proposal for an IN would require implementing the components that would be required if the network hardware were real, plus a detailed model of that hardware.

4. Synchronization mechanisms

In the previous sections we explained how full-system simulation of multicomputers is carried out via the combination of a collection of different simulators. These simulators are separate software entities that have different views of the passing of (simulated) time. This means that they have different simulation clocks, with different time units, and may even have different mechanisms to make those clocks advance. For example, Simics is event-driven and time is measured in CPU cycles, whose translation to actual time depend on the CPU speed, while INSEE is cycle-driven and its unit of time is a more abstract cycle: the time needed to route and move a phit (*physical* transmission *unit*) from an input port to an output port. Obviously, mechanisms are required to coordinate and synchronize those clocks in such a way that simulators for compute elements and IN advance at the same pace, as if a global clock was in use. The synchronization module takes care of this task.

The synchronization model can be strict or relaxed. Strict models are unapproachable, in terms of execution time, when performing a full-system simulation of a multicomputer, because they make exploitation of available parallelism (in the simulation platform) almost impossible. Thus, we will only consider relaxed models. In order to keep discussion simple only two simulators are considered: one that takes care of compute elements, and another one for the IN. However, discussed mechanisms can (and will) be extended to consider several, concurrent simulators for the compute elements.

One synchronization alternative is to allow the simulators to advance in lock-step mode. The compute elements simulator advances for a given amount of time (let us call it slice) and then stops. The IN simulator starts its execution and simulates the same amount of time (an equivalent one, if a translation of time units is required). It then stops and the compute elements simulator resumes its operation. Note that both simulators never run in parallel.

When a message is generated at a compute element, it is stored (with a timestamp) at an interfacing queue. Later, the IN simulator will simulate the same time slice. It will process the queue, taking care of this injection, at the right time. When the IN signals that a message has to be delivered to a compute element, again this event is stored at an interfacing queue. However, we have a problem here: that queue will not be processed until the next slice. The compute

elements simulator cannot process a message from the past; therefore all messages received during a given slice will be processed at the beginning of the next slice. In other words, messages will suffer, due to this relaxed synchronization approach, a false delay, ranging from 0 to the duration of the slice.

The other alternative allows the exploitation of parallelism in the simulation environment. We can let both simulators advance in parallel, without interchanging information. After consuming a slice, both simulators exchange lists of events. The compute elements simulator passes the list of messages generated at the slice just consumed, to be processed by the network, and the IN simulator passes the list of messages that have arrived to the destination compute elements. Note how this approach introduces two artificial delays: injection is delayed until the beginning of the next slice. Delivery, as in the previous option, is also delayed. Again, the importance of these delays depends on the slice length. A second effect is that message injections into the IN are done in bursts, at the beginning of each slice, which may impose unnecessary contention.

In both models, a very short slice length would provide maximum fidelity, but at the cost of stopping simulators very often. A long slice substantially accelerates experimentation, but introduces artificial delays that can have important, negative effects on our measurements

5. A proposal for full-system simulation of multicomputers

As we have already stated, our tool of choice to simulate the compute nodes that interchange packets through a network is Simics. From the options described in Section 3 to interchange information between simulators, we have chosen the second one: we have substituted the module that models an Ethernet NIC (a DEC21143), using another one almost identical, but capable of communicating with an external Traffic Manager module. Regarding synchronization, we use the second of the options presented in Section 4: the network simulator and Simics run in lock-step node. A certain degree of parallel simulation is performed, but only when running Simics – further details will be given later in this section.

INSEE provides a flexible environment to perform simulations of INs. Its two main modules (see again Figure 1) are FSIN (a cycle-driven, functional simulator of interconnection networks) and TrGen (a traffic-generation module). The latter allows us to feed simulations with three

different kinds of workloads: synthetic traffic patterns defined by statistical distributions, traces obtained from actual parallel application executions, and full-system simulations as described in this paper.

In the experiments detailed in the next sections, we will present results for INs built using ring topologies. The reader should note that we have chosen rings *only* because they are more prone to congestion than other kind of networks like torus of higher dimensions or fat-trees, not because we think this is a good choice of topology; in fact, INSEE can deal deal with *k*-ary *n*-cube networks of higher degree and radix and even with multistage networks as fat-trees. At any rate, the network is composed of a collection of routers, each of one is connected to several (for this work, just two) neighboring routers and to a compute node. Figure 4 represents a model of these routers. Three virtual channels (VCs) share each physical channel of the router: an Escape channel (governed by the bubble routing rules [29]), and two adaptive channels.

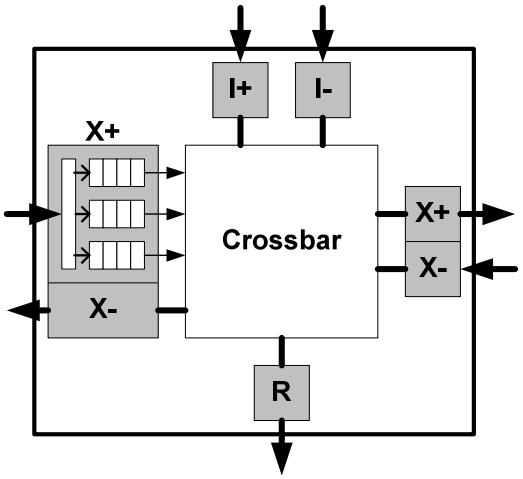


Figure 4. Model of router simulated by FSIN for 1D networks, with a detailed view of the X+ input port showing the 3 virtual channels that share this link.

Note that a ring has just one minimal path from source to destination, *i.e.* packets cannot adapt. Thus, the only difference between the Escape VC and the other two is that accesses to the adaptive VCs are not restricted by the bubble rules. Each node is able to simultaneously consume several packets arriving to the reception port. There are two injection ports, and the interface should perform a pre-routing decision: packets moving towards the X+ axis are stored in the I+ injection port, and those towards X– go to the I– injection port. Transit and injection queues are

able to store 4 packets of 16 phits (unit of transit through the wires) each. Phit length is 4 bytes and therefore the link bandwidth is 32 bits per cycle.

Regarding the simulation of the compute nodes, we use 8 instances of Simics, each one simulating 8 nodes, for a total of 64 simulated nodes. Each node runs a full Red Hat 7.3 operating system, and can be configured to use a variety of MPI implementations. As we use an Ethernet-like simulated NIC, we can use either TCP/IP/Ethernet or UDP/IP/Ethernet as low-level protocol stacks.

The process of sending a message to another different process in a multicomputer comprises several steps. In Figure 5 we expose the different modules performing these actions in our environment. First, the message is segmented into network (Ethernet) frames, which are encapsulated with different headers depending on the protocol stack used on the multicomputer. Then, these frames are sent to the NIC driver that injects them into the IN. Instead of injecting them on a real IN, the Traffic Manager module extracts them from the simulated hardware NIC and sends them to INSEE. This Traffic Manager module is responsible for sending and receiving frames from the TrGen module of INSEE. As we use a cluster of commodity computers to simulate a multicomputer, the traffic Manager sends the frames to the computer that runs INSEE.

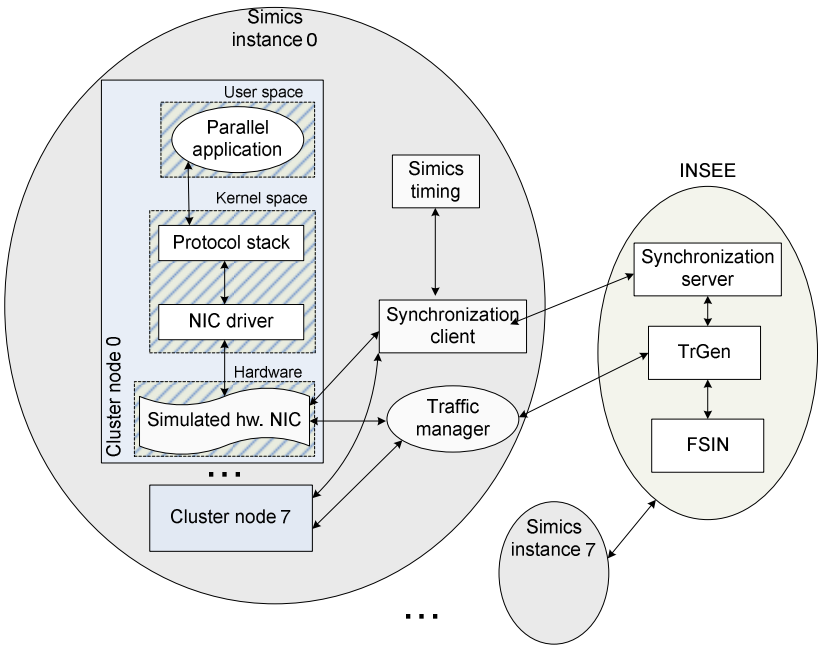


Figure 5. Elements of our full-system simulation environment that simulates an MPI application running on top of an INSEE (simulated) network.

TrGen is in charge of providing the workload for FSIN, the IN simulator core of INSEE. Frames received in TrGen are further divided into packets that are put in the right injection queue of the corresponding FSIN router. When the synchronization server signals FSIN to run, it simulates how packets travel through the network, and delivers them to their destination routers. Once all packets of a frame have arrived at the destination, TrGen will regenerate the frame putting packets together and will send the frame to the corresponding Traffic Manager on the destination node.

The Traffic Manager injects the received frames into the simulated hardware NIC. When the synchronization client resumes the run of Simics, the arrival of a frame causes an interruption that will be attended by the NIC driver. The frame is then processed, *i.e.* the headers of the protocol stack are removed and the message is rebuilt and delivered to the application process.

The ability of running several compute nodes in each Simics instance requires the utilization of *two* different synchronization mechanisms. The first one is used to coordinate all compute nodes inside a Simics instance: every node runs a specific number of cycles (slice) and then the next compute node and so on, in a round-robin fashion. This mechanism is integrated into the own Simics working. The second synchronization mechanism is used to coordinate the compute nodes and INSEE. We use a client-server model in a lock-step mode to do it. There is a synchronization server in INSEE and a synchronization client on each Simics instance, as shown in Figure 5. When all nodes in a Simics instance have completed a slice, the synchronization client stops that instance, and then sends a *timestamp signal* to the synchronization server asking for permission to run another slice. During a running slice, all generated network traffic is kept in the injection queues.

When the synchronization server has received timestamp signals from all Simics instances, INSEE allows FSIN to run a slice, routing the received packets up to their FSIN router destination. When FSIN finishes its slice, it sends a multicast timestamp signal to all synchronization clients, allowing them to resume the execution of the compute nodes. This is shown in Figure 6.

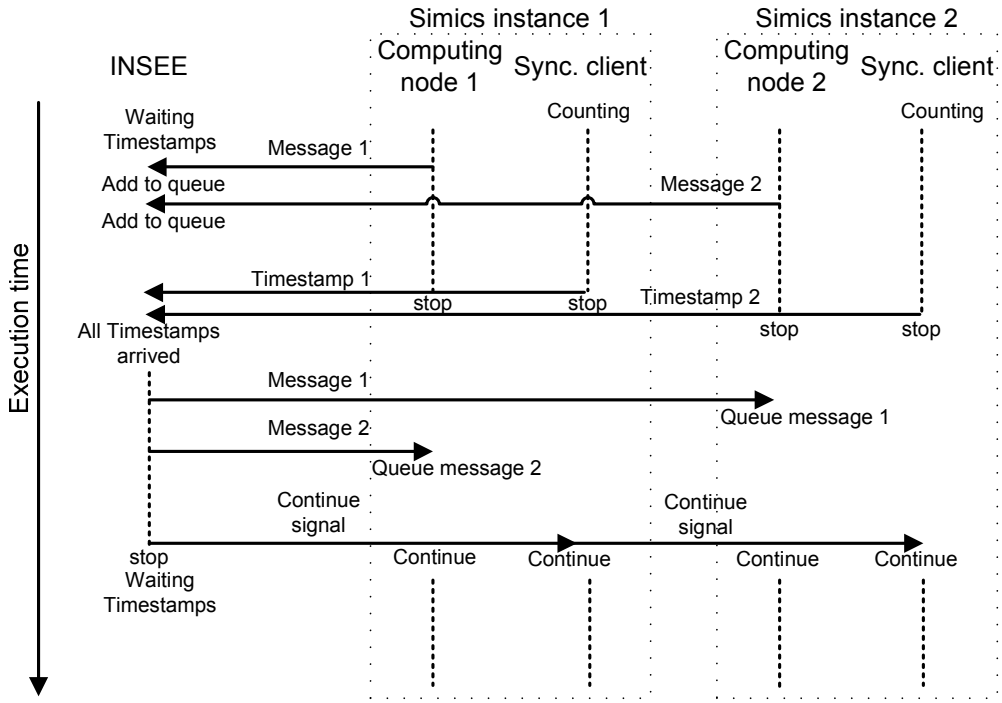


Figure 6. Synchronization mechanism between Simics instances and INSEE, and message routing. Only one computing element per Simics instance is shown

Recall that this synchronization mechanism allows network injections to be simulated precisely, but deliveries are artificially delayed until the start of the next slice. The main difficulty here is to find a trade-off between the high execution speed provided by long slices and the accuracy obtained from short ones.

6. Experimental environment

As an example of the capability of our environment, our proposal has been used to study the effects of network-based congestion control in the execution speed of MPI parallel applications. In order to better understand the experimental work, we start this section defining the congestion control mechanisms under study.

6.1 Network congestion control

Congestion may appear when the utilization of resources inside the IN is close to its limits; its negative effects include throughput and delay degradation. If no action is taken when congestion appears, it soon spreads through the whole network. Congestion control techniques usually limit

packet injection as soon as the network presents signs of congestion. There are different ways of diagnosing these signs and techniques to avoid congestion [9], based on global knowledge like in [36], distributed like RECN [6] or based on information of the local router buffers like LBR [17]. The torus network of the IBM BlueGene/L includes a mechanism that works prioritizing in-transit traffic—we call this IPR (In-transit Priority Restriction).

In an initial set of experiments, we evaluated two of these congestion control mechanisms for INs, IPR and LBR, both based on locally available information, with good performance and minimal implementation costs [17]. When using IPR, priority is given to in-transit traffic for a given number *P* of cycles. In those cycles, the 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). The Local Buffer Restriction (LBR) mechanism has been designed specifically for adaptive routers that rely on Bubble Flow Control [29] to avoid deadlock in the escape sub-network. LBR extends the bubble restriction to all new packets that enter the network. Subsequently, 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 level of buffer utilization.

6.2 The experimental set-up

The effects of congestion were studied on the previously described environment. The testbed system is composed by 64 compute nodes connected through a ring network. The number of nodes (64) is a consequence of the availability of resources to obtain actual traces and run the full-system simulation environment. As we stated before, the choice of a ring (instead of a more reasonable 8x8 torus) is because the ring is, for a given number of nodes, more prone to congestion than a 2D torus. The system was composed by 64 Intel Pentium-4 processors, running Red Hat 7.3 with kernel 2.4 at 200 MHz (1 Simics cycle = 5 ns), with 64 MB of RAM. Unless otherwise stated, the synchronization slice was 10000:200, meaning that 200 INSEE cycles are made equivalent to 10000 Simics cycles; this results in a link bandwidth of 128 Mb/s, approximately the speed of a Fast Ethernet.

We used the IPR and LBR network congestion control mechanisms. For the following experiments, we fixed the values: P=1 (the maximum priority) and B=3 (inject in an adaptive channel only when its queue is empty or almost empty). Note that the "Base" case corresponds to both mechanisms being deactivated, *i.e.* B=0 and P=0.

The applications used to perform the experiments were a subset of the A class of the NAS Parallel Benchmarks [21] (NPB), a well-known, allegedly representative set of parallel application workloads often used to assess the performance of multicomputers. To simplify descriptions and discussions, we have focused on three of these benchmarks: BT, CG and IS. BT is computation-biased (although not embarrassingly parallel) and should show how gains at the network level do not have a great impact on execution time. In contrast, CG and IS are communication-intensive, so the effects of network changes should be more clearly visible.

We started our study with an initial evaluation using traces captured in a small-size cluster with 8 dual-core Intel Xeon nodes, connected via Gigabit Ethernet. Several tasks time-shared each processor, so that the timing information about CPU intervals stored in the trace records is not valid. Nevertheless, in the trace-based simulations we do not use CPU times, just keep causal relationship between messages, as if we were using infinite-speed processors. The mechanisms used to capture the traces, as well as the methodology to perform the simulation, are described in detail in [18] [31]. With this trace-based simulation we obtained some *predictions* of the maximum performance improvement achievable with the use of congestion control mechanisms. Then, we used our full-system simulation environment to verify these predictions. Obviously, results obtained with this environment should be consistent with those predicted by the trace-based study, but reduced in magnitude because of the inclusion of the computation part.

As our environment allows us to use different protocol stacks, we have tested a few, different ones: from the collection shown in Figure 3 we have evaluated those with Ethernet at the hardware level. When TCP is used at the transport level, we use the usual version of TCP Reno [9].

7. Experiments and discussion of results

In this section we describe in detail the experiments carried out, and analyze the results. The first set of experiments consists of several full-system simulations designed to evaluate the effects of network-level congestion control on the execution speed of NPB applications. The results of the trace-based study are also plotted in the figures, to compare the predictions obtained this way with the results from the full-system study. From this set we learn about the interactions between end-to-end (TCP) and network-level congestion control mechanism, an issue that is further studied in the second set of experiments. The last part of this Section is devoted to explore the effects of synchronization frequency in simulation accuracy and execution times.

Note that all the reported results normalize execution times to the Base case, which is, as described before, the one without IN-based congestion control. Experiments are repeated 10 times, and in the figures we plot the average values of results. Variances, summarized in tables, are included too.

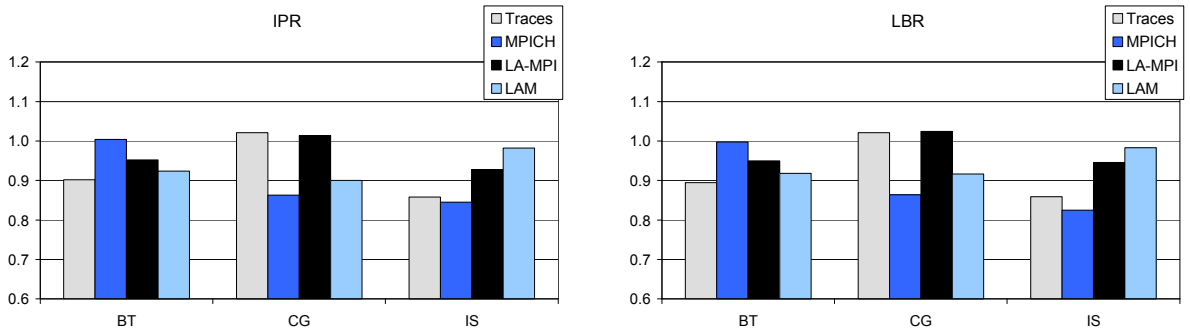
7.1 Trace-based simulation

The most relevant results of these experiments are plotted in Figure 7 (light-grey bars). They predict that the reduction of execution time achievable from applying congestion control mechanisms should be, in the best case, a 15% (IS benchmark). However, we should expect small performance drops in applications such as CG: around a 3%. Both network-level congestion control mechanisms are beneficial for BT and IS, because their traffic patterns consist of large messages that are able to saturate the IN. In contrast, CG is harmed by these mechanisms, due to its traffic pattern, which consists of sequences of small messages, arranged by dependency chains. Thus, congestion does not appear, and any mechanism that restricts packet injection is harmful, because it imposes unnecessary delays that accumulate and increase the overall execution time.

The reader should note that the method we use to carry out trace-based simulation exaggerates the potential advantages / disadvantages of a given congestion control mechanism, because we do not take into account the CPU time (we only simulate the interchange of messages). Real applications only spend a portion of its time performing communications; hence our prediction applies only to this portion. This limitation does not apply to full-system simulations.

7.2 Assessing the effects of network-level congestion control

The full-system simulation did not, to our surprise, confirm the predictions of the trace-based simulation: results (summarized in Figure 7 and Table 1, along with those obtained with traces) were in some cases much more favorable than expected, and in some others poorer than expected. The causes of these mismatches were some unaccounted-for interactions between end-to-end congestion control (TCP incorporates its own mechanisms) and the network-level congestion control (those under test, IPR and LBR).



a) **b)**

Figure 7. Measured execution times relative to the base case (IPR and LBR deactivated). a) Results obtained when IPR is activated. b) Results obtained when LBR is activated.

Table 1. Variances of the experiments plotted in Figure 7

	Base			IPR			LBR		
	BT	**CG**	**IS**	**BT**	**CG**	**IS**	**BT**	**CG**	**IS**
MPICH	2.76E-04	8.12E-04	5.23E-04	2.70E-04	1.57E-05	6.47E-04	9.36E-04	5.85E-06	1.86E-03
LA-MPI	2.68E-04	9.31E-05	1.55E-03	2.80E-04	1.54E-03	1.41E-03	4.04E-04	1.95E-03	2.16E-03
LAM	7.96E-04	4.13E-04	4.98E-03	2.33E-04	1.03E-03	4.11E-03	4.29E-04	1.19E-03	4.40E-03

In the case of experiments made with MPICH and without any network congestion control mechanism, application execution times were negatively affected by TCP's wrong estimation of timeouts, triggering retransmissions and continuously activating the slow start protocol [8]. Those made the whole execution very slow in saturated networks. In contrast, when IPR or LBR were applied, jitter was reduced, which helped TCP to determine its timeout values which reduced the retransmissions and, consequently, the occurrences of the slow start mechanism. Therefore, both IPR and LBR caused two overlapping effects:

- For most applications, the flow of packets through the network was accelerated.
- In all cases, they helped TCP, allowing the applications to reach higher throughput.

The second effect was unexpected, and was more significant than the first. This explains the mismatch in our predictions.

In order to avoid these interactions, we tested other MPI implementations with protocol stacks that do not include TCP. One of those is LA-MPI over UDP. This implementation performs error control at application level to avoid message losses (messages are dropped when intermediate buffers are full). This error control slightly affects the execution time. Despite this, the absence of TCP made LA-MPI a good platform for experimentation. However, as this project is no longer supported, we decided to search for another non-TCP based MPI implementation.

LAM over UDP routes all messages through a daemon present in every compute element. This adds two hops to every message. And these daemons also do the flow control and error recovery not available while using UDP. For this reasons this configuration is very slow and also affects the performance in a similar way as TCP does. This can be seen in Table 2.

We can conclude that the reutilization of components, in this case protocol stacks, may look as a good idea, because it reduces the time of setting-up an evaluation environment and reduces programming errors, but the price to pay may be too high: it may introduce unforeseen interactions that can magnify, hide or even invalidate the results obtained. A more detailed explanation of this issue is provided in [23].

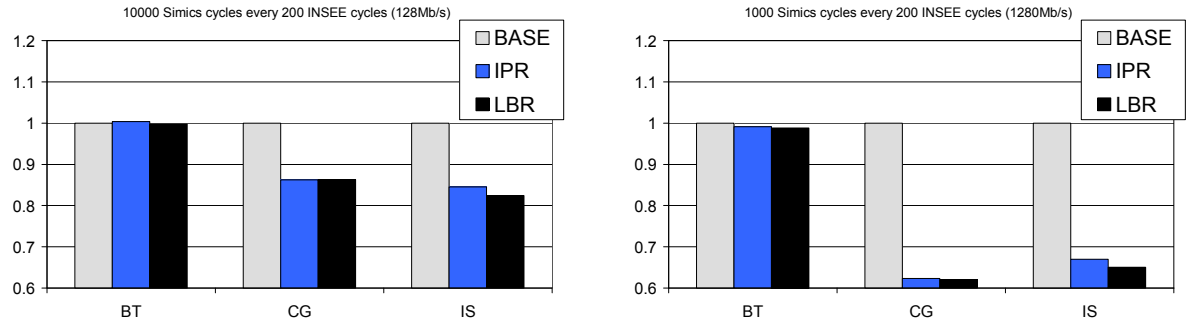
Table 2. Simulated time needed to run an iteration of each benchmark for different MPI implementations in the full-system simulation environment. Average of 10 runs

	BT	CG	IS
MPICH	4.52s	5.89s	4.21s
LA-MPI	4.51s	5.60s	4.10s
LAM/UDP	5.09s	10.88s	10.38s

7.3 Congestion control and network speed

Once the interaction between end-to-end and network congestion control mechanisms was detected, we carried out additional experiments in order to assess the influence of these mechanisms for different network speeds. In addition to the experiments described in the previous section, for a network of 128 Mb/s, we repeated the experiments on a faster network of 1280 Mb/s and only with MPICH, one of the most widely-used noncommercial MPI implementations. Results are summarized in Figure 8 (averages) and Table 3 (variances).

An increase in network speed makes TCP perform even worse when there is no network-based congestion control mechanism. When the traffic is not enough to fill up the network and the network speed is high, TCP estimates correctly its timers, and is able to deliver traffic with high throughput levels. However, in those phases where contention for network resources appears, because the traffic pattern requires it, packets slow down. Some of them even "get lost" from the point of view of TCP, because they arrive too late. This jitter activates TCP's flow control and error recovery mechanism. This activation slows down the performance in the same way as a slower network because the slow-start algorithm is network independent. Again, the utilization of network-level congestion control reduces jitter, and improves performance. As we can see, the effect of IPR/LBR is more beneficial at higher network speeds, not only in terms of average values but also in reduction of variance; this means that the adverse behavior of TCP gets worse in faster networks. Note the great differences in performance for the CG and IS benchmarks, those that are more communication-intensive.



a) **b)**

Figure 8. Measured execution times relative to base case (IPR and LBR deactivated). a) Obtained in a 128Mb/s network. b) Obtained in a 1280Mb/s network

Table 3. Variances of the experiments plotted in Figure 8

	128 Mb/s			1280 Mb/s		
	BT	**CG**	**IS**	**BT**	**CG**	**IS**
Base	2.76E-04	8.12E-04	5.23E-04	4.90E-05	2.94E-03	1.38E-02
IPR	2.70E-04	1.57E-05	6.47E-04	8.32E-05	2.24E-04	3.90E-04
LBR	9.36E-04	5.85E-06	1.86E-03	1.03E-04	3.84E-04	8.17E-05

7.4 The simulation speed / accuracy trade-off

We explained in previous sections that an issue when linking two different simulators is to fine-tune the synchronization among them. In particular, we need to define the slice duration: the

period of time in which a given simulator advances without synchronizing with the rest. The duration of the slice affects both the accuracy and the simulation performance. The longer the slice, the larger the delay and jitter, hence the accuracy decreases. In addition, every time the simulators synchronize, a penalty has to be paid; therefore simulation performance is better for longer slices because the actual time to run the simulation is shorter. The effect of a short slice is just the opposite: more synchronization overhead, but better accuracy because delay and jitter are reduced.

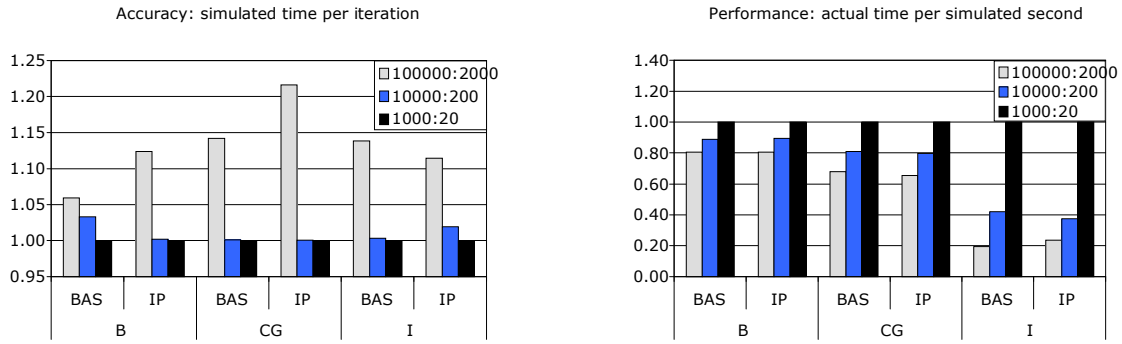
We ran another set of experiments to measure the accuracy versus performance relation due to the synchronization mechanism. We used an MPICH over TCP protocol stack, fixed the network speed at 128 Mb/s, and used three different synchronization values: 100000:2000, 10000:200 and 1000:20. We took the time reported by the simulator to execute an iteration of the benchmarks under study (BT, CG and IS) with and without IPR, and also the wall-clock time that the simulator required to perform the experiment. Results are plotted in Figure 9.

Figure 9a shows that, in terms of simulator-reported time, there is little difference in terms of simulated time per iteration between the highly synchronized case (1000:20) and the one in which synchronization frequency is reduced to one tenth of that (10000:200), in which benchmark execution is at worst only 3.28% slower (BT Base). This small difference can be explained because of the additional delays introduced by the lock-step synchronization. In the 10000:200 case, a packet generated at a node may need to wait up to 9999 Simics cycles (50μs) before being injected into the network. This resulted in large delay variations. In the 1000:20 experiments, the worst-case additional delay was reduced to 999 Simics cycles (5μs) – reducing jitter and allowing TCP to perform better.

When synchronization is infrequent, as in the 100000:2000 case, results change in a significant way. Huge delays and jitter are artificially introduced, as we force packets to wait up to 99999 (500μs). With these unstable, long delays TCP cannot make good estimations of delays, and activate too often the slow-start mechanism.

Besides, we could unnecessarily saturate the IN. The maximum injection rate of a Fast Ethernet is of 148.800 [10] frames of minimum size (72 Bytes, that will require 2 FSIN packets) per second without collisions, which gives us an injection rate of a frame every 1344 Simics cycles, using the parameters of our simulation environment. This means that, at every

synchronization step, every node could have up to 70 frames pending for injection, which could heavily saturate the network. It must be taken into account that our environment uses a regular device (Fast Ethernet) on top of regular protocols (MPICH/TCP/IP/Ethernet) over a custom IN, which it is not designed for them. Therefore, simulation artifacts are triggering mechanisms of TCP that should not operate, generating inaccurate results.



a) **b)**

Figure 9. Time measurements for each benchmark with different synchronization parameters. Results relative to the most synchronized case (1000:20) a) Simulated time to complete each iteration. b) Actual time to simulate a second

Figure 9b represents the wall-clock time to simulate a second of each benchmark. This time is inversely proportional to the slice length. The longer the slice, the shorter the time needed to simulate a second, up to only 20% of the time needed if we compare the longer slice (100000:2000) with the shorter slice (1000:20) for the IS base case. The savings of time can be between 10% up to 60% if we increase a magnitude the slice duration. Therefore, it is clear that a carefully study is needed to choose the synchronization slice duration. We must choose a slice short enough to get reliable and correct results, and long enough to complete simulations in a reasonable time. In our case, the 10000:200 slice appears to be a good choice, because we only loose a 3% on accuracy on the worst case, and the simulation lasts up to 60% less time than the case with the shortest synchronization slice.

In summary, as a performance study requires many simulation runs, researchers are encouraged to spend some time exploring this speed/accuracy trade-off, searching for those slice durations that provide faster simulation runs but without sacrificing truthfulness of results. Actual values to explore depend on factors such as (relative) speeds of processors and network,

so we cannot provide a rule of thumb to get the right durations. Also, some precision can be sacrificed in initial simulation runs, switching to slower set-ups when accuracy is essential.

8. Conclusions

Full-system simulation is a very complex matter, complexity that is greatly increased when trying to simulate not just a computer, but a collection of networked machines – especially if the network and the interfaces differ from the traditional LAN devices available in simulation environments. It is also an intensive resource-consuming task. The simulation of a cluster of computers may require an actual machine with similar characteristics to the one under study, but the speed at which we obtain performance results would be several orders of magnitude slower.

In this paper, we show that full-system simulation of INs requires a large collection of interrelated (software) components, which, in many cases, have to be done from scratch, or re-used from those provided by the simulation environment being used. The reutilization allows for important reductions of implementations effort and errors, but implies some risks of using, for a given purpose, components designed for different (although related) purposes, and may lead to inaccurate or even invalid simulation results. The synchronization between parts of the simulation environment needs also to be carefully designed, in order to find a good trade-off between simulation fidelity and execution time of the experiments. We have presented our proposal of combining INSEE with Simics, which has allowed us to learn, by experience, about the plethora of factors that have an effect on the quality of results: protocol stacks, MPI implementations, drivers, synchronization modules, etc. Still, this environment allows us to evaluate, on realistic scenarios, the effects of network-level congestion control, and the interactions between end-to-end and network level mechanisms. As far as we know, no other tool (or combination of tools) is available providing the same set of features.

As future lines of work, we plan to improve our toolset including models of the hardware NICs used in current super-clusters; in particular, our target is to have a model of an Infiniband HCA, and to add models of Infiniband networks into FSIN. Also, as full-system simulation is very slow, we are working in improving the other, significantly faster, traffic-generation modules

of INSEE (synthetic patterns and traces), with special focus on synthetic traffic that, while algorithmically generated, is inspired in actual applications [22].

9. 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. Mr. Javier Navaridas is supported by a doctoral grant of the UPV/EHU.

10. References

[1] N.R. Adiga et al., "Blue Gene/L torus interconnection network." IBM Journal of Research and Development, Volume 49, Number 2/3, 2005.

[2] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, S. K. Reinhardt. "The M5 Simulator: Modeling Networked Systems". IEEE Micro, vol. 26, no. 4, pp. 52-60, July/August, 2006.

[3] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW), Feb. 2003.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, et al. "Myrinet. A gigabit per second local area network". IEEE-Micro, Vol.15, No.1, February 1995, pp.29-36.

[5] The Chaotic Routing Project at the U. of Washington. Chaos Router Simulator. Available (May 2008): http://www.cs.washington.edu/research/projects/lis/chaos/www/chaos.html

[6] P.J. García, F. J. Quiles, J. Flich, J. Duato, I. Jhonson, F. Naven. "Efficient, scalable congestion management for interconnection networks". IEEE MICRO 26 (5): pp 52-66 Sep.-Oct 2006.

[7] IBM. "IBM Full-System Simulator for the Cell Broadband Engine Processor". Available (May 2008) at http://alphaworks.ibm.com/tech/cellsystemsim

[8] Jacobson, V., "Congestion Avoidance and Control", Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988.

[9] R. Jain. "Congestion control in computer networks: issues and trends". IEEE Network, v.4 n.3, pp 24-30, May 1990.

[10] S. Karlin and L. Peterson. Maximum Packet Rates for Full-Duplex Ethernet. Technical Report TR--645--02, Princeton University, February 2002

[11] LA-MPI Home Page "The Los Alamos Message Passing Interface" Available (May 2008) at http://public.lanl.gov/lampi/

[12] LAM/MPI Home Page "LAM/MPI Parallel Computing." Available (Apr. 2008) http://www.lam-mpi.org/

[13] J. Liu, J. Wu, and D. K. Panda. "High Performance RDMA-Based MPI Implementation over InfiniBand", International Journal of Parallel Programming, 2004.

[14] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg amd Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. "Simics: A full system simulation platform". IEEE Computer, 35(2):50–58, February 2002.

[15] M.M.K. Martin et al., "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," Sigarch Computer Architecture News, v. 33, n. 4, Sept.05, pp. 92-99.

[16] C.J. Mauer, M.D. Hill, and D.A. Wood. “Full-System Timing-First Simulation”, ACM SIGMETRICS, June 2002.

[17] J. Miguel-Alonso, C. Izu, J.A. Gregorio. “Improving the Performance of Large Interconnection Networks using Congestion-Control Mechanisms”. Performance Evaluation 65 (2008) 203–211.

[18] J. Miguel-Alonso, J. Navaridas, F. J. Ridruejo. “Interconnection Network Simulation Using Traces of MPI Applications”. Int J Parallel Prog (to appear). DOI 10.1007/s10766-008-0089-y

[19] MPI Forum. “MPICH Home Page”. Available (May 2008) at http://www-unix.mcs.anl.gov/mpi/mpich/

[20] Myricom Documentation and Software Downloads. Available (May 2008) at http://www.myri.com/scs/

[21] NASA Advanced Supercomputing (NAS) division. “NAS Parallel Benchmarks” Available (May 2008) at http://www.nas.nasa.gov/Resources/Software/npb.html

[22] J Navaridas, J Miguel-Alonso, FJ Ridruejo. "On synthesizing workloads emulating MPI applications". The 9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC-08). April 14-18, 2008, Miami, Florida, USA.

[23] J. Navaridas, F.J. Ridruejo, J. Miguel-Alonso. "Evaluation of Interconnection Networks Using Full-System Simulators: Lessons Learned". Proc. 40th Annual Simulation Symposium, Norfolk, VA, March 26-28, 2007

[24] The Network Simulator ns-2. Available (May 2008) at http://www.isi.edu/nsnam/ns/

[25] OPNET Technologies, Inc. corporate web page, available (May 2008) at http://www.opnet.com

[26] V.S. Pai, P. Ranganathan, and S.V.Adve. “RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors”. IEEE TCCA New., Oct. 1997.

[27] Gregory F. Pfister “Aspects of the InfiniBand(tm) Architecture”. Third IEEE International Conference on Cluster Computing (CLUSTER'01). October 2001. pp. 369

[28] V. Puente, J.A. Gregorio, R.Beivide (2002). SICOSYS: An Integrated Framework for studying Interconnection Network in Multiprocessor Systems, Proceedings of the IEEE 10th Euromicro Workshop on Parallel and Distributed Processing. Gran Canaria, Spain.

[29] V. Puente, C. Izu, J.A. Gregorio, R. Beivide, and F. Vallejo, “The Adaptive Bubble router”, J. of Parallel and Distributed Computing, v. 61, n. 9, pp.1180-1208 Sep. 2001.

[30] V. Puente, J. A.Gregorio, F. Vallejo and R. Beivide, “Immunet: A Cheap and Robust Fault-Tolerant Packet Routing Mechanism”, International Symposium on Computer Architecture (ISCA), June 2004. pp 198-211

[31] F.J. Ridruejo, A. Gonzalez, J. Miguel-Alonso. "TrGen: a Traffic Generation System for Interconnection Network Simulators". International Conference on Parallel Processing, 2005. 1st. Int. Workshop on Performance Evaluation of Networks for Parallel, Cluster and Grid Computing Systems (PEN-PCGCS'05). ICPP 2005 Workshops. 14-17 June 2005 Page(s): 547 - 553

[32] F.J. Ridruejo, J. Miguel-Alonso. “INSEE: an Interconnection Network Simulation and Evaluation Environment”. Lecture Notes in Computer Science, Volume 3648 / 2005 (Proc. Euro-Par 2005), pp. 1014 - 1023.

[33] M. Rosenblum et al., “Complete Computer System Simulation: The SimOS Approach,” IEEE Parallel & Distributed Tech., vol. 3, no. 4, Winter 1995, pp. 34-43.

[34] Lambert Schaelicke, Mike Parker, "ML-RSIM Reference Manual," tech. report 02-10, Department of Computer Science and Engineering, Univ. of Notre Dame, Notre Dame, Ind., 2002.

[35] SMART group at the U. of Southern California. FlexSim 1.2. Available (May 2008) at http://ceng.usc.edu/smart/FlexSim/flexsim.html

[36] Mithuna Thottethodi, Alvin R. Lebeck, Shubhendu S. Mukherjee, “Exploiting Global Knowledge to Achieve Self-Tuned Congestion Control for k-Ary n-Cube Networks”, IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 3, pp. 257-272, Mar., 2004.

[37] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, J. C. Hoe, “SimFlex: Statistical Sampling of Computer System Simulation”, IEEE Micro, vol. 26, no. 4, pp. 18-31, Jul-Aug 2006.