Full-system simulation of distributed memory parallel computers using Simics

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

Abstract

In this paper we discuss environments for the full-system simulation of multicomputers. These are composed of a large collection of modules that simulate the compute nodes and the network, plus glue elements that perform communication and synchronization. We present our own environment based on Simics and INSEE. We reuse as many Simics modules as possible to reduce the effort of hardware modeling, and simulate standard machines running unmodified operating systems. We explain how experiments reveal unforeseen interactions among all modules and components, providing results that are difficult to interpret. Another important issue is the synchronization among simulators: a trade-off has to be found between simulation speed and accuracy of results.

1. Introduction

The design of a supercomputer is a complex task that comprises the selection and design of multiple components, such as computing elements, storage, interconnection network, access elements and the software it uses, from the operating system to high performance libraries and parallel applications. 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 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.

When system components have been chosen, a validation of the whole design is required to confirm that the behavior is as expected, and to check that there are no undesirable interactions between components that may have passed unnoticed before due to the simplicity 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 *glue* work to put them to work 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 – something that 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.

In this paper we describe the components that take part in a full-system simulation of a multicomputer, and our proposal that mixes two very different simulators, Interconnection Network Simulation and Evaluation Environment [16] (INSEE for short) in charge of the IN, and Simics [7], used to simulate the compute nodes.

This work has been done with the support of the Spanish Ministerio de Educación y Ciencia, grant TIN2004-07440-C02-02. Mr. J. Navaridas is supported by a doctoral grant of the UPV/EHU.

We also discuss several other 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 modules. 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 is about options to synchronize these simulators. Section 5 describes our simulation environment. Section 6 describes some experimental work done with this environment, and identifies some issues detected. Finally, in section 7 we enumerate the conclusions of this work.

2. Related work

There are many 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 find several full-system simulators in [9].

When evaluating parallel computers, models used for the IN are often too simplistic. Most 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 sophisticated IN models inside available full-system simulation tools; however, IN simulators are already there as standalone tools (SICOSYS [14], the Chaos Router Simulator [2], FlexSim [17] and many others). Instead of starting from scratch, designing a sophisticated Simics module of the target IN, it would be easier for designers to make available tools collaborate. For example, SICOSYS can interface with RSIM [13] to do full-system simulation of shared-memory parallel computers, providing an accurate timing model. However, this setup consumes a huge amount of resources (memory, CPU time) and only allows for the simulation of tens (a few hundreds at most) of interconnected compute nodes.

GEMS [8] and SIMFLEX [4] are based on Simics. The Simics environment provides the system simulation, and the other tools provide an accurate timing model which allows doing a high-fidelity performance evaluation of systems. This is required because Simics timing is a very simple mechanism in which all instructions and memory accesses take the same time. These tools in combination do accurate performance evaluation of shared-memory parallel computers via full-system simulation. They have detailed multiprocessor memory systems but lack detailed I/O models and multiple-system capability.

As we can see, these tools are heavyweight, and focus on shared-memory machines. Our interest is on multicomputers, and we already use INSEE as a simulation environment to evaluate proposals for the INs used in this kind of architectures. The interaction of INSEE with Simics, the tool of choice to provide full-system simulation of computing elements, provides a great environment to experiment with cluster and MPP technologies.

3. Interfacing IN and node simulators

There are many possible approaches to perform full-system simulation of multicomputers. In Fig. 1 we can observe a collection of components that take part into the simulation. Some of them are pure software, some others are also software, but simulate pieces of hardware.

One instance of an IN simulator simulates the flow of packets through a network. Several instances of node simulators simulate in detail the operation of the compute elements. The elements that are of interest for our purposes are: (1) A NIC that interfaces with the IN; actually, this is a software module that simulates the NIC. (2) A driver, in kernel space, for that NIC. (3) A protocol stack, in kernel space, providing higher-level access to the IN. (4) A library on top of that stack, providing the MPI API and run-time. (5) A process of a parallel application.

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 (packets) between node and IN simulators, making format translations if required

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

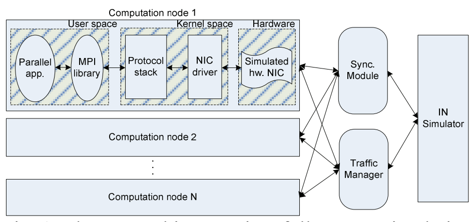

Fig. 1. Elements taking part in a full-system simulation of a multicomputer

3.1. Substitution of the NIC driver

The first option is to substitute the NIC driver with another one that 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 used protocol stack because otherwise it would be impossible to reuse it. Note that the main *trick* here would be to have a driver running in a simulated machine interact 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, adding capabilities to interact with the Traffic Manager (sending and receiving packets) but mimicking the operations of the *original* one. Using this option 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, Infiniband or the torus network of the Bluegene/L [1] is not readily available. These environments provide mechanisms to add new, user-designed hardware modules, than 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 over 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 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. Results would be very realistic—but only if we are able to provide good-quality, bug-free software. This is, in fact, the 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 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. They can 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 (time needed to route and move a phit from the input ports to an output port). Obviously, mechanisms are required to coordinate and synchronize those clocks, so that simulators for nodes 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 only consider more relaxed models. In this discussion we consider only two simulators: one that takes care of nodes, and another one for the IN. However, discussed mechanisms can be extended to consider several, concurrent simulators for the nodes.

One synchronization alternative is to allow the simulators to advance in lock-step mode. The nodes 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 nodes simulator resumes its operation. Note that both simulators never run in parallel.

When a message is generated at a node, 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 node, 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 nodes simulator cannot process a message from the past, so 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 is by exploiting 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 nodes 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 nodes.

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 duration. A second effect is that message injections in the IN is 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

We have chosen Simics [7] as the tool to simulate the compute nodes that interchange packets through a network. From the options described in Section 3, 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.

INSEE provides a flexible environment to perform simulations of IN. It consists of two main modules: a cycle-driven, functional simulator of interconnection networks (FSIN), and a traffic-generation module (TrGen). The later module 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 our experiments we will present results for 64-node rings. We will explain later the reasons to make this unusual choice. The network is composed of a collection of routers, each of one is connected to a two neighboring routers and to a compute node. Fig. 2 represents a model of these routers. Each physical channel of the router is shared by three virtual channels (VCs): an Escape channel (governed by the bubble routing rules [15]), and two adaptive channels.

Note that a ring has just one minimal path from source to destination, so packets cannot adapt. Thus, the only difference between the

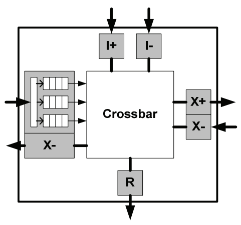

Fig. 2. Model of router simulated by FSIN for 1D networks, with a detailed view of the X+ input port showing the 3 virtual channels that share its link.

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, so 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. Each node runs a full Red Hat 7.3 operating system, and can be configured to use some MPI implementations [12]. For this work we chose MPICH [10] because it is widely used and supports several protocol stacks, depending on the underlying IN (Ethernet, Infiniband, Myrinet, …). As we use an Ethernet-like simulated NIC, we use the P4/TCP/IP/Ethernet protocol stack.

In a real multi-computer the flow of information between two application processes is as follows. Whenever a node wants to send a message to another, this message passes through several software layers to build one or many adequately formatted packets. First, the message is segmented and encapsulated in kernel space by a protocol stack (in our case TCP/IP/Ethernet). Then the driver of the NIC injects the generated packets into the network interface card that, in turn, injects the packets into the IN. When a packet arrives from the IN to a NIC, the driver is signaled and a process to obtain the original message (maybe reassembly several packets) is performed, in order to deliver the message to the right application process.

In our environment everything is as described here, with a few exceptions. See, in Fig. 3, the collection of components taking part in the simulation. The network is not a real Ethernet. Instead, it is simulated by FSIN. The hardware module that simulates the DEC21143 fast Ethernet NIC receives a collection of packets (actually, Ethernet frames) that are used, with the help of the Traffic Manager, as workload for FSIN. The interchange of workload is performed using an actual network because INSEE and Simics run on different machines.

Packets are received by the TrGen module in INSEE that is in charge of providing the workload for FSIN. TrGen puts a received packet into the right injection queue at the corresponding FSIN router. Then, FSIN simulates the way that packet travels through the network, sharing its resources with other packets, and delivers it to the appropriate destination router. When this happens, the packet is sent back to TrGen, which uses the Traffic Manager to inject it into the NIC at the destination node. When a packet is injected at a (simulated) NIC, this arrival causes an interruption that is attended by the NIC driver. The rest of the process that end with a message being received by an application process is exactly the same that happens with a real multicomputer.

The synchronization among all the Simics instances and INSEE is done at two levels. Simics

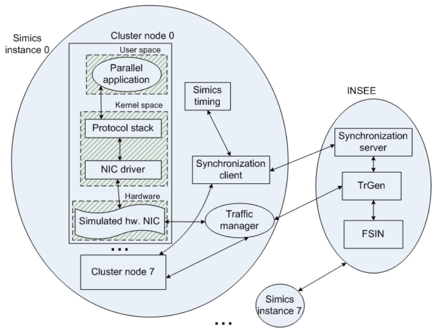

Fig. 3. Elements of our full-system simulation environment that simulates an MPI application running on top of an INSEE (simulated) network.

incorporates its own synchronization mechanism, which is used to coordinate the eight computers simulated by each Simics instance. This is done in round-robin fashion: each node runs for a specific number of cycles, then the next node and so on.

The second level of synchronization is among Simics instances and INSEE, in a lock-step way, using a client-server model. Each instance of Simics includes a synchronization client, and INSEE includes a synchronization server. We can see these two modules in Fig. 3. A synchronization client allows a Simics instance to run for a pre-defined number of cycles (slice). After completing the slice, the Simics instance (thus, all the nodes simulated by it) stops, and a *timestamp* signal is sent to the synchronization server. During a slice, computing nodes can send messages to other nodes, but those are stored in a synchronization queue.

When the synchronization server has received timestamp signals from all the Simics instances, INSEE runs for a number of FSIN cycles (a slice), routing and sending the received messages to their corresponding destinations, before sending, via multicast, a *continue* signal to all the clients – which allows the Simics instances to resume their executions.

Remember 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.

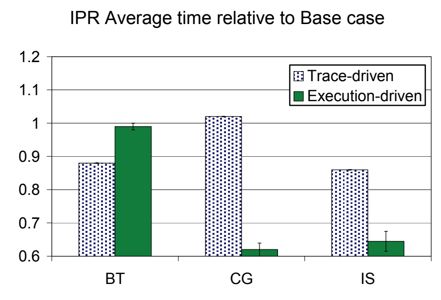

Fig. 4. Relative times to complete a run of NPB's BT, CG and IS, and 99% confidence intervals.

6. Experimental work

The environment described in the previous section has been used to study the effects of network-based congestion control in the execution speed of MPI parallel applications. 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, based on global knowledge like in [18], distributed like RECN [3] or based on information of the local router buffers like LBR [5]. The torus network of the IBM BlueGene/L [1] 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 several of these congestion control mechanisms for INs using trace-based traffic, obtaining some predictions of performance improvements. Then, we used our full-system simulation of IN to validate these predictions – see [12] for the full details. In particular, we studied the effects of IPR on a ring of 64 nodes with multiple injection sources. The number of nodes (64) is a consequence of the availability of resources to obtain actual traces and run the full-system simulation environment. The choice of topology, a ring (instead of a more reasonable 8x8 torus) is because we want to study network congestion, and a ring is more prone to congestion than a 2D torus—for the same number of nodes.

Trace-based and full-system simulation were performed using the A class of NAS Parallel Benchmarks [11] (NPB), a well-known, allegedly representative set of parallel application workloads often used to assess the performance of parallel systems. The details of the full-system simulation were as follows. The system was composed by 64 Intel Pentium-4 processors, running at 200 MHz (1 Simics cycle = 5 ns), with 64 MB of RAM. The slice is 200 INSEE cycles, equivalent to 1000 Simics cycles; this results in a link bandwidth of 1280 Mb/s, approximately the speed of a Gigabit Ethernet.

We can see in Fig. 4 the relative average times predicted by trace-driven simulation and the unexpected results in our full-system simulator. 99% confidence intervals are also plotted. These were much better than those predicted by traces.

We analyzed the causes of this mismatch, and found it in some undesirable interactions between host-based congestion control (performed by the TCP implementations at the hosts, see Figure 2) and the network based congestion control we were evaluating, in this case IPR. Our trace-based simulation did not include TCP, so these interactions were not taken into account.

Without IPR, application execution times were negatively affected by TCP's wrong estimation of buffers, retransmissions and slow start protocol [6], that made the whole execution very slow in saturated networks. In contrast, when IPR was applied, jitter was reduced, and this helps TCP to determine its timeout and buffer values, so there were less retransmissions and slow starts.

Thus, IPR caused two overlapped effects: (1) For most applications, it accelerated the flow of packets through the IN. (2) In all cases, it helped TCP, allowing the applications to reach higher throughput. This second effect was unexpected, and was way more significant than the first, explaining the mismatch in our predictions.

We can conclude that the reutilization of components 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.

As we explained in the previous sections, another issue when gluing together two different simulators is to fine-tune the synchronization among them. In particular, we need to define the slice duration. We ran some additional experiments to explore this issue. The IN was the same described before and in [12], using again MPICH/TCP/IP/Ethernet as the protocol stack. However, we used a slower network. Results are shown in Fig. 4. In the first row there were a set of experiments tuned to run the benchmarks executing 200 INSEE cycles per 10000 Simics cycles. Experiments on second row were tuned to run 20 INSEE cycles per 1000 Simics cycles – meaning that simulators synchronized 10 times more often, but the network speed was 128Mb/s in both cases because they kept the same relation between Simics cycles and INSEE cycles in every step of execution.

This difference in synchronization frequency had an impact on obtained results, due to 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 before being injected into the network. This resulted in significant delay variations. In the 1000:20 experiments, the worst-case additional delay was reduced to 999 Simics cycles – thus jitter was reduced too. We already know that TCP is very sensitive to jitter, and we observed it in the results. 10000:200 experiment rows took much more simulated time than their analogous 1000:20 experiments.

It is important to point out that 10000:200 experiments were faster (about 5 times) in terms of actual time (not simulated time) because they synchronize less often. But the price to pay for being faster was a lower fidelity in timing, which resulted in a worst performance of TCP.

Despite all problems listed above, our full-system simulator has allowed us to validate, after careful fine-tuning, our expectations about the performance of congestion control, that were tested previously with other methods like synthetic traffic or trace-based simulation – techniques that are significantly faster but are considered less accurate.

Slice duration	BT	CG	IS
10000:200 base	4.64s	5.96s	4.21s
1000:20 base	2.51s	2.83s	2.87s

Table 1. Simulated execution times with different set of synchronization parameters.

7. Conclusions

Full-system simulation is a very complex issue, more so when trying to simulate not a computer, but a collection of networked machines – especially if the network and the interfaces differ from the traditional LAN devices available from simulation environments. It is also a very resource-consuming task. The simulation of a cluster of computers may require an actual machine with similar characteristics to the one under study, and the execution of applications will be several orders of magnitude slower.

As we have shown, full-system simulation of multicomputers requires a large collection of interrelated (software) components, which have in many cases 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 compute nodes and IN simulators also requires a very careful design. A trade-off has to be found between execution speed and simulation fidelity.

Our experience has shown that there are many factors that can interfere in the quality of results: selection of protocol stacks, including MPI implementation, drivers, synchronization modules… In fact, there are so many of these, that sometimes is almost impossible to detect the isolated effects of a given architectural proposal.

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] The Chaotic Routing Project at the U. of Washington. “Chaos Router Simulator”. Available (May 2007) at: http://www.cs.washington.edu/research/projects/lis/chaos/www/chaos.html

[3] 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.

[4] N. Hardavellas et al., “SimFlex: A Fast, Accurate, Flexible Full-System Simulation Framework for Performance Evaluation of Server Architecture,” ACM Sigmetrics Performance Evaluation Rev., v. 31, n. 4, Mar. 24, pp. 31-35.

[5] 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.

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

[7] P.S. Magnusson, et al. “Simics: A full system simulation platform”. IEEE Computer, 35(2):50–58, February 2002.

[8] 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.

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

[10] MPI Forum. “MPICH Home Page”. Available (May 2007) at: http://www-unix.mcs.anl.gov/mpi/mpich/

[11] NASA Advanced Supercomputing (NAS) division. “NAS Parallel Benchmarks” Avail. (May 2007) at: http://www.nas.nasa.gov/Resources/Software/npb.html

[12] 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

[13] 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.

[14] 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.

[15] V. Puente, et al. “The Adaptive Bubble router”, J. of Parallel and Distributed Computing, v. 61, n. 9, pp.1180-1208 Sep. 2001.

[16] F.J. Ridruejo, J. Miguel-Alonso. “INSEE: an Interconnection Network Simulation and Evaluation Environment”. LNCS, Vol. 3648 / 2005 (Proc. Euro-Par 2005), pp. 1014 - 1023.

[17] SMART group at the U. of Southern California. “FlexSim 1.2”. Available (May 2007) at: http://ceng.usc.edu/smart/FlexSim/flexsim.html

[18] M. Thottethodi, A.R. Lebeck, S.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