Evaluation of Interconnection Networks Using Full-System Simulators: Lessons Learned

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

Abstract

In this paper we show the difficulties encountered when performing full system simulation of a distributed memory parallel system. To illustrate the problem, we have chosen a workbench that evaluates the impact on application performance of some simple congestion-control mechanism that can be implemented in the interconnection network. Applications of choice are some of those included in the NAS Parallel Benchmarks. Running a full-system, execution-driven simulation that combines Simics with an interconnection network simulator, we observe some unexpected, negative interactions of overlapping congestion control techniques implemented at the network level and at the host level.

Each MPI implementation uses a different protocol stack, and some of them work without TCP. We compare and contrast results obtained with MPICH, LA-MPI and LAM..

1. Introduction

Full-system simulation has been embraced by hardware and software producers as an invaluable help in the development of new products. For example, extensive tests of an operating system implementation for a yet-to-be released machine, and the applications running on top of it, can be done in a simulated system, while obtaining accurate performance measurements. Virtutech's Simics [19] is one of the most prominent examples of this kind of simulators; this product is able to simulate PowerPC, MIPS, Alpha, ARM, IA-64, SPARC V9, x86, and AMD64 architectures. IBM has a full-system simulator for the Cell Broadband Engine™ (Cell BE) processor, codenamed Mambo **[6]**. And many others are available for different architectures and processors.

In the research area, these simulators have been adopted with enthusiasm. In the scientific literature we can find plenty of references to research done with these tools – just see the list made available by Virtutech[1]. In fact, it has become common by conference reviewers, when examining a simulation-based research work, to ask for a full-system simulation of the systems under study, including large parallel or distributed systems – in many cases, even ignoring the high cost of this kind of simulations (processors, memory, execution time, etc.)

In our experience, full-system simulation is not an appropriate tool to evaluate the performance of a large parallel system, because the resources required would be even larger than the system under evaluation. Thus, to study the complete system we need to simulate a simplified model. The detailed view of the system, essential to fine-tune pieces of hardware and software can be done only for a limited number of nodes. In [1], where the design of the torus interconnection network of the BG/L is described, we can read:

"Given the complexity and scale of the BG/L interconnection network, having an accurate performance simulator was essential during the design phase of the project (...) We also recognized the difficulties in developing an execution-driven simulator (...) for a system with up to 64K nodes. We therefore decided upon a simulator that would be driven primarily by application pseudocodes, in which message-passing calls could be easily passed to the simulator".

Another important requirement of a full-system simulation is, precisely, that it requires the simulation of all the aspects of a system. When we are designing an element that has to be implemented in hardware we require modeling and programming many pieces of software. Let us take a network interface card (NIC) as an example. We need:

- A model of the NIC

[1] http://www.virtutech.com/about/research/selected-pubs.html

- A driver for the operating system to interact with the NIC
- If the new NIC does not behave as a standard network interface (for example, an Ethernet interface) and we want to bypass the operating system and/or the TCP/IP protocol stack, we also need a new protocol stack and API
- If we have an application (or middleware) for which we want to take advantage of the new NIC, we need to modify it to use the new API

Of course, all these pieces have to be carefully crafted and debugged, including their interactions. At the end, when we are doing an experiment on performance evaluation, it is extremely difficult to determine if an unexpected bad result is due to a bad NIC design, a buggy driver, or some unanticipated driver / operating system interactions.

An additional issue comes into the light when comparing systems. In the previous example, if we introduce a design variation into the NIC, and we keep the rest of the software layers unmodified, this variation may result in unexpected interactions that can hide or exaggerate its impact on performance.

Part of our research work include the evaluation of interconnection networks (INs) and, to that purpose, we modeled a switch and a NIC and use these elements in a Simics-based environment. In order to make fair comparisons with other alternatives, and to minimize the number of pieces of software to program (thus, also reducing the opportunities of introducing bugs or, simply, bad programs) we reused as many modules as possible. To that extent, our NIC behaved as an Ethernet NIC. Any MPI implementation with support for IP (most, if not all, available ones include this support) could then use our network. As an example of the kind of evaluations that could be done in this environment, we chose to compare different congestion-control techniques.

Many computing clusters use TCP/IP to run MPI applications, because they work “out of the box”, without acquiring a dedicated network or using new protocol stacks. This reinforced the idea of having a NIC which behaves as an Ethernet card. In the literature we can find many research works assessing the performance of applications running on top of MPI implementations based on TCP/IP. Conclusions of these works include:

- The software overhead associated to TCP/IP is very high, and good performance can be achieved by using alternative, lightweight protocols [2]
- Performance measurements (available bandwidth, delay) on TCP-based networks are highly variable, so that average values may not be of much interest [5].
- Good performance on a TCP/IP-based MPI implementation can only be obtained after fine-tuning the Ethernet drivers and the TCP parameters [18].

What apparently was a good idea resulted in a nightmare, because unexpected interactions between the congestion control mechanisms implemented in the simulated IN (those we were evaluating) and those implemented by TCP or user-level protocols. In this paper we report this experience, and the conclusions learned from it. In particular, we use a Simics-based environment, combined with an in-house developed IN simulator to evaluate the impact on performance of some congestion-control strategies. A collection of parallel applications are run on Linux-based (simulated) machines that interface with the IN using an Ethernet-compatible interface. Different MPI implementations and protocol stacks are compared, in order to assess the impact of these set-ups in the performance of the applications.

The rest of the paper is organized as follows. In section 2 we define some terms, required to fully understand this work. In Section 3 the experimental set-up is defined. In Section 4 we explain the results of our experiments, for each MPI implementation. Section 5 analyzes and summarizes these results. Finally in section 6 we enumerate the conclusions of this work, together with some follow-on research lines.

2. Some definitions

In this section we define two congestion control techniques that are used in our experiments and evaluations. The reader should note that the main focus of this paper is *not* on congestion control. Techniques such as IPR and LBR, defined in the next paragraphs, are *only* used to illustrate the difficulties of doing a full-system simulation while trying to evaluate the exact bearing on performance of a change in the IN behavior.

Congestion appears in a network when compute nodes impose a load close to that the network is able to cope with. As new packets fill up the router buffers, the router will not only stop accepting additional traffic, but will also delay any in-transit traffic. Although congestion may emerge in localized areas, if injection pressure is not reduced it may quickly spread through the whole network. Some networks are able to operate at maximum capacity (in saturation) while exhibiting only minor levels of congestion. However, many others show signs of throughput degradation at loads beyond their saturation points [7].

As congestion appears when the network is overloaded, congestion control techniques deal with it

by limiting packet injection as soon as the network exhibits signs of being congested. There are different ways of diagnosing the apparition of congestion and dealing with it, see for example [9]. In this work, we focus only on two simple methods, based on information locally available at the routing elements that showed good performance with minimal implementation costs:

- In-transit Priority Restriction (**IPR**). For a given fraction *P* of cycles, priority is given to in-transit traffic, meaning that, in those cycles, injection of a new packet is only allowed if it does not compete with packets already in the network. P may vary from 0 (no restriction) to 1 (absolute priority to in-transit traffic). This is the method applied in IBM's BG/L torus network [1].
- Local Buffer Restriction (**LBR**). Most routers split each physical link into several virtual channels (see next section) in such a way that the combination of an escape sub-network with one or more adaptive sub-networks provides deadlock-free adaptive routing [3]. The LBR mechanism has been designed specifically for adaptive routers that rely on Bubble Flow Control [15] to avoid deadlock in the escape sub-network. A previous study showed the bubble restriction also provides congestion control for the escape sub-network [7]. LBR extends this mechanism to all new packets that enter the network. That is, a packet can only be injected into an adaptive virtual channel if such action leaves room for at least *B* packets in the transit buffer associated to that virtual channel. The parameter *B* indicates the number of buffers reserved for in-transit traffic. In other words, congestion is estimated by the current buffer occupancy.

The parameters (*P* for IPR and *B* for LBR) allow us to vary the degree of restriction. In this work, we have set these parameters to values that, in previous experiments, proved to perform well in most scenarios: *P*=1 (the maximum) and *B*=3 (inject in an adaptive channel only when its queue is empty or almost empty). Note that the adaptive bubble router in [15] corresponds to the "Base" case of *B*=0 and *P*=0.

3. The experimental set-up

Experiments have been performed using the in-house developed Interconnection Network Simulation and Evaluation Environment (INSEE for short) described in [16]. It consists of two main modules: an interconnection network simulator (FSIN), and a traffic-generation module (TrGen), which either provides traffic (synthetically or from traces) or interfaces with a Simics-based full system simulation environment [19]. In our set-up Simics performs the full system simulation of the nodes and, when a packet needs to be interchanged between two nodes, this interchange is actually simulated by FSIN.

Resource availability restrictions limit our ability to make full-system simulations to multicomputers of up to 64 nodes. As our experiments deal with congestion control, we have chosen a 1D ring IN topology, more prone to congestion problems than others with higher degrees of connectivity (such as 2D torus, which would be more adequate for a real system). The router model for the network is depicted in Fig. 1.

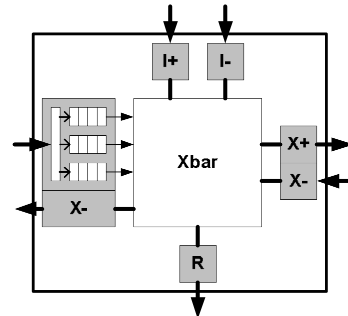

Fig. 1. Model of router simulated by FSIN for 1D networks, with a detailed view of the X+ input port showing the 3 virtual channels that share its link

Each network channel in the router is split into three virtual channels (VCs): an Escape channel (governed by the bubble routing rules [15]), and two adaptive channels. Note that a ring network has just one minimal path from source to destination, so 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.

Each instance of Simics can simulate a variety of real hosts, both hardware and software including the operating system. We simulate a cluster of 64 Intel Pentium-4 processors, running at 200 MHz (1 Simics cycle = 50 ns), with 64 MB of RAM. Every 8 nodes are simulated using a single instance of Simics. Nodes run a RedHat 7.3 operating system, and are configured to use some implementations of MPI: MPICH [12], LA-MPI [10] and LAM/MPI [11].

The link bandwidth of FSIN routers is 32 bits by FSIN cycle. Interaction between Simics and FSIN has

been adjusted to run 200 FSIN cycles per 10000 Simics cycles. Thus, the bandwidth of links is 128 Mb/s.

We have selected these MPI implementations because of the different protocol stack and methods they use to perform the parallel communication, and because they are of widespread use. All of them support several protocol stacks depending on the underlying interconnection network, for example Ethernet, Infiniband or Myrinet. As we use an Ethernet-like simulated NIC, we use MPICH with the P4/TCP/IP/Ethernet protocol stack, which uses the TCP mechanisms to recover lost messages and for congestion control. LA-MPI supports two possible configurations on top of Ethernet, one using TCP/IP/Ethernet and another with UDP/IP/Ethernet. We have chosen the UDP one to evaluate a protocol stack without TCP. In LA-MPI over UDP, flow control and error recovery is done in user space, and not in kernel space (as is done TCP-based stacks). LAM also supports UDP and TCP communication. For the reasons we explain in the next paragraph, we have selected an UDP-based stack. In this case, LAM works by launching a communication daemon at each node, which is in charge of all the interchanges of messages; these daemons communicate using UDP. These protocol stacks are summarized in Fig. 2.

	MPICH	LA-MPI	LAM
User space	NPB	NPB	NPB
User space	P4	Error recovery	Daemon Flow control & error recov.
Kernel space	TCP Flow control & error recovery	UDP	UDP
Kernel space	IP	IP	IP
Kernel space	Ethernet	Ethernet	Ethernet

Fig. 2. Protocol stacks used in the MPI implementations used in the experiments

Our IN simulator does not lose packets. However, when there is congestion in the network, latencies grow. Protocols such as TCP use timers to detect errors and congestion. When a timeout for a packet is triggered, the protocol assumes that the packet is lost, probably due to congestion. The packet is re-transmitted and a flow-control mechanism that restricts the injection of new packets is activated, in order to alleviate the congestion. If the interconnection network itself implements its own congestion-control mechanism, interferences with TCP are unavoidable. For this reason we selected, when possible, protocol stacks without this protocol.

Fig. 3 depicts all the elements that interact in our full-system simulation environment. We will now explain these interactions.

The MPI application of choice (one of the NAS parallel benchmarks) is launched in the first Simics node, which then launches the application on the rest of nodes. Whenever a node wants to send a *message* to another, it passes through several software layers to build one or many adequately formatted *packets* for each message. First, the message is segmented and encapsulated in the kernel by a protocol stack, in example TCP/IP/Ethernet. Then the driver of the network interface card (NIC) injects the generated packets into the network interface card that, in turn, injects the packets into the network. In our simulated environment the NIC is a Simics software module that models a DEC21143 fast Ethernet. We have implemented a Traffic Manager module into Simics that intercepts the sending packets from the NIC and send them (using a real network) to our network simulator (INSEE).

Packets are received by the TrGen module in INSEE that is in charge of providing an interface with the network simulator (FSIN). TrGen puts received packets into its injection queues at FSIN routers. Then, FSIN simulates the way packets travel through the network, and delivers them to its destination router. When this happens, the packet is sent back to TrGen, which sends the packet to its Simics destination node. The packet is received and queued at the Traffic Manager module, and will leave this module to be injected at the simulated DEC21143 card.

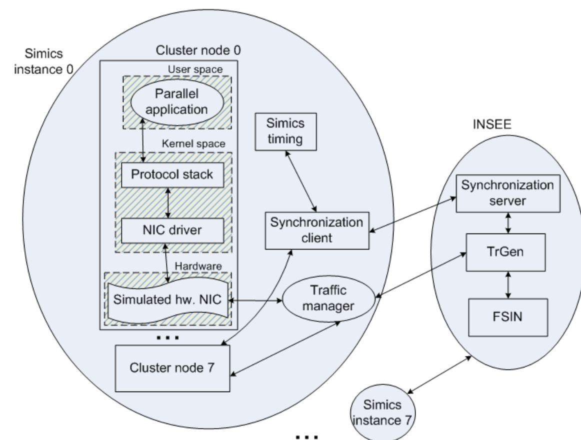

Fig. 3. Elements of our full-system simulation environment that simulates an MPI application running on top of an INSEE (simulated) network

Once the packet is injected at the simulated NIC, this arrival will cause an interruption that will be attended by the NIC driver, already in the kernel space

of the host. The driver will process the packet and send it to the kernel protocol stack that will remove the several protocol envelopes that the packet has around the data which the parallel application awaits.

The synchronization among all nodes and the IN is done at two levels. The first one is the synchronization within each Simics instance that runs 8 nodes. This synchronization is done using a round-robin mechanism: each node executes a determined number of cycles, then the next node and so on. This is provided by the own Simics itself, using a "timing" module.

The second level of synchronization is among Simics instances and INSEE, in a lock-step fashion, using a client-server model. Each instance of Simics includes a synchronization client, and INSEE includes a synchronization server. The client allows Simics instances to run for a pre-defined number of cycles, and then sends a "timestamp" signal to the server. When the synchronization server has received the signals from all the clients, INSEE runs for a number of FSIN cycles, before sending, via multicast, a "continue" signal to all the clients – which allows the Simics instances to resume its execution.

With this synchronization mechanism, all packets that enter from a node into the INSEE are stored in a queue, waiting until the next scheduled run of FSIN to be injected into it. However, packets that arrive to a node are injected at that precise moment into the NIC, but are not processed by Simics until its next scheduled run.

4. Experiments

We completed all the experiments using some of the NAS Parallel benchmarks [13] as target applications. We have chosen three representative benchmarks: CG, BT, IS, with the standard size A. MG is not included in the study because it behaves like CG. The same occurs with LU and SP, that behave like BT. It is important to know that the IS benchmark makes intensive use of the collective operation MPI_Alltoall.

In all figures, execution times are not represented as they are, but are normalized to the base case (without congestion control). This is because times of each benchmark are remarkably different. For each experiment, we plot the mean value together with the 99% confidence level intervals.

4.1. Traces

Before performing full-system simulations, we have used other way of dealing with actual traffic from applications: trace-driven simulation. For this, traces of some NPB applications are obtained in a real system (no simulation is involved) using a modified version of MPICH that shows all point-to-point operations involved in the MPI collectives. With them, we feed our simulator taking into account only the causal relationship of the messages, so all CPU events are discarded. This traffic generation methodology emulates infinitely fast processors, so that the network is the bottleneck of the system. Thus, we can obtain an optimistic estimation of the advantages of congestion control techniques (or any other characteristic of the IN under evaluation). It is "optimistic" in the sense that, in an actual system, processes do not only communicate, but also compute, so that the actual utilization of the network is less intensive, and congestion is less prone to appear.

We can see the results obtained for both congestion control mechanism in Fig. 4. Both congestion control mechanisms are beneficial for BT and IS, because their traffic patterns consist of large messages that, when decomposed into packets, are able to congest 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.

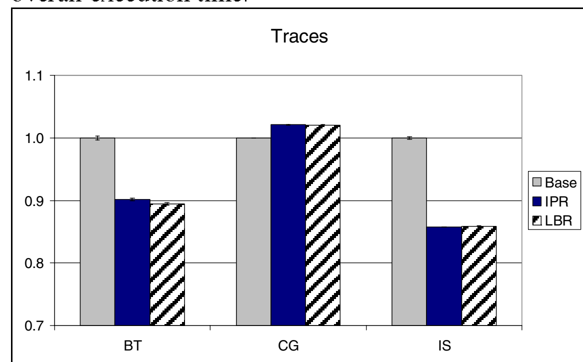

Fig. 4. Results from NPB traces using INSEE trace-driven simulation. Relative times to complete a run of benchmarks BT, CG and IS, and 99% confidence intervals. Base time (1.0) corresponds to the run with congestion control deactivated

4.2. MPICH

Our first attempt to obtain results from a full-system simulation was using the MPICH implementation of MPI. This is one of the most widely used MPI implementations. The traces described above were obtained using this implementation, so we expected the results to be similar but attenuated compared to those described in the previous section, because actual

applications include computation phases in which the network is not used, so that congestion is less prone to appear. However, this is not the case, as we can see in Fig. 5. What we see is that BT does not experiment any significant change in performance when using IPR or LBR, and that CG and IS benefit from congestion control at the network level.

When we analyzed the reasons of this behavior, we discovered some unexpected interactions between TCP congestion control and those implemented in the network. When IPR or LBR is activated, the latency values provided by the network are in a smaller range than in the base case (in other words, there is less jitter). TCP can, for this reason, estimate values for its timeouts more accurately. In the base case, when the estimation of timeouts is not very precise, there are many packet retransmissions, TCP's mechanism to deal with congestion (basically, slow start) are activated very often, and actual throughput is exceedingly poor. When we help TCP by reducing the jitter, these mechanisms are triggered much less often, and the applications observe a higher network throughput. This effect is clearly visible in CG and IS, because both applications make intensive use of the network (the computation phases are short). In contrast, as BT is more CPU-intensive, the interaction between congestion control mechanisms is hidden by its low network utilization.

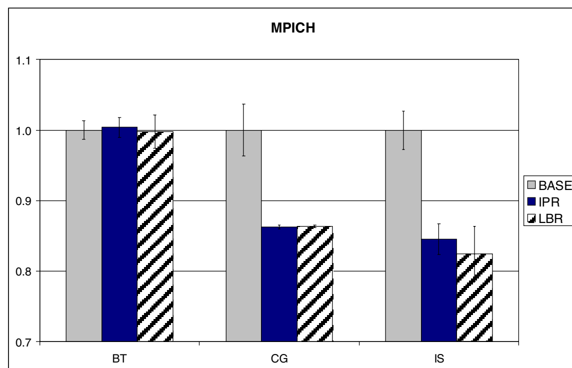

Fig. 5. Results of the full-system simulation using MPICH

This adverse effect could be reduced if we used an optimized version of TCP instead of the slow-start algorithm for flow and congestion control implemented in the "standard" TCP (often called "Reno TCP" [8]). There are a variety of TCP versions that implement flow and congestion control algorithms with a more efficient management of the transmission window, designed for modern networks with faster speeds and more reliable links. Some examples are HighSpeed TCP [4], Vegas TCP [14], H-TCP [17] and BIC TCP [20]. The most recent Linux kernels include some of these variations of TCP; however, our experimental set-up was done with a Red Hat 7.3 distribution, whose kernel did not allow us to experiment with TCP. A kernel upgrade could have been done, but they would make simulations slower or even unstable. We opted for abandoning TCP-based implementations of MPI, which included MPICH.

4.3. LA-MPI

Once the decision of avoiding TCP was taken, we searched for another (free) MPI implementation providing support for MPI directly over Ethernet, but we did not find any. However, we found LA-MPI, which provides an implementation of MPI over UDP (over Ethernet). Since UDP does not implement error and congestion control, and its implementation over IP/Ethernet is very simple (it basically adds some headers to the Ethernet frames), we repeated our experiments running the same benchmarks.

LA-MPI performs an application level error control to avoid packet losses. A packet may be dropped when intermediate buffers are full. We had to tune the "RETRANS_TIME" parameter in the source code of LA-MPI in order to modify the maximum time a message sender has to wait for the ACK before retransmitting it. The standard value for this parameter is 10 s.; we changed it to 1 s. in order to accelerate retransmissions.

We can see the results in Fig. 6. They are similar to those from the traces, but slightly smoothed. This is true for all the benchmarks, although in some cases the confidence intervals overlap. Thus, LA-MPI looked like a good platform for experimentation. However, this project was abandoned some time ago and it is no longer supported, so we decide search for another MPI implementation over UDP/IP.

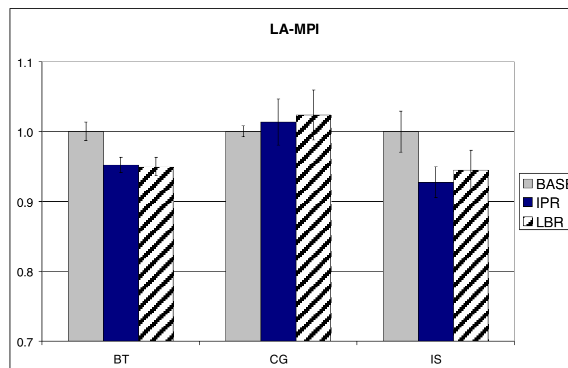

Fig. 6. Results of the full-system simulation using LA-MPI

4.4. LAM/MPI

LAM/MPI is also a very well-known and widely used MPI implementation. It can work in different modes: direct TCP communication among MPI processes, or indirect communication via LAM daemons (*lamd*). This module creates a daemon in every node which is in charge of all communications and uses UDP/IP. As related in the LAM User Guide:

"Rather than send messages directly from one MPI process to another, all messages are routed through the local LAM daemon, the remote LAM daemon (if the target process is on a different node), and then finally to the target MPI process. This potentially adds two hops to each MPI message".

So latencies are larger and messages could be lost when the network is congested. This indirect version of LAM looked like a viable option to experiment without dealing with TCP interferences. But initial runs using LAM were *very* slow, because of the *lamds*. When their buffers are full, instead of using any kind of backpressure to tell the processor to stop injection, they drop the messages they can not store. The dropped messages are resent only when a timeout is triggered. We modified the LAM_TO_DLO_ACK parameter in LAM from 0.5 s to 0.5 ms. This parameter defines the maximum time to wait for an ACK from a node that should have received a message; if this time expires, the message will be retransmitted. We can see the results after this modification in Fig. 7.

In summary, LAM implements, at the application level, an error recovery mechanism similar to TCP that, again, mixes-up congestion control and error control – and, again, interferes with congestion control applied at the network.

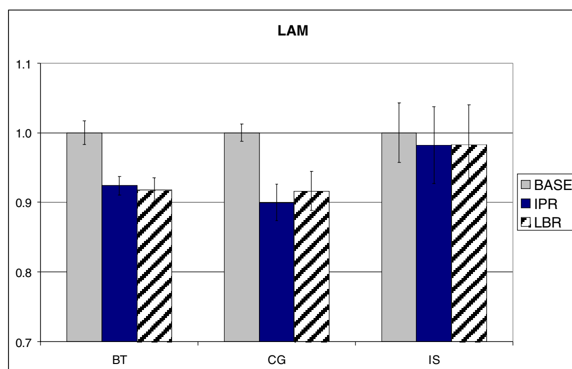

Fig. 7. Results from Simics using LAM/MPI

5. Analysis of the results

Our search for a full-system simulation environment to evaluate our proposals in the field of IN design has been far from successful. These environments are complex, because they include many elements: applications, operating systems, protocol stacks, drivers, and so on. Trying to keep things simple we decided to reuse as many elements as possible, and this proved to be a bad decision. Focusing on MPI implementations, we have seen that the most used ones, which work on top of TCP or UDP, are not adequate for measuring the performance of INs, because these protocols were developed for other kind of networks (the Internet), whose needs are different.

TCP includes end to end flow and congestion controls that are very sensitive to jitter. Our hardware-implemented congestion control mechanism provides a more stable value of packet latency. This maintains TCP transmission window more stable, providing a better transmission rate. When running applications on MPICH over TCP, the utilization of network-based congestion control resulted in execution times much shorter that those obtained without this control.

If TCP is not used, we find that the MPI middleware implements its own versions of error control and congestion control, so we are not free from interactions with overlapping mechanisms implemented at the network. Experiments with LA-MPI shows how a lightweight MPI implementation on top of UDP (that is, almost directly over the network interface) is able to expose the characteristics of the network. Unfortunately, LA-MPI can be considered obsolete. LAM/MPI also work on top of UDP, via a collection of communicating *lamds*, but this way of working is very inefficient and should not be used in actual, production work. Execution times reported by our simulator for each benchmark (Table 1) clearly show that LAM/MPI using *lamds* is the worst performer of the studied MPI implementations.

Table 1. Average number of FSIN cycles needed to run an iteration of each benchmark

	BT	CG	IS
MPICH	1806022	2355266	1684316
LA-MPI	1805120	2238120	1639150
LAM/MPI	**2034136**	**4350458**	**4151681**

6. Conclusions and future work

The utilization of full-system simulation environments may be far from adequate when testing non-existent networks. A complete evaluation would require the simulation of the hardware plus drivers, libraries, MPI implementations and even applications.

The reutilization of common pieces of software allows, theoretically, for a fairer comparison but may be influenced by unexpected side-effects. The implementation of ad-hoc protocol stacks may be more

adequate to test the potential of a given solution but, when comparing two alternatives, it would be difficult to find where the performance differences may lie: hardware or any piece of the software.

In this paper we have evaluated the effects of network-level congestion control techniques on the performance of MPI applications, using a variety of MPI implementations. We have observed that this form of congestion control results in lower latencies and jitter – as expected. This behavior has a side-effect not foreseen: it makes TCP behave better, which results in great performance gains when using protocol stacks with this protocol. Ideally, TCP should be put out of the protocol stacks but, as we have shown, this is not an easy task: publicly available implementations of MPI make use of this protocol, or implement their own congestion or flow control mechanisms at the application level.

For the future, we need to take a modular implementation of MPI, such as MPICH, and implement a module that provides MPI support directly on top of our simulated network. The result will be an evaluation environment with less interference. However, this module has to be done with enormous care because, otherwise, it could be the cause of unexpected performance results.

Acknowledgements

This work has been done with the support of the Spanish Ministerio de Educación y Ciencia, grant TIN2004-07440-C02-01. Mr. J. Navaridas is supported by a doctoral grant of the UPV/EHU.

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] A.F. Diaz, J. Ortega, A. Canas, F.J. Fernandez, A. Prieto, "The Lightweight Protocol CLIC: Performance of an MPI implementation on CLIC," cluster, p. 391, 3rd IEEE International Conference on Cluster Computing (CLUSTER'01), 2001.

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

[4] S. Floyd. “HighSpeed TCP for Large Congestion Windows”, RFC3649, experimental, December 2003

[5] D.A. Grove and P.D. Coddington. “Communication Benchmarking and Performance Modelling of MPI Programs on Cluster Computers”. International Journal of Supercomputing, vol. 34, 201-217 (2005).

[6] IBM. “IBM Full-System Simulator for the Cell Broadband Engine Processor”. Available (nov. 2006) at http://alphaworks.ibm.com/tech/cellsystemsim

[7] 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.

[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, vol.4 no.3, pp 24-30, May 1990.

[10] LA-MPI Home Page “The Los Alamos Message Passing Interface” Available (Oct. 2006) at http://public.lanl.gov/lampi/

[11] LAM/MPI Home Page “LAM/MPI Parallel Computing.” Available (Oct. 2006) at http://www.lam-mpi.org/

[12] MPI Forum. “MPICH Home Page”. Available (Oct. 2006) at http://www-unix.mcs.anl.gov/mpi/mpich/

[13] NASA Advanced Supercomputing (NAS) division. “NAS Parallel Benchmarks” Available (Oct. 2006) at http://www.nas.nasa.gov/Resources/Software/npb.html

[14] S. W. O’Malley, L. S. Brakmo, L. L. Peterson. “TCP Vegas: New Techniques for Congestion Detection and Avoidance”, SIGCOMM, 1994

[15] V. Puente, C. Izu, J.A. Gregorio, R. Beivide, and F. Vallejo, “The Adaptive Bubble router”, Journal of Parallel and Distributed Computing, vol 61, no. 9, pp.1180-1208 September 2001.

[16] 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.

[17] R. N. Shorten, D. J. Leith. “H-TCP: TCP for High Speed and Long Distance Networks”, Proc. PFLDnet, Argonne, 2004

[18] D. Turner and X. Chen. “Protocol-Dependent Message-Passing Performance on Linux Clusters”. Proc of Cluster 2002, Chicago, September 25th, 2002.

[19] Virtutech Inc. “Simics page”. Available (Jan. 2006) at http://www.virtutech.se/products/

[20] L. Xu, K. Harfoush, I.Rhee. “Binary Increase Congestion Control for Fast, Long Distance Networks”. IEEE INFOCOM, March 2004.