Full Circle: Simulating Linux Clusters on Linux Clusters

Luis Ceze[1], Karin Strauss[1], George Almasi[2], Patrick J. Bohrer[3], José R. Brunheroto[2], Calin Caşcaval[2], José G. Castaños[2], Derek Lieber[2], Xavier Martorell[2], José E. Moreira[2], Alda Sanomiya[2], and Eugen Schenfeld[2]
{luisceze,kstrauss}@uiuc.edu
{gheorghe,pbohrer,brunhe,cascaval,castanos,lieber, xavim,jmoreira,sanomiya,eugen}@us.ibm.com

[1] Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801
[2] IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218
[3] IBM Austin Research Laboratory
Austin, TX 78758

Abstract. `BGLsim` is a complete system simulator for parallel machines. It is currently being used in hardware validation and software development for the BlueGene/L cellular architecture machine. `BGLsim` is capable of functionally simulating multiple nodes of this machine operating in parallel. It simulates instruction execution in each node and the communication that happens between nodes. `BGLsim` allows us to develop, test, and run the exactly same code that will be used in the real system. Using `BGLsim`, we can gather data that helps us debug and enhance software (including parallel software) and evaluate hardware. To illustrate the capabilities of `BGLsim`, we describe experiments running the NAS Parallel Benchmark IS on a simulated BlueGene/L machine. `BGLsim` is a parallel application that runs on Linux clusters. It executes fast enough to run complete operating systems and complex MPI codes.

1 Introduction

Linux clusters have revolutionized high-performance computing by delivering large amounts of compute cycles at a low price. This has enabled computing at a scale that was previously not affordable to many people. In this paper, we report our experience in leveraging the vast amount of computational power delivered by modern Linux clusters to help us develop the system software for a new machine: The BlueGene/L supercomputer.

BlueGene/L [1] is a massively parallel cellular architecture system being developed at the IBM T. J. Watson Research Center in cooperation with Lawrence Livermore National Laboratory. This machine is based on a large number of PowerPC processors, augmented with a novel floating-point unit, interconnected in a three-dimensional torus network. The processors in BlueGene/L are organized into groups called *processing sets*, or psets. Each pset is under control of one Linux operating system image, so the machine looks like a large (1,024-way) Linux cluster.

Early in the BlueGene/L project we realized the need for tools that would help us develop and test the system software for BlueGene/L in advance of hardware availability. Our experience in previous large system projects had shown that architectural simulation of large machines was an effective exploratory tool. Therefore, we endeavored in the development of an architecturally accurate, full system simulator of the BlueGene/L supercomputer.

A logic level simulator of BlueGene/L was available as a side effect of our VHDL hardware design methodology. VHDL simulations play an important role in the detailed analysis and verification of the design, but they are too slow to be used for simulations of large configurations executing real workloads. It was essential to provide a feasible way to support the development of both system software and end user software, in addition to hardware studies.

Our approach was to develop `BGLsim`, an architecturally accurate simulator at the instruction-set level. `BGLsim` exposes all architected features of the hardware, including processors, floating-point units, caches, memory, interconnection networks, and other supporting devices. The simulator can run exactly the same software that will run in the real machine. This approach allows running complete and unmodified code, from simple self-contained executables to full Linux images. The simulator includes interaction mechanisms for inspecting detailed machine state, providing more information than what is possible with real hardware. By simulating at the instruction-set level, `BGLsim` is several orders of magnitude faster than VHDL simulation at the logic design level.

We have organized `BGLsim` as a parallel application that can run in multiple nodes of a Linux cluster. Each task (process) of this parallel application simulates one BlueGene/L node. The larger the Linux cluster we have access to, the larger the BlueGene/L machine we can simulate. Since BlueGene/L itself is essentially a Linux cluster, we have this situation in which a host (real) Linux cluster is used as the platform for a simulated (virtual) Linux cluster.

To illustrate the capabilities of the `BGLsim` simulation environment, we present an experiment with a parallel application on a multi-node BlueGene/L machine. The application chosen is the NAS Parallel Benchmark IS, an MPI application [10]. We used `BGLsim` to investigate the scalability characteristics of IS, regarding both its computation and communication components.

The rest of this paper is organized as follows. Section 2 compares our work to other simulation efforts that already exist or are being developed. Section 3 provides a brief overview of the BlueGene/L machine. Section 4 explains the structure of our single-node simulator and its capabilities. Section 5 explains how we implement a multi-node system simulator an how nodes communicate with each other. Practical experience with `BGLsim` is described in Section 6. Finally, Section 7 summarizes our contributions.

2 Related work

Augmint [13] is a multiprocessor simulator, intended for performance analysis of parallel architectures. It simulates the execution of multiple processes running on multiple processors in parallel. Its target architecture is Intel x86, which is not appropriate for

the processor we wanted to simulate. Augmint is not a complete machine simulator, so we could not simulate specific BlueGene/L devices. Furthermore, Augmint works by instrumenting code before execution. For our purposes, we needed a system that would support running unmodified kernel and user code, making it easier to debug software, correlate events with the source code, and validate results against detailed VHDL simulations.

SimpleScalar [4] offers a set of various processor simulators, including PowerPC architecture [12]. However, these simulators do not model all the devices in BlueGene/L. Because we wanted to use a simulator to help in the development of kernel level software, SimpleScalar was not appropriate. In contrast, `BGLsim` accurately models all architected features of the BlueGene/L hardware at the instruction-set level. Thus, we can run a complete software stack, including kernel and user level code.

MPI-SIM [11] is a library for execution-driven parallel simulation of task and data parallel programs. It simulates only user level code and its main focus is evaluating the performance of parallel programs. The parallel programs can be written directly using MPI for message-passing, or in UC, a data parallel language, and then compiled to use message-passing. MPI-SIM does not provide access to the underlying layers of communication and interconnection devices. Since we wanted to develop several layers of communication libraries for BlueGene/L, from the hardware layer to the message-passing layer, MPI-SIM was not the complete solution we were looking for.

Some existing simulators, like SimOS [7, 8] and SimICS [6, 2, 9] are complete machine simulators, but do not model the specific processor and devices in our hardware design. Furthermore, the inter-simulator communication scheme (for multi-node simulation) in SimOS did not provide good performance and security. In BlueGene/L, we implemented a different mechanism for inter-simulator communication (see Section 5), which provides less overhead when the communication traffic is high.

We decided to develop our own simulator for BlueGene/L. This simulator is based to a great extent on the Mambo single- and multi-processor simulator developed at the IBM Austin Research Laboratory. Using concepts from other system simulators, particularly SimOS, we have created a simulator that models all the architected features of BlueGene/L, including multi-node operation.

3 BlueGene/L overview

BlueGene/L is a new high-performance parallel supercomputer. It is based on low cost embedded PowerPC technology. The interested reader is referred to [1] for a more detailed description of the architecture. This section presents an overview of the machine as background for our discussion on `BGLsim`, one of its simulation environments.

3.1 Overall organization

The BlueGene/L machine is constructed based on a custom system-on-a-chip building block. This BlueGene/L *node* is composed of processors, memory and communication logic, all in the same piece of silicon. Every chip contains two standard 32-bit embedded PowerPC 440 cores with a custom floating-point unit (described in the next paragraph),

and a memory subsystem. The memory subsystem is composed by an L1 instruction cache and an L1 data cache, as well as an L2 cache for each core, and a shared L3. The L1 caches are 32KB, have 32-byte lines, and are not coherent. The L2 caches are 2KB, have 128-byte lines, are not inclusive (they are smaller than the L1 caches) and are coherent. The L2 caches act as a prefetch buffer to the L1 caches. The two cores share a 4MB L3 EDRAM cache, which is also included in the chip.

Each core is connected to a custom FPU, which is a 128-bit "double" FPU: its two 64-bit "subunits" are driven by the same double instructions, issued by the core and extended from the standard PowerPC floating point instruction set. Each subunit is a conventional FPU with a 64-bit register file of 32 registers. One of these subunits is called primary and the other one is called secondary. The primary side can be used as a standard 440 FPU, executing the complete PowerPC floating point standard instruction set. The two subunits can be used together to perform SIMD-like floating point operations from the extended instruction set. Since each subunit is able to perform multiply-and-add operations, the custom FPU can perform four double precision floating point operations per cycle.

The typical usage scenario for the two processors in a node is to allocate only one of the 440 cores to run user applications; the other core is destined to drive the networks. The target frequency of the machine is currently 700MHz. At this frequency, each core achieves 2.8GFlop/s of peak performance. If both cores are used for user applications, the peak performance per node reaches 5.6GFlop/s.

Multiprocessor support is not provided by the standard PPC440 cores. As described in the beginning of this section, the L1 caches are not coherent. Besides, the standard PPC440 instruction set does not provide any atomic memory operations. In order to provide synchronization and communication capabilities between the two cores, some devices were added to the chip: lockbox, shared SRAM, L3 scratchpad, and the blind device. All devices are configured via registers and used via memory mapped locations. The lockbox unit provides fast atomic test-and-sets and barriers to a limited number of memory locations. The shared SRAM has 16KB and can be used to exchange data between the two cores. The EDRAM L3 cache can have some regions reserved as an addressable scratchpad. The blind device provides explicit cache management. A special interrupt controller is designed and connected in such a way that the cores can interrupt each other.

The BlueGene/L packaging, depicted in Figure 1, is very dense, due to the low power characteristics of the architecture. A compute card contains two of the nodes described above (each node is a chip with two cores) and SDRAM-DDR memory. An external memory of 2GB can be supported by each node, but in the current configuration each node has 256MB of main memory, accessible at a bandwidth of 5.6GB/s and 75-cycle latency. A node board can hold up to 16 compute cards (32 nodes). A cabinet can hold two midplanes, each containing 16 node boards for a total of 1024 compute nodes (2048 processors), with a peak performance of 2.9/5.7TFlop/s per cabinet. The BlueGene/L complete system is composed by 64 cabinets and 16TB of memory.

Besides the 64K compute nodes, the BlueGene/L machine also has 1024 *I/O nodes*. I/O nodes and compute nodes are very similar and built out of the same chip. One of the differences is that the I/O nodes have larger memory (512MB). The enabled networks

for each type of node also varies: in our current implementation, an I/O node has its tree and Ethernet devices active while a compute node has its torus and tree devices active. These networks will be further explained in section 3.2. Each group of 1 I/O node and 64 compute nodes is called a processing set (pset), and it behaves as one Linux machine. The Linux operating system runs on the I/O node, which controls and distributes applications to the compute nodes of the pset.

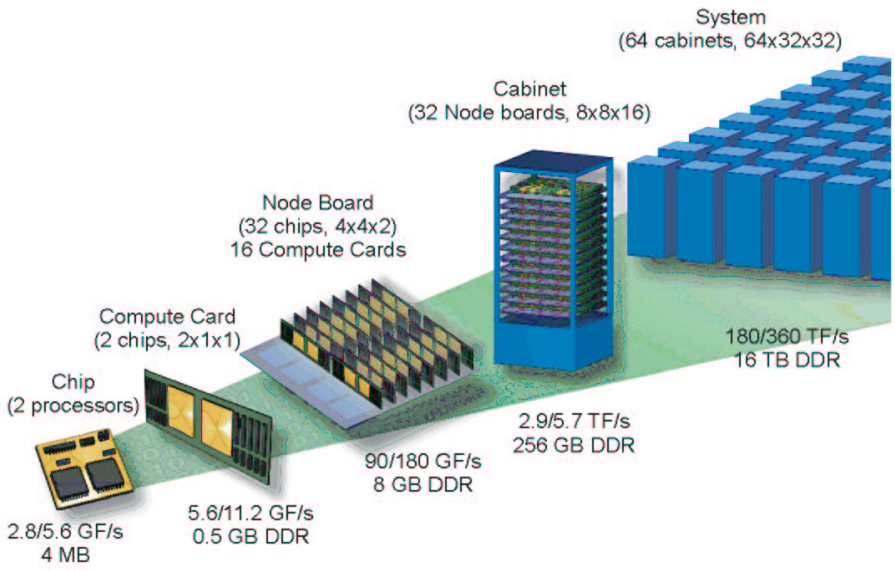

Fig. 1. High-level organization of the BlueGene/Lsupercomputer. All 65,536 compute nodes are organized in a $64 \times 32 \times 32$ three-dimensional torus.

3.2 BlueGene/L communications hardware

A PowerPC processor in a compute (or I/O) node can only address the node's local memory. To communicate with other nodes, a node needs to send inter-node messages. Therefore, all inter-node communication is based on message exchange. The BlueGene/L system-on-a-chip supports five different networks: torus, tree, Ethernet, JTAG and global interrupts. The three-dimensional torus is the main communication network for point-to-point messages. Each node has six links for this network, which are connected to its nearest neighbors. The 64K compute nodes are organized in a 64x32x32 three-dimensional torus, which can be partitioned in smaller units. I/O nodes are not part of this network. The variable length torus packets (up to 256B) have guaranteed reliable, unordered, deadlock-free delivery by a minimal adaptive routing algorithm. It is also possible to use this network to send simple broadcast packets that deposit the

contents of the packet in every node along a route. The bandwidth in each direction of a torus link is 1.4Gb/s, and the bisection bandwidth of a 64K node system is 360GB/s (each direction).

The tree network provides point-to-point, broadcast and reduction operations with packets, with low hardware latency. Incoming packets in a node can be combined using bitwise and integer operations, and the result is forwarded in a new combined packet along the tree. Floating point operations can be performed in two steps (the first to combine the exponents and the second to combine the mantissas). Both I/O and compute nodes participate in this network, which is used for communication between the two types of nodes. The global interrupts are supported by a different set of links that run in parallel with the tree links and provide global AND/OR operations with low latency. This feature was designed to support fast barrier synchronization.

The BlueGene/L machine can be divided into partitions. Each partition is allocated to execute a single job. Partitions are subsets of the machine that are self-contained and electrically isolated from each other. The interconnection of the networks described above between midplanes are routed through "link chips" which are custom chips that can be configured dynamically to go around faulty midplanes while maintaining the networks functional. The "link chips" can also be used to partition the torus network into independent tori, that can be as small as a midplane (512 nodes). It is possible to create smaller meshes (128 compute nodes and 2 I/O nodes in the current packaging) by disabling some of the links on the boundary chips of a partition.

Each node also contains a 1Gb/s Ethernet device for external connectivity, but only I/O nodes are actually connected to an Ethernet network. Therefore, the BlueGene/L machine has $1024 \times$ 1Gb/s links to outside world, used primarily to communicate with file servers.

Finally, each node has a serial JTAG interface to support booting, control and monitoring of the system through a separate control network. The JTAG network is part of this control network, which also includes a control Ethernet network (not the same network that is directly connected to the I/O nodes). This control Ethernet network is connected to one or a cluster of service nodes, and IDo chips, which act as gateways between the control Ethernet network and the JTAG network. The service nodes are commercially available computers that control the rest of the BlueGene/L machine via the control network. Each node board and each midplane contains an IDo chip that receives commands, encapsulated in UDP packets, from the service nodes over the Ethernet control network. The IDo chips can interface with the machine components via a number of serial protocols. The I^2C network is used to control power supplies, temperature sensors and fans. A small EPROM containing the serial number of each core can be read using the SPI protocol. Each chip has 16KB of SRAM, which can be read and written using the JTAG protocol. The JTAG protocol can also be used to read and write registers, and to suspend and reset the cores. The IDo chips have direct access to the hardware and do not rely on any software support in the nodes they control.

4 Single-node simulation

This section describes how a single BlueGene/L node is simulated. Each BlueGene/L node is simulated by one `BGLsim` process. Several communicating `BGLsim` processes compose a virtual BlueGene/L machine. Multi-node simulation will be addressed in Section 5.

`BGLsim` was developed to run in a Linux/x86 environment. Each `BGLsim` process implements a virtual BlueGene/L node that, in turn, can run a custom Linux/PPC operating system image. Figure 2 shows the simulation stack for a single node `BGLsim` running Linux/PPC. At the bottom of the stack we have the host hardware, an x86 machine. We run an image of the Linux operating system in this host hardware and `BGLsim` is an application process on top of this Linux platform. The `BGLsim` process implements a (virtual) instance of the BlueGene/L node hardware. We run an image of the Linux operating system on this BlueGene/L node and test applications are run as processes on top of this second Linux platform.

Fig. 2. Stack when `BGLsim` is running Linux-PPC

4.1 `BGLsim` internals

Figure 3 shows in more detail what is being simulated by a `BGLsim` process. That is, the internals of a BlueGene/L node. The simulation of the processors and the devices exposes to the software running on the simulator an architecturally accurate view of the hardware. This allows the execution of unmodified operating system and application images. The devices are memory mapped and are also accessible for configuration through DCRs (device control registers). When a memory address associated with a device is accessed, the corresponding device model is activated and takes the appropriate actions, like sending a packet to the network or reading the console input buffer.

The JTAG control port (a device in the real hardware) provides a set of basic low-level operations, like read/write of SRAM and DCRs, and stopping/resuming/resetting a processor core. Using this port is possible to precisely control and inspect the state of a `BGLsim` node. Using the functionality of the simulated JTAG control port, it is possible to use the same exact control software infrastructure that will be used in the real machine.

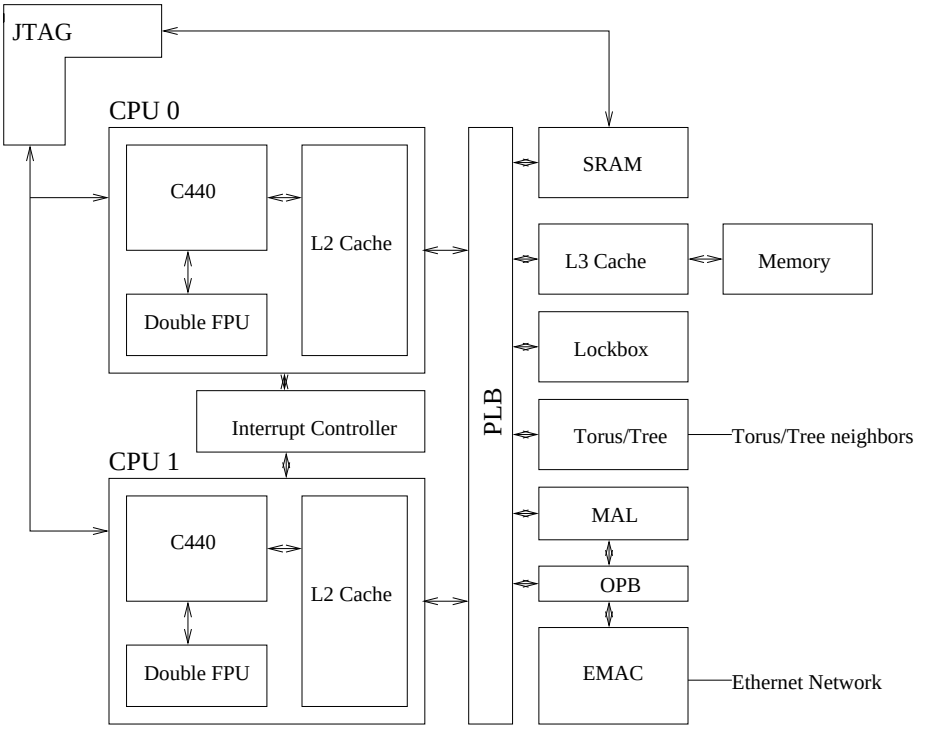

Fig. 3. The internals of a BlueGene/L node.

`BGLsim` includes models for all the cache levels of the memory subsystem. By using these models, it is possible to analyze cache behavior of applications and the OS. The BlueGene/L node has incoherent (between the two processors) L1 caches and we correctly model that cache behavior. This is important to support testing of software that manages the cache coherence explicitly.

Besides providing full system functionality, our simulator also provides networking connectivity. Each simulated node has its own Ethernet MAC address and, when running Linux, its own IP stack. This makes it possible to telnet or ftp into a simulated node either from a real machine or from another simulated node. The torus and tree interconnect devices have also been implemented. We give more details about the interconnection scheme in Section 5.

`BGLsim` is centered around the instruction simulator. This instruction simulator is organized as an infinite loop, shown in Figure 4, with each iteration performing the following steps: instruction fetching, instruction decoding, instruction executing (may include memory operation and register updating), asynchronous devices polling, and timer update. A memory operation is composed by the following sub-steps: address translation, permission checking, and memory access (read or write). When the memory operation accesses a memory mapped device, simulation of that device is activated.

For each instruction executed, BGLsim can generate trace information which provides detailed information of the operation performed. That trace information can be passed to on- or off-line analysis programs, such as the IBM SIGMA tool chain [5].

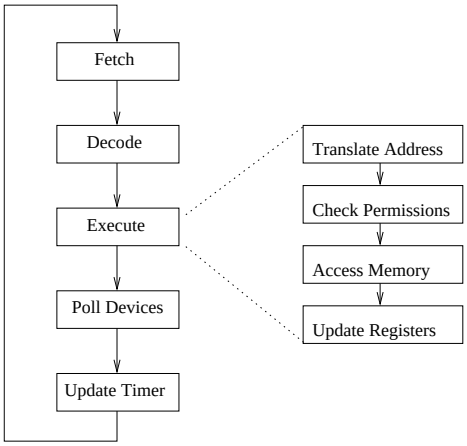

Fig. 4. Main simulation loop of BGLsim.

4.2 Running code with BGLsim

A version of the Linux operating system has been ported to BlueGene/L using BGLsim. This allows executing unmodified applications, collecting statistics, debugging new applications and validating design choices before building the actual hardware. At this time, we have successfully run database applications (eDB2), linear algebra kernels, and codes from the Splash-2, Spec, and NAS Parallel Benchmarks suite.

When modeling a single-node machine with 256MB of memory, BGLsim can simulate up to 2 million BlueGene/L instructions per second when running on a 1GHz Pentium III machine with 512MB of RAM. (Even in multi-node simulation, this speed is still maintained if each node in the host cluster is simulating one BlueGene/L node. More information about multi-node simulation is given in Section 5.) The simulation rate drops to about half when tracing is enabled. In order to reduce the overhead and the trace file size imposed by the trace functionality, we have devised a scheme that allows selective trace acquisition. It is possible to gather trace information selectively for a given process in the Linux image, ignoring activity related to other processes. This selective tracing facility relies on the *call-through* scheme of BGLsim, explained below.

The simulator provides additional services to executing code through a call-through instruction that is not part of the processor instruction set. When the simulator finds this instruction during execution, it invokes its call-through handler. Parameters are passed

to the simulator through registers of the simulated machine and, depending on those parameters, specific actions are performed by the call-through handler. This instruction can be called by any type of program that runs on the simulated machine. Some of the services provided through this mechanism are described below.

- *Host real time clock*: this information is useful when evaluating the simulator itself. It is possible to use this information to measure the time an application takes to run in the simulated environment.
- *Instruction counter*: if an operating system is running on the simulated hardware, this call-through returns the number of instructions already executed in the current process, otherwise it returns the global instruction counter.
- *Trace on/off*: turns tracing information generation on and off. It is possible to specify it to be either global trace or per-process trace.
- *Histogram on/off*: same as above except that the trace information is dynamically summarized into an instruction histogram.
- *Stop/suspend simulation*: the program being run has the ability to stop the simulation, either going to the simulator command line or completely exiting the simulator.
- *Message output to trace*: the program can send strings to be included in the trace file.
- *Environment variables from host*: the program being run can read the value of environment variables from the host. This is useful to parametrize the workload when running several instances of the simulator.
- *Task operation*: informs the simulator that the operating system has switched to a different process, created a new process, or terminated an existing process.

In order to support OS dependent information, we instrumented Linux to inform the simulator about events like context switching and termination of a process. This was done using single instruction call-throughs, in order to have the least interference possible. The parameters of these call-throughs inform the simulator about the action being performed (process creation, context switching, process termination), and the processes involved.

4.3 Debugging support in `BGLsim`

Since the focus of `BGLsim` is on assisting with software development, we have provided extensive debugging support in it. The debugging scenarios possible with this simulator are:

- *Single-instruction step*: It is possible to run a program instruction by instruction and see it disassembled, along with machine state information. This feature is useful for debugging code generation tools, such as compilers and assemblers.
- *Running gdb on simulated machine*: Because the simulator is capable of running unmodified code, it is possible to run gdb on the simulated machine. However, the overhead caused by this process costs time on a simulated machine.

– *Running gdb on a separate machine, with TCP stubs on the simulated machine*: In this case, only a simple stub runs on the simulated machine and an external gdb process that runs on a real machine can connect to this stub through the network. This reduces the overhead caused by the debugging process running on the simulated machine and is appropriate for debugging user code.
– *Kernel debugging:* Kernel debugging can be provided by direct communication between the simulator and debugging tools running on the host machine. We have implemented our own debugging client, that communicates directly with the simulator through a non-architected back door. This same debugger can also be used with user code.

4.4 Validating `BGLsim`

We verify the correctness of our simulator using VHDL simulations. We run the same test cases in both simulators. Both of them give us the same result, differing only in their timing accuracy. Our simulator is functionally accurate, has some timing information, and runs faster. The VHDL simulator is cycle accurate and runs slower. Because our simulator runs much faster than the VHDL simulator, it is appropriate for software development. If we need to get more detailed performance measurements, then the VHDL simulator is used.

5 Multi-node simulation

BlueGene/L is an inherently parallel machine. Therefore, it is vital that `BGLsim` be capable of simulating multi-node configurations, interconnected in the same way that they will be in the real machine. In this section we describe how we implemented multi-node simulation in `BGLsim`.

`BGLsim` runs on a Linux/x86 platform. A multi-node configuration of `BGLsim` is an MPI application, running on an Linux/x86 cluster, simulating a cluster of BlueGene/L nodes (a virtual BlueGene/L machine). Thus, if the `BGLsim` instances are running Linux/PPC, we have a virtual Linux/PPC cluster running on top of a Linux/x86 cluster. When an MPI application is running in BlueGene/L, the simulation stack would look like Figure 5. Again, at the bottom of the stack we have the host hardware, which is an x86 cluster. Each node of that cluster runs an image of the Linux operating system. `BGLsim` is an MPI application that implements a (virtual) BlueGene/L machine. We run an image of the Linux operating system on each simulated node and then run an MPI test application.

As described in Section 3, BlueGene/L provides several ways of exchanging data among nodes and also between nodes and external entities, like the control system and a file server. This section describes the approach for providing a scalable and robust mechanism of supporting a multi-node simulation.

5.1 Organization of multi-node `BGLsim`

When running a multi-node simulation, `BGLsim` is an asymmetric parallel application. Several, not necessarily identical processes run in different host cluster nodes and communicate with each other to implement the multi-node simulation. All the processes in

Fig. 5. Stack when `BGLsim`is running an MPI application

`BGLsim` use a common library called CommFabric for all the communication services involved in the simulation. Figure 6 shows the general architecture of a multi-node simulation. Note that although torus and tree appear in the figure as clouds, their simulation is done in the **bglsim** process. The clouds appear in the figure to represent how simulator messages are routed in each simulated network.

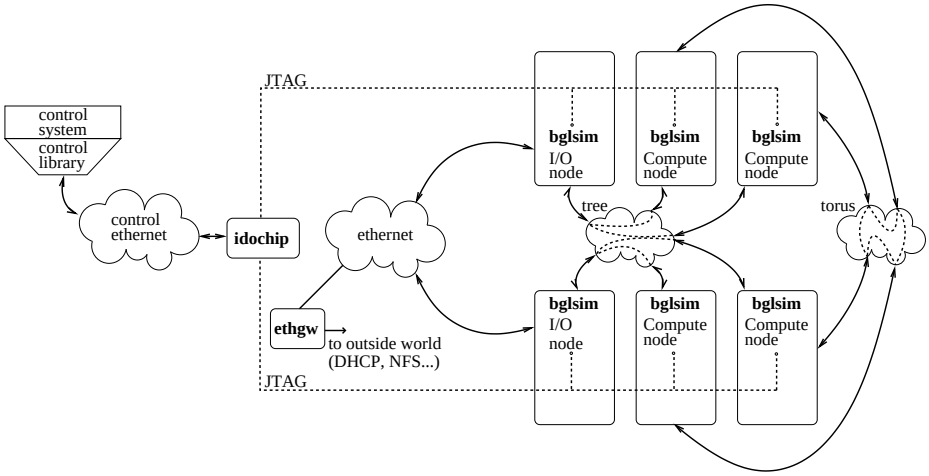

Fig. 6. `BGLsim` multi-node simulation architecture.

There are three different types of processes involved in a `BGLsim` multi-node simulation:

bglsim This process is a single BlueGene/L node simulator. The simulation environment will have as many `BGLsim` processes as the number of nodes being simulated.

ethgw This process provides connectivity services between the virtual Ethernet network in the multi-node simulation environment and the external network. This gateway makes the real external network visible to the simulated BlueGene/L nodes. This allows, for instance, one `BGLsim` node running Linux to mount volumes served by a real NFS server.

idochip This process makes the interface between the external control system and the nodes involved in the simulation. It translates commands encapsulated in IP packets to JTAG transactions. Using its services, the control system, external to the simulation environment, can send low level commands to specific nodes.

When a program running on top of `BGLsim` sends a message through any of the interconnection devices, the corresponding device model sends a message to the simulated destination node. This node then gets the message and puts it in the corresponding device model's incoming buffer, that is then read by the device driver.

In order to have an organized way to describe a simulated machine, a data structure called `bglmachine` was created. This data structure holds information for each node in the simulated machine, like memory and cache sizes, network addresses, and torus coordinates. In addition to information about nodes, `bglmachine` also holds information about the topology of the interconnection networks, which is essential for a proper simulation of the routing process.

Since there are many processes involved in a multi-node simulation, it can get complicated to start a simulation. In order to solve this problem, we created a simulation launcher called `simboot`. When `simboot` is executed, it reads an XML file with the description of the simulated machine, and builds an instance of `bglmachine` that holds this information, the processes are then launched among the available nodes in the host Linux/x86 cluster.

5.2 Communication services

As mentioned before, `BGLsim` supports inter-process communication through a library called CommFabric. CommFabric exposes an abstraction API to `BGLsim` for the communication services. For instance if a message is sent through the Ethernet network, the device model calls `send_ethernet(void *packet)`, and CommFabric then takes care of mapping the destination MAC address present in the header of the Ethernet frame to the right `BGLsim` node. CommFabric provides similar services for sending data through the simulated torus, tree, and JTAG networks. CommFabric uses MPI as its transport layer, so it maps addresses for the various networks to MPI ranks of the corresponding processes. `BGLsim` processes have separate threads for communication and simulation.

It would be simpler to make CommFabric send packets directly from the source to the destination node when transporting packets through the virtual torus interconnect and the virtual tree interconnect. However, we decided to make packets follow the actual route they will take in the real machine. In the torus interconnect, there is a deposit option that leaves a copy of the packet in each node that the packet traverses in a given direction. To simulate this feature we need to do the routing. Correspondingly, the tree network is a class routing network, and the routing depends on routing tables present in

each node. Therefore, to properly simulate the communication and validate the routing tables, we had to implement actual routing.

`BGLsim` multi-node simulation supports the following interconnection networks:

torus This is the main interconnection network between compute nodes. As described in section 3, this is a three-dimensional torus interconnect. When a message is sent through the torus interconnection device, the CommFabric library performs the actual routing through the simulated torus based on the (x, y, z) address of the destination node.

tree This interconnection network is organized as a tree of nodes. It is intended mainly for I/O operations and global reduction operations. In this case, when a message is sent, CommFabric implements the routing of messages by following a tree routing table in each node.

global interrupts This interconnection network is also organized as a tree, and provides global AND/OR operations. CommFabric implements the routing of signals in this network.

ethernet This interconnection network is an Ethernet network that involves all the I/O nodes in the simulation. When a packet is sent through this network, CommFabric maps the MAC address of the destination node to the MPI rank of the node and sends the message directly. If the destination MAC address is not part of the local network, this message is sent to the Ethernet gateway (**ethgw**) and the message is then relayed to the real Ethernet network.

jtag This service, implemented by the **idochip** process, provides communication between the control system network and individual nodes. The kind of messages it transports are low-level control messages (JTAG). When a message arrives in the **idochip** from the control network, it uses the CommFabric to send the message to the corresponding `BGLsim` node.

6 Practical experience with `BGLsim`

Examples of successful use of `BGLsim` include: Porting of Linux to the BlueGene/L I/O nodes, development of a lightweight kernel for the BlueGene/L compute nodes, testing of the BlueGene/L compilers, development of an MPI implementation for BlueGene/L, executing some unmodified MPI benchmarks, porting of LoadLeveler (IBM's job management system) to BlueGene/L, and development of the control system for BlueGene/L. In this section, we give more details on three particular experiences with `BGLsim`: The testing of BlueGene/L compilers, experiments with the NAS Parallel Benchmark IS, and evaluation of simulation speed.

6.1 Compiler testing and validation with `BGLsim`

The BlueGene/L machine double FPU unit has a novel design that cannot be exploited by conventional compilers. For that reason, there was an effort to develop a compiler (and assembler) that would be able to take advantage of the new features of the double FPU. Since the compiler is very specific to the architecture, we could not test its

new capabilities on other hardware. Since the BlueGene/L machine hardware was not available at that time, `BGLsim` was of great help in the compiler debugging process. We were able to find and repair bugs in the compiler. For example, in one situation the compiler was generating a binary instruction with its operands in the wrong position. With the available instrumentation in `BGLsim`, we were able to generate traces and identify the problem. In other cases, we were able to detect wrong code generation for specific operations and to identify problems in the register allocation algorithm.

6.2 NAS IS Parallel Benchmark experiment

In order to validate and demonstrate our simulation environment, we chose to run an MPI application over the Ethernet network. We believe this to be a stressing case for the simulator, as it exercises IP stacks and Linux in all nodes. (In the "normal" BlueGene/L configuration, Linux runs only on the I/O nodes.)

The application chosen is the IS (Integer Sort) kernel from the NAS Parallel Benchmark suite version 2.3 [3, 10]. It is a parallel integer sort problem that sequentially generates keys at the beginning and then sorts them. In its performance measurements, it accounts for communication and computation. The problem size we used was the sample (S) size, which was kept the same when varying the number of processors. We changed the program timer to not only get the time on the simulated machine but also the process's instruction counter. This takes advantage of our per process instruction counters.

To run the experiments, we cross-compiled the LAM/MPI 6.5.4 and common Linux network utilities such as telnetd and rshd. We also built a ramdisk with all the basic utilities, lamd and the cross-compiled IS benchmark for 1, 2, 4 ,8, 16 and 32 processors. In a cluster of Linux/x86 host machines, we booted a 32-node `BGLsim` simulation, and ran the experiments.

In Figure 7, we plot the number of instructions spent on communication, the number of instructions spent on computation and the total number of instructions per process, as a function of the number of processes. As expected, the number of instructions for computation decreases as the number of processors increases and the number of instructions used for communication increases with the number of processors.

By analyzing the total number of instructions, we can verify that the configuration for this problem size that results in the least number of executed instructions (per node) is the one that uses 16 nodes, when the communication overhead is not so high and there is still a significant gain in splitting the computation.

6.3 NAS Parallel Benchmarks slowdown

In order to analyze the slowdown introduced by our simulator in the execution of an application, we have executed all the NAS Parallel Benchmarks with a varying number of nodes. We compare the execution time of the benchmarks when running natively on our x86 Linux host machines and when running on the simulated BlueGene/L machine.

Table 1 presents the results obtained, in the form of slowdown. That is, the ratio of simulation time and native time. We used the class S size for all benchmarks. Most of

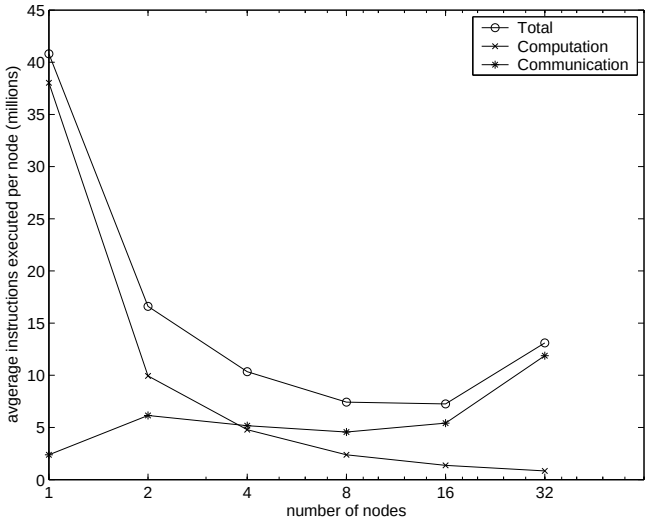

Fig. 7. Computation and communication instructions executed per node in the NAS Parallel Benchmark IS (size S), as a function of the number of nodes used in the computation.

the benchmarks were run on 2, 4, 8, and 16 nodes. The BT and SP benchmarks require a square grid and therefore were executed on 4, 9 and 16 nodes. The class S of LU can only be executed with 2 and 4 nodes, due to the problem size.

We observe that the mean slowdown for all these benchmarks is around 600, that is, the simulation takes 600 times the real execution time of the benchmark running natively on the host system. Among the benchmarks, EP is the one that suffers a significantly higher slowdown. In general, the slowdown of the simulations decrease when the number of processors is increased. The likely reason is that the slowdown for our simulation of computations is higher than the slowdown for our simulation of communications.

benchmark / nodes	2	4	8/9	16
BT	-	660	570	490
SP	-	530	450	400
IS	260	220	170	150
CG	640	540	510	500
FT	790	570	480	230
LU	670	340	-	-
EP	1750	1670	1530	1250
MG	680	350	270	280

Table 1. Slowdown of the simulated NAS Benchmarks.

7 Conclusions

We have presented a complete parallel system simulator that is being used for software development and hardware analysis and verification. It is capable of running the same exact code that runs on real hardware. System software has been instrumented to collect additional important information that would be impossible to gather in real hardware. The simulator has been useful in porting compiler and operating system code, in developing MPI libraries, control and monitoring systems, and job scheduling and management tools.

Experiments were performed to validate the functionality of the simulator and measure simulation speed. We were able to gather data from individual processes in the simulated system through the use of joint instrumentation between the simulator and the operating system.

The simulator we developed is not cycle accurate. It is useful for developing and evaluating code for a large-scale parallel machine and verifying its architectural design before it is deployed. We believe it can provide significant input for future designs of parallel systems. To make the simulator even more useful, we are currently extending it with timing models that account for functional unit latencies, memory system behavior and inter-node communications. These timing models will provide approximate performance information for applications and system software. Although not cycle accurate, we believe the timing models can have enough accuracy to support design decisions, in both hardware and software.

References

1. N.R. Adiga et al. An overview of the bluegene/l supercomputer. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
2. Lars Albertsson and Peter S. Magnusson. Using complete system simulation for temporal debugging of general purpose operating systems and workloads. In *Proceedings of Mascots 2000*, 2000.
3. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks. Technical Report RNR-94-007, RNR, March 1994.
4. Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison and Intel Corporation, June 1997.
5. Luiz DeRose, K. Ekanadham, and Jeffrey K. Hollingsworth. Sigma: A simulator infrastructure to guide memory analysis. In *Proceedings of SC2002*, Baltimore, Maryland, November 2002.
6. P. Magnusson et al. Simics/sun4m: A virtual workstation. In *Usenix 1998 Annual Technical Conference*, 1998.
7. Steve Herrod, Mendel Rosenblum, Edouard Bugnion, Scott Devine, Robert Bosch, John Chapin, Kinshuk Govil, Dan Teodosiu, Emmett Witchel, and Ben Verghese. The simos simulation environment. Technical report, Computer Systems Laboratory - Stanford University, February 1996.
8. Steve Alan Herrod. *Using Complete Machine Simulation to Understand Computer System Behavior*. PhD thesis, Stanford University, 1998.
9. Shubhendu S. Mukherjee, Sarita V. Adve, Todd Austin, Joel Emer, and Peter S. Magnusson. Performance simulation tools. *Computer Magazine*, February 2002.

10. Nas parallel benchmarks. `http://www.nas.nasa.gov/Software/NPB`.
11. Sundeep Prakask, Ewa Deelman, and Rajive Bagrodia. Asynchronous parallel simulation of parallel programs. *Software Engineering*, 26:385–400, 2000.
12. Karthikeyan Sankaralingam, Ramadass Nagarajan, Stephen W. Keckler, and Doug Burger. Simplescalar simulation of the powerpc instruction set architecture. Technical Report TR2000-04, Computer Architecture and Technology Laboratory - Department of Computer Science - The University of Texas at Austin, 2000.
13. Arun Sharma, Anthony-Trung Nguyen, and Josep Torrellas. Augmint - a multiprocessor simulation environment for intel x86 architectures. Technical report, Center for Supercomputing Research and Development - University of Illinois at Urbana-Champaign, March 1996.