The IBM Blue Gene/Q Interconnection Fabric

This article describes the IBM Blue Gene/Q interconnection network and message unit. Blue Gene/Q is the third generation in the IBM Blue Gene line of massively parallel supercomputers and can be scaled to 20 petaflops and beyond. For better application scalability and performance, Blue Gene/Q has new routing algorithms and techniques to parallelize the injection and reception of packets in the network interface.

Dong Chen
Noel A. Eisley
Philip Heidelberger
Robert M. Senger
Yutaka Sugawara
Sameer Kumar
Valentina Salapura
David L. Satterfield
IBM T.J. Watson Research Center

Burkhard Steinmacher-Burow
IBM Germany Research & Development

Jeffrey J. Parker
IBM Systems & Technology Group

The IBM Blue Gene/Q (BG/Q) system is the third generation in the IBM Blue Gene line of massively parallel supercomputers and can be scaled to 20 petaflops (2×10^{16} floating-point operations per second) and beyond.[1,2] This article summarizes the BG/Q network and message unit (MU). (A more detailed description is available elsewhere.[3]) The highly parallel MU provides the functionality of a network interface. Both the network logic and the MU are integrated with the processors and cache memory on a single chip and occupy 8 percent of the chip's total area.

The network is a five-dimensional (5D) torus. BG/Q integrates both collective and global barrier functions into this torus network. A thin software layer enables applications to achieve low latency and high throughput. A nearest-neighbor exchange benchmark achieves up to 35.4 Gbytes/s, which is 98.3 percent of the theoretical peak. This demonstrates the effectiveness of the highly parallel and integrated network and MU design.

In addition to describing the network and the MU, this article discusses the reliability, availability, and serviceability (RAS) and physical design, as well as verification and testing methodologies. We also describe the software interfaces to the network and MU, and we report on our initial hardware performance measurements.

Interconnection network

BG/Q systems consist of compute nodes and I/O nodes. Applications run on compute nodes, whereas file I/O is shipped to I/O nodes that interface via PCI Express to a file system. Compute nodes are interconnected via a 5D torus requiring 10 2-Gbyte/s bidirectional links (2 Gbytes/s for each sender and 2 Gbytes/s for each receiver). An additional 11th bidirectional 2-Gbytes/s link connects compute nodes to I/O nodes. A subset of compute nodes, called *bridge nodes*, have their I/O links attached to I/O nodes. To match an I/O node's 4-Gbyte/s PCI Express bandwidth, two 2-Gbyte/s torus links are typically attached to each I/O node. I/O traffic routes deterministically over the compute-node torus to the bridge node specified in the packet header, and then goes over the I/O link to the attached I/O node.

Let us now compare the compute-node torus in BG/Q to that in BG/L[4] and

Published by the IEEE Computer Society

BG/P,[5] the first- and second-generation Blue Gene systems.. The 3D torus network in BG/P is essentially identical to that in BG/L,[6] the major difference being an increase in link bandwidth. BG/L has 175-Mbyte/s links, whereas BG/P has 425-Mbyte/s links. Thus, each BG/Q link at 2 Gbytes/s is 11.4 (4.7) times faster than a BG/L (BG/P) link, and the total single compute-node torus bandwidth in BG/Q is 19 (7.8) times than in BG/L (BG/P).

Other supercomputers, including the 3D Cray networks[7,8] and the 6D Fujitsu Tofu interconnect,[9] have also used torus networks. In contrast, the IBM Power7-IH uses a high-radix direct-connect switch.[10]

We chose a 5D torus for BG/Q because of three main reasons. First, from a performance perspective, it achieves high nearest-neighbor bandwidth while increasing bisection bandwidth and reducing the maximum number of hops (and latency) compared to a lower-dimensional torus. For example, a 20-petaflops 16 × 16 × 16 × 12 × 2 BG/Q has about 46 (19) times the bisection bandwidth of a 64 × 48 × 32 BG/L (BG/P) with the same number of nodes. Compared to BG/L, 11.4× of the 46× comes from increasing the link bandwidth, and 4× comes from reducing the length of the maximum dimension.

Second, the torus permits partitioning a large machine into independent submachines; applications running in different partitions don't affect one another, except possibly for file I/O. Third, the torus permits most links within a 512-node (4 × 4 × 4 × 4 × 2) midplane to be electrical rather than optical, thus reducing cost. Links between midplanes connect through separate link chips via optical fibers. The torus dimensions are labeled A, B, C, D, and E. The last dimension, E, is constrained to length 2, thereby reducing wiring and cabling.

Also, because many applications don't use point-to-point and collective messaging at the same time, BG/Q integrates barrier and collective functionality onto the torus network. This reduces cost, simplifies cabling, and maintains partitioning capability.

Data packets on BG/Q include a 32-byte header, a 0- to 512-byte data payload in increments of 32 bytes, and an 8-byte trailer for link-level packet checks. Including the protocol overhead, at most 90 percent of the 2-Gbytes/s raw link bandwidth (or 1.8 Gbytes/s) can be used for user data.

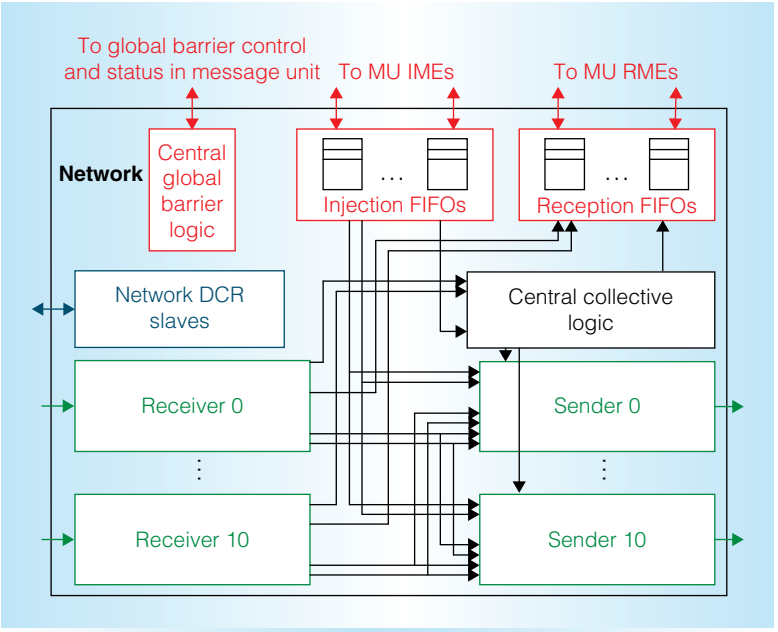

Figure 1. The BG/Q network router logic. Point-to-point packets are injected into the network by the message unit (MU) injection messaging engines (IMEs), and they are routed from sender to receiver (on another node) until they arrive at the destination where they are received by the MU reception messaging engines (RMEs). (DCR: device-control register.)

The on-chip per-hop latency for point-to-point packets on BG/Q is approximately 40 ns, compared to approximately 97 ns for BG/L and 46 ns for BG/P. Of the 40 ns, which represent 20 network cycles, 8 cycles are for the network logic (compared to 12 network cycles for BG/L), and the rest are for the serializer/deserializer (SerDes) and high-speed signaling. Hardware measurements have confirmed this per-hop latency.[3] The worst-case hardware one-way point-to-point latency on a large 16 × 16 × 16 × 12 × 2 system is expected to be less than 3 µs, including circuit-board-trace and cable delays.

Network logic

Figure 1 shows the network logic in BG/Q. There are 11 sender and 11 receiver units, one for each of the 5D torus links and one for the I/O link. The MU injects packets into network injection first in, first

out (FIFO) buffers and pulls packets from network reception FIFOs. The network logic allocates multiple injection and reception FIFOs to normal-priority user point-to-point data on the 5D torus. In addition, dedicated FIFOs are allocated to intranode local transfers, high-priority user and system point-to-point data, and user and system collective data. The number of FIFOs is sufficient to ensure that all links can remain busy simultaneously. Injection FIFOs are not tied to torus links, thus permitting point-to-point packets from any injection FIFO to use any link. When a normal-priority point-to-point packet arrives at its destination, it goes into the reception FIFO associated with the receiver unit on which the packet arrived. For example, packets arriving at the A- receiver always go into the A- reception FIFO.

Virtual channels and point-to-point routing

BG/Q has separate virtual channels (VCs) to support an integrated network with user, system, and collective traffic. Each receiver has separate buffers for each VC, and there is virtual cut-through logic and token flow control similar to BG/L.[6,11] To improve performance, we increased the number of packets that each VC can store. To reduce head-of-line blocking, BG/L and BG/P have two VCs for dynamic (adaptive) routing, but the deterministic VC is managed as a single-queue FIFO. Improving on that, BG/Q employs a form of virtual output queuing. Every point-to-point VC maintains multiple queues, each holding one or more packets. Packets in different queues requiring different links don't block one another, but deterministically routed packets in different queues requiring the same link are handled in first-come, first-served order, thus maintaining in-order delivery.

As in BG/L and BG/P, dynamic routing in BG/Q follows the shortest paths to the destination and chooses one link from among the available links at every hop. When a dynamic packet is placed in a VC buffer, arbitration logic selects the queue with the fewest packets in it. Packets in different dynamic VC queues don't block one another. As in BG/L, there are multiple data paths from each receiver so that multiple packets can be transferred simultaneously.

Point-to-point routing in BG/Q is similar to that in BG/L. *Hint bits* specify which links can be used (at most, one per dimension). However, there are several important improvements to boost performance, especially on asymmetric tori. First, the dimension order for deterministically routed packets is now programmable to permit routing the longest dimension first, which is typically more efficient.

Second, the BG/Q network uses programmable *zone routing*, which constrains the order in which dimensions are dynamically routed. For example, in a $16 \times 16 \times 12 \times 12 \times 2$ torus, routing can be configured to route the longest dimensions first: A or B, then C or D, then E. Because BG/L routing is very efficient on symmetric tori, zone routing effectively breaks the routing into symmetric zones of decreasing size, moving packets from the busy links to either equally busy links or more lightly loaded links.[12] This reduces pressure on internal network buffers, thereby improving performance. Detailed near-cycle-accurate parallel simulations of an all-to-all pattern of a $16 \times 8 \times 8 \times 8$ torus showed that performance improved from 66 percent of the network peak without zone routing to 93 percent of the peak with zone routing. Simulations of zone routing on a $16 \times 16 \times 16 \times 8$ torus achieved 99 percent of the network peak.

Collective and barrier support

BG/Q has better hardware support than BG/L for collective operations within the network. First, whereas BG/L requires two passes over the collective network for floating-point reductions, BG/Q incorporates logic for one-pass double-precision floating-point sums. Second, BG/Q supports collective operations over message passing interface (MPI) subcommunicators, provided they are contiguous subrectangles of the torus (over lines, planes, 3D subcubes, and so on).

The nodes participating in a collective operation specify a contiguous tree within the torus network. Packets go up the tree, being reduced on each hop, then turn around at the tree's root and travel back

down the tree. Class routes define and identify the tree by specifying which of the 11 links (including I/O) are inputs on the up-tree path, which link is the up-tree output, and whether there is a contribution from the local node. Each node can participate in up to 16 different class routes, but the entire machine can have far more than 16 class routes. For example, a class route could exist for each 2D AB plane, and all of these could be active simultaneously without any interference between them. Packets from a given message flowing up a tree can overlap with packets flowing down the same tree. Packets from different messages on noninterfering trees can proceed in parallel; however, at any given time, the up-tree collective logic can process packets from only one tree, while the down-tree logic can process packets from the same or a different tree. For reductions, up-tree packets are stored in the receiver's VC buffers until packets from all of the tree's inputs have been received.

The collective unit supports several floating-point operations (add, min, and max) and fixed-point operations (signed and unsigned add; min and max; and bitwise AND, OR, and XOR). The collective logic also supports operations such as broadcast, reduce to any single node, and all-reduce to all nodes. The collective logic incorporates a bit-reproducible floating-point unit that operates close to the link bandwidth (up to 86 percent of the raw link bandwidth). Floating-point reductions add an average of 12 ns to the per-hop point-to-point latency, as hardware measurements confirm.[3] The hardware latency for a short all-reduce on a 16 × 16 × 16 × 12 × 2 BG/Q is expected to be about 6.5 µs.

BG/Q also implements global barriers and interrupts using special packets and logic. Sixteen barrier classes can be configured, similar to collective classes. BG/Q performs a global OR of the inputs of each class on each node. When that changes, a packet is sent up or back down the tree. The hardware latency for a global barrier on a 16 × 16 × 16 × 12 × 2 BG/Q is expected to be about 6.3 µs.

Link chip and partitioning

Each group of 32 nodes has nine link chips to provide support for the optical modules, including encoding and decoding, fiber sparing, and on-the-fly single-bit error correction using IBM 8b/10b-P (8-bit/10-bit with parity) code.[13] This code is a modification of the standard 8-bit/10-bit code to use Hamming distance 2. The parity of several data lanes is encoded and sent over a spare fiber. If one single-bit error occurs over the parallel 8-bit words, the parity can correct it. Error counters on the link chip can detect when fibers are going bad; the data sent on a bad fiber can then be rerouted to a spare fiber with only a momentary impact on the application.

For partitioning, link chips can be programmed to loop a midplane's links back to the same midplane or to the next midplane, or individual links can be held in reset. This makes it impossible for packets to cross from one partition to another.

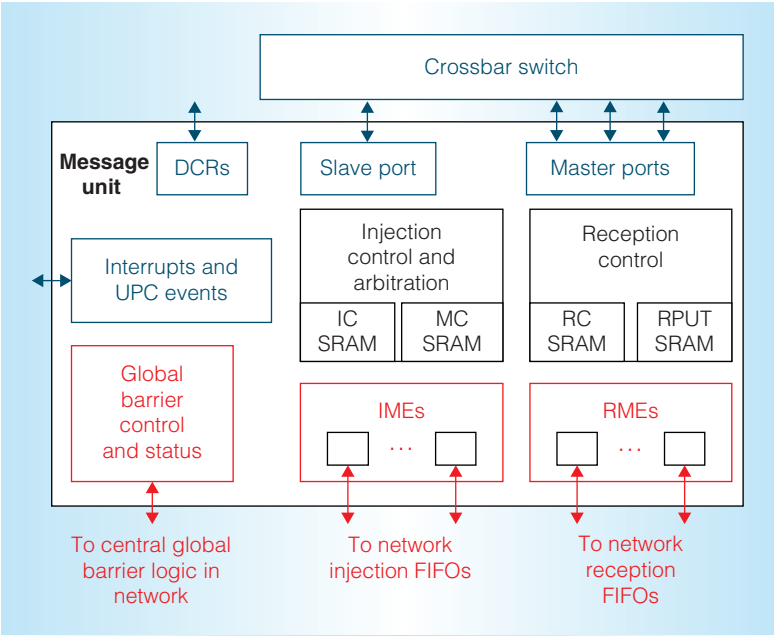

Figure 2. The BG/Q message unit logic. The IMEs packetize messages for injection into the network, whereas the RMEs receive packets from the network and store them in memory. (IC: injection control; MC: message control; RC: reception control; RPUT: remote put; SRAM: static RAM; UPC: universal performance counter.)

Message unit

The MU acts as an interface between the network and the BG/Q memory system, as Figure 2 shows. We designed the MU to provide low latency and high throughput—enough to keep all the links busy. The MU

supports direct puts (remote direct memory access [RDMA] write), remote gets (RDMA read), and memory FIFO messages. The MU maintains pointers to circular buffers in memory called injection memory (IM) FIFOs and reception memory (RM) FIFOs. There are 544 IM FIFOs (32 per core) and 272 RM FIFOs (16 per core)—enough so that each processor thread can have its own set without locking.

The MU transfers data to and from memory, packetizes messages, and provides simple address translation on the reception side. Messages can have arbitrary byte alignments, and incoming packets can optionally trigger processor interrupts. Dynamically routed packets from the same message can arrive in a different order from which they were sent, and the MU makes no attempt to reorder such packets. The MU exploits the BG/Q Level-2 (L2) cache atomic functionality, updating message byte counters in memory via atomic increments. The MU can also perform atomic operations during memory transfers. This can be useful for locks, work queues, packet flow control, and so on.

Whereas the BG/P DMA has two engines, one for injecting packets and one for receiving packets, the BG/Q MU has many parallel engines: one injection messaging engine (IME) for each network injection FIFO, and one reception messaging engine (RME) for each network reception FIFO. The IMEs and RMEs share master ports on the BG/Q memory system crossbar switch. All MU reads and writes pass through the globally shared L2 cache, which manages coherency across the node's memory system. The master ports' bandwidth is sufficient to support all 10 (user mode) torus network links when the messages fit in the L2 cache. The MU also has a crossbar slave port that provides memory-mapped addresses for software to update FIFO pointers, set up address translation, and handle certain interrupt conditions. The MU supports *local memory copy*, meaning the MU copies a message to another area in the local memory using loopback network FIFOs.

Message injection

On the sending side, injection-control logic selects the next message descriptor to fetch and inject into the network. A 64-byte message descriptor contains network-routing information, the message type, a pointer to the start of the message, the message length, and where to store the message on the destination node. The injection-control static RAM (IC SRAM) stores start, size, head, and tail pointers for each IM FIFO. To send a message, software copies a descriptor into an IM FIFO at the location pointed to by the tail, and then moves the tail past the descriptor. The MU sees that the IM FIFO is nonempty and fetches the descriptor at the FIFO's head pointer. Once a descriptor's message has been sent, the MU moves the head past that descriptor and begins to process the next descriptor, until the IM FIFO is empty.

Whereas descriptors in a given IM FIFO are processed sequentially, descriptors in different IM FIFOs are processed in parallel—hence, the benefit of using multiple IM FIFOs. The message control (MC) SRAM holds descriptors fetched by the injection-control logic. After an IME injects a packet from a descriptor, arbitration logic selects a new packet for injection from among the MC SRAM's available descriptors.

An IME builds a packet for the assigned descriptor and stores it in the corresponding network injection FIFO for transmission. It issues master port requests to fetch the message data and puts this data into the packet payload. As the packet flows out of the network injection FIFO, the network logic packs the payload contiguously. Multiple IMEs can process different message descriptors independently in parallel, keeping the network links busy.

Message reception

On the reception side, each RME retrieves packets from its corresponding network reception FIFO and stores the packets in their designated memory locations. The RMEs operate independently in parallel to drain the network reception FIFOs more quickly than the network can fill them. The RMEs distinguish between three types of packets, and the MU hardware performs different operations accordingly:

- *Memory FIFO.* The MU copies the packet into the RM FIFO specified in the packet header.

- *RDMA write.* The MU copies the packet payload directly into the predetermined memory location. After storing the payload, the MU updates a message reception byte counter, located anywhere in local memory, via an L2 atomic operation. The write locations for the data and byte counter are specified in the packet. In a typical usage, the byte counter is initialized by software, which then polls this counter to detect message completion.
- *RDMA read.* The MU treats the received packet payload as message descriptors and injects them into the IM FIFO specified in the packet, so that the injection MU hardware can process them. Typically, these descriptors generate RDMA write packets that transmit data back to the sender of the RDMA read packet.

Each RM FIFO has head and tail pointers, managed by the reception control (RC) SRAM. When a memory FIFO packet arrives, the MU copies the packet at the tail of the RM FIFO designated in the packet, and moves the tail past the received packet. When software sees that the tail has moved (via polling), it processes the packet (for example, it copies the payload to a user buffer), moves the head past the packet, and continues processing packets until the RM FIFO is empty.

To allow multiple RMEs to process packets targeting the same RM FIFO simultaneously, the RC SRAM maintains a hardware shadow tail, in addition to the regular tail that is visible to software. The RM FIFO's shadow tail locates where FIFO space has been reserved for RMEs to store packets, whereas the regular tail locates where packets have been completely stored. When an RME starts receiving a new memory FIFO packet, it reads the shadow tail and atomically increments it by the size of the packet for the next RME. Then the RME stores the packet in the location to which the original shadow tail pointed. After storing the packet, the RME increments the regular tail past the received packet, provided all previous packets have been fully received.

For virtual-to-physical memory translation in RDMA write transfers, the receiver uses a 544-entry base address table (BAT) stored in the MU to determine the target physical address. The packet header contains one BAT index for storing the message payload and another for the reception counter. These base addresses are added with *put offset* and *counter offset* in the packet header to determine the memory address for storing payload data and updating the counter, respectively. The operating system manages memory to efficiently exploit this simplified translation mechanism.

RAS and physical design

A rich set of RAS features in the network protocol, the network logic, and the message unit help provide the high level of reliability that is critical for high-performance petascale machines.

BG/Q can detect corrupted packets and resend them. A copy of sent packets is stored in the sender's retransmission FIFO for later retransmission if an appropriate acknowledgement isn't received within a programmable timeout. Because of the 8-bit/10-bit encoding that occurs when packets go over optical links between midplanes, a single-bit error on a fiber can result in an 8-bit error burst. Two such errors in the same packet can exceed the guaranteed detection properties of most cyclic redundancy checks (CRCs). So, we used symbol-oriented Reed-Solomon (RS) codes instead of CRCs, and these codes are appended to the end of each packet. An end-to-end 10-bit RS code word covers a packet's unchanging part and follows it through the network. In addition, five 10-bit-per-hop RS code words cover the entire packet. This code is capable of detecting any five symbol errors and has an escape rate of 2^{-50}.

In addition, the packet's network header contains extensive error consistency checks, including an 8-bit Hamming code that detects any three bits of error in the header. CRCs cover all packets sent and all packets received over a link. At the end of a run, the paired sender and receiver CRCs must be equal; if they are not equal, then a packet-level RS code escape has occurred.

The network and MU logic have extensive logic and checks to separate users from system traffic and to prevent user-space

errors from interfering with system messaging. All data paths and internal buffers in the network logic and the MU are protected by single-bit correction, double-bit detection error-correction code (ECC). Critical control-flow latches, such as state-machine latches, use hardened latches and have parity detection.

In BG/Q, the 64-bit PowerPC cores operate at 1.6 GHz, whereas the memory system and MU operate at 800 MHz. The internal network logic operates at 500 MHz; the network handles 4 bytes per network cycle, thereby matching the 2 Gbyte/s link bandwidth. The network logic comprises just 3 percent of the chip's area, with the SerDes consuming an additional 4 percent of the chip area. The MU takes up only 1 percent of the total chip area. We described the MU and network portions of the BG/Q chip in VHDL, synthesized them using application-specific integrated circuit (ASIC) methodology, and then fabricated them in IBM's copper 45-nm silicon-on-insulator (SOI) technology. More than 85 percent of the latches in the network logic and MU are clock-gated to save power.

Verification and testing

Sophisticated VHDL hardware logic is required to implement the complex network and MU functionality, such as support of multiple packet types; extensive use of pipelining to achieve performance targets; and widespread error detection, correction, and reporting. These units are too complex for formal verification techniques, so we used a testbench approach to simulate and verify the logic behavior across a large range of pseudorandom scenarios. Constructing a testbench and VHDL logic simulation incorporating thousands of interconnected network nodes isn't efficient. So, we built several different single-node unit testbenches of only the network and MU logic. Our approach also included a full-chip testbench to simulate the entire chip's logic by executing C programs that sent and received messages and verified the results.

We constructed two network-only testbenches, and one MU testbench that could be run as either *MU only* or *MU plus network*. Each of these testbenches simulates the logic from a single node. We reused portions of the C++ testbench code across multiple testbenches. Moreover, we used standard IBM logic verification tools that implemented a runtime environment in which testbench code drove and checked the VHDL logic on a cycle-by-cycle basis.

Standard IBM coverage tools helped ensure that we reached the appropriate logic corner cases. The logic is annotated with coverage events; a count is bumped each time an event is hit during the simulation, and then is stored in a database from which coverage reports can be produced. Examples of interesting coverage events are that a FIFO is full, or that a packet enters a FIFO at the same time another packet exits the FIFO.

Network unit testbenches

In the two network-only testbenches, C++ drivers emulate the MU; implement bus protocols between the network and the MU; and inject, receive, and check packets. These testbenches check that every byte of every packet is correct, and that every packet is received in the right place and in the correct order. The first loopback testbench uses a single network unit and configures the network links in loopback, so that, for instance, the A+ sender is connected to the A- receiver.

In this testbench, all packets are injected into and received from the network through emulated MU code. By setting multiple hint bits in the packet's header, we can test the point-to-point cut-through logic. For example, if the A+ and C- hint bits are set, a packet can be placed in a network injection FIFO via the MU emulation code, exit the A+ sender, enter the A- receiver, exit the C- sender, enter the C+ receiver, and then go into the network reception FIFO, where the MU emulation logic will pull it out.

We tested the link-level retransmission protocol by corrupting packets on the emulated links. The collective logic's capability to combine (sum) data packets inherently requires multiple nodes and can't be tested in this loopback testbench, nor can the barrier packet logic.

To test collective operations and barriers, and to more flexibly and aggressively test the network logic, we built a second link driver

testbench. This testbench emulates a multinode network surrounding the single-node logic under test. The testbench constructs packets on the emulated neighboring nodes, emulates the link-level protocol, passes the packets into the network logic, and then checks packets when they arrive at either an emulated MU or an emulated node. This testbench varies packet types, stresses certain links more than others (for example, by sampling the hint bit for those links with higher probability), and better covers the link-level retransmission protocol and the spacing of packets compared to the loopback testbench.

Message unit testbench

The message unit testbench simulates a single node's MU logic in network loopback. The testbench has two modes. In the first mode, the network logic is also simulated and verified. In the second mode, C++ drivers emulate the network logic and provide easier debugging and faster testbench execution. C++ drivers implement all bus protocols between the MU and the rest of the chip. This testbench consists of objects that create messaging scenarios and verify the resulting MU logic behavior.

At the top level, one or more objects each emulate a message-passing application. Within an application object, one or more objects each emulate an IM FIFO. Within an IM FIFO object, one or more objects each emulate a message descriptor. A descriptor object contains a header object and a payload object. Similar objects exist for RM FIFOs, message byte counters, and every other aspect of MU functionality. Each testbench object verifies the corresponding MU behavior. For example, the injected-message payload object verifies exact crossbar bus transactions when it is loaded by the MU logic. Each object has characteristics to fully exercise the MU functionality. For example, an application object can have user or system privileges. To improve coverage, the testbench randomly chooses some object characteristics during a given simulation run, while constraining other characteristics. Thus, for example, a simulation run could include short user FIFO messages and long system put messages.

Both the network and MU testbenches perform considerable testing of error conditions for bad paths. For example, we attached an ECC corruptor, which randomly created single-bit errors, to every appropriate place in the logic. We used a bad-path corrupter to test reporting of rare fatal errors, such as double-bit errors in arrays or parity errors on hardened state-machine latches.

Full-chip testing

In the full-chip testing framework, we simulate an HDL model of the entire chip in a cycle-accurate manner. The main purposes of full-chip testing are to verify unit interfaces such as between the MU, crossbar, and L2 cache; verify interactions with software on the cores; and verify performance. Because the full-chip model's large size limits the simulation speed, full-chip tests typically simulate fewer cycles than unit-level tests. A limited number of special hardware accelerator devices were available for speeding up the full-chip simulations. The full-chip tests use the production Blue Gene/Q initialization code and software libraries running on the simulated cores. Examples of full-chip test cases include performance tests and bad-path tests to verify that the hardware and software can detect, report, and recover from error conditions.

Bring-up testing

Bring-up testing can begin after we receive hardware. Bring-up tests expand on logic verification tests, but they are orders of magnitude faster and span multiple nodes with real application messaging and memory access patterns. When testing a parallel-networked system such as BG/Q, we must start with single-node tests running in loopback and then scale up to progressively larger multinode configurations. Besides testing logical functionality, these tests also quantify the system's electrical properties.

We wrote a complex C-language network random test to parameterize and vary workloads and configuration settings. There were hundreds of parameters, including message size and type, packet size, collective class maps and operations, and routing hint bits. We varied parameters randomly, with special emphasis on probability distributions that

Table 1. Cumulative nearest-neighbor link throughput (the sum of the sender and receiver bandwidths) for all 10 links.

Message size (Kbytes)	Throughput (Gbytes/s)	Percent of raw link bandwidth	Percent of effective peak data utilization
4	17.0	42.5	47.2
8	23.0	57.5	63.9
16	27.9	69.8	77.5
32	31.3	78.3	86.9
64	33.3	83.3	92.5
128	34.2	85.4	94.9
256	34.9	87.1	96.8
512	35.2	88.0	97.8
1,024	35.4	88.4	98.3
2,048	26.4	66.0	73.3
4,096	24.7	61.8	68.6
262,144	25.0	62.5	69.4

create stress scenarios such as network hot spots. We checked messages for correctness, and we checked in-order delivery for deterministically routed packets. We ran approximately half a million node hours of network and MU tests during bring-up to qualify hardware correctness.

Software support

The BG/Q system's software provides highly optimized C inlines via system programming interfaces (SPIs) for applications, message layer libraries, kernels, and system software to program the MU and the network. The SPIs are thin software abstraction layers of the BG/Q network and the MU hardware. They are flexible, enabling the implementation of many message-passing and resource allocation algorithms. For example, a message-layer library can implement point-to-point and collective algorithms optimized for the semantics of MPI and partitioned global access space (PGAS) runtimes, whereas an application that requires different resource management or different point-to-point or collective algorithms can directly program to the SPIs for optimized performance. In addition, kernels such as the BG/Q Compute Node Kernel can program the MU for function-shipping I/O system calls to the I/O nodes. (More details on the BG/Q software stack are available elsewhere.[3])

Performance measurements

We carefully tracked network and MU performance in every phase of BG/Q development, from prelogic simulation to unit and full-chip simulation, to bring-up testing. We obtained performance results for up to 2,048 BG/Q pass-2 prototype nodes, and we ran C-language SPI benchmarks, including nearest-neighbor, all-to-all, and all-reduce.

Nearest neighbor

An SPI RDMA write benchmark measured the achievable throughput to nearest neighbors on a single node, where the 5D torus links are looped back. For example, the A+ direction is looped back to the A- direction, so all outgoing data in the A+ direction arrives at the A- links, and vice versa.

Table 1 shows the cumulative nearest-neighbor link throughput (the sum of the sender and receiver bandwidths) for all 10 links. At 1 Mbyte, the cumulative throughput achieved on the 10 links in both directions was 35.4 Gbytes/s, which constitutes an efficiency of 88.4 percent of the raw link throughput, and 98.3 percent of the peak 90 percent effective data utilization of the links. When message sizes were less than or equal to 1 Mbyte, all 20 sender and receiver streams fit in the L2 cache, and there was sufficient memory bandwidth between the L2 cache and the MU. Messages that were 2 Mbytes or larger spilled out of the L2 cache, causing throughput degradation because of double data rate (DDR) DRAM bandwidth limitations.

All-to-all

We ran an SPI-based all-to-all performance test, in which each node sent one RDMA write message to each of the other $(n - 1)$ nodes in the system. On 512 nodes, with a 4-Kbyte message size and dynamic routing, the network achieved 95 percent of theoretical peak performance. This improved to 97 percent of peak for 32-Kbyte message size. Deterministic routing achieved only 70 percent of peak for this near-symmetric configuration.

To demonstrate the effects of zone routing, we ran the same test on 1,024 and

2,048 nodes configured as a 4 × 4 × 8 × 4 × 2 and a 4 × 4 × 16 × 4 × 2 torus, respectively. We compared the performance with three different routing configurations:

- deterministic routing (longest to shortest order);
- default dynamic zone routing (longest to shortest, but with dimensions of the same size grouped into a single zone—that is, C dimension first, then adaptive routing in A, B, and D dimensions, then E); and
- unconstrained dynamic routing in all dimensions.

The first two routing configurations both achieved 92 percent of peak on 1,024 nodes, and 94 percent of peak on 2,048 nodes. We expected this result because both configurations route the bottleneck (longest) dimension first. The third configuration achieved only 80 percent of peak on 1,024 nodes, and 59 percent on 2,048 nodes.

Collective

Table 2 reports the throughput of collective floating-point add operations on 512 nodes using RDMA write messages. Owing to complex logic for collective sums, the percent of peak was slightly lower than for point-to-point communication.

Table 2. Collective throughput.

Message size (Kbytes)	Throughput (Gbytes/s)	Percent of raw link bandwidth	Percent of effective peak data utilization
0.5	0.26	13.0	14.4
1	0.45	22.5	25.0
2	0.72	36.0	40.0
4	1.01	50.5	56.1
8	1.27	63.5	70.6
16	1.45	72.5	80.6
32	1.57	78.5	87.2
64	1.64	81.7	90.8
128	1.67	83.5	92.8
256	1.69	84.4	93.7
512	1.70	84.9	94.3
1,024	1.70	85.1	94.6

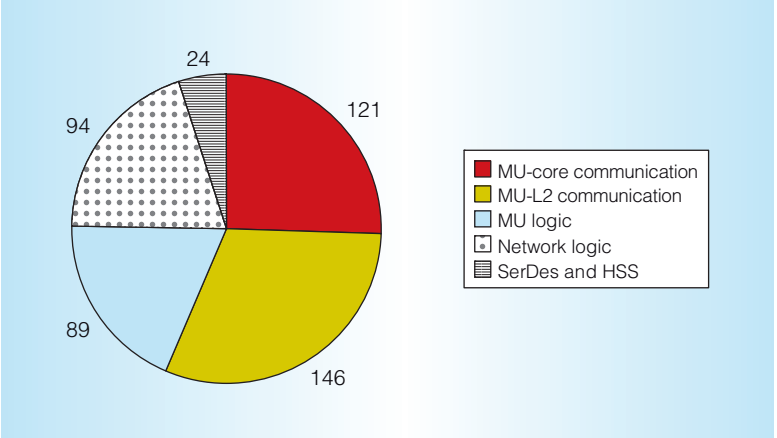

Figure 3. Message latency breakdown (in nanoseconds). The latency includes time for software to inject an 8-byte remote direct memory access (RDMA) write message, route through one hop on the torus network, be received by the MU, and be processed by the software. (HSS: high-speed serial interface; L2: Level-2 cache; SerDes: serializer/deserializer.)

Message latency

Figure 3 shows the breakdown of hardware latency for a short (8-byte) RDMA write message in local-node loopback mode. We obtained this baseline latency using a cycle-accurate HDL simulator. This latency includes message injection, one 40-ns torus hop, and message reception.

More than half (267 ns) of the total 474-ns hardware latency is in the local memory system, with the remainder divided roughly evenly between the MU (89 ns) and the network (94 ns). The SerDes and the high-speed serial (HSS) interface used an additional 24 ns. Of the 267-ns memory system latency, 146 ns is spent by the MU fetching the descriptor and payload data, and storing the payload and counter to memory. The 121-ns portion of the memory latency includes time to propagate the updated RDMA atomic counter value to the core, and the core-to-MU control write latency. The former delay varies, depending on how aggressively software polls the counter value. The MU latency is split almost evenly between the injection and reception sides. The network logic incurs more than half of its latency processing the network injection and reception FIFOs, whereas the latency for allocating the link and performing data transfer is relatively small. In the production software stack, there is additional overhead, such as for copying the descriptor into the

IM FIFO and pulling packets from the reception FIFOs.

Large-scale parallel benchmarks such as Linpack (http://www.top500.org/list/2011/11/100) and the Graph500 (http://www.graph500.org) are already taking advantage of the efficient design of the Blue Gene/Q network and message unit. BG/Q systems have ranked first for power efficiency on the three most recent Green500 (http://www.green500.org) lists (November 2010 through November 2011). A relatively small 4,096-node BG/Q ranked first on the network and data-intensive Graph500 benchmark in November 2011, achieving 254 billion traversed edges per second. As the machine becomes more widely deployed, performance data on a broad set of applications will become available. MICRO

Acknowledgments

The Blue Gene/Q project has been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the US Department of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331. We acknowledge the collaboration and support of Columbia University and the University of Edinburgh. Blue Gene is a team effort. We especially thank the following for their contributions: Ralph Bellofatto, Peter Boyle, Arthur Bright, George Chiu, Paul Coteus, Mark Giampapa, Tom Gooding, Ruud Haring, Michael Kaufmann, Kyu-Hyoun Kim, Gerry Kopcsay, Luis Lastras, Jim Marcella, Moyra McManus, Tom Musta, Ben Nathanson, Martin Ohmacht, Bryan Rosenburg, Krishnan Sugavanam, Todd Takken, Jim Van Oosten, Amith Mamidala, Jose Brunheroto, Gabor Dozsa, Doug Miller, Michael Blocksome, and Brian Smith.

References

1. IBM Blue Gene Team, ''IBM Blue Gene/Q Compute Chip,'' Hot Chips Conf. (Hot Chips 23), 2011; http://www.hotchips.org/conference-archives/hot-chips-23.
2. R.A. Haring et al., ''The IBM Blue Gene/Q Compute Chip,'' to be published in *IEEE Micro,* Mar./Apr. 2012.
3. D. Chen et al., ''The IBM Blue Gene/Q Interconnection Network and Message Unit,'' *Proc. Int'l Conf. High Performance Computing, Networking, Storage and Analysis* (SC 11), ACM Press, 2011, article 26.
4. A. Gara et al., ''Overview of the Blue Gene/L System Architecture,'' *IBM J. Research and Development,* vol. 49, nos. 2-3, 2005, pp. 195-212.
5. IBM Blue Gene Team, ''An Overview of the Blue Gene/P Project,'' *IBM J. Research and Development,* vol. 52, nos. 1-2, 2008, pp. 199-220.
6. N.R. Adiga et al., ''Blue Gene/L Torus Interconnection Network,'' *IBM J. Research and Development,* vol. 49, nos. 2-3, 2005, pp. 265-276.
7. S. Scott and G. Thorson, ''The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus,'' *Proc. Symp. High Performance Interconnects* (Hot Interconnects IV), IEEE CS Press, 1996, pp. 147-156.
8. R. Alverson, D. Roweth, and L. Kaplan, ''The Gemini System Interconnect,'' *Proc. 18th IEEE Symp. High Performance Interconnects* (HOTI 10), IEEE CS Press, 2010, pp. 83-87.
9. Y. Ajima et al., ''The Tofu Interconnect,'' *Proc. IEEE 19th Ann. Symp. High Performance Interconnects* (HOTI 11), IEEE CS Press, 2011, pp. 87-94.
10. R. Rajamony, L.B. Arimilli, and K. Gildea, ''PERCS: The IBM POWER7-IH High-Performance Computing System,'' *IBM J. Research and Development,* vol. 55, no. 3, 2011, article 3.
11. V. Puente et al., ''Adaptive Bubble Router: A Design to Improve Performance in Torus Networks,'' *Proc. Int'l Conf. Parallel Processing* (ICPP 99), IEEE CS Press, 1999, pp. 58-67.
12. S. Kumar et al., ''Optimization of All-to-All Communication on the Blue Gene/L Supercomputer,'' *Proc. 37th Int'l Conf. Parallel Processing* (ICPP 08), IEEE CS Press, 2008, pp. 320-329.
13. A. Widmer, *Transmission Code Having Local Parity,* US patent 5,699,062, to IBM Corp., Patent and Trademark Office, 1997.

Dong Chen is a research staff member on the Blue Gene supercomputer project at the IBM T.J. Watson Research Center in

Yorktown Heights, New York. His research interests include the architecture and design of Blue Gene systems and their applications. Chen has a PhD in physics from Columbia University. He is a member of IEEE.

Noel A. Eisley is a research staff member on the Blue Gene supercomputer project at the IBM T.J. Watson Research Center. His research interests include interconnection networks, chip multiprocessors, and networks on chips. Eisley has a PhD in electrical engineering from Princeton University. He is a member of IEEE and the American Association for the Advancement of Science (AAAS).

Philip Heidelberger is a research staff member on the Blue Gene supercomputer project hardware team at the IBM T.J. Watson Research Center. His research focuses on various aspects of Blue Gene, including network architecture and design, logic verification, hardware bring-up, network software interfaces, and efficient communications algorithms. Heidelberger has a PhD in operations research from Stanford University. He is an IBM master inventor, a member of the IBM Academy of Technology, and a fellow of IEEE and the ACM.

Robert M. Senger is a research staff member on the Blue Gene supercomputer project at the IBM T.J. Watson Research Center. His research interests include low-power architectures and circuits, interconnection network design, and messaging hardware interfaces. Senger has a PhD in electrical engineering from the University of Michigan, Ann Arbor. He is a member of IEEE.

Yutaka Sugawara is a research staff member on the Blue Gene supercomputer project at the IBM T.J. Watson Research Center. His research focuses on interconnect and host interface architecture for high-performance computing systems. Sugawara has a PhD in computer science and technology from the University of Tokyo. He is a member of IEEE.

Sameer Kumar is a research staff member and the research lead of the messaging-software stack for Blue Gene/Q at the IBM T.J. Watson Research Center. His research interests include designing message-passing software on million-processor machines, optimizing performance for parallel applications, and designing next-generation supercomputer interconnection networks. Kumar has a PhD in computer science from the University of Illinois at Urbana-Champaign.

Valentina Salapura is a system architect at the IBM T.J. Watson Research Center, where she works in the Services Innovation Lab. Her research interests include cloud computing; data center design and operation; virtualization and virtual-machine provisioning; multiprocessor interconnects; multiprocessor synchronization; and multithreaded, multicore architecture design and evaluation. Salapura has a PhD in computer science from Technische Universität Wien in Vienna. She is an IBM master inventor, a member of the IBM Academy of Technology and the ACM, and a fellow of IEEE.

David L. Satterfield is a senior engineer on the Blue Gene supercomputer project at the IBM T.J. Watson Research Center. His research interests include the architecture and design of Blue Gene systems and their applications. Satterfield has a BS in electrical engineering from Tufts University.

Burkhard Steinmacher-Burow is a senior technical staff member at IBM Germany Research & Development. His research interests include Blue Gene hardware and software systems, particularly network and memory systems. Steinmacher-Burow has a PhD in physics from the University of Toronto.

Jeffrey J. Parker is an operating-system software engineer at the IBM Systems & Technology Group in Rochester, Minnesota. His technical interests include the IBM System/36 and iSeries systems, Blue Gene/L I/O performance, and Blue Gene/P and Blue Gene/Q messaging software. Parker has a BS in computer science from the University of Minnesota.

Direct questions and comments about this article to Robert M. Senger, IBM T.J. Watson Research Center, 1101 Kitchawan Rd., Rt. 134, Yorktown Heights, NY 10598; rmsenger@us.ibm.com.