RC 21999 (98824) 19 March 2001 **Computer Science**

IBM Research Report

Developing System Software for Blue Gene

George Almasi, Calin Cascaval, Jose G. Castanos, Derek Lieber, Jose E. Moreira
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

This page intentionally left blank.

Developing System Software for Blue Gene

George Almasi Călin Caşcaval José G. Castaños Derek Lieber José E. Moreira
{galmasi,cascaval,castanos,lieber,jmoreira}@us.ibm.com
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598-0218

Abstract

In this paper we introduce the software development environment for Blue Gene, a massively parallel system being developed at the IBM T. J. Watson Research Center. While the hardware is still in a development phase, we are able to provide a complete system software stack consisting of compiler, kernel, run-time libraries, and visualizer that can execute many user-level applications such as the Splash-2 benchmark suite. These applications execute in an instruction-level simulator that we have developed to validate our architecture and tools. We analyze the data obtained from these runs and make performance estimates that influence architectural decisions.

Keywords: cellular architecture, parallel computing, simulation, trace visualization, performance evaluation

1 Introduction

Blue Gene is a massively parallel system of unprecedented scale. With a projected peak performance of 1 Petaflop/second (10^{15} double-precision floating-point operations per second), Blue Gene will enable significant advances in computational science and engineering. The scale and complexity of Blue Gene give rise to many technical problems that have to be solved before the project can be successful. In this paper, we address the challenges of developing correct, efficient system software for Blue Gene.

The high levels of performance in Blue Gene are achieved through massive parallelism. Full exploitation of the machine by a single application requires decomposing that application into up to 8 million concurrent threads of execution. Parallelism on this scale can only be implemented economically if we adopt a *cellular architecture* approach. The machine is built through regular replication of a basic module which contains processing logic, memory, and interconnection hardware. On a machine of this scale it is certain that some of the components will be broken at any given time, with a fairly high (order of days) hard-failure rate. Despite these failures, the machine as a whole must continue to operate effectively.

In addition to the unusual scale, the Blue Gene hardware implements a new instruction set architecture (ISA). Currently there is no hardware that can execute this instruction set. Yet, we are faced with two important missions: (i) verifying correctness and performance of the architecture and providing feedback to the hardware designers, and (ii) developing a complete system software stack, which must be ready when initial operational hardware is built. Examples of system software components that comprise this stack include compilers, resident run-time kernel, user-level libraries (math, message-passing, shared-memory, input/output), and debugging and performance analysis tools. Also of great importance are software components that handle job scheduling, system test, boot, and management, and file system management. These components are expected to run on a host system attached to the Blue Gene core. Finally, it is necessary to

develop or port benchmark applications to Blue Gene, both to investigate the performance characteristics of the machine and to explore different programming models.

Our approach to developing and assessing software components for Blue Gene is based on multiple levels of simulation. At the highest level, the Blue Gene EMulator (BGEM) implements the API exported by the Blue Gene resident kernel on a Linux cluster. Thus all layers above the kernel, including user-level libraries (*e.g.*, communications, shared-memory, input/output) can be developed and tested on conventional machines, using native tools.

At the next level of detail, the Blue Gene SIMulator (BGSIM) is a complete architecture-level simulator that directly executes binaries produced by the Blue Gene compiler. Although not cycle accurate, BGSIM models the performance characteristics of Blue Gene, leading to detailed performance predictions. The Blue Gene kernel itself was developed and tested with BGSIM. Most of the time, a simple recompilation is all it takes to move a software component from BGEM to BGSIM. The Blue Gene VISualizer (BGVIS) attaches to BGSIM to provide a graphical interface to the inner workings of Blue Gene. With BGVIS it is possible to observe the behavior of an application over time, identifying critical performance characteristics. BGSIM and BGVIS are the main tools we have for developing, testing, and tuning Blue Gene code, from libraries to applications. For that reason, they are described in greater detail in this paper.

The most detailed simulator for Blue Gene is a gate-level simulation obtained directly from the logic design of Blue Gene. This simulator is cycle accurate but, because of the complexities of gate-level simulation, it is restricted to simulating only small Blue Gene configurations. The gate-level simulator is not ready yet. We expect to use it to validate the Blue Gene kernel and to investigate the behavior of some small benchmarks.

The rest of this paper is organized as follows. Section 2 describes the Blue Gene reference architecture. Section 3 describes the Blue Gene simulator (BGSIM) that we use to validate and assess the behavior of Blue Gene code. Section 4 (BGVIS) presents details of the Blue Gene visualizer, on which we rely to better understand what happens inside Blue Gene. Section 5 describes the main components of the system software stack we have developed so far. Section 6 discusses results from running parallel benchmarks through our system software stack on BGSIM. Finally, Section 7 presents conclusions and future work for this activity.

2 The Blue Gene Reference Architecture

In this section we describe an early prototype of the Blue Gene architecture which was used during most of our system software development. This is also the architecture for which we discuss experimental results in Section 6.

The main characteristic of the Blue Gene design is the integration of embedded DRAM and processing logic on the same chip. The proximity of memory and processors results in a flat memory hierarchy which overcomes the Von Neumann bottleneck (processor performance improves faster than the capacity of memory to serve them) observed in conventional designs. Even with the memory embedded on the chip, accessing a memory block takes more than one cycle. The solution adopted by Blue Gene is to use multiple threads in a single chip so that, if a thread stalls after a memory reference, other threads can make progress.

The building block of our cellular architecture is a *node*. A node is implemented in a single silicon chip and contains memory, processing elements, and interconnection elements. A node can be viewed as a single-chip shared-memory multiprocessor. In the design simulated in this paper, a node contains 16 MB of shared memory and 256 instruction execution units. Each instruction execution unit is associated with one thread of execution, giving 256 simultaneous threads of execution in one node, and executes instructions in a thread strictly in order. Every thread unit includes a register file (with its own program counter) and a fixed point unit for integer and logical operations. The register file contains 64 32-bit registers which can be paired to form double precision registers.

Each group of 8 threads (a *processor*) share one instruction cache and one floating-point unit. Such simplifications are necessary to keep instruction units simple, resulting in large numbers of them in a die. The instruction caches have two levels, with the L1 I-cache implemented in SRAM (8 KB) and the L2 I-cache in DRAM. The floating-point units (32 in a node) are pipelined and can perform a multiply and an add on every cycle. With a 500 MHz clock cycle, this translates into 1 Gflop/s of peak performance per floating-point unit, or 32 Gflop/s of peak performance per node. The thread units compete for access to their shared floating point unit. Hardware selects a winner if two threads try to issue floating point instructions that use the same functional unit in the same cycle.

The in-chip memory is organized in 32 banks of 4 Megabits of EDRAM, giving a total of 16MB per chip. This memory is globally addressable and shared by all threads. The D-caches are 8KB each and implemented in SRAM. These caches are on the memory side (one cache per pair of DRAM banks) rather than on the processor side. Thus, there is no cache coherence problem.

Blue Gene processors do not support out-of-order execution, speculative execution or register renaming. Instead, extensive multithreading is used to cover latencies. In a machine like this, performance is derived from its massive parallelism rather than by improving the speed of one particular task. Blue Gene uses a new instruction set. This instruction set is very simple, it contains approximately 100 instructions. This ISA can be characterized as a three-operand, load/store RISC architecture. In addition it has instructions that are appropriate for parallel processing, such as test-and-set and sync.

Multiple nodes are interconnected in a regular fashion to form a much larger system. Each node has six channels of communication, for direct interconnection with up to six other nodes, forming a three-dimensional mesh. (Nodes on the faces or along the edges of the mesh have fewer connections.) We use a mesh topology because of its regularity and because it can be built without any additional hardware. With a $32 \times 32 \times 32$ three-dimensional mesh of nodes, we build a system of 32,768 nodes. Since each node attains a peak computation rate of 32 Gflop/s, the entire system delivers a peak computation rate of approximately 1 Petaflop/s. We directly connect the communication channels of a node to the communication channels of its neighbors. Nodes communicate by streaming data through these channels. With 16-bit wide channels operating at 500 MHz, a communication bandwidth of 1 GB/s per channel, on each direction, is achieved.

Communication in Blue Gene is by message passing with fixed length packets of up to 256 bytes. The network hardware in the chip provides a fixed number input and output buffers. To send a packet, the computation thread copies the payload into an empty output buffer and marks it ready. Software also specifies the route followed by the packet through the mesh until it appears in an input buffer of the destination node. All the routing is performed by hardware: threads in intermediate nodes do not get interrupted. The network guarantees message delivery but not message arrival order even if two messages follow the same route.

Our design for a node is ambitious, but within the realm of current or near-future silicon technology. Combined logic-memory microelectronics processes will soon deliver chips with hundreds of millions of transistors. Several research groups have advanced processor-in-memory designs that rely on that technology. Examples include the Illinois FlexRAM [6, 12], Berkeley IRAM [11] and Gilgamesh [15] projects.

An application running on this Petaflop machine must exploit both inter- and intra-node parallelism. First, the application is decomposed into multiple tasks and each task assigned to a particular node. As discussed previously, the tasks can communicate only through data streams. Second, each task is decomposed into multiple threads, each thread operating on a subset of the problem assigned to the task. The threads in a task interact through shared-memory. The results presented in Section 6 only address the latter form of parallelism. For results on the former form of parallelism refer to [1].

3 Blue Gene Simulator

The Blue Gene simulator (BGSIM) is an architecturally accurate simulator. Because Blue Gene is a completely new and complex design, we need tools to validate new architectural features and to test software before the hardware is available. One could argue that a gate-level simulator provides a more accurate view of hardware behavior. However, there are two major drawbacks in using a gate-level simulator in early stages of the development process. First, gate-level simulation needs the hardware design to be completed because the simulator is based on the actual VLSI specification. Often, this leads to using the gate-level simulator as a correctness tool, i.e., changes are made only if the simulation finds functional errors, mostly because architects are reluctant to introduce major changes in the design, a costly process once a layout is obtained. The second disadvantage is the speed of gate-level simulation. A typical VHDL simulator can execute about 1000 machine cycles per second, and it can simulate only small systems (for example, the AWAN simulator can simulate systems up to four Blue Gene nodes). For these reasons, a relatively fast (between 60,000 and 120,000 machine cycles per second, depending on the number of threads simulated) architectural simulator such as BGSIM, allows us to run real applications and measure the impact of different architectural features on these applications.

Traditionally, simulators have been categorized into two classes: *trace-driven* and *execution-driven.* Trace-driven simulators use traces of instructions or memory references of applications produced on different machines to analyze the behavior of proposed architectural features. Although the traces capture the behavior of realistic applications, it is hard to account for timing differences between the simulated system and the system on which the traces were collected [7, 4]. Execution-driven simulators [3, 5, 13, 9, 8, 10] interleave the execution of the application instructions and escapes in the simulator at specific events to model the target architecture. Execution-driven simulators usually simulate target architectures with ISAs very close to the host system.

BGSIM is an *instruction-driven* simulator. It interprets the instructions in the Blue Gene instruction set, and, although it does not model the microarchitecture of the Blue Gene processors, it executes each instruction inside the simulator. We have taken this approach because the Blue Gene ISA is a new instruction set and it is radically different from the x86 instruction set on which the simulator is running.

3.1 Features

BGSIM executes instructions from the Blue Gene instruction set, modeling resource contention between instructions, and thus, estimating the number of cycles each instruction executes. The simulator models the sharing of resources at all levels in the node's hierarchy. Register files are shared by instructions, instruction caches and floating point units are shared by threads, data caches, memory and communication switches are shared by processors. Also, buses inside the chip are shared by all processors on the node. The simulator is parametrized such that different architectural features can be specified when the program runs. Examples of parameters that can vary are: the number of thread units in a processor, the amount of memory in the chip, the size, line size and associativity of the caches.

BGSIM produces instruction traces, instruction histograms and resource utilization statistics, such as floating point unit usage and contention, cache hit and miss ratio, memory accesses and contention. The outputs can be turned on and off from the user program, using instructions addressing unarchitected special registers (escapes into the simulator). The statistics can be collected at thread unit level or node level (i.e., for all the thread units in the node). Instruction traces can be visualized on-line or analyzed off-line by other tools, such as BGVIS presented in Section 4.

3.2 Implementation

The Blue Gene simulator is an object-oriented design, implemented in C++. BGSIM models the hierarchical nature of the Blue Gene chip, having objects for most of the hardware features (one feature that we do not model is the JTAG/scan-in mechanism for the chip, because this mechanism is irrelevant for the application performance). For example, there are objects that implement the thread unit, the memory, the data and instruction caches, and communication switches. All the resources available to instructions are modeled with default parameters for latencies, and the simulator adjusts these latencies depending on the contention.

Blue Gene instructions are also modeled as objects, and have three parameters: *execution time*, *pipeline delay* and *resource delay*. The *execution time* represents the number of cycles for which the execution unit is used exclusively by the instruction. The *pipeline delay* represents the number of cycles in which the instruction executes in the execution unit's pipeline, thus being able to share the execution unit with other instructions. The *resource delay* is the number of cycles that an instruction has to wait for resources (registers, execution units, memory banks, etc.) to become available. For all instructions except memory instructions, the execution time and the pipeline delay are given by the architectural specifications. In the case of memory instructions, the execution time is fixed (the time to execute in the load-store unit); the pipeline delay depends on where the addressed datum is located in the memory hierarchy. The resource delay is also variable, depending on resource availability.

The simulator models the instruction execution in three phases. First, an instruction probes resource availability in the thread unit, or, depending on the resource requirements, in the processor or in the chip. There are three outcomes possible: (i) the resources are available, thus the instruction can start executing; (ii) the resources are not available but the resource delay can be computed. In this case, the instruction stalls the thread for resource delay cycles (the thread units issue instructions in program order, and although they can execute several independent instructions per cycle, the instructions are also retired in program order), and it starts executing after that many cycles have passed; (iii) and finally, the resources are not available, and the delay can not be computed. For example, atomic instructions block the access to a memory location until the operation is completed, thus another thread will keep the "resource" (the memory location) busy for an unspecified amount of time. In this case, the instruction stalls the thread unit for one cycle, and it will probe for resources again.

The next phase in the instruction execution is starting the execution. In this phase the instruction allocates resources for execution. And finally, there is a completion phase, in which the results are computed and stored in registers (or memory). Also in this phase the resources are deallocated.

BGSIM loads the Blue Gene kernel (see Section 5), which in turn, loads and runs the application program. The application is optionally augmented with simulator escapes in order to produce results for selected regions of the code. Otherwise, results for the whole program execution, including the booting of the kernel, are reported.

Because of the non-trivial size of the Blue Gene machine, the simulator is designed to simulate several Blue Gene nodes in one process and several instances of the BGSIM collaborate using MPI, each simulating a subset of the machine. Blue Gene nodes communicate with each other using communication channels. BGSIM simulates these channels. Packets in the channel are routed to the appropriate node objects, intra-process if the nodes are simulated on the same BGSIM process, or using MPI messages if the nodes are simulated on different BGSIM processes.

The platform for our simulation is a Linux cluster, comprised of 80 dual processor PIII nodes, running at 600 MHz. On this system we have simulated up to 400 Blue Gene nodes running a molecular dynamics application [1]. The simulator runs at a speed of 60 to 120 KHz on the cluster, depending on the number of threads simulated. This corresponds to a 8000-4000 slowdown compared to the Blue Gene hardware. However, the main goal of the current implementation is functionality not performance. We hope to report

much better results in the final version of the paper.

4 Blue Gene Visualizer

The Blue Gene Visualizer (BGVIS) is the "face" of Blue Gene. Traditional debuggers and performance analyzers with command line interfaces cannot cope with the level of complexity and the size of Blue Gene. Even graphical debuggers will have a very hard time displaying information on 8 million threads running simultaneously. BGVIS is our attempt to develop tools that can help understanding the behavior of massively parallel machines while executing programs. It has already shown its usefulness by helping decide between 32 and 64 bytes cache lines for the data cache. Second, the visualizer can help identify bottlenecks in the application, and here again, we have shown that just by looking at the instruction mix for all the threads in the node (explained below) we can immediately pinpoint load imbalance between different threads.

The visualizer is designed to work with the BGSIM to show and allow assessment of how the user's program is running on the machine. BGVIS consumes traces produced by the simulator, and displays graphically and audibly the status of different components of the machine. It provides hierarchical views of the machine, customizable to the desired level of details needed by the analysis. It shows different parameters such as: instruction mix, data and instruction cache miss ratio, the state of cache lines and the contents of the registers in the register file, the line in the high level source code which is currently executing, etc. The visualizer displays two types of values: cumulative and instantaneous. Cumulative values are aggregated over the course of execution, for example instruction mixes and cache miss ratios. Instantaneous values are computed over an adjustable window of instructions. These values provide valuable feedback on the dynamic behavior of the program, since the values presented to the user are not averages over a large number of instructions that hide the transitory effects.

The main panels of the graphical interface are illustrated in Figure 1. We can explain better the functionality of the visualizer with a usage scenario: consider an application that uses all 256 threads in one node, out of which two threads, on two different processors, are to be followed more closely. This view is reflected in the top main window of the visualizer, where the two processors are highlighted on the right panel. The left panel shows the cumulative instruction mix for all the threads in the chip. The small windows correspond to the threads showing the line in the source code that corresponds to the current instruction executed by each thread. The central window in Figure 1 displays more details on the execution in one of the processors. Shown in the figure is mainly the cache behavior for this processor. The state of the cache lines in both instruction and data caches is color coded. Also, a time-line with values for certain parameters (the ratio of floating point operations to total instructions, cache miss ratios, etc.) is displayed on the right.

Another interesting feature of BGVIS is its ability to single step through the trace allowing the user to focus on a greater level of details, without being afraid of missing important information. This feature, together with the linking of the current instruction back to high level source code, makes the visualizer invaluable for debugging and performance tuning.

Not shown in the figure, but of much interest, are the sound effects. Since there are so many components in Blue Gene, we believe that, to provide a total view, we must "immerse" the user in the tool, thus we must act upon all the senses. We have started with sounds, and for example, we have used the frequency of dirty line cast-outs as an audible indicator of memory bus traffic. The Java sound and MIDI classes provide many ways to provide audible feedback that is effective, pleasant and maybe even musically interesting.

Written in Java using the JDK1.3 swing and sound classes and the Sitraka JClass chart classes, the visualizer is fast enough to accept output directly from the simulator, a useful feature that allows the option of not saving trace files to disk. We will provide direct integration of these tools in the future. The hierarchy of objects in the program reflects that of the hardware, to accommodate design updates. The graphs can be edited interactively and saved in several formats. A client/server mode is provided whose communication

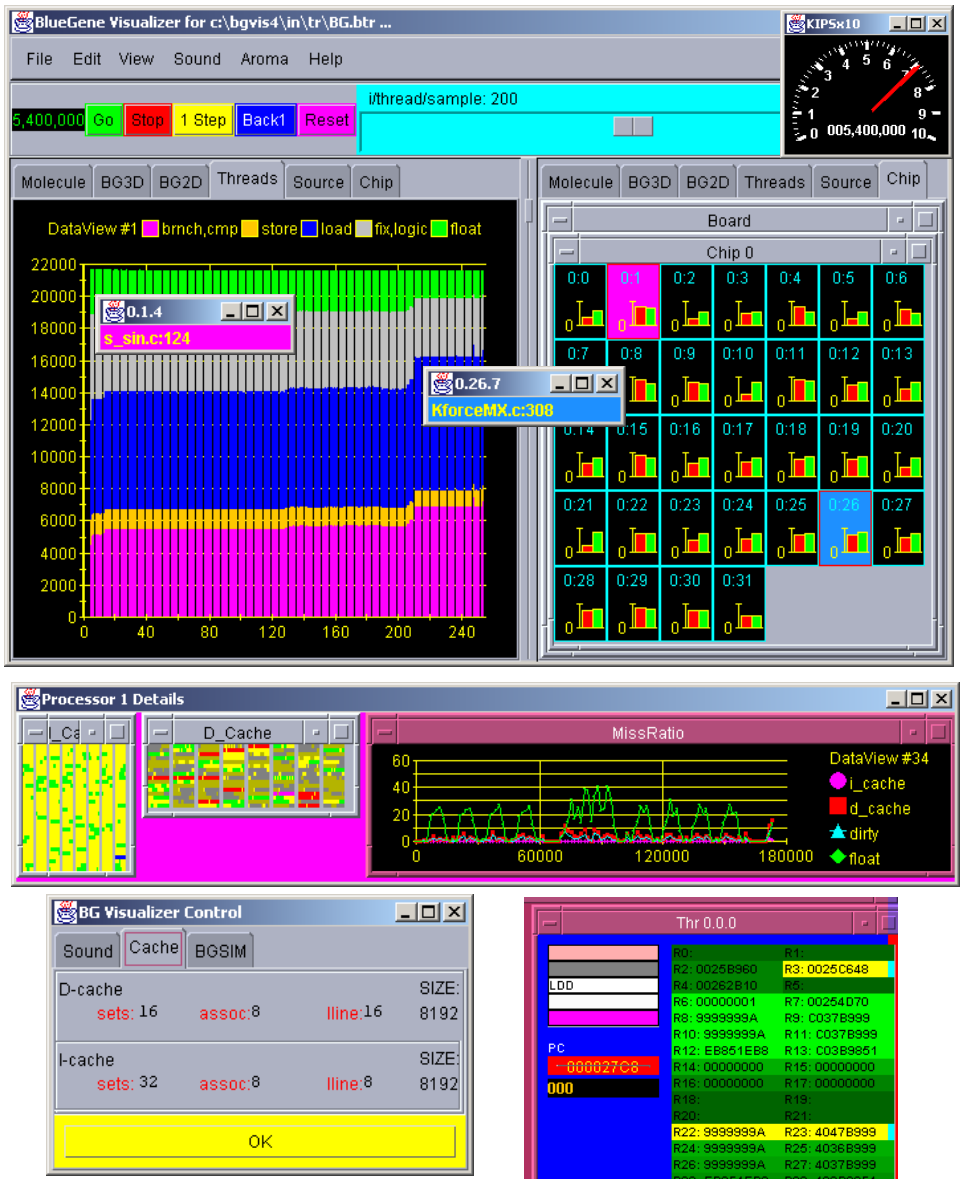

Figure 1: The Blue Gene visualizer. Top: the main window. The left panel shows a thread instruction-mix histogram, with two open thread source-line windows. The corresponding processors are highlighted in the right panel. Center: processor details panel, with cache states and miss ratio history. Lower left: control panel. Lower right: register details.

protocol is sufficiently spartan to allow full functionality even with only a 28.8K modem connection. We are continuing to work with the visualizer and sonifier to add expanded debugging capabilities and novel modes appropriate to the massive parallelism of Blue Gene.

5 System Software Stack for Blue Gene

The Blue Gene system software stack consists of a compiler, kernel and runtime libraries. These tools provide a programming and operating environment within the constraints dictated by the Blue Gene hardware. With the addition of the Blue Gene simulator (Section 3) and visualizer (Section 4) our environment permits the development of applications for Blue Gene. We use the results obtained from simulations to validate the architecture, to measure the scalability of applications on Blue Gene and to evaluate the impact of architectural changes.

Our environment exports a familiar (POSIX) interface without the complexity of a full Unix system residing on each node. In Blue Gene, only the most critical system tasks are handled by the computational core. The remaining support routines execute on an attached host, a conventional cluster. As a result, the resident portion of the system software stack is small (30 KB of memory and 3,000 lines of C code) and simple when compared with current operating systems and it is well adapted to the limited resources available in each node. Its simplicity provides the high reliability necessary to effectively manage several thousand nodes. It also delivers high performance since it does not require traversing many layers of software to access the hardware.

The kernel, libraries, and applications are currently being generated with a cross-compiler based on the GNU toolkit, re-targeted for the Blue Gene instruction set architecture. This cross-compiler supports C, C++ (including the Standard Template Library), and FORTRAN 77.

The resident runtime (or *kernel*) supports single user, single program applications within each Blue Gene partition. Its purpose is two-fold: first, to protect machine resources (threads, memory, communication channels) from accidental corruption by a misbehaving application program, so that resources can be used reliably for error detection, debugging, and performance monitoring; and second, to isolate application libraries from details of the underlying Blue Gene hardware and host communications protocols, so as to minimize the impact of evolutionary changes in those areas.

The kernel executes with supervisor privileges. Context switching between user and supervisor mode is inexpensive, requiring a few cycles. The kernel provides a thin interface to the hardware for communication, synchronization, timers, and interrupts. Data is exchanged with other nodes via message passing. File system input/output and program loading is accomplished via message passing to externally attached host computers, which appear as *virtual* nodes on the external faces of the Blue Gene cube.

The kernel exposes a single-address space shared by all threads. Due to the small address space and large number of hardware threads available, no resource virtualization is performed in software: virtual addresses map directly to physical addresses (no paging) and software threads map directly to hardware threads. The kernel does not support preemption (except in debugging mode), scheduling or prioritization. Every software thread is preallocated with a fixed size stack (currently 2KB per thread), resulting in fast thread creation and reuse.

We support inter-node communication using an active message mechanism for delivering packets closely matching Blue Gene hardware. The system supports a variable number of computation and communication threads. At system level messages are created by providing the payload, the route that the message follows to its destination, and the address of a function to be executed by a communication thread on the destination node upon message arrival. Communication threads continuously poll input buffers until an input buffer is marked full. A communication thread processes the incoming packet by invoking the function specified in the message, supplying the payload as an argument.

As a convenience to programmers porting code from other machines, we provide a C library (libc) with familiar Unix-style functions for file access: fopen, fread, fwrite, fprintf, etc. These operations are implemented with function shipping: the actual file access is performed on the host; Blue Gene nodes route user requests to the host using the Blue Gene communication fabric. We also provide a subset of

the pthreads API for thread management and synchronization. Higher performance can be obtained by using application libraries that exploit specific features of the Blue Gene hardware, such as intra-chip thread barriers and interrupts. To simplify the development of parallel applications we implemented a message passing library with functions for many common parallel programming idioms (broadcasts, reductions, etc) which are tailored to the needs of fine grain communications.

6 Parallel Benchmarks to Validate the Software Stack

To verify and measure the behavior of our entire system software infrastructure, we selected the Splash-2 benchmark suite [14]. This suite is a collection of applications used to study the characteristics and performance of shared memory multiprocessors. These applications cover a wide range of areas which allows us to compare and analyze variations of our basic hardware design using a representative set of codes.

In this section we describe data produced by our simulations. It is essential to emphasize that these codes are not optimized for Blue Gene. For this reason, the results should not be taken as an absolute measure of the capabilities of our hardware. On the other hand, they demonstrate our ability to run real applications on the software stack and allow us to evaluate architectural characteristics. Although a single Blue Gene chip has a peak performance comparable to high performance servers available just a few years ago, the Blue Gene architecture has many significant differences from traditional architectures, such as limited memory, shared functional units and unconventional caches. These sample codes do not take advantage of these features.

The set of applications that are part of Splash-2 are summarized in Table 1. They consist of 4 simple kernels (FFT, blocked LU decomposition with and without contiguous blocks, Cholesky decomposition and radix sort) and 8 more complex applications which range between 1,000 and 12,500 lines of C code. These codes support synchronization between multiple threads using barriers and protect access to shared data with locks. To simplify the portability of the suite between different architectures the synchronization primitives are coded using PARMACS [2] macros.

After defining the macros for Blue Gene, the modifications required to port the codes were minimal. Difficulties appeared in just two codes: one that invokes a function recursively in the set-up phase – cholesky; another thet allocates large vectors from the stack – raytrace. Both these codes overrun our small stacks and were changed. When possible, we used the default inputs for each benchmark. In several situations, because of memory limitations, we had to use a reduced problem size. The input parameters are presented in Table 1, third column.

Figure 2 shows the speedups relative to single thread execution for most of the Splash-2 codes. For simplicity, many benchmarks execute on a number of processors that is a power of two. In Blue Gene, it is necessary to reserve at least one thread to handle communication. Thus, the maximum number of computation threads we used to run Splash-2 without major rewriting is 128.

The kernel currently supports a simple thread allocation policy: software threads are mapped to hardware threads sequentially. As a result, the sharing of resources (floating point units and caches) is not optimal. For example, when executing these tests with 64 threads, only 8 FPUs out of the 32 FPUs in a node are actually used. Finding and expressing optimal policies for thread allocation is one of our areas of future research.

The resources available to the 256 threads in a Blue Gene node are not scaled by the same number of threads, i.e., threads share resources that in a more traditional architectures are not shared. Even with these constraints, we obtained speedups similar to the ones reported in [14] for a number of benchmarks: FFT, LU, water-nsquared, water-spatial and raytrace. Other benchmarks, cholesky, barnes, and ocean, show speedups within a factor of two compared to [14].

Table 1: Description of the Splash-2 kernels and applications and input parameters selected.

Problem	Description	Input
cholesky		tk14.o
fft	1D FFT	forward and reverse FFT of 64K complex numbers
lu	blocked LU decomposition	256 by 256 matrix of doubles
radix	radix sort	256K keys that can take 512K values and a radix of 1024
barnes	Barnes-Hut hierarchical n-body simulation	2048 bodies, 3 time steps
fmm	Fast Multipole	2048 bodies, uniform distribution, 5 time steps
ocean	simulation of ocean movements with Gauss-Seidel multigrid solver	130 by 130 grid size
radiosity	iterative hierarchical method	test problem
raytrace	3D scene rendering with hierarchical grid	teapot
volrend	render a volume with raycasting	head-scaleddown2
water-nsquared	molecular dynamics of a water box ($O(n^2)$ algorithm)	512 water molecules, 3 time steps, 6.21A cutoff radius
water-spatial	molecular dynamics of a water box using a spatial decomposition	512 water molecules, 3 time steps, 6.21A cutoff radius

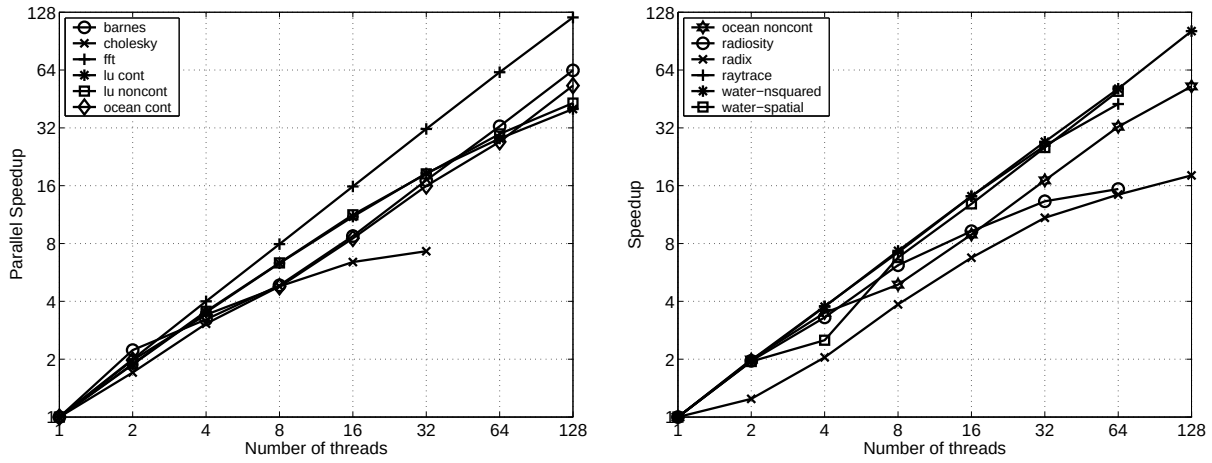

Figure 2: Parallel speedups of Splash-2 kernels and applications.

BGSIM reports the number and type of instructions per thread and per node. An example of these outputs is illustrated in Figure 3 for the barnes and water-nsquared benchmarks. The instructions in the ISA are grouped into eight categories: loads and stores, integer and floating point operations, logical operations, branches, atomic operations (test-and-set) and system instructions (kernel traps and other special

instructions). These instruction histograms are used analyze the scaling of the application. For example, the histogram for barnes for 64 and 128 threads shows a significant increase in the number of memory and branch instructions for a similar amount of floating point operations. On the other hand, the water-nsquared histogram shows better scalability – about the same total number of instructions is executed for all thread configurations.

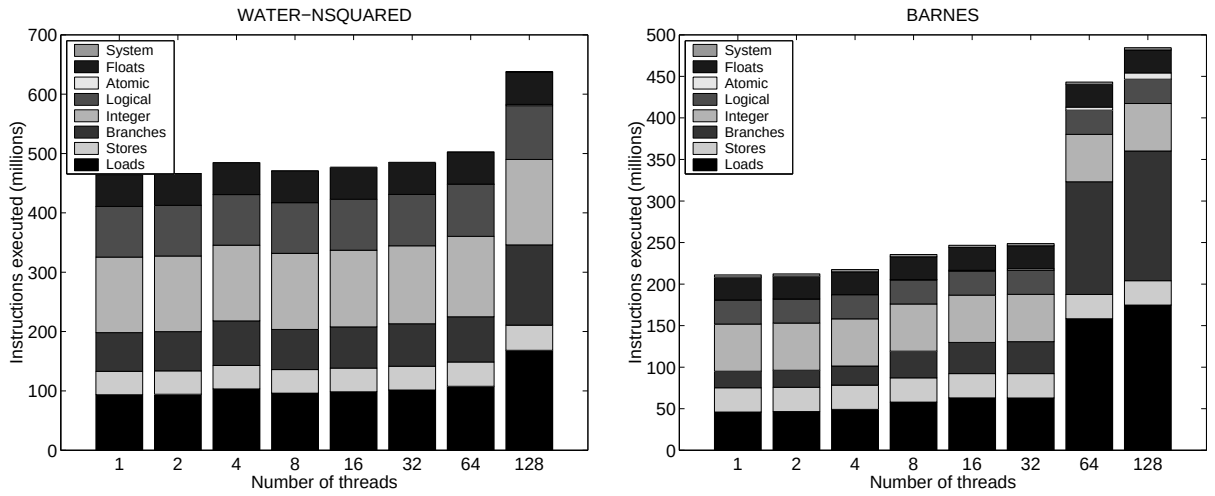

Figure 3: Instruction histograms for the water-nsquared (left) and barnes (right) benchmarks.

During execution, thread units can be in a number of states: running, stalled waiting for resources (either FPUs or data from memory), executing branch instructions or fetching instructions. The total number of cycles spent by all computation threads in each of these states for the fft and radix benchmarks is presented in Figure 4.

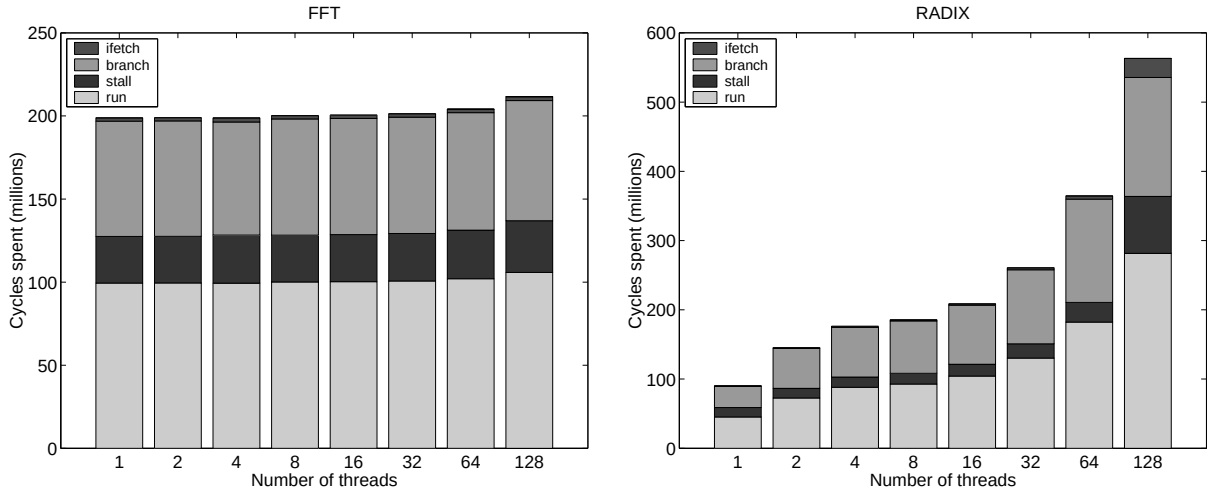

Figure 4: Breakdown of thread runtime for the fft (left) and radix (right) benchmarks.

The cache simulation results summarized in Figure 5 are obtained from BGVIS. The figure shows the

cache miss ratio versus cache size and associativity for 32 and 64 bytes cache line for the FFT benchmark. In this benchmark the extra hardware needed for 8-way associativity in the data cache paid for itself in terms of improved cache miss ratio, and revealed that a longer cache line (64 bytes vs. 32 bytes) is also a better choice.

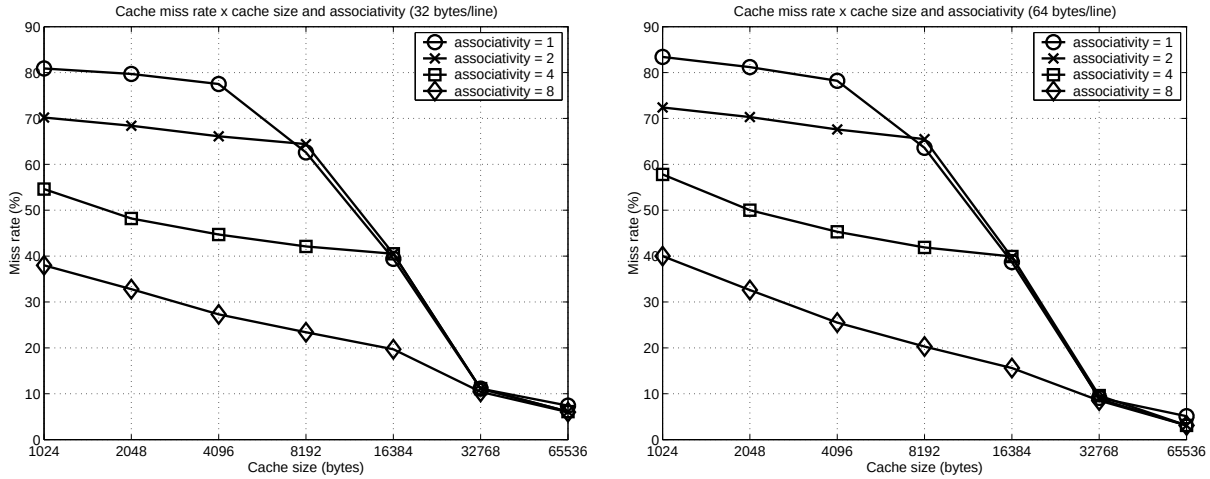

Figure 5: Simulated data cache miss ratios vs. cache parameters for FFT forward only, 64K complex numbers.

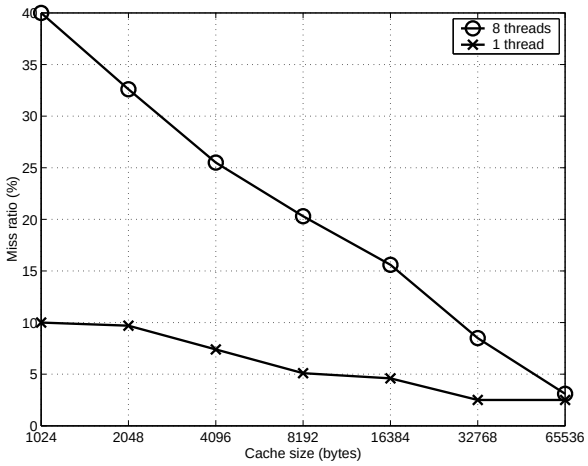

Figure 6: Effect of multiple threads sharing data cache for the FFT benchmark.

Figure 6 compares the miss ratios when only a single thread is using the data cache vs. the case in figure 5, where 8 threads are sharing the data cache. The single-thread results are consistent with earlier Splash-2 simulation results published for the Stanford DASH – for example, the miss ratio for an 8KB cache for the FFT problem is about 5% in both cases. This miss ratio rises to 20% when 8 threads share the cache, a less than linear increase with the number of threads. However, it should also be noted that one thread with a 1KB cache results in only a 10% miss ratio, half the value obtained when 8 threads share an 8KB cache.

7 Conclusions

Blue Gene is a massively parallel system with an entirely new instruction set architecture and system organization. The goals of the system software team, at this stage of the project, are to verify correctness and performance of the architecture and to develop a complete software stack for running applications. For those purposes, we developed a complete instruction-level simulator (BGSIM) for the Blue Gene architecture. The simulator is resource- and timing-aware, and can compute performance parameters from the execution of benchmarks. To better understand the inner workings of the Blue Gene architecture, and the behavior of Blue Gene code, we developed a visualizer (BGVIS) that presents simulation results in a more human-readable form.

With BGSIM we were able to develop a complete system software stack, comprising compilers, a runtime kernel, user-level libraries, and debugging tools. This software stack, together with the simulator and visualizer, were used to run the Splash-2 benchmark suite. Using these benchmarks, we conducted a validation and performance study of the architecture. Our performance evaluation capabilities include speedup numbers, analysis of instruction mix with different numbers of threads, analysis of processing units behavior, and parameterized cache behavior studies. These studies are important to define some hardware parameters such as cache line size and associativity.

The next step is to extend our simulation and software capabilities to system level. We need to be able to simulate a complete mesh of nodes, and to execute multi-node applications that communicate through messages. From a system software perspective, this requires communication libraries and parallel debugging tools. The visualizer also needs to be extended to the system level. We also need to simulate all the support infrastructure for Blue Gene, including file system, testing and booting, job scheduling, and system management. All these services are part of system software that will execute on a host computer. Although we use commercially available computers as hosts, the interaction with the Blue Gene *core* (the mesh of nodes) needs to be properly simulated to support development of the software components.

While we were developing the compiler, kernel and simulator the hardware design continued to evolve, in part as a response from our early performance evaluations and in part as a result of manufacturing constraints. However, the basic principles of the project remained constant. Some of the latest changes are: (i) fewer threads per node while maintaining the total number of FPUs; (ii) less memory per node, with the ability to access off-chip memory; (iii) new cache architecture; (iv) new instructions; and (v) an additional communication mode. We are in the process of updating our simulator, visualizer, and software stack to the new design, and we will soon have new and updated performance results.

References

[1] G. S. Almasi, C. Caşcaval, J. G. Castaños, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J. E. Moreira, D. Newns, M. Snir, and J. Henry S. Warren. Demonstrating the scalability of a molecular dynamics application on a Petaflop computer. Technical Report RC21965, IBM T. J. Watson Research Center, February 2001.

[2] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[3] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[4] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. Technical Report CSL-TR-92-546, Stanford University, September 1992.

[5] S. A. Herrod. Tango Lite: A multiprocessor simulation environment. Technical report, Stanford University, November 1993.

[6] Y. Kang, M. Huang, S.-M. Yoo, Z. Ge, D. Keen, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the International Conference on Computer Design (ICCD)*, October 1999.

[7] E. J. Koldinger, S. J. Eggers, and H. M. Levy. On the validity of trace-driven simulation for multiprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 244–253, May 1991.

[8] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. In *Proceedings of the Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[9] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of the International Conference on Computer Design (ICCD)*, October 1996.

[10] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. *IEEE Technical Committee on Computer Architecture newsletter*, 1997.

[11] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent RAM: IRAM. In *Proceedings of IEEE Micro*, April 1997.

[12] J. Torrellas, L. Yang, and A.-T. Nguyen. Toward a cost-effective DSM organization that exploits processor-memory integration. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[13] J. E. Veenstra and R. J. Fowler. MINT tutorial and user manual. Technical Report 452, University of Rochester, June 1993.

[14] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1995.

[15] H. P. Zima. The Gilgamesh processor-in-memory architecture and its execution model. Presentation at IBM TJ Watson Research Center, March 2001.