High-performance server systems and the next generation of online games

B. D'Amora
K. Magerlein
A. Binstock
A. Nanda
B. Yee

Developing a massively multiplayer online game which utilizes physically based simulation to provide realistic behaviors requires numerical integration functions with inherently high computational costs. This simulation, performed on the individual clients of a peer-to-peer networked game or for a client/server online game, presents challenges due to many factors, including limited computing resources at the client level and network latency in the propagation of a client's state to other clients. Computationally intensive simulation may adversely affect performance and result in a situation in which little processing capacity is left for other aspects of the game. In this paper, we explore how a game developer who is aware of these issues might create a game for IBM's recently announced Cell Broadband Engine™ processor; we also present an example of the development of a game in which multiple human and robotic characters interact with static and dynamic objects in a virtual environment. Although our experience suggests that porting code to the Cell Broadband Engine core with minimal use of its synergistic processing elements (SPEs) should not be expected to produce significant performance gains at this time, the potential of the Cell SPEs to improve performance is considerable. We discuss performance and design and implementation decisions, with programmability issues being especially noted.

INTRODUCTION

The video game industry had an annual revenue of approximately $25.4B in 2004, and this is projected to increase to $54.6B by 2009, according to Price Waterhouse Coopers.[1] This figure represents a 16.5 percent compound annual growth rate. Game platforms (both PCs and consoles) host non-networked and multiplayer networked games. Although revenue from offline games has been dominant over the years, analysts' predictions suggest that multiplayer online game sales will eventually dwarf those of traditional console PC games, with revenues approaching $5.2B by 2006.[2]

Currently, online games are predominantly turn-based role-playing games with limited behavioral dynamics incorporated into game play. Several problems, such as low network bandwidth and high latency as well as low server-side computing density, are limiting the state of the art in massively multiplayer online games (MMOGs). As the number of households with broadband access increases in the United States and around the world, the network limitations should be significantly reduced. The next generation of MMOGs will likely employ a much higher level of physically based modeling, artificial intelligence, dynamic story lines, user-created content, and gesture-based input devices, and a higher level of visual quality, motivated by a new family of high-performance game consoles equipped with integrated Ethernet controllers. Powerful servers based on the same microprocessor technology used in these consoles can enable the simulation of virtual worlds on a massive scale.

Game developers using current technology are forced to use a design model that binds the scope, quality, quantity, and complexity of interactivity between players. This limits both the type of game that developers can create and the market penetration of the game. The design model is limited due to many problems, including communications latency, bandwidth constraints, algorithmic and computational complexity, development and implementation costs, implementation time, and the inherent complexities of combining visual, musical, and literary arts with technology. MMOGs have traditionally addressed these challenges by imposing design constraints on the level of interactivity or realism provided in the game. For example, games such as EverQuest** or Ultima Online** do not attempt to provide a level of realism that depends on physical simulation, whereas Guild Wars** provides a higher level of visual realism and attempts to include better simulation than traditional role-playing games.

Physical simulation allows computer-generated objects to behave in a physically realistic manner. This is accomplished by formulating and evaluating a mathematical model consistent with the laws of physics governing the behaviors being simulated. Physical simulation is very computationally intensive and consequently very slow. It can also result in unrealistic behaviors due to instabilities at extremes in the simulation, such as very heavy or very light objects, extreme forces exerted at joints, or very large velocities. These challenges often lead simulation models to use physics for some aspects, but rely on nonphysical simulations for others. For example, the simulation of a multibody object using articulated joints may constrain one body to move itself after each time step to maintain a consistent position or orientation to an adjacent body. Higher-performing and more stable "pseudo-physics" is typically used in video games, but may be inadequate for other simulations such as those required for simulating animal behavior or mechanical systems.

The numerical integration functions required for physically based simulation to provide realistic behaviors in MMOGs have inherently high computational costs.[3] To alleviate this somewhat, the simulation may be performed on the individual clients of a peer-to-peer networked game, but this is problematic due to limited computing resources at the client level and network latency in propagating a client's state to the other clients. A client/server online game offers another alternative, providing a powerful compute server that can be authoritative over the state of the entire game world. Although many clients currently have the capacity to perform physical simulation for all objects in a player's view, with current technology this approach leaves little processing capacity for all other aspects of the game.

The performance challenges inherent in physical simulation can be addressed by utilizing a high-performance computing system to provide the computing power necessary for the complex mathematics used to model the simulation. Traditionally, the application can be parallelized and distributed over a cluster or grid of Intel Architecture (IA) computing nodes, but the communication latency and bandwidth between the nodes becomes a bottleneck in achieving real-time performance. Additionally, the cost of such a system can be prohibitive for some users. High performance and high volume mitigate both of these problems. High-performance processors can provide ten times the computing power of commercial off-the-shelf (COTS) PC/workstation processors. If used in a server cluster or grid, the computing density is much greater than servers based on a COTS processor could deliver. Consequently, there is a need for fewer nodes and less internode communication.

The Cell Broadband Engine** (BE) processor is a compute server which can meet these challenges. The Cell BE processor consists of a 64-bit PowerPC*

core augmented with eight synergistic processing elements (SPEs) suitable for high-speed, numerically intensive computation.[4] In this paper, we explore how an independent software vendor such as a game developer might create a game for the Cell BE processor with multiple human and robotic characters interacting with static and dynamic objects in a virtual environment.

Each Cell BE processor represents nine computing nodes over which an application can be distributed with very high bandwidth and low latency communications between processors. A system using multiple nodes based on the Cell BE architecture can distribute the application further across the nodes. Hence, the physical simulation application can leverage the layered bandwidth of the server system based on the Cell BE architecture to distribute the latency- or bandwidth-sensitive operations over multiple processing elements within the Cell BE processor and the less sensitive operations over multiple Cell BE nodes.

PROJECT OVERVIEW

The purpose of the project presented here was to assess the viability of using Cell BE technology to implement high-performance compute servers for the next generation of online games. The methodology used was to develop a client/server-based online game which would utilize a Cell BE server to implement rigid body dynamics in addition to global state management, relying on a Wintel (Microsoft Windows** and Intel) client for rendering. The team focused on determining the programmability and performance of the Cell BE server by porting preexisting Wintel code and developing new code using the tool chain provided for the Cell BE architecture. The most powerful Cell BE architectural characteristic for which we designed was the eight SPEs. Their 256 GFLOPS of computing power could potentially provide a significant performance boost to physically based simulations not available through current Intel Architecture IA-32 or IA-64 implementations.

In our implementation we began as a game developer might by simply porting existing code to the relatively mature and well-understood PowerPC core. Following this, we utilized the SPEs to perform some of the more computationally intensive physics simulation required for a game server. We used the SPEs to perform a hybrid integration calculation required to compute rigid body displacements in a multibody game scene. We also investigated the potential for utilizing the SPEs to perform narrow-phase collision detection. Although our experience suggests that simply porting code to the Cell BE processor PowerPC core with minimal use of the SPEs should not be expected to produce significant performance gains early in the process, the potential of the Cell BE processor SPEs to improve performance is considerable.

Our research was conducted in three phases. Phase I utilized a Wintel-based prototype with a single client and server. Performance was characterized on this system by determining frame rates and time spent in specific sections of code. Frame rates in this context indicate the number of frames simulated per second, rather than the number rendered. Streaming SIMD (single instruction multiple data) extensions (SSE) were utilized to accelerate server-side computationally intensive functions, such as numerical integration and collision detection. SIMD extensions are API (application programming interface) extensions to the C programming language that allow the programmer to utilize the SIMD extensions to the Cell BE architecture without having to write assembly language. The Wintel system serves as a reference performance benchmark. During Phase II of this study, server code was ported to the Mambo simulation environment for the Cell BE processor. We utilized a stand-alone 3D physics editor to determine specific bottlenecks in the physical simulations and to compute projected frame rates. Phase III focused on porting the server-side code base to engineering prototype boards connected via gigabit Ethernet networks to Wintel client systems. Performance was characterized and compared with previous projections. We describe design and implementation decisions, with programmability issues being especially noted.

GAME DESCRIPTION AND CHALLENGES

The story line for the game involves mechanical robots attacking a city inhabited by humans who defend themselves by using a variety of weapons. These weapons include handheld rocket launchers, machine guns, and satchel explosives. A robot can be destroyed by firing weapons at vulnerable points. Alternatively, humans can destroy static structures, causing indirect damage to a robot.

Several technical challenges were involved in implementing the articulation of robot and human

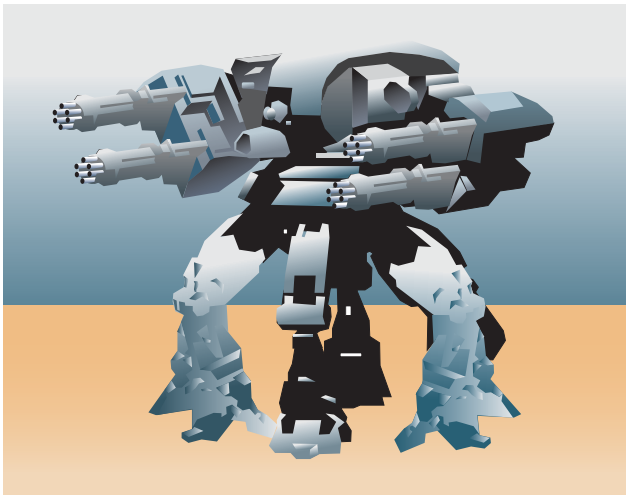

Figure 1
Mechanical robot

joints resulting in realistic movement. First, collision detection was needed to determine when moving bodies intersected with other moving or static bodies, such as walkways, buildings, and terrain. Collision detection was implemented with a two-phase approach, using a broad phase to quickly eliminate bodies that could not physically collide with each other during any given frame update and a narrow phase to specifically determine if pairs of bodies would intersect within each given frame update. This process was both integer- and floating-point-intensive. *Figure 1* shows a large robot with many articulating joints. Each pair of joints is represented in a database of collision bodies.

Robot and player movement and balance were expected to be a difficult problem. Specifically, we did not want moving bodies to look like they were "floating" over terrain or paved surfaces. The articulation of joints was intended to look smooth and coordinated. This was especially challenging because network bandwidth limitations would require differential updates of bodies to reduce data flow from server to client. We had to consider what granularity of updates in the time line would be appropriate to ensure that articulating joints did not "drift" and separate over several frames.

Client/server synchronization was another important challenge. Due to network latency time, we would need to compensate on the client for player response time that could not tolerate a round-trip communication between client and server. For a low-performance client, this compensation would require approximations that could result in a desynchronization of game-world state between the client and the server, which computes more accurately.

Ultimately, aggregate outbound server network bandwidth was limited to 100 Mb/s, and this presented a challenge for data movement to the client during state updates and packaging of information. Client-to-server communication was considerably less, and was more susceptible to the limitations of higher latency than those of overall bandwidth.

GAME DESIGN

The prototypical game was designed to execute on both client and server. The client used in this project was an IBM Intellistation* M-Pro with a 3.2 GHz P4 processor and 1 GB of RAM. The graphics adapter used was an nVidia GeForce** 6800 adapter with 256 MB of unified frame buffer memory. Client execution primarily involved the rendering of each frame. The client software processed player inputs in the form of mouse and keyboard actions. In order to accommodate very low-latency game actions, the client software received entity state input from the server and performed an "approximated" simulation. The client model of the game-world state was then updated, and the frame was rendered. A client world-state update was subsequently sent to the server. The approximated simulation could be performed on the client by using an extrapolation of entity positions from the last known positions or an interpolation based on information transferred from the server running the simulation at a slightly higher frame rate than the client updates.

The server was an engineering prototype board based upon the Cell BE architecture. The Cell BE processor was running at 2.4 MHz, and the available system memory was 512 MB. The Ethernet controller was a PCI (Peripheral Component Interconnect) E1000 card and could achieve a bandwidth of up to 100 Mb/s. The focus of the project was to test the feasibility of using the server to perform accurate real-time physical modeling of the rigid body dynamics required to simulate the movement of robot entities as they attack a virtual city. This modeling consisted of repeatedly performing collision detection by using the current positions of the

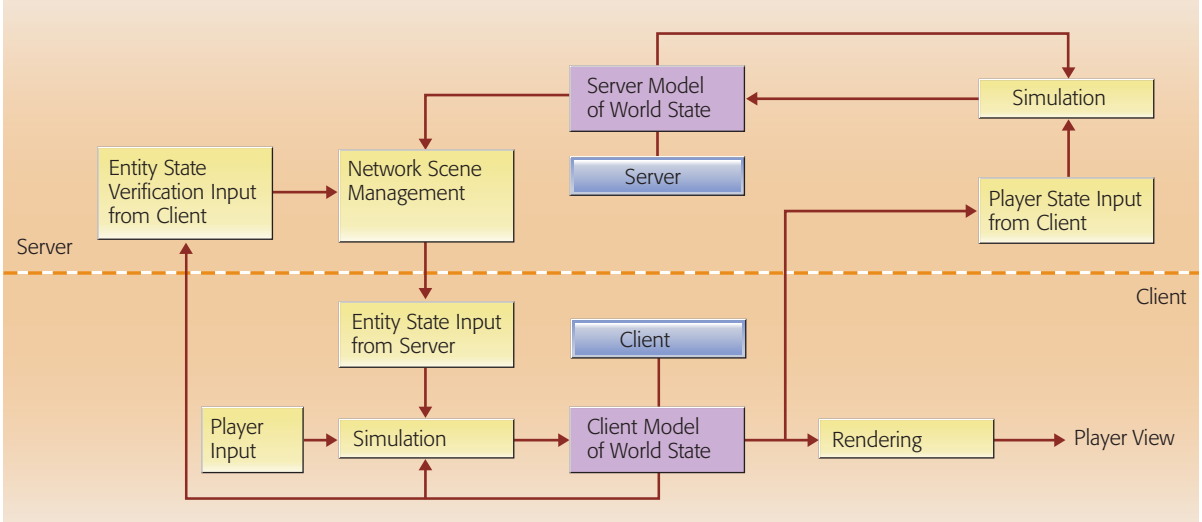

Figure 2
Data flow and game functionality

objects in the game world followed by numerical integration to obtain the positions and orientations of the objects for the next time step, based on the application of an external force. Both collision detection and integration are very computationally intensive tasks. They accounted for at least 90 percent of the server-simulation computational workload.

Finally, the server executable was tasked with keeping an inventory of state changes for the entire virtual world, a complete copy of the entire entity database representing all assets, managing network communications including determination of client updates, packaging of data, input event processing, player login, and so forth. The state of each body can change from frame to frame. These tasks, although many, did not represent a significant amount of the server workload on the reference platform. *Figure 2* describes the overall flow of data between client and server and the distribution of game functionality.

GAME PHYSICS

The physics engine provided the real-time simulation of a simplified model of real-world physics. The goal was not to provide a perfectly realistic model of reality, but rather to robustly and efficiently provide a reasonable model for building game worlds.

The engine supported a large number of rigid bodies. The collision representation was a union of convex hulls. Various control forces were provided to allow interaction. Several types of constraints were provided to allow construction of hinges, joints, and so forth. Client code interfaced with the physics engine via an object-oriented C++ interface. Client code instantiated rigid body, force, and simulator objects.

The most important distinction among physics engines is the method of integration. Physics simulation is fundamentally about solving the measure differential inclusions that describe the time evolution of a dynamic system. A common method is to reformulate this as a mixed complementary problem.[5] However, an alternate method was chosen for this project that was more robust, efficient, and better suited to SIMD hardware, namely semi-implicit integration[6] of a penalty force-based system, that is, a dynamic system that enforces constraints by applying a restorative force when the constraint is violated.

Contact and other constraints were internally reformulated as "stiff" penalty forces designed to enforce the desired behavior.[7] (Forces are said to be stiff if they are difficult or expensive to integrate explicitly.) The system was evolved by numerically integrating a system of ordinary differential equations. To handle the stiffness of these forces, implicit

integration of the equations of motion was required. A fully implicit integration requires solving a system of nonlinear equations, or alternatively, minimizing a nonlinear function of the system, which can be very expensive. Thus, a semi-implicit formulation was used. First, a first-order Taylor series of the equations of motion was expanded around the current state, producing a local model that was a linear approximation of the actual nonlinear system. Then, the integration step was achieved by solving a large sparse linear algebra problem.[3]

Continuous collision detection was not supported, and collision detection was implemented only at a fixed frequency. This frequency was a small multiple of the game update frequency. Because of this, tunneling (the complete passage of a fast-moving object through another object due to running collision detection only at discrete time steps) could occur. This presented a game-design constraint, imposing a restriction on the size versus the speed of simulated objects. Objects that did not fit within this size/speed envelope, such as bullets, were simulated outside of the physics engine.

Offline force construction

For the semi-implicit integrator to work, the derivatives of forces needed to be calculated. Often this is performed with an in-place automatic differentiation library.[8] However, we took a better approach: in a preprocessing step, force expressions were compiled, and derivative code was generated.

For example, we examine the functioning of a point-to-point constraint. The constraint function

$$C = Pb - Pa$$

indicates that the constraint is satisfied (i.e., $C = 0$) when the positions of the constraint points are equal (note this is a vector-valued equation). The formula is hard-coded as a C++ expression in the force compiler tool.

Pa and Pb are functions of the underlying state variables:

$$Pa = Xa + Ra * Pa\_bs$$

where Xa is the position of the center of mass of body a, Ra is the orientation matrix for body a, and Pa_bs is the body space position of the constraint point.

Next, we convert the constraint equation to a penalty function:

$$Fa = \partial C/\partial Ca * (Ks * C + Kd * dC/dt),$$

where Fa is the force on body a and Ca is the vector of the 6 degrees of freedom of body a's position.

The force compiler symbolically calculates this expression and then symbolically generates the partial derivative matrices required for the integrators as follows:

$$\partial Fa/\partial Xa$$
$$\partial Fa/\partial Va$$
$$\partial Fa/\partial Cb$$
$$\partial Fa/\partial Vb$$

where Va is the vector of the 6 degrees of freedom of body a's velocity. Each partial derivative is a 6×6 matrix. Similar expressions are evaluated for Fb.

The expressions are internally represented as an expression tree that allows the taking of derivatives. The output expressions are then optimized with some simple identity rules, passed through a common subexpression eliminator, and output as C code. The C code is compiled into the game runtime.

The preceding description presents the science of force formulation. The art is in the manner in which constraints are formulated. For example, we also could have written the point-to-point constraint as

$$C = ||Pb - Pa||$$

This expression provides a continuous function that is zero when the constraint is satisfied and nonzero when it is not. This second formulation works, but results in significantly worse stability (and therefore performance), because the first formulation is a much more linear function of the degrees of freedom than the second and therefore, is much better approximated by the first order Taylor series approximation used in semi-implicit integration.

Runtime step

A runtime step consists of collision detection using the current positions of the rigid bodies in the game world, revising the groups of active versus sleeping objects based on the result, and integrating to obtain the positions of the rigid bodies for the next step.

Collision detection

The broad phase of collision detection compares objects pair-wise based on their axis-aligned

bounding boxes. Alternatively, bounding spheres could have been used, but ultimately boxes were chosen because they can be made to tightly fit a variety of geometric shapes. If the bounding boxes intersect, the objects are passed along to the narrow phase for a more detailed examination to see whether the objects intersect. Bodies that are static are referred to as "sleeping," and moving bodies are referred to as "active."

In the broad phase, all of the active bodies' bounding boxes are tested against each other. Each active body is then tested against all of the sleeping bodies. It is not necessary to test the sleeping bodies against each other because they are not moving.

The narrow phase collision detection takes a pair of convex hulls and outputs a set of contacts. First, a best separating plane is found, as shown in *Figure 3*. The definition of "separating" is generalized to handle the case of penetration, when ordinary separation is not possible. In this case, a generalized conception of separation could be a plane that has the normal of the direction of minimum translation (to separate) and a position that converges to the ordinary separating plane as the two hulls separate.

The best separating plane P on hull A versus hull B is the plane that produces the maximal separating distance, where separating distance is the minimum signed distance of all the vertices of hull B from plane P. Planes from all the faces of both hulls are tested. If at any point an actual separating plane is found, the algorithm ends because the hulls are disjoint. The contact list is then generated. The two hulls are "intersected," that is, an algorithm is run which finds some attributes of the volume of intersection, and a contact point is generated for each edge-face intersection and each contained vertex.

Waking, partitioning, and sleeping

With collision detection done, the new contact list is checked to see if any active bodies have collided with any sleeping bodies. Any sleep groups (i.e., a group of bodies that was an integration group but has been put to sleep) which have been affected are moved to the list of active bodies. Partitioning takes the active bodies and separates them into non-interacting groups (islands), using the disjoint set forest algorithm.[9]

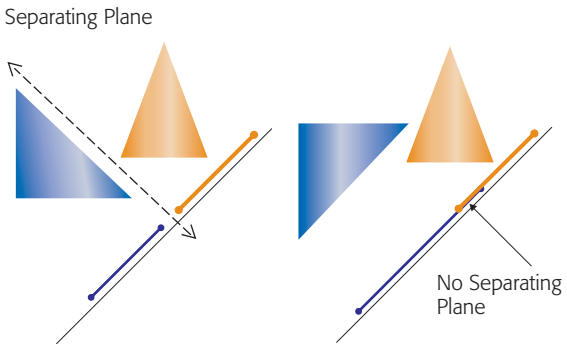

Figure 3
Narrow-phase collision detection

Each group is tested to see if it meets the requirements for being put to sleep (velocity tolerance tests over several frames). If so, it is changed to a sleeping group and not integrated.

Decoupling

Decoupling forcibly breaks up integration groups (i.e., a group of bodies that must be integrated due to their interactions, constraints, etc.) that are too large to fit in SPE memory. Decoupling is a constrained combinatoric graph optimization problem, with the bodies as the nodes of the graph and the forces, constraints, and contacts as the edges. The problem can be stated as follows: find the smallest set of partitions, such that each partition passes some cost function (involving SPE memory size), which maximizes some quality function (involving which edges are broken). As such, it is probably an NP (nondeterministic polynomial) problem. The algorithm we currently use for decoupling uses a very simple heuristic that could not be called "greedy," as it does not optimize even locally. It randomly picks a node to start with and indiscriminately attempts to grow the group until it cannot fit any more nodes under the memory limit.

An intrinsic part of the quality of the decoupling is that the decoupled partitions are randomized from step to step so that a given group is decoupled differently at each step. This is necessary in order to spread decoupling-induced error around the group. The requirement for randomization makes any caching system impracticable.

Step control

When the integration step fails for a workload (due to failure to converge when solving the linear

system), the step control system acts to keep the system from "exploding." This is done by discarding all results from solution failures (despite the fact that they may have resulted in a solution which was close to adequate), reducing the step size, and reintegrating. For performance and stability, collision detection is not rerun. Should step size reduction fail, there are a number of backup methods to keep the integration from totally failing.

Integration step

In the following, we present a mathematical overview and a description of the algorithm used for performing the integration step.

Mathematical overview

Numerical integration schemes integrate a time-dependent system Y from state $Y0 = Y(t0)$ to state $Y1 = Y(t1)$, where the time derivative function of Y is $dY/dt = F(Y)$. The step time is referred to as $h = t1 - t0$.

The simplest integration scheme is known as Euler integration:

$$Y1 = Y0 + h*F(Y0).$$

Euler integration is easy and fast to evaluate. However, it is inaccurate and, more important, unstable when presented with stiff systems.

Implicit integration handles stiff forces much more robustly. The simplest implicit integration scheme is known as backward Euler integration:

$$Y1 = Y0 + h*F(Y1).$$

However, as $Y1$ depends on $F(Y1)$, one cannot simply calculate $Y1$ as one can with forward Euler integration. Instead, one must solve for a new state $Y1$ that satisfies the above equation. As the function F is generally nonlinear, this requires solving a system of nonlinear equations. The problem can also be cast as a nonlinear minimization problem:

$$\min f(Y) = r(Y) \cdot r(Y),$$

where

$$r(Y) = Y - Y0 - h*F(Y).$$

Semi-implicit integration approximates $Y1$ by a first order Taylor expansion of F around $Y0$:

$$F(Y0 + delta\_y) \approx F(Y0) + F'(Y0)*delta\_y$$

where

$$F'(Y) = dF(Y)/dY$$

and

$$delta\_y = Y1 - Y0.$$

This gives us

$$Y1 = Y0 + h*(F(Y0) + F'(Y0)*delta\_y)$$
$$delta\_y = h*F(Y0) + h*F'(Y0)*delta\_y.$$

In order to solve for $delta\_y$, we obtain:

$$(I - h*F'(Y0))*delta\_y = h*F(Y0).$$

This is a square linear system that can be solved for $delta\_y$.

Second-order rigid body dynamics can conceptually be turned into a first order integration problem like this:

$$Y = \begin{pmatrix} x & [0] \\ v & [0] \\ x & [1] \\ v & [1] \\ \dots & \end{pmatrix}$$

where $x[0]$ is the generalized position of body 0 (or the 0th element of the system x vector), and $v[0]$ is the generalized velocity of body 0.

The derivatives are:

$$F(Y) = \begin{pmatrix} v[0] & \\ W[0] & * f[0] \\ v[1] & \\ W[1] & * f[1] \\ \dots & \end{pmatrix}$$

where

$W[0] = M[0]^{-1}$, is the inverse of the generalized mass matrix of body 0, and $f[0]$ is the force on body 0, that is, the 0th component of the force vector f (not to be confused with system derivative F).

We can see that this results in Newton's familiar formulation of dynamics:

$$dx/dt = v$$
$$M*dv/dt = f$$

We could directly use such as setup to do a semi-implicit integration, but by directly expanding $F(Y)$ in terms of $delta\_x$ and $delta\_v$, we can halve the size of the linear solution:

$$F(x0 + delta\_x, v0 + delta\_v)$$
$$\approx F(x0, v0) + dF(x0, v0)/dx*delta\_x$$
$$+ dF(x0, v0)/dv*delta\_v.$$

Because

$$delta\_x = h*v1 = h*(v0 + delta\_v),$$

we can remove *delta_x* and solve for *delta_v* only:

$$\left(I - h*W*\begin{pmatrix} df(x0,v0)/dv+ \\ h*df(x0,v0)/dx \end{pmatrix}\right) * delta\_v = h*(W*(f0 + h*(/df(x0,v0)\ /dx*v0))).$$

By introducing some abbreviations,

$$df\_dx = df(x0,v0)/dx,$$
$$df\_dv = df(x0,v0)/dv,$$

we arrive at our final problem:

$$(I - h*W*(df\_dv + h*df\_x)) * delta\_v = h*(W*(f0 + h*(df\_dx*v0)))$$

Once again, this is a linear algebra problem, but now we are only solving for *delta_v*.

$$A * delta\_v = b$$

where

$$A = I - h * W * (df\_dv + h * df\_dx)$$
$$b = h * (W * (f0 + h * (df\_dx * v0)))$$

In the maximal coordinates representation, each body contributes six components to v (three linear and three angular). *df_dx* and *df_dv* are of size $6 * num\_bodies$ by $6 * num\_bodies$ and are 6×6 block sparse. The sparseness pattern is such that the diagonal blocks are nonzero, and there are a pair of off-diagonal nonzero blocks for each pair of interacting bodies (i.e., those with a force between them).

Algorithm

The integration algorithm consists of several logical steps:

1. Calculate the components of A and b; $v0$ and W are trivial to extract, $f0$ must be calculated, and *df_dx* and *df_dv* both require considerable computational effort to calculate.
2. Form A and b.
3. Solve $A * delta\_v = b$ by a conjugate gradient method.
4. Step the system from $Y0$ to $Y1$ by *delta_v*.

IMPLEMENTATION

This project was executed on a much accelerated schedule and was intended to assess the programmability of the Cell BE architecture. To that end, the team started with an established code base that included a general game database, game engine, network manager, event handlers, and a 2D physics engine. Four fundamental challenges were identified up front:

1. Creating a 3D physics engine optimized for the eight SPEs;
2. Porting some portion of the existing C++ code to C because there was no SPE compiler support for C++;
3. Porting Windows-specific code to Linux**;
4. Providing additional network management logic to handle endianness differences between Cell BE- and Intel-based server and client platforms.

These four challenges resulted in a fairly quick divergence between the Wintel and Cell BE code base. In order to create a stable, robust reference system, the team decided to develop the game on a Wintel platform first. In reality, the Wintel version of the game was not completed before work started on the Cell BE version, and there was some dual development that occurred throughout the latter stages of the project.

Implementing an application for the Cell BE architecture requires that programmers design for one of its key architectural features, that is, the eight asynchronous SPEs. The SPEs represent an aggregate 256 GFLOPS of vector float performance. Fundamentally, to achieve optimal performance, the code needs to be partitioned across both the PPE (PowerPC Processor Element) and the SPEs. Ideally, it was our goal to port most of the code to the SPEs; practically, this was not possible, given the time constraints of the project. We focused our efforts on porting the code that represented the bulk of the workload.

Because the application relied heavily on rigid body dynamics to provide interesting game play, it was heavily dependent on collision detection and numerical integration. Either task-level or data-level parallelism could have been employed across the SPEs, but because the integration code was compact enough to reside in the 256-KB local storage area of a single SPE, we chose a data parallelization scheme across all available SPEs. This also facilitated workload balancing and hence optimal SPE utilization. The scheme was applied to the port of the numerical integration code onto the SPEs. Numerical integration required an iterative solution to the

conjugate gradient squared algorithm[10] as outlined in the following pseudocode:

```
r = rt = p = u = b − A(x)
rho = dot (r,rt)
if rho==0, return (method fails)
while (1){
        vhat = A(p)
        alpha = rho/dot (rt,vhat)
        q = u − (alpha)vhat
        u += q
        x += (alpha) u
        qhat = A(u)
        r −= (alpha)qhat
        if converged, return (ok)
        if maximum iteration count
                exceeded, return
                (not converged)
        rhoprev = rho
        rho = dot(r, rt)
        if rho==0, return (method fails)
        beta = rho/rhoprev
        u = r + (beta)q
        p = q + (beta)p
        p = u + (beta)p
}
```

A biconjugate gradient algorithm, as shown in the following pseudocode, was also tested:

```
r = rt = p = pt = b − A(x)
rho = dot (r,rt)
if rho==0, return (method fails)
while (1){
        q = A(p)
        qt = (transpose(A))(pt)
        alpha = rho/dot(pt, q)
        x += (alpha)p
        r −= (alpha)q
        rt −= (alpha)qt
        if converged, return (ok)
        if maximum iteration count
                exceeded, return
                (not converged)
        rhoprev = rho
        rho = dot(r, rt)
        if rho==0, return (method fails)
        beta = rho/rhoprev
        p = r + (beta)p
        pt = r + (beta)pt
}
```

The latter algorithm required some additional storage for the multiplication by the transposition of the matrix. The two algorithms yielded similar performance results, and the conjugate gradient squared algorithm was ultimately chosen because of its smaller memory footprint. DMA (direct memory access) was driven from the SPEs, and a single-buffer implementation was used for input and output data storage. The option of double-buffering data I/O between the SPE local store and system memory was identified, but the decision of whether to do this was deferred until the performance analysis phase of the project. The following input and output data structures were used for transfer of rigid body metrics to and from the SPEs:

```
struct Rigid_Body{

        //---------state---------
        Vec3 position;
        Quaternion orientation;
        Vec3 velocity;
        Vec3 angular_velocity;

        //----mass parameters----
        float inverse_mass;
        Matrix33 inverse_inertia;

        //----other parameters----
        float coefficient_friction;
        float coefficient_damping;
        ...
} bodies[num_bodies];


struct Rigid_Body_Step {
        Vec3 delta_velocity;
        Vec3 delta_angular_velocity;
} delta[num_bodies];
```

The remainder of the server code was targeted to run on the PPE. This included the network-management, arithmetic-coding, and particle-systems code. The network-management code was broken into two levels. The team initially focused on networked scene management. Even with low-level compression, because there was far too much data to send all of it to each client as changes occurred, the server needed to have a "scene management" layer that decided which state updates needed to be sent and to which clients. This layer was constantly working to minimize the error in each client's view

of the world and, at the same time, stay within a bandwidth budget.

Initially, the problem of networked scene management may seem difficult to allocate to the SPEs because it requires random access to all client-observable game states, and this involves much more data than can fit into an SPE's local memory. However, even in a monolithic CPU situation, because the "brute force" version of this algorithm requires prohibitive amounts of memory, the algorithm must be modified to reduce memory expenditure. Generally, this involves developing some heuristics that coarsely approximate each client's state-knowledge error. Such approximations often involve grouping entities into equivalence classes; such a grouping strategy, if properly chosen, could also help the system run on the SPEs.

The nature of this grouping heuristic depends on many factors that are not yet finalized (the number of objects in the game world, how many data items describing them need to be communicated and at what precision, how quickly and predictably the values tend to change, and so forth). The most likely approach would be to perform the broad phase, which requires many random memory accesses, on the PPE and allocate the narrow phase to the SPEs. This model does not fit perfectly because networked scene management is a somewhat stream-oriented rather than a batch-oriented task, but it can be a good starting point. This would constitute a server-side optimization. Whereas the client does tend to perform some tasks related to networked scene management, these are not very expensive compared to what the server does. The server must perform computations for each client, and these computations are more expensive.

PERFORMANCE

We were concerned primarily with server-side performance, based on the premise of the project, which was that servers based on the Cell BE architecture could be used to accelerate game play for MMOGs. We profiled the server-side components to determine both qualitative and quantitative performance differences between an Intel Pentium-4 (P4) 3.0 GHz system and a Cell BE processor 2.4 GHz system. The Cell BE processor performance profiling was done on early hardware that was not running at full clock speed (4 GHz) and had only six (as opposed to eight) functioning SPEs.

During the implementation phase, we had considered double buffering data I/O to and from the SPE local store. Profiling determined that numerical integration executing on the SPE had an extremely high ratio of computation time to DMA time (approximately 182); that is, DMA time (single buffered) was less than 1 percent of all SPE execution time. This suggested that double buffering input and output buffers would likely reduce overall performance by lowering maximum workload size without providing a speedup.

Two benchmarks were used to determine relative performance differences between the Intel and Cell BE processor platforms:

- The "Ben25" benchmark (see *Figure 4* for a screen capture) is a synthetic benchmark. It contains 25 "ben" robots and is designed to stress integration.
- The "City_Busy" benchmark (see *Figure 5* for a screen capture) is a realistic game scenario. It was created with data captured from a level during actual game play. It contains two "ben" robots, two "sparkimus_prime" robots, two "heshbot" robots, and a lot of rubble.

The results are normalized so that the P4 performance is 1.0.

Ben25 benchmark

Figure 6A shows the relative performance of the Ben25 benchmark using various processor combinations. The first obvious result is that the PPE (VMX) is less than a fifth of the speed of the P4 (SSE). This result has roughly held true for all P4-to-PPE comparisons made in this code base (VMX-to-SSE or scalar-to-scalar).

The SPE versus PPE performance gap is huge. Measuring the exact SPE performance shows that one SPE runs at more than 11 times the speed of the PPE and at more than twice the speed of the P4. Figure 6A shows a 1.5 times speedup for one SPE, which indicates that even with one SPE, the PPE is a major drag on performance. This is despite the fact that most of the PPE integration code is run in parallel on the SPE. The PPE overhead in this benchmark is mostly due to packing and unpacking SPE workloads, although there is some small overhead for SPE scheduling. The multi-SPE scheduling is less than optimal, as well. The multi-SPE

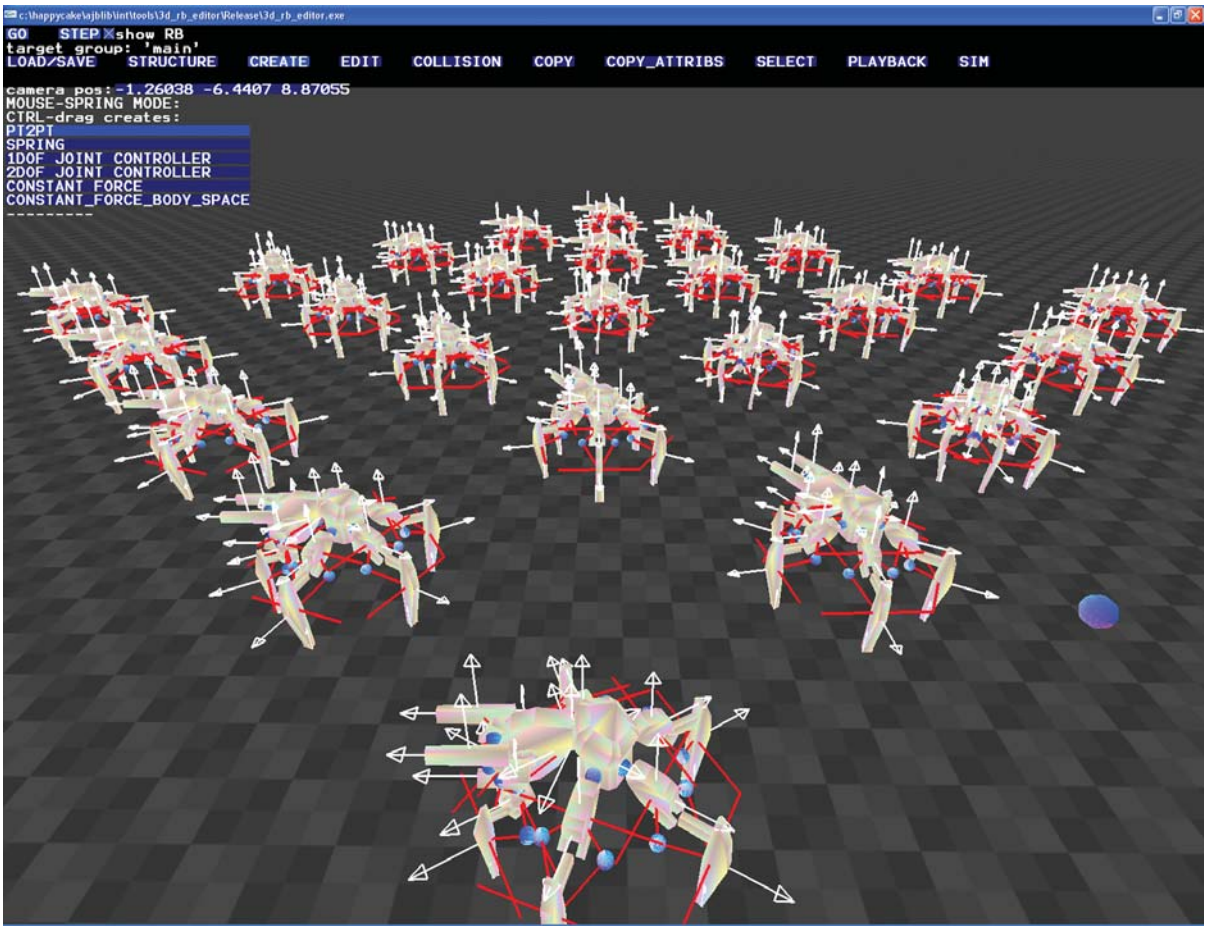

Figure 4
Ben-25 benchmark screen capture

performance continues to scale until the PPE overhead dominates at around three or four SPEs.

Figure 7B shows that the non-SPE parts of the code take a much larger proportion of the time on the Cell BE processor than on the P4, in particular when performing decoupling. The PPE is obviously a limiting factor in the current SPE optimized code. SPE optimizing of 90 percent of the P4 profile (of the integration core) results in 1.3 times the performance.

City_Busy benchmark

Compared to the Ben25 benchmark, we see that for the City_Busy benchmark (*Figure 6B*), the SPEs perform less well. This is probably due to less ideal workload generation, resulting in higher PPE overhead. This is confirmed by a higher ratio of the number of workloads to the number of bodies and the leveling off of performance at two SPEs. Another factor is that the City_Busy benchmark is less stiff than the Ben25 test (as indicated by the ratio of the number of solution steps to the number of bodies).

As shown in Figure 7B, the non-SPE parts of the code take a much larger proportion of time on the Cell BE processor than on the P4. In particular, collision detection time now dominates the Cell BE processor profile. SPE optimizing of 61 percent of the P4 profile (of the integration core) results in 0.4 times the overall performance. The modest performance gains in SPE code are swamped by huge performance losses in the PPE code.

Next steps

Clearly, the PPE is a bottleneck in the current implementation. Since much of the game code runs on the PPE, most of the advantage of the SPEs is negated. In order to improve performance, we are

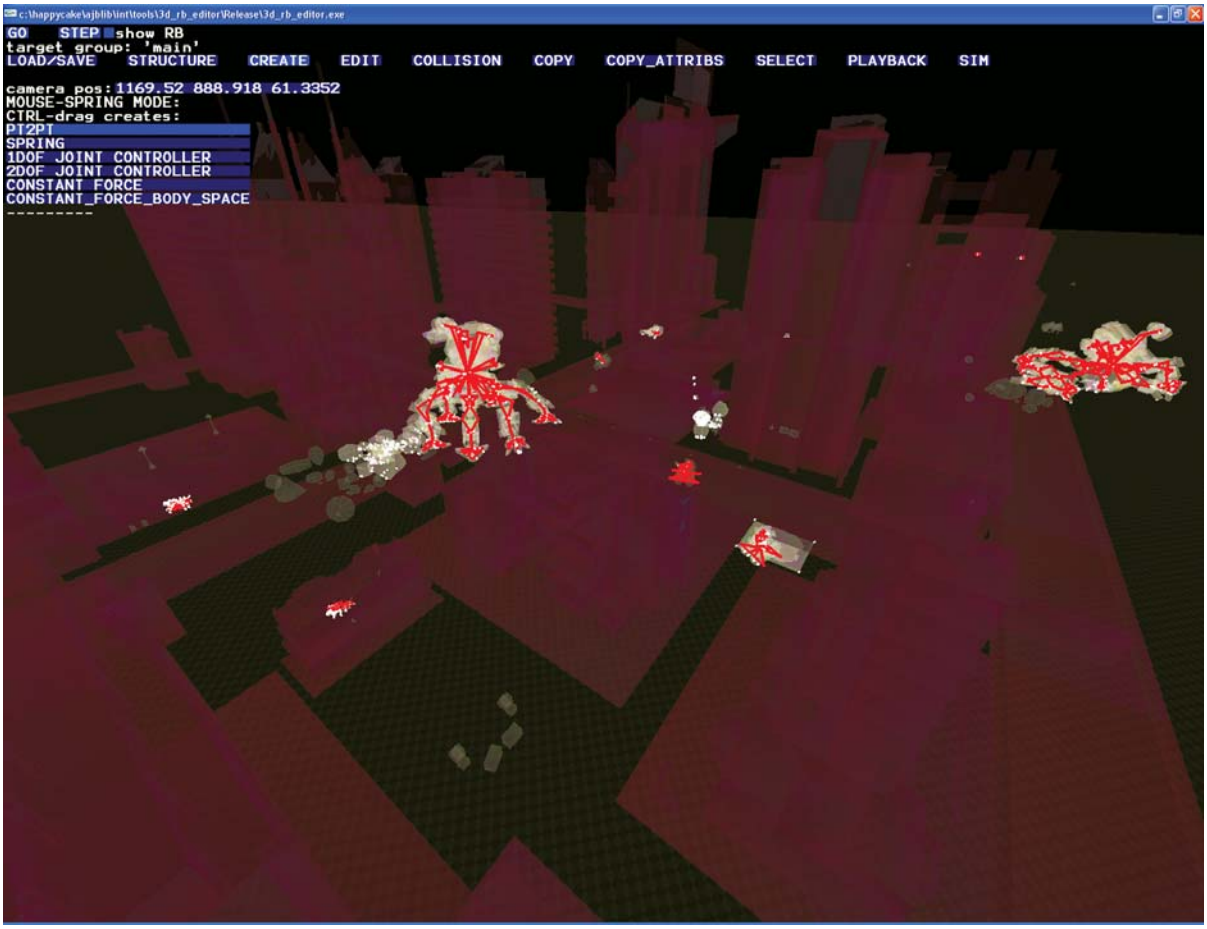

Figure 5
City-busy benchmark screen capture

focusing on moving more code to the SPEs. Ultimately, this will require considerable redesign of the current game code. One component that is currently being ported to the SPEs is collision detection. The narrow-phase function is highly vectorizable, and this would reduce a significant bottleneck in current game play.

Another important design change that would alleviate the PPE bottleneck involves the data structures used to store the game scene data, which must be transferred to the SPEs for the collision-detection and integration calculations. These structures, which describe such things as rigid bodies, collision bodies, and forces, were designed as C++ structures in the code base with which we started, and tend to be somewhat complex. For example, a collision body includes (among other things) a vector of shared faces, each of which has a normal, a vector of edges, and a vector indicating which face is on the opposite side of each edge. This complexity allows a degree of abstraction in C++ that makes algorithm development much easier. When we moved the integration to the SPEs, we had two options: packing the information from the various structures needed for each workload into contiguous storage on the PPE side and copying it to the SPE as one "chunk," or sending the addresses of the structures to the SPE and letting it crawl through the C++ structures to get the necessary data. We chose the former approach because the latter would have been difficult to program and error-prone, and we expected the PPE performance to be somewhat better than it turned out to be. However, the packing step is highly inefficient and further burdens the PPE with a data-processing task that not only would be unnecessary on an Intel platform, but would execute five times faster. An extension to the SPE compiler which

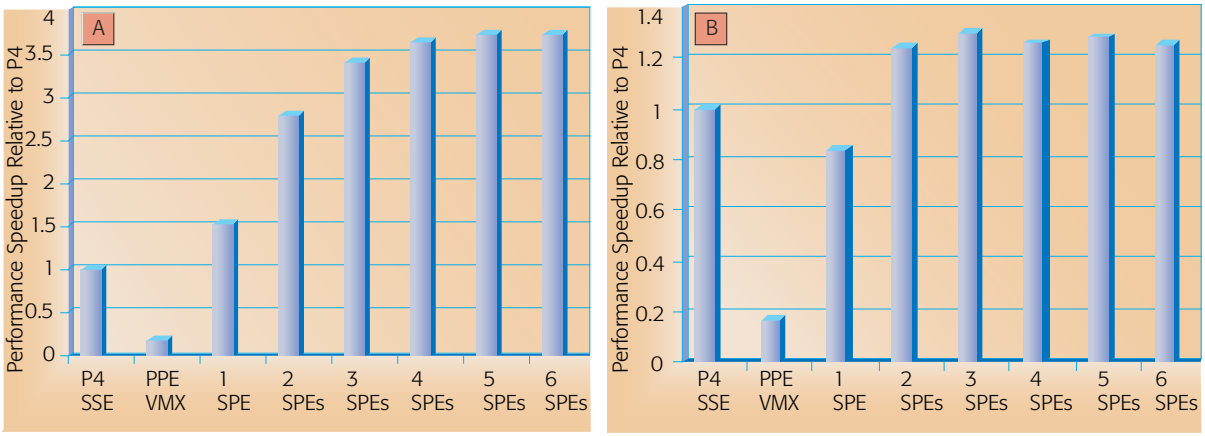

Figure 6
Relative performance for P4 versus Cell BE processor; (A) using Ben-25 benchmark, (B) using City-busy benchmark

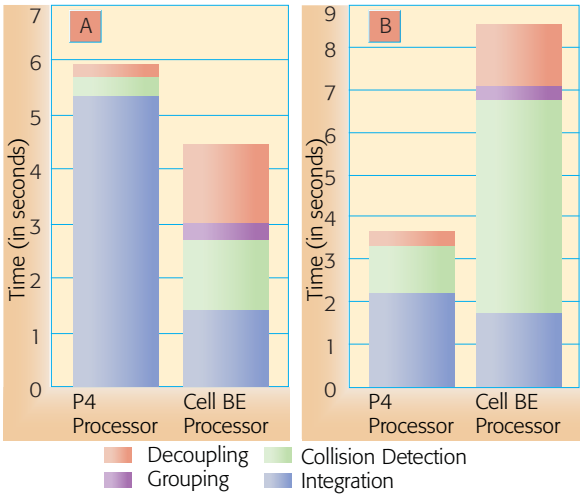

Figure 7
Time spent for code segments; (A) using Ben-25 benchmark, (B) using City-busy benchmark

would read the PPE's C++ data structures would be helpful in porting the code, but ultimately the data structures need to be simplified to obtain maximum performance.

For collision detection, we are investigating ways to simplify the data structures on the PPE side so that we can transfer blocks of contiguous storage, with pointers to vectors (again in contiguous storage) for some of the variable-sized data structures. This will require some additional discipline on the PPE side. We can store pointers to C++ vectors without resetting them every time we send them to the SPE as long as we do not add elements to the vectors during game play, because this may cause the data to be moved. This simplification also requires more complex code on the SPE side to transfer the data, but should make it possible to improve performance significantly. If so, we would investigate doing something similar with the data structures that are used in the integration step in order to break the bottleneck there.

The goal of providing a high-performance server platform for MMOGs requires significant design work to support a multiprocessor implementation. The game currently supports a "stateless" model to distribute the data for entities in the virtual world. The entities in the virtual world are maintained by one or more nodes in the system, which dispatch work to helper nodes. Workload balancing is easier to accomplish with this scheme, but the dispatcher node can become a bottleneck, and there may be long latencies to move data to remote helper nodes. Another approach would be the "territorial" model, in which the virtual-world entities are distributed throughout all the nodes in the system, and each node is responsible for processing data in a particular geographic area of the world. This approach presents several challenges. First, as objects or players move from one geographical region to another, their representative data must be transferred between the appropriate nodes, leading to longer latencies in this event. Second, if play is concentrated in a subset of the geographical regions, other nodes are idle, making load balancing a challenge.

CONCLUSIONS

It is clear that from the point of view of theoretical maximum performance, the Cell BE processor's low-latency local store with bandwidth-efficient DMA and manually managed memory latency offers significant advantages. However, from a software-engineering perspective, the impact of porting some types of legacy game software is daunting. It is not enough to identify computationally expensive subprocesses and fit them onto the SPEs; the data which supports those subprocesses must be stored in a compact way on the PPE side to simplify data transfer without requiring repacking by the PPE. It is possible to envision applications for which this could be done without great difficulty, but our game was not one of them.

The lower performance of the PPE is the major factor in considering overall system performance when porting legacy code. The SPEs performed very well, beyond our expectations, and we did not experience any DMA-related performance issues. Nevertheless, it is clear that with the current Cell BE hardware, there is not a performance advantage unless most of the non-SPE-optimized profile is moved to SPE code. Even using the SPEs to process scalar codes would offer a significant advantage over executing the same code on the PPE.

ACKNOWLEDGMENTS

We would like to thank the following people who contributed to the success of this project: Jonathan Blow of Episode, Inc., Randy Moulic of the IBM Thomas J. Watson Research Center, Robert E. Hanson and Sidney Manning of the IBM Systems and Technology Group, and Sreenivasulu Kesavarapu, Ioana Boier-Martin, Fred Mintzer, and Jeff Burns, of the IBM Thomas J. Watson Research Center.

CITED REFERENCES

1. Global Entertainment and Media Outlook, 2005–2009, Price Waterhouse Coopers (2005), http://www.pwc.com/extweb/industry.nsf/docid/6BB1D7B2F2463E1885256CE8006C6ED6?opendocument&vendor=none.
2. *The Online Game Market 2004*, DFC Intelligence (2004), http://www.dfcint.com/game_report/Online_Game_toc.html.
3. D. Baraff and A. Witkin, "Large Steps in Cloth Simulation," *Proceedings of SIGGRAPH 98* (1998), pp. 43–54, http://www.cs.cmu.edu/~baraff/papers/sig98.pdf.
4. D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, "The Design and Implementation of a First-Generation CELL Processor," *Proceedings of the IEEE International Solid-State Circuits Symposium (ISSCC 2005)* (February 2005), pp. 184–186, http://www.research.ibm.com/cellcompiler/papers/pham-ISSCC05.pdf.
5. D. E. Stewart and J. C. Trinkle, "Dynamics, Friction, and Complementarity Problems," In *Complementarity and Variational Problems: State of the Art, Proceedings of the 1995 International Conference on Complementarity Problems*, J.-S. Pang and M. C. Ferris, Editors, pp. 425–439, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1997.
6. D. Baraff, *Physically Based Modeling: Principles and Practice—Implicit Methods for Differential Equations*, Carnegie Mellon University (1997), http://www.cs.cmu.edu/~baraff/sigcourse/notese.pdf.
7. D. Wu, *Penalty Methods for Contact Resolution*, http://www.pseudointeractive.com/games/penaltymethods.ppt.
8. A. Griewank, "On Automatic Differentiation," in *Mathematical Programming: Recent Developments and Applications*, M. Iri and K. Tanabe, Editors, Kluwer Academic Publishers, Amsterdam (1989), pp. 83–108.
9. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, Chapter 21, 2nd Edition, MIT Press, Cambridge, MA (2001).
10. J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method without the Agonizing Pain*, Carnegie Mellon University, School of Computer Science (August 1994), http://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

Accepted for publication September 6, 2005.
Published online January 12, 2006.

Bruce D'Amora
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (damora@us.ibm.com).
Mr. D'Amora is a researcher in the Emerging Systems Software group at the Watson Research Center in Hawthorne, New York. His research interests are in 3D rendering and physical simulation. Mr. D'Amora is currently focusing on the design and programmability of Cell Broadband Engine™ (BE) processor-based systems targeted at video game development and digital entertainment. His previous project was pervasive 3D viewing for product data management for which he developed a 3D renderer on a network-enabled handheld device. Prior to arriving at Watson, Mr. D'Amora was the Chief Software Architect for the 3D graphics development group at IBM Austin and the IBM representative on the OpenGL architectural review board. He holds a B.A. degree in microbiology and a B.S. degree in applied mathematics from the University of Colorado. Mr. D'Amora also holds an M.S. degree in computer science from National Technological University.

Karen Magerlein
IBM Research Division, Thomas J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 (kmager@us.ibm.com).
Ms. Magerlein is an advisory engineer with the Emerging Systems Software department at the Watson Research Center. She received a B.S. degree in computer science from Duke University in 1980 and joined the Image Technologies department at IBM's Research Division the same year. She has worked on the development of programming tools, techniques for binary image manipulation, the JPEG compression standard, and image-processing software for digital library applications such as Russia's State Hermitage Museum and the Egyptian Museum. She completed her M.S. degree in computer science at Columbia University in 1995, and is the co-author of over 25 articles and patents (mostly under the name Karen L. Anderson).

Atman Binstock
Episode Technologies, 119 South Highland Ave., South Nyack, NY 10960 (atman.binstock@gmail.com). Mr. Binstock is an independent developer (at Electric Sheep Games) and a consultant in the game industry. He received a B.A. degree in computer science from New York University in 1995. That same year, he joined Utopia Technologies/Sandbox Studios/Digital Illusions, first as lead programmer, and later as designer and technical director of the studio. He has worked on the development of 3D graphics engines, procedural animation techniques, and real-time physics systems. He is credited with the development of at least 12 games, playing a lead role in the development of five of them.

Ashwini Nanda
IBM Research Division, Thomas J. Watson Research Center, 1101 Kitchawan Rd., Route 134/P.O. Box 218, Yorktown Heights, NY 10598 (ashwini@us.ibm.com). Dr. Nanda is a research staff member at the Watson Research Center, where he currently leads research and technology strategy on Cell Broadband Engine™ (BE) processor-based systems, clusters, and their applications. Dr. Nanda is also the chief architect of the Cell BE processor Blades prototype and road map in IBM. Earlier, he established and managed the Scalable Server Architecture Group in IBM Research for several years. His key research contributions at IBM include MemorIES (Memory Instrumentation and Emulation System), high-throughput coherence controllers, the Watson Commercial Server Performance lab, Cell BE processor Blades, and applications of Cell BE processor Blades. Prior to joining IBM, Dr. Nanda worked on the Amazon superscalar processor at Texas Instruments in Dallas, and led the development of a multiprocessor system at Wipro Technologies in Bangalore for India's missile research program. Dr. Nanda has been a co-general chair of the International Symposium on High Performance Computer Architecture (HPCA-7), served on the editorial board of IEEE Transactions on Parallel and Distributed Systems, and co-edited a special issue of IEEE Computer magazine. He holds ten patents and has published over 40 papers on computer system architecture, design, and performance.

Bernard Yee
Episode Technologies, 119 South Highland Ave., South Nyack, NY 10960 (bernie@episodetech.com). Mr. Yee has extensive experience in online games. At ABC Interactive/Disney Interactive, he was Director of Product Development for the PC football title Monday Night Football '97, which was Computer Gaming World's PC football game of the year and the first game to introduce head-to-head tournament-style online play. At Sony Online Entertainment, Mr. Yee managed the premium games unit, which acquired and launched EverQuest®, one of the most successful online games of all time. He helped run the Asia-Pacific launch of the United States-developed massively multiplayer game Shadowbane, the first MMOG to have a simultaneous global rollout. Mr. Yee has consulted on games for companies including Take Two, Electronic Arts, Lehman Brothers, and IBM, and has covered the industry as a journalist. He is currently the Vice President of Game Assessment and Direction at Atari, Inc. in New York City. ■